第 1 章 Python 编程基础 9
1.编程语言是什么 10
2.编译型语言和解释型语言 11
3.Python 是什么 15
4.Python 具有哪些特点（优点和缺点） 17
5.学 Python，不需要有编程基础！ 22
6.学 Python 能干什么 23
7.怎样学习 Python，才能成为 Python 高手？ 26
8.Python 2.x 和 Python 3.x，初学者应如何选择？ 29
9.Python 3 和 Python 2 的区别 32
10.Python 2to3：自动将 Python 2.x 代码转换成 Python3.x 代码 40
11.作为程序猿，怎样才能及时了解 Python 的最新变化呢？ 42
12.Python 底层是用什么语言实现的？ 44
13.了解 Jupyter Notebook，你已然超越了 90%的 Python 程序员 50
第 2 章 Python 初探 54
1.Windows 安装 Python 55
2.Linux 安装 Python 58
3.Mac 安装 Python 环境 61
4.python 不是内部或外部命令的解决方法 63
5.第一个 Python 程序 68
6.Python 交互式解释器 74
7.IDE（集成开发环境）是什么 77
8.Python IDE 有哪些，哪款适合初学者？ 78
9.Python IDLE 使用方法（包含常用快捷键） 82
10.Python 注释 88
11.Python 缩进规则（包含快捷键） 92
12.Python 编码规范（PEP 8） 95
13.Python 标识符命名规范 98
14.一文彻底搞懂编码规范的重要性 101
15.Python 关键字（保留字） 105
16.Python 内置函数 106
第 3 章 变量类型和运算符 109
1.Python 变量及其使用 109
2.Python 数值类型（整形、浮点型和复数） 113
3.为什么 Python 浮点类型存在误差？ 119
4.Python 字符串 121
5.python 字符串使用哪种编码格式？ 127
6.Python bytes 131
7.Python bool 布尔类型 135
8.Python len()函数：获取字符串长度或字节数 136
9.Python input()函数：获取用户输入的字符串 138
10.Python print()函数高级用法 141
11.Python 格式化字符串 143
12.Python 转义字符 147

13.Python 数据类型转换 150
14.Python 算术运算符 152
15.Python 赋值运算符 158
16.Python 位运算符 161
17.Python 比较运算符 169
18.Python 逻辑运算符 172
19.Python 三目运算符 175
20.Python 运算符优先级 178
第 4 章 列表、元组、字典和集合 180
1.什么是序列，Python 序列详解 180
2.Python 列表（list） 187
3.Python list 列表添加元素 193
4.Python list 列表删除元素 197
5.Python list 列表修改元素 200
6.Python list 常用方法 202
7.Python range()快速初始化数字列表 209
8.Python tuple 元组详解 212
9.Python 元组和列表的区别 220
10.Python 列表和元组的底层实现是怎样的？ 222
11.Python dict 字典 225
12.Python dict 字典基本操作 233
13.Python dict 字典方法完全攻略 236
14.Python 使用字典格式化字符串 244
15.Python set 集合 245
16.Python set 集合基本操作 250
17.Python set 集合方法 254
18.Python frozenset 集合 262
19.深入底层了解 Python 字典和集合，一眼看穿他们的本质！ 264
20.Python 深拷贝和浅拷贝详解 272
第 5 章 Python 字符串常用方法 279
1.Python 字符串拼接（包含字符串拼接数字） 279
2.Python 截取字符串 281
3.Python split()分割字符串方法 284
4.Python join()合并字符串方法 287
5. Python count()方法：统计字符串出现的次数 288
6. Python find()方法：检测字符串中是否包含某子串 290
7.Python index()方法：检测字符串中是否包含某子串 292
8.Python 字符串对齐方法（ljust()、rjust()和 center()）详 294
9.Python startswith()和 endswith() 298
10.Python 字符串大小写转换 300
11.Python 去除字符串中空格 302
12.Python format()格式化输出方法 305
13.Python encode()和 decode()方法 308
14.Python dir()和 help()帮助函数 313

第 6 章 Python 流程控制 316
1.Python if else 条件语句 316
2.Python if else 语句用法规范 322
3.Python if 语句嵌套 328
4.Python pass 331
5.Python assert 断言 332
6.如何合理使用 assert，千万不要和 if 混用！ 333
7.Python while 循环语句 336
8.Python for 循环 340
9.Python 循环结构中 else 用法 350
10.Python 循环嵌套 352
11.Python break：跳出当前循环体 354
12.Python continue：直接执行下次循环 358
13.教你一招，彻底告别死（无限）循环！ 359
14.Python 推导式，快速初始化各种序列！ 361
15.Python zip 函数 370
16.Python reversed 函数 372
17.Python sorted 函数 373
18.Python 项目实战之数字转人民币读法 375
19.Python 项目实战之绕圈圈面试题 378
20.Python 项目实战之猜数字游戏（含实现代码） 382
第 7 章 函数和 lambda 表达式 384
1.Python 函数（函数定义、函数调用）用法详解 384
2.Python 函数值传递和引用传递（包括形式参数和实际参数） 391
3.Python 函数参数传递机制 393
4.Python 位置参数 401
5.Python 关键字参数 404
6.Python 默认参数 406
7.Python 函数如何传入任意个参数？ 410
8.Python 如何用序列中元素给函数传递参数？ 414
9.Python return 函数返回值 417
10.Python 函数返回多个值 420
11.Python 函数递归 422
12.Python 变量作用域（全局变量和局部变量） 423
13.Python 如何在函数中使用同名的全局变量？ 428
14.Python 局部函数 430
15.Python 函数高级用法 433
16.Python 闭包 438
17.Python lambda 表达式（匿名函数） 440
18.Python 函数式编程（map()、filter()和 reduce()）详解 444
19.函数注解，号称 Python3 新增的最独特的功能！ 449
20.如何才能提高代码颜值，让代码变得有逼格？ 451
第 8 章 Python 类和对象 455
1.Python 面向对象 455

2.Python class：定义类.....458
3.Python __init__()类构造方法.....460
4.Python 类对象的创建和使用.....463
5.Python self.....468
6.Python 类属性和实例属性.....472
7.Python 实例方法、静态方法和类方法.....479
8.Python 类调用实例方法.....483
9.为什么说 Python 类是独立的命令空间？.....485
10.Python 描述符.....487
11.Python property().....490
12.Python @property.....493
13.Python 封装.....496
14.Python 继承机制.....501
15.在子类中，Python 到底是如何找到父类的属性和方法的？（深度揭秘）.....505
16.Python 父类方法重写.....510
使用未绑定方法调用被重写的方法.....511
17.如何使用 Python 继承机制提高开发效率？.....512
18.Python super().....514
19.切记，super()只能在新式类中使用！.....518
20.使用 super()，这些“坑”千万别踩！.....519
21.Python __slots__限制类实例动态添加属性和方法.....523
22.Python type()函数：动态创建类.....526
23.Python MetaClass 元类.....529
24.Python 底层是如何实现 MetaClass 元类的？.....532
25.Python 多态.....535
26.Python 枚举类.....538
27.Python 项目实战：利用面向对象思想实现搜索引擎.....543
第 9 章 Python 异常处理机制.....547
1.什么是异常处理.....548
2.浅谈 Python 异常处理机制.....554
3.Python try except.....557
4.Python try except else.....565
5.Python try except finally.....569
6.Python raise.....575
7.Python sys.exc_info()获取异常信息.....584
8.Python traceback.....587
9.Python 自定义异常类.....591
10.正确使用 Python 异常处理机制.....594
11.Python IDLE 调试程序.....598
12.Python assert 调试程序.....603
第 10 章 类特殊成员（属性和方法）.....605
1.Python __new__.....605
2.Python __repr__.....608

3.Python __del__ 610
4.Python __dir__ 611
5.Python __dict__ 612
6.Python setattr、getattr、hasattr 613
7.Python issubclass 和 isinstance 616
8.Python __call__ 618
9.Python 序列相关操作 620
10.Python 迭代器 623
11.Python 生成器 626
12.Python @函数装饰器 635
13.结合实例，深入了解装饰器！ 642
第 11 章 Python 模块和包 645
1.什么是模块 645
2.Python import 导入模块 647
3.Python 自定义模块 654
4.Python __name__=='__main__' 659
5.Python 导入模块的 3 种方式 661
6.Python 导入模块的本质 672
7.Python __all__变量 676
8.Python 包：存放多个模块的文件夹 678
9.Python 创建包，导入包 680
10.Python __init__.py 的作用 686
11. Python 查看模块（变量、函数、类）方法 688
12.Python __doc__ 690
13.Python __file__ 692
14.Python 第三方库（模块）下载和安装 693
第 12 章 Python 常见模块 696
1.Python sys 696
2.Python os 704
3.Python random 710
4.Python time 714
5.Python json 721
6.Python re 正则表达式模块及其用法 735
7.Python set 和 frozenset 744
8.Python queue（双端队列）模块及用法 749
9.Python heapq（堆操作） 755
10.Python ChainMap 762
11.Python Counter 766
12.Python defaultdict 771
13.Python namedtuple 772
14.Python OrderedDict 776
15.Python itertools 779
16.Python functools 786
第 13 章 Tkinter 799

1.什么是 GUI，Python GUI 常用库汇总 799
2.Python GUI 编程组件及用法 801
3.Python Tkinter Pack 817
4.Python Tkinter Grid 822
5.Python Tkinter Place 布局管理器及用法 825
6.Python Tkinter command 和 bind 事件处理（绑定）用法详解 829
7.Python Tkinter ttk 组件及用法 839
8.Python Tkinter Variable 842
9.Python Tkinter compound 选项使用方法 844
10.Python Tkinter Entry 和 Text 控件用法(包含两者区别) 846
11.Python Tkinter Radiobutton 和 Checkbutton 控件用法 853
12.PythonTkinter Listbox 和 Combobox 控件用法 860
13.Python Tkinter Spinbox 870
14.Python Tkinter Scale 和 LabeledScale 872
15.Python Tkinter LabelFrame 879
16.Python Tkinter Panedwindow 882
17.Python Tkinter OptionMenu 888
18.Python Tkinter 对话框 891
19.Python Tkinter 自定义对话框 895
20.Python Tkinter 输入对话框 900
21.Python Tkinter 文件对话框 903
22.Python Tkinter askcolor 颜色选择对话框用法 907
23.Python Tkinter 消息框 909
24.Python Tkinter Menu 菜单 915
25.Python tkinter Canvas 画布完全攻略 928
26.tkinter Canvas tag_bind：指定图形项绑定事件处理函数或方法 953
27.Python tkinter Canvas 绘制动画 963
第 14 章 Python 文件操作（I/O） 967
1.Python 文件基本操作 968
2.Python open()函数详解：打开指定文件 970
3.以文本格式和二进制格式打开文件，到底有什么区别？ 976
4.Python read()函数：按字节（字符）读取文件 978
5.Python readline()和 readlines()函数：按行读取文件 981
6.Python write()和 writelines()：向文件中写入数据 985
7.Python close()函数：关闭文件 988
8.Python seek 和 tell 990
9.Python with as 用法详解 994
10.什么是上下文管理器，深入底层了解 Python with as 语句 996
11.Python fileinput 模块：逐行读取多个文件 1001
12.Python linecache 模块用法：随机读取文件指定行 1004
13.Python pathlib 模块 1005
14.Python os.path 模块常见函数用法（实例+详细注释） 1017
15.Pythonfnmatch 模块：用于文件名的匹配 1019
16.Python os 模块 1021

17.Python tempfile 模块：生成临时文件和临时目录.....1030
第 15 章 Python 数据库编程.....1034
1.Python 数据库 API.....1034
2.Python SQLite 创建数据表.....1040
3.SQLite Expert 安装和使用.....1044
4.Python execute()和 executemany().....1048
5.Python fetchone()、fetchmany()和 fetchall().....1052
6.Python executescript().....1056
7.Python SQLite create_function().....1057
8.Python SQLite create_aggregate().....1059
9.Python SQLite create_collation()方法：创建自定义比较函数.....1062
10.MySQL 数据库下载和安装.....1064
11. Python pip（管理模块工具）基础用法.....1070
12.Python MySQL 数据库执行 DDL 语句.....1074
13.Python MySQL 数据库执行 DML 语句.....1077
14.Python MySQL 数据库执行查询语句.....1081
15.Python MySQL callproc()方法：调用数据库存储过程.....1083
16.Python PyMySQL 模块下载和安装.....1085
第 16 章 Python 并发编程.....1087
1.Python 进程和线程.....1087
2.Python 创建线程.....1089
3.Python 线程的生命周期.....1096
4.Python Thread join().....1103
5.Python 守护线程.....1105
6.Python sleep()函数用法：线程睡眠.....1107
7. Python 互斥锁（Lock）：解决多线程安全问题.....1108
8.Python 线程死锁.....1117
9.Python condition 实现线程通信.....1121
10.Python Queue 队列实现线程通信.....1128
11.Python Event 实现线程通信.....1132
12.Python 线程池.....1136
13.Python threading Local()函数用法：返回线程局部变量.....1144
14.Python Timer 定时器：控制函数在特定时间执行.....1147
15.Python schedule 任务调度及其用法.....1149
16.Python os.fork()方法：创建新进程.....1151
17.PythonProcess 创建进程（2 种方法）详解.....1153
18.Python 设置进程启动的 3 种方式.....1157
19.多进程编程和多线程编程的优缺点.....1160
20.Python 使用进程池管理进程.....1162
21.Python 进程间通信 2 种实现方法（Queue 和 Pipe）.....1165
22.Python Futures 并发编程.....1169
23.Python Asyncio 并发编程.....1176

24.深入底层了解 Python GIL 全局解释器锁 1182
25.深度解析 Python 垃圾回收机制 1188
第 17 章 Python 网络编程 1198
1.什么是计算机网络，TCP/IP 协议和端口号 1198
2.Python 网络编程模块 1204
3.Python urllib.parseurllib.parse 模块用法详解 1206
4.Python urllib.request 1213
5.Python http.cookiejar 模块：管理 cookie 1222
6. 什么是 TCP 协议，TCP 协议和 IP 协议的关系 1228
7.Python socket 建立 TCP 连接 1230
8.Python 多线程实现 socket 通信 1236
9.Python socket shutdown 方法详解 1240
10.Python selectors 模块用法：实现非阻塞式编程 1243
11.UDP 是什么，UDP 协议及优缺点（包含和 TCP 协议的区别） 1248
12.socket 发送和接受数据（基于 UDP 协议） 1249
13.UDP 多点广播 1254
14.Python smtplib 模块详解：发送邮件 1257
15.Pythonpoplib 模块：收取邮件 1264
第 18 章 Python 文档和测试 1270
1.编写 Python 开发文档的重要性 1270
2.读完本文，你也能编写高大上的 Python 软件开发文档 1272
3.Python docstring 添加注释 1277
4.Python pydoc 1279
5.Python doctest 1291
6.Python unittest（PyUnit） 1296
7.Python TestSuite 1307
8.Python setUp 和 tearDown 1311
9.Python unittest 跳过测试用例 1315
第 19 章 Python 打包和发布 1318
1.Python 打包的发展历程 1318
2.如何创建一个程序包？ 1321
3.如何在开发期间安装测试 Python 程序包？ 1335
4.Python 命名空间包 1337
5.如何向 PyPI 上传 Python 程序包？ 1343
6.Python 源码包和二进制包 1346
7.Python zipapp 打包 1352
8.Python PyInstaller 1358
9.怎样阻止别人反编译我们的程序包？ 1365
第 20 章 Python 数据可视化 1367
1.Python Matplotlib 安装 1367
2.Python Matplotlib plot 1369
3.Python Matplotlib legend 1374

4.Python Matplotlib time、xlabel、ylabel、xticks、yticks.....1379
5.Python Matplotlib subplot.....1384
6.Python Matplotlib pie.....1392
7.Python Matplotlib 绘制柱状图（bar 和 barh 函数）.....1394
8.Python Matplotlib scatter.....1402
9.Python Matplotlib contour 和 contourf.....1405
10.Python plot_surface.....1408
11.Python Pygal.....1410
12.Python Pygal 常见数据图.....1419
13.Python 读取 csv 文件.....1437
14.Python 读取 JSON 文件.....1444
15.Python 读取网络数据.....1451
第 21 章 Python Scrapy 网络爬虫.....1457
1.网络爬虫是什么.....1457
2.Python Scrapy 安装.....1459
3.Python Scrapy 项目创建.....1463
4.Python Scrapy shell.....1467
5.Python Scrapy 爬虫项目开发过程.....1474
6.Python scrapy 爬虫数据保存到 MySQL 数据库.....1485
7.Python Scrapy 突破反爬虫机制.....1487
8.Python Scrapy 反爬虫解决方案.....1496
9.Python Scrapy 整合 Selenium.....1500

第 1 章 Python 编程基础

伴随着大数据和人工智能的兴起，Python 这门“古老”的语言重新焕发出耀眼的光彩。实际上 Python 一直是一门优秀的编程语言，不仅简洁、易用，而且功能强大，它能做到的事情太多了，既可用于开发桌面应用，也可用于做网络编程，还可用于开发 Web 应用……可能正因为它能做到的方面太多，反而显得没有特别突出的一面。

另外，由于 Python 非常简单，很多非专业人士都可使用 Python，这可能导致一些专业程序员对 Python 抱有偏见。现在情况发生了改变，Python 在大数据和人工智能两个领域大放异彩，使得 Python 语言变得非常流行（目前 Python 排在商用语言排行榜的第 4 位）。

本教程将会向读者详细介绍 Python 这门优秀的编程语言。本章重点介绍如何搭建 Python 的开发环境。

1.编程语言是什么

其实，程序指的就是一系列指令，用来告诉计算机做什么，而编写程序的关键在于，我们需要用计算机可以理解的语言来提供这些指令。

虽然借助 Siri（Apple）、Google Now（Android）、Cortana（Microsoft）等技术，我们可以使用汉语直接告诉计算机做什么，比如“Siri，打开酷狗音乐”，但使用过这些系统的读者都知道，它尚未完全成熟，再加上我们语言充满了模糊和不精确因素，使得设计一个完全理解人类语言的计算机程序，仍然是一个有待解决的问题。

为了有效避开所有影响给计算机传递指令的因素，计算机科学家设计了一些符号，这些符号各有其含义，且之间无二义性，通常被它们为编程语言。编程语言中的每个结构，都有固定的使用格式（称为语法）以及精确的含义（称为语义）。换句话说，编程语言指定了成套的规则，用来编写计算机可以理解的指令。习惯上，我们将这一条条指令称为计算机代码，而用编程语言来编写算法的过程称为编码。

本教程要讲解的 Python 就是一种编程语言，除此之外，你肯定也听说过其他一些编程语言，如 C、C++、Java、Ruby 等。至今，计算机科学家已经开发了成千上万种编程语言，且随着时间演变，这些编程语言又产生了多个不同的版本。但无论是哪个编程语言，也无论有多少个版本，虽然它们在细节上可能有所不同，无疑它们都有着固定的、无二义性的语法和语义。

以上提到的编程语言，都是高级计算机语言，设计它们的目的是为了方便程序员理解和使用。

但严格来说，计算机硬件只能理解一种非常低级的编程语言，称为机器语言。

比如说，让计算机对 2 个数做求和操作，那么 CPU 可能要执行以下指令：

1. 将位于内存空间位置在 2001 的数加载到 CPU 中；
2. 再将位于内存空间位置在 2002 的数也加载到 CPU 中；
3. 在 CPU 中，对这 2 个数做求和操作；
4. 将结果存储在位置为 2003 的内存空间。

可以看到，对 2 个数执行求和操作需要做这么多工作，且这还只是笼统地描述，实际会更加复杂。

而使用 Python 这样的高级语言，对 2 个数求和可以很自然地用 c = a + b 表示，但由此带来的问题是，我们需要设计一种方法，将高级语言翻译成计算机可以执行的机器语言，有两种方法可以实现，分别是使用编译器和解释器。

使用编译器将自身等效转换成机器语言的高级语言，通常称为编译型语言；而使用解释器将自身转换成机器语言的高级语言，称为解释型语言，Python 就是解释型编程语言的一种。

> 关于编译型语言和解释型语言的含义和区别，后续章节会进行详细介绍。

2.编译型语言和解释型语言

Python 是一门解释型的编程语言，因此它具有解释型语言的运行机制。

计算机程序，其实就是一组计算机指令的集合，能真正驱动机器运行的是机器指令，但让普通开发者直接编写机器指令是不现实的，因此就出现了计算机高级语言。高级语言允许使用自然语言（通常就是英语）来编程，但高级语言的程序最终必须被翻译成机器指令来执行。

高级语言按照程序的执行方式，可以分为编译型和解释型两种。

编译型语言

编译型语言是指使用专门的编译器，针对特定平台（操作系统）将某种高级语言源代码，一次性“翻译”成可被该平台硬件执行的机器语言（包括机器指令和操作数），并包装成该平台所能识别的可执行程序的格式，这个转换过程称为编译（Compile）。编译生成的可执行程序可以脱离开发环境，在特定的平台上独立运行。

> 编译器，可以理解为就是一个比较复杂的计算机程序，它可以接受一个以高级语言编写的程序，并将其翻译成以某个计算机的机器语言表达的等效程序。

有些程序编译结束后，还可能需要对其他编译好的目标代码进行链接，即组装两个以上的目标代码模块生成最终的可执行程序，通过这种方式实现低层次的代码复用。

因为编译型语言是一次性编译成机器语言的，所以可以脱离开发环境独立运行，而且通常运行效率较高。但因为编译型语言的程序被编译成特定平台上的机器码，因此编译生成的可执行程序通常无法移植到其他平台上运行，如果需要移植，则必须将源代码复制到特定平台上，针对特定平台进行修改，至少需要采用特定平台上的编译器重新编译。

现有的 C 、C++、Objective-C、Pascal 等高级语言都属于编译型语言。

半编译型语言

此外，还有一种伪编译型语言，如 Visual Basic，它属于半编译型语言，并不是真正的编译型语言。它首先被编译成 P-code 代码（中间代码），并将解释引擎封装在可执行程序内，当运行程序时，P-code 代码才会被解析成真正的二进制代码。

从表面上看，Visual Basic 可以编译生成可执行的 EXE 文件，而且这个 EXE 文件也可以脱离开发环境，在特定平台上运行，非常像编译型语言。实际上，在这个 EXE 文件中，既有程序的启动代码，也有链接解释程序的代码，而这部分代码负责启动 Visual Basic 解释程序，再对 Visual Basic 代码进行解释并执行。

解释型语言

解释型语言是指使用专门的解释器，将源程序逐行解释成特定平台的机器代码并立即执行的语言。

解释型语言通常不会进行整体性的编译和链接处理，解释型语言相当于把编译型语言中的编译和解释过程混合到一起同时完成。

可以这样认为，每次执行解释型语言的程序都需要进行一次编译，因此解释型语言的程序运行效率通常较低，而且不能脱离解释器独立运行。但解释型语言有一个优势，就是跨平台比

较容易，只需提供特定平台的解释器即可，每个特定平台上的解释器都负责将源程序解释成特定平台的机器指令。

> 也就是说，解释型语言可以方便地实现源程序级的移植，但这是以牺牲程序执行效率为代价的。

编译型语言 PK 解释型语言

编译型语言和解释型语言的对比如图 1 所示。

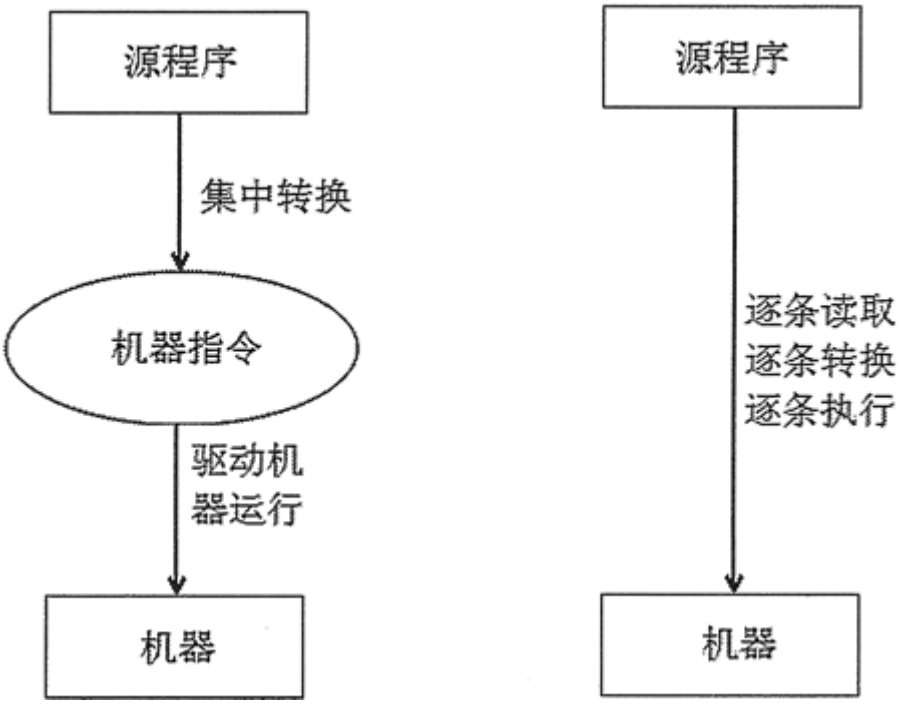


图 1 编译型语言和解释型语言

借助图 1 不难理解，编译型语言和解释型语言的区别在于，编译是对高级语言程序进行一次性翻译，这样的好处是，一旦源程序被彻底翻译，它就可以重复运行，且今后都不再需要编译器和源代码；而如果使用解释器，则高级语言程序每次运行，都需要借助源程序和解释

器，其最大的好处就是，程序有很好的可移植性。

本教程所讲的 Python 语言，属于解释型语言，因此运行 Python 程序时需要使用特定的解释器进行解释、执行。解释型的 Python 语言天生具有跨平台的能力，只要为 Python 提供相应平台的解释器即可。

3.Python 是什么

前面提到，编程语言有“高低”之分，而高级语言又有诸多版本，比如 C 、C++、Java 等，「Python」也是其中之一。Python 语言算得上一门“古老”的编程语言，其流行这么久，必然有它的独到之处，本节我们简单介绍 Python 的相关情况。

Python，本义是指“蟒蛇”。1989 年，荷兰人 Guido van Rossum（后续简称 Guido）为了打发圣诞节的无趣，决心开发一个新的脚本解释程序，于是便诞生了一种面向对象的解释型高级编程语言，被命名为 Python，其标记如图 1 所示。

图 1 Python 的标志

有人说，之所以 Guido 会选择 Python 作为该编程语言的名字，是因为他是一个叫 Monty Python 戏剧团体的忠实粉丝。

Python 语言是基于 ABC 教学语言的，而 ABC 这种语言非常强大，是专门为非专业程序员设计的。但 ABC 语言并没有获得广泛的应用，Guido 认为是非开放造成的。

> ABC 是一种交互式、结构化的高级语言，旨在替代 BASIC、Pascal 等语言，用于教学以及原型软件设计。值得一提的是，Python 创始人 Guido 曾在 ABC 系统开发中工作了数年。

Guido 在 Python 中避免了 ABC 不够开放的劣势，加强了 Python 和其他语言如 C、C++ 和 Java 的结合性，令其具有丰富而强大的库，能够把使用其他语言制作的各种模块（尤其是 C 和 C++）很轻松地联结在一起，因此 Python 又常被称为“胶水”语言。

> 这里的模块，可以简单理解为一个个可实现各种不同功能的方法（也可称为函数）。

Python 的设计哲学为优雅、明确、简单，实际上，Python 始终贯彻着这一理念，以至于如今网络上流传着“人生苦短，我用 Python”的说法。Python 上手非常简单，它的语法非常像自然语言，对非软件专业人士而言，选择 Python 的成本最低，因此某些医学甚至艺术专业背景的人，往往会选择 Python 作为编程语言。

正是由于 Python 所具有的这些特性，使得其自 1991 年 Python 第一个公开发行版问世后，自 2004 年起 Python 的使用率呈线性增长，不断受到编程者的欢迎和喜爱；2010 年，Python 荣膺 TIOBE 2010 年度语言桂冠；2017 年，IEEE Spectrum 发布的 2017 年度编程语言排行榜中，Python 位居第 1 位。

直至现在（2019 年 6 月份），根据 TIOBE 排行榜的显示，Python 也居于第 3 位，且有继续提升的态势（如表 2 所示）。

表 2 TIOBE 2019 年 6 月份编程语言排行榜（前 10 名）

Jun 2019	Jun 2018	Change	Programming Language	Ratings
1	1		Java	15.004%
2	2		C	13.300%
3	4	↑	Python	8.530%
4	3	↓	C++	7.384%
5	6	↑	Visual Basic .NET	4.624%
6	5	↓	C#	4.483%
7	8	↑	JavaScript	2.716%
8	7	↓	PHP	2.567%
9	9		SQL	2.224%
10	16	⇈	Assembly language	1.479%

4.Python 具有哪些特点（优点和缺点）

我们已经不止一次讲过，Python 是一种面向对象、解释型的脚本语言，同时也是一种功能

强大而完善的通用型语言。相比其他编程语言（比如 Java），Python 代码非常简单，上手非常容易。

举个简单的例子，比如要完成某个功能，如果用 Java 需要 100 行代码，但用 Python 可能只需要 20 行代码，这是 Python 具有巨大吸引力的一大特点。

不仅如此，Python 还有其他的特点，下面就给大家做一下总结。

Python 特点：简单易学

Python 是一种代表简单注意思想的语言，阅读一个良好的 Python 程序，即使是在 Python 语法要求非常严格的大环境下，给人的感觉也像是在读英语段落一样。

换句话说，Python 编程语言最大的优点之一，是其具有伪代码的特质，它可以让我们在开发 Python 程序时，专注于解决问题，而不是搞明白语言本身。

Python 特点：开源

Python 是 FLOSS（自由/开源源码软件）之一，简单地理解就是，用户使用 Python 进行开发和发布自己编写的程序，不需要支付任何费用，也不用担心版权问题，即使作为商业用途，Python 也是免费的。

开源正在成为软件行业的一种发展趋势，现在有很多商业软件公司都开始将自己的产品变成

开源的（例如 Java）。也许，Python 的开源正是它如此优秀的原因之一，因为会有这么一群人，他们希望看到一个更加优秀的 Python，从而为了这个目标，不断地对 Python 进行创造，不断地改进。

Python 特点：高级语言

Python 是高级语言，因此当使用 Python 语言编写程序时，我们无需再考虑一些底层细节方面的问题。例如，如何管理程序使用的内存等等。

Python 特点：解释型语言

一个用编译型语言（如 C 或 C++）写的程序，可以从源文件转换到一个计算机使用的语言。这个过程主要通过编译器完成。当运行程序的时候，我们可以把程序从硬盘复制到内存中并且运行。

而 Python 语言写的程序，则不需要编译成二进制代码，可以直接从源代码运行程序。在计算机内部，由 Python 解释器把源代码转换成字节码的中间形式，然后再把它翻译成计算机使用的机器语言并运行。

事实上，由于不再担心如何编译程序，使得使用 Python 变得更加简单，我们只需要将 Python 程序复制到另外一台计算机上，它就可以工作了。因此，Python 程序更加易于移植。

Python 特点：可移植性

由于 Python 是开源的，它已经被移植到许多平台上。如果能够避免使用依赖系统的特性，那就意味着，所有 Python 程序都无需修改就可以在好多平台上运行，包括 Linux 、Windows、FreeBSD、Solaris 等等，甚至还有 PocketPC、Symbian 以及 Google 基于 Linux 开发的 Android 平台。

解释型语言几乎天生就是跨平台的。Python 作为一门解释型的语言，它天生具有跨平台的特征，只要为平台提供了相应的 Python 解释器，Python 就可以在该平台上运行。

Python 特点：面向对象

Python 既支持面向过程编程，也支持面向对象编程。在“面向过程”的语言中（如 C 语言），程序仅仅是由可重用代码的函数构建起来的；而在“面向对象”的语言（如 C++）中，程序是由数据和功能组合而成的对象构建起来的。

与其他编程语言（如 C++ 和 Java）相比，Python 是以一种非常强大，而又简单的方式实现的面向对象编程。

Python 特点：强大的功能

Python 强大的功能也许才是很多用户支持 Python 的最重要的原因，从字符串处理到复杂的 3D 图形编程，Python 借助扩展模块都可以轻松完成。

实际上，Python 的核心模块已经提供了足够强大的功能，使用 Python 精心设计的内置对象可以完成许多功能强大的操作。

此外，Python 的社区也很发达，即使一些小众的应用场景，Python 往往也有对应的开源模块来提供解决方案。

Python 特点：可扩展性

Python 的可扩展性体现为它的模块，Python 具有脚本语言中最丰富和强大的类库，这些类库覆盖了文件 I/O、GUI、网络编程、数据库访问、文本操作等绝大部分应用场景。

Python 可扩展性一个最好的体现是，当我们需要一段关键代码运行的更快时，可以将其用 C 或 C++ 语言编写，然后在 Python 程序中使用它们即可。

除了以上几个特点（也可称之为优点）之外，作为一个解释型语言，Python 自然也有一些弱点，比如：

- 速度慢：Python 程序比 Java、C、C++ 等程序的运行效率都要慢。
- 源代码加密困难：不像编译型语言的源程序会被编译成目标程序，Python 直接运行源程序，因此对源代码加密比较困难。

其实，这两个缺点并不是什么大问题，首先，由于目前计算机的硬件速度越来越快，软件工程往往更关注开发过程的效率和可靠性，而不是软件的运行效率；至于第二个问题就更不是

问题了，现在软件行业的大势本就是开源，就像 Java 程序同样很容易反编译，但丝毫不会影响它的流行。

5.学 Python，不需要有编程基础！

“编程零基础，可以学习 Python 吗”，这是很多初学者经常问我的一个问题。当然，在计算机方面的基础越好，对学习任何一门新的编程语言越有利。但如果你在编程语言的学习上属于零基础，也不用担心，因为无论用哪门语言作为学习编程的入门语言，总是要有一个开始。

就我个人的观点，Python 作为学习编程的入门语言是再合适不过的。凡是在大学计算机专业学习过 C 语言的同学都感同身受，认为 C 语言不是很好的入门语言，很多曾经立志学习编程的读者，在学习了 C 语言之后，就决心不再学习编程。因此，是否学会 C 语言，好像成为了进入编程行业的筛选标准。

但是，如果将 Python 作为编程入门语言，就不会出现类似 C 语言的那些窘境问题。目前，逐渐有高校开始使用 Python 作为软件专业大学生（甚至也包含非软件专业）的入门编程语言。

本教程始终贯彻的思想就是，零基础也能学 Python，教程的目标就是和初学者一起，从零基础开始学习 Python。因此，编程零基础的你，无需犹豫，尽管放胆来学。

除此之外，很多初学者还面临这样一个问题，那就是教程已经学完啦，教程中的代码也都已

经亲自编写并运行通过了，但还是不知道如何开发一个真正的应用程序，面对问题还是不知道如何下手解决。

如果你深有同感，只能说明你缺乏练习，代码编辑量太少。从编程零基础到专业程序员的过程，除了学习更多的基础知识，更要刻意地去培养自己的编程思维，这没有捷径，只有靠不断积累自己的代码量。

当然，增加代码量并不是要我们去盲目地编写代码，如果找不到增加代码量的方向，可以从阅读别人的代码开始。需要注意的是，在阅读他人编写的代码时，要边阅读边思考，多问几个为什么，例如代码为什么要这么写，有什么意图，有没有更简单的方法可以实现等等，必要情况下还可以给代码进行必要的注释。不仅如此，在完全理解他人代码的前提下，还可以试图对代码做修改，实现一些自己的想法。做到这些，才能说明你将别人的代码消化吸收了。

初学者在写代码或者调试代码的过程中，肯定会遇到很多问题，怎么办呢？最好的方法就是借助网络寻找答案，看看类似的问题别人是如何解决的，千万不要总是局限在自己的思维范围中。在这里，给大家推荐一个专门针对编程答疑解惑的网站 Stack OverFlow。

6.学 Python 能干什么

Python 作为一种功能强大的编程语言，因其简单易学而受到很多开发者的青睐。那么，Python 的应用领域有哪些呢？

概括起来，Python 的应用领域主要有如下几个。

Web 应用开发

Python 经常被用于 Web 开发。例如，通过 mod_wsgi 模块，Apache 可以运行用 Python 编写的 Web 程序。Python 定义了 WSGI 标准应用接口来协调 HTTP 服务器与基于 Python 的 Web 程序之间的通信。

不仅如此，一些 Web 框架（如 Django、TurboGears、web2py 等等）可以让程序员轻松地开发和管理复杂的 Web 程序。

举个最直观的例子，全球最大的搜索引擎 Google，在其网络搜索系统中就广泛使用 Python 语言。另外，我们经常访问的集电影、读书、音乐于一体的豆瓣网（如图 1 所示），也是使用 Python 实现的。

图 1 用 Python 实现的豆瓣网

操作系统管理、自动化运维开发

很多操作系统中，Python 是标准的系统组件，大多数 Linux 发行版以及 NetBSD、OpenBSD 和 Mac OS X 都集成了 Python，可以在终端下直接运行 Python。

有一些 Linux 发行版的安装器使用 Python 语言编写，例如 Ubuntu 的 Ubiquity 安装器、Red Hat Linux 和 Fedora 的 Anaconda 安装器等等。

另外，Python 标准库中包含了多个可用来调用操作系统功能的库。例如，通过 pywin32 这个软件包，我们能访问 Windows 的 COM 服务以及其他 Windows API；使用 IronPython，我们能够直接调用 .Net Framework。

通常情况下，Python 编写的系统管理脚本，无论是可读性，还是性能、代码重用度以及扩展性方面，都优于普通的 shell 脚本。

游戏开发

很多游戏使用 C++ 编写图形显示等高性能模块，而使用 Python 或 Lua 编写游戏的逻辑。和 Python 相比，Lua 的功能更简单，体积更小；而 Python 则支持更多的特性和数据类型。

比如说，国际上指明的游戏 Sid Meier's Civilization（文明，如图 2 所示）就是使用 Python 实现的。

图 2 Python 开发的游戏

编写服务器软件

Python 对于各种网络协议的支持很完善，所以经常被用于编写服务器软件以及网络爬虫。

比如说，Python 的第三方库 Twisted，它支持异步网络编程和多数标准的网络协议（包含客户端和服务器端），并且提供了多种工具，因此被广泛用于编写高性能的服务器软件。

科学计算

NumPy、SciPy、Matplotlib 可以让 Python 程序员编写科学计算程序。

以上都只是 Python 应用领域的冰山一角，总的来说，Python 语言不仅可以应用到网络编程、游戏开发等领域，还可以在图形图像处理、只能机器人、爬取数据、自动化运维等多方面展露头角，为开发者提供简约、优雅的编程体验。

7.怎样学习 Python，才能成为 Python 高手？

在学习过程中，很多小伙伴经常抱怨，计算机编程语言种类太多，根本学不过来，可能用了 Java 很多年，突然最近的项目需要用 Python，就感到不知所措，压力山大。

再举个例子，Facebook 的主流语言是 Hack（PHP 的进化版本），但对于刚刚入职的工程师而言，100 个里至少有 95 个之前根本没有用过 Hack 或者 PHP。不过，这些人上手都特别快，基本上只需要 1~2 周，日常编程就变得毫无压力。

那么，他们是怎么做到的呢？

编程语言都是相通的

首先，如果你具有一定的编程基础，接触一门新的编程语言时会发现，不同的编程语言之间其实是相通的，因为编程语言本就是人类控制计算机的一系列指令，即便是不同的编程语言，

它们在语法规则方面也大同小异。

因此，在原有编程基础上再学习一门新的编程语言，并没有那么难。学习过程中，首先要做到的就是明确区别。例如，学习 Python 的条件和循环语句时，可以比对 C 语言或者 C++ 语言的语法是怎样的；学习 Python 中的字符串相加时，可以对比 Java 语言中字符串相加的语法。

除了能够明确区分各编程语言的不同点，还要能将它们联系起来灵活运用。比如说，Python 语言的优势是擅长数据分析，因此它广泛应用于人工智能（AI）、机器学习等领域（例如机器学习用到的 TensorFlow 框架就是用 Python 写的），但是涉及到底层的矩阵运算等操作时，就需要依赖 C++ 语言，因为它的速度快，运行效率更高。

事实上，很多公司都是根据不同的需求选择不同的编程语言进行开发。毕竟，哪怕只是几十到几百毫秒的速度差距，对用户体验来说都是决定性的。

Python 对初学者很友好

如果 Python 是你接触的第一门编程语言，那也不必担心。我们知道，虽然同为人机交互的桥梁，但 Python 比 C++、Java 等，语言更简洁，也更接近英语，对初学者很友好，这也是 Python 语言的一个显著特点。

对于初学者来说，要做的就是专注于 Python 这一门语言，明确学习的重点，把握好学习节奏，由浅入深循序渐进地学习。根据本人多年的学习工作经验，把编程语言的学习分为以

下 3 步，无论你是否有编程基础，都可以对照着来做。

1) 多实践，积累代码量

任何一门编程语言，其囊括的知识面都是非常广泛的，从基本的变量赋值、条件循环、到文件操作、并发编程等，千万不要等到把所有知识点都学完了才开始练习，因为到那时你会发现，前面好不容易记住的知识都忘记了。

学习编程，是十分讲究实战的，没有捷径可走，越早练习，练得越多越勤，学习效果就越好。

2) 时刻注意代码规范

学习编程语言，在追求快和高效的同时，每一种编程语言都有必要的编码规范，一定不能忽略。

要想成为一名优秀的程序员，从起步阶段就要刻意地去培养自己的编程规范。例如，在刚开始编写代码时，不要将原本多行的代码全部写到一行，也不要随便用 a、b、c 等字母作为变量的名称。

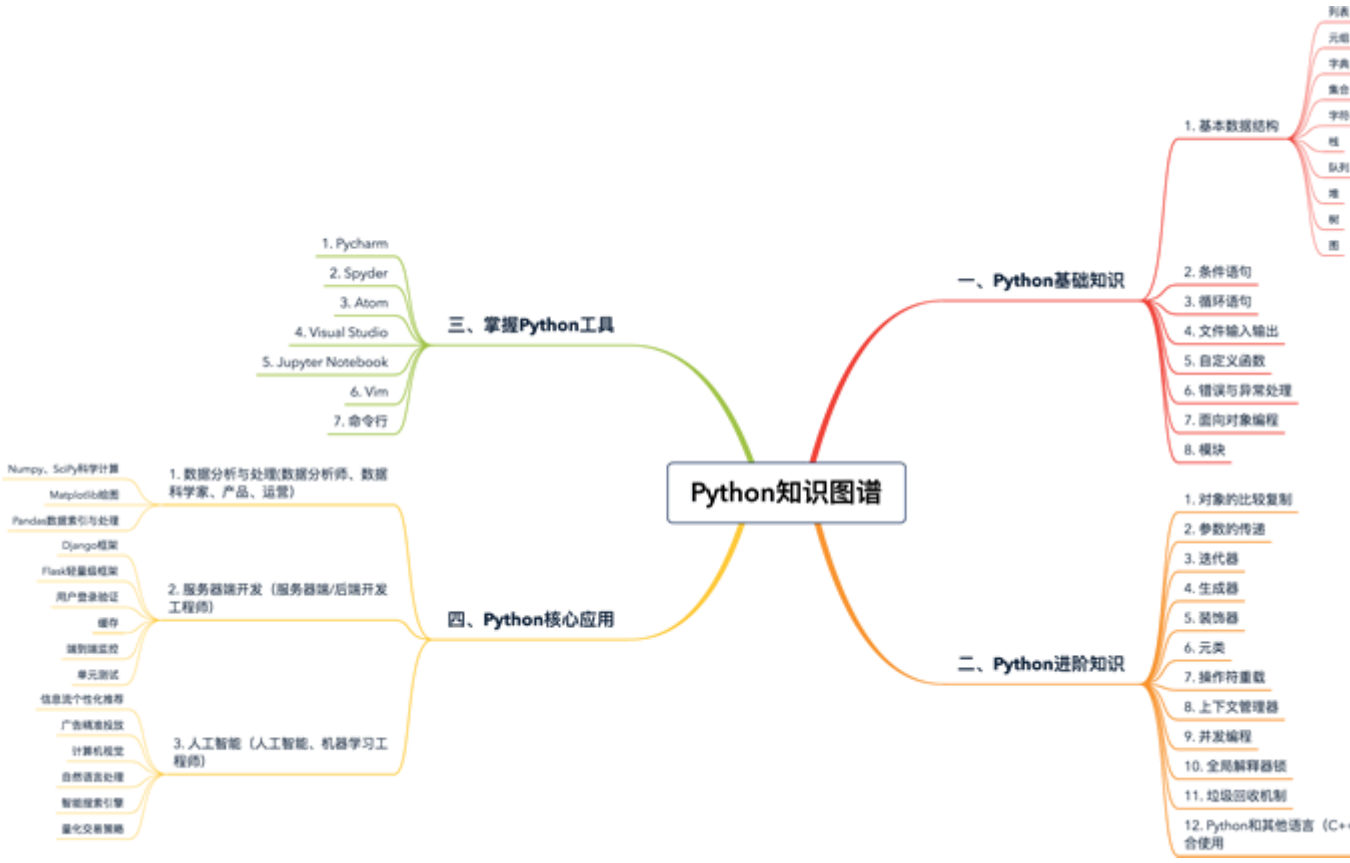
3) 开发经验是必不可少的

要想真正熟练地掌握 Python（或者其它编程语言），拥有大中型产品的开发经验是必不可少的，它会让你站得更高，望得更远。

例如，我们几乎每天都会用搜索引擎，但你了解一个搜索引擎的服务器端实现吗？这是一个

典型的面向对象设计，需要定义一系列相关的类和函数，还要从产品需求、代码复杂度、效率以及可读性等多个方面综合考量，同时在上线后还要进行各种优化等。

本教程中无法让你完成一个上亿用户级产品的实践，但设置有一些小项目，可以帮你掌握必要的开发知识。最后，这里为你准备了一章 Python 学习的知识图谱，涵盖了 Python 的核心知识，其中的大部分内容本教程都会做详细讲解。



Python 知识体系框架（点击查看高清大图）

8.Python 2.x 和 Python 3.x，初学者应如何选择？

Python 自发布以来，主要有 3 个版本，分别是：

1. 1994 年发布的 Python 1.0 版本（已过时）；
2. 2000 年发布的 Python 2.0 版本，截止到 2019 年 3 月份，已经更新到 2.7.16；
3. 2008 年发布的 Python 3.0 版本，截止到 2019 年 3 月份，已经更新到 3.7.3；

3 个版本中，Python 3.0 是一次重大的升级，为了避免引入历史包袱，Python 3.0 没有考虑与 Python 2.x 的兼容，这也就导致很长时间以来，Python 2.x 的用户不愿意升级到 Python 3.0。

除此之外，造成目前这种状况的另一个原因是，将现有应用从 Python 2.x 迁移到 Python 3.x 是一项不小的挑战。虽然有 2to3（后续会介绍）之类的工具可以进行代码的自动转换，但无法保证转换后的代码 100% 正确。而且，如果不做人工修改的话，转换后的代码性能可能还不如转换前。因此，将现有的复杂代码库迁移到 Python 3.x 上可能需要付出巨大的经历和成本，某些公司无法负担这些成本。

目前，根据统计显示，使用 Python 2.x 的开发者仍占 63.7%，而 Python 3.x 的用户占 36.3%，由此可见，使用 Python 2.x 的用户还是占多数。在 2014 年，Python 创始人宣布，将 Python 2.7 支持时间延长到 2020 。

那么，初学者应该选择什么版本呢？本教程建议大家选择 Python 3.x 版本，理由有以下几点：

- 使用 Python 3.x 已经是大势所趋

目前，虽然使用 Python 2.x 的开发者居多，但使用 Python 3.x 的开发者正在迅速增加，如图 1 所示：

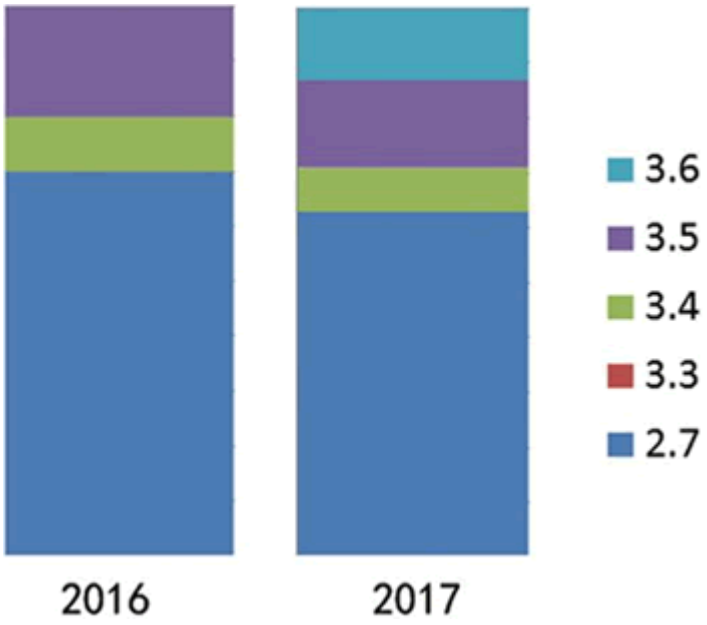


图 1 Python 3.x 是大势所趋

另外，根据 PEP-404 这份文档，Python 2.x 版本将不会再发布 2.8 版本，且未来非常重要的项目（如 Django、Flask 和 Numpy）可能都将放弃对 2.x 的支持，仅支持 Python 3.x。因此，从长远来看，学习 Python 3.x 只有好处。

本教程也是以 Python 3.x 来介绍 Python 编程，因此，为了同步，强烈建议初学者选择 Python 3.x。

Python 3.x 在 Python 2.x 的基础上做了功能升级

Python 3.x 对 Python 2.x 的标准库进行了一定程序的重新拆分和整合，比 Python 2.x 更容易理解，特别是在字符编码方面。Python 2.x 中对于中文字符串的支持性能不够好，需要编写单独的代码对中文进行处理，否则不能正确显示中文，而 Python 3.x 已经将该问题成功解决了。

Python 3.x 和 Python 2.x 思想基本是共通的

Python 3.x 和 Python 2.x 思想基本上是共通的，只有少量的语法差别，学会的 Python 3.x，只要稍微花点时间学习 Python 2.x 的语法，即可灵活运用这两个不同版本了。

注意，选择 Python 3.x 也不是没有弊端，很多扩展库的发行总是会滞后于 Python 的发行

版本，甚至目前还有很多库不支持 Python 3.x。

因此，在选择 Python 时，一定要先考虑清楚自己的学习目的，比如说，打算做哪方面的开发，此方向需要用法哪些扩展库，以及这些扩展库支持的最高 Python 版本等，明确这些问题后，再选择适合自己的版本。

> 关于 Python 3.x 和 Python 2.x 具体的区别，可阅读《Python 3 和 Python 2 区别》一节。

9.Python 3 和 Python 2 的区别

和 Python 2.x 版本相比，Python 3.x 版本在语句输出、编码、运算和异常等方面做出了一些调整，本节就对这些调整一一做一下简单介绍。

> 本节适合有 Python 基础的学员阅读，初学者可先跳过本节，整体学完 Python 之后，再回过头来阅读。

Python 3.x print 函数代替了 print 语句

在 Python2.x 中，输出数据使用的是 Print 语句，例如：

```
>>> print "3,4"
3,4
或者
>>> print(3,4)
(3,4)
```

但是在 Python 3.x 中，print 语句没有了，取而代之的是 print 函数，例如：

```
>>> print(3,4)
3 4
```

如果还像 Python 2.x 中那样使用 print 语句，Python 编译器就会报错，例如：

```
>>> print "3,4"
  File "<stdin>", line 1
    print "3,4"
          ^
SyntaxError: Missing parentheses in call to 'print'
```

Python 3.x 默认使用 UTF-8 编码

相比 Python 2.x 默认采用的 ASCII 编码，Python 3.x 默认使用 UTF-8 编码，可以很好地支持中文或其它非英文字符。

例如，输出一句中文，使用 Python 2.x 和 Python 3.x 的区别如下：

```
#Python 2.x
>>>str ="C 语言中文网"
>>>str
'C\xe8\xaf\xad\xe8\xa8\x80\xe4\xb8\xad\xe6\x96\x87\xe7\xbd\x91'
```

```
#Python 3.x
>>>str ="C 语言中文网"
>>>str
'C 语言中文网'
```

不仅如此，在 Python 3.x 中，下面的代码也是合法的：

```
>>>中国="China"
>>>print(中国)
China
```

Python 3.x 除法运算

和其他语言相比，Python 的除法运算要高端很多，它的除法运算包含 2 个运算符，分别是 / 和 //，这 2 个运算符在 Python 2.x 和 Python 3.x 的使用方法如下：

/ 运算符

在 Python 2.x 中，使用运算符 / 进行除法运算的方式和 Java、C 语言类似，整数相除的结果仍是一个整数，浮点数除法会保留小数点部分，例如：

```
>>>1/2
0
```

```
>>>1.0/2

0.5
```

但是在 Python 3.x 中使用 / 运算符，整数之间做除法运算，结果也会是浮点数。例如：

```
>>>1/2

0.5
```

运算符 //

使用运算符 // 进行的除法运算叫做 floor 除法，也就是输出不大于结果值的一个最大的整数（向下取整）。此运算符的用法在 Python 2.x 和 Python 3.x 中是一样的，举个例子：

```
#Python 2.x

>>> -1//2

-1



#Python 3.x

>>> -1//2

-1
```

Python 3.x 异常

在 Python 3.x 版本中，异常处理改变的地方主要在以下几个方面：

1. 在 Python 2.x 版本中 ,所有类型的对象都是直接被抛出的 ,但是在 Python 3.x 版本中，只有继承 BaseException 的对象才可以被抛出。
2. 在 Python 2.x 版本中 ,捕获异常的语法是“except Exception ,var:” ;但在 Python 3.x 版本中，引入了 as 关键字，捕获异常的语法变更为 “except Exception as var:”。
3. 在 Python 3.x 版本中，处理异常用 “raise Exception(args)” 代替了 “raise Exception，args”。
4. Python 3.x 版本中，取消了异常类的序列行为和 .message 属性。

有关 Python 2.x 版本和 Python 3.x 版本处理异常的示例代码如下所示：

```
#Python 2.x
>>> try:
...   raise TypeError,"类型错误"
... except TypeError,err:
...   print err.message
...
类型错误


#Python 3.x
>>> try:
...     raise TypeError("类型错误")
```

```
... except TypeError as err:
...       print(err)
...
类型错误
```

Python 3.x 八进制字面量表示

在 Python 3.x 中，表示八进制字面量的方式只有一种，并且必须写成“0o1000”这样的方式，原来“01000”的方式不能使用了。举个例子：

```
#Python 2.x
>>> 0o1000
512
>>> 01000
512


#Python 3.x
>>> 01000
  File "<stdin>", line 1
    01000
        ^
SyntaxError: invalid token
```

```
>>> 0o1000

512
```

Python 3.x 不等于运算符

Python 2.x 中的不等于运算符有 2 种写法，分别为 != 和 <>，但在 Python 3.x 中去掉了 <>，只有 ！= 这一种写法，例如：

```
#Python 2.x

>>> 1!=2

True

>>> 1<>2

True



#Python 3.x

>>> 1!=2

True

>>> 1<>2

  File "<stdin>", line 1

    1<>2

      ^

SyntaxError: invalid syntax
```

Python 3.x 数据类型

Python 3.x 中对数据类型也做了改动，比如说：

- Python 3.x 去除了 long 类型，现在只有一种整形 int，但它的行为就像是 Python 2.x 版本中的 long。
- Python 3.x 新增了 bytes 类型，对应 Python 2.x 版本的八位串，定义 bytes 字面量的方法如下所示：

```
>>>b=b'China'
>>>type(b)
<type 'bytes'>
```

字符串对象和 bytes 对象可以使用 .encode() 或者 .decode()方法相互转化，例如：

```
>>>s=b.decode()
>>>s
'China'
>>>b1=s.encode()
>>>b1
b'China'
```

- Python 3.x 中，字典的 keys()、items() 和 values() 方法用返回迭代器，且之前的 iterkeys() 等函数都被废弃。同时去掉的还有 dict.has_key()，改为用 in 替代。

本节所介绍的只是 Python 3.x 的一部分改动，由于篇幅有限，这里不再具体指出，教程中涉及到时再给大家详细介绍。

10.Python 2to3：自动将 Python 2.x 代码转换成 Python3.x 代码

本节内容针对已有 Python 基础的学员，初学者可直接跳过本节内容，继续往下学习。

由于 Python 2.x 和 Python 3.x 的差别较大，因此 Python 2.x 的多数代码无法直接在 Python 3.x 环境中运行。而由于兼容性的原因，我们在网络上查找的资源多数是 Python 2.x 的代码，如果想要在 Python 3.x 环境下运行，就需要修改源代码。

针对这一问题，Python 官方提供了一个将 Python 2.x 代码自动转换为 Python 3.x 代码的小工具，它就是 2to3.py ，通过该工具可以将大部分 Python 2.x 代码转换为 Python 3.x 代码。

接下来，就给大家详细介绍一下，如何使用 2to3.py 将 Python 2.x 代码转换成 Python 3.x 代码。

假设我们现在有 Python 2.x 的代码，要将其转换成 Python 3.x 代码，需要按照一下几个步骤进行操作：

1. 找到 2to3.py 文件，该文件保存在 Python 安装路径下的 “Tools\scripts” 目录中。比如说，我们将 Python 安装在了“G:\Python\Python36”目录中，那么 2to3.py 文件则保存在 “G:\Python\Python36\Tools\scripts ” 目录中，如图 1 所示：

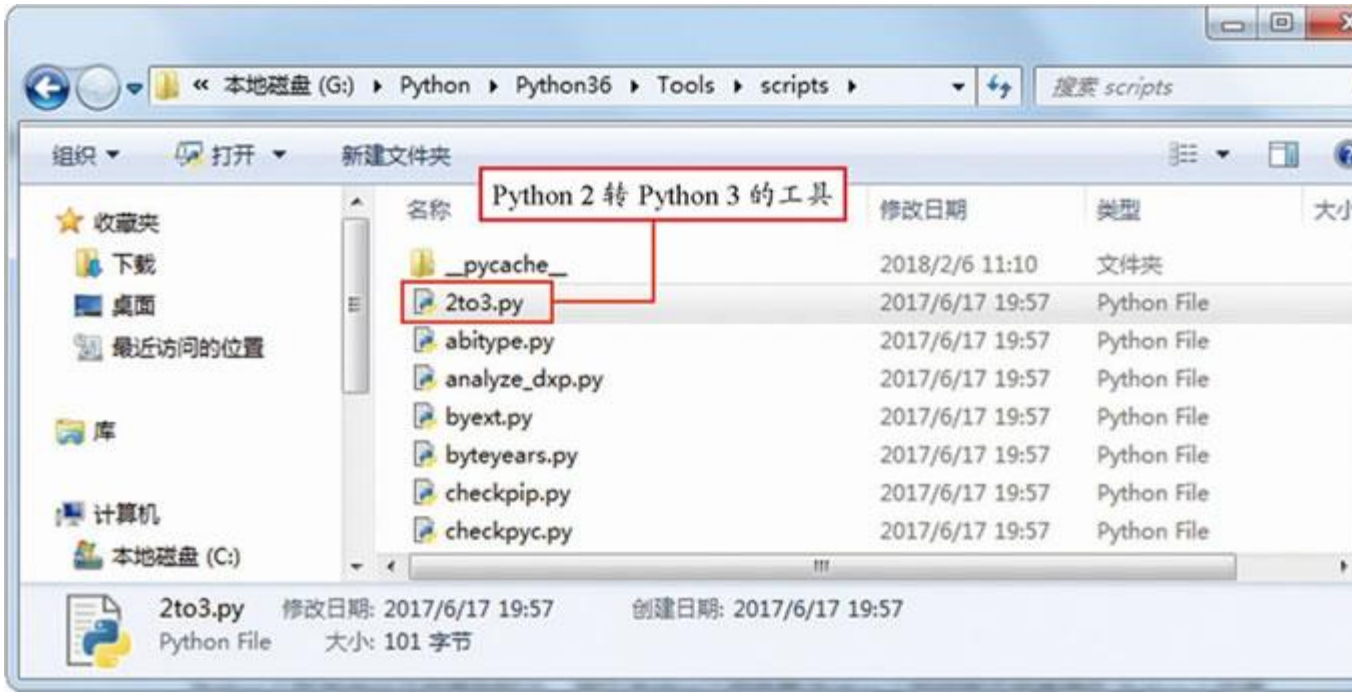


图 1 Python 2.x 转 Python 3.x 的工具

2. 将 2to3.py 文件复制到要转换代码所在的目录中。

3. 打开命令行窗口，并进入要转换代码文件所在的目录。例如，该文件保存在“E:\change”目录下，则在命令行窗口中可输入如下命令：

```
C:\users\Demo>E:

E:\>cd change

E:\change>
```

4. 调用 2to3.py 工具转化代码。例如，要转化的文件名称为 demo.py 文件，可以使用下面的代码：

```
Python 2to3.py -w demo.py
```

执行此行代码之后，将会在 “E:\change” 目录下创建一个 demo.py 文件的备份文件，名称为 demo.py.bak ,同时 ,原 demo.py 文件的内容被转换为 Python 3.x 对应的代码。

注意，在使用 2to3.py 转换 python 2.x 代码前，尽量不要把要转换的代码保存在 C 盘中，因此如果保存在 C 盘，可能会因权限问题导致转换不能正常完成。

11.作为程序猿，怎样才能及时了解 Python 的最新变化呢？

通过前面章节对 Python 语言的介绍，到目前为止，Python 已经历经了 3 个版本的迭代，读者不禁要问，Python 为什么要不断的升级呢？作为程序猿的我们，怎样才能及时了解 Python 的最新变化呢？

首先解决第一个问题，Python 不断升级的原因很简单，因为用户有了更高的需求。人们之所以设计新的编程语言 ,是因为他们发现现有的语言已经无法以最佳方式来解决问题。此外，

Python 的使用范围越来越广，人们发现它有许多可以改进的地方，应该做出这样的改进。

Python 的很多改进都是有特定应用领域的需求驱动的，其中最重要的领域是 Web 开发，这一领域需要 Python 改进对并发的处理。

还有一些是由于 Python 的历史原因导致的，随着使用 Python 的不断深入，发现了 Python 的一些不合理之处。比如，有些是标准库模块结构混乱或冗余，有些是程序设计缺陷。

那么，怎样才能及时追踪 Python 的更新动态呢？这就需要借助 Python PEP 文档。

Python PEP 文档

要知道，虽然各种各样的 Python 语句修改方案会以邮件列表的形式进行讨论，但 Python 社区有一种应对变化的固定方法，即只有发布了名为 PEP 的新文档，新的变化才会生效。

PEP（Python Enhancement Proposal），全称是 Python 改进方案。它是提交 Python 变化的书面文档，也是社区对这一变化进行讨论的出发点。值得一提得是，PEP 文档的整个目的，格式以及工作流程的标准格式，都包含 PEP 1 文档中。

PEP 文档对 Python 十分重要，它主要有以下 3 个用途：

1. 通知：汇总 Python 核心开发者重要的信息，并通过 Python 发布日程；
2. 标准化：提供代码风格、文档或者其他指导意见；
3. 设计：对提交的功能进行说明。

所有提交过的 PEP 都被汇总在 PEP 0 文档中。

需要注意的是，如果读者对 Python 语言的未来发展方向感兴趣，但苦于没有时间追踪 Python 邮件列表中的讨论，那么 PEP 0 是信息来源的不错选择，它会告诉你哪些文档已被接受但尚未实施，哪些文档仍在审议中。

不仅如此，PEP 还有其他的用途，比如说，人们常常会问以下类似的问题：

- A 功能为什么要以这样的方式运行？
- Python 为什么没有 B 功能？

多数情况下，关于该功能的某个 PEP 文档已经给出了上述问题的详细回答。

另外，还有关于 Python 语言功能的 PEP 文档并没有通过，这些文档可作为历史资料来参考。

12.Python 底层是用什么语言实现的？

确切地说，本教程介绍的是用 C 语言编写实现的 Python，又称为 CPython。平时我们所讨论的 Python，指的其实就是 CPython。

随着编程语言的不断发展，Python 的实现方式也发生了变化，除了用 C 语言实现外，Python 还有其他的实现方式。例如，用 Java 语言实现的 Python 称为 JPython，用 .net 实现的 Python 称为 IronPython 等等。

Python 的这些实现方式虽然诞生比 CPython 晚，但一直在努力地跟上主流，并在不同的生产环境中不断地使用并推广 Python。

Python 的实现方式有很多种，Python 官网上介绍了 20 多种语言变体、方言或 C 语言之外的 Python 解释器实现。其中一些只是实现了语言核心语法、功能和内置扩展的一个子集，但至少有几个与 CPython 几乎完全兼容。更重要的是，在这些不同的实现方式中，虽然有些知识玩具项目或实验，但大部分都是为了解决某些实际问题而创建的，这些问题要么使用 CPython 无法解决，要么需要开发人员花费巨大的精力，这里举几个例子：

- 在嵌入式系统中运行 Python 代码。
- 与运行框架（如 Java 或 .NET）或其他语言做代码集成。
- 在 Web 浏览器中运行 Python 代码。

由于受到篇幅的限制，本节仅给大家介绍几种 Python 开发人员最常用的几种 Python 实现方式。

Stackless Python

Stackless Python 自称 Python 增强版。之所以名为 Stackless（无栈），是因为它没有依赖 C 语言的调用栈，实际上就是对 CPython 做了一些修改，添加了一些新的功能。

在新添加的功能中，最重要就是由解释器管理的微线程，用来替代依赖系统内核上下文切换和任务调度的普通线程，既轻量化又节约资源。

Stackless Python 最新可用的版本是 2.7.9 和 3.3.5 ,分别实现的是 Python 2.7 和 3.3。在 Stackless Python 中，所有的额外功能都是内置 stackless 模块内的框架。

Stackless Python 并不是最有名的 Python 实现，但很值得一提，因为它引入的思想对编程语言社区有很大的影响。例如，将 Stackless Python 中的内核切换功能提取出来并作为一个独立包发布，名为 greenlet，是许多有用的库和框架的基础。

此外，Stackless Python 的大部分功能都在 PyPy 中重新实现，PyPy 是另一个 Python 实现，我们将稍后介绍。

JPython

Jython 是 Python 语言的 Java 实现。它将代码编译为 Java 字节代码，开发人员在 Python 模块中可以无缝使用 Java 类。

Jython 允许人们在复杂应用系统（例如 J2EE）中使用 Python 作为顶层脚本语言，它还将 Java 应用引入到 Python 中，一个很好的例子就是，在 Python 程序中可以使用 Apache Jackrabbit（这是一个基于 JCR 的文档仓库 API）。

Jython 最新可用的版本是 Jython 2.7，对应的是 Python 2.7 版。它宣称几乎实现了 Python 所有的核心标准库，并使用相同的回归测试套件。Jython 3.x 版正在开发中。

Jython 与 CPython 实现的主要区别如下所示：

- 真正的 Java 垃圾回收，而不是引用计数。
- 没有全局解释器锁（GlobakInterpreter Lock，GIL），在多线程应用中可以充分利用多个内核。

这一语言实现的主要缺点是缺少对 C/Python 扩展 API 的支持，因此用 C 语言编写的 Python 扩展在 Jython 中无法运行。这种情况未来可能会发生改变，因为 Jython 3.x 计划支持 C/Python 扩展 API。

某些 Python Web 框架（例如 Pylons）被认为是促进 Jython 的开发，使其可用于 Java 世界。

IronPython

IronPython 将 Python 引入 .NET 框架中，这个项目受到微软的支持，因为 IronPython 的主要开发人员都在微软工作。可以这么说，IronPython 是推广语言的一种重要实现。

除了 Java，.NET 社区是最大的开发者社区之一。

值得一提的是，微软提供了一套免费开发工具，名为 PTVS（Python Tools for Visual Studio，用于 Visual Studio 的 Python 工具），可以将 Visual Studio 转换为成熟的 Python IDE。这是作为 Visual Studio 的插件发布的，在 GitHub 可以找到其开源代码。

IronPython 最新的稳定版本是 2.7.5，与 Python 2.7 兼容。与 Jython 类似，Python 3.x 的实现也在开发中，但还没有可用的稳定版本。

虽然 .NET 主要在微软 Windows 系统上运行，但是 IronPython 也可以在 Mac OS X 和 Linux 系统上运行，这一点要感谢 Mono，一个跨平台的开源 .NET 实现。

与 CPython 相比，IronPython 的主要区别或优点如下：

- 与 Jython 类似，没有全局解释器锁（Global Interpreter Lock，GIL），在多线程应用中可以充分利用多个内核。
- 用 C# 和其他 .NET 语言编写的代码可以轻松集成到 IronPython 中，反之亦然。
- 通过 Silverlight，在所有主流 Web 浏览器中都可以运行。

说到弱点，IronPython 也与 Jython 非常类似，因为它也不支持 C/Python 扩展 API。对于想要使用主要基于 C 扩展的 Python 包（例如 NumPy）的开发人员来说，这一点很重要。

有一个叫作 ironclad 的项目，其目的是在 IronPython 中无缝使用这些扩展，其最新支持的版本是 2.6，开发已经停止。

PyPy

PyPy 可能是最令人兴奋的 Python 实现，因为其目标就是将 Python 重写为 Python。在 PyPy 中，Python 解释器本身是用 Python 编写的。

在 Python 的 CPython 实现中，有一个 C 代码层来实现具体细节。但在 PyPy 实现中，这个 C 代码层用 Python 完全重写。这样，你可以在代码运行期间改变解释器的行为，并实现 CPython 难以实现的代码模式。

目前 PyPy 的目的是与 Python 2.7 完全兼容，而 PyPy3 则与 Python 3.2.5 版兼容。

以前对 PyPy 感兴趣主要是理论上的原因，只有喜欢深入钻研语言细节的人才会对它感兴趣。PyPy 通常不用于生产环境，但这些年来这种状况已经发生改变，PyPy 通常比 CPython 实现要快得多。基于这一特性，使得越来越多的开发人员决定在生产环境中切换到 PyPy。

PyPy 与 CPython 实现的主要区别在于以下几个方面：

- 使用垃圾回收，而不是引用计数。
- 集成跟踪 JIT 编译器，可以显著提高性能。
- 借鉴 Stackless Python 在应用层的无栈特性。

与几乎所有其他的 Python 实现类似，PyPy 也缺乏对 C/Python 扩展 API 的完全官方支持。但它至少通过 CPyExt 子系统为 C 扩展提供了某种程度的支持，虽然文档不完整，功能也尚未完善。此外，社区正在努力将 NumPy 迁移到 PyPy 中，因为这是最需要的功能。

13.了解 Jupyter Notebook，你已然超越了 90%的 Python 程序员

在 2019 年 8 月份 TOIBE 编程语言社区公布的编程语言排行榜中，Python 已经超过了 C++，稳居排行榜第 3 名（如图 1 所示）。

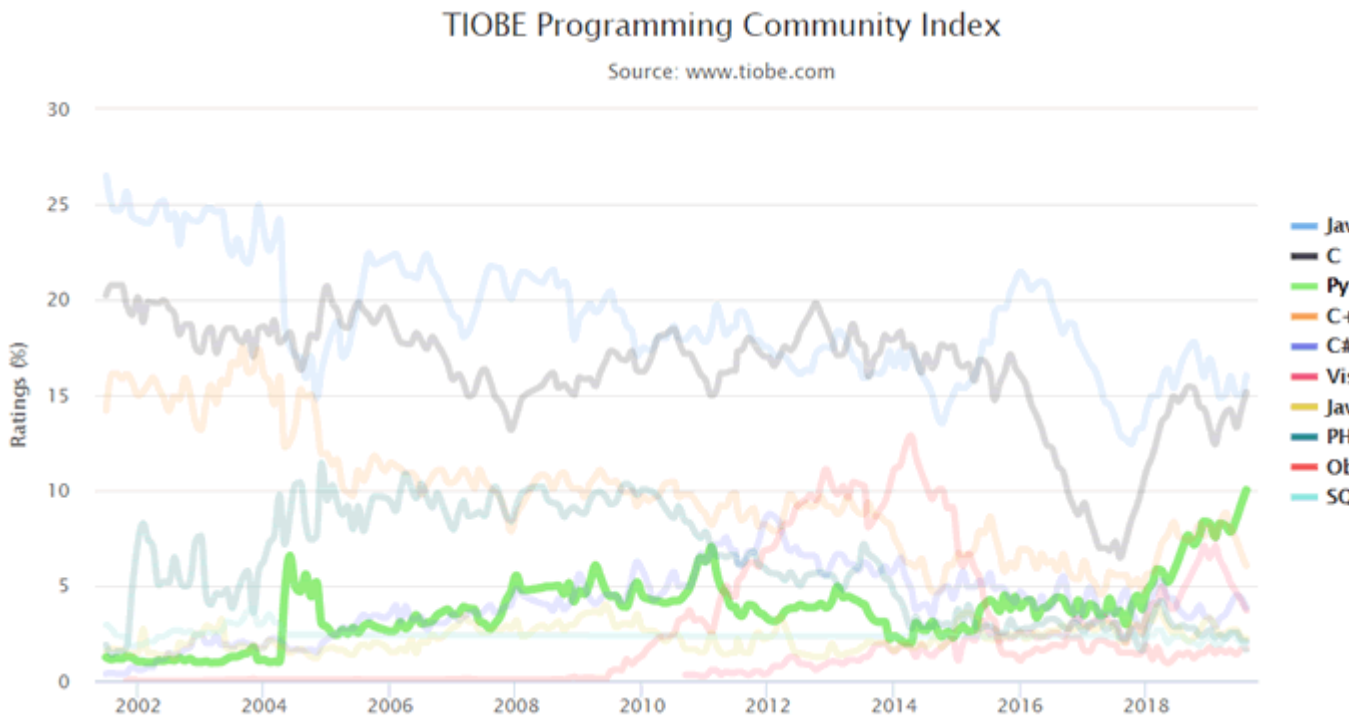


图 1 TOIBE 编程社区排行榜

我们应该知道，Python 之所有在 14 年后“崛起”，得益于机器学习和数学统计应用的兴起。至于 Python 适合数学统计和机器学习的原因，很多读者可能并不会想到，Jupyter Notebook 功不可没。可以好不夸张地说，作为一名 Python 工程师，如果不会使用 Jupyter Notebook，可能会真的太落伍了。

本节，就带领大家学习一下 Jupyter Notebook。

什么是 Jupyter Notebook

说了这么多，到底什么是 Jupyter Notebook 呢？按照 Jupyter 创始人的说法，起初他是想做一个综合 Ju（Julia）、Py（Python）和 R 三种科学运行语言的计算工具平台，所以将其命名为 Jupyter。

Jupyter 发展到现在，已经成为了一个几乎支持所有语言，能够把软件代码、计算输出、解释文档、多媒体资源整合在一起的多功能科学运行平台。Jupyter Notebook 的工作界面如图 2 所示。

图 2 Jupyter Notebook 界面（点此查看高清大图）

如图 2 所示，只要直接输入代码并运行，它就会直接在代码下面显示输出结果。那么，

Jupyter Notebook 对 Python 的影响真的有那么大吗？

Jupyter Notebook 的影响力

衡量一个技术的影响力，或者说想利用自己的技术影响世界时，必定绕不开技术对教育界的

影响力。

以微软的 Word 文本处理系统为例，从纯技术角度来讲，它的单机设计理念早已过时，但以 Google Doc 为代表的在线文档系统，并没有想象中实现对 Word 的降维打击。最直接的原因归咎于用户习惯，多数用户已经习惯使用 Word 来编辑、修改文档啦，之所以会这样，是因为我们从小学、中学，一直到大学期间，都在学习使用 Word。而且到了工作中，老员工还会带着新员工继续使用 Word。

从 2017 年开始，已经有大量的北美顶尖计算机课程开始使用 Jupyter Notebook 作为教学工具，而在这之前，一直使用的都是 Python 命令行的形式。不仅如此，Jupyter Notebook 在工业界的影响力更大。

例如，在 Facebook 中，几乎所有的中小型程序（比如内部的线下分析软件，机器学习模块的训练等）都是借助与 Jupyter Notebook 完成。同时，在硅谷其他的一线大厂中，也全部使用 Jupyter Notebook（他们用的是改进定制型的 Jupyter Notebook，名称 Google Colab）。

了解了以上这些，相信你已经认可了 Jupyter Notebook 的江湖地位。

但需要注意的是，不是说那个技术流行，就盲目跟从。对于技术的选择，要学习独立的思考，切勿人云亦云。以 Facebook 为例，之所以它选择 Jupyter Notebook 等技术，很大程序上因为它有几百个产品线，几万个工程师，而如果是几个人的团队，使用同样的技术反倒

成了拖累。

Jupyter Notebook 的优势

这里给大家总结了几点 Jupyter Notebook 的优势：

1）整合了所有资源

在软件开发过程中，频繁地进行上下文切换，会影响生产效率。举个例子，假设你需要切换窗口去看一些文档，再切换窗口去用另一个工具画图，不断地切换窗口就会成为影响效率的因素。

而 Jupyter Notebook 则不同，它会将所有和软件编写的资源全部放在一个地方，无需切换窗口就可以轻松找到。

2）交互性编程体验

在机器学习和数据统计领域，Python 编程的实验性特别强，比如为了测试 100 种不同的方法，有时就需要将一小块代码重写 100 遍，在这种情况下，如果使用传统的 Python 开发流程，每一次测试都要将所有代码重新跑一遍，会花费开发者很多时间。

Jupyter Notebook 引进了 Cell 的概念。每次测试可以只跑一小块的代码，并且在代码下方立刻就能看到运行结果。

如此强的交互性，满足了 Python 程序员可以专注于问题本身，不会被频繁的工具链拖累，

也不用在命令行之间来回切换，所有工作都能在 Jupyter Notebook 上完成。

3）轻松运行他人编写的代码

同样是在机器学习和数学统计领域，我们可能会借鉴他人分享的代码，但当我们拷贝过来想要运行时，却需要使用 pip 安装一大堆依赖的库，足以让人抓狂。而 Jupyter Notebook 就可以解决这个问题。

例如，Jupyter 官方的 Binder 平台以及 Google 提供的 Google Colab 环境，它们可以让 Jupyter Notebook 变得和 Google Doc 在线文档一样。比如用 Binder 打开一份 GitHub 上的 Jupyter Notebook 时，就不需要安装任何 Python 库，直接在打开代码就能运行。

本节，仅是为了让初学者对 Jupyter Notebook 有一个初步的了解，具体 Jupyter Notebook 的安装和使用，网络上有很多详细的教程，这里不再做过多描述。

注意，本教程仍旧使用 Python 内置的 IDLE 作为教学工具。

第 2 章 Python 初探

系统学习 Python 之前，首先要了解它的语法特点和编程规范。
本章将围绕第一个 Python 入门程序，介绍 Python 编程过程中需要遵循的注释规则、代码缩进、编程规范等。
和其他高级编程语言相同的是，Python 也有一套自己的关键字和标识符命令规范，不同之处在于，Python 是以冒号和缩进作为区分代码之间层次的。

1.Windows 安装 Python

使用任何高级编程语言都需要有一个自己的编程环境，Python 也不例外。只要使用 Python，就必须要配置 Python 的开发和运行环境。

Python 本身有非常优秀的可移植性，本节我们只介绍如何在 Windows 系统上安装 Python 环境，后续章节还会介绍如何在 Linux 以及 Mac OS X 上安装 Python 环境。

在 Windows 系统上安装 Python，可以按如下步骤进行。

首先，登录 https://www.python.org/downloads/ 页面，可以在该页面上看到两类下载链接，分别是 Python 3.x 和 Python 2.x 版本。因为 Python 在同时维护着 3.x 和 2.x 两个版本，这样既可让早期项目继续使用 Python 2.x，也可让新的项目使用 Python 3.x 。

在该页面下方的“Looking for a specific release？”列表中选择“Python 3.6.x”，可以看到如图 1 所示的下载列表。

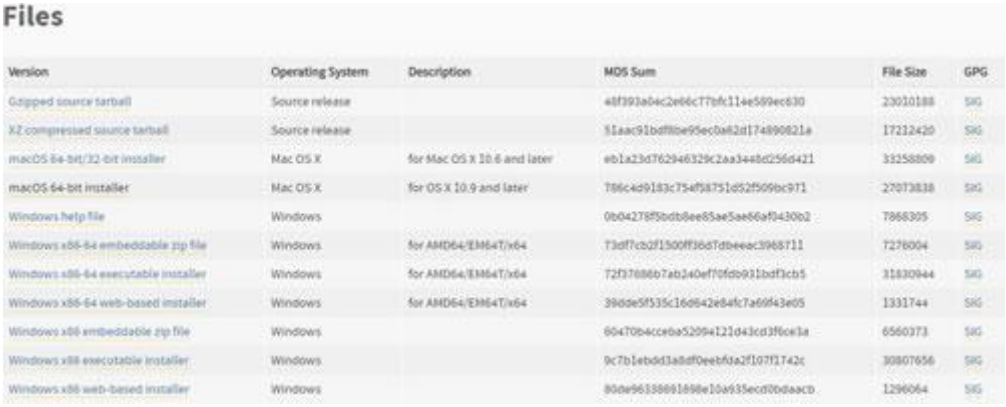


图 1 下载 Python 的安装程序（点此查看高清大图）

在下载列表中，以“Windows x86-64”开头的链接才是 64 位的 Python 安装程序；以“Windows x86”开头的链接是 32 位的 Python 安装程序。

根据 Windows 系统平台下载合适的安装程序（64 位的平台下载 64 位的安装程序，32 位的平台下载 32 位的安装程序），本教程以 64 位的安装程序为例，下载完成后得到 python-3.6.x-amd64.exe 安装文件。

如果在“Looking for a specific release？”列表中选择“Python 2.7.x”，也可以看到类似如图 1 所示的下载页面，通过这种方式下载平台对应的 Python 2.7.x 安装程序，下载完成后会得到一个 python-2.7.x.amd64.msi 安装文件。

双击 python-3.6.x-amd64.exe 文件，系统将会开启 Python 安装向导，如图 2 所示。

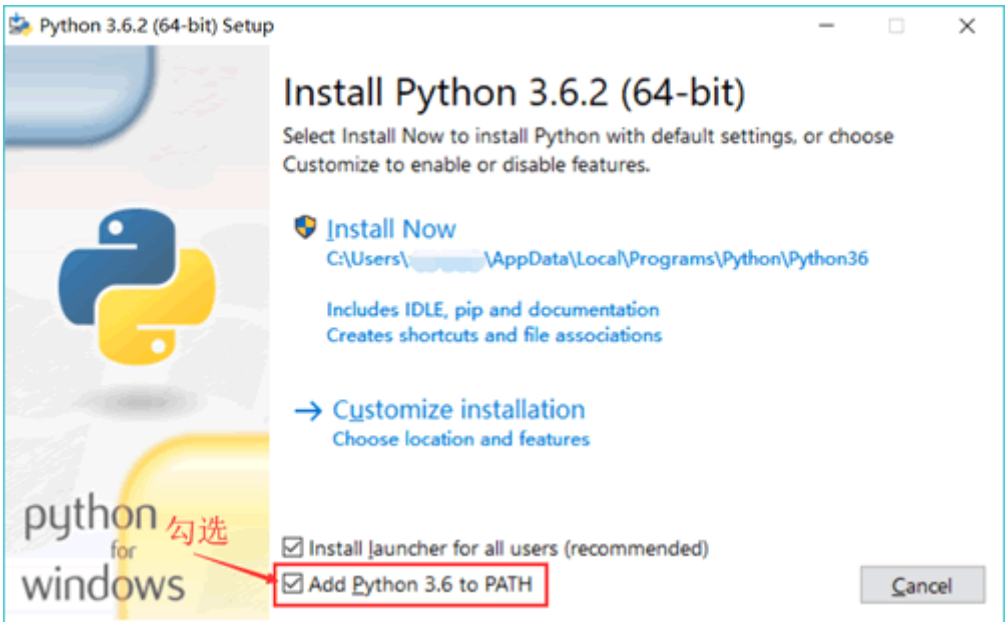


图 2 Python 安装向导

勾选“Add Python 3.6 to PATH” 复选框，可以将 Python 命令工具所在目录添加到系统 Path 环境变量中；单击“Customize installation” ，可以在安装时指定自定义的安装路径。单击该选项即可开始安装。

在接下来的安装界面中只要改变 Python 的安装路径（建议安装在指定盘符的根路径下，比如安装在 D:\Python\ 目录下），其他地方使用默认选项即可。

安装完成后，启动 Windows 的命令行程序，在命令行窗口中输入“python”命令（字母 p 是小写的），如果出现 Python 提示符（>>>），就说明安装成功了，如图 3 所示。

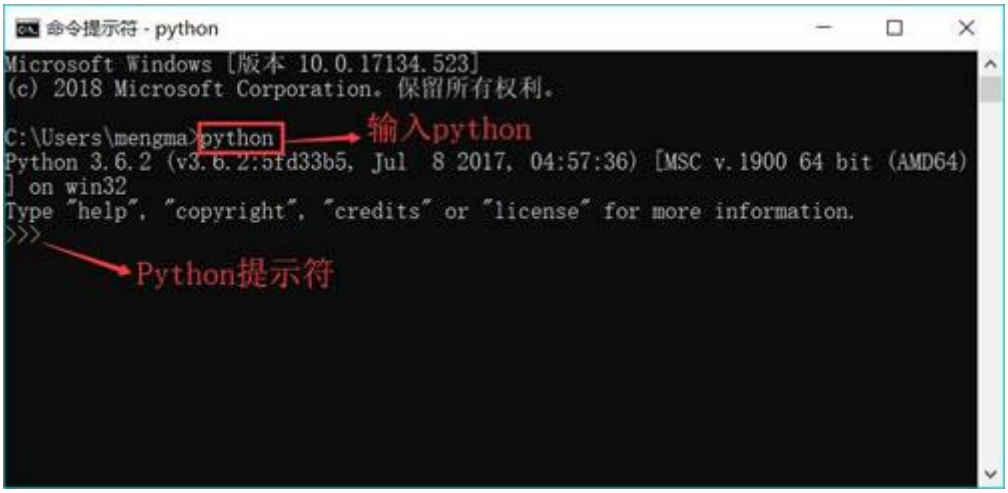


图 3 Python 安装成功示意图

使用 python 命令启动的就是 Python 的交互式解释器，如果要退出交互式解释器，则可按“Ctrl+Z”快捷键或使用 exit() 命令。

使用 python-2.7.x.amd64.msi 文件安装 Python 解释器，同以上方式完全一样，根据选择安装其中一个即可。

需要额外说明的是，在 Windows 的“开始”菜单中，也可找到 Python 3.x（或者 Python 2.x）的菜单组，在这些菜单组中可找到 Python 提供的 IDLE 工具（如图 4 所示），该工具是一个简易开发环境，提供了简易的 Python 编辑工具，编辑完成后按 F5 键即可运行 Python 程序。

图 4 Python IDLE 简易开发环境

2.Linux 安装 Python

在通常情况下，Linux 系统默认自带了 Python 开发环境。下面以 Ubuntu（Linux 发行版之一）为例来介绍在 Linux 系统上安装 Python 的步骤。

只要安装了 Ubuntu 这个操作系统，默认就已安装有 Python 开发环境（最新版的 Ubuntu 中可能预装了 Python 的 2 个版本）。我们可以通过系统的 Terminal（可通过“Ctrl+Alt+T”快捷键）启动命令窗口，在该命令行窗口中输入 python 命令（注意，字

母 p 是小写的）进行校验：

```
$ python
Python 2.7.12 (default, Dec 4 2017, 14:50:18)
[GCC 5.4.0 20160609] on linux2
Type "help","copyright","credits" or "license" for more information.
>>>
```

可以看到，执行结果最后显示了 Python 提示符（>>>），这表明此 Ubuntu 系统上已经存在 Python 2 开发环境，因此，我们可以直接执行 python 命令来启动 Python 2 开发环境。

使用 python 命令启动的是 Python 的交互式解释器，如果希望退出该交互式解释器，则可按“Ctrl+D”快捷键或使用 exit() 命令。

如果要检查在 Ubuntu 系统上是否安装了 Python 3，则可在 Terminal 命令行窗口中输入 python3 命令。

```
$ Python3
Python 3.5.2 (default , Nov 23 2017 , 16:37:01)
[GCC 5.4.0 20160609] on linux2
Type "help","copyright","credits" or "license" for more information.
>>>
```

如果上面的命令行同样显示了 Python 提示符（>>>），则表明该 Ubuntu 系统上已经存在 Python 3 开发环境，只需执行 python 3 命令就可以启动 Python 3 开发环境。

如果感觉 Ubuntu 系统内置的 Python 3 版本不够新，或者希望安装指定版本的 Python 交互式解释器，那么可以通过执行如下 2 条命令更新 Python 版本：

```
$sudo apt-get update
$sudo apt-get install python3.6
```

第一条命令用来指定更新 /etc/apt/sources.list 和 /etc/apt/sources.list.d 所列出的源地址，这样保证能获得最新的软件包；第二条命令则指定安装 Python 3.6。

在成功执行上面的命令之后，再次在 Temlinal 命令行窗口中输入 python3 命令，即可看到 Python 3 交互式解释器更新到 Python 3.6。

有读者问，每次使用 Python3 命令比较麻烦，能否使用 Python 命令调用 Python 3 环境呢？当然没问题，只需执行如下 2 行命令即可：

```
$sudo unlink /usr/bin/python
$sudo ln -s /usr/bin/python3.6 /usr/bin/python   <--注意，这里 Python 3 环境的路径以及版本一定要正确
```

由此，当你再次使用 Python 命令时，出现的就是 Python 3 交互解释器啦。

以上仅是在 Ubuntu 系统已有 Python 的情况下，除此之外，如果读者要重新安装 Python 环境，则可以在终端模式下按照顺序运行如下命令：

1. 到 Python 的官方网站上下载源码，执行如下命令：

```
$ wget https://www.python.org/ftp/python/3.6.2/Python-3.6.2.tgz
```

2. 解压源码包：

```
$ tar -zxvf Python-3.6.2.tgz
```

3. 编译：

```
$ ./configure  --prefix=/usr/local

$ make&&sudo make install
```

这里的 --prefix=/usr/local 用于指定安装目录（建议指定）。如果不指定，就会使用默认的安装目录。

经过以上几步命令，我们就安装好了 Python，这时就可以进入终端，输入 Python 指令，验证是否已安装成功。

3.Mac 安装 Python 环境

最新版的 Mac OS X 系统通常已经安装了 Python 2 ，为了检查系统中是否已安装 Python，可以启动系统中的终端窗口（Terminal），在该窗口中输入“python”命令，将会看到系统提示 Python 2.x 已安装成功：

```
$ python
Python 2.7.10 (default, Jul 15 2017, 17:16:57)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.31)] on darwin
Type "help","copyright","credits" or "license" for more information.
```

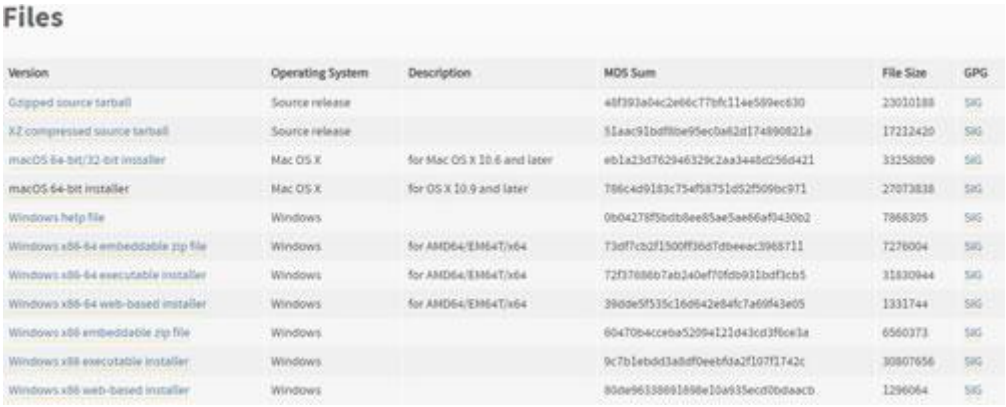
```
>>>
```

从上面的运行结果可以看出，该 Mac OS X 系统已经安装了 Python 2.7.10。

如果要检查 Mac OS X 上是否安装了 Python 3，可以在终端窗口中输入“python3” 命令，如果系统提示“command not found” ，则表明该系统暂未安装 Python 3。

在 Mac OS X 上安装 Python 3 可以按如下步骤进行。

登录 https://www.python.org/downloads/ 页面，在该页面下方的“Looking for a specific release？” 列表中选择 “Python 3.6.x” ，可以看到如图 1 所示的下载列表。



Files

Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		4bf393a04c2e66cf7bfc114e589ec630	23010188	SIG
XZ compressed source tarball	Source release		51aac91bdf8be95ecba62df174890821a	17212420	SIG
macOS 64-bit/32-bit installer	Mac OS X	for Mac OS X 10.6 and later	eb1a23d76294632 9c2aa344bd256d421	33258809	SIG
macOS 64-bit installer	Mac OS X	for OS X 10.9 and later	786c4d9183c754f58751d52f509bc971	27073838	SIG
Windows help file	Windows		0b04278f5bdb8ee85ae5ae66af0430b2	7868305	SIG
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64	73df7cb2f1500ff36d7dbeeac3968711	7276004	SIG
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64	72f37686b7ab1a0ef70fdb931bdf3cb5	31830944	SIG
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	39dde5f535c16d642e84fc7a69f43e05	1331744	SIG
Windows x86 embeddable zip file	Windows		60470b4cce6a5209412 1d43cd3f6ce1a	6560373	SIG
Windows x86 executable installer	Windows		9c7b1ebdd3a8df0eebfda2f107f1742c	30807656	SIG
Windows x86 web-based installer	Windows		80de96138691898e10a935ecd0bdaacb	1296064	SIG

图 1 下载 Python 的安装程序（点此查看高清大图）

在下载列表中单击以“Mac OS X”开头的链接（在 Mac OS X 平台上 Python 安装程序同时兼容 32 位和 64 位系统），下载完成后得到一个 python-3.6.4-macosx10.6.pkg 安装包。

双击 python-3.6.4-macosx10.6.pkg 文件，系统将会开启 Python 安装向导，按照安装向导一步一步向下安装，一切保持默认即可。

安装完成后，电脑上将同时存在 Python 3.x 和 Python 2.x 的运行环境，直接在命令行窗口中输入“python”命令将会运行 Python 2；在命令行窗口中输入“python3”命令将会运行 Python3。

```
$ python3
Python 3.6.4 (v3.6.4:d48ecebad5, Dec 18 2017, 21:07:28)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help","copyright","credits" or "license" for more information.
>>>
```

与 Windows 平台类似的是，安装 Python 3 之后，在 Mac OS X 的程序列表中同样会多出一个 IDLE 工具，该工具就是 Python 3 的简易开发环境。

4.python 不是内部或外部命令的解决方法

> 安装 Python 环境没有问题的读者，可直接跳过本节，继续阅读后续文章。

有些读者，在命令行窗口（Linux 称为终端）中输出 python 命令后，却显示“‘python’不是内部或外部命令，也不是可运行的程序或批处理文件”，如图 1 所示：

图 1 输出 python 命令后出错

出现此问题，是因为在当前的路径中找不到 Python.exe 可执行程序，最直接的解决方法就是手动配置环境变量，具体方法如下：

1. 在“计算机”图标上单击右键，然后在弹出的快捷菜单中选择“属性”，在属性对话框中单击“高级系统设置”超链接，会出现如图 2 所示的“系统属性”对话框。

图 2 “系统属性”对话框示意图

2. 如图 2 所示，单击“环境变量”按钮，将弹出“环境变量”对话框，如图 3 所示：

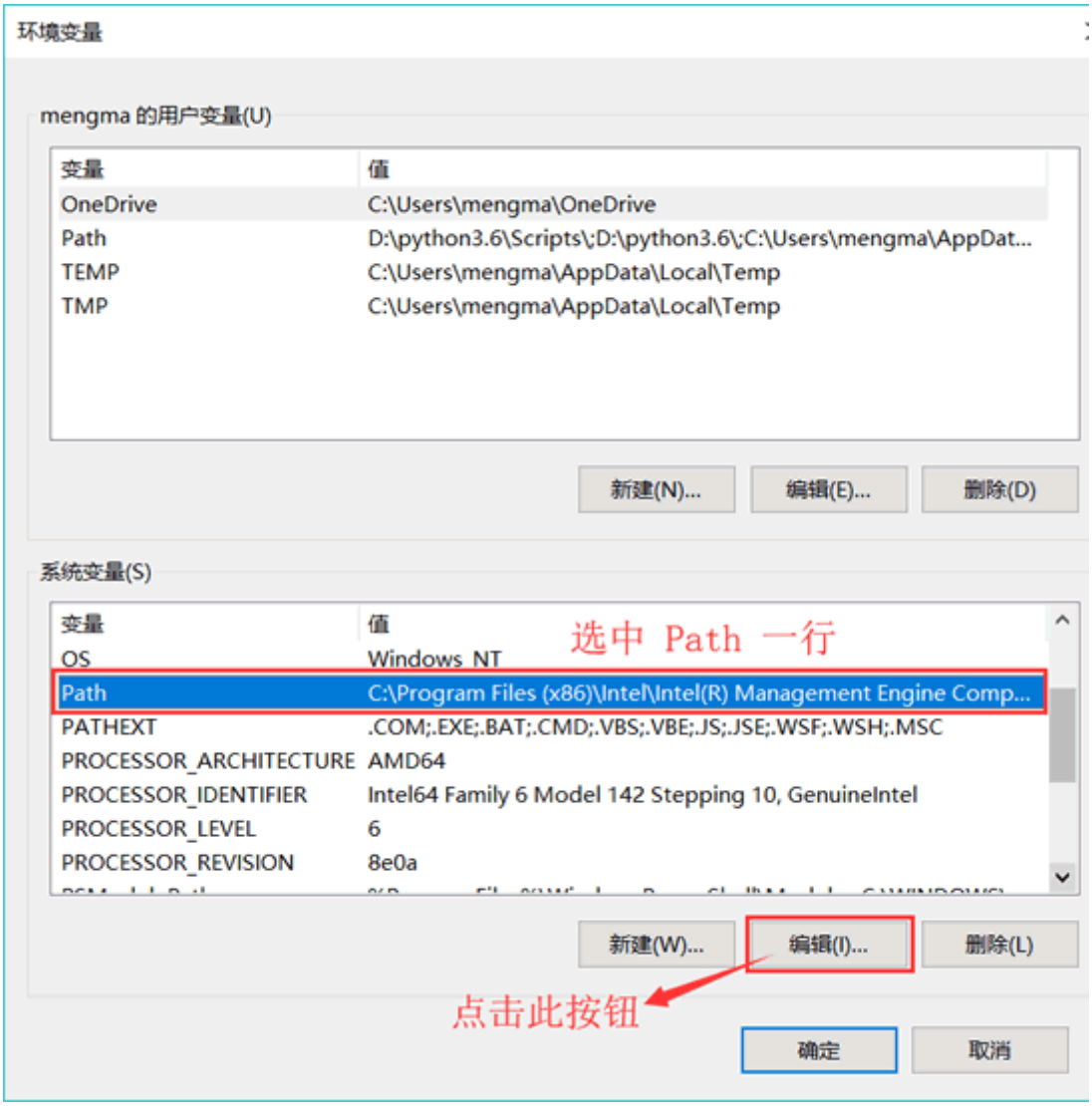


图 3 “环境变量”对话框

3. 在图 3 所示的对话框中，选中“系统变量”栏中的 Path 变量，然后单击“编辑”按钮。此时会弹出“编辑系统变量”的对话框，如图 4 所示：

图 4 “编辑系统变量”对话框

4. 正如图 4 所示的这样，我们需要在现有 Path 变量值的最前端，手动添加“G:Python\Python36;G:Python\Python36/Scripts;”，注意，中间和最后的分号不要丢掉，另外将 Python 安装路径修改成自己的，然后单击确定按钮，就完成了对环境变量的设置。

> 再次强调，不要删除系统变量 Path 中原本存在的变量值，并且其中的分号是英文状态下输出的，否则会产生错误。

修改完成后，再在命令行窗口中输入 python 命令，就应该可以成功进入 Python 交互式解释器。

> 本节所介绍的解决方法，是以 Windows 操作系统为例进行说明的，但此方式同样适用于其他的操作系统，只是修改过程略有差异，使用其他操作系统的读者，需要自己找到修改 Path 变量的入口，然后按照此方式修改即可。

5.第一个 Python 程序

无论读者使用的是哪种操作系统，相信都已经安装好了 Python 环境，可以通过命令行窗口或者 Python 自带的 IDLE 成功启动交互式解释器（如图 1 所示）。

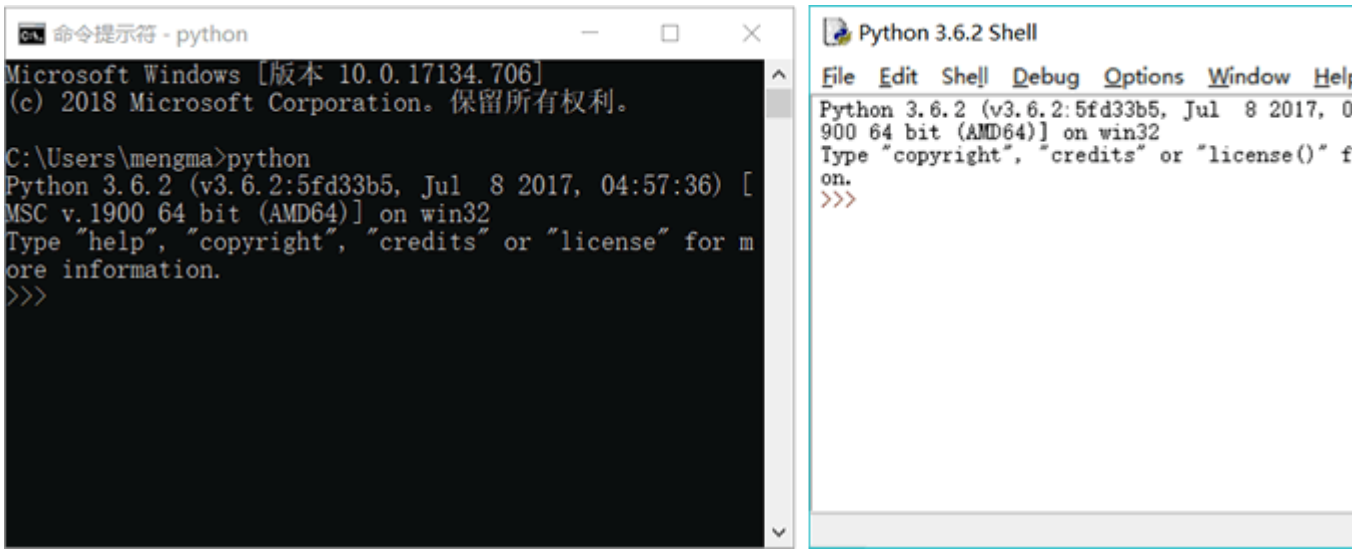


图 1 启动 Python 交互式解释器

本节将带领读者正式编写第一个 Python——输出“Hello World”。

在图 1 中的提示符 >>> 的由此，直接输入如下代码：

```
print ("Hello World")
```

这行 Python 程序的功能就是在屏幕（不是打印机）上输出字符串“Hello World”。需要注意的是，在这行代码中，小括号和双引号都必须在英文半角状态下输入，且 print 全部为小写字母，因为 Python 语言是严格区分大小写字母的。

Python 程序不要求语句使用分号结尾，当然也可以使用分号，但并没有实质的作用（除非同一行有更多的代码），而且这种做法也不是 Python 推荐的。

如果读者使用的 Python 2.x 环境，那么上面代码也可以写成如下形式：

```
print "Hello World"  <--Python 3 中不再支持这种写法
```

输入完成后，按下回车（Enter）键，可以看到如图 2 所示的输出结果。

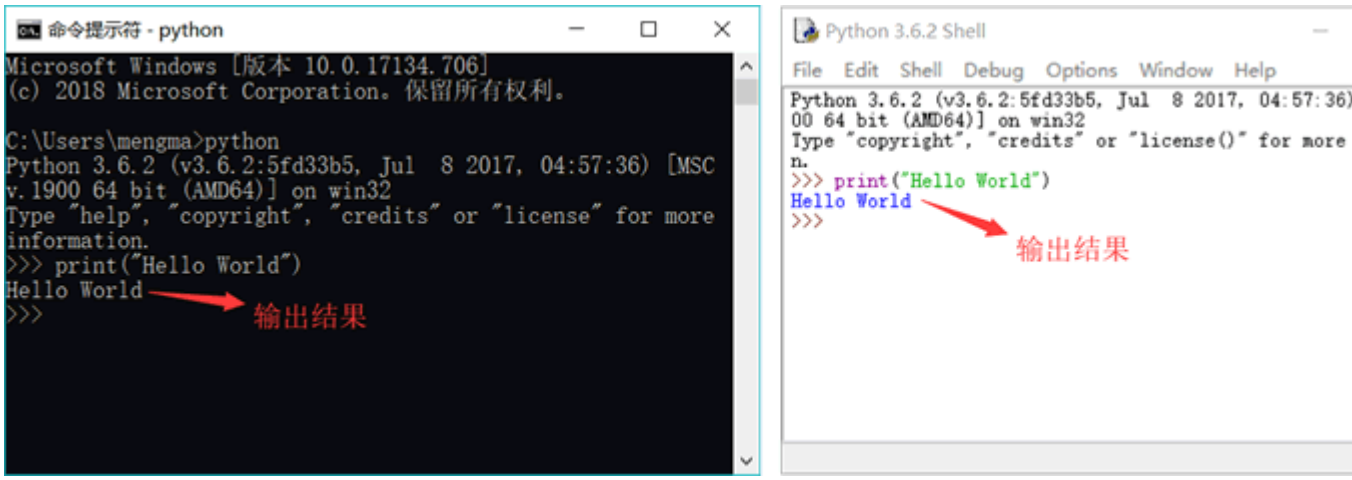


图 2 程序输出结果

文本编辑器编写并执行 Python 程序

如果读者不习惯直接将程序卸载命令行窗口或 IDLE 中，也可使用任何熟悉的无格式文本编辑器编写 Python 程序，比如说，在 Windows 操作系统上可使用 EditPlus、Notepad++、UltraEdit 等程序；在 Linux 平台上可使用 vim、gedit 等工具；在 Mac OS X 上可使用 TextEdit、Sublime Text 等工具。

> 注意，编写 Python 程序不要使用写字板，更不可使用 Word 等文档编辑器。因为写字板、Word 等工具是有格式的编辑器，当使用它们编辑一份文档时，这个文档中会包含一些隐藏的格式化字符，这些隐藏的字符会导致程序无法正常编译、运行。

使用文本编辑器编写 Python 程序，首先要新建一个文本文件，假设将此文件命名为 hello_world.py（无论名称是什么，.py 后缀名不能改变），并输入如下代码：

```
print ("Hello World")
```

编写完成后将文件保存，则该文件就是 Python 程序的源程序文件。

运行源文件中的 Python 程序有 2 种方法：

1. 使用 IDLE 工具，通过“file -> open ” 菜单打开 hello_world.py 源文件，然后在源文件中的菜单栏中，选择“Run->Run Module”（或直接按 F5 键），就可以执行文件中的程序了。

运行后的效果如图 3 所示。有关 IDLE 的具体用法，可阅读《Python IDLE 使用方法》一节。

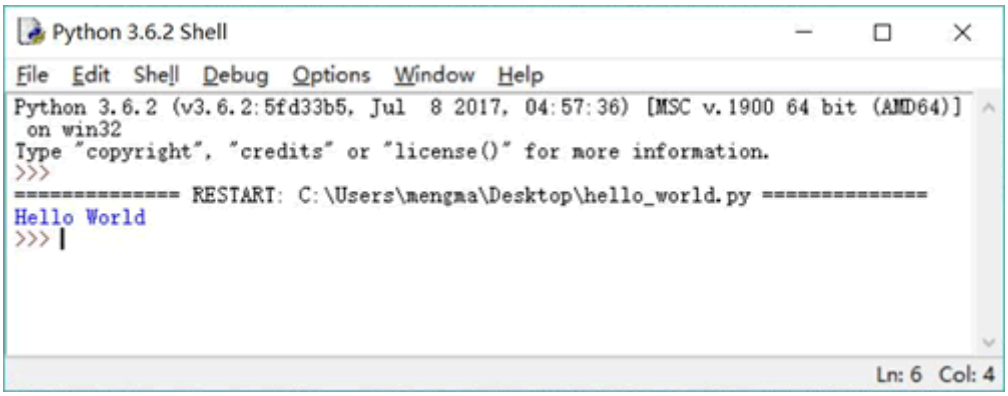


图 3 使用 IDLE 运行 Python 程序

2. 启动命令行窗口，并进入 hello_world.py 文件所在的位置，然后在命令行窗口中输入如下命令：

```
python hello_world.py
```

运行上面的命令，如果看到以下输出结果，则证明程序运行成功：

```
Hello World
```

这里简单介绍一下 python 命令，它的语法非常简单，其基本格式如下：

```
python <源程序路径>
```

这里的源程序路径，可以是自盘符（C 盘、D 盘）开始写的绝对路径，也可以在执行 python

命令之前，先进行目标程序文件所在的路径，然后只传递文件名（使用相对路径）。例如，hello_world.py 文件的完整路径为 D:\hello_world.py，图 4 分别演示了使用这 2 种方式运行源文件中的程序。

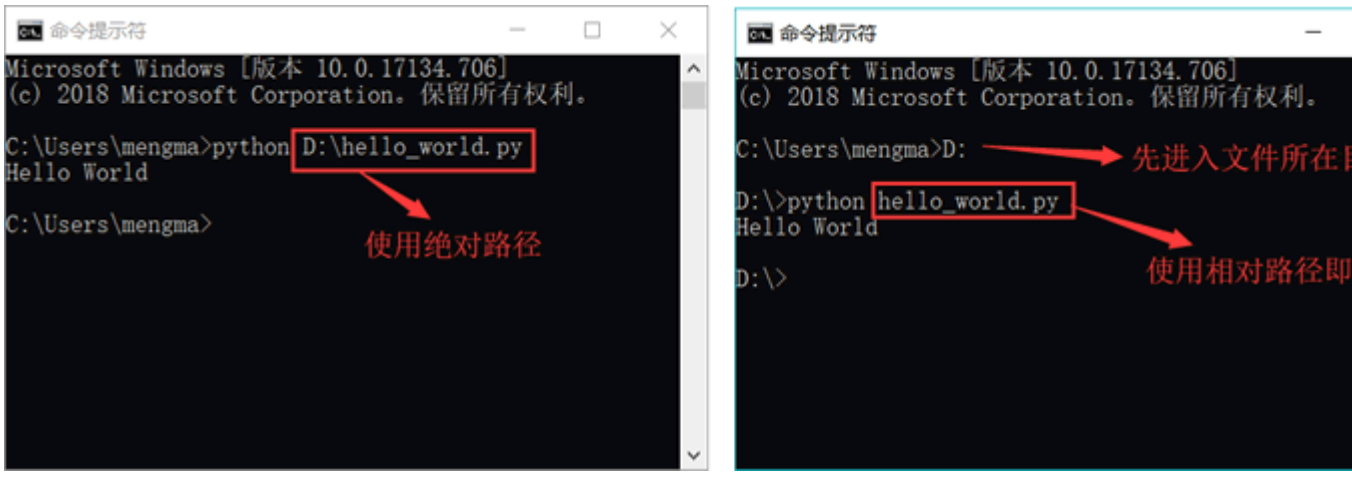


图 4 两种方式运行源文件中的 Python 程序

注意，Windows 系统不区分大小写，在 Windows 平台上输入源程序路径时可以不区分大小写。但是 Mac OS X 或 Linux 系统都区分大小写，因此在这两个平台上输入 Python 源程序路径时一定要注意大小写问题。

Sublime Text 编辑并运行 Python 程序

比较特殊的是，如果在 Mac OS X 上使用 Sublime Text 编辑 Python 程序，可以集成

“python”命令来运行 Python 程序，从而避免每次都启动终端窗口。在 Sublime Text 工具中集成运行 Python 程序的“python”命令，可按如下步骤进行：

1. 在 Sublime Text 的主菜单中选择“Tools -> Build System -> New Build System...”菜单项，Sublime Text 工具将会打开一个新的配置文件，将该配置文件中的内容改为如下形式：

```
{
"cmd":["python3","-u","$file"]
}
```

上面文件指定运行“python3”命令来解释运行当前文件（$file 代表当前文件）。将该配置命名为“Python3.sublime-build”，并保存在默认目录下（当选择保存时 Sublime Text 默认打开的目录）。

Python3.sublime.build 就是 Sublime Text 自定义构建系统的配置文件，该文件默认保存在 /Users/用户名/Library/Application Support/Sublime Text 3/Packages/User 目录下。

2. 重启 Sublime Text 工具，此时即可在 Sublime Text 的“Tools->Build System”菜单中看到新增的“Python 3”菜单项，勾边该菜单项，就是告诉 Sublime Text 将要使用前一步定义的配置文件。

3. 选择 Sublime Text 的“Tools->Build”菜单项（或按“command+B”快捷键），即可运行当前打开的 Python 程序。在 Sublime Text 窗口底部将会显示运行结果，如图 5 所示。

图 5 在 Sublime Text 中集成 Python 工具

上面介绍的方法是在 Sublime Text 中集成“python3”命令，实际上也可按该步骤“集成 python2”命令。

6.Python 交互式解释器

为了让开发者能快速学习、测试 Python 的各种功能，Python 提供的“python”命令不仅能用于运行 Python 程序，也可作为一个交互式解释器——开发者逐行输入 Python 代码，它逐行解释执行。

当输入“python”命令时，可以看到如下输出结果：

```
C:\Users\mengma>python
Python 3.6.2 (v3.6.2:5fd33b5,Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32
Type "help","copyright","credits" or "license" for more information.
>>>
```

交互式解释器提示当前使用的是 Python 3.6.2，后面的“>>>”就是交互式解释器的提示符。

接下来用户可在该提示符后输入如下命令：

```
>>>print ("Hello World")
```

按回车键后，交互式解释器就会解释执行这行代码，生成如下输出结果：

```
Hello World
>>>
```

从上面的输出结果可以看出，交互式解释器执行完 print("Hello World") 之后，再次显示“>>>”，用于提示用户可以再次输入 Python 命令。

如果随便输入一段内容，例如输入 crazyit，将可以看到如下输出结果：

```
>>>crazyit
Traceback (most recent call last ) :
```

```
    File "<stdin>", line 1, in<module>
NameError: name 'crazyit' is not defined
>>>
```

交互式解释器提示 crazyit 没有定义。这表明该交互式解释器完全可作为一个“快速演练场”，既可用于学习各种新语法，也可用于测试各种功能。

再比如输入 5**4（其中**是乘方运算符），可以看到如下输出：

```
>>> 5**4
625
>>>
```

从上面的输出结果可以看出，交互式解释器帮我们计算了 5 的 4 次方的结果。

需要说明的是，如果直接在 Python 程序中写中写 5**4 不会有任何效果，这是因为 5**4 只是表示一次计算，既没有让 Python 输出计算结果，也没有将该结果赋值给任何变量。但交互式解释器总是将所有表达式的值打印出来，这样才能与开发者交互，所以可以看到 5**4 的结果是 625。

实际上，你可以在其中输入任何复杂的算式（甚至包括复数运算，Python 支持复数运算），交互式解释器总可以帮你得到正确的结果。这也是很多非专业程序员喜欢 Python 的一个原因：即使你不是程序员，但只要输入想执行的运算，交互式解释器就能告诉你正确的答案。从这个角度来看，Python 的交互式解释器相当于一个功能无比强大的“计算器”，比 Windows 、Mac OS X 系统自带的计算器的功能强大多了。

交互式解释器的运行效果如图 1 所示：

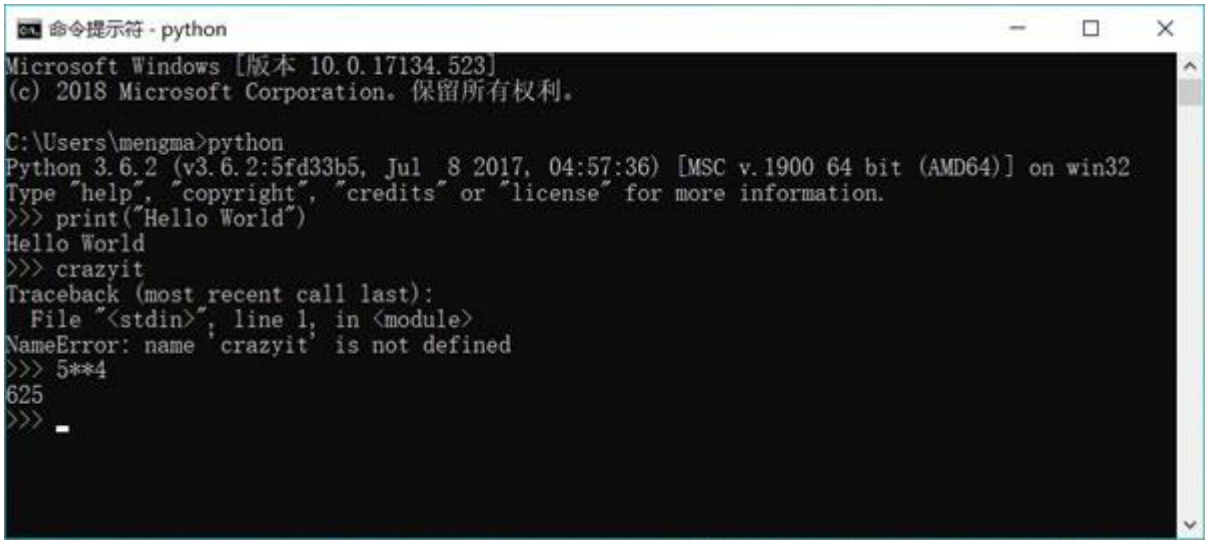


图 1 交互式解释器

7.IDE（集成开发环境）是什么

IDE 是 Intergreated Development Environment 的缩写，中文称为集成开发环境，用来表示辅助程序员开发的应用软件，是它们的一个总称。

通过前面章节的学习我们知道，运行 C 语言（或 Java 语言）程序必须有编译器，而运行 Python 语言程序必须有解释器。在实际开发中，除了运行程序必须的工具外，我们往往还需要很多其他辅助软件，例如语言编辑器、自动建立工具、除错器等等。这些工具通常被打包在一起，统一发布和安装，例如 PythonWin、MacPython、PyCharm 等，它们统称为集成开发环境（IDE）。

因此可以这么说，集成开发环境就是一系列开发工具的组合套装。这就好比台式机，一个台式机的核心部件是主机，有了主机就能独立工作了，但是我们在购买台式机时，往往还要附带上显示器、键盘、鼠标、U 盘、摄像头等外围设备，因为只有主机太不方便了，必须有外设才能玩的爽。

> 需要注意的是，虽然有一些 IDE 支持多种程序语言的开发（如 Eclipse、NetBeans、VS），但通常来说，IDE 主要还是针对某一特定的程序语言而量身打造的（如 VB）。

一般情况下，程序员可选择的 IDE 类别是很多的，比如说，用 Python 语言进行程序开发，既可以选用 Python 自带的 IDLE，也可以选择使用 PyCharm 和 Notepad++ 作为 IDE。并且，为了称呼方便，人们也常常会将集成开发环境称为编译器或编程软件，对此读者没必要较真儿，就把它当做“乡间俗语”吧。

8.Python IDE 有哪些，哪款适合初学者？

百度搜索“Python IDE”会发现支持 Python 编程的 IDE 有很多，那么对于零基础的初学者，应该使用哪款 IDE 呢？

我个人推荐初学者使用 Python 自带的 IDLE。因为 IDLE 的使用方法很简单，非常适合初学者入门。

当然，除了 IDLE，还有很多其他的 IDE 供大家选择，这里列出常用的几个，喜欢探索的读者可自行安装使用。

PyCharm

这是由 JetBrains 公司开发的一款 Python 开发工具，在 Windows、Mac OS 和 Linux 操作系统中都可以使用。

PyCharm 具有语法高亮显示、Project（项目）管理代码跳转、智能提示、自动完成、调试、单元测试以及版本控制等一般开发工具都具有的功能，除此之外，它还支持 Django（Python 的 Web 开发框架）框架下进行 Web 开发。

PyCharm 的主窗口如图 1 所示。

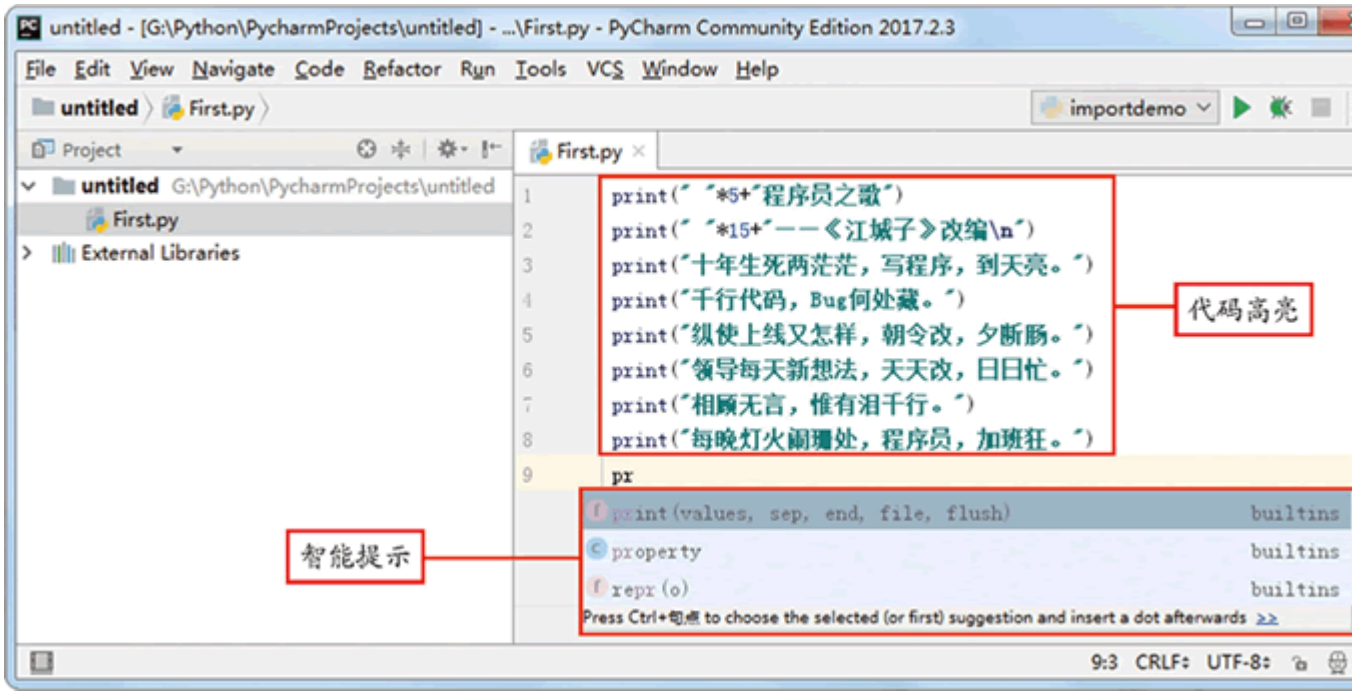


图 1 PyCharm 主窗口示意图

PyCharm 开发工具可通过其官方网站(http://www.jetbrains.com/pycharm/)下载获取。需要注意的是，该网站提供了 2 个版本，一个是社区版（免费并且提供源代码，适合多数读者），另一个是专业版（免费试用）。

Microsoft Visual Studio

Microsoft Visual Studio ,简称 VS ,是 Microsoft(微软)公司开发的一款用于进行 C# 和 ASP.NET 等应用的开发工具，但它也可以作为 Python 的开发工具，只需要在安装时，选择安装 PTVS 插件即可。

PTVS 插件是一个开源插件，它支持编辑、浏览、智能感知、混合 Python/C++ 调试、Django 等，适用于 Windows、Linux 和 Mac OS 客户端的云计算。

当 VS 安装 PTVS 插件之后 ,就可以进行 Python 应用开发了 ,其开发界面如图 2 所示。

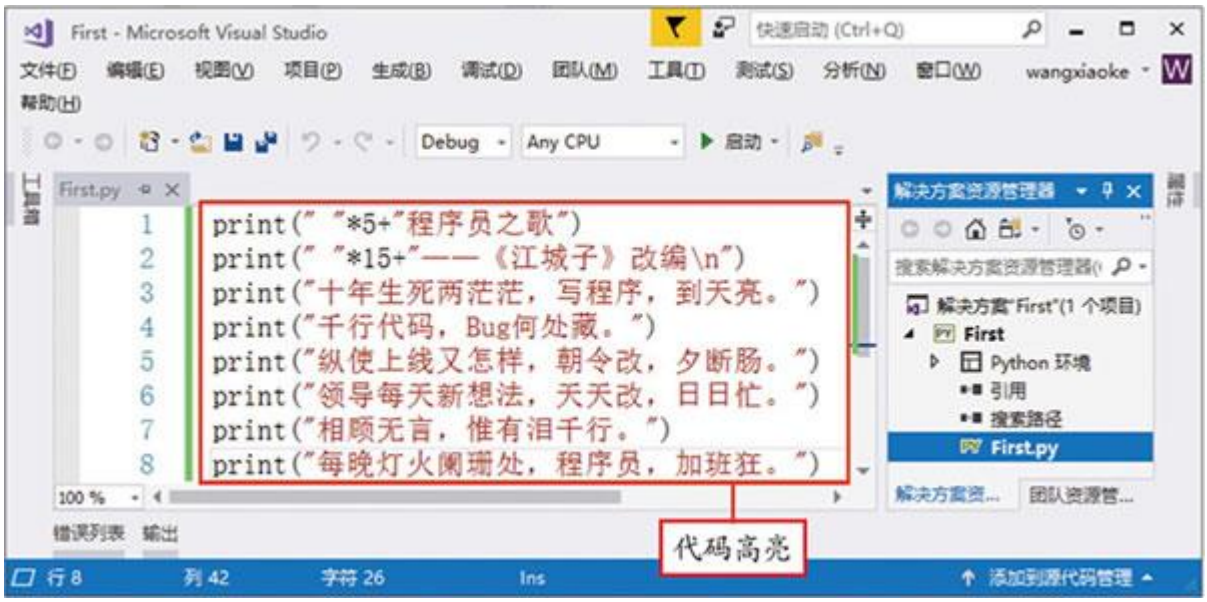


图 2 应用 VS 开发 Python 项目

Eclipse+PyDev

Eclipse 是一个开源的、基于 Java 的可扩展开发平台，最初主要用于 Java 语言的开发。该平台可通过安装不同的插件，进行不同语言的开发。

PyDev 是一款功能强大的 Eclipse 插件，它可以提供语法高亮、语法分析、语法错误提示，以及大纲视图显示导入的类、库和函数、源代码内部的超链接、运行和调试等功能。

当 Eclipse 在安装 PyDev 插件后，就可以进行 Python 应用开发。其开发界面如图 3 所示。

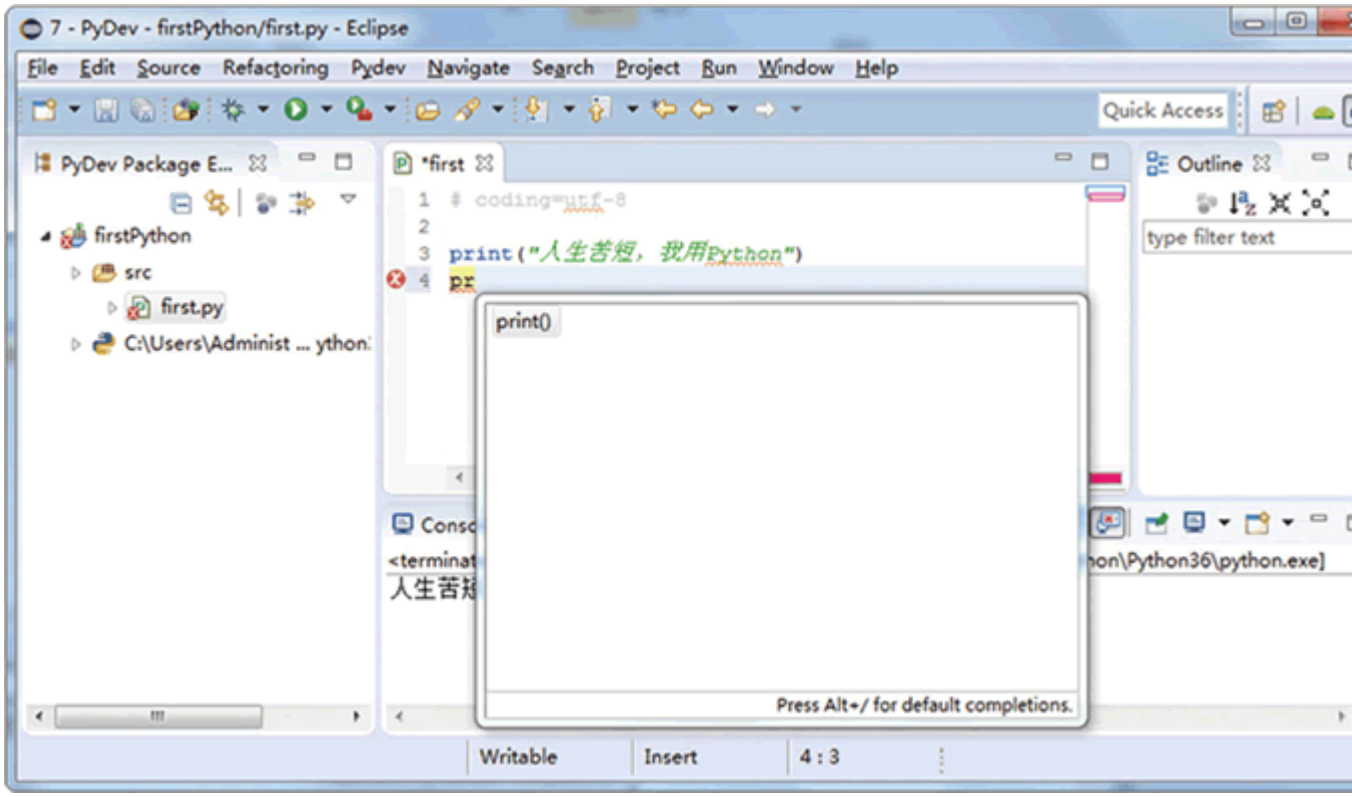


图 3 Eclipse+PyDev 开发界面

除此之外，还有 PythonWin（只针对 Win32 平台）、MacPython IDE（PythonWin 对应的 Mac 版本）、Emacs 和 Vim（功能强大的文本编辑器，可以用来编写 Python 程序）等，都可以作为执行 Python 程序的 IDE。

> 需要注意的是，虽然适合作为 Python 开发工具的 IDE 有很多，选择一个适合自己的即可。

9.Python IDLE 使用方法（包含常用快捷键）

在安装 Python 后，会自动安装一个 IDLE，它是一个 Python Shell (可以在打开的 IDLE 窗口的标题栏上看到)，程序开发人员可以利用 Python Shell 与 Python 交互。

本节将以 Windows7 系统中的 IDLE 为例，详细介绍如何使用 IDLE 开发 Python 程序。

单击系统的开始菜单，然后依次选择“所有程序 -> Python 3.6 -> IDLE (Python 3.6 64-bit)”菜单项，即可打开 IDLE 窗口，如图 1 所示。

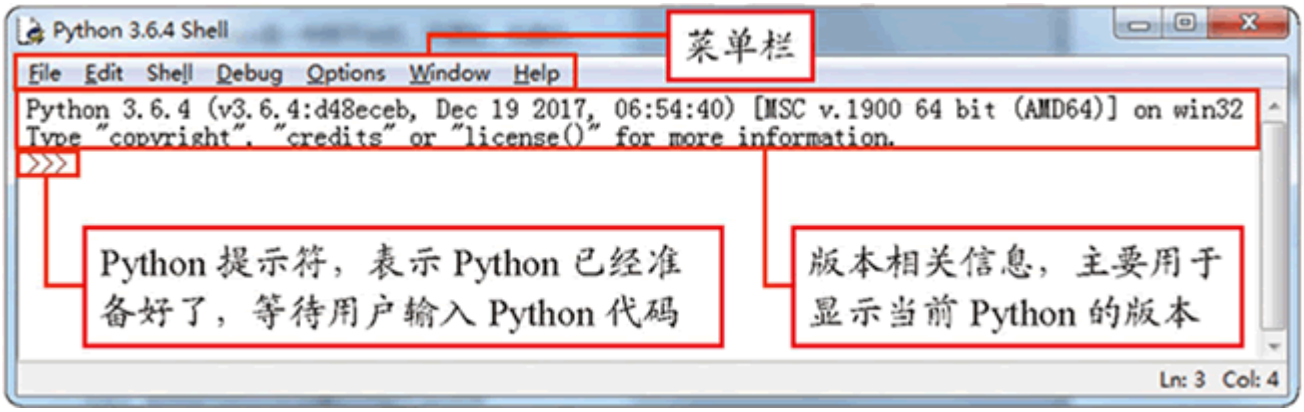


图 1 IDLE 主窗口

前面我们已经应用 IDLE 输出了简单的语句，但在实际开发中，通常不能只包含一行代码，当需要编写多行代码时，可以单独创建一个文件保存这些代码，在全部编写完成后一起执行。具体方法如下：

1. 在 IDLE 主窗口的菜单栏上，选择“File -> New File”菜单项，将打开一个新窗口，在该窗口中，可以直接编写 Python 代码。

在输入一行代码后再按下 <Enter> 键，将自动换到下一行，等待继续输入，如图 2 所示。

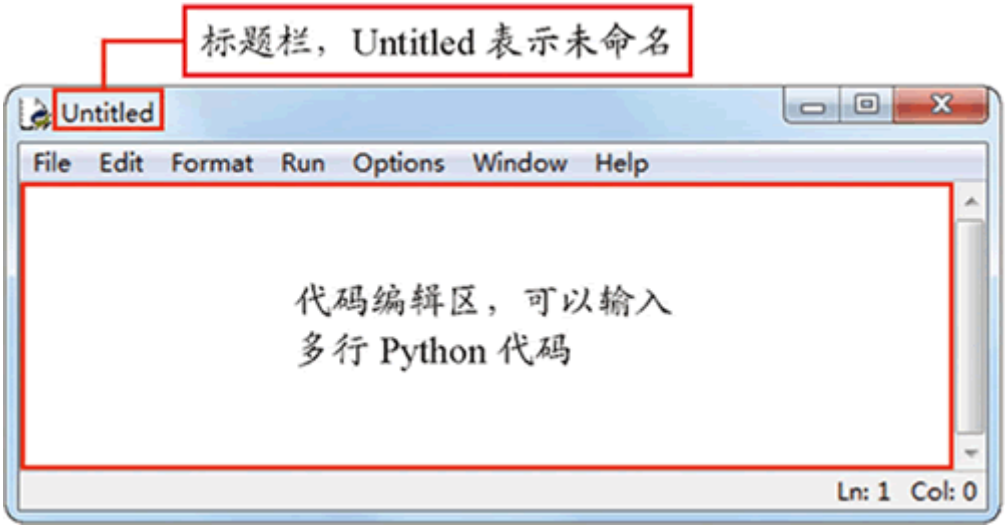


图 2 新创建的 Python 文件窗口

2. 在代码编辑区中，编写多行代码。例如，输出由宋词《江城子》改编而成的《程序员之歌》，代码如下：

```
print(" "*5+"程序员之歌")
print(" "*15+"——《江城子》改编\n")
print("十年生死两茫茫，写程序，到天亮，")
print("千行代码，Bug 何处藏。")
print("纵使上线又怎样，朝令改，夕断肠。")
print("领导每天新想法，天天改，日日忙。")
print("相顾无言，惟有泪千行")
print("每晚灯火阑珊处，程序员，加班狂。")
```

编写代码后的 Python 文件窗口如图 3 所示。

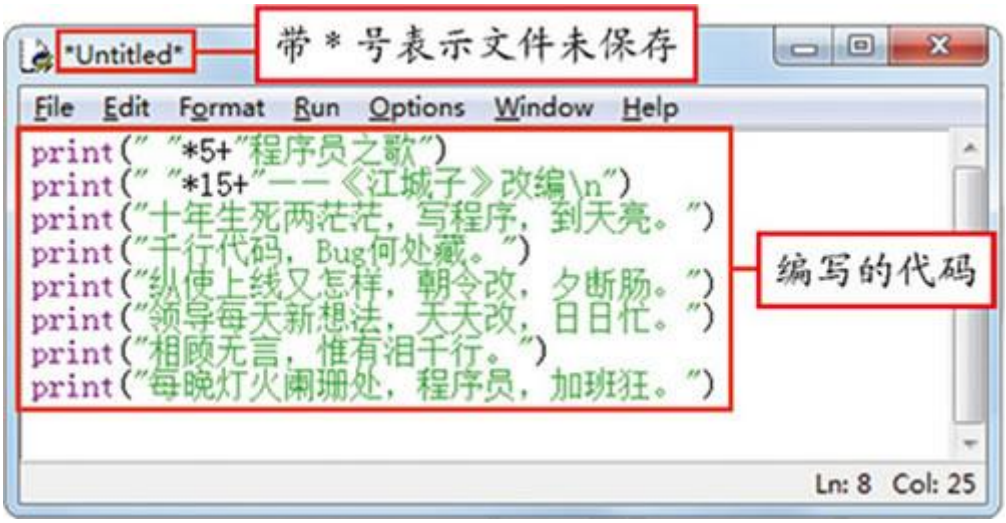


图 3 编写代码后的 Python 文件窗口

3. 按下快捷键 <Ctrl+S> 保存文件，这里将文件名称设置为 demo.py。其中，.py 是 Python 文件的扩展名。在菜单栏中选择“Run -> Run Module”菜单项（也可以直接按下快捷键 <F5>），运行程序，如图 4 所示。

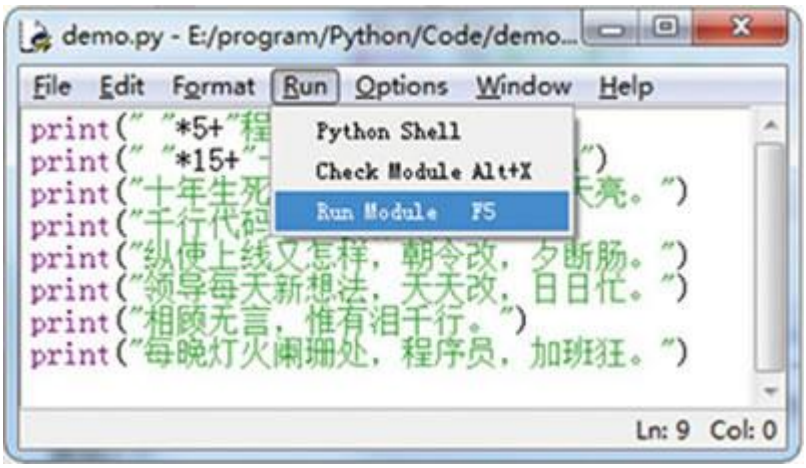


图 4 运行程序

4. 运行程序后，将打开 Python Shell 窗口显示运行结果，如图 5 所示。

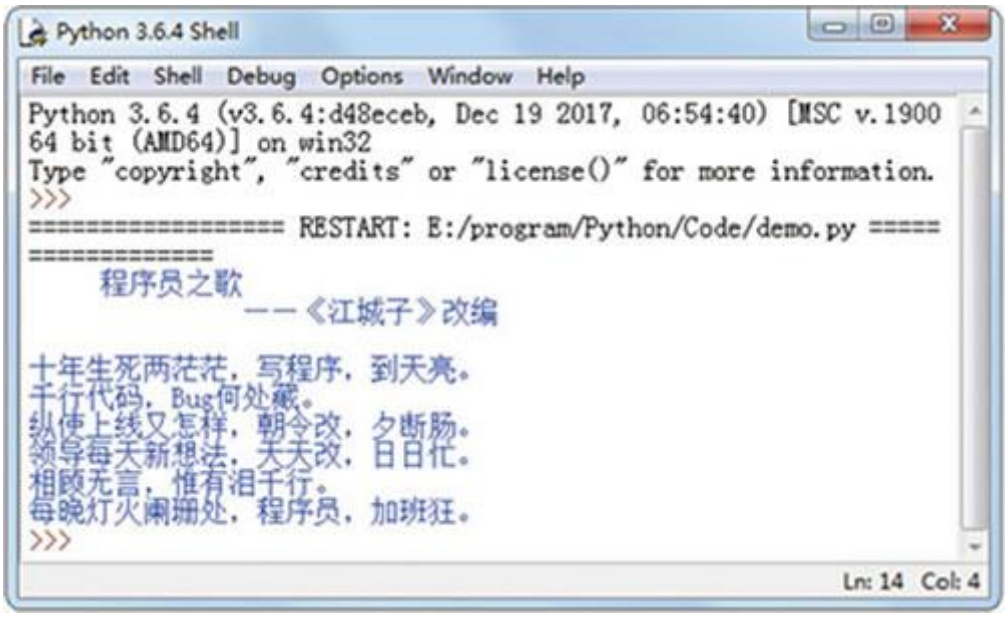


图 5 运行结果

Python IDLE 常用快捷键

在程序开发过程中，合理使用快捷键不但可以减少代码的错误率，而且可以提高开发效率。在 IDLE 中，可通过选择“Options -> Configure IDLE”菜单项，在打开的“Settings”对话框的‘Keys”选项卡中查看，但是该界面是英文的，不便于查看。为方便读者学习，表 6 列出了 IDLE 中一些常用的快捷键。

表 6 IDLE 提供的常用快捷键

快提键	说 明	适用范围
F1	打开 Python 帮助文档	Python 文件窗口和 Shell 均可用

Alt+P	浏览历史命令（上一条）	仅 Python Shell 窗口可用
Alt+N	浏览历史命令（下一条）	仅 Python Shell 窗口可用
Alt+/	自动补全前面曾经出现过的单词，如果之前有多个单词具有相同前缀，可以连续按下该快捷键，在多个单词中间循环选择	Python 文件窗口和 Shell 窗口均可用
Alt+3	注释代码块	仅 Python 文件窗口可用
Alt+4	取消代码块注释	仅 Python 文件窗口可用
Alt+g	转到某一行	仅 Python 文件窗口可用
Ctrl+Z	撤销一步操作	Python 文件窗口和 Shell 窗口均可用
Ctrl+Shift+Z	恢复上一次的撤销操作	Python 文件窗口和 Shell 窗口均可用
Ctrl+S	保存文件	Python 文件窗口和 Shell 窗口均可用

Ctrl+]	缩进代码块	仅 Python 文件窗口可用
Ctrl+[	取消代码块缩进	仅 Python 文件窗口可用
Ctrl+F6	重新启动 Python Shell	仅 Python Shell 窗口可用

由于 IDLE 简单、方便，很适合联系，因此本教程如果没有特殊说明，均使用 IDLE 作为开发工具。

10.Python 注释

学会向程序中添加必要的注释，也是很重要的。注释不仅可以用来解释程序某些部分的作用和功能（用自然语言描述代码的功能），在必要时，还可以将代码临时移除，是调试程序的好帮手。

当然，添加注释的最大作用还是提高程序的可读性！很多时候，笔者宁愿自己写一个应用，也不愿意去改进别人的代码，没有合理的注释是一个重要原因。虽然良好的代码可自成文挡，但我们永远也不清楚今后读这段代码的人是谁，他是否和你有相同的思路。或者一段时间以后，你自己也不清楚当时写这段代码的目的了。

> 总的来说，一旦程序中注释掉某部分内容，则该内容将会被 Python 解释器忽略，换句话说，此部分内容将不会被执行。

通常而言，合理的代码注释应该占源代码的 1/3 左右。Python 语言允许在任何地方插入空字符或注释，但不能插入到标识符和字符串中间。

在 Python 中，通常包括 3 种类型的注释，分别是单行注释、多行注释和中文编码声明注释。

Python 单行注释

Python 中使用井号（‘#’）作为单行注释的符号，语法格式为：

```
# 注释内容
```

也就是说，从符号‘#’处开始，直到换行处结束，此部分内容都作为注释的内容，当程序执行时，这部分内容会被忽略。

单行注释放置的位置，既可以是要注释代码的前一行，例如：

```
1.          #这是一行简单的注释
2.          print ("Hello World!")
```

也可以是注释代码的右侧，例如：

```
1.          print ("Hello World!") #这是一行简单的注释
```

基于注释在程序中所起到的作用，以上两种方式虽然放置位置不同，但程序的执行结果是相

同的，都是输出“Hello World!”。

Python 多行注释

多行注释指的是可以一次性注释程序中多行的内容（包含一行）。多行注释的语法有 2 种，分别是：

```
'''
使用 3 个单引号分别作为注释的开头和结尾
可以一次性注释多行内容
这里面的内容全部是注释内容
'''
```

或者

```
"""
使用 3 个双引号分别作为注释的开头和结尾
可以一次性注释多行内容
这里面的内容全部是注释内容
"""
```

多行注释通常用来为 Python 文件、模块、类或者函数等添加版权、功能描述等信息。需要注意的是，在 Python 中，如果多行注释标记（三单引号或三双引号）作为语句的一部分出现，就不能再将它们视为多行注释的标记，而应看做是字符串的标志（同双引号的作用相同），例如：

```
print('''Hello,World!''')
```

该语句的执行结果为：`Hello，World!`。由此可见，Python 解释器没有将这里的三个单引号看做是注释，而是将它们看作是字符串的标志。

Python 中文编码声明注释

Python 还提供了一种特殊的中文编码声明注释，其主要用来解决 Python 2.x 中不支持直接写中文的问题。虽然此问题在 Python 3.x 中已经不存在啦，但为了规范编码，增强代码的可执行性，方便其他程序员及时了解程序所用的编码，建议初学者在程序开头处加上中文编码声明注释。

中文编码声明注释的语法有如下 2 种：

```
# -*- coding:编码 -*-
或者
# coding=编码
```

语法中的编码，指的是编写程序所用的字符编码类型，比如 UTF-8、GBK 编码等。另外，在第一种语法中，‘-*-’并没有实际意义，只是为了美观才加上去了，因此，第一种语法格式中可以直接将前后的 ‘-*-’ 去掉。

举个例子，如果指定编码为 UTF-8，则可以使用下面的中文编码声明注释：

```
# coding:utf-8
或者
# coding=utf-8
```

注意，给代码添加注释是是注释的基本用法，它还有一个更实用的功能，即用来调试程序。举个例子，如果你觉得某段代码可能有问题，可以先把这段代码注释起来，让 Python 解

释器忽略这段代码，然后运行。如果程序可以正常执行，则可以说明错误就是由这段代码引起的；反之，如果依然出现相同的错误，则可以说明错误不是由这段代码引起的。

由此可见，在调试程序时，学会注释代码，可以缩小错误所在的范围，提高调试程序的效率。

11.Python 缩进规则（包含快捷键）

和其它程序设计语言（如 Java、C 语言）采用大括号“{}”分隔代码块不同，Python 采用代码缩进和冒号（ ： ）来区分代码块之间的层次。

在 Python 中，对于类定义、函数定义、流程控制语句、异常处理语句等，行尾的冒号和下一行的缩进，表示下一个代码块的开始，而缩进的结束则表示此代码块的结束。

注意，Python 中实现对代码的缩进，可以使用空格或者 Tab 键实现。但无论是手动敲空格，还是使用 Tab 键，通常情况下都是采用 4 个空格长度作为一个缩进量（默认情况下，一个 Tab 键就表示 4 个空格）。

例如，下面这段 Python 代码中（涉及到了目前尚未学到的知识，初学者无需理解代码含义，只需体会代码块的缩进规则即可）：

```
1.          height=float(input("输入身高："))  #输入身高
2.          weight=fioat(input("输入体重："))  #输入体重
3.          bmi=weight/(height*height)        #计算 BMI 指数
4.
5.          #判断身材是否合理
6.          if bmi<18.5:
```

```
7.              #下面 2 行同属于 if 分支语句中包含的代码，因此属于同一作用域
8.              print("BMI 指数为: "+str(bmi)) #输出 BMI 指数
9.              print("体重过轻")
10.         if bmi>=18.5 and bmi<24.9:
11.             print("BMI 指数为: "+str(bmi)) #输出 BMI 指数
12.             print("正常范围，注意保持")
13.         if bmi>=24.9 and bmi<29.9:
14.             print("BMI 指数为: "+str(bmi)) #输出 BMI 指数
15.             print("体重过重")
16.         if bmi>=29.9:
17.             print(BMI 指数为: "+str(bmi)) #输出 BMI 指数
18.             print("肥胖")
```

Python 对代码的缩进要求非常严格，同一个级别代码块的缩进量必须一样，否则解释器会报 SyntaxError 异常错误。例如，对上面代码做错误改动，将位于同一作用域中的 2 行代码，它们的缩进量分别设置为 4 个空格和 3 个空格，如下所示：

```
1.          if bmi<18.5:
2.              print("BMI 指数为: "+str(bmi)) #输出 BMI 指数
3.             print("体重过轻")
```

可以看到，第二行代码和第三航代码本来属于同一作用域，但我们手动修改了各自的缩进量，这会导致 SyntaxError 异常错误，如图 1 所示。

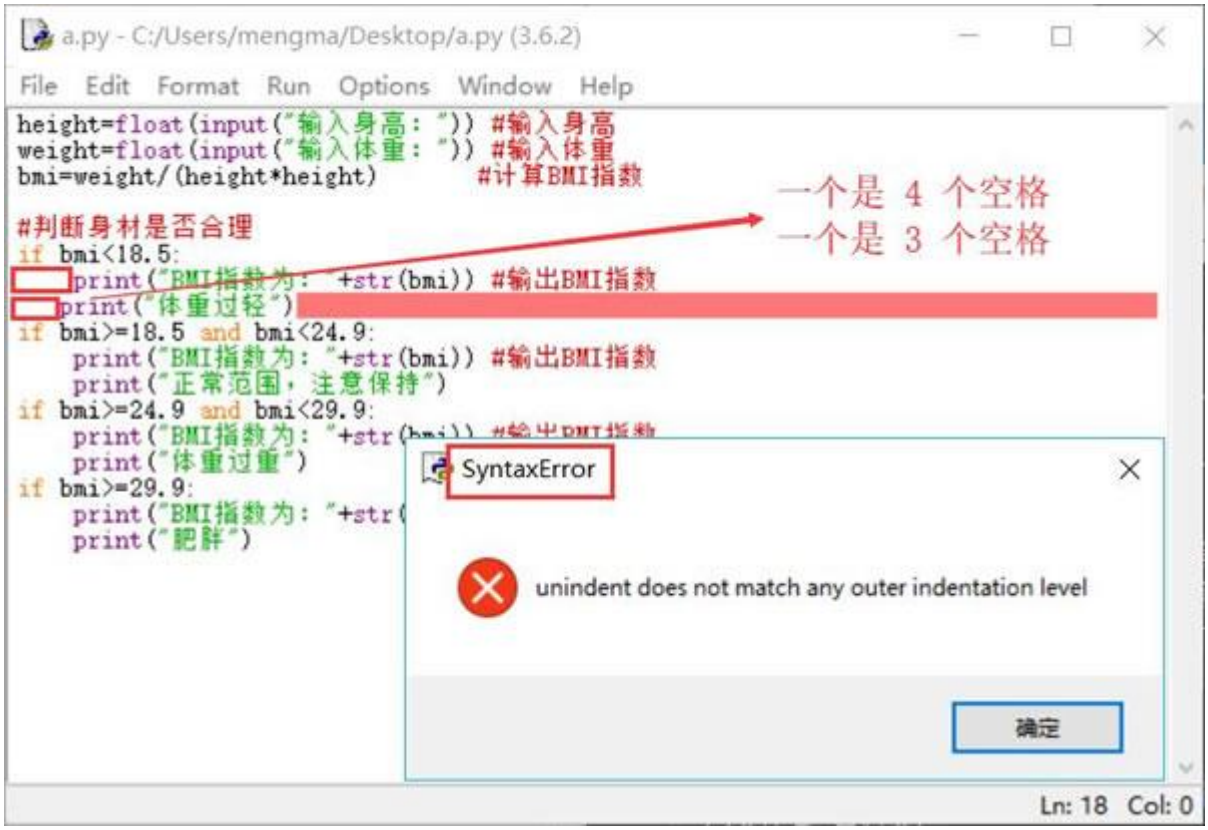
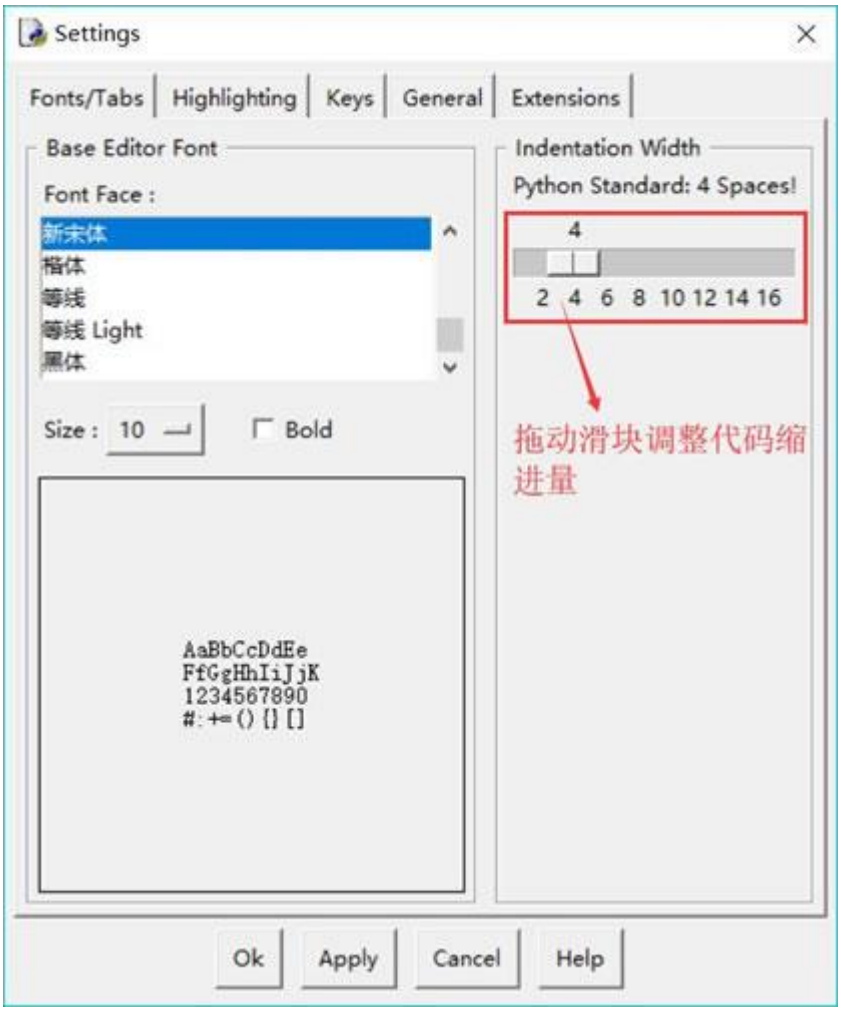


图 1 缩进不符合规范导致异常

对于 Python 缩进规则，初学者可以这样理解，Python 要求属于同一作用域中的各行代码，它们的缩进量必须一致，但具体缩进量为多少，并不做硬性规定。

IDLE 开发环境对缩进量的设置

在 IDLE 开发环境中，默认是以 4 个空格作为代码的基本缩进单位。不过，这个值是可以手动改变的，在菜单栏中选择 Options -> Configure，会弹出如下对话框：

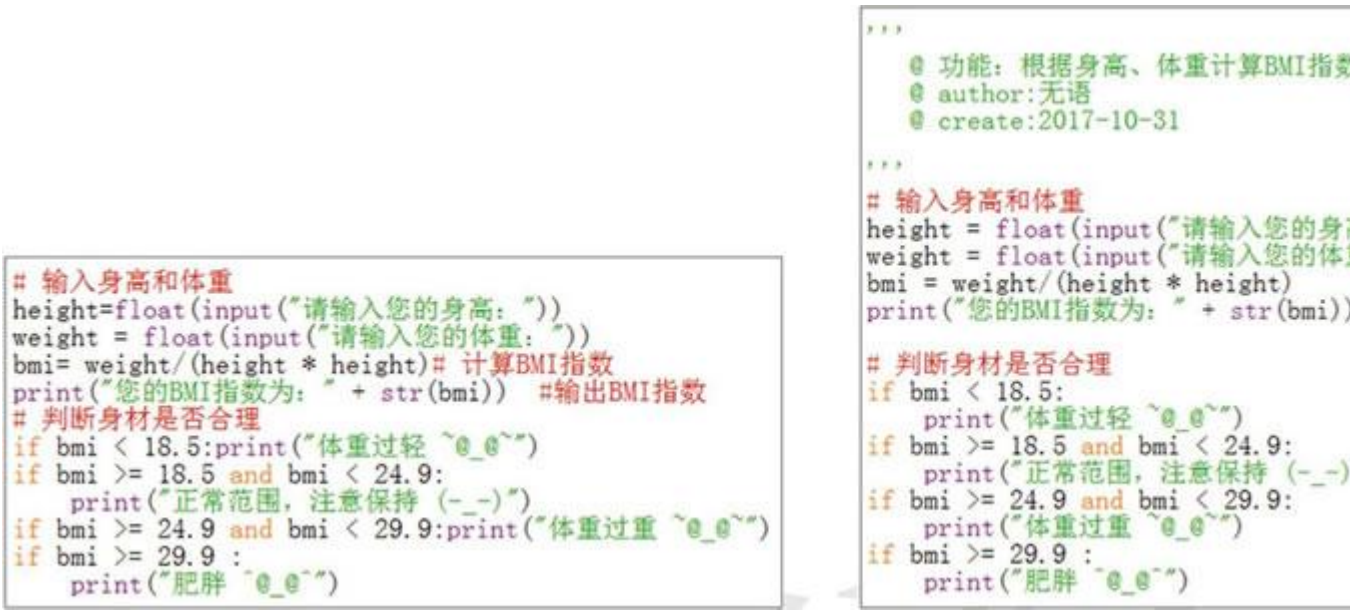


如图所示，通过拖动滑块，即可改变默认的代码缩进量，例如拖动至 2，则当你使用 Tab 键设置代码缩进量时，会发现按一次 Tab 键，代码缩进 2 个空格的长度。

不仅如此，在使用 IDLE 开发环境编写 Python 代码时，如果想通过设置多行代码的缩进量，可以使用 Ctrl+] 和 Ctrl+[快捷键，此快捷键可以使所选中代码快速缩进（或反缩进）。

12.Python 编码规范（PEP 8）

在讲解具体的 Python 编码规范之前，先来看看图 1 中的代码：



```
# 输入身高和体重
height=float(input("请输入您的身高："))
weight = float(input("请输入您的体重："))
bmi= weight/(height * height)# 计算BMI指数
print("您的BMI指数为：" + str(bmi))  #输出BMI指数
# 判断身材是否合理
if bmi < 18.5:print("体重过轻 ~@_@~")
if bmi >= 18.5 and bmi < 24.9:
    print("正常范围，注意保持 (-_-)")
if bmi >= 24.9 and bmi < 29.9:print("体重过重 ~@_@~")
if bmi >= 29.9 :
    print("肥胖 ~@_@~")
```

```
'''
    @ 功能：根据身高、体重计算BMI指
    @ author:无语
    @ create:2017-10-31
'''
# 输入身高和体重
height = float(input("请输入您的身
weight = float(input("请输入您的体
bmi = weight/(height * height)
print("您的BMI指数为：" + str(bmi))
# 判断身材是否合理
if bmi < 18.5:
    print("体重过轻 ~@_@~")
if bmi >= 18.5 and bmi < 24.9:
    print("正常范围，注意保持 (-_-)
if bmi >= 24.9 and bmi < 29.9:
    print("体重过重 ~@_@~")
if bmi >= 29.9 :
    print("肥胖 ~@_@~")
```

图 1 两段功能相同的 Python 代码

对比图 1 中的两段代码你会发现，它们所包含的代码时完全相同的，但很明显，右侧的代码编写格式看上去比左侧的代码段更加规整，阅读起来也会比较轻松、畅快，因为它遵循了最基本的 Python 代码编写规范。

Python 采用 PEP 8 作为编码规范，其中 PEP 是 Python Enhancement Proposal（Python 增强建议书）的缩写，8 代表的是 Python 代码的样式指南。下面仅给大家列出 PEP 8 中初学者应严格遵守的一些编码规则：

1. 每个 import 语句只导入一个模块，尽量避免一次导入多个模块，例如：

```
#推荐
import os
import sys
```

```
#不推荐
import os,sys
```

关于 import 的含义和用法会在后续介绍，这里不必深究。

2. 不要在行尾添加分号，也不要用分号将两条命令放在同一行，例如：

```
#不推荐
height=float(input("输入身高：")) ; weight=fioat(input("输入体重：")) ;
```

建议每行不超过 80 个字符，如果超过，建议使用小括号将多行内容隐式的连接起来，而不推荐使用反斜杠 \ 进行连接。例如，如果一个字符串文本无法实现一行完全显示，则可以使用小括号将其分开显示，代码如下：

```
#推荐
s=("C 语言中文网是中国领先的 C 语言程序设计专业网站，"
"提供 C 语言入门经典教程、C 语言编译器、C 语言函数手册等。")
#不推荐
s="C 语言中文网是中国领先的 C 语言程序设计专业网站，\
提供 C 语言入门经典教程、C 语言编译器、C 语言函数手册等。"
```

注意，此编程规范适用于绝对大多数情况，但以下 2 种情况除外：

- 导入模块的语句过长。
- 注释里的 URL。

3. 使用必要的空行可以增加代码的可读性，通常在顶级定义（如函数或类的定义）之间空两行，而方法定义之间空一行，另外在用于分隔某些功能的位置也可以空一行。比如说，在图 1 右侧这段代码中，if 判断语句同之前的代码多实现的功能不同，因此这里可以使用空行进行分隔。

4. 通常情况下，在运算符两侧、函数参数之间以及逗号两侧，都建议使用空格进行分隔。

以上就是初学者应该遵循的部分 Python 编码规范，如果想了解更多 PEP 8 的详细信息，可访问 PEP 8 官方介绍。

13.Python 标识符命名规范

简单地理解，标识符就是一个名字，就好像我们每个人都有属于自己的名字，它的主要作用就是作为变量、函数、类、模块以及其他对象的名称。

Python 中标识符的命名不是随意的，而是要遵守一定的命令规则，比如说：

1. 标识符是由字符（A~Z 和 a~z）、下划线和数字组成，但第一个字符不能是数字。
2. 标识符不能和 Python 中的保留字相同。有关保留字，后续章节会详细介绍。
3. Python 中的标识符中，不能包含空格、@、% 以及 $ 等特殊字符。

例如，下面所列举的标识符是合法的：

```
UserID

name

mode12

user_age
```

以下命名的标识符不合法：

```
4word      #不能以数字开头
try             #try 是保留字，不能作为标识符
$money #不能包含特殊字符
```

4. 在 Python 中，标识符中的字母是严格区分大小写的，也就是说，两个同样的单词，如果大小格式不一样，多代表的意义也是完全不同的。比如说，下面这 3 个变量之间，就是完全独立、毫无关系的，它们彼此之间是相互独立的个体。

```
number = 0
Number = 0
NUMBER = 0
```

5. Python 语言中，以下划线开头的标识符有特殊含义，例如:

- 以单下划线开头的标识符（如 _width），表示不能直接访问的类属性，其无法通过 from...import* 的方式导入；
- 以双下划线开头的标识符（如__add）表示类的私有成员；
- 以双下划线作为开头和结尾的标识符（如 __init__），是专用标识符。

因此，除非特定场景需要，应避免使用以下划线开头的标识符。

另外需要注意的是，Python 允许使用汉字作为标识符，例如:

```
C 语言中文网 = "http://c.biancheng.net"
```

但我们应尽量避免使用汉字作为标识符，这会避免遇到很多奇葩的错误。

标识符的命名，除了要遵守以上这几条规则外，不同场景中的标识符，其名称也有一定的规范可循，例如：

- 当标识符用作模块名时，应尽量短小，并且全部使用小写字母，可以使用下划线分割多个字母，例如 game_mian、game_register 等。
- 当标识符用作包的名称时，应尽量短小，也全部使用小写字母，不推荐使用下划线，例如 com.mr、com.mr.book 等。
- 当标识符用作类名时，应采用单词首字母大写的形式。例如，定义一个图书类，可以命名为 Book。
- 模块内部的类名，可以采用 "下划线+首字母大写" 的形式，如 _Book;
- 函数名、类中的属性名和方法名，应全部使用小写字母，多个单词之间可以用下划线分割；
- 常量命名应全部使用大写字母，单词之间可以用下划线分割；

有读者可能会问，如果不遵守这些规范，会怎么样呢？答案是程序照样可以运行，但遵循以上规范的好处是，可以更加直观地了解代码所代表的含义，以 Book 类为例，我们可以很容易就猜到此类与书有关，虽然将类名改为 a（或其它）不会影响程序运行，但通常不这么做。

14.一文彻底搞懂编码规范的重要性

很多去 Google 参观的人，在用完洗手间后都有这样的疑惑，马桶前面的门上怎么会贴着 Python 编码规范？要知道，Google 对编码规范的要求极其严格，这也能从侧面说明编码规范的重要性。

对于编码规范的认知，很多初学者还仅停留在初级阶段，即只知道编码规范有用，比如命名时使用驼峰式的格式（如 TheFirstDemo），而至于为什么要求这样严格，就不是很清楚了。

本节，将给读者扫除以下 2 个盲区：

1. Python 编码规范到底有多么重要，它对于业务开发来说，究竟有哪些帮助？
2. 有哪些流程和工具，可以强制你遵循规定好的编码规范呢？

注意，在讲解过程，会引用以下 2 个编码规范来举例，分别是：

- 《8 号 Python 增强规范》，通常称之为 PEP8；
- 《Google Python 风格规范》 简称为 Google Style，这是源自 Google 内部公开发布的社区版本，其目的是为了让 Google 旗下所有 Python 开源项目的编程风格统一。

以上这 2 个编码规范，Google Style 比 PEP8 更为严格，因为 PEP8 的主要面向群体是个人和小团队开发者，而 Google Style 则能够胜任大团队甚至是企业。

Python 编码规范到底有多么重要

Python 编码规范重要性的原因用一句话来概括就是：统一的编码规范可以提高开发效率。

而影响开发效率的有 3 类对象，分别是阅读者、编程者和机器，它们的优先级是阅读者>>编程者>>机器（>>表示远远大于）。

阅读者>>编程者

写过代码的人应该深有体会，在实际工作中真正用来码代码的时间，远比阅读或者调试的时间要少。事实也是如此，有研究表明，软件工程中 80% 的时间都在阅读代码。

因此，如果想提高开发效率，首先要优化的不是码代码的速度，而是阅读代码的体验。

其实，很多编码规范本身就是为优化读者体验而存在的，拿命名原则来说，PEP8 第 38 条规定命名不能是无意义的单字母，有意义的名称可以很大程序提高阅读者的体验。

编程者>>机器

说完了阅读者的体验，再来聊聊编程者的体验。笔者常常见到的一个错误倾向就是过度简化自己的代码，这样做会大大降低代码的可阅读性，并且一旦出现 BUG，也不容易检查出来。

例如，阅读如下这行程序：

```
result = [(x, y) for x in range(10) for y in range(5) if x * y > 10]
```

上面这行代码还可以改写成如下这种形式：

```
result = []
for x in range(10):
  for y in range(5):
    if x * y > 10:
      result.append((x, y))
```

以上代码，涉及到了列表和判断循环结构的相关知识，由于还未学到，初学者不需要理解。

对比这 2 种写法，显然后者调理更清楚，更容易理解，编写起来也更轻松。

机器体验也很重要

每个人都希望自己编写的代码能正确、高效地在电脑上执行，但是一些危险的编程风格，不仅会影响程序的正确性，也容易成为代码效率的瓶颈。

例如，PEP8 和 Google Style 都特别强调了，何时使用 is， 何时使用 ==，何时使用隐式布尔转换。不仅如此，Google Style 2.8 还对遍历方式的选择作出了明确限制。

在编程过程中，只要严格遵守编码规范，编写出的代码通常都很健壮，可移植性也很高。

编码规范的自动化工具

既然编码规范的终极目标是提高开发效率。所以，如果每次写代码都需要在代码规范上额外花很多时间，就达不到我们的初衷了。

首先，你需要根据自己的具体工作环境，选择或者制定适合自己公司或团队的编码规范。市面上可以参考的规范，也就是在文章开头提到的 PEP8 和 Google Style。

要知道，没有放之四海而皆准的规范，我们必须要因地制宜。例如在 Google 中，因为历史原因 C++ 不使用异常，引入异常对整个代码库带来的风险已经远大于它的益处，所以在它的 C++ 代码规范中，禁止使用异常。

一旦确定了整个团队所遵从的编码规范，就一定要强制执行，有什么好的办法呢？靠强制代码评审和强制静态或者动态 linter。具体流程是：

1. 在代码评审工具里，添加必须的编码规范环节；
2. 把团队确定的代码规范写进 Pylint 里，能够在每份代码提交前自动检查，不通过的代码无法提交。

整合之后，你的团队工作流程就会变成图 1 所示的这样。

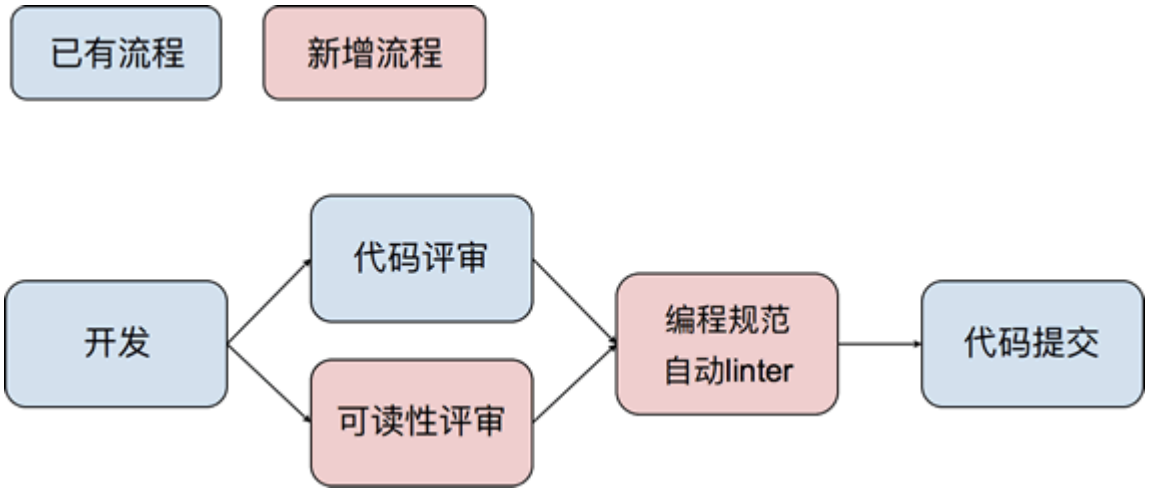


图 1 自动检查编码规范的工作流程

学到这里，相信你对代码风格的重要性有了全新的认识。

15.Python 关键字（保留字）

保留字是 Python 语言中一些已经被赋予特定意义的单词，这就要求开发者在开发程序时，不能用这些保留字作为标识符给变量、函数、类、模板以及其他对象命名。

Python 包含的保留字可以执行如下命令进行查看：

```
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif',
'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal',
'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

所有的保留字，如下表所示：

表 1 Python 保留字一览表

and	as	assert	break	class	continue
def	del	elif	else	except	finally
for	from	False	global	if	import

in	is	lambda	nonlocal	not	None
or	pass	raise	return	try	True
while	with	yield			

需要注意的是，由于 Python 是严格区分大小写的，保留字也不例外。所以，我们可以说 if 是保留字，但 IF 就不是保留字。

在实际开发中，如果使用 Python 中的保留字作为标识符，则解释器会提示“invalid syntax” 的错误信息，如图 2 所示。

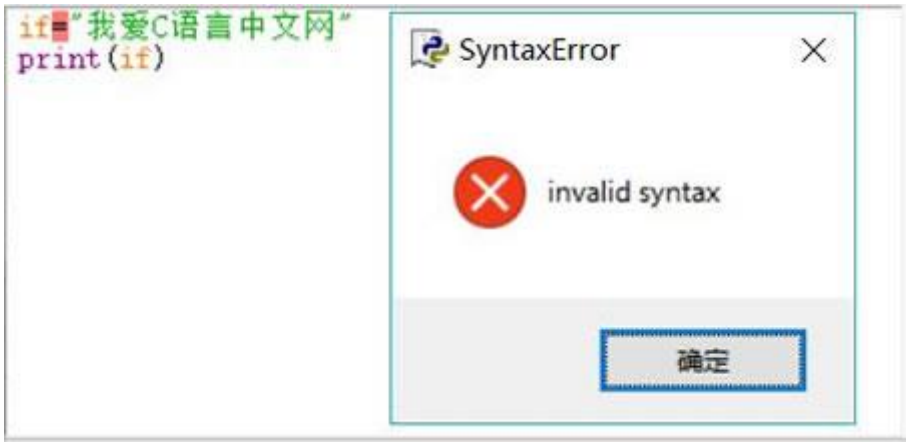


图 2 保留字作标识符报错信息示意图

16.Python 内置函数

为了提高程序员的开发效率，Python 提供了很多可以直接拿来用的函数（初学者可以先理解为方法），每个函数都可以帮助程序员实现某些具体的功能。

举个例子，在 Python 2.x 中 print 只是一个关键字，但在 Python 3.x 中，print 是用于打印输出的函数，而且还是一个内置函数，通过此函数，我们可以直接将数据做输出操作，比如说：

```
>>>print("C 语言中文网")
C 语言中文网
```

除了 print() 函数之外，Python 还提供了很多内置函数，表 1 中罗列出了 Python 3.x 环境中的所有内置函数。

表 1 Python 3.x 内置函数

内置函数				
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slicea()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()

bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

表 1 中各个内置函数的具体功能和用法，可通过访问 https://docs.python.org/zh-cn/3/library/functions.html 进行查看。

需要注意的是，开发者不建议使用以上内置函数的名字作为标识符使用（作为某个变量、函数、类、模板或其他对象的名称），虽然这样做 Python 解释器不会报错，但这会导致同名的内置函数被覆盖，从而无法使用。例如：

```
>>> print="C 语言中文网"   #Python 解释器不会报错
>>> print(print)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print(print)
TypeError: 'str' object is not callable
```

第 3 章 变量类型和运算符

所有编程语言的第一个功能肯定是定义变量，变量是编程的起始点，程序用到的各种数据都是存储在变量内的。

Python 是一门弱类型语言，弱类型包含两方面的含义：
所有的变量无须声明即可使用，或者说对从未用过的变量赋值就是声明了应变量；
变量的数据类型可以随时改变，同一个变量可以一会儿是数值型，一会儿是字符串型。

本章作为学习 Python 编程的基础，将会向读者详细介绍使用变量的方法。此外，本章会重点介绍 Python 的整型、浮点型、字符串等基础类型，并通过示例介绍各种数据类型的功能、用法，以及相关注意事项。

本章另一个重点内容是运算符。Python 提供了一系列功能丰富的运算符，包括所有的算术运算符，以及功能丰富的位运算符、比较运算符、逻辑运算符，这些运算符是 Python 编程的基础。将运算符和操作数连接在一起就形成了表达式。

1.Python 变量及其使用

无论使用什么语言编程，其最终目的都是对数据进行处理。程序在编程过程中，为了处理数据更加方便，通常会将其存储在变量中。

形象地看，变量就像一个个小容器，用于“盛装”程序中的数据。除了变量，还有常量，它也能用来“盛装”数据，它们的区别是，常量一旦保存某个数据之后，该数据就不能发生改变；但变量保存的数据则可以多次发生改变，只要程序对变量重新赋值即可。

Python 使用等号（ = ）作为赋值运算符，例如 a = 20 就是一条赋值语句，这条语句用于将 20 装入变量 a 中，这个过程就被称为赋值，即将 20 赋值给变量 a。

注意，变量名其实就是标识符，因此在命名时，既要遵守标识符的命名规范，还要避免和 Python 的内置函数重名。

如果你接触过 C、C++、Java 这些高级编程语言会发现，它们在使用变量存储数据时，必须指定该变量所能存储的数据类型，以 C 语言为例：

```
int a = 20
```

这行 C 语言代码的意思是，声明一个整形变量 a 来存储整数 20，这就意味着，我们无法用 a 存储小数、字符以及字符串等其他类型数据，这类编程语言被称为强类型语言。

和强类型语言相对应的是弱类型语言，Python 就是典型代表之一。弱类型语言有以下 2 个特征：

1. 变量无须声明即可直接赋值，对一个不存在的变量赋值就相当于定义了一个新变量。
2. 变量的数据类型可以动态改变：同一个变量可以一会儿被赋值为整数值，一会儿被赋值为字符串。

对于没有编程基础的读者，可以先不编写真正的 Python 程序，而是先打开 Python 的交互式解释器，在这个交互式解释器中“试验” Python。首先，在 Python 解释器中输入如下内容：

```
>>> a = 5
>>>
```

上面代码没有生成任何输出，只是向交互式解释器中存入了一个变量 a，该变量 a 的值为 5。

如果我们想看到某个变量的值，可以直接在交互式解释器中输入该变量。例如，此处想看到变量 a 的值，可以直接输入 a。

```
>>> a
5
>>>
```

可以看到，Python 解释器输出变量 a 的值为 5。

接下来，我们尝试改变变量 a 的值，将新的值赋给变量 a，例如：

```
>>>a = 'Hello , Charlie'
>>>
```

这会导致变量原来的值被新值覆盖掉，换句话说，此时变量 a 的值就不再是 5 了，而是字符串“Hello , Charlie”，a 的类型也变成了字符串。现在再输入 a，让交互式解释器显示 a 的值：

```
>>> a
'Hello , Charlie'
```

如果想查看此时 a 的类型，可以使用 Python 的 type() 内置函数。

```
>>> type(a)
<class 'str'>
>>>
```

可以看到，a 的类型是 str（表示字符串类型）。

形象地说，函数就相当于一个有魔法的“黑盒子”，你可以向这个“黑盒子“提供”一些数据，这个“黑盒子”会对这些数据进行处理，这种处理包括转换和输出结果。比如 print() 也是一个函数，它的作用就是输出传入的数据。此处 type() 函数的作用则用于输出传入数据的类型。

最后，如果想把上面的交互过程转换成真正的 Python 程序，只要将过程中输入的每行代码放在一个文件中，并使用 print() 函数来输出变量，最后将该文件保存为以 .py 结尾的源文件即可。

在交互式解释器中，只要输入变量名，交互式解释器就会输出变量的值，但在 Python 程序中则必须使用 print() 函数来输出变量。

上面的交互过程对应的程序如下：

```
1.          #定义一个数值类型变量
2.          a = 5
3.          print(a)
4.          #重新将字符串赋值给 a 变量
5.          a = 'Hello , Charlie'
```

```
6.          print(a)
7.          print(type(a))
```

通过使用 python 命令或者 IDLE 中的“Run->Run Module”菜单项运行程序（具体操作可阅读《第一个 Python 程序》一节），即可看到如下输出结果：

```
5
Hello , Charlie
<type 'str'>
```

2.Python 数值类型（整形、浮点型和复数）

实际开发中，我们经常需要使用数字记录游戏中用户的得分、游戏中角色的生命值、伤害值等信息，Python 语言提供了数值类型用于保存这些数值。

需要注意的是，Python 中这些数值类型都是不可改变的，也就是说，如果我们要修改数字类型变量的值，那么其底层实现的过程是，先将新值存放到内存中，然后修改变量让其指向新的内存地址，换句话说，Python 中修改数值类型变量的值，其实只是修改变量名所表示的内存空间。

数值类型只是一个泛泛的统称，Python 中的数值类型主要包括整形、浮点型和复数类型。

Python 整型

整形专门用来表示整数，即没有小数部分的数。在 Python 中，整数包括正整数、0 和负

整数。

和其他强类型语言不同，它们会提供多种整形类型，开发者要根据数值的大小，分别用不同的整形类型存储，以 C 语言为例，根据数值的大小，开发者要合理选择 short、int、long 整形类型存储，大大增加了开发难度。

Python 则不同，它的整型支持存储各种整数值，无论多大或者多小，Python 都能轻松处理（当所用数值超过计算机自身的计算功能时，Python 会自动转用高精度计算）。

例如如下代码：

```
#定义变量 a，赋值为 56
a = 56
print(a)
#为 a 赋值一个大整数
a = 999999999999999999999
print(a)
#type()函数用于返回变量的类型
print(type (a))
```

对于没有编程基础的读者， 同样可以在交互式解释器中逐行“试验”上面程序来观看运行效果。由于篇幅限制，本教程后面的程序不再详细列出在交互式解释器中逐行“试验”的过程。

上面程序中，将 999999999999999999999 大整数赋值给变量 a，Python 也不会发生溢出等问题，程序运行一样正常，这足以证明 Python 的强大。

使用 Python 3.x 运行上面程序，可以看到如下输出结果：

```
56
99999999999999999999
<class 'int'>
```

从上面的输出结果可以看出，此时 a 依然是 int 类型。

但如果用 Python 2.x 运行上面程序，由于输入的数值比较大，Python 会自动在其后面加上字母 L（或小写字母 l），如下输出结果：

```
56
99999999999999999999L
<type 'long'>
```

对比两种输出结果，不难发现：不管是 Python 3.x 还是 Python 2.x，Python 完全可以正常处理很大的整数，只是 Python 2.x 底层会将大整数当成 long 类型处理，但开发者通常不需要理会这种细节。

除此之外，Python 的整型还支持 None 值（空值），例如如下代码：

```
a = None
print(a) #什么都不输出
```

Python 的整型数值有 4 种表示形式：

1. 十进制形式：最普通的整数就是十进制形式的整数，在使用十进制表示整数值时，不能以 0（零）作为十进制数的开头（数值是 0 除外）。
2. 二进制形式 :由 0 和 1 组成 ,以 0b 或 0B 开头。例如 ,101 对应十进制数是 5；
3. 八进制形式：八进制整数由 0~7 组成，以 0o 或 0O 开头（第一个字母是零，第二个字母是大写或小写的 O）。需要注意的是，在 Python 2.x 中，八进制数值还可以直接以 0（零）开头。
4. 十六进制形式：由 0~9 以及 A~F（或 a~f）组成，以 0x 或 0X 开头，

下面代码片段使用了其他进制形式的数：

```
#以 0x 或 0X 开头的整型数值是十六进制形式的整数
hex_value1 = 0x13
hex_value2 = 0xaF
print("hexValue1 的值为：",hex_value1)
print("hexValue2 的值为：",hex_value2)
#以 0b 或 0B 开头的整型数值是二进制形式的整数
bin_val = 0b111
print('bin_val 的值为：',bin_val)
bin_val = 0B101
print('bin_val 的值为：',bin_val)
#以 0o 或 0O 开头的整型数值是八进制形式的整数
oct_val = 0o54
print('oct_val 的值为：',oct_val)
oct_val = 0017
print('oct_val 的值为：',oct_val)
```

为了提高数值（包括浮点型）的可读性，Python 3.x 允许为数值（包括浮点型）增加下画线作为分隔符。这些下画线并不会影响数值本身。例如如下代码：

```
#在数值中使用下画线
one_million = 1000000
print(one_million)
price = 234_234_234 #price 实际的值为 234234234
android = 12341234 #android 实际的值为 12341234
```

Python 浮点型

浮点型数值用于保存带小数点的数值，Python 的浮点数有两种表示形式：

- 十进制形式：这种形式就是平常简单的浮点数，例如 5.12、512.0、0.512。浮点数必须包含一个小数点，否则会被当成整数类型处理。
- 科学计数形式：例如 5.12e2（即 5.12×10^2）、5.12E2（也是 5.12×10^2）。

必须指出的是，只有浮点型数值才可以使用科学计数形式表示。例如 51200 是一个整型值，但 512E2 则是浮点型值。

下面程序示范了上面介绍的关于浮点数的各个知识点：

```
af1 = 5.2345556
#输出 af1 的值
print("af1 的值为：",af1)
af2 = 25.2345
print("af2 的类型为：",type(af2))
f1=5.12e2
print("f1 的值为：",f1)
```

```
8.          f2 = 5e3
9.          print("f2 的值为：",f2)
10.         print("f2 的类型为：",type(f2))#看到类型为 float
```

通过第 8 行代码可以看出，虽然 5e3 的值是 5000，但它依然是浮点型值，而不是整型值，因为 Python 会自动将该数值变为 5000.0。

注意，使用浮点数进行计算时，可能会出现小数位数不确定的情况，比如说，计算 0.1+0.2，本来应该得到 0.3，但经过测试你会发现，Python 解释器的运行结果为 0.30000000000000004。几乎所有编程语言都存在这个问题，这与浮点数在底层的存储方式有关系（后续章节会做详细介绍）。

Python 复数

Python 甚至可以支持复数，复数的虚部用 j 或 J 来表示。

> 如果读者对虚数虚部感到困惑，请直接跳过本节，大部分编程并不会用到复数这么“高级”的数学知识。

如果需要在程序中对复数进行计算，可导入 Python 的 cmath 模块（c 代表 complex），在该模块下包含了各种支持复数运算的函数。

> 模块就是一个 Python 程序，Python 正是通过模块提高了自身的可扩展性的；Python 本身内置了大量模块，此外还有大量第三方模块，导入这些模块即可直接使用这些程序中定义的函数。

下面程序示范了复数的用法：

纯文本复制

```
ac1 = 3 + 0.2j
print (ac1)
print(type(ac1)) #输出复数类型
ac2 = 4 - 0.1j
print(ac2)
#复数运行
print(ac1 + ac2) #输出(7+0.1j)
#导入 cmath 模块
import cmath
#sqrt()是 cmath 模块下的商数，用于计算平方根
ac3 = cmath.sqrt(-1)
print (ac3) #输出 1j
```

3.为什么 Python 浮点类型存在误差？

前面章节提到，Python 中浮点类型之间的运算，其结果并不像我们想象的那样，例如：

```
>>> 0.1+0.2
0.30000000000000004
>>> 0.1+0.1-0.2
0.0
>>> 0.1+0.1+0.1-0.3
5.551115123125783e-17
>>> 0.1+0.1+0.1-0.2
0.10000000000000003
```

为什么在计算这么简单的问题上，计算机会出现这样的低级错误呢？真正的原因在于十进制和数和二进制数的转换。

我们知道，计算机其实是不认识十进制数，它只认识二进制数，也就是说，当我们以十进制数进行运算的时候，计算机需要将各个十进制数转换成二进制数，然后进行二进制间的计算。

以类似 0.1 这样的浮点数为例，如果手动将其转换成二进制，其结果为：

$$0.1_{(10)}=0.0001100110011001..._{(2)}$$

可以看到，结果是无限循环的，也就是说，0.1 转换成二进制数后，无法精确到等于十进制数的 0.1。同时，由于计算机存储的位数是有限制的，所以如果要存储的二进制位数超过了计算机存储位数的最大值，其后续位数会被舍弃（舍弃的原则是“0 舍 1 入”）。

这种问题不仅在 Python 中存在，在所有支持浮点数运算的编程语言中都会遇到，它不光是 Python 的 Bug。

明白了问题产生的原因之后，那么该如何解决呢？就 Python 的浮点数运算而言，大多数计算机每次计算误差不会超过 2^{53}，这对于大多数任务来说已经足够了。

如果需要非常精确的结果，可以使用 decimal 模块（其实就是别人开发好的程序，我们可以直接拿来用），它实现的十进制数运算适合会计方面的应用和有高精度要求的应用。例如：

```
#使用模块前，需要使用 import 引入
import decimal
a = decimal.Decimal("10.0")
b = decimal.Decimal("3")
```

```
5.          print(10.0/3)
6.          print(a/b)
```

运行结果为：

```
3.3333333333333335

3.33333333333333333333333333
```

可以看到，相比普通运算的结果，使用 decimal 模块得到的结果更精确。

如果 decimal 模块还是无法满足需求，还可以使用 fractions 模块，例如：

```
1.          #引入 decimal 模块
2.          from fractions import Fraction
3.          print(10/3)
4.          print(Fraction(10,3))
```

运行结果为：

```
3.3333333333333335

10/3
```

可以看到，通过 fractions 模块能很好地解决浮点类型数之间运算的问题。

本节涉及到了有关模块的使用，读者不用研究细节，会用即可，后续章节会详细介绍模块。

4.Python 字符串

简单地理解，字符串就是“一串字符”，也就是用引号内的任何数据，比如“Hello,Charlie”是一个字符串，“How are you？”也是一个字符串。

Python 要求，字符串必须使用引号括起来，可以使用单引号或者双引号，只要成对即可。

字符串中的内容几乎可以包含任何字符，英文字符也行，中文字符也行。

> Python 3.x 对中文字符支持较好，但 Python 2.x 则要求在源程序中增加“#coding:utf-8”才能支持中文字符。

至于字符串是用单引号括起来，还是用双引号括起来，在 Python 语言中，它们没有任何区别。比如说：

```
str1 = 'c.biancheng.net'
str2 = "C 语言中文网"
print(str1)
print(str2)
```

但需要说明的是，Python 有时候没有我们期望的那么聪明。如果字符串内容本身包含了单引号或双引号，此时就需要进行特殊处理：

1. 使用不同的引号将字符串括起来。
2. 对引号进行转义。

先看第一种处理方式。假如字符串内容中包含了单引号，则可以使用双引号将字符串括起来。例如：

```
str3 = "I'm a coder"
```

由于上面字符串中包含了单引号，此时 Python 会将字符串中的单引号与第一个单引号配

对，这样就会把 'I' 当成字符串，而后面的 m a coder' 就变成了多余的内容，从而导致语法错误。

为了避免这种问题，可以将上面代码改为如下形式：

```
str3 = "I'm a coder"
```

上面代码使用双引号将字符串括起来，此时 Python 就会把字符串中的单引号当成字符串内容，而不是和字符串开始的引号配对。

假如字符串内容本身包含双引号，则可使用单引号将字有串括起来，例如如下代码：

```
str4 = '"Spring is here,let us jam!", said woodchuck.'
```

接下来看第二种处理方式：使用转义字符。Python 允许使用反斜线（\）将字符串中的特殊字符进行转义。假如字符串既包含单引号，又包含双引号，此时就可以使用转义字符，例如：

```
str5 = '"we are scared,Let\'s hide in the shade",says the bird'
```

通过使用转义字符，向 Python 解释器表明了此单引号并不是和最前面的单引号进行配对的另一半，从而避免了发生语法错误。

此外，Python 不是格式自由的语言，它对程序的换行、缩进都有其规定的语法。Python 允许使用转义字符（\）对换行符进行转义，转义之后的换行符不会“中断”字符串。例如：

```
1.          s2 = 'The quick brown fox \
2.          jumps over the lazy dog'
3.          print(s2)
```

上面 s2 字符串的内容较长，故程序使用了转义字符（\）对内容进行了转义，这样就可以把一个字符串写成两行。

同样的，Python 的表达式也不允许随便换行。但如果程序需要对 Python 表达式换行，就需要使用转义字符（\），例如：

```
1.          num = 20 + 3 / 4 + \
2.              2 * 3
3.          print(num)
```

上面程序中有一个表达式，为了对该表达式换行，程序用到了转义字符。

> 有关转义字符，后续章节会做详细介绍。

Python 长字符串

前面介绍 Python 多行注释时，提到使用三个引号（单引号、双引号都行）来包含多行注释内容，其实这是长字符串的写法，只是由于在长字符串中可以放置任何内容，包括放置单引号、双引号都可以，如果所定义的长字符串没有赋值给任何变量，那么这个字符串就相当于被解释器忽略了，也就相当于注释掉了。

实际上，使用三个引号括起来的长字符串完全可以赋值给变量，例如如下程序：

```
s = '''"Let's go fishing", said Mary.
"OK, Let's go", said her brother.
they walked to a lake'''
print(s)
```

上面程序使用三个引号定义了长字符串，该长字符串中既可包含单引号，也可包含双引号。

当程序中有大段文本内容要定义成字符串时，优先推荐使用长字符串形式，因为这种形式非常强大，可以让字符串中包含任何内容，既可包含单引号，也可包含双引号。

Python 原始字符串

由于字符串中的反斜线都有特殊的作用，因此当字符串中包含反斜线时，就需要使用转义字符 \ 对字符串中包含的每个 '\' 进行转义。

比如说，我们要写一个关于 Windows 路径 G:\publish\codes\02\2.4 这样的字符串，如果在 Python 程序中直接这样写肯定是不行的，需要使用 \ 转义字符，对字符串中每个 '\' 进行转义，即写成 G:\\publish\\codes\\02\\2.4 这种形式才行。

有没有觉得这种写法很啰嗦，有没有更好的解决办法呢？答案是肯定的，借助于原始字符串可以很好地解决这个问题。

原始字符串以“r”开头，它不会把反斜线当成特殊字符。因此，上面的 Windows 路径可

直接写成如下这种形式：

```
s1 = r'G:\publish\codes\02\2.4'
print(s1)
```

如果原始字符串中包含引号，程序同样需要对引号进行转义（否则 Python 同样无法对字符串的引号精确配对），但此时用于转义的反斜线会变成字符串的一部分。

例如如下代码：

```
# 原始字符串包含的引号，同样需要转义
s2 = r'"Let\'s go", said Charlie'
print(s2)
```

上面代码会生成如下输出结果：

```
"Let\'s go", said Charlie
```

由于原始字符串中的反斜线会对引号进行转义，因此原始字符串的结尾处不能是反斜线，否则字符串结尾处的引号会被转义，导致字符串不能正确结束。

如果确实要在原始字符串的结尾处添加反斜线怎么办呢？一种方式是不要使用原始字符串，而是改为使用长字符串写法（三引号字符串）；另一种方式就是将反斜线单独写。

例如如下代码：

```
s3 = r'Good Morning' '\\'
print(s3)
```

上面代码开始写了一个原始字符串 r'Good Morning'，紧接着程序使用 '\\' 写了一个包含反斜线的字符串，Python 会自动将这两个字符串拼接在一起。运行上面代码会生成如下输出结果：

```
Good Morning\
```

由于这种方式涉及到了字符串拼接的相关知识，这里读者只需要了解即可，后续会对字符串拼接做详细介绍。

5.python 字符串使用哪种编码格式？

在实践中，很多初学者都遇到过“文件显示乱码”的情况，其多数都是由于在打开文件时，没有选对编码格式导致的。因此，学习 Python 中的字符或字符串，了解其底层的编码格式是非常有必要的。

鉴于有些读者并不了解什么是编码格式，本节先从编码开始讲起。

什么是编码？

虽然很多教程中有关于编码的定义，但对初学者来说并不容易理解，这里先举一个例子。古代打仗，击鼓为号、鸣金收兵，即把要传达给士兵的命令对应为公认的其他形式，这就和编码有相似之处。

以发布进攻命令为例，相比用嗓子喊，敲鼓发出的声音传播的更远，并且士兵听到后也不会引起歧义，因此长官下达进攻命令后，传令员就将此命令转化为对应的鼓声，这个转化的过

程称为编码；由于士兵都接受过训练，听到鼓声后，他们可以将其转化为对应的进攻命令，这个转化的过程称为解码。

需要说明的是，此例只是形象地描述了编码和解码的原理，真实的编码和解码过程比这要复杂的多。

了解了编码的含义之后，接下来再介绍一下字符编码。

什么是字符编码？

我们知道，计算机是以二进制的形式来存储数据的，即它只认识 0 和 1 两个数字。 20 世纪 60 年代，是计算机发展的早期，这时美国是计算机领域的老大，它制定了一套编码标准，解决了 128 个英文字符与二进制之间的对应关系，被称为 ASCII 字符编码(简称 ASCII 码)。

ASCII 码，全称为美国信息交换标准代码，是基于拉丁字母的一套字符编码，主要用于显示现代英语，因为万维网的出现，使得 ASCII 码广为使用，其直到 2007 年 12 月才逐渐被 Unicode 取代。

虽然英语用 128 个字符编码已经够用，但计算机不仅仅用于英语，如果想表示其他语言，128 个符号显然不够用，所以很多其他国家都在 ASCII 的基础上发明了很多别的编码，例如包含了汉语简体中文格式的 GB2312 编码格式（使用 2 个字节表示一个汉字）。

也正是由于出现了很多种编码格式，导致了“文件显示乱码”的情况。比如说，发送邮件时，如果发信人和收信人使用的编码格式不一样，则收信人很可能看到乱码的邮件。基于这个原因，Unicode 字符集应运而生。

Unicode 字符集又称万国码、国际码、统一码等。从名字就可以看出来，它是以统一符号为目标的字符集。Unicode 对世界上大部分的文字系统进行了整理、编码，使得电脑可以用更简单的方式来呈现和处理文字。

注意，在实际使用时，人们常常混淆字符集和字符编码这两个概念，我认为它们是不同的：

- 字符集定义了字符和二进制的对应关系，为每个字符分配了唯一的编号。可以将字符集理解成一个很大的表格，它列出了所有字符和二进制的对应关系，计算机显示文字或者存储文字，就是一个查表的过程；
- 而字符编码规定了如何将字符的编号存储到计算机中，要知道，有些字符编码（如 GB2312 和 GBK）规定，不同字符在存储时所占用的字节数是不一样的，因此为了区分一个字符到底使用了几个字节，就不能将字符的编号直接存储到计算机中，字符编号在存储之前必须要经过转换，在读取时还要再逆向转换一次，这套转换方案就叫做字符编码。

Unicode 字符集可以使用的编码方案有三种，分别是：

1. UTF-8：一种变长的编码方案，使用 1~6 个字节来存储；
2. UTF-32：一种固定长度的编码方案，不管字符编号大小，始终使用 4 个字节来存储；
3. UTF-16：介于 UTF-8 和 UTF-32 之间，使用 2 个或者 4 个字节来存储，长度既固定又可变。

其中，UTF-8 是目前使用最广的一种 Unicode 字符集的实现方式，可以说它几乎已经一统江湖了。

Python 使用哪种字符编码？

了解了什么是编码，以及什么是字符编码之后，最后解决“Python 使用哪种字符编码？”这个问题。

Python 3.x 中，字符串采用的是 Unicode 字符集，可以用如下代码来查看当前环境的编码格式：

```
>>> import sys
>>> sys.getdefaultencoding()
'utf-8'
```

同时，在 Python 3.x 中也可以用 ord() 和 chr() 函数实现字符和编码数字之间的转换，例如：

```
>>> ord('Q')
81
>>> chr(81)
'Q'
>>> ord("网")
```

```
32593
>>> chr(32593)
'网'
```

Python 2.x 中无法使用 ord() 得到指定字符对应的编码数字。

由此可以知道，在 Unicode 字符集中，字符‘Q’对应的编码数字为 81，而中文‘网’对应的编码数字为 32593。

值得一提的是，虽然 Python 默认采用 UTF-8 编码，但它也提供了 encode() 方法，可以轻松实现将 Unicode 编码格式的字符串转化为其它编码格式。有关 encode() 方法的用法，可阅读《Python encode()和 decode()方法》 一节。

6.Python bytes

Python 3 新增了 bytes 类型，用于代表字节串（这是本教程创造的一个词，用来和字符串对应）。字符串（str）由多个字符组成，以字符为单位进行操作；字节串（bytes）由多个字节组成，以字节为单位进行操作。

bytes 和 str 除操作的数据单元不同之外，它们支持的所有方法都基本相同，bytes 也是不可变序列。

bytes 对象只负责以字节（二进制格式）序列来记录数据，至于这些数据到底表示什么内容，完全由程序决定。如果采用合适的字符集，字符串可以转换成字节串；反过来，字节串也可以恢复成对应的字符串。

由于 bytes 保存的就是原始的字节（二进制格式）数据，因此 bytes 对象可用于在网络上传输数据，也可用于存储各种二进制格式的文件，比如图片、音乐等文件。

如果希望将一个字符串转换成 bytes 对象，有如下三种方式：

1. 如果字符串内容都是 ASCII 字符，则可以通过直接在字符串之前添加 b 来构建字节串值。
2. 调用 bytes() 函数（其实是 bytes 的构造方法）将字符串按指定字符集转换成字节串，如果不指定字符集，默认使用 UTF-8 字符集。
3. 调用字符串本身的 encode() 方法将字符串按指定字符集转换成字节串，如果不指定字符集，默认使用 UTF-8 字符集。

例如，如下程序示范了如何创建字节串：

```
# 创建一个空的 bytes
b1 = bytes()
# 创建一个空的 bytes 值
b2 = b''
# 通过 b 前缀指定 hello 是 bytes 类型的值
b3 = b'hello'
print(b3)
print(b3[0])
print(b3[2:4])
# 调用 bytes 方法将字符串转成 bytes 对象
b4 = bytes('我爱 Python 编程',encoding='utf-8')
print(b4)
# 利用字符串的 encode()方法编码成 bytes，默认使用 utf-8 字符集
```

```
14.         b5 = "学习 Python 很有趣".encode('utf-8')
15.         print(b5)
```

上面程序中 b1～b5 都是字节串对象，该程序示范了以不同方式来构建字节串对象。其中 b2、b3 都是直接在 ASCII 字符串前添加 b 前缀来得到字节串的：b4 调用 bytes() 函数来构建字节串；而 b5 则调用字符串的 encode 方法来构建字节串。

运行上面程序，可以看到如下输出结果：

```
b'hello'
104
b'll'
b'\xe6\x88\x91\xe7\x88\xb1Python\xe7\xbc\x96\xe7\xa8\x8b'
b'\xe5\xad\xa6\xe4\xb9\xa0Python\xe5\xbe\x88\xe6\x9c\x89\xe8\xb6\xa3'
```

从上面的输出结果可以看出，字节串和字符串非常相似，只是字节串里的每个数据单元都是 1 字节。

> 计算机底层有两个基本概念：位（bit）和字节（Byte），其中 bit 代表 1 位，要么是 0，要么是 1，就像一盏灯，要么打开，要么熄灭；Byte 代表 1 字节，1 字节包含 8 位。

在字节串中每个数据单元都是字节，也就是 8 位，其中每 4 位（相当于 4 位二进制数，最小值为 0 ，最大值为 15）可以用一个十六进制数来表示，因此每字节需要两个十六进制数表示，所以可以看到上面的输出是 b'\xe6\x88\x91\xe7\x88\xb1Python\xe7\xbc\x96\xe7\xa8\x8b'，比如 \xe6 就表示 1

字节，其中 \x 表示十六进制，e6 就是两位的十六进制数。

如果程序获得了 bytes 对象，也可调用 bytes 对象的 decode() 方法将其解码成字符串，例如，在上面程序中添加如下代码：

```
#将 bytes 对象解码成字符串，默认使用 UTF-8 进行解码
st = b5.decode('utf-8')
print(st)#学习 Python 很有趣
```

运行上面程序，可以看到如下输出结果：

```
学习 Python 很有趣
```

这里简单介绍一下字符集的概念。计算机底层并不能保存字符，但程序总是需要保存各种字符的，那该怎么办呢？计算机“科学家”就想了一个办法：为每个字符编号，当程序要保存字符时，实际上保存的是该字符的编号；当程序读取字符时，读取的其实也是编号，接下来要去查“编号—字符对应表”（简称码表）才能得到实际的字符。

因此，所谓的字符集，就是所有字符的编号组成的总和。早期美国人给英文字符、数字、标点符号等字符进行了编号，他们认为所有字符加起来顶多 100 多个，只要 1 字节（8 位，支持 256 个字符编号）即可为所有字符编号——这就是 ASCII 字符集。

后来，亚洲国家纷纷为本国文字进行编号，即制订本国的字符集，但这些字符集并不兼容。于是美国人又为世界上所有书面语言的字符进行了统一编号，这次他们用了两个字节(16 位，支持 65536 个字符编号），这就是 Unicode 字符集。实际使用的 UTF-8, UTF-16 等

其实都属于 Unicode 字符集。

由于不同人对字符的编号完全可以很随意，比如同一个“爱”字，我可以为其编号为 99，你可以为其编号为 199，所以同一个编号在不同字符集中代表的字符完全有可能是不同的。因此，对于同一个字符串，如果采用不同的字符集来生成 bytes 对象，就会得到不同的 bytes 对象。

7.Python bool 布尔类型

Python 提供了 bool 类型来表示真（对）或假（错），比如常见的 5 > 3 比较算式，这个是正确的，在程序世界里称之为真（对），Python 使用 True 来代表；再比如 4 > 20 比较算式，这个是错误的，在程序世界里称之为假（错），Python 使用 False 来代表。

> True 和 False 是 Python 中的关键字，当作为 Python 代码输入时，一定要注意字母的大小写，否则解释器会报错。

值得一提的是，布尔类型可以当做整数来对待，即 True 相当于整数值 1，False 相当于整数值 0。因此，下边这些运算都是可以的：

```
>>> False+1
1
>>> True+1
2
```

注意，这里只是为了说明 True 和 Flase 对应的整形值，在实际应用中是不妥的，不要这么用。

总的来说，bool 类型就是用于代表某个事情的真（对）或假（错），如果这个事情是正确的，用 True（或 1）代表；如果这个事情是错误的，用 False（或 0）代表。

【例 1】

```
>>> 5>3
True
>>> 4>20
False
```

在 Python 中，所有的对象都可以进行真假值的测试，包括字符串、元组、列表、字典、对象等，由于目前尚未学习，因此这里不做过多讲述，后续遇到时会做详细的介绍。

8.Python len()函数：获取字符串长度或字节数

Python 中，要想知道一个字符串有多少个字符（获得字符串长度），或者一个字符串占用多少个字节，可以使用 len 函数。

len 函数的基本语法格式为：

```
len（string）
```

其中 string 用于指定要进行长度统计的字符串。

例如，定义一个字符串，内容为“http://c.biancheng.net”，然后用 len() 函数计算该字符串的长度，执行代码如下：

```
>>> a='http://c.biancheng.net'
>>> len(a)
22
```

在实际开发中，除了常常要获取字符串的长度外，有时还要获取字符串的字节数。

在 Python 中，不同的字符所占的字节数不同，数字、英文字母、小数点、下划线以及空格，各占一个字节，而一个汉字可能占 2~4 个字节，具体占多少个，取决于采用的编码方式。例如，汉字在 GBK/GB2312 编码中占用 2 个字节，而在 UTF-8 编码中一般占用 3 个字节。

以 UTF-8 编码为例，字符串“人生苦短，我用 Python”所占用的字节数如图 1 所示。

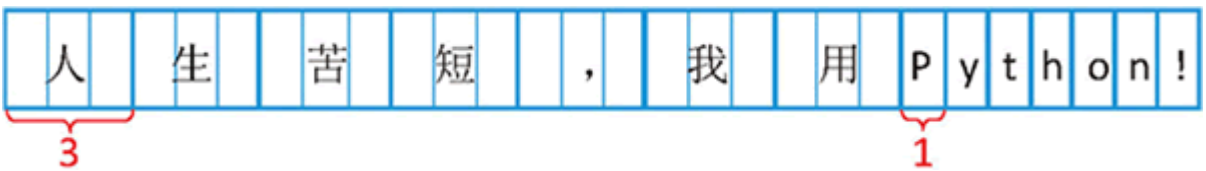


图 1 汉字和英文所占字节数

我们可以通过使用 encode() 方法，将字符串进行编码后再获取它的字节数。例如，采用 UTF-8 编码方式，计算“人生苦短，我用 Python”的字节数，可以执行如下代码：

```
>>> str1 = "人生苦短，我用 Python"
>>> len(str1.encode())
27
```

因为汉字加中文标点符号共 7 个，占 21 个字节，而英文字母和英文的标点符号占 6 个字节，一共占用 27 个字节。

同理，如果要获取采用 GBK 编码的字符串的长度，可以执行如下代码：

```
>>> str1 = "人生苦短，我用 Python"
>>> len(str1.encode('gbk'))
20
```

9.Python input()函数：获取用户输入的字符串

input() 函数用于向用户生成一条提示，然后获取用户输入的内容。由于 input() 函数总会将用户输入的内容放入字符串中，因此用户可以输入任何内容，input() 函数总是返回一个字符串。

例如如下程序：

```
msg = input("请输入你的值：")
print (type(msg))
print(msg)
```

第一次运行该程序，我们输入一个整数，运行过程如下：

```
请输入你的值：2
<class 'str'>
2
```

第二次运行该程序，我们输入一个浮点数，运行过程如下：

```
请输入你的值： 1.2
<class 'str'>
1.2
```

第三次运行该程序，我们输入一个字符串，运行过程如下：

```
请输入你的值：Hello
<class 'str'>
Hello
```

从上面的运行过程可以看出，无论输入哪种内容，始终可以看到 input() 函数返回字符串，程序总会将用户输入的内容转换成字符串。

需要指出的是，Python 2.x 提供了一个 raw_input() 函数，该 raw_input() 函数就相当于 Python 3.x 中的 input() 函数。

而 Python 2.x 也提供了一个 input() 函数，该 input() 函数则比较怪异：要求用户输入的必须是符合 Python 语法的表达式。通常来说，用户只能输入整数、浮点数、复数、字符串等。重点是格式必须正确，比如输入字符串时必须使用双引号，否则 Python 就会报错。

使用 Python 2.x 来运行上面程序，假如输入一个整数，运行过程如下：

```
请输入你的值：2
<class 'int'>
2
```

使用 Python 2.x 来运行上面程序，假如输入一个复数，运行过程如下：

```
请输入你的值： 2+3j
<type 'complex'>
(2+3j)
```

使用 Python 2.x 来运行上面程序，假如输入一个字符串，运行过程如下：

```
请输入你的值：Hello
NameError : name 'Hello' is not defined
```

上面程序报错的原因是：Python 2.x 的 input() 函数要求用户输入字符串时必须用引号把字符串括起来。

> 在 Python 2.x 中应该尽量使用 raw_input() 函数来获取用户输入，因为 Python 2.x 中的 raw_input() 等同于 Python 3.x 中的 input()。

10.Python print()函数高级用法

前面使用 print() 函数时，都只输出了一个变量，但实际上 print() 函数完全可以同时输出多个变量，而且它具有更多丰富的功能。

print() 函数的详细语法格式如下：

```
print (value,...,sep='',end='\n',file=sys.stdout,flush=False)
```

从上面的语法格式可以看出，value 参数可以接受任意多个变量或值，因此 print() 函数完全可以输出多个值。例如如下代码：

```
1.          user_name = 'Charlie'
2.          user_age = 8
3.          #同时输出多个变量和字符串
4.          print("读者名：",user_name,"年龄：",user_age)
```

运行上面代码，可以看到如下输出结果：

```
读者名： Charlie 年龄： 8
```

从输出结果来看，使用 print() 函数输出多个变量时，print() 函数默认以空格隔开多个变量，如果读者希望改变默认的分隔符，可通过 sep 参数进行设置。例如输出语句：

```
#同时输出多个变量和字符串，指定分隔符
print("读者名：" ,user_name,"年龄：",user_age,sep='|')
```

运行上面代码，可以看到如下输出结果：

```
读者名：|Charlie|年龄：|8
```

在默认情况下，print() 函数输出之后总会换行，这是因为 print() 函数的 end 参数的默认值是“\n” ，这个“\n”就代表了换行。如果希望 print() 函数输出之后不会换行，则重设 end 参数即可，例如如下代码：

```
#设置 end 参数，指定输出之后不再换行
print(40,'\t',end="")
print(50,'\t',end="")
print(60,'\t',end="")
```

上面三条 print() 语句会执行三次输出，但由于它们都指定了 end = ""，因此每条 print() 语句的输出都不会换行，依然位于同一行。运行上面代码，可以看到如下输出结果：

```
40    50    60
```

file 参数指定 print() 函数的输出目标，file 参数的默认值为 sys.stdout，该默认值代表了系统标准输出，也就是屏幕，因此 print() 函数默认输出到屏幕。实际上，完全可以通过改变该参数让 print() 函数输出到特定文件中，例如如下代码：

```
f = open("demo.txt","w")#打开文件以便写入
print('沧海月明珠有泪',file=f)
print('蓝回日暖玉生烟',file=f)
f.close()
```

上面程序中，open() 函数用于打开 demo.txt 文件，接连 2 个 print 函数会将这 2 段字符串依次写入此文件，最后调用 close() 函数关闭文件，教程后续章节还会详细介绍关于文件操作的内容。

print() 函数的 flush 参数用于控制输出缓存，该参数一般保持为 False 即可，这样可以获得较好的性能。

11.Python 格式化字符串

Python 提供了“%”对各种类型的数据进行格式化输出，例如如下代码：

```
price = 108
print ("the book's price is %s" % price)
```

上面程序中的 print 函数包含以下三个部分，第一部分是格式化字符串（相当于字符串模板），该格式化字符串中包含一个“%s”占位符，它会被第三部分的变量或表达式的值代替；第二部分固定使用“%”作为分隔符。

格式化字符串中的“%s”被称为转换说明符（Conversion Specifier），其作用相当于一个占位符，它会被后面的变量或表达式的值代替。“%s”指定将变量或值使用 str() 函数转换为字符串。

如果格式化字符串中包含多个“%s”占位符，第三部分也应该对应地提供多个变量，并且使用圆括号将这些变量括起来。例如如下代码：

```
user = "Charli"
age = 8
# 格式化字符串有两个占位符，第三部分提供 2 个变量
print("%s is a %s years old boy" % (user , age))
```

在格式化字符串中难道只能使用“%s”吗？还有其他转换说明符吗？如果只有“%s”这一种形式，Python 的格式化功能未免也太单一了。

实际上， Python 提供了如表 1 所示的转换说明符：

表 1 Python 转换说明符

转换说明符	说明
%d，%i	转换为带符号的十进制形式的整数
%o	转换为带符号的八进制形式的整数
%x，%X	转换为带符号的十六进制形式的整数
%e	转化为科学计数法表示的浮点数（e 小写）
%E	转化为科学计数法表示的浮点数（E 大写）
%f，%F	转化为十进制形式的浮点数
%g	智能选择使用 %f 或 %e 格式

%G	智能选择使用 %F 或 %E 格式
%c	格式化字符及其 ASCII 码
%r	使用 repr() 将变量或表达式转换为字符串
%s	使用 str() 将变量或表达式转换为字符串

当使用上面的转换说明符时，可指定转换后的最小宽度，例如如下代码：

```
num = -28
print("num is: %6i" % num)
print("num is: %6d" % num)
print("num is: %6o" % num)
print("num is: %6x" % num)
print("num is: %6X" % num)
print("num is: %6s" % num)
```

运行上面代码，可以看到如下输出结果：

```
num is:   -28
num is:   -28
num is:   -34
num is:   -1c
num is:   -1C
num is:   -28
```

从上面的输出结果可以看出，此时指定了字符串的最小宽度为 6，因此程序转换数值时总宽度为 6，程序自动在数值前面补充了三个空格。

在默认情况下，转换出来的字符串总是右对齐的，不够宽度时左边补充空格。Python 也允许在最小宽度之前添加一个标志来改变这种行为，Python 支持如下标志：

- -：指定左对齐。
- +：表示数值总要带着符号（正数带“+”，负数带“-”）。
- 0：表示不补充空格，而是补充 0。

> 提示：这三个标志可以同时存在。

例如如下代码：

```
num2 = 30
# 最小宽度为 0，左边补 0
print("num2 is: %06d" % num2)
# 最小宽度为 6，左边补 0，总带上符号
print("num2 is: %+06d" % num2)
# 最小宽度为 6，右对齐
print("num2 is: %-6d" % num2)
```

运行上面代码，可以看到如下输出结果：

```
num2 is: 000030

num2 is: +00030

num2 is: 30
```

对于转换浮点数，Python 还允许指定小数点后的数字位数：如果转换的是字符串，Python 允许指定转换后的字符串的最大字符数。这个标志被称为**精度值**，该精度值被放在最小宽度之后，中间用点 () 隔开。例如如下代码：

```
my_value = 3.001415926535
# 最小宽度为 8，小数点后保留 3 位
print("my_value is: %8.3f" % my_value)
# 最小宽度为 8，小数点后保留 3 位，左边补 0
print("my_value is: %08.3f" % my_value)
# 最小宽度为 8，小数点后保留 3 位，左边补 0，始终带符号
print("my_value is: %+08.3f" % my_value)
the_name = "Charlie"
# 只保留 3 个字符
print("the name is: %.3s" % the_name) # 输出 Cha
# 只保留 2 个字符，最小宽度 10
print("the name is: %10.2s" % the_name)
```

运行上面代码，可以看到如下输出结果：

```
my_value is:    3.001
my_value is: 0003.001
my_value is: +003.001
the name is: Cha
the name is:         Ch
```

12.Python 转义字符

在前面的章节中，我们曾经简单学习过转义字符，所谓转义，可以理解为“采用某些方式暂时取消该字符本来的含义”，这里的“某种方式”指的就是在指定字符前添加反斜杠 \，以

此来表示对该字符进行转义。

举个例子，在 Python 中单引号（或双引号）是有特殊作用的，它们常作为字符（或字符串）的标识（只要数据用引号括起来，就认定这是字符或字符串），而如果字符串中包含引号（例如 'I'm a coder'），为了避免解释器将字符串中的引号误认为是包围字符串的“结束”引号，就需要对字符串中的单引号进行转义，使其在此处取消它本身具有的含义，告诉解释器这就是一个普通字符。

因此这里需要使用单引号 ' 的转义字符 \'，尽管它由 2 个字符组成，但通常将它看做是一个整体，是一个转义字符。我们已经见过很多类似的转义字符，包括 \'、\"、\\ 等。

Python 不只有以上几个转义字符， Python 中常用的转义字符如表 1 所示：

表 1 Python 支持的转义字符

转义字符	说明
\	在行尾的续行符，即一行未完，转到下一行继续写
\'	单引号
\"	双引号
\0	空
\n	换行符

\r	回车符
\t	水平制表符，用于横向跳到下一制表位
\a	响铃
\b	退格（Backspace）
\\	反斜线
\0dd	八进制数，dd 代表字符，如 \012 代表换行
\xhh	十六进制数，hh 代表字符，如 \x0a 代表换行

掌握了上面的转义字符之后，下面在字符串中使用它们，例如如下代码：

```
s = 'Hello\nCharlie\nGood\nMorning'
print(s)
```

运行上面代码，可以看到如下输出结果：

```
Hello
Charlie
Good
Morning
```

也可以使用制表符进行分隔，例如如下代码：

纯文本复制

```
s2 = '商品名\t\t 单价\t\t 数量\t\t 总价'
s3 = 'C 语言小白变怪兽\t99\t\t2\t\t198'
```

```
        print(s2)
        print (s3)
```

运行上面代码，可以看到如下输出结果：

```
商品名                 单价      数量      总价
C 语言小白变怪兽  99          2           198
```

13.Python 数据类型转换

虽然 Python 是弱类型编程语言，不需要像 Java 或 C 语言那样还要在使用变量前声明变量的类型，但在一些特定场景中，仍然需要用到类型转换。

比如说，我们想通过使用 print() 函数输出信息“您的身高：”以及浮点类型 height 的值，如果在交互式解释器中执行如下代码：

```
>>> height = 70.0
>>> print("您的身高"+height)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print("您的身高"+height)
TypeError: must be str, not float
```

你会发现这是错误的，解释器提示我们字符串和浮点类型变量不能直接相连，需要提前将浮点类型变量 height 转换为字符串才可以。

庆幸的是，Python 已经为我们提供了多种可实现数据类型转换的函数，如表 1 所示。

表 1 常用数据类型转换函数

函 数	作 用
int(x)	将 x 转换成整数类型
float(x)	将 x 转换成浮点数类型
complex(real，[,imag])	创建一个复数
str(x)	将 x 转换为字符串
repr(x)	将 x 转换为表达式字符串
eval(str)	计算在字符串中的有效 Python 表达式，并返回一个对象
chr(x)	将整数 x 转换为一个字符
ord(x)	将一个字符 x 转换为它对应的整数值
hex(x)	将一个整数 x 转换为一个十六进制字符串
oct(x)	将一个整数 x 转换为一个八进制的字符串

需要注意的是，在使用类型转换函数时，提供给它的数据必须是有意义的。例如，int() 函数无法将一个非数字字符串转换成整数：

```
>>> int("123") #转换成功
123
>>> int("123 个") #转换失败
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    int("123 个")
ValueError: invalid literal for int() with base 10: '123 个'
>>>
```

14.Python 算术运算符

算术运算符是处理四则运算的符号，在数字的处理中应用得最多。Python 支持所有的基本算术运算符，如表 1 所示。

表 1 Python 常用算术运算符

运算符	说明	实例	结果
+	加	12.45 + 15	27.45
-	减	4.56 - 0.26	4.3
*	乘	5 * 3.6	18.0
/	除	7 / 2	3.5

%	取余，即返回除法的余数	7 % 2	1
//	整除，返回商的整数部分	7 // 2	3
**	幂，即返回 x 的 y 次方	2 ** 4	16，即 2^4

接下来将对表 1 中各个算术运算符的用法，一一做详细讲解。

+ 加法运算符

例如如下代码：

```
a = 5.2
b = 3.1
the_sum = a + b
# sum 的值为 8.3
print("the_sum 的值为: ", the_sum)
```

除此之外，“+”还可以作为字符串（包括后续要介绍的序列）的连接运算符。例如如下代码：

```
s1 = 'Hello, '
s2 = 'Charlie'
# 使用+连接两个字符串
print(s1 + s2)
```

- 减法运算符

例如如下代码：

```
c = 5.2
d = 3.1
```

```
sub = c - d
# sub 的值为 2.1
print("sub 的值为：", sub)
```

此外，“-”除了可以作为减法运算符之外，还可以作为求负的运算符。请看如下代码：

```
# 定义变量 x，其值为-5.0
x = -5.0
# 将 x 求负，其值变成 5.0
x = -x
print(x)
```

但单目运算符“+”则不对操作数做任何改变，例如如下代码：

```
# 定义变量 y，其值为-5.0
y = -5.0
# y 值依然是-5.0
y = +y
print(y)
```

*乘法运算符

例如如下代码：

```
e = 5.2
f = 3.1
multiply = e * f
# multiply 的值为 16.12
print("multiply 的值为：", multiply)
```

此外，“*”还可以作为字符串（包括后续要介绍的序列）的连接运算符，表示将 N 个字符串连接起来。例如如下代码：

```
s3 = 'crazyit '
# 使用*将 5 个字符串连接起来
```

```
3.          print(s3 * 5)
```

上面代码将会输出：

```
crazyit crazyit crazyit crazyit crazyit
```

/ 和 // 除法运算符

Python 的除法运算符有两个："/" 表示普通除法，使用它除出来的结果与平常数学计算的结果是相同的（即除不尽时，会产生小数部分）；而 "//" 表示整除，使用它除出来的结果只有整数部分，小数部分将会被舍弃。

记住，在 Python 3.x 中，除法运算的结果都是浮点类型。

例如如下代码：

```
1.          print("19/4 的结果是:", 19/4)
2.          print("19//4 的结果是:", 19//4)
3.          aa = 5.2
4.          bb = 3.1
5.          # aa / bb 的值将是 1.67741935483871
6.          print("aa/bb 的值是:", aa / bb)
7.          # aa // bb 值将是 1.0
8.          print("aa//bb 的值是:", aa // bb)
```

此外，Python 不允许使用 0 作为除数，否则将会引发 ZeroDivisionError 错误。

在有些编程语言中，0 作为除数会得到无穷大，包括正无穷大或负无穷大。

对于 Python 2.x 而言，它只提供了一个 "/" 运算符，该运算符是 Python 3.x 中 "/" 和 "//" 的综合版。对于 Python 2.x 中的 "/" 而言，它既是整除运算符，也是非整除运算

符。规则如下：

- 当两个操作数都是整数时，Python 2.x 中的“/”就是整除运算符。
- 当两个操作数有一个是浮点数（或两个都是浮点数）时，Python 2.x 中的“/”就是非整除运算符。

例如，在 Python 2.x 的交互式解释执行器中执行如下代码：

```
>>>print(5/2)
2
>>>print(5.0/2)
2.5
>>>print(5/2.0)
2.5
>>>print(5.0/2.0)
2.5
```

% 求余运算符

Python 不要求求余运算符的两个操作数都是整数，Python 的求余运算符完全支持对浮点数求余。求余运算的结果不一定总是整数，它是使用第一个操作数来除以第二个操作数，得到一个整除的结果后剩下的值就是余数。

由于求余运算也需要进行除法运算，因此求余运算的第二个操作数不能是 0，否则程序会

报出 ZeroDivisionError 错误。

例如如下程序：

```
print("5%3 的值为：", 5 % 3) # 输出 2
print("5.2%3.1 的值为：",5.2 % 3.1) # 输出 2.1
print("-5.2%-3.1 的值为：", -5.2 % -3.1) # 输出-2.1
print("5.2%-2.9 的值为：", 5.2 % -2.9) # 输出-0.6
print("5.2%-1.5 的值为：", 5.2 % -1.5) # 输出-0.8
print("-5.2%1.5 的值为：", -5.2 % 1.5) # 输出 0.8
#print("5 对 0.0 求余的结果是:", 5 % 0.0) # 导致错误
```

这里需要注意的一点是，使用 % 求余，如果除数（第二个操作数）是负数，那么取得的结果也一定是负数。因此，运行上面程序，可以看到如下输出结果：

```
5%3 的值为： 2
5.2%3.1 的值为： 2.1
-5.2%-3.1 的值为： -2.1
5.2%-2.9 的值为： -0.5999999999999996
5.2%-1.5 的值为： -0.7999999999999998
-5.2%1.5 的值为： 0.7999999999999998
```

前三个算式的运行结果比较简单，它们进行的都是很简单的求余计算。但 5.2 % -2.9 的结果有点奇怪，我们预计它为 -0.6，但实际输出的是 -0.5999999999999996。这里有两个问题：

1. 第一个问题，为什么预计 5.2 % 2.9 的结果是 -0.6 呢？因为 Python 求余运算的逻辑是用被除数减去除数的 N 倍，此处的 N 是 -2，因此得到结果是 0.6。

2. 第二个问题，为什么实际输出的是 -0.5999999999999996 呢？这是由浮点数的存储机制导致的。计算机底层的浮点数的存储机制并不是精确保存每一个浮点数的值，读者暂时不需要花太多的时间去理解浮点数的存储机制，只要知道浮点数在 Python 中可能产生精度丢失的问题就行。比如此处正常计算的结果应该是 -0.6，但实际计算出来的结果是一个非常接近 -0.6 的值。

** 乘方运算符

Python 支持使用“**”作为乘方运算符，这是一个使用非常方便的运算符。由于开方其实是乘方的逆运算，因此实际上使用“**”也可进行开方运算。例如如下代码：

```
print('5 的 2 次方：', 5 ** 2) # 25
print('4 的 3 次方：', 4 ** 3) # 64
print('4 的开平方：', 4 ** 0.5) # 2.0
print('27 的开 3 次方：',27 **(1 / 3 )) # 3.0
```

15.Python 赋值运算符

赋值运算符主要用来为变量（或常量）赋值，在使用时，既可以直接用基本赋值运算符“=”将右侧的值赋给左侧的变量，右侧也可以在进行某些运算后再赋值给左侧的变量。

= 基本赋值运算符

Python 使用“=”作为赋值运算符，常用于将表达式的值赋给另一个变量。例如如下代码：

```
# 为变量 st 赋值为 Python
st = "Python"
# 为变量 pi 赋值为 3.14
```

```
pi = 3.14
# 为变量 visited 赋值为 True
visited  = True
```

除此之外，也可使用赋值运算利将一个变量的值赋给另一个变量。例如，如下代码也是正确的：

```
# 将变量 st 的值赋给 st2
st2 = st
print(st2)
```

值得指出的是，Python 的赋值表达式是有值的，赋值表达式的值就是被赋的值，因此 Python 支持连续赋值。例如，如下代码也是正确的：

```
a = b = c = 20
```

上面程序将 c=20。这个表达式的值赋给变量 b，由于赋值表达式本身也有值，就是被赋的值，因此 c=20 这个表达式的值就是 20，故 b 也被赋值为 20；依此类推，变量 a 也被赋值为 20。

赋值运算符还可用于将表达式的值赋给变量。例如，如下代码也是正确的：

```
d1 = 12.34
# 将表达式的值赋给 d2
d2 = d1 + 5
# 输出 d2 的值
print("d2 的值为：%g" % d2 ) # 17.34
```

需要初学者特别注意的是，= 和 == 的含义完全不同，前者是赋值号，后者是等号，千万

不要混淆。

扩展后的赋值运算符

= 赋值运算符还可与其他运算符（算术运算符、位运算符等）结合，成为功能更强大的赋值运算符，如表 1 所示。

表 1 Python 常用赋值运算符

运算符	说 明	举 例	展开形式
=	最基本的赋值运算	x = y	x = y
+=	加赋值	x += y	x = x + y
-=	减赋值	x -= y	x = x - y
*=	乘赋值	x *= y	x = x * y
/=	除赋值	x /= y	x = x / y
%=	取余数赋值	x %= y	x = x % y
**=	幂赋值	x **= y	x = x ** y
//=	取整数赋值	x //= y	x = x // y
|=	按位或赋值	x |= y	x = x | y
^=	按位与赋值	x ^= y	x = x ^ y
<<=	左移赋值	x <<= y	x = x << y，这里的 y 指的是左移的位数

>>=	右移赋值	x >>= y	x = x >> y，这里的 y 指的是右移的位数

通常情况下，只要能使用扩展后的赋值运算符，都推荐使用这种赋值运算符。

16.Python 位运算符

位运算符通常在图形、图像处理和创建设备驱动等底层开发中使用。使用位运算符可以直接操作数值的原始 bit 位，尤其是在使用自定义的协议进行通信时，使用位运算符对原始数据进行编码和解码也非常有效。

> 位运算符对于初学者来说有些难度，因此初学者可先跳过本节内容。

位运算符的操作对象是整数类型，它会把数字看做对应的二进制数来进行计算。Python 支持的位运算符如表 1 所示。

表 1 Python 位运算符

位运算符	说 明	使用形式	举 例
&	按位与	a & b	4 & 5
|	按位或	a | b	4 | 5
^	按位异或	a ^ b	4 ^ 5
~	按位取反	~a	~4
<<	按位左移	a << b	4 << 2，表示数字 4 按位左移 2 位

>>	按位右移	a >> b	4 >> 2，表示数字 4 按位右移 2 位

& 按位与运算符

按位与运算的运算符是 &，它有 2 个操作数，其运算法则是，按位将 2 个操作数对应的二进制数一一对应，只有对应数位都是 1 时，此为对应的结果位才是 1；反之，就是 0。

按位与运算的运算法则如表 2 所示。

表 2 & 运算符的运算法则

第一个操作数	第二个操作数	结果位的值
0	0	0
0	1	0
1	0	0
1	1	1

例如，在 Python 交互式解释器中，计算 12 & 8 的值，执行过程如下：

```
>>> 12 & 8
8
```

计算过程如图 3 所示。

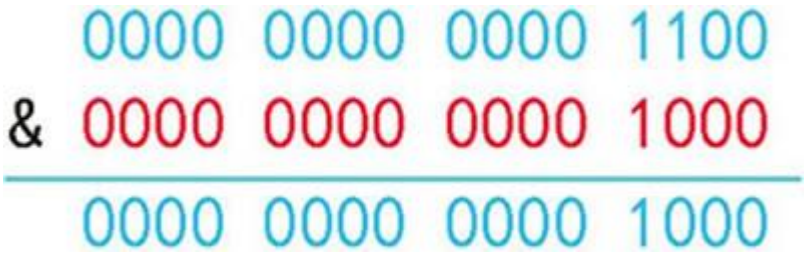


图 3 12 & 8 的计算过程

| 按位或运算符

按位或运算的运算符是 |，它有 2 个操作数，运算法则是，按位将 2 个操作数对应的二进制数一一对应，只有对应数位都是 0，所得结果才是 0；反之，就是 1。

按位或运算的运算法则如表 4 所示。

表 4 | 运算符的运算法则

第一个操作数	第二个操作数	结果位的值
0	0	0
0	1	1
1	0	1
1	1	1

例如，在 Python 交互式解释器上计算 4 | 8 的值，执行过程如下：

```
>>> 4 | 8

12
```

计算过程如图 5 所示。

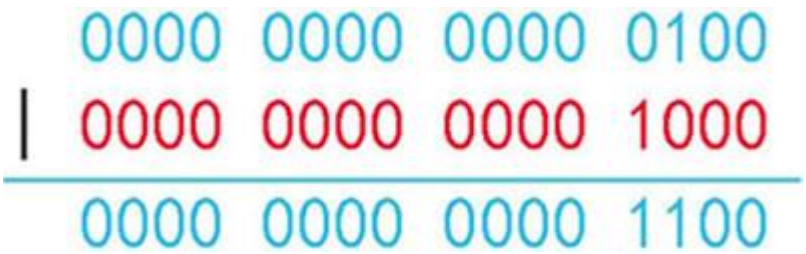


图 5 4 | 8 的计算过程

^按位异或运算符

按位异或运算的运算符是 ^，它有 2 个操作数，运算法则是，按位将 2 个操作数对应的二进制数一一对应，当对应位的二进制值相同（同为 0 或同为 1）时，所得结果为 0；反之，则为 1。

^ 运算符的运算法则如表 6 所示。

表 6 ^ 运算符的运算法则

第一个操作数	第二个操作数	结果位的值
0	0	0
0	1	1

1	0	1
1	1	0

例如，在 Python 交互式解释器中，计算 31 ^ 22 的值，其执行过程为：

```
>>> 31 ^ 22
9
```

计算过程如图 7 所示。

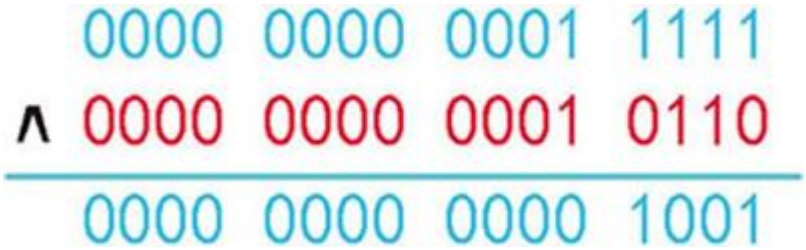


图 7 31^22 的计算过程

~按位取反运算符

按位取反，也常称为“按位非”运算，此运算的运算符为 ~，它只有 1 个操作数，其运算法则为：将操作数的所有二进制位，1 改为 0，0 改为 1。

例如，使用 Python 交互式解释器计算 -5 取反后的结果，其执行过程为：

```
>>> ~-5
4
```

注意，此运算过程涉及与计算机存储相关的内容，首先需要了解什么是原码、反码以及补码：

- 原码是直接将一个数值换算成二进制数。有符号整数的最高位是符号位，符号位为 0 代表正数，符号位为 1 代表负数。无符号整数则没有符号位，因此无符号整数只能表示 0 和正数。
- 反码的计算规则是：对原码按位取反，只是最高位（符号位）保持不变。
- 补码的计算规则是：正数的补码和原码完全相同，负数的补码是其反码 +1；

其实，所有数值在计算机底层都是以二进制形式存在的，为了方便计算，计算机底层以补码的形式保存所有的整数（如图 8 所示）：

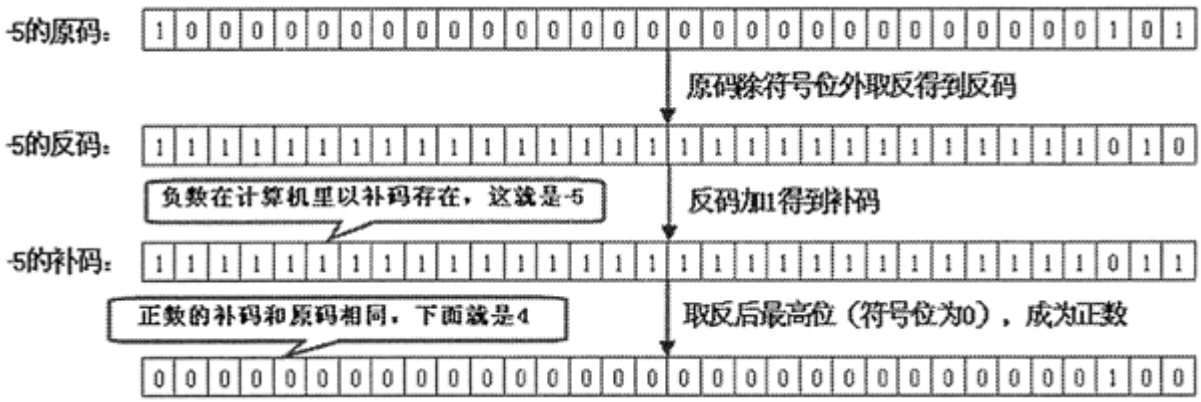


图 8 ~-5 的运算过程（点此查看高清大图）

通过图 8 可以得出，按位取反运算，实际上就是对存储在计算机底层中，以补码形式存储的整数进行按位取反所得的最终值。

注意，本节涉及到的所有按位运算符，操作的二进制数，都是操作数的补码形式。

<<左移运算符

左移运算符是将操作数补码形式的二进制数，整体左移指定位数，左移后，左边溢出的位直接丢弃，右边空出来的位以 0 来填充。例如如下代码：

```
>>> 5 << 2

20

>>>-5 << 2

-20
```

图 9 示范了 -5 左移两位的运算过程。

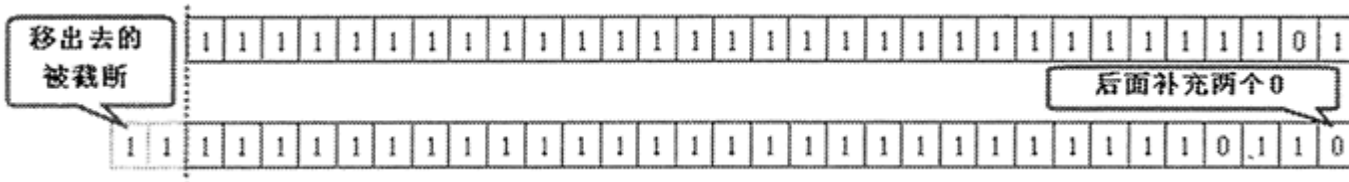


图 9 -5 左移两位的运算过程

在图 5 中，上面的 32 位数是 -5 的补码，左移两位后得到一个二进制补码，这个二进制补码的最高位是 1，表明是一个负数，换算成十进制数就是 -20。

>>右移运算符

Python 的右移运算符为 >> ,其运行法则是 ,把操作数补码形式的二进制右移指定位数后，

左边空出来的位以符号位来填充，右侧溢出位直接丢弃。

请看下面代码：

```
>>> -5 >> 2

-2
```

图 10 给出了-5 >> 2 的运算过程。

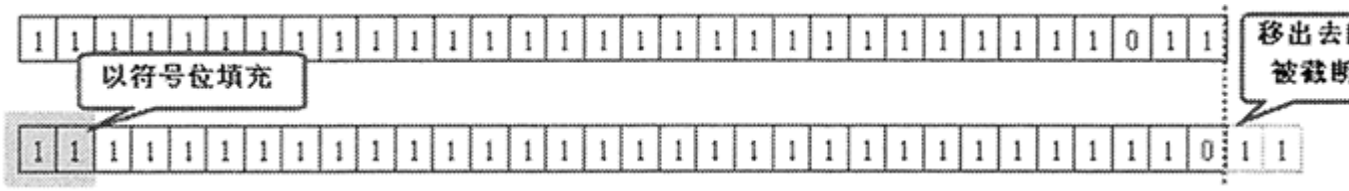


图 10 -5>>2 的运算过程

从图 10 来看，-5 右移两位后左边空出两位，空出来的两位以符号位来填充，右边溢出的两位被直接丢弃。因此，右移运算后得到的结果的正负与第一个操作数的正负相同。右移后的结果依然是一个负数，我们知道，这是一个负数的补码（负数的补码和原码不同），换算成十进制数就是 -2。

必须指出的是，无论是左移运算符还是右移运算符，它们都只适合对整型数进行运算。

在进行位移运算时，不难发现，左移 n 位就相当于来以 2 的 n 次方，右移 n 位则相当于除以 2 的 n 次方(如果不能整除 ,实际返回的结果是小于除得结果数值的最大整数的)。不仅如此 ,进行位移运算只是得到了一个新的运算结果 ,而原来的操作数本身是不会改变的。

17.Python 比较运算符

比较运算符，也成关系运算符，用于对常量、变量或表达式的结果进行大小、真假等比较，如果比较结果为真，则返回 True；反之，则返回 False。

Python 支持的比较运算符如表 1 所示。

表 1 Python 比较运算符

比较运算符	功能
>	大于，如果运算符前面的值大于后面的值，则返回 True；否则返回 False
>=	大于或等于，如果运算符前面的值大于或等于后面的值，则返回 True；否则返回 False
<	小于，如果运算符前面的值小于后面的值，则返回 True；否则返回 False
<=	小于或等于，如果运算符前面的值小于或等于后面的值，则返回 True；否则返回 False

==	等于，如果运算符前面的值等于后面的值，则返回 True；否则返回 False
!=	不等于，如果运算符前面的值不等于后面的值，则返回 True；否则返回 False
is	判断两个变量所引用的对象是否相同，如果相同则返回 True
is not	判断两个变量所引用的对象是否不相同，如果不相同则返回 True

下面程序示范了比较运算符的基本用法：

```
>>> print("5 是否大于 4：", 5 > 4)
5 是否大于 4： True
>>> print("3 的 4 次方是否大于等于 90.0：", 3 ** 4 >= 90)
3 的 4 次方是否大于等于 90.0： False
>>> print("20 是否大于等于 20.0：", 20 >= 20.0)
20 是否大于等于 20.0： True
>>> print("5 和 5.0 是否相等：", 5 == 5.0)
5 和 5.0 是否相等： True
>>> print("True 和 False 是否相等：", True == False)
True 和 False 是否相等： False
>>>
```

比较运算符 == 和 is 的区别

初学 Python，大家可能对 is 比较陌生，很多人会误将它和 == 的功能混为一谈，但其

实 is 与 == 有本质上的区别，完全不是一码事儿。

== 用来比较两个变量的值是否相等，而 is 则用来比对两个变量引用的是否是同一个对象，例如：

```
import time
# 获取当前时间
a = time.gmtime()
b =  time.gmtime()
print(a == b) # a 和 b 两个时间相等，输出 True
print(a is b) # a 和 b 不是同一个对象，输出 False
```

上面代码中 a、b 两个变量都代表当前系统时间，因此 a、b 两个变量的时间值是相等的（代码运行速度很快，能保证是同一时间），故程序使用“==”判断返回 True。但由于 a、b 两个变量分别引用不同的对象（每次调用 gmtime() 函数都返回不同的对象），因此 `a is b` 返回 False。

那么，如何判断两个变量是否属于一个对象呢？Python 提供了一个全局的 id() 函数，它可以用来判断变量所引用的对象的内存地址（相当于对象在计算机内存中存储位置的门牌号），如果两个对象所在的内存地址相同（相当于它们住在同一个房间内，计算机同一块内存在任一时刻只能存放一个对象），则说明这两个对象其实是同一个对象。

由此可见，is 判断其实就是要求通过 id() 函数计算两个对象时返回相同的地址。例如，使用 id() 函数判断上面的 a、b 两个变量：

```
print(id(a))

print(id(b))
```

从运行结果中将会看到 a、b 两个变量所引用的对象的内存地址是不同的，这样通过 is 来判断 a、b 两个变量自然也就输出 False 了。

再举一个例子：

```
>>> a = 'cheesezh'

>>> b = 'cheesezh'

>>> id(a)

2680257978480

>>> id(b)

2680257978480

>>> a is b

True
```

18.Python 逻辑运算符

逻辑运算符是对真和假两种布尔值进行运算（操作 bool 类型的变量、常量或表达式），逻辑运算的返回值也是 bool 类型值。

Python 中的逻辑运算符主要包括 and（逻辑与）、or（逻辑或）以及 not（逻辑非），

它们的具体用法和功能如表 1 所示。

表 1 Python 逻辑运算符及功能

逻辑运算符	含义	基本格式	功能
and	逻辑与（简称“与”）	a and b	有 2 个操作数 a 和 b，只有它们都是 True 时，才返回 True，否则返回 False。
or	逻辑或（简称“或”）	a or b	有 2 个操作数 a 和 b ，只有它们都是 False 时，才返回 False，否则返回 True。
not	逻辑非（简称“非”）	not a	只需要 1 个操作数 a，如果 a 的值为 True，则返回 False；反之，如果 a 的值为 False，则返回 True。

下面代码示范了与、或、非这三个逻辑运算符的使用：

```
# 直接对 False 求非运算，将返回 True
print(not False)
# 5>3 返回 True，20.0 大于 10，因此结果返回 True
print(5 > 3 and 20.0 > 10)
# 4>=5 返回 False，"c">"a"返回 True。求或后返回 True
print(4 >= 5 or "c" > "a")
```

有些时候，程序需要使用多个逻辑运算符来组合复杂的逻辑。例如，假设想表达如下逻辑：

需要购买一套必须以“Python”结尾的教程，且教程价格小于 50 元或该教程是基于“正式版”的。假如使用如下程序：

```
bookName = "C 语言中文网 Python"
price = 99
version = "正式版"
if bookName.endswith('Python') and price < 50 or version == "正式版" :
    print("打算购买这套 Python 教程")
else:
    print("不购买！")
```

编译、运行上面程序，可以看到程序输出“打算购买这套 Python 教程”。那么上面程序是否有问题呢？

其实问题是存在的，这是因为程序会先计算 bookName.endswith('Python') and price < 50，即使该逻辑表达式中的两个条件都是 False，但只要后面的 version==“正式版” 返回 True，整个表达式就会返回 True，从而导致程序依然会输出“打算购买这套 Python 教程”。因此，即使把上面程序中的 bookName 改为不以“Python”结尾，程序也依然会输出“打算购买这套 Python 教程”。

运算结果显然与逻辑需求并不一致，逻辑需求是：需要购买一套名以“Python” 结尾的教程。此时应该使用圆括号来保证程序先对 price < 50 || version =="正式版" 求值，然后再与 bookName.endswith('Python') 的结果求与。

因此，应该把程序改为如下形式：

```
bookName = "C 语言中文网 Python"
```

```
        price = 99
        version = "正式版"
        if bookName.endswith('Python') and (price < 50 or version == "正式版") :
            print("打算购买这套 Python 教程")
        else:
            print("不购买！")
```

从上面代码可以看出，对于组合逻辑来说，使用圆括号保证运算顺序非常重要。

需要说明的是，即使不是为了保证逻辑运算的顺序，且有括号和没括号的输出结果是一样的，本教程也依然建议使用圆括号来提高程序的可读性。

19.Python 三目运算符

Python 可通过 if 语句来实现三目运算符的功能，因此可以近似地把这种 if 语句当成三目运算符。作为三目运算符的 if 语句的语法格式如下：

```
True_statements if expression else False_statements
```

三目运算符的规则是：先对逻辑表达式 expression 求值，如果逻辑表达式返回 True，则执行并返回 True_statements 的值；如果逻辑表达式返回 False，则执行并返回 False_statements 的值。看如下代码：

```
        a = 5
        b = 3
        st = "a 大于 b" if a > b else  "a 不大于 b"
        # 输出"a 大于 b"
        print(st)
```

实际上，如果只是为了在控制台输出提示信息，还可以将上面的三目运算符表达式改为如下形式：

```
# 输出"a 大于 b"
print("a 大于 b") if a > b else print("a 不大于 b")
```

Python 允许在三目运算符的 True_statements 或 False_statements 中放置多条语句。Python 主要支持以下两种放置方式：

1. 多条语句以英文逗号隔开：每条语句都会执行，程序返回多条语句的返回值组成的元组。
2. 多条语句以英文分号隔开：每条语句都会执行，程序只返回第一条语句的返回值。

先看第一种情形，使用如下代码：

```
# 第一个返回值部分使用两条语句，逗号隔开
st = print("crazyit"), 'a 大于 b' if a > b else  "a 不大于 b"
print(st)
```

上面程序中 True_statements 为 print("crazyit")，'a 大于 b'，这两条语句都会执行，程序将会返回这两条语句的返回值组成的元组。由于 print() 函数没有返回值，相当于它的返回值是 None。运行上面代码，将看到如下结果：

```
crazyit
(None,'a 大于 b')
```

如果将上面语句中的逗号改为分号，将逗号之后的语句改为赋值语句，即写成如下形式：

```
# 第一个返回值部分使用两条语句，分号隔开
st = print("crazyit"); x = 20 if a > b else  "a 不大于 b"
print(st)
print(x)
```

此时虽然 True_statements 包含两条语句，但程序只会返回第一条语句 print(" crazyit") 的返回值，该语句同样返回 None，因此相当于 str 的返回值为 None。运行上面代码，将看到如下结果：

```
crazyit

None

20
```

需要指出的是，三目运算符支持嵌套，通过嵌套三目运算符，可以执行更复杂的判断。例如，下面代码需要判断 c、d 两个变量的大小关系：

```
c = 5
d = 5
# 下面将输出 c 等于 d
print("c 大于 d") if c > d else (print("c 小于 d") if c < d else print("c 等于 d"))
```

上面代码首先对 c>d 求值，如果该表达式为 True，程序将会执行并返回第一个表达式：print(" c 大于 d")；否则系统将会计算 else 后面的内容：(print("c 小于 d") if c < d else print("c 等于 d"))，这个表达式又是一个嵌套的三目运算符表达式。注意，进入该表达式时只剩下“c 小于 d”或“c 等于 d”两种情况，因此该三目运算符再次判断 c<d，如果该表达式为 True，将会输出“c 小于 d” ；否则只剩下“c 等于 d”一种情况，自然就输出该字符串了。

20.Python 运算符优先级

所谓运算符的优先级，指的是在含有多个逻辑运算符的式子中，到底应该先计算哪一个，后计算哪一个，这与数学中四则运算应遵循“先乘除后加减”是一个道理。

Python 中运算符的运算规则是，优先级高的运算符先执行，优先级低的运算符后执行，同一优先级的运算符按照从左到右的顺序进行。需要注意的是，Python 语言中大部分运算符都是从左向右执行的，只有单目运算符（例如 not 逻辑非运算符）、赋值运算符和三目运算符例外，它们是从右向左执行的。

表 1 按照优先级从高到底的顺序，罗列出了包括分隔符在内的所有运算符的优先级顺序。

表 1 运算符的优先级

运算符说明	Python 运算符	优先级
索引运算符	x[index] 或 x[index:index2[:index3]]	18、19
属性访问	x.attrbute	17
乘方	**	16
按位取反	~	15
符号运算符	+（正号）或 -（负号）	14
乘、除	*、/、//、%	13

加、减	+、-	12
位移	>>、<<	11
按位与	&	10
按位异或	^	9
按位或	|	8
比较运算符	==、!=、>、>=、<、<=	7
is 运算符	is、is not	6
in 运算符	in、not in	5
逻辑非	not	4
逻辑与	and	3
逻辑或	or	2

根据表 1 中运算符的优先级，我们分析 4+4<<2 语句的执行结果。程序先执行 4+4 得到结果 8，再执行 8<<2 得到 32。如果使用“()”就可以改变程序的执行顺序，比如 4+(4<<2)，则先执行 4<<2 得到结果 16，再执行 4+16 得到 20。

虽然 Python 运算符存在优先级的关系，但并不推荐过度依赖运算符的优先级，因为这会导致程序的可读性降低。因此，在这里要提醒读者：

- 不要把一个表达式写得过于复杂，如果一个表达式过于复杂，则把它分成几步来完成。
- 不要过多地依赖运算符的优先级来控制表达式的执行顺序，这样可读性太差，应尽量使用“()”来控制表达式的执行顺序。

第 4 章 列表、元组、字典和集合

本章将会介绍 Python 内置的四种常用数据结构：列表（list）、元组（tuple）、字典（dict）以及集合（set）。

这四种数据结构一但都可用于保存多个数据项，这对于编程而言是非常重要的，因为程序不仅需要使用单个变量来保存数据，还需要使用多种数据结构来保存大量数据，而列表、元组、字典和集合就可满足保存大量数据的需求。

列表（list）和元组（tuple）比较相似，它们都按顺序保存元素，每个元素都有自己的索引，因此列表和元组都可通过索引访问元素。二者的区别在于元组是不可修改的，但列表是可修改的。

字典（dict）和集合（set）类似，它们存储的数据都是无序的，其中字典是用 key-value 的形式保存数据。

1.什么是序列，Python 序列详解

所谓序列，指的是一块可存放多个值的连续内存空间，这些值按一定顺序排列，可通过每个值所在位置的编号（称为索引）访问它们。

为了更形象的认识序列，可以将它看做是一家旅店，那么店中的每个房间就如同序列存储数据的一个个内存空间，每个房间所特有的房间号就相当于索引值。也就是说，通过房间号（索引）我们可以找到这家旅店（序列）中的每个房间（内存空间）。

在 Python 中，序列类型包括字符串、列表、元组、集合和字典，这些序列支持以下几种通用的操作，但比较特殊的是，集合和字典不支持索引、切片、相加和相乘操作。

> 字符串也是一种常见的序列，它也可以直接通过索引访问字符串内的字符。

序列索引

序列中，每个元素都有属于自己的编号（索引）。从起始元素开始，索引值从 0 开始递增，如图 1 所示。

图 1 序列索引值示意图

除此之外，Python 还支持索引值是负数，此类索引是从右向左计数，换句话说，从最后一个元素开始计数，从索引值 -1 开始，如图 2 所示。

图 2 负值索引示意图

注意，在使用负值作为列序中各元素的索引值时，是从 -1 开始，而不是从 0 开始。

无论是采用正索引值，还是负索引值，都可以访问序列中的任何元素。以字符串为例，访问“C 语言中文网”的首元素和尾元素，可以使用如下的代码：

```
str="C 语言中文网"
print(str[0],"==",str[-6])
print(str[5],"==",str[-1])
```

输出结果为：

```
C == C
网 == 网
```

序列切片

切片操作是访问序列中元素的另一种方法，它可以访问一定范围内的元素，通过切片操作，可以生成一个新的序列。

序列实现切片操作的语法格式如下：

```
sname[start : end : step]
```

其中，各个参数的含义分别是：

- sname：表示序列的名称；
- start :表示切片的开始索引位置(包括该位置),此参数也可以不指定 ,会默认为 0，也就是从序列的开头进行切片；

- end：表示切片的结束索引位置（不包括该位置），如果不指定，则默认为序列的长度；
- step：表示在切片过程中，隔几个存储位置（包含当前位置）取一次元素，也就是说，如果 step 的值大于 1，则在进行切片去序列元素时，会“跳跃式”的取元素。如果省略设置 step 的值，则最后一个冒号就可以省略。

例如，对字符串“C 语言中文网”进行切片：

```
str="C 语言中文网"
#取索引区间为[0,2]之间（不包括索引 2 处的字符）的字符串
print(str[:2])
#隔 1 个字符取一个字符，区间是整个字符串
print(str[::2])
#取整个字符串，此时 [] 中只需一个冒号即可
print(str[:])
```

运行结果为：

```
C 语
C 言文
C 语言中文网
```

序列相加

Python 中，支持两种类型相同的序列使用“+”运算符做相加操作，它会将两个序列进行连接，但不会去除重复的元素。

这里所说的“类型相同”，指的是“+”运算符的两侧序列要么都是序列类型，要么都是元组类型，要么都是字符串。

例如，前面章节中我们已经实现用“+”运算符连接 2 个（甚至多个）字符串，如下所示：

```
str="c.biancheng.net"
print("C 语言"+"中文网:"+str)
```

输出结果为：

```
C 语言中文网：c.biancheng.net
```

序列相乘

Python 中，使用数字 n 乘以一个序列会生成新的序列，其内容为原来序列被重复 n 次的结果。例如：

```
str="C 语言中文网"
print(str*3)
```

输出结果为：

```
'C 语言中文网 C 语言中文网 C 语言中文网'
```

比较特殊的是，列表类型在进行乘法运算时，还可以实现初始化指定长度列表的功能。例如如下的代码，将创建一个长度为 5 的列表，列表中的每个元素都是 None，表示什么都没有。

```
#列表的创建用 []，后续讲解列表时会详细介绍
list = [None]*5
print(list)
```

输出结果为：

```
[None, None, None, None, None]
```

检查元素是否包含在序列中

Python 中，可以使用 in 关键字检查某元素是否为序列的成员，其语法格式为：

```
value in sequence
```

其中，value 表示要检查的元素，sequence 表示指定的序列。

例如，检查字符 'c' 是否包含在字符串 "c.biancheng.net" 中，可以执行如下代码：

```
str="c.biancheng.net"
print('c' in str)
```

运行结果为：

```
True
```

和 in 关键字用法相同，但功能恰好相反的，还有 not in 关键字，它用法检查某个元素是否不包含在指定的序列中，比如说：

```
str="c.biancheng.net"
```

```
2.          print('c' not in str)
```

输出结果为：

```
False
```

和序列相关的内置函数

Python 提供了几个内置函数（表 3 所示），可用于实现与序列相关的一些常用操作。

表 3 序列相关的内置函数

函数	功能
len()	计算序列的长度，即返回序列中包含多少个元素。
max()	找出序列中的最大元素。注意，对序列使用 sum() 函数时，做加和操作的必须都是数字，不能是字符或字符串，否则该函数将抛出异常，因为解释器无法判定是要做连接操作（+ 运算符可以连接两个序列），还是做加和操作。
min()	找出序列中的最小元素。
list()	将序列转换为列表。
str()	将序列转换为字符串。
sum()	计算元素和。
sorted()	对元素进行排序。
reversed()	反向序列中的元素。

enumerate()	将序列组合为一个索引序列，多用在 for 循环中。

这里给大家给几个例子：

纯文本复制

```
str="c.biancheng.net"
#找出最大的字符
print(max(str))
#找出最小的字符
print(min(str))
#对字符串中的元素进行排序
print(sorted(str))
```

输出结果为：

```
t

.

['.', '.', 'a', 'b', 'c', 'c', 'e', 'e', 'g', 'h', 'i', 'n', 'n', 'n', 't']
```

2.Python 列表（list）

在实际开发中，经常需要将一些（不只一个）数据暂储起来，以便将来使用。说到这里，一些读者可能知道或听说过数组，它就可以把多个数据挨个存储到一起，通过数组下标可以访问数组中的各个元素。但使用数组存储数据有一个弊端，就是所存储的数据必须类型一致。

需要明确的是，Python 中没有数组，但是加入了更加强大的列表。如果把数组看做是一个集装箱，那么 Python 的列表就是一个工厂的仓库。

从形式上看，列表会将所有元素都放在一对中括号 [] 中，相邻元素之间用逗号分隔，如下所示：

```
[element1,element2,element3,...,elementn]
```

格式中，element1~elementn 表示列表中的元素，个数没有限制，只要是 Python 支持的数据类型就可以。

从内容上看，列表可以存储整数、实数、字符串、列表、元组等任何类型的数据，并且和数组不同的是，在同一个列表中元素的类型也可以不同。比如说：

```
["c.biancheng.net" , 1 , [2,3,4] , 3.0]
```

可以看到，列表中同时包含字符串、整数、列表、浮点数这些数据类型。

注意，在使用列表时，虽然可以将不同类型的数据放入到同一个列表中，但通常情况下不这么做，同一列表中只放入同一类型的数据，这样可以提高程序的可读性。

另外，在其它 Python 教程中，经常用 list 代指列表，这是因为列表的数据类型就是 list，通过 type() 函数就可以知道，例如：

```
>>> type(["c.biancheng.net" , 1 , [2,3,4] , 3.0])
<class 'list'>
```

可以看到，它的数据类型为 list，就表示它是一个列表。

Python 创建列表

在 Python 中，创建列表的方法可分为 2 种，下面分别进行介绍。

使用 = 运算符直接创建列表

和其他类型的 Python 变量一样，创建列表时，也可以使用赋值运算符“=”直接将一个列表赋值给变量，其语法格式如下：

```
listname = [element1 , element2 , element3 , ... , elementn]
```

其中，listname 表示列表的名称，注意，在命名时既要符合 Python 命名规范，也要尽量避开与 Python 的内置函数重名。

例如，下面定义的列表都是合法的：

```
num = [1,2,3,4,5,6,7]
name = ["C 语言中文网","http://c.biancheng.net"]
program = ["C 语言","Python","Java"]
```

另外，使用此方式创建列表时，列表中元素可以有多个，也可以一个都没有，例如：

```
emptylist = []
```

这表明，emptylist 是一个空列表。

使用 list()函数创建列表

除使用前面介绍的方括号语法创建列表之外，Python 还提供了一个内置的 list() 函数来创建列表，它可用于将元组、区间（range）等对象转换为列表，例如：

```
a_tuple = ('crazyit', 20, -1.2)
# 将元组转换成列表
a_list = list(a_tuple)
print(a_list)
```

输出结果为：

```
['crazyit', 20, -1.2]
```

此程序的功能是将一个元组（'crazyit', 20, -1.2）转换成列表。由于尚未学习元组，因此初学者只需要知道，用圆括号 () 括起来的多个数据，就是一个元组。

再举个例子：

```
# 使用 range()函数创建区间（range）对象
a_range = range(1, 5)
# 将区间转换成列表
b_list = list(a_range)
print(b_list)
```

输出结果为：

```
[1, 2, 3, 4]
```

程序中，range(1,5) 函数的功能是生成从 1（包括 1）到 5（不包括 5）之间的整数，也

就是 1、2、3、4。

有关 range() 函数的具体用法，后续章节会作详细介绍。

访问列表元素

在 Python 中，如果想将列表的内容输出也比较简单，直接使用 print() 函数即可。例如，前面已经创建了一个名为 name 的列表，输出此列表的执行代码如下：

```
1.          name = ["C 语言中文网","http://c.biancheng.net"]
2.          print(name)
```

运行结果为：

```
['C 语言中文网', 'http://c.biancheng.net']
```

可以看到，输出整个列表时，是包括左右两侧的中括号的。

如果不想要输出全部的元素，可以通过列表的索引获取指定的元素。例如，获取 name 列表中索引值为 1 的元素，可以使用如下代码：

```
1.          name = ["C 语言中文网","http://c.biancheng.net"]
2.          print(name[1])
```

运行结果为：

```
http://c.biancheng.net
```

从执行结果可以看出，在输出单个列表元素时，是不带中括号的，且如果是字符串，还不包

括左右的引号。

除了一次性访问列表中的单个元素外，列表还可以通过切片操作实现一次性访问多个元素。

> 有关切片操作的具体格式，可阅读《Python 序列详解》一节，这里不再赘述。

例如：

```
num = [1, 2, 3, 4, 5, 6, 7]
print(num[2:4])
```

输出结果为：

```
[3, 4]
```

可以看到，通过切片操作，最终得到的是一个新的列表。

Python 删除列表

对于已经创建的列表，如果不再使用，可以使用 del 语句将其删除。

> 实际开始时，del 语句不常用，因为 Python 自带的垃圾回收机制会自动销毁不用的列表，所以即使开发者不手动将其删除，Python 也会自动将其回收。

del 的语法格式为：

```
del listname
```

其中，listname 表示要删除列表的名称。

例如，删除前面创建的 name 列表，可以使用下面的代码：

```
1.          name = name = ["C 语言中文网","http://c.biancheng.net"]
2.          print(name)
3.          del name
4.          print(name)
```

运行结果为：

```
['C 语言中文网', 'http://c.biancheng.net']
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 4, in <module>
    print(name)
NameError: name 'name' is not defined
```

3.Python list 列表添加元素

实际开发中，经常需要对列表进行更新，包括向列表中添加元素、修改表中元素以及删除元素。本节先来学习如何向列表中添加元素。

通过前面的学习我们知道，通过使用“+”运算符可以将多个序列进行连接，列表也不例外，例如：

```
name = ["C 语言中文网"]
address = ["http://c.biancheng.net"]
print(name+address)
```

运行结果为：

```
['C 语言中文网', 'http://c.biancheng.net']
```

可以看到，使用“+”运算符，确实可以像列表中添加元素。但是这种方式的执行效率并不高，更建议大家使用列表提供的 append() 方法。

Python append()方法添加元素

append() 方法用于在列表的末尾追加元素，该方法的语法格式如下：

```
listname.append(obj)
```

其中，listname 指的是要添加元素的列表；obj 表示到添加到列表末尾的数据，它可以是单个元素，也可以是列表、元组等。

例如：

```
a_list = ['crazyit', 20, -2]
# 追加元素
a_list.append('fkit')
print(a_list)
a_tuple = (3.4, 5.6)
# 追加元组，元组被当成一个元素
a_list.append(a_tuple)
print(a_list)
# 追加列表，列表被当成一个元素
```

```
10.         a_list.append(['a', 'b'])
11.         print(a_list)
```

运行结果为：

```
['crazyit', 20, -2, 'fkit']

['crazyit', 20, -2, 'fkit', (3.4, 5.6)]

['crazyit', 20, -2, 'fkit', (3.4, 5.6), ['a', 'b']]
```

可以看到，即便给 append() 方法传递列表或者元组，此方法也只会将其视为一个元素，直接添加到列表中，从而形成包含列表和元组的新列表。

Python extend()方法添加元素

当然，如果希望不将被追加的列表或元组当成一个整体，而是只追加列表中的元素，则可使用列表提供的 extend() 方法。

extend()方法的语法格式如下：

```
listname.extend(obj)
```

例如：

```
1.          b_list = ['a', 30]
2.          # 追加元组中的所有元素
3.          b_list.extend((-2, 3.1))
4.          print(b_list)
5.          # 追加列表中的所有元素
6.          b_list.extend(['C', 'R', 'A'])
```

```
print(b_list)
# 追加区间中的所有元素
b_list.extend(range(97, 100))
print(b_list)
```

运行结果为：

```
['a', 30, -2, 3.1]
['a', 30, -2, 3.1, 'C', 'R', 'A']
['a', 30, -2, 3.1, 'C', 'R', 'A', 97, 98, 99]
```

Python insert()方法插入元素

如果希望在列表中间增加元素，则可使用列表的 insert() 方法，此方法的语法格式为：

```
listname.insert(index , obj)
```

其中，index 参数指的是将元素插入到列表中指定位置处的索引值。

使用 insert() 方法向列表中插入元素，和 append() 方法一样，无论插入的对象是列表还是元组，都只会将其整体视为一个元素。

例如：

纯文本复制

```
c_list = list(range(1, 6))
print(c_list)
# 在索引 3 处插入字符串
c_list.insert(3, 'CRAZY' )
```

```
5.          print(c_list)
6.          # 在索引 3 处插入列表
7.          c_list.insert(3, ["crazy"])
8.          print(c_list)
```

输出结果为：

```
[1, 2, 3, 4, 5]

[1, 2, 3, 'CRAZY', 4, 5]

[1, 2, 3, ['crazy'], 'CRAZY', 4, 5]
```

注意，insert() 主要用来插入元素，当向列表末尾添加元素时，还是应该使用 append() 方法。

4.Python list 列表删除元素

在列表中删除元素，主要分为以下 3 种应用场景：

1. 根据目标元素所在位置的索引值进行删除，可使用 del 语句；
2. 根据元素的值进行删除，可使用列表（list 类型）提供的 remove() 方法；
3. 将列表中所有元素全部删除，可使用列表（list 类型）提供的 clear() 方法。

根据索引值删除元素

删除列表中指定元素，和删除列表类似，也可以使用 del 语句实现。

del 语句是 Python 中专门用于执行删除操作的语句，不仅可用于删除列表的元素，也可用于删除变量等。

例如，定义一个保存 3 个元素的列表，若指定删除最后一个元素，可以使用如下的代码：

```
a_list=[20,2.4,(3,4)]
del a_list[-1]
print(a_list)
```

输出结果为:

```
[20, 2.4]
```

不仅如此，del 语句还可以直接删除列表的中间一段。例如如下代码：

```
a_list = ['crazyit', 20, -2.4, (3, 4), 'fkit']
# 删除第 2 个到第 4 个（不包含）元素
del a_list[1: 3]
print(a_list)
```

输出结果为：

```
['crazyit', (3, 4), 'fkit']
```

根据元素值进行删除

除使用 del 语句之外，Python 还提供了 remove() 方法来删除列表元素，该方法并不是根据索引来删除元素的，而是根据元素本身的值来执行删除操作的。

remove() 方法会删除第一个和指定值相同的元素，如果找不到该元素，该方法将会引发 ValueError 错误。

如下代码示范了使用 remove() 方法删除元素：

```
c_list = [20, 'crazyit', 30, -4, 'crazyit', 3.4]
# 删除第一次找到的 30
c_list.remove(30)
print(c_list)
# 删除第一次找到的'crazyit'
c_list.remove('crazyit')
print(c_list)
#再次尝试删除 30，会引发 ValueEroor 错误
c_list.remove(30)
```

输出结果为:

```
[20, 'crazyit', -4, 'crazyit', 3.4]
[20, -4, 'crazyit', 3.4]
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 9, in <module>
    c_list.remove(30)
ValueError: list.remove(x): x not in list
```

在使用 remove() 方法删除列表元素之前，最好提前判断一下指定的元素是否存在，所以此方法常与 count() 方法组合使用。

删除列表所有元素

list 列表还包含一个 clear() 方法，正如它的名字所暗示的，该方法用于清空列表的所有元素。例如如下代码：

```
c_list = [20, 'crazyit', 30, -4, 'crazyit', 3.4]
c_list.clear()
print(c_list)
```

输出结果为:

```
[]
```

5.Python list 列表修改元素

列表的元素相当于变量，因此程序可以对列表的元素赋值，这样即可修改列表的元素。例如如下代码：

```
a_list = [2, 4, -3.4, 'crazyit', 23]
# 对第 3 个元素赋值
a_list[2] = 'fkit'
print(a_list) # [2, 4, 'fkit', 'crazyit', 23]
# 对倒数第 2 个元素赋值
a_list[-2] = 9527
print(a_list) # [2, 4, 'fkit', 9527, 23]
```

上面代码通过索引到列表元素赋值，程序既可使用正数索引，也可使用负数索引，这都没有问题。

此外，程序也可通过 slice 语法对列表其中一部分赋值。在执行这个操作时，并不要求新赋值的元素个数与原来的元素个数相等。这意味着通过这种方式既可为列表增加元素，也可为列表删除元素。

例如，如下代码示范了对列表中间一段赋值：

```
b_list = list(range(1, 5))
print(b_list)
# 将第 2 个到第 4 个（不包含）元素赋值为新列表的元素
b_list[1: 3] = ['a', 'b']
print(b_list) # [1, 'a', 'b', 4]
```

如果对列表中空的 slice 赋值，就变成了为列表插入元素。例如如下代码：

```
# 将第 3 个到第 3 个（不包含）元素赋值为新列表的元素，就是插入
b_list[2: 2] = ['x', 'y']
print(b_list) # [1, 'a', 'x', 'y', 'b', 4]
```

如果将列表其中一段赋值为空列表，就变成了从列表中删除元素。例如如下代码：

```
# 将第 3 个到第 6 个（不包含）元素赋值为空列表，就是删除
b_list[2: 5] = []
print(b_list) # [1, 'a', 4]
```

对列表使用 slice 语法赋值时，不能使用单个值；如果使用字符串赋值，Python 会自动把字符串当成序列处理，其中每个字符都是一个元素。例如如下代码：

```
# Python 会自动将 str 分解成序列
b_list[1: 3] = 'Charlie'
print(b_list) # [1, 'C', 'h', 'a', 'r', 'l', 'i', 'e']
```

在使用 slice 语法赋值时，也可指定 step 参数。但如果指定了 step 参数，则要求所赋值的列表元素个数与所替换的列表元素个数相等。例如如下代码：

```
c_list = list(range(1, 10))
# 指定 step 为 2，被赋值的元素有 4 个，因此用于赋值的列表也必须有 4 个元素
```

```
3.          c_list[2: 9: 2] = ['a', 'b', 'c', 'd']
4.          print(c_list) # [1, 2, 'a', 4, 'b', 6, 'c', 8, 'd']
```

6.Python list 常用方法

除前面章节介绍的增加元素、删除元素、修改元素方法之外，列表还包含了一些常用的方法。

例如，在交互式解释器中输入 dir(list) 即可看到列表包含的所有方法，如下所示：

```
>>> dir(list)
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
>>>
```

在上面输出结果中已经剔除了那些以双下画线开头的方法。按照约定，这些方法都具有特殊的意义，不希望被用户直接调用。

上面有些方法前面已经介绍过了，接下来给大家介绍另外的一些常用方法。

count()方法

此方法用于统计列表中某个元素出现的次数，其基本语法格式为：

```
listname.count(obj)
```

其中，listname 代表列表名，obj 表示判断是否存在的元素。

下面代码示范了 count() 方法的用法：

```
a_list = [2, 30, 'a', [5, 30], 30]
# 计算列表中 30 的出现次数
print(a_list.count(30))
# 计算列表中[5, 30]的出现次数
print(a_list.count([5, 30]))
```

运行结果为：

```
2

1
```

index()用法

index() 方法用于定位某个元素在列表中出现的位置（也就是索引），如果该元素没有出现，则会引发 ValueError 错误。

此方法的基本语法格式为：

```
listname.index(obj,start,end)
```

同 count() 方法不同，index() 方法还可传入 start、end 参数，用于在列表的指定范围内搜索元素。

如下代码示范了 index() 方法的用法：

```
a_list = [2, 30, 'a', 'b', 'crazyit', 30]
# 定位元素 30 的出现位置
```

```
print(a_list.index(30))
# 从索引 2 处开始、定位元素 30 的出现位置
print(a_list.index(30, 2))
# 从索引 2 处到索引 4 处之间定位元素 30 的出现位置，因为找不到该元素，会引发 ValueError 错误
print(a_list.index(30, 2, 4))
```

运行结果为：

```
1

5

Traceback (most recent call last):

  File "C:\Users\mengma\Desktop\1.py", line 7, in <module>

    print(a_list.index(30, 2, 4)) # ValueError

ValueError: 30 is not in list
```

pop()用法

pop() 方法会移除列表中指定索引处的元素，如果不指定，默认会移除列表中最后一个元素。该方法的基本语法格式为：

```
listname.pop(index)
```

例如：

```
a_list=[1,2,3]
#移除列表的元素 3
print(a_list.pop())
print(a_list)
#移除列表中索引为 0 的元素 1
```

```
print(a_list.pop(0))
print(a_list)
```

运行结果为:

```
3
[1, 2]
1
[2]
```

注意，使用 pop() 方法可以实现一种常见的数据结构——栈。栈是一种特殊的数据结构，它可实现先入后出（FILO）功能，即先加入栈的元素，反而后出栈。

在其他编程语言所实现的“栈”中，往往会提供一个 push() 方法，用于实现入栈操作，但 Python 的列表并没有提供 push() 方法，我们可以使用 append() 方法来代替 push() 方法实现入栈操作。

下面代码示范了使用列表作为“栈”的示例：

```
stack = []
# 向栈中“入栈”3 个元素
stack.append("fkit")
stack.append("crazyit")
stack.append("Charlie")
print(stack) # ['fkit', 'crazyit', 'Charlie']
# 第一次出栈：最后入栈的元素被移出栈
print(stack.pop())
```

```
9.          print(stack) # ['fkit', 'crazyit']
10.         # 再次出栈
11.         print(stack.pop())
12.         print(stack) # ['fkit']
```

运行结果为：

```
['fkit', 'crazyit', 'Charlie']
Charlie
['fkit', 'crazyit']
crazyit
['fkit']
```

上面程序中，第 8 行实现了第一次出栈操作，该操作将会把最后一次添加的元素移出栈，且该方法会返回出栈的元素。因此，执行这行代码将会看到输出 Charlie。

> 与所有编程语言类似的是，出栈操作既会移出列表的最后一个元素，也会返回被移出的元素。

reverse()用法

reverse() 方法会将列表中所有元素反向存放。该方法的基本语法格式为：

```
listname.reverse()
```

例如：

```
1.          a_list = list(range(1, 8))
2.          # 将 a_list 列表元素反转
3.          a_list.reverse()
```

```
4.          print(a_list)
```

运行结果为：

```
[7, 6, 5, 4, 3, 2, 1]
```

从上面的运行结果可以看出，调用 reverse() 方法将反转列表中的所有元素。

sort()用法

sort() 方法用于对列表元素进行排序，排序后原列表中的元素顺序会方发生改变。sort() 方法的语法格式如下：

```
listname.sort(key=None, reserse=False)
```

可以看到，和其他方法不同，此方法中多了 2 个参数，它们的作用分别是：

- key 参数用于指定从每个元素中提取一个用于比较的键。例如，使用此方法时设置 key=str.lower 表示在排序时不区分字母大小写。
- reverse 参数用于设置是否需要反转排序，默认 False 表示从小到大排序；如果将该参数设为 True，将会改为从大到小排序。

例如如下代码：

```
1.          a_list = [3, 4, -2, -30, 14, 9.3, 3.4]
2.          # 对列表元素排序
3.          a_list.sort()
4.          print(a_list)
```

```
        b_list = ['Python', 'Swift', 'Ruby', 'Go', 'Kotlin', 'Erlang']
        # 对列表元素排序：默认按字符串包含的字符的编码大小比较
        b_list.sort()
        print(b_list) # ['Erlang', 'Go', 'Kotlin', 'Python', 'Ruby', 'Swift']
```

运行结果为：

```
[-30, -2, 3, 3.4, 4, 9.3, 14]
['Erlang', 'Go', 'Kotlin', 'Python', 'Ruby', 'Swift']
```

如下代码示范了 key 和 reverse 参数的用法：

```
        b_list = ['Python', 'Swift', 'Ruby', 'Go', 'Kotlin', 'Erlang']
        # 指定 key 为 len，指定使用 len 函数对集合元素生成比较的键，也就是按字符串的长度比较大小
        b_list.sort(key=len)
        print(b_list)
        # 指定反向排序
        b_list.sort(key=len, reverse=True)
        print(b_list)
```

运行结果为：

```
['Go', 'Ruby', 'Swift', 'Python', 'Kotlin', 'Erlang']
['Python', 'Kotlin', 'Erlang', 'Swift', 'Ruby', 'Go']
```

上面两次排序时都将 key 参数指定为 len，这意味着程序将会使用 len() 函数对集合元素生成比较大小的键，即根据集合元素的字符串长度比较大小。

注意，采用 sort() 方法对列表进行排序时，对中文支持不好，其排序结果与常用的音序排序法或者笔画排序法都不一致，因此，如果需要实现对中文内容的列表排序，还需要重新编写相应的方法进行处理，而不能直接使用 sort() 方法。

除此之外，需要指出的是，在 Python 2.x 中，列表的 sort() 方法还可传入一个比较大小的函数，该函数负责比较列表元素的大小。该函数包含两个参数，当该函数返回正整数时，代表该函数的第一个参数大于第二个参数；当该函数返回负整数时，代表该函数的第一个参数小于第二个参数；返回 0，则意味着两个参数相等。

下面代码示范了使用比较函数调用 sort() 方法。以下代码只能在 Python 2.x 中执行：

```
# 以下代码只能在 Python 2.x 中执行
# 定义一个根据长度比较大小的比较函数
def len_cmp(x, y):
    # 下面代码比较大小的逻辑是：长度大的字符串就算更大
    return 1 if len(x) > len(y) else (-1 if len(x) < len(y) else 0)
b_list = ['Python', 'Swift', 'Ruby', 'Go', 'Kotlin', 'Erlang']
b_list.sort(len_cmp)
print(b_list)
```

运行结果为：

```
['Go', 'Ruby', 'Swift', 'Erlang', 'Kotlin', 'Python']
```

7.Python range()快速初始化数字列表

注意：本节需具备最基本的 Python 循环结构的基础，初学者可先跳过本节。

实际场景中，经常需要存储一组数字。例如在游戏中，需要跟踪每个角色的位置，还可能需要跟踪玩家的几个最高得分。在数据可视化中，处理的几乎都是由数字（如温度、距离、人口数量、经度和纬度等）组成的集合。

列表非常适合用于存储数字集合，并且 Python 提供了 range() 函数，可帮助我们高效地处理数字列表，即便列表需要包含数百万个元素，也可以快速实现。

Python range() 函数能够轻松地生成一系列的数字。例如，可以像如下这样使用 range() 来打印一系列数字：

```
1.          for value in range(1,5):
2.              print(value)
```

输出结果为：

```
1

2

3

4
```

注意，在这个示例程序中，range() 只是打印数字 1~4，因为 range() 函数的用法是：让 Python 从指定的第一个值开始，一直数到指定的第二个值停止，但不包含第二个值（这里为 5）。

因此，如果想要上面程序打印数字 1~5，需要使用 range(1,6)。

另外需要指明的是，range() 函数的返回值并不直接是列表类型（list），例如：

```
>>> type([1,2,3,4,5])
<class 'list'>
>>> type(range(1,6))
<class 'range'>
```

可以看到，range() 函数的返回值类型为 range，而不是 list。而如果想要得到 range() 函数创建的数字列表，还需要借助 list() 函数，比如：

```
>>> list(range(1,6))
[1, 2, 3, 4, 5]
```

可以看到，如果将 range() 作为 list() 的参数，其输出就是一个数字列表。

不仅如此，在使用 range() 函数时，还可以指定步长。例如，下面的代码打印 1~10 内的偶数：

```
1.          even_numbers = list(range(2,11,2))
2.          print(even_numbers)
```

在这个示例中，函数 range() 从 2 开始数，然后不断地加 2，直到达到或超过终值，因此输出如下：

```
[2, 4, 6, 8, 10]
```

注意，即便 range() 第二个参数恰好符合条件，最终创建的数字列表中也不会包含它。

实际使用时，range() 函数常常和 Python 循环结构、推导式（后续会讲，这里先不涉及）一起使用，几乎能够创建任何需要的数字列表。

例如，创建这样一个列表，其中包含前 10 个整数（即 1~10）的平方，实现代码如下：

```
squares = []
for value in range(1,11):
    square = value**2
    squares.append(square)
print(squares)
```

运行结果为：

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

8.Python tuple 元组详解

元组是 Python 中另一个重要的序列结构，和列表类似，也是由一系列按特定顺序排序的元素组成。和列表不同的是，列表可以任意操作元素，是可变序列；而元组是不可变序列，即元组中的元素不可以单独修改。

元组可以看做是不可变的列表。通常情况下，元组用于保存不可修改的内容。

从形式上看，元组的所有元素都放在一对小括号“()”中，相邻元素之间用逗号“,”分隔，

如下所示：

```
(element1, element2, ... , elementn)
```

其中 element1~elementn 表示元组中的各个元素，个数没有限制，且只要是 Python 支持的数据类型就可以。

从存储内容上看，元组可以存储整数、实数、字符串、列表、元组等任何类型的数据，并且在同一个元组中，元素的类型可以不同，例如：

```
("c.biancheng.net",1,[2,'a'],("abc",3.0))
```

在这个元组中，有多种类型的数据，包括整形、字符串、列表、元组。

另外，我们都知道，列表的数据类型是 list，那么元组的数据类型是什么呢？通过 type() 函数，就可以查看到元组的数据类型，例如：

```
>>> type(("c.biancheng.net",1,[2,'a'],("abc",3.0)))
<class 'tuple'>
```

可以看到，元组是 typle 类型，这也是很多教程中用 tuple 指代元组的原因。

Python 创建元组

Python 提供了多种创建元组的方法，下面一一进行介绍。

= 运算符直接创建元组

和其他类型的 Python 变量一样，在创建元组时，可以使用赋值运算符“=”直接将一个元组赋值给变量，其语法格式如下：

```
tuplename = (element1,element2,...,elementn)
```

其中，tuplename 表示创建的元组名，可以使用任何符合 Python 命名规则，且不和 Python 内置函数重名的标识符作为元组名。

再次强调，创建元组的语法和创建列表的语法非常相似，唯一的不同在于，创建列表使用的是 []，而创建元组使用的是 ()。

例如，下面定义的元组都是合法的：

```
num = (7,14,21,28,35)
a_tuple = ("C 语言中文网","http://c.biancheng.net")
python = ("Python",19,[1,2],('c',2.0))
```

在 Python 中，元组通常都是使用一对小括号将所有元素括起来的，但小括号不是必须的，只要将各元素用逗号隔开，Python 就会将其视为元组，举个例子：

```
a_tuple = "C 语言中文网","http://c.biancheng.net"
print(a_tuple)
```

运行结果为：

```
('C 语言中文网', 'http://c.biancheng.net')
```

需要额外注意的一点是，当创建的元组中只有一个元素时，此元组后面必须要加一个逗号“,”，否则 Python 解释器会将其误认为字符串。例如：

```
#创建元组 a_typle
a_tuple =("C 语言中文网",)
print(type(a_tuple))
print(a_tuple)

#创建字符串 a
a = ("C 语言中文网")
print(type(a))
print(a)
```

运行结果为：

```
<class 'tuple'>
('C 语言中文网',)
<class 'str'>
C 语言中文网
```

显然，前者 a_tuple 才是元组类型，而变量 a 只是一个字符串。

使用 tuple()函数创建元组

除了第一种最常见的创建方式外，Python 还提供了 tuple() 函数来创建元组，它可以直接将列表、区间（range）等对象转换成元组。

tuple 函数的语法格式如下：

```
tuple(data)
```

其中，data 表示可以转化为元组的数据，其类型可以是字符串、元组、range 对象等。

例如如下代码：

```
# 将列表转换成元组
a_list = ['crazyit', 20, -1.2]
a_tuple = tuple(a_list)
print(a_tuple)

# 使用 range()函数创建区间（range）对象
a_range = range(1, 5)
print(a_range)
# 将区间转换成元组
b_tuple = tuple(a_range)
print(b_tuple)
# 创建区间时还指定步长
c_tuple = tuple(range(4, 20, 3))
print(c_tuple)
```

运行结果为：

```
('crazyit', 20, -1.2)

range(1, 5)

(1, 2, 3, 4)

(4, 7, 10, 13, 16, 19)
```

Python 访问元组元素

和列表完全一样，如果想访问元组中的指定元素，可以使用元组中各元素的索引值获取，例如，定义一个包含 3 个元素的元组，若想访问第 2 个元素，可以使用如下的代码：

```
1.          a_tuple = ('crazyit', 20, -1.2)
2.          print(a_tuple[1])
```

运行结果为：

```
20
```

在此基础上，元组也支持采用切片方式获取指定范围内的元素，例如，访问 a_tuple 元组中前 2 个元组，可以执行如下代码：

```
1.          a_tuple = ('crazyit', 20, -1.2)
2.          #采用切片方式
3.          print(a_tuple[:2])
```

运行结果为：

```
('crazyit', 20)
```

有关切片的语法格式以及用法，已在《Python 序列》一节中做了详细介绍。

Python 修改元组元素

前面已经讲过，元组是不可变序列，元组中的元素不可以单独进行修改。但是，元组也不是完全不能修改。

比如，我们可以对元组进行重新赋值：

```
a_tuple = ('crazyit', 20, -1.2)
print(a_tuple)
#对元组进行重新赋值
a_tuple = ('c.biancheng.net',"C 语言中文网")
print(a_tuple)
```

运行结果为：

```
('crazyit', 20, -1.2)
('c.biancheng.net', 'C 语言中文网')
```

另外，还可以通过连接多个元组的方式向元组中添加新元素。例如：

```
a_tuple = ('crazyit', 20, -1.2)
print(a_tuple)
#连接多个元组
a_tuple = a_tuple + ('c.biancheng.net',)
print(a_tuple)
```

运行结果为：

```
('crazyit', 20, -1.2)
('crazyit', 20, -1.2, 'c.biancheng.net')
```

需要注意的是，在使用此方式时，元组连接的内容必须都是元组，不能将元组和字符串或列表进行连接，否则或抛出 TypeError 错误。例如：

```
1.          a_tuple = ('crazyit', 20, -1.2)
2.          #元组连接字符串
3.          a_tuple = a_tuple + 'c.biancheng.net'
4.          print(a_tuple)
```

运行结果为：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 4, in <module>
    a_tuple = a_tuple + 'c.biancheng.net'
TypeError: can only concatenate tuple (not "str") to tuple
```

Python 删除元组

当已经创建的元组确定不再使用时，可以使用 del 语句将其删除，例如：

```
1.          a_tuple = ('crazyit', 20, -1.2)
2.          print(a_tuple)
3.          #删除 a_tuple 元组
4.          del(a_tuple)
5.          print(a_tuple)
```

运行结果为：

```
('crazyit', 20, -1.2)
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 4, in <module>
    print(a_tuple)
NameError: name 'a_tuple' is not defined
```

前面已经介绍过，在实际开发中，del() 语句并不常用，因为 Python 自带的垃圾回收机制会自动销毁不用的元组。

9.Python 元组和列表的区别

元组和列表同属序列类型，且都可以按照特定顺序存放一组数据，数据类型不受限制，只要是 Python 支持的数据类型就可以。那么，元组和列表有哪些区别呢？

元组和列表最大的区别就是，列表中的元素可以进行任意修改，就好比是用铅笔在纸上写的字，写错了还可以擦除重写；而元组中的元素无法修改，除非将元组整体替换掉，就好比是用圆珠笔写的字，写了就擦不掉了，除非换一张纸。

需要注意的是，这样的差异势必会影响两者的存储方式，我们来直接看下面的例子：

```
>>> listdemo = []
>>> listdemo.__sizeof__()
40
>>> tupleDemo = ()
>>> tupleDemo.__sizeof__()
24
```

可以看到，对于列表和元组来说，虽然它们都是空的，但元组却比列表少占用 16 个字节，这是为什么呢？

事实上，就是由于列表是动态的，它需要存储指针来指向对应的元素（占用 8 个字节）。另外，由于列表中元素可变，所以需要额外存储已经分配的长度大小（占用 8 个字节）。但是对于元组，情况就不同了，元组长度大小固定，且存储元素不可变，所以存储空间也是固定的。

读者可能会问题，既然列表这么强大，还要元组这种序列类型干什么？

通过对比列表和元组存储方式的差异，我们可以引申出这样的结论，即元组要比列表更加轻量级，所以从总体上来说，元组的性能速度要由于列表。

另外，Python 会在后台，对静态数据做一些资源缓存。通常来说，因为垃圾回收机制的存在，如果一些变量不被使用了，Python 就会回收它们所占用的内存，返还给操作系统，以便其他变量或其他应用使用。

但是对于一些静态变量（比如元组），如果它不被使用并且占用空间不大时，Python 会暂时缓存这部分内存。这样的话，当下次再创建同样大小的元组时，Python 就可以不用再向操作系统发出请求去寻找内存，而是可以直接分配之前缓存的内存空间，这样就能大大加快程序的运行速度。

下面的例子，是计算初始化一个相同元素的列表和元组分别所需的时间。我们可以看到，元组的初始化速度要比列表快 5 倍。

```
C:\Users\mengma>python -m timeit 'x=(1,2,3,4,5,6)'
20000000 loops, best of 5: 9.97 nsec per loop
C:\Users\mengma>python -m timeit 'x=[1,2,3,4,5,6]'
5000000 loops, best of 5: 50.1 nsec per loop
```

当然，如果你想要增加、删减或者改变元素，那么列表显然更优。因为对于元组来说，必须得通过新建一个元组来完成。

总的来说，元组确实没有列表那么多功能，但是元组依旧是很重要的序列类型之一，元组的不可替代性体现在以下这些场景中：

1. 元组作为很多内置函数和序列类型方法的返回值存在，也就是说，在使用某些函数或者方法时，它的返回值会元组类型，因此你必须对元组进行处理。
2. 元组比列表的访问和处理速度更快，因此，当需要对指定元素进行访问，且不涉及修改元素的操作时，建议使用元组。
3. 元组可以在映射（和集合的成员）中当做“键”使用，而列表不行。这会在后续章节中作详解介绍。

10.Python 列表和元组的底层实现是怎样的？

有关列表（list）和元组（tuple）的底层实现，本节分别从它们的源码来进行分析。

首先来分析 list 列表，它的具体结构如下所示：

```
1.          typedef struct {
```

```
2.              PyObject_VAR_HEAD
3.              /* Vector of pointers to list elements.  list[0] is ob_item[0], etc.
    */
4.              PyObject **ob_item;
5.
6.              /* ob_item contains space for 'allocated' elements.  The number
7.               * currently in use is ob_size.
8.               * Invariants:
9.               *     0 <= ob_size <= allocated
10.              *     len(list) == ob_size
11.              *     ob_item == NULL implies ob_size == allocated == 0
12.              * list.sort() temporarily sets allocated to -1 to detect mutations.
13.              *
14.              * Items must normally not be NULL, except during construction when
15.              * the list is not yet visible outside the function that builds it.
16.              */
17.             Py_ssize_t allocated;
18.         } PyListObject;
```

有兴趣的读者，可直接阅读 list 列表实现的源码文件 listobject.h 和 listobject.c。

list 本质上是一个长度可变的连续数组。其中 ob_item 是一个指针列表，里边的每一个指针都指向列表中的元素，而 allocated 则用于存储该列表目前已被分配的空间大小。

需要注意的是，allocated 和列表的实际空间大小不同，列表实际空间大小，指的是 len(list) 返回的结果，也就是上边代码中注释中的 ob_size，表示该列表总共存储了多少个元素。而在实际情况中，为了优化存储结构，避免每次增加元素都要重新分配内存，列表预分配的空间 allocated 往往会大于 ob_size。

因此 allocated 和 ob_size 的关系是：allocated >= len(list) = ob_size >= 0。

如果当前列表分配的空间已满（即 allocated == len(list)），则会向系统请求更大的内存空间，并把原来的元素全部拷贝过去。

接下来再分析元组，如下所示为 Python 3.7 tuple 元组的具体结构：

```
        typedef struct {
            PyObject_VAR_HEAD
            PyObject *ob_item[1];

            /* ob_item contains space for 'ob_size' elements.
             * Items must normally not be NULL, except during construction when
             * the tuple is not yet visible outside the function that builds it.
             */
        } PyTupleObject;
```

有兴趣的读者，可阅读 tuple 元组实现的源码文件 tupleobject.h 和 tupleobject.c。

tuple 和 list 相似，本质也是一个数组，但是空间大小固定。不同于一般数组，Python 的 tuple 做了许多优化，来提升在程序中的效率。

举个例子，为了提高效率，避免频繁的调用系统函数 free 和 malloc 向操作系统申请和释放空间，tuple 源文件中定义了一个 free_list：

```
static PyTupleObject *free_list[PyTuple_MAXSAVESIZE];
```

所有申请过的，小于一定大小的元组，在释放的时候会被放进这个 free_list 中以供下次使用。也就是说，如果以后需要再去创建同样的 tuple，Python 就可以直接从缓存中载入。

11.Python dict 字典

和列表相同，字典也是许多数据的集合，属于可变序列类型。不同之处在于，它是无序的可变序列，其保存的内容是以“键值对”的形式存放的。

字典类型是 Python 中唯一的映射类型。“映射”是数学中的术语，简单理解，它指的是元素之间相互对应的关系，即通过一个元素，可以唯一找到另一个元素。如图 1 所示。

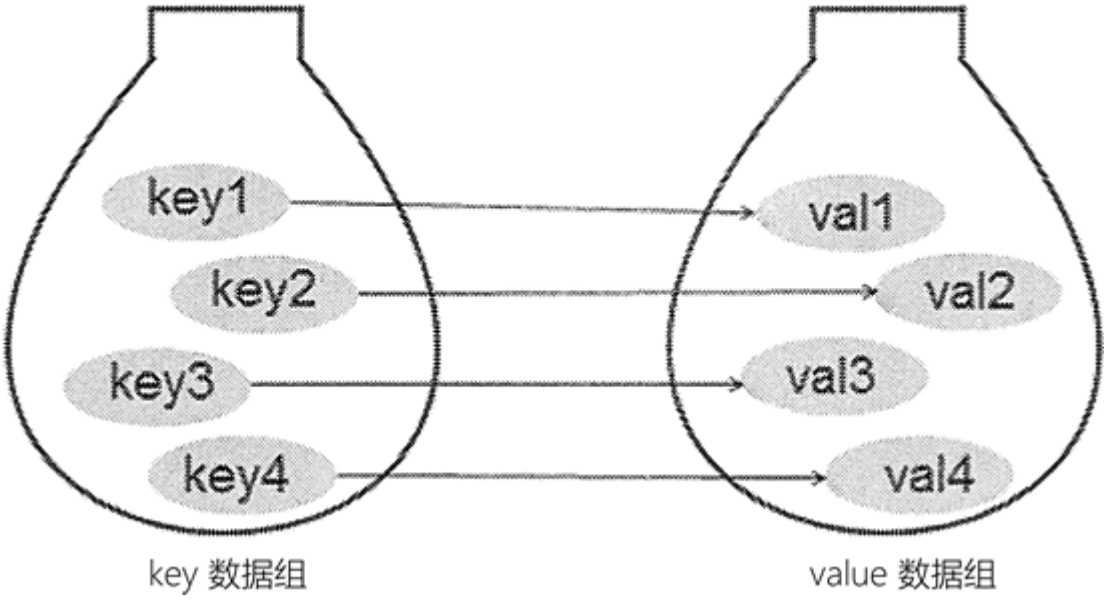


图 1 映射关系示意图

字典中，习惯将各元素对应的索引称为键（key），各个键对应的元素称为值（value），键及其关联的值称为“键值对”。

字典类型很像学生时代常用的新华字典。我们知道，通过新华字典中的音节表，可以快速找到想要查找的汉字。其中，字典里的音节表就相当于字典类型中的键，而键对应的汉字则相

当于值。

总的来说，字典类型所具有的主要特征如表 1 所示。

表 1 Python 字典特征

主要特征	解释
通过键而不是通过索引来读取元素	字典类型有时也称为关联数组或者散列表（hash）。它是通过键将一系列的值联系起来的，这样就可以通过键从字典中获取指定项，但不能通过索引来获取。
字典是任意数据类型的无序集合	和列表、元组不同，通常会将索引值 0 对应的元素称为第一个元素。而字典中的元素是无序的。
字典是可变的，并且可以任意嵌套	字典可以在原处增长或者缩短（无需生成一个副本），并且它支持任意深度的嵌套，即字典存储的值也可以是列表或其它的字典。
字典中的键必须唯一	字典中，不支持同一个键出现多次，否则，只会保留最后一个键值对。
字典中的键必须不可变	字典中的值是不可变的，只能使用数字、字符串或者元组，不能使用列表。

Python 中的字典类型相当于 Java 或者 C++ 中的 Map 对象。

和列表、元组一样，字典也有它自己的类型。Python 中，字典的数据类型为 dict，通过 type() 函数即可查看：

```
>>> a = {'one':1,'two':2,'three':3} #a 是一个字典类型
>>> type(a)
<class 'dict'>
```

Python 创建字典

创建字典的方式有很多，下面一一做介绍。

1）花括号语法创建字典

由于字典中每个元素都包含 2 部分，分别是键和值，因此在创建字典时，键和值之间使用冒号分隔，相邻元素之间使用逗号分隔，所有元素放在大括号 {} 中。

Python 字典类型的语法格式如下：

```
dictname = {'key':'value1','key2':'value2',...,'keyn':valuen}
```

其中 dictname 表示字典类型名，keyn : valuen 表示各个元素的键值对。需要注意的是，同一字典中各个元素的键值必须唯一。

如下代码示范了使用花括号语法创建字典：

```
1.          scores = {'语文': 89, '数学': 92, '英语': 93}
2.          print(scores)
```

```
3.          # 空的花括号代表空的 dict
4.          empty_dict = {}
5.          print(empty_dict)
6.          # 使用元组作为 dict 的 key
7.          dict2 = {(20, 30):'good', 30:[1,2,3]}
8.          print(dict2)
```

运行结果为：

```
{'语文': 89, '数学': 92, '英语': 93}

{}

{(20, 30): 'good', 30: [1, 2, 3]}
```

可以看到，同一字典中，键值可以是整数、字符串或者元组，只要符合唯一和不可变的特性；对应的值可以是 Python 支持的任意数据类型。

2）通过 fromkeys() 方法创建字典

Python 中，还可以使用 dict 字典类型提供的 fromkeys() 方法创建所有键值为空的字典，使用此方法的语法格式为：

```
dictname = dict.fromkeys(list，value=None)
```

其中，list 参数表示字典中所有键的列表，value 参数默认为 None，表示所有键对应的值。

例如：

```
1.          knowledge = {'语文', '数学', '英语'}
```

```
2.          scores = dict.fromkeys(knowledge)
3.          print(scores)
```

运行结果为：

```
{'语文': None, '数学': None, '英语': None}
```

可以看到，knowledge 列表中的元素全部作为了 scores 字典的键，而各个键对应的值都为空（None）。此种创建方式，通常用于初始化字典，设置 value 的默认值。

3）通过 dict() 映射函数创建字典

通过 dict() 函数创建字典的写法有多种，表 2 罗列出了常用的几种方式，它们创建的都是同一个字典 a。

表 2 dict() 函数创建字典

创建格式	注意事项
>>> a = dict(one=1,two=2,three=3)	注意，其中的 one、two、three 都是字符串，但使用此方式创建字典时，字符串不能带引号。

>>> demo = [('two',2),('one',1),('three',3)] #方式 1
>>> demo = [['two',2],['one',1],['three',3]] #方式 2
>>> demo =
	向 dict() 函数传入列表或元组，而它们中的元素又各自是包含 2 个元素的列表或元组，其中第一个元素作为键，第二个元素作为值。

(('two',2),('one',1),('three',3)) #方式 3
>>> demo =
(['two',2],['one',1],['three',3]) #方式 4
>>> a = dict(demo)	

>>> demokeys = ['one','two','three'] #还可以是字符串或元组
>>> demovalues = [1,2,3] #还可以是字符串或元组
>>> a =
dict(zip(demokeys,demovalues))
	通过应用 dict() 函数和 zip() 函数，可将前两个列表转换为对应的字典。

注意，无论采用以上哪种方式创建字典，字典中各元素的键都只能是字符串、元组或数字，不能是列表。

如果不为 dict() 函数传入任何参数，则代表创建一个空的字典。例如如下代码：

```
1.          # 创建空的字典
2.          dict5 = dict()
3.          print(dict5)
```

运行结果为：

```
{}
```

Python 访问字典

和列表、元组不同，它们访问元素都是通过下标，而字典不同，它是通过键来访问对应的元素值。

因为字典中元素是无序的，所以不能像列表、元组那样，采用切片的方式一次性访问多个元素。

例如，如果想访问刚刚建立的字典 a 中，获取元素 1，可以使用下面的代码：

```
>>> a['one']
1
```

在使用此方法获取指定键的值时，如果键不存在，则会抛出如下异常：

```
>>> a['four']
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    a['four']
KeyError: 'four'
```

另外，除了上面这种方式外，Python 更推荐使用 dict 类型提供的 get() 方法获取指定键的值。get() 方法的语法格式为：

```
dict.get(key[,default])
```

其中，dict 指的是所创建的字典名称；key 表示指定的键；default 用于指定要查询的键不存在时，此方法返回的默认值，如果不手动指定，会返回 None。

例如，通过 get() 方法获取字典 a 中“two”对应的值，执行代码如下：

```
>>> a = dict(one=1,two=2,three=3)
>>>a.get('two')
2
```

当然，为了防止在获取指定键的值时，因不存在该键而导致抛出异常，在使用 get() 方法时，可以为其设置默认值，这样，即便指定的键不存在，也不回报错。例如：

```
>>> a = dict(one=1,two=2,three=3)
>>> a.get('four','字典中无此键')
'字典中无此键'
```

Python 删除字典

和删除列表、元组一样，手动删除字典也可以使用 del 语句。例如：

```
>>> a = dict(one=1,two=2,three=3)
>>> a
{'one': 1, 'two': 2, 'three': 3}
>>> del(a)
```

```
>>> a
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    a
NameError: name 'a' is not defined
```

12.Python dict 字典基本操作

由于字典属于可变序列，所以我们可以任意操作字典中的键值对（ key-value 对 ）。

Python 中，常见的字典操作有以下几种：

1. 向现有字典中添加新的键值对。
2. 修改现有字典中的键值对。
3. 从现有字典中删除指定的键值对。
4. 判断现有字典中是否存在指定的键值对。

初学者要牢记，字典中常常包含多个键值对，而 key 是字典的关键数据，字典的基本操作都是围绕 key 值实现的。

Python 字典添加键值对

如果要为 dict 添加键值对，只需为不存在的 key 赋值即可。实现此操作的语法格式如下：

```
dict[key] = value
```

此格式中，各个参数的含义如表 1 所示。

表 1 Python 字典添加键值对语法参数

参数	含义
dict	表示字典名称。
key	表示要添加元素的键。注意，既然是添加新的元素，那么就要保证此元素的键和字典中现有元素的键互不相同。
value	表示要添加数据的值，只要是 Python 支持的数据类型就可以。

例如，下面代码演示了在现有字典 a 的基础上，添加新元素的过程：

```
a = {'数学':95}
print(a)
#添加新键值对
a['语文'] = 89
print(a)
#再次添加新键值对
a['英语'] = 90
print(a)
```

运行结果为：

```
{'数学': 95}
{'数学': 95, '语文': 89}
{'数学': 95, '语文': 89, '英语': 90}
```

Python 字典修改键值对

“修改键值对”并不是同时修改某一键值对的键和值，而只是修改某一键值对中的值。

由于在字典中，各元素的键必须是唯一的，因此，如果新添加元素的键与已存在元素的键相同，原来键所对应的值就会被新的值替换掉。例如：

```
a = {'数学': 95, '语文': 89, '英语': 90}
a['语文']=100
print(a)
```

运行结果为：

```
{'数学': 95, '语文': 100, '英语': 90}
```

可以看到，字典中没有再添加一个 {'语文':100} 键值对，而是对原有键值对 {'语文': 89} 中的值做了修改。

Python 字典删除键值对

如果要删除字典中的键值对，还是可以使用 del 语句。例如：

```
# 使用 del 语句删除键值对
a = {'数学': 95, '语文': 89, '英语': 90}
del a['语文']
del a['数学']
print(a)
```

运行结果为：

```
{'英语': 90}
```

判断字典中是否存在指定键值对

如果要判断字典中是否存在指定键值对，首先应判断字典中是否有对应的键。判断字典是否包含指定键值对的键，可以使用 in 或 not in 运算符。

> 需要指出的是，对于 dict 而言，in 或 not in 运算符都是基于 key 来判断的。

例如如下代码：

```
a = {'数学': 95, '语文': 89, '英语': 90}
# 判断 a 中是否包含名为'数学'的 key
print('数学' in a) # True
# 判断 a 是否包含名为'物理'的 key
print('物理' in a) # False
```

运行结果为：

```
True
False
```

通过 in（或 not in）运算符，我们可以很轻易地判断出现有字典中是否包含某个键，如果存在，由于通过键可以很轻易的获取对应的值，因此很容易就能判断出字典中是否有指定的键值对。

13.Python dict 字典方法完全攻略

我们知道，字典的数据类型为 dict，我们可使用 `dir(dict)` 来查看该类包含哪些方法。在交

互式解释器中输入 dir(dict) 命令，将看到如下输出结果：

```
>>> dir(dict)
['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update',
'values']
```

这些方法中，clear()、fromkeys()、get() 方法的功能和用法已经在前面章节中讲过，这里不再重复介绍。下面给大家一一介绍另外的这些方法。

Python keys()、values() 和 items()方法

这 3 个方法之所以放在一起介绍，是因为它们都用来获取字典中的特定数据。keys() 方法用于返回字典中的所有键；values() 方法用于返回字典中所有键对应的值；items() 用于返回字典中所有的键值对。

例如：

```
1.          a = {'数学': 95, '语文': 89, '英语': 90}
2.          print(a.keys())
3.          print(a.values())
4.          print(a.items())
```

运行结果为：

```
dict_keys(['数学', '语文', '英语'])
dict_values([95, 89, 90])
dict_items([('数学', 95), ('语文', 89), ('英语', 90)])
```

注意，在 Python 2.x 中，这三个方法的返回值是列表类型。但在 Python 3 中，并不是我们常见的列表和元组类型，因为 Python 3 不希望用户直接操作这几个方法的返回值。如果想使用返回的数据，有以下 2 种方法：

1. 使用 list() 函数，将它们返回的数据转换成列表，例如：

```
a = {'数学': 95, '语文': 89, '英语': 90}
b = list(a.keys())
print(b)
```

运行结果为：

```
['数学', '语文', '英语']
```

2. 也可以利用多重赋值的技巧，利用循环结构将键或值分别赋给不同的变量，比如说：

```
a = {'数学': 95, '语文': 89, '英语': 90}
for k in a.keys():
    print(k, end=' ')
print("\n---------------")
for v in a.values():
    print(v, end=' ')
print("\n---------------")
for k,v in a.items():
    print("key:",k," value:",v)
```

运行结果为：

```
数学 语文 英语
---------------
95 89 90
---------------
key: 数学  value: 95
key: 语文  value: 89
key: 英语  value: 90
```

由于目前尚未学习 for 循环，初学者可先跳过此种方法，等学会 for 循环后再回过头来学习。

Python copy()方法

copy() 方法用于返回一个具有相同键值对的新字典，例如：

```
1.          a = {'one': 1, 'two': 2, 'three': [1,2,3]}
2.          b = a.copy()
3.          print(b)
```

运行结果为：

```
{'one': 1, 'two': 2, 'three': [1, 2, 3]}
```

可以看到，通过 copy() 方法，就可以将字典 a 的数据拷贝给字典 b。

注意，copy() 方法所遵循的拷贝原理，既有深拷贝，也有浅拷贝。拿拷贝字典 a 为例，copy() 方法只会对最表层的键值对进行深拷贝，也就是说，它会再申请一块内存用来存放 {'one': 1,

'two': 2, 'three': []}；而对于某些列表类型的值来说，此方法对其做的是浅拷贝，也就是说，b 中的 [1,2,3] 的值不是自己独有，而是和 a 共有。

例如：

```
1.          a = {'one': 1, 'two': 2, 'three': [1,2,3]}
2.          b = a.copy()
3.          #向 a 中添加新键值对，由于 b 已经提前将 a 所有键值对都深拷贝过来，因此 a 添加新键值对，不会影响 b。
4.          a['four']=100
5.          print(a)
6.          print(b)
7.          #由于 b 和 a 共享[1,2,3]（浅拷贝），因此移除 a 中列表中的元素，也会影响 b。
8.          a['three'].remove(1)
9.          print(a)
10.         print(b)
```

运行结果为：

```
{'one': 1, 'two': 2, 'three': [1, 2, 3], 'four': 100}
{'one': 1, 'two': 2, 'three': [1, 2, 3]}
{'one': 1, 'two': 2, 'three': [2, 3], 'four': 100}
{'one': 1, 'two': 2, 'three': [2, 3]}
```

从运行结果不难看出，对 a 增加新键值对，b 不变；而修改 a 某键值对中列表内的元素，b 也会相应改变。

Python update()方法

update() 方法可使用一个字典所包含的键值对来更新已有的字典。

在执行 update() 方法时，如果被更新的字典中已包含对应的键值对，那么原 value 会被覆盖；如果被更新的字典中不包含对应的键值对，则该键值对被添加进去。

例如：

```
1.          a = {'one': 1, 'two': 2, 'three': 3}
2.          a.update({'one':4.5, 'four': 9.3})
3.          print(a)
```

运行结果为：

```
{'one': 4.5, 'two': 2, 'three': 3, 'four': 9.3}
```

从上面的执行过程可以看出，由于被更新的 dict 中已包含 key 为“one”的键值对，因此更新时该键值对的 value 将被改写；但如果被更新的 dict 中不包含 key 为“four”的键值对，那么更新时就会为原字典增加一个键值对。

Python pop 方法

pop() 方法用于获取指定 key 对应的 value，并删除这个键值对。

如下方法示范了 pop() 方法的用法：

```
1.          a = {'one': 1, 'two': 2, 'three': 3}
2.          print(a.pop('one'))
3.          print(a)
```

运行结果为：

```
1

{'two': 2, 'three': 3}
```

此程序中，第 2 行代码将会获取“one”对应的值 1，并删除该键值对。

Python popitem()方法

popitem() 方法用于随机弹出字典中的一个键值对。

注意，此处的随机其实是假的，它和 list.pop() 方法一样，也是弹出字典中最后一个键值对。但由于字典存储键值对的顺序是不可知的，因此 popitem() 方法总是弹出底层存储的最后一个键值对。

如下代码示范了 popitem() 方法的用法：

```
1.          a = {'one': 1, 'two': 2, 'three': 3}
2.          print(a)
3.          # 弹出字典底层存储的最后一个键值对
4.          print(a.popitem())
5.          print(a)
```

运行结果为：

```
{'one': 1, 'two': 2, 'three': 3}

('three', 3)

{'one': 1, 'two': 2}
```

实际上，由于 popitem 弹出的是一个元组，因此我们也可以通过序列解包的方式，用两个变量分别接收 key 和 value。

例如如下代码：

```
a = {'one': 1, 'two': 2, 'three': 3}
# 将弹出项的 key 赋值给 k、value 赋值给 v
k, v = a.popitem()
print(k, v)
```

运行结果为：

```
three 3
```

Python setdefault()方法

setdefault() 方法也用于根据 key 来获取对应 value 的值。但该方法有一个额外的功能，即当程序要获取的 key 在字典中不存在时，该方法会先为这个不存在的 key 设置一个默认的 value，然后再返回该 key 对应的 value。

也就是说，setdefault() 方法总能返回指定 key 对应的 value；如果该键值对存在，则直接返回该 key 对应的 value；如果该键值对不存在，则先为该 key 设置默认的 value，

然后再返回该 key 对应的 value。

如下代码示范了 setdefault() 方法的用法：

纯文本复制

```
a = {'one': 1, 'two': 2, 'three': 3}
# 设置默认值，该 key 在 dict 中不存在，新增键值对
print(a.setdefault('four', 9.2))
print(a)
# 设置默认值，该 key 在 dict 中存在，不会修改 dict 内容
print(a.setdefault('one', 3.4))
print(a)
```

运行结果为：

```
9.2
{'one': 1, 'two': 2, 'three': 3, 'four': 9.2}
1
{'one': 1, 'two': 2, 'three': 3, 'four': 9.2}
```

14.Python 使用字典格式化字符串

前面章节介绍过，在格式化字符串时，如果要格式化的字符串模板中包含多个变量，后面就需要按顺序给出多个变量，这种方式对于字符串模板中包含少量变量的情形是合适的，但如果字符串模板中包含大量变量，这种按顺序提供变量的方式则有些不合适。

这时，就可以使用字典对字符串进行格式化输出，具体方法是：在字符串模板中按 key 指

定变量，然后通过字典为字符串模板中的 key 设置值。

例如如下程序：

```
纯文本复制
```

```
1.          # 字符串模板中使用 key
2.          temp = '教程是:%(name)s, 价格是:%(price)010.2f, 出版社是:%(publish)s'
3.          book = {'name':'Python 基础教程', 'price': 99, 'publish': 'C 语言中文网
'}
4.          # 使用字典为字符串模板中的 key 传入值
5.          print(temp % book)
6.          book = {'name':'C 语言小白变怪兽', 'price':159, 'publish': 'C 语言中文网
'}
7.          # 使用字典为字符串模板中的 key 传入值
8.          print(temp % book)
```

运行上面程序，可以看到如下输出结果：

```
教程是:Python 基础教程, 价格是:0000099.00, 出版社是:C 语言中文网

教程是:C 语言小白变怪兽, 价格是:0000159.00, 出版社是:C 语言中文网
```

15.Python set 集合

Python 中的集合，和数学中的集合概念一样，用来保存不重复的元素，即集合中的元素都是唯一的，互不相同。

从形式上看，和字典类似，Python 集合会将所有元素放在一对大括号 {} 中，相邻元素之间用“,”分隔，如下所示：

```
{element1,element2,...,elementn}
```

其中，elementn 表示集合中的元素，个数没有限制。

从内容上看，同一集合中，只能存储不可变的数据类型，包括整形、浮点型、字符串、元组，无法存储列表、字典、集合这些可变的数据类型，否则 Python 解释器会抛出 TypeError 错误。比如说：

```
>>> {{'a':1}}
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    {{'a':1}}
TypeError: unhashable type: 'dict'
>>> {[1,2,3]}
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    {[1,2,3]}
TypeError: unhashable type: 'list'
>>> {{1,2,3}}
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    {{1,2,3}}
TypeError: unhashable type: 'set'
```

并且需要注意的是，数据必须保证是唯一的，因为集合对于每种数据元素，只会保留一份。

例如：

```
>>> {1,2,1,(1,2,3),'c','c'}
{1, 2, 'c', (1, 2, 3)}
```

由于 Python 中的 set 集合是无序的，所以每次输出时元素的排序顺序可能都不相同。

其实，Python 中有两种集合类型，一种是 set 类型的集合，另一种是 frozenset 类型的集合，它们唯一的区别是，set 类型集合可以做添加、删除元素的操作，而 forzenset 类型集合不行。本节先介绍 set 类型集合，后续章节再介绍 forzenset 类型集合。

Python 创建 set 集合

Python 提供了 2 种创建 set 集合的方法，分别是使用 {} 创建和使用 set() 函数将列表、元组等类型数据转换为集合。

1）使用 {} 创建

在 Python 中，创建 set 集合可以像列表、元素和字典一样，直接将集合赋值给变量，从而实现创建集合的目的，其语法格式如下：

```
setname = {element1,element2,...,elementn}
```

其中 ,setname 表示集合的名称 ,起名时既要符合 Python 命名规范 ,也要避免与 Python

内置函数重名。

举个例子：

```
a = {1,'c',1,(1,2,3),'c'}
print(a)
```

运行结果为：

```
{1, 'c', (1, 2, 3)}
```

2) set()函数创建集合

set() 函数为 Python 的内置函数，其功能是将字符串、列表、元组、range 对象等可迭代对象转换成集合。该函数的语法格式如下：

```
setname = set(iteration)
```

其中，iteration 就表示字符串、列表、元组、range 对象等数据。

例如：

```
set1 = set("c.biancheng.net")
set2 = set([1,2,3,4,5])
set3 = set((1,2,3,4,5))
print("set1:",set1)
print("set2:",set2)
print("set3:",set3)
```

运行结果为：

```
set1: {'a', 'g', 'b', 'c', 'n', 'h', '.', 't', 'i', 'e'}
set2: {1, 2, 3, 4, 5}
set3: {1, 2, 3, 4, 5}
```

注意，如果要创建空集合，只能使用 set() 函数实现。因为直接使用一对 {}，Python 解释器会将其视为一个空字典。

Python 访问 set 集合元素

由于集合中的元素是无序的，因此无法向列表那样使用下标访问元素。Python 中，访问集合元素最常用的方法是使用循环结构，将集合中的数据逐一读取出来。

例如：

```
a = {1,'c',1,(1,2,3),'c'}
for ele in a:
    print(ele,end=' ')
```

运行结果为：

```
1 c (1, 2, 3)
```

由于目前尚未学习循环结构，以上代码初学者只需初步了解，后续学习循环结构后自然会明白。

Python 删除 set 集合

和其他序列类型一样，手动函数集合类型，也可以使用 del() 语句，例如：

```
a = {1,'c',1,(1,2,3),'c'}
print(a)
del(a)
print(a)
```

运行结果为：

```
{1, 'c', (1, 2, 3)}
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 4, in <module>
    print(a)
NameError: name 'a' is not defined
```

Python set 集合最常用的操作是向集合中添加、删除元素，以及集合之间做交集、并集、差集等运算。受到篇幅的限制，这些知识会放到下节进行详细讲解。

16.Python set 集合基本操作

Python set 集合最常用的操作是向集合中添加、删除元素，以及集合之间做交集、并集、差集等运算，本节将一一讲解这些操作的具体实现。

向 set 集合中添加元素

set 集合中添加元素，可以使用 set 类型提供的 add() 方法实现，该方法的语法格式为：

```
setname.add(element)
```

其中，setname 表示要添加元素的集合，element 表示要添加的元素内容。

需要注意的是，使用 add() 方法添加的元素，只能是数字、字符串、元组或者布尔类型(True 和 False) 值，不能添加列表、字典、集合这类可变的数据，否则 Python 解释器会报 TypeError 错误。例如：

```
a = {1,2,3}
a.add((1,2))
print(a)
a.add([1,2])
print(a)
```

运行结果为：

```
{(1, 2), 1, 2, 3}
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 4, in <module>
    a.add([1,2])
TypeError: unhashable type: 'list'
```

从 set 集合中删除元素

删除现有 set 集合中的指定元素，可以使用 remove() 方法，该方法的语法格式如下：

```
setname.remove(element)
```

使用此方法删除集合中元素，需要注意的是，如果被删除元素本就不包含在集合中，则此方法会抛出 KeyError 错误，例如：

```
a = {1, 2, 3}
a.remove(1)
print(a)
a.remove(1)
print(a)
```

运行结果为：

```
{2, 3}
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 4, in <module>
    a.remove(1)
KeyError: 1
```

上面程序中，由于集合中的元素 1 已被删除，因此当再次尝试使用 remove() 方法删除时，会引发 KeyError 错误。

如果我们不想在删除失败时令解释器提示 KeyError 错误，还可以使用 discard() 方法，此方法和 remove() 方法的用法完全相同，唯一的区别就是，当删除集合中元素失败时，此方法不会抛出任何错误。

例如：

```
a = {1, 2, 3}
a.remove(1)
print(a)
a.discard(1)
print(a)
```

运行结果为：

```
{2, 3}

{2, 3}
```

Python set 集合做交集、并集、差集运算

集合最常做的操作就是进行交集、并集、差集以及对称差集运算，首先有必要给大家普及一下各个运算的含义。

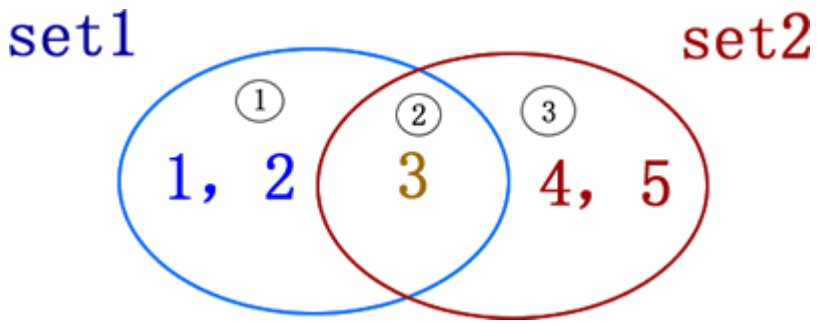


图 1 集合示意图

图 1 中，有 2 个集合，分别为 set1={1,2,3} 和 set2={3,4,5}，它们既有相同的元素，也有不同的元素。以这两个集合为例，分别做不同运算的结果如表 1 所示。

表 1 Python set 集合运算

运算操作	Python 运算符	含义	例子
交集	&	取两集合公共的元素	>>> set1 & set2 {3}

并集	|	取两集合全部的元素	>>> set1 | set2 {1,2,3,4,5}

差集	-	取一个集合中另一集合没有的元素	>>> set1 - set2
			{1,2}
			>>> set2 - set1
			{4,5}

对称差集	^	取集合 A 和 B 中不属于 A&B 的元素	>>> set1 ^ set2 {1,2,4,5}

17.Python set 集合方法

前面学习了 set 集合，本节来一一学习 set 类型提供的方法。首先，通过 dir(set) 命令可以查看它有哪些方法：

```
>>> dir(set)
['add', 'clear', 'copy', 'difference', 'difference_update', 'discard', 'intersection',
'intersection_update', 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove',
'symmetric_difference', 'symmetric_difference_update', 'union', 'update']
```

各个方法的具体语法结构及功能如表 1 所示。

表 1 Python set 方法

方法名	语法格式	功能	实例

add()	set1.add()	向set1集合中添加数字、字符串、元组或者布尔类型	>>> set1 = {1,2,3}
			>>> set1.add((1,2))
			>>> set1
			{(1, 2), 1, 2, 3}

clear()	set1.clear()	清空set1集合中所有元素	>>> set1 = {1,2,3}
			>>> set1.clear()
			>>> set1
			set()
			
			set()才表示空集合，{}表示的是空字典

copy()	set2 = set1.copy()	拷贝	>>> set1 = {1,2,3}

		set1 集合给 set2	>>> set2 = set1.copy()
			>>> set1.add(4)
			>>> set1
			{1, 2, 3, 4}
			>>> set1
			{1, 2, 3}

difference()	set3 = set1.difference(set2)	将 set1 中有而 set2 没有的元素给 set3	>>> set1 = {1,2,3}
			>>> set2 = {3,4}
			>>> set3 = set1.difference(set2)
			>>> set3
			{1, 2}

difference_update()	set1.difference_update(set2)	从 set1 中删除与 set2 相同	>>> set1 = {1,2,3}
			>>> set2 = {3,4}
			>>> set1.difference_update(set2)
			>>> set1

		的元素	{1, 2}

discard()	set1.discard(elem)	删除 set1 中的 elem 元素	>>> set1 = {1,2,3}
			>>> set1.discard(2)
			>>> set1
			{1, 3}
			>>> set1.discard(4)
			{1, 3}

intersection()	set3 = set1.intersection(set2)	取 set1 和 set2 的交集给 set3	>>> set1 = {1,2,3}
			>>> set2 = {3,4}
			>>> set3 = set1.intersection(set2)
			>>> set3
			{3}

intersection_update()	set1.intersection_update(set2)	取 set1 和 set2 的交集，	>>> set1 = {1,2,3}
			>>> set2 = {3,4}
			>>> set1.intersection_update(set2)
			>>> set1

		并更新给 set1	{3}

isdisjoint()	set1.isdisjoint(set2)	判断 set1 和 set2 是否没有交集，有交集返回 Fals e ;没有交集返回 True	>>> set1 = {1,2,3}
			>>> set2 = {3,4}
			>>> set1.isdisjoint(set2)
			False

issubset()	set1.issubset(set2)	判断	>>> set1 = {1,2,3}

		set1 是否是 set2 的子集	>>> set2 = {1,2}
			>>> set1.issubset(set2)
			False

issuperset()	set1.issuperset(set2)	判断 set2 是否是 set1 的子集	>>> set1 = {1,2,3}
			>>> set2 = {1,2}
			>>> set1.issuperset(set2)
			True

pop()	a = set1.pop()	取 set1 中一个元素，并赋值给 a	>>> set1 = {1,2,3}
			>>> a = set1.pop()
			>>> set1
			{2,3}
			>>> a
			1

remove()	set1.remove(elem)	移除set1中的elem元素	>>> set1 = {1,2,3}
			>>> set1.remove(2)
			>>> set1
			{1, 3}
			>>> set1.remove(4)
			Traceback (most recent call last):
			File "<pyshell#90>", line 1, in <module>
			set1.remove(4)
			KeyError: 4

symmetric_difference()	set3 = set1.symmetric_difference(set2)	取set1和set2中互不相同的元素，给	>>> set1 = {1,2,3}
			>>> set2 = {3,4}
			>>> set3 = set1.symmetric_difference(set2)
			>>> set3
			{1, 2, 4}

		set3	

symmetric_difference_update()	set1.symmetric_difference_update(set2)	取set1和set2中互不相同的元素，并更新给set1	>>> set1 = {1,2,3}
			>>> set2 = {3,4}
			>>> set1.symmetric_difference_update(set2)
			>>> set1
			{1, 2, 4}

union()	set3 = set1.union(set2)	取set1和set2的并集，赋给set3	>>> set1 = {1,2,3}
			>>> set2 = {3,4}
			>>> set3=set1.union(set2)
			>>> set3
			{1, 2, 3, 4}

update()	set1.update(elem)	添加列表或集合中的元素到 set1	>>> set1 = {1,2,3}
			>>> set1.update([3,4])
			>>> set1
			{1,2,3,4}

18.Python frozenset 集合

frozenset 是 set 的不可变版本，因此 set 集合中所有能改变集合本身的方法（如 add、remove、discard、xxx_update 等），frozenset 都不支持；set 集合中不改变集合本身的方法，fronzenset 都支持。

在交互式解释器中输入 dir(frozenset) 命令来查看 frozenset 集合的全部方法，可以看到如下输出结果：

```
>>> dir(frozenset)
['copy', 'difference', 'intersection', 'isdisjoint', 'issubset', 'issuperset',
'symmetric_difference', 'union']
```

很明显，frozenset 的这些方法和 set 集合同名方法的功能完全相同。

frozenset 的作用主要有两点：

1. 当集合元素不需要改变时，使用 frozenset 代替 set 更安全。
2. 当某些 API 需要不可变对象时 必须用 frozenset 代替 set。比如 dict 的 key 必须是不可变对象，因此只能用 frozenset；再比如 set 本身的集合元素必须是不可变的，因此 set 不能包含 set，set 只能包含 frozenset。

如下程序示范了在 set 中添加 frozenset：

```
s = set()
#创建 frozenset 不可变集合，使用 frozenset() 函数
frozen_s = frozenset('Kotlin')
# 为 set 集合添加 frozenset
s.add(frozen_s)
print('s 集合的元素：', s)
sub_s = {'Python'}
# 为 set 集合添加普通 set 集合，程序报错
s.add(sub_s)
```

上面程序中第 4 行代码为 set 添加 frozenset，程序完全没有问题，这样 set 中就包括一个 frozenset 子集合 ;第 8 行代码试图向 set 中添加普通 set ,程序会报出 TypeError 异常。运行上面代码，可以看到如下输出结果：

```
s 集合的元素： {frozenset({'t', 'o', 'i', 'K', 'l', 'n'})}
Traceback (most recent call last):
```

```
  File "C:\Users\mengma\Desktop\1.py", line 8, in <module>
    s.add(sub_s)
TypeError: unhashable type: 'set'
```

19.深入底层了解 Python 字典和集合，一眼看穿他们的本质！

字典和集合是进行过性能高度优化的数据结构，特别是对于查找、添加和删除操作。本节将结合实例介绍它们在具体场景下的性能表现，以及与列表等其他数据结构的对比。

例如，有一个存储产品信息（产品 ID、名称和价格）的列表，现在的需求是，借助某件产品的 ID 找出其价格。则实现代码如下：

```
1.          def find_product_price(products, product_id):
2.              for id, price in products:
3.                  if id == product_id:
4.                      return price
5.              return None
6.          products = [
7.              (111, 100),
8.              (222, 30),
9.              (333, 150)
10.         ]
11.         print('The price of product 222 is {}'.format(find_product_price(products,
    222)))
```

运行结果为：

```
The price of product 222 is 30
```

在上面程序的基础上，如果列表有 n 个元素，因为查找的过程需要遍历列表，那么最坏情况下的时间复杂度就为 O(n)。即使先对列表进行排序，再使用二分查找算法，也需要 O(logn) 的时间复杂度，更何况列表的排序还需要 O(nlogn) 的时间。

但如果用字典来存储这些数据，那么查找就会非常便捷高效，只需 O(1) 的时间复杂度就可以完成，因为可以直接通过键的哈希值，找到其对应的值，而不需要对字典做遍历操作，实现代码如下：

```
1.          products = {
2.            111: 100,
3.            222: 30,
4.            333: 150
5.          }
6.          print('The price of product 222 is {}'.format(products[222]))
```

运行结果为：

```
The price of product 222 is 30
```

有些读者可能对时间复杂度并没有直观的认识，没关系，再给大家列举一个实例。下面的代码中，初始化了含有 100,000 个元素的产品，并分别计算出了使用列表和集合来统计产品价格数量的运行时间：

```
1.          #统计时间需要用到 time 模块中的函数，了解即可
2.          import time
```

```
3.
4.          def find_unique_price_using_list(products):
5.              unique_price_list = []
6.              for _, price in products: # A
7.                  if price not in unique_price_list: #B
8.                      unique_price_list.append(price)
9.              return len(unique_price_list)
10.
11.         id = [x for x in range(0, 100000)]
12.         price = [x for x in range(200000, 300000)]
13.         products = list(zip(id, price))
14.
15.         # 计算列表版本的时间
16.         start_using_list = time.perf_counter()
17.         find_unique_price_using_list(products)
18.         end_using_list = time.perf_counter()
19.         print("time elapse using list: {}".format(end_using_list -
    start_using_list))
20.
21.         #使用集合完成同样的工作
22.         def find_unique_price_using_set(products):
23.             unique_price_set = set()
24.             for _, price in products:
25.                 unique_price_set.add(price)
26.             return len(unique_price_set)
27.
28.         # 计算集合版本的时间
29.         start_using_set = time.perf_counter()
30.         find_unique_price_using_set(products)
31.         end_using_set = time.perf_counter()
32.         print("time elapse using set: {}".format(end_using_set -
    start_using_set))
```

运行结果为：

```
time elapse using list: 68.78650900000001

time elapse using set: 0.010747099999989018
```

可以看到，仅仅十万的数据量，两者的速度差异就如此之大。而往往企业的后台数据都有上亿乃至十亿数量级，因此如果使用了不合适的数据结构，很容易造成服务器的崩溃，不但影响用户体验，并且会给公司带来巨大的财产损失。

那么，字典和集合为什么能如此高效，特别是查找、插入和删除操作呢？

字典和集合的工作原理

字典和集合能如此高效，和它们内部的数据结构密不可分。不同于其他数据结构，字典和集合的内部结构都是一张哈希表：

- 对于字典而言，这张表存储了哈希值（hash）、键和值这 3 个元素。
- 而对集合来说，哈希表内只存储单一的元素。

对于之前版本的 Python 来说，它的哈希表结构如下所示：

```
  | 哈希值（hash）  键（key）  值（value）
. |          ...
0 |    hash0      key0    value0
. |          ...
1 |    hash1      key1    value1
. |          ...
```

```
2 |    hash2      key2    value2
. |           ...
```

这种结构的弊端是，随着哈希表的扩张，它会变得越来越稀疏。比如，有这样一个字典：

```
{'name': 'mike', 'dob': '1999-01-01', 'gender': 'male'}
```

那么它会存储为类似下面的形式：

```
entries = [
['--', '--', '--']
[-230273521, 'dob', '1999-01-01'],
['--', '--', '--'],
['--', '--', '--'],
[1231236123, 'name', 'mike'],
['--', '--', '--'],
[9371539127, 'gender', 'male']
]
```

显然，这样非常浪费存储空间。为了提高存储空间的利用率，现在的哈希表除了字典本身的结构，会把索引和哈希值、键、值单独分开，也就是采用如下这种结构：

```
Indices
-----------------------------------------------
None | index | None | None | index | None | index ...
-----------------------------------------------


Entries
--------------------
hash0   key0  value0
```

```
---------------------
hash1   key1  value1
---------------------
hash2   key2  value2
---------------------
        ...
---------------------
```

在此基础上，上面的字典在新哈希表结构下的存储形式为：

```
indices = [None, 1, None, None, 0, None, 2]
entries = [
[1231236123, 'name', 'mike'],
[-230273521, 'dob', '1999-01-01'],
[9371539127, 'gender', 'male']
]
```

通过对比可以发现，空间利用率得到很大的提高。

清楚了具体的设计结构，接下来再分析一下如何使用哈希表完成对数据的插入、查找和删除操作。

哈希表插入数据

当向字典中插入数据时，Python 会首先根据键(key)计算出对应的哈希值(通过 hash(key) 函数)，而向集合中插入数据时， Python 会根据该元素本身计算对应的哈希值（ 通过

hash(valuse) 函数）。

例如：

```
dic = {"name":1}
print(hash("name"))
setDemo = {1}
print(hash(1))
```

运行结果为：

```
8230115042008314683
1
```

得到哈希值（例如为 hash）之后，再结合字典或集合要存储数据的个数（例如 n），就可以得到该元素应该插入到哈希表中的位置（比如，可以用 hash%n 的方式）。

如果哈希表中此位置是空的，那么此元素就可以直接插入其中；反之，如果此位置已被其他元素占用，那么 Python 会比较这两个元素的哈希值和键是否相等：

- 如果相等，则表明该元素已经存在，再比较他们的值，不相等就进行更新；
- 如果不相等，这种情况称为哈希冲突（即两个元素的键不同，但求得的哈希值相同）。

这种情况下，Python 会使用开放定址法、再哈希法等继续寻找哈希表中空余的位置，直到找到位置。

具体遇到哈希冲突时，各解决方法的具体含义可阅读《哈希表详解》一节做详细了解。

哈希表查找数据

在哈希表中查找数据，和插入操作类似，Python 会根据哈希值，找到该元素应该存储到哈希表中的位置，然后和该位置的元素比较其哈希值和键（集合直接比较元素值）：

- 如果相等，则证明找到；
- 反之，则证明当初存储该元素时，遇到了哈希冲突，需要继续使用当初解决哈希冲突的方法进行查找，直到找到该元素或者找到空位为止。

> 这里的找到空位，表示哈希表中没有存储目标元素。

哈希表删除元素

对于删除操作，Python 会暂时对这个位置的元素赋于一个特殊的值，等到重新调整哈希表的大小时，再将其删除。

需要注意的是，哈希冲突的发生往往会降低字典和集合操作的速度。因此，为了保证其高效性，字典和集合内的哈希表，通常会保证其至少留有 1/3 的剩余空间。随着元素的不停插入，当剩余空间小于 1/3 时，Python 会重新获取更大的内存空间，扩充哈希表，与此同时，表内所有的元素位置都会被重新排放。

虽然哈希冲突和哈希表大小的调整，都会导致速度减缓，但是这种情况发生的次数极少。所以，平均情况下，仍能保证插入、查找和删除的时间复杂度为 `O(1)`。

20.Python 深拷贝和浅拷贝详解

对于浅拷贝（shallow copy）和深度拷贝（deep copy），本节并不打算一上来抛出它们的概念，而是先从它们的操作方法说起，通过代码来理解两者的不同。

Python 浅拷贝

常见的浅拷贝的方法，是使用数据类型本身的构造器，比如下面两个例子：

```
1.          list1 = [1, 2, 3]
2.          list2 = list(list1)
3.          print(list2)
4.          print("list1==list2 ?",list1==list2)
5.          print("list1 is list2 ?",list1 is list2)
6.
7.          set1= set([1, 2, 3])
8.          set2 = set(set1)
9.          print(set2)
10.         print("set1==set2 ?",set1==set2)
11.         print("set1 is set2 ?",set1 is set2)
```

运行结果为：

```
[1, 2, 3]

list1==list2 ? True

list1 is list2 ? False

{1, 2, 3}

set1==set2 ? True

set1 is set2 ? False
```

在上面程序中，list2 就是 list1 的浅拷贝，同理 set2 是 set1 的浅拷贝。

当然，对于可变的序列，还可以通过切片操作符 “:” 来完成浅拷贝，例如：

```
1.          list1 = [1, 2, 3]
2.          list2 = list1[:]
3.          print(list2)
4.          print("list1 == list2 ?",list1 == list2)
5.          print("list1 is list2 ?",list1 is list2)
```

运行结果为：

```
[1, 2, 3]
list1 == list2 ? True
list1 is list2 ? False
```

除此之外，Python 还提供了对应的函数 copy.copy() 函数，适用于任何数据类型。其用法如下：

```
1.          import copy
2.          list1 = [1, 2, 3]
3.          list2 = copy.copy(list1)
4.          print(list2)
5.          print("list1 == list2 ?",list1 == list2)
6.          print("list1 is list2 ?",list1 is list2)
```

运行结果为：

```
[1, 2, 3]

list1 == list2 ? True

list1 is list2 ? False
```

不过需要注意的是，对于元组，使用 tuple() 或者切片操作符 ':' 不会创建一份浅拷贝，相反它会返回一个指向相同元组的引用：

```
1.          tuple1 = (1, 2, 3)
2.          tuple2 = tuple(tuple1)
3.          print(tuple2)
4.          print("tuple1 == tuple2 ?",tuple1 == tuple2)
5.          print("tuple1 is tuple2 ?",tuple1 is tuple2)
```

运行结果为：

```
(1, 2, 3)

tuple1 == tuple2 ? True

tuple1 is tuple2 ? True
```

此程序中，元组 (1, 2, 3) 只被创建一次，t1 和 t2 同时指向这个元组。

看到这里，也许你可能对浅拷贝有了初步的认识。浅拷贝，指的是重新分配一块内存，创建一个新的对象，但里面的元素是原对象中各个子对象的引用。

对数据采用浅拷贝的方式时，如果原对象中的元素不可变，那倒无所谓；但如果元素可变，浅拷贝通常会出现一些问题，例如：

```
list1 = [[1, 2], (30, 40)]
list2 = list(list1)

list1.append(100)
print("list1:",list1)
print("list2:",list2)

list1[0].append(3)
print("list1:",list1)
print("list2:",list2)

list1[1] += (50, 60)
print("list1:",list1)
print("list2:",list2)
```

运行结果为：

```
list1: [[1, 2], (30, 40), 100]

list2: [[1, 2], (30, 40)]

list1: [[1, 2, 3], (30, 40), 100]

list2: [[1, 2, 3], (30, 40)]

list1: [[1, 2, 3], (30, 40, 50, 60), 100]

list2: [[1, 2, 3], (30, 40)]
```

此程序中，首先初始化了 list1 列表，包含一个列表和一个元组；然后对 list1 执行浅拷贝，赋予 list2。因为浅拷贝里的元素是对原对象元素的引用，因此 list2 中的元素和 list1 指向同一个列表和元组对象。

接着往下看，list1.append(100) 表示对 list1 的列表新增元素 100。这个操作不会对 list2

产生任何影响，因为 list2 和 list1 作为整体是两个不同的对象，并不共享内存地址。操作过后 list2 不变，list1 会发生改变。

再来看 ,list1[0].append(3) 表示对 list1 中的第一个列表新增元素 3。因为 list2 是 list1 的浅拷贝 ,list2 中的第一个元素和 list1 中的第一个元素 ,共同指向同一个列表 ,因此 list2 中的第一个列表也会相对应的新增元素 3。

最后是 list1[1] += (50, 60)，因为元组是不可变的，这里表示对 list1 中的第二个元组拼接，然后重新创建了一个新元组作为 list1 中的第二个元素，而 list2 中没有引用新元组，因此 list2 并不受影响。

通过这个例子 ,你可以很清楚地看到使用浅拷贝可能带来的副作用。如果想避免这种副作用，完整地拷贝一个对象，就需要使用深拷贝。所谓深拷贝，是指重新分配一块内存，创建一个新的对象，并且将原对象中的元素，以递归的方式，通过创建新的子对象拷贝到新对象中。因此，新对象和原对象没有任何关联。

Python 中以 copy.deepcopy() 来实现对象的深度拷贝。比如上述例子写成下面的形式，就是深度拷贝：

```
1.          import copy
2.          list1 = [[1, 2], (30, 40)]
3.          list2 = copy.deepcopy(list1)
4.
5.          list1.append(100)
6.          print("list1:",list1)
```

```
7.          print("list2:",list2)
8.
9.          list1[0].append(3)
10.         print("list1:",list1)
11.         print("list2:",list2)
12.
13.         list1[1] += (50, 60)
14.         print("list1:",list1)
15.         print("list2:",list2)
```

运行结果为：

```
list1: [[1, 2], (30, 40), 100]

list2: [[1, 2], (30, 40)]

list1: [[1, 2, 3], (30, 40), 100]

list2: [[1, 2], (30, 40)]

list1: [[1, 2, 3], (30, 40, 50, 60), 100]

list2: [[1, 2], (30, 40)]
```

可以看到，无论 list1 如何变化，list2 都不变。因为此时的 list1 和 list2 完全独立，没有任何联系。

不过，深度拷贝也不是完美的，往往也会带来一系列问题。如果被拷贝对象中存在指向自身的引用，那么程序很容易陷入无限循环，例如：

```
1.          import copy
2.          list1 = [1]
3.          list1.append(list1)
4.          print(list1)
5.
```

```
6.          list2 = copy.deepcopy(list1)
7.          print(list2)
```

运行结果为：

```
[1, [...]]

[1, [...]]
```

此例子中，列表 x 中有指向自身的引用，因此 x 是一个无限嵌套的列表。但是当深度拷贝 x 到 y 后，程序并没有出现栈溢出的现象。这是为什么呢？

其实，这是因为深度拷贝函数 deepcopy 中会维护一个字典，记录已经拷贝的对象与其 ID。拷贝过程中，如果字典里已经存储了将要拷贝的对象，则会从字典直接返回。通过查看 deepcopy 函数实现的源码就会明白：

```
1.          def deepcopy(x, memo=None, _nil=[]):
2.              """Deep copy operation on arbitrary Python objects.
3.
4.              See the module's __doc__ string for more info.
5.              """
6.
7.              if memo is None:
8.                  memo = {}
9.              d = id(x) # 查询被拷贝对象 x 的 id
10.             y = memo.get(d, _nil) # 查询字典里是否已经存储了该对象
11.             if y is not _nil:
12.                 return y # 如果字典里已经存储了将要拷贝的对象，则直接返回
13.                 ...
```

第 5 章 Python 字符串常用方法

在了解字符串的基本使用之后，本章将介绍 Python 字符串类型常用的几个方法。
在 Python 开发过程中，经常需要对字符串进行一些特殊处理，比如拼接字符串、截取字符串、格式化字符串等，这些操作无需开发者自己设计实现，只需调用相应的字符串方法即可。
注意，本章所讲的字符串方法，和内置函数（如 len() ）不同，它们是字符串类型（str）所特有的，因此这些字符串方法的调用方式也比较特殊。

1.Python 字符串拼接（包含字符串拼接数字）

前面我们见过书写字符串的一种特殊方式，即直接将两个字符串紧挨着写在一起，这样的话 Python 就会自动拼接它们，比如说：

```
1.          s1 = "Hello,"'Charlie'
2.          print(s1)
```

上面代码将会输出：

```
Hello,Charlie
```

其实，这种写法只是书写字符串的一种特殊方法，并不能算是真正意义上的拼接字符串。

Python 使用加号（ + ）作为字符串的拼接运算符，例如如下代码：

```
1.          s2 = "Python "
2.          s3 = "iS Funny"
3.          #使用+拼接字符串
4.          s4 = s2 + s3
5.          print(s4)
```

Python 字符串拼接数字

除此之外，在某些特殊场景中，我们需要将字符串与数字进行拼接，而 Python 不允许直接拼接数字和字符串，程序必须先将数字转换成字符串。

为了将数字转换成字符串，可以使用 str() 或 repr() 函数，例如如下代码：

```
s1 = "这是数字: "
p = 99.8
#字符串直接拼接数值，程序报错
print(s1 + p)
#使用 str()将数值转换成字符串
print(s1 + str(p))
#使用 repr()将数值转换成字符串
print(s1 + repr(p))
```

上面程序中直接拼接字符串和数值，程序会报错。

str() 和 repr() 函数都可以将数字转换成字符串，其中 str 是 Python 内置的类型（和 int、float 一样），而 repr() 则只是一个函数。此外，repr 还有一个功能，它会以 Python 表达式的形式来表示值。对比如下代码：

```
st = "I will play my fife"
print (st)
print(repr(st))
```

上面代码中，st 本身就是一个字符串，但程序依然使用了 repr() 对字符串进行转换。运行上面程序，可以看到如下输出结果：

```
I will play my fife
'I will play my fife'
```

通过上面的输出结果可以看出，如果直接使用 print() 函数输出字符串，将只能看到字符串的内容，没有引号；但如果先使用 repr() 函数对字符串进行处理，然后再使用 print() 执行输出，将可以看到带引号的字符串，这就是 Python 字符串的表达式形式。

在交互式解释器中输入一个变量（或表达式）时，Python 会自动使用 repr() 函数处理该变量（或表达式）。

2.Python 截取字符串

字符串本质上就是由多个字符组成的，Python 允许通过索引来操作字符，比如获取指定索引处的字符，获取指定字符在字符串中的位置等。

Python 字符串直接在方括号（[]）中使用索引即可获取对应的字符，其基本语法格式为：

```
string[index]
```

这里的 string 表示要截取的字符串，index 表示索引值。Python 规定，字符串中第一个字符的索引为 0、第二个字符的索引为 1，后面各字符依此类推。此外，Python 也允许从后面开始计算索引，最后一个字符的索引为 -1，倒数第二个字符的索引为 -2，依此类推。

【例 1】

```
s = 'crazyit.org is very good'
# 获取 s 中索引 2 处的字符
print(s[2]) # 输出 a
# 获取 s 中从右边开始，索引 4 处的字符
print(s[-4]) # 输出 g
```

除可获取单个字符之外，Python 也可以在方括号中使用范围来获取字符串的中间“一段”（被称为子串），其基本语法格式为：

```
string[start : end : step]
```

此格式中，各参数的含义如下：

- string：要截取的字符串；
- start：表示要截取的第一个字符所在的索引（截取时包含该字符）。如果不指定，默认为 0，也就是从字符串的开头截取；
- end：表示要截取的最后一个字符所在的索引（截取时不包含该字符）。如果不指定，默认为字符串的长度；
- step 指的是从 start 索引处的字符开始，每 step 个距离获取一个字符，直至 end 索引出的字符。step 默认值为 1，当省略该值时，最后一个冒号也可以省略。

【例 2】基本用法

```
s = 'crazyit.org is very good'
# 获取 s 中从索引 3 处到索引 5 处（不包含）的子串
print(s[3: 5]) # 输出 zy
# 获取 s 中从索引 3 处到倒数第 5 个字符的子串
print(s[3: -5]) # 输出 zyit.org is very
# 获取 s 中从倒数第 6 个字符到倒数第 3 个字符的子串
print(s[-6: -3]) # 输出 y g
#每隔 1 个，取一个字符
print(s[::2]) # 输出 caytogi eygo
```

【例 3】 start、end 以及 step 都可以省略。

```
# 获取 s 中从索引 5 处到结束的子串
print(s[5: ]) # 输出 it.org is very good
# 获取 s 中从倒数第 6 个字符到结束的子串
print(s[-6: ]) # 输出 y good
# 获取 s 中从开始到索引 5 处的子串
print(s[: 5]) # 输出 crazy
# 获取 s 中从开始到倒数第 6 个字符的子串
print(s[: -6]) #输出 crazyit.org is ver
```

此外，Python 字符串还支持用 in 运算符判断是否包含某个子串。例如如下代码：

```
# 判断 s 是否包含'very'子串
print('very' in s) # True
print('fkit' in s) # False
```

还可使用全局内置的 min() 和 max() 函数获取字符串中最小字符和最大字符。例如如下代码：

```
# 输出 s 字符串中最大的字符
print(max(s)) # z
# 输出 s 字符串中最大的字符
print(min(s)) # 空格
```

3.Python split()分割字符串方法

Python 中，除了可以使用一些内建函数获取字符串的相关信息外（例如 len() 函数获取字符串长度），字符串类型本身也拥有一些方法供我们使用。

注意，这里所说的方法，指的是字符串类型 str 本身所提供的，由于涉及到类和对象的知识，初学者不必深究，只需要知道方法的具体用法即可。

从本节开始，将给大家介绍一些常用的字符串类型方法，本节先介绍分割字符串的 split() 方法。

split() 方法可以实现将一个字符串按照指定的分隔符切分成多个子串，这些子串会被保存到列表中（不包含分隔符），作为方法的返回值反馈回来。该方法的基本语法格式如下：

```
str.split(sep,maxsplit)
```

此方法中各部分参数的含义分别是：

1. str：表示要进行分割的字符串；
2. sep：用于指定分隔符，可以包含多个字符。此参数默认为 None，表示所有空字符，包括空格、换行符“\n”、制表符“\t”等。
3. maxsplit：可选参数，用于指定分割的次数，最后列表中子串的个数最多为maxsplit+1。如果不指定或者指定为 -1，则表示分割次数没有限制。

在 split 方法中，如果不指定 sep 参数，那么也不能指定 maxsplit 参数。

同内建函数（如 len）的使用方式不同，字符串变量所拥有的方法，只能采用“字符串.方法名()”的方式调用。这里不用纠结为什么，学完类和对象之后，自然会明白。

例如，定义一个保存 C 语言中文网网址的字符串，然后用 split() 方法根据不同的分隔符进行分隔，执行过程如下：

```
>>> str = "C 语言中文网 >>> c.biancheng.net"
>>> str
'C 语言中文网 >>> c.biancheng.net'
>>> list1 = str.split() #采用默认分隔符进行分割
>>> list1
['C 语言中文网', '>>>', 'c.biancheng.net']
>>> list2 = str.split('>>>') #采用多个字符进行分割
>>> list2
['C 语言中文网 ', ' c.biancheng.net']
>>> list3 = str.split('.') #采用 . 号进行分割
>>> list3
['C 语言中文网 >>> c', 'biancheng', 'net']
>>> list4 = str.split(' ',4) #采用空格进行分割，并规定最多只能分割成 4 个子串
>>> list4
['C 语言中文网', '>>>', 'c.biancheng.net']
>>> list5 = str.split('>') #采用 > 字符进行分割
>>> list5
['C 语言中文网 ', '', '', ' c.biancheng.net']
>>>
```

需要注意的是，在未指定 sep 参数时，split() 方法默认采用空字符进行分割，但当字符串

中有连续的空格或其他空字符时，都会被视为一个分隔符对字符串进行分割，例如：

```
>>> str = "C 语言中文网    >>>    c.biancheng.net"  #包含 3 个连续的空格
>>> list6 = str.split()
>>> list6
['C 语言中文网', '>>>', 'c.biancheng.net']
>>>
```

4.Python join()合并字符串方法

join() 方法也是非常重要的字符串方法，它是 split() 方法的逆方法，用来将列表（或元组）中包含的多个字符串连接成一个字符串。

想详细了解 split() 方法的读者，可阅读《Python split()方法》一节。

使用 join() 方法合并字符串时，它会将列表（或元组）中多个字符串采用固定的分隔符连接在一起。例如，字符串“c.biancheng.net”就可以看做是通过分隔符“.”将 ['c','biancheng','net'] 列表合并为一个字符串的结果。

join() 方法的语法格式如下：

```
newstr = str.join(iterable)
```

此方法中各参数的含义如下：

1. newstr：表示合并后生成的新字符串；
2. str：用于指定合并时的分隔符；
3. iterable：做合并操作的源字符串数据，允许以列表、元组等形式提供。

【例 1】将列表中的字符串合并成一个字符串。

```
>>> list = ['c','biancheng','net']
>>> '.'.join(list)
'c.biancheng.net'
```

【例 2】将元组中的字符串合并成一个字符串。

```
>>> dir = '','usr','bin','env'
>>> type(dir)
<class 'tuple'>
>>> '/'.join(dir)
'/usr/bin/env'
```

5. Python count()方法：统计字符串出现的次数

count 方法用于检索指定字符串在另一字符串中出现的次数，如果检索的字符串不存在，则返回 0，否则返回出现的次数。

count 方法的语法格式如下：

```
str.count(sub[,start[,end]])
```

此方法中，各参数的具体含义如下：

1. str：表示原字符串；
2. sub：表示要检索的字符串；
3. start：指定检索的起始位置，也就是从什么位置开始检测。如果不指定，默认从头开始检索；
4. end：指定检索的终止位置，如果不指定，则表示一直检索到结尾。

【例 1】检索字符串“c.biancheng.net”中“.”出现的次数。

```
>>> str = "c.biancheng.net"
>>> str.count('.')
2
```

【例 2】

```
>>> str = "c.biancheng.net"
>>> str.count('.',1)
2
```

```
>>> str.count('.',2)
1
```

前面讲过，字符串中各字符对应的检索值，从 0 开始，因此，本例中检索值 1 对应的是第 2 个字符 ‘.’ ，从输出结果可以分析出，从指定索引位置开始检索，其中也包含此索引位置。

【例 3】

```
>>> str = "c.biancheng.net"
>>> str.count('.',2,-3)
1
>>> str.count('.',2,-4)
0
```

6. Python find()方法：检测字符串中是否包含某子串

find() 方法用于检索字符串中是否包含目标字符串，如果包含，则返回第一次出现该字符串的索引；反之，则返回 -1。

find() 方法的语法格式如下：

```
str.find(sub[,start[,end]])
```

此格式中各参数的含义如下：

1. str：表示原字符串；
2. sub：表示要检索的目标字符串；
3. start：表示开始检索的起始位置。如果不指定，则默认从头开始检索；
4. end：表示结束检索的结束位置。如果不指定，则默认一直检索到结尾。

【例 1】用 find() 方法检索 “c.biancheng.net” 中首次出现 “.” 的位置索引。

```
>>> str = "c.biancheng.net"
>>> str.find('.')
1
```

【例 2】手动指定起始索引的位置。

```
>>> str = "c.biancheng.net"
>>> str.find('.',2)
11
```

【例 3】手动指定起始索引和结束索引的位置。

```
>>> str = "c.biancheng.net"
>>> str.find('.',2,-4)
-1
```

位于索引（2，-4）之间的字符串为“biancheng”，由于其不包含“.”，因此 find() 方法的返回值为 -1。

注意，Python 还提供了 rfind() 方法，与 find() 方法最大的不同在于，rfind() 是从字符串右边开始检索。例如：

```
>>> str = "c.biancheng.net"
>>> str.rfind('.')
11
```

7.Python index()方法：检测字符串中是否包含某子串

同 find() 方法类似，index() 方法也可以用于检索是否包含指定的字符串，不同之处在于，当指定的字符串不存在时，index() 方法会抛出异常。

index() 方法的语法格式如下：

```
str.index(sub[,start[,end]])
```

此格式中各参数的含义分别是：

1. str：表示原字符串；
2. sub：表示要检索的子字符串；
3. start：表示检索开始的起始位置，如果不指定，默认从头开始检索；
4. end：表示检索的结束位置，如果不指定，默认一直检索到结尾。

【例 1】用 index() 方法检索“c.biancheng.net”中首次出现“.”的位置索引。

```
>>> str = "c.biancheng.net"
>>> str.index('.')
1
```

【例 2】当检索失败时，index()会抛出异常。

```
>>> str = "c.biancheng.net"
>>> str.index('z')
Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    str.index('z')
ValueError: substring not found
```

同 find() 和 rfind() 一样，字符串变量还具有 rindex() 方法，其作用和 index() 方法类似，不同之处在于它是从右边开始检索，例如：

```
>>> str = "c.biancheng.net"
>>> str.rindex('.')
11
```

8.Python 字符串对齐方法（ljust()、rjust()和 center()）详

Python str 提供了 3 种可用来进行文本对齐的方法，分别是 ljust()、rjust() 和 center() 方法，本节就来一一介绍它们的用法。

Python ljust()方法

ljust() 方法的功能是向指定字符串的右侧填充指定字符，从而达到左对齐文本的目的。

ljust() 方法的基本格式如下：

```
S.ljust(width[, fillchar])
```

其中各个参数的含义如下：

- S：表示要进行填充的字符串；
- width：表示包括 S 本身长度在内，字符串要占的总长度；
- fillchar：作为可选参数，用来指定填充字符串时所用的字符，默认情况使用空格。

【例 1】

```
S = 'http://c.biancheng.net/python/'
addr = 'http://c.biancheng.net'
print(S.ljust(35))
print(addr.ljust(35))
```

输出结果为：

```
http://c.biancheng.net/python/
http://c.biancheng.net
```

注意，该输出结果中除了明显可见的网址字符串外，其后还有空格字符存在，每行一共 35 个字符长度。

【例 2】

```
1.          S = 'http://c.biancheng.net/python/'
2.          addr = 'http://c.biancheng.net'
3.          print(S.ljust(35,'-'))
4.          print(addr.ljust(35,'-'))
```

输出结果为：

```
http://c.biancheng.net/python/-----
http://c.biancheng.net----------
```

此程序和例 1 的唯一区别是，填充字符从空格改为 '-'。

Python rjust()方法

rjust() 和 ljust() 方法类似，唯一的不同在于，rjust() 方法是向字符串的左侧填充指定字符，从而达到右对齐文本的目的。

rjust() 方法的基本格式如下：

```
S.rjust(width[, fillchar])
```

其中各个参数的含义和 ljust() 完全相同，所以这里不再重复描述。

【例 3】

```
S = 'http://c.biancheng.net/python/'
addr = 'http://c.biancheng.net'
print(S.rjust(35))
print(addr.rjust(35))
```

输出结果为：

```
    http://c.biancheng.net/python/
            http://c.biancheng.net
```

可以看到，每行字符串都占用 35 个字节的位置，实现了整体的右对齐效果。

【例 4】

```
S = 'http://c.biancheng.net/python/'
addr = 'http://c.biancheng.net'
print(S.rjust(35,'-'))
print(addr.rjust(35,'-'))
```

输出结果为：

```
-----http://c.biancheng.net/python/
-------------http://c.biancheng.net
```

Python center()方法

center() 字符串方法与 ljust() 和 rjust() 的用法类似，但它让文本居中，而不是左对齐或

右对齐。

center() 方法的基本格式如下：

```
S.center(width[, fillchar])
```

其中各个参数的含义和 ljust()、rjust() 方法相同。

【例 5】

```
S = 'http://c.biancheng.net/python/'
addr = 'http://c.biancheng.net'
print(S.center(35,))
print(addr.center(35,))
```

输出结果为：

```
   http://c.biancheng.net/python/
       http://c.biancheng.net
```

【例 6】

```
S = 'http://c.biancheng.net/python/'
addr = 'http://c.biancheng.net'
print(S.center(35,'-'))
print(addr.center(35,'-'))
```

输出结果为：

```
---http://c.biancheng.net/python/--
-------http://c.biancheng.net------
```

9.Python startswith()和 endswith()

除了前面介绍的几个方法外，Python 字符串变量还可以使用 startswith() 和 endswith() 方法。

startswith()方法

startswith() 方法用于检索字符串是否以指定字符串开头，如果是返回 True；反之返回 False。此方法的语法格式如下：

```
str.startswith(sub[,start[,end]])
```

此格式中各个参数的具体含义如下：

1. str：表示原字符串；
2. sub：要检索的子串；
3. start：指定检索开始的起始位置索引，如果不指定，则默认从头开始检索；
4. end：指定检索的结束位置索引，如果不指定，则默认一直检索在结束。

【例 1】判断“c.biancheng.net”是否以“c”子串开头。

```
>>> str = "c.biancheng.net"
>>> str.startswith("c")
True
```

【例 2】

```
>>> str = "c.biancheng.net"
>>> str.startswith("http")
False
```

【例 3】从指定位置开始检索。

```
>>> str = "c.biancheng.net"
>>> str.startswith("b",2)
True
```

endswith()方法

endswith() 方法用于检索字符串是否以指定字符串结尾，如果是则返回 True；反之则返回 False。该方法的语法格式如下：

```
str.endswith(sub[,start[,end]])
```

此格式中各参数的含义如下：

1. str：表示原字符串；
2. sub：表示要检索的字符串；
3. start：指定检索开始时的起始位置索引（字符串第一个字符对应的索引值为 0），如果不指定，默认从头开始检索。

4. end：指定检索的结束位置索引，如果不指定，默认一直检索到结束。

【例 4】检索“c.biancheng.net”是否以“net”结束。

```
>>> str = "c.biancheng.net"
>>> str.endswith("net")
True
```

10.Python 字符串大小写转换

Python 中，为了方便对字符串中的字母进行大小写转换，字符串变量提供了 3 种方法，分别是 title()、lower() 和 upper()。

Python title()方法

title() 方法用于将字符串中每个单词的首字母转为大写，其他字母全部转为小写，转换完成后，此方法会返回转换得到的字符串。如果字符串中没有需要被转换的字符，此方法会将字符串原封不动地返回。

title() 方法的语法格式如下：

```
str.title()
```

其中，str 表示要进行转换的字符串。

【例 1】

```
>>> str = "c.biancheng.net"
>>> str.title()
'C.Biancheng.Net'
>>> str = "I LIKE C"
>>> str.title()
'I Like C'
```

Python lower()方法

lower() 方法用于将字符串中的所有大写字母转换为小写字母，转换完成后，该方法会返回新得到的字符串。如果字符串中原本就都是小写字母，则该方法会返回原字符串。

lower() 方法的语法格式如下：

```
str.lower()
```

其中，str 表示要进行转换的字符串。

【例 2】

```
>>> str = "I LIKE C"

>>> str.lower()

'i like c'
```

Python upper()方法

upper() 的功能和 lower() 方法恰好相反，它用于将字符串中的所有小写字母转换为大写字母，和以上两种方法的返回方式相同，即如果转换成功，则返回新字符串；反之，则返回原字符串。

upper() 方法的语法格式如下：

```
str.upper()
```

其中，str 表示要进行转换的字符串。

【例 3】

```
>>> str = "i like C"

>>> str.upper()

'I LIKE C'
```

需要注意的是，以上 3 个方法都仅限于将转换后的新字符串返回，而不会修改原字符串。

11.Python 去除字符串中空格

用户输入数据时，很有可能会无意中输入多余的空格，或者在一些场景中，字符串前后不允

许出现空格和特殊字符，此时就需要去除字符串中的空格和特殊字符。

> 这里的特殊字符，指的是制表符（\t）、回车符（\r）、换行符（\n）等。

Python 中，字符串变量提供了 3 种方法来删除字符串中多余的空格和特殊字符，它们分别是：

1. strip()：删除字符串前后（左右两侧）的空格或特殊字符。
2. lstrip()：删除字符串前面（左边）的空格或特殊字符。
3. rstrip()：删除字符串后面（右边）的空格或特殊字符。

注意，Python 的 str 是不可变的（不可变的意思是指，字符串一旦形成，它所包含的字符序列就不能发生任何改变），因此这三个方法只是返回字符串前面或后面空白被删除之后的副本，并不会改变字符串本身。

Python strip()方法

strip() 方法用于删除字符串左右两个的空格和特殊字符，该方法的语法格式为：

```
str.strip([chars])
```

其中，str 表示原字符串，[chars] 用来指定要删除的字符，可以同时指定多个，如果不手动指定，则默认会删除空格以及制表符、回车符、换行符等特殊字符。

【例 1】

```
>>> str = "   c.biancheng.net \t\n\r"
>>> str.strip()
'c.biancheng.net'
>>> str.strip(" ,\r")
'c.biancheng.net \t\n'
>>> str
'   c.biancheng.net \t\n\r'
```

分析运行结果不难看出，通过 strip() 确实能够删除字符串左右两侧的空格和特殊字符，但并没有真正改变字符串本身。

Python lstrip()方法

lstrip() 方法用于去掉字符串左右的空格和特殊字符。该方法的语法格式如下：

```
str.lstrip([chars])
```

其中，str 和 chars 参数的含义，分别同 strip() 语法格式中的 str 和 chars 完全相同。

【例 2】

```
>>> str = "   c.biancheng.net \t\n\r"
>>> str.lstrip()
'c.biancheng.net \t\n\r'
```

Python rstrip()方法

rstrip() 方法用于删除字符串右侧的空格和特殊字符，其语法格式为：

```
str.rstrip([chars])
```

str 和 chars 参数的含义和前面 2 种方法语法格式中的参数完全相同。

【例 3】

```
>>> str = "   c.biancheng.net \t\n\r"
>>> str.rstrip()
'   c.biancheng.net'
```

12.Python format()格式化输出方法

前面章节介绍了如何使用 % 操作符对各种类型的数据进行格式化输出，这是早期 Python 提供的方法。自 Python 2.6 版本开始，字符串类型（str）提供了 format() 方法对字符串进行格式化，本节就来学习此方法。

format() 方法的语法格式如下：

```
str.format(args)
```

此方法中，str 用于指定字符串的显示样式；args 用于指定要进行格式转换的项，如果有多项，之间有逗号进行分割。

学习 format() 方法的难点，在于搞清楚 str 显示样式的书写格式。在创建显示样式模板时，需要使用{}和：来指定占位符，其完整的语法格式为：

```
{ [index][ : [ [fill] align] [sign] [#] [width] [.precision] [type] ] }
```

注意，格式中用 [] 括起来的参数都是可选参数，即可以使用，也可以不使用。各个参数的含义如下：

- index :指定 :后边设置的格式要作用到 args 中第几个数据 ,数据的索引值从 0 开始。如果省略此选项，则会根据 args 中数据的先后顺序自动分配。
- fill : 指定空白处填充的字符。注意，当填充字符为逗号(,)且作用于整数或浮点数时，该整数（或浮点数）会以逗号分隔的形式输出，例如（1000000 会输出 1,000,000）。
- align：指定数据的对齐方式，具体的对齐方式如表 1 所示。

表 1 align 参数及含义

align	含义
<	数据左对齐。
>	数据右对齐。
=	数据右对齐，同时将符号放置在填充内容的最左侧，该选项只对数字类型有效。
^	数据居中，此选项需和 width 参数一起使用。

- sign：指定有无符号数，此参数的值以及对应的含义如表 2 所示。

表 2 sign 参数以含义

sign 参数	含义

+	正数前加正号，负数前加负号。
-	正数前不加正号，负数前加负号。
空格	正数前加空格，负数前加负号。
#	对于二进制数、八进制数和十六进制数，使用此参数，各进制数前会分别显示 0b、0o、0x 前缀；反之则不显示前缀。

- width：指定输出数据时所占的宽度。
- .precision：指定保留的小数位数。
- type：指定输出数据的具体类型，如表 3 所示。

表 3 type 占位符类型及含义

type 类型值	含义
s	对字符串类型格式化。
d	十进制整数。
c	将十进制整数自动转换成对应的 Unicode 字符。
e 或者 E	转换成科学计数法后，再格式化输出。
g 或 G	自动在 e 和 f（或 E 和 F）中切换。
b	将十进制数自动转换成二进制表示，再格式化输出。
o	将十进制数自动转换成八进制表示，再格式化输出。
x 或者 X	将十进制数自动转换成十六进制表示，再格式化输出。
f 或者 F	转换为浮点数（默认小数点后保留 6 位），再格式化输出。
%	显示百分比（默认显示小数点后 6 位）。

【例 1】

```
1.          str="网站名称：{:>9s}\t 网址：{:s}"
2.          print(str.format("C 语言中文网","c.biancheng.net"))
```

输出结果为：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 2, in
    print(str.format("C 语言中文网","c.biancheng.net"))
ValueError: cannot switch from automatic field numbering to manual field
specification
```

【例 2】
在实际开发中，数值类型有多种显示需求，比如货币形式、百分比形式等，使用 format() 方法可以将数值格式化为不同的形式。

纯文本复制

```
#以货币形式显示
print("货币形式：{:,d}".format(1000000))
#科学计数法表示
print("科学计数法：{:E}".format(1200.12))
#以十六进制表示
print("100 的十六进制：{:#x}".format(100))
#输出百分比形式
print("0.01 的百分比表示：{:.0%}".format(0.01))
```

输出结果为：

```
货币形式：1,000,000
科学计数法：1.200120E+03
100 的十六进制：0x64
0.01 的百分比表示：1%
```

13.Python encode()和 decode()方法

前面章节在介绍 bytes 类型时，已经对 encode() 和 decode() 方法的使用做了简单的介绍，本节将对这 2 个方法做详细地说明。

我们知道，最早的字符串编码是 ASCII 编码，它仅仅对 10 个数字、26 个大小写英文字母以及一些特殊字符进行了编码。ASCII 码做多只能表示 256 个符号，每个字符只需要占用 1 个字节。

随着信息技术的发展，各国的文字都需要进行编码，于是相继出现了 GBK、GB2312、UTF-8 编码等，其中 GBK 和 GB2312 是我国制定的中文编码标准，规定英文字符母占用 1 个字节，中文字符占用 2 个字节；而 UTF-8 是国际通过的编码格式，它包含了全世界所有国家需要用到的字符，其规定英文字符占用 1 个字节，中文字符占用 3 个字节。

> Python 3.x 默认采用 UTF-8 编码格式，有效地解决了中文乱码的问题。

在 Python 中，有 2 种常用的字符串类型，分别为 str 和 bytes 类型，其中 str 用来表示 Unicode 字符，bytes 用来表示二进制数据。str 类型和 bytes 类型之间就需要使用 encode() 和 decode() 方法进行转换。

Python encode()方法

encode() 方法为字符串类型（str）提供的方法，用于将 str 类型转换成 bytes 类型，这个过程也称为“编码”。

encode() 方法的语法格式如下：

```
str.encode([encoding="utf-8"][,errors="strict"])
```

注意，格式中用 [] 括起来的参数为可选参数，也就是说，在使用此方法时，可以使用 [] 中的参数，也可以不使用。

该方法各个参数的含义如表 1 所示。

表 1 encode()参数及含义

参数	含义
str	表示要进行转换的字符串。
encoding = "utf-8"	指定进行编码时采用的字符编码，该选项默认采用 utf-8 编码。例如，如果想使用简体中文，可以设置 gb2312。 当方法中只使用这一个参数时，可以省略前边的“encoding=”，直接写编码格式，例如 str.encode("UTF-8")。
errors = "strict"	指定错误处理方式，其可选择值可以是： strict：遇到非法字符就抛出异常。 ignore：忽略非法字符。 replace：用“？”替换非法字符。 xmlcharrefreplace：使用 xml 的字符引用。 该参数的默认值为 strict。

注意，使用 encode() 方法对原字符串进行编码，不会直接修改原字符串，如果想修改原字符串，需要重新赋值。

【例 1】将 str 类型字符串“C 语言中文网”转换成 bytes 类型。

```
>>> str = "C 语言中文网"
>>> str.encode()
b'C\xe8\xaf\xad\xe8\xa8\x80\xe4\xb8\xad\xe6\x96\x87\xe7\xbd\x91'
```

此方式默认采用 UTF-8 编码，也可以手动指定其它编码格式，例如：

```
>>> str = "C 语言中文网"
>>> str.encode('GBK')
b'C\xd3\xef\xd1\xd4\xd6\xd0\xce\xc4\xcd\xf8'
```

Python decode()方法

和 encode() 方法正好相反，decode() 方法用于将 bytes 类型的二进制数据转换为 str 类型，这个过程也称为“解码”。

decode() 方法的语法格式如下：

```
bytes.decode([encoding="utf-8"][,errors="strict"])
```

该方法中各参数的含义如表 2 所示。

表 2 decode()参数及含义

参数	含义
bytes	表示要进行转换的二进制数据。
encoding="utf-8"	指定解码时采用的字符编码，默认采用 utf-8 格式。当方法中只使用这一个参数时，可以省略“encoding=”，直接写编码方式即可。 注意，对 bytes 类型数据解码，要选择和当初编码时一样的格式。
errors = "strict"	指定错误处理方式，其可选择值可以是： strict：遇到非法字符就抛出异常。 ignore：忽略非法字符。 replace：用“？”替换非法字符。 xmlcharrefreplace：使用 xml 的字符引用。 该参数的默认值为 strict。

【例 2】

```
>>> str = "C 语言中文网"
>>> bytes=str.encode()
```

```
>>> bytes.decode()
'C 语言中文网'
```

注意，如果编码时采用的不是默认的 UTF-8 编码，则解码时要选择和编码时一样的格式，否则会抛出异常，例如：

```
>>> str = "C 语言中文网"
>>> bytes = str.encode("GBK")
>>> bytes.decode()  #默认使用 UTF-8 编码，会抛出以下异常
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    bytes.decode()
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xd3 in position 1: invalid continuation byte
>>> bytes.decode("GBK")
'C 语言中文网'
```

14.Python dir()和 help()帮助函数

前面，我们已经学习了很多字符串变量提供的方法，如 split()、join()、find()、index() 等，但这远远不是它的全部方法。由于篇幅有限，本章只能给大家列举一些最常用的方法。至于其他的方法，读者可通过本节介绍的 2 个帮助函数自行查看。

Python 非常方便，它不需要用户查询文档，只需掌握如下两个帮助函数，即可查看 Python 中的所有函数（方法）以及它们的用法和功能：

1. dir()：列出指定类或模块包含的全部内容（包括函数、方法、类、变量等）。
2. help()：查看某个函数或方法的帮助文档。

例如，要查看字符串变量（它的类型是 str 类型）所能调用的全部内容，可以在交互式解释器中输入如下命令：

```
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__',
'__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize',
'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>>
```

上面列出了字符串类型（str）提供的所有方法，其中以“_”开头、“_”结尾的方法被约定成私有方法，不希望被外部直接调用。

其实，本章中一直在说的“字符串变量”，只是为了方便初学者理解，严格意义上说是不对的，它应该是字符串 str 类的对象，由于涉及到了 Python 类的知识，后续章节才会介绍，这里不过多赘述，初学者只需要知道，以“_”开头、“_”结尾的方法不能以“字符串变量.方法”的形式调用就可以了。

如果希望查看某个方法的用法，则可使用 help() 函数。例如，在交互式解释器中输入如下命令：

```
>>> help(str.title)
Help on method_descriptor:

title(...)
    S.title() -> str

    Return a titlecased version of S, i.e. words start with title case
    characters, all remaining cased characters have lower case.

>>>
```

从上面介绍可以看出，title() 方法的使用形式是“str.title()”，其功能是将字符串中所有单词的首字母大写，其他所有字符全部改为小写。

通过使用 dir() 和 help() 函数，我们就可以查看字符串变量所能调用的所有方法，包括他们的使用方法和功能等。

第 6 章 Python 流程控制

Python 同样提供了现代编程语言都支持的两种基本流程控制结构，分支结构和循环结构：
分支结构用于实现根据条件来选择性地执行某段代码；
循环结构用于实现根据循环条件重复执行某段代码；

Python 使用 if 语句提供分支支持，提供了 while、 for-in 循环，也提供了 break 和 continue 控制程序的循环结构。

1.Python if else 条件语句

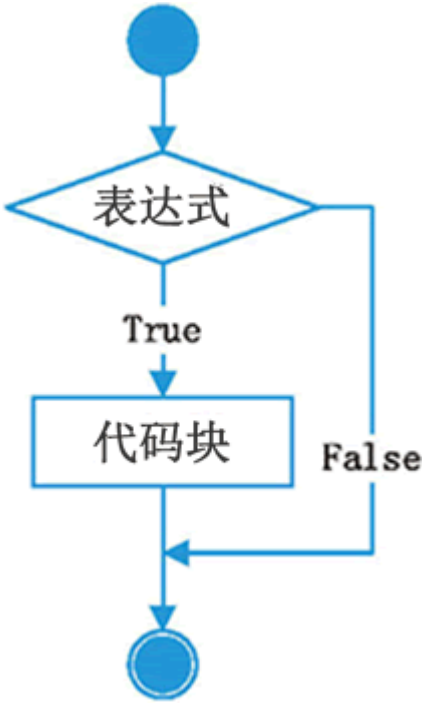
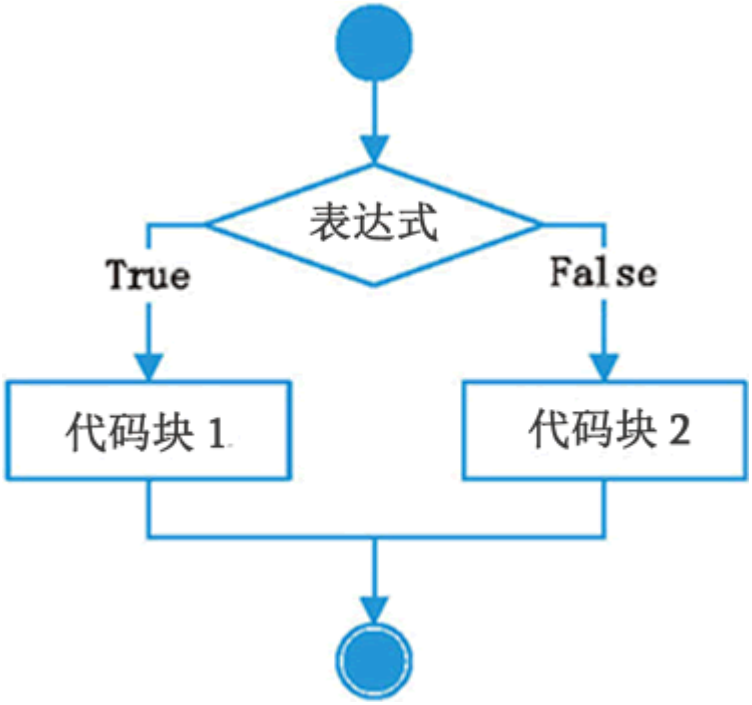
在前面章节的学习，我们所接触到的程序，都只是从第一行代码开始逐条执行，但这远远不能满足开发的需求，因为在很多场景中，我们需要选择性地执行一部分语句。

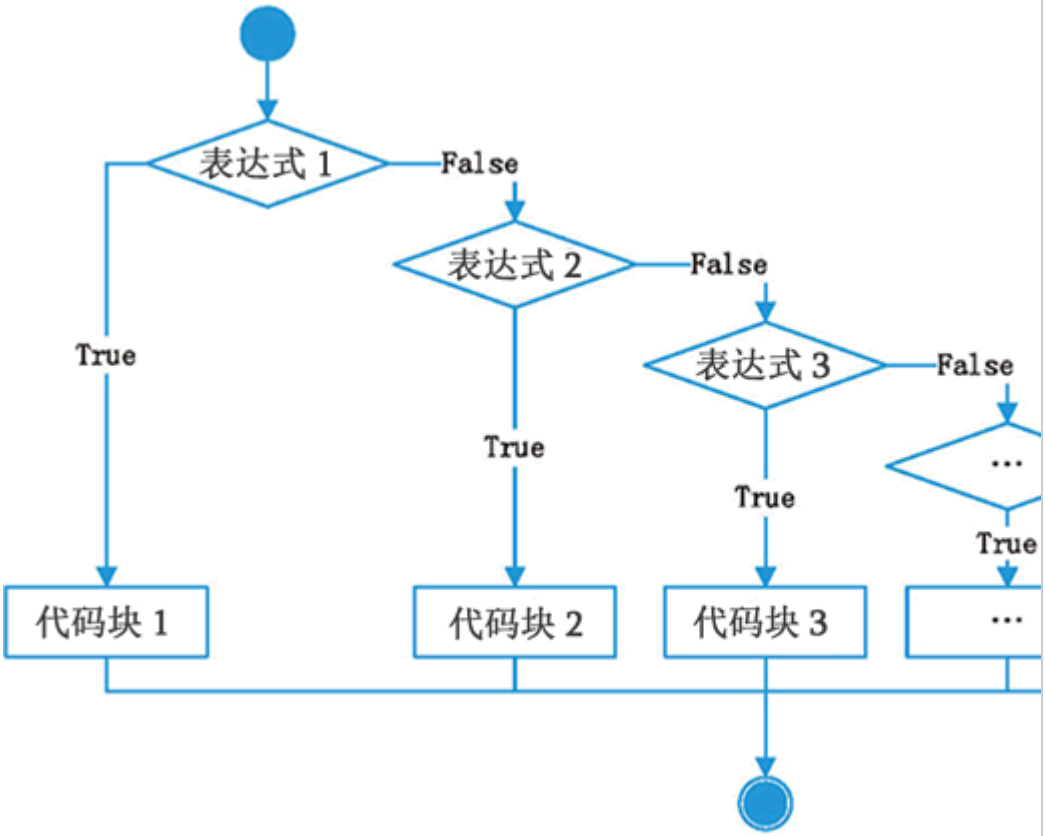
比如说，编程实现“只有当用户输入用户名和密码正确时，才能成功登陆系统；反之则登陆失败，无法进入系统”，要实现这样地功能，只凭借前面所学的知识是做不到的，需要使用本节所讲的选择（条件）语句才能实现。

Python 中，选择（条件）语句可细分为 3 种形式，分别是 if 语句、if else 语句和 if elif else 语句。这 3 种形式分支语句的语法格式和执行流程，如表 1 所示。

表 1 if 分支语句 3 种形式

语法	执行流程

格式	
if 表达式： 代码块	
if 表达式： 代码块 1 else： 代码块 2	

if 表达式 1： 代码块 1 elif 表达式 2： 代码块 2 elif 表达式 3： 代码块 3	

...//可以有零条或多条elif语句 else： 代码块n	

以上三种形式中，第二种形式和第三种形式是相通的，如果第三种形式中的 elif 块不出现，则变成了第二种形式。另外，elif 和 else 必须和 if 一起使用，不能单独使用。

以上 3 种形式的分支语句中，表达式可以是一个单纯的布尔值或变量，也可以是比较表达式或逻辑表达式（表达式真假值的判断，本节后续会详细介绍）；代码块指的是具有相同缩进格式的多行代码。一般情况下，一个代码块会被当成一个整体来执行（除非在运行过程中遇到 return、break、continue 等关键字），因此这个代码块也被称为条件执行体；

对于上面的 if 分支语句，执行过程是非常简单的，即如果 if 中的表达式为“真”，程序就

会执行 if 中的代码块；否则就会依次判断 elif 中的表达式（如果有），如果为“真”，程序就会执行 elif 中的代码块……如果前面所有条件都为“假”，程序就会执行 else 后的代码块（如果有）。

例如如下程序：

```
        s_age = input("请输入您的年龄:")
        age = int(s_age)
        #使用第一种形式的 if 分支结构
        if age > 20 :
            # 只有当 age > 20 时，下面用整体缩进的代码块才会执行
            # 整体缩进的语句是一个代码块
            print("年龄已经大于 20 岁了")
            print("20 岁以上的人应该学会承担责任...")
```

运行上面代码，如果输入年龄小于 20，将会看到如下运行结果：

```
请输入您的年龄：18
```

从上面代码可以看出，如果输入的年龄小于 20，则程序没有任何输出，代码快中的代码都不会执行。

运行上面代码，如果输入年龄大于 20，将会看到如下运行结果：

```
请输入您的年龄:24
年龄已经大于 20 岁了
20 岁以上的人应该学会承担责任...
```

从上面代码可以看出，如果输入的年龄大于 20 ，则程序会执行整体缩进的代码块。

再次强调，Python 是一门很“独特”的语言，它的代码块是通过缩进来标记的，具有相同缩进的多行代码属于同一个代码块。如果代码莫名其妙地乱缩进，Python 解释器会报错（详解内容可阅读《Python 缩进规则》一节）。有关 if else 分支语句的使用注意规范，可阅读 《Python if else 分支语句使用规范》一节。

> 在其他语言中（如 C、C++、Java 等），选择语句还包括 switch 语句，也可以实现多重选择，但是在 Python 中没有 switch 语句，所以当要实现多重选择的功能时，只能使用 if（elif else）分支语句。

if 表达式真假值的判断方法

从前面的示例可以看到，Python 执行 if 语句时，会判断 if 表达式的值是 True 还是 False 。那么是不是只能使用 bool 类型的表达式呢？

不是。表达式可以是任意类型，当下面的值作为 bool 表达式时，会被解释器当作 False 处理：

> False、None、0、""、()、[]、{}

从上面介绍可以看出，除了 False 本身，各种代表“空”的 None、空字符串、空元组、空列表、空字典都会被当成 False 处理。

例如如下代码：

```
s = ""
if s :
    print('s 不是空字符串')
else:
    print('s 是空字符串')
# 定义空列表
my_list = []
if my_list:
    print('my_list 不是空列表')
else:
    print('my_list 是空列表')
# 定义空字典
my_dict = {}
if my_dict:
    print('my_dict 不是空字典')
else:
    print('my_dict 是空字典')
```

从以上代码中可以看出，这些 if 表达式分别使用了字符串（str）类型、列表（list）类型、字典（dict）类型，由于这些 str、list、dict 都是空值，因此 Python 会把它们当成 False 处理。

2.Python if else 语句用法规范

在使用 if elif else 条件语句时，需要初学者根据实际应用场景，自行设计表达式和代码块，而在这两部分中，有些细节需要初学者注意。

1) 代码块不要忘记缩进

代码块一定要缩进，否则就不是代码块。例如如下程序：

```
s_age = input("请输入您的年龄:")
age = int(s_age)
```

```
3.          if age > 20 :
4.          print("年龄已经大于 20 岁了")
```

上面程序的 if 条件与下面的 print 语句位于同一条竖线上，这样在 if 条件下就没有受控制的代码块了。因此，上面程序执行时会报出如下错误：

```
IndentationError: expected an indented block
```

if 条件后的条件执行体（代码块）一定要缩进。只有缩进后的代码才能算条件执行体。

可能读者会产生一个疑问，条件执行体到底要缩进多少呢？这个随意。你可以缩进 1 个空格、2 个空格、3 个空格…… 或 1 个 Tab 位，这都是符合语法要求的。但从编程习惯来看，Python 通常建议缩进 4 个空格。

有些时候，Python 解释器不会报错，但并不代表程序没有错误。例如如下代码：

```
1.          s_age = input("请输入您的年龄:")
2.          age = int(s_age)
3.          if age > 20 :
4.              print("年龄已经大于 20 岁了")
5.          print("20 岁以上的人应该学会承担责任...")
```

解释执行上面程序，程序不会报任何错误。但如果输入一个小于 20 的年龄，则可看到如下运行过程：

```
请输入您的年龄:12
20 岁以上的人应该学会承担责任...
```

从运行过程可以看出，我们输入的年龄明明小于 20，但运行结果还是会打印“20 岁以上...”。

这是为什么呢？就是因为这条 print 语句没有缩进。如果这行代码不缩进，那么 Python 就不会把这行代码当成条件执行体，它就不受 if 条件的控制，因此无论用户输入什么年龄，print 语句总会执行。

如果忘记正确地缩进，很可能导致程序的运行结果超出我们的预期。例如如下程序：

```
#定义变量 b，并为其赋值
b = 5
if b > 4 :
    #如果 b>4，则执行下面的条件执行体，只有一行代码作为代码块
    print("b 大于 4")
else:
    #否则，执行下面的条件执行体，只有一行代码作为代码块
    b -= 1
#对于下面代码而言，它已经不再是条件执行体的一部分，因此总会执行
print("b 不大于 4")
```

上面代码中，最后一行代码总会执行，因为这行代码没有缩进，因此它就不属于 else 后的条件执行体，else 后的条件执行体只有 `b -= 1` 这一行代码。

如果要让 print("b 不大于 4") 语句也处于 else 控制之下，则需要让这行代码也缩进 4 个空格。

if、else、elif 后的条件执行体必须使用相同缩进的代码块，将这个代码块整体作为条件执行体。当 if 后有多条语句作为条件执行体时，如果忘记了缩进某一行代码，则会引起语法错误。看下面代码：

```
# 定义变量 c，并为其赋值
```

```
    c = 5
    if c > 4:
        # 如果 c>4，则执行下面的执行体，将只有 c--一行代码为执行体
        c -= 1
    # 下面是一行普通代码，不属于执行体
    print("c 大于 4")
    # 此处的 else 将没有 if 语句，因此编译出错
    else
        # 否则，执行下面的执行体，只有一行代码作为代码块
        print("c 不大于 4")
```

在上面代码中，因为 if 后的条件执行体的最后一条语句没有缩进，所以系统只把 `c -= 1` 一行代码作为条件执行体，当 `c -= 1` 语句执行结束后，if 语句也就执行结束了。后面的 print("c 大于 4") 已经是一行普通代码，不再属于条件执行体，从而导致 else 语句没有 if 语句，引发编译错误。

运行上面代码，将看到如下错误：

```
SyntaxError : invalid syntax
```

为了改正上面的代码，需要让 print("c 大于 4") 也缩进 4 个空格。

2) if 代码块不要随意缩进

需要说明的是，虽然 Python 语法允许代码块随意缩进 N 个空格，但同一个代码块内的代码必须保持相同的缩进，不能一会缩进 2 个空格，一会缩进 4 个空格。

例如如下代码：

```
    s_age = input("请输入您的年龄:")
```

```
2.          age = int(s_age)
3.          if age > 20 :
4.              print("年龄已经大于 20 岁了")
5.               print("20 岁以上的人应该学会承担责任...")
```

上面程序中第 2 个 print 语句缩进了 5 个空格，在这样的情况下，Python 解释器认为这条语句与前一条语句（缩进了 4 个空格）不是同一个代码块（这就是游标卡尺笑话的由来），因此 Python 解释器会报错。运行上面代码，将会报出如下错误：

```
IndentationError : unexpected indent
```

把代码改为如下形式：

```
1.          s_age = input("请输入您的年龄:")
2.          age = int(s_age)
3.          if age > 20 :
4.              print("年龄已经大于 20 岁了")
5.             print("20 岁以上的人应该学会承担责任...")
```

上面程序中第二条 print 语句只缩进了 3 个空格，它与前一条 print 语句（缩进了 4 个空格）同样不属于同一个代码块，因此 Python 解释器还是会报错。运行上面代码，则会报出如下错误：

```
IndentationError: unindent does not match any outer indentation level
```

通过上面介绍可以看出，Python 代码块中的所有语句必须保持相同的缩进，换句话说，位于同一个代码块中的所有语句必须保持相同的缩进，既不能多，也不能少。

另外，需要说明的是，对于不需要使用代码块的地方，千万不要随意缩进，否则程序也会报错。例如如下简单的程序：

```
1.          msg = "Hello, Charlie"
2.              print(msg)
```

上面程序只有两条简单的执行语句，并没有包括分支、循环等流程控制，因此不应该使用缩进。解释执行上面代码，将会看到如下错误：

```
Indentat ionError:unexpected indent
```

3) if 表达式不要遗忘冒号

从 Python 语法解释器的角度来看，Python 冒号精确表示代码块的开始点这个功能不仅在条件执行体中如此，后面的循环体、方法体、类体全部遵守该规则。

如果程序遗忘了冒号，那么 Python 解释器就无法识别代码块的开始点。例如如下程序：

```
1.          age = 24
2.          if age > 20
3.              print("年龄已经大于 20 岁了")
4.              print("20 岁以上的人应该学会承担责任...")
```

上面 if 条件后忘了写冒号，因此 Python 就不知道条件执行体的开始点。运行上面程序，将会报出如下错误：

```
SyntaxError : invalid syntax
```

上面介绍的有关代码块的知识，不仅适用于 if 分支的代码块，也适用于作为循环体的代码块等。后面我们还会见到大量的 Python 代码块缩进、代码块必须以冒号开头的示例。

3.Python if 语句嵌套

前面章节中，详细介绍了 3 种形式的条件语句，即 if、if else 和 if elif else，这 3 种条件语句之间可以相互嵌套。

例如，在最简单的 if 语句中嵌套 if else 语句，形式如下：

```
if 表达式 1：
    if 表示式 2：
        代码块 1
    else：
        代码块 2
```

再比如，在 if else 语句中嵌套 if else 语句，形式如下：

```
if 表示式 1：
    if 表达式 2：
        代码块 1
    else：
        代码块 2
else：
```

```
    if 表达式 3：
        代码块 3
    else：
        代码块 4
```

Python 中，if、if else 和 if elif else 之间可以相互嵌套。因此，在开发程序时，需要根据场景需要，选择合适的嵌套方案。需要注意的是，在相互嵌套时，一定要严格遵守不同级别代码块的缩进规范。

【实例】判断是否为酒后驾车

如果规定，车辆驾驶员的血液酒精含量小于 20mg/100ml 不构成酒驾；酒精含量大于或等于 20mg/100ml 为酒驾；酒精含量大于或等于 80mg/100ml 为醉驾。先编写 Python 程序判断是否为酒后驾车。

通过梳理思路，是否构成酒驾的界限值为 20mg/100ml；而在已确定为酒驾的范围（大于 20mg/100ml）中，是否构成醉驾的界限值为 80mg/100ml，整个代码执行流程应如图 1 所示。

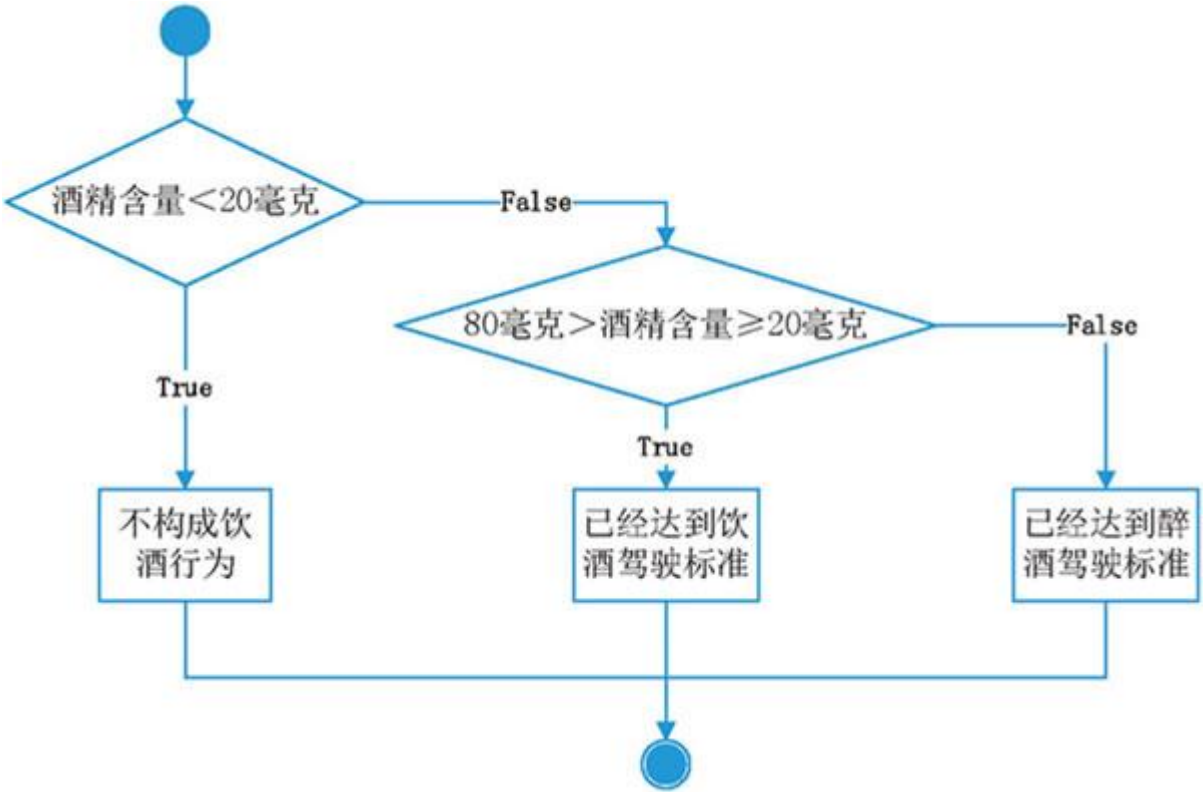


图 1 执行流程示意图

由此，我们可以使用两个 if else 语句嵌套来实现：

```
1.          proof = int(input("输入驾驶员每 100ml 血液酒精的含量："))
2.          if proof < 20:
3.              print("驾驶员不构成酒驾")
4.          else:
5.              if proof < 80:
6.                  print("驾驶员已构成酒驾")
7.              else:
8.                  print("驾驶员已构成醉驾")
```

运行结果为：

```
输入驾驶员每 100ml 血液酒精的含量：10

驾驶员不构成酒驾
```

当然，这个例题单独使用 if elif else 也可以实现，这里只是为了让初学者熟悉 if 分支嵌套的用法而已。

除此之外，if 分支结构中还可以嵌套循环结构，同样，循环结构中也可以嵌套分支结构。不过，由于目前尚未系统学习循环结构，因此这部分知识会放到后续章节中作详细讲解。

4.Python pass

很多程序都提供了“空语句”支持，Python 也不例外，Python 的 pass 语句就是空语句。

有时候程序需要占一个位、放一条语句，但又不希望这条语句做任何事情，此时就可通过 pass 语句来实现。通过使用 pass 语句，可以让程序更完整。

如下程序示范了 pass 作为空语句的用法：

```
1.          s = input("请输入一个整数: ")
2.          s = int(s)
3.          if s > 5:
4.              print("大于 5")
5.          elif s < 5:
6.              # 空语句，相当于占位符
7.              pass
8.          else:
9.              print("等于 5")
```

正如从上面程序所看到的 对于 s 小于 5 的情形 程序暂时不想处理(或不知道如何处理)，此时程序就需要通过空语句来占一个位，这样即可使用 pass 语句了。

5.Python assert 断言

assert 断言语句和 if 分支有点类似，它用于对一个 bool 表达式进行断言，如果该 bool 表达式为 True，该程序可以继续向下执行；否则程序会引发 AssertionError 错误。

有读者可能会问，明明 assert 会令程序崩溃，为什么还要使用它呢？这是因为，与其让程序在晚些时候崩溃，不如在错误条件出现时，就直接让程序崩溃。通常，assert 语句用在检查函数参数的属性（是参数是否是按照设想的要求传入），或者作为初期测试和调试过程中的辅助工具。

例如如下程序：

```
1.          s_age = input("请输入您的年龄:")
2.          age = int(s_age)
3.          assert 20 < age < 80
4.          print("您输入的年龄在 20 和 80 之间")
```

上面程序中，使用 asser 语句断言 age 必须位于 20 到 80 之间。运行上面程序，如果输入的 age 处于执行范围之内，则可看到如下运行过程：

```
请输入您的年龄:23
您输入的年龄在 20 和 80 之间
```

如果输入的 age 不处于 20 到 80 之间，将可以看到如下运行过程：

```
请输入您的年龄:1
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 3, in <module>
    assert 20 < age < 80
AssertionError
```

从上面的运行过程可以看出，断言也可以对逻辑表达式进行判断，因此实际上断言也相当于一种特殊的分支。

assert 断言的执行逻辑是：

```
if 表达式的值为 True：
    程序继续运行；
else：      # 表示式的值为 False
    程序引发 AssertionError 错误
```

6.如何合理使用 assert，千万不要和 if 混用！

讲完了 assert 的基本语法之后，本节通过一些实际应用的例子，给大家演示一下 assert 在 Python 中的用法，并明确 assert 的使用场景。

第一个例子，假设 C 语言中文网想做 VIP 促销活动，准备进行打折，现需要写一个 apply_discount() 函数，要求是，向该函数传入原来的价格和折扣力度，该函数返回打折

后的价格。

apply_discount() 大致应该写成如下这样：

```
1.          #price 为原价，discount 为折扣力度
2.          def apply_discount(price, discount):
3.              updated_price = price * (1 - discount)
4.              assert 0 <= updated_price <= price, '折扣价应在 0 和原价之间'
5.              return updated_price
```

可以看到，在计算新价格的后面，添加了一个 assert 语句，用来检查折后价格，这里要求新折扣价格必须大于等于 0、小于等于原来的价格，否则就抛出异常。

我们可以试着输入几组数，来验证一下这个功能：

```
1.          print(apply_discount(100,0.2))
2.          print(apply_discount(100,1.1))
```

运行结果为：

```
80.0
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 7, in <module>
    print(apply_discount(100,1.1))
  File "C:\Users\mengma\Desktop\demo.py", line 4, in apply_discount
    assert 0 <= updated_price <= price, '折扣价应在 0 和原价之间'
AssertionError: 折扣价应在 0 和原价之间
```

可以看到，当 discount 是 0.2 时，输出 80 没有问题，但是当 discount 为 1.1 时，程序便抛出下面 AssertionError 异常。

这样一来，如果开发人员修改相关的代码，或者是加入新的功能，导致 discount 数值异常时，只要运行程序就很容易能发现问题，这也从侧面印证了前面多讲的，assert 的加入可以有效预防程序漏洞，提高程序的健壮性。

另外，在实际工作中，assert 还有一些很常见的用法，例如：

```
        def func(input):
            assert isinstance(input, list), '输入内容必须是列表'
            # 下面的操作都是基于前提：input 必须是 list
            if len(input) == 1:
                ...
            elif len(input) == 2:
                ...
            else:
                ...
```

上面代码中，func() 函数中的所有操作都基于输入必须是列表这个前提。所以很有必要在开头加一句 assert 的检查，防止程序出错。

以上给大家介绍了 2 个有关 assert 的使用场景，很多读者可能觉得，assert 的作用和 if 语句非常接近，那么他们之间是否可以相互替代呢？

要注意，前面讲过，assert 的检查是可以被关闭的，比如在命令行模式下运行 Python 程序时，加入 -O 选项就可以使程序中的 assert 失效。一旦 assert 失效，其包含的语句也

就不会被执行。

还是拿 C 语言中文网用户来说，只有 VIP 用户才可以阅读 VIP 文章，我们可以设计如下这个函数来模式判断用户身份的功能：

```
1.          def login_user_identity(user_id):
2.              #凭借用户 id 判断该用户是否为 VIP 用户
3.              assert user_is_Vip(user_id) "用户必须是 VIP 用户，才能阅读 VIP 文章"
4.              read()
```

此代码从代码功能角度上看，并没有问题，但在实际场景中，基本上没人会这么写，因为一旦 assert 失效，则就造成任何用户都可以阅读 VIP 文章，这显然是不合理的。

所以正确的做法是，使用 if 条件语句替代 assert 语句进行相关的检查，并合理抛出异常：

```
1.          def login_user_identity(user_id):
2.              #凭借用户 id 判断该用户是否为 VIP 用户
3.              if not user_is_Vip(user_id):
4.                  raise Exception("用户必须是 VIP 用户，才能阅读 VIP 文章")
5.              read()
```

总之，不能滥用 assert，很多情况下，程序中出现的不同情况都是意料之中的，需要用不同的方案去处理，有时用条件语句进行判断更为合适，而对于程序中可能出现的一些异常，要记得用 try except 语句处理（后续章节会做详细介绍）。

7.Python while 循环语句

Python 中，while 循环和 if 条件分支语句类似，即在条件（表达式）为真的情况下，会执行相应的代码块。不同之处在于，只要条件为真，while 就会一直重复执行那段代码块。

while 语句的语法格式如下：

```
while 条件表达式：
    代码块
```

这里的代码块，还是指的缩进格式相同的多行代码，不过在循环结构中，它又称为循环体。

while 语句执行的具体流程为：首先判断条件表达式的值，其值为真（True）时，则执行代码块中的语句，当执行完毕后，再回过头来重新判断条件表达式的值是否为真，若仍为真，则继续重新执行代码块...如此循环，直到条件表达式的值为假（False），才终止循环。

while 循环结构的执行流程如图 1 所示。

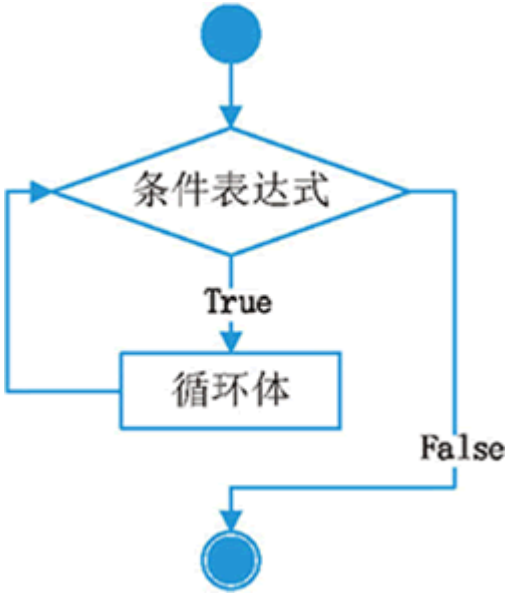


图 1 while 循环语句执行流程示意图

例如，打印 1~100 的所有数字，就可以使用 while 循环，实现代码如下：

```
        # 循环的初始化条件
        num = 1
        # 当 num 小于 100 时，会一直执行循环体
        while num < 100 :
            print("num=", num)
            # 迭代语句
            num += 1
        print("循环结束!")
```

运行程序会发现，程序只输出了 1~99，却没有输出 100。这是因为，当循环至 num 的值为 100 时，此时条件表达式为假（100<100），当然就不会再去执行代码块中的语句，因此不会输出 100。

注意，在使用 while 循环时，一定要保证循环条件有变成假的时候，否则这个循环将成为一个死循环，永远无法结束这个循环。

例如，将上面 while 循环中的 `num += 1` 代码注释掉再运行，你会发现，Python 解释器一直在输出"num= 1"，除非我们强制关闭解释器。

另外，与 if 分支语句类似的是，while 循环的循环体中所有代码必须使用相同的缩进，否则 Python 也会引发错误。例如如下程序：

```
        # 循环的初始化条件
        num = 1
        # 当 num 小于 100 时，会一直执行循环体
```

```
        while num < 100 :
            print("num=", num)
            # 迭代语句
        num += 1
        print("循环结束!")
```

运行上面程序，将会看到执行一个死循环。这是由于 `num += 1` 代码没有缩进，这行代码就不属于循环体。这样程序中的 num 将一直是 1，从而导致 1 < 100 一直都是 True，因此该循环就变成了一个死循环。

使用 while 循环遍历列表和元组

由于列表和元组的元素都是有索引的，因此程序可通过 while 循环、列表或元组的索引来遍历列表和元组中的所有元素。例如如下程序：

```
        a_tuple = ('fkit', 'crazyit', 'Charli')
        i = 0
        # 只有 i 小于 len(a_list)，继续执行循环体
        while i < len(a_tuple):
            print(a_tuple[i]) # 根据 i 来访问元组的元素
            i += 1
```

运行上面程序，可以看到如下输出结果：

```
fkit crazyit Charli
```

按照上面介绍的方法，while 循环也可用于遍历列表。

下面示范一个小程序，实现对一个整数列表的元素进行分类，能整除 3 的放入一个列表中；除以 3 余 1 的放入另一个列表中；除以 3 余 2 的放入第三个列表中：

纯文本复制

```
src_list = [12, 45, 34,13, 100, 24, 56, 74, 109]
a_list = [] # 定义保存整除 3 的元素
b_list = [] # 定义保存除以 3 余 1 的元素
c_list = [] # 定义保存除以 3 余 2 的元素
# 只要 src_list 还有元素，继续执行循环体
while len(src_list) > 0:
    # 弹出 src_list 最后一个元素
    ele = src_list.pop()
    # 如果 ele % 2 不等于 0
    if ele % 3 == 0 :
        a_list.append(ele) # 添加元素
    elif ele % 3 == 1:
        b_list.append(ele) # 添加元素
    else:
        c_list.append(ele) # 添加元素
print("整除 3:", a_list)
print("除以 3 余 1:",b_list)
print("除以 3 余 2:",c_list)
```

8.Python for 循环

Python 中的循环语句有 2 种，分别是 while 循环和 for 循环，前面章节已经对 while 做了详细的讲解，本节给大家介绍 for 循环，它常用于遍历字符串、列表、元组、字典、集合等序列类型，逐个获取序列中的各个元素。

for 循环的语法格式如下：

```
for 迭代变量 in 字符串|列表|元组|字典|集合：
    代码块
```

格式中，迭代变量用于存放从序列类型变量中读取出来的元素，所以一般不会在循环中对迭代变量手动赋值；代码块指的是具有相同缩进格式的多行代码（和 while 一样），由于和循环结构联用，因此代码块又称为循环体。

for 循环语句的执行流程如图 1 所示。

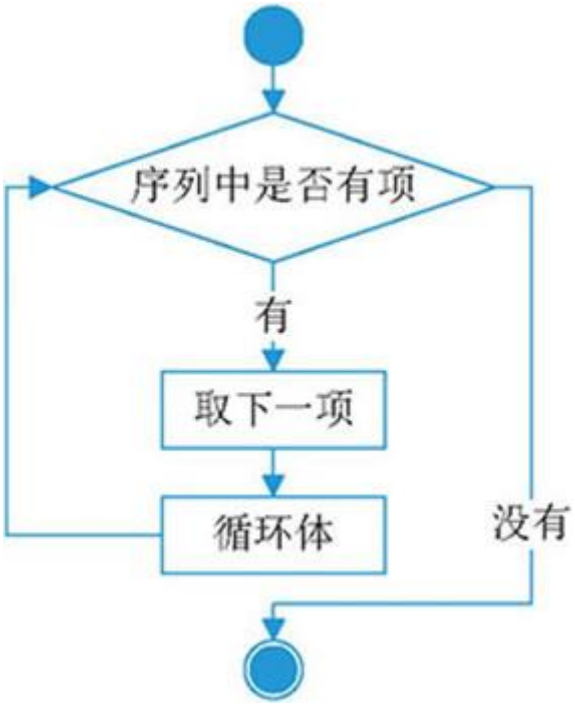


图 1 for 循环语句的执行流程图

例如：

```
1.          name = '张三'
2.          #变量 name，逐个输出各个字符
```

```
3.          for ch in name:
4.              print(ch)
```

运行结果为：

```
张
三
```

可以看到，使用 for 循环遍历 “张三” 字符串的过程中，迭代变量 ch 先后被赋值为‘张’和‘三’，并代入循环体中运行，只不过例子中的循环体比较简单，只有一行输出代码。

for 进行数值循环

在使用 for 循环时，最基本的应用就是进行数值循环。比如说，想要实现从 1 到 100 的累加，可以执行如下代码：

```
1.          print("计算 1+2+...+100 的结果为：")
2.          #保存累加结果的变量
3.          result = 0
4.          #逐个获取从 1 到 100 这些值，并做累加操作
5.          for i in range(101):
6.              result += i
7.          print(result)
```

运行结果为：

```
计算 1+2+...+100 的结果为：
5050
```

上面代码中，使用了 range() 函数，此函数是 Python 内置的函数，用于生成一系列连续

的整数，多用于 for 循环中。

range() 函数的语法格式如下：

```
range(start,end,step)
```

此函数中各参数的含义如下：

1. start：用于指定计数的起始值，如果省略不写，则默认从 0 开始。
2. end：用于指定计数的结束值（不包括此值），此参数不能省略。
3. step：用于指定步长，即两个数之间的间隔，如果省略，则默认步长为 1。

总之，在使用 range() 函数时，如果只有一个参数，则表示指定的是 end；如果有两个参数，则表示指定的是 start 和 end。

例如：

```
print("输出 10 以内的所有奇数：")
for i in range(1,10,2):
    print(i,end=' ')
```

运行结果为：

```
输出 10 以内的所有奇数：
1 3 5 7 9
```

在 Python 2.x 中，除提供 range() 函数外，还提供了一个 xrange() 函数，它可以解决 range() 函数不经意间耗掉所有可用内存的问题。但在 Python 3.x 中，已经将 xrange() 更名为 range() 函数，并删除了老的 xrange() 函数。

for 循环遍历列表和元组

在使用 for 循环遍历列表和元组时，列表或元组有几个元素，for 循环的循环体就执行几次，针对每个元素执行一次，迭代变量会依次被赋值为元素的值。

如下代码使用 for 循环遍历元组：

```
1.          a_tuple = ('crazyit', 'fkit', 'Charlie')
2.          for ele in a_tuple:
3.              print('当前元素是:', ele)
```

运行结果为：

```
当前元素是: crazyit
当前元素是: fkit
当前元素是: Charlie
```

当然，也可按上面方法来遍历列表。例如，下面程序要计算列表中所有数值元素的总和、平均值：

```
1.          src_list = [12, 45, 3.4, 13, 'a', 4, 56, 'crazyit', 109.5]
2.          my_sum = 0
3.          my_count = 0
```

```
        for ele in src_list:
            # 如果该元素是整数或浮点数
            if isinstance(ele, int) or isinstance(ele, float):
                print(ele)
                # 累加该元素
                my_sum += ele
                # 数值元素的个数加 1
                my_count += 1
        print('总和:', my_sum)
        print('平均数:', my_sum / my_count)
```

运行结果为：

```
12
45
3.4
13
4
56
109.5
总和: 242.9
平均数: 34.7
```

上面程序使用 for 循环遍历列表的元素，并对几何元素进行判断：只有当列表元素是数值（int、float）时，程序才会累加它们，这样就可以计算出列表中数值元素的总和。

不仅如此，程序中还使用了 Python 的 isinstance() 函数，该函数用于判断某个变量是否为指定类型的实例，其中前一个参数是要判断的变量，后一个参数是类型。我们可以在

Python 的交互式解释器中测试该函数的功能，例如如下运行过程：

```
>>> isinstance(2,int)
True
>>> isinstance('a',int)
False
>>> isinstance('a',str)
True
```

从上面的运行过程可以看出，使用 isinstance() 函数判断变量是否为指定类型非常方便、有效。

如果需要，for 循环也可根据索引来遍历列表或元组，即只要让迭代变量取 0 到列表长度的区间，就可通过该迭代变量访问列表元素。例如如下程序：

```
a_list = [330, 1.4, 50, 'fkit', -3.5]
# 遍历 0 到 len(a_list)的范围
for i in range(0, len(a_list)) :
    # 根据索引访问列表元素
    print("第%d 个元素是 %s" % (i , a_list[i]))
```

运行结果为：

```
第 0 个元素是 330
第 1 个元素是 1.4
第 2 个元素是 50
```

```
第 3 个元素是 fkit
第 4 个元素是 -3.5
```

for 循环遍历字典

使用 for 循环遍历字典其实也是通过遍历普通列表来实现的。前面在介绍字典时已经提到，字典包含了如下三个方法：

1. items()：返回字典中所有 key-value 对的列表。
2. keys()：返回字典中所有 key 的列表。
3. values()：返回字典中所有 value 的列表。

因此，如果要遍历字典，完全可以先调用字典的上面三个方法之一来获取字典的所有 key-value 对、所有 key、所有 value，再进行遍历。如下程序示范了使用 for 循环来遍历字典：

```
my_dict = {'语文': 89, '数学': 92, '英语': 80}
# 通过 items()方法遍历所有 key-value 对
# 由于 items 方法返回的列表元素是 key-value 对，因此要声明两个变量
for key, value in my_dict.items():
    print('key:', key)
    print('value:', value)
print('--------------')
# 通过 keys()方法遍历所有 key
for key in my_dict.keys():
    print('key:', key)
    # 在通过 key 获取 value
    print('value:', my_dict[key])
```

```
13.         print('-------------')
14.         # 通过 values()方法遍历所有 value
15.         for value in my_dict.values():
16.             print('value:', value)
```

运行结果为：

```
key: 语文

value: 89

key: 数学

value: 92

key: 英语

value: 80

------------

key: 语文

value: 89

key: 数学

value: 92

key: 英语

value: 80

------------

value: 89

value: 92

value: 80
```

上面程序通过三个 for 循环分别遍历了字典的所有 key-value 对、所有 key、所有 value。尤其是通过字典的 items() 遍历所有的 key-value 对时，由于 items() 方法返回的是字典中所有 key-value 对组成的列表，列表元素都是长度为 2 的元组，因此程序要声明两个变量来分别代表 key、value（这也是序列解包的应用）。

假如需要实现一个程序，用于统计列表中各元素出现的次数。由于我们并不清楚列表中包含多少个元素，因此考虑定义一个字典，以列表的元素为 key，该元素出现的次数为 value。程序如下：

纯文本复制

```
src_list = [12, 45, 3.4, 12, 'fkit', 45, 3.4, 'fkit', 45, 3.4]
statistics = {}
for ele in src_list:
    # 如果字典中包含 ele 代表的 key
    if ele in statistics:
        # 将 ele 元素代表出现次数加 1
        statistics[ele] += 1
    # 如果字典中不包含 ele 代表的 key，说明该元素还未出现过
    else:
        # 将 ele 元素代表出现次数设为 1
        statistics[ele] = 1
# 遍历 dict，打印出各元素的出现次数
for ele, count in statistics.items():
    print("%s 的出现次数为:%d" % (ele, count))
```

运行结果为：

```
12 的出现次数为:2
45 的出现次数为:3
```

```
3.4 的出现次数为:3

fkit 的出现次数为:2
```

9.Python 循环结构中 else 用法

Python 中，无论是 while 循环还是 for 循环，其后都可以紧跟着一个 else 代码块，它的作用是，当循环条件为 False 跳出循环时，程序会最先执行 else 代码块中的代码。

如下代码示范了为 while 循环定义 else 代码块：

```
1.          count_i = 0
2.          while count_i < 5:
3.              print('count_i 小于 5: ', count_i)
4.              count_i += 1
5.          else:
6.              print('count_i 大于或等于 5: ', count_i)
```

运行上面程序，可以看到如下输出结果：

```
count_i 小于 5:  0

count_i 小于 5:  1

count_i 小于 5:  2

count_i 小于 5:  3

count_i 小于 5:  4

count_i 大于或等于 5:  5
```

从上面的运行过程来看，当循环条件 count i < 5 变成 False 时，程序执行了 while 循环的 else 代码块。

简单来说，程序在结束循环之前，会先执行 else 代码块。从这个角度来看，else 代码块其实没有太大的价值，将 else 代码块直接放在循环体之外即可。也就是说，上面代码其实可改为如下形式：

```
        count_i = 0
        while count_i < 5:
            print('count_i 小于 5: ', count_i)
            count_i += 1
        print('count_i 大于或等于 5: ', count_i)
```

上面代码直接将 else 代码块放在 while 循环体之外，程序执行结果与使用 else 代码块的执行结果完全相同。

> 循环的 else 代码块是 Python 的一个很特殊的语法(其他编程语言通常不支持),else 代码块的主要作用是便于生成更优雅的 Python 代码。

for 循环同样可使用 else 代码块，当 for 循环把区间、元组或列表的所有元素遍历一次之后， for 循环会执行 else 代码块，在 else 代码块中，迭代变量的值依然等于最后一个元素的值。例如如下代码：

```
        a_list = [330, 1.4, 50, 'fkit', -3.5]
        for ele in a_list:
            print('元素: ', ele)
        else:
            # 访问循环计数器的值，依然等于最后一个元素的值
```

```
6.                 print('else 块: ', ele)
```

运行上面程序，可以看到如下输出结果：

```
元素:  330

元素:  1.4

元素:  50

元素:  fkit

元素:  -3.5

else 块:  -3.5
```

10.Python 循环嵌套

Python 程序中，如果把一个循环放在另一个循环体内，那么就可以形成循环嵌套。循环嵌套既可以是 for 循环嵌套 while 循环，也可以是 while 循环嵌套 for 循环，即各种类型的循环都可以作为外层循环，各种类型的循环也都可以作为内层循环。

当程序遇到循环嵌套时，如果外层循环的循环条件允许，则开始执行外层循环的循环体，而内层循环将被外层循环的循环体来执行（只是内层循环需要反复执行自己的循环体而已）。只有当内层循环执行结束且外层循环的循环体也执行结束时，才会通过判断外层循环的循环条件，决定是否再次开始执行外层循环的循环体。

根据上面分析，假设外层循环的循环次数为 n 次，内层循环的循环次数为 m 次，那么内层循环的循环体实际上需要执行 n × m 次。循环嵌套的执行流程图如图 1 所示：

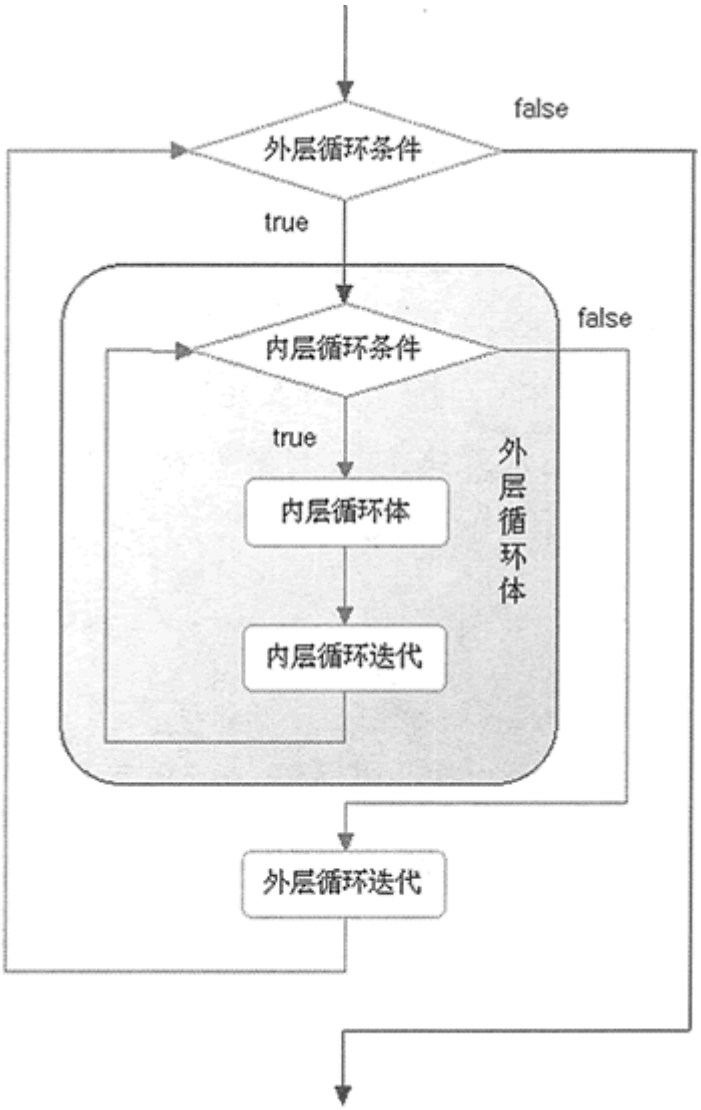


图 1 循环嵌套的执行流程图

从图 1 来看，循环嵌套就是把内层循环当成外层循环的循环体。只有内层循环的循环条件为假时，才会完全跳出内层循环，才可以结束外层循环的当次循环，开始下一次循环。
下面是一个循环嵌套的示例代码：

```
1.          # 外层循环
2.          for i in range(0, 5) :
3.              j = 0
4.              # 内层循环
5.              while j < 3 :
6.                  print("i 的值为: %d , j 的值为: %d" % (i, j))
7.                  j += 1
```

运行上面程序，将看到如下运行结果：

```
i 的值为: 0 , j 的值为: 0

i 的值为: 0 , j 的值为: 1

i 的值为: 0 , j 的值为: 2

.....
```

从上面的运行结果可以看出，当进入嵌套循环时，循环变量 i 开始为 0，这时即进入了外层循环。当进入外层循环后，内层循环把 i 当成一个普通变量，其值为 0。在外层循环的当次循环中，内层循环就是一个普通循环。

实际上，嵌套循环不仅可以是两层嵌套，还可以是三层嵌套、四层嵌套……不论循环如何嵌套，都可以把内层循环当成外层循环的循环体来对待，区别只是这个循环体中包含了需要反复执行的代码。

11.Python break：跳出当前循环体

我们知道，在执行 while 循环或者 for 循环时，只要循环条件满足，程序将会一直执行循环体，不停地转圈。但在某些场景，我们可能希望在循环结束前就手动离开循环，Python 提供了 2 种强制离开当前循环体的办法：

1. 使用 continue 语句，可以跳过执行本次循环体中剩余的代码，转而执行下一次的循环。
2. 只用 break 语句，可以完全终止当前循环。

本节先讲解 break 的用法，continue 语句放到下节做详细介绍。

在某些场景中，如果需要在某种条件出现时强行中止循环，而不是等到循环条件为 False 时才退出循环，就可以使用 break 来完成这个功能。

break 用于完全结束一个循环 ,跳出循环体。不管是哪种循环 ,一旦在循环体中遇到 break，系统就将完全结束该循环 ,开始执行循环之后的代码。这就好比在操场上跑步 ,原计划跑 10 圈，可是当跑到第 2 圈的时候，突然想起有急事要办，于是果断停止跑步并离开操场，这就相当于使用了 break 语句提前终止了循环。

> break 语句一般会结合 if 语句进行搭配使用，表示在某种条件下，跳出循环体，如果使用嵌套循环，break 语句将跳出当前的循环体。

break 语句的语法非常简单，只需要在相应 while 或 for 语句中直接加入即可。例如如下程序：

```
# 一个简单的 for 循环
for i in range(0, 10) :
    print("i 的值是: ", i)
    if i == 2 :
        # 执行该语句时将结束循环
        break
```

运行结果为：

```
i 的值是:  0
i 的值是:  1
i 的值是:  2
```

通过运行结果可以看出，break 语句导致了 i==2 时执行结束，因为当 i==2 时，在循环体内遇到了 break 语句，程序跳出该循环。

需要注意的是，对于带 else 块的 for 循环，如果使用 break 强行中止循环，程序将不会执行 else 块。例如如下程序：

```
# 一个简单的 for 循环
for i in range(0, 10) :
    print("i 的值是: ", i)
    if i == 2 :
        # 执行该语句时将结束循环
        break
else:
    print('else 块: ', i)
```

上面程序同样会在 i==2 时跳出循环，而且此时 for 循环不会执行 else 块。

在使用 break 语句的情况下，循环的 else 代码块与直接放在循环体后是有区别的，即如果将代码块放在 else 块中，当程序使用 break 中止循环时，循环不会执行 else 块；如果将代码块直接放在循环体后面，当程序使用 break 中止循环时，程序自然会执行循环体之后的代码块。

另外，针对嵌套的循环结构来说，Python 的 break 语句只能结束其所在的循环体，而无法结束嵌套所在循环的外层循环。例如：

```
for i in range(0,4) :
    print("此时 i 的值为: ",i)
    for j in range(5):
        print("    此时 j 的值为:",j)
        break
    print("跳出内层循环")
```

运行结果为：

```
此时 i 的值为： 0
    此时 j 的值为: 0
跳出内层循环
此时 i 的值为： 1
    此时 j 的值为: 0
跳出内层循环
```

```
此时 i 的值为： 2
    此时 j 的值为: 0
跳出内层循环
此时 i 的值为： 3
    此时 j 的值为: 0
跳出内层循环
```

分析运行结果不难看出，每次执行内层循环体时，第一次循环就会遇到 break 语句，即做跳出所在循环体的操作，转而执行外层循环体的代码。

如果想达到 break 语句不仅跳出单前所在循环，同时跳出外层循环的目的，可先定义 bool 类型的变量来标志是否需要跳出外层循环，然后在内层循环、外层循环中分别使用两条 break 语句来实现。例如如下程序：

```
1.          exit_flag = False
2.          # 外层循环
3.          for i in range(0, 5) :
4.              # 内层循环
5.              for j in range(0, 3 ) :
6.                  print("i 的值为: %d, j 的值为: %d" % (i, j))
7.                  if j == 1 :
8.                      exit_flag = True
9.                      # 跳出里层循环
10.                     break
11.             # 如果 exit_flag 为 True，跳出外层循环
12.             if exit_flag :
13.                 break
```

上面程序在内层循环中判断 j 是否等于 i，当 j 等于 i 时，程序将 exit_flag 设为 True，并跳出内层循环；接下来程序开始执行外层循环的剩下语句，由于 exit_flag 为 True，因此也会执行外层循环的 break 语句来跳出外层循环。

运行上面程序，将看到如下运行结果：

```
i 的值为: 0, j 的值为: 0

i 的值为: 0, j 的值为: 1
```

程序从外层循环进入内层循环后，当 j 等于 i 时，程序将 exit_flag 设为 True，并跳出内层循环；接下来程序又执行外层循环的 break 语句，从而跳出外层循环。

12.Python continue：直接执行下次循环

和 break 语句相比，continue 语句的作用则没有那么强大，它只能终止本次循环而继续执行下一次循环。

仍然以在操作跑步为例，原计划跑 10 圈，但是当跑到第 2 圈一半的时候，突然接到一个电话，停止了跑步。等挂断电话后，回到起点直接从第 3 圈继续跑。

continue 语句的用法和 break 语句一样，只要 while 或 for 语句中的相应位置加入即可。例如：

```
1.          # 一个简单的 for 循环
2.          for i in range(0, 3 ) :
3.              print("i 的值是: ", i)
4.              if i == 1 :
5.                  # 忽略本次循环的剩下语句
6.                  continue
7.              print("continue 后的输出语句")
```

运行上面程序，将看到如下运行结果：

```
i 的值是:  0

continue 后的输出语句
```

```
i 的值是:  1

i 的值是:  2

continue 后的输出语句
```

从上面的运行结果来看，当 i==1 时，程序没有输出“continue 后的输出语句”字符串，因为程序执行到 continue 时，忽略了当次循环中 continue 语句后的代码。从这个意义上看，如果把一条 continue 语句放在当次循环的最后一行，那么这条 continue 语句是没有任何意义的，因为它仅仅忽略了一片空白，没有忽略任何程序语句。

13.教你一招，彻底告别死（无限）循环！

要知道，每个循环结构（while 或 for）都必须有停止运行的途径，这样才不会没完没了地执行下去。

例如，下面的循环从 1 数到 5：

```
1.          x = 1
2.          while x <= 5:
3.              print(x)
4.              x += 1
```

运行结果为：

```
1

2

3
```

```
4

5
```

但如果像下面这样不小心遗漏了代码行 x += 1，那么这个循环将没完没了地运行：

```
1.          #这个循环将没完没了地运行
2.          x = 1
3.          while x <= 5:
4.              print(x)
```

在这里，x 的初始值为 1，但根本不会变，因此条件测试 x <= 5 始终为 True，导致 while 循环没完没了地打印 1，运行结果如下所示：

```
1

1

1

1

--snip--
```

每个程序员都会偶尔因为不小心而编写出无限（死）循环，在循环的退出条件比较微妙时尤其如此。

如果程序陷入无限循环，可按组合键 Ctrl+C，也可关闭显示程序输出的终端窗口，如果前 2 种方式都无效，比如有些编辑器（如 Sublime Text）内嵌了输出窗口，这可能导致难以结束无限循环，因此不得不关闭编辑器来结束无限循环。

读者可能会问，怎样才能有效避免编写出包含无限循环的程序呢？

要避免编写无限（死）循环，应务必对每个 while 循环（或 for 循环）进行测试，确保它按预期那样结束。如果希望程序在用户输入特定值时结束，可运行程序并输入这样的值，如果此时程序没有结束，则说明该循环结构会无限循环，应检查程序处理这个值的方式。

总之，解决无限（死）循环的方法只有一个，即确认循环结构中至少有一个位置能让循环条件为 False 或让 break 语句得以执行。

14.Python 推导式，快速初始化各种序列！

推导式（又称解析器），是 Python 独有的一种特性。使用推导式可以快速生成列表、元组、字典以及集合类型的数据，因此推导式又可细分为列表推导式、元组推导式、字典推导式以及集合推导式。

Python 列表推导式

列表推导式可以利用 range 区间、元组、列表、字典和集合等数据类型，快速生成一个满足指定需求的列表。

列表推导式的语法格式如下：

```
[表达式 for 迭代变量 in 可迭代对象 [if 条件表达式] ]
```

此格式中，[if 条件表达式] 不是必须的，可以使用，也可以省略。

通过列表推导式的语法格式，明显会感觉到它和 for 循环存在某些关联。其实，除去 [if 条件表达式] 部分，其余各部分的含义以及执行顺序和 for 循环是完全一样的（表达式其实就是 for 循环中的循环体），即它的执行顺序如下所示：

```
for 迭代变量 in 可迭代对象
    表达式
```

初学者可以这样认为，它只是对 for 循环语句的格式做了一下简单的变形，并用 [] 括起来而已，只不过最大的不同之处在于，列表推导式最终会将循环过程中，计算表达式得到的一系列值组成一个列表。

例如如下代码（程序一）：

```
1.          a_range = range(10)
2.          # 对 a_range 执行 for 表达式
3.          a_list = [x * x for x in a_range]
4.          # a_list 集合包含 10 个元素
5.          print(a_list)
```

上面代码的第 3 行会对 a_range 执行迭代，由于 a_range 相当于包含 10 个元素，因此程序生成的 a_list 同样包含 10 个元素，且每个元素都是 a_range 中每个元素的平方（由表达式 x*x 控制）。

运行上面代码，可以看到如下输出结果：

```
[0 , 1 , 4 , 9 , 16 , 25 , 36 , 49 , 64, 81]
```

不仅如此，我们还可以在列表推导式中添加 if 条件语句，这样列表推导式将只迭代那些符合条件的元素。例如如下代码：

```
1.          b_list = [x * x for x in a_range if x % 2 == 0]
2.          # a_list 集合包含 5 个元素
3.          print(b_list)
```

第一行代码与程序一中第 3 行代码大致相同，只是为这里给列表推导式增加了 if 条件语句 ,这会导致推导式只处理 range 区间的偶数 ,因此程序生成的 b_list 只包含 5 个元素。

运行上面代码，可以看到如下输出结果：

```
[0 ,4 , 16, 36, 64]
```

另外，以上所看到的列表推导式都只有一个循环，实际上它可使用多个循环，就像嵌套循环一样。例如如下代码：

```
1.          d_list = [(x, y) for x in range(5) for y in range(4)]
2.          # d_list 列表包含 20 个元素
3.          print(d_list)
```

上面代码中，x 是遍历 range(5) 的迭代变量（计数器），因此该 x 可迭代 5 次；y 是遍历 range(4) 的计数器 ,因此该 y 可迭代 4 次。因此 ,该(x,y)表达式一共会迭代 20 次。上面的 for 表达式相当于如下嵌套循环：

```
dd_list = []
for x in range(5):
    for y in range(4):
        dd_list.append((x, y))
```

运行上面代码，可以看到如下输出结果：

```
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3), (2, 0), (2, 1), (2, 2), (2, 3), (3, 0), (3, 1), (3, 2), (3, 3), (4, 0), (4, 1), (4, 2), (4, 3)]
```

当然，也支持类似于三层嵌套的 for 表达式，例如如下代码：

```
e_list = [[x, y, z] for x in range(5) for y in range(4) for z in range(6)]
# e_list 列表包含 120 个元素
print(e_list)
```

对于包含多个循环的 for 表达式，同样可指定 if 条件。假如我们有一个需求：程序要将两个列表中的数值按“能否整除”的关系配对在一起。比如 src_a 列表中包含 30，src_b 列表中包含 5，其中 30 可以整除 5，那么就将 30 和 5 配对在一起。对于上面的需求使用 for 表达式来实现非常简单，例如如下代码：

```
src_a = [30, 12, 66, 34, 39, 78, 36, 57, 121]
src_b = [3, 5, 7, 11]
# 只要 y 能整除 x，就将它们配对在一起
result = [(x, y) for x in src_b for y in src_a if y % x == 0]
print(result)
```

运行上面代码，可以看到如下输出结果：

```
[(3, 30), (3, 12), (3, 66), (3, 39), (3, 78), (3, 36), (3, 57), (5, 30), (11, 66), (11, 121)]
```

Python 元组推导式

元组推导式可以利用 range 区间、元组、列表、字典和集合等数据类型，快速生成一个满足指定需求的元组。

元组推导式的语法格式如下：

```
(表达式 for 迭代变量 in 可迭代对象 [if 条件表达式] )
```

其中，用 [] 括起来的部分，可以使用，也可以省略。

通过和列表推导式做对比，你会发现，除了元组推导式是用 () 圆括号将各部分括起来，而列表推导式用的是 []，其它完全相同。不仅如此，元组推导式和列表推导式的用法也完全相同。

例如，我们可以使用下面的代码生成一个包含数字 1~9 的元组：

```
1.          a = (x for x in range(1,10))
2.          print(a)
```

运行结果为：

```
<generator object <genexpr> at 0x0000020BAD136620>
```

从上面的执行结果可以看出，使用元组推导式生成的结果并不是一个元组，而是一个生成器对象（后续会介绍），这一点和列表推导式是不同的。

如果我们想要使用元组推导式获得新元组或新元组中的元素，有以下三种方式：

1. 使用 tuple() 函数，可以直接将生成器对象转换成元组，例如：

```
a = (x for x in range(1,10))
print(tuple(a))
运行结果为：
(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

2. 直接使用 for 循环遍历生成器对象，可以获得各个元素，例如：

```
a = (x for x in range(1,10))
for i in a:
    print(i,end=' ')
print(tuple(a))
```

运行结果为：

```
1 2 3 4 5 6 7 8 9 ()
```

3. 使用 __next__() 方法遍历生成器对象，也可以获得各个元素，例如：

```
a = (x for x in range(3))
print(a.__next__())
print(a.__next__())
print(a.__next__())
a = tuple(a)
print("转换后的元组：",a)
```

运行结果为：

```
0

1

2

转换后的元组： ()
```

注意，无论是使用 for 循环遍历生成器对象，还是使用 __next__() 方法遍历生成器对象，遍历后原生成器对象将不复存在，这就是遍历后转换原生成器对象却得到空元组的原因。

Python 字典推导式

Python 中，使用字典推导式可以借助列表、元组、字典、集合以及 range 区间，快速生成符合需求的字典。

字典推导式的语法格式如下：

```
{表达式 for 迭代变量 in 可迭代对象 [if 条件表达式]}
```

其中，用 [] 括起来的部分，可以使用，也可以省略。

可以看到，和其它推导式的语法格式相比，唯一不同在于，字典推导式用的是大括号{}。

【例 1】

```
1.          listdemo = ['C 语言中文网','c.biancheng.net']
2.          #将列表中各字符串值为键，各字符串的长度为值，组成键值对
```

```
3.          newdict = {key:len(key) for key in listdemo}
4.          print(newdict)
```

运行结果为：

```
{'C 语言中文网': 6, 'c.biancheng.net': 15}
```

【例 2】交换现有字典中各键值对的键和值。

```
1.          olddict={'C 语言中文网': 6, 'c.biancheng.net': 15}
2.          newdict = {v: k for k, v in olddict.items()}
3.          print(newdict)
```

运行结果为：

```
{6: 'C 语言中文网', 15: 'c.biancheng.net'}
```

【例 3】使用 if 表达式筛选符合条件的键值对。

```
1.          olddict={'C 语言中文网': 6, 'c.biancheng.net': 15}
2.          newdict = {v: k for k, v in olddict.items() if v>10}
3.          print(newdict)
```

运行结果为：

```
{15: 'c.biancheng.net'}
```

Python 集合推导式

Python 中，使用集合推导式可以借助列表、元组、字典、集合以及 range 区间，快速生

成符合需求的集合。

集合推导式的语法格式和字典推导式完全相同，如下所示：

```
{ 表达式 for 迭代变量 in 可迭代对象 [if 条件表达式] }
```

其中，用 [] 括起来的部分，可以使用，也可以省略。

有读者可能会问，集合推导式和字典推导式的格式完全相同，那么给定一个类似的推导式，如何判断是哪种推导式呢？最简单直接的方式，就是根据表达式进行判断，如果表达式以键值对（key：value）的形式，则证明此推导式是字典推导式；反之，则是集合推导式。

【例 1】

```
1.          setnew = {i**2 for i in range(3)}
2.          print(setnew)
```

运行结果为：

```
{0, 1, 4}
```

【例 2】既然生成的是集合，那么其保存的元素必须是唯一的。

```
1.          tupledemo = (1,1,2,3,4,5,6,6)
2.          setnew = {x**2 for x in tupledemo if x%2==0}
3.          print(setnew)
```

运行结果为:

```
{16, 4, 36}
```

【例 3】

纯文本复制

```
dictdemo = {'1':1,'2':2,'3':3}
setnew = {x for x in dictdemo.keys()}
print(setnew)
```

运行结果为：

```
{'2', '1', '3'}
```

15.Python zip 函数

zip() 函数可以把两个列表“压缩”成一个 zip 对象（可迭代对象），这样就可以使用一个循环并行遍历两个列表。为了测试 zip() 函数的功能，我们可以先在交互式解释器中“试验”一下该函数的功能。

```
>>> a = ['a','b','c']
>>> b = [1, 2, 3]
>>> [x for x in zip(a,b)]
[('a', 1), ('b', 2), ('c', 3)]
```

从上面的测试结果来看，zip() 函数压缩得到的可迭代对象所包含的元素是由原列表元素组成的元组。

Pthon 2.x 的 zip() 函数直接返回列表，而不是返回 zip 对象。Python 2.x 的 zip() 函数返回的列表所包含的元素和 Python 3.x 的 zip() 返回的 zip 对象所包含的元素相同。

例如：

```
>>> c = [0.1 ,0.2]
>>> [x for x in zip(a,c)]
[('a', 0.1), ('b', 0.2)]
```

从上面代码可以看出，如果 zip() 函数压缩的两个列表长度不相等，那么 zip() 函数将以长度更短的列表为准。

zip() 函数不仅可以压缩两个列表，也可以压缩多个列表。比如下面试验同时压缩 3 个列表：

```
>>> [x for x in zip(a, b, c)]
[('a', 1, 0.1), ('b', 2, 0.2)]
```

从上面代码可以看出，如果使用 zip() 函数压缩 N 个列表，那么 zip() 函数返回的可迭代对象的元素就是长度为 N 的元组。

下面代码示范了使用 zip() 函数来实现并行遍历的效果：

```
books = ['疯狂 Kotlin 讲义', '疯狂 Swift 讲义', '疯狂 Python 讲义']
prices = [79, 69, 89]
# 使用 zip()函数压缩两个列表，从而实现并行遍历
for book, price in zip(books, prices):
    print("%s 的价格是: %5.2f" % (book, price))
```

16.Python reversed 函数

有些时候，程序需要进行反向遍历，此时可通过 reversed() 函数，该函数可接收各种序列（元组、列表、区间等）参数，然后返回一个“反序排列”的法代器，该函数对参数本身不会产生任何影响。

在交互式解释器中，测试该函数的过程如下：

```
>>> a = range(10)
>>> [x for x in reversed(a)]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

从上面代码可以看出，通过 reversed() 函数得到了 range(10) 的反转序列；但如果再次访问 a，将会看到 a 并没有发生改变：

```
>>> a
range(0, 10)
```

reversed() 当然也可以对列表、元组进行反转。例如如下测试代码：

```
>>> b = ['a', 'fkit', 20, 3.4, 50]
>>> [x for x in reversed(b)]
[50, 3.4, 20, 'fkit', 'a']
```

前面提到过，str 其实也是序列，因此也可通过该函数实现在不影响字符串本身的前提下，对字符串进行反序遍历。例如如下测试代码：

```
>>> c = 'Hello,Charlie'
>>> [x for x in reversed(c)]
['e', 'i', 'l', 'r', 'a', 'h', 'C', ',', 'o', 'l', 'l', 'e', 'H']
```

17.Python sorted 函数

sorted() 函数与 reversed() 函数类似，该函数接收一个可迭代对象作为参数，返回一个对元素排序的列表。

在交互式解释器中测试该函数，可以看到如下运行过程：

```
>>> a = [20, 30, -1.2, 3.5, 90, 3.6]
>>> sorted(a)
[-1.2, 3.5, 3.6, 20, 30, 90]
>>> a
[20, 30, -1.2, 3.5, 90, 3.6]
```

从上面的运行过程来看，sorted() 函数也不会改变所传入的可迭代对象，该函数只是返回一个新的、排序好的列表。

在使用 sorted() 函数时，还可传入一个 reverse 参数，如果将该参数设置为 True，则表示反向排序。例如如下测试过程：

```
>>> sorted(a, reverse = True)
[90, 30, 20, 3.6, 3.5, -1.2]
```

在调用 sorted() 函数时，还可传入一个 key 参数，该参数可指定一个函数来生成排序的关键值。比如希望根据字符串长度排序，则可为 key 参数传入 len 函数。例如如下运行过程：

```
>>> b = ['fkit', 'crazyit', 'charlie', 'fox', 'Emily']
>>> sorted(b, key = len)
['fox', 'fkit', 'Emily', 'crazyit', 'charlie']
```

通过 sorted() 函数的帮助，程序可对可迭代对象按照由小到大的顺序进行遍历。例如如下程序：

```
1.          my_list = ['fkit', 'crazyit', 'Charlie', 'fox', 'Emily']
2.          for s in sorted(my_list, key=len):
3.              print(s)
```

18.Python 项目实战之数字转人民币读法

本节实现在实际开发中常用的一个工具函数：将一个浮点数转换成人民币读法的字符串，这个程序需要使用数组。

实现这个函数的思路是，首先把这个浮点数分成整数部分和小数部分。提取整数部分很容易，直接将这个浮点数强制类型转换成一个整数即可，这个整数就是浮点数的整数部分；再使用浮点数减去整数就可以得到这个浮点数的小数部分。

然后分开处理整数部分和小数部分。小数部分的处理比较简单，直接截断保留 2 位数字，转换成几角几分的字符串。整数部分的处理则稍微复杂一点，但只要认真分析不难发现，中国的数字习惯是 4 位一节的，一个 4 位的数字可被转成几千几百几十几，至于后面添加什么单位则不确定：

- 如果这节 4 位数字出现在 1~4 位，则后面添加单位“元”；
- 如果这节 4 位数字出现在 5~8 位，则后面添加单位“万”；
- 如果这节 4 位数字出现在 9~12 位，则后面添加单位“亿”；

> 多于 12 位就暂不考虑了。

因此，实现这个程序的关键就是把一个 4 位的数字字符串转换成中文读法。下面程序把这个需求实现了一部分：

```
1.          '''
2.              把一个浮点数分解成整数部分和小数部分字符串
3.              num 需要被分解的浮点数
```

```
4.              返回分解出来的整数部分和小数部分。
5.              第一个数组元素是整数部分，第二个数组元素是小数部分
6.          '''
7.          def divide(num):
8.              # 将一个浮点数强制类型转换为 int 型，即得到它的整数部分
9.              integer = int(num)
10.             # 浮点数减去整数部分，得到小数部分，小数部分乘以 100 后再取整得到 2
    位小数
11.             fraction = round((num - integer) * 100)
12.             # 下面把整数转换为字符串
13.             return (str(integer), str(fraction))
14.
15.         han_list = ["零", "壹", "贰", "叁", "肆",\
16.             "伍", "陆", "柒", "捌", "玖"]
17.         unit_list = ["十", "百", "千"]
18.         '''
19.           把一个四位的数字字符串变成汉字字符串
20.           num_str 需要被转换的四位的数字字符串
21.           返回四位的数字字符串被转换成汉字字符串
22.         '''
23.         def four_to_hanstr(num_str):
24.             result = ""
25.             num_len = len(num_str)
26.             # 依次遍历数字字符串的每一位数字
27.             for i in range(num_len) :
28.                 # 把字符串转成数值
29.                 num = int(num_str[i])
30.                 # 如果不是最后一位数字，而且数字不是零，则需要添加单位（千、百、
    十）
31.                 if i != num_len - 1 and num != 0 :
32.                     result += han_list[num] + unit_list[num_len - 2 - i]
33.                 # 否则不要添加单位
34.                 else :
35.                     result += han_list[num]
36.             return result
37.         '''
```

```
        把数字字符串变成汉字字符串
        num_str 需要被转换的数字字符串
        返回数字字符串被转换成汉字字符串
    '''
    def integer_to_str(num_str):
        str_len = len(num_str)
        if str_len > 12 :
            print('数字太大，翻译不了')
            return
        # 如果大于 8 位，包含单位亿
        elif str_len > 8:
            return four_to_hanstr(num_str[:-8]) + "亿" + \
                four_to_hanstr(num_str[-8: -4]) + "万" + \
                four_to_hanstr(num_str[-4:])
        # 如果大于 4 位，包含单位万
        elif str_len > 4:
            return four_to_hanstr(num_str[:-4]) + "万" + \
                four_to_hanstr(num_str[-4:])
        else:
            return four_to_hanstr(num_str)
    num = float(input("请输入一个浮点数: "))
    # 测试把一个浮点数分解成整数部分和小数部分
    integer, fraction = divide(num)
    # 测试把一个四位的数字字符串变成汉字字符串
    print(integer_to_str(integer))
    print(fraction)
```

运行上面程序，将看到如下运行结果：

```
请输入一个浮点数: 361092004.456
叁亿陆千壹百零玖万贰千零零肆
46
```

从上面程序的运行结果来看，初步实现了所需功能，但这个程序并不是这么简单的，对零的

处理比较复杂。例如，有两个零连在一起时该如何处理呢？还有小数部分如何翻译？因此，这个程序还需要继续完善，有兴趣的读者可自行完善此程序。

19.Python 项目实战之绕圈圈面试题

下面是来自某知名公司的一道“面试题”：给定 4，应该输出如下形式的数据：

```
01 12 11 10

02 13 16 09

03 14 15 08

04 05 06 07
```

给定 5，应该输出如下形式的数据：

```
01 16 15 14 13

02 17 24 23 12

03 18 25 22 11

04 19 20 21 10

05 06 07 08 09
```

仔细观察上面的试题，不难发现程序就是“绕圈圈”填入整数，如图 1 所示：

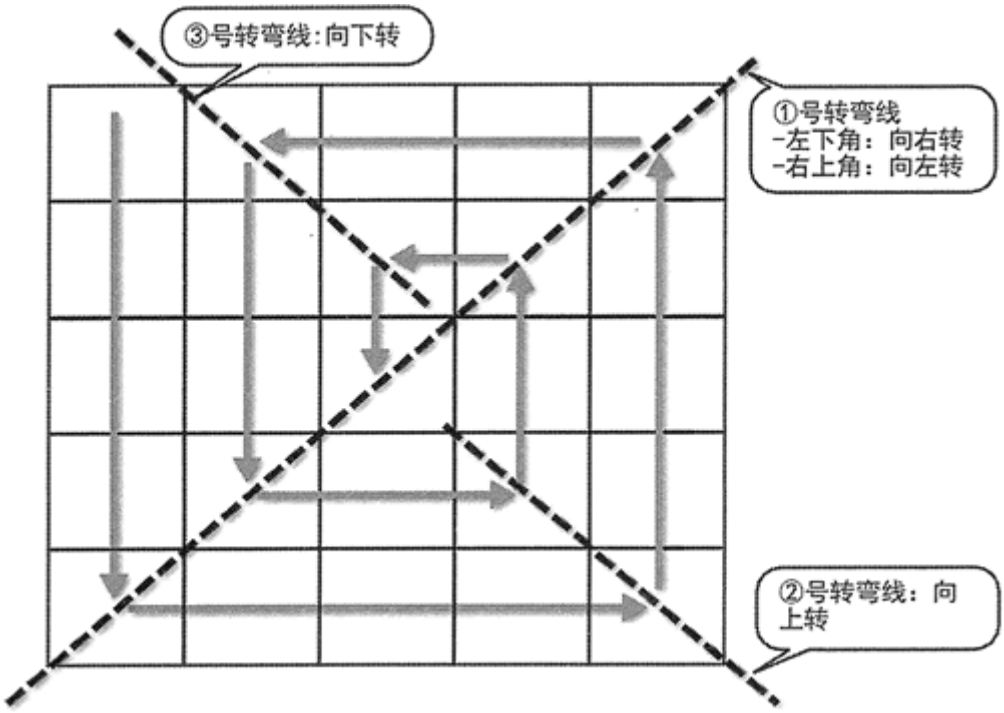


图 1 填数规则

掌握上面的规律之后，我们打算使用列表嵌套列表（相当于二维列表）的方式来存储这些整数，将数值存入嵌套列表时需要遵守这种“绕圈圈”的规则，然后再以二维方式将这个嵌套列表打印出来。

为了控制“绕圈”，该程序的关键点就是控制绕固的拐弯点。在图 1 中标出的对角线上的位置，就是重要的拐弯点。

找到图中 ①、②、③ 号转弯线之后，可以发现如下规则：

- 位于 ① 号转弯线的行索引与列索引总和为 n - 1（即给定整数值减 1）。
- 位于 ② 号转弯线的行索引与列索引相等。

- 位于 ③ 号转弯线的行索引等于列索引减 1。

总结出上面的规则之后，接下来就可实现如下程序：

纯文本复制

```
SIZE = 7
array = [[0] * SIZE]
# 创建一个长度 SIZE * SIZE 的二维列表
for i in range(SIZE - 1):
    array += [[0] * SIZE]
# 该 orient 代表绕圈的方向
# 其中 0 代表向下，1 代表向右，2 代表向左，3 代表向上
orient = 0
# 控制将 1~SIZE * SIZE 的数值填入二维列表中
# 其中 j 控制行索引，k 控制列索引
j = 0
k = 0
for i in range(1, SIZE * SIZE + 1) :
    array[j][k] = i
    # 如果位于图 4.2 中①号转弯线上
    if j + k == SIZE - 1 :
        # j>k，位于左下角
        if j > k :
            orient = 1
        # 位于右上角
        else :
            orient = 2
    # 如果位于图 5.2 中②号转弯线上
    elif (k == j) and (k >= SIZE / 2) :
        orient = 3
    # 如果 j 位于图 5.2 中③号转弯线上
    elif (j == k - 1) and (k <= SIZE / 2) :
        orient = 0
    # 根据方向来控制行索引、列索引的改变
```

```
            # 如果方向为向下绕圈
            if orient == 0 :
                j += 1
            # 如果方向为向右绕圈
            elif orient == 1:
                k += 1
            # 如果方向为向左绕圈
            elif orient == 2:
                k -= 1
            # 如果方向为向上绕圈
            elif orient == 3:
                j -= 1
        # 采用遍历输出上面的二维列表
        for i in range(SIZE) :
            for j in range(SIZE) :
                print('%02d ' % array[i][j], end = "")
            print("")
```

上面程序的重点就在于当处于转弯线上时绕圈的方向。一旦正确控制了绕圈的方向，接下来就可通过对 j（行索引）、k（列索引）的增减来控制绕圈了。

运行该程序，可以看到如下输出结果：

```
01 24 23 22 21 20 19

02 25 40 39 38 37 18

03 26 41 48 47 36 17

04 27 42 49 46 35 16

05 28 43 44 45 34 15

06 29 30 31 32 33 14

07 08 09 10 11 12 13
```

20.Python 项目实战之猜数字游戏（含实现代码）

猜数字游戏，旨在提高初学者对 Python 变量类型以及循环结构的使用。

此游戏的游戏规则如下：程序随机内置一个位于一定范围内的数字作为猜测的结果，由用户猜测此数字。用户每猜测一次，由系统提示猜测结果：太大了、太小了或者猜对了，直到用户猜对结果或者猜测次数用完导致失败。

这里有必要对“太大了”、“太小了”做一下解释，举个例子，程序选择了位于 0~20 之间的数字 13 作为猜测结果，假设用户第一次猜测 10，由于 13 > 10，因此程序应提示用户“太小了”；反之，如果用户猜测 15，由于 13 < 15，则程序应提示用户“太大了”。

除此之外，此项目中还涉及到一个没有学习的知识，即如何产生随机数，这里给大家提供一种方法，如下所示：

```
#引入生成随机数的模块
import random
#程序设定生成 1-20 之间的一个随机数
secretNum = random.randint(1,20)
```

通过使用 random 模块中的 randint() 函数，即可生成一定范围内的随机数。

假设只给用户 3 此猜测的机会，则猜数字游戏的实现代码如下（仅供参考）：

```
#引入生成随机数的模块
import random
#程序设定生成 1-20 之间的一个随机数，让用户猜
```

```
4.          secretNum = random.randint(1,20)
5.
6.          print("这是一个位于 1-20 之间的数")
7.          #设定用户只能猜 3 次
8.          for number in range(1,4):
9.              print("请输入猜测的数：")
10.             guess = int(input())
11.             if guess == 0:
12.                 break
13.             if guess < secretNum:
14.                 print("太小啦")
15.             elif guess > secretNum:
16.                 print("太大啦")
17.             else:
18.                 break
19.         if(guess == secretNum):
20.             print("真厉害，猜对啦，就是",str(guess))
21.         else:
22.             print("很遗憾，正确的答案应该是",str(secretNum))
```

运行示例：

这是一个位于 1-20 之间的数

请输入猜测的数：

10

太大啦

请输入猜测的数：

5

太小啦

请输入猜测的数：

7

太大啦

很遗憾，正确的答案应该是 6

第 7 章 函数和 lambda 表达式

函数是执行特定任务的一段代码，程序通过将一段代码定义成函数，并为该函数指定一个函数名，这样即可在需要的时候多次调用这段代码。因此，函数是代码复用的重要手段。

学习函数需要重点掌握定义函数、调用函数的方法。此外，本章也会介绍大量有关 Python 的高级内容，读者应该紧跟本章的讲解要点。

与函数紧密相关的另一个知识点是 lambda 表达式。lambda 表达式可作为表达式、函数参数或函数返回值，因此使用 lambda 表达式可以让程序更加简洁。

1.Python 函数（函数定义、函数调用）用法详解

Python 中，函数的应用非常广泛，前面章节中我们已经接触过多个函数，比如 input() 、print()、range()、len() 函数等等，这些都是 Python 的内置函数，可以直接使用。

除了可以直接使用的内置函数外，Python 还支持自定义函数，即将一段有规律的、可重复使用的代码定义成函数，从而达到一次编写、多次调用的目的。

比如，在程序中定义了一段代码，这段代码用于实现一个特定的功能。问题来了，如果下次需要实现同样的功能，难道要把前面定义的代码复制一次？如果这样做实在太傻了，这意味着每次当程序需要实现该功能时，都要将前面定义的代码复制一次。正确的做法是，将实现特定功能的代码定义成一个函数，每次当程序需要实现该功能时，只要执行（调用）该函数

即可。

通俗来讲，所谓函数，就是指为一段实现特定功能的代码“取”一个名字，以后即可通过该名字来执行（调用）该函数。使用函数，可以大大提高代码的重复利用率。

通常，函数可以接收零个或多个参数，也可以返回零个或多个值。从函数使用者的角度来看，函数就像一个“黑匣子”，程序将零个或多个参数传入这个“黑匣子”，该“黑匣子”经过一番计算即可返回零个或多个值。

对于“黑匣子”的内部细节（就是函数的内部实现细节），函数的使用者并不需要关心。就像前面在调用 len()、max()、min() 等函数时，我们只负责传入参数、接收返回值，至于函数内部的实现细节，我们并不关心。

如图 1 所示为函数调用示意图：

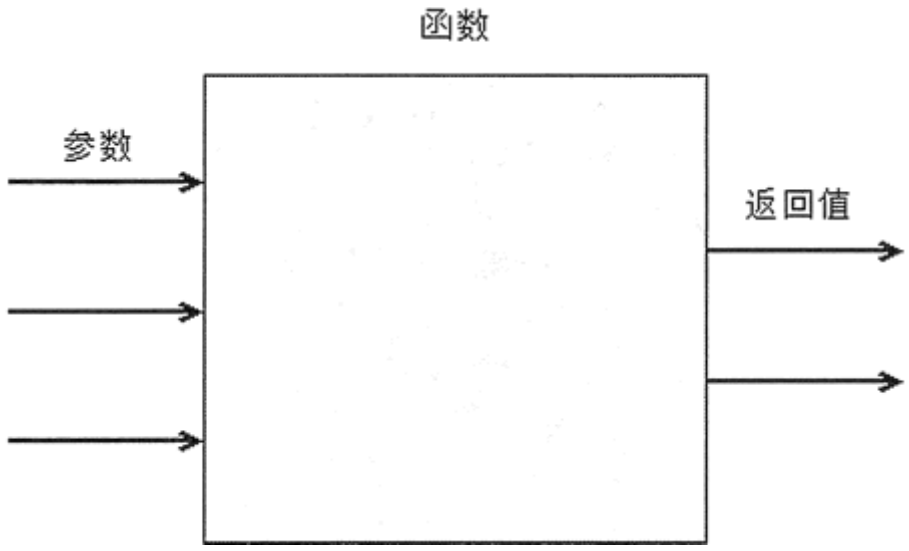


图 1 函数调用示意图

从函数定义者（实现函数的人）的角度来看，其至少需要想清楚以下 3 点：

1. 函数需要几个关键的需要动态变化的数据，这些数据应该被定义成函数的参数。
2. 函数需要传出几个重要的数据（就是调用该函数的人希望得到的数据），这些数据应该被定义成返回值。
3. 函数的内部实现过程。

> 不难看出，定义函数比调用函数要难得多，而本节正是教你如何定义函数的。不过不用担心，对于实现过程复杂的函数，定义本身就很费力，有时候实现不出来也完全正常。

Python 函数的定义

定义函数，也就是创建一个函数，可以理解为创建一个具有某些用途的工具。定义函数需要用 def 关键字实现，具体的语法格式如下：

```
def 函数名(形参列表):
    //由零条到多条可执行语句组成的代码块
    [return [返回值]]
```

其中，用 [] 括起来的为可选择部分，即可以使用，也可以省略。

此格式中，各部分参数的含义如下：

- 函数名：从语法角度来看，函数名只要是一个合法的标识符即可；从程序的可读性角度来看，函数名应该由一个或多个有意义的单词连缀而成，每个单词的字母全部小写，单词与单词之间使用下画线分隔。
- 形参列表：用于定义该函数可以接收的参数。形参列表由多个形参名组成，多个形参名之间以英文逗号（,）隔开。一旦在定义函数时指定了形参列表，调用该函数时就必须传入相应的参数值，也就是说，谁调用函数谁负责为形参赋值。

> 注意，在创建函数时，即使函数不需要参数，也必须保留一对空的“()”，否则 Python 解释器将提示“invaild syntax”错误。另外，如果想定义一个没有任何功能的空函数，可以使用 pass 语句作为占位符。

下面程序定义了两个函数：

```
def my_max(x, y) :
    # 定义一个变量 z，该变量等于 x、y 中较大的值
    z = x if x > y else y
    # 返回变量 z 的值
    return z
# 定义一个函数，声明一个形参
def say_hi(name) :
    print("===正在执行 say_hi()函数===")
    return name + "，您好！"
```

Python 函数的调用

调用函数也就是执行函数。如果把创建的函数理解为一个具有某种用途的工具，那么调用函数就相当于使用该工具。

函数调用的基本语法格式如下所示：

```
函数名([形参值])
```

其中，函数名即指的是要调用的函数的名称；形参值指的是当初创建函数时要求传入的各个形参的值。需要注意的是，创建函数有多少个形参，那么调用时就需要传入多少个值，且顺序必须和创建函数时一致。即便该函数没有参数，函数名后的小括号也不能省略。

例如，调用前面创建的那 2 个函数，执行代码如下：

```
        a = 6
        b = 9
        # 调用 my_max()函数，将函数返回值赋值给 result 变量
        result = my_max(a , b) # ①
        print("result:", result)
        # 调用 say_hi()函数，直接输出函数的返回值
        print(say_hi("孙悟空")) # ②
```

上面程序中，分别在 ① 号、② 号代码处调用了 my_max() 和 say_hi() 这两个函数。从下面的运行结果可以看出，当程序调用一个函数时，既可以把调用函数的返回值赋值给指定变量，也可以将函数的返回值传给另一个函数，作为另一个函数的参数。

运行上面程序，将可以看到如下运行结果：

```
result: 9
===正在执行 say_hi()函数===
孙悟空，您好！
```

另外，在函数体中使用 return 语句可以显式地返回一个值，return 语句返回的值既可是有值的变量，也可是一个表达式。例如上面的 my_max() 函数，实际上也可简写为如下形式：

```
1.          def my_max(x, y) :
2.              # 返回一个表达式
3.              return x if x > y else y
```

为函数提供说明文档

前面介绍过可以使用 Python 内置的 help() 函数查看其他函数的帮助文档，我们也经常通过 help() 函数查看指定函数的帮助信息，这对于 Python 开发者来说非常重要。

我们还可以为函数编写说明文档，只要把一段字符串放在函数声明之后、函数体之前，这段字符串将被作为函数的部分，这个文档就是函数的说明文档。

程序既可通过 help() 函数查看函数的说明文档，也可通过函数的 __doc__ 属性访问函数的说明文档。下面程序示范了为函数编写说明文档：

纯文本复制

```
1.          def my_max(x, y) :
2.              '''
3.              获取两个数值之间较大数的函数。
```

```
4.
5.              my_max(x, y)
6.                  返回 x、y 两个参数之间较大的那个
7.              '''
8.              # 定义一个变量 z，该变量等于 x、y 中较大的值
9.              z = x if x > y else y
10.             # 返回变量 z 的值
11.             return z
```

上面程序使用多行字符串的语法为 my_max() 函数编写了说明文档，接下来程序既可通过 help() 函数查看该函数的说明文档，也可通过 __doc__ 属性访问该函数的说明文档。

```
1.          # 使用 help()函数查看 my_max 的帮助文档
2.          help(my_max)
3.          #或者 print(my_max.__doc__)
```

运行上面代码，可以看到如下运行结果：

```
Help on function my_max in module __main__:


my_max(x, y)
    获取两个数值之间较大数的函数。


    my_max(x, y)
        返回 x、y 两个参数之间较大的那个
```

2.Python 函数值传递和引用传递（包括形式参数和实际参数）

通常情况下，定义函数时都会选择有参数的函数形式，函数参数的作用是传递数据给函数，令其对接收的数据做具体的操作处理。

在使用函数时，经常会用到形式参数（简称“形参”）和实际参数（简称“实参”），二者都叫参数，之间的区别是：

- 形式参数：在定义函数时，函数名后面括号中的参数就是形式参数，例如:

```
#定义函数时，这里的函数参数 obj 就是形式参数
def demo(obj)
    print(obj)
```

- 实际参数：在调用函数时，函数名后面括号中的参数称为实际参数，也就是函数的调用者给函数的参数。例如:

```
a = "C 语言中文网"
#调用已经定义好的 demo 函数，此时传入的函数参数 a 就是实际参数
demo(a)
```

> 实参和形参的区别，就如同剧本选主角，剧本中的角色相当于形参，而演角色的演员就相当于实参。

明白了什么是形参和实参后，再来想一个问题，那就是实参是如何传递给形参的呢？

Python 中，根据实际参数的类型不同，函数参数的传递方式可分为 2 种，分别为值传递和引用（地址）传递：

1. 值传递：适用于实参类型为不可变类型（字符串、数字、元组）；
2. 引用（地址）传递：适用于实参类型为可变类型（列表，字典）；

值传递和引用传递的区别是，函数参数进行值传递后，若形参的值发生改变，不会影响实参

的值；而函数参数继续引用传递后，改变形参的值，实参的值也会一同改变。

例如，定义一个名为 demo 的函数，分别为传入一个字符串类型的变量（代表值传递）和列表类型的变量（代表引用传递）：

```
1.          def demo(obj) :
2.              obj += obj
3.              print("形参值为：",obj)
4.          print("--------值传递------")
5.          a = "C 语言中文网"
6.          print("a 的值为：",a)
7.          demo(a)
8.          print("实参值为：",a)
9.          print("------引用传递------")
10.         a = [1,2,3]
11.         print("a 的值为：",a)
12.         demo(a)
13.         print("实参值为：",a)
```

运行结果为：

```
-------值传递-----

a 的值为： C 语言中文网

形参值为： C 语言中文网 C 语言中文网

实参值为： C 语言中文网

-----引用传递-----

a 的值为： [1, 2, 3]

形参值为： [1, 2, 3, 1, 2, 3]

实参值为： [1, 2, 3, 1, 2, 3]
```

分析运行结果不难看出，在执行值传递时，改变形式参数的值，实际参数并不会发生改变；而在进行引用传递时，改变形式参数的值，实际参数也会发生同样的改变。

对于初学者来说，本节只需要了解形参和实参，值传递和引用传递的区别即可。对于函数参数的传递方法，如果读者想深入探究其原因，可阅读《Python 函数参数传递机制》一节。

3.Python 函数参数传递机制

Python 中，函数参数由实参传递给形参的过程，是由参数传递机制来控制的。通过学习《Python 函数值传递和引用传递》一节我们知道，根据实际参数的类型不同，函数参数的传递方式分为值传递和引用传递（又称为地址传递），本节将对这两种传递机制做深度剖析。

Python 函数参数的值传递机制

所谓值传递，实际上就是将实际参数值的副本（复制品）传入函数，而参数本身不会受到任何影响。

> 值传递的方式，类似于《西游记》里的孙悟空，它复制一个假孙悟空，假孙悟空具有的能力和真孙悟空相同，可除妖或被砍头。但不管这个假孙悟空遇到什么事，真孙悟空都不会受到任何影响。与此类似，传入函数的是实际参数值的复制品，不管在函数中对这个复制品如何操作，实际参数值本身不会受到任何影响。

下面程序演示了函数参数进行值传递的效果：

```
1.          def swap(a , b) :
2.              # 下面代码实现 a、b 变量的值交换
3.              a, b = b, a
4.              print("swap 函数里，a 的值是", \
5.                  a, "; b 的值是", b)
6.          a = 6
7.          b = 9
8.          swap(a , b)
```

```
9.          print("交换结束后，变量 a 的值是", \
10.             a , "；变量 b 的值是", b)
```

运行上面程序，将看到如下运行结果：

```
swap 函数里，a 的值是 9 ；b 的值是 6
交换结束后，变量 a 的值是 6 ；变量 b 的值是 9
```

从上面的运行结果来看，在 swap() 函数里，a 和 b 的值分别是 9、6，交换结束后，变量 a 和 b 的值依然是 6、9。从这个运行结果可以看出，程序中实际定义的变量 a 和 b，并不是 swap() 函数里的 a 和 b 。

正如前面所讲的，swap() 函数里的 a 和 b 只是主程序中变量 a 和 b 的复制品。下面通过示意图来说明上面程序的执行过程。

上面程序开始定义了 a、b 两个局部变量，这两个变量在内存中的存储示意图如图 1 所示。

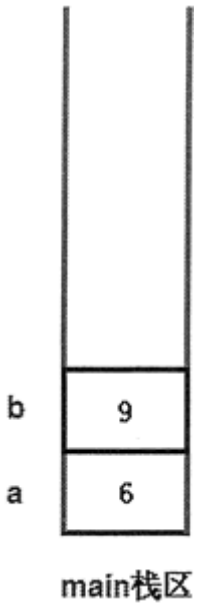


图 1 主栈区中 a、b 变量存储示意图

当程序执行 swap() 函数时，系统进入 swap() 函数，并将主程序中的 a、b 变量作为参数值传入 swap() 函数，但传入 swap() 函数的只是 a、b 的副本，而不是 a、b 本身。进入 swap() 函数后 ,系统中产生了 4 个变量 ,这 4 个变量在内存中的存储示意图如图 2 所示。

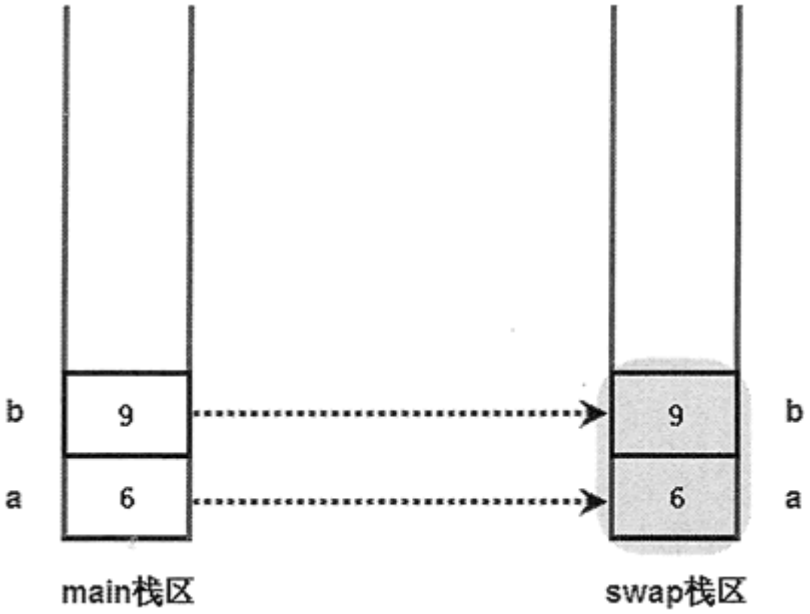


图 2 主栈区的变量作为参数值传入 swap() 函数后存储示意图

当在主程序中调用 swap() 函数时，系统分别为主程序和 swap() 函数分配两块栈区，用于保存它们的局部变量。将主程序中的 a、b 变量作为参数值传入 swap() 函数，实际上是在 swap() 函数栈区中重新产生了两个变量 a、b，并将主程序栈区中 a、b 变量的值分别赋值给 swap() 函数栈区中的 a、b 参数（就是对 swap() 函数的 a、b 两个变量进行初始化）。此时，系统存在两个 a 变量、两个 b 变量，只是存在于不同的栈区中而已。

程序在 swap() 函数中交换 a、b 两个变量的值，实际上是对图 2 中灰色区域的 a、b 变量进行交换。交换结束后，输出 swap() 函数中 a、b 变量的值，可以看到 a 的值为 9，b 的值为 6，此时在内存中的存储示意图如图 3 所示。

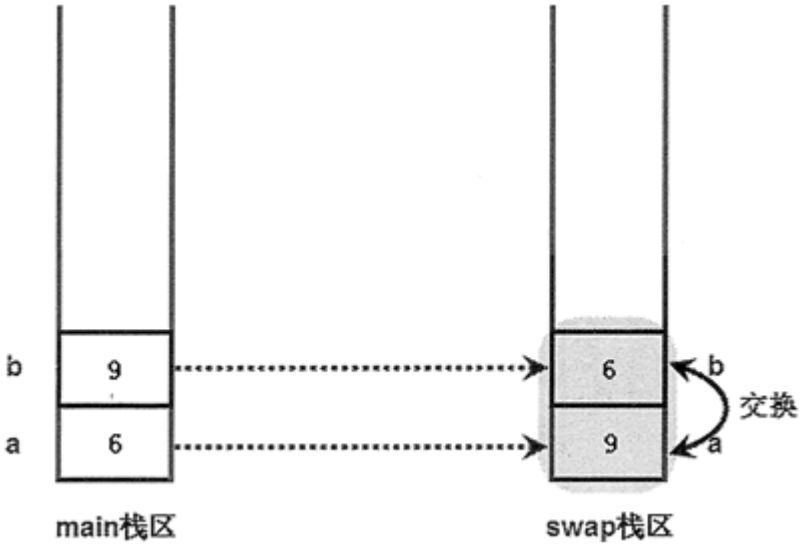


图 3 swap() 函数中 a、b 交换之后的存储示意图

对比图 3 与图 1，可以看到两个示意图中主程序栈区中 a、b 的值并未有任何改变，程序

改变的只是 swap() 函数栈区中 a、b 的值。这就是值传递的实质：当系统开始执行函数时，系统对形参执行初始化，就是把实参变量的值赋给函数的形参变量，在函数中操作的并不是实际的实参变量。

Python 函数参数的引用传递

如果实际参数的数据类型是可变对象（列表、字典），则函数参数的传递方式将采用引用传递方式。需要注意的是，引用传递方式的底层实现，采用的依然还是值传递的方式。

下面程序示范了引用传递参数的效果：

```
        def swap(dw):
            # 下面代码实现 dw 的 a、b 两个元素的值交换
            dw['a'], dw['b'] = dw['b'], dw['a']
            print("swap 函数里，a 元素的值是",\
                dw['a'], "；b 元素的值是", dw['b'])
        dw = {'a': 6, 'b': 9}
        swap(dw)
        print("交换结束后，a 元素的值是",\
            dw['a'], "；b 元素的值是", dw['b'])
```

运行上面程序，将看到如下运行结果：

```
swap 函数里，a 元素的值是 9 ；b 元素的值是 6
交换结束后，a 元素的值是 9 ；b 元素的值是 6
```

从上面的运行结果来看，在 swap() 函数里，dw 字典的 a、b 两个元素的值被交换成功。不仅如此，当 swap() 函数执行结束后，主程序中 dw 字典的 a、b 两个元素的值也被交换了。这很容易造成一种错觉，即在调用 swap() 函数时，传入 swap() 函数的就是 dw 字

典本身，而不是它的复制品。但这只是一种错觉，下面还是结合示意图来说明程序的执行过程。

程序开始创建了一个字典对象，并定义了一个 dw 引用变量（其实就是一个指针）指向字典对象，这意味着此时内存中有两个东西：对象本身和指向该对象的引用变量。此时在系统内存中的存储示意图如图 4 所示：

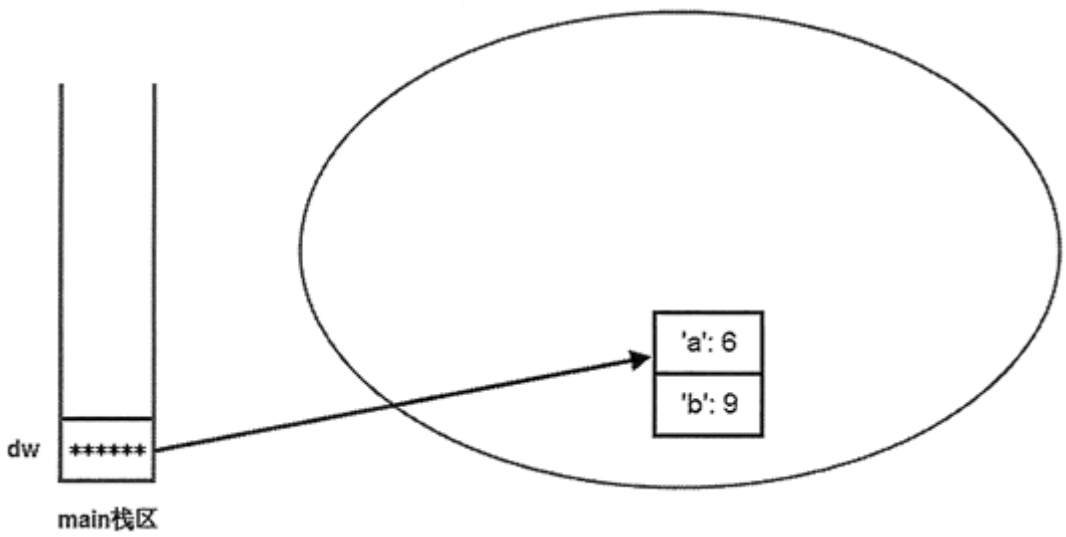


图 4 主程序创建了字典对象后存储示意图

接下来主程序开始调用 swap() 函数，在调用 swap() 函数时，dw 变量作为参数传入 swap() 函数，这里依然采用值传递方式：把主程序中 dw 变量的值赋给 swap() 函数的 dw 形参，从而完成 swap() 函数的 dw 参数的初始化。值得指出的是，主程序中的 dw 是一个引用变量（也就是一个指针），它保存了字典对象的地址值，当把 dw 的值赋给 swap() 函数的 dw 参数后，就是让 swap() 函数的 dw 参数也保存这个地址值，即也会引用到同一个字典对象。图 5 显示了 dw 字典传入 swap() 函数后的存储示意图。

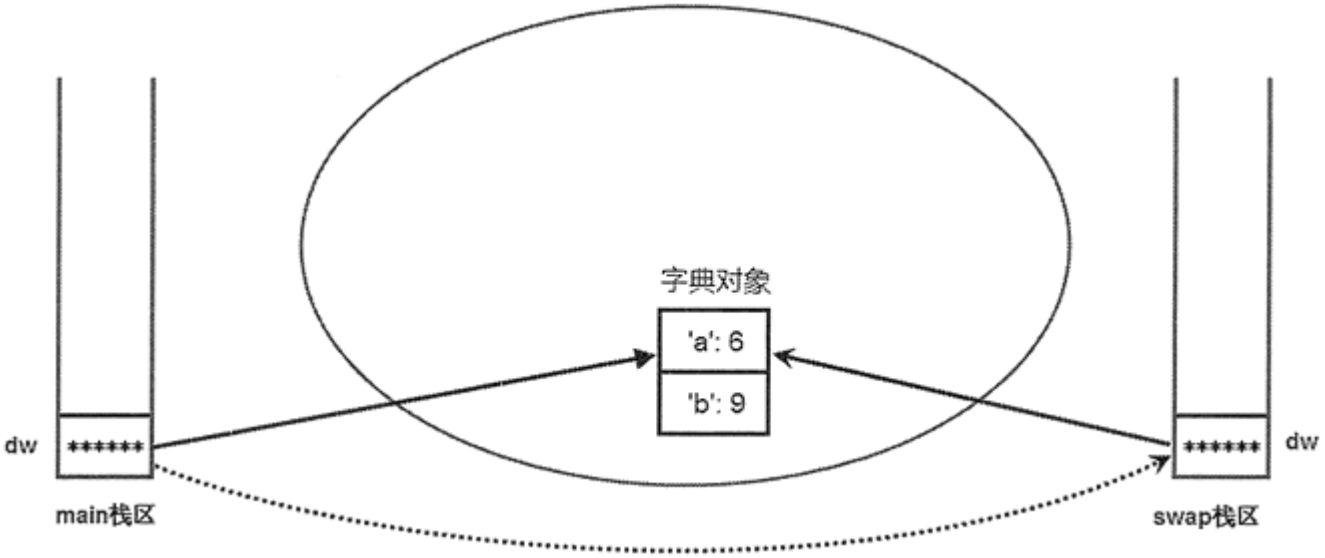


图 5 dw 字典传入 swap() 函数后存储示意图

从图 5 来看，这种参数传递方式是不折不扣的值传递方式，系统一样复制了 dw 的副本传入 swap() 函数。但由于 dw 只是一个引用变量，因此系统复制的是 dw 变量，并未复制字典本身。

当程序在 swap() 函数中操作 dw 参数时，由于 dw 只是一个引用变量，故实际操作的还是字典对象。此时，不管是操作主程序中的 dw 变量，还是操作 swap() 函数里的 dw 参数，其实操作的都是它们共同引用的字典对象，它们引用的是同一个字典对象。因此，当在 swap() 函数中交换 dw 参数所引用字典对象的 a、b 两个元素的值后，可以看到在主程序中 dw 变量所引用字典对象的 a、b 两个元素的值也被交换了。

为了更好地证明主程序中的 dw 和 swap() 函数中的 dw 是两个变量，在 swap() 函数的最后一行增加如下代码：

```
#把 dw 直接赋值为 None，让它不再指向任何对象
dw = None
```

运行上面代码，结果是 swap() 函数中的 dw 变量不再指向任何对象，程序其他地方没有任何改变。主程序调用 swap() 函数后，再次访问 dw 变量的 a、b 两个元素，依然可以输出 9、6。可见，主程序中的 dw 变量没有受到任何影响。实际上，当在 swap() 函数中增加“dw =None”代码后，在内存中的存储示意图如图 6 所示。

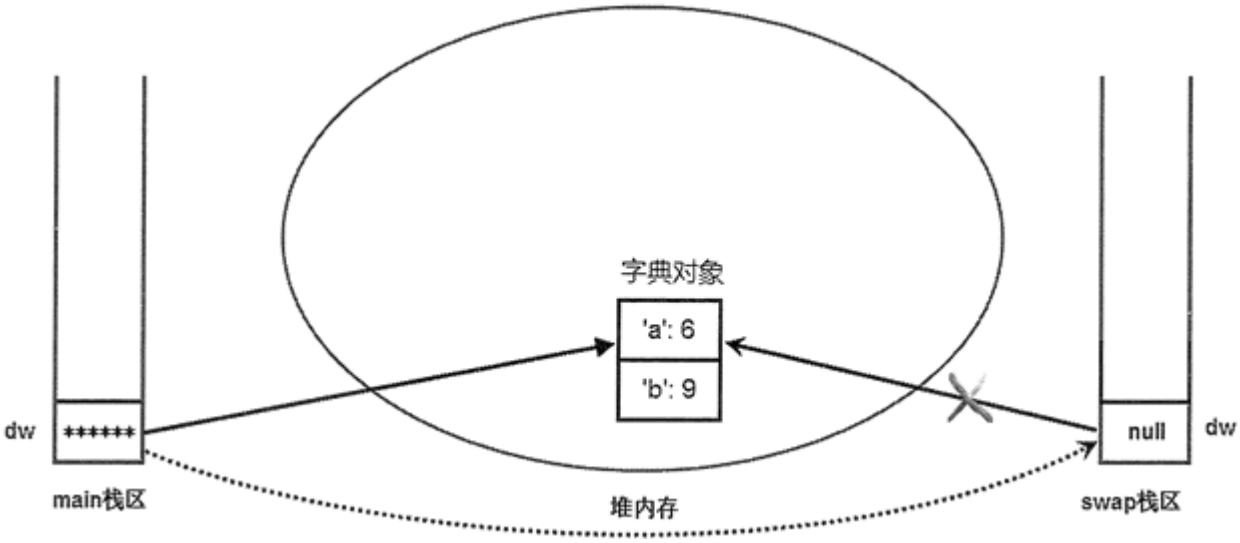


图 6 将 swap() 函数中的 dw 赋值为 None 后存储示意图

从图 6 来看，把 swap() 函数中的 dw 赋值为 None 后，在 swap() 函数中失去了对字典对象的引用，不可再访问该字典对象。但主程序中的 dw 变量不受任何影响，依然可以引用该字典对象，所以依然可以输出字典对象的 a、b 元素的值。

通过上面介绍可以得出如下两个结论：

1. 不管什么类型的参数，在 Python 函数中对参数直接使用“=”符号赋值是没用的，直接使用“=”符号赋值并不能改变参数。
2. 如果需要让函数修改某些数据，则可以通过把这些数据包装成列表、字典等可变对象，然后把列表、字典等可变对象作为参数传入函数，在函数中通过列表、字典的方法修改它们，这样才能改变这些数据。

4.Python 位置参数

位置参数，有时也称必备参数，指的是必须按照正确的顺序将实际参数传到函数中，换句话说，调用函数时传入实际参数的数量和位置都必须和定义函数时保持一致。

实参和形参数量必须一致

在调用函数，指定的实际参数的数量，必须和形式参数的数量一致（传多传少都不行），否则 Python 解释器会抛出 TypeError 异常，并提示缺少必要的位置参数。

例如：

```
1.          def girth(width , height):
2.              return 2 * (width + height)
3.          #调用函数时，必须传递 2 个参数，否则会引发错误
4.          print(girth(3))
```

运行结果为：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 4, in <module>
    print(girth(3))
TypeError: girth() missing 1 required positional argument: 'height'
```

可以看到，抛出的异常类型为 TypeError，具体是指 girth() 函数去烧一个必要的 height 参数。

同样，多传参数也会抛出异常：

```
1.          def girth(width , height):
2.              return 2 * (width + height)
3.          #调用函数时，必须传递 2 个参数，否则会引发错误
4.          print(girth(3,2,4))
```

运行结果为：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 4, in <module>
    print(girth(3,2,4))
TypeError: girth() takes 2 positional arguments but 3 were given
```

通过 TypeErroe 异常信息可以知道 ,girth() 函数本只需要 2 个参数 ,但是却传入了 3 个参数。

实参和形参位置必须一致

在调用函数时，传入实际参数的位置必须和形式参数位置一一对应，否则会产生以下 2 种结果：

1. 抛出 TypeError 异常

当实际参数类型和形式参数类型不一致，并且在函数种，这两种类型之间不能正常转换，此时就会抛出 TypeError 异常。

例如：

```
def area(height,width):
    return height*width/2
print(area("C 语言中文网",3))
```

输出结果为：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 3, in <module>
    print(area("C 语言中文网",3))
  File "C:\Users\mengma\Desktop\1.py", line 2, in area
    return height*width/2
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

以上显示的异常信息，就是因为字符串类型和整形数值做除法运算。

2. 产生的结果和预期不符

调用函数时，如果指定的实际参数和形式参数的位置不一致，但它们的数据类型相同，那么程序将不会抛出异常，只不过导致运行结果和预期不符。

例如，设计一个求梯形面积的函数，并利用此函数求上底为 4cm，下底为 3cm，高为 5cm 的梯形的面积。但如果交互高和下低参数的传入位置，计算结果将导致错误：

```
1.  def area(upper_base,lower_bottom,height):
2.      return (upper_base+lower_bottom)*height/2
3.  print("正确结果为：",area(4,3,5))
4.  print("错误结果为：",area(4,5,3))
```

运行结果为:

```
正确结果为： 17.5
错误结果为： 13.5
```

因此，在调用函数时，一定要确定好位置，否则很有可能产生类似示例中的这类错误，还不容易发现。

5.Python 关键字参数

目前为止，我们使用函数时所用的参数都是位置参数，即传入函数的实际参数必须与形式参数的数量和位置对应。而本节将介绍的关键字参数，则可以避免牢记参数位置的麻烦，令函数的调用和参数传递更加灵活方便。

关键字参数是指使用形式参数的名字来确定输入的参数值。通过此方式指定函数实参时，不再需要与形参的位置完全一致，只要将参数名写正确即可。

> 因此，Python 函数的参数名应该具有更好的语义，这样程序可以立刻明确传入函数的每个参数的含义。

例如如下程序：

```
1.          # 定义一个函数
2.          def girth(width , height):
3.              print("width: ", width)
4.              print("height: ", height)
5.              return 2 * (width + height)
6.          # 传统调用函数的方式，根据位置传入参数
7.          print(girth(3.5, 4.8))
8.          # 根据关键字参数来传入参数
9.          print(girth(width = 3.5, height = 4.8))
10.         # 使用关键字参数时可交换位置
11.         print(girth(height = 4.8, width = 3.5))
12.         # 部分使用关键字参数，部分使用位置参数
13.         print(girth(3.5, height = 4.8))
```

上面程序定义了一个简单的 girth() 函数，该函数包含 width、height 两个参数，该函数与前面定义的函数并没有任何区别。

接下来在调用该函数时，既可使用传统的根据位置参数来调用（如上面程序中第 7 行代码所示），也可根据关键字参数来调用（如上面程序中第 9 行代码所示），在使用关键字参数调用时可交换参数的位置（如上面程序中第 11 行代码所示），还可混合使用位置参数和关键字参数（如上面程序中第 13 行代码所示）。

需要说明的是，如果希望在调用函数时混合使用关键字参数和位置参数，则关键字参数必须位于位置参数之后。换句话说，在关键字参数之后的只能是关键字参数。例如如下代码是错误的：

```
# 位置参数必须放在关键字参数之前，下面代码错误
print(girth(width = 3.5, 4.8))
```

运行上面代码，将会提示如下错误：

```
SyntaxError: positional argument follows keyword argument
```

6.Python 默认参数

我们知道，在调用函数时，如果不指定某个参数，解释器会抛出异常。为了解决这个问题，Python 允许为参数设置默认值，即在定义函数时，直接给形式参数指定一个默认值，这样的话，即便调用函数时没有给拥有默认值的形参传递参数，该参数可以直接使用定义函数时设置的默认值。

定义带有默认值参数的函数，其语法格式如下：

```
def 函数名(...，形参名=默认值)：
    代码块
```

注意，在使用此格式定义函数时，指定有默认值的形式参数必须在所有没默认值参数的最后，否则会产生语法错误。

例如，如下程序为 name、message 形参指定了默认值：

```
1.          # 为两个参数指定默认值
```

```
2.          def say_hi(name = "孙悟空", message = "欢迎来到 C 语言中文网"):
3.              print(name, ", 您好")
4.              print("消息是：", message)
5.          # 全部使用默认参数
6.          say_hi()
7.          # 只有 message 参数使用默认值
8.          say_hi("白骨精")
9.          # 两个参数都不使用默认值
10.         say_hi("白骨精", "欢迎学习 Python")
11.         # 只有 name 参数使用默认值
12.         say_hi(message = "欢迎学习 Python")
```

运行结果为：

```
孙悟空 , 您好
消息是： 欢迎来到 C 语言中文网
白骨精 , 您好
消息是： 欢迎来到 C 语言中文网
白骨精 , 您好
消息是： 欢迎学习 Python
孙悟空 , 您好
消息是： 欢迎学习 Python
```

从上面程序可以看出，如果只传入一个位置参数，由于该参数位于第一位，系统会将该参数值传给 name 参数。因此，不能按如下方式调用 say_hi() 函数：

```
say_hi("欢迎学习 Python")
```

上面调用时传入的“欢迎学习 Python”字符串将传给 name 参数，而不是 message 参数。

更不能按如下方式来调用 say_hi() 函数：

```
say_hi(name="白骨精", "欢迎学习 Python")
```

因为 Python 规定，关键字参数必须位于位置参数的后面。因此提示错误：positional argument follows keyword argument。

那么，能不能单纯地将上面两个参数交换位置呢？

```
say_hi("欢迎学习 Python"，name="白骨精")
```

上面调用依然是错误的，因为第一个字符串没有指定关键字参数，因此将使用位置参数为 name 参数传入参数值，第二个参数使用关键字参数的形式再次为 name 参数传入参数值，这意味着两个参数值其实都会传给 name 参数，程序为 name 参数传入了多个参数值。因此提示错误：say_hi() got multiple values for argument 'name'。

将函数调用改为如下两种形式是正确的：

```
1.          say_hi("白骨精", message="欢迎学习 Python")
2.          say_hi(name="白骨精", message="欢迎学习 Python")
```

上面第一行代码先使用位置参数为 name 参数传入参数值，再使用关键字参数为 message 参数传入参数值；第二行代码中的 name、message 参数都使用关键字参数传

入参数值。

再次强调，由于 Python 要求在调用函数时关键字参数必须位于位置参数的后面，因此在定义函数时指定了默认值的参数（关键字参数）必须在没有默认值的参数之后。例如如下代码：

纯文本复制

```
# 定义一个打印三角形的函数，有默认值的参数必须放在后面
def printTriangle(char, height = 5) :
    for i in range(1, height + 1) :
        # 先打印一排空格
        for j in range(height - i) :
            print(' ', end = '')
        # 再打印一排特殊字符
        for j in range(2 * i - 1) :
            print(char, end = '')
        print()
printTriangle('@', 6)
printTriangle('#', height=7)
printTriangle(char = '*')
```

上面程序定义了 printTriangle() 函数，其中，有默认值的 height 形参，必须放在 char 形参的后面；反之，将会造成语法错误。

在 Python 中，可以使用“函数名.__defaults__”查看函数的默认值参数的当前值，其返回值是一个元组。例如，显示上面定义的 printTriangle 函数的默认值参数的当前值，可以使用 printTriangle.__defaults__ ，其结果为 (5,)。

7.Python 函数如何传入任意个参数？

很多编程语言都允许定义个数可变的参数，这样可以在调用函数时传入任意多个参数。

Python 也不例外，在定义函数时也可以使用可变参数。

可变参数，又称不定长参数，即传入函数中的实际参数可以是任意多个。Python 定义可变参数，主要有以下 2 种形式。

1) 可变参数：形参前添加一个 '*'

此种形式的语法格式如下所示：

```
*args
```

其中，args 表示创建一个名为 args 的空元组，该元组可接受任意多个外界传入的非关键字实参。

下面程序定义了一个形参个数可变的函数：

```
1.          # 定义了支持参数收集的函数
2.          def test(a, *books) :
3.              print(books)
4.              # books 被当成元组处理
5.              for b in books :
6.                  print(b)
7.              # 输出整数变量 a 的值
8.              print(a)
9.          # 调用 test()函数
10.         test(5 , "C 语言中文网" , "Python 教程")
```

运行上面程序，将看到如下运行结果：

```
('C 语言中文网', 'Python 教程')
C 语言中文网
Python 教程
5
```

从上面的运行结果可以看出，当调用 test() 函数时，books 元组可存储任意个字符串。从 test() 的函数体代码来看，参数收集的本质就是一个元组： Python 会将传给 books 参数的多个值收集成一个元组。

Python 允许个数可变的形参可以处于形参列表的任意位置(不要求是形参列表的最后一个参数) 但同时普通参数必须以关键字实参的形式传值，例如如下程序：

```
1.          # 定义了支持参数收集的函数
2.          def test(*books ,num) :
3.              print(books)
4.              # books 被当成元组处理
5.              for b in books :
6.                  print(b)
7.              print(num)
8.          # 调用 test()函数
9.          test("C 语言中文网", "Python 教程", num = 20)
```

运行结果为：

```
('C 语言中文网', 'Python 教程')
C 语言中文网
```

```
Python 教程
20
```

正如从上面程序中所看到的，test() 函数的第一个参数就是个数可变的形参，由于该参数可接收个数不等的参数值，因此如果需要给后面的参数传入参数值，则必须使用关键字参数，否则程序会把所传入的多个值都当成是传给 books 参数的。

2) 可变参数：形参前添加两个'*'

这种形式的语法格式如下：

```
**kwargs
```

*kwargs 表示创建一个名为 kwargs 的空字典。该字典可以接收任意多个以关键字参数赋值的实际参数。

例如如下代码：

```
1.          # 定义了支持参数收集的函数
2.          def test(x, y, z=3, *books, **scores) :
3.              print(x, y, z)
4.              print(books)
5.              print(scores)
6.          test(1, 2, 3, "C 语言中文网" , "Python 教程", 语文=89, 数学=94)
```

上面程序在调用 test() 函数时，前面的 1、2、3 将会传给普通参数 x、y、z；接下来的两个字符串将会由 books 参数收集成元组；最后的两个关键字参数将会被收集成字典。运行上面代码，会看到如下输出结果：

```
1 2 3
('C 语言中文网', 'Python 教程')
{'语文': 89, '数学': 94}
```

这里需要注意一点，对于以上面方式定义的 test() 函数，参数 z 的默认值几乎不能发挥作用。比如按如下方式调用 test() 函数：

```
test(1, 2, "C 语言中文网"，"Python 教程"，语文=89，数学=94)
```

上面代码在调用 test() 函数时，前面的 1、2、"C 语言中文网" 将会传递给普通参数 x、y、z；接下来的一个字符串将会由 books 参数收集成元组；最后的两个关键字参数将会被收集成字典。运行上面代码，会看到如下输出结果：

```
1 2 C 语言中文网
('Python 教程',)
{'语文': 89, '数学': 94}
```

如果希望让 z 参数的默认值发挥作用，则需要只传入两个位置参数。例如如下调用代码：

```
test(1, 2，语文=89，数学=94)
```

上面代码在调用 test() 函数时，前面的 1、2 将会传给普通参数 x、y，此时 z 参数将使用默认的参数值 3，books 参数将是一个空元组；接下来的两个关键字参数将会被收集成

字典。运行上面代码，会看到如下输出结果：

```
1 2 3
()
{'语文': 89, '数学': 94}
```

8.Python 如何用序列中元素给函数传递参数？

所谓逆向参数收集，指的是在程序已有列表、元组、字典等对象的前提下，把它们的元素“拆开”后传给函数的参数。逆向参数收集需要在传入的列表、元组参数之前添加一个星号，在字典参数之前添加两个星号。例如如下代码：

```
1.          def test(name, message):
2.              print("用户是：", name)
3.              print("欢迎消息：", message)
4.          my_list = ['小明', '欢迎来 C 语言中文网']
5.          test(*my_list)
```

运行结果为：

```
用户是： 小明
欢迎消息： 欢迎来 C 语言中文网
```

程序中定义了一个需要两个参数的函数，而 my_list 列表包含两个元素，为了让程序将 my_list 列表的两个元素传给 test() 函数，程序在传入的 my_list 参数之前添加了一个星号。

实际上，即使是可变参数，如果程序需要将一个元组传给该参数，那么同样需要使用逆向收集。例如如下代码：

```
def foo(name, *nums):
    print("name 参数: ", name)
    print("nums 参数: ", nums)
my_tuple = (1, 2, 3)
# 使用逆向收集，将 my_tuple 元组的元素传给 nums 参数
foo('fkit', *my_tuple)
```

上面程序中，调用将 'fkit' 传给 foo() 函数的 name 参数，然后使用逆向收集将 my_tuple 包含的多个元素传给 nums 参数，nums 再将 my_tuple 的多个元素收集成元组。

运行上面代码，将看到如下输出结果：

```
name 参数:  fkit
nums 参数:  (1, 2, 3)
```

此外，也可使用如下方式调用 foo() 函数：

```
# 使用逆向收集，将 my_tuple 元组的第一个元素传给 name 参数，剩下参数传给 nums 参数
foo(*my_tuple)
```

此时程序会对 my_tuple 进行逆向收集，其中第一个元素传给 name 参数，后面剩下的元素传给 nums 参数。运行上面代码，将看到如下输出结果：

```
name 参数:  1
nums 参数:  (2, 3)
```

如果不使用逆向收集（不在元组参数之前添加星号），整个元组将会作为一个参数，而不是将元组的元素作为多个参数。例如按如下方式调用 foo() 函数：

```
# 不使用逆向收集，my_tuple 元组整体传给 name 参数
foo(my_tuple)
```

上面调用没有使用逆向收集，因此 my_tuple 整体作为参数值传给 name 参数。运行上面代码，将看到如下输出结果：

```
name 参数:  (1, 2, 3)
nums 参数:  ()
```

字典也支持逆向收集，字典将会以关键字参数的形式传入。例如如下代码：

纯文本复制

```
def bar(book, price, desc):
    print(book, "VIP 价格是:", price)
    print('描述信息', desc)
my_dict = {'price': 159, 'book': 'C 语言中文网', 'desc': '这是一个精美而实用的网站'}
# 按逆向收集的方式将 my_dict 的多个 key-value 传给 bar()函数
bar(**my_dict)
```

运行结果为：

```
C 语言中文网 VIP 价格是: 159
描述信息 这是一个精美而实用的网站
```

上面程序中，bar() 需要三个参数。接下来程序定义了一个 my_dict 字典，该字典正好包含三个 key-value 对 ,程序使用逆向收集即可将 my_dict 包含的三个 key-value 对以关键字参数的形式传给 bar() 函数。

9.Python return 函数返回值

在 Python 中，有一个特殊的常量 None（N 必须大写）。和 False 不同，它不表示 0，也不表示空字符串，而表示没有值，也就是空值。

这里的空值并不代表空对象，即 None 和 []、 “” 不同：

```
>>> None is []
False
>>> None is ""
False
```

None 有自己的数据类型 , 我们可以在 IDLE 中使用 type() 函数查看它的类型 , 执行代码如下：

```
>>> type(None)
<class 'NoneType'>
```

可以看到，它属于 NoneType 类型。

需要注意的是，None 是 NoneType 数据类型的唯一值（其他编程语言可能称这个值为 null、nil 或 undefined），也就是说，我们不能再创建其它 NoneType 类型的变量，但是可以将 None 赋值给任何变量。如果希望变量中存储的东西不与任何其它值混淆，就可以使用 None。

除此之外，None 常用于 assert、判断以及函数无返回值的情况。举个例子，在前面章节中我们一直使用 print() 函数输出数据，其实该函数的返回值就是 None。因为它的功能是在屏幕上显示文本，根本不需要返回任何值，所以 print() 就返回 None。

```
>>> spam = print('Hello!')
Hello!
>>> None == spam
True
```

另外，对于所有没有 return 语句的函数定义，Python 都会在末尾加上 return None，使用不带值的 return 语句（也就是只有 return 关键字本身），那么就返回 None。

到目前为止，我们创建的函数都只是对传入的数据进行了处理，处理完了就结束。但实际上，在某些场景中，我们还需函数将处理的结果反馈回来，就好像主管向下级员工下达命令，让其去打印文件，员工打印好文件后并没有完成任务，还需要将文件交给主管。

Python 中，用 def 语句创建函数时，可以用 return 语句指定应该返回的值，该返回值可

以是任意类型。需要注意的是，return 语句在同一函数中可以出现多次，但只要有一个得到执行，就会直接结束函数的执行。

函数中，使用 return 语句的语法格式如下：

```
return [返回值]
```

其中，返回值参数可以指定，也可以省略不写（将返回空值 None）。

【例 1】

```
def add(a,b):
    c = a + b
    return c
#函数赋值给变量
c = add(3,4)
print(c)
#函数返回值作为其他函数的实际参数
print(add(3,4))
```

运行结果为：

```
7
7
```

本例中，add() 函数既可以用来计算两个数的和，也可以连接两个字符串，它会返回计算的结果。

通过 return 语句指定返回值后，我们在调用函数时，既可以将该函数赋值给一个变量，用

变量保存函数的返回值，也可以将函数再作为某个函数的实际参数。

【例 2】

```
        def isGreater0(x):
            if x > 0:
                return True
            else:
                return False
        print(isGreater0(5))
        print(isGreater0(0))
```

运行结果为：

```
True
False
```

可以看到，函数中可以同时包含多个 return 语句，但需要注意的是，最终真正执行的做多只有 1 个，且一旦执行，函数运行会立即结束。

以上实例中，我们通过 return 语句，都仅返回了一个值，但其实通过 return 语句，可以返回多个值，读者可以阅读《Python 函数返回多个值》一节做详细了解。

10.Python 函数返回多个值

如果程序需要有多个返回值，则既可将多个值包装成列表之后返回，也可直接返回多个值。

如果 Python 函数直接返回多个值，Python 会自动将多个返回值封装成元组。

如下程序示范了函数直接返回多个值的情形：

纯文本复制

```
        def sum_and_avg(list):
            sum = 0
            count = 0
            for e in list:
                # 如果元素 e 是数值
                if isinstance(e, int) or isinstance(e, float):
                    count += 1
                    sum += e
            return sum, sum / count
        my_list = [20, 15, 2.8, 'a', 35, 5.9, -1.8]
        # 获取 sum_and_avg 函数返回的多个值，多个返回值被封装成元组
        tp = sum_and_avg(my_list) #①
        print(tp)
```

上面程序中的第 9 行代码返回了多个值，当 ① 号代码调用该函数时，该函数返回的多个值将会被自动封装成元组，因此程序看到 tp 是一个包含两个元素（由于被调用函数返回了两个值）的元组。

此外，也可使用 Python 提供的序列解包功能，直接使用多个变量接收函数返回的多个值。例如如下代码：

```
        # 使用序列解包来获取多个返回值
        s, avg = sum_and_avg(my_list) #②
        print(s)
        print(avg)
```

上面程序中的 ② 号代码直接使用两个变量来接收 sum_and_avg() 函数返回的两个值，

这就是利用了 Python 提供的序列解包功能。

11.Python 函数递归

在一个函数体内调用它自身，被称为函数递归。函数递归包含了一种隐式的循环，它会重复执行某段代码，但这种重复执行无须循环控制。

例如有如下数学题。已知有一个数列：f(0) = 1，f(1) = 4，f(n + 2) = 2*f(n+ 1) +f(n)，其中 n 是大于 0 的整数，求 f(10) 的值。这道题可以使用递归来求得。下面程序将定义一个 fn() 函数，用于计算 f(10) 的值。

```
1.          def fn(n) :
2.              if n == 0 :
3.                  return 1
4.              elif n == 1 :
5.                  return 4
6.              else :
7.                  # 函数中调用它自身，就是函数递归
8.                  return 2 * fn(n - 1) + fn(n - 2)
9.          # 输出 fn(10)的结果
10.         print("fn(10)的结果是:", fn(10))
```

在上面的 fn() 函数体中再次调用了 fn() 函数，这就是函数递归。注意在 fn() 函数体中调用 fn 的形式：

```
return 2 * fn(n - 1) + fn(n - 2)
```

对于 fn(10)，即等于 2*fn(9)+fn(8)，其中 fn(9) 又等于 2*fn(8)+fn(7)……依此类推，最终会计算到 fn(2) 等于 2*fn(1)+fn(0)，即 fn(2) 是可计算的，这样递归带来的隐式循环就有结束的时候，然后一路反算回去，最后就可以得到 fn(10) 的值。

仔细看上面递归的过程，当一个函数不断地调用它自身时，必须在某个时刻函数的返回值是

确定的，即不再调用它自身：否则，这种递归就变成了无穷递归，类似于死循环。因此，在定义递归函数时有一条最重要的规定： 递归一定要向已知方向进行。

例如，如果把上面数学题改为如此。已知有一个数列：f(20)=1，f(21)=4，f(n + 2)=2*f(n+1)+f(n)，其中 n 是大于 0 的整数，求 f(10) 的值。那么 f(10) 的函数体应该改为如下形式：

```
        def fn(n) :
            if n == 20 :
                return 1
            elif n == 21 :
                return 4
            else :
                # 函数中调用它自身，就是函数递归
                return fn(n + 2) - 2*fn(n + 1)
```

从上面的 fn() 函数来看，当程序要计算 fn(10) 的值时，fn(10) 等于 fn(12)-2*fn(11)，而 fn(11) 等于 fn(13)-2*fn(12)……依此类推，直到 fn(19) 等于 fn(21)-2*fn(20)，此时就可以得到 fn(19) 的值，然后依次反算到 fn(10) 的值。这就是递归的重要规则：对于求 fn(10) 而言，如果 fn(0) 和 fn(1) 是已知的，则应该采用 fn(n)=2*fn(n-1)+fn(n-2) 的形式递归，因为小的一端已知；如果 fn(20) 和 fn(21) 是已知的，则应该采用 fn(n)=fn(n+2)-2*fn(n+1) 的形式递归，因为大的一端已知。

递归是非常有用的，例如程序希望遍历某个路径下的所有文件，但这个路径下的文件夹的深度是未知的，那么就可以使用递归来实现这个需求。系统可定义一个函数，该函数接收一个文件路径作为参数，该函数可遍历出当前路径下的所有文件和文件路径，即在该函数的函数体中再次调用函数自身来处理该路径下的所有文件路径。

总之，只要在一个函数的函数体中调用了函数自身，就是函数递归。递归一定要向已知方向进行。

12.Python 变量作用域（全局变量和局部变量）

在程序中定义一个变量时，这个变量是有作用范围的，变量的作用范围被称为它的作用域。

换句话说，变量的作用域指的是程序代码能够访问该变量的区域，如果超过该区域，将无法访问该变量。

根据定义变量的位置（有效范围），可以将变量分为局部变量和全局变量。

Python 局部变量

局部变量是指在函数内部定义并使用的变量，它只在函数内部有效。

每个函数在执行时，系统都会为该函数分配一块“临时内存空间”，所有的局部变量都被保存在这块临时内存空间内。当函数执行完成后，这块内存空间就被释放了，这些局部变量也就失效了，因此离开函数之后就不能再访问局部变量了，否则解释器会抛出 NameError 错误。

例如：

```
1.          def text():
2.              demo = 'C 语言中文网'
3.              print(demo)
4.          text()
5.          #此处获取局部变量值会引发错误
6.          print('局部变量 demo 的值为：',demo)
```

运行结果为：

```
C 语言中文网
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 5, in <module>
    print('局部变量 demo 的值为：',demo)
NameError: name 'demo' is not defined
```

Python 全局变量

和局部变量相对应，全局变量指的是能作用于函数内外的变量，即全局变量既可以在各个函数的外部使用，也可以在各函数内部使用。

定义全局变量的方式有以下 2 种：

- 在函数体外定义的变量，一定是全局变量，例如：

```
demo = "C 语言中文网"
def text():
    print("函数体内访问：",demo)
text()
print('函数体外访问：',demo)
```

运行结果为：

```
函数体内访问： C 语言中文网
函数体外访问： C 语言中文网
```

- 在函数体内定义全局变量。即使用 global 关键字对变量进行修饰后，该变量就会变为全局变量。例如：

```
def text():
    global demo
    demo = "C 语言中文网"
    print("函数体内访问：",demo)
text()
print('函数体外访问：',demo)
```

运行结果为：

```
函数体内访问： C 语言中文网
函数体外访问： C 语言中文网
```

注意，在使用 global 关键字修饰变量名时，不能直接给变量赋初值，否则会引发语法错误。

获取指定作用域范围中的变量

不管是在函数的局部范围内还是在全局范围内，都可能存在多个变量，每个变量“持有”该变量的值。从这个角度来看，不管是局部范围还是全局范围，这些变量和它们的值就像一个“看不见”的字典，其中变量名就是字典的 key，变量值就是字典的 value。实际上，Python 提供了如下三个工具函数来获取指定范围内的“变量字典”：

1. globals()：该函数返回全局范围内所有变量组成的“变量字典”。
2. locals()：该函数返回当前局部范围内所有变量组成的“变量字典”。
3. vars(object)：获取在指定对象范围内所有变量组成的“变量字典”。如果不传入 object 参数，vars() 和 locals() 的作用完全相同。

globals() 和 locals() 看似完全不同，但它们实际上也是有联系的，关于这两个函数的区别和联系大致有以下两点：

- locals() 总是获取当前局部范围内所有变量组成的“变量字典”，因此，如果在全局范围内（在函数之外）调用 locals() 函数，同样会获取全局范围内所有变量组成的“变量字典”；而 globals() 无论在哪里执行，总是获取全局范围内所有变量组成的“变量字典”。

- 一般来说，使用 locals() 和 globals() 获取的“变量字典”只应该被访问，不应该被修改。但实际上，不管是使用 globals() 还是使用 locals() 获取的全局范围内的“变量字典”，都可以被修改，而这种修改会真正改变全局变量本身：但通过 locals() 获取的局部范围内的“变量字典”，即使对它修改也不会影响局部变量。

下面程序示范了如何使用 locals()、globals() 函数访问局部范围和全局范围内的“变量字典”：

纯文本复制

```
        def test ():
            age = 20
            # 直接访问 age 局部变量
            print(age) # 输出 20
            # 访问函数局部范围的“变量数组”
            print(locals()) # {'age': 20}
            # 通过函数局部范围的“变量数组”访问 age 变量
            print(locals()['age']) # 20
            # 通过 locals 函数局部范围的“变量数组”改变 age 变量的值
            locals()['age'] = 12
            # 再次访问 age 变量的值
            print('xxx', age) # 依然输出 20
            # 通过 globals 函数修改 x 全局变量
            globals()['x'] = 19
        x = 5
        y = 20
        print(globals()) # {..., 'x': 5, 'y': 20}
        # 在全局访问内使用 locals 函数，访问的是全局变量的“变量数组”
        print(locals()) # {..., 'x': 5, 'y': 20}
        # 直接访问 x 全局变量
        print(x) # 5
        # 通过全局变量的“变量数组”访问 x 全局变量
```

```
print(globals()['x']) # 5
# 通过全局变量的“变量数组”对 x 全局变量赋值
globals()['x'] = 39
print(x) # 输出 39
# 在全局范围内使用 locals 函数对 x 全局变量赋值
locals()['x'] = 99
print(x) # 输出 99
```

从上面程序可以清楚地看出，locals() 函数用于访问特定范围内的所有变量组成的“变量字典”，而 globals() 函数则用于访问全局范围内的全局变量组成的“变量字典”。

> 在使用 globals() 或 locals() 访问全局变量的“变量字典”时，将会看到程序输出的“变量字典”默认包含了很多变量，这些都是 Python 主程序内置的，读者暂时不用理会它们。

13.Python 如何在函数中使用同名的全局变量？

我们知道，全局变量默认可以在所有函数内被访问，但是，如果当函数中定义了与全局变量同名的变量时，就会发生局部变量遮蔽（hide）全局变量的情形，例如如下程序：

```
name = 'Charlie'
def test ():
    # 直接访问 name 全局变量
    print(name) # Charlie
test()
print(name)
```

上面程序中，第 4 行直接访问 name 变量，这是允许的，此时程序将会输出 Charlie。

如果在第 4 行代码之后（仍添加在函数内部）再增加如下一行代码：

```
name = '孙悟空'
```

再次运行该程序，将会看到如下错误：

```
UnboundLocalError : local variable  ‘name’  referenced before assignment
```

该错误提示所访问的 name 变量还未定义。这是什么原因呢？这正是由于程序在 test() 函数中增加了“name='孙悟空'”一行代码造成的。

Python 语法规定，在函数内部对不存在的变量赋值时，默认就是重新定义新的局部变量，这会使得函数内部遮蔽重名的 name 全局变量。由于局部变量 name 在 print(name) 后才初始化，所以程序会报错。

为了避免这个问题，可以通过以下 2 种方式来修改上面程序：

1. 访问被遮蔽的全局变量。如果希望程序依然能访问 name 全局变量，且在函数中可重新定义 name 局部变量，也就是在函数中可以访问被遮蔽的全局变量，此时可通过 globals() 函数来实现，将上面程序改为如下形式即可：

```
name = 'Charlie'
def test ():
    # 直接访问 name 全局变量
    print(globals()['name'])  # Charlie
    name = '孙悟空'
test()
print(name)  # Charlie
```

2. 在函数中声明全局变量。为了避免在函数中对全局变量赋值（不是重新定义局部变量），可使用 global 语句来声明全局变量。因此，可将程序改为如下形式：

```
name = '孙悟空'
def test ():
    # 声明 name 是全局变量，后面的赋值语句不会重新定义局部变量
    global name
    # 直接访问 name 全局变量
    print(name)  # 孙悟空
    name = '猪八戒'
test()
print(name)  # 猪八戒
```

增加了“global name”声明之后，程序会把 name 变量当成全局变量，这意味着 test() 函数后面对 name 赋值的语句只是对全局变量赋值，而不是重新定义局部变量。

14.Python 局部函数

前面所看到的函数都是在全局范围内定义的，它们都是全局函数。Python 还支持在函数体内定义函数，这种被放在函数体内定义的函数称为局部函数。

在默认情况下，局部函数对外部是隐藏的，局部函数只能在其封闭（enclosing）函数内有效，其封闭函数也可以返回局部函数，以便程序在其他作用域中使用局部函数。

```
def get_math_func(type, nn) :
    # 定义一个计算平方的局部函数
    def square(n) :  # ①
        return n * n
    # 定义一个计算立方的局部函数
    def cube(n) :  # ②
```

```
            return n * n * n
        # 定义一个计算阶乘的局部函数
        def factorial(n) :   # ③
            result = 1
            for index in range(2, n + 1) :
                result *= index
            return result
        # 调用局部函数
        if type == "square" :
            return square(nn)
        elif type == "cube":
            return cube(nn)
        else:
            return factorial(nn)
print(get_math_func("square", 3)) # 输出 9
print(get_math_func("cube", 3)) # 输出 27
print(get_math_func("", 3)) # 输出 6
```

上面程序中第一行粗体字代码定义了 get_math_func() 函数，接下来程序的 ①、②、③ 号代码定义了 3 个局部函数，而 get_math_func() 函数则根据参数选择调用不同的局部函数。

如果封闭函数没有返回局部函数，那么局部函数只能在封闭函数内部调用，如上面程序所示。另外，还会出现一种情况，如果封闭函数将局部函数返回，且程序使用变量保存了封闭函数的返回值，那么这些局部函数的作用域就会被扩大。因此程序完全可以自由地调用它们，就像它们都是全局函数一样（函数返回函数的情况可阅读《Python 函数高级方法》一节）。

局部函数内的变量也会遮蔽它所在函数内的局部变量（这句话有点拗口），请看如下代码：

```
def foo ():
```

```
        # 局部变量 name
        name = 'Charlie'
        def bar ():
            # 访问 bar 函数所在的 foo 函数的 name 局部变量
            print(name) # Charlie
            name = '孙悟空'
        bar()
    foo()
```

运行上面代码，会导致如下错误：

```
UnboundLocalError: local variable 'name' referenced before assignment
```

该错误是由局部变量遮蔽局部变量导致的，在 bar() 函数中定义的 name 局部变量遮蔽了它所在 foo() 函数内的 name 局部变量，又因为 bar() 函数中的 name 局部变量定义在 print() 输出函数之后，使的 print(name) 执行时找不到合适的 name，因此导致程序报错。

为了声明 bar() 函数中的“name='孙悟空'”赋值语句不是定义新的局部变量，只是访问它所在 foo() 函数内的 name 局部变量，Python 提供了 nonlocal 关键字，通过 nonlocal 语句即可声明访问赋值语句只是访问该函数所在函数内的局部变量。将上面程序改为如下形式：

```
    def foo ():
        # 局部变量 name
        name = 'Charlie'
        def bar ():
         nonlocal name
            # 访问 bar 函数所在的 foo 函数的 name 局部变量
            print(name) # Charlie
```

```
8.                  name = '孙悟空'
9.              bar()
10.         foo()
```

增加上面程序中第 5 行之后，接下来 bar() 函数中的“name='孙悟空'”就不再是定义新的局部变量，而是访问它所在函数（foo() 函数）内的 name 局部变量。

> nonlocal 和前面介绍的 global 功能大致相似，区别只是 global 用于声明访问全局变量，而 nonlocal 用于声明访问当前函数所在函数内的局部变量。

15.Python 函数高级用法

Python 的函数是“一等公民”，因此函数本身也是一个对象，函数既可用于赋值，也可用作其他函数的参数，还可作为其他函数的返回值。

使用函数变量

Python 的函数也是一种值：所有函数都是 function 对象，这意味着可以把函数本身赋值给变量，就像把整数、浮点数、列表、元组赋值给变量一样。

当把函数赋值给变量之后，接下来程序也可通过该变量来调用函数。例如如下代码：

```
1.          # 定义一个计算乘方的函数
2.          def pow(base, exponent) :
3.              result = 1
4.              for i in range(1, exponent + 1) :
5.                  result *= base
```

```
        return result
    # 将 pow 函数赋值给 my_fun，则 my_fun 可当成 pow 使用
    my_fun = pow
    print(my_fun(3 , 4)) # 输出 81
    # 定义一个计算面积的函数
    def area(width, height) :
        return width * height
    # 将 area 函数赋值给 my_fun，则 my_fun 可当成 area 使用
    my_fun = area
    print(my_fun(3, 4)) # 输出 12
```

从上面代码可以看出，程序依次将 pow()、area() 函数赋值给 my_fun 变量，接下来即可通过 my_fun 变量分别调用 pow()、area() 函数。

> 其实 python 已经内置了计算乘方的方法，因此此处的 pow() 函数并没有太大的实际意义，只是作为示范使用。

通过对 my_fun 变量赋值不同的函数，可以让 my_fun 在不同的时间指向不同的函数，从而让程序更加灵活。由此可见，使用函数变量的好处是让程序更加灵活。

除此之外，程序还可使用函数作为另一个函数的形参和（或）返回值。

使用函数作为函数形参

有时候需要定义一个函数，该函数的大部分计算逻辑都能确定，但某些处理逻辑暂时无法确定，这意味着某些程序代码需要动态改变，如果希望调用函数时能动态传入这些代码，那么就需要在函数中定义函数形参，这样即可在调用该函数时传入不同的函数作为参数，从而动态改变这段代码。

Python 支持像使用其他参数一样使用函数参数，例如如下程序：

```
# 定义函数类型的形参，其中 fn 是一个函数
def map(data, fn) :
    result = []
    # 遍历 data 列表中每个元素，并用 fn 函数对每个元素进行计算
    # 然后将计算结果作为新数组的元素
    for e in data :
        result.append(fn(e))
    return result
# 定义一个计算平方的函数
def square(n) :
    return n * n
# 定义一个计算立方的函数
def cube(n) :
    return n * n * n
# 定义一个计算阶乘的函数
def factorial(n) :
    result = 1
    for index in range(2, n + 1) :
        result *= index
    return result
data = [3 , 4 , 9 , 5, 8]
print("原数据: ", data)
# 下面程序代码 3 次调用 map()函数，每次调用时传入不同的函数
print("计算数组元素的平方")
print(map(data , square))
print("计算数组元素的立方")
print(map(data , cube))
print("计算数组元素的阶乘")
print(map(data , factorial))
```

上面程序中定义了一个 map() 函数，该函数的第二个参数是一个函数类型的参数，这意味

着每次调用函数时可以动态传入一个函数，随着实际传入函数的改变，就可以动态改变 map() 函数中的部分计算代码。

Pthon 3 本身也提供了一个 map() 函数，Python 3 内置的 map() 函数的功能和此处定义的 map() 函数功能类似，但更加强大。

接下来的三行粗体字代码调用了 map() 函数三次，三次调用依次传入了 square、cube、factorial 函数作为参数，这样每次调用 map() 函数时实际的执行代码是有区别的。

编译、运行上面程序，可以看到如下输出结果：

```
原数据:  [3, 4, 9, 5, 8]
计算数组元素的平方
[9, 16, 81, 25, 64]
计算数组元素的立方
[27, 64, 729, 125, 512]
计算数组元素的阶乘
[6, 24, 362880, 120, 40320]
```

从上面介绍不难看出，通过使用函数作为参数可以在调用函数时动态传入函数，实际上就可以动态改变被调用函数的部分代码。

在程序最后添加如下一行：

```
# 获取 map 的类型

print(type(map))
```

运行上面代码，将会看到如下输出结果：

```
<class 'function'>
```

使用函数作为返回值（闭包）

前面已经提到，Python 还支持使用函数作为其他函数的返回值。例如如下程序：

纯文本复制

```
        def get_math_func(type) :
            # 定义一个计算平方的局部函数
            def square(n) :  # ①
                return n * n
            # 定义一个计算立方的局部函数
            def cube(n) :  # ②
                return n * n * n
            # 定义一个计算阶乘的局部函数
            def factorial(n) :   # ③
                result = 1
                for index in range(2 , n + 1):
                    result *= index
                return result
            # 返回局部函数
            if type == "square" :
                return square
            if type == "cube" :
                return cube
            else:
                return factorial
        # 调用 get_math_func()，程序返回一个嵌套函数
```

```
22.         math_func = get_math_func("cube") # 得到 cube 函数
23.         print(math_func(5)) # 输出 125
24.         math_func = get_math_func("square") # 得到 square 函数
25.         print(math_func(5)) # 输出 25
26.         math_func = get_math_func("other") # 得到 factorial 函数
27.         print(math_func(5)) # 输出 120
```

程序中，定义了一个 get_math_func() 函数，该函数将返回另一个函数。接下来在 get_math_func() 函数体内的 ①、②、③ 号代码分别定义了三个局部函数，最后 get_math_func() 函数会根据所传入的参数，使用这三个局部函数之一作为返回值。

在定义了会返回函数的 get_math_func() 函数之后，接下来程序调用 get_math_func() 函数时即可返回所需的函数，如上面程序中代码所示。

16.Python 闭包

前面章节中，已经对 Python 闭包做了初步的讲解，本节将详解介绍到底什么是闭包，以及使用闭包有哪些好处。

闭包其实和前面讲的嵌套函数类似，不同之处在于，在嵌套的函数中，外部函数返回的不是一个具体的值，而是一个函数。一般情况下，返回的函数会赋予一个变量，这个变量可以在后面被继续执行调用。

例如，计算一个数的 n 次幂，用闭包可以写成下面的代码：

```
1.          def nth_power(exponent):
2.              def exponent_of(base):
3.                  return base ** exponent
```

```
4.              return exponent_of # 返回值是 exponent_of 函数
5.          square = nth_power(2) # 计算一个数的平方
6.          cube = nth_power(3) # 计算一个数的立方
7.
8.          print(square(2))  # 计算 2 的平方
9.          print(cube(2)) # 计算 2 的立方
```

运行结果为：

```
4

8
```

在上面程序中，外部函数 nth_power() 的返回值是函数 exponent_of()，而不是一个具体的数值。

需要注意的是，在执行完 square = nth_power(2) 和 cube = nth_power(3) 后，外部函数 nth_power() 的参数 exponent 仍然会被内部函数 exponent_of() 记住，这样在之后调用 square(2) 或者 cube(2) 时，程序就能顺利地输出结果，而不会报错说参数 exponent 没有定义。

看到这里，读者可能会问，为什么要闭包呢？上面的程序，完全可以写成下面的形式：

```
1.          def nth_power_rewrite(base, exponent):
2.              return base ** exponent
```

上面程序确实可以实现相同的功能，不过使用闭包，可以让程序变得更简洁易读。设想一下，比如需要计算很多个数的平方，那么读者觉得写成下面哪一种形式更好呢？

```
1.          # 不使用闭包
2.          res1 = nth_power_rewrite(base1, 2)
3.          res2 = nth_power_rewrite(base2, 2)
4.          res3 = nth_power_rewrite(base3, 2)
5.          # 使用闭包
6.          square = nth_power(2)
7.          res1 = square(base1)
8.          res2 = square(base2)
9.          res3 = square(base3)
```

显然第二种方式表达更为简洁，在每次调用函数时，都可以少输入一个参数。

其次，和缩减嵌套函数的优点类似，函数开头需要做一些额外工作，当需要多次调用该函数时，如果将那些额外工作的代码放在外部函数，就可以减少多次调用导致的不必要开销，提高程序的运行效率。

17.Python lambda 表达式（匿名函数）

lambda 表达式（又称匿名函数）是现代编程语言争相引入的一种语法，如果说函数是命名的、方便复用的代码块，那么 lambda 表达式则是功能更灵活的代码块，它可以在程序中被传递和调用。

回顾局部函数

回顾《Python 函数高级用法》一节中，get_math_func() 函数将返回三个局部函数之一。

该函数代码如下：

```
def get_math_func(type) :
    # 定义三个局部函数
    ...
    # 返回局部函数
    if type == "square" :
        return square
    if type == "cube" :
        return cube
    else:
        return factorial
```

由于局部函数的作用域默认仅停留在其封闭函数之内，因此这三个局部函数的函数名的作用太有限了，即仅仅是在程序的 if 语句中作为返回值使用。一旦离开了 get_math_func() 函数体，这三个局部函数的函数名就失去了意义。

既然局部函数的函数名没有太大的意义，那么就考虑使用 lambda 表达式来简化局部函数的写法。

使用 lambda 表达式代替局部函数

如果使用 lambda 表达式来简化 get_math_func() 函数，则可以将程序改写成如下形式：

```
        def get_math_func(type) :
            result=1
            # 该函数返回的是 Lambda 表达式
            if type == 'square':
                return lambda n: n * n  # ①
            elif type == 'cube':
                return lambda n: n * n * n  # ②
            else:
                return lambda n: (1 + n) * n / 2 # ③
        # 调用 get_math_func()，程序返回一个嵌套函数
        math_func = get_math_func("cube")
        print(math_func(5)) # 输出 125
        math_func = get_math_func("square")
        print(math_func(5)) # 输出 25
        math_func = get_math_func("other")
        print(math_func(5)) # 输出 15.0
```

在上面代码中，return 后面的部分使用 lambda 关键字定义的就是 lambda 表达式，Python 要求 lambda 表达式只能是单行表达式。

注意：由于 lambda 表达式只能是单行表达式，不允许使用更复杂的函数形式，因此上面③ 号代码处改为计算 1+2+3+...+n 的总和。

lambda 表达式的语法格式如下：

```
lambda [parameter_list]：表达式
```

从上面的语法格式可以看出 lambda 表达式的几个要点：

- lambda 表达式必须使用 lambda 关键字定义。
- 在 lambda 关键字之后、冒号左边的是参数列表，可以没有参数，也可以有多个参数。如果有多个参数，则需要用逗号隔开，冒号右边是该 lambda 表达式的返回值。

实际上，lambda 表达式的本质就是匿名的、单行函数体的函数。因此，lambda 表达式可以写成函数的形式。

例如，对于如下 lambda 表达式：

```
lambda x , y:x + y
```

可改写为如下函数形式：

```
def add(x, y):
    return x+ y
```

上面定义函数时使用了简化语法：当函数体只有一行代码时，可以直接把函数体的代码放在与函数头同一行。

总体来说，函数比 lambda 表达式的适应性更强，lambda 表达式只能创建简单的函数对象（它只适合函数体为单行的情形）。但 lambda 表达式依然有如下两个用途：

- 对于单行函数，使用 lambda 表达式可以省去定义函数的过程，让代码更加简洁。
- 对于不需要多次复用的函数，使用 lambda 表达式可以在用完之后立即释放，提高了性能。

下面代码示范了通过 lambda 表达式来调用 Python 内置的 map() 函数：

纯文本复制

```
# 传入计算平方的 lambda 表达式作为参数
x = map(lambda x: x*x , range(8))
print([e for e in x]) # [0, 1, 4, 9, 16, 25, 36, 49]
# 传入计算平方的 lambda 表达式作为参数
y = map(lambda x: x*x if x % 2 == 0 else 0, range(8))
print([e for e in y]) # [0, 0, 4, 0, 16, 0, 36, 0]
```

正如从上面代码所看到的，内置的 map() 函数的第一个参数需要传入函数，此处传入了函数的简化形式：lambda 表达式，这样程序更加简洁，而且性能更好。

总结

本节所介绍的 lambda 表达式是 Python 编程的核心机制之一。Python 语言既支持面向过程编程，也支持面向对象编程。而 lambda 表达式是 Python 面向过程编程的语法基础，因此读者必须引起重视。

Python 的 lambda 表达式只是单行函数的简化版本，因此 lambda 表达式的功能比较简单。

18.Python 函数式编程（map()、filter()和 reduce()）详解

所谓函数式编程，是指代码中每一块都是不可变的，都由纯函数的形式组成。这里的纯函数，是指函数本身相互独立、互不影响，对于相同的输入，总会有相同的输出。

除此之外，函数式编程还具有一个特点，即允许把函数本身作为参数传入另一个函数，还允许返回一个函数。

例如，想让列表中的元素值都变为原来的两倍，可以使用如下函数实现：

```
1.          def multiply_2(list):
2.              for index in range(0, len(list)):
3.                  list[index] *= 2
4.              return list
```

需要注意的是，这段代码不是一个纯函数的形式，因为列表中元素的值被改变了，如果多次调用 multiply_2() 函数，那么每次得到的结果都不一样。

而要想让 multiply_2() 成为一个纯函数的形式，就得重新创建一个新的列表并返回，也就

是写成下面这种形式：

```
        def multiply_2_pure(list):
            new_list = []
            for item in list:
                new_list.append(item * 2)
            return new_list
```

函数式编程的优点，主要在于其纯函数和不可变的特性使程序更加健壮，易于调试和测试；缺点主要在于限制多，难写。

注意，纯粹的函数式编程语言（比如 Scala），其编写的函数中是没有变量的，因此可以保证，只要输入是确定的，输出就是确定的；而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出。

Python 允许使用变量，所以它并不是一门纯函数式编程语言。Python 仅对函数式编程提供了部分支持，主要包括 map()、filter() 和 reduce() 这 3 个函数，它们通常都结合 lambda 匿名函数一起使用。接下来就对这 3 个函数的用法做逐一介绍。

Python map()函数

map() 函数的基本语法格式如下：

```
map(function, iterable)
```

其中，function 参数表示要传入一个函数，其可以是内置函数、自定义函数或者 lambda 匿名函数；iterable 表示一个或多个可迭代对象，可以是列表、字符串等。

map() 函数的功能是对可迭代对象中的每个元素 ,都调用指定的函数 ,并返回一个 map 对象。

注意，该函数返回的是一个 map 对象，不能直接输出，可以通过 for 循环或者 list() 函数来显示。

【例 1】还是对列表中的每个元素乘以 2。

```
listDemo = [1, 2, 3, 4, 5]
new_list = map(lambda x: x * 2, listDemo)
print(list(new_list))
```

运行结果为：

```
[2， 4， 6， 8， 10]
```

【例 2】map() 函数可传入多个可迭代对象作为参数。

```
listDemo1 = [1, 2, 3, 4, 5]
listDemo2 = [3, 4, 5, 6, 7]
new_list = map(lambda x,y: x + y, listDemo1,listDemo2)
print(list(new_list))
```

运行结果为：

```
[4, 6, 8, 10, 12]
```

注意，由于 map() 函数是直接由用 C 语言写的，运行时不需要通过 Python 解释器间接调用，并且内部做了诸多优化，所以相比其他方法，此方法的运行效率最高。

Python filter()函数

filter()函数的基本语法格式如下：

```
filter(function, iterable)
```

此格式中，funcition 参数表示要传入一个函数，iterable 表示一个可迭代对象。

filter() 函数的功能是对 iterable 中的每个元素，都使用 function 函数判断，并返回 True 或者 False，最后将返回 True 的元素组成一个新的可遍历的集合。

【例 3】返回一个列表中的所有偶数。

```
1.          listDemo = [1, 2, 3, 4, 5]
2.          new_list = filter(lambda x: x % 2 == 0, listDemo)
3.          print(list(new_list))
```

运行结果为：

```
[2, 4]
```

【例 4】filter() 函数可以接受多个可迭代对象。

```
1.          listDemo = [1, 2, 3, 4, 5]
2.          new_list = map(lambda x,y: x-y>0,[3,5,6],[1,5,8] )
3.          print(list(new_list))
```

运行结果为：

```
[True, False, False]
```

Python reduce()函数

reduce() 函数通常用来对一个集合做一些累积操作，其基本语法格式为：

```
reduce(function, iterable)
```

其中，function 规定必须是一个包含 2 个参数的函数；iterable 表示可迭代对象。

注意，由于 reduce() 函数在 Python 3.x 中已经被移除，放入了 functools 模块，因此在使用该函数之前，需先导入 functools 模块。

【例 5】计算某个列表元素的乘积。

```
import functools
listDemo = [1, 2, 3, 4, 5]
product = functools.reduce(lambda x, y: x * y, listDemo)
print(product)
```

运行结果为：

```
120
```

总结

通常来说，当对集合中的元素进行一些操作时，如果操作非常简单，比如相加、累积这种，那么应该优先考虑使用 map()、filter()、reduce() 实现。另外，在数据量非常多的情况下

（比如机器学习的应用），一般更倾向于函数式编程的表示，因为效率更高。

当然，在数据量不多的情况下，使用 for 循环等方式也可以。不过，如果要对集合中的元素做一些比较复杂的操作，考虑到代码的可读性，通常会使用 for 循环。

19.函数注解，号称 Python3 新增的最独特的功能！

函数注解是 Python 3 最独特的功能之一，关于它的介绍，官方文档是这么说的，“函数注解是关于用户自定义函数使用类型的完全可选的元信息”。也就是说，官方将函数注解的用途归结为：为函数中的形参和返回值提供类型提示信息。

下面是对 Python 官方文档中的示例稍作修改后的程序，可以很好的展示如何定义并获取函数注解：

```
1.          def f(ham:str,egg:str='eggs')->str:
2.            pass
3.          print(f.__annotations__)
```

输出结果为：

```
{'ham': <class 'str'>, 'egg': <class 'str'>, 'return': <class 'str'>}
```

如上所示，给函数中的参数做注解的方法是在形参后添加冒号“：”，后接需添加的注解（可以是类（如 str、int 等），也可以是字符串或者表示式）；给返回值做注解的方法是将注解添加到 def 语句结尾的冒号和 -> 之间。

注意，如果参数有默认值，参数注解位于冒号和等号之间。比如 eggs:str='eggs'，它表示 eggs 参数的默认值为 'eggs'，添加的注解为 str。

给函数定义好注解之后，可以通过函数对象的 __annotations__ 属性获取，它是一个字典，在应用运行期间可以获取。这里再举一个例子：

```
1.          def square(number:"一个数字")->"返回 number 的平方":
2.            return number**2
3.          print(square(10))
4.          print(square.__annotations__)
```

运行结果为：

```
100
{'number': '一个数字', 'return': '返回 number 的平方'}
```

事实上，函数注解并不局限于类型提示，而且在 Python 及其标准库中也没有单个功能可以利用这种注解，这也是这个功能独特的原因。

注意，函数注解没有任何语法上的意义，只是为函数参数和返回值做注解，并在运行获取这些注解，仅此而已。换句话说，为函数做的注解，Python 不做检查，不做强制，不做验证，什么操作都不做，函数注解对 Python 解释器没任何意义。

函数注解可能的用法

PEP 3107 作为提议函数注解的官方文档，其中列出了以下可能的使用场景：

- 提供类型信息：包括类型检查、让 IDE 显示函数接受和返回的类型、适配、与其他语言的桥梁、数据库查询映射、RPC 参数编组等；
- 其他信息：函数参数和返回值的文档。

总之，虽然函数注解存在的时间和 Python 3 一样长，但目前仍未找到任一常见且积极维护的包，将函数注解用作类型检查之外的功能。Python 3 最初发布时包含函数注解的最初目的也仅是用于试验和玩耍。

20.如何才能提高代码颜值，让代码变得有逼格？

学习过程中，我们经常会阅读他人写的代码，如果注意观察就会发现，好的代码本身就是一份文档，解决同样的问题，不同的人编写的代码，其可读性千差万别。

有些人的设计风格和代码风格犹如热刀切黄油，从顶层到底层的代码看下来酣畅淋漓，注释详尽又精简；深入到细节代码，无需注释也能理解清清楚楚。而有些人，代码勉勉强强能跑起来，遇到稍微复杂的情况就会出崩溃，且代码中处处都是堆积在一起的变量、函数和类，很难理清代码的实现思路。

Python 创始人 Guido van Rossum（吉多·范罗苏姆）说过，代码的阅读频率远高于编写代码的频率。毕竟是在编写代码的时候，我们自己也需要对代码进行反复阅读和调试，来确认代码能够按照期望运行。

本节，在读者学会如何使用 Puython 函数的基础上，教大家怎么才能合理分解代码，提高代码的可读性。

首先，大家在编程过程中，一定要围绕一个中心思想：不写重复性的代码。因为，重复代码往往是可以通过使用条件、循环、构造函数和类（后续章节会做详细介绍）来解决的。

例如，仔细观察下面的代码：

```
1.          if i_am_rich:
2.              money = 100
3.              send(money)
4.          else:
5.              money = 10
6.              send(money)
```

这段代码中，同样的 send 语句出现了两次，其实它完全可以进行合并，把代码改造成下面这样：

```
1.          if i_am_rich:
```

```
        money = 100
    else:
        money = 10
    send(money)
```

与此同时，还要学会刻意地减少代码的迭代层数，尽可能让 Python 代码扁平化。例如：

```
def send(money):
    if is_server_dead:
        LOG('server dead')
        return
    else:
        if is_server_timed_out:
            LOG('server timed out')
            return
        else:
            result = get_result_from_server()
            if result == MONEY_IS_NOT_ENOUGH:
                LOG('you do not have enough money')
                return
            else:
                if result == TRANSACTION_SUCCEED:
                    LOG('OK')
                    return
                else:
                    LOG('something wrong')
                    return
```

上面这段代码层层缩进，如果我们没有比较强的逻辑分析能力，理清这段代码是比较困难。其实，这段代码完全可以改成如下这样：

```
def send(money):
    if is_server_dead:
        LOG('server dead')
        return
```

```
5.
6.              if is_server_timed_out:
7.                  LOG('server timed out')
8.                  return
9.
10.             result = get_result_from_server()
11.
12.             if result == MONET_IS_NOT_ENOUGH:
13.                 LOG('you do not have enough money')
14.                 return
15.
16.             if result == TRANSACTION_SUCCEED:
17.                 LOG('OK')
18.                 return
19.
20.             LOG('something wrong')
```

可以看到，所有的判断语句都位于同一层级，同之前的代码格式相比，代码层次清晰了很多。

另外，在使用函数时，函数的粒度应该尽可能细，不要让一个函数做太多的事情。往往一个复杂的函数，我们要尽可能地把它拆分成几个功能简单的函数，然后合并起来。

如何拆分函数呢？这里，举一个二分搜索的例子。给定一个非递减整数数组，和一个 target 值，要求你找到数组中最小的一个数 x，满足 x*x > target，如果不存在，则返回 -1。

大家不妨先独立完成，写完后再对照着来看下面的代码，找出自己的问题：

```
1.          def solve(arr, target):
2.              l, r = 0, len(arr) - 1
3.              ret = -1
4.              while l <= r:
5.                  m = (l + r) // 2
6.                  if arr[m] * arr[m] > target:
7.                      ret = m
8.                      r = m - 1
9.                  else:
10.                     l = m + 1
11.             if ret == -1:
```

```
                return -1
            else:
                return arr[ret]

        print(solve([1, 2, 3, 4, 5, 6], 8))
        print(solve([1, 2, 3, 4, 5, 6], 9))
        print(solve([1, 2, 3, 4, 5, 6], 0))
        print(solve([1, 2, 3, 4, 5, 6], 40))
```

对于上面这样的写法，应付算法比赛和面试已经绰绰有余。但如果从工程的角度考虑，还需要进行深度优化：

```
        def comp(x, target):
            return x * x > target

        def binary_search(arr, target):
            l, r = 0, len(arr) - 1
            ret = -1
            while l <= r:
                m = (l + r) // 2
                if comp(arr[m], target):
                    ret = m
                    r = m - 1
                else:
                    l = m + 1
            return ret

        def solve(arr, target):
            id = binary_search(arr, target)
            if id != -1:
                return arr[id]
            return -1

        print(solve([1, 2, 3, 4, 5, 6], 8))
        print(solve([1, 2, 3, 4, 5, 6], 9))
        print(solve([1, 2, 3, 4, 5, 6], 0))
```

```
25.         print(solve([1, 2, 3, 4, 5, 6], 40))
```

在这段代码中，我们把不同功能的代码单独提取出来作为独立的函数。其中，comp() 函数作为核心判断，提取出来之后，可以让整个程序更清晰；同时，还把二分搜索的主程序提取了出来，只负责二分搜索；最后的 solve() 函数拿到结果，决定返回不存在，还是返回值。这样一来，每个函数各司其职，阅读性也能得到一定提高。

第 8 章 Python 类和对象

在设计之初，Python 就被设计成支持面向对象的编程语言，因此 Python 完全能以面向对象的方式编程。而且 Python 的面向对象比较简单，它不像其他面向对象语言提供了大量繁杂的面向对象特征，它致力于提供简单、够用的语法功能。

正因为如此，在 Python 中创建一个类和对象都很容易。Python 支持面向对象的三大特征：封装、继承和多态，子类继承父类同样可以继承到父类的变量和方法。

1.Python 面向对象

读者肯定听过 Python 中“一切皆对象”的说法，但可能并不了解它的具体含义，只是在学习的时候听说 Python 是面向对象的编程语言，本节将向大家详细介绍 Python 面向对象的含义。

面向对象编程是在面向过程编程的基础上发展来的，它比面向过程编程具有更强的灵活性和扩展性。面向对象编程是程序员发展的分水岭，很多初学者会因无法理解面向对象而放弃学习编程。

面向对象编程（Object-oriented Programming，简称 OOP），是一种封装代码的方法。其实，在前面章节的学习中，我们已经接触了封装，比如说，将乱七八糟的数据扔进列表中，这就是一种简单的封装，是数据层面的封装；把常用的代码块打包成一个函数，这也是一种封装，是语句层面的封装。

代码封装，其实就是隐藏实现功能的具体代码，仅留给用户使用的接口，就好像使用计算机，用户只需要使用键盘、鼠标就可以实现一些功能，而根本不需要知道其内部是如何工作的。

本节所讲的面向对象编程，也是一种封装的思想，不过显然比以上两种封装更先进，它可以更好地模拟真实世界里的事物（将其视为对象），并把描述特征的数据和代码块（函数）封装到一起。

打个比方，若在某游戏中设计一个乌龟的角色，应该如何来实现呢？使用面向对象的思想会更简单，可以分为如下两个方面进行描述：

1. 从表面特征来描述，例如，绿色的、有 4 条腿、重 10 kg、有外壳等等。
2. 从所具有的的行为来描述，例如，它会爬、会吃东西、会睡觉、会将头和四肢缩到壳里，等等。

如果将乌龟用代码来表示，则其表面特征可以用变量来表示，其行为特征可以通过建立各种函数来表示。参考代码如下所示：

```
class tortoise:
    bodyColor = "绿色"
    footNum = 4
    weight = 10
    hasShell = True

    #会爬
    def crawl(self):
        print("乌龟会爬")
```

```
10.             #会吃东西
11.             def eat(self):
12.                 print("乌龟吃东西")
13.             #会睡觉
14.             def sleep(self):
15.                 print("乌龟在睡觉")
16.             #会缩到壳里
17.             def protect(self):
18.                 print("乌龟缩进了壳里")
```

> 注意，以上代码仅是为了演示面向对象的编程思想，具体细节后续会做详细介绍。

因此，从某种程序上，相比较只用变量或只用函数，使用面向对象的思想可以更好地模拟现实生活中的事物。

不仅如此，在 Python 中，所有的变量其实也都是对象，包括整形（int）、浮点型（float）、字符串（str）、列表(list)、元组(tuple)、字典（dict）和集合（set）。以字典（dict）为例，它包含多个函数供我们使用，例如使用 keys() 获取字典中所有的键，使用 values() 获取字典中所有的值，使用 item() 获取字典中所有的键值对，等等。

面向对象相关术语

在系统学习面向对象编程之前，初学者要了解有关面向对象的一些术语。当和其他人讨论代码的时候，或者尝试查找我们遇到的问题的解决方案时，知道正确的术语会很有帮助。

面向对象中，常用术语包括：

类：可以理解是一个模板，通过它可以创建出无数个具体实例。比如，前面编写的 tortoise 表示的只是乌龟这个物种，通过它可以创建出无数个实例来代表各种不同特征的乌龟（这一

过程又称为类的实例化）。
对象：类并不能直接使用，通过类创建出的实例（又称对象）才能使用。这有点像汽车图纸和汽车的关系，图纸本身（类）并不能为人们使用，通过图纸创建出的一辆辆车（对象）才能使用。
属性：类中的所有变量称为属性。例如，tortoise 这个类中，bodyColor、footNum、weight、hasShell 都是这个类拥有的属性。
方法：类中的所有函数通常称为方法。不过，和函数所有不同的是，类方法至少要包含一个 self 参数（后续会做详细介绍）。例如，tortoise 类中，crawl()、eat()、sleep()、protect() 都是这个类所拥有的方法，类方法无法单独使用，只能和类的对象一起使用。

2.Python class：定义类

前面章节中已经提到，类仅仅充当图纸的作用，本身并不能直接拿来用，而只有根据图纸造出的实际物品（对象）才能直接使用。因此，Python 中使用类的顺序是：先创建（定义）类，然后再创建类的实例对象，通过实例对象实现特定的功能。本节来先学习如何创建一个类。

Python 中，创建一个类使用 class 关键字实现，其基本语法格式如下：

```
class 类名：
    零个到多个类属性...
    零个到多个类方法...
```

注意，类中属性和方法所在的前后顺序没有任何影响，且各成员之间可以相互调用。

类名只要是一个合法的标识符即可，但这仅仅满足的是 Python 的语法要求：如果从程序的可读性方面来看，Python 的类名必须是由一个或多个有意义的单词连缀而成的，每个单词首字母大写，其他字母全部小写，单词与单词之间不要使用任何分隔符，例如类名为

“TheFirstDemo”。

从上面定义来看，Python 的类定义有点像函数定义，都是以冒号（:）作为类体的开始，以统一缩进的部分作为类体的。区别只是函数定义使用 def 关键字，而类定义则使用 class 关键字。

Python 的类定义由类头（指 class 关键字和类名部分）和统一缩进的类体构成，在类体中最主要的两个成员就是属性和方法。如果不为类定义任何属性和方法，那么这个类就相当于一个空类，如果空类不需要其他可执行语句，则可使用 pass 语句作为占位符。例如，如下类定义是允许的：

```
class Empty:
    pass
```

通常来说，空类没有太大的实际意义。

下面程序将定义一个 Person 类：

```
class Person :
    '''这是一个学习 Python 定义的一个 Person 类'''
    # 下面定义了一个类属性
    hair = 'black'
    # 下面定义了一个 say 方法
    def say(self, content):
        print(content)
```

与函数类似的是，Python 也允许为类定义说明文档，该文档同样被放在类声明之后、类体之前，如上面程序中第二行的字符串所示。

可以看到，上面代码创建了一个名为 Person 的类，该类中创建了一个名为 hair 的类属性，需要注意的是，根据定义属性位置的不同，在各个类方法之外定义的称为类属性或类变量（如 hair 属性），而在类方法中定义的属性称为实例属性（或实例变量），它们的区别和用法可阅读《Python 类变量和实例变量》一节。

不仅如此，Person 类中还创建了一个 say() 类方法，细心的读者可能已经看到，该方法包含两个参数，分别是 self 和 content。可以肯定的是，content 参数就只是一个普通参数，没有特殊含义，但 self 比较特殊，并不是普通的参数，它的作用和用法会在后续章节中详细介绍。

> 注意，更确切地说，say() 是一个实例方法，除此之外，Python 类中还可以定义类方法和静态方法，这 3 种类方法的区别和具体用法，可阅读《Python 实例方法、静态方法和类方法》。

3.Python __init__()类构造方法

在创建类时，我们可以手动添加一个 __init__() 方法，该方法是一个特殊的类实例方法，称为构造方法（或构造函数）。

构造方法用于创建对象时使用，每当创建一个类的实例对象时，Python 解释器都会自动调用它。Python 类中，手动添加构造方法的语法格式如下：

```
def __init__(self,...):
    代码块
```

注意，此方法的方法名中，开头和结尾各有 2 个下划线，且中间不能有空格。Python 中很多这种以双下划线开头、双下划线结尾的方法，都具有特殊的意义，教程后面还会详细介绍这些特殊的方法。

另外，__init__() 方法可以包含多个参数，但必须包含一个名为 self 的参数，且必须作为第一个参数。也就是说，类的构造方法最少也要有一个 self 参数。

例如，仍以 Person 类为例，添加构造方法的代码如下所示：

```
1.          class Person :
2.              '''这是一个学习 Python 定义的一个 Person 类'''
3.              def __init__(self):
4.                  print("调用构造方法")
```

注意，如果开发者没有为该类定义任何构造方法，那么 Python 会自动为该类创建一个只包含 self 参数的默认的构造方法。

在以上代码的基础上，添加如下代码：

```
1.          zhangsan = Person()
```

这行代码的含义是创建一个名为 zhangsan 的 Person 类对象。运行代码可看到如下结果：

调用构造方法

显然，在创建 zhangsan 这个对象时，隐式调用了类的构造方法。

不仅如此，在 __init__() 构造方法中，除了 self 参数外，还可以自定义一些参数，参数之间使用逗号“,”进行分割。例如，下面的代码在创建 __init__() 方法时，额外指定了 2 参数，分别是 name 和 age：

```
class Person :
    '''这是一个学习 Python 定义的一个 Person 类'''
    def __init__(self,name,age):
        print("这个人的名字是：",name," 年龄为：",age)
#创建 zhangsan 对象，并传递参数给构造函数
zhangsan = Person("张三",20)
```

注意，由于创建对象时会调用类的构造方法，如果构造函数有多个参数时，需要手动传递参数，传递方式如代码中所示（后续章节会做详细讲解）。

运行以上代码，执行结果为：

```
这个人的名字是： 张三  年龄为： 20
```

可以看到，虽然构造方法中有 self、name、age 3 个参数，但实际需要传参的仅有 name 和 age，也就是说，self 不需要手动传递参数。

关于 self 参数，后续章节会做详细介绍，这里只需要知道，在创建类对象时，无需给 self 传参即可。

4.Python 类对象的创建和使用

使用 class 语句只能创建一个类，而无法创建类的对象，因此要想使用已创建好的类，还需要手动创建类的对象，创建类对象的过程又称为类的实例化。

对已创建的类进行实例化，其语法格式如下：

```
类名(参数)
```

当创建类时，若没有显式创建 __init()__ 构造方法或者该构造方法中只有一个 self 参数，则创建类对象时的参数可以省略不写。

例如，如下代码创建了名为 Python 的类，并对其进行了实例化：

```
class Person :
    '''这是一个学习 Python 定义的一个 Person 类'''
    # 下面定义了 2 个类变量
    name = "zhangsan"
    age = "20"
    def __init__(self,name,age):
        #下面定义 2 个实例变量
        self.name = name
        self.age = age
        print("这个人的名字是：",name," 年龄为：",age)
    # 下面定义了一个 say 实例方法
    def say(self, content):
        print(content)
# 将该 Person 对象赋给 p 变量
p = Person("张三",20)
```

在上面的程序中，由于构造方法除 self 参数外，还包含 2 个参数，且这 2 个参数没有设置默认参数，因此在实例化类对象时，需要传入相应的 name 值和 age 值（self 参数是特殊参数，不需要手动传值，Python 会自动传给它值）。

> 类变量和实例变量，简单地理解，定义在各个类方法之外（包含在类中）的变量为类变量（或者类属性），定义在类方法中的变量为实例变量（或者实例属性），二者的具体区别和用法可阅读《Python 类变量和实例变量》

Python 类对象的使用

创建对象之后，接下来即可使用该对象了。Python 的对象大致有以下作用：

- 操作对象的实例变量，包括访问、修改实例变量的值、以及给对象添加或删除实例变量）。
- 调用对象的方法，包括调用对象的方法，已经给对象动态添加方法。

类对象访问变量或方法

使用已创建好的类对象访问类中实例变量的语法格式如下：

```
对象名.变量名
```

使用类对象调用类中方法的语法格式如下：

```
对象名.方法名(参数)
```

注意，对象名和变量名以及方法名之间用点 "." 连接。

下面代码通过 Person 对象来调用 Person 的实例变量和方法：

```
        # 输出 p 的 name、age 实例变量
        print(p.name, p.age)
        # 访问 p 的 name 实例变量，直接为该实例变量赋值
        p.name = '李刚'
        # 调用 p 的 say()方法，声明 say()方法时定义了 2 个形参，但第一个形参（self）不需要传值，因此调用该方法只需为第二个形参指定一个值
        p.say('Python 语言很简单，学习很容易！')
        # 再次输出 p 的 name、age 实例变量
        print(p.name, p.age)
```

程序运行结果为：

```
这个人的名字是： 张三  年龄为： 20
张三 20
Python 语言很简单，学习很容易！
李刚 20
```

给类对象动态添加变量

Python 支持为已创建好的对象动态增加实例变量，方法也很简单，只要为它的新变量赋值即可，比如说：

```
        # 为 p 对象增加一个 skills 实例变量
        p.skills = ['programming', 'swimming']
        print(p.skills)
```

运行结果为：

```
['programming', 'swimming']
```

上面程序为 p 对象动态增加了一个 skills 实例变量，即只要对 p 对象的 skills 实例变量赋值就是新增一个实例变量。也可以动态删除实例变量，使用 del 语句即可删除，例如如下代码：

```
# 删除 p 对象的 name 实例变量
del p.name
# 再次访问 p 的 name 实例变量
print(p.name)
```

程序中调用 del 删除了 p 对象的 name 实例变量，但由于类中还有同名的 name 类变量，因此程序不会报错；否则会导致 AttributeError 错误，并提示：'Person' object has no attribute 'name'。

给类对象动态添加方法

注意，初学者在理解下面内容之前，需明白 self 参数的含义和作用，可阅读《Python self 用法》详细了解。

Python 也允许为对象动态增加方法。比如上面程序中在定义 Person 类时只定义了一个 say() 方法，但程序完全可以为 p 对象动态增加方法。

但需要说明的是，为 p 对象动态增加的方法，Python 不会自动将调用者自动绑定到第一个参数（即使将第一个参数命名为 self 也没用）。例如如下代码：

```
# 先定义一个函数
```

```
2.          def info(self):
3.              print("---info 函数---", self)
4.          # 使用 info 对 p 的 foo 方法赋值（动态绑定方法）
5.          p.foo = info
6.          # Python 不会自动将调用者绑定到第一个参数，
7.          # 因此程序需要手动将调用者绑定为第一个参数
8.          p.foo(p)  # ①
9.
10.         # 使用 lambda 表达式为 p 对象的 bar 方法赋值（动态绑定方法）
11.         p.bar = lambda self: print('--lambda 表达式--', self)
12.         p.bar(p) # ②
```

上面的第 5 行和第 11 行代码分别使用函数、lambda 表达式为 p 对象动态增加了方法，但对于动态增加的方法，Python 不会自动将方法调用者绑定到它们的第一个参数，因此程序必须手动为第一个参数传入参数值，如上面程序中 ① 号、② 号代码所示。

如果希望动态增加的方法也能自动绑定到第一个参数，则可借助于 types 模块下的 MethodType 进行包装。例如如下代码：

纯文本复制

```
1.          def intro_func(self, content):
2.              print("我是一个人，信息为：%s" % content)
3.          # 导入 MethodType
4.          from types import MethodType
5.          # 使用 MethodType 对 intro_func 进行包装，将该函数的第一个参数绑定为 p
6.          p.intro = MethodType(intro_func, p)
7.          # 第一个参数已经绑定了，无需传入
8.          p.intro("生活在别处")
```

正如从上面代码所看到的，通过 MethodType 包装 intr_func 函数之后（包装时指定了将该函数的第一个参数绑定为 p），为 p 对象动态增加的 intro() 方法的第一个参数已经绑定，因此程序通过 p 调用 intro() 方法时无须传入第一个参数，就像定义类时已经定义了 intro() 方法一样。

5.Python self

在学习如何定义类的过程中，无论是显式创建类的构造方法，还是向类中添加实例方法，都要求将 self 参数作为方法的第一个参数。例如，定义如下 Dog 类：

```
class Dog:
    def __init__(self):
        print("正在执行构造方法")
    # 定义一个 jump()实例方法
    def jump(self):
        print("正在执行 jump 方法")
```

本节将对 self 参数做详细的讲解。

Python 要求，类方法（构造方法和实例方法）中至少要包含一个参数，但并没有规定此参数的名称（完全可以叫任意参数名），之所以将类方法的第一个参数命名为 self，只是 Python 程序员约定俗成的一种习惯，这会使程序具有更好的可读性。

那么，作为类方法的第一个参数，self 参数的具体作用是什么呢？打个比方，如果把类比作造房子的图纸，那么对类实例化后的对象才是真正可以住的房子，根据一张图纸，我们可以设计出成千上万的房子，虽然每个房子长相相似，但它们都有各自的主人。而类方法的 self 参数，就相当于每个房子的门钥匙，它可以保证，每个房子的主人仅能进入自己的房子。

如果你接触过其他面向对象的编程语言（例如 C++），其实 Python 类方法中的 self 参数就相当于 C++ 中的 this 指针。

也就是说，同一个类可以产生多个对象，当某个对象调用类方法时，该对象会把自身的引用作为第一个参数自动传给该方法，换句话说，Python 会自动绑定类方法的第一个参数指向调用该方法的对象。如此，Python 解释器就能知道到底要操作哪个对象的方法了。

> 对于构造方法来说，self 参数（第一个参数）代表该构造方法正在初始化的对象。

因此，程序在调用实例方法和构造方法时，不需要为第一个参数传值。例如，更改前面的 Dog 类，如下所示：

```
class Dog:
    def __init__(self):
        print(self,"在调用构造方法")
    # 定义一个 jump()方法
    def jump(self):
        print(self,"正在执行 jump 方法")
    # 定义一个 run()方法，run()方法需要借助 jump()方法
    def run(self):
        print(self,"正在执行 run 方法")
        # 使用 self 参数引用调用 run()方法的对象
        self.jump()
dog1 = Dog()
dog1.run()
dog2 = Dog()
dog2.run()
```

上面代码中，jump() 和 run() 中的 self 代表该方法的调用者，即谁在调用该方法，那么 self 就代表谁，因此，该程序的运行结果为：

```
<__main__.Dog object at 0x00000276B14B12B0> 在调用构造方法
<__main__.Dog object at 0x00000276B14B12B0> 正在执行 run 方法
```

```
<__main__.Dog object at 0x00000276B14B12B0> 正在执行 jump 方法
<__main__.Dog object at 0x00000276B14B1F28> 在调用构造方法
<__main__.Dog object at 0x00000276B14B1F28> 正在执行 run 方法
<__main__.Dog object at 0x00000276B14B1F28> 正在执行 jump 方法
```

上面程序中值得一提的是，当一个 Dog 对象调用 run() 方法时，run() 方法需要依赖该对象自己的 jump() 方法。在现实世界里，对象的一个方法依赖另一个方法的情形很常见，例如，吃饭方法依赖拿筷子方法，写程序方法依赖敲键盘方法，这种依赖都是同一个对象的两个方法之间的依赖。

注意，当 Python 对象的一个方法调用另一个方法时，不可以省略 self。也就是说，将上面的 run()方法改为如下形式是不正确的：

```
1.          # 定义一个 run()方法，run()方法需要借助 jump()方法
2.          def run():
3.              #省略 self，代码会报错
4.              self.jump()
5.              print("正在执行 run 方法")
```

再比如，分析如下代码：

```
1.          class InConstructor :
2.              def __init__(self) :
3.                  # 在构造方法里定义一个 foo 变量（局部变量）
4.                  foo = 0
5.                  # 使用 self 代表该构造方法正在初始化的对象
```

```
6.                  # 下面的代码将会把该构造方法正在初始化的对象的 foo 实例变量设为
    6
7.                  self.foo = 6
8.          # 所有使用 InConstructor 创建的对象的 foo 实例变量将被设为 6
9.          print(InConstructor().foo) # 输出 6
```

在 InConstructor 的构造方法中，self 参数总是引用该构造方法正在初始化的对象。程序中将正在执行初始化的 InConstructor 对象的 foo 实例变量设为 6，这意味着该构造方法返回的所有对象的 foo 实例变量都等于 6。

需要说明的是，自动绑定的 self 参数并不依赖具体的调用方式，不管是以方法调用还是以函数调用的方式执行它，self 参数一样可以自动绑定。例如如下程序：

```
1.          class User:
2.              def test(self):
3.                  print('self 参数: ', self)
4.          u = User()
5.          # 以方法形式调用 test()方法
6.          u.test() # <__main__.User object at 0x00000000021F8240>
7.          # 将 User 对象的 test 方法赋值给 foo 变量
8.          foo = u.test
9.          # 通过 foo 变量（函数形式）调用 test()方法。
10.         foo() # <__main__.User object at 0x00000000021F8240>
```

上面程序中，第 6 行代码以方法形式调用 User 对象的 test() 方法，此时方法调用者当然会自动绑定到方法的第一个参数(self 参数) 程序中第 10 行代码以函数形式调用 User 对象的 test() 方法，看上去没有调用者，但程序依然会把实际调用者绑定到方法的第一个参数，因此输出结果完全相同。

当 self 参数作为对象的默认引用时，程序可以像访问普通变量一样来访问这个 self 参数，甚至可以把 self 参数当成实例方法的返回值。看下面程序：

```
class ReturnSelf :
    def grow(self):
        if hasattr(self, 'age'):
            self.age += 1
        else:
            self.age = 1
        # return self 返回调用该方法的对象
        return self
rs = ReturnSelf()
# 可以连续调用同一个方法
rs.grow().grow().grow()
print("rs 的 age 属性值是:", rs.age)
```

可以看出，如果在某个方法中把 self 参数作为返回值，则可以多次连续调用同一个方法，从而使得代码更加简洁。但是这种把 self 参数作为返回值的方法可能会造成实际意义的模糊，例如上面的 grow 方法用于表示对象的生长，即 age 属性的值加 1，实际上不应该有返回值。

6.Python 类属性和实例属性

我们知道，无论是在类中定义的属性还是方法，在类的外部，都无法直接调用它们，因此，我们完全可以把类看做是一个独立的作用域（称为类命名空间），则类属性其实就是定义在类命名空间内的变量（类方法其实就是定义的类命名空间中的函数）。

根据定义属性的位置不同，类属性又可细分为类属性（后续用类变量表示）和实例属性（后续用实例变量表示）。

类变量（类属性）

类变量指的是定义在类中，但在各个类方法外的变量。类变量的特点是：所有类的实例化对象都可以共享类变量的值，即类变量可以在所有实例化对象中作为公用资源。

注意，类变量推荐直接用类名访问，但也可以使用对象名访问。

例如，下面代码定义了一个 Address 类，并为该类定义了多个类属性：

```
1.          class Address :
2.              detail = '广州'
3.              post_code = '510660'
4.              def info (self):
5.                  # 尝试直接访问类变量
6.                  #print(detail) # 报错
7.                  # 通过类来访问类变量
8.                  print(Address.detail) # 输出 广州
9.                  print(Address.post_code) # 输出 510660
10.         #创建 2 个类对象
11.         addr1 = Address()
12.         addr1.info()
13.         addr2 = Address()
14.         addr2.info()
15.         # 修改 Address 类的类变量
16.         Address.detail = '佛山'
17.         Address.post_code = '460110'
18.         addr1.info()
19.         addr2.info()
```

该程序中，第二、三行代码为 Address 定义了两个类变量。当程序中第一次调用 Address 对象的 info() 方法输出两个类变量时，将会输出这两个类变量的初始值。接下来程序通过

Address 类修改了两个类变量的值，因此当程序第二次通过 info() 方法输出两个类变量时，将会输出这两个类变量修改之后的值。

运行上面代码，将会看到如下输出结果：

```
广州
510660
广州
510660
佛山
460110
佛山
460110
```

通过输出结果可以看到，addr1 和 addr2 共享类变量，换句话说，改变类变量的值会作用于该类所有的实例化对象。

当然，Python 也支持使用对象来访问该对象所属类的类属性（此方式不推荐使用）。例如如下程序：

```
1.          class Record:
2.              # 定义两个类变量
3.              item = '鼠标'
4.              date = '2016-06-16'
5.              def info (self):
6.                  print('info 方法中: ', self.item)
```

```
                print('info 方法中: ', self.date)
        rc = Record()
        print(rc.item) # '鼠标'
        print(rc.date) # '2016-06-16'
        rc.info()
```

上面程序的 Record 中定义了两个类变量，接下来程序完全可以使用 Record 对象来访问这两个类变量。

因此可以看到，在 Record 类的 info() 方法中，程序使用 self 访问 Record 类的类变量，此时 self 代表 info() 方法的调用者，也就是 Record 对象，因此这是合法的；在主程序代码区，程序创建了 Record 对象，并通过对象调用 Record 对象的 item、date 类变量，这也是合法的。

在 Python 中，除了可以通过类名访问类属性之外，还可以动态地为类和对象添加类变量。例如，在上面代码的基础，添加以下代码：

```
        Address.depict ="佛山很美"
        print(addr1.depict)
        print(addr2.depict)
```

运行结果为：

```
佛山很美
佛山很美
```

实例变量（实例属性）

实例变量指的是定义在类的方法中的属性，它的特点是：只作用于调用方法的对象。

注意，实例变量只能通过对象名访问，无法通过类名直接访问。

Python 允许通过对象访问类变量，但无法通过对象修改类变量的值。因为，通过对象修改类变量的值，不是在给“类变量赋值”，而是定义新的实例变量。

例如如下程序：

```
1.          class Inventory:
2.              # 定义两个类变量
3.              item = '鼠标'
4.              quantity = 2000
5.              # 定义实例方法
6.              def change(self, item, quantity):
7.                  # 下面赋值语句不是对类变量赋值，而是定义新的实例变量
8.                  self.item = item
9.                  self.quantity = quantity
10.         # 创建 Inventory 对象
11.         iv = Inventory()
12.         iv.change('显示器', 500)
13.         # 访问 iv 的 item 和 quantity 实例变量
14.         print(iv.item) # 显示器
15.         print(iv.quantity) # 500
16.         # 访问 Inventory 的 item 和 quantity 类变量
17.         print(Inventory.item) # 鼠标
18.         print(Inventory.quantity) # 2000
```

上面程序中，第 8、9 行代码通过实例对 item、quantity 变量赋值，看上去很像是对类

变量赋值，但并不是，它们的作用是：重新定义了两个实例变量。

> 类中，实例变量和类变量可以同名，但是在这种情况下，使用类对象将无法调用类变量，因为它会首选实例变量，因此这也是不推荐“类变量使用对象名调用”的原因。

上面程序在调用 Inventory 对象的 change() 方法之后，访问 Inventory 对象的 item、quantity 变量，由于该对象本身已有这两个实例变量，因此程序将会输出该对象的实例变量的值；接下来程序通过 Inventory 访问它的 item、quantity 两个类变量，此时才是真的访问类变量。

运行上面程序，将看到如下输出结果：

```
显示器
500
鼠标
2000
```

即便程序通过类修改了两个类变量的值，程序中 Inventory 的实例变量的值也不会受到任何影响。例如如下代码：

```
1.          Inventory.item = '类变量 item'
2.          Inventory.quantity = '类变量 quantity'
3.          # 访问 iv 的 item 和 quantity 实例变量
4.          print(iv.item)
5.          print(iv.quantity)
```

运行上面代码，可以看到如下输出结果：

```
显示器
500
```

上面程序开始就修改了 Inventory 类中两个类变量的值，但这种修改对 Inventory 对象的实例变量没有任何影响。

同样，如果程序对一个对象的实例变量进行了修改，这种修改也不会影响类变量的值，更不会影响其他对象中实例变量的值。例如如下代码：

```
        iv2 = Inventory()
        iv2.change('键盘',300)
        iv.item = 'iv 实例变量 item'
        iv.quantity = 'iv 实例变量 quantity'
        print(Inventory.item)
        print(Inventory.quantity)

        print(iv.item)
        print(iv.quantity)
```

运行上面代码，将会看到如下输出结果：

```
类变量 item
类变量 quantity
iv 实例变量 item
iv 实例变量 quantity
```

```
键盘

300
```

从输出结果很容易看出，修改一个对象的实例变量，既不会影响类变量的值，也不会影响其它对象的实例变量。

和动态为类添加类变量不同，Python 只支持为特定的对象添加实例变量。例如，在之前代码的基础上，为 iv 对象添加 color 实例变量，实现代码为：

```
1.          iv.color="red"
2.          print(iv.color)
3.          #因为 color 实例变量仅 iv 对象有，iv2 对象并没有，因此下面这行代码会报错
4.          #print(iv2.color)
```

7.Python 实例方法、静态方法和类方法

和类属性可细分为类属性和实例属性一样，类中的方法也可以有更细致的划分，具体可分为类方法、实例方法和静态方法，本节将详细介绍这 3 种类方法的特点和用法。

Python 类实例方法

通常情况下，在类中定义的方法默认都是实例方法。前面章节中，我们已经定义了不只一个实例方法。不仅如此，类的构造方法理论上也属于实例方法，只不过它比较特殊。

例如：

```
class Person :
    #类构造方法，也属于实例方法
    def __init__(self, name = 'Charlie', age=8):
        self.name = name
        self.age = age
    # 下面定义了一个 say 实例方法
    def say(self, content):
        print(content)
```

实例方法最大的特点就是，它最少也要包含一个 self 参数，用于绑定调用此方法的实例对象。实例方法通常会用类对象直接调用，当然也可以用类名调用，例如：

```
#创建一个类对象
person = Person()
#类对象调用实例方法
person.say("类对象调用实例方法")
#类名调用实例方法，需手动给 self 参数传值
Person.say(person,"类名调用实例方法")
```

运行结果：

```
类对象调用实例方法
类名调用实例方法
```

有关使用类名直接调用实例方法的更多介绍，可阅读《Python 类调用实例方法》一节。

Python 类方法

Python 类方法和实例方法相似，它最少也要包含一个参数，只不过，类方法中通常将其命名为 cls，且 Python 会自动将类本身绑定给 cls 参数（而不是类对象）。因此，在调用类

方法时，无需显式为 cls 参数传参。

> 和 self 一样，cls 参数的命名也不是规定的（可以随意命名），只是 Python 程序员约定俗称的习惯而已。

除此之外，和实例方法最大的不同在于，类方法需要使用@classmethod进行修饰，例如：

```
class Bird:
    # classmethod 修饰的方法是类方法
    @classmethod
    def fly (cls):
        print('类方法 fly: ', cls)
```

注意，如果没有 @classmethod，则 Python 解释器会将 fly() 方法认定为实例方法，而不是类方法。

类方法推荐使用类名直接调用，当然也可以使用实例对象来调用（不推荐），例如：

```
# 调用类方法，Bird 类会自动绑定到第一个参数
Bird.fly()  #①
b = Bird()
# 使用对象调用 fly()类方法，其实依然还是使用类调用，
# 因此第一个参数依然被自动绑定到 Bird 类
b.fly()  #②
```

运行结果为：

```
类方法 fly:  <class '__main__.Bird'>

类方法 fly:  <class '__main__.Bird'>
```

可以看到，不管程序是使用类还是对象调用类方法，Python 都会将类方法的第一个参数绑定到类本身。

Python 类静态方法

静态方法，其实就是我们学过的函数，和函数唯一的区别是，静态方法定义在类这个空间（类命名空间）中，而函数则定义在程序所在的空间（全局命名空间）中。

静态方法没有类似 self、cls 这样的特殊参数，因此 Python 解释器不会对它包含的参数做任何类或对象的绑定，也正是因为如此，此方法中无法调用任何类和对象的属性和方法，静态方法其实和类的关系不大。

静态方法需要使用`@staticmethod`修饰，例如：

```
1.          class Bird:
2.              # staticmethod 修饰的方法是静态方法
3.              @staticmethod
4.              def info (p):
5.                  print('静态方法 info: ', p)
```

静态方法的调用，既可以使用类名，也可以使用类对象，例如：

```
1.          #类名直接调用静态方法
2.          Bird.info("类名")
3.          #类对象调用静态方法
4.          b = Bird()
5.          b.info("类对象")
```

运行结果为：

```
静态方法 info:  类名

静态方法 info:  类对象
```

在使用 Python 编程时，一般不需要使用类方法或静态方法，程序完全可以使用函数来代替类方法或静态方法。但是在特殊的场景（比如使用工厂模式）下，类方法或静态方法也是不错的选择。至于静态方法和类方法的具体应用场景，后续章节用到时再做详细介绍。

8.Python 类调用实例方法

前面讲过，在 Python 的类体中定义的方法默认都是实例方法，也示范了通过对象来调用实例方法。

但要提醒大家的是，Python 的类在很大程度上可看做是一个独立的空间（称为类命名空间），当程序在类体中定义变量、方法时，与前面介绍的定义变量、定义函数其实并没有太大的不同。对比如下代码：

```
1.          # 定义全局空间的 foo 函数
2.          def foo ():
3.              print("全局空间的 foo 方法")
4.          # 全局空间的 bar 变量
5.          bar = 20
6.          class Bird:
7.              # 定义 Bird 空间的 foo 函数
8.              def foo():
9.                  print("Bird 空间的 foo 方法")
10.             # 定义 Bird 空间的 bar 变量
11.             bar = 200
12.         # 调用全局空间的函数和变量
13.         foo()
14.         print(bar)
15.         # 调用 Bird 空间的函数和变量
16.         Bird.foo()
17.         print(Bird.bar)
```

上面代码在全局空间和 Bird 类（Bird 空间）中分别定义了 foo() 函数和 bar 变量，从定义它们的代码来看，几乎没有任何区别，只是在 Bird 类中定义它们时需要缩进。

接下来程序在调用 Bird 空间内的 bar 变量和 foo() 函数（方法）时，只要添加 Bird. 前缀即可，这说明完全可以通过 Bird 类来调用 foo() 函数（方法）。这就是类调用实例方法的证明。

现在问题来了，如果使用类调用实例方法，那么该方法的第一个参数（self）怎么自动绑定呢？例如如下程序：

```
class User:
    def walk (self):
        print(self, '正在慢慢地走')
# 通过类调用实例方法
User.walk()
```

运行上面代码，程序会报出如下错误：

```
TypeError: walk() missing 1 required positional argument:'self'
```

请看程序最后一行代码，调用 walk() 方法缺少传入的 self 参数，所以导致程序出错。这说明在使用类调用实例方法时，Python 不会自动为第一个参数绑定调用者。实际上也没法自动绑定，因此实例方法的调用者是类本身，而不是对象。

如果程序依然希望使用类来调用实例方法，则必须手动为方法的第一个参数传入参数值。例如，将上面的最后一行代码改为如下形式：

```
u = User()
# 显式为方法的第一个参数绑定参数值
User.walk(u)
```

此代码显式地为 walk() 方法的第一个参数绑定了参数值，这样的调用效果完全等同于执行 u.walk()。

实际上，当通过 User 类调用 walk() 实例方法时，Python 只要求手动为第一个参数绑定参数值，并不要求必须绑定 User 对象，因此也可使用如下代码进行调用：

```
# 显式为方法的第一个参数绑定 fkit 字符串参数值
User.walk('fkit')
```

如果按上面方式进行绑定，那么 'fkit' 字符串就会被传给 walk() 方法的第一个参数 self。

因此，运行上面代码，将会看到如下输出结果：

```
fkit 正在慢慢地走
```

总结

Python 的类可以调用实例方法，但使用类调用实例方法时，Python 不会自动为方法的第一个参数 self 绑定参数值；程序必须显式地为第一个参数 self 传参，这种方式调用的方法被称为“未绑定方法”。

> 用类的实例对象访问的类成员方法称为绑定方法；用类名调用的类成员方法称为非绑定方法。

9.为什么说 Python 类是独立的命令空间？

前面章节中，已经不只一次提到，所有位于 class 语句中的代码，其实都位于特殊的命名空间中，通常称之为类命名空间。Python 中，编写的整个程序默认处于全局命名空间内，而类体则处于类命名空间内。

Python 允许在全局范围内放置可执行代码，当 Python 执行该程序时，这些代码就会获得执行的机会。类似地，Python 同样允许在类范围内放置可执行代码，当 Python 执行该类定义肘，这些代码同样会获得执行的机会。

例如，如下程序测试了类命名空间：

```
1.          class Item:
2.              # 直接在类空间中放置执行性质代码
3.              print('正在定义 Item 类')
4.              for i in range(10):
5.                  if i % 2 == 0 :
```

```
                print('偶数:', i)
            else:
                print('奇数:', i)
```

运行结果为：

```
正在定义 Item 类
偶数: 0
奇数: 1
偶数: 2
奇数: 3
偶数: 4
奇数: 5
偶数: 6
奇数: 7
偶数: 8
奇数: 9
```

正如从上面代码所看到的，程序直接在 Item 类体中放置普通的输出语句、循环语句、分支语句，这都是合法的。当程序执行 Item 类时，Item 类命名空间中的这些代码都会被执行。

从执行效果来看，这些可执行代码被放在 Python 类命名空间与全局空间并没有太大的区别。确实如此，这是因为程序并没有定义“成员”（变量或函数），这些代码执行之后就完了，不会留下什么。

但下面代码就有区别。下面代码示范了在全局空间和类命名空间内分别定义 lambda 表达式：

```
global_fn = lambda p: print('执行 lambda 表达式，p 参数: ', p)
class Category:
    cate_fn = lambda p: print('执行 lambda 表达式，p 参数: ', p)
```

```
4.          # 调用全局范围内的 global_fn，为参数 p 传入参数值
5.          global_fn('fkit')  # ①
6.          c = Category()
7.          # 调用类命名空间内的 cate_fn，Python 自动绑定第一个参数
8.          c.cate_fn()  # ②
```

上面程序分别在全局空间、类命名空间内定义了两个 lambda 表达式，在全局空间内定义的 lambda 表达式就相当于一个普通函数，因此程序使用调用函数的方式来调用该 lambda 表达式，并显式地为第一个参数绑定参数值，如上面程序中 ① 号代码所示。

对于在类命名空间内定义的 lambda 表达式，则相当于在该类命名空间中定义了一个函数，这个函数就变成了实例方法，因此程序必须使用调用方法的方式来调用该 lambda 表达式，Python 同样会为该方法的第一个参数（相当于 self 参数）绑定参数值，如上面程序中 ② 号代码所示。

10.Python 描述符

Python 中，通过使用描述符，可以让程序员在引用一个对象属性时自定义要完成的工作。

本质上看，描述符就是一个类，只不过它定义了另一个类中属性的访问方式。换句话说，一个类可以将属性管理全权委托给描述符类。

描述符是 Python 中复杂属性访问的基础，它在内部被用于实现 property、方法、类方法、静态方法和 super 类型。

描述符类基于以下 3 个特殊方法，换句话说，这 3 个方法组成了描述符协议：

- __set__(self, obj, type=None)：在设置属性时将调用这一方法（本节后续用 setter 表示）；
- __get__(self, obj, value)：在读取属性时将调用这一方法（本节后续用 getter 表示）；
- __delete__(self, obj)：对属性调用 del 时将调用这一方法。

其中，实现了 setter 和 getter 方法的描述符类被称为数据描述符；反之，如果只实现了 getter 方法，则称为非数据描述符。

实际上，在每次查找属性时，描述符协议中的方法都由类对象的特殊方法 __getattribute__() 调用（注意不要和 __getattr__() 弄混）。也就是说，每次使用类对象.属性（或者 getattr(类对象，属性值)）的调用方式时，都会隐式地调用 __getattribute__()，它会按照下列顺序查找该属性：

1. 验证该属性是否为类实例对象的数据描述符；
2. 如果不是，就查看该属性是否能在类实例对象的 __dict__ 中找到；
3. 最后，查看该属性是否为类实例对象的非数据描述符。

为了表达清楚，这里举个例子：

```
#描述符类
class revealAccess:
    def __init__(self, initval = None, name = 'var'):
        self.val = initval
        self.name = name

    def __get__(self, obj, objtype):
        print("Retrieving",self.name)
        return self.val

    def __set__(self, obj, val):
        print("updating",self.name)
        self.val = val
class myClass:
```

```
15.             x = revealAccess(10,'var "x"')
16.             y = 5
17.         m = myClass()
18.         print(m.x)
19.         m.x = 20
20.         print(m.x)
21.         print(m.y)
```

运行结果为：

```
Retrieving var "x"

10

updating var "x"

Retrieving var "x"

20

5
```

从这个例子可以看到，如果一个类的某个属性有数据描述符，那么每次查找这个属性时，都会调用描述符的 __get__() 方法，并返回它的值；同样，每次在对该属性赋值时，也会调用 __set__() 方法。

注意，虽然上面例子中没有使用 __del__() 方法，但也很容易理解，当每次使用 del 类对象.属性（或者 delattr(类对象，属性)）语句时，都会调用该方法。

除了使用描述符类自定义类属性被调用时做的操作外，还可以使用 property() 函数或者 @property 装饰器，它们会在后续章节做详细介绍。

11.Python property()

前面章节中，我们一直在用“类对象.属性”的方式访问类中定义的属性，其实这种做法是欠妥的，因为它破坏了类的封装原则。换句话说，正常情况下的类，它包含的属性应该是隐藏的，只允许通过类提供的方法来间接实现对类属性的访问和操作。

因此，在不破坏类封装原则的基础上，为了能够有效操作类中的属性，类中应包含读（或写）类属性的多个 getter（或 setter）方法，这样就可以通过“类对象.方法(参数)”的方式操作属性，例如：

```
1.          class Rectangle:
2.              # 定义构造方法
3.              def __init__(self, width, height):
4.                  self.width = width
5.                  self.height = height
6.              # 定义 setsize()函数
7.              def setsize (self , size):
8.                  self.width, self.height = size
9.              # 定义 getsize()函数
10.             def getsize (self):
11.                 return self.width, self.height
12.              # 定义 delsize()函数
13.             def delsize (self):
14.                 self.width, self.height = 0, 0
15.         rect = Rectangle(3 , 4)
16.         rect.setsize((6,8))
17.         print(rect.getsize())
```

运行结果为：

```
(6,8)
```

可能有读者觉得，这种操作类属性的方式比较麻烦，更习惯使用“类对象.属性”这种方式。庆幸的是，Python 中提供了 property() 函数，可以实现在不破坏类封装原则的前提下，让开发者依旧使用“类对象.属性”的方式操作类中的属性。

property() 函数的基本使用格式如下：

属性名=property(fget=None, fset=None, fdel=None, doc=None)
其中，fget 参数用于指定获取该属性值的类方法，fset 参数用于指定设置该属性值的方法，fdel 参数用于指定删除该属性值的方法，最后的 doc 是一个文档字符串，用于提供说明此函数的作用。

开发者调用 property() 函数时，可以传入 0 个（既不能读，也不能写的属性）、1 个（只读属性）、2 个（读写属性）、3 个（读写属性，也可删除）和 4 个（读写属性，也可删除，包含文档说明）参数。

例如，对前面的 Rectangle 类做适当的修改，使用 property() 函数定义一个 size 属性：

```
1.          class Rectangle:
2.              # 定义构造方法
3.              def __init__(self, width, height):
4.                  self.width = width
5.                  self.height = height
6.              # 定义 setsize()函数
7.              def setsize (self , size):
8.                  self.width, self.height = size
9.              # 定义 getsize()函数
10.             def getsize (self):
11.                 return self.width, self.height
12.              # 定义 getsize()函数
13.             def delsize (self):
14.                 self.width, self.height = 0, 0
15.             # 使用 property 定义属性
16.             size = property(getsize, setsize, delsize, '用于描述矩形大小的属性
    ')
17.         # 访问 size 属性的说明文档
18.         print(Rectangle.size.__doc__)
19.         # 通过内置的 help()函数查看 Rectangle.size 的说明文档
20.         help(Rectangle.size)
21.         rect = Rectangle(4, 3)
22.         # 访问 rect 的 size 属性
23.         print(rect.size) # (4, 3)
24.         # 对 rect 的 size 属性赋值
25.         rect.size = 9, 7
26.         # 访问 rect 的 width、height 实例变量
```

```
        print(rect.width) # 9
        print(rect.height) # 7
        # 删除 rect 的 size 属性
        del rect.size
        # 访问 rect 的 width、height 实例变量
        print(rect.width) # 0
        print(rect.height) # 0
```

程序中，使用 property() 函数定义了一个 size 属性，在定义该属性时一共传入了 4 个参数，这意味着该属性可读、可写、可删除，也有说明文档。所以，该程序尝试对 Rectangle 对象的 size 属性进行读、写、删除操作，其实这种读、写、删除操作分别被委托给 getsize()、setsize() 和 delsize() 方法来实现。

运行上面程序，将会看到如下输出结果：

```
用于描述矩形大小的属性
Help on property:


    用于描述矩形大小的属性


(4, 3)
9
7
0
0
```

在使用 property() 函数定义属性时，也可根据需要只传入少量的参数。例如，如下代码使用 property() 函数定义了一个读写属性，该属性不能删除：

纯文本复制

```
        class User :
```

```
        def __init__ (self, first, last):
            self.first = first
            self.last = last
        def getfullname(self):
            return self.first + ',' + self.last
        def setfullname(self, fullname):
            first_last = fullname.rsplit(',');
            self.first = first_last[0]
            self.last = first_last[1]
        # 使用 property()函数定义 fullname 属性，只传入 2 个参数
        # 该属性是一个读写属性，但不能删除
        fullname = property(getfullname, setfullname)
u = User('悟空', '孙')
# 访问 fullname 属性
print(u.fullname)
# 对 fullname 属性赋值
u.fullname = '八戒,朱'
print(u.first)
print(u.last)
```

此程序中使用 property() 定义了 fullname 属性，该程序使用 property() 函数时只传入两个参数，分别作为 getter 和 setter 方法，因此该属性是一个读写属性，不能删除。

运行上面程序，将看到如下输出结果：

```
悟空,孙

八戒

朱
```

在某些编程语言中，类似于这种 property() 函数合成的属性被称为计算属性。这种属性并不真正存储任何状态，它的值其实是通过某种算法计算得到的。当程序对该属性赋值时，被赋的值也会被存储到其他实例变量中。

12.Python @property

既要保护类的封装特性，又要让开发者可以使用“对象.属性”的方式操作操作类属性，除了使用 property() 函数，Python 还提供了 @property 装饰器。通过 @property 装饰器，可

以直接通过方法名来访问方法，不需要在方法名后添加一对“（）”小括号。

@property 的语法格式如下：

```
@property
def 方法名(self)
    代码块
```

例如，定义一个矩形类，并定义用 @property 修饰的方法操作类中的 area 私有属性，

代码如下：

```
class Rect:
    def __init__(self,area):
        self.__area = area
    @property
    def area(self):
        return self.__area
rect = Rect(30)
#直接通过方法名来访问 area 方法
print("矩形的面积是：",rect.area)
```

运行结果为：

```
矩形的面积为： 30
```

上面程序中，使用 @property 修饰了 area() 方法，这样就使得该方法变成了 area 属性的 getter 方法。需要注意的是，如果类中只包含该方法，那么 area 属性将是一个只读属性。

也就是说，在使用 Rect 类时，无法对 area 属性重新赋值，即运行如下代码会报错：

```
rect.area = 90
print("修改后的面积：",rect.area)
```

运行结果为：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 10, in <module>
    rect.area = 90
AttributeError: can't set attribute
```

而要想实现修改 area 属性的值，还需要为 area 属性添加 setter 方法，就需要用到 setter 装饰器，它的语法格式如下：

```
@方法名.setter
def 方法名(self, value):
    代码块
```

例如，为 Rect 类中的 area 方法添加 setter 方法，代码如下：

```
        @area.setter
        def area(self, value):
            self.__area = value
```

再次运行如下代码：

```
        rect.area = 90
        print("修改后的面积：",rect.area)
```

运行结果为：

```
修改后的面积: 90
```

这样，area 属性就有了 getter 和 setter 方法，该属性就变成了具有读写功能的属性。

除此之外，还可以使用 deleter 装饰器来删除指定属性，其语法格式为：

```
@方法名.deleter
def 方法名(self):
    代码块
```

例如，在 Rect 类中，给 area() 方法添加 deleter 方法，实现代码如下：

```
1.          @area.deleter
2.          def area(self):
3.              self.__area = 0
```

然后运行如下代码：

```
1.          del rect.area
2.          print("删除后的 area 值为：",rect.area)
```

运行结果为：

```
删除后的 area 值为： 0
```

13.Python 封装

封装（Encapsulation）是面向对象的三大特征之一（另外两个是继承和多态），它指的是将对象的状态信息隐藏在对象内部，不允许外部程序直接访问对象内部信息，而是通过该类所提供的方法来实现对内部信息的操作和访问。

就好比使用计算机，我们只需要使用计算机提供的键盘，就可以达到操作计算机的目的，至于在敲击键盘时计算机内部是如何工作，我们根本不需要知道。

封装机制保证了类内部数据结构的完整性，因为使用类的用户无法直接看到类中的数据结构，只能使用类允许公开的数据，很好地避免了外部对内部数据的影响，提高了程序的可维护性。总的来说，对一个类或对象实现良好的封装，可以达到以下目的：

- 隐藏类的实现细节。
- 让使用者只能通过事先预定的方法来访问数据，从而可以在该方法里加入控制逻辑，限制对属性的不合理访问。
- 可进行数据检查，从而有利于保证对象信息的完整性。
- 便于修改，提高代码的可维护性。

为了实现良好的封装，需要从以下两个方面来考虑：

1. 将对象的属性和实现细节隐藏起来，不允许外部直接访问。
2. 把方法暴露出来，让方法来控制对这些属性进行安全的访问和操作。

因此，实际上封装有两个方面的含义：把该隐藏的隐藏起来，把该暴露的暴露出来。

Python 并没有提供类似于其他语言的 private 等修饰符，因此 Python 并不能真正支持隐藏。为了隐藏类中的成员，Python 玩了一个小技巧：只要将 Python 类的成员命名为

以双下画线开头的，Python 就会把它们隐藏起来。

例如，如下程序示范了 Python 的封装机制：

```
class User :
    def __hide(self):
        print('示范隐藏的 hide 方法')
    def getname(self):
        return self.__name
    def setname(self, name):
        if len(name) < 3 or len(name) > 8:
            raise ValueError('用户名长度必须在 3～8 之间')
        self.__name = name
    name = property(getname, setname)
    def setage(self, age):
        if age < 18 or age > 70:
            raise ValueError('用户名年龄必须在 18 在 70 之间')
        self.__age = age
    def getage(self):
        return self.__age
    age = property(getage, setage)
# 创建 User 对象
u = User()
# 对 name 属性赋值，实际上调用 setname()方法
u.name = 'fk' # 引发 ValueError: 用户名长度必须在 3～8 之间
```

上面程序将 User 的两个实例变量分别命名为 __name 和 __age ,这两个实例变量就会被隐藏起来，这样程序就无法直接访问 __name、__age 变量，只能通过 setname()、getname()、setage()、getage() 这些访问器方法进行访问，而 setname()、setage() 会对用户设置的 name、age 进行控制，只有符合条件的 name、age 才允许设置。

上面程序用到了 raise 关键字来抛出异常，关于 raise 关键字和异常的相关信息，后续章节会详细介绍。

上面程序尝试将 User 对象的 name 设为 fk，这个字符串的长度为“2”不符合实际要求，因此运行程序最后一行包含如下错误：

```
ValueError：用户名长度必须在 3-8 之间
```

将最后一行代码注释掉，并在程序尾部添加如下代码：

```
        u.name = 'fkit'
        u.age = 25
        print(u.name) # fkit
        print(u.age) # 25
```

此时程序对 name、age 所赋的值都符合要求，因此上面两行赋值语句完全可以正常运行。

运行上面代码，可以看到如下输出结果：

```
fkit
25
```

从该程序可以看出封装的好处，程序可以将 User 对象的实现细节隐藏起来，程序只能通过暴露出来的 setname()、setage() 方法来改变 User 对象的状态，而这两个方法可以添加自己的逻辑控制，这种控制对 User 的修改始终是安全的。

上面程序还定义了一个 __hide() 方法，这个方法默认是隐藏的。如果程序尝试执行如下代码：

```
        # 尝试调用隐藏的__hide()方法
        u.__hide()
```

将会提示如下错误：

```
AttributeError:'User' object has no attribute 'hide'
```

最后需要说明的是，Python 其实没有真正的隐藏机制，双下画线只是 Python 的一个小技巧，Python 会“偷偷”地改变以双下画线开头的方法名，会在这些方法名前添加单下画线和类名。因此上面的 __hide() 方法其实可以按如下方式调用（通常并不推荐这么干）：

```
1.          # 调用隐藏的__hide()方法
2.          u._User__hide()
```

运行上面代码，可以看到如下输出结果：

```
示范隐藏的 hide 方法
```

通过上面调用可以看出，Python 并没有实现真正的隐藏。

类似的是，程序也可通过为隐藏的实例变量添加下画线和类名的方式来访问或修改对象的实例变量。例如如下代码：

```
1.          # 对隐藏的__name 属性赋值
2.          u._User__name = 'fk'
3.          # 访问 User 对象的 name 属性（实际上访问__name 实例变量）
4.          print(u.name)
```

上面粗体字代码实际上就是对 User 对象的 name 实例变量进行赋值，通过这种方式可“绕开”setname() 方法的检查逻辑，直接对 User 对象的 name 属性赋值。运行这两行代码，可以看到如下输出结果：

```
fk
```

总结

Python 并没有提供真正的隐藏机制，所以 Python 类定义的所有成员默认都是公开的；如果程序希望将 Python 类中的某些成员隐藏起来，那么只要让该成员的名字以双下画线开头即可。

即使通过这种机制实现了隐藏，其实也依然可以绕过去。

14.Python 继承机制

继承是面向对象的三大特征之一，也是实现代码复用的重要手段。继承经常用于创建和现有类功能类似的新类，又或是新类只需要在现有类基础上添加一些成员（属性和方法），但又不想直接将现有类代码复制给新类。

例如，有一个 Shape 类，该类的 draw() 方法可以在屏幕上画出指定的形状，现在需要创建一个 Rectangle 类，要求此类不但可以在屏幕上画出指定的形状，还可以计算出所画形状的面积。要创建这样的 Rectangle 类，除了将 draw() 方法直接复制到新类中，并添加计算面积的方法，其实还有更简单的方法，即让 Rectangle 类继承 Shape 类，这样当 Rectangle 类对象调用 draw() 方法时，Python 解释器会自动去 Shape 类中调用该方法，如此，我们只需在 Rectangle 类中添加计算面积的方法即可。

Python 中，实现继承的类称为子类，被继承的类称为父类（也可称为基类、超类）。子类继承父类的语法是：在定义子类时，将多个父类放在子类之后的圆括号里。语法格式如下：

```
class 类名(父类 1, 父类 2, ...)：
    #类定义部分
```

注意，Python 的继承是多继承机制，即一个子类可以同时拥有多个直接父类。

从上面的语法格式来看，定义子类的语法非常简单，只需在原来的类定义后增加圆括号，并在圆括号中添加多个父类，即可表明该子类继承了这些父类。如果在定义一个 Python 类

时，并未显式指定这个类的直接父类，则这个类默认继承 object 类。

> object 类是所有类的父类，要么是直接父类，要么是间接父类。

父类和子类的关系，就好像水果和苹果的关系，苹果是一种特殊的水果，苹果是水果的子类，水果是苹果的父类，因此可以说，苹果继承了水果。不仅如此，由于子类是一种特殊的父类，因此父类包含的范围总比子类包含的范围要大，所以可以认为父类是大类，而子类是小类。

从实际意义上看，子类是对父类的扩展，子类是一种特殊的父类。从这个意义上看，使用继承来描述子类和父类的关系是错误的，用扩展更恰当。因此，这样的说法更加准确：苹果扩展了水果这个类。

> 从子类的角度来看，子类扩展（extend）了父类；但从父类的角度来看，父类派生（derive）出子类。也就是说，扩展和派生所描述的是同一个动作，只是观察角度不同而已。

下面程序示范了子类继承父类的特点。下面是 Fruit 类的代码：

```
class Fruit:
    def info(self):
        print("我是一个水果！重%g 克" % self.weight)

class Food:
    def taste(self):
        print("不同食物的口感不同")

# 定义 Apple 类，继承了 Fruit 和 Food 类
class Apple(Fruit, Food):
```

```
11.             pass
12.
13.         # 创建 Apple 对象
14.         a = Apple()
15.         a.weight = 5.6
16.         # 调用 Apple 对象的 info()方法
17.         a.info()
18.         # 调用 Apple 对象的 taste()方法
19.         a.taste()
```

运行结果为：

```
我是一个水果！重 5.6 克
不同食物的口感不同
```

上面程序中，定义了两个父类：Fruit 类和 Food 类，接下来定义了一个 Apple 类，该类本身是空类，但其继承自 Food 类和 Fruit 类。因此，在主程序部分创建了 Apple 对象之后，可以访问 Apple 对象的 info() 和 taste() 方法，这表明 Apple 对象也具有了 info() 和 taste() 方法，这就是继承的作用，即子类扩展（继承）了父类，将可以继承得到父类定义的方法，这样子类就可复用父类的方法了。

关于 Python 的多继承

大部分面向对象的编程语言（除了 C++）都只支持单继承，而不支持多继承，这是由于多继承不仅增加了编程的复杂度，而且很容易导致一些莫名的错误。

> Python 虽然在语法上明确支持多继承，但通常推荐如果不是很有必要，则尽量不要使用多继承，而是使用单继承，这样可以保证编程思路更清晰，而且可以避免很多麻烦。

当一个子类有多个直接父类时，该子类会继承得到所有父类的方法，这一点在前面示例中已经做了示范。现在的问题是，如果多个父类中包含了同名的方法，此时会发生什么呢？此时排在前面的父类中的方法会“遮蔽”排在后面的父类中的同名方法。例如如下代码：

```
1.          class Item:
2.              def info (self):
3.                  print("Item 中方法:", '这是一个商品')
4.          class Product:
5.              def info (self):
6.                  print("Product 中方法:", '这是一个工业产品')
7.          class Mouse(Item, Product): # ①
8.              pass
9.          m = Mouse()
10.         m.info()
```

上面 ① 号代码让 Mouse 继承了 Item 类和 Product 类，由于 Item 排在前面，因此 Item 中定义的方法优先级更好，Python 会优先到 Item 父类中搜寻方法，一旦在 Item 父类中搜寻到目标方法，Python 就不会继续向下搜寻了。

运行上面程序，将看到如下输出结果：

```
Item 中方法：这是一个商品
```

如果将程序中第 7 行代码改为如下形式：

```
class Mouse(Product,Itern): #①
```

此时 Product 父类的优先级高于 Item 父类，因此 Product 中的 info() 方法将会起作用。运行上面程序，将会看到如下输出结果：

```
Product 中方法：这是一个工业产品
```

15.在子类中，Python 到底是如何找到父类的属性和方法的？（深度揭秘）

我们知道，Python 类是支持（多）继承的，一个类的方法和属性可能定义在当前类，也可能定义在基类。针对这种情况，当调用类方法或类属性时，就需要对当前类以及它的基类进行搜索，以确定方法或属性的位置，而搜索的顺序就称为方法解析顺序。

方法解析顺序（Method Resolution Order），简称 MRO。对于只支持单继承的编程语言来说，MRO 很简单，就是从当前类开始，逐个搜索它的父类；而对于 Python，它支持多继承，MRO 相对会复杂一些。

实际上，Python 发展至今，经历了以下 3 种 MRO 算法，分别是：

1. 从左往右，采用深度优先搜索（DFS）的算法，称为旧式类的 MRO；
2. 自 Python 2.2 版本开始，新式类在采用深度优先搜索算法的基础上，对其做了优化；
3. 自 Python 2.3 版本，对新式类采用了 C3 算法。由于 Python 3.x 仅支持新式类，所以该版本只使用 C3 算法。

有关旧式类和新式类的讲解，可阅读《Python super()使用注意事项》一文。

有读者可能会好奇，为什么 MRO 弃用了前两种算法，而选择最终的 C3 算法呢？原因很

简单，前 2 种算法都存在一定的问题。

旧式类 MRO 算法

在使用旧式类的 MRO 算法时，以下面代码为例（程序一）：

```
class A:
    def method(self):
      print("CommonA")
class B(A):
    pass
class C(A):
    def method(self):
      print("CommonC")
class D(B, C):
    pass
print(D().method())
```

通过分析可以想到，此程序中的 4 个类是一个“菱形”继承的关系，当使用 D 类对象访问 method() 方法时，根据深度优先算法，搜索顺序为 D->B->A->C->A。

旧式类的 MRO 可通过使用 inspect 模块中的 getmro(类名) 函数直接获取。例如 inspect.getmro(D) 表示获取 D 类的 MRO。

因此，使用旧式类的 MRO 算法最先搜索得到的是基类 A 中的 method() 方法，即在 Python 2.x 版本中，此程序的运行结果为：

```
CommonA
```

但是，这个结果显然不是想要的，我们希望搜索到的是 C 类中的 method() 方法。

新式类 MRO 算法

为解决旧式类 MRO 算法存在的问题，Python 2.2 版本推出了新的计算新式类 MRO 的方法，它仍然采用从左至右的深度优先遍历，但是如果遍历中出现重复的类，只保留最后一个。

仍以上面程序为例，通过深度优先遍历，其搜索顺序为 `D->B->A->C->A`，由于此顺序中有 2 个 A，因此仅保留后一个，简化后得到最终的搜索顺序为 `D->B->C->A`。

新式类可以直接通过 类名.__mro__ 的方式获取类的 MRO，也可以通过 类名.mro() 的形式，旧式类是没有 __mro__ 属性和 mro() 方法的。

可以看到，这种 MRO 方式已经能够解决“菱形”继承的问题，但是可能会违反单调性原则。所谓单调性原则，是指在类存在多继承时，子类不能改变基类的 MRO 搜索顺序，否则会导致程序发生异常。

例如，分析如下程序（程序二）：

```
1.          class X(object):
2.            pass
3.          class Y(object):
4.            pass
5.          class A(X,Y):
6.            pass
7.          class B(Y,X):
8.            pass
9.          class C(A, B):
10.           pass
```

通过进行深度遍历，得到搜索顺序为 `C->A->X->object->Y->object->B->Y->object->X->object`，再进行简化（相同取后者），得到 `C->A->B->Y->X->object`。

下面来分析这样的搜索顺序是否合理，我们来看下各个类中的 MRO：

- 对于 A，其搜索顺序为 A->X->Y->object；
- 对于 B，其搜索顺序为 B->Y->X->object；
- 对于 C，其搜索顺序为 C->A->B->X->Y->object。

可以看到，B 和 C 中，X、Y 的搜索顺序是相反的，也就是说，当 B 被继承时，它本身的搜索顺序发生了改变，这违反了单调性原则。

MRO C3

为解决 Python 2.2 中 MRO 所存在的问题，Python 2.3 采用了 C3 方法来确定方法解析顺序。多数情况下，如果某人提到 Python 中的 MRO，指的都是 C3 算法。

在 Python 2.3 及后续版本中，运行程序一，得到如下结果：

```
CommonC
```

运行程序二，会产生如下异常：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 9, in <module>
    class C(A, B):
TypeError: Cannot create a consistent method resolution
order (MRO) for bases X, Y
```

由此可见，C3 可以有效解决前面 2 种算法的问题。那么，C3 算法是怎样实现的呢？

以程序一为主，C3 把各个类的 MRO 记为如下等式：

- 类 A：L[A] = merge(A , object)
- 类 B：L[B] = [B] + merge(L[A] , [A])
- 类 C：L[C] = [C] + merge(L[A] , [A])
- 类 D：L[D] = [D] + merge(L[A] , L[B] , [A] , [B])

注意，以类 A 等式为例，其中 merge 包含的 A 称为 L[A] 的头，剩余元素（这里仅有一个 object）称为尾。

这里的关键在于 merge，它的运算方式如下：

1. 检查第一个列表的头元素（如 L[A] 的头），记作 H。
2. 若 H 未出现在 merge 中其它列表的尾部，则将其输出，并将其从所有列表中删除，然后回到步骤 1；否则，取出下一个列表的头部记作 H，继续该步骤。

重复上述步骤，直至列表为空或者不能再找出可以输出的元素。如果是前一种情况，则算法

结束；如果是后一种情况，Python 会抛出异常。

由此，可以计算出类 B 的 MRO，其计算过程为：

```
L[B] = [B] + merge(L[A],[A])
     = [B] + merge([A,object],[A])
     = [B,A] + merge([object])
     = [B,A,object]
```

同理，其他类的 MRO 也可以轻松计算得出。这里不再赘述，有兴趣的读者可自行推算

16.Python 父类方法重写

子类扩展了父类，子类是一种特殊的父类。大部分时候，子类总是以父类为基础，额外增加新的方法。但在一些场景中，子类需要重写父类的方法。

例如，鸟类都包含了飞翔方法，其中驼鸟是一种特殊的鸟类，因此驼鸟应该是鸟的子类，它也将从鸟类获得飞翔方法，但这个飞翔方法明显不适合驼鸟，为此，驼鸟需要重写鸟类的方法。

如下程序示范了子类重写父类的方法：

```
1.          class Bird:
2.              # Bird 类的 fly()方法
3.              def fly(self):
4.                  print("我在天空里自由自在地飞翔...")
5.          class Ostrich(Bird):
6.              # 重写 Bird 类的 fly()方法
7.              def fly(self):
```

```
8.                  print("我只能在地上奔跑...")
9.
10.         # 创建 Ostrich 对象
11.         os = Ostrich()
12.         # 执行 Ostrich 对象的 fly()方法，将输出"我只能在地上奔跑..."
13.         os.fly()
```

运行上面程序，将看到运行 os.fly() 时执行的不再是 Bird 类的 fly() 方法，而是 Ostrich 类的 fly() 方法。

这种子类包含与父类同名的方法的现象被称为方法重写（Override），也被称为方法覆盖。可以说子类重写了父类的方法，也可以说子类覆盖了父类的方法。

使用未绑定方法调用被重写的方法

如果在子类中调用重写之后的方法，Python 总是会执行子类重写的方法，不会执行父类中被重写的方法。如果需要在子类中调用父类中被重写的实例方法，那该怎么办呢？

别忘了，Python 类相当于类空间，因此 Python 类中的方法本质上相当于类空间内的函数。所以，即使是实例方法，Python 也允许通过类名调用。区别在于：在通过类名调用实例方法时，Python 不会为实例方法的第一个参数 self 自动绑定参数值，而是需要程序显式绑定第一个参数 self。这种机制被称为未绑定方法。

通过使用未绑定方法即可在子类中再次调用父类中被重写的方法。例如如下程序：

```
1.          class BaseClass:
2.              def foo (self):
```

```
3.                 print('父类中定义的 foo 方法')
4.         class SubClass(BaseClass):
5.             # 重写父类的 foo 方法
6.             def foo (self):
7.                 print('子类重写父类中的 foo 方法')
8.             def bar (self):
9.                 print('执行 bar 方法')
10.                # 直接执行 foo 方法，将会调用子类重写之后的 foo()方法
11.                self.foo()
12.                # 使用类名调用实例方法（未绑定方法）调用父类被重写的方法
13.                BaseClass.foo(self)
14.        sc = SubClass()
15.        sc.bar()
```

上面程序中 SubClass 继承了 BaseClass 类，并重写了父类的 foo() 方法。接下来程序在 SubClass 类中定义了 bar() 方法，该方法的第 11 行代码直接通过 self 调用 foo() 方法，Python 将会执行子类重写之后的 foo() 方法；第 13 行代码通过未绑定方法显式调用 BaseClass 中的 foo 实例方法，并显式为第一个参数 self 绑定参数值，这就实现了调用父类中被重写的方法。

17.如何使用 Python 继承机制提高开发效率？

我们知道，Python 中内置有一个 object 类，它是所有内置类型的共同祖先，也是所有没有显式指定父类的类（包括用户自定义的）的共同祖先。因此在实际编程过程中，如果想实现与某个内置类型具有类似行为的类时，最好的方法就是将这个内置类型子类化。

内置类型子类化，其实就是自定义一个新类，使其继承有类似行为的内置类，通过重定义这个新类实现指定的功能。

举个例子，如下所示创建了一个名为 newDict 的类，其中 newDictError 是自定义的异常类：

```
1.         class newDictError(ValueError):
2.           """如果向 newDict 添加重复值，则引发此异常"""
3.
4.         class newDict(dict):
5.           """不接受重复值的字典"""
6.           def __setitem__(self,key,value):
7.             if value in self.values():
8.               if ((key in self and self[key]!=value) or (key not in self)):
```

```
                raise newDictError("这个值已经存在，并对应不同的键")
            super().__setitem__(key,value)
demoDict = newDict()
demoDict['key']='value'
demoDict['other_key']='value2'
print(demoDict)
demoDict['other_key']='value'
print(demoDict)
```

运行结果为：

```
{'key': 'value', 'other_key': 'value2'}
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 15, in <module>
    demoDict['other_key']='value'
  File "C:\Users\mengma\Desktop\demo.py", line 9, in __setitem__
    raise newDictError("这个值已经存在，并对应不同的键")
newDictError: 这个值已经存在，并对应不同的键
```

可以看到，newDict 是 Python 中 dict 类型的子类，所以其大部分行为都和 dict 内置类相同，唯一不同之处在于，newDict 不允许字典中多个键对应相同的值。如果用户试图添加具有相同值的新元素，则会引发 newDictError 异常，并给出提示信息。

> 由于目前尚未学习如何处理异常，因此这里没有 newDictError 做任何处理，异常处理会在后续章节做详细讲解。

另外，如果查看现有代码你会发现，其实很多类都是对 Python 内置类的部分实现，它们作为子类的速度更快，代码更整洁。

比如，list 类型用来管理序列，如果一个类需要在内部处理序列，那么就可以对 list 进行子类化，示例代码如下：

```
class myList(list):
```

```
        def __init__(self,name):
            self.name = name

        def dir(self,nesting = 0):
            offset = " " * nesting
            print("%s%s/" % (offset,self.name))

            for element in self:
                if hasattr(element , 'dir'):
                    element.dir(nesting + 1)
                else:
                    print("%s %s" % (offset,element))

demoList = myList('C 语言中文网')
demoList.append('http://c.biancheng.net')
print(demoList.dir())
```

运行结果如下：

```
C 语言中文网/
http://c.biancheng.net
None
```

其实，除了 Python 中常用的基本内置类型，collections 模块中还额外提供了很多有用的容器，这些容器可以满足大部分情况。

18.Python super()

Python 的子类也会继承得到父类的构造方法，但如果子类有多个直接父类，那么会优先选择排在最前面的父类的构造方法。例如如下代码：

```
class Employee :
    def __init__ (self, salary):
        self.salary = salary
```

```
4.              def work (self):
5.                  print('普通员工正在写代码，工资是:', self.salary)
6.          class Customer:
7.              def __init__ (self, favorite, address):
8.                  self.favorite = favorite
9.                  self.address = address
10.             def info (self):
11.                 print('我是一个顾客，我的爱好是: %s,地址是%s' % (self.favorite,
    self.address))
12.         # Manager 继承了 Employee、Customer
13.         class Manager (Employee, Customer):
14.             pass
15.         m = Manager(25000)
16.         m.work()  #①
17.         #m.info()  #②
```

上面程序中第 13 行代码定义了 Manager 类，该类继承了 Employee 和 Customer 两个父类。接下来程序中的 Manager 类将会优先使用 Employee 类的构造方法（因为它排在前面），所以程序使用 Manager(25000）来创建 Manager 对象。该构造方法只会初始化 salary 实例变量，因此执行上面程序中 ① 号代码是没有任何问题的。

但是当执行到 ② 号代码时就会引发错误，这是由于程序在使用 Employee 类的构造方法创建 Manager 对象时，程序并未初始化 Customer 对象所需的两个实例变量：favorite 和 address，因此程序引发错误。

如果将程序中第 13 行代码改为如下形式：

```
class Manager (Customer, Employee)
```

上面 Manager 类将优先使用 Customer 类的构造方法，因此程序必须使用如下代码来创建 Manager 对象：

```
m = Manager('IT 产品','广州')
```

上面代码为 Manager 的构造方法传入两个参数，这明显是调用从 Customer 类继承得到的两个构造方法，此时程序将可以初始化 Customer 类中的 favorite 和 address 实例变量，但它又不能初始化 Employee 类中的 salary 实例变量。因此，此时程序中的 ② 号代码可以正常运行，但 ① 号代码会报错。

为了让 Manager 能同时初始化两个父类中的实例变量，Manager 应该定义自己的构造方法，即重写父类的构造方法。Python 要求，如果子类重写了父类的构造方法，那么子类的构造方法必须调用父类的构造方法。

子类的构造方法调用父类的构造方法有两种方式：

1. 使用未绑定方法，这种方式很容易理解。因为构造方法也是实例方法，当然可以通过这种方式来调用。
2. 使用 super() 函数调用父类的构造方法。

注意，当子类继承多个父类是，super() 函数只能用来调用第一个父类的构造方法，而其它父类的构造方法只能使用未绑定的方式调用。

在交互式解释器中输入 help(super) 查看 super() 函数的帮助信息，可以看到如下输出信息：

```
class super(object)
|  super() -> same as super(__class__, <first argument>)
|  super(type) -> unbound super object
|  super(type, obj) -> bound super object; requires isinstance(obj, type)
|  super(type, type2) -> bound super object; requires issubclass(type2, type)
|  Typical use to call a cooperative superclass method:
|  class C(B):
|      def meth(self, arg):
|          super().meth(arg) #①
|  This works for class methods too:
|  class C(B):
|      @classmethod
|      def cmeth(cls, arg):
|          super().cmeth(arg) #②
...
```

从上面介绍可以看出，super 其实是一个类，因此调用 super() 的本质就是调用 super 类的构造方法来创建 super 对象。

从上面的帮助信息可以看到，使用 super() 构造方法最常用的做法就是不传入任何参数（这种做法与 super(type, obj) 的效果相同），然后通过 super 对象的方法既可调用父类的实例方法，也可调用父类的类方法。在调用父类的实例方法时，程序会完成第一个参数 self 的自动绑定，如上帮助信息中 ① 号信息所示。在调用类方法时，程序会完成第一个参数 cls 的自动绑定，如上面帮助信息中 ② 号信息所示。

掌握了 super() 函数的用法之后，接下来可以将上面程序改为如下形式：

纯文本复制

```
1.          # Manager 继承了 Employee、Customer
2.          class Manager(Employee, Customer):
3.              # 重写父类的构造方法
4.              def __init__(self, salary, favorite, address):
5.                  print('--Manager 的构造方法--')
6.                  # 通过 super()函数调用父类的构造方法
7.                  super().__init__(salary)
8.                  # 与上一行代码的效果相同
9.                  #super(Manager, self).__init__(salary)
10.                 # 使用未绑定方法调用父类的构造方法
11.                 Customer.__init__(self, favorite, address)
12.         # 创建 Manager 对象
13.         m = Manager(25000, 'IT 产品', '广州')
14.         m.work()  #①
15.         m.info()  #②
```

上面程序中，第 7、11 行代码分别示范了两种方式调用父类的构造方法。通过这种方式，Manager 类重写了父类的构造方法，并在构造方法中显式调用了父类的两个构造方法执行初始化，这样两个父类中的实例变量都能被初始化。

运行上面程序，可以看到如下运行结果：

```
--Manager 的构造方法--

普通员工正在写代码，工资是：2500。

我是一个顾客，我的爱好是：IT 产品，地址是广州
```

从上面的运行结果可以看到，此时程序中 ①、② 号代码都可正常运行，这正是程序对 Manager 重写的构造方法分别调用了两个父类的构造方法来完成初始化的结果。

19.切记，super()只能在新式类中使用！

前面已经讲解了 super() 函数的用法，值得一提的是，Python 2 中 super() 函数的用法和 Python 3 大致相同，唯一的区别在于，Python 2 中不能使用零参数形式的格式，必须提供至少一个参数。

对于想要编写跨版本兼容代码的程序员来说，还要注意一件事，即 Python 2 中的 super() 函数只适用于新式类，在旧式类中不能使用 super()。

那么，什么是旧式类和新式类呢？在早期版本的 Python 中，所有类并没有一个共同的祖先 object，如果定义一个类，但没有显式指定其祖先，那么就被解释为旧式类，例如：

纯文本复制

```
class oldStyle1:
  pass
class oldStyle2:
  pass
```

其中，oldStyle1 和 oldStyle2 都属于旧式类。

Python 2.x 版本中，为了向后兼容保留了旧式类。该版本中的新式类必须显式继承 object 或者其他新式类：

```
class newStyleClass(object):
  pass
class newStyleClass(newStyleClass):
  pass
```

显然，以上两个类都属于新式类。

而在 Python 3.x 版本中，不再保留旧式类的概念。因此，没有继承任何其他类的类都隐式地继承自 object。

可以说，在 Python 3.x 中，显式声明某个类继承自 object 似乎是冗余的。但如果考虑跨版本兼容，那么就必须将 object 作为所有基类的祖先，因为如果不这么做的话，这些类将被解释为旧式类，最终会导致难以诊断的问题。

20.使用 super()，这些“坑”千万别踩！

Python 中，由于基类不会在 __init__() 中被隐式地调用，需要程序员显式调用它们。这种情况下，当程序中包含多重继承的类层次结构时，使用 super 是非常危险的，往往会在类的初始化过程中出现问题。

混用 super 与显式类调用

分析如下程序，C 类使用了 __init__() 方法调用它的基类，会造成 B 类被调用了 2 次：

```
class A:
    def __init__(self):
        print("A",end=" ")
        super().__init__()
class B:
    def __init__(self):
        print("B",end=" ")
        super().__init__()
class C(A,B):
    def __init__(self):
        print("C",end=" ")
        A.__init__(self)
        B.__init__(self)
print("MRO:",[x.__name__ for x in C.__mro__])
C()
```

运行结果为：

```
MRO: ['C', 'A', 'B', 'object']
C A B B
```

出现以上这种情况的原因在于，C 的实例调用 A.__init__(self)，使得 super(A,self).__init__() 调用了 B.__init__() 方法。换句话说，super 应该被用到整个类的层次结构中。

但是，有时这种层次结构的一部分位于第三方代码中，我们无法确定外部包的这些代码中是否使用 super()，因此，当需要对某个第三方类进行子类化时，最好查看其内部代码以及 MRO 中其他类的内部代码。

不同种类的参数

使用 super 的另一个问题是初始化过程中的参数传递。如果没有相同的签名，一个类怎么能调用其基类的 __init__() 代码呢？这会导致下列问题：

```
1.          class commonBase:
2.              def __init__(self):
3.                  print("commonBase")
4.                  super().__init__()
5.
6.          class base1(commonBase):
7.              def __init__(self):
8.                  print("base1")
9.                  super().__init__()
10.
11.         class base2(commonBase):
12.             def __init__(self):
13.                 print("base2")
14.                 super().__init__()
15.
16.         class myClass(base1,base2):
17.             def __init__(self,arg):
18.                 print("my base")
19.                 super().__init__(arg)
20.         myClass(10)
```

运行结果为：

```
my base
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 20, in <module>
    myClass(10)
```

```
  File "C:\Users\mengma\Desktop\demo.py", line 19, in __init__
    super().__init__(arg)
TypeError: __init__() takes 1 positional argument but 2 were given
```

一种解决方法是使用 *args 和 **kwargs 包装的参数和关键字参数，这样即使不使用它们，所有的构造函数也会传递所有参数，如下所示：

```
1.  class commonBase:
2.      def __init__(self,*args,**kwargs):
3.          print("commonBase")
4.          super().__init__()
5.
6.  class base1(commonBase):
7.      def __init__(self,*args,**kwargs):
8.          print("base1")
9.          super().__init__(*args,**kwargs)
10.
11. class base2(commonBase):
12.     def __init__(self,*args,**kwargs):
13.         print("base2")
14.         super().__init__(*args,**kwargs)
15.
16. class myClass(base1,base2):
17.     def __init__(self,arg):
18.         print("my base")
19.         super().__init__(arg)
20. myClass(10)
```

运行结果为：

```
my base
base1
```

```
base2

commonBase
```

不过，这是一种很糟糕的解决方法，由于任何参数都可以传入，所有构造函数都可以接受任何类型的参数，这会导致代码变得脆弱。另一种解决方法是在 MyClass 中显式地使用特定类的 __init__() 调用，但这无疑会导致第一种错误。

总结

如果想要避免程序中出现以上的这些问题，这里给出几点建议：

- 尽可能避免使用多继承，可以使用一些设计模式来替代它；
- super 的使用必须一致，即在类的层次结构中，要么全部使用 super，要么全不用。混用 super 和传统调用是一种混乱的写法；
- 如果代码需要兼容 Python 2.x，在 Python 3.x 中应该显式地继承自 object。在 Python 2.x 中，没有指定任何祖先地类都被认定为旧式类。
- 调用父类时应提前查看类的层次结构，也就是使用类的 __mro__ 属性或者 mro() 方法查看有关类的 MRO。

21.Python __slots__限制类实例动态添加属性和方法

实例都添加方法，则可通过为类添加方法来实现。例如如下代码：

```
1.          class Cat:
2.              def __init__(self, name):
3.                  self.name = name
4.          def walk_func(self):
5.              print('%s 慢慢地走过一片草地' % self.name)
6.          d1 = Cat('Garfield')
```

```
d2 = Cat('Kitty')
#d1.walk() # AttributeError
# 为 Cat 动态添加 walk()方法，该方法的第一个参数会自动绑定
Cat.walk = walk_func  # ①
# d1、d2 调用 walk()方法
d1.walk()
d2.walk()
```

上面程序定义了一个 Cat 类，该 Cat 类只定义了一个构造方法，并未提供任何方法。因此，程序第 8 行代码调用 d1.walk() 方法时会出现异常，Cat 类并没有 walk() 方法。

程序中 ① 号代码为 Cat 动态添加了 walk() 方法，为类动态添加方法时不需要使用 MethodType 进行包装，该函数的第一个参数会自动绑定。为 Cat 动态添加 walk() 方法之后，Cat 类的两个实例 d1、d2 都具有了 walk() 方法，因此上面程序中最后两行 d1、d2 都可调用 walk() 方法。

Python 的这种动态性固然有其优势，但也给程序带来了一定的隐患，即程序定义好的类，完全有可能在后面被其他程序修改，这就带来了一些不确定性。如果程序要限制为某个类动态添加属性和方法，则可通过 __slots__ 属性来指定。

__slots__ 属性的值是一个元组，该元组的所有元素列出了该类的实例允许动态添加的所有属性名和方法名（对于 Python 而言，方法相当于属性值为函数的属性）。例如如下程序：

```
class Dog:
    __slots__ = ('walk', 'age', 'name')
    def __init__(self, name):
        self.name = name
    def test():
```

```
6.                  print('预先定义的 test 方法')
7.          d = Dog('Snoopy')
8.          from types import MethodType
9.          # 只允许动态为实例添加 walk、age、name 这 3 个属性或方法
10.         d.walk = MethodType(lambda self: print('%s 正在慢慢地走' % self.name), d)
11.         d.age = 5
12.         d.walk()
13.         d.foo = 30 # AttributeError
```

上面程序中第 2 行代码定义了 __slots__=('walk','age', 'name')，这意味着程序只允许为 Dog 实例动态添加 walk、age、name 这三个属性或方法。因此上面程序中第 10 行、第 11 行代码为 Dog 对象动态添加 walk() 方法和 age 属性都是允许的。

但如果程序尝试为 Dog 对象添加其他额外属性，程序就会引发 AttributeError 错误，如上面最后一行代码所示。运行上面程序，可以看到如下输出结果：

```
Snoopy 正在慢慢地走
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 13, in <module>
    d.foo = 30 # AttributeError
AttributeError: 'Dog' object has no attribute 'foo'
```

需要说明的是，__slots__ 属性并不限制通过类来动态添加属性或方法，因此下面代码是合法的：

```
1.          # __slots__属性并不限制通过类来动态添加方法
2.          Dog.bar = lambda self: print('abc') # AttributeError
```

```
3.          d.bar()
```

下面代码通过 Dog 类动态添加了 bar() 方法，这样 Dog 对象就可以调用该 bar() 方法了。

此外，__slots__ 属性指定的限制只对当前类的实例起作用，对该类派生出来的子类是不起作用的。例如如下代码：

```
1.          class GunDog(Dog):
2.              def __init__(self, name):
3.                  super().__init__(name)
4.              pass
5.          gd = GunDog('Puppy')
6.          # 完全可以为 Gundog 实例动态添加属性
7.          gd.speed = 99
8.          print(gd.speed)
```

正如从上面代码所看到的，Dog 的子类 GunDog 的实例完全可以动态添加 speed 属性，这说明 __slots__ 属性指定的限制只对当前类起作用。

如果要限制子类的实例动态添加属性和方法，则需要在子类中也定义 __slots__ 属性，这样，子类的实例允许动态添加属性和方法就是子类的 __slots__ 元组加上父类的 __slots__ 元组的和。

22.Python type()函数：动态创建类

type() 函数更适合于动态地创建相对简单的类，如果要创建更复杂的类，则需要通过 MetaClass（元类）的方式。

MetaClass（元类），简单的理解，就是创建类的类，即创建类之后，再由类来创建实例进行应用。使用元类可以在创建类时动态修改类定义。为了使用元类动态修改类定义，程序需要先定义元类。

注意，不要从字面上去理解元类的含义，事实上，MetaClass 中的 Meta 这个词根，起源于希腊语词汇 meta，包含“超越”和“改变”的意思。

定义元类时，需令其继承与 type 类，且默认的命名习惯是，让类名以 MetaClass 结尾。不仅如此，元类中需要定义并实现 __new__() 方法（一定要有返回值）。因为元类在创建类时，该 __new__() 方法将会被调用，用来生成新建的类。

> 之所有要求元类继承 type 并实现 __new__() 方法，是因为在创建类时，内部调用了 type 的 __new__() 方法为这个类分配内存空间，当内存分配完成后，便会调用 type 的 _init__ 方法初始化。

下面程序定义了一个元类类：

```
# 定义 Item 元类，继承 type
class ItemMetaClass(type):
    # cls 代表动态修改的类
    # name 代表动态修改的类名
    # bases 代表被动态修改的类的所有父类
    # attr 代表被动态修改的类的所有属性、方法组成的字典
    def __new__(cls, name, bases, attrs):
        # 动态为该类添加一个 cal_price 方法
        attrs['cal_price'] = lambda self: self.price * self.discount
        return type.__new__(cls, name, bases, attrs)
```

上面程序定义了一个 ItemMetaClass，该类继承了 type 类，并重写了 __new__ 方法，在重写该方法时为目标类动态添加了一个 cal_price 方法。

> 元类的 __new__ 方法的作用是：当程序使用 class 定义新类时，如果指定了元类，那么元类的 __new__ 方法就会被自动执行。

例如，如下程序使用元类定义了两个类：

```
# 定义 Book 类
class Book(metaclass=ItemMetaClass):
    __slots__ = ('name', 'price', '_discount')
    def __init__(self, name, price):
        self.name = name
        self.price = price
    @property
    def discount(self):
        return self._discount
    @discount.setter
    def discount(self, discount):
        self._discount = discount
# 定义 cellPhone 类
class CellPhone(metaclass=ItemMetaClass):
    __slots__ = ('price', '_discount' )
    def __init__(self, price):
        self.price = price
    @property
    def discount(self):
        return self._discount
    @discount.setter
    def discount(self, discount):
        self._discount = discount
```

上面程序定义了 Book 和 CellPhone 两个类，在定义这两个类时都指定了元类信息，因此当 Python 解释器在创建这两个类时，ItemMetaClass 的 __new__ 方法就会被调用，用于修改这两个类。

ItemMetaClass 类的 __new__ 方法会为目标类动态添加 cal_price 方法，因此，虽然在定义 Book、CellPhone 类时没有定义 cal_price() 方法，但这两个类依然有 cal_price() 方法。如下程序测试了 Book、CellPhone 两个类的 cal_price() 方法：

纯文本复制

```
b = Book("Python 基础教程", 89)
b.discount = 0.76
# 创建 Book 对象的 cal_price()方法
print(b.cal_price())
cp = CellPhone(2399)
cp.discount = 0.85
# 创建 CellPhone 对象的 cal_price()方法
print(cp.cal_price())
```

运行上面程序，可以看到如下运行结果：

```
67.64
2039.1499999999999
```

从上面的运行结果来看，通过使用元类可以动态修改程序中的一批类，对它们集中进行某种修改。这个功能在开发一些基础性框架时非常有用，程序可以通过使用元类为某一批需要具有通用功能的类添加方法。

23.Python MetaClass 元类

要理解 MetaClass 的底层原理，首先要深入理解 Python 类型模型。本节将从以下 2 点对 Python 类型模型做详细的介绍。

1）所有的 Python 的用户定义类，都是 type 这个类的实例

事实上，类本身不过是一个名为 type 类的实例，可以通过如下代码进行验证：

```
class MyClass:
  pass

instance = MyClass()
print(type(instance))
```

```
6.          print(type(MyClass))
```

输出结果为：

```
<class '__main__.MyClass'>

<class 'type'>
```

可以看到，instance 是 MyClass 的实例，而 MyClass 是 type 的实例。

2) 用户自定义类，只不过是 type 类的 __call__ 运算符重载。

当定义完成一个类时，真正发生的情况是 Python 会调用 type 类的 __call__ 运算符。

简单来说，当定义一个类时，例如下面语句：

```
1.          class MyClass:
2.            data = 1
```

Python 底层执行的是下面这段代码：

```
1.          class = type(classname, superclasses, attributedict)
```

其中等号右边的 type(classname, superclasses, attributedict) 就是 type 的 __call__ 运算符重载，它会进一步调用下面这 2 个函数：

```
1.          type.__new__(typeclass, classname, superclasses, attributedict)
2.          type.__init__(class, classname, superclasses, attributedict)
```

以上整个过程，可以通过如下代码进行论证：

```
1.          class MyClass:
2.            data = 1
```

```
3.
4.          instance = MyClass()
5.          print(MyClass,instance)
6.          print(instance.data)
7.          MyClass = type('MyClass', (), {'data': 1})
8.          instance = MyClass()
9.
10.         print(MyClass,instance)
11.         print(instance.data)
```

运行结果为：

```
<class '__main__.MyClass'> <__main__.MyClass object at 0x000001CB469F7400>
1
<class '__main__.MyClass'> <__main__.MyClass object at 0x000001CB46A50828>
1
```

由此可见，正常的 MyClass 定义，和手工调用 type 运算符的结果是完全一样的。

总之，正是 Python 的类创建机制，给了 metaclass 大展身手的机会，即一旦把一个类型 MyClass 设置成元类 MyMeta，那么它就不再由原生的 type 创建，而是会调用 MyMeta 的 __call__ 运算符重载：

```
1.          class = type(classname, superclasses, attributedict)
2.          # 变为了
3.          class = MyMeta(classname, superclasses, attributedict)
```

使用 metaclass 的风险

正如上面所看到的那样，metaclass 这样“逆天”的存在，会"扭曲变形"正常的 Python 类

型模型，所以，如果使用不慎，对于整个代码库造成的风险是不可估量的。

换句话说，metaclass 仅仅是给小部分 Python 开发者，在开发框架层面的 Python 库时使用的。而在应用层，metaclass 往往不是很好的选择。

建议初学者不要轻易尝试使用 mateclass。

24.Python 底层是如何实现 MetaClass 元类的？

对于弱类型的语言来说，变量并没有声明类型，因此同一个变量完全可以在不同的时间引用不同的对象。当同一个变量在调用同一个方法时，完全可能呈现出多种行为（具体呈现出哪种行为由该变量所引用的对象来决定），这就是所谓的多态（Polymorphism）。

先看下面程序：

```
class Bird:
    def move(self, field):
        print('鸟在%s 上自由地飞翔' % field)
class Dog:
    def move(self, field):
        print('狗在%s 里飞快的奔跑' % field)
# x 变量被赋值为 Bird 对象
x = Bird()
# 调用 x 变量的 move()方法
x.move('天空')
# x 变量被赋值为 Dog 对象
x = Dog()
# 调用 x 变量的 move()方法
x.move('草地')
```

上面程序中 x 变量开始被赋值为 Bird 对象，因此当 x 变量执行 move() 方法时，它会表现出鸟类的飞翔行为。接下来 x 变量被赋值为 Dog 对象，因此当 x 变量执行 move() 方法时，它会表现出狗的奔跑行为。

运行上面程序，可以看到如下运行结果：

```
鸟在天空上自由地飞翔
狗在草地里飞快的奔跑
```

从上面的运行结果可以看出，同一个变量 x 在执行同一个 move() 方法时，由于 x 指向的对象不同，因此它呈现出不同的行为特征，这就是多态。

看到这里，可能有读者感到失望，这个多态有什么用啊？不就是创建对象、调用方法吗？看不出多态有什么优势啊？

实际上，多态是一种非常灵活的编程机制。假如我们要定义一个 Canvas（画布）类，这个画布类定义一个 draw_pic() 方法，该方法负责绘制各种图形。该 Canvas 类的代码如下：

```
1.          class Canvas:
2.              def draw_pic(self, shape):
3.                  print('--开始绘图--')
4.                  shape.draw(self)
```

从上面代码可以看出，Canvas 的 draw_pic() 方法需要传入一个 shape 参数，该方法就是调用 shape 参数的 draw() 方法将自己绘制到画布上。

从上面程序来看，Canvas 的 draw_pic() 传入的参数对象只要带一个 draw() 方法就行，至于该方法具有何种行为（到底执行怎样的绘制行为），这与 draw_pic() 方法是完全分离的，这就为编程增加了很大的灵活性。下面程序定义了三个图形类，并为它们都提供了 draw() 方法，这样它们就能以不同的行为绘制在画布上，这就是多态的实际应用。看如下示例程序：

```
class Canvas:
    def draw_pic(self, shape):
        print('--开始绘图--')
        shape.draw(self)

class Rectangle:
    def draw(self, canvas):
        print('在%s 上绘制矩形' % canvas)
class Triangle:
    def draw(self, canvas):
        print('在%s 上绘制三角形' % canvas)
class Circle:
    def draw(self, canvas):
        print('在%s 上绘制圆形' % canvas)
c = Canvas()
# 传入 Rectangle 参数，绘制矩形
c.draw_pic(Rectangle())
# 传入 Triangle 参数，绘制三角形
c.draw_pic(Triangle())
# 传入 Circle 参数，绘制圆形
c.draw_pic(Circle())
```

运行上面代码， 可以看到如下输出结果：

```
--开始绘图--
在<__main__.Canvas object at 0x0000021CA36364A8>上绘制矩形
```

```
--开始绘图--
在<__main__.Canvas object at 0x0000021CA36364A8>上绘制三角形
--开始绘图--
在<__main__.Canvas object at 0x0000021CA36364A8>上绘制圆形
```

从上面这个例子可以体会到 Python 多态的优势。当程序涉及 Canvas 类的 draw_pic() 方法时，该方法所需的参数是非常灵活的，程序为该方法传入的参数对象只要具有指定方法就行，至于该方法呈现怎样的行为特征，则完全取决于对象本身，这大大提高了 draw_pic() 方法的灵活性。

25.Python 多态

对于弱类型的语言来说，变量并没有声明类型，因此同一个变量完全可以在不同的时间引用不同的对象。当同一个变量在调用同一个方法时，完全可能呈现出多种行为（具体呈现出哪种行为由该变量所引用的对象来决定），这就是所谓的多态（Polymorphism）。

先看下面程序：

```
class Bird:
    def move(self, field):
        print('鸟在%s 上自由地飞翔' % field)
class Dog:
    def move(self, field):
        print('狗在%s 里飞快的奔跑' % field)
# x 变量被赋值为 Bird 对象
x = Bird()
# 调用 x 变量的 move()方法
x.move('天空')
```

```
        # x 变量被赋值为 Dog 对象
        x = Dog()
        # 调用 x 变量的 move()方法
        x.move('草地')
```

上面程序中 x 变量开始被赋值为 Bird 对象，因此当 x 变量执行 move() 方法时，它会表现出鸟类的飞翔行为。接下来 x 变量被赋值为 Dog 对象，因此当 x 变量执行 move() 方法时，它会表现出狗的奔跑行为。

运行上面程序，可以看到如下运行结果：

```
鸟在天空上自由地飞翔
狗在草地里飞快的奔跑
```

从上面的运行结果可以看出，同一个变量 x 在执行同一个 move() 方法时，由于 x 指向的对象不同，因此它呈现出不同的行为特征，这就是多态。

看到这里，可能有读者感到失望，这个多态有什么用啊？不就是创建对象、调用方法吗？看不出多态有什么优势啊？

实际上，多态是一种非常灵活的编程机制。假如我们要定义一个 Canvas（画布）类，这个画布类定义一个 draw_pic() 方法，该方法负责绘制各种图形。该 Canvas 类的代码如下：

```
        class Canvas:
            def draw_pic(self, shape):
                print('--开始绘图--')
                shape.draw(self)
```

从上面代码可以看出，Canvas 的 draw_pic() 方法需要传入一个 shape 参数，该方法就是调用 shape 参数的 draw() 方法将自己绘制到画布上。

从上面程序来看，Canvas 的 draw_pic() 传入的参数对象只要带一个 draw() 方法就行，至于该方法具有何种行为（到底执行怎样的绘制行为），这与 draw_pic() 方法是完全分离的，这就为编程增加了很大的灵活性。下面程序定义了三个图形类，并为它们都提供了 draw() 方法，这样它们就能以不同的行为绘制在画布上，这就是多态的实际应用。看如下示例程序：

```
class Canvas:
    def draw_pic(self, shape):
        print('--开始绘图--')
        shape.draw(self)

class Rectangle:
    def draw(self, canvas):
        print('在%s 上绘制矩形' % canvas)
class Triangle:
    def draw(self, canvas):
        print('在%s 上绘制三角形' % canvas)
class Circle:
    def draw(self, canvas):
        print('在%s 上绘制圆形' % canvas)
c = Canvas()
# 传入 Rectangle 参数，绘制矩形
c.draw_pic(Rectangle())
# 传入 Triangle 参数，绘制三角形
c.draw_pic(Triangle())
# 传入 Circle 参数，绘制圆形
c.draw_pic(Circle())
```

运行上面代码， 可以看到如下输出结果：

```
--开始绘图--
在<__main__.Canvas object at 0x0000021CA36364A8>上绘制矩形
--开始绘图--
在<__main__.Canvas object at 0x0000021CA36364A8>上绘制三角形
--开始绘图--
在<__main__.Canvas object at 0x0000021CA36364A8>上绘制圆形
```

从上面这个例子可以体会到 Python 多态的优势。当程序涉及 Canvas 类的 draw_pic() 方法时，该方法所需的参数是非常灵活的，程序为该方法传入的参数对象只要具有指定方法就行，至于该方法呈现怎样的行为特征，则完全取决于对象本身，这大大提高了 draw_pic() 方法的灵活性。

26.Python 枚举类

在某些情况下，一个类的对象是有限且固定的，比如季节类，它只有 4 个对象；再比如行星类，目前只有 8 个对象。这种实例有限且固定的类，在 Python 中被称为枚举类。

程序有两种方式来定义枚举类：

1. 直接使用 Enum 列出多个枚举值来创建枚举类。
2. 通过继承 Enum 基类来派生枚举类。

如下程序示范了直接使用 Enum 列出多个枚举值来创建枚举类：

```
import enum
# 定义 Season 枚举类
Season = enum.Enum('Season', ('SPRING', 'SUMMER', 'FALL', 'WINTER'))
```

上面程序使用 Enum() 函数（就是 Enum 的构造方法）来创建枚举类，该构造方法的第一个参数是枚举类的类名；第二个参数是一个元组，用于列出所有枚举值。

在定义了上面的 Season 枚举类之后，程序可直接通过枚举值进行前问，这些枚举值都是该枚举的成员，每个成员都有 name、value 两个属性，其中 name 属性值为该枚举值的变量名，value 代表该枚举值的序号（序号通常从 1 开始）。

例如，如下代码测试了枚举成员的用法：

```
# 直接访问指定枚举
print(Season.SPRING)
# 访问枚举成员的变量名
print(Season.SPRING.name)
# 访问枚举成员的值
print(Season.SPRING.value)
```

运行该程序，可以看到如下输出结果：

```
Season.SPRING
SPRING
1
```

程序除可直接使用枚举之外，还可通过枚举变量名或枚举值来访问指定枚举对象。例如如下

代码：

```
# 根据枚举变量名访问枚举对象
print(Season['SUMMER']) # Season.SUMMER
# 根据枚举值访问枚举对象
print(Season(3)) # Season.FALL
```

此外，Python 还为枚举提供了一个 __members__ 属性，该属性返回一个 dict 字典，字典包含了该枚举的所有枚举实例。程序可通过遍历 __members__ 属性来访问枚举的所有实例。例如如下代码：

```
# 遍历 Season 枚举的所有成员
for name, member in Season.__members__.items():
    print(name, '=>', member, ',', member.value)
```

运行上面代码，可以看到如下输出结果：

```
SPRING => Season.SPRING , 1
SUMMER => Season.SUMMER , 2
FALL => Season.FALL , 3
WINTER => Season.WINTER , 4
```

如果要定义更复杂的枚举，则可通过继承 Enum 来派生枚举类，在这种方式下程序就可以为枚举额外定义方法了。例如如下程序：

```
import enum
class Orientation(enum.Enum):
    # 为序列值指定 value 值
    EAST = '东'
```

```
5.              SOUTH = '南'
6.              WEST = '西'
7.              NORTH = '北'
8.              def info(self):
9.                  print('这是一个代表方向【%s】的枚举' % self.value)
10.         print(Orientation.SOUTH)
11.         print(Orientation.SOUTH.value)
12.         # 通过枚举变量名访问枚举
13.         print(Orientation['WEST'])
14.         # 通过枚举值来访问枚举
15.         print(Orientation('南'))
16.         # 调用枚举的 info()方法
17.         Orientation.EAST.info()
18.         # 遍历 Orientation 枚举的所有成员
19.         for name, member in Orientation.__members__.items():
20.             print(name, '=>', member, ',', member.value)
```

上面程序通过继承 Enum 派生了 Orientation 枚举类，通过这种方式派生的枚举类既可额外定义方法，如上面的 info() 方法所示，也可为枚举指定 value（value 的值默认是 1、2、3、...）。

虽然此时 Orientation 枚举的 value 是由类型，但该枚举同样可通过 value 来访问特定枚举，如上面程序中的 Orientation('南')，这是完全允许的。运行上面代码，可以看到如下输出结果：

```
Orientation.SOUTH
南
Orientation.WEST
Orientation.SOUTH
```

这是一个代表方向【东】的枚举

EAST => Orientation.EAST，东

SOUTH => Orientation.SOUTH，南

WEST => Orientation.WEST，西

NORTH => Orientation.NORTH，北

枚举的构造器

枚举也是类，因此枚举也可以定义构造器。为枚举定义构造器之后，在定义枚举实例时必须为构造器参数设置值。例如如下程序：

```
import enum
class Gender(enum.Enum):
    MALE = '男', '阳刚之力'
    FEMALE = '女', '柔顺之美'
    def __init__(self, cn_name, desc):
        self._cn_name = cn_name
        self._desc = desc
    @property
    def desc(self):
        return self._desc
    @property
    def cn_name(self):
        return self._cn_name
# 访问 FEMALE 的 name
print('FEMALE 的 name:', Gender.FEMALE.name)
# 访问 FEMALE 的 value
print('FEMALE 的 value:', Gender.FEMALE.value)
# 访问自定义的 cn_name 属性
print('FEMALE 的 cn_name:', Gender.FEMALE.cn_name)
# 访问自定义的 desc 属性
print('FEMALE 的 desc:', Gender.FEMALE.desc)
```

上面程序定义了 Gender 枚举类，并为它定义了一个构造器，调用该构造器需要传入 cn_name 和 desc 两个参数，因此程序使用如下代码来定义 Gender 的枚举值。

```
MALE = '男', '阳刚之力'
FEMALE = '女', '柔顺之美'
```

上面代码为 MALE 枚举指定的 value 是‘男’和‘阳刚之力’这两个字符串，其实它们会被自动封装成元组后传给 MALE 的 value 属性；而且此处传入的‘男’和‘阳刚之力’这两个参数值正好分别传给 cnname 和 desc 两个参数。简单来说，枚举的构造器需要几个参数，此处就必须指定几个值。

运行上面程序，可以看到如下输出结果：

```
FEMALE 的 name: FEMALE
FEMALE 的 value: ('女', '柔顺之美')
FEMALE 的 cn_name: 女
FEMALE 的 desc: 柔顺之美
```

27.Python 项目实战：利用面向对象思想实现搜索引擎

引擎一词尤如其名，听起来非常酷炫，搜索引擎是新世纪初期互联网发展最重要的入口之一，依托搜索引擎，中国和美国分别诞生了百度、谷歌等巨型公司。

搜索引擎极大地方便了互联网生活，也成为上网必不可少的刚需工具。依托搜索引擎发展起

来的互联网广告，则成了硅谷和中国巨头的核心商业模式；而搜索本身，也在持续进步着，Facebook 和微信也一直有意向在自家社交产品架设搜索平台。

关于搜索引擎的价值就不必多说了，本节主要来看一下搜索引擎的核心构成。

要知道，一个搜索引擎由搜索器、索引器、检索器和用户接口四个部分组成。

搜索器，通俗来讲就是我们常提到的爬虫（scrawler），它能在互联网上大量爬取各类网站的内容，送给索引器。索引器拿到网页和内容后，会对内容进行处理，形成索引（index），存储于内部的数据库等待检索。

最后的用户接口很好理解，是指网页和 App 前端界面，例如百度和谷歌的搜索页面。用户通过用户接口，向搜索引擎发出询问（query），询问解析后送达检索器；检索器高效检索后，再将结果返回给用户。

爬虫知识不是本节学习的重点，不做深入介绍，我们假设搜索样本存在于本地磁盘上。

为了方便，这里只提供五个文件的检索，各文件存放的内容分别如下：

```
# 1.txt
C 语言中文网
# 2.txt
http://c.biancheng.net
# 3.txt
```

「C 语言中文网」是一个在线学习编程的网站，我们发布了多套文字教程，它们都通俗易懂，深入浅出。

4.txt

C 语言中文网成立于 2012 年初，目前已经运营了将近 7 年，我们致力于分享精品教程，帮助对编程感兴趣的读者。

5.txt

坚持做好一件事情，做到极致，让自己感动，让用户心动，这就是足以传世的作品！

先来定义 SearchEngineBase 基类，代码如下所示：

```
class SearchEngineBase(object):
    def __init__(self):
        pass

    def add_corpus(self, file_path):
        with open(file_path, 'rb') as fin:
            text = fin.read().decode('utf-8')
        self.process_corpus(file_path, text)

    def process_corpus(self, id, text):
        raise Exception('process_corpus not implemented.')

    def search(self, query):
        raise Exception('search not implemented.')

def main(search_engine):
    for file_path in ['1.txt', '2.txt', '3.txt', '4.txt', '5.txt']:
        search_engine.add_corpus(file_path)
    while True:
        query = input()
```

```
21.                 results = search_engine.search(query)
22.                 print('found {} result(s):'.format(len(results)))
23.                 for result in results:
24.                     print(result)
```

SearchEngineBase 类可以被继承，继承的类分别代表不同的算法引擎。每一个引擎都应该实现 process_corpus() 和 search() 两个函数，对应刚刚提到的索引器和检索器。main() 函数提供搜索器和用户接口，于是一个简单的包装界面就有了。

具体来看这段代码，其中 add_corpus() 函数负责读取文件内容，将文件路径作为 ID，连同内容一起送到 process_corpus 中；process_corpus 需要对内容进行处理，然后文件路径为 ID ，将处理后的内容存下来。处理后的内容，就叫做索引（index）；search 则给定一个询问，处理询问，再通过索引检索，然后返回。

理解了上面这些概念后，接下来实现一个最基本的可以工作的搜索引擎，代码如下：

纯文本复制

```
1.         class SimpleEngine(SearchEngineBase):
2.             def __init__(self):
3.                 super(SimpleEngine, self).__init__()
4.                 self.__id_to_texts = {}
5.
6.             def process_corpus(self, id, text):
7.                 self.__id_to_texts[id] = text
8.
9.             def search(self, query):
10.                 results = []
11.                 for id, text in self.__id_to_texts.items():
12.                     if query in text:
13.                         results.append(id)
```

```
14.                 return results
15.
16.         search_engine = SimpleEngine()
17.         main(search_engine)
```

运行结果为：

```
C 语言中文网
found 3 result(s):
1.txt
3.txt
4.txt
```

可以看到，只需要短短十来行代码就可以实现一个基础的搜索引擎。下面给大家分析一下这段代码：

SimpleEngine 实现了一个继承 SearchEngineBase 的子类，继承并实现了 process_corpus 和 search 接口，同时，也顺手继承了 add_corpus 函数（当然重写也是可行的），因此可以在 main() 函数中直接调取。

在新的构造函数中，self.__id_to_texts = {} 初始化了自己的私有变量，也就是这个用来存储文件名到文件内容的字典。

process_corpus() 函数则非常直白地将文件内容插入到字典中。这里注意，ID 需要是唯一的，不然相同 ID 的新内容会覆盖掉旧的内容。

search 直接枚举字典，从中找到要搜索的字符串。如果能够找到，则将 ID 放到结果列表中，最后返回。

是不是非常简单呢？整个过程始终贯穿着面向对象的思想。

第 9 章 Python 异常处理机制

异常机制已经成为判断一门编程语言是否成熟的标准，除传统的像 C 语言没有提供异常机制之外，目前主流的编程语言如 Python、 Java、 Kotlin 等都提供了成熟的异常机制。

异常机制可以使程序中的异常处理代码和正常业务代间分离，保证程序代码更加优雅，并可

以提高程序的健壮性。

Python 的异常机制主要依赖 try 、except 、else、finally 和 raise 五个关键字：

- try 关键字后缩进的代码块简称 try 块，它里面放置的是可能引发异常的代码；
- 在 except 后对应的是异常类型和一个代码块，用于表明该 except 块处理这种类型的代码块；
- 在多个 except 块之后可以放一个 else 块，表明程序不出现异常时还要执行 else 块；
- 最后还可以跟一个 finally 块，finally 块用于回收在 try 块里打开的物理资源，异常机制会保证 finally 块总被执行；
- raise 用于引发一个实际的异常，raise 可以单独作为语句使用，引发一个具体的异常对象；

1.什么是异常处理

开发人员在编写程序时，难免会遇到错误，有的是编写人员疏忽造成的语法错误，有的是程序内部隐含逻辑问题造成的数据错误，还有的是程序运行时与系统的规则冲突造成的系统错误，等等。

总的来说，编写程序时遇到的错误可大致分为 2 类，分别为语法错误和运行时错误。

Python 语法错误

语法错误，也就是解析代码时出现的错误。当代码不符合 Python 语法规则时，Python 解释器在解析时就会报出 SyntaxError 语法错误，与此同时还会明确指出最早探测到错误的语句。例如：

```
print "Hello,World!"
```

我们知道，Python 3 已不再支持上面这种写法，所以在运行时，解释器会报如下错误：

```
SyntaxError: Missing parentheses in call to 'print'
```

语法错误多是开发者疏忽导致的，属于真正意义上的错误，是解释器无法容忍的，因此，只有将程序中的所有语法错误全部纠正，程序才能执行。

Python 运行时错误

运行时错误，即程序在语法上都是正确的，但在运行时发生了错误。例如：

```
a = 1/0
```

上面这句代码的意思是“用 1 除以 0，并赋值给 a 。因为 0 作除数是没有意义的，所以运行后会产生如下错误：

```
>>> a = 1/0
Traceback (most recent call last):
```

```
  File "<pyshell#2>", line 1, in <module>
    a = 1/0
ZeroDivisionError: division by zero
```

以上运行输出结果中，前两段指明了错误的位置，最后一句表示出错的类型。在 Python 中，把这种运行时产生错误的情况叫做**异常（Exceptions）**。这种异常情况还有很多，常见的几种异常情况如表 1 所示。

表 1 Python 常见异常类型

异常类型	含义	实例

AssertionError	当 assert 关键字后的条件为假时，程序运行会停止并抛出 AssertionError 异常	>>> demo_list = ['C语言中文网']
		>>> assert len(demo_list) > 0
		>>> demo_list.pop()
		'C 语言中文网'
		>>> assert len(demo_list) > 0
		Traceback (most recent call last):
		  File "<pyshell#6>", line 1, in <module>
		    assert len(demo_list) > 0
		AssertionError

AttributeError	当试图访问的对象属性不存在时抛出的异常	>>> demo_list = ['C语言中文网']
		>>> demo_list.len
		Traceback (most recent call last):
		File "<pyshell#10>", line 1, in <module>
		demo_list.len
		AttributeError: 'list' object has no attribute 'len'

IndexError	索引超出序列范围会引发此异常	>>> demo_list = ['C语言中文网']
		>>> demo_list[3]
		Traceback (most recent call last):
		File "<pyshell#8>", line 1, in <module>
		demo_list[3]
		IndexError: list index out of range

KeyError	字典中查找一个不存在的关键字时	>>> demo_dict={'C 语言中文网

	引发此异常	':"c.biancheng.net"} >>> demo_dict["C 语言"] Traceback (most recent call last): File "<pyshell#12>", line 1, in <module> demo_dict["C 语言"] KeyError: 'C 语言'
NameError	尝试访问一个未声明的变量时，引发此异常	>>> C 语言中文网 Traceback (most recent call last): File "<pyshell#15>", line 1, in <module> C 语言中文网 NameError: name 'C语言中文网' is not defined
TypeError	不同类型数据之间的无效操作	>>> 1+'C 语言中文网' Traceback (most recent call last): File "<pyshell#17>", line 1, in <module>

		1+'C 语言中文网' TypeError: unsupported operand type(s) for +: 'int' and 'str'
ZeroDivisionError	除法运算中除数为 0 引发此异常	>>> a = 1/0 Traceback (most recent call last): File "<pyshell#2>", line 1, in <module> a = 1/0 ZeroDivisionError: division by zero

提示：表中的异常类型不需要记住，只需简单了解即可。

当一个程序发生异常时，代表该程序在执行时出现了非正常的情况，无法再执行下去。默认情况下，程序是要终止的。如果要避免程序退出，可以使用捕获异常的方式获取这个异常的名称，再通过其他的逻辑代码让程序继续运行，这种根据异常做出的逻辑处理叫作异常处理。

开发者可以使用异常处理全面地控制自己的程序。异常处理不仅仅能够管理正常的流程运行，还能够在程序出错时对程序进行必的处理。大大提高了程序的健壮性和人机交互的友好性。

那么，应该如何捕获和处理异常呢？可以使用 try 语句来实现。有关 try 语句的语法和用法，会在后续章节继续详解。

2.浅谈 Python 异常处理机制

异常处理机制已经成为衡量一门编程语言是否成熟的标准之一，使用异常处理机制的 Python 程序具有更好的容错性，更加健壮。

无论是多优秀的程序员，都无法保证自己写的程序永远不会出错，即便程序没有错误，也无法保证用户总是按自己的意愿来输入，就算用户都是非常“聪明而且配合”的，也无法保证运行该程序的操作系统永远稳定，无法保证运行该程序的硬件不会突然坏掉，无法保证网络永远通畅……无法保证的情况太多了。但作为一个程序设计人员，我们需要尽可能预知所有可能发生的情况，尽可能保证程序在所有糟糕的情形下都能正常运行。

考虑实现一个五子棋程序时，当用户输入下棋坐标时，程序要判断用户输入是否合法。如果保证程序有较好的容错性，将会有如下伪码：

```
if 用户输入包含除逗号之外的其他非数字字符：
    alert 坐标只能是数值
    goto retry
elif 用户输入不包含逗号：
    alert 应使用逗号分隔两个坐标值
    goto retry
elif 用户输入的坐标值超出了有效范围：
```

```
    alert 用户输入的坐标应位于棋盘坐标之内
    goto retry
elif 用户输入的坐标已有棋子：
    alert "只能在没有棋子的地方下棋"
    goto retry
else:
    #业务实现代码
    ...
```

上面代码还未涉及任何有效处理，只是考虑了 4 种可能的错误，代码量就已经急剧增加了。但实际上，上面考虑的 4 种情况还远未包括所有可能的情况。事实上，世界上的意外是不可穷举的，程序可能发生的异常情况总是多于程序员所能考虑到的意外情况。而且正如前面所提到的，高傲的程序员在开发程序时更倾向于认为“对，错误也许会发生，但那是别人造成的，不关我的事”。

如果每次在实现真正的业务逻辑之前，都需要不厌其烦地考虑各种可能出错的情况，针对各种错误情况给出补救措施，这是多么乏味的事情啊。程序员喜欢解决问题，喜欢开发带来的“创造”快感，但不喜欢像一个“堵漏”工人，去堵那些由外在条件造成的“漏洞”。

对于构造大型、健壮、可维护的应用而言，错误处理是整个应用需要考虑的重要方面，程序员不能仅仅只做“对”的事情，程序员开发程序的过程，是一个创造的过程，这个过程需要有全面的考虑，仅做“对”的事情是远远不够的。

对于上面的错误处理机制，主要有如下两个缺点：

1. 无法穷举所有的异常情况。因为人类知识的限制，异常情况总比可以考虑到的情况多，总有“漏网之鱼”的异常情况，所以程序总是不够健壮。
2. 错误处理代码和业务实现代码混杂。这种错误处理和业务实现混杂的代码严重影响程序的可读性，会增加程序维护的难度。

程序员希望有一种强大的机制来解决上面的问题，希望将上面程序改成如下伪码：

```
if  用户输入不合法:
    alert  输入不合法
    goto retry
else :
    #业务实现代码
    ...
```

上面伪码提供了一个非常强大的“if 块”，即程序不管输入错误的原因是什么，只要用户输入不满足要求，程序就一次处理所有的错误。这种处理方法的好处是，使得错误处理代码变得更有条理，只需在一个地方处理错误。

现在的问题是，“用户输入不合法”这个条件怎么定义？当然，对于这个简单的要求，可以使用正则表达式对用户输入进行匹配，当用户输入与正则表达式不匹配时即可判断“用户输

入不合法”。但对于更复杂的情形，就没有这么简单了。使用 Python 的异常处理机制就可以解决这个问题。

3.Python try except

Python 提供了 try except 语句捕获并处理异常，该异常处理语句的基本语法结构如下：

```
try:
    可能产生异常的代码块
except [(Error1, Error2, ...) [as e]]:
    处理异常的代码块 1
except [(Error3, Error4, ...) [as e]]:
    处理异常的代码块 2
```

该格式中，[] 括起来的部分可以使用，也可以省略；(Error1,Error2,...) 、(Error3,Error4,...) 表示各自的 except 块可以处理异常的具体类型；[as e] 表示将异常类型赋值给变量 e(方便在 except 块中调用异常类型)。

注意，except 后面也可以不指定具体的异常名称，这样的话，表示要捕获所有类型的异常。

另外，从 try except 的基本语法格式可以看出，try 块仅有一个，但 except 代码块可以有多个，这是为了针对不同的异常类型提供不同的异常处理方式。当程序发生不同的意外情况时，会对应不同的异常类型，Python 解释器就会根据该异常类型来决定使用哪个 except 块来处理该异常。

通过在 try 块后提供多个 except 块可以无须在异常处理块中使用 if 判断异常类型，但依然可以针对不同的异常类型提供相应的处理逻辑，从而提供更细致、更有条理的异常处理逻辑。

try except 语句的执行流程如下：

1. 首先执行 try 中的代码块，如果执行过程中出现异常，系统会自动生成一个异常对象，该异常对象会提交给 Python 解释器，此过程被称为引发异常。
2. 当 Python 解释器收到异常对象时，会寻找能处理该异常对象的 except 块，如果找到合适的 except 块，则把该异常对象交给该 except 块处理，这个过程被称为捕获异常。如果 Python 解释器找不到捕获异常的 except 块，则程序运行终止，Python 解释器也将退出。

事实上，不管程序代码块是否处于 try 块中，甚至包括 except 块中的代码，只要执行该代码块时出现了异常，系统总会自动生成一个 Error 对象。如果程序没有为这段代码定义任何的 except 块，则 Python 解释器无法找到处理该异常的 except 块，程序就会停止运行；反之，如果程序发生异常，并且该异常经 try 捕获并由 except 处理完成，则程序会继续执行。

举个例子：

```
1.          try:
2.              a = int(input("输入被除数："))
3.              b = int(input("输入除数："))
```

```
            c = a / b
            print("您输入的两个数相除的结果是：", c )
        except (ValueError, ArithmeticError):
            print("程序发生了数字格式异常、算术异常之一")
        except :
            print("未知异常")
        print("程序继续运行")
```

程序运行结果为：

```
输入被除数：a
程序发生了数字格式异常、算术异常之一
程序继续运行
```

上面程序中，第 6 行代码使用了（ValueError, ArithmeticError）来指定所捕获的异常类型，这就表明该 except 块可以同时捕获这 2 种类型的异常；第 8 行代码只有 except 关键字，并未指定具体要捕获的异常类型，这种省略异常类的 except 语句也是合法的，它表示可捕获所有类型的异常，一般会作为异常捕获的最后一个 except 块。

除此之外，由于 try 块中引发了异常，并被 except 块成功捕获，因此程序才可以继续执行，才有了“程序继续运行”的输出结果。

访问异常信息

如果程序需要在 except 块中访问异常对象的相关信息，可以通过为 except 块添加 as a 来实现。当 Python 解释器决定调用某个 except 块来处理该异常对象时，会将异常对象赋值给 except 块后的异常变量，程序即可通过该变量来获得异常对象的相关信息。

所有的异常对象都包含了如下几个常用属性和方法：

- args：该属性返回异常的错误编号和描述字符串。
- errno：该属性返回异常的错误编号。
- strerror：该属性返回异常的描述字符串。
- with_traceback()：通过该方法可处理异常的传播轨迹信息。

下面例子演示了程序如何访问异常信息：

```
1.          def foo():
2.              try:
3.                  fis = open("a.txt");
4.              except Exception as e:
5.                  # 访问异常的错误编号和详细信息
6.                  print(e.args)
7.                  # 访问异常的错误编号
8.                  print(e.errno)
9.                  # 访问异常的详细信息
10.                 print(e.strerror)
11.         foo()
```

从上面程序可以看出，如果要访问异常对象，只要在单个异常类或异常类元组（多异常捕获）之后使用 as 再加上异常变量即可。

> 在 Python 的早期版本中，直接在单个异常类或异常类元组（多异常捕获）之后添加异常变量，中间用逗号隔开即可。

上面程序调用了 Exception 对象的 args 属性（该属性相当于同时返回 errno 属性和

strerror 属性）访问异常的错误编号和详细信息。运行上面程序，会看到如下运行结果：

```
(2, 'No such file or directory')
2
No such file or directory
```

从上面的运行结果可以看到，由于程序尝试打开的文件不存在，因此引发的异常错误编号为 2，异常详细信息为：No such file or directory。

关于如何处理异常的传播轨迹信息，以及使用 open() 方法来打开一个文件，用于读取磁盘文件的内容，后续章节还有更详细的介绍，此处暂不详细讲解。

异常类的继承体系

当 Python 解释器接收到异常对象时，如何为该异常对象寻找 except 块呢？注意上面程序中 except 块的 except Exception，这意味着此 except 块专门用来处理该异常类以及其子类的异常实例。

当 Python 解释器接收到异常对象后，会依次判断该异常对象是否是 except 块后的异常类或其子类的实例，如果是，Python 解释器将调用该 except 块来处理该异常；否则，再次拿该异常对象和下一个 except 块里的异常类进行比较。

Python 异常捕获流程示意图如图 1 所示：

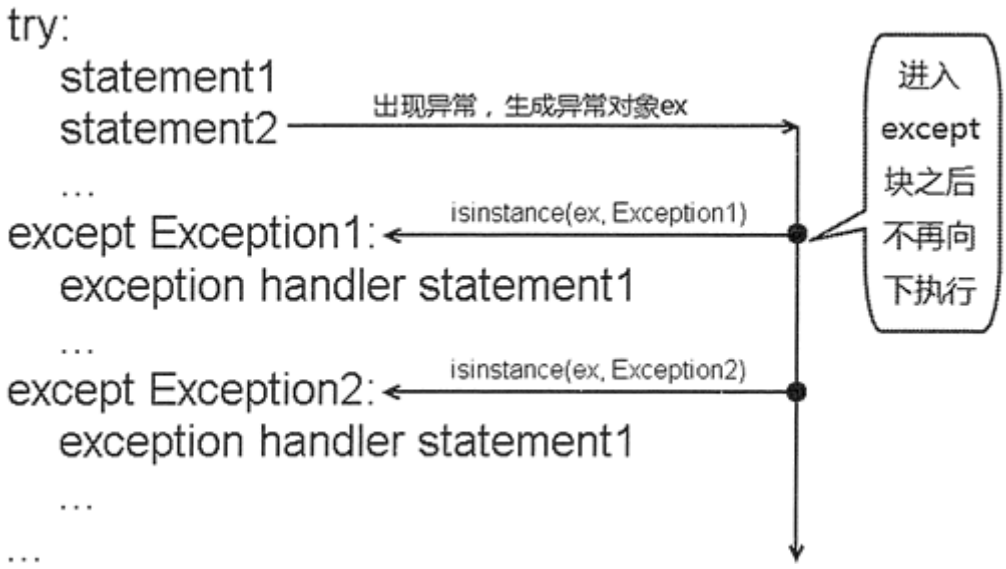


图 1 Python 异常捕获流程示意图

从图 1 中可以看出 ,在通常情况下 ,如果 try 块被执行一次 ,则 try 块后只有一个 except 块会被执行，不可能有多个 except 块被执行。除非在循环中使用了 continue 开始下一次循环，下一次循环又重新运行了 try 块，这才可能导致多个 except 块被执行。

Python 的所有异常类都从 BaseException 派生而来，提供了丰富的异常类，这些异常类之间有严格的继承关系，图 2 显示了 Python 的常见异常类之间的继承关系。

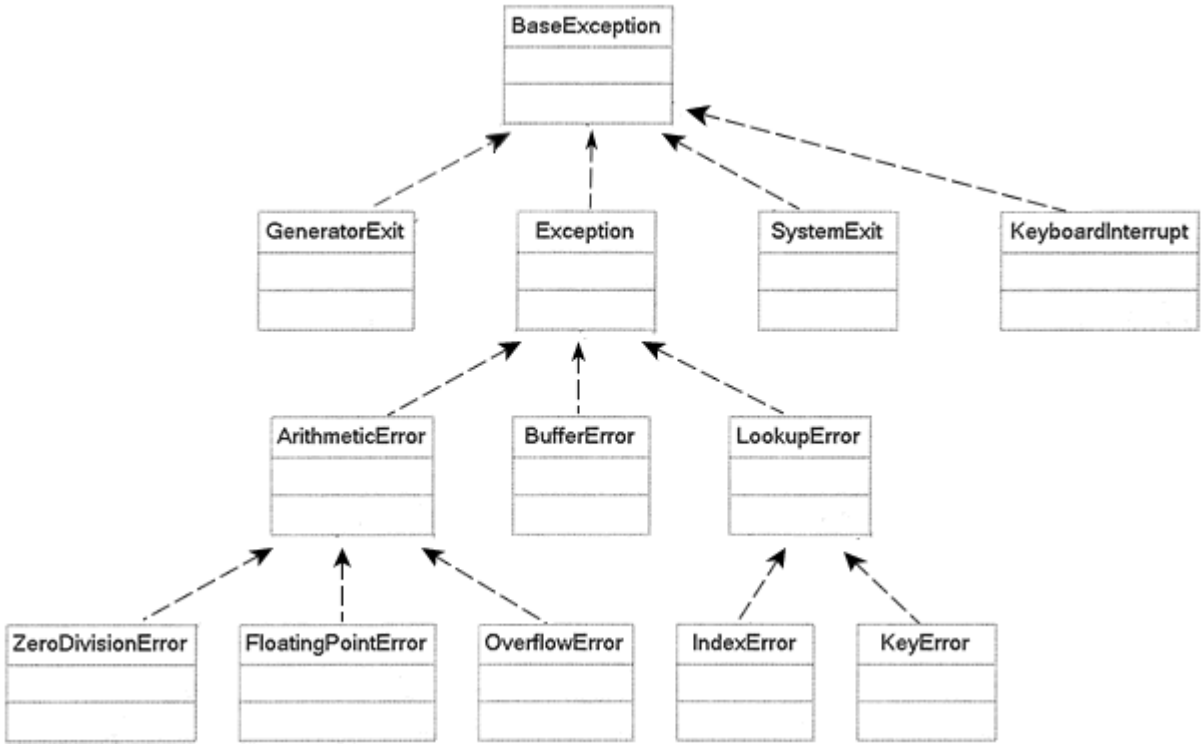


图 2 Python 的常见异常类之间的继承关系

从图 2 中可以看出，Python 的所有异常类的基类是 BaseException，但如果用户要实现自定义异常，则不应该继承这个基类，而是应该继承 Exception 类。

> BaseException 的主要子类就是 Exception，不管是系统的异常类，还是用户自定义的异常类，都应该从 Exception 派生。

下面看几个简单的异常捕获的例子：

```
1.          try:
2.                  a = int(input("输入被除数："))
3.                  b = int(input("输入除数："))
4.                  c = a / b
5.                  print("您输入的两个数相除的结果是：", c )
```

```
        except ValueError:
            print("数值错误：程序只能接收整数参数")
        except ArithmeticError:
            print("算术错误")
        except Exception:
            print("未知异常")
```

上面程序针对 ValueError、ArithmeticError 类型的异常，提供了专门的异常处理逻辑。该程序运行时的异常处理逻辑可能有如下几种情形：

- 如果在运行该程序时输入的参数不是数字，而是字母，将发生数值错误，Python 将调用 ValueError 对应的 except 块处理该异常。
- 如果在运行该程序时输入的第二个参数是 0，将发生除 0 异常，Python 将调用 ArithmeticError 对应的 except 块处理该异常。
- 如果在程序运行时出现其他异常，该异常对象总是 Exception 类或其子类的实例，Python 将调用 Exception 对应的 except 块处理该异常。

> 上面程序中的异常类型，都是非常常见的运行时异常，读者应该记住这些异常，并掌握在哪些情况下可能出现这些异常。

正如在前面程序中所看到的，程序总是把对应 Exception 类的 except 块放在最后，这是为什么呢？想一下图 1 所示的 Python 异常捕获流程，可能你就会明白，如果把 Exception 类对应的 except 块排在其他 except 块的前面，Python 解释器将直接进入该 except 块（因为所有的异常对象都是 Exception 或其子类的实例），而排在它后面的 except 块将永远也不会获得执行的机会。

实际上，在进行异常捕获时，不仅应该把 Exception 类对应的 except 块放在最后，而且所有父类异常的 except 块都应该排在子类异常的 except 块的后面(即 :先处理小异常，再处理大异常)。

> 虽然 Python 语法没有要求，但在实际编程时一定要记住先捕获小异常，再捕获大异常。

4.Python try except else

在 Python 中，还有另一种异常处理结构，就是 try except else 语句，也就是在原来 try except 语句的基础上再添加一个 else 子句，其作用是指定当 try 块中没有发现异常时要执行的代码。换句话说，当 try 块中发现异常，则 else 块中的语句将不会被执行。

例如如下程序：

```
1.          s = input('请输入除数:')
2.          try:
3.              result = 20 / int(s)
4.              print('20 除以%s 的结果是: %g' % (s , result))
5.          except ValueError:
6.              print('值错误，您必须输入数值')
7.          except ArithmeticError:
8.              print('算术错误，您不能输入 0')
9.          else:
10.             print('没有出现异常')
11.         print("程序继续运行")
```

上面程序为异常处理流程添加了 else 块，当程序中的 try 块没有出现异常时，程序就会执行 else 块。运行上面程序，如果用户输入导致程序中的 try 块出现了异常，则运行结

果如下：

```
请输入除数：a
值错误，您必须输入数值
程序继续运行
```

如果用户输入让程序中的 try 块顺利完成，则运行结果如下：

```
请输入除数：3
20 除以 3 的结果是：6.66667
没有出现异常
程序继续运行
```

看到这里，可能有读者觉得奇怪，既然只有当 try 块没有异常时才会执行 else 块，那么直接把 else 块的代码放在 try 块的代码的后面不就行了？

实际上大部分语言的异常处理都没有 else 块，它们确实是将 else 块的代码直接放在 try 块的代码的后面的，因为对于大部分场景而言，直接将 else 块的代码放在 try 块的代码的后面即可。

但 Python 的异常处理使用 else 块绝不是多余的语法，当 try 块没有异常，而 else 块有异常时，就能体现出 else 块的作用了。例如如下程序：

```
1.          def else_test():
```

```
            s = input('请输入除数:')
            result = 20 / int(s)
            print('20 除以%s 的结果是: %g' % (s , result))
        def right_main():
            try:
                print('try 块的代码，没有异常')
            except:
                print('程序出现异常')
            else:
                # 将 else_test 放在 else 块中
                else_test()
        def wrong_main():
            try:
                print('try 块的代码，没有异常')
                # 将 else_test 放在 try 块代码的后面
                else_test()
            except:
                print('程序出现异常')
        wrong_main()
        right_main()
```

上面程序中定义了一个 else_test() 函数，该函数在运行时需要接收用户输入的参数，随着用户输入数据的不同可能导致异常。接下来程序定义了 right_main() 和 wrong_main() 两个函数，其中 right_main() 将 else_test() 函数放在 else 块内；而 wrong_main() 将 else_test() 函数放在 try 块的代码的后面。

正如上面所介绍的，当 try 块和 else 块都没有异常时，将 else_test() 函数放在 try 块的代码的后面和放在 else 块中没有任何区别。例如，如果用户输入的数据没有导致程序出现异常，则将看到程序产生如下输出结果：

```
try 块的代码，没有异常
请输入除数:4
20 除以 4 的结果是: 5
try 块的代码，没有异常
请输入除数:4
20 除以 4 的结果是: 5
```

但如果用户输入的数据让 else_test() 函数出现异常（try 块依然没有任何异常），此时程序就会产生如下输出结果：

```
try 块的代码，没有异常
请输入除数:0
程序出现异常
try 块的代码，没有异常
请输入除数:0
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 21, in <module>
    right_main()
  File "C:\Users\mengma\Desktop\1.py", line 12, in right_main
    else_test()
  File "C:\Users\mengma\Desktop\1.py", line 3, in else_test
    result = 20 / int(s)
ZeroDivisionError: division by zero
```

对比上面两个输出结果，用户输入的都是 0，这样都会导致 else_test() 函数出现异常。如果将 else_test() 函数放在 try 块的代码的后面，此时 else_test() 函数运行产生的异常将会被 try 块对应的 except 捕获，这正是 Python 异常处理机制的执行流程：但如果将 else_test() 函数放在 else 块中，当 else_test() 函数出现异常时，程序没有 except 块来处理该异常，该异常将会传播给 Python 解释器，导致程序中止。

> 对比上面两个输出结果，不难发现，放在 else 块中的代码所引发的异常不会被 except 块捕获。

所以，如果希望某段代码的异常能被后面的 except 块捕获，那么就应该将这段代码放在 try 块的代码之后；如果希望某段代码的异常能向外传播（不被 except 块捕获），那么就应该将这段代码放在 else 块中。

5.Python try except finally

Python 中，finally 语句是与 try 和 except 语句配合使用的，其通常是用来做清理工作的。无论 try 中的语句是否跳入 except 中，最终都要进入 finally 语句，并执行其中的代码块。

有些时候，程序在 try 块里打开了一些物理资源（例如数据库连接、网络连接和磁盘文件等），这些物理资源都必须被显式回收。Python 的垃圾回收机制不会回收任何物理资源，只能回收内存中对象所占用的内存。

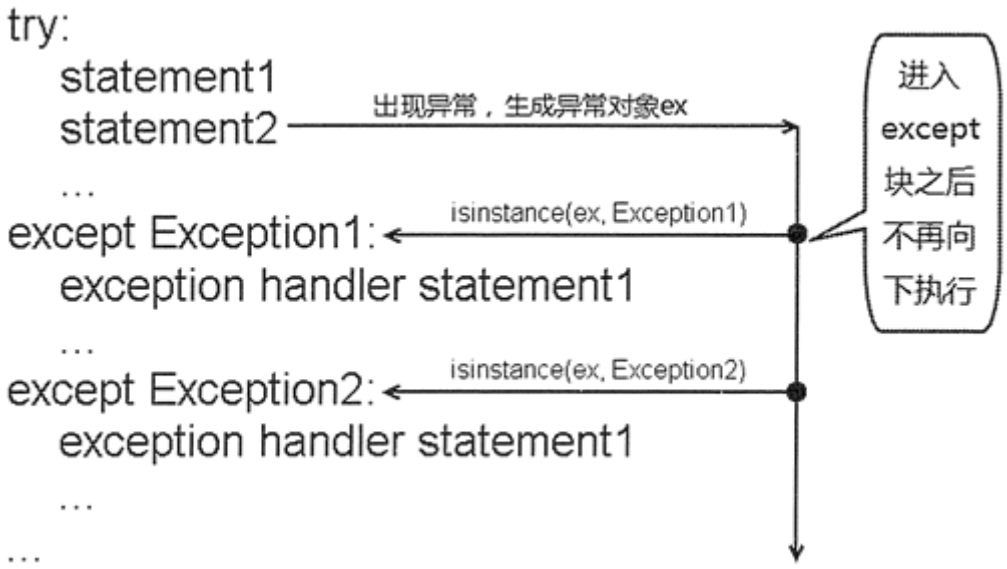


图 1 Python 异常捕获流程示意图

那么在哪里回收这些物理资源呢？在 try 块里回收，还是在 except 块中进行回收？假设程序在 try 块里进行资源回收，根据图 1 所示的异常捕获流程，如果 try 块的某条语句引发了异常，该语句后的其他语句通常不会获得执行的机会，这将导致位于该语句之后的资源回收语句得不到执行。如果在 except 块里进行资源回收，因为 except 块完全有可能得不到执行，这将导致不能及时回收这些物理资源。

为了解决这个问题，一般会在 except 的最后加个 finally 语句。不管 try 块中的代码是否出现异常，也不管哪一个 except 块被执行，甚至在 try 块或 except 块中执行了 return 语句，finally 块总会被执行。

Python 完整的异常处理语法结构如下：

```
try:
    #业务实现代码
except SubException as e:
    #异常处理块 1
    ...
except SubException2 as e:
    #异常处理块 2
    ...
else:
    #正常处理块
finally :
    #资源回收块
    ...
```

各个语句块的具体执行过程，如图 1 所示：

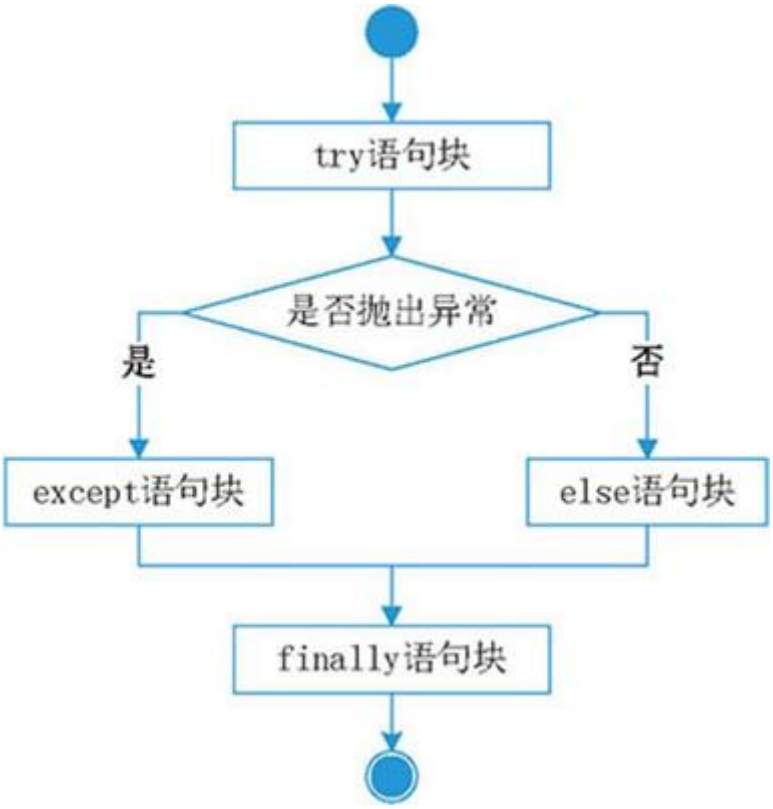


图 1 异常处理语句块的执行流程

在异常处理语法结构中，只有 try 块是必需的，也就是说：

- 如果没有 try 块，则不能有后面的 except 块和 finally 块；
- except 块和 finally 块都是可选的，但 except 块和 finally 块至少出现其中之一，也可以同时出现；
- 可以有多个 except 块，但捕获父类异常的 except 块应该位于捕获子类异常的 except 块的后面；
- 不能只有 try 块，既没有 except 块，也没有 finally 块；
- 多个 except 块必须位于 try 块之后 ,finally 块必须位于所有的 except 块之后。

finally 语句块和 else 语句块的区别是，else 语句块只有在没有异常发生的情况下才会执行，而 finally 语句则不管异常是否发生都会执行。不仅如此，无论是正常退出、异常退出，还是通过 break、continue、return 语句退出，finally 语句块都会执行。

例如如下程序：

```
import os
def test():
    fis = None
    try:
        fis = open("a.txt")
    except OSError as e:
        print(e.strerror)
        # return 语句强制方法返回
        return        # ①
        #os._exit(1)     # ②
    finally:
        # 关闭磁盘文件，回收资源
        if fis is not None:
            try:
                # 关闭资源
                fis.close()
            except OSError as ioe:
                print(ioe.strerror)
        print("执行 finally 块里的资源回收!")
test()
```

上面程序在 try 块后增加了 finally 块，用于回收在 try 块中打开的物理资源。注意在程序的 except 块中 ① 处有一条 return 语句，该语句强制方法返回。在通常情况下，一旦在方法里执行到 return 语句，程序将立即结束该方法：现在不会了，虽然 return 语句也强制方法结束，但一定会先执行 finally 块的代码。

运行上面程序，将看到如下运行结果：

```
No such file or directory
执行 finally 里的资源回收!
```

上面的运行结果表明在方法返回之前执行了 finally 块的代码。将 ① 处的 return 语句注释掉，取消 ② 处代码的注释，即在异常处理的 except 块中使用 os._exit(1) 语句来退出 Python 解释器。运行上面代码，将看到如下运行结果：

```
No such file or directory
```

上面的运行结果表明 finally 块没有被执行。如果在异常处理代码中使用 os.exit(1) 语句来退出 Python 解释器，则 finally 块将失去执行的机会。

除非在 try 块、except 块中调用了退出 Python 解释器的方法，否则不管在 try 块、except 块中执行怎样的代码，出现怎样的情况，异常处理的 finally 块总会被执行。调用 sys.exit() 方法退出程序不能阻止 finally 块的执行，这是因为 sys.exit() 方法本身就是通过引发 SystemExit 异常来退出程序的。

在通常情况下，不要在 finally 块中使用如 return 或 raise 等导致方法中止的语句(raise 语句将在后面介绍),一旦在 finally 块中使用了 return 或 raise 语句，将会导致 try 块、except 块中的 return、raise 语句失效。看如下程序：

```
1.          def test():
```

```
        try:
            # 因为 finally 块中包含了 return 语句
            # 所以下面的 return 语句失去作用
            return True
        finally:
            return False
    a = test()
    print(a)
```

上面程序在 finally 块中定义了一条 return False 语句，这将导致 try 块中的 return true 失去作用。运行上面程序，将打印出 False 的结果。

如果 Python 程序在执行 try 块、except 块时遇到了 return 或 raise 语句，这两条语句都会导致该方法立即结束，那么系统执行这两条语句并不会结束该方法，而是去寻找该异常处理流程中的 finally 块，如果没有找到 finally 块，程序立即执行 return 或 raise 语句，方法中止；如果找到 finally 块，系统立即开始执行 finally 块，只有当 finally 块执行完成后，系统才会再次跳回来执行 try 块、except 块里的 return 或 raise 语句；如果在 finally 块里也使用了 return 或 raise 等导致方法中止的语句，finally 块已经中止了方法，系统将不会跳回去执行 try 块、except 块里的任何代码。

> 尽量避免在 finally 块里使用 return 或 raise 等导致方法中止的语句，否则可能出现一些很奇怪的情况。

6.Python raise

有读者可能会问，编写代码时能否手动抛出一个异常吗？答案是肯定的，Python 允许程序

自行引发异常，使用 raise 语句即可。

> 异常是一种很“主观”的说法，以下雨为例，假设大家约好明天去爬山郊游，如果第二天下雨了，这种情况会打破既定计划，就属于一种异常；但对于正在期盼天降甘霖的农民而言，如果第二天下雨了，他们正好随雨追肥，这就完全正常。

很多时候，系统是否要引发异常，可能需要根据应用的业务需求来决定，如果程序中的数据、执行与既定的业务需求不符，这就是一种异常。由于与业务需求不符而产生的异常，必须由程序员来决定引发，系统无法引发这种异常。

如果需要在程序中自行引发异常，则应使用 raise 语句，该语句的基本语法格式为：

```
raise [exceptionName [(reason)]]
```

其中，用 [] 括起来的为可选参数，其作用是指定抛出的异常名称，以及异常信息的相关描述。如果可选参数全部省略，则 raise 会把当前错误原样抛出；如果仅省略（reason），则在抛出异常时，将不附带任何的异常描述信息。

也就是说，raise 语句有如下三种常用的用法：

1. raise：单独一个 raise。该语句引发当前上下文中捕获的异常（比如在 except 块中），或默认引发 RuntimeError 异常。
2. raise 异常类名称 :raise 后带一个异常类名称。该语句引发指定异常类的默认实例。
3. raise 异常类名称(描述信息)：在引发指定异常的同时，附带异常的描述信息。

上面三种用法最终都是要引发一个异常实例（即使指定的是异常类，实际上也是引发该类的默认实例），raise 语句每次只能引发一个异常实例。

例如：

```
>>> raise
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    raise
RuntimeError: No active exception to reraise
>>> raise ZeroDivisionError
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    raise ZeroDivisionError
ZeroDivisionError
>>> raise ZeroDivisionError("除数不能为零")
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    raise ZeroDivisionError("除数不能为零")
ZeroDivisionError: 除数不能为零
```

注意，即使是用户自行引发的异常，也可以使用 `try except` 来捕获它。当然也可以不管它，

让该异常向上（先调用者）传播，如果该异常传到 Python 解释器，那么程序就会中止。

下面示例示范了处理用户引发异常的两种方式：

```
        def main():
            try:
                # 使用 try...except 来捕捉异常
                # 此时即使程序出现异常，也不会传播给 main 函数
                mtd(3)
            except Exception as e:
                print('程序出现异常:', e)
            # 不使用 try...except 捕捉异常，异常会传播出来导致程序中止
            mtd(3)
        def mtd(a):
            if a > 0:
                raise ValueError("a 的值大于 0，不符合要求")
        main()
```

从上面程序可以看到，程序既可在调用 mtd(3) 时使用 `try except` 来捕获异常，这样该异常将会被 except 块捕获；也可直接调用 mtd(3)，这样该函数的异常就会直接导致程序中止。

运行上面程序，可以看到如下输出结果：

```
程序出现异常: a 的值大于 0，不符合要求
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 13, in <module>
    main()
```

```
  File "C:\Users\mengma\Desktop\1.py", line 9, in main
    mtd(3)
  File "C:\Users\mengma\Desktop\1.py", line 12, in mtd
    raise ValueError("a 的值大于 0，不符合要求")
ValueError: a 的值大于 0，不符合要求
```

上面第一行输出是第一次调用 mtd (3) 的结果，该方法引发的异常被 except 块捕获并处理。后面的大段输出则是第二次调用 mtd(3) 的结果，由于该异常没有被 except 块捕获，导致程序中止。

第二次调用 mtd(3) 引发的以“File”开头的三行输出，其实显示的就是异常的传播轨迹信息。也就是说，如果程序不对异常进行处理，Python 默认会在控制台输出异常的传播轨迹信息。

raise 不需要参数

正如前面所看到的，在使用 raise 语句时可以不带参数，此时 raise 语句处于 except 块中，它将会自动引发当前上下文激活的异常；否则，通常默认引发 RuntimeError 异常。

例如，将上面程序改为如下形式：

```
1.          class AuctionException(Exception): pass
2.          class AuctionTest:
3.              def __init__(self, init_price):
4.                  self.init_price = init_price
5.              def bid(self, bid_price):
```

```
6.                  d = 0.0
7.                  try:
8.                      d = float(bid_price)
9.                  except Exception as e:
10.                     # 此处只是简单地打印异常信息
11.                     print("转换出异常：", e)
12.                     # 再次引发自定义异常
13.                     raise
14.                 if self.init_price > d:
15.                     raise AuctionException("竞拍价比起拍价低，不允许竞拍！")
16.                 initPrice = d
17.         def main():
18.             at = AuctionTest(20.4)
19.             try:
20.                 at.bid("df")
21.             except AuctionException as ae:
22.                 # 再次捕获到 bid()方法中的异常，并对该异常进行处理
23.                 print('main 函数捕捉的异常：', ae)
24.         main()
```

正如从 13 行代码所看到的，此时程序在 except 块中只是简单地使用 raise 语句来引发异常，那么该 raise 语句将会再次引发该 except 块所捕获的异常。运行该程序，可以看到如下输出结果：

```
转换出异常： could not convert string to float: 'df'
main 函数捕获的异常：<class 'ValueError'>
```

从输出结果来看，此时 main() 函数再次捕获了 ValueError，它就是在 bid() 方法中 except 块所捕获的原始异常。

except 和 raise 同时使用

在实际应用中对异常可能需要更复杂的处理方式。当一个异常出现时，单靠某个方法无法完全处理该异常，必须由几个方法协作才可完全处理该异常。也就是说，在异常出现的当前方法中，程序只对异常进行部分处理，还有些处理需要在该方法的调用者中才能完成，所以应该再次引发异常，让该方法的调用者也能捕获到异常。

为了实现这种通过多个方法协作处理同一个异常的情形，可以在 except 块中结合 raise 语句来完成。如下程序示范了 except 和 raise 同时使用的方法：

```
1.          class AuctionException(Exception): pass
2.          class AuctionTest:
3.              def __init__(self, init_price):
4.                  self.init_price = init_price
5.              def bid(self, bid_price):
6.                  d = 0.0
7.                  try:
8.                      d = float(bid_price)
9.                  except Exception as e:
10.                     # 此处只是简单地打印异常信息
11.                     print("转换出异常：", e)
12.                     # 再次引发自定义异常
13.                     raise AuctionException("竞拍价必须是数值，不能包含其他字符！
    ")  # ①
14.                     raise AuctionException(e)
15.                 if self.init_price > d:
16.                     raise AuctionException("竞拍价比起拍价低，不允许竞拍！")
17.                 initPrice = d
18.         def main():
19.             at = AuctionTest(20.4)
20.             try:
```

```
                at.bid("df")
            except AuctionException as ae:
                # 再次捕获到 bid()方法中的异常，并对该异常进行处理
                print('main 函数捕捉的异常：', ae)
        main()
```

上面程序中 9~13 行代码对应的 except 块捕获到异常后，系统打印了该异常的字符串信息，接着引发一个 AuctionException 异常，通知该方法的调用者再次处理该 AuctionException 异常。所以程序中的 main() 函数，也就是 bid() 方法的调用者还可以再次捕获 AuctionException 异常，并将该异常的详细描述信息打印出来。

这种 except 和 raise 结合使用的情况在实际应用中非常常用。实际应用对异常的处理通常分成两个部分：

1. 应用后台需要通过日志来记录异常发生的详细情况；
2. 应用还需要根据异常向应用使用者传达某种提示；

在这种情形下，所有异常都需要两个方法共同完成，也就必须将 except 和 raise 结合使用。

如果程序需要将原始异常的详细信息直接传播出去，Python 也允许用自定义异常对原始异常进行包装，只要将上面 ① 号代码改为如下形式即可：

```
raise AuctionException(e)
```

上面就是把原始异常 e 包装成了 AuctionException 异常 ,这种方式也被称为异常包装或异常转译。

自定义异常类

很多时候，程序可选择引发自定义异常，因为异常的类名通常也包含了该异常的有用信息。所以在引发异常时，应该选择合适的异常类，从而可以明确地描述该异常情况。在这种情形下，应用程序常常需要引发自定义异常。

用户自定义异常都应该继承 Exception 基类或 Exception 的子类，在自定义异常类时基本不需要书写更多的代码，只要指定自定义异常类的父类即可。

下面程序创建了一个自定义异常类：

```
class AuctionException(Exception):
    pass
```

上面程序创建了 AuctionException 异常类 ,该异常类不需要类体定义 ,因此使用 pass 语句作为占位符即可。

在大部分情况下，创建自定义异常类都可采用与上面程序相似的代码来完成，只需改变 AuctionException 异常的类名即可，让该异常的类名可以准确地描述该异常。

7.Python sys.exc_info()获取异常信息

在实际调试程序的过程中，有时只获得异常的类型是远远不够的，还需要借助更详细的异常信息才能解决问题。

捕获异常时，有 2 种方式可获得更多的异常信息，分别是：

1. 使用 sys 模块中的 exc_info 方法；
2. 使用 traceback 模块中的相关函数。

本节首先介绍如何使用 sys 模块中的 exc_info() 方法获得更多的异常信息。

> 有关 sys 模块更详细的介绍，可阅读《Python sys 模块》。

模块 sys 中，有两个方法可以返回异常的全部信息，分别是 exc_info() 和 last_traceback()，这两个函数有相同的功能和用法，本节仅以 exc_info() 方法为例。

exc_info() 方法会将当前的异常信息以元组的形式返回，该元组中包含 3 个元素，分别为 type、value 和 traceback，它们的含义分别是：

- type：异常类型的名称，它是 BaseException 的子类（有关 Python 异常类，可阅读《Python 常见异常类型》一节）
- value：捕获到的异常实例。
- traceback：是一个 traceback 对象。

举个例子：

```
#使用 sys 模块之前，需使用 import 引入
import sys
try:
    x = int(input("请输入一个被除数："))
    print("30 除以",x,"等于",30/x)
except:
    print(sys.exc_info())
    print("其他异常...")
```

当输入 0 时，程序运行结果为：

```
请输入一个被除数：0
(<class 'ZeroDivisionError'>, ZeroDivisionError('division by zero',), <traceback
object at 0x000001FCF638DD48>)
其他异常...
```

输出结果中，第 2 行是抛出异常的全部信息，这是一个元组，有 3 个元素，第一个元素是一个 ZeroDivisionError 类；第 2 个元素是异常类型 ZeroDivisionError 类的一个实例；第 3 个元素为一个 traceback 对象。其中，通过前 2 个元素可以看出抛出的异常类型以及描述信息，对于第 3 个元素，是一个 traceback 对象，无法直接看出有关异常的信息，还需要对其做进一步处理。

要查看 traceback 对象包含的内容，需要先引进 traceback 模块，然后调用 traceback

模块中的 print_tb 方法，并将 sys.exc_info() 输出的 traceback 对象作为参数参入。例如：

```
#使用 sys 模块之前，需使用 import 引入
import sys
#引入 traceback 模块
import traceback
try:
    x = int(input("请输入一个被除数："))
    print("30 除以",x,"等于",30/x)
except:
    #print(sys.exc_info())
    traceback.print_tb(sys.exc_info()[2])
    print("其他异常...")
```

输入 0，程序运行结果为：

```
请输入一个被除数：0
  File "C:\Users\mengma\Desktop\demo.py", line 7, in <module>
    print("30 除以",x,"等于",30/x)
其他异常...
```

可以看到，输出信息中包含了更多的异常信息，包括文件名、抛出异常的代码所在的行数、抛出异常的具体代码。

print_tb 方法也仅是 traceback 模块众多方法中的一个，有关 traceback 模块如何获取更多异常信息，后续章节会做详细介绍。

8.Python traceback

除了使用 sys.exc_info() 方法获取更多的异常信息之外，还可以使用 traceback 模块，该模块可以用来查看异常的传播轨迹，追踪异常触发的源头。

下面示例显示了如何显示异常传播轨迹：

```
1.          class SelfException(Exception):
2.              pass
3.          def main():
4.              firstMethod()
5.          def firstMethod():
6.              secondMethod()
7.          def secondMethod():
8.              thirdMethod()
9.          def thirdMethod():
10.             raise SelfException("自定义异常信息")
11.         main()
```

上面程序中 main() 函数调用 firstMethod()，firstMethod() 调用 secondMethod()，secondMethod() 调用 thirdMethod()，thirdMethod() 直接引发一个 SelfException 异常。运行上面程序，将会看到如下所示的结果：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 11, in <module>
    main()
  File "C:\Users\mengma\Desktop\1.py", line 4, in main                    <--mian 函数
    firstMethod()
```

```
  File "C:\Users\mengma\Desktop\1.py", line 6, in firstMethod          <--第三个
    secondMethod()
  File "C:\Users\mengma\Desktop\1.py", line 8, in secondMethod    <--第二个
    thirdMethod()
  File "C:\Users\mengma\Desktop\1.py", line 10, in thirdMethod       <--异常源头
    raise SelfException("自定义异常信息")
SelfException: 自定义异常信息
```

从输出结果可以看出，异常从 thirdMethod() 函数开始触发，传到 secondMethod() 函数，再传到 firstMethod() 函数，最后传到 main() 函数，在 main() 函数止，这个过程就是整个异常的传播轨迹。

在实际应用程序的开发中，大多数复杂操作都会被分解成一系列函数或方法调用。这是因为，为了具有更好的可重用性，会将每个可重用的代码单元定义成函数或方法，将复杂任务逐渐分解为更易管理的小型子任务。由于一个大的业务功能需要由多个函数或方法来共同实现，在最终编程模型中，很多对象将通过一系列函数或方法调用来实现通信，执行任务。

所以，当应用程序运行时，经常会发生一系列函数或方法调用，从而形成“函数调用战”。异常的传播则相反，只要异常没有被完全捕获（包括异常没有被捕获，或者异常被处理后重新引发了新异常），异常就从发生异常的函数或方法逐渐向外传播，首先传给该函数或方法的调用者，该函数或方法的调用者再传给其调用者，直至最后传到 Python 解释器，此时 Python 解释器会中止该程序，并打印异常的传播轨迹信息。

很多初学者一看到输出结果所示的异常提示信息，就会惊慌失措，他们以为程序出现了很多严重的错误，其实只有一个错误，系统提示那么多行信息，只不过是显示异常依次触发的轨迹。

其实，上面程序的运算结果显示的异常传播轨迹信息非常清晰，它记录了应用程序中执行停止的各个点。最后一行信息详细显示了异常的类型和异常的详细消息。从这一行向上，逐个记录了异常发生源头、异常依次传播所经过的轨迹，并标明异常发生在哪个文件、哪一行、哪个函数处。

使用 traceback 模块查看异常传播轨迹，首先需要将 traceback 模块引入，该模块提供了如下两个常用方法：

- traceback.print_exc()：将异常传播轨迹信息输出到控制台或指定文件中。
- format_exc()：将异常传播轨迹信息转换成字符串。

可能有读者好奇，从上面方法看不出它们到底处理哪个异常的传播轨迹信息。实际上我们常用的 print_exc() 是 print_exc([limit[, file]]) 省略了 limit、file 两个参数的形式。而 print_exc([limit[, file]]) 的完整形式是 print_exception(etype, value, tb[,limit[, file]])，在完整形式中，前面三个参数用于分别指定异常的如下信息：

- etype：指定异常类型；
- value：指定异常值；
- tb：指定异常的 traceback 信息；

当程序处于 except 块中时，该 except 块所捕获的异常信息可通过 sys 对象来获取，其中 sys.exc_type、sys.exc_value、sys.exc_traceback 就代表当前 except 块内的异常类型、异常值和异常传播轨迹。

简单来说， print_exc([limit[, file]]) 相当于如下形式：

```
print_exception(sys.exc_etype, sys.exc_value, sys.exc_tb[, limit[, file]])
```

也就是说，使用 print_exc([limit[, file]]) 会自动处理当前 except 块所捕获的异常。该方法还涉及两个参数：

1. limit：用于限制显示异常传播的层数，比如函数 A 调用函数 B，函数 B 发生了异常，如果指定 limit=1，则只显示函数 A 里面发生的异常。如果不设置 limit 参数，则默认全部显示。
2. file：指定将异常传播轨迹信息输出到指定文件中。如果不指定该参数，则默认输出到控制台。

借助于 traceback 模块的帮助，我们可以使用 except 块捕获异常，并在其中打印异常传播信息，包括把它输出到文件中。例如如下程序：

纯文本复制

```
# 导入 trackback 模块
import traceback
class SelfException(Exception): pass
```

```
4.
5.          def main():
6.              firstMethod()
7.          def firstMethod():
8.              secondMethod()
9.          def secondMethod():
10.             thirdMethod()
11.         def thirdMethod():
12.             raise SelfException("自定义异常信息")
13.         try:
14.             main()
15.         except:
16.             # 捕捉异常，并将异常传播信息输出控制台
17.             traceback.print_exc()
18.             # 捕捉异常，并将异常传播信息输出指定文件中
19.             traceback.print_exc(file=open('log.txt', 'a'))
```

上面程序第一行先导入了 traceback 模块，接下来程序使用 except 捕获程序的异常，并使用 traceback 的 print_exc() 方法输出异常传播信息，分别将它输出到控制台和指定文件中。

运行上面程序，同样可以看到在控制台输出异常传播信息，而且在程序目录下生成了一个 log.txt 文件，该文件中同样记录了异常传播信息。

9.Python 自定义异常类

前面的例子里充斥了很多 Python 内置的异常类型，读者也许会问，我可以创建自己的异常类型吗？

答案是肯定的，Python 允许用户自定义异常类型。实际开发中，有时候系统提供的异常类型不能满足开发的需求。这时就可以创建一个新的异常类来拥有自己的异常。

其实，在前面章节中，已经涉及到了异常类的创建，例如：

```
        class SelfExceptionError(Exception):
            pass
        try:
            raise SelfExceptionError()
        except SelfExceptionError as err:
            print("捕捉到自定义异常")
```

运行结果为：

```
捕捉到自定义异常
```

可以看到，此程序中就自定义了一个名为 SelfExceptionError 的异常类，只不过该类是一个空类。

> 由于大多数 Python 内置异常的名字都以 "Error" 结尾，所以实际命名时尽量跟标准的异常命名一样。

需要注意的是，自定义一个异常类，通常应继承自 Exception 类（直接继承），当然也可以继承自那些本身就是从 Exception 继承而来的类（间接继承 Exception）。

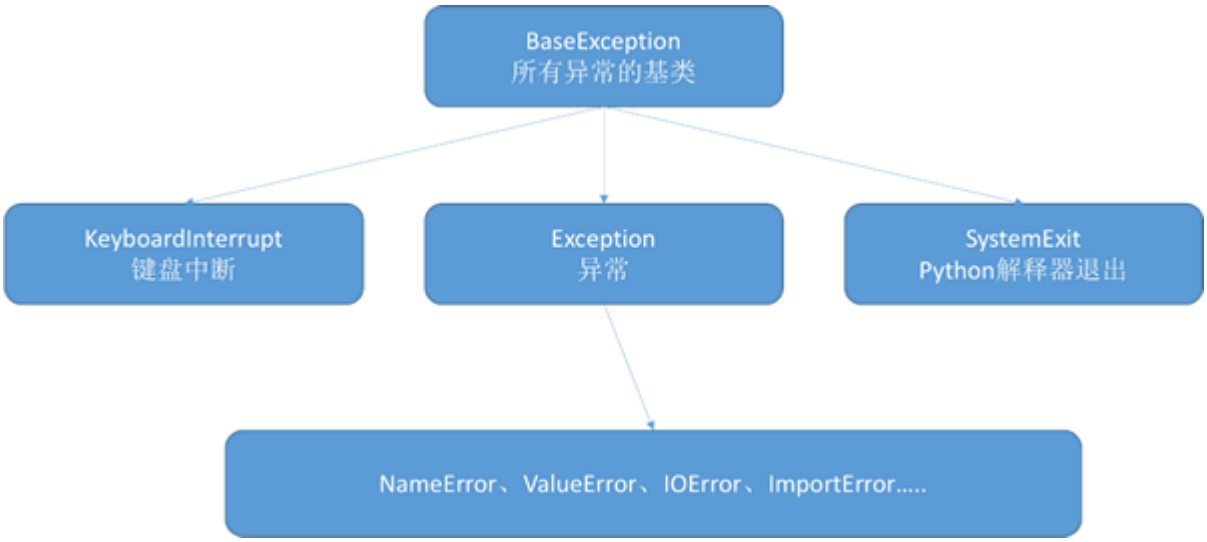


图 1 Python 异常类继承图

注意，虽然所有类同时继承自 BaseException，但它是为系统退出异常而保留的，假如直接继承 BaseException，可能会导致自定义异常不会被捕获，而是直接发送信号退出程序运行，脱离了我们自定义异常类的初衷。

另外，系统自带的异常只要触发会自动抛出（比如 NameError、ValueError 等），但用户自定义的异常需要用户自己决定什么时候抛出。也就是说，自定义的异常需要使用 raise 手动抛出。

下面也是自定义的异常类，和上面的异常类相比，其内部实现了 __init__() 方法和 __str__() 方法：

```
1.          class InputError(Exception):
2.              '''当输出有误时，抛出此异常'''
3.              #自定义异常类型的初始化
4.              def __init__(self, value):
5.                  self.value = value
```

```
6.              # 返回异常类对象的说明信息
7.              def __str__(self):
8.                  return ("{} is invalid input".format(repr(self.value)))
9.
10.         try:
11.             raise InputError(1) # 抛出 MyInputError 这个异常
12.         except InputError as err:
13.             print('error: {}'.format(err))
```

运行结果为：

```
error: 1 is invalid input
```

注意，只要自定义的类继承自 Exception，则该类就是一个异常类，至于此类中包含的内容，并没有做任何规定。

10.正确使用 Python 异常处理机制

前面介绍了使用异常处理的优势、便捷之处，本节将进一步从程序性能优化、结构优化的角度给出异常处理的一般规则。

成功的异常处理应该实现如下 4 个目标：

1. 使程序代码混乱最小化。
2. 捕获并保留诊断信息。
3. 通知合适的人员。
4. 采用合适的方式结束异常活动。

下面介绍达到这些效果的基本准则。

不要过度使用异常

不可否认，Python 的异常机制确实方便，但滥用异常机制也会带来一些负面影响。过度使用异常主要表现在两个方面：

1. 把异常和普通错误混淆在一起，不再编写任何错误处理代码，而是以简单地引发异常来代苦所有的错误处理。
2. 使用异常处理来代替流程控制。

熟悉了异常使用方法后，程序员可能不再愿意编写烦琐的错误处理代码，而是简单地引发异常。实际上这样做是不对的，对于完全已知的错误和普通的错误，应该编写处理这种错误的代码，增加程序的健壮性。只有对于外部的、不能确定和预知的运行时错误才使用异常。

对比前面五子棋游戏中，处理用户输入坐标点已有棋子的两种方式。如果用户试图下棋的坐标点已有棋子：

```
1.          #如果要下棋的点不为空
2.          if board[int(y_str) - 1) [int(x_str) - 1] !="╋" :
3.              inputStr = input ("您输入的坐标点已有棋子了，请重新输入\n")
4.              continue
```

上面这种处理方式检测到用户试图下棋的坐标点已经有棋子，立即打印一条提示语句，并重新开始下一次循环。这种处理方式简洁明了、逻辑清晰，程序的运行效率也很好程序进入 if

块后，即结束了本次循环。

如果将上面的处理机制改为如下方式：

```
        #如果要下棋的点不为空
        if board[int(y_str) - 1) [int(x_str) - 1) != "十":
            #引发默认的 RuntimeError 异常
            raise
```

上面这种处理方式没有提供有效的错误处理代码，当程序检测到用户试图下棋的坐标点已经有棋子时，并没有提供相应的处理，而是简单地引发一个异常。这种处理方式虽然简单，但 Python 解释器接收到这个异常后，还需要进入相应的 except 块来捕获该异常，所以运行效率要差一些。而且用户下棋重复这个错误完全是可预料的，所以程序完全可以针对该错误提供相应的处理，而不是引发异常。

> 必须指出，异常处理机制的初衷是将不可预期异常的处理代码和正常的业务逻辑处理代码分离，因此绝不要使用异常处理来代替正常的业务逻辑判断。

另外，异常机制的效率比正常的流程控制效率差，所以不要使用异常处理来代替正常的程序流程控制。例如，对于如下代码：

```
        #定义一个字符串列表
        my_list =["Hello", "Python", "Spring"]
        #使用异常处理来遍历 arr 数组的每个元素
        try:
            i = 0
            while True:
            print (my_list [i])
            i += 1
```

```
        except:
            pass
```

运行上面程序确实可以实现遍历 my_list 列表的功能，但这种写法可读性较差，而且运行效率也不高。程序完全有能力避免产生 indexError 异常，程序“故意”制造这种异常，然后使用 except 块去捕获该异常，这是不应该的。将程序改为如下形式肯定要好得多：

```
        i = 0
        while i < len(my_list):
            print(my_list[i])
            i += 1
```

注意，异常只应该用于处理非正常的情况，不要使用异常处理来代替正常的流程控制。对于一些完全可预知，而且处理方式清楚的错误，程序应该提供相应的错误处理代码，而不是将其笼统地称为异常。

不要使用过于庞大的 try 块

很多初学异常机制的读者喜欢在 try 块里放置大量的代码，这看上去很“简单”，但这种“简单”只是一种假象，只是在编写程序时看上去比较简单。但因为 try 块里的代码过于庞大，业务过于复杂，就会造成 try 块中出现异常的可能性大大增加，从而导致分析异常原因的难度也大大增加。

而且当时块过于庞大时，就难免在 try 块后紧跟大量的 except 块才可以针对不同的异常提供不同的处理逻辑。在同一个 try 块后紧跟大量的 except 块则需要分析它们之间的逻辑关系，反而增加了编程复杂度。

正确的做法是，把大块的 try 块分割成多个可能出现异常的程序段落，并把它们放在单独的 try 块中，从而分别捕获并处理异常。

不要忽略捕获到的异常

不要忽略异常！既然已捕获到异常，那么 except 块理应做些有用的事情，及处理并修复异常。except 块整个为空，或者仅仅打印简单的异常信息都是不妥的！

except 块为空就是假装不知道甚至瞒天过海，这是最可怕的事情，程序出了错误，所有人都看不到任何异常，但整个应用可能已经彻底坏了。仅在 except 块里打印异常传播信息稍微好一点，但仅仅比空白多了几行异常信息。通常建议对异常采取适当措施，比如：

- 处理异常。对异常进行合适的修复，然后绕过异常发生的地方继续运行；或者用别的数据进行计算，以代替期望的方法返回值；或者提示用户重新操作……总之，程序应该尽量修复异常，使程序能恢复运行。
- 重新引发新异常。把在当前运行环境下能做的事情尽量做完，然后进行异常转译，把异常包装成当前层的异常，重新传给上层调用者。
- 在合适的层处理异常。如果当前层不清楚如何处理异常，就不要在当前层使用 except 语句来捕获该异常，让上层调用者来负责处理该异常。

11.Python IDLE 调试程序

在程序开发过程中，免不了会出现一些错误，既有语法方面的，也有逻辑方面的。语法方面的相对比较好检测，因为当程序中有语法错误时，程序运行会直接停止，同时 Python 解释

器会给出错误提示。而对于逻辑错误，可能并不太容易发现，因为程序本身运行没有问题，只是运行结果是错误的。

当遇到程序有逻辑错误时，最好的解决方法就是对程序进行调试，即通过观察程序的运行过程，以及运行过程中变量（局部变量和全局变量）值的变化，可以快速找到引起运行结果异常的根本原因，从而解决逻辑错误。

掌握一定的程序调试方法，是每一名合适的程序员的必备技能。多数的集成开发工具都提供了程序调试功能，本教程中使用的 IDLE 也不例外。本节将给大家演示如何使用 IDLE 调试 Python 程序。

在保证程序没有语法错误的前提下，使用 IDLE 调试程序的基本步骤如下：

1. 打开 Python Shell，在主菜单上选择“Debug -> Debugger”选项，打开 Debug Control 对话框，同时 Python Shell 窗口中会显示“[DEBUG ON]”，表示已经处于调试状态，如图 1 所示：

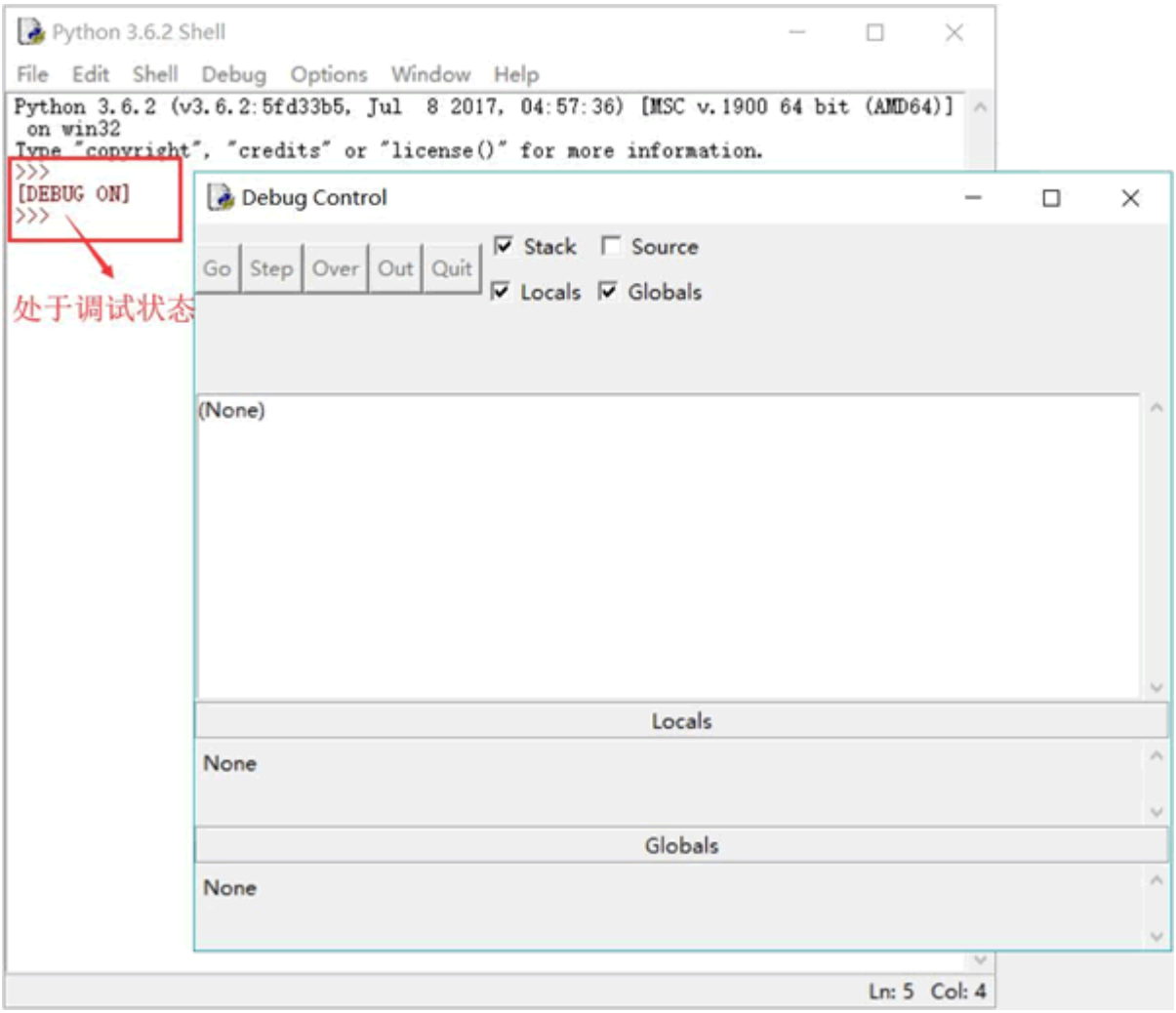


图 1 处于调试状态的 Python Shell

2. 在 Python Shell 窗口中，选择“File -> Open”菜单项，打开要调试的程序文件，并向程序中的代码添加断点，其作用是：当程序执行至断点位置时，会暂时中断执行。根据需要，程序还可以恢复执行。

向程序中添加断点，不能胡乱地添加，要有目的的添加。一般情况下，当想要查看某个变量运行至某处代码的值，就可以在该代码位置添加一个断点。

程序中添加断点的方法是：在想要添加断点的行上，点击鼠标右键，在弹出的快捷菜单中选择“Set BreakPoint”菜单项，添加断点的代码行，其背景会变成黄色，如图 2 所示。

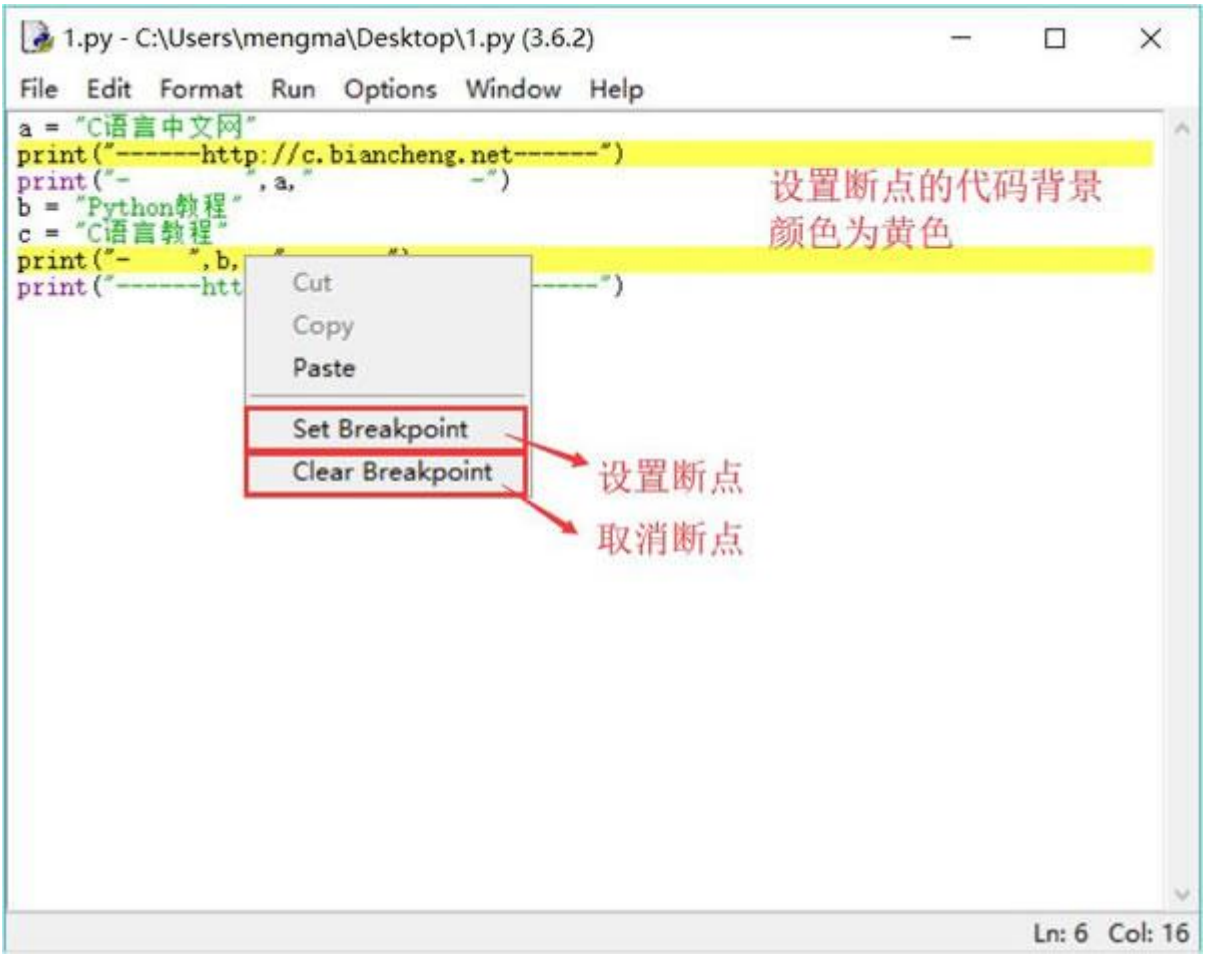


图 2 给代码添加断点

同样 ,如果想删除已添加的断点 ,可以选中已添加断点的行 ,然后点击鼠标右键 ,选择 “Clear Breakpoint” 。

3. 添加完断点之后 ,可以按 F5 快捷键 ,或者在打开的程序文件菜单栏中选择 “Run -> Run Module” 执行程序 , 这时 Debug Control 对话框中将显示程序的执行信息。如图 3 所示。

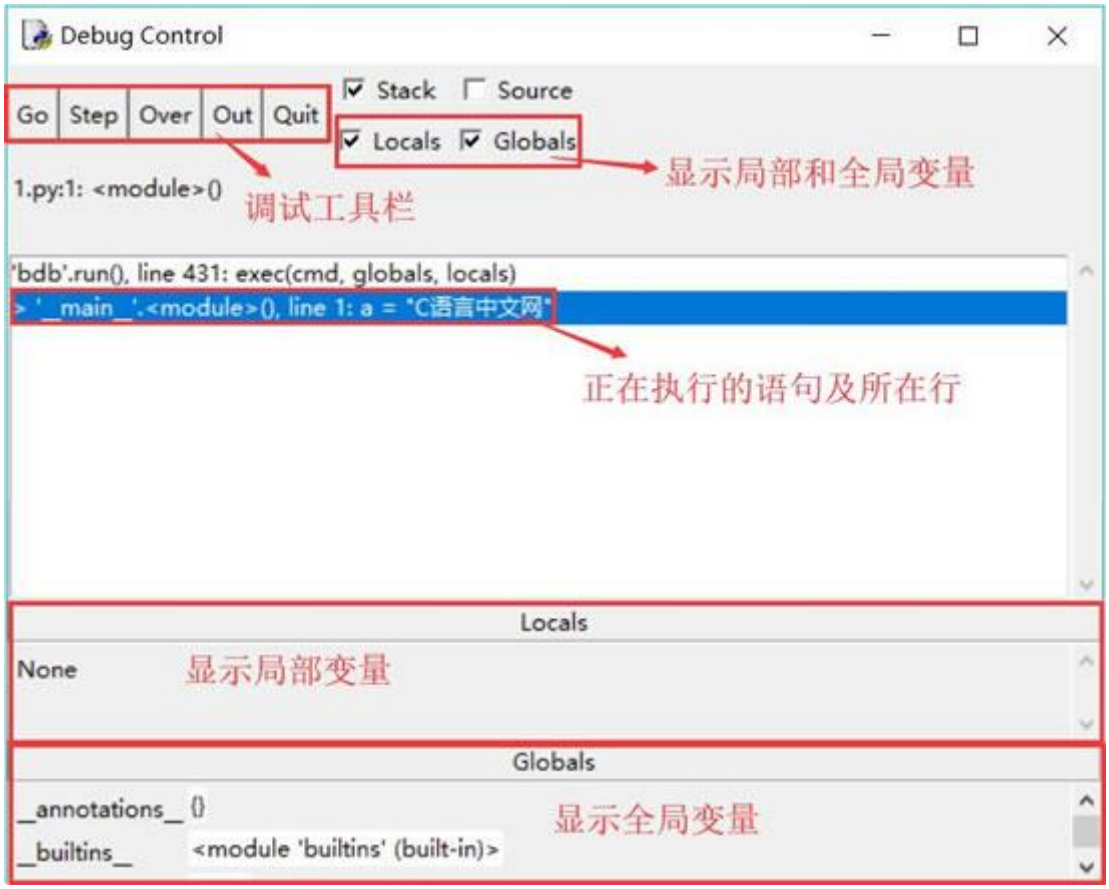


图 3 显示程序的执行信息

需要注意的是，勾选 Globals 复选框，将显示全局变量，Debug Control 默认只显示局部变量。

4. 图 3 中，调试工具栏中的 5 个按钮的作用分别是：

- Go 按钮：直接运行至下一个断点处；
- Step 按钮：用于进入要执行的函数；
- Over 按钮：表示单步执行；
- Out 按钮：表示跳出当前运行的函数；
- Quit 按钮：表示结束调试。

通过使用这 5 个按钮，可以查看程序执行过程中各个变量值的变化，直至程序运行结束。程序调试完毕后，可以关闭 Debug Control 窗口，此时在 Python Shell 窗口中将显示“[DEBUG OFF]”，表示已经结束调试。

12.Python assert 调试程序

前面章节介绍了如何使用 IDLE 自身的调试工具调试程序，除此之外，Python 还提供了 assert 语句，也可以用来调试程序。

《Python assert 断言》一节中，已经对 assert 的基本用法做了简单介绍，assert 语句的完整语法格式为：

```
assert 条件表达式 [,描述信息]
```

assert 语句的作用是：当条件表达式的值为真时，该语句什么也不做，程序正常运行；反之，若条件表达式的值为假，则 assert 会抛出 AssertionError 异常。其中，[,描述信息] 作为可选参数，用于对条件表达式可能产生的异常进行描述。

例如：

```
s_age = input("请输入您的年龄:")
age = int(s_age)
assert 20 < age < 80 , "年龄不在 20-80 之间"
print("您输入的年龄在 20 和 80 之间")
```

程序运行结果为：

```
请输入您的年龄:10
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 3, in <module>
    assert 20 < age < 80 , "年龄不在 20-80 之间"
AssertionError: 年龄不在 20-80 之间
```

通过运行结果可以看出，当 assert 中条件表达式的值为假时，程序将抛出异常，并附带异常的描述性信息，与此同时，程序立即停止执行。

通常情况下，assert 可以和 try except 异常处理语句配合使用，以前面代码为例：

```
try:
    s_age = input("请输入您的年龄:")
    age = int(s_age)
    assert 20 < age < 80 , "年龄不在 20-80 之间"
    print("您输入的年龄在 20 和 80 之间")
except AssertionError as e:
    print("输入年龄不正确",e)
```

程序运行结果为：

```
请输入您的年龄:10
输入年龄不正确 年龄不在 20-80 之间
```

通过在程序的适当位置，使用 assert 语句判断变量或表达式的值，可以起到调试代码的作用。

当在命令行模式运行 Python 程序时，传入 -O（注意是大写）参数，可以禁用程序中包含的 assert 语句。

第 10 章 类特殊成员（属性和方法）

在 Python 类中有些方法名、属性名的前后都添加了双下画线，这种方法、属性通常都属于 Python 的特殊方法和特殊属性，开发者可以通过重写这些方法或直接调用这些方法来实现特殊的功能。

最常见的特殊方法就是前面介绍的构造方法：__init__，开发者可以通过重写类中的 __init__ 方法来实现自己的初始化逻辑。

Python 是一门尽量简单的语言，它不像某些语言（如 Java）需要让类实现接口，并实现接口中的方法。Python 采用的是一种“约定”的机制，Python 按照约定，以特殊名字的方法、属性未提供特殊的功能。

Python 类中的特殊方法、特殊属性有些需要开发者重写，有些则可以直接调用，掌握这些常见的特殊方法、特殊属性也是非常重要的。

1.Python __new__

__new__() 是一种负责创建类实例的静态方法，它无需使用 staticmethod 装饰器修饰，且该方法会优先 __init__() 初始化方法被调用。

一般情况下，覆写 __new__() 的实现将会使用合适的参数调用其超类的

super().__new__()，并在返回之前修改实例。例如：

```
class demoClass:
    instances_created = 0
    def __new__(cls,*args,**kwargs):
        print("__new__():",cls,args,kwargs)
        instance = super().__new__(cls)
        instance.number = cls.instances_created
        cls.instances_created += 1
        return instance
    def __init__(self,attribute):
        print("__init__():",self,attribute)
        self.attribute = attribute
test1 = demoClass("abc")
test2 = demoClass("xyz")
print(test1.number,test1.instances_created)
print(test2.number,test2.instances_created)
```

输出结果为：

```
__new__(): <class '__main__.demoClass'> ('abc',) {}
__init__(): <__main__.demoClass object at 0x0000026FC0DF8080> abc
__new__(): <class '__main__.demoClass'> ('xyz',) {}
__init__(): <__main__.demoClass object at 0x0000026FC0DED358> xyz
0 2
1 2
```

__new__() 通常会返回该类的一个实例，但有时也可能会返回其他类的实例，如果发生了这种情况，则会跳过对 __init__() 方法的调用。而在某些情况下（比如需要修改不可变类实例（Python 的某些内置类型）的创建行为），利用这一点会事半功倍。比如：

```
class nonZero(int):
    def __new__(cls,value):
        return super().__new__(cls,value) if value != 0 else None
    def __init__(self,skipped_value):
        #此例中会跳过此方法
        print("__init__()")
        super().__init__()
print(type(nonZero(-12)))
print(type(nonZero(0)))
```

运行结果为：

```
__init__()
<class '__main__.nonZero'>
<class 'NoneType'>
```

那么，什么情况下使用 __new__() 呢？答案很简单，在 __init__() 不够用的时候。

例如，前面例子中对 Python 不可变的内置类型（如 int、str、float 等）进行了子类化，这是因为一旦创建了这样不可变的对象实例，就无法在 __init__() 方法中对其进行修改。

有些读者可能会认为，__new__() 对执行重要的对象初始化很有用，如果用户忘记使用 super()，可能会漏掉这一初始化。虽然这听上去很合理，但有一个主要的缺点，即如果使用这样的方法，那么即便初始化过程已经是预期的行为，程序员明确跳过初始化步骤也会变得更加困难。不仅如此，它还破坏了“__init__() 中执行所有初始化工作”的潜规则。

注意，由于 __new__() 不限于返回同一个类的实例，所以很容易被滥用，不负责任地使用这种方法可能会对代码有害，所以要谨慎使用。一般来说，对于特定问题，最好搜索其他可用的解决方案，最好不要影响对象的创建过程，使其违背程序员的预期。比如说，前面提到

的覆写不可变类型初始化的例子，完全可以用工厂方法（一种设计模式）来替代。

Python 中大量使用 __new__() 方法且合理的，就是 MetaClass 元类。有关元类的介绍，可阅读《Python MetaClass 元类》一节。

2.Python _repr_

先看下面程序：

```
class Item:
    def __init__ (self, name, price):
        self.name = name
        self.price = price
# 创建一个 Item 对象，将之赋给 im 变量
im = Item('鼠标', 29.8)
# 打印 im 所引用的 Item 对象
print(im)
```

上面程序创建了一个 Item 对象，然后使用 print() 方法输出 Item 对象。编译、运行上面程序，将看到如下输出结果：

```
<__main__.Item object at 0x000001DB45D406D8>
```

当读者运行上面程序时，可能会看到不同的输出结果，at 后的 16 位十六进制数字可能发生改变。但这个输出结果是怎么来的呢？按道理来说，print() 函数只能在控制台打印字符串，而 Item 实例是内存中的一个对象，怎么能直接转换为字符串输出呢？

事实上，当使用该方法输出 Item 对象时，实际上输出的是 Item 对象的 __repr__() 方法的返回值。也就是说，下面两行代码的效果完全一样：

```
print (im)
print (im.__repr__)
```

__repr__() 是 Python 类中的一个特殊方法，由于 object 类已提供了该方法，而所有的 Python 类都是 object 类的子类，因此所有的 Python 对象都具有 __repr__() 方法。

因此，当程序需要将任何对象与字符串进行连接时，都可先调用 __repr__() 方法将对象转换成字符串，然后将两个字符串连接在一起。例如如下代码：

```
im str = im.__repr() + ""
```

__repr__() 是一个非常特殊的方法，它是一个“自我描述”的方法，该方法通常用于实现这样一个功能：当程序员直接打印该对象时，系统将会输出该对象的“自我描述”信息，用来告诉外界该对象具有的状态信息。

object 类提供的 __repr__() 方法总是返回该对象实现类的“类名+object at+内存地址”值，这个返回值并不能真正实现“自我描述”的功能，因此，如果用户需要自定义类能实现“自我描述”的功能，就必须重写 __repr__() 方法。例如下面程序：

纯文本复制

```
class Apple:
    # 实现构造器
    def __init__(self, color, weight):
        self.color = color;
        self.weight = weight;
    # 重写__repr__方法，用于实现 Apple 对象的“自我描述”
    def __repr__(self):
        return "Apple[color=" + self.color +\
            ", weight=" + str(self.weight) + "]"
a = Apple("红色" , 5.68)
# 打印 Apple 对象
print(a)
```

编译、运行上面程序，可以看到如下运行结果：

```
Apple[color=红色, weight=5.68]
```

从上面的运行结果可以看出，通过重写 Apple 类的 __repr__() 方法，就可以让系统在打印 Apple 对象时打印出该对象的“自我描述”信息。

大部分时候，重写 __repr__() 方法总是返回该对象的所有令人感兴趣的信息所组成的字符串。通常可返回如下格式的字符串：

```
类名[field1 = 值 1, field2 =值 2, ...]
```

3.Python __del__

与 __init__() 方法对应的是 __del__() 方法，__init__() 方法用于初始化 Python 对象，而 __del__() 则用于销毁 Python 对象，即在任何 Python 对象将要被系统回收之时，系统都会自动调用该对象的 __del__() 方法。

当程序不再需要一个 Python 对象时，系统必须把该对象所占用的内存空间释放出来，这个过程被称为垃圾回收（GC，Garbage Collector），Python 会自动回收所有对象所占用的内存空间，因此开发者无须关心对象垃圾回收的过程。

> Python 采用自动引用计数（ARC）方式来回收对象所占用的空间，当程序中有一个变量引用该 Python 对象时，Python 会自动保证该对象引用计数为 1；当程序中有两个变量引用该 Python 对象时，Python 会自动保证该对象引用计数为 2，依此类推，如果一个对象的引用计数变成了 0，则说明程序中不再有变量引用该对象，表明程序不再需要该对象，因此 Python 就会回收该对象。

大部分时候，Python 的 ARC 都能准确、高效地回收系统中的每个对象。但如果系统中出现循环引用的情况，比如对象 a 持有一个实例变量引用对象 b，而对象 b 又持有一个实例变量引用对象 a，此时两个对象的引用计数都是 1，而实际上程序已经不再有变量引用它们，系统应该回收它们，此时 Python 的垃圾回收器就可能没那么快，要等专门的循环垃圾回收器（Cyclic Garbage Collector）来检测并回收这种引用循环。

当一个对象被垃圾回收时，Python 就会自动调用该对象的 __del__ 方法。需要说明的是，不要以为对一个变量执行 del 操作，该变量所引用的对象就会被回收，只有当对象的引用计数变成 0 时，该对象才会被回收。因此，如果一个对象有多个变量引用它，那么 del 其中一个变量是不会回收该对象的。

```
class Item:
    def __init__ (self, name, price):
        self.name = name
        self.price = price
    # 定义析构函数
    def __del__ (self):
        print('del 删除对象')
# 创建一个 Item 对象，将之赋给 im 变量
im = Item('鼠标', 29.8)
```

```
        x = im    # ①
        # 打印 im 所引用的 Item 对象
        del im
        print('--------------')
```

程序中重写了 Item 类的 __del__() 方法，该方法就是 Item 类的析构函数，当系统将要回收 Item 时，系统会自动调用 Item 对象的 __del__() 方法。

上面程序先创建了一个 Item 对象，并将该对象赋值给 im 变量，① 号代码又将 im 赋值给变量 x，这样程序中有两个变量引用 Item 对象，接下来程序执行 del im 代码删除 im 对象，此时由于还有变量引用该 Item 对象，因此程序并不会回收 Item 对象。

运行上面程序，可以看到如下输出结果：

```
--------------
del 删除对象
```

从上面程序的输出结果可以看到，del im 执行之后，程序并没有回收 Item 对象，只有等到程序执行将要结束时（系统必须回收所有对象），系统才会回收 Item 对象。

如果将程序中 ① 号代码注释掉，再次运行上面程序，将会看到如下输出结果：

```
del 删除对象
--------------
```

注释掉 ① 号代码之后，当程序执行 del im 之后，此时程序中不再有任何变量引用该 Item 对象，因此系统会立即回收该对象，则无须等到程序结束之前。

最后需要说明的是，如果父类提供了 __del__() 方法，则系统重写 __del__() 方法时必须显式调用父类的 __del__() 方法，这样才能保证合理地回收父类实例的部分属性。

4.Python __dir__

对象的 __dir__ 方法用于列出该对象内部的所有属性（包括方法）名，该方法将会返回包含所有属性（方法）名的序列。

当程序对某个对象执行 dir(object) 函数时，实际上就是将该对象的 __dir__() 方法返回值

进行排序，然后包装成列表。

例如，如下程序示范了 __dir__() 方法的功能：

```
class Item:
    def __init__ (self, name, price):
        self.name = name
        self.price = price
    def info ():
        pass
# 创建一个 Item 对象，将之赋给 im 变量
im = Item('鼠标', 29.8)
print(im.__dir__())  # 返回所有属性（包括方法）组成列表
print(dir(im))  # 返回所有属性（包括方法）排序之后的列表
```

运行上面程序，可以看到程序不仅会输出我们为对象定义的 name、 price、 info 三个属性和方法，而且还有大量系统内置的属性和方法，如前面章节介绍的 __repr__ 和 __del__ 方法。

5.Python __dict__

__dict__ 属性用于查看对象内部存储的所有属性名和属性值组成的字典，通常程序直接使用该属性即可。

程序使用 __dict__ 属性既可查看对象的所有内部状态，也可通过字典语法来访问或修改指定属性的值。例如如下程序：

```
1.          class Item:
2.              def __init__ (self, name, price):
3.                  self.name = name
4.                  self.price = price
5.          im = Item('鼠标', 28.9)
```

```
print(im.__dict__)  # ①
# 通过__dict__访问 name 属性
print(im.__dict__['name'])
# 通过__dict__访问 price 属性
print(im.__dict__['price'])
im.__dict__['name'] = '键盘'
im.__dict__['price'] = 32.8
print(im.name) # 键盘
print(im.price) # 32.8
```

上面程序中 ① 号代码直接输出对象的 dict 属性，这样将会直接输出该对象内部存储的所有属性名和属性值组成的 dict 对象；接下来的两行代码通过 dict 属性访问对象的 name、price 两个属性；再后边两行代码通过 __dict__ 属性对 name、 price 两个属性赋值。

运行上面程序，可以看到如下输出结果：

```
{'name': '鼠标', 'price': 28.9}
鼠标
28.9
键盘
32.8
```

6.Python setattr、getattr、hasattr

在动态检查对象是否包含某些属性（包括方法）相关的函数有如下几个：

hasattr(obj, name)：检查 obj 对象是否包含名为 name 的属性或方法。
getattr(object, name[, default])：获取 object 对象中名为 name 的属性的属性值。
setattr(obj, name, value，/)：将 obj 对象的 name 属性设为 value。

下面程序示范了通过以上函数来动态操作 Python 对象的属性：

```
class Comment:
```

```
2.              def __init__ (self, detail, view_times):
3.                  self.detail = detail
4.                  self.view_times = view_times
5.              def info ():
6.                  print("一条简单的评论，内容是%s" % self.detail)
7.
8.          c = Comment('疯狂 Python 讲义很不错', 20)
9.          # 判断是否包含指定的属性或方法
10.         print(hasattr(c, 'detail')) # True
11.         print(hasattr(c, 'view_times')) # True
12.         print(hasattr(c, 'info')) # True
13.         # 获取指定属性的属性值
14.         print(getattr(c, 'detail')) # '疯狂 Python 讲义很不错'
15.         print(getattr(c, 'view_times')) # 20
16.         # 由于 info 是方法，故下面代码会提示：name 'info' is not defined
17.         #print(getattr(c, info, '默认值'))
18.         # 为指定属性设置属性值
19.         setattr(c, 'detail', '天气不错')
20.         setattr(c, 'view_times', 32)
21.         # 输出重新设置后的属性值
22.         print(c.detail)
23.         print(c.view_times)
```

上面程序先定义了一个 Comment 类，接下来程序创建了 Comment 类的实例。程序后面部分示范了 hasattr() 、getattr() 、setatttr() 三个函数的用法。

从上面的示例代码可以看出，hasattr() 函数既可判断属性，也可判断方法，但 getattr() 则只能获取属性的属性值。

运行上面代码，可以看到如下输出结果：

```
True

True

True

疯狂 Python 讲义很不错
```

```
20
天气不错
32
```

从上面最后两行输出来看，程序使用 setattr() 函数可改变 Python 对象的属性值；如果使用该函数对 Python 对象设置的属性不存在，那么就表现为添加属性反正 Python 是动态语言。看如下代码：

```
#设置不存在的属性，即为对象添加属性
setattr(c, 'test', '新增的测试属性')
print(c.test) # 新增的测试属性
```

实际上 setattr() 函数还可对方法进行设置，在使用 setattr() 函数重新设置对象的方法时，新设置的方法是未绑定方法。例如如下代码：

```
def bar ():
    print('一个简单的 bar 方法')
# 将 c 的 info 方法设为 bar 函数
setattr(c, 'info', bar)
c.info()
```

上面程序先定义了一个 bar() 函数，在该函数中不能定义 self 参数（否则需要程序员显式为参数传入参数值，系统不会自动为该参数绑定参数值）。接下来程序调用 setattr() 函数将 Comment 对象的 info() 方法设置为 bar() 函数，然后程序调用 Comment 对象的 info() 方法，其实就是调用程序中的 bar() 函数。

运行上面代码，可以看到运行 c.info() 方法时实际执行的是前面定义的 bar() 函数。

不仅如此，程序完全可通过 setattr() 函数将 info() 方法设置成普通值，这样将会把 info 变成一个属性，而不是方法，例如如下代码：

```
# 将 c 的 info 设置为字符串'fkit'
setattr(c, 'info', 'fkit')
c.info()
```

上面第一行代码将 c 的 info（原本是方法）设置为字符串 'fkit'，这样 info 就变成了属性，

而不是方法。运行上面第二行代码，将会看到如下错误提示：

```
TypeError：'str' object is not callable
```

7.Python issubclass 和 isinstance

Python 提供了如下两个函数来检查类型：

issubclass(cls, class_or_tuple) :检查 cls 是否为后一个类或元组包含的多个类中任意类的子类。

isinstance(obj, class_or_tuple)：检查 obj 是否为后一个类或元组包含的多个类中任意类的对象。

通过使用上面两个函数，程序可以方便地先执行检查，然后才调用方法，这样可以保证程序不会出现意外情况。

如下程序示范了通过这两个函数来检查类型：

```
1.          # 定义一个字符串
2.          hello = "Hello";
3.          # "Hello"是 str 类的实例，输出 True
4.          print('"Hello"是否是 str 类的实例：', isinstance(hello, str))
5.          # "Hello"是 object 类的子类的实例，输出 True
6.          print('"Hello"是否是 object 类的实例：', isinstance(hello, object))
7.          # str 是 object 类的子类，输出 True
8.          print('str 是否是 object 类的子类：', issubclass(str, object))
9.          # "Hello"不是 tuple 类及其子类的实例，输出 False
10.         print('"Hello"是否是 tuple 类的实例：', isinstance(hello, tuple))
```

```
        # str 不是 tuple 类的子类，输出 False
        print('str 是否是 tuple 类的子类：', issubclass(str, tuple))
        # 定义一个列表
        my_list = [2, 4]
        # [2, 4]是 list 类的实例，输出 True
        print('[2, 4]是否是 list 类的实例：', isinstance(my_list, list))
        # [2, 4]是 object 类的子类的实例，输出 True
        print('[2, 4]是否是 object 类及其子类的实例：', isinstance(my_list,
object))
        # list 是 object 类的子类，输出 True
        print('list 是否是 object 类的子类：', issubclass(list, object))
        # [2, 4]不是 tuple 类及其子类的实例，输出 False
        print('[2, 4]是否是 tuple 类及其子类的实例：', isinstance([2, 4], tuple))
        # list 不是 tuple 类的子类，输出 False
        print('list 是否是 tuple 类的子类：', issubclass(list, tuple))
```

通过上面程序可以看出，issubclass() 和 isinstance() 两个函数的用法差不多，区别只是 issubclass() 的第一个参数是类名，而 isinstance() 的第一个参数是变量，这也与两个函数的意义对应：issubclass 用于判断是否为子类，而 isinstance() 用于判断是否为该类或子类的实例。

issubclass() 和 isinstance() 两个函数的第二个参数都可使用元组。例如如下代码：

```
        data = (20, 'fkit')
        print('data 是否为列表或元组：', isinstance(data, (list, tuple))) # True
        # str 不是 list 或者 tuple 的子类，输出 False
        print('str 是否为 list 或 tuple 的子类：', issubclass(str, (list, tuple)))
        # str 是 list 或 tuple 或 object 的子类，输出 True
        print('str 是否为 list 或 tuple 或 object 的子类 ', issubclass(str, (list,
tuple, object)))
```

此外，Python 为所有类都提供了一个 __bases__ 属性，通过该属性可以查看该类的所有直接父类，该属性返回所有直接父类组成的元组。例如如下代码：

```
        class A:
            pass
        class B:
            pass
```

```
5.          class C(A, B):
6.              pass
7.          print('类 A 的所有父类:', A.__bases__)
8.          print('类 B 的所有父类:', B.__bases__)
9.          print('类 C 的所有父类:', C.__bases__)
```

运行上面程序，可以看到如下运行结果：

```
类 A 的所有父类: (<class 'object'>,)

类 B 的所有父类: (<class 'object'>,)

类 C 的所有父类: (<class '__main__.A'>, <class '__main__.B'>)
```

从上面的运行结果可以看出，如果在定义类时没有显式指定它的父类，则这些类默认的父类是 object 类。

Python 还为所有类都提供了一个 __subclasses__() 方法，通过该方法可以查看该类的所有直接子类，该方法返回该类的所有子类组成的列表。例如在上面程序中增加如下两行：

```
1.          print('类 A 的所有子类:', A.__subclasses__())
2.          print('类 B 的所有子类:', B.__subclasses__())
```

运行上面代码，可以看到如下输出结果：

```
类 A 的所有子类: [<class '__main__.C'>]

类 B 的所有子类: [<class '__main__.C'>]
```

8.Python __call__

前面章节中，我们用 hasattr() 函数判断指定属性（或方法）是否存在，但到底是属性还是方法，则需要进一步判断它是否可调用。程序可通过判断该属性（或方法）是否包含 __call__ 属性来确定它是否可调用。

例如如下程序：

```
class User:
    def __init__(self, name, passwd):
        self.name = name
        self.passwd = passwd
    def validLogin (self):
        print('验证%s 的登录' % self.name)
u = User('crazyit', 'leegang')
# 判断 u.name 是否包含__call__方法，即判断是否可调用
print(hasattr(u.name, '__call__')) # False
# 判断 u.passwd 是否包含__call__方法，即判断是否可调用
print(hasattr(u.passwd, '__call__')) # False
# 判断 u.validLogin 是否包含__call__方法，即判断是否可调用
print(hasattr(u.validLogin, '__call__')) # True
```

上面程序中粗体字代码分别判断 User 对象的 name、 passwd、 validLogin 是否包含 __call__ 方法，如果包含该方法，则表明它是可调用的，否则就说明它是不可调用的。

从上面程序的输出结果不难看到，对于 name、 passwd 两个属性，由于它们都是不可调用的，因此程序在判断它们是否包含 __call__ 方法时输出 False；对于 validLogin 方法，由于它是可调用的，因此程序在判断它是否包含 __call__ 方法时输出 True。

实际上，一个函数（甚至对象）之所以能执行，关键就在于 __call__() 方法。实际上 x(arg1, arg2,...) 只是 x.__call__(arg1, arg2, ...) 的快捷写法，因此我们甚至可以为自定义类添加 __call__ 方法，从而使得该类的实例也变成可调用的。例如如下代码：

```
# 定义 Role 类
class Role:
    def __init__ (self, name):
        self.name = name
    # 定义__call__方法
    def __call__(self):
        print('执行 Role 对象')
r = Role('管理员')
# 直接调用 Role 对象，就是调用该对象的__call__方法
```

```
10.         r()
```

上面程序中最后一行代码使用调用函数的语法来调用对象，这看上去似乎是错误的，但由于该 Role 类提供了 __call__ 方法，因此调用对象的本质就是执行该对象的 __call__ 方法。运行上面代码，将看到如下输出结果：

```
执行 Role 对象
```

对于程序中的函数，同样既可使用函数的语法来调用它，也可把函数当成对象，调用它的 __call__ 方法。例如如下示例代码：

```
1.          def foo ():
2.              print('--foo 函数--')
3.          # 下面示范了 2 种方式调用 foo()函数
4.          foo()
5.          foo.__call__()
```

运行上面代码，可以看到 foo() 和 foo.__call__() 的效果完全相同。

9.Python 序列相关操作

序列最重要的特征就是可包含多个元素， 因此和序列有关的特殊方法有如下几个：

__len__(self)：该方法的返回值决定序列中元素的个数。
__getitem__(self, key)：该方法获取指定索引对应的元素。该方法的 key 应该是整数值或 slice 对象，否则该方法会引发 KeyError 异常。
__contains__(self, item)：该方法判断序列是否包含指定元素。
__setitem__(self, key, value)：该方法设置指定索引对应的元素。该方法的 key 应该是整数值或 slice 对象，否则该方法会引发 KeyError 异常。
__delitem__(self, key)：该方法删除指定索引对应的元素。

如果程序要实现不可变序列（程序只能获取序列中的元素，不能修改），只要实现上面前 3 个方法就行；如果程序要实现可变序列（程序既能获取序列中的元素，也可修改），则需要实现上面

5 个方法。

下面程序将会实现一个字符串序列，在该字符串序列中默认每个字符串的长度都是 3，该序列的元素按 AAA、 AAB、 AAC..... 这种格式排列：

```
        def check_key (key):
            '''
            该函数将会负责检查序列的索引，该索引必须是整数值，否则引发 TypeError
            且程序要求索引必须为非负整数，否则引发 IndexError
            '''
            if not isinstance(key, int): raise TypeError('索引值必须是整数')
            if key < 0: raise IndexError('索引值必须是非负整数')
            if key >= 26 ** 3: raise IndexError('索引值不能超过%d' % 26 ** 3)
        class StringSeq:
            def __init__(self):
                # 用于存储被修改的数据
                self.__changed = {}
                # 用于存储已删除元素的索引
                self.__deleted = []
            def __len__(self):
                return 26 ** 3
            def __getitem__(self, key):
                '''
                根据索引获取序列中元素
                '''
                check_key(key)
                # 如果在 self.__changed 中找到已经修改后的数据
                if key in self.__changed :
                    return self.__changed[key]
                # 如果 key 在 self.__deleted 中，说明该元素已被删除
                if key in self.__deleted :
                    return None
                # 否则根据计算规则返回序列元素
                three = key // (26 * 26)
                two = ( key - three * 26 * 26) // 26
```

```
                one = key % 26
                return chr(65 + three) + chr(65 + two) + chr(65 + one)
            def __setitem__(self, key, value):
                '''
                根据索引修改序列中元素
                '''
                check_key(key)
                # 将修改的元素以 key-value 对的形式保存在__changed 中
                self.__changed[key] = value
            def __delitem__(self, key):
                '''
                根据索引删除序列中元素
                '''
                check_key(key)
                # 如果__deleted 列表中没有包含被删除 key，添加被删除的 key
                if key not in self.__deleted : self.__deleted.append(key)
                # 如果__changed 中包含被删除 key，删除它
                if key in self.__changed : del self.__changed[key]
        # 创建序列
        sq = StringSeq()
        # 获取序列的长度，实际上就是返回__len__()方法的返回值
        print(len(sq))
        print(sq[26*26])
        # 打印没修改之后的 sq[1]
        print(sq[1]) # 'AAB'
        # 修改 sq[1]元素
        sq[1] = 'fkit'
        # 打印修改之后的 sq[1]
        print(sq[1]) # 'fkit'
        # 删除 sq[1]
        del sq[1]
        print(sq[1]) # None
        # 再次对 sq[1]赋值
        sq[1] = 'crazyit'
        print(sq[1]) # crazyit
```

上面程序实现了一个 StringSeq 类，并为该类实现了一个 __len__()、 __getitem__()、__setitem__() 和 __delitem__() 方法，其中 __len__() 方法返回该序列包含的元素个数，__getitem__() 方法根据索引返回元素， __setitem__() 方法根据索引修改元素的值，而__delitem__() 方法则用于根据索引删除元素。

该序列本身并不保存序列元素，序列会根据索引动态计算序列元素，因此该序列需要保存被修改、被删除的元素。该序列使用 __changed 实例变量保存被修改的元素，使用 __deleted 实例变量（列表）保存被删除的索引。

在定义了字符串序列之后， 接下来程序创建了序列对象，并调用序列方法测试该工具类。运行该程序，可以看到如下输出结果：

```
17576

BAA

AAB

fkit

None

crazyit
```

从上面的输出结果来看，程序中序列的第二个元素 sq[1] 恰好为 'AAB'，程序既可对序列元素赋值，也可删除、修改序列元素，这完全是一个功能完备的序列。

10.Python 迭代器

前面介绍了使用 for 循环遍历列表、元组和字典等，这些对象都是可迭代的，因此它们都属于迭代器。迭代器其实就是一个实现了迭代器协议的容器类对象。

简单的理解容器，就是用来存放各种元素的，是各种元素的集合。常用的容器包括元组、列表、字典、集合等。

迭代器基于以下两个方法实现：

1. __next__(self)：返回容器的下一个元素。

2. __iter__(self)：该方法返回一个迭代器（iterator）。

迭代器可以利用 Python 内置的 iter() 函数和一个序列来创建。例如：

```
# 将列表转换为迭代器
my_iter = iter([2, 'fkit', 4])
# 依次获取迭代器的下一个元素
print(my_iter.__next__())
print(my_iter.__next__())
print(my_iter.__next__())
print(my_iter.__next__())
```

运行结果为：

```
2
fkit
4
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 7, in <module>
    print(my_iter.__next__())
StopIteration
```

可以看到，当遍历完序列之后如果继续遍历，会引发 StopIteration 异常，这样迭代器就可以与循环兼容，因为可以捕获这个异常并停止循环。例如：

```
for i in iter([2, 'fkit', 4]):
  print(i,end=" ")
```

运行结果为：

```
2 fkit 4
```

Python 也允许我们自定义一个迭代器，只需要实现 __next__() 和 __iter__() 特殊方法即可。例如：

```
class countDown:
  def __init__(self,step):
    self.step = step
  def __next__(self):
    if self.step <= 0:
      raise StopIteration
    self.step -= 1
    return self.step
  def __iter__(self):
    return self
for element in countDown(10):
  print(element,end=" ")
```

输出结果为：

```
9 8 7 6 5 4 3 2 1 0
```

迭代器本身是一个底层的特性和概念，在程序中并不常用，但它为生成器这一更有趣的特性提供了基础。

有关生成器的相关知识，会在后续章节中介绍。

11.Python 生成器

生成器和迭代器的功能非常相似，它也会提供 __next__() 方法，这意味着程序同样可调用内置的 next() 函数来获取生成器的下一个值，也可使用 for 循环来遍历生成器。

生成器与迭代器的区别在于，迭代器通常是先定义一个迭代器类，然后通过创建实例来创建迭代器；而生成器则是先定义一个包含 yield 语句的函数，然后通过调用该函数来创建生成器。

生成器是一种非常优秀的语法，Python 使用生成器可以让程序变得很优雅。

创建生成器

创建生成器需要两步操作：

1. 定义一个包含 yield 语句的函数。
2. 调用第 1 步创建的函数得到生成器。

下面程序使用生成器来定义一个差值递增的数列。程序先定义了一个包含 yield 语句的函数：

```
1.          def test(val, step):
2.              print("--------函数开始执行------")
3.              cur = 0
4.              # 遍历 0～val
5.              for i in range(val):
```

```
6.                  # cur 添加 i*step
7.                  cur += i * step
8.                  yield cur
```

上面函数与前面介绍的普通函数的最大区别在于 yield cur 这行，如果将这行代码改为 print(cur)，那么这个函数就显得比较普通了，该函数只是简单地遍历区间，并将循环计数器乘以 step 后添加到 cur 变量上，该数列中两个值之间的差值会逐步递增。

如果将上面的 yield cur 语句改为 print(cur, end ='')，执行 test(10, 2) 函数将会看到如下输出结果：

```
--------函数开始执行------
0 2 6 12 20 30 42 56 72 90
```

yield cur 语句的作用有两点：

1. 每次返回一个值，有点类似于 return 语句。
2. 冻结执行，程序每次执行到 yield 语句时就会被暂停。

在程序被 yield 语句冻结之后，当程序再次调用 next() 函数获取生成器的下一个值时，程序才会继续向下执行。

需要指出的是，调用包含 yield 语句的函数并不会立即执行，它只是返回一个生成器。只有当程序通过 next() 函数调用生成器或遍历生成器时，函数才会真正执行。

保留上面函数中的 yield cur 语句，执行如下语句：

```
# 执行函数，返回生成器
t = test(10, 2)
print('================')
# 获取生成器的第一个值
print(next(t)) # 0，生成器“冻结”在 yield 处
print(next(t)) # 2，生成器再次“冻结”在 yield 处
```

运行上面代码，可以看到如下输出结果：

```
==================

--------函数开始执行------

0

2
```

从上面的输出结果不难看出，当程序执行 `t = test(10, 2)` 调用函数时，程序并未开始执行 test() 函数；当程序第一次调用 next(t) 时，test() 函数才开始执行。

> Python 2.x 不使用 next() 函数来获取生成器的下一个值，而是直接调用生成器的 next() 方法。也就是说，在 Python 2.x 中应该写成 t.next()。

当程序调用 next(t) 时，生成器会返回 yield cur 语句返回的值（第一次返回 0），程序被“冻结”在 yield 语句处，因此可以看到上面生成器第一次输出的值为 0。

当程序第二次调用 next(t) 时，程序的“冻结”被解除，继续向下执行，这一次循环计数器 i 变成 1，在执行 `cur += i * step` 之后，cur 变成 2 ，生成器再次返回 yield cur 语

句返回的值（这一次返回 2），程序再次被“冻结”在该 yield 语句处，因此可以看到上面生成器第二次输出的值为 2。

程序可使用 for 循环来遍历生成器，相当于不断地使用 next() 函数获取生成器的下一个值。例如如下代码：

```
        for ele in t:
        print(ele, end=' ')
```

运行上面循环代码，会生成如下输出结果：

```
6 12 20 30 42 56 72 90
```

由于前面两次调用 next() 函数已经获取了生成器的前两个值，因此此处循环时第一次输出的值就是 6。

此外，程序可使用 list() 函数将生成器能生成的所有值转换成列表，也可使用 tuple() 函数将生成器能生成的所有值转换成元组。例如如下代码：

```
        #再次创建生成器
        t = test(10, 1)
        #将生成器转换成列表
        print (list (t))
        #再次创建生成器
        t = test(10, 3)
        #将生成器转换成元组
        print(tuple(t))
```

运行上面代码，可以看到如下输出结果：

```
--------函数开始执行------
[0 , 1 , 3, 6 , 10 , 15 , 21 , 28 , 36 , 45]
--------函数开始执行------
(0 , 3 , 9 , 18 , 30 , 45 , 63 , 84 , 108, 135)
```

系统学习本教程的读者应该知道，前面章节还介绍过使用 for 循环来创建生成器（将 for 表达式放在圆括号里）。可见，Python 主要提供了以下两种方式来创建生成器：

1. 使用 for 循环的生成器推导式。
2. 调用带 yield 语句的生成器函数。

生成器是 Python 的一个特色功能，在其他语言中往往没有对应的机制，因此很多 Python 开发者对生成器机制不甚了解。但实际上生成器是一种非常优秀的机制，以我们实际开发的经验来看，使用生成器至少有以下几个优势：

- 当使用生成器来生成多个数据时，程序是按需获取数据的，它不会一开始就把所有数据都生成出来，而是每次调用 next() 获取下一个数据时，生成器才会执行一次，因此可以减少代码的执行次数。比如前面介绍的示例，程序不会一开始就把生成器函数中的循环都执行完成，而是每次调用 next() 时才执行一次循环体。
- 当函数需要返回多个数据时，如果不使用生成器，程序就需要使用列表或元组来收集函数返回的多个值，当函数要返回的数据量较大时，这些列表、元组会带来一定的内存开销。如果使用生成器就不存在这个问题，生成器可以按需、逐个返回数据。
- 使用生成器的代码更加简洁。

生成器的方法

当生成器运行起来之后，开发者还可以为生成器提供值，通过这种方式让生成器与“外部程序”动态地交换数据。

为了实现生成器与“外部程序” 动态地交换数据，需要借助于生成器的 send() 方法，该方法的功能与前面示例中所使用的 next() 函数的功能非常相似，它们都用于获取生成器所生成的下一个值，并将生成器“冻结”在 yield 语句处；但 send() 方法可以接收一个参数，该参数值会被发送给生成器函数。

在生成器函数内部，程序可通过 yield 表达式来获取 send() 方法所发送的值，这意味着此时程序应该使用一个变量来接收 yield 语句的值。如果程序依然使用 next() 函数来获取生成器所生成的下一个值，那么 yield 语句返回 None。

对于上面详细的描述，归纳起来就是两句话：

1. 外部程序通过 send() 方法发送数据。
2. 生成器函数使用 yield 语句接收收据。

另外，需要说明的是，只有等到生成器被“冻结”之后，外部程序才能使用 send() 方法向生成器发送数据。获取生成器第一次所生成的值，应该使用 next() 函数；如果程序非要使用 send() 方法获取生成器第一次所生成的值，也不能向生成器发送数据，只能为该方法传入 None 参数。

下面程序示范了向生成器发送数据。该程序会依次生成每个整数的平方值，但外部程序可以向生成器发送数据，当生成器接收到外部数据之后会生成外部数据的平方值：

```
1.          def square_gen(val):
2.              i = 0
3.              out_val = None
4.              while True:
5.                  # 使用 yield 语句生成值，使用 out_val 接收 send()方法发送的参数值
6.                  out_val = (yield out_val ** 2) if out_val is not None else (yield
    i ** 2)
7.                  # 如果程序使用 send()方法获取下一个值，out_val 会获取 send()方法
    的参数
8.                  if out_val is not None : print("====%d" % out_val)
9.                  i += 1
10.
11.         sg = square_gen(5)
12.         # 第一次调用 send()方法获取值，只能传入 None 作为参数
13.         print(sg.send(None))  # 0
14.         print(next(sg))  # 1
15.         print('--------------')
16.         # 调用 send()方法获取生成器的下一个值，参数 9 会被发送给生成器
17.         print(sg.send(9))  # 81
18.         # 再次调用 next()函数获取生成器的下一个值
19.         print(next(sg))  # 9
```

该程序与前面的简单生成器程序的区别就在于第 6 行代码，这行代码在 yield 语句（yield 语句被放在 if 表达式中，整个表达式只会运回一个 yield 语句）的左边放了一个变量，该变量就用于接受生成器 send() 方法所发送的值。

上面程序第一次使用生成器的 send() 方法来获取生成器的下一个值，因此只能为 send()

方法传入 None 作为参数。程序执行到第 9 行代码，由于此时 out_val 为 None ，因此程序执行 yield i**2（生成器返回 0），程序被“冻结”。注意，当程序被“冻结” 时，程序还未对 out_val 变量赋值，因此看到第一次获取生成器的值为 0 。

通过上面的执行过程不难看出，生成器根本不能获取第一次调用 send() 方法发送的参数值，这就是 Python 要求生成器第一次调用 send() 方法时只能发送 None 参数的原因。

接下来程序调用 next(sg) 获取生成器的下一个值，程序从“冻结”处（对 out_val 赋值）向下执行。由于此处调用 next() 函数获取生成器的下一个值，因此 out_val 被赋值为 None，所以程序执行 yield i**2（生成器返回 1），程序再次被“冻结”。

接下来程序调用 sg.send(9) 获取生成器的下一个值，程序从“冻结”处（对 out_val 赋值）向下执行。由于此处调用 send(9) 方法获取生成器的下一个值，因此 out_val 被赋值为 9，所以程序执行 yield out_val**2（生成器返回 81），程序再次被“冻结”。因此看到本次获取生成器的值为 81。

程序再次调用 next(sg) 获取生成器的下一个值，程序从“冻结”处（对 out_val 赋值）向下执行。由于此处调用 next() 函数获取生成器的下一个值，因此 out_val 被赋值为 None，所以程序执行 yield i**2（此时 i 已经递增到 3，因此生成器返回 9），程序再次被“冻结”。因此看到本次获取生成器的值为 9。

运行上面程序，可以看到如下输出结果：

```
0

1

--------------

====9

81

9
```

此外，生成器还提供了如下两个常用方法：

1. close()：该方法用于停止生成器。
2. throw()：该方法用于在生成器内部（yield 语句内）引发一个异常。

例如，在程序中增加如下代码：

```
# 让生成器引发异常
sg.throw(ValueError)
```

运行上面代码，将看到如下输出结果：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 21, in <module>
    sg.throw(ValueError)
  File "C:\Users\mengma\Desktop\1.py", line 6, in square_gen
    out_val = (yield out_val ** 2) if out_val is not None else (yield i ** 2)
ValueError
```

从上面的输出结果可以看到，在程序调用生成器的 throw() 方法引发异常之后，程序就会在 yield 语句中引发该异常。

将上面的 sg.throw(ValueError) 代码注释掉，为程序增加如下两行代码来示范 stop() 方法的用法。在程序调用 stop() 方法关闭生成器之后，程序就不能再去获取生成器的下一个值，否则就会引发异常。

```
        # 关闭生成器
        sg.close()
        print(next(sg)) # StopIteration
```

运行上面代码，可以看到如下输出结果：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 24, in <module>
    print(next(sg)) # StopIteration
StopIteration
```

12.Python @函数装饰器

前面章节中，我们已经讲解了 Python 内置的 3 种函数装饰器，分别是 @staticmethod、@classmethod 和 @property，其中 staticmethod、classmethod 和 property 都是 Python 的内置函数。那么，我们是否可以开发自定义的函数装饰器呢？

答案是肯定的。当程序使用“@函数”（比如函数 A）装饰另一个函数（比如函数 B）时，

实际上完成如下两步：

1. 将被修饰的函数（函数 B）作为参数传给 @ 符号引用的函数（函数 A）。
2. 将函数 B 替换（装饰）成第 1 步的返回值。

从上面介绍不难看出，被“@函数”修饰的函数不再是原来的函数，而是被替换成一个新的东西（取决于装饰器的返回值）。其实所谓的装饰器，就是通过装饰器函数，来修改原函数的一些功能，使得原函数不需要修改。

为了让大家厘清函数装饰器的作用，下面看一个非常简单的示例：

```
def funA(fn):
    print('A')
    fn() # 执行传入的 fn 参数
    return 'fkit'
'''
下面装饰效果相当于：funA(funB),
funB 将会替换（装饰）成 funA() 语句的返回值;
由于 funA()函数返回 fkit，因此 funB 就是 fkit
'''
@funA
def funB():
    print('B')
print(funB) # fkit
```

上面程序使用 @funA 修饰 funB，这意味着程序要完成两步操作：

- 将 funB 作为 funA() 的参数，也就是上面代码中 @funA 相当于执行 funA(funB)。

- 将 funB 替换成 funA() 执行的结果，funA() 执行完成后返回 fkit，因此 funB 就不再是函数，而是被替换成一个字符串。

其实，简单地理解函数装饰器的作用，上面程序可以等价地转换成如下程序：

```
def funA(fn):
    print('A')
    fn() # 执行传入的 fn 参数
    return 'fkit'
def funB():
    print('B')
funB = funA(funB)
print(funB) # fkit
```

注意，此程序中的 funB = funA(funB) 就等同于上面程序中 @funA 所起的作用。

运行上面 2 段程序，可以看到相同的输出结果：

```
A
B
Fkit
```

通过这个例子，相信读者对函数装饰器的执行关系已经有了一个较为清晰的认识，但读者可能会产生另一个疑问，这个函数装饰器导致被修饰的函数变成了字符串，那么函数装饰器有什么用？

别忘记了，被修饰的函数总是被替换成 @ 符号所引用的函数的返回值，因此被修饰的函数会变成什么，完全由于 @ 符号所引用的函数的返回值决定，换句话说，如果 @ 符号所引

用的函数的返回值是函数，那么被修饰的函数在替换之后还是函数。

带参数的函数装饰器

读者可能会想到，如果原函数 funB() 中有参数需要传递给函数装饰器，应该如何实现？

一个简单的办法是，可以在对应的函数装饰器 funA() 上，添加相应的参数，例如：

```
1.          def foo(fn):
2.                  # 定义一个嵌套函数
3.                  def bar(a):
4.                      fn(a * (a - 1))
5.                      print("*" * 15)
6.                      return fn(a * (a - 1))
7.                  return bar
8.          '''
9.          下面装饰效果相当于：foo(my_test),
10.         my_test 将会替换（装饰）成该语句的返回值；
11.         由于 foo()函数返回 bar 函数，因此 my_test 就是 bar
12.         同时，my_test 的参数 a 对应 bar 函数的参数 a
13.         '''
14.         @foo
15.         def my_test(a):
16.                 print("==my_test 函数==", a)
17.         # 打印 my_test 函数，将看到实际上是 bar 函数
18.         print(my_test)
19.         # 下面代码看上去是调用 my_test()，其实是调用 bar()函数
20.         my_test(10)
```

上面程序定义了一个装饰器函数 foo ,该函数执行完成后并不是返回普通值 ,而是返回 bar 函数（这是关键），这意味着被该 @foo 修饰的 my_test() 函数最终都会被替换成 bar() 函数。

上面程序使用 @foo 修饰 my_test() 函数，因此程序同样会执行 foo(my_test)，并将 my_test 替换成 foo() 函数的返回值：bar 函数。所以，上面程序第 18 行代码在打印 my_test 函数时，实际上输出的是 bar 函数，这说明 my_test 已经被替换成 bar 函数。接下来程序调用 my_test() 函数，实际上就是调用 bar() 函数。

运行上面程序，可以看到如下输出结果：

```
<function foo.<locals>.bar at 0x0000012D7E246598>
==my_test 函数== 90
***************
==my_test 函数== 90
```

在此基础上，还有一个问题，如果程序中另外还有一个函数，也需要使用 funA 装饰器，但是这个新的函数有 2 个参数，此时又该怎么办呢？例如：

```
1.          @foo
2.          def new_test(a,b):
3.              ....
```

在这种情况下，最简单的解决方式是用 *args 和 **kwargs 作为 foo 函数装饰器内部函数 bar() 的参数，*args 和 **kwargs 表示接受任意数量和类型的参数，因此函数装饰器可以写成下面的形式：

```
1.          def foo(fn):
2.              # 定义一个嵌套函数
```

```
            def bar(*args,**kwargs):
                fn(*args,**kwargs)
            return bar

        @foo
        def my_test(a):
            print("==my_test 函数==", a)

        @foo
        def new_test(a,b):
            print("==new_test 函数==",a," ",b)

        my_test(10)
        new_test(6, 5)
```

运行结果为：

```
==my_test 函数== 10

==new_test 函数== 6  5
```

带自定义参数的函数装饰器

其实，函数装饰器还有更大程度的灵活性。刚刚说了，装饰器可以接受原函数任意类型和数量的参数，除此之外，它还可以接受自己定义的参数。

举个例子，比如要定义一个参数，来表示装饰器内部函数被执行的次数，那么就可以写成下面这种形式：

```
        def foo(num):
            def my_decorator(fn):
                def bar(*args,**kwargs):
```

```
                    for i in range(num):
                        fn(*args,**kwargs)
                return bar
            return my_decorator

        @foo(3)
        def my_test(a):
            print("==my_test 函数==", a)

        @foo(5)
        def new_test(a,b):
            print("==new_test 函数==",a," ",b)

        my_test(10)
        new_test(6, 5)
```

运行结果为：

```
==my_test 函数== 10

==my_test 函数== 10

==my_test 函数== 10

==new_test 函数== 6  5

==new_test 函数== 6  5

==new_test 函数== 6  5

==new_test 函数== 6  5

==new_test 函数== 6  5
```

函数装饰器也可以嵌套

上面示例中，都是使用一个装饰器的情况，但实际上，Python 也支持多个装饰器，比如：

```
@decorator1
@decorator2
@decorator3
def func():
    ...
```

上面程序的执行顺序是里到外，所以它等效于下面这行代码：

```
decorator1( decorator2( decorator3(func) ) )
```

这里不再给出具体实例，有兴趣的读者可自行编写程序进行测试。

13.结合实例，深入了解装饰器！

前面章节已经讲解了装饰器的基本概念及用法，本节将结合实际工作中的几个例子，带读者加深对它的理解。

装饰器用于身份认证

首先是最常见的身份认证的应用。这个很容易理解，举个最常见的例子，大家登录微信，需要输入用户名密码，然后点击确认，这样服务器端便会查询你的用户名是否存在、是否和密码匹配等等。如果认证通过，就可以顺利登录；反之，则提示你登录失败。

再比如一些网站，你不登录也可以浏览内容，但如果你想要发布文章或留言，在点击发布时，服务器端便会查询你是否登录。如果没有登录，就不允许这项操作等等。

如下是一个实现身份认证的简单示例：

```
import functools

def authenticate(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        request = args[0]
        # 如果用户处于登录状态
        if check_user_logged_in(request):
            # 执行函数 post_comment()
```

```
                    return func(*args, **kwargs)
                else:
                    raise Exception('Authentication failed')
            return wrapper

        @authenticate
        def post_comment(request, ...)
            ...
```

注意，对于函数来说，它也有自己的一些属性，例如 __name__ 属性，代码中 @functools.wraps(func) 也是一个装饰器，如果不使用它，则 post_comment.__name__ 的值为 wrapper。而使用它之后，则 post_comment.__name__ 的值依然为 post_comment。

上面这段代码中，定义了装饰器 authenticate，函数 post_comment() 则表示发表用户对某篇文章的评论，每次调用这个函数前，都会先检查用户是否处于登录状态，如果是登录状态，则允许这项操作；如果没有登录，则不允许。

装饰器用于日志记录

日志记录同样是很常见的一个案例。在实际工作中，如果你怀疑某些函数的耗时过长，导致整个系统的延迟增加，想在线上测试某些函数的执行时间，那么，装饰器就是一种很常用的手段。

我们通常用下面的方法来表示：

```
        import time
        import functools

        def log_execution_time(func):
            @functools.wraps(func)
            def wrapper(*args, **kwargs):
                start = time.perf_counter()
                res = func(*args, **kwargs)
                end = time.perf_counter()
                print('{} took {} ms'.format(func.__name__, (end - start) * 1000))
                return res
            return wrapper

        @log_execution_time
```

```
15.         def calculate_similarity(items):
16.             ...
```

这里，装饰 log_execution_time 记录某个函数的运行时间，并返回其执行结果。如果你想计算任何函数的执行时间，在这个函数上方加上@log_execution_time 即可。

装饰器用于输入合理性检查

在大型公司的机器学习框架中，调用机器集群进行模型训练前，往往会用装饰器对其输入（往往是很长的 json 文件）进行合理性检查。这样就可以大大避免输入不正确对机器造成的巨大开销。

它的写法往往是下面的格式：

```
1.          import functools
2.
3.          def validation_check(input):
4.              @functools.wraps(func)
5.              def wrapper(*args, **kwargs):
6.                  ... # 检查输入是否合法
7.
8.          @validation_check
9.          def neural_network_training(param1, param2, ...):
10.             ...
```

其实在工作中，很多情况下都会出现输入不合理的现象。因为我们调用的训练模型往往很复杂，输入的文件有成千上万行，很多时候确实也很难发现。

试想一下，如果没有输入的合理性检查，很容易出现“模型训练了好几个小时后，系统却报错说输入的一个参数不对，成果付之一炬”的现象。这样的“惨案”，大大减缓了开发效率，也对机器资源造成了巨大浪费。

缓存装饰器

关于缓存装饰器的用法，其实十分常见，这里以 Python 内置的 LRU cache 为例来说明。

LRU cache，在 Python 中的表示形式是 @lru_cache。@lru_cache 会缓存进程中的函数参数和结果，当缓存满了以后，会删除最近最久未使用的数据。

正确使用缓存装饰器，往往能极大地提高程序运行效率。举个例子，大型公司服务器端的代码中往往存在很多关于设备的检查，比如使用的设备是安卓还是 iPhone，版本号是多少。

这其中的一个原因，就是一些新的功能，往往只在某些特定的手机系统或版本上才有（比如 Android v200+）。

这样一来，我们通常使用缓存装饰器来包裹这些检查函数，避免其被反复调用，进而提高程序运行效率，比如写成下面这样：

```
1.      @lru_cache
2.      def check(param1, param2, ...) # 检查用户设备类型，版本号等等
```

第 11 章 Python 模块和包

前面在介绍异常时已经用到了系统提供的 sys、 OS、 traceback 这三个模块，可能已有读者对模块这个“神秘”的东西感到很好奇，Python 每次导入模块之后，Python 似乎就增加了某种特别的功能，模块怎么这么厉害呢？这些模块到底是从哪里来的？

本章将会带着大家去真正了解模块这个强大的工具。实际上，Python 语言之所以能被广泛应用于各行各业，在很大程度上得益于它的模块化系统。在 Python 的标准安装中包含了一组自带的模块，这些模块被称为“标准库”。

Python 3 的标准库请参考 https://docs.python.org/3/library/index.html。
更重要的是，开发者完全可以根据自己的需要不断地为 Python 增加扩展库。各行各业的 Python 用户贡献了大量的扩展库，这些扩展库极大地丰富了 Python 的功能，这些扩展库从某种程度上也形成了 Python 的“生态圈”。

1.什么是模块

Python 提供了强大的模块支持，主要体现在，不仅 Python 标准库中包含了大量的模块（称为标准模块），还有大量的第三方模块，开发者自己也可以开发自定义模块。通过这些强大的模块可以极大地提高开发者的开发效率。

那么，模块到底指的是什么呢？模块，英文为 Modules，至于模块到底是什么，可以用一句话总结：模块就是 Python 程序。换句话说，任何 Python 程序都可以作为模块，包括在前面章节中写的所有 Python 程序，都可以作为模块。

模块可以比作一盒积木，通过它可以拼出多种主题的玩具，这与前面介绍的函数不同，一个函数仅相当于一块积木，而一个模块（.py 文件）中可以包含多个函数，也就是很多积木。模块和函数的关系如图 1 所示。

图 1 模块和函数的关系

经过前面的学习，读者已经能够将 Python 代码写到一个文件中，但随着程序功能的复杂，程序体积会不断变大，为了便于维护，通常会将其分为多个文件（模块），这样不仅可以提高代码的可维护性，还可以提高代码的可重用性。

> 代码的可重用性体现在，当编写好一个模块后，只要编程过程中需要用到该模块中的某个功能（由变量、函数、类实现），无需做重复性的编写工作，直接在程序中导入该模块即可使用该功能。

前面讲了封装，并且还介绍了很多具有封装特性的结构，比如说：

- 诸多容器，例如列表、元组、字符串、字典等，它们都是对数据的封装；
- 函数是对 Python 代码的封装；
- 类是对方法和属性的封装，也可以说是对函数和数据的封装。

本节所介绍的模块，可以理解为是对代码更高级的封装，即把能够实现某一特定功能的代码编写在同一个 .py 文件中，并将其作为一个独立的模块，这样既可以方便其它程序或脚本导入并使用，同时还能有效避免函数名和变量名发生冲突。

举个简单的例子，在某一目录下（桌面也可以）创建一个名为 hello.py 文件，其包含的代码如下：

```
def say ():
    print("Hello,World!")
```

在同一目录下，再创建一个 say.py 文件，其包含的代码如下：

```
#通过 import 关键字，将 hello.py 模块引入此文件
import hello
hello.say()
```

运行 say.py 文件，其输出结果为：

```
Hello,World!
```

读者可能注意到，say.py 文件中使用了原本在 hello.py 文件中才有的 say() 函数，相对于 day.py 来说，hello.py 就是一个自定义的模块（有关自定义模块，后续章节会做详细讲解），我们只需要将 hellp.py 模块导入到 say.py 文件中，就可以直接在 say.py 文件中使用模块中的资源。

与此同时，当调用模块中的 say() 函数时，使用的语法格式为“模块名.函数”，这是因为，相对于 say.py 文件，hello.py 文件中的代码自成一个命名空间，因此在调用其他模块中的函数时，需要明确指明函数的出处，否则 Python 解释器将会报错。

2.Python import 导入模块

使用 Python 进行编程时，有些功能没必须自己实现，可以借助 Python 现有的标准库或者其他人提供的第三方库。比如说，在前面章节中，我们使用了一些数学函数，例如余弦函数 cos()、绝对值函数 fabs() 等，它们位于 Python 标准库中的 math（或 cmath）模块中，只需要将此模块导入到当前程序，就可以直接拿来用。

前面章节中，已经看到使用 import 导入模块的语法，但实际上 import 还有更多详细的用法，主要有以下两种：

1. `import 模块名 1 [as 别名 1], 模块名 2 [as 别名 2]，...`：使用这种语法格式的 import 语句，会导入指定模块中的所有成员（包括变量、函数、类等）。不仅如此，当需要使用模块中的成员时，需用该模块名（或别名）作为前缀，否则 Python 解释器会报错。
2. `from 模块名 import 成员名 1 [as 别名 1]，成员名 2 [as 别名 2]，...`： 使用这种语法格式的 import 语句，只会导入模块中指定的成员，而不是全部成员。同时，当程序中使用该成员时，无需附加任何前缀，直接使用成员名（或别名）即可。

注意，用 [] 括起来的部分，可以使用，也可以省略。

> 其中，第二种 import 语句也可以导入指定模块中的所有成员，即使用 form 模块名 import * ，但此方式不推荐使用，具体原因本节后续会做详细说明。

import 模块名 as 别名

下面程序使用导入整个模块的最简单语法来导入指定模块：

```
1.  # 导入 sys 整个模块
2.  import sys
3.  # 使用 sys 模块名作为前缀来访问模块中的成员
4.  print(sys.argv[0])
```

上面第 2 行代码使用最简单的方式导入了 sys 模块，因此在程序中使用 sys 模块内的成员时，必须添加模块名作为前缀。

运行上面程序，可以看到如下输出结果(sys 模块下的 argv 变量用于获取运行 Python 程序的命令行参数，其中 argv[0] 用于获取当前 Python 程序的存储路径)：

```
C:\Users\mengma\Desktop\hello.py
```

导入整个模块时，也可以为模块指定别名。例如如下程序：

```
1.          # 导入 sys 整个模块，并指定别名为 s
2.          import sys as s
3.          # 使用 s 模块别名作为前缀来访问模块中的成员
4.          print(s.argv[0])
```

第 2 行代码在导入 sys 模块时才指定了别名 s ,因此在程序中使用 sys 模块内的成员时，必须添加模块别名 s 作为前缀。运行该程序，可以看到如下输出结果：

```
C:\Users\mengma\Desktop\hello.py
```

也可以一次导入多个模块，多个模块之间用逗号隔开。例如如下程序：

```
1.          # 导入 sys、os 两个模块
2.          import sys,os
3.          # 使用模块名作为前缀来访问模块中的成员
4.          print(sys.argv[0])
5.          # os 模块的 sep 变量代表平台上的路径分隔符
6.          print(os.sep)
```

上面第 2 行代码一次导入了 sys 和 os 两个模块，因此程序要使用 sys、os 两个模块内的成员，只要分别使用 sys、os 模块名作为前缀即可。在 Windows 平台上运行该程序，可以看到如下输出结果（os 模块的 sep 变量代表平台上的路径分隔符）：

```
C:\Users\mengma\Desktop\hello.py

\
```

在导入多个模块的同时，也可以为模块指定别名，例如如下程序：

```
1.          # 导入 sys、os 两个模块，并为 sys 指定别名 s，为 os 指定别名 o
2.          import sys as s,os as o
3.          # 使用模块别名作为前缀来访问模块中的成员
4.          print(s.argv[0])
5.          print(o.sep)
```

上面第 2 行代码一次导入了 sys 和 os 两个模块，并分别为它们指定别名为 s、o，因此程序可以通过 s、o 两个前缀来使用 sys、os 两个模块内的成员。在 Windows 平台上运行该程序，可以看到如下输出结果：

```
C:\Users\mengma\Desktop\hello.py

\
```

from 模块名 import 成员名 as 别名

下面程序使用了 from...import 最简单的语法来导入指定成员：

```
1.          # 导入 sys 模块的 argv 成员
2.          from sys import argv
3.          # 使用导入成员的语法，直接使用成员名访问
4.          print(argv[0])
```

第 2 行代码导入了 sys 模块中的 argv 成员，这样即可在程序中直接使用 argv 成员，无须使用任何前缀。运行该程序，可以看到如下输出结果：

```
C:\Users\mengma\Desktop\hello.py
```

导入模块成员时，也可以为成员指定别名，例如如下程序：

```
1.          # 导入 sys 模块的 argv 成员，并为其指定别名 v
2.          from sys import argv as v
3.          # 使用导入成员（并指定别名）的语法，直接使用成员的别名访问
4.          print(v[0])
```

第 2 行代码导入了 sys 模块中的 argv 成员，并为该成员指定别名 v，这样即可在程序中通过别名 v 使用 argv 成员，无须使用任何前缀。运行该程序，可以看到如下输出结果：

```
C:\Users\mengma\Desktop\hello.py
```

form...import 导入模块成员时，支持一次导入多个成员，例如如下程序：

```
1.          # 导入 sys 模块的 argv,winver 成员
2.          from sys import argv, winver
3.          # 使用导入成员的语法，直接使用成员名访问
4.          print(argv[0])
5.          print(winver)
```

上面第 2 行代码导入了 sys 模块中的 argv、 winver 成员，这样即可在程序中直接使用 argv、winver 两个成员，无须使用任何前缀。运行该程序，可以看到如下输出结果（ sys 的 winver 成员记录了该 Python 的版本号 ）：

```
C:\Users\mengma\Desktop\hello.py
3.6
```

一次导入多个模块成员时，也可指定别名，同样使用 as 关键字为成员指定别名，例如如下程序：

```
1.          # 导入 sys 模块的 argv,winver 成员，并为其指定别名 v、wv
2.          from sys import argv as v, winver as wv
3.          # 使用导入成员（并指定别名）的语法，直接使用成员的别名访问
4.          print(v[0])
5.          print(wv)
```

上面第 2 行代码导入了 sys 模块中的 argv、winver 成员，并分别为它们指定了别名 v、wv，这样即可在程序中通过 v 和 wv 两个别名使用 argv、winver 成员，无须使用任何前缀。运行该程序，可以看到如下输出结果：

```
C:\Users\mengma\Desktop\hello.py
3.6
```

不推荐使用 from import 导入模块所有成员

在使用 from...import 语法时，可以一次导入指定模块内的所有成员（此方式不推荐），例如如下程序：

```
1.          #导入 sys 棋块内的所有成员
2.          from sys import *
3.          #使用导入成员的语法，直接使用成员的别名访问
4.          print(argv[0])
5.          print(winver)
```

上面代码一次导入了 sys 模块中的所有成员，这样程序即可通过成员名来使用该模块内的所有成员。该程序的输出结果和前面程序的输出结果完全相同。

需要说明的是，一般不推荐使用“from 模块 import”这种语法导入指定模块内的所有成员，因为它存在潜在的风险。比如同时导入 module1 和 module2 内的所有成员，假如这两个模块内都有一个 foo() 函数，那么当在程序中执行如下代码时：

```
foo()
```

上面调用的这个 foo() 函数到底是 module1 模块中的还是 module2 模块中的？因此，这种导入指定模块内所有成员的用法是有风险的。

但如果换成如下两种导入方式：

```
import module1

import module2 as m2
```

接下来要分别调用这两个模块中的 foo() 函数就非常清晰。程序可使用如下代码：

纯文本复制

```
#使用模块 module1 的模块名作为前缀调用 foo()函数
module1.foo()
#使用 module2 的模块别名作为前缀调用 foo()函数
m2.foo()
```

或者使用 from...import 语句也是可以的：

```
#导入 module1 中的 foo 成员，并指定其别名为 foo1
from module1 import foo as foo1
#导入 module2 中的 foo 成员，并指定其别名为 foo2
from module2 import foo as foo2
```

此时通过别名将 module1 和 module2 两个模块中的 foo 函数很好地进行了区分，接下来分别调用两个模块中 foo() 函数就很清晰：

```
foo1() #调用 module1 中的 foo()函数

foo2() #调用 module2 中的 foo()函数
```

3.Python 自定义模块

到目前为止，读者已经掌握了导入 Python 标准库并使用其成员（主要是函数）的方法，接下来要解决的问题是，怎样自定义一个模块呢？

前面章节中讲过，Python 模块就是 Python 程序，任何 Python 程序都可以作为模块。也就是说，随便写的一个 Python 程序，都可以作为模块。例如，下面程序定义了一个简单的模块（编写在 module1.py 文件中）：

```
1.          print('这是 module 1')
2.          my_book = 'Python 入门教程'
3.          def say_hi(user):
4.              print('%s,您好，欢迎学习 Python' % user)
5.          class User:
6.              def __init__(self, name):
7.                  self.name = name
8.              def walk(self):
9.                  print('%s 正在慢慢地走路' % self.name)
10.             def __repr__(self):
11.                 return 'User[name=%s]' % self.name
```

上面程序中，第一行代码执行了一条简单的输出语句，第二行代码定义了一个 my_book 变量。接下来程序定义了一个 say_hi() 函数，然后定义了一个 User 类。不难发现，这个程序和我们前面所写的 Python 程序并没有太大的区别，它就可以作为模块。

使用模块的好处在于，如果将程序将频繁使用的代码段（比如刚刚定义的 say_hi() 函数、User 类）定义在模块中，后面不管哪个程序，只要导入该模块，此程序就可以直接使用该模块所包含的代码段，从而避免每个程序都需要重复编写这些代码段，提高了程序的可复用性。

> 模块文件的文件名就是它的模块名，比如 module1.py 的模块名就是 module1。

自定义模块编写测试代码

当自定义模块编写完成之后，需要编写一些测试代码，检验模块中各个功能是否都能正常运行。由于模块就是一段 Python 程序，因此只要模块中包含可执行代码，就可以直接执行模块中的程序。

比如，利用 Python IDLE 打开 module1.py 文件并运行，即可看到如下输出结果：

> 这是 module 1

上面这行输出是因为 module1.py 程序中包含了一条 print 语句。但模块中的变量、函数、类都没有得到测试，因此还应该为这些变量、函数、类提供测试程序。

例如，可以为上面的 module1.py 增加如下测试代码：

```
1.          # ===以下部分是测试代码===
2.          def test_my_book ():
3.              print(my_book)
4.          def test_say_hi():
```

```
5.              say_hi('孙悟空')
6.              say_hi(User('Charlie'))
7.          def test_User():
8.              u = User('白骨精')
9.              u.walk()
10.             print(u)
11.         test_my_book()
12.         test_say_hi()
13.         test_User()
```

上面代码为 module1 定义了三个函数，分别用于测试模块中的变量、函数和类，如果此时再运行该模块，可以得到如下运行结果：

```
这是 module 1
Python 入门教程
孙悟空,您好，欢迎学习 Python
User[name=Charlie],您好，欢迎学习 Python
白骨精正在慢慢地走路
User[name=白骨精]
```

对于实际开发的项目，对每个函数、类可能都需要使用更多的测试用例进行测试，这样才能达到各种覆盖效果（比如语句覆盖、条件覆盖等）。本教程并不介绍测试相关内容，因此就不深入展开了。

需要注意的是，像上面这样，直接向模块程序中添加测试代码，会导致一个问题，即当其它程序导入该模块时，其中的测试函数会自动执行。例如，在 module1.py 同目录下，再创建一个新的 module2.py 文件，向该文件中添加如下代码：

```
1.          import module1
```

运行 module2.py 文件，可以看到如下运行结果：

```
这是 module 1
Python 入门教程
孙悟空,您好，欢迎学习 Python
User[name=Charlie],您好，欢迎学习 Python
白骨精正在慢慢地走路
User[name=白骨精]
```

这显然不是期望看到的结果。我们希望的效果是 如果直接运行模块文件(相当于测试文件)，程序会执行该模块的测试函数；如果是其他程序导入该模块，程序不应该执行该模块的测试函数。

要实现这个效果，需要借助 Python 内置的 __name__ 变量。当直接运行一个模块时，name 变量的值为 __main__；而将模块被导入其他程序中并运行该程序时，处于模块中的 __name__ 变量的值就变成了模块名。因此，如果希望测试函数只有在直接运行模块文件时才执行，则可在调用测试函数时增加判断，即只有当 __name__ =='__main__' 时才调用测试函数。

例如，修改 module1.py 模块文件的测试代码，如下所示：

```
1.          # ===以下部分是测试代码===
2.          if __name__=='__main__':
3.              def test_my_book ():
```

```
4.                  print(my_book)
5.              def test_say_hi():
6.                  say_hi('孙悟空')
7.                  say_hi(User('Charlie'))
8.              def test_User():
9.                  u = User('白骨精')
10.                 u.walk()
11.                 print(u)
12.             test_my_book()
13.             test_say_hi()
14.             test_User()
```

修改完成后，再次执行模块文件（module1.py），测试代码会执行；而执行导入模块文件的module2.py，模块中的测试代码不会执行。由此，就实现了我们想要的效果。

自定义模块编写说明文档
与前面介绍的函数、类相同的是，在实际开发中往往也应该为模块编写说明文档；否则，其他开发者将不知道该模块有什么作用，以及包含哪些功能。

为模块编写说明文档很简单，只要在模块开头的位置定义一个字符串即可。例如，在上面程序的第一行代码之前添加如下内容：

```
'''

这是我们编写的第一个模块，该模块包含以下内容：

my_book：字符串变量

say_hi：简单的函数

User：代表用户的类

'''
```

这段字符串内容将会作为该模块的说明文挡，可通过模块的 __doc__ 属性来访问文档。例如，在 module2.py 中添加如下代码：

```
1.          import module1
2.          print(module1.__doc__)
```

运行 module2.py，执行结果为：

```
这是 module 1


这是我们编写的第一个模块，该模块包含以下内容：
my_book：字符串变量
say_hi：简单的函数
User：代表用户的类
```

4.Python __name__=='__main__'

前面章节已经对模块及其用法做了详解的介绍，相信有很多读者已经开始去尝试阅读别人的代码了（通常阅读比自己牛的人写的代码，会让自己的技术水平飞速提高）。不过，在阅读别人写的自定义模块时，经常会看到有如下这行判断语句：

```
1.          if __name__ == '__main__':
```

这行代码的作用是什么呢？本节就详解讲解以下它的作用。

一般情况下，当我们写完自定义的模块之后，都会写一个测试代码，检验一些模块中各个功能是否能够成功运行。例如，创建一个 candf.py 文件，并编写如下代码：

```
1.          '''
2.          摄氏度和华氏度的相互转换模块
3.          '''
4.          def c2f(cel):
5.              fah = cel * 1.8 + 32
```

```
6.              return fah
7.          def f2c(fah):
8.              cel = (fah - 32) / 1.8
9.              return cel
10.         def test():
11.             print("测试数据：0 摄氏度 = %.2f 华氏度" % c2f(0))
12.             print("测试数据：0 华氏度 = %.2f 摄氏度" % f2c(0))
13.         test()
```

单独运行此模块文件，可以看到如下运行结果：

```
测试数据：0 摄氏度 = 32.00 华氏度

测试数据：0 华氏度 = -17.78 摄氏度
```

在 candf.py 模块文件的基础上，在同目录下再创建一个 demo.py 文件，并编写如下代码：

```
1.          import candf
2.          print("32 摄氏度 = %.2f 华氏度" % candf.c2f(32))
3.          print("99 华氏度 = %.2f 摄氏度" % candf.f2c(99))
```

运行 demo.py 文件，其运行结果如下所示：

```
测试数据：0 摄氏度 = 32.00 华氏度

测试数据：0 华氏度 = -17.78 摄氏度

32 摄氏度 = 89.60 华氏度

99 华氏度 = 37.22 摄氏度
```

可以看到，Python 解释器将模块（candf.py）中的测试代码也一块儿运行了，这并不是我们想要的结果。想要避免这种情况的关键在于，要让 Python 解释器知道，当前要运行的程度代码，是模块文件本身，还是导入模块的其它程序。

为了实现这一点，就需要使用 Python 内置的系统变量 __name__，它用于标识所在模块的模块名。例如，在 demo.py 程序文件中，添加如下代码：

```
print(__name__)
print(candf.__name__)
其运行结果为：
__main__
candf
```

可以看到，当前运行的程序，其 __name__ 的值为 __main__，而导入到当前程序中的模块，其 __name__ 值为自己的模块名。

因此，if __name__ == '__main__': 的作用是确保只有单独运行该模块时，此表达式才成立，才可以进入此判断语法，执行其中的测试代码；反之，如果只是作为模块导入到其他程序文件中，则此表达式将不成立，运行其它程序时，也就不会执行该判断语句中的测试代码。

5.Python 导入模块的 3 种方式

在编写完 Python 模块之后，如果直接用 import 或 from...import 来导入该模块，有时并不能正确地加载该模块。道理很简单，仅通过我们提供的模块文件名，Python 无法找到该模块文件。

编程其实并不难，就是用合适的语法告诉计算机，让它帮助完成某个工作。因此计算机能完成的事情，其实都是程序员预先告诉它的。

通常情况下，当使用 import 语句导入模块后，Python 会按照以下顺序查找指定的模块文件：

- 在当前目录，即当前执行的程序文件所在目录下查找；
- 到 PYTHONPATH（环境变量）下的每个目录中查找；
- 到 Python 默认的安装目录下查找。

以上所有涉及到的目录，都保存在标准模块 sys 的 sys.path 变量中，通过此变量，可以输出指定程序文件支持查找的所有目录。因此，如果要导入的模块没有存储在 sys.path 显示的目录中，那么导入该模块并运行程序时，Python 解释器会抛出 ModuleNotFoundError（未找到模块）异常。

为了让 Python 能找到自定义（或第三方提供）的模块，可以用以下 3 种方式来告诉它：

1. 向 sys.path 中临时添加模块文件存储位置的完整路径；
2. 将模块放在 sys.path 变量中已包含的模块加载路径中。
3. 设置系统环境变量。

介绍这 3 种方式之前，为了能更方便地讲解，本节使用前面章节已建立好的 hello.py 自定义模块文件（D:\python_module\hello.py）和 say.py 程序文件（C:\Users\mengma\Desktop\say.py），它们各自包含的代码如下：

```
#hello.py
def say ():
    print("Hello,World!")

#say.py
import hello
hello.say()
```

可以看到，hello.py 文件和 say.py 文件并不在同一目录，此时运行 say.py 文件，其运行结果为：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\say.py", line 1, in <module>
    import hello
ModuleNotFoundError: No module named 'hello'
```

可以看到，Python 解释器抛出了 ModuleNotFoundError 异常。接下来，分别用以上 3 种方法解决这个问题。

导入模块方式一：临时添加模块完整路径

模块文件的存储位置，可以临时添加到 sys.path 变量中，即向 sys.path 中添加 D:\python_module（hello.py 所在目录），在 say.py 中的开头位置添加如下代码：

```
import sys
sys.path.append('D:\\python_module')
```

注意：在添加完整路径中，路径中的 '\' 需要使用 \ 进行转义，否则会导致语法错误。再次运行 say.py 文件，运行结果如下：

```
Hello,World!
```

可以看到，程序成功运行。在此基础上，我们在 say.py 文件中输出 sys.path 变量的值，会得到以下结果：

```
['C:\\Users\\mengma\\Desktop', 'D:\\python3.6\\Lib\\idlelib',
'D:\\python3.6\\python36.zip', 'D:\\python3.6\\DLLs', 'D:\\python3.6\\lib',
'D:\\python3.6',
'C:\\Users\\mengma\\AppData\\Roaming\\Python\\Python36\\site-packages',
'D:\\python3.6\\lib\\site-packages', 'D:\\python3.6\\lib\\site-packages\\win32',
'D:\\python3.6\\lib\\site-packages\\win32\\lib',
'D:\\python3.6\\lib\\site-packages\\Pythonwin', 'D:\\python_module']
```

该输出信息中，红色部分就是临时添加进去的存储路径。需要注意的是，通过该方法添加的目录，只能在执行当前文件的窗口中有效，窗口关闭后即失效。

导入模块方式二：将模块保存到指定位置

如果要安装某些通用性模块，比如复数功能支持的模块、矩阵计算支持的模块、图形界面支持的模块等，这些都属于对 Python 本身进行扩展的模块，这种模块应该直接安装在 Python 内部，以便被所有程序共享，此时就可借助于 Python 默认的模块加载路径。

Python 程序默认的模块加载路径保存在 sys.path 变量中，因此，我们可以在 say.py 程序文件中先看看 sys.path 中保存的默认加载路径，向 say.py 文件中输出 sys.path 的

值，如下所示：

```
['C:\\Users\\mengma\\Desktop', 'D:\\python3.6\\Lib\\idlelib',
'D:\\python3.6\\python36.zip', 'D:\\python3.6\\DLLs', 'D:\\python3.6\\lib',
'D:\\python3.6',
'C:\\Users\\mengma\\AppData\\Roaming\\Python\\Python36\\site-packages', 'D:\
\python3.6\\lib\\site-packages', 'D:\\python3.6\\lib\\site-packages\\win32',
'D:\\python3.6\\lib\\site-packages\\win32\\lib',
'D:\\python3.6\\lib\\site-packages\\Pythonwin']
```

上面的运行结果中，列出的所有路径都是 Python 默认的模块加载路径，但通常来说，我们默认将 Python 的扩展模块添加在 `lib\site-packages` 路径下，它专门用于存放 Python 的扩展模块和包。

所以，我们可以直接将我们已编写好的 hello.py 文件添加到 `lib\site-packages` 路径下，就相当于为 Python 扩展了一个 hello 模块，这样任何 Python 程序都可使用该模块。

移动工作完成之后，再次运行 say.py 文件，可以看到成功运行的结果：

```
Hello,World!
```

导入模块方式三：设置环境变量

Python 将会根据 PYTHONPATH 环境变量的值来确定到哪里去加载模块。

PYTHONPATH 环境变量的值是多个路径的集合，这样 Python 就会依次搜索 PYTHONPATH 环境变量所指定的多个路径，试图从中找到程序想要加载的模块。

下面介绍在不同平台上设置环境变量的方式。

在 Windows 平台上设置环境变量

右击桌面上的“计算机”图标，出现右键菜单；单击“属性”菜单项，系统显示“控制面板\所有控制面板项\系统”窗口，单击该窗口左边栏中的“高级系统设置”链接，出现“系统属性”对话框，如图 1 所示。

图 1 系统属性对话框

单击该对话框中“高级” Tab 页中的“环境变量”按钮，将看到如图 2 所示的“环境变量”对话框，通过该对话框可以添加或修改环境变量。

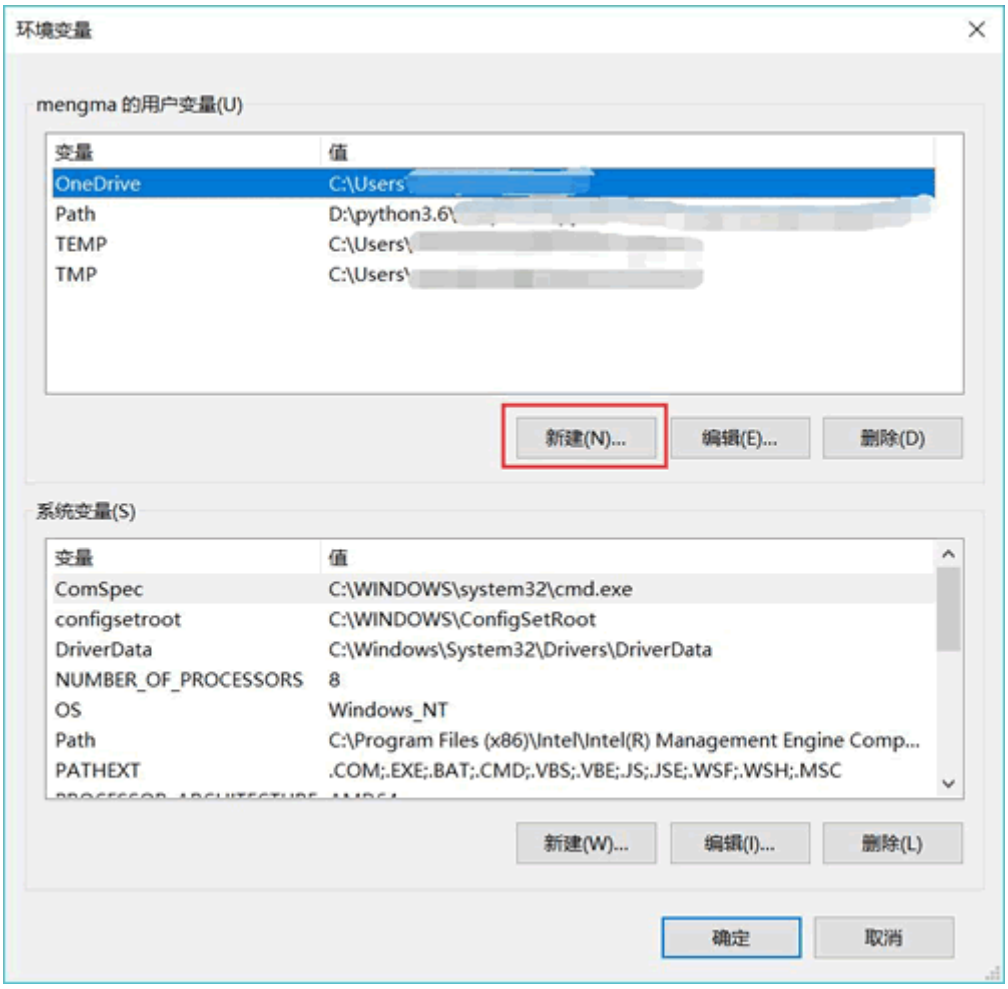


图 2 环境变量对话框

在如图 2 所示的对话框中，上面的“用户变量”部分用于设置当前用户的环境变量，下面的“系统变量”部分用于设置整个系统的环境变量。

一般建议设置“用户变量”即可，因为用户变量只对当前用户有效，而系统变量对所有用户有效。为了减少自己所做的修改对其他人的影响，故设置用户变量。对于当前用户而言，设置用户变量和系统变量的效果大致相同，不过系统变量的路径排在用户变量的路径之前。

单击用户变量中的“新建”按钮， 系统显示如图 3 所示的对话框。

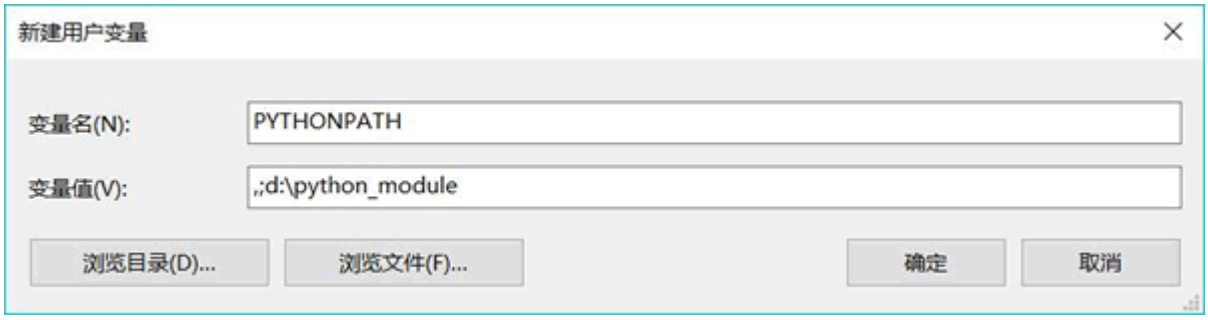


图 3 新建 PYTHONPATH 环境变量

在“变量名”文本框内输入 PYTHONPATH，表明将要建立名为 PYTHONPATH 的环境变量；在“变量值”文本框内输入 .;d:\python_ module，这就是该环境变量的值，该值其实包含了两条路径（以 ；分隔）：

- 第一条路径为一个点（.），这个点代表当前路径，表明当运行 Python 程序时，Python 总能从当前路径加载模块；
- 第二条路径为 d:\python_ module，表明当运行 Python 程序时，Python 总能从 d:\python_ module 加载模块。

在成功设置了上面的环境变量之后，接下来只要把前面定义的模块（Python 程序）放在与当前所运行 Python 程序相同的路径中（或放在 d:\python_ module 路径下），该模块就能被成功加载了。

在 Linux 上设置环境变量

启动 Linux 的终端窗口(命令行界面),进入当前用户的 home 路径下 ,然后在 home 路径下输入如下命令：

```
ls - a
```

该命令将列出当前路径下所有的文件，包括隐藏文件。Linux 平台的环境变量是通过 .bash_profile 文件来设置的，使用无格式编辑器打开该文件，在该文件中添加 PYTHONPATH 环境变量。也就是为该文件增加如下一行：

```
#设置 PYTHON PATH 环境变量
PYTHONPATH=.:/home/mengma/python_module
```

Linux 与 Windows 平台不一样，多个路径之间以冒号 (:) 作为分隔符，因此上面一行同样设置了两条路径，点 (.) 代表当前路径，还有一条路径是 `/home/mengma/python_module`（mengma 是在 Linux 系统的登录名)。

在完成了 PYTHONPATH 变量值的设置后，在 .bash_profile 文件的最后添加导出 PYTHONPATH 变量的语句。

```
#导出 PYTHONPATH 环境变量
export PYTHONPATH
```

重新登录 Linux 平台，或者执行如下命令：

```
source.bash_profile
```

这两种方式都是为了运行该文件，使在文件中设置的 PYTHONPATH 变量值生效。

在成功设置了上面的环境变量之后，接下来只要把前面定义的模块（Python 程序）放在与当前所运行 Python 程序相同的路径中(或放在 `/home/mengma/python_module` 路径下)，该模块就能被成功加载了。

在 Mac OS X 上设置环境变量

在 Mac OS X 上设置环境变量与 Linux 大致相同（因为 Mac OS X 本身也是类 UNIX 系统）。启动 Mac OS X 的终端窗口（命令行界面），进入当前用户的 home 路径下，然后在 home 路径下输入如下命令：

```
ls -a
```

该命令将列出当前路径下所有的文件，包括隐藏文件。Mac OS X 平台的环境变量也可通过，bash_profile 文件来设置，使用无格式编辑器打开该文件，在该文件中添加 PYTHONPATH 环境变量。也就是为该文件增加如下一行：

```
#设置 PYTHON PATH 环境变盘
PYTHONPATH=.:/Users/mengma/python_module
```

Mac OS X 的多个路径之间同样以冒号（:）作为分隔符，因此上面一行同样设置了两条路

径 :点(.)代表当前路径 ,还有一条路径是 /Users/mengma/python_module (memgma 是作者在 Mac OS X 系统的登录名)。

在完成了 PYTHONPATH 变量值的设置后，在 .bash_profile 文件的最后添加导出 PYTHONPATH 变量的语句。

```
#导出 PYTHON PATH 环境变量
export PYTHONPATH
```

重新登录 Mac OS X 系统，或者执行如下命令：

```
source.bash_profile
```

这两种方式都是为了运行该文件，使在文件中设置的 PYTHONPATH 变量值生效。

在成功设置了上面的环境变量之后，接下来只要把前面定义的模块 (Python 程序) 放在与当前所运行 Python 程序相同的路径中 (或放在 Users/mengma/python_module 路径下) , 该模块就能被成功加载了。

6.Python 导入模块的本质

为了帮助大家更好地理解导入模块，下面定义一个新的模块，该模块比较简单，所以不再为之编写测试代码。该模块代码如下 (编写在 fk_module.py 文件中) :

```
1.          '一个简单的测试模块: fk_module'
2.          print("this is fk_module")
3.          name = 'fkit'
```

```
4.          def hello():
5.              print("Hello, Python")
```

接下来，在相同的路径下定义如下程序来使用该模块：

```
1.          import fk_module
2.          print("================")
3.          # 打印 fk_module 的类型
4.          print(type(fk_module))
5.          print(fk_module)
```

由于前面在 PYTHONPATH 环境变量中已经添加了点(.)，因此 Python 程序总可以加载相同路径下的模块。所以，上面程序可以成功导入 fk_module 模块。

运行上面程序，可以看到如下输出结果：

```
this is fk_module
================
<class 'module'>
<module 'fk_module' from 'C:\\Users\\mengma\\Desktop\\fk_module.py'>
```

从输出结果来看，当程序导入 fk_module 时，该模块中的输出语句会在 import 时自动执行。该程序中还包含一个与模块同名的变量，该变量的类型是 module。

使用“import fk_module”导入模块的本质就是，将 fk_module.py 中的全部代码加载到内存并执行，然后将整个模块内容赋值给与模块同名的变量，该变量的类型是 module，而在该模块中定义的所有程序单元都相当于该 module 对象的成员。

下面再试试使用 from...import 语句来执行导入，例如使用如下程序来测试该模块：

```
from fk_module import name, hello
print("================")
print(name)
print(hello)
# 打印 fk_module
print(fk_module)
```

运行上面程序，可以看到如下输出结果：

```
this is fk_module
================
fkit
<function hello at 0x0000000001E7BAE8>
Traceback (most recent call last):
  File "fk_module_test2.py", line 22, in <module>
print(fk_module)
NameError: name 'fk_module' is not defined
```

从上面的输出结果可以看出，即便使用 from...import 只导入模块中部分成员，该模块中的输出语句也会在 import 时自动执行，这说明 Python 依然会加载并执行模块中的代码。

使用“from fk_module import name, hello”导入模块中成员的本质就是将 fk_module.py 中的全部代码加载到内存并执行，然后只导入指定变量、函数等成员单元，

并不会将整个模块导入，因此上面程序在输出 fk_module 时将看到错误提示：`name 'fk module' is not defined`。

在导入模块后，可以在模块文件所在目录下看到一个名为“__pycache__”的文件夹，打开该文件夹，可以看到 Python 为每个模块都生成一个 *.cpython-36.pyc 文件，比如 Python 为 fk_module 模块生成一个 fk_ module.cpython-36.pyc 文件，该文件其实是 Python 为模块编译生成的字节码，用于提升该模块的运行效率。

导入同一个模块多次，Python 只执行一次

先看一个例子，将本节开头处的程序文件修改成如下内容：

```
import fk_module
import fk_module
print("================")
# 打印 fk_module 的类型
print(type(fk_module))
print(fk_module)
```

运行结果为：

```
this is fk_module
================
<class 'module'>
<module 'fk_module' from 'C:\\Users\\mengma\\Desktop\\fk_module.py'>
```

可以看到，修改后的程序中导入了 2 次 fk_module 模块，其实完全没必要，此处导入两次只是为了说明一点：Python 很智能。虽然上面程序两次导入了 fk_module 模块，但最后运行程序，我们看到输出语句只输出一条“this is fk_module”，这说明第二次导入的 fk_module 模块并没有起作用，这就是 Python 的“智能”之处。

当程序重复导入同一个模块时，Python 只会导入一次。道理很简单，因为这些变量、函数、

类等程序单元都只需要定义一次即可，何必导入多次呢？相反，如果 Python 允许导入多次，反而可能会导致严重的后果。比如程序定义了 foo 和 bar 两个模块，假如 foo 模块导入了 bar 模块，而 bar 模块又导入了 foo 模块，这似乎形成了无限循环导入，但由于 Python 只会导入一次，所以这个无限循环导入的问题完全可以避免。

7.Python __all__变量

在默认情况下，如果使用“from 模块名 import *”这样的语句来导入模块，程序会导入该模块中所有不以下画线开头的成员（包括变量、函数和类）。但在一些场景中，我们并不希望每个成员都被暴露出来供外界使用，此时可借助于模块的 __all__ 变量，将变量的值设置成一个列表，只有该列表中的成员才会被暴露出来。

例如，下面程序定义了一个包含 __all__ 变量的模块：

```
'''测试__all__变量的模块'''
def hello():
    print("Hello, Python")
def world():
    print("Pyhton World is funny")
def test():
    print('--test--')
# 定义__all__变量，指定默认只导入 hello 和 world 两个成员
__all__ = ['hello', 'world']
```

上面的 __all__ 变量指定该模块默认只被导入 hello 和 world 两个成员。下面程序示范了模块中 __all__ 变量的用处：

```
# 导入 all_module 模块内所有成员
from all_module import *
hello()
world()
test() # 会提示找不到 test()函数
```

上面第 2 行代码使用“from all_module import *”导入了 all_module 模块下所有的成员。由于该模块包含了 __all__ 变量，因此该语句只导入 __all__ 变量所列出的成员。

运行上面程序，可以看到如下输出结果：

```
Hello, Python
Python World is funny
Traceback (most recent call last) :
  File "all_module_test.py", line 20, in <module>
    test() #会提示找不到 test()函数
NameError: name 'test' is not defined
```

从上面的输出结果可以看到，通过“from all_module import *”语句确实只能导入 __all__ 变量所列出的全部成员，没有列出的 test 就没有被导入进来。

事实上，__all__ 变量的意义在于为模块定义了一个开放的公共接口。通常来说，只有 __all__ 变量列出的成员，才是希望该模块被外界使用的成员。因此，为模块设置 __all__ 变量还是比较有用的。比如一个实际的大模块可能包含了大量其他程序不需要使用的变量、函数和类，那么通过 __all__ 变量即可把它们自动过滤掉，这还是非常酷的。

如果确实希望程序使用模块内 __all__ 列表之外的成员，有两种解决方法：

- 第一种是使用“import 模块名”来导入模块。在通过这种方式导入模块之后，总可以通过模块名前缀（如果为模块指定了别名，则可以使用模快的别名作为前缀）来调用模块内的成员。
- 第二种是使用“from 模块名 import 成员”来导入指定成员。在这种方式下，即使想导入的成员没有位于 __all__ 列表中，也依然可以导入。

8.Python 包：存放多个模块的文件夹

实际开发中，一个大型的项目往往需要使用成百上千的 Python 模块，如果将这些模块都堆放在一起，势必不好管理。而且，使用模块可以有效避免变量名或函数名重名引发的冲突，但是如果模块名重复怎么办呢？因此，Python 提出了包（Package）的概念。

什么是包呢？简单理解，包就是文件夹，只不过在该文件夹下必须存在一个名为“__init__.py” 的文件。

注意，这是 Python 2.x 的规定，而在 Python 3.x 中，__init__.py 对包来说，并不是必须的。

每个包的目录下都必须建立一个 __init__.py 的模块，可以是一个空模块，可以写一些初始化代码，其作用就是告诉 Python 要将该目录当成包来处理。

注意，__init__.py 不同于其他模块文件，此模块的模块名不是 __init__，而是它所在的包名。例如，在 settings 包中的 __init__.py 文件，其模块名就是 settings。

包是一个包含多个模块的文件夹，它的本质依然是模块，因此包中也可以包含包。例如，在前面章节中，我们安装了 numpy 模块之后可以在 Lib\site-packages 安装目录下找到名为 numpy 的文件夹，它就是安装的 numpy 模块（其实就是一个包），它所包含的内容如图 1 所示。

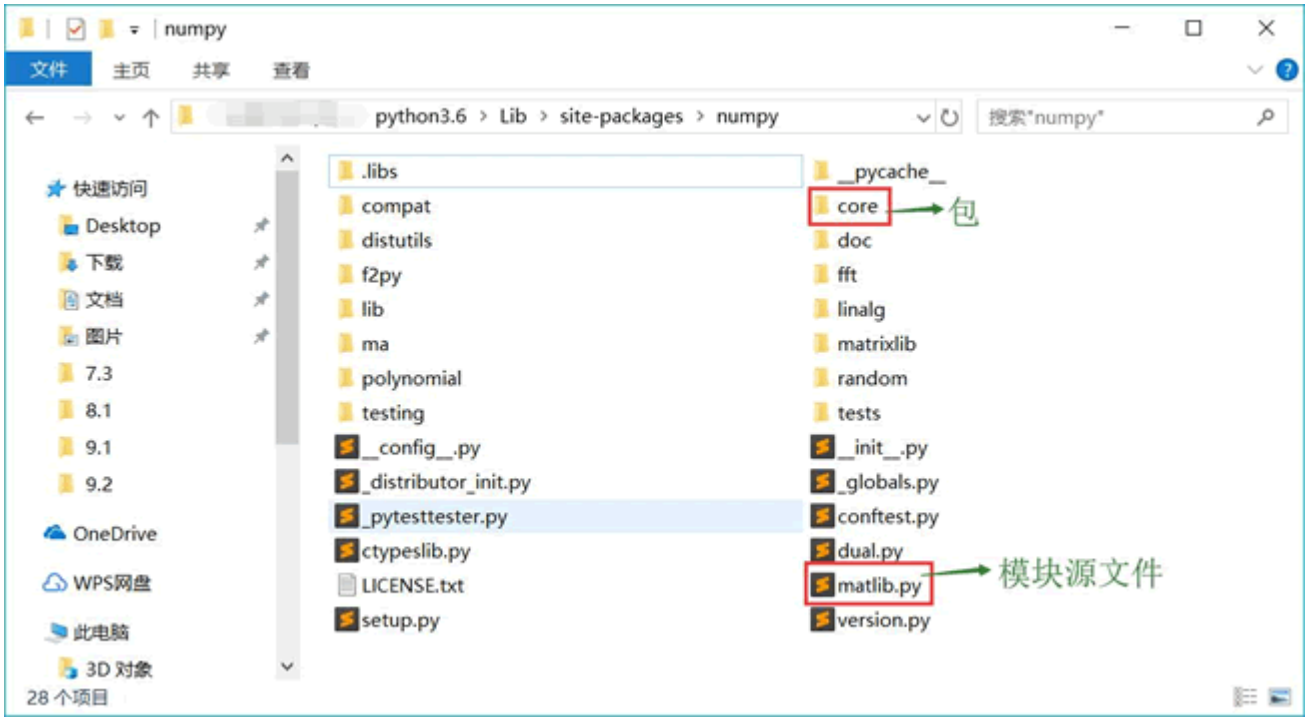


图 1 numpy 包（模块）

从图 1 可以看出，在 numpy 包（模块）中，有必须包含的 __init__.py 文件，还有 matlib.py 等模块源文件以及 core 等子包（也是模块）。这正印证了我们刚刚讲过的，包的本质依然是模块，包可以包含包。

Python 库：相比模块和包，库是一个更大的概念，例如在 Python 标准库中的每个库都有好多个包，而每个包中都有若干个模块。

9.Python 创建包，导入包

掌握了包是什么之后，接下来继续学习如何创建和导入包。

Python 包的创建

创建一个包，主要分为如下两步：

1. 创建一个文件夹，该文件夹的名字就是该包的包名。
2. 在该文件夹内创建一个名为 __init__.py 的 Python 文件，在此文件中，可以不编写任何代码，也可以编写一些 Python 初始化代码，在此文件中编写的代码，其他程序文件导入包时，会自动执行（本节后续会有实例）。

例如，现在我们创建一个非常简单的包，该包的名称为 first_package，可以按照仿照以上 2 步进行：

1. 创建一个文件夹，其名称设置为 first_package
2. 在该文件夹中添加一个 __init__.py 文件，此文件中可以不编写任何代码，不过，本例中向该文件编写如下代码：

```
'''
这是学习包的第一个示例
'''
print('this is first_package')
```

可以看到，__init__.py 文件中，包含了 2 部分信息，分别是该包的说明文档和一条简单的输出语句。

由此，就创建好了一个包。之后，就可以向包中添加模块了（也可以添加包），这里给 first_package 包添加 2 个模块，分别是 module1.py、module2.py，各自包含的代码分别如下所示（读者可直接复制下来）：

```
#module1.py 模块文件
display(n):
    print(n)

#module2.py 模块文件
class Item:
    display(self):
        print("调用模块中类的方法")
```

现在，我们就创建好了一个具有如下文件结构的包：

```
fk_package
├── __init__.py
├── module1.py
└── module2.py
```

当然，包中还有容纳其它的包，不过这里不再演示，有兴趣的读者可以自行调整包的文件结构。

Python 包的导入

再次强调，包的本质就是模块，因此导入包和导入模块的语法非常类似。无论导入我们自定义的包，还是导入从他处下载的第三方包，导入方法可归结为以下 3 种。

1. `import 包名[.模块名 [as 别名]]`
2. `from 包名 import 模块名 [as 别名]`
3. `from 包名.模块名 import 成员名 [as 别名]`

用 [] 括起来的部分，是可选部分，即在使用时可以忽略。

与模块类似，包被导入之后，会在包目录下生成一个 __pycache__ 文件夹，并在该文件夹内为包生成一个 __init__.cpython-36.pyc 文件。

1) import 包名[.模块名 [as 别名]]

以前面创建好的 first_package 包为例，导入 module1 模块并使用该模块中成员可以使用如下代码：

```
import first_package.module1
first_package.module1.display(20)
```

运行结果为：

```
this is first_package
20
```

可以看到，通过此语法格式导入包中的指定模块后，在使用该模块中的成员（变量、函数、类）时，需添加“包名.模块名”为前缀。当然，如果使用 as 给包名.模块名”起一个别名的话，就使用直接使用这个别名作为前缀使用该模块中的方法了，例如：

```
import first_package.module1 as module
module.display(20)
```

运行该程序会发现，其结果和上面完全一样。

另外，从以上输出结果可以看出，当导入指定包时，程序会自动执行该包所对应文件夹下的 __init__.py 文件中的代码。需要注意的是，当使用此语法格式仅导入包名时，例如：

```
import first_package
first_package.module1.display(20)
```

直接导入包名，并不会将包中所有模块全部导入到程序中，它的作用仅仅是导入并执行包下的 __init__.py 文件，因此，运行该程序，在执行 __init__.py 文件中代码的同时，还会抛出 AttributeError 异常（访问的对象不存在）：

```
this is first_package
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 2, in <module>
    first_package.module1.display(20)
AttributeError: module 'first_package' has no attribute 'module1'
```

我们知道，包的本质就是模块，导入模块时，当前程序中会包含一个和模块名同名且类型为 module 的变量，导入包也是如此：

```
# 导入 first_package 包（模块）
import first_package
print('==========')
print(first_package.__doc__)
print(type(first_package))
print(first_package)
```

运行结果为：

```
this is first_package

==========


这是学习包的第一个示例


<class 'module'>

<module 'first_package' from

'C:\\Users\\mengma\\Desktop\\first_package\\__init__.py'>
```

2) from 包名 import 模块名 [as 别名]

仍以导入 first_package 包中的 module1 模块为例，使用此语法格式的实现代码如下：

```
1.          from first_package import module1
2.          module1.display(20)
```

运行结果为：

```
this is first_package

20
```

可以看到，使用此语法格式导入包中模块后，在使用其成员时不需要带包名前缀，但需要带模块名前缀。

当然，我们也可以使用 as 为导入的指定模块定义别名，例如：

```
1.          from first_package import module1 as module
2.          module.display(20)
```

此程序的输出结果和上面程序完全相同。

同样，既然包也是模块，那么这种语法格式自然也支持 `from 包名 import *` 这种写法，它和 import 包名 的作用一样，都只是将该包的 __init__.py 文件导入并执行。

3) from 包名.模块名 import 成员名 [as 别名]

此语法格式用于向程序中导入“包.模块”中的指定成员（变量、函数或类）。通过该方式导入的变量（函数、类），在使用时可以直接使用变量名（函数名、类名）调用，例如：

```
1.          from first_package.module1 import display
2.          display(20)
```

运行结果为：

```
this is first_package
20
```

当然，也可以使用 as 为导入的成员起一个别名，例如：

```
1.          from first_package.module1 import display as a
2.          a(20)
```

另外，在使用此种语法格式加载指定包的指定模块时，可以使用 * 代替成员名，表示加载该模块下的所有成员。例如：

```
1.          from first_package.module1 import *
2.          display(20)
```

10.Python __init__.py 的作用

前面章节中，已经对包的创建和导入进行了详细讲解，并提供了大量的实例，这些实例虽然可以正常运行，但存在一个通病，即为了调用包内模块的成员（变量、函数或者类），代码中包含了诸多的 import 导入语句，非常繁琐。

要解决这个问题，就需要搞明白包内 __init__.py 文件的作用和用法。

我们知道，导入包就等同于导入该包中的 __init__.py 文件，因此完全可以在 __init__.py 文件中直接编写实现模块功能的变量、函数和类，但实际上并推荐大家这样做，因为包的主要作用是包含多个模块。因此 __init__.py 文件的主要作用是导入该包内的其他模块。

也就是说，通过在 __init__.py 文件使用 import 语句将必要的模块导入，这样当向其他程序中导入此包时，就可以直接导入包名，也就是使用 import 包名（或 from 包名 import *）的形式即可。

上节中，我们已经创建好的 first_package 包，该包名包含 module1 模块、module2 模块和 __init__.py 文件。现在向 first_package 包的 __init__.py 文件中编写如下代码：

```
# 从当前包导入 module1 模块
from . import module1
#from .module1 import *
# 从当前包导入 module2 模块
#from . import module2
from .module2 import *
```

可以看到，在 __init__.py 文件中用点（.）来表示当前包的包名，除此之外，from import 语句的用法和在程序中导入包的用法完全相同。

有关 from...import 语句的用法，可阅读《Python 创建包，导入包》一节详细了解。

总的来说，__init__.py 文件是通过如下 2 种方式来导入包中模块的：

```
# 从当前包导入指定模块
from . import 模块名
# 从.模块名 导入所有成员到包中
from .模块名 import *
```

第 1 种方式用于导入当前包（模块）中的指定模块，这样即可在包中使用该模块。当在其他程序使用模块内的成员时，需要添加“包名.模块名”作为前缀，例如：

```
import first_package
first_package.module1.display(20)
```

运行结果为：

```
20
```

第 2 种方式表示从指定模块中导入所有成员，采用这种导入方式，在其他程序中使用该模块的成员时，只要使用包名作为前缀即可。例如如下程序：

```
import first_package
item = first_package.Item()
item.display()
```

运行结果为：

```
调用模块中类的方法
```

11. Python 查看模块（变量、函数、类）方法

在导入模块之后，开发者往往需要了解模块包含哪些功能，比如包含哪些变量、哪些函数、哪些类等，还希望能查看模块中各成员的帮助信息，掌握这些信息才能正常地使用该模块。

为了查看模块包含什么，可以通过如下两种方式：

1. 使用 dir() 函数。
2. 使用模块本身提供的 __all__ 变量。

前面章节已经介绍过 dir() 函数的基本用法 ,该函数可用于返回模块或类所包含的全部成员（包括变量、函数、类和方法等），但直接使用 dir() 函数默认会列出模块内所有的成员，包括以下画线开头的成员，而这些以下画线开头的成员其实并不希望被外界使用。

比如在 Python 的交互式解释器中执行如下命令来导入 string 模块（Python 内置的用于丰富字符串功能的模块）：

```
>>> import string
```

然后通过 dir() 函数来查看该模块的内容，将可以看到如下输出结果：

```
>>> dir(string)
['Formatter', 'Template', '_ChainMap', '_TemplateMetaclass', '__all__', '__builtins__',
'__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
'_re', '_string', 'ascii_letters', 'ascii_lowercase', 'ascii_uppercase', 'capwords', 'digits',
'hexdigits', 'octdigits', 'printable', 'punctuation', 'whitespace']
```

很明显，该模块内有大量以下画线开头的成员，其实这些成员并不希望被其他程序使用，因此列出这些成员意义不大。

为了过滤这些以下画线开头的成员，我们可以使用如下列表推导式来列出模块中的成员：

```
>>> [e for e in dir(string) if not e.startswith('_')]
['Formatter', 'Template', 'ascii_letters', 'ascii_lowercase', 'ascii_uppercase',
'capwords', 'digits', 'hexdigits', 'octdigits', 'printable', 'punctuation', 'whitespace']
```

此命令使用 for 循环的列表推导式列出了 dir(string) 返回的所有不以下画线开头的成员，它们才是该模块希望被其他程序使用的成员。

此外，本章前面还介绍过模块中的 __all__ 变量，该变量相当于该模块开放的功能接口，因此也可通过该模块的 __all__ 变量来查看模块内的成员。例如，在交互式解释器中输入如下命令：

```
>>> string.__all__
['ascii_letters', 'ascii_lowercase', 'ascii_uppercase', 'capwords', 'digits', 'hexdigits',
'octdigits', 'printable', 'punctuation', 'whitespace', 'Formatter', 'Template']
```

对比前面列表推导式列出的结果和此处 __all__ 变量列出的结果，不难发现二者的输出结果大致相同，这说明使用这两种方式都可以查看到模块所包含的成员。

注意，并不是所有模块都会提供__all__变量的，有些模块并不提供__all__变量，在这种情况下，只能使用列表推导式来查看模块中的成员。

12.Python __doc__

前面介绍了使用 help() 函数来查看程序单元的帮助信息。比如导入 string 模块之后，即可使用 help() 函数来查看指定程序单元的帮助信息。

例如，在交互式解释器中输入如下命令来查看 string 模块下 capwords() 函数的作用：

```
>>> help(string.capwords)
Help on function capwords in module string:

capwords(s, sep=None)
    capwords(s [,sep]) -> string

    Split the argument into words using split, capitalize each
    word using capitalize, and join the capitalized words using
    join.  If the optional second argument sep is absent or None,
    runs of whitespace characters are replaced by a single space
    and leading and trailing whitespace are removed, otherwise
    sep is used to split and join the words.
```

通过上面描述可以看到，capwords() 函数的作用就是将给定的 s 字符串中每个单词的首字母变成大写的。该函数可通过 sep 参数指定分隔符：如果不指定 sep 参数，该字符串默认

以空白作为分隔符。

在查看了帮助信息之后，接下来通过如下命令来测试 string.capwords() 函数的用法：

```
>>>string.capwords('abc xyz')

'Abc Xyz'

>>> string.capwords('abc;xyz',sep=';')

'Abc;Xyz'
```

上面代码在第一次使用 capwords() 函数时，没有指定 sep 参数，因此默认以空格为分隔符，这意味着程序将 abc xyz 分成 abc 和 xyz 两个单词，因此该函数将 a、x 两个字母变成大写的：在第二次使用 capwords() 函数时，指定 sep 参数为“;”，这意味着以“,”为分隔符将 abc;xyz 分成 abc 和 xyz 两个单词，因此程序将 a、x 两个字母变成大写的。

需要说明的是，使用 help() 函数之所以能查看到程序单元的帮助信息，其实完全是因为该程序单元本身有文档信息，也就是有 __doc__ 属性。换句话说，使用 help() 函数查看的其实就是程序单元的 __doc__ 属性值。

例如，使用 print(string.capwords.__doc__) 命令来查看 capwords() 的帮助信息，将会看到如下输出结果：

```
>>> print(string.capwords.__doc__)

capwords(s [,sep]) -> string


    Split the argument into words using split, capitalize each

    word using capitalize, and join the capitalized words using

    join.  If the optional second argument sep is absent or None,

    runs of whitespace characters are replaced by a single space
```

```
    and leading and trailing whitespace are removed, otherwise

    sep is used to split and join the words
```

对比 help(string.capwords) 和 print(string.capwords.__doc__) 两个命令的输出结果，不难看到它们的输出结果完全相同，这说明使用 help() 函数查看的就是程序单元的 __doc__ 属性值。

从理论上说，应该为每个程序单元都编写完备而详细的文档信息，这样开发者只要通过 help() 函数即可查看该程序单元的文档信息，完全不需要查看文档。但不得不说的是，有些程序单元的文档信息并不是很详细，此时可能需要借助于 Python 库的参考文档：https://docs.python.org/3/library/index.html.

13.Python __file__

除可以查看模块的帮助信息之外，还可以直接阅读模块的源代码来掌握模块功能，提升 Python 编程能力。

不管学习哪种编程语言，认真阅读那些优秀的框架、库的源代码都是非常好的学习方法。

通过模块的 __file__ 属性即可查看到指定模块的源文件路径。例如，在交互式解释器中输入 string.__file__ 即可看到如下输出结果：

```
>>> import string

>>> string.__file__

'D:\\python3.6\\lib\\string.py'
```

这说明 string 模块对应的文件就是保存在 D:\Python\Python36\lib\ 目录下的 string.py 文件，开发者完全可以直接打开该文件来查看该模块的全部源代码。

需要说明的是，并不是所有模块都是使用 Python 语言编写的，有些与底层交互的模块可能是用 C 语言编写的，而且是 C 程序编译之后的效果，因此这种模块可能没有 file 属性。

14.Python 第三方库（模块）下载和安装

进行 Python 程序开发时，除了使用 Python 内置的标准模块以及我们自定义的模块之外，还有很多第三方模块可以使用，这些第三方模块可以借助 Python 官方提供的查找包页面（https://pypi.org/）找到。

使用第三方模块之前，需要先下载并安装该模块，然后就能像使用标准模块和自定义模块那样导入并使用了。因此，本节主要讲解如何下载并安装第三方模块。

下载和安装第三方模块，可以使用 Python 提供的 pip 命令实现。pip 命令的语法格式如下：

```
pip install|uninstall|list 模块名
```

其中，install、uninstall、list 是常用的命令参数，各自的含义为：

1. install：用于安装第三方模块，当 pip 使用 install 作为参数时，后面的模块名不能省略。
2. uninstall：用于卸载已经安装的第三方模块，选择 uninstall 作为参数时，后面的模块名也不能省略。
3. list：用于显示已经安装的第三方模块。

以安装 numpy 模块为例（该模块用于进行科学计算），可以在命令行窗口中输入以下代码：

```
pip install numpy
```

执行此代码，它会在线自动安装 numpy 模块。安装完成后，将显示图 1 所示的结果：

图 1 numpy 模块安装成功示意图

pip 命令会将下载完成的第三方模块，默认安装到 Python 安装目录中的 \Lib\site-packages 目录下。打开此目录，你就会发现 numpy 包，也就是 numpy 文件夹，如图 2 所示。

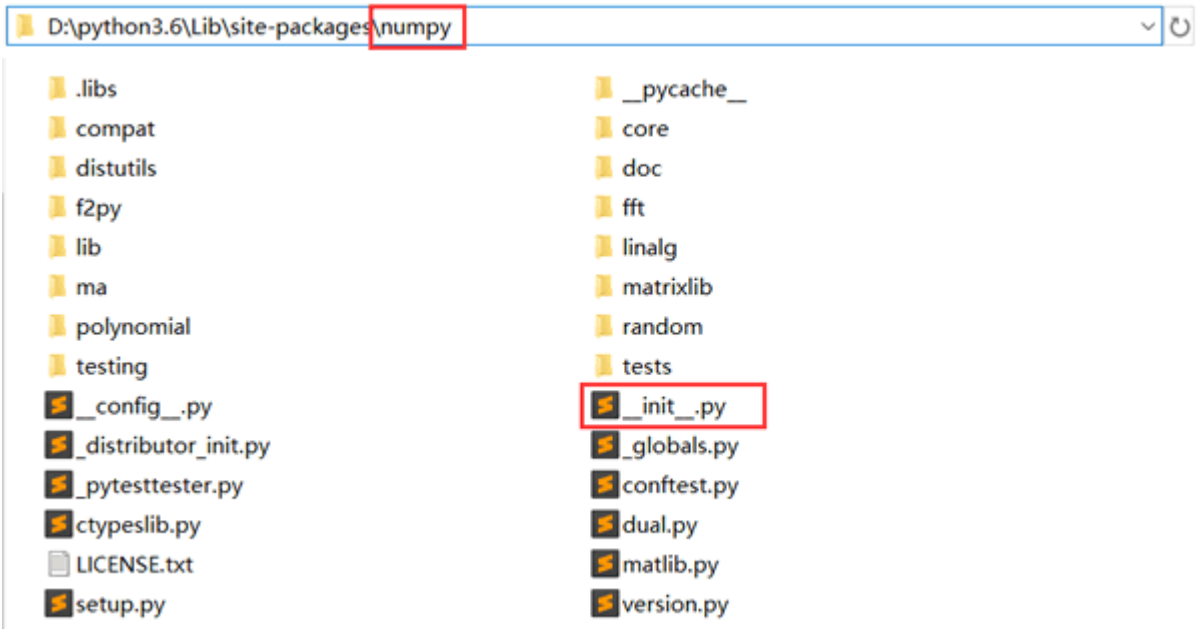


图 2 numpy 包内容示意图

前面讲过，对于向程序中导入模块，\Lib\site-packages 目录是 Python 肯定会搜索的目录，因此位于此目录的模块，可以直接使用 import 语句引入，例如：

纯文本复制

```
#直接导入 numpy 模块即可
import numpy as nu
#用 numpy 模块中的开发函数
print(nu.sqrt(16))
```

运行结果为：

```
4.0
```

通过 pip 命令，我们可以下载并安装很多第三方模块，如果想要查看 Python 中目前有哪些模块（包括标准模块和第三方模块），可以在 IDLE 中输入以下命令：

```
help('modules')
```

在此基础上，如果只是想要查看已经安装的第三方模块，可以在使用如下命令：

```
pip list
```

提示：在大型程序中，往往需要导入很多模块，建议初学者在导入模块时，优先导入 Python 提供的标准模块，然后再导入第三方模块，最后导入自定义模块。

第 12 章 Python 常见模块

第 9 章介绍了 Python 模块的相关知识，读者已经掌握了如何自定义模块。但在实际开发中，Python 的很多功能都已有了成熟的第三方实现，一般不需要开发者“重复造轮子”，当开发者需要完成某种功能时，通过搜索引擎进行搜索，通常就可以找到第三方在 Python 中为该功能所扩展的模块。

实际上，Python 语言本身也内置了大量模块，对于常规的日期、时间、正则表达式、JSON 支持、容器类等，Python 内置的模块已经非常完备，而本章就将带着读者来熟悉 Python 自带的这些模块。

需要说明的是，Python 内置的模块总是在不断的更新中，阅读本章内容只是掌握 Python 内置模块的入门之路，关于更详细、更完备的模块介绍文档可参考 Python 库的参考手册：https://docs.python.org/3/library/index.html。

1.Python sys

sys 是一个和 Python 解释器关系密切的标准库，它和帮助我们访问和 Python 解释器联

系紧密的变量和函数。

在 Python 的交互式解释器中，先导入 sys 模块，然后输入 `[e for e in dir(sys) if not e.startswith('_')]` 命令（sys 模块没有 __all__ 变量），可以看到如下输出结果：

```
>>> [e for e in dir(sys) if not e.startswith('_')]
['api_version', 'argv', 'base_exec_prefix', 'base_prefix', 'builtin_module_names',
'byteorder', 'call_tracing', 'callstats', 'copyright', 'displayhook', 'dllhandle',
'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix', 'executable', 'exit',
'flags', 'float_info', 'float_repr_style', 'get_asyncgen_hooks',
'get_coroutine_wrapper', 'getallocatedblocks', 'getcheckinterval',
'getdefaultencoding', 'getfilesystemencodeerrors', 'getfilesystemencoding',
'getprofile', 'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
'gettrace', 'getwindowsversion', 'hash_info', 'hexversion', 'implementation',
'int_info', 'intern', 'is_finalizing', 'last_traceback', 'last_type', 'last_value', 'maxsize',
'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache',
'platform', 'prefix', 'set_asyncgen_hooks', 'set_coroutine_wrapper',
'setcheckinterval', 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace',
'stderr', 'stdin', 'stdout', 'thread_info', 'version', 'version_info', 'warnoptions',
'winver']
```

上面列出的就是 sys 模块所包含的全部成员（包括变量、函数等），读者不要被它们吓着了，以为这些全都需要记下来。实际上完全没有必要，通常都是用到哪些模块就去查阅其对应的

说明文档和参考手册。

sys 模块的参考页面为 https://docs.python.org/3/library/sys.html。

需要说明的是，大部分时候用不到 sys 模块里很冷僻的功能，因此本节只介绍 sys 模块中常用的属性和函数（如表 1 所示）。

表 1 sys 模块常用成员及功能

sys 成员（变量和函数）	功能描述
sys.argv	获取运行 Python 程序的命令行参数。其中 sys.argv[0] 通常就是指该 Python 程序，sys.argv[1] 代表为 Python 程序提供的第一个参数，sys.argv[2] 代表为 Python 程序提供的第二个参数……依此类推。
sys.path	是一个字符串列表，其中每个字符串都是一个目录名，在使用 import 语句导入模块时，解释器就会从这些目录中查找指定的模块。
sys.exit()	通过引发 SystemExit 异常来退出程序。将其放在 try 块中不能阻止 finally 块的执行。你可以提供一个整数作为参数（默认为 0 ，标识成功），用来标识程序是否成功运行，这是 UNIX 的一个惯例。
sys.modules	返回模块名和载入模块对应关系的字典。

sys.platform	此变量是一个字符串，标识解释器当前正在运行的平台名称，它可能是标识操作系统的名称，也可能是标识其他种类的平台，如果运行 Jython 的话，就是 Java 虚拟机。
sys.stdin 、 sys.stdout 、 sys.stderr	这三个模块变量是类文件流对象，分别表示标准输入、标准输出和标准错误。简单理解，Python 利用 sys.stdin 获得输入，利用 sys.stdout 输出。
sys.flags	该只读属性返回运行 Python 命令时指定的旗标。
sys.getfilesystemencoding()	返回在当前系统中保存文件所用的字符集。
sys.getrefcount(object)	返回指定对象的引用计数。前面介绍过，当 object 对象的引用计数为 0 时，系统会回收该对象。
sys.getrecursionlimit()	返回 Python 解释器当前支持的递归深度。该属性可通过 setrecursionlimit() 方法重新设置。
sys.getswitchinterval()	返回在当前 Python 解释器中线程切换的时间间隔。该属性可通过 setswitchinterval() 函数改变。
sys.implementation	返回当前 Python 解释器的实现。
sys.maxsize	返回 Python 整数支持的最大值。在 32 位平台上，该属性值为 2**31-1 ;在 64 位平台上 ,该属性值为 2**63-1。
sys.executable	该属性返回 Python 解释器在磁盘上的存储路径。
sys.byteorder	显示本地字节序的指示符。如果本地字节序是大端模式，

	则该属性返回 big；否则返回 little。
sys.copyright	该属性返回与 Python 解释器有关的版权信息。
sys.version	返回当前 Python 解释器的版本信息。
sys.winver	返回当前 Python 解释器的主版本号。

下面程序示范了使用 sys 模块的部分功能：

```
import sys
# 显示本地字节序的指示符。
print(sys.byteorder)
# 显示 Python 解释器有关的版权信息
print(sys.copyright)
# 显示 Python 解释器在磁盘上的存储路径。
print(sys.executable)
# 显示当前系统上保存文件所用的字符集。
print(sys.getfilesystemencoding())
# 显示 Python 整数支持的最大值
print(sys.maxsize)
# 显示 Python 解释器所在平台
print(sys.platform)
# 显示当前 Python 解释器的版本信息。
print(sys.version)
# 返回当前 Python 解释器的主版本号。
print(sys.winver)
```

上面程序分别调用了 sys 模块的部分属性和函数。运行该程序，可以看到如下输出结果：

```
little

Copyright (c) 2001-2017 Python Software Foundation.
```

```
All Rights Reserved.

Copyright (c) 2000 BeOpen.com.

All Rights Reserved.


Copyright (c) 1995-2001 Corporation for National Research Initiatives.

All Rights Reserved.


Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.

All Rights Reserved.

D:\python3.6\pythonw.exe

utf-8

9223372036854775807

win32

3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)]

3.6
```

从上面的输出结果可以看出，Windows 7 系统（作者使用的）的字节序是小端模式，将 Python 解释器保存在 `D:\python3.6\pythonw.exe` 处，当前 Python 版本是 3.6.2。

sys 获取运行参数

通过 sys 模块的 argv 属性可获取运行 Python 程序的命令行参数。argv 属性值是一个列表，其列表元素和运行参数的关系如图 1 所示。

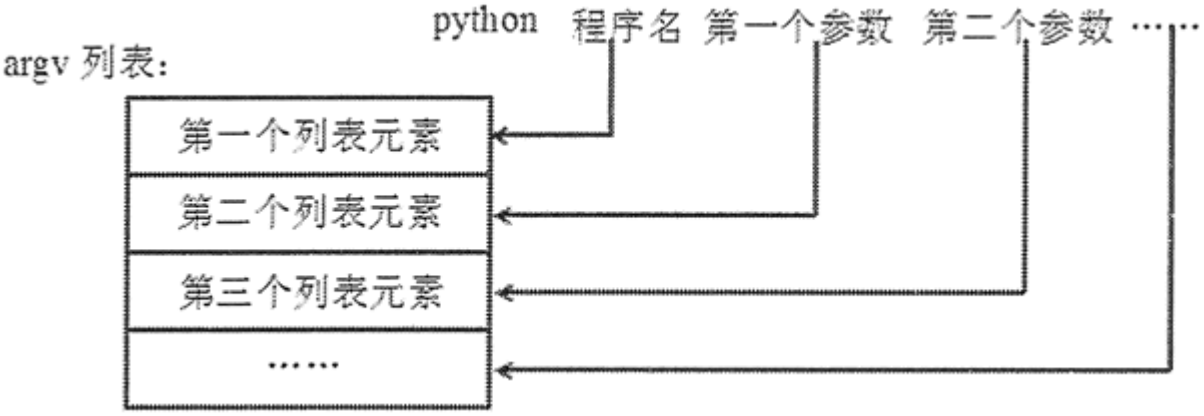


图 1 运行 Python 程序时命令行参数与 argv 列表的关系

因此，如果需要获取运行 Python 程序时传入的参数，可以通过 argv[1]、argv[2]……来获取。例如下面程序（编写在 argv_test.py 文件中）：

```
from sys import argv
# 输出 argv 列表的长度
print(len(argv))
# 遍历 argv 列表的每个元素
for arg in argv:
    print(arg)
```

上面程序是最简单的“Hello World”级的程序，只是这个程序增加了输出 argv 列表的长度、遍历 argv 列表元素的代码。使用“python argv_test.py”命令运行上面程序，可以看到如下输出结果：

```
1

argv_test.py
```

此时看到 argv 列表的长度为 1，argv 的第一个元素就是被运行的 Python 程序。

如果改为使用如下命令来运行该程序：

```
python argv_test.py Python Swift
```

可以看到如下输出结果：

```
3
argv_test.py
Python
Swift
```

上面两次运行的结果和前面介绍的内容完全一致。

如果某个参数本身包含了空格，则应该将该参数用双引号（""）括起来；否则，Python 会把这个空格当成参数分隔符，而不是参数本身。例如，采用如下命令来运行上面程序：

```
python argv test.py "Python Swift"
```

可以看到 argv 列表的长度是 2，第一个列表元素是被运行的 Python 程序，第二个列表元素的值是“Python Swift”。

动态修改模块加载路径

前面介绍了使用 PYTHONPATH 环境变量来添加 Python 模块的加载路径，但这种方式必须预先设置好。如果需要在程序运行时动态改变 Python 模块的加载路径，则可通过

sys.path 属性来实现。

sys.path 也是很有用的一个属性，它可用于在程序运行时为 Python 动态修改模块加载路径。例如，如下程序在运行时动态指定加载 g:\fk_ext 目录下的模块：

```
import sys
# 动态添加 g:\fk_ext 路径作为模块加载路径
sys.path.append('g:\\fk_ext')
# 加载 g:\fk_ext 路径下的 hello 模块
```

```
import hello
```

为了成功运行该程序，需要在 G:\ 盘中创建 fk_ext 目录，并在该目录下添加 hello.py 模块文件。

2.Python os

os 模块代表了程序所在的操作系统，主要用于获取程序运行所在操作系统的相关信息。

在 Python 的交互式解释器中先导入 os 模块，然后输入 os.__all__ 命令（__all__ 变量代表了该模块开放的公开接口），即可看到该模块所包含的全部变量和函数。

> 开发者同样不需要完全记住这些变量和函数的含义，在需要用时可参考
>
> https://docs.python.org/3/library/os.html。

由于该模块包含的内容比较多，表 1 仅介绍 os 模块中常用的变量和函数。

表 1 os 模块常用变量和函数

os 成员（变量、函数和类）	功能描述

os.name	返回导入依赖模块的操作系统名称，通常可返回 'posix'、'nt'、'java' 等值其中之一。
os.environ	返回在当前系统上所有环境变量组成的字典。
os.sep	返回路径分隔符。
os.fsencode(filename)	该函数对类路径（path-like）的文件名进行编码。
os.fsdecode(filename)	该函数对类路径（path-like）的文件名进行解码。
os.PathLike	这是一个类，代表一个类路径（path-like）对象。
os.getenv(key, default=None)	获取指定环境变量的值。
os.getlogin()	返回当前系统的登录用户名。与该函数对应的还有 os.getuid()、os.getgroups()、os.getgid() 等函数，用于获取用户 ID、用户组、组 ID 等，这些函数通常只在 UNIX 系统上有效。
os.getpid()	获取当前进程 ID。
os.getppid()	获取当前进程的父进程 ID。
os.putenv(key, value)	该函数用于设置环境变量。
os.cpu_count()	返回当前系统的 CPU 数量。
os.pathsep	返回当前系统上多条路径之间的分隔符。一般在 Windows 系

	统上多条路径之间的分隔符是英文分号（;）；在 UNIX 及类 UNIX 系统（如 Linux、Mac os X）上多条路径之间的分隔符是英文冒号（:）。
os.linesep	返回当前系统的换行符。一般在 Windows 系统上换行符是“\r\n”：在 UNIX 系统上换行符是“\n”；在 Mac os X 系统上换行符是“\r”。
os.urandom(size)	返回适合作为加密使用的、最多由 N 个字节组成的 bytes 对象。该函数通过操作系统特定的随机性来源返回随机字节，该随机字节通常是不可预测的，因此适用于绝大部分加密场景。

下面程序示范了 os 模块的大部分函数的用法：

```
import os
# 显示导入依赖模块的操作系统的名称
print(os.name)
# 获取 PYTHONPATH 环境变量的值
print(os.getenv('PYTHONPATH'))
# 返回当前系统的登录用户名
print(os.getlogin())
# 获取当前进程 ID
print(os.getpid())
# 获取当前进程的父进程 ID
print(os.getppid())
# 返回当前系统的 CPU 数量
print(os.cpu_count())
# 返回路径分隔符
print(os.sep)
# 返回当前系统的路径分隔符
```

```
17.         print(os.pathsep)
18.         # 返回当前系统的换行符
19.         print(os.linesep)
20.         # 返回适合作为加密使用的、最多 3 个字节组成的 bytes
21.         print(os.urandom(3))
```

运行上面程序，可以看到如下输出结果：

```
nt
None
yeeku
9904
12036
8
\
;



b'\x12\x1e\xcf'
```

从上面的输出结果可以看出，在 Windows 系统上 Python 导入依赖模块的操作系统名称为“nt”；当前系统的登录用户名是“yeeku”：当前进程 ID 为“9904”；当前进程的父进程 ID 为“12036”；当前系统上有 8 个 CPU；当前系统（Windows）的路径分隔符是“\”；当前系统（Windows）上多条路径之间的分隔符是分号（;）；但在当前系统（Windows）上换行符不能明显看到，这是因为当在控制台输出“\r\n” 时才会产生两个

空行。

此外，在 os 模块下还包含各种进程管理函数，它们可用于启动新进程、中止己有进程等。在 os 模块下与进程管理相关的函数如表 2 所示。

表 2 os 进程管理函数

os 进程管理函数	功能描述
os.system(command)	运行操作系统上的指定命令。
os.abort()	生成一个 SIGABRT 信号给当前进程。在 UNIX 系统上，默认行为是生成内核转储；在 Windows 系统上，进程立即返回退出代码 3。
os.execl(path, arg0, arg1, ...)	该函数还有一系列功能类似的函数，比如 os.execle()、os.execlp() 等，这些函数都是使用参数列表 arg0, arg1,...来执行 path 所代表的执行文件的。由于 os.exec*() 函数都是 PosIX 系统的直接映射，因此如采使用该命令来执行 Python 程序，传入的 arg0 参数没有什么作用。os._exit(n) 用于强制退出 Python 解释器。将其放在 try 决中可以阻止 finally 块的执行。
os.forkpty()	fork 一个子进程。
os.kill(pid, sig)	将 sig 信号发送到 pid 对应的过程，用于结束该进程。

os.killpg(pgid, sig)	将 sig 信号发送到 pgid 对应的进程组。
os.popen(cmd, mode='r', buffering=-1)	用于向 cmd 命令打开读写管道(当 mode 为 r 时为只读管道 ,当 mode 为 rw 时为读写管道) ,buffering 缓冲参数与内置的 open() 函数有相同的含义。该函数返回的文件对象用于读写字符串，而不是字节。
os.spawnl(mode, path, ...)	该函数还有一系列功能类似的函数，比如 os.spawnle()、os.spawnlp() 等，这些函数都用于在新进程中执行新程序。
os.startfile(path[,operation])	对指定文件使用该文件关联的工具执行 operation 对应的操作。如果不指定 operation 操作，则默认执行打开（ open ）操作。operation 参数必须是有效的命令行操作项目，比如 open (打开)、edit (编辑)、print (打印) 等。

下面程序示范了在 os 模块中与进程管理相关的函数的功能：

```
1.          import os
2.          # 运行平台上的 cmd 命令
3.          os.system('cmd')
4.          # 使用 Excel 打开 g:\abc.xls 文件
5.          os.startfile('g:\\abc.xls')
6.          os.spawnl(os.P_NOWAIT, 'E:\\Tools\\编辑工具
\\Notepad++.7.5.6.bin.x64\\notepad++.exe', ' ')
7.          # 使用 python 命令执行 os_test.py 程序
8.          os.execl("D:\\Python\\Python36\\python.exe", " ", 'os_test.py', 'i')
```

如果直接运行上面程序，可以看到程序运行后使用 Excel 打开了 abe.xls 文件，也打开了 Notepad++ 工具，还使用 python 命令运行了 os_test.py 文件。但如果将程序中粗体字代码取消注释，将看到程序运行后只是启动了 cmd 命令行程序，这是因为使用 as.system() 函数来运行程序时，新程序所在的进程会替代原有的进程。

在使用 os.execl() 函数时新进程之后，也会取代原有的进程，因此上面程序将这行代码放在了最后。

不仅如此，在 os 模块下还包含大量操作文件和目录的功能函数，本教程将会在后续章节专门介绍这些功能函数。

3.Python random

random 模块包括返回随机数的函数，可以用于模拟或者任何产生随机输出的程序。

事实上，所产生的随机数都是伪随机数，换句话说，这些随机数看似随机，但实际上，它们是以一个可预测的系统作为基础。不过由于这个系统模块在伪装随机方面十分优秀，因此也就不必对此过多担心啦。

在 Python 的交互式解释器中先导入 random 模块，然后输入 random.__all__ 命令（__all__ 变量代表了该模块开放的公开接口），即可看到该模块所包含的全部变量和函数：

```
>>> random.__all__
['Random', 'seed', 'random', 'uniform', 'randint', 'choice', 'sample', 'randrange',
'shuffle', 'normalvariate', 'lognormvariate', 'expovariate', 'vonmisesvariate',
'gammavariate', 'triangular', 'gauss', 'betavariate', 'paretovariate', 'weibullvariate',
'getstate', 'setstate', 'getrandbits', 'choices', 'SystemRandom']
```

开发者同样不需要完全记住这些属性和函数的含义，在使用时可参考

https://docs.python.org/3/library/random.html。

在 random 模块下提供了表 1 所示的常用函数。

表 1 random 模块常用函数

random 模块常用函数	功能描述
random.seed(a=None, version=2)	指定种子来初始化伪随机数生成器。
random.randrange(start, stop[, step])	返回从 start 开始到 stop 结束、步长为 step 的随机数。其实就相当于 choice(range(start, stop, step)) 的效果，只不过实际底层并不生成区间对象。
random.randint(a, b)	生成一个范围为 a≤N≤b 的随机数。其等同于 randrange(a, b+1) 的效果。
random.choice(seq)	从 seq 中随机抽取一个元素，如果 seq 为空，则引发 IndexError 异常。
random.choices(seq, weights=None, cum_weights=None, k=1)	从 seq 序列中抽取 k 个元素，还可通过 weights 指定各元素被抽取的权重（代表被抽取的可能性高低）。
random.shuffle(x[, random])	对 x 序列执行洗牌“随机排列”操作。

random.sample(population, k)	从 population 序列中随机抽取 k 个独立的元素。
random.random()	生成一个从 0.0（包含）到 1.0（不包含）之间的伪随机浮点数。
random.uniform(a, b)	生成一个范围为 a≤N≤b 的随机数。
random.expovariate(lambd)	生成呈指数分布的随机数。其中 lambd 参数(其实应该是 lambda，只是 lambda 是 Python 关键字，所以简写成 lambd）为 1 除以期望平均值。如果 lambd 是正值，则返回的随机数是从 0 到正无穷大；如果 lambd 为负值，则返回的随机数是从负无穷大到 0。

关于生成伪随机浮点数的函数，Python 还提供了 random.triangular(low, high, mode)、random.gauss(mu, sigma) 等，它们用于生成呈对称分布、高斯分布的随机数。由于这种随机数需要一些数学知识来理解它们，而且平时开发并不常用，因此这里就不深入展开介绍了。

下面程序示范了 random 模块中常见函数的功能和用法：

```
1.          import random
2.          #生成范围为 0.0≤x<1.0 的伪随机浮点数
3.          print (random.random())
4.          #生成范围为 2.5≤x<10.0 的伪随机浮点数
5.          print (random.uniform(2.5, 10.0))
6.          #生成呈指数分布的伪随机浮点数
```

```
print (random.expovariate(1/5))
#生成从 0 到 9 的伪随机整数
print(random.randrange(10))
#生成从 0 到 100 的随机偶数
print (random.randrange(0, 101 , 2))
#随机抽取一个元素
print (random.choice (['Python','Swift','Kotlin']))
book_list = ['Python','Swift','Kotlin']
#对列表元素进行随机排列
random.shuffle (book_list)
print (book_list)
#随机抽取 4 个独立的元素
print (random.sample([10, 20 , 30 , 40 , 50], k=4))
```

运行上面程序，可以看到如下输出结果：

```
0.7097266660597923
6.4845555071809535
9.707899380000603
2
70
Python
['Swift', 'Python', 'Kotlin']
[20, 10, 30, 50]
```

实际上，使用 random 模块中的随机函数可以做很多很有趣的事情。比如下面程序：

纯文本复制

```
import random
import collections
#指定随机抽取 6 个元素，各元素被抽取的权重（概率）不同
```

```
print(random.choices(['Python','Swift','Kotlin'], [5, 5, 1], k=6))
#下面模拟从 52 张扑克牌中抽取 20 张
#在被拍到的 20 张牌中，牌面为 10（包括 J、Q、K 的的牌占多大比例
#生成一个 16 个 tens（代表 10）和 36 个 low_cards（代表其他牌）的集合
deck= collections.Counter(tens=16, low_cards=36)
#从 52 张牌中随机抽取 20 张
seen= random.sample(list(deck.elements()), k=20)
#统计 tens 元素有多少个，再除以 20
print(seen.count('tens')/20)
```

运行上面程序，可以看到如下输出结果：

```
['Swift', 'Swift', 'Swift', 'Swift', 'Python', 'Swift']
0.45
```

从上面的输出结果来看，在第一次抽取的 6 个元素中 Kotlin 完全没有被抽取到，这是因为它的被抽取比例太低了。

4.Python time

time 模块主要包含各种提供日期、时间功能的类和函数。该模块既提供了把日期、时间格式化为字符串的功能，也提供了从字符串恢复日期、时间的功能。

在 Python 的交互式解释器中先导入 time 模块，然后输入 [e for e in dir(time) if not e.startswith('_')] 命令，即可看到该模块所包含的全部属性和函数：

```
>>> [e for e in dir(time) if not e.startswith('_')]
['altzone', 'asctime', 'clock', 'ctime', 'daylight', 'get_clock_info', 'gmtime', 'localtime',
```

```
'mktime', 'monotonic', 'perf_counter', 'process_time', 'sleep', 'strftime', 'strptime',
'struct_time', 'time', 'timezone', 'tzname']
```

在 time 模块内提供了一个 time.struct_time 类，该类代表一个时间对象，它主要包含 9 个属性，每个属性的信息如表 1 所示：

表 1 time.struct_time 类中各属性的含义

字段名	字段含义	值
tm_year	年	如 2017、2018 等
tm_mon	月	如 2、3 等，范围为 1~12
tm_mday	日	如 2、3 等，范围为 1~31
tm_hour	时	如 2、3 等，范围为 0~23
tm_min	分	如 2、3 等，范围为 0~59
tm_sec	秒	如 2、3 等，范围为 0~61
tm_wday	周	周一为 0，范围为 0~6
tm_yday	一年内第几天	如 65，范围 1~366
tm_isdst	夏时令	0、1 或 -1

注意，秒的范围 0~61 是为了应付闰秒和双闰秒。

比如，Python 可以用 time.struct_time(tm_year=2018, tm_mon=5, tm_mday=2, tm_hour=8, tm_min=0, tm_sec=30, tm_wday=3, tm_yday=1, tm_isdst=0) 很清晰地

代表时间。

此外，Python 还可以用一个包含 9 个元素的元组来代表时间，该元组的 9 个元素和 struct_time 对象中 9 个属性的含义是一一对应的。比如程序可以使用（2018, 5, 2, 8, 0, 30, 3, 1, 0）来代表时间。

在日期、时间模块内常用的功能函数如表 1 所示。

表 1 time 模块常用函数

time 常用函数	功能描述
time.asctime([t])	将时间元组或 struct_time 转换为时间字符串。如果不指定参数 t，则默认转换当前时间。
time.ctime([secs])	将以秒数代表的时间（格林威治时间）转换为时间字符串。
time.gmtime([secs])	将以秒数代表的时间转换为 struct_time 对象。如果不传入参数，则使用当前时间。
time.localtime([secs])	将以秒数代表的时间转换为代表当前时间的 struct_time 对象。如果不传入参数，则使用当前时间。
time.mktime(t)	它是 localtime 的反转函数，用于将 struct_time 对象或元组代表的时间转换为从 1970 年 1 月 1 日 0 点整到现在过了多少秒。

time.perf_counter()	返回性能计数器的值。以秒为单位。
time.process_time()	返回当前进程使用 CPU 的时间，以秒为单位。
time.sleep(secs)	暂停 secs 秒，什么都不干。
time.strftime(format[, t])	将时间元组或 struct_time 对象格式化为指定格式的时间字符串。如果不指定参数 t，则默认转换当前时间。
time.strptime(string[, format])	将字符串格式的时间解析成 struct_time 对象。
time.time()	返回从 1970 年 1 月 1 日 0 点整到现在过了多少秒。
time.timezone	返回本地时区的时间偏移，以秒为单位。
time.tzname	返回本地时区的名字。

Python 可以用从 1970 年 1 月 1 日 0 点整到现在所经过的秒数来代表当前时间(又称格林威治时间)，比如我们写 30 秒，那么意味着时间是 1970 年 1 月 1 日 0 点 0 分 30 秒。但需要注意的是，在实际输出时可能会受到时区的影响，比如中国处于东八区，因此实际上会输出 1970 年 1 月 1 日 8 点 0 分 30 秒。

下面程序示范了 time 棋块的功能函数：

```
1.          import time
2.          # 将当前时间转换为时间字符串
3.          print(time.asctime())
4.          # 将指定时间转换时间字符串，时间元组的后面 3 个元素没有设置
```

```
5.          print(time.asctime((2018, 2, 4, 11, 8, 23, 0, 0 ,0))) # Mon Feb  4 11:08:23
    2018
6.          # 将以秒数为代表的时间转换为时间字符串
7.          print(time.ctime(30)) # Thu Jan  1 08:00:30 1970
8.          # 将以秒数为代表的时间转换为 struct_time 对象。
9.          print(time.gmtime(30))
10.         # 将当前时间转换为 struct_time 对象。
11.         print(time.gmtime())
12.         # 将以秒数为代表的时间转换为代表当前时间的 struct_time 对象
13.         print(time.localtime(30))
14.         # 将元组格式的时间转换为秒数代表的时间
15.         print(time.mktime((2018, 2, 4, 11, 8, 23, 0, 0 ,0))) # 1517713703.0
16.         # 返回性能计数器的值
17.         print(time.perf_counter())
18.         # 返回当前进程使用 CPU 的时间
19.         print(time.process_time())
20.         #time.sleep(10)
21.         # 将当前时间转换为指定格式的字符串
22.         print(time.strftime('%Y-%m-%d %H:%M:%S'))
23.         st = '2018 年 3 月 20 日'
24.         # 将指定时间字符串恢复成 struct_time 对象。
25.         print(time.strptime(st, '%Y 年%m 月%d 日'))
26.         # 返回从 1970 年 1970 年 1 月 1 日 0 点整到现在过了多少秒。
27.         print(time.time())
28.         # 返回本地时区的时间偏移，以秒为单位
29.         print(time.timezone) # 在国内东八区输出-28800
```

运行上面程序，可以看到如下输出结果：

```
Fri Feb 22 11:28:39 2019

Mon Feb  4 11:08:23 2018

Thu Jan  1 08:00:30 1970
```

```
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0,
tm_sec=30, tm_wday=3, tm_yday=1, tm_isdst=0)
time.struct_time(tm_year=2019, tm_mon=2, tm_mday=22, tm_hour=3, tm_min=28,
tm_sec=39, tm_wday=4, tm_yday=53, tm_isdst=0)
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=8, tm_min=0,
tm_sec=30, tm_wday=3, tm_yday=1, tm_isdst=0)
1517713703.0
0.0
0.140625
2019-02-22 11:28:39
time.struct_time(tm_year=2018, tm_mon=3, tm_mday=20, tm_hour=0, tm_min=0,
tm_sec=0, tm_wday=1, tm_yday=79, tm_isdst=-1)
1550806119.4960592
-28800
```

time 模块中的 strftime() 和 strptime() 两个函数互为逆函数，其中 strftime() 用于将 struct_time 对象或时间元组转换为时间字符串；而 strptime() 函数用于将时间字符串转换为 struct_time 对象。这两个函数都涉及编写格式模板，比如上面程序中使用 %Y 代表年、%m 代表月、%d 代表日、%H 代表时、%M 代表分、%S 代表秒。这两个函数所需要的时间格式字符串支持的指令如表 2 所示：

表 2 Python 时间格式字符串所支持的指令

指令	含义

%a	本地化的星期几的缩写名，比如 Sun 代表星期天
%A	本地化的星期几的完整名
%b	本地化的月份的缩写名，比如 Jan 代表一月
%B	本地化的月份的完整名
%c	本地化的日期和时间的表示形式
%d	代表一个月中第几天的数值，范固： 01~31
%H	代表 24 小时制的小时，范围：00~23
%I	代表 12 小时制的小时，范围：01~12
%j	一年中第几天，范围：001~366
%m	代表月份的数值，范围：01~12
%M	代表分钟的数值，范围：00~59
%p	上午或下午的本地化方式。当使用 strptime() 函数并使用 %I 指令解析小时时，%p 只影响小时字段
%S	代表分钟的数值，范围：00~61。该范围确实是 00~61，60 在表示闰秒的时间戳时有效，而 61 则是由于一些历史原因造成的
%U	代表一年中表示第几周，以星期天为每周的第一天，范围：00~53。在这种方式下，一年中第一个星期天被认为处于第一周。当使用 strptime() 函数解析时间字符串时，只有同时指定了星期几和年份该指令才会有效

%w	代表星期几的数值，范围：0~6，其中 0 代表周日
%W	代表一年小第几周，以星期一为每周的第一天，范围：00~53。在这种方式下，一年中第一个星期一被认为处于第一周。当使用 strptime() 函数解析时间字符串时，只有同时指定了星期几和年份该指令才会有效
%x	本地化的日期的表示形式
%X	本地化的时间的表示形式
%y	年份的缩写，范围：00~99，比如 2018 年就简写成 18
%Y	年份的完整形式。如 2018
%z	显示时区偏移
%Z	时区名（如果时区不行在，则显示为空）
%%	用于代表%符号

5.Python json

JSON 的全称是 JavaScript Object Notation，即 JavaScript 对象符号，它是一种轻量级、跨平台、跨语言的数据交换格式，其设计意图是把所有事情都用设计的字符串来表示，这样既方便在互联网上传递信息，也方便人进行阅读。

最早的时候，JSON 是 JavaScript 语言的数据交换格式，后来慢慢发展成一种语言无关的数据交换格式，这一点非常类似于 XML。

JSON 主要在类似于 C 的编程语言中广泛使用，这些语言包括 C、C++、C#、Java、JavaScript、Perl、Python 等。JSON 提供了多种语言之间完成数据交换的能力，因此，JSON 也是一种非常理想的数据交换格式。

JSON 主要有如下两种数据结构：

- 由 key-value 对组成的数据结构。这种数据结构在不同的语言中有不同的实现。例如，在 JavaScript 中是一个对象；在 Python 中是一种 dict 对象；在 C 语言中是一个 struct；在其他语言中，则可能是 record、dictionary、hash table 等。
- 有序集合。这种数据结构在 Python 中对应于列表；在其他语言中，可能对应于 list、vector、数组和序列等。

上面两种数据结构在不同的语言中都有对应的实现，因此这种简便的数据表示方式完全可以实现跨语言。所以，JSON 可以作为程序设计语言中通用的数据交换格式。

在 JavaScript 中主要有两种 JSON 语法，其中一种用于创建对象，另一种用于创建数组。

使用 JSON 语法创建对象

使用 JSON 语法创建对象是一种更简单的方式。使用 JSON 语法可避免书写函数，也可避免使用 new 关键字，而是可以直接获取一个 JavaScript 对象。对于早期的 JavaScript 版本，如果要使用 JavaScript 创建一个对象，通常可能会这样写：

```
1.          //定义一个函数，可以作为该类的构造器
2.          function Person(name, gender)
3.          {
4.              this.name = name;
5.              this.gender = gender;
6.          }
7.          //创建一个 Person 实例
8.          var p = new Person('yeeku', 'male');
9.          //输出 Person 实例的 name 属性
10.         alert (p.name);
```

从 JavaScript 1.2 开始，创建对象有了一种更快捷的语法，如下所示：

```
1.          var p = {"name":'yeeku',"gender":'male'};
2.          alert(p);
```

这种语法就是一种 JSON 语法。显然，使用 JSON 语法创建对象更加简捷、方便。如图 1 所示是使用 JSON 创建对象的语法示意图：

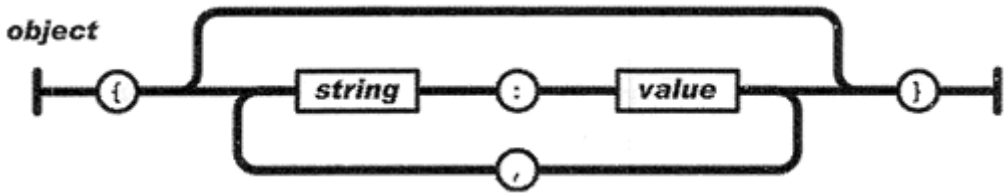


图 1 使用 JSON 创建对象的语法示意图

从图 1 可以看出，在创建对象 object 时，总以“{”开始，以“}”结束，对象的每个属性名和属性值之间以英文冒号（:）隔开，多个属性定义之间以英文逗号（,）隔开。语法格式如下：

```
object =
{
    propertyName1 : propertyValue1,
    propertyName2 : propertyValue2,
    ...
}
```

必须注意的是，并不是在每个属性定义的后面都有英文逗号，必须当后面还有属性定义时才需要有逗号。因此，下面的对象定义是错误的：

```
person =
{
    name : 'yeeku',
    gender : 'male',
}
```

因为在 gender 属性定义的后面多出了一个英文逗号。如果在最后一个属性定义的后面直接以“}”结束了，则不应该再有英文逗号。

当然，在使用 JSON 语法创建 JavaScript 对象时，属性值不仅可以是普通字符串，也可以是任何基本数据类型，还可以是函数、数组，甚至是另外一个使用 JSON 语法创建的对象。例如：

```
1.          person =
```

```
        {
            name : 'yeeku',
            gender : 'male',
            //使用 JSON 语法为其指定一个属性
            son : {
            name : 'tiger',
            grade : 1
            },
            //使用 JSON 语法为 person 直接分配一个方法
            info : function()
            {
            console.log("姓名：" + this.name + "性别：" + this.sex);
            }
        }
```

使用 JSON 语法创建数组

使用 JSON 语法创建数组也是非常常见的情形，在早期的 JavaScript 语法中，我们通过如下方式来创建数组：

```
        //创建数组对象
        var a = new Array();
        //为数组元素赋值
        a[0] = 'yeeku';
        //为数组元素赋值
        a[1] = 'nono';
```

或者，通过如下方式创建数组：

```
        //在创建数组对象时直接赋值
        var a = new Array ('yeeku', 'nono');
```

但如果使用 JSON 语法，则可以通过如下方式创建数组：

```
        //使用 JSON 语法创建数组
```

```
2.          var a = ['yeeku', 'nono'];
```

如图 2 所示是使用 JSON 创建数组的语法示意图：

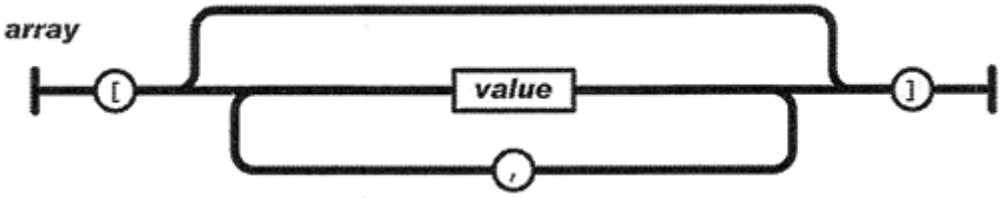


图 2 使用 JSON 创建数组的语法示意图

正如从图 2 中所看到的，使用 JSON 语法创建数组总是以英文方括号（[）开始，然后依次放入数组元素，元素与元素之间以英文逗号隔开，最后一个数组元素后面不需要英文逗号，但以英文反方括号（]）结束。使用 JSON 创建数组的语法格式如下：

```
arr= [value1, value2, ...]
```

与使用 JSON 语法创建对象相似的是，在数组的最后一个元素的后面不能有英文逗号。

鉴于 JSON 语法的简单易用，而且作为数据传输载体时，数据传输量更小，因此在跨平台的数据交换中，往往采用 JSON 作为数据交换格式。假设需要交换一个对象 person，其 name 属性为 yeeku，gender 属性为 male，age 属性为 29，使用 JSON 语法可以简单写成如下形式：

```
1.          person =
2.          {
3.              name : 'yeeku',
4.              gender : 'male',
```

```
5.                 age : 29
6.             }
```

而 Python 则提供了将符合格式的 JSON 字符串恢复成对象的函数,也提供了将对象转换成 JSON 字符串的方法。JSON 的官方站点是 http://www.json.org，读者可以登录该站点了解关于 JSON 的更多信息。

Python 的 JSON 支持

json 模块提供了对 JSON 的支持，它既包含了将 JSON 字符串恢复成 Python 对象的函数，也提供了将 Python 对象转换成 JSON 字符串的函数。

当程序把 JSON 对象或 JSON 字符串转换成 Python 对象时，从 JSON 类型到 Python 类型的转换关系如表 3 所示：

表 3 JSON 类型转换 Python 类型的对应关系

JSON 类型	Python 类型
对象（object）	字典（dict）
数组（array）	列表（list）
字符串（string）	字符串（str）
整数（number(int)）	整数（int）
实数（number(real)）	浮点数（float）

true	True
false	False
null	None

当程序把 Python 对象转换成 JSON 格式字符串时 ,从 Python 类型到 JSON 类型的转换关系如表 4 所示 :

表 4 Python 类型转换 JSON 类型的对应关系

Python 类型	**JSON 类型**
字典（dict）	对象（object）
列表（list）和元组（tuple）	数组（array）
字符串（str）	字符串（string）
整型、浮点数 ,以及整型、浮点型派生的枚举(float,int-& float-derived Enums)	数值型（number）
True	true
False	false
None	null

在 Python 的交互式解释器中先导入 json 模块，然后输入 json.all 命令，即可看到该模

块所包含的全部属性和函数：

```
>>> json.__all__
['dump', 'dumps', 'load', 'loads', 'JSONDecoder', 'JSONDecodeError',
'JSONEncoder']
```

json 模块中常用的函数和类的功能如下：

- json.dump(obj, fp, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw)：将 obj 对象转换成 JSON 字符串输出到 fp 流中，fp 是一个支持 write() 方法的类文件对象。
- json.dumps(obj, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan= True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw)：将 obj 对象转换为 JSON 字符串，并返回该 JSON 字符串。
- json.load(fp, *, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw)：从 fp 流读取 JSON 字符串，将其恢复成 JSON 对象，其中 fp 是一个支持 write() 方法的类文件对象。
- json.loads(s, *, encoding=None, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw)：将 JSON 字符串 s 恢复成 JSON 对象。

通过上面 4 个功能函数就可以实现 JSON 的两个主要应用场景，由于 JSON 只是一种轻量级的数据交换格式，因此 JSON 的主要应用场景如图 5 所示：

图 5 Python 的 JSON 支持

下面程序示范了 dumps() 和 dump() 函数的 encode 操作（将 Python 对象转换成 JSON 字符串）：

```
import json
# 将 Python 对象转 JSON 字符串（元组会当成数组）
s = json.dumps(['yeeku', {'favorite': ('coding', None, 'game', 25)}])
print(s) # ["yeeku", {"favorite": ["coding", null, "game", 25]}]
# 简单的 Python 字符串转 JSON
s2 = json.dumps("\"foo\bar")
print(s2) #"\"foo\bar"
# 简单的 Python 字符串转 JSON
s3 = json.dumps('\\')
print(s3) #"\\"
# Python 的 dict 对象转 JSON，并对 key 排序
s4 = json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True)
print(s4) #{"a": 0, "b": 0, "c": 0}
# 将 Python 列表转 JSON，
# 并指定 JSON 分隔符：逗号和冒号之后没有空格（默认有空格）
s5 = json.dumps([1, 2, 3, {'x': 5, 'y': 7}], separators=(',', ':'))
```

```
        # 输出的 JSON 字符串中逗号和冒号之后没有空格
        print(s5) # '[1,2,3,{"4":5,"6":7}]'
        # 指定 indent 为 4，意味着转换的 JSON 字符串有缩进
        s6 = json.dumps({'Python': 5, 'Kotlin': 7}, sort_keys=True, indent=4)
        print(s6)
        # 使用 JSONEncoder 的 encode 方法将 Python 转 JSON
        s7 = json.JSONEncoder().encode({"names": ("孙悟空", "齐天大圣")})
        print(s7) # {"names": ["\u5b59\u609f\u7a7a",
"\u9f50\u5929\u5927\u5723"]}
        f = open('a.json', 'w')
        # 使用 dump()函数将转换得到 JSON 字符串输出到文件
        json.dump(['Kotlin', {'Python': 'excellent'}], f)
```

上面程序主要是调用 dumps() 函数执行 encode 操作，程序在调用 dumps() 函数时指定了不同的选项。上面程序最后一行代码调用 dump() 函数将通过 encode 操作得到的 JSON 字符串输出到文件中。实际上，dumps() 和 dump() 函数的功能、所支持的选项基本相同，只是 dumps() 函数直接返回转换得到的 JSON 字符串，而 dump() 函数则将转换得到的 JSON 字符串输出到文件中。

运行上面程序，可以看到如下输出结果：

```
["yeeku", {"favorite": ["coding", null, "game", 25]}]
"\"foo\bar"
"\\"
{"a": 0, "b": 0, "c": 0}
[1,2,3,{"x":5,"y":7}]
{
    "Kotlin": 7,
```

```
    "Python": 5
}
{"names": ["\u5b59\u609f\u7a7a", "\u9f50\u5929\u5927\u5723"]}
```

程序运行结束后，会在程序所在目录生成一个 a.json 文件，该文件内容就是转换得到的 JSON 字符串。

正如从上面程序中所看到的，程序调用 json.JSONEncoder 对象的 encode() 方法也可以将 Python 对象转换为 JSON 字符串。而 dumps() 和 dump() 函数是更高级的调用方式，一般调用 dumps() 和 dump() 函数对 Python 对象执行转换即可。

下面程序示范了 loads() 和 load() 函数的 decode 操作（将 JSON 字符串转换成 Python 对象）：

```
1.          import json
2.          # 将 JSON 字符串恢复成 Python 列表
3.          result1 = json.loads('["yeeku", {"favorite": ["coding", null, "game",
    25]}]')
4.          print(result1) # ['yeeku', {'favorite': ['coding', None, 'game', 25]}]
5.          # 将 JSON 字符串恢复成 Python 字符串
6.          result2 = json.loads('"\\"foo\\"bar"')
7.          print(result2) # "foo"bar
8.          # 定义一个自定义的转化函数
9.          def as_complex(dct):
10.             if '__complex__' in dct:
11.                 return complex(dct['real'], dct['imag'])
12.             return dct
13.         # 使用自定义的恢复函数
14.         # 自定义回复函数将 real 数据转成复数的实部，将 imag 转成复数的虚部
```

```
15.         result3 = json.loads('{"__complex__": true, "real": 1, "imag": 2}',\
16.             object_hook=as_complex)
17.         print(result3) # (1+2j)
18.         f = open('a.json')
19.         # 从文件流恢复 JSON 列表
20.         result4 = json.load(f)
21.         print(result4) # ['Kotlin', {'Python': 'excellent'}]
```

上面程序开始调用 loads() 函数从 JSON 字符串恢复 Python 列表、Python 字符串等。接下来程序示范了一个比较特殊的例子，程序定义了一个自定义的恢复函数，该函数负责将一个原本应该恢复成 dict 对象的 JSON 字符串恢复成复数，并负责将字典中 real 对应的值转换成复数的实部，将字典中 imag 对应的值转换成复数的虚部。

通过使用自定义的恢复函数，可以完成 JSON 类型到 Python 特殊类型（如复数、矩阵）的转换。上面程序最后使用 load() 函数示范了从文件流来恢复 JSON 列表。

运行上面程序，可以看到如下输出结果：

```
['yeeku', {'favorite': ['coding', None, 'game', 25]}]
"foo"bar
(1+2j)
['Kotlin', {'Python': 'excellent'}]
```

此外，我们还需要考虑一个问题，Python 支持更多的 JSON 所不支持的类型，比如复数、矩阵等，如果直接使用 dumps() 或 dump() 函数进行转换，程序肯定会出问题。此时就需要开发者对 JSONEncoder 类进行扩展，通过这种扩展来完成从 Python 特殊类型到

JSON 类型的转换。

例如，如下程序示范了通过扩展 JSONEncoder 来实现从 Python 复数到 JSON 字符串的转换：

纯文本复制

```
import json
# 定义 JSONEncoder 的子类
class ComplexEncoder(json.JSONEncoder):
    def default(self, obj):
        # 如果要转换的对象是复数类型，程序负责处理
        if isinstance(obj, complex):
            return {"__complex__": 'true', 'real': obj.real, 'imag': obj.imag}
        # 对于其他类型，还使用 JSONEncoder 的默认处理
        return json.JSONEncoder.default(self, obj)
s1 = json.dumps(2 + 1j, cls=ComplexEncoder)
print(s1) # '{"__complex__": "true", "real": 2.0, "imag": 1.0}'
s2 = ComplexEncoder().encode(2 + 1j)
print(s2) # '{"__complex__": "true", "real": 2.0, "imag": 1.0}'
```

上面程序扩展了 JSONEncoder 类的子类，并重写了它的 default() 方法，在方法中判断如果要转换的目标类型是复数（complex），程序就会进行自定义转换，即将复数转换成 JSON 对象，且该对象包含 "_complex_":'true' 属性。

一旦扩展了 JSONEncoder 的子类之后，程序有两种方式来使用自定义的子类：

1. 在 dumps() 或 dump() 函数中通过 cls 属性指定使用 JSONEncoder 的自定义子类。
2. 直接使用 JSONEncoder 的自定义子类的 encode() 方法来执行转换。

运行该程序，可以看到如下输出结果：

```
{"__complex__": "true", "real": 2.0, "imag": 1.0}

{"__complex__": "true", "real": 2.0, "imag": 1.0}
```

6.Python re 正则表达式模块及其用法

正则表达式（Regular Expression）用于描述一种字符串匹配的模式，它可用于检查一个字符串是否含有某个子串，也可用于从字符串中提取匹配的子串，或者对字符串中匹配的子串执行替换操作。

很多读者都会觉得正则表达式是非常神奇、高级的知识，实际上正则表达式确实是一种非常实用的工具。正则表达式的入门并不难，任意字符串都可以被当成正则表达式来使用，例如“abc”，它也是一个正则表达式，只是它只能匹配“abc”字符串。

当然，如果正则表达式仅能匹配“abc”这样的字符串，那么正则表达式也就不值得学习了。事实上，正则表达式包含的知识点比较多，它的模式匹配能力也非常强，初学者可以由浅入深地学习。

对于 Python 开发者来说，掌握正则表达式确实是一个很重要的技能。在掌握了正则表达式之后，Python 开发者也可使用正则表达式来开发数据抓取、网络爬虫等程序。

实际上，掌握 Python 的正则表达式并不难，无非就是几个简单的函数。在 Python 的交互式解释器中先导入 re 模块，然后输入 re.__all__ 命令，即可看到该模块所包含的全部属性和函数：

```
>>> re.__all__

['match', 'fullmatch', 'search', 'sub', 'subn', 'split', 'findall', 'finditer', 'compile',

'purge', 'template', 'escape', 'error', 'A', 'I', 'L', 'M', 'S', 'X', 'U', 'ASCII', 'IGNORECASE',

'LOCALE', 'MULTILINE', 'DOTALL', 'VERBOSE', 'UNICODE']
```

从上面的输出结果可以看出，re 模块包含了为数不多的几个函数和属性（用于控制正则表达式匹配的几个选项）。下面先介绍这些函数的作用：

- re.compile(pattern, flags=0)：该函数用于将正则表达式字符串编译成 _sre.SRE_Pattern 对象，该对象代表了正则表达式编译之后在内存中的对象，它可以缓存并复用正则表达式字符串。如果程序需要多次使用同一个正则表达式字符串，则可考虑先编译它。

  该函数的 pattern 参数就是它所编译的正则表达式字符串，flags 则代表了正则表达式的匹配旗标。编译得到的 _sre.SRE_Pattern 对象包含了 re 模块中绝大部分函数对应的方法。比如下面两行代码表示先编译正则表达式，然后调用正则表达式的 search() 方法执行匹配：

```
#先编译正则表达式
p = re.compile('abc')
#调用_sre.SRE_Pattern 对象的 search()方法
p.search("www.abc.com")
```

上面两行代码和下面代码的效果基本相同：

```
#直接用正则表达式匹配目标字符串
re.search('abc','www.abc.com')
```

对于上面两种方式，由于第一种方式预编译了正则表达式，因此程序可复用 p 对象（该对象缓存了正则表达式字符串），所以具有更好的性能。

re.match(pattern, string, flags=0)：尝试从字符串的开始位置来匹配正则表达式，如果从开始位置匹配不成功，match() 函数就返回 None 。其中 pattern 参数代表正则表达式；string 代表被匹配的字符串；flags 则代表正则表达式的匹配旗标。该函数返回 _sre.SRE_Match 对象，该对象包含的 span(n) 方法用于获取第 n+1 个组的匹配位置，group(n) 方法用于获取第 n+1 个组所匹配的子串。

re.search(pattern, string, flags=0)：扫描整个字符串，并返回字符串中第一处匹配 pattern 的匹配对象。其中 pattern 参数代表正则表达式；string 代表被匹配的字符串；flags 则代表正则表达式的匹配旗标。该函数也返回 _sre.SRE_Match 对象。

根据上面介绍不难发现，match() 与 search() 的区别在于，match() 必须从字符串开始处就

匹配，但 search() 可以搜索整个字符串。例如如下程序：

```
import re
m1 = re.match('www', 'www.fkit.org')# 开始位置可以匹配
print(m1.span())  # span 返回匹配的位置
print(m1.group()) # group 返回匹配的组
print(re.match('fkit', 'www.fkit.com')) # 开始位置匹配不到，返回 None
m2 = re.search('www', 'www.fkit.org') # 开始位置可以匹配
print(m2.span())
print(m2.group())
m3 = re.search('fkit', 'www.fkit.com') # 中间位置可以匹配，返回 Match 对象
print(m3.span())
print(m3.group())
```

运行上面程序，可以看到如下输出结果：

```
(0, 3)
www
None
(0, 3)
www
(4, 8)
fkit
```

从上面的输出结果可以看出，match() 函数要求必须从字符串开始处匹配，而 search() 函数则可扫描整个字符串，从中间任意位置开始匹配。

re.findall(pattern, string, flags=0)：扫描整个字符串，并返回字符串中所有匹配 pattern 的子串组成的列表。其中 pattern 参数代表正则表达式；string 代表被匹配的字符串；flags 则代表正则表达式的匹配旗标。

re.finditer(pattern, string, flags=0)：扫描整个字符串，并返回字符串中所有匹配 pattern 的子串组成的迭代器，迭代器的元素是 _sre.SRE_Match 对象。其中 pattern 参数代表正则表达式；string 代表被匹配的字符串；flags 则代表正则表达式的匹配旗标。

从上面介绍不难看出，findall() 与 finditer() 函数的功能基本相似，区别在于它们的返回值不同，findall() 函数返回所有匹配 patten 的子串组成的列表；而 finditer() 函数则返回所有匹配 pattern 的子串组成的迭代器。

如果对比 findall()、finditer() 和 search() 函数，它们的区别也很明显，search() 只返回字符串中第一处匹配 pattern 的子串；而 findall() 和 finditer() 则返回字符串中所有匹配 pattern 的子串。

```
18.         import re
19.         # 返回所有匹配 pattern 的子串组成的列表，忽略大小写
20.         print(re.findall('fkit', 'FkIt is very good , Fkit.org is my favorite', re.I))
21.         # 返回所有匹配 pattern 的子串组成的迭代器，忽略大小写
22.         it = re.finditer('fkit', 'FkIt is very good , Fkit.org is my favorite', re.I)
23.         for e in it:
24.             print(str(e.start()) + "-->" + e.group())
```

re.fullmatch(pattem, string, flags=0)：该函数要求整个字符串能匹配 pattern，如果匹配则返回包含匹配信息的 _sre.SRE_Match 对象；否则返回 None。

re.sub(pattern, repl, string, count=0, flags=0)：该函数用于将 string 字符串中所有匹配 pattern 的内容替换成 repl；repl 既可是被替换的字符串，也可是一个函数。count 参数控制最多替换多少次，如果指定 count 为 0 ，则表示全部首换。如下程序示范了 sub() 函数的简单用法：

```
25.         import re
26.         my_date = '2008-08-18'
27.         # 将 my_date 字符串里中画线替换成斜线
28.         print(re.sub(r'-', '/' , my_date))
29.         # 将 my_date 字符串里中画线替换成斜线，只替换一次
30.         print(re.sub(r'-', '/' , my_date, 1))
```

运行上面程序，可以看到如下输出结果：

2008/08/18

2008/08-18

上面程序所使用的 r'-' 是原始字符串，其中 r 代表原始字符串，通过使用原始字符串，可以避免对字符串中的特殊字符进行转义。

在某些情况下，所执行的替换要基于被替换内容进行改变。比如下面程序需要将字符串中的每个英文单词都变成一套教程的名字：

```
        # 在匹配的字符串前后添加内容
        def fun(matched):
            # matched 就是匹配对象，通过该对象的 group()方法可获取被匹配的字符串
            value = "《基础" + (matched.group('lang')) + "教程》"
            return value
        s = 'Python 很好，Kotlin 也很好'
        # 将 s 里面的英文单词（用 re.A 旗标控制）进行替换
        # 使用 fun 函数指定替换的内容
        print(re.sub(r'(?P<lang>\w+)', fun, s, flags=re.A))
```

上面程序使用 sub() 函数执行替换时，指定使用 fun() 函数作为替换内容，而 fun() 函数则负责在 pattern 匹配的字符串之前添加“《基础”，在 pattern 匹配的字符串之后添加“教程》”。运行上面程序，可以看到如下输出结果：

《基础 Python 教程》很好，《基础 Kotlin 教程》也很好
由于此时还未深入介绍正则表达式的语法，因此前面所使用的正则表达式都很简单，但此处使用了一个稍微复杂的正则表达式： r'(? P<lang>\w+)'。

r'(?P<lang>\w+)' 正则表达式用圆括号表达式创建了一个组，并使用“?P”选项为该组起名为 lang（所起的组名要放在尖括号内）。剩下的“\w+”才是正则表达式的内容，其中“\w”代表任意字符；“+”用于限定前面的“\w”可出现一次到多次，因此“\w+”代表一个或多个任意字符。又由于程序执行 sub() 函数时指定了 re.A 选项，这样“\w”就只能代表 ASCII 字符，不能代表汉字。

当使用 sub() 函数执行替换时，正则表达式“\w+”所匹配的内容可以通过组名“lang”来获取，这样 fun() 函数就调用了 matched.group('lang') 来获取“\w+”所匹配的内容。
re.split(pattem, string, maxsplit=0, flags=0)：使用 pattern 对 string 进行分割，该函数返回分割得到的多个子串组成的列表。其中 maxsplit 参数控制最多分割几次。如下程序示范了 split() 函数的用法：

```
import re
# 使用逗号对字符串进行分割
print(re.split(', ', 'fkit, fkjava, crazyit'))
# 输出：['fkit', 'fkjava', 'crazyit']
# 指定只分割 1 次，被切分成 2 个子串
print(re.split(', ', 'fkit, fkjava, crazyit', 1))
# 输出：['fkit', 'fkjava, crazyit']
# 使用 a 进行分割
print(re.split('a', 'fkit, fkjava, crazyit'))
# 输出：['fkit, fkj', 'va, crazyit']
# 使用 x 进行分割，没有匹配内容，则不会执行分割
print(re.split('x', 'fkit, fkjava, crazyit'))
# 输出：['fkit, fkjava, crazyit']
```

re.purge()：清除正则表达式缓存。
re.escape(pattern)：对模式中除 ASCII 字符、数值、下画线（_）之外的其他字符进行转义。如下程序示范了 escape() 函数的用法：

```
import re
# 对模式中特殊字符进行转义
print(re.escape(r'www.crazyit.org is good, i love it!'))
# 输出：www\.crazyit\.org\ is\ good\,\ i\ love\ it\!
print(re.escape(r'A-Zand0-9?'))
# 输出：A\-Zand0\-9\?
```

从上面的输出结果可以看出，使用 escape() 函数对模式执行转义之后，模式中除 ASCII 字符、数值、下画线之外的其他字符都被添加了反斜线进行转义。

此外，在 re 模块中还包含两个类，它们分别是正则表达式对象（其具体类型为 _sre.SRE_Pattern）和匹配（Match）对象，其中正则表达式对象就是调用 re.compile() 函数的返回值。该对象的方法与前面介绍的 re 模块中的函数大致对应。

相比之下，正则表达式对象的 search()、match()、fullmatch()、findall()、finditer() 方法的功能更强大一些，因为这些方法都可额外指定 pos 和 endpos 两个参数，用于指定只处理目标字符串从 pos 开始到 endpos 结束之间的子串。如下程序示范了使用正则表达式的方法来执行匹配：

```
import re
# 编译得到正则表达式对象
pa = re.compile('fkit')
```

```
# 调用 match 方法，原本应该从开始匹配，
# 此处指定从索引为 4 的地方开始匹配，就可以成功匹配了
print(pa.match('www.fkit.org', 4).span()) # (4, 8)
# 此处指定从索引为 4 到索引 6 之间执行匹配，匹配失败
print(pa.match('www.fkit.org', 4, 6)) # None
# 此处指定从索引为 4 到索引 8 之间执行全匹配，匹配成功
print(pa.fullmatch('www.fkit.org', 4, 8).span()) # (4, 8)
```

上面程序示范了使用正则表达式调用 match()、fullmatch() 方法时指定 pos 和 endpos 参数的效果，在指定这两个参数之后，程序就可以只处理目标字符串的中间一段。此外，通过上面程序也可以体会到编译正则表达式的好处，即程序使用 compile() 函数编译正则表达式之后，该函数所返回的对象就会缓存该正则表达式，从而可以多次利用该正则表达式执行匹配。比如上面程序多次使用 pa 对象（它缓存了正则表达式）来执行匹配。

re 模块中的 Match 对象（其具体类型为 _sre.SRE_Match）则是 match()、search() 方法的返回值，该对象中包含了详细的正则表达式匹配信息，包括正则表达式匹配的位置、正则表达式所匹配的子串。

sre.SRE_Match 对象包含了如下方法或属性：
match.group([group1,...])：获取该匹配对象中指定组所匹配的字符串。
match.__getitem__(g)：这是 match.group(g) 的简化写法。由于 match 对象提供了 __getitem__() 方法，因此程序可使用 match[g] 来代替 match.group(g)。
match.groups(default=None)：返回 match 对象中所有组所匹配的字符串组成的元组。
match.groupdict(default= None)：返回 match 对象中所有组所匹配的字符串组成的字典。
match.start([group])：获取该匹配对象中指定组所匹配的字符串的开始位置。
match.end([group])：获取该匹配对象中指定组所匹配的字符串的结束位置。
match.span([group])：获取该匹配对象中指定组所匹配的字符串的开始位置和结束位置。该方法相当于同时返回 start() 和 end() 方法的返回值。

上面这些方法都涉及了组的概念，组是正则表达式中很常见的一个东西，用圆括号将多个表达式括起来形成组。如果正则表达式中没有圆括号，那么整个表达式就属于一个默认组。如下程序示范了正则表达式包含组的情形：

```
import re
# 在正则表达式中使用组
m = re.search(r'(fkit).(org)', r"www.fkit.org is a good domain")
print(m.group(0)) # fkit.org
# 调用简化写法，底层是调用 m.__getitem__(0)
print(m[0]) # fkit.org
print(m.span(0)) # (4, 12)
print(m.group(1)) # fkit
```

```
        # 调用简化写法，底层是调用 m.__getitem__(1)
        print(m[1]) # fkit
        print(m.span(1)) # (4, 8)
        print(m.group(2)) # org
        # 调用简化写法，底层是调用 m.__getitem__(2)
        print(m[2]) # org
        print(m.span(2)) # (9, 12)
        # 返回所有组所匹配的字符串组成的元组
        print(m.groups())
```

上面程序中 search() 函数使用了一个正则表达式： r'(tkit).(org)'，在该正则表达式内包含两个组，即 (fkit) 和 (org)，因此程序可以依次获取 group(0)、group(1)、group(2) 的值，也就是依次获取整个正则表达式所匹配的子串、第一个组所匹配的子串和第二个组所匹配的子串；程序也可以依次获取 span(0)、span(1)、span(2) 的值，也就是依次获取整个正则表达式所匹配的子串的开始位置和结束位置、第一个组所匹配的子串的开始位置和结束位置、第二个组所匹配的子串的开始位置和结束位置。

运行上面程序，可以看到如下输出结果：

```
fkit.org

fkit.org

(4, 12)

fkit

fkit

(4, 8)

org

org

(9, 12)

('fkit', 'org')
```

从该程序可以看出，只要正则表达式能匹配得到结果，则不管正则表达式是否包含组，group(0)、span(0) 总能获得内容，因为它们分别是获取整个正则表达式所匹配的子串，以及该子串的开始位置和结束位置。

如果在正则表达式中为组指定了名字（用 ?P<名字> 为正则表达式的组指定名字），就可以调用 groupdict() 方法来获取所有组所匹配的字符串组成的字典（其中组名作为字典的 key）。例如如下代码：

```
        # 正则表达式定义了 2 个组，并为组指定了名字
        m2 = re.search(r'(?P<prefix>fkit).(?P<suffix>org)',\
            r"www.fkit.org is a good domain")
        print(m2.groupdict()) # {'prefix': 'fkit', 'suffix': 'org'}
```

上面程序为正则表达式的第一个组指定名字为 prefix，为第二个组指定名字为 suffix。运行上面程序，可以看到如下输出结果：

```
{'prefix': 'fkit', 'suffix': 'org'}
```

从上面的输出结果可以看到，此处返回的字典的 key 为正则表达式中的组名，value 为该组所匹配的子串。

match.pos：该属性返回传给正则表达式对象的 search()、match() 等方法的 pos 参数。

match.endpos：该属性返回传给正则表达式对象的 search()、match() 等方法的 endpos 参数。

match.lastindex：该属性返回最后一个匹配的捕获组的整数索引。如果没有组匹配，该属性返回 None。例如用 (a)b、((a)(b)) 或 ((ab)) 对字符串‘ab’执行匹配，该属性都会返回 1；但如果使用 (a)(b) 正则表达式对‘ab’执行匹配，则 lastindex 等于 2。

match.lastgroup：该属性返回最后一个匹配的捕获组的名字；如果该组没有名字或根本没有组匹配，该属性返回 None。

match.re：该属性返回执行正则表达式匹配时所用的正则表达式。

match.string：该属性返回执行正则表达式匹配时所用的字符串。

7.Python set 和 frozenset

set 集合是可变容器，程序可以改变容器中的元素。与 set 对应的还有 frozenset 集合，frozenset 是 set 的不可变版本，它的元素是不可变的。

Python set 集合

set 集合有如下两个特征：

- set 不记录元素的添加顺序。
- 元素不允许重复。

在交互式解释器中输入 `[e for e in dir(set) if not e.startswith('_')]` 来查看 set 集合的全部方法，可以看到如下输出结果：

```
>>> [e for e in dir(set) if not e.startswith('_')]
['add', 'clear', 'copy', 'difference', 'difference_update', 'discard', 'intersection',
'intersection_update', 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove',
'symmetric_difference', 'symmetric_difference_update', 'union', 'update']
```

对于上面这些方法，其方法名已经暗示了它们的作用，比如 add() 很明显就是向 set 集合中添加元素，remove()、discard() 就是删除元素，clear() 就是清空白 set 集合，等等。

remove() 与 discard() 方法都用于删除集合中的元素，但区别在于，如果集合中不包含被删除的元素，remove() 方法会报出 KeyError 异常，而 discard() 方法则什么也不做。

下面程序示范了 set 集合的方法的用法：

```
# 使用花括号构建 set 集合
c = {'白骨精'}
# 添加元素
c.add("孙悟空")
c.add(6)
print("c 集合的元素个数为:" , len(c)) # 输出 3
# 删除指定元素
c.remove(6)
print("c 集合的元素个数为:" , len(c)) # 输出 2
# 判断是否包含指定字符串
print("c 集合是否包含'孙悟空'字符串:" , ("孙悟空" in c)) # 输出 True
c.add("轻量级 Java EE 企业应用实战")
print("c 集合的元素: " , c)
# 使用 set()函数（构造器）来创建 set 集合
books = set()
books.add("轻量级 Java EE 企业应用实战")
books.add("疯狂 Java 讲义")
print("books 集合的元素: " , books)
# issubset()方法判断是否为子集合
print("books 集合是否为 c 的子集合？", books.issubset(c)) # 输出 False
# issubset()方法与<=运算符效果相同
print("books 集合是否为 c 的子集合？", (books <= c)) # 输出 False
# issuperset()方法判断是否为父集合
# issubset 和 issuperset 其实就是倒过来判断
print("c 集合是否完全包含 books 集合？", c.issuperset(books)) # 输出 False
# issuperset()方法与>=运算符效果相同
print("c 集合是否完全包含 books 集合？", (c >= books)) # 输出 False
# 用 c 集合减去 books 集合里的元素，不改变 c 集合本身
result1 = c - books
```

```
30.         print(result1)
31.         # difference()方法也是对集合做减法，与用-执行运算的效果完全一样
32.         result2 = c.difference(books)
33.         print(result2)
34.         # 用 c 集合减去 books 集合里的元素，改变 c 集合本身
35.         c.difference_update(books)
36.         print("c 集合的元素：", c)
37.         # 删除 c 集合里的所有元素
38.         c.clear()
39.         print("c 集合的元素：", c)
40.         # 直接创建包含元素的集合
41.         d = {"疯狂 Java 讲义", '疯狂 Python 讲义', '疯狂 Kotlin 讲义'}
42.         print("d 集合的元素：", d)
43.         # 计算两个集合的交集，不改变 d 集合本身
44.         inter1 = d & books
45.         print(inter1)
46.         # intersection()方法也是获取两个集合的交集，与用&执行运算的效果完全一样
47.         inter2 = d.intersection(books)
48.         print(inter2)
49.         # 计算两个集合的交集，改变 d 集合本身
50.         d.intersection_update(books)
51.         print("d 集合的元素：", d)
52.         # 将 range 对象包装成 set 集合
53.         e = set(range(5))
54.         f = set(range(3, 7))
55.         print("e 集合的元素：", e)
56.         print("f 集合的元素：", f)
57.         # 对两个集合执行异或运算
58.         xor = e ^ f
59.         print('e 和 f 执行 xor 的结果：', xor)
60.         # 计算两个集合的并集，不改变 e 集合本身
61.         un = e.union(f)
62.         print('e 和 f 执行并集的结果：', un)
63.         # 计算两个集合的并集，改变 e 集合本身
64.         e.update(f)
65.         print('e 集合的元素：', e)
```

上面程序基本示范了 set 集合中所有方法的用法。不仅如此，该程序还示范了 set 集合支持的如下几个运算符：

- <=：相当于调用 issubset() 方法，判断前面的 set 集合是否为后面的 set 集合的子集合。
- >=：相当于调用 issuperset() 方法，判断前面的 set 集合是否为后面的 set 集合的父集合。
- -：相当于调用 difference() 方法，用前面的 set 集合减去后面的 set 集合的元素。
- &：相当于调用 intersection() 方法，用于获取两个 set 集舍的交集。
- ^：计算两个集合异或的结果，就是用两个集合的并集减去交集的元素。

此外，由于 set 集合本身是可变的，因此它除了提供 add()、remove()、discard() 方法来操作单个元素，还支持进行集合运算来改变集合内的元素。因此，它的集合运算方法都有两个版本：

1. 交集运算：intersection() 和 intersection_update()，前者不改变集合本身，而是返回两个集合的交集；后者会通过交集运算改变第一个集合。
2. 并集运算：union() 和 update()，前者不改变集合本身，而是返回两个集合的并集；后者会通过并集运算改变第一个集合。
3. 减法运算：difference() 和 difference_update()，前者不改变集合本身，而是返回两个集合做咸法的结果；后者改变第一个集合。

Python frozenset 集合

frozenset 是 set 的不可变版本，因此 set 集合中所有能改变集合本身的方法（如 add、remove、discard、xxx_update 等），frozenset 都不支持；set 集合中不改变集合本身的方法，fronzenset 都支持。

在交互式解释器中输入 `[e for e in dir(frozenset) if not e.startswith('_')]` 命令来查看 frozenset 集合的全部方法，可以看到如下输出结果：

```
[e for e in dir(frozenset) if not e.startswith('_')]
['copy', 'difference', 'intersection', 'isdisjoint', 'issubset', 'issuperset',
'symmetric_difference', 'union']
```

很明显，frozenset 的这些方法和 set 集合的同名方法的功能完全相同。

frozenset 的作用主要有两点：

- 当集合元素不需要改变时，使用 frozenset 代替 set 更安全。
- 当某些 API 需要不可变对象时，必须用 frozenset 代替 set。比如 dict 的 key 必须是不可变对象，因此只能用 frozenset；再比如 set 本身的集合元素必须是不可变的，因此 set 不能包含 set，set 只能包含 frozenset。

如下程序示范了在 set 中添加 frozenset：

```
s = set()
frozen_s = frozenset('Kotlin')
# 为 set 集合添加 frozenset
s.add(frozen_s)
print('s 集合的元素：', s)
sub_s = {'Python'}
# 为 set 集合添加普通 set 集合，程序报错
s.add(sub_s)
```

上面程序中第 4 行代码为 set 添加 frozenset，程序完全没有问题，这样 set 中就包括一个 frozenset 子集合 ;第 8 行代码试图向 set 中添加普通 set ,程序会报出 TypeError 异常。运行上面代码，可以看到如下输出结果：

```
s 集合的元素： {frozenset({'t', 'o', 'i', 'K', 'l', 'n'})}
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 8, in <module>
    s.add(sub_s)
TypeError: unhashable type: 'set'
```

8.Python queue（双端队列）模块及用法

在“数据结构”课程中最常讲授的数据结构有栈、队列、双端队列。

栈是一种特殊的线性表，它只允许在一端进行插入、删除操作，这一端被称为栈顶（top），另一端则被称为栈底（bottom）。

从栈顶插入一个元素被称为进栈，将一个元素插入栈顶被称为“压入栈”，对应的英文说法为 push；相应地，从栈顶删除一个元素被称为出栈，将一个元素从栈顶删除被称为“弹出栈”，对应的英文说法为 pop。

对于栈而言，最先入栈的元素位于栈底，只有等到上面所有元素都出栈之后，栈底的元素才能出栈。因此栈是一种后进先出（LIFO）的线性表。如图 1 所示为栈的操作示意图。

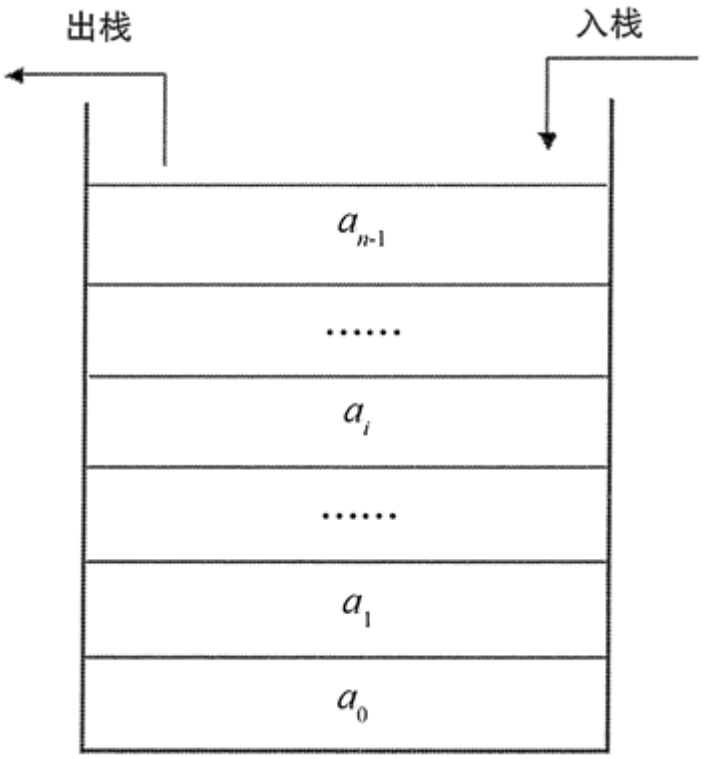


图 1 栈的操作示意图

队列也是一种特殊的线性表，它只允许在表的前端（front）进行删除操作，在表的后端（rear）进行插入操作。进行插入操作的端被称为队尾，进行删除操作的端被称为队头。

对于一个队列来说，每个元素总是从队列的 rear 端进入队列的，然后等待该元素之前的所

有元素都出队之后，当前元素才能出队。因此，队列是一种先进先出（FIFO）的线性表。

队列示意图如图 2 所示。

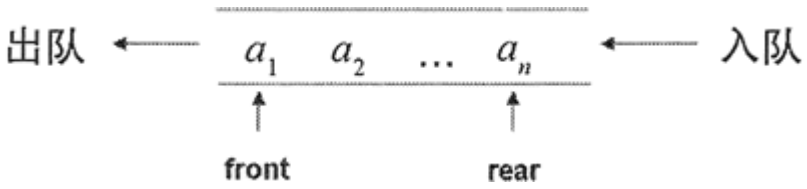


图 2 队列示意图

双端队列（即此处介绍的 deque）代表一种特殊的队列，它可以在两端同时进行插入、删除操作，如图 3 所示。

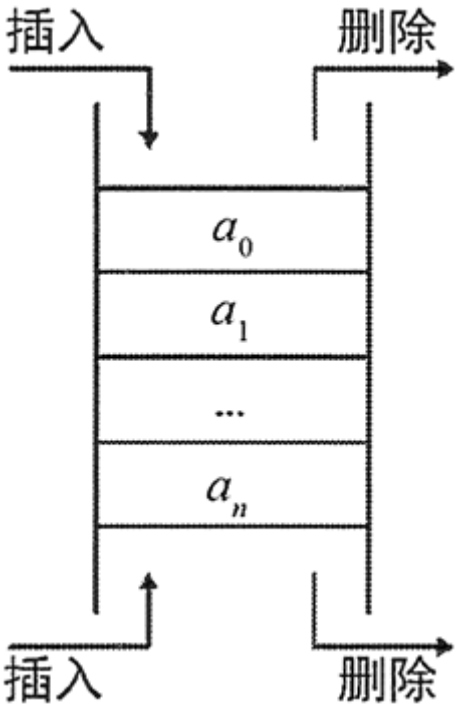


图 3 双端队列示意图

对于一个双端队列来说，它可以从两端分别进行插入、删除操作，如果程序将所有的插入、

删除操作都固定在一端进行，那么这个双端队列就变成了栈；如果固定在一端只添加元素，在另一端只删除元素，那么它就变成了队列。因此，deque 既可被当成队列使用，也可被当成栈使用。

deque 位于 collections 包下，在交互式解释器中先导入 collections 包，然后输入 [e for e in dir(collections.deque) if not e.startswith('_')] 命令来查看 deque 的全部方法，可以看到如下输出结果：

```
>>> from collections import deque
>>> [e for e in dir(deque) if not e.startswith('_')]
['append', 'appendleft', 'clear', 'copy', 'count', 'extend', 'extendleft', 'index', 'insert', 'maxlen', 'pop', 'popleft', 'remove', 'reverse', 'rotate']
```

从上面的方法可以看出，deque 的方法基本都有两个版本，这就体现了它作为双端队列的特征。deque 的左边（left）就相当于它的队列头（front），右边（right）就相当于它的队列尾（rear）：

- append 和 appendleft：在 deque 的右边或左边添加元素，也就是默认在队列尾添加元素。
- pop 和 popleft：在 deque 的右边或左边弹出元素，也就是默认在队列尾弹出元素。
- extend 和 extendleft：在 deque 的右边或左边添加多个元素，也就是默认在队列尾添加多个元素。

deque 中的 clear() 方法用于清空队列：insert() 方法则是线性表的方法，用于在指定位置插入元素。

假如程序要把 deque 当成栈使用，则意味着只在一端添加、删除元素，因此调用 append 和 pop 方法即可。例如如下程序：

```
from collections import deque
stack = deque(('Kotlin', 'Python'))
# 元素入栈
stack.append('Erlang')
stack.append('Swift')
print('stack 中的元素：' , stack)
# 元素出栈，后添加的元素先出栈
print(stack.pop())
print(stack.pop())
print(stack)
```

运行上面程序，可以看到如下运行结果：

```
stack 中的元素： deque(['Kotlin', 'Python', 'Erlang', 'Swift'])
Swift
Erlang
deque(['Kotlin', 'Python'])
```

从上面的运行结果可以看出，程序最后入栈的元素“Swift”最先出栈，这体现了栈的 LIFO 的特征。

假如程序要把 deque 当成队列使用，则意味着一端只用来添加元素，另一端只用来删除元素，因此调用 append、popleft 方法即可。例如如下程序：

```
from collections import deque
q = deque(('Kotlin', 'Python'))
# 元素加入队列
q.append('Erlang')
q.append('Swift')
print('q 中的元素：', q)
# 元素出队列，先添加的元素先出队列
print(q.popleft())
print(q.popleft())
print(q)
```

运行上面程序，可以看到如下运行结果：

```
q 中的元素： deque(['Kotlin', 'Python', 'Erlang', 'Swift'])
Kotlin
Python
deque(['Erlang', 'Swift'])
```

从上面的运行结果可以看出，程序先添加的元素“Katlin”最先出队列，这体现了队列的 FIFO 的特征。

此外，deque 还有一个 rotate() 方法，该方法的作用是将队列的队尾元素移动到队头，使之首尾相连。例如如下程序：

```
from collections import deque
q = deque(range(5))
```

```
        print('q 中的元素：', q)
        # 执行旋转，使之首尾相连
        q.rotate()
        print('q 中的元素：', q)
        # 再次执行旋转，使之首尾相连
        q.rotate()
        print('q 中的元素：', q)
```

运行上面程序，可以看到如下输出结果：

```
q 中的元素： deque([0, 1, 2, 3, 4])
q 中的元素： deque([4, 0, 1, 2, 3])
q 中的元素： deque([3, 4, 0, 1, 2])
```

从上面的输出结果来看，每次执行 rotate() 方法，deque 的队尾元素都会被移到队头，这样就形成了首尾相连的效果。

9.Python heapq（堆操作）

Python 提供了关于堆的操作，下面先简单介绍有关堆的概念。

假设有 n 个数据元素的序列 k_0，k_1，...，k_{n-1}，当且仅当满足 $k_i \le k_{2i+1}$ 且 $k_i \le k_{2i+2}$（其中 i=0,2,...,(n-1)/2)时，可以将这组数据称为小顶堆（小根堆）；或者满足 $k_i \ge k_{2i+1}$ 且 $k_i \ge k_{2i+2}$（其中 i=0,2,...,(n-1)/2 ）时，可以将这组数据称为大顶堆（大根堆）。

对于满足小顶堆的数据序列 k_0，k_1，...，k_{n-1}，如果将它们顺序排成一棵完全二叉树，则此树的特点是：树中所有节点的值都小于其左、右子节点的值，此树的根节点的值必然最小。

反之，对于满足大顶堆的数据序列 k_0，k_1，…，k_{n-1}，如果将它们顺序排成一棵完全二叉树，则此树的特点是：树中所有节点的值都大于其左、右子节点的值，此树的根节点的值必然最大。

通过上面介绍不难发现，小顶堆的任意子树也是小顶堆，大顶堆的任意子树还是大顶堆。

Python 提供的是基于小顶堆的操作，因此 Python 可以对 list 中的元素进行小顶堆排列，这样程序每次获取堆中元素时，总会取得堆中最小的元素。

例如，判断数据序列 9, 30, 49, 46, 58, 79 是否为堆，可以将其转换为一棵完全二叉树，如图 1 所示。

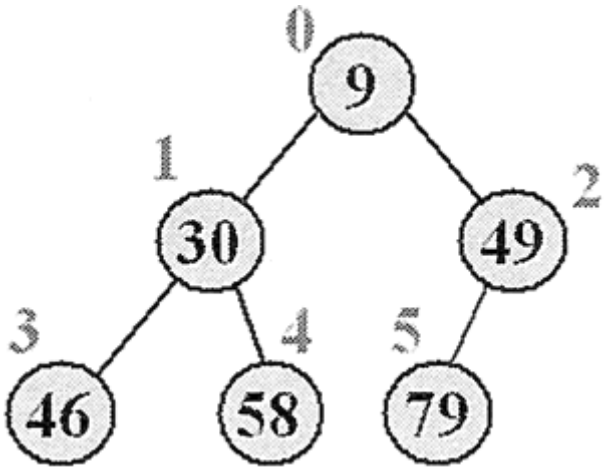


图 1 完全二叉树

在图 1 中，每个节点上的灰色数字代表该节点数据在底层数组中的索引。图 1 所示的完全二叉树完全满足小顶堆的特征，每个父节点的值总小于或等于它的左、右子节点的值。

Python 并没有提供“堆”这种数据类型，它是直接把列表当成堆处理的。Python 提供的 heapq 包中有一些函数，当程序用这些函数来操作列表时，该列表就会表现出“堆”的行为。

在交互式解释器中先导入 heapq 包，然后输入 heapq.__all__ 命令来查看 heapq 包下的全部函数，可以看到如下输出结果：

```
>>> heapq.__all__
['heappush', 'heappop', 'heapify', 'heapreplace', 'merge', 'nlargest', 'nsmallest',
'heappushpop']
```

上面这些函数就是执行堆操作的工具函数，这些函数的功能大致如下：

- heappush(heap, item)：将 item 元素加入堆。
- heappop(heap)：将堆中最小元素弹出。
- heapify(heap)：将堆属性应用到列表上。
- heapreplace(heap, x)：将堆中最小元素弹出，并将元素 x 入堆。
- merge(*iterables, key=None, reverse=False)：将多个有序的堆合并成一个大的有序堆，然后再输出。
- heappushpop(heap, item)：将 item 入堆，然后弹出并返回堆中最小的元素。
- nlargest(n, iterable, key=None)：返回堆中最大的 n 个元素。
- nsmallest(n, iterable, key=None)：返回堆中最小的 n 个元素。

下面程序示范了这些函数的用法：

```
from heapq import *
my_data = list(range(10))
my_data.append(0.5)
# 此时 my_data 依然是一个 list 列表
print('my_data 的元素：', my_data)
# 对 my_data 应用堆属性
heapify(my_data)
print('应用堆之后 my_data 的元素：', my_data)
heappush(my_data, 7.2)
print('添加 7.2 之后 my_data 的元素：', my_data)
```

上面程序开始创建了一个 list 列表，接下来程序调用 heapify() 函数对列表执行堆操作，执行之后看到 my_data 的元素顺序如下：

```
应用堆之后 my_data 的元素：[0 , 0.5, 2, 3, 1, 5, 6, 7, 8, 9, 4]
```

这些元素看上去是杂乱无序的，但其实并不是，它完全满足小顶堆的特征。我们将它转换为完全二叉树，可以看到如图 2 所示的效果。

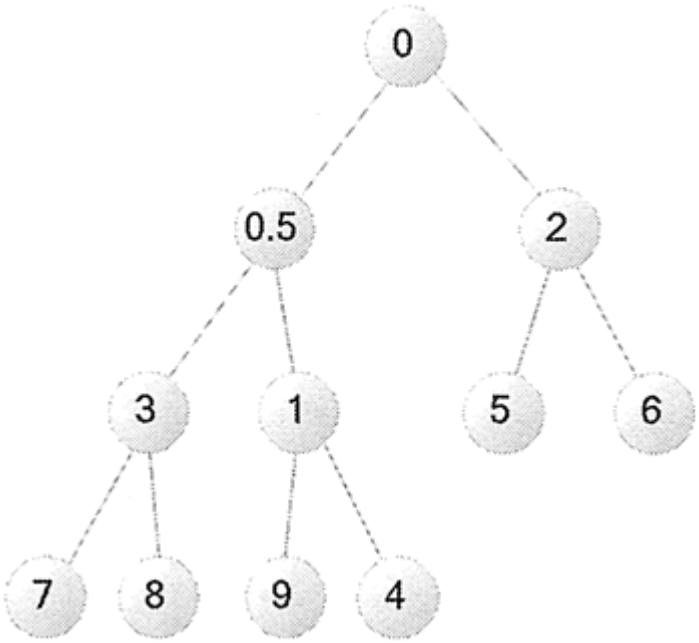


图 2 小顶堆对应的完全二叉树

当程序再次调用 heappush(my_data, 7.2) 向堆中加入一个元素之后，输出该堆中元素，可以看到如下输出结果：

```
添加 7.2 之后 my_data 的元素：[0, 0.5, 2, 3, 1, 5, 6, 7, 8, 9, 4,7.2]
```

此时将它转换为完全二叉树，可以看到如图 3 所示的效果。

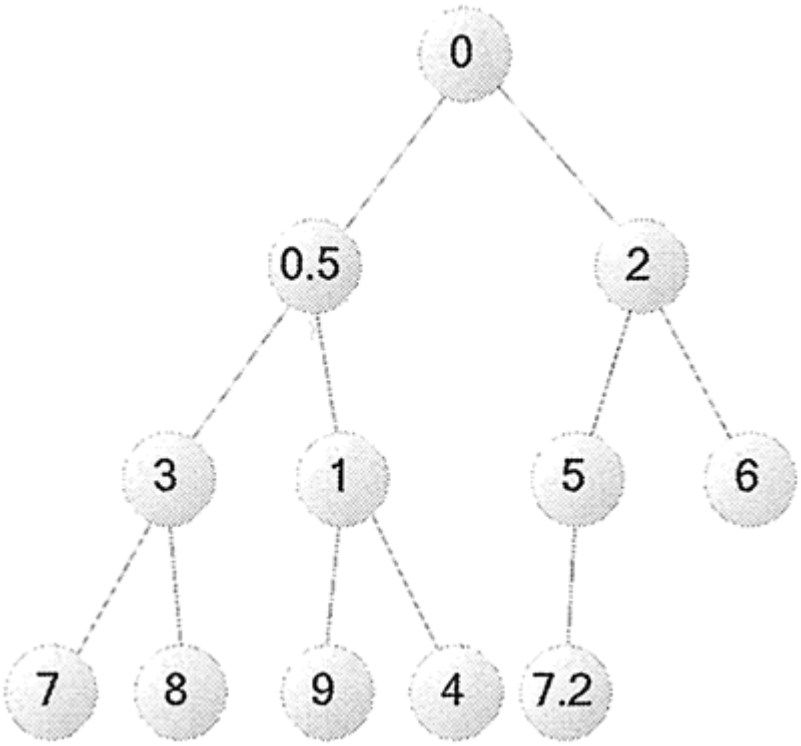


图 3 添加 7.2 之后的小顶堆对应的完全二叉树

接下来程序尝试从堆中弹出两个元素：

```
# 弹出堆中最小的元素
print(heappop(my_data)) # 0
print(heappop(my_data)) # 0.5
print('弹出两个元素之后 my_data 的元素：', my_data)
```

上面三行代码的输出如下：

```
0
0.5
弹出两个元素之后 my_data 的元素： [1, 3, 2, 7, 4, 5, 6, 7.2, 8, 9]
```

从最后输出的 my_data 的元素来看，此时 my_data 的元素依然满足小顶堆的特征。

下面代码示范了 replace() 函数的用法：

```
# 弹出最小元素，压入指定元素
print(heapreplace(my_data, 8.1))
print('执行 replace 之后 my_data 的元素：', my_data)
```

执行上面两行代码，可以看到如下输出结果：

```
1
执行 replace 之后 my_data 的元素： [2, 3, 5, 7, 4, 8.1, 6, 7.2, 8, 9]
```

也可以测试通过 nlargest()、nsmallest() 来获取最大、最小的 n 个元素，代码如下：

```
print('my_data 中最大的 3 个元素：', nlargest(3, my_data))
print('my_data 中最小的 4 个元素：', nsmallest(4, my_data))
```

运行上面程序，可以看到如下输出结果：

```
my_data 中最大的 3 个元素： [9, 8.1, 8]
my_data 中最小的 4 个元素： [2, 3, 4, 5]
```

通过上面程序不难看出，Python 的 heapq 包中提供的函数，其实就是提供对排序算法中“堆排序”的支持。Python 通过在底层构建小顶堆，从而对容器中的元素进行排序，以便程序能快速地获取最小、最大的元素，因此使用起来非常方便。

提示：当程序要获取列表中最大的 n 个元素，或者最小的 n 个元素时，使用堆能缓存列表的排序结果，因此具有较好的性能。

10.Python ChainMap

ChainMap 是一个方便的工具类，它使用链的方式将多个 dict “链”在一起，从而允许程序可直接获取任意一个 dict 所包含的 key 对应的 value。

简单来说，ChainMap 相当于把多个 dict 合并成一个大的 dict，但实际上底层并未真正合并这些 dict，因此程序无须调用多个 update() 方法将多个 dict 进行合并。所以说 ChainMap 是一种合并，但实际用起来又具有较好的效果。

需要说明的是，由于 ChainMap 只是将多个 dict 链在一起，并未真正合并它们，因此在多个 dict 中完全可能具有重复的 key，在这种情况下，排在“链”前面的 dict 中的 key 具有更高的优先级。

例如，下面程序示范了 ChainMap 的用法：

```
from collections import ChainMap
# 定义 3 个 dict 对象
a = {'Kotlin': 90, 'Python': 86}
b = {'Go': 93, 'Python': 92}
c = {'Swift': 89, 'Go': 87}
# 将 3 个 dict 对象链在一起，就像变成了一个大的 dict
cm = ChainMap(a, b , c)
```

```
print(cm)
# 获取 Kotlin 对应的 value
print(cm['Kotlin']) # 90
# 获取 Python 对应的 value
print(cm['Python']) # 86
# 获取 Go 对应的 value
print(cm['Go']) # 93
```

上面程序 a、b、c 三个 dict 链在一起，组成一个 ChainMap，这个链的顺序就是 a、b、c，因此 a 的优先级最高，b 的优先级次之，c 的优先级最低。运行上面程序，可以看到如下输出结果：

```
ChainMap({'Kotlin': 90, 'Python': 86}, {'Go': 93, 'Python': 92}, {'Swift': 89, 'Go': 87})
90
86
93
```

从上面第一行输出可以看到，ChainMap 其实并未将这三个 dict 合并成一个大的 dict，只是将它们链在一起而已。

掌握了 ChainMap 的用法之后，接下来介绍的示例程序借鉴了 Python 库文档中的一个例子，该例子将局部范围的定义、全局范围的定义、Python 内置定义链成一个 ChainMap，当程序通过该 ChainMap 获取变量时，将会按照局部定义、全局定义、内置定义的顺序执行搜索。

```
from collections import ChainMap
import builtins
my_name = '孙悟空'
```

```
    def test():
        my_name = 'yeeku'
        # 将 locals、globals、builtins 的变量链接成 ChainMap
        pylookup = ChainMap(locals(), globals(), vars(builtins))
        # 访问 my_name 对应的 value，优先使用局部范围的定义
        print(pylookup['my_name']) # 'yeeku'
        # 访问 len 对应的 value，由于局部范围、全区范围都找不到，因此访问内置定义的 len 函数
        print(pylookup['len']) # <built-in function len>
    test()
```

下面程序示范了优先使用运行程序的指定参数，然后是系统环境变量，最后才使用系统默认值的实现，程序同样将这三个搜索范围链成 ChainMap：

```
#程序文件名为 ChainMap_test.py
from collections import ChainMap
import os, argparse
# 定义默认参数
defaults = {'color': '蓝色', 'user': 'yeeku'}
# 创建程序参数解析器
parser = argparse.ArgumentParser()
# 为参数解析器添加-u（--user）和-c（--color）参数
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
# 解析运行程序的参数
namespace = parser.parse_args()
# 将程序参数转换成 dict
command_line_args = {k:v for k, v in vars(namespace).items() if v}
# 将 command_line_args(解析程序参数而来)、os.environ(环境变量)、defaults 链成 ChainMap
combined = ChainMap(command_line_args, os.environ, defaults)
# 获取 color 对应的 value
print(combined['color'])
# 获取 user 对应的 value
print(combined['user'])
```

```
21.         # 获取 PYTHONPATH 对应的 value
22.         print(combined['PYTHONPATH'])
```

上面程序将 command_line_args、as.environ、defaults 链成 ChairMap，其中 command_line_args 由程序参数解析而来，它的优先级最高。

假如使用如下命令来运行该程序：

```
python ChainMap_test.py -c 红色 -u Charlie
```

由于上面命令指定了 -c（对应于 color）和 -u（对应于 user）参数，在命令行指定的参数的优先级是最高的，因此可以看到如下输出结果：

```
红色
Charlie
.;d:\python_module
```

上面输出的最后一行是 PYTHONPATH 环境变量的值。如果使用如下命令来运行该程序：

```
python ChainMap_test.py
```

由于上面命令没有指定任何命令行参数，因此程序访问 user、color 时将会使用 defaults 字典中 key 对应的值，所以可以看到如下输出结果：

蓝色

yeeku

.;d:\python_ module

11.Python Counter

collections 包下的 Counter 也是一个很有用的工具类 ,它可以自动统计容器中各元素出现的次数。

Counter 的本质就是一个特殊的 dict，只不过它的 key 都是其所包含的元素，而它的 value 则记录了该 key 出现的次数。因此，如果通过 Counter 并不存在的 key 访问 value，将会输出 0（代表该 key 出现了 0 次）。

程序可通过任何可法代对象参数来创建 Counter 对象，此时 Counter 将会自动统计各元素出现的次数，并以元素为 key，出现的次数为 value 来构建 Counter 对象；程序也能以 dict 为参数来构建 Counter 对象；还能通过关键字参数来构建 Counter 对象。例如如下程序：

```
1.          from collections import Counter
2.          # 创建空的 Counter 对象
3.          c1 = Counter()
4.          # 以可迭代对象创建 Counter 对象
5.          c2 = Counter('hannah')
6.          print(c2)
7.          # 以可迭代对象创建 Counter 对象
```

```
c3 = Counter(['Python', 'Swift', 'Swift', 'Python', 'Kotlin', 'Python'])
print(c3)
# 以 dict 来创建 Counter 对象
c4 = Counter({'red': 4, 'blue': 2})
print(c4)
# 使用关键字参数的语法创建 Counter
c5 = Counter(Python=4, Swift=8)
print(c5)
```

运行上面程序，可以看到如下输出结果：

```
Counter({'h': 2, 'a': 2, 'n': 2})
Counter({'Python': 3, 'Swift': 2, 'Kotlin': 1})
Counter({'red': 4, 'blue': 2})
Counter({'Swift': 8, 'Python': 4})
```

事实上，Counter 继承了 dict 类，因此它完全可以调用 dict 所支持的方法。此外，Counter 还提供了如下三个常用的方法：

- elements()：该方法返回该 Counter 所包含的全部元素组成的迭代器。
- most_common([n])：该方法返回 Counter 中出现最多的 n 个元素。
- subtract([iterable-or-mapping])：该方法计算 Counter 的减法，其实就是计算减去之后各元素出现的次数。

下面程序示范了 Counter 类中这些方法的用法示例：

```
from collections import Counter
# 创建 Counter 对象
```

```
cnt = Counter()
# 访问并不存在的 key，将输出该 key 的次数为 0.
print(cnt['Python']) # 0
for word in ['Swift', 'Python', 'Kotlin', 'Kotlin', 'Swift', 'Go']:
    cnt[word] += 1
print(cnt)
# 只访问 Counter 对象的元素
print(list(cnt.elements()))
# 将字符串（迭代器）转换成 Counter
chr_cnt = Counter('abracadabra')
# 获取出现最多的 3 个字母
print(chr_cnt.most_common(3))  # [('a', 5), ('b', 2), ('r', 2)]
c = Counter(a=4, b=2, c=0, d=-2)
d = Counter(a=1, b=2, c=3, d=4)
# 用 Counter 对象执行减法，其实就是减少各元素的出现次数
c.subtract(d)
print(c) # Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
e = Counter({'x': 2, 'y': 3, 'z': -4})
# 调用 del 删除 key-value 对，会真正删除该 key-value 对
del e['y']
print(e)
# 访问'w'对应的 value，'w'没有出现过，因此返回 0
print(e['w']) # 0
# 删除 e['w']，删除该 key-value 对
del e['w']
# 再次访问'w'对应的 value，'w'还是没有，因此返回 0
print(e['w']) # 0
```

上面程序中第 10 行代码调用了 Counter 对象的 elements() 方法，该方法返回容器中所有元素组成的迭代器，Counter 记录了几个元素的出现次数，该方法就会返回几个对应的元素。

程序中第 14 行代码调用了 Counter 对象的 most_common(3) 方法，该方法会返回容

器中出现次数最多的三个元素。

程序中第 18 行代码调用了 Counter 对象的 subtract() 方法执行减法，实质上就是对元素出现的次数执行减法。

运行上面代码，可以看到如下输出结果：

```
0
Counter({'Swift': 2, 'Kotlin': 2, 'Python': 1, 'Go': 1})
['Swift', 'Swift', 'Python', 'Kotlin', 'Kotlin', 'Go']
[('a', 5), ('b', 2), ('r', 2)]
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
Counter({'x': 2, 'z': -4})
0
0
```

此外，对于 Counter 对象还有一些很常用的操作，比如把 Counter 对象转换成 set（集合）、list（列表）、dict（字典）等，程序还可对 Counter 执行加、减、交、并运算，对 Counter 进行求正、求负运算等。对 Counter 执行各种运算的含义如下：
加：将两个 Counter 对象中各 key 出现的次数相加，且只保留出现次数为正的元素。
减：将两个 Counter 对象中各 key 出现的次数相喊，且只保留出现次数为正的元素。
交：取两个 Counter 对象中都出现的 key 且各 key 对应的次数的最小数。
并：取两个 Counter 对象中各 key 对应的出现次数的最大数。
求正：只保留 Counter 对象中出现次数为 0 或正数的 key-value 对。
求负：只保留 Counter 对象中出现次数为负数的 key-value 对，并将出现次数改为正数。

下面程序示范了对 Counter 对象进行的这些常用操作：

```
from collections import Counter
# 创建 Counter 对象
c = Counter(Python=4, Swift=2, Kotlin=3, Go=-2)
# 统计 Counter 中所有出现次数的总和
print(sum(c.values())) # 7
# 将 Counter 转换为 list，只保留各 key
print(list(c)) # ['Python', 'Swift', 'Kotlin', 'Go']
# 将 Counter 转换为 set，只保留各 key
print(set(c)) # {'Go', 'Python', 'Swift', 'Kotlin'}
# 将 Counter 转换为 dict
print(dict(c)) # {'Python': 4, 'Swift': 2, 'Kotlin': 3, 'Go': -2}
# 将 Counter 转换为 list，列表元素都是(元素, 出现次数)组
list_of_pairs = c.items()
print(list_of_pairs) # dict_items([('Python', 4), ('Swift', 2), ('Kotlin', 3), ('Go', -2)])
# 将列表元素为(元素, 出现次数)组的 list 转换成 Counter
c2 = Counter(dict(list_of_pairs))
print(c2) # Counter({'Python': 4, 'Kotlin': 3, 'Swift': 2, 'Go': -2})
# 获取 Counter 中最少出现的 3 个元素
print(c.most_common()[:-4:-1]) # [('Go', -2), ('Swift', 2), ('Kotlin', 3)]
# 清空所有 key-value 对
c.clear()
print(c) # Counter()
c = Counter(a=3, b=1, c=-1)
d = Counter(a=1, b=-2, d=3)
# 对 Counter 执行加法
print(c + d)  # Counter({'a': 4, 'd': 3})
# 对 Counter 执行减法
print(c - d)  # Counter({'b': 3, 'a': 2})
Counter({'a': 2})
# 对 Counter 执行交运算
print(c & d) # Counter({'a': 1})
print(c | d) # Counter({'a': 3, 'd': 3, 'b': 1})
print(+c) # Counter({'a': 3, 'b': 1})
print(-d) # Counter({'b': 2})
```

上面程序演示了前面介绍的 Counter 的各种通用方法和运算符，并且还给出了每次运算的输出结果，读者可通过这些输出结果来理解 Counter 对象的方法和运算符的功能。

12.Python defaultdict

defaultdict 是 dict 的子类，因此 defaultdict 也可被当成 dict 来使用，dict 支持的功能，defaultdict 基本都支持。但它与 dict 最大的区别在于，如果程序试图根据不存在的 key 采访问 dict 中对应的 value，则会引发 KeyError 异常；而 defaultdict 则可以提供一个 default_factory 属性，该属性所指定的函数负责为不存在的 key 来生成 value。

通过下面程序进行对比：

```
1.          from collections import defaultdict
2.          my_dict = {}
3.          # 使用 int 作为 defaultdict 的 default_factory
4.          # 将 key 不存在时，将会返回 int()函数的返回值
5.          my_defaultdict = defaultdict(int)
6.          print(my_defaultdict['a']) # 0
7.          print(my_dict['a']) # KeyError
```

上面程序分别创建了空的 dict 对象和 defaultdict 对象，当程序试图访问 defaultdict 中不存在的 key 对应的 value 时，程序输出 defaultdict 的 default_factory 属性（int 函数）的返回值 0；如果程序试图访问 dict 中不存在的 key 对应的 value，就会引发 KeyError 异常。

假如程序中包含多个 key-value 对数据，在这些 key-value 对中有些 key 是重复的，程序希望对这些 key-value 对进行整理，key 对应一个 list，该 list 中包含这组数据中该 key 对应的所有 value。

下面先使用普通 dict 来完成这项工作：

```
1.          s = [('Python', 1), ('Swift', 2), ('Python', 3), ('Swift', 4), ('Python',
9)]
2.          d = {}
3.          for k, v in s:
4.              # setdefault()方法用于获取指定 key 对应的 value.
5.              # 如果该 key 不存在，则先将该 key 对应的 value 设置为默认值:[]
6.              d.setdefault(k, []).append(v)
7.          print(list(d.items()))
```

正如从上面第 6 行代码所看到的，如果使用普通 dict 来处理，就需要处理 key 不存在的

情况。程序中使用了 dict 的 setdefault() 方法，该方法用于获取指定 key 对应的 value，但如果该 key 不存在，setdefault() 方法就会先为该 key 设置一个默认的 value。

运行上面程序，可以看到如下输出结果：

```
[('Python', [1, 3, 9]), ('Swift', [2, 4])]
```

如果使用 defaultdict 来处理则简单得多，因为程序可以直接为 defaultdict 中不存在的 key 设置默认的 value。该处理程序如下：

纯文本复制

```
from collections import defaultdict
s = [('Python', 1), ('Swift', 2), ('Python', 3), ('Swift', 4), ('Python', 9)]
# 创建 defaultdict，设置由 list()函数来生成默认值
d = defaultdict(list)
for k, v in s:
    # 直接访问 defaultdict 中指定 key 对应的 value 即可。
    # 如果该 key 不存在，defaultdict 会自动为该 key 生成默认值
    d[k].append(v)
print(list(d.items()))
```

对比该程序中的第 8 行代码和前一个程序中的第 6 行代码，不难发现使用 defaultdict 更加方便，原因是程序直接访问 defaultdict 中指定的 key 对应的 value，如果该 key 不存在，程序在创建 defaultdict 时传入的 list 函数将会为之生成默认的 value。

13.Python namedtuple

namedtuple() 是一个工厂函数，使用该函数可以创建一个 tuple 类的子类，该子类可以为 tuple 的每个元素都指定字段名，这样程序就可以根据字段名来访问 namedtuple 的各元素了。当然，如果有需要，程序依然可以根据索引来访问 namedtuple 的各元素。

namedtuple 是轻量级的 ,性能很好 ,其并不比普通 tuple 需要更多的内存。namedtuple 函数的语法格式如下：

```
namedtuple(typename, field_names, *, verbose=False, rename=False,
module=None)
```

关于该函数参数的说明如下；

- typename :该参数指定所创建的 tuple 子类的类名 ,相当于用户定义了一个新类。
- field_names：该参数是一个字符串序列，如 ['x'，'y']。此外，field_names 也可直接使用单个字符串代表所有字段名，多个字段名用空格、逗号隔开，如 'x y' 或 'x,y'。任何有效的 Python 标识符都可作为字段名(不能以下画线开头)。有效的标识符可由字母、数字、下画线组成，但不能以数字、下面线开头，也不能是关键字 (如 return、global、pass、raise 等) 。
- rename：如果将该参数设为 True，那么无效的字段名将会被自动替换为位置名。例如指定 ['abc','def','ghi','abc'] ,它将会被替换为 ['abc', '_1','ghi','_3'] ,这是因为 def 字段名是关键字，而 abc 字段名重复了。
- verbose：如果该参数被设为 True，那么当该子类被创建之后，该类定义就会被立即打印出来。
- module：如果设置了该参数，那么该类将位于该模块下，因此该自定义类的 __module__ 属性将被设为该参数值。

下面程序示范了使用 namedtuple 工厂函数来创建命名元组：

```
from collections import namedtuple
# 定义命名元组类：Point
Point = namedtuple('Point', ['x', 'y'])
# 初始化 Point 对象，即可用位置参数，也可用命名参数
p = Point(11, y=22)
# 像普通元组一样用根据索引访问元素
print(p[0] + p[1]) # 33
# 执行元组解包，按元素的位置解包
a, b = p
print(a, b) # 11, 22
# 根据字段名访问各元素
print(p.x + p.y) # 33
print(p) # Point(x=11, y=22)
```

上面程序先创建了一个命名元组类 Point，它是 tuple 的子类，因此程序既可使用命名参数的方式设置元组成员，也可使用索引的方式设置元组成员。

在创建了 Point 对象之后，Point 就代表一个命名元组，程序既可使用普通元组的方式来访问元组的成员，也可使用字段名来访问元组的成员。

除上面介绍的用法之外，Python 还为命名元组提供了如下方法和属性：

- _make(iterable)：类方法。该方法用于根据序列或可迭代对象创建命名元组对象。
- _asdict()：将当前命名元组对象转换为 OrderedDict 字典。
- _replace(**kwargs)：替换命名元组中一个或多个字段的值。
- _source：该属性返回定义该命名元组的源代码。
- _fields：该属性返回该命名元组中所有字段名组成的元组。

下面代码示范了上面方法和属性的用法：

纯文本复制

```
my_data = ['East', 'North']
# 创建命名元组对象
p2 = Point._make(my_data)
print(p2) # Point(x='East', y='North')
# 将命名元组对象转换成 OrderedDict
print(p2._asdict()) # OrderedDict([('x', 'East'), ('y', 'North')])
# 替换命名元组对象的字段值
p2._replace(y='South')
print(p2) # Point(x='East', y='North')
# 输出 p2 包含的所有字段
print(p2._fields) # ('x', 'y')
# 再次定义一个命名元组类
Color = namedtuple('Color', 'red green blue')
# 再次定义命名元组类，其字段由 Point 的字段加上 Color 的字段
Pixel = namedtuple('Pixel', Point._fields + Color._fields)
# 创建 Pixel 对象，分别为 x、y、red、green、blue 字段赋值
pix = Pixel(11, 22, 128, 255, 0)
print(pix) # Pixel(x=11, y=22, red=128, green=255, blue=0)
```

运行上面程序， 可以看到如下输出结果：

```
33

11 22

33

Point(x=11, y=22)

Point(x='East', y='North')

OrderedDict([('x', 'East'), ('y', 'North')])
```

```
Point(x='East', y='North')

('x', 'y')

Pixel(x=11, y=22, red=128, green=255, blue=0)
```

14.Python OrderedDict

OrderedDict 也是 dict 的子类，其最大特征是，它可以“维护”添加 key-value 对的顺序。简单来说，就是先添加的 key-value 对排在前面，后添加的 key-value 对排在后面。

由于 OrderedDict 能维护 key-value 对的添加顺序，因此即使两个 OrderedDict 中的 key-value 对完全相同，但只要它们的顺序不同，程序在判断它们是否相等时也依然会返回 false。

例如如下程序：

```
1.          from collections import OrderedDict
2.          # 创建 OrderedDict 对象
3.          dx = OrderedDict(b=5, c=2, a=7)
4.          print(dx) # OrderedDict([('b', 5), ('c', 2), ('a', 7)])
5.          d = OrderedDict()
6.          # 向 OrderedDict 中添加 key-value 对
7.          d['Python'] = 89
8.          d['Swift'] = 92
9.          d['Kotlin'] = 97
10.         d['Go'] = 87
11.         # 遍历 OrderedDict 的 key-value 对
12.         for k,v in d.items():
13.             print(k, v)
```

上面程序首先创建了 OrderedDict 对象，接下来程序向其中添加了 4 个 key-value 对，OrderedDict 完全可以“记住”它们的添加顺序。运行该程序，可以看到如下输出结果：

```
OrderedDict([('b', 5), ('c', 2), ('a', 7)])
Python 89
Swift 92
Kotlin 97
Go 87
```

正如前面所说的，两个 OrderedDict 中即使包含的 key-value 对完全相同，但只要它们的顺序不同，程序也依然会判断出两个 OrderedDict 是不相等的。例如如下程序：

```
        # 创建普通的 dict 对象
        my_data = {'Python': 20, 'Swift':32, 'Kotlin': 43, 'Go': 25}
        # 创建基于 key 排序的 OrderedDict
        d1 = OrderedDict(sorted(my_data.items(), key=lambda t: t[0]))
        # 创建基于 value 排序的 OrderedDict
        d2 = OrderedDict(sorted(my_data.items(), key=lambda t: t[1]))
        print(d1) # OrderedDict([('Go', 25), ('Kotlin', 43), ('Python', 20), ('Swift', 32)])
        print(d2) # OrderedDict([('Python', 20), ('Go', 25), ('Swift', 32), ('Kotlin', 43)])
        print(d1 == d2) # False
```

上面程序先创建了一个普通的 dict 对象，该对象中包含 4 个 key-value 对；接下来程序分别使用 sorted() 函数对 my_data（dict 对象）的 items 进行排序：d1 是按 key 排序的；d2 是按 value 排序的，这样得到的 d1、d2 两个 OrderedDict 中的 key-value 对是一样的，只不过顺序不同。

运行上面程序，可以看到如下输出结果：

```
OrderedDict([('Go', 25), ('Kotlin', 43), ('Python', 20), ('Swift', 32)])
OrderedDict([('Python', 20), ('Go', 25), ('Swift', 32), ('Kotlin', 43)])
False
```

从上面的输出结果可以看到，虽然两个 OrderedDict 所包含的 key-value 对完全相同，但由于它们的顺序不同，因此程序判断它们不相等。

此外，由于 OrderedDict 是有序的，因此 Python 为之提供了如下两个方法：

1. popitem(last=True)：默认弹出并返回最右边（最后加入）的 key-value 对；如果将 last 参数设为 False，则弹出并返回最左边（最先加入）的 key-value 对。
2. move_to_end(key, last=True)：默认将指定的 key-value 对移动到最右边（最后加入）；如果将 last 改为 False，则将指定的 key-value 对移动到最左边（最先加入）。

下面程序示范了 OrderedDict 的两个方法的用法：

```
from collections import OrderedDict
d = OrderedDict.fromkeys('abcde')
# 将 b 对应的 key-value 对移动到最右边（最后加入）
d.move_to_end('b')
print(d.keys()) # odict_keys(['a', 'c', 'd', 'e', 'b'])
# 将 b 对应的 key-value 对移动到最左边（最先加入）
d.move_to_end('b', last=False)
print(d.keys()) # odict_keys(['b', 'a', 'c', 'd', 'e'])
```

```
9.          # 弹出并返回最右边（最后加入）的 key-value 对
10.         print(d.popitem()[0]) # e
11.         # 弹出并返回最左边（最先加入）的 key-value 对
12.         print(d.popitem(last=False)[0]) # b
```

运行上面程序，可以看到如下输出结果：

```
odict_keys(['a', 'c', 'd', 'e', 'b'])
odict_keys(['b', 'a', 'c', 'd', 'e'])
e
b
```

通过上面的输出结果可以看出，使用 OrderedDict 的 move_to_end() 方法可以方便地将指定的 key-value 对移动到 OrderedDict 的任意一端；而 popitem() 方法则可用于弹出并返回 OrderedDict 任意一端的 key-value 对。

15.Python itertools

itertools 模块中主要包含了一些用于生成迭代器的函数。在 Python 的交互式解释器中先导入 itertools 模块，然后输入 [e for e in dir(itertools) if not e.startswith('_')] 命令，即可看到该模块所包含的全部属性和函数：

```
>>> [e for e in dir(itertools) if not e.startswith('_')]
['accumulate', 'chain', 'combinations', 'combinations_with_replacement',
'compress', 'count', 'cycle', 'dropwhile', 'filterfalse', 'groupby', 'islice', 'permutations',
'product', 'repeat', 'starmap', 'takewhile', 'tee', 'zip_longest']
```

从上面的输出结果可以看出，itertools 模块中的不少函数都可以用于生成迭代器。

先看 itertools 模块中三个生成无限迭代器的函数：

1. count(start, [step])：生成 start, start+step, start+2*step，... 的迭代器，其中 step 默认为 1。比如使用 count(10) 生成的迭代器包含：10, 11 , 12 , 13, 14 , ...。
2. cycle(p)：对序列 p 生成无限循环 $p_0, p_1,..., p_0, p_1,...$ 的迭代器。比如使用 cycle('ABCD') 生成的迭代器包含：A,B,C,D,A,B,C,D,...。
3. repeat(elem [,n])：生成无限个 elem 元素重复的迭代器，如果指定了参数 n，则只生成 n 个 elem 元素。比如使用 repeat(10, 3) 生成的法代器包含：10, 10, 10。

下面程序示范了使用上面三个函数来生成迭代器：

```
import itertools as it
# count(10, 3)生成 10、13、16……迭代器
for e in it.count(10, 3):
    print(e)
    # 用于跳出无限循环
    if e > 20:
        break
print('---------')
my_counter = 0
# cycle 用于对序列生成无限循环的迭代器
for e in it.cycle(['Python', 'Kotlin', 'Swift']):
    print(e)
    # 用于跳出无限循环
    my_counter += 1
    if my_counter > 7:
        break
```

```
17.         print('---------')
18.         # repeat 用于生成 n 个元素重复的迭代器
19.         for e in it.repeat('Python', 3):
20.             print(e)
```

在 itertools 模块中还有一些常用的迭代器函数，如下所示：

- accumulate(p[,func])：默认生成根据序列 p 元素累加的迭代器，p_0, p_0+p_1, $p_0+p_1+p_2$，...序列，如果指定了 func 函数，则用 func 函数来计算下一个元素的值。
- chain(p, q, ...)：将多个序列里的元素“链”在一起生成新的序列。
- compress(data, selectors)：根据 selectors 序列的值对 data 序列的元素进行过滤。如果 selector[0] 为真，则保留 data[0]；如果 selector[1] 为真，则保留 data[1]..... 依此类推。
- dropwhile(pred, seq)：使用 pred 函数对 seq 序列进行过滤，从 seq 中第一个使用 pred 函数计算为 False 的元素开始，保留从该元素到序列结束的全部元素。
- takewhile(pred, seq)：该函数和上一个函数恰好相反。使用 pred 函数对 seq 序列进行过滤，从 seq 中第一个使用 pred 函数计算为 False 的元素开始，去掉从该元素到序列结束的全部元素。
- filterfalse(pred, seq)：使用 pred 函数对 seq 序列进行过滤，保留 seq 中使用 pred 计算为 True 的元素。比如 filterfalse(lambda x:x%2, range(10))，得到 0, 2, 4, 6, 8。
- islice(seq, [start,] stop [, step])：其功能类似于序列的 slice 方法，实际上就是返回 seq[start:stop:step] 的结果。
- starmap(func, seq)：使用 func 对 seq 序列的每个元素进行计算，将计算结果作为新的序列元素。当使用 func 计算序列元素时，支持序列解包。比如 seq 序列的元素长

度为 3，那么 func 可以是一个接收三个参数的函数，该函数将会根据这三个参数来计算新序列的元素。

- zip_longest(p,q,...)：将 p、q 等序列中的元素按索引合并成元组，这些元组将作为新序列的元素。

上面这些函数的测试程序如下：

```
import itertools as it
# 默认使用累加的方式计算下一个元素的值
for e in it.accumulate(range(6)):
    print(e, end=', ') # 0, 1, 3, 6, 10, 15
print('\n---------')
# 使用 x*y 的方式来计算迭代器下一个元素的值
for e in it.accumulate(range(1, 6), lambda x, y: x * y):
    print(e, end=', ') # 1, 2, 6, 24, 120
print('\n---------')
# 将两个序列“链”在一起，生成新的迭代器
for e in it.chain(['a', 'b'], ['Kotlin', 'Swift']):
    print(e, end=', ') # 'a', 'b', 'Kotlin', 'Swift'
print('\n---------')
# 根据第二个序列来筛选第一个序列的元素，
# 由于第二个序列只有中间两个元素为 1（True），因此前一个序列只保留中间两个元素
for e in it.compress(['a', 'b', 'Kotlin', 'Swift'], [0, 1, 1, 0]):
    print(e, end=', ') # 只有: 'b', 'Kotlin'
print('\n---------')
# 获取序列中从长度不小于 4 的元素开始、到结束的所有元素
for e in it.dropwhile(lambda x:len(x)<4, ['a', 'b', 'Kotlin', 'x', 'y']):
    print(e, end=', ') # 只有: 'Kotlin', 'x', 'y'
print('\n---------')
# 去掉序列中从长度不小于 4 的元素开始、到结束的所有元素
for e in it.takewhile(lambda x:len(x)<4, ['a', 'b', 'Kotlin', 'x', 'y']):
```

```
            print(e, end=', ')  # 只有: 'a', 'b'
        print('\n---------')
        # 只保留序列中从长度不小于 4 的元素
        for e in it.filterfalse(lambda x:len(x)<4, ['a', 'b', 'Kotlin', 'x',
'y']):
            print(e, end=', ') # 只有: 'Kotlin'
        print('\n---------')
        # 使用 pow 函数对原序列的元素进行计算，将计算结果作为新序列的元素
        for e in it.starmap(pow, [(2,5), (3,2), (10,3)]):
            print(e, end=', ') # 32, 9, 1000
        print('\n---------')
        # 将'ABCD'、'xy'的元素按索引合并成元组，这些元组作为新序列的元素
        # 长度不够的序列元素使用'-'字符代替
        for e in it.zip_longest('ABCD', 'xy', fillvalue='-'):
            print(e, end=', ') # ('A', 'x'), ('B', 'y'), ('C', '-'), ('D', '-')
```

运行上面程序，可以看到如下输出结果：

```
0, 1, 3, 6, 10, 15,
---------
1, 2, 6, 24, 120,
---------
a, b, Kotlin, Swift,
---------
b, Kotlin,
---------
Kotlin, x, y,
---------
a, b,
```

```
---------

Kotlin,

---------

32, 9, 1000,

---------

('A', 'x'), ('B', 'y'), ('C', '-'), ('D', '-'),
```

在 itertools 模块中还有一些用于生成排列组合的工具函数：

- product(p, q, ...[repeat= 1)]：用序列 p 、q 、... 中的元素进行排列组合，就相当于使用嵌套循环组合。
- permutations(p[, r])：从序列 p 中取出 r 个元素组成全排列，将排列得到的元组作为新迭代器的元素。
- combinations(p, r)：从序列 p 中取出 r 个元素组成全组合，元素不允许重复，将组合得到的元组作为新迭代器的元素。
- combinations with_replacement(p, r)，从序列 p 中取出 r 个元素组成全组合，元素允许重复，将组合得到的元组作为新迭代器的元素。

如下程序示范了上面 4 个函数的用法：

```
1.          import itertools as it
2.          # 使用两个序列进行排列组合
3.          for e in it.product('AB', 'CD'):
4.              print(''.join(e), end=', ') # AC, AD, BC, BD,
```

```
5.          print('\n---------')
6.          # 使用一个序列、重复 2 次进行全排列
7.          for e in it.product('AB', repeat=2):
8.              print(''.join(e), end=', ') # AA, AB, BA, BB,
9.          print('\n---------')
10.         # 从序列中取 2 个元素进行排列
11.         for e in it.permutations('ABCD', 2):
12.             print(''.join(e), end=', ') # AB, AC, AD, BA, BC, BD, CA, CB, CD, DA,
    DB, DC,
13.         print('\n---------')
14.         # 从序列中取 2 个元素进行组合、元素不允许重复
15.         for e in it.combinations('ABCD', 2):
16.             print(''.join(e), end=', ') # AB, AC, AD, BC, BD, CD,
17.         print('\n---------')
18.         # 从序列中取 2 个元素进行组合、元素允许重复
19.         for e in it.combinations_with_replacement('ABCD', 2):
20.             print(''.join(e), end=', ') # AA, AB, AC, AD, BB, BC, BD, CC, CD, DD,
```

上面程序用到了一个字符串的 join() 方法，该方法用于将元组的所有元素连接成一个字符串。运行程序，可以看到如下输出结果：

```
AC, AD, BC, BD,

---------

AA, AB, BA, BB,

---------

AB, AC, AD, BA, BC, BD, CA, CB, CD, DA, DB, DC,

---------

AB, AC, AD, BC, BD, CD,
```

```
---------

AA, AB, AC, AD, BB, BC, BD, CC, CD, DD,
```

16.Python functools

functools 模块中主要包含了一些函数装饰器和便捷的功能函数。在 Python 的交互式解释器中先导入 functools 模块，然后输入 [e for e in dir(functools) if not e.startswith('_')] 命令，即可看到该模块所包含的全部属性和函数：

```
>>> [e for e in dir(functools) if not e.startswith('_')]
['MappingProxyType', 'RLock', 'WRAPPER_ASSIGNMENTS', 'WRAPPER_UPDATES',
'WeakKeyDictionary', 'cmp_to_key', 'get_cache_token', 'lru_cache', 'namedtuple',
'partial', 'partialmethod', 'recursive_repr', 'reduce', 'singledispatch', 'total_ordering',
'update_wrapper', 'wraps']
```

在 functools 模块中常用的函数装饰器和功能函数如下：

- functools.cmp_to_key(func)：将老式的比较函数（func）转换为关键字函数（key function）。在 Python 3 中比较大小、排序都是基于关键字函数的，Python 3 不支持老式的比较函数。
- @functools.lru_cache(maxsize=128, typed=False) :该函数装饰器使用 LRU（最近最少使用）缓存算法来缓存相对耗时的函数结果，避免传入相同的参数重复计算。同时，

缓存并不会无限增长，不用的缓存会被释放。其中 maxsize 参数用于设置缓存占用的最大字节数，typed 参数用于设置将不同类型的缓存结果分开存放。

- @functools.total_ordering：这个类装饰器（作用类似于函数装饰器，只是它用于修饰类）用于为类自动生成比较方法。通常来说，开发者只要提供 __lt__()、__le__()、__gt__()、__ge__() 其中之一（最好能提供 __eq__() 方法），@functools.total_ordering 装饰器就会为该类生成剩下的比较方法。
- functools.partial(func, *args, **keywords) :该函数用于为 func 函数的部分参数指定参数值，从而得到一个转换后的函数，程序以后调用转换后的函数时，就可以少传入那些已指定值的参数。
- functools.partialmethod(func, *args, **keywords)：该函数与上一个函数的含义完全相同，只不过该函数用于为类中的方法设置参数值。
- functools.reduce(function, iterable[, initializer])：将初始值（默认为 0，可由 initializer 参数指定）、迭代器的当前元素传入 function 函数，将计算出来的函数结果作为下一次计算的初始值、迭代器的下一个元素再次调用 function 函数……依此类推，直到迭代器的最后一个元素。
- @functools.singledispatch：该函数装饰器用于实现函数对多个类型进行重载。比如同样的函数名称，为不同的参数类型提供不同的功能实现。该函数的本质就是根据参数类型的变换，将函数转向调用不同的函数。
- functools.update_wrapper(wrapper, wrapped, assigned=WRAPPER_ASSIGNMENTS, updated=WRAPPER_UPDATES)：对 wrapper 函数进行包装，使之看上去就像 wrapped（被包装）函数。

- @functools.wraps(wrapped, assigned=WRAPPER_ASSIGNMENTS, updated=WRAPPER_UPDATES)：该函数装饰器用于修饰包装函数，使包装函数看上去就像 wrapped 函数。

> 通过介绍不难发现，functools.update_wrapper 和 @functools.wraps 的功能是一样的，只不过前者是函数，因此需要把包装函数作为第一个参数传入；而后者是函数装饰器，因此使用该函数装饰器修饰包装函数即可，无须将包装函数作为第一个参数传入。

下面程序示范了 functiontools 模块中部分函数或函数装饰器的用法：

```
from functools import *
# 以初始值（默认为 0）为 x，以当前序列元素为 y，x+y 的和作为下一次的初始值
print(reduce(lambda x,y: x + y, range(5))) # 10
print(reduce(lambda x,y: x + y, range(6))) # 15
# 设初始值为 10
print(reduce(lambda x,y: x + y, range(6), 10)) # 25
print('----------------')
class User:
    def __init__(self, name):
        self.name = name
    def __repr__(self):
        return 'User[name=%s' % self.name
# 定义一个老式的大小比较函数，User 的 name 越长，该 User 越大
def old_cmp(u1 , u2):
    return len(u1.name) - len(u2.name)
my_data = [User('Kotlin'), User('Swift'), User('Go'), User('Java')]
# 对 my_data 排序，需要关键字参数（调用 cmp_to_key 将 old_cmp 转换为关键字参数
my_data.sort(key=cmp_to_key(old_cmp))
```

```
19.         print(my_data)
20.         print('-----------------')
21.         @lru_cache(maxsize=32)
22.         def factorial(n):
23.             print('~~计算%d 的阶乘~~' % n)
24.             if n == 1:
25.                 return 1
26.             else:
27.                 return n * factorial(n - 1)
28.         # 只有这行会计算，然后会缓存 5、4、3、2、1 的解乘
29.         print(factorial(5))
30.         print(factorial(3))
31.         print(factorial(5))
32.         print('-----------------')
33.         # int 函数默认将 10 进制的字符串转换为整数
34.         print(int('12345'))
35.         # 为 int 函数的 base 参数指定参数值
36.         basetwo = partial(int, base=2)
37.         basetwo.__doc__ = '将二进制的字符串转换成整数'
38.         # 相当于执行 base 为 2 的 int()函数
39.         print(basetwo('10010'))
40.         print(int('10010', 2))
```

上面程序中第 3 行代码调用 reduce() 函数来计算序列的“累计”结果，在调用该函数时传入的第一个参数（函数）决定了累计算法，此处使用的累计算法是“累加”。

程序第 18 行代码调用 cmp_to_ key() 函数将老式的大小比较函数（old_cmp）转换为关键字函数，这样该关键字函数即可作为列表对象的 sort() 方法的参数。

程序第 21 行代码调用 @lru_cache 对函数结果进行缓存，后面程序第一次执行 factorial(5) 时将会看到执行结果；但接下来调用 factorial(3)、factorial(5) 时都不会看到

执行结果，因为它们的结果已被缓存起来。

程序第 36 行代码调用 paitial() 函数为 int() 函数的 base 参数绑定值“2”，这样程序以后调用该函数时实际上就相当于调用 base 为 2 的 int() 函数。所以，上面程序中最后两行代码的本质是完全一样的。

partialmethod() 与 partial() 函数的作用基本相似，区别只是 partial() 函数用于为函数的部分参数绑定值；而 partialmethod() 函数则用于为类中方法的部分参数绑定值。如下程序示范了 partialmethod() 函数的用法：

```
from functools import *
class Cell:
    def __init__(self):
        self._alive = False
    # @property 装饰器指定该方法可使用属性语法访问
    @property
    def alive(self):
        return self._alive
    def set_state(self, state):
        self._alive = bool(state)
    # 指定 set_alive()方法就是将 set_state()方法的 state 参数指定为 True
    set_alive = partialmethod(set_state, True)
    # 指定 set_dead()方法就是将 set_state()方法的 state 参数指定为 False
    set_dead = partialmethod(set_state, False)
c = Cell()
print(c.alive)
# 相当于调用 c.set_state(True)
c.set_alive()
print(c.alive)
# 相当于调用 c.set_state(False)
```

```
        c.set_dead()
        print(c.alive)
```

上面程序定义了一个 Cell（细胞）类，在该类中定义了一个 set_state() 方法，该方法用于设置该细胞的状态。程序接下来使用 partialmethod() 函数为 set_state() 方法绑定了参数值；将 set_state() 方法的参数值绑定为 True 之后赋值给了 set_alive() 方法；将 set_state() 方法的参数值绑定为 False 之后赋值给了 set_dead() 方法。

因此 ,程序调用 c.set_alive() 就相当于调用 c.set_state(True) ;程序调用 c.set_dead() 就相当于调用 c.set_state(False)。

下面程序示范了 @total_ordering 类装饰器的作用：

```
from functools import *
#@total_ordering
class User:
    def __init__(self, name):
        self.name = name
    def __repr__(self):
        return 'User[name=%s' % self.name
    # 根据是否有 name 属性来决定是否可比较
    def _is_valid_operand(self, other):
        return hasattr(other, "name")
    def __eq__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        # 根据 name 判断是否相等（都转成小写比较、忽略大小写）
        return self.name.lower()  == other.lastname.lower()
    def __lt__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
```

```
19.                 # 根据 name 判断是否相等（都转成小写比较、忽略大小写）
20.                 return self.lastname.lower() < other.lastname.lower()
21.         # 打印被装饰之后的 User 类中的__gt__方法
22.         print(User.__gt__)
```

上面程序定义了一个 User 类，并为该类提供了 __eq__、__lt__ 两个比较大小的方法。程序中使用 @total_ordering 装饰器修饰了该 User 类，这样该装饰器将会为该类提供 __gt__、__ge__、__le__、__ne__ 这些比较方法。上面程序中最后一行输出了 User 类的 __gt__ 方法。运行该程序，可以看到如下输出结果：

```
<function _gt_from_lt at 0x00000000028C2D90>
```

从上面的输出结果可以看到，此时的 __gt__ 方法是根据 __lt__ 方法“生产”出来的。

但如果将上面程序中 @total_ordering 注释掉，再次运行该程序，则可以看到如下输出结果：

```
<slot wrapper '__gt__' of 'object' objects>
```

从上面的输出结果可以看到，此时该 __gt__ 方法其实来自父类 object。

@singledispatch 函数装饰器的作用是根据函数参数类型转向调用另一个函数，从而实现函数重载的功能。例如，如下程序示范了该函数装饰器的用法：

```
1.         from functools import *
2.         @singledispatch
3.         def test(arg, verbose):
```

```
4.              if verbose:
5.                  print("默认参数为：", end=" ")
6.              print(arg)
7.          # 限制 test 函数第一个参数为 int 型的函数版本
8.          @test.register(int)
9.          def _(argu, verbose):
10.             if verbose:
11.                 print("整型参数为：", end=" ")
12.             print(argu)
13.         test('Python', True)  # ①
14.         # 调用第一个参数为 int 型的版本
15.         test(20, True)  # ②
```

上面程序中第 2 行代码使用 @singledispatch 装饰器修饰了 test() 函数，接下来程序即可通过 test() 函数的 register() 方法来注册被转向调用的函数。第 8 行代码使用 @test.register(int) 修饰了目标函数，这意味着如果 test() 函数的第一个参数为 int 类型，实际上则会转向调用被 @test.register(int) 修饰的函数。

使用 @singledispatch 装饰器修饰之后的函数就有了 register() 方法，该方法用于为指定类型注册被转向调用的函数。

程序中 ① 号代码在调用 test() 函数时第一个参数是 str 类型，因此程序依然调用 test() 函数本身；程序中 ② 号代码在调用 test() 函数时第一个参数是 int 类型，因此将会转向调用被 @test.register(int) 修饰的函数。

运行上面程序，可以看到如下输出结果：

```
默认参数为：Python

整型参数为：20
```

程序还可继续使用 @test.register() 装饰器来绑定被转向调用的函数。例如如下代码：

```
1.          # 限制 test 函数第一个参数为 list 型的函数版本
2.          @test.register(list)
3.          def _(argb, verbose=False):
4.              if verbose:
5.                  print("列表中所有元素为:")
6.              for i, elem in enumerate(argb):
7.                  print(i, elem, end=" ")
8.          test([20, 10, 16, 30, 14], True) # ③
9.          print("\n---------------")
```

上面第 2 行代码显示 test() 函数的第一个参数是 list 时将转向调用被 @test.register(list) 修饰的函数。而上面程序中 ③ 号代码在调用 test() 函数时第一个参数是 list 对象，因此这行代码将会转向调用被 @test.register(list) 修饰的函数。运行上面代码，将看到如下输出结果：

```
0 20 1 10 2 16 3 30 4 14

---------------
```

此外，程序也可使用 register（类型，被转向调用的函数）方法来执行绑定。这种方式与前面使用函数装饰器的本质是一样的，只不过这种语法没有修饰被转向调用的函数，因此额外多传入一个参数。例如如下代码：

```
1.          print("\n---------------")
2.          # 定义一个函数，不使用函数装饰器修饰
3.          def nothing(arg, verbose=False):
```

```
        print("~~None 参数~~")
    # 当 test 函数第一个参数为 None 类型时，转向为调用 nothing 函数
    test.register(type(None), nothing)
    test(None, True) # ④
    print("\n----------------")
```

上面程序中指定调用 test() 函数的第一个参数为 None 类型时，程序将会转向调用 nothing 函数。而 ④ 号代码在调用 test() 函数时第一个参数是 None，因此这行代码将会转向调用 nothing 函数。运行上面代码，可以看到如下输出结果：

```
~~None 参数~~


---------------
```

此外，@singledispatch 也允许为参数的多个类型绑定同一个被转向调用的函数：只要使用多个 @ 函数名.register() 装饰器即可。例如如下代码：

```
    from decimal import Decimal
    # 限制 test 函数第一个参数为 float 或 Decimal 型的函数版本
    @test.register(float)
    @test.register(Decimal)
    def test_num(arg, verbose=False):
        if verbose:
            print("参数的一半为:", end=" ")
        print(arg / 2)
```

程序中使用 @test.register(float)、@test.register(Decimal) 修饰 test_num 函数，这意味着程序在调用 test() 函数时无论第一个参数是 float 类型还是 Decimal 类型，其实都会转向调用 test_num 函数。

当程序为 @singledispatch 函数执行绑定之后，程序就可以通过该函数的 dispatch（类型）方法来找到该类型所对应转向的函数。例如如下代码：

```
        # test.dispatch(类型)即可获取它转向的函数
        # 当 test()函数第一个参数为 float 时将转向到调用 test_num
        print(test_num is test.dispatch(float)) # True
        # 当 test()函数第一个参数为 Decimal 时将转向到调用 test_num
        print(test_num is test.dispatch(Decimal)) # True
        # 直接调用 test 并不等于 test_num
        print(test_num is test) # False
```

由于程序在调用 test() 函数时无论第一个参数类型是 float 还是 Decimal ,都会转向调用 test_num 函数，因此 test.dispatch(float) 和 test.dispatch(Decimal) 其实就是 test_num 函数。运行上面代码，将看到如下输出结果：

```
True
True
False
```

此外，如果想访问 @singledispatch 函数所绑定的全部类型及对应的 dispatch 函数，则可通过该函数的只读属性 registry 来实现，该属性相当于一个只读的 dict 对象。例如如下代码：

```
        # 获取 test 函数所绑定的全部类型
        print(test.registry.keys())
        # 获取 test 函数为 int 类型绑定的函数
        print(test.registry[int])
```

运行上面代码，可以看到如下输出结果：

```
dict_keys([<class 'object'>, <class 'int'>, <class 'list'>, <class 'NoneType'>, <class 'decimal.Decimal'>, <class 'float'>])
<function _ at 0x0000025910082730>
```

@wraps(wrapped_func) 函数装饰器与 update_wrapper(wrapper, wrapped_func) 函数的作用是一样的，都用于让包装函数看上去就像被包装函数（主要就是让包装函数的 __name__、__doc__ 属性与被包装函数保持一致）。区别是 @wraps(wrapped_ func) 函数装饰器直接修饰包装函数，因此不需要传入包装函数作为参数；而 update_wrapper(wrapper, wrapped_func) 则需要同时传入包装函数、被包装函数作为参数。

如下程序示范了 @wraps(wrapped_func) 函数装饰器的用法：

```
from functools import wraps

def fk_decorator(f):
    # 让 wrapper 函数看上去就像 f 函数
    @wraps(f)
    def wrapper(*args, **kwds):
        print('调用被装饰函数')
        return f(*args, **kwds)
    return wrapper
@fk_decorator
def test():
    """test 函数的说明信息"""
    print('执行 test 函数')
test()
print(test.__name__)
print(test.__doc__)
```

程序中第 5 行代码的作用是让被包装函数（wrapper）就像 f 函数。程序使用 @fk_decorator 修饰 test() 函数，因此在调用 test() 函数时，实际上是调用 fk_decorator 的返回值 wrapper 函数（这是前面的函数装饰器的功能）。

也就是说，上面程序中最后三行代码看上去是访问 test 函数，实际上是访问 wrapper 函数。由于程序使用 @wraps(f) 修饰了 wrapper 函数，因此该函数看上去就像 test 函数。所以，程序在输出 test.__name__ 和 test.__doc__ 时（注意此处的 test 其实是 wrapper 函数），输出的依然是 test 函数的函数名、描述文档。

运行上面代码，将看到如下输出结果：

```
调用被装饰函数
执行 test 函数
test
test 函数的说明信息
```

如果注释掉程序中第 5 行代码，此时将不能让 wrapper 函数看上去像 test 函数。如果再次运行该程序，将看到如下输出结果：

```
调用被装饰函数
执行 test 函数
wrapper
None
```

第 13 章 Tkinter

Python GUI 开发也叫 Python 图形界面开发，通俗地讲，就是开发出带界面的客户端程序，像 QQ、迅雷一样。
Python GUI 库有很多，大部分都是第三方提供的，并且有各自的特点，这给开发者造成了“选择障碍”，所以，本专题干脆以 Python 内置的 Tkinter 库来讲解图形界面开发。
这套 Python Tkinter 教程通俗易懂，简洁明了，同时包含了大量实例，你的学习将会非常轻松逾越。

1.什么是 GUI，Python GUI 常用库汇总

前面介绍的所有程序都是基于命令行的，这些程序只有专业的计算机人士才会使用，用户更青睐于有直观操作界面的程序，通过拖动鼠标、单击等动作就可以操作整个应用。实际上，Windows 之所以广为人知，其最初的吸引力就是来自它所提供的图形用户界面。
作为一个程序设计者，必须优先考虑用户的感受，一定要让用户感到“爽”，程序才会被需要、被使用，这样的程序才有价值。
GUI，是 Graphics User Interface（图形用户界面）的缩写。在 GUI 中，并不只是输入文本和返回文本，用户可以看到窗口、按钮、文本框等组件，还可以通过鼠标和键盘操作应用。

GUI 是程序交互的一种不同的方式，使用 GUI 开发的程序，和命令行程序一样，都具有输入数据、处理数据和输出数据这 3 个基本要素，只不过，使用 GUI 开发的程序，它们的输入和输出方式更丰富，更有趣。

值得一提的是，Python GUI 库有很多，表 1 给大家罗列了常用的几种 GUI 库。

表 1 Python GUI 常用库

库名	特点
wxPython	wxPython 是一个跨平台的 GUI 工具集，以流行的 wxWidgets（原名 wxWindows）为基础，提供了良好的跨平台外观。简单来说，wxPython 在 Windows 上调用 Windows 的本地组件、在 Mac OS 上调用 Mac OS X 的本地组件、在 Linux 上调用 Linux 的本地组件，这样可以让 GUI 程序在不同的平台上显示平台对应的风格。wxPython 是一个非常流行的跨平台的 GUI

	库。其官方网址是 http://www.wxpython.org/。
PyQt	PyQt 是 Python 编程语言和 Qt 库的成功融合。Qt 本身是一个扩展的 C++ GUI 应用开发框架，Qt 可以在 UNIX、Windows 和 Mac OS X 上完美运行，因此 PyQt 是建立在 Qt 基础上的 Python 包装。所以 PyQt 也能跨平台使用。
PyGTK	PyGTK 基于老版本的 GTK+2 的库提供绑定，借助于底层 GTK+2 所提供的各种可视化元素和组件，同样可以开发出在 GNOME 桌面系统上运行的软件，因此它主要适用于 Linux/UNIX 系统。PyGTK 对 GTK+2 的 C 语言进行了简单封装，提供了面向对象的编程接口。其官方网址是 http://www.pygtk.org/。
Pywin32	Windows Pywin32 库允许我们像 VC 一样的形式使用 Python 开发 win32 应用。
Kivy	Kivy 是一个开源库，它能够让使用相同源代码创建的程序实现跨平台运行。同时，Kivy 主要关注创新型用户界面开发，例如多点触摸应用程序。
Flexx	Flexx 是一个纯 Python 工具包，可以用来创建图形化界面程序，还支持使用 Web 技术进行界面的渲染。

每个 GUI 库都有其优缺点，因此 GUI 库的选择取决于实现的应用场景。比如，考虑开发跨平台的图形用户界面，则推荐使用 PyQt 或 wsPython。

除了以上这些 Python GUI 库之外，Python 还有一个官方御用的 GUI 库，即 Tkinter 库，我们常用的 IDLE 就是用 Tkinter 开发的。

Tkinter 是 Python 标准 GUI 库，其最初是为 Tcl（这是一门工具命令语言，而不是某个电视机品牌）设计的，由于其良好的可移植性和灵活性，加上非常容易使用，因此逐渐被移植到很多脚本语言中，包括 Perl、Ruby 和 Python。

本章就来重点介绍 Tkinter 库。由于它是 Python 自带的 GUI 库，因此无须进行额外的下载安装，只要导入 tkinter 包即可。

2.Python GUI 编程组件及用法

如果从程序员的角度来看一个窗口，这个窗口不是一个整体（有点庖丁解牛的感觉），而是由多个部分组合而成的，如图 1 所示：

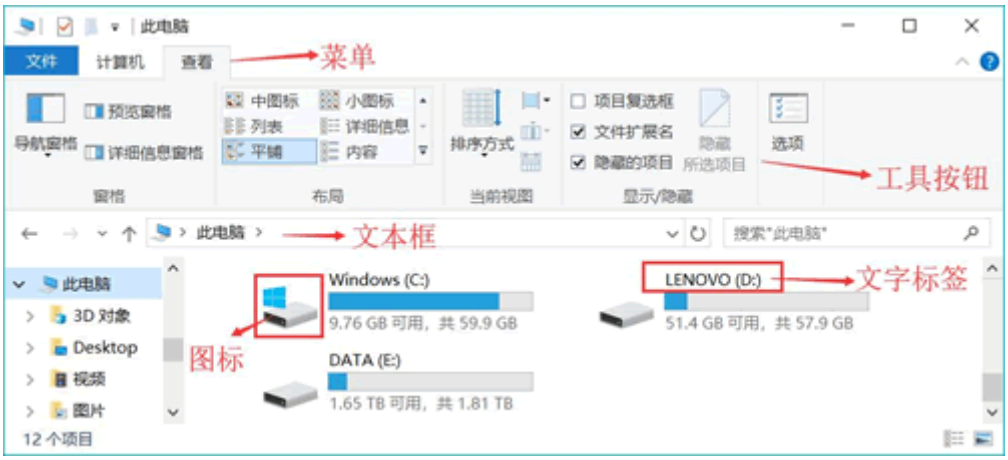


图 1 窗口的“分解”

从图 1 中可以看出，任何窗口都可被分解成一个空的容器，容器里盛装了大量的基本组件，通过设置这些基本组件的大小、位置等属性，就可以将该空的容器和基本组件组成一个整体的窗口。

实际上，图形界面编程非常简单，它非常类似于小朋友玩的拼图游戏，容器类似于拼图的“母板”，而普通组件（如 Button、Listbox 之类的）则类似于拼图的图块。创建图形用户界

面的过程就是完成拼图的过程。

使用 Tkinter 进行 GUI 编程与其他语言的 GUI 编程基本相似，都是使用不同的“积木块”来堆出各种各样的界面。因此，学习 GUI 编程的总体步骤大致可分为三步：

1. 了解 GUI 库大致包含哪些组件，就相当于熟悉每个积木块到底是些什么东西。
2. 掌握容器及容器对组件进行布局的方法，就相当于掌握拼图的“母板”，以及母板怎么固定积木块的方法。
3. 逐个掌握各组件的用法，则相当于深入掌握每个积木块的功能和用法。

下面先完成第 1 步，大致介绍 Tkinter 库包含的各 GUI 组件。由于这些组件之间存在错综复杂的继承关系，因此先通过类图来了解各 GUI 组件，以及它们之间的关系。Tkinter 的 GUI 组件之间的继承关系如图 2 所示。

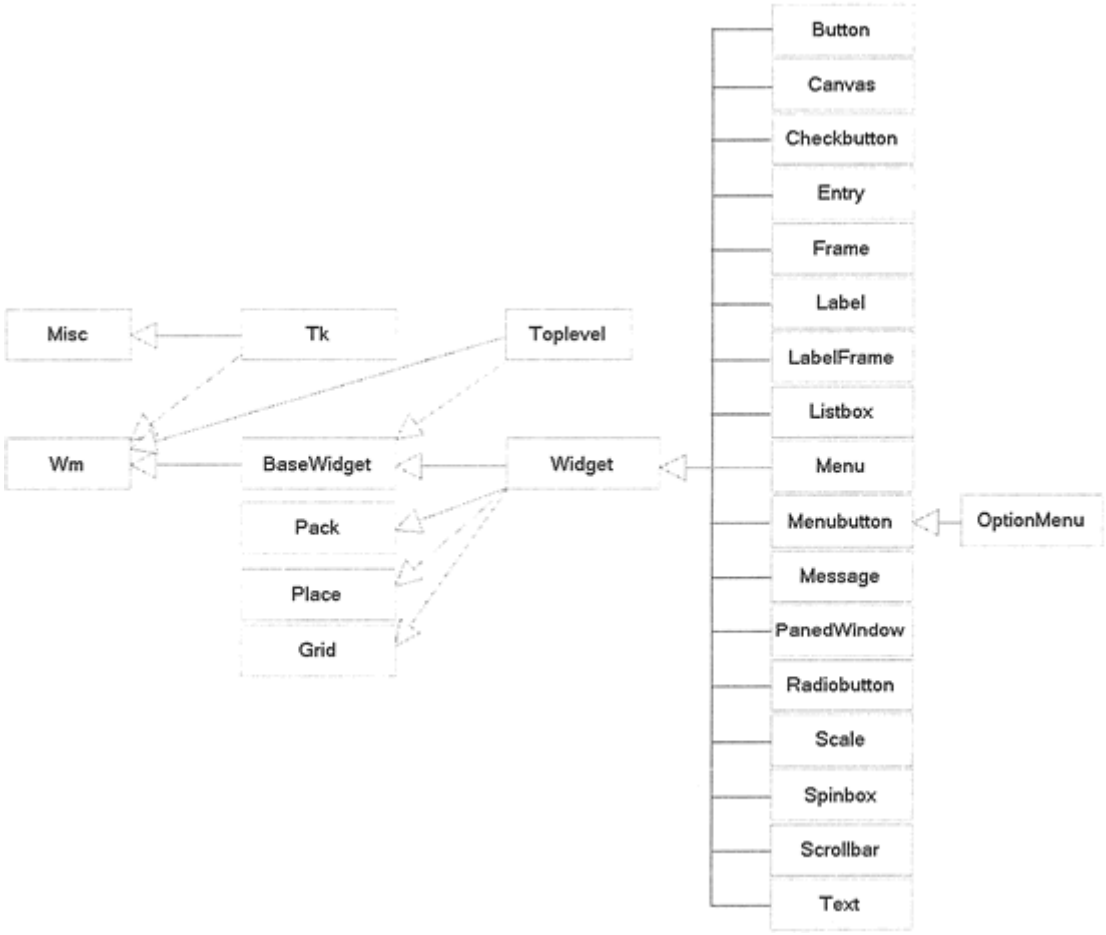


图 2 Tkinter 的 GUI 组件之间的继承关系

可以看到，Tkinter 的 GUI 组件有两个根父类，它们都直接继承了 object 类：

- Misc：它是所有组件的根父类。
- Wm：它主要提供了一些与窗口管理器通信的功能函数。

对于 Misc 和 Wm 两个基类而言，GUI 编程并不需要直接使用它们，但由于它们是所有 GUI 组件的父类，因此 GUI 组件都可以直接使用它们的方法。

Misc 和 Wm 派生了一个子类 Tk，它代表应用程序的主窗口。因此所有 Tkinter GUI 编程通常都需要直接或间接使用该窗口类。

BaseWidget 是所有组件的基类，它还派生了一个子类 Widget。Widget 代表一个通用的 GUI 组件，Tkinter 所有的 GUI 组件都是 Widget 的子类。

再来看 Widget 的父类。Widget 一共有四个父类，除 BaseWidget 之外，还有 Pack、Place 和 Grid，这三个父类都是布局管理器，它们负责管理所包含的组件的大小和位置。

剩下的就是图 2 右边所显示的 Widget 的子类了，它们都是 Tkinter GUI 编程的各种 UI 组件，也就是前面所说的各种“积木块”。表 3 简单介绍了各 GUI 组件的功能。

表 3 各 GUI 组件的功能

Tkinter 类	名称	简介
Toplevel	顶层	容器类，可用于为其他组件提供单独的容器；Toplevel 有点类似于窗口
Button	按钮	代表按钮组件
Canvas	画布	提供绘图功能，包括绘制直线、矩形、椭圆、多边形、位图等
Checkbutton	复选框	可供用户勾选的复选框
Entry	单行输	用户可输入内容

	入框	
Frame	容器	用于装载其它 GUI 组件
Label	标签	用于显示不可编辑的文本或图标
LabelFrame	容器	也是容器组件，类似于 Frame，但它支持添加标题
Listbox	列表框	列出多个选项，供用户选择
Menu	菜单	菜单组件
Menubutton	菜单按钮	用来包含菜单的按钮（包括下拉式、层叠式等）
OptionMenu	菜单按钮	Menubutton 的子类，也代表菜单按钮，可通过按钮打开一个菜单
Message	消息框	类似于标签，但可以显示多行文本；后来当 Label 也能显示多行文本之后，该组件基本处于废弃状态
PanedWindow	分区窗口	该容器会被划分成多个区域，每添加一个组件占一个区域，用户可通过拖动分隔线来改变各区域的大小
Radiobutton	单选钮	可供用户点边的单选钮
Scale	滑动条	拖动滑块可设定起始值和结束值，可显示当前位置的精确值
Spinbox	微调选择器	用户可通过该组件的向上、向下箭头选择不同的值

Scrollbar	滚动条	用于为组件（文本域、画布、列表框、文本框）提供滚动功能
Text	多行文本框	显示多行文本

此处只是简单地介绍了这些 GUI 组件，读者有一个大致印象即可，先不用太熟悉它们。下面通过一个简单的例子来创建一个窗口：

```
1.          # Python 2.x 使用这行
2.          #from Tkinter import *
3.          # Python 3.x 使用这行
4.          from tkinter import *
5.          # 创建 Tk 对象，Tk 代表窗口
6.          root = Tk()
7.          # 设置窗口标题
8.          root.title('窗口标题')
9.          # 创建 Label 对象，第一个参数指定该 Label 放入 root
10.         w = Label(root, text="Hello Tkinter!")
11.         # 调用 pack 进行布局
12.         w.pack()
13.         # 启动主窗口的消息循环
14.         root.mainloop()
```

上面程序主要创建了两个对象 Tk 和 Label。其中 Tk 代表顶级窗口，Label 代表一个简单的文本标签，因此需要指定将该 Label 放在哪个容器内。上面程序在创建 Label 时第一个参数指定了 root，表明该 Label 要放入 root 窗口内。

运行上面程序，可以看到如图 3 所示的运行效果：

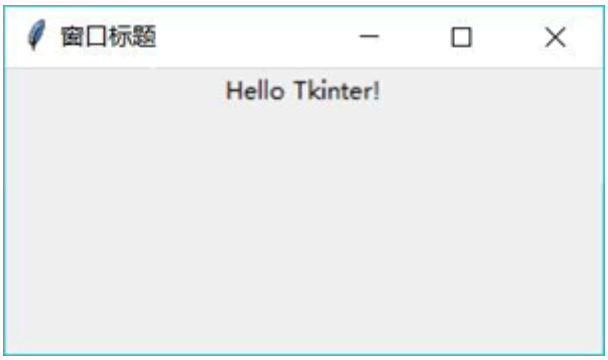


图 3 简单窗口

此外，还有一种方式是不直接使用 Tk，只要创建 Frame 的子类，它的子类就会自动创建 Tk 对象作为窗口。例如如下程序：

```
# Python 2.x 使用这行
#from Tkinter import *
# Python 3.x 使用这行
from tkinter import *

# 定义继承 Frame 的 Application 类
class Application(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        # 调用 initWidgets()方法初始化界面
        self.initWidgets()
    def initWidgets(self):
        # 创建 Label 对象，第一个参数指定该 Label 放入 root
        w = Label(self)
        # 创建一个位图
        bm = PhotoImage(file = 'serial.png')
        # 必须用一个不会被释放的变量引用该图片，否则该图片会被回收
        w.x = bm
        # 设置显示的图片是 bm
```

```
                w['image'] = bm
                w.pack()
                # 创建 Button 对象，第一个参数指定该 Button 放入 root
                okButton = Button(self, text="确定")
                okButton['background'] = 'yellow'
                okButton.configure(background='yellow')
                okButton.pack()
        # 创建 Application 对象
        app = Application()
        # Frame 有个默认的 master 属性，该属性值是 Tk 对象（窗口）
        print(type(app.master))
        # 通过 master 属性来设置窗口标题
        app.master.title('窗口标题')
        # 启动主窗口的消息循环
        app.mainloop()
```

上面程序创建了 Frame 的子类 Application ,并在该类的构造方法中调用了 initWidges() 方法（这个方法可以是任意方法名），程序在 initWidgets() 方法中创建了两个组件，即 Label 和 Button。

上面程序只是创建了 Application 的实例（Frame 容器的子类），并未创建 Tk 对象（窗口），那么这个程序有窗口吗？答案是肯定的。如果程序在创建任意 Widget 组件（甚至 Button）时没有指定 master 属性（即创建 Widget 组件时第一个参数传入 None），那么程序会自动为该 Widget 组件创建一个 Tk 窗口，因此 Python 会自动为 Application 实例创建 Tk 对象来作为它的 master。

该程序与上一个程序的差别在于，程序在创建 Label 和 Button 之后，对 Label 进行了配置，设置了 Label 显示的背景图片；也对 Button 进行了配置，设置了 Button 的背景色。

可能有读者对上面程序中的如下代码感到好奇：

```
w.x = bm
```

这行代码负责为 w 的 x 属性赋值。这行代码有什么作用呢？因为程序在 initWidgets() 方法中创建了 PhotoImage 对象，这是一个图片对象。当该方法结束时，如果该对象没有被其他变量引用，这个图片可能会被系统回收，此处由于 w（Label 对象）需要使用该图片，因此程序就让 w 的 x 属性引用该 PhotoImage 对象，阻止系统回收 PhotoImage 的图片。

运行上面程序，可以看到如图 4 所示的效果：

图 4 配置 Label 和 Button

仔细体会上面程序中 initWidgets() 方法的代码，虽然看上去代码量不小，但实际上只有 3 行代码：

- 创建 GUI 组件。相当于创建“积木块”。
- 添加 GUI 组件，此处使用 pack() 方法添加。相当于把“积木块”添加进去。
- 配置 GUI 组件。

其中创建 GUI 组件的代码很简单，与创建其他 Python 对象并没有任何区别，但通常至少要指定一个参数，用于设置该 GUI 组件属于哪个容器（Tk 组件例外，因为该组件代表顶级窗口）。

配置 GUI 组件的方法有以下 2 种：

1. 以关键字参数的方式配置。例如 Button(self, text="确定") ,其中 text = "确定" 就指定了该按钮上的文本是“确定”。
2. 以字典语法进行配置。例如 okButton['background']='yellow'，这种语法使得 okButton 看上去就像一个字典，它用于配置 okButton 的 background 属性，从而改变该按钮的背景色。

上面两种方式完全可以切换。比如，可以在创建按钮之后配置该按钮上的文本：

```
okButton ['text']= '确定'
```

同时，上面这行代码等同于如下代码：

```
okButton.configure(text = '确定')
```

也可以在创建按钮时就配置它的文本和背景色，例如如下代码：

```
1.          #创建 Button 对象，在创建时就配置它的文本和背景色
2.          okButton = Button (self, text = "确定", background = 'yellow')
```

这里又产生了一个问题，除可配置 background、image 等选项之外，GUI 组件还可配置哪些选项呢？可以通过该组件的构造方法的帮助文档来查看。例如，查看 Button 的构造方法的帮助文档，可以看到如下输出结果：

```
>>> help(tkinter.Button.__init__)
Help on function __init__ in module tkinter:

__init__(self, master=None, cnf={}, **kw)
    Construct a button widget with the parent MASTER.

    STANDARD OPTIONS

        activebackground, activeforeground, anchor,
        background, bitmap, borderwidth, cursor,
        disabledforeground, font, foreground
        highlightbackground, highlightcolor,
        highlightthickness, image, justify,
        padx, pady, relief, repeatdelay,
```

```
    repeatinterval, takefocus, text,
    textvariable, underline, wraplength

 WIDGET-SPECIFIC OPTIONS

    command, compound, default, height,
    overrelief, state, width
```

上面的帮助文档指出，Button 支持两组选项：标准选项（STANDARD OPTIONS）和组件特定选项（WIDGET-SPECIFIC OPTIONS）。至于这些选项的含义，基本上通过它们的名字就可猜出来。

这里简单介绍了一些 GUI 组件的常见选项以及各自的含义（如表 5 所示）。

表 5 GUI 组件支持的通用选项

选项名（别名）	含义	单位	典型值
activebackground	指定组件处于激活状态时的背景色	color	'gray25'或'#ff4400'
activeforeground	指定组件处于激活状态时的前景色	color	'gray25'或'#ff4400'
anchor	指定组件内的信息（比如文本或图片）		CENTER

	在组件中如何显示。必须为下面的值之一：N、NE、E、SE、S、SW、W、NW 或 CENTER。比如 NW（NorthWest）指定将信息显示在组件的左上角		
background(bg)	指定组件正常显示时的背景色	color	'gray25'或 '#ff4400'
bitmap	指定在组件上显示该选项指定的位图，该选项值可以是 Tk_GetBitmap 接收的任何形式的位图。位图的显示方式受 anchor、justify 选项的影响。如果同时指定了 bitmap 和 text 那么 bitmap 覆盖文本；如果同时指定了 bitmap 和 image，那么 image 覆盖 bitmap		
borderwidth	指定组件正常显示时的 3D 边框的宽度，该值可以是 Tk_GetPixels 接收的任何格式	pixel	2
cursor	指定光标在组件上的样式。该值可以是 Tk_GetCursors 接受的任何格式	cursor	gumby
disabledforeground	指定组件处于禁用状态时的前景色	color	'gray25'或 '#ff4400'

font	指定组件上显示的文本字体	font	'Helvetica' 或 ('Verdana', 8)
foreground(fg)	指定组件正常显示时的前景色	color	'gray'或 '#ff4400'
highlightbackground	指定组件在高亮状态下的背景色	color	'gray'或 '#ff4400'
highlightcolor	指定组件在高亮状态下的前景色	color	'gray'或 '#ff4400'
highlightthickness	指定组件在高亮状态下的周围方形区域的宽度，该值可以是 Tk_GetPixels 接收的任何格式	pixel	2
height	指定组件的高度，以 font 选项指定的字体的字符高度为单位，至少为 1	integer	14
image	指定组件中显示的图像，如果设置了 image 选项，它将会覆盖 text、bitmap 选项	image	
justify	指定组件内部内容的对齐方式，该选项支持 LEFT（左对齐）、CENTER（居中	constant	RIGHT

	对齐）或 RIGHT（右对齐）这三个值		
padx	指定组件内部在水平方向上两边的空白，该值可以是 Tk_GctPixels 接收的任何格式	pixel	12
pady	指定组件内部在垂直方向上两地的空白，该值可以是 Tk_GctPixels 接收的任何格式	pixel	12
relief	指定组件的 3D 效果，该选项支持的值包括 RAISED、SUNKEN、FLAT、RIDGE、SOLID、GROOVE。该值指出组件内部相对于外部的外观样式，比如 RAISED 表示组件内部相对于外部凸起	constant	GROOVE RAISED
selectbackground	指定组件在选中状态下的背景色	color	'gray'或 '#ff4400'
selectborderwidth	指定组件在选中状态下的 3D 边框的宽度，该值可以是 Tk_GetPixels 接收的任何格式	pixel	2
selectforeground	指定组在选中状态下的前景色	color	'gray'或 '#ff4400'
state	指定组件的当前状态。该选项支持	constant	NORMAL

	NORMAL（正常）、DISABLE（禁用）这两个值		
takefocus	指定组件在键盘遍历（Tab 或 Shift+Tab）时是否接收焦点，将该选项设为 1 表示接收焦点；设为 0 表示不接收焦点	boolean	1 或 YES
text	指定组件上显示的文本，文本显示格式由组件本身、anchor 及 justify 选项决定	str	'确定'
textvariable	指定一个变量名，GUI 组件负责显示该变量值转换得到的字符串，文本显示格式由组件本身、anchor 及 justify 选项决定	variable	bnText
underline	指定为组件文本的第几个字符添加下画线，该选项就相当于为组件绑定了快捷键	integer	2
width	指定组件的宽度，以 font 选项指定的字体的字符高度为单位，至少为 1	integer	14
wraplength	对于能支持字符换行的组件，该选项指定每行显示的最大字符数，超过该数量	integer	20

	的字符将会转到下行显示		
xscrollcommand	通常用于将组件的水平滚动改变（包括内容滚动或宽度发生改变）与水平滚动条的 set 方法关联 ,从而让组件的水平滚动改变传递到水平滚动条	function	scroll.set
yscrollcommand	通常用于将组件的垂直滚动改变（包括内容滚动或高度发生改变）与垂直滚动条的 set 方法关联，从而让组件的垂直滚动改变传递到垂直滚动条	function	scroll.set

3.Python Tkinter Pack

GUI 编程就相当于小孩子搭积木，每个积木块应该放在哪里，每个积木块显示为多大，也就是对大小和位置都需要进行管理，而布局管理器正是负责管理各组件的大小和位置的。此外，当用户调整了窗口的大小之后，布局管理器还会自动调整窗口中各组件的大小和位置。

Tkinter 中，有 3 种布局管理器，分别为 pack、grid 和 place。本节先介绍 pack 布局管理器。使用 Pack 布局各个组件，当程序向容器中添加组件时，这些组件会依次向后排列，排列方向既可是水平的，也可是垂直的。

下面程序简单示范了 Pack 布局的用法，该程序向窗口中添加了三个 Label 组件：

```
1.          # Python 2.x 使用这行
2.          #from Tkinter import *
3.          # Python 3.x 使用这行
4.          from tkinter import *
5.
6.          # 创建窗口并设置窗口标题
7.          root = Tk()
8.          # 设置窗口标题
```

```
9.          root.title('Pack 布局')
10.         for i in range(3):
11.             lab = Label(root, text="第%d 个 Label" % (i + 1), bg='#eeeeee')
12.             # 调用 pack 进行布局
13.             lab.pack()
14.         # 启动主窗口的消息循环
15.         root.mainloop()
```

上面程序创建了一个窗口，然后使用循环创建了三个 Label，并对这三个 Label 使用了 pack() 方法进行默认的 Pack 布局。运行该程序，可以看到如图 1 所示的界面：

图 1 使用 Pack 布局

图 1 使用的是默认的 Pack 布局，实际上程序在调用 pack() 方法时，可传入多个选项，表 2 罗列出了一些常用的选项以及各自的功能。

表 2 Tkinter pack 常用选项及功能

常用选项	功能
anchor	当可用空间大于组件所需求的大小时，该选项决定组件被放置在容器的何处。该选项支持 N（北，代表上）、E（东，代表右）、S（南，代表下）、W（西，代

	表左）、NW（西北，代表左上）、NE（东北，代表右上）、SW（西南，代表左下）、SE（东南，代表右下）、CENTER（中，默认值）这些值。
expand	该 bool 值指定当父容器增大时才是否拉伸组件。
fill	设置组件是否沿水平或垂直方向填充。该选项支持 NONE、X、Y、BOTH 四个值，其中 NONE 表示不填充，BOTH 表示沿着两个方向填充。
ipadx	指定组件在 x 方向（水平）上的内部留白（padding）。
ipady	指定组件在 y 方向（水平）上的内部留白（padding）。
padx	指定组件在 x 方向（水平）上与其他组件的间距。
pady	指定组件在 y 方向（水平）上与其他组件的间距。
side	设置组件的添加位置，可以设置为 TOP、BOTTOM、LEFT 或 RIGHT 这四个值的其中之一。

pack 布局管理器支持的选项，可以通过 help(tkinter.Label.pack) 命令来查看，有兴趣的读者可自行查看。

当程序界面比较复杂时，就需要使用多个容器（Frame）分开布局，然后再将 Frame 添加到窗口中。例如如下程序：

```
1.          # Python 2.x 使用这行
2.          #from Tkinter import *
3.          # Python 3.x 使用这行
4.          from tkinter import *
5.          class App:
6.              def __init__(self, master):
7.                  self.master = master
8.                  self.initWidgets()
```

```
9.              def initWidgets(self):
10.                 # 创建第一个容器
11.                 fm1 = Frame(self.master)
12.                 # 该容器放在左边排列
13.                 fm1.pack(side=LEFT, fill=BOTH, expand=YES)
14.                 # 向 fm1 中添加 3 个按钮
15.                 # 设置按钮从顶部开始排列，且按钮只能在垂直（X）方向填充
16.                 Button(fm1, text='第一个').pack(side=TOP, fill=X, expand=YES)
17.                 Button(fm1, text='第二个').pack(side=TOP, fill=X, expand=YES)
18.                 Button(fm1, text='第三个').pack(side=TOP,  fill=X, expand=YES)
19.                 # 创建第二个容器
20.                 fm2 = Frame(self.master)
21.                 # 该容器放在左边排列，就会挨着 fm1
22.                 fm2.pack(side=LEFT, padx=10, expand=YES)
23.                 # 向 fm2 中添加 3 个按钮
24.                 # 设置按钮从右边开始排列
25.                 Button(fm2, text='第一个').pack(side=RIGHT, fill=Y, expand=YES)
26.                 Button(fm2, text='第二个').pack(side=RIGHT, fill=Y, expand=YES)
27.                 Button(fm2, text='第三个').pack(side=RIGHT, fill=Y, expand=YES)
28.                 # 创建第三个容器
29.                 fm3 = Frame(self.master)
30.                 # 该容器放在右边排列，就会挨着 fm1
31.                 fm3.pack(side=RIGHT, padx=10, fill=BOTH, expand=YES)
32.                 # 向 fm3 中添加 3 个按钮
33.                 # 设置按钮从底部开始排列，且按钮只能在垂直（Y）方向填充
34.                 Button(fm3, text='第一个').pack(side=BOTTOM, fill=Y,
    expand=YES)
35.                 Button(fm3, text='第二个').pack(side=BOTTOM, fill=Y,
    expand=YES)
36.                 Button(fm3, text='第三个').pack(side=BOTTOM, fill=Y,
    expand=YES)
37.         root = Tk()
38.         root.title("Pack 布局")
39.         display = App(root)
40.         root.mainloop()
```

上面程序创建了三个 Frame 容器，其中第一个 Frame 容器内包含三个从顶部（TOP）开始排列的按钮，这意味着这三个按钮会从上到下依次排列，且这三个按钮能在水平（X）方向上填充；第二个 Frame 容器内包含三个从右边（RIGHT）开始排列的按钮，这意味着这三个按钮会从右向左依次排列；第三个 Frame 容器内包含三个从底部（BOTTOM）开始排列的按钮，这意味着这三个按钮会从下到上依次排列，且这三个按钮能在垂直（Y）方向上填充。

运行上面程序，将看到如图 3 所示的界面。

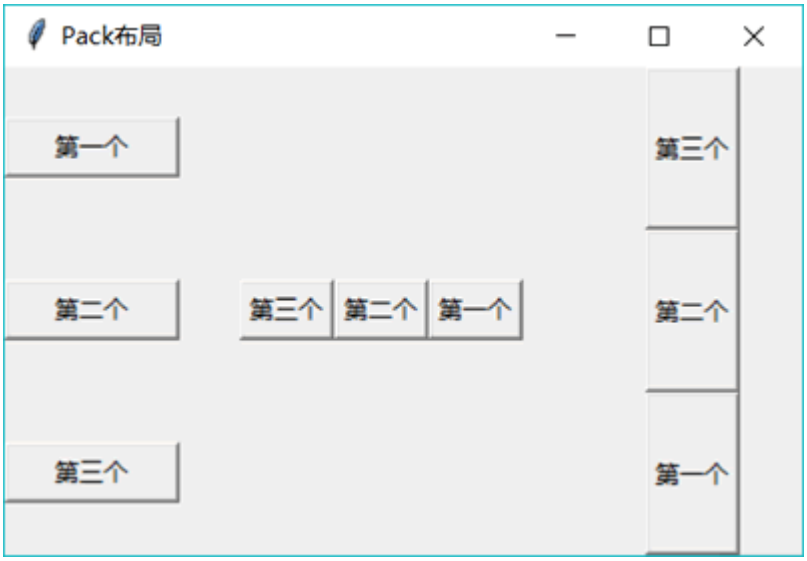


图 3 复杂的 Pack 布局

从图 3 中可以看到，为运行效果添加了三个框，分别代表 fm1、fm2、fm3（实际上容器是看不到的），此时可以看到 fm1 内的三个按钮从上到下排列，并且可以在水平方向上填充；fm3 内的三个按钮从下到上排列，并且可以在垂直方向上填充。

可能有读者会有疑问，fm2 内的三个按钮也都设置了 fill=Y，expand=YES，这说明它们也能在垂直方向上填充，为啥舍看不到呢？仔细看 fm2.pack(side=LEFT, padx=10, expand= YES)这行代码，它说明 fm2 本身不在任何方向上填充，因此 fm2 内的三个按钮都不能填充。

如果希望看到 fm2 内的三个按钮也能在垂直方向上填充，则可将 fm2 的 pack() 方法改为如下代码：

```
fm2.pack(side=LEFT, padx=10, fill=BOTH, expand=YES)
```

再次运行程序，得到的界面如图 4 所示。

图 4 修改后的界面

通过上面介绍不难发现，Pack 布局其实还是非常灵活的，它完全可以实现很复杂的用户界面。这里有一个界面分解的常识需要说明，无论看上去多么复杂、古怪的界面，其实大多可分解为水平排列和垂直排列，而 Pack 布局既可实现水平排列，也可实现垂直排列，然后再通过多个容器进行组合，这样就可以开发出更复杂的界面了。

对于打算使用 Pack 布局的开发者来说，首先要做的事情是将程序界面进行分解，分解成水平排列的容器和垂直排列的容器（有时候甚至要容器嵌套容器），然后使用多个 Pack 布局的容器将它们组合在一起。

4.Python Tkinter Grid

很多时候 Tkinter 界面编程都会优先考虑使用 Pack 布局，但实际上 Tkinter 后来引入的 Grid 布局管理器不仅简单易用，而且管理组件也非常方便。

Grid 把组件空间分解成一个网格进行维护，即按照行、列的方式排列组件，组件位置由其所在的行号和列号决定，行号相同而列号不同的几个组件会被依次上下排列，列号相同而行号不同的几个组件会被依次左右排列。

> 可见，在很多场景下 Grid 是最好用的布局方式。相比之下，Pack 布局在控制细节方面反而显得有些力不从心。

使用 Grid 布局的过程就是为各个组件指定行号和列号的过程，不需要为每个网格都指定大

小，Grid 布局会自动为它们设置合适的大小。

程序调用组件的 grid() 方法就进行 Grid 布局，在调用 grid() 方法时可传入多个选项，该方法支持的 ipadx、ipady、padx、pady 与 pack() 方法的这些选项相同，除此之外，grid() 方法还额外增加了如表 1 所示的这些选项。

表 1 Tkinter Grid 常用选项

Grid 选项	功能
column	指定将组件放入哪列，第一列的索引为 0。
columnspan	指定组件横跨多少列。
row	指定组件放入哪行，第一行的索引为 0。
rowspan	指定组件横跨多少行。
sticky	类似 pack() 方法的 anchor 选项，同样支持 N（北，代表上）、E（东，代表右）、S（南，代表下）、W（西，代表左）、NW（西北，代表左上）、NE（东北，代表右上）、SW（西南，代表左下）、SE（东南，代表右下）、CENTER（中，默认值）这些值。

Grid 支持的选项，可以通过执行 help(tkinter.Label.grid) 命令查看（执行此命令之前，一定要先引入 Tkinter 包）。

下面程序使用 Grid 布局来实现一个计算器界面：

```
1.          # Python 2.x 使用这行
2.          #from Tkinter import *
3.          # Python 3.x 使用这行
4.          from tkinter import *
```

```
5.
6.          class App:
7.              def __init__(self, master):
8.                  self.master = master
9.                  self.initWidgets()
10.             def initWidgets(self):
11.                 # 创建一个输入组件
12.                 e = Entry(relief=SUNKEN, font=('Courier New', 24), width=25)
13.                 # 对该输入组件使用 Pack 布局，放在容器顶部
14.                 e.pack(side=TOP, pady=10)
15.                 p = Frame(self.master)
16.                 p.pack(side=TOP)
17.                 # 定义字符串的元组
18.                 names = ("0" , "1" , "2" , "3"
19.                     , "4" , "5" , "6" , "7" , "8" , "9"
20.                     , "+" , "-" , "*" , "/" , ".", "=")
21.                 # 遍历字符串元组
22.                 for i in range(len(names)):
23.                     # 创建 Button，将 Button 放入 p 组件中
24.                     b = Button(p, text=names[i], font=('Verdana', 20), width=6)
25.                     b.grid(row=i // 4, column=i % 4)
26.         root = Tk()
27.         root.title("Grid 布局")
28.         App(root)
29.         root.mainloop()
```

上面程序实际上使用了两个布局管理器进行嵌套，先使用 Pack 布局管理两个组件，分别是 Entry（输入组件）和 Frame（容器），这两个组件就会按照从上到下的方式排列。

接下来程序使用 Grid 布局管理 Frame 容器中的 16 个按钮，分别将 16 个按钮放入不同的行、不同的列。运行上面程序，可以看到如图 1 所示的界面。

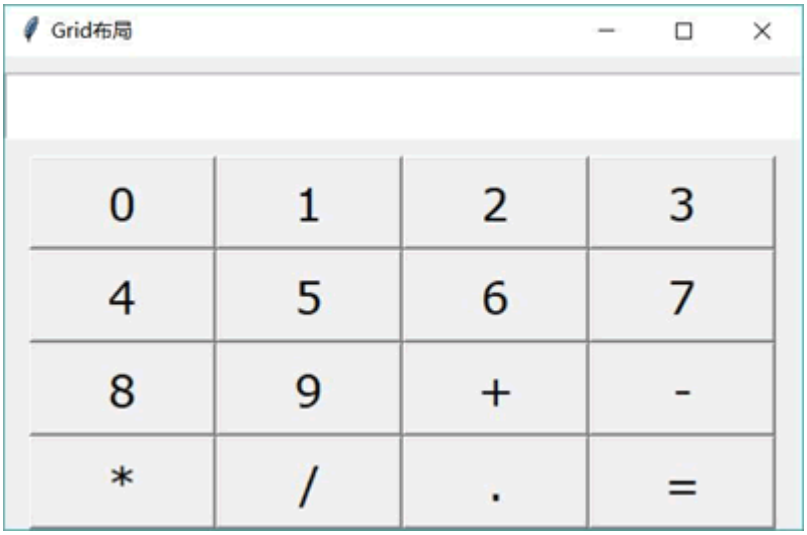


图 1 使用 Grid 布局实现计算器界面

5.Python Tkinter Place 布局管理器及用法

Place 布局就是其他 GUI 编程中的“绝对布局”，这种布局方式要求程序显式指定每个组件的绝对位置或相对于其他组件的位置。

如果要使用 Place 布局，调用相应组件的 place() 方法即可。在使用该方法时同样支持一些详细的选项，关于这些选项的介绍如表 1 所示。

表 1 Tkinter place()方法常用选项

place 选项	功能
x	指定组件的 X 坐标。x 为 0 代表位于最左边。
y	指定组件的 Y 坐标。y 为 0 代表位于最右边。
relx	指定组件的 X 坐标 ,以父容器总宽度为单位 1 ,该值应该在 0.0~1.0 之间 ,其中 0.0 代表位于窗口最左边 ,1.0 代表位于窗口最右边 ,0.5 代表位于窗口中间。

rely	指定组件的 Y 坐标，以父容器总高度为单位 1，该值应该在 0.0~1.0 之间，其中 0.0 代表位于窗口最上边，1.0 代表位于窗口最下边，0.5 代表位于窗口中间。
width	指定组件的宽度，以 pixel 为单位。
height	指定组件的高度，以 pixel 为单位。
relwidth	指定组件的宽度，以父容器总宽度为单位 1，该值应该在 0.0~1.0 之间，其中 1.0 代表整个窗口宽度，0.5 代表窗口的一半宽度。
relheight	指定组件的高度，以父容器总高度为单位 1，该值应该在 0.0~1.0 之间，其中 1.0 代表整个窗口高度，0.5 代表窗口的一半高度。
bordermode	该属性支持“inside”或“outside” 属性值，用于指定当设置组件的宽度、高度时是否计算该组件的边框宽度。

place() 方法支持的所有选项，可通过执行 help(tkinter.Label.place) 命令查看（一定要先导入 Tkinter 包）。

当使用 Place 布局管理容器中的组件时，需要设置组件的 x、y 或 relx、rely 选项，Tkinter 容器内的坐标系统的原点 (0，0) 在左上角，其中 X 轴向右延伸，Y 轴向下延伸，如图 1 所示：

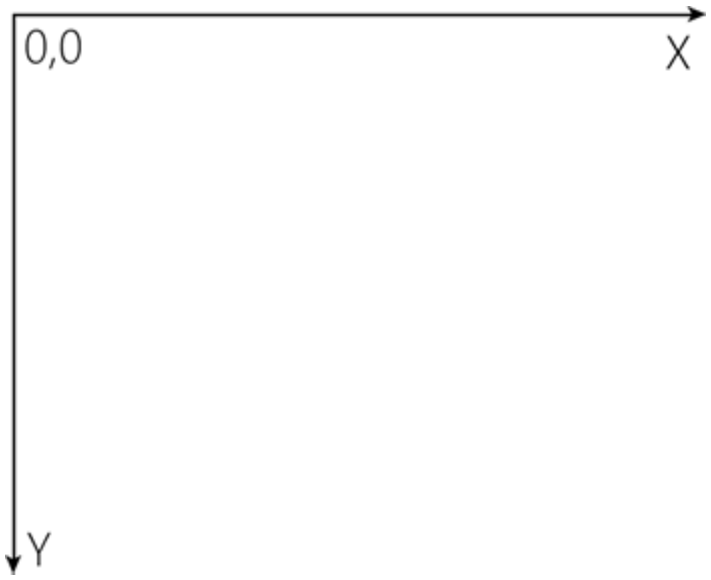


图 1 Tkinter 容器坐标系

如果通过 x、y 指定坐标，单位就是 pixel（像素）；如果通过 relx、rely 指定坐标，则以整个父容器的宽度、高度为 1。不管通过哪种方式指定坐标，通过图 1 不难发现，通过 x 指定的坐标值越大，该组件就越靠右；通过 y 指定的坐标值越大，该组件就越靠下。

下面介绍一个使用 Place 进行布局的例子，该示例将会动态计算各 Label 的大小和位置，并通过 place() 方法设置各 Label 的大小和位置。

```
1.          # Python 2.x 使用这行
2.          #from Tkinter import *
3.          # Python 3.x 使用这行
4.          from tkinter import *
5.          import random
6.          class App:
7.              def __init__(self, master):
8.                  self.master = master
9.                  self.initWidgets()
10.             def initWidgets(self):
11.                 # 定义字符串元组
12.                 books = ('疯狂 Python 讲义', '疯狂 Swift 讲义', '疯狂 Kotlin 讲义
    ',\
13.                     '疯狂 Java 讲义', '疯狂 Ruby 讲义')
14.                 for i in range(len(books)):
```

```
15.                     # 生成 3 个随机数
16.                     ct = [random.randrange(256) for x in range(3)]
17.                     grayness = int(round(0.299*ct[0] + 0.587*ct[1] +
    0.114*ct[2]))
18.                     # 将元组中 3 个随机数格式化成 16 进制数,转成颜色格式
19.                     bg_color = "#%02x%02x%02x" % tuple(ct)
20.                     # 创建 Label，设置背景色和前景色
21.                     lb = Label(root,
22.                         text=books[i],
23.                         fg = 'White' if grayness < 120 else 'Black',
24.                         bg = bg_color)
25.                     # 使用 place()设置该 Label 的大小和位置
26.                     lb.place(x = 20, y = 36 + i*36, width=180, height=30)
27.         root = Tk()
28.         root.title("Place 布局")
29.         # 设置窗口的大小和位置
30.         # width x height + x_offset + y_offset
31.         root.geometry("250x250+30+30")
32.         App(root)
33.         root.mainloop()
```

上面程序中第 26 行代码就是调用 place() 方法执行 Place 布局的关键代码。在调用 place() 方法时主要设置了 x（X 坐标）、y（Y 坐标）、width（宽度）、height（高度）这四个选项，通过这四个边项即可控制各 Label 的位置和大小。

为了增加一些趣味性，上面程序使用随机数计算了 Label 组件的背景色，并根据背景色的灰度值来计算 Label 组件的前景色：如果 grayness 小于 125，则说明背景色较深，前景色使用白色；否则说明背景色较浅，前景色使用黑色。运行上面程序，可以看到如图 2 所示的界面。

图 2 使用 Place 布局

6.Python Tkinter command 和 bind 事件处理（绑定）用法详解

前面介绍了如何放置各种组件，从而得到了丰富多彩的图形界面，但这些界面还不能响应用户的任何操作。比如单击窗口上的按钮，该按钮并不会提供任何响应。这就是因为程序没有为这些组件绑定任何事件处理的缘故。

command 绑定事件处理方法

简单的事件处理可通过 command 选项来绑定，该选项绑定为一个函数或方法，当用户单击指定按钮时，通过该 command 选项绑定的函数或方法就会被触发。

下面程序示范了为按钮的 command 绑定事件处理方法：

```
# Python 2.x 使用这行
#from Tkinter import *
# Python 3.x 使用这行
from tkinter import *
import random

class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        self.label = Label(self.master, width=30)
        self.label['font'] = ('Courier', 20)
        self.label['bg'] = 'white'
        self.label.pack()
        bn = Button(self.master, text='单击我', command=self.change)
        bn.pack()
    # 定义事件处理方法
    def change(self):
        self.label['text'] = '欢迎学习 Python'
        # 生成 3 个随机数
        ct = [random.randrange(256) for x in range(3)]
        grayness = int(round(0.299*ct[0] + 0.587*ct[1] + 0.114*ct[2]))
        # 将元组中 3 个随机数格式化成 16 进制数,转成颜色格式
        bg_color = "#%02x%02x%02x" % tuple(ct)
        self.label['bg'] = bg_color
        self.label['fg'] = 'black' if grayness > 125 else 'white'
root = Tk()
root.title("简单事件处理")
App(root)
root.mainloop()
```

上面程序中第 16 行代码为 Button 的 command 选项指定为 self.change ,这意味着当该按钮被单击时，将会触发当前对象的 change() 方法。该 change() 方法会改变界面上 Label 的文本和背景色。

运行该程序，单击界面上的“单击我”按钮，将看到如图 1 所示的界面。

图 1 使用 command 绑定事件处理

bind 绑定事件处理方法

上面这种简单的事件绑定方式虽然简单，但它存在较大的局限性：
程序无法为具体事件（比如鼠标移动、按键事件）绑定事件处理方法。
程序无法获取事件相关信息。

为了弥补这种不足，Python 提供了更灵活的事件绑定方式，所有 Widget 组件都提供了一个 bind() 方法，该方法可以为“任意” 事件绑定事件处理方法。

下面先看一个为按钮的单击、双击事件绑定事件处理方法的示例：

```
1.          # 将 tkinter 写成 Tkinter 可兼容 Python 2.x
2.          from tkinter import *
3.          class App:
4.              def __init__(self, master):
5.                  self.master = master
6.                  self.initWidgets()
7.              def initWidgets(self):
8.                  self.show = Label(self.master, width=30, bg='white',
font=('times', 20))
9.                  self.show.pack()
```

```
10.                 bn = Button(self.master, text='单击我或双击我')
11.                 bn.pack(fill=BOTH, expand=YES)
12.                 # 为左键单击事件绑定处理方法
13.                 bn.bind('<Button-1>', self.one)
14.                 # 为左键双击事件绑定处理方法
15.                 bn.bind('<Double-1>', self.double)
16.             def one(self, event):
17.                 self.show['text'] = "左键单击:%s" % event.widget['text']
18.             def double(self, event):
19.                 print("左键双击击, 退出程序:", event.widget['text'])
20.                 import sys; sys.exit()
21.         root = Tk()
22.         root.title('简单绑定')
23.         App(root)
24.         root.mainloop()
```

上面程序中第 13、15 行代码为 Button 按钮的单击、双击事件绑定了事件处理方法，其中第 13 行代码为 '<Btutton-1>' 事件绑定了 self.one 作为事件处理方法；第 15 行代码为 '<Double-1>' 事件绑定了 self.double 作为事件处理方法。

此时 self.one 和 self.double 方法都可定义一个 event 参数，该参数代表了传给该事件处理方法的事件对象，因此上面程序示范了通过事件来获取事件源的方式（通过 event.widget 获取即可）。对于鼠标事件来说，鼠标相对当前组件的位置可通过 event 对象中的 x 和 y 属性来获取。

运行上面程序，单击界面上的按钮，将看到如图 2 所示的运行结果：

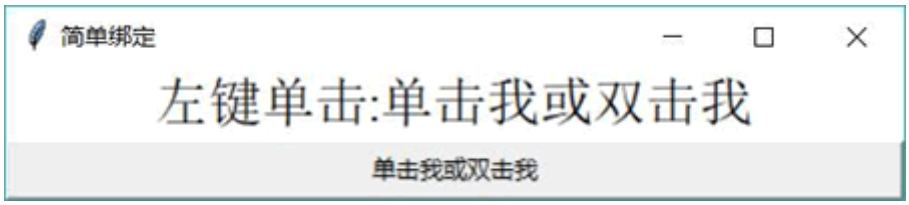


图 2 为单击、双击事件绑定事件处理方法

从上面的例子可以看到，Tkinter 直接使用字符串来代表事件类型，比如使用 <Button-1> 代表鼠标左键单击事件，使用 <Double-1> 代表鼠标左键双击事件。那问题来了，其他事件应该怎么写呢？

代表 Tkinter 事件的字符串大致遵循如下格式：

```
<modifier-type-detail>
```

其中 type 是事件字符串的关键部分，用于描述事件的种类，比如鼠标事件、键盘事件等；modifer 则代表事件的修饰部分，比如单击、双击等；detail 用于指定事件的详情，比如指定鼠标左键、右键、滚轮等。

Tkinter 支持的各种鼠标、键盘事件如表 3 所示。

表 3 Tkinter 支持的各种鼠标、键盘事件

事件	简介
<Button-detail>	鼠标按键的单击事件，detail 指定哪一个鼠标键被单击。比如单击鼠标左键为 <Button-1>，单击鼠标中键为 <Button-2>，单击鼠标右键为 <Button-3>，单击向上滚动的滚轮为 <Button-4>，单击向下滚动的滚轮为 <Button-5>
<modifier Motion>	鼠标在组件上的移动事件，modifier 指定要求按住哪个鼠标键。比如按住鼠标左键移动为 <B1-Motion>，锁住鼠杯中键移动为 <B2-Motion>，按住鼠标右键移动为 <B3-Motion>
<ButtonRelease-detail>	鼠标按键的释放事件，detail 指定哪一个鼠标键被释放。比如鼠标左键被释放为 <ButtonRelease-1>，鼠标中键被释放为 <ButtonRelease-2>，鼠标右键被释放为 <ButtonRelease-3>

<Double-Button-detail>或<Double-detail>	用户双击某个鼠标键的事件，detail 指定哪一个鼠标键被双击。比如双击鼠标左键为 <Double-1>，双击鼠标中键为<Double-2>，双击鼠标右键为 <Double-3>，双击向上滚动的滚轮为 <Double-4>，双击向下滚动的滚轮为<Double-5>
<Enter>	鼠标进入组件的事件。注意，<Enter> 事件不是按下回车键事件，按下回车键的事件是 <Return>
<Leave>	鼠标移出组件事件
<Focusln>	组件及其包含的子组件获得焦点
<FocusOut>	组件及其包含的子组件失去焦点
<Return>	按下回车键的事件。实际上可以为所有按键绑定事件处理方法。特殊键位名称包括 Cancel、BackSpace、Tab、Return（回车）、Shift_L（左 Shift，如果只写 Shift 则代表任意 Shift）、Control_L（左 Ctrl，如果只写 Control 则代表任意 Ctrl）、Alt_L（左 Alt，如果只写 Alt 则代表任意 Alt）、Pause、Caps_Lock、Escape、Prior(Page Up)、Next(Page Down)、End、Home、Left、Up、Right、Down、Print、Insert、Delete、F1、F2、F3、F4、F5、F6、F7、F8、F9、F10、F11、F12、Num_Lock 和 Scroll_Lock
<Key>	键盘上任意键的单击事件，程序可通过 event 获取用户单

	击了哪个键
a	键盘上指定键被单击的事件。比如‘a’代表 a 键被单击，‘b’代表 b 键被单击（不要尖括号）……
<Shift-Up>	在 Shift 键被按下时按 Up 键。类似的还有 <Shift-Left>、<Shift-Down>、<Alt-Up>、<Control-Up> 等
<Configure>	组件大小、位置改变的事件。组件改变之后的大小、位置可通过 event 的 width、height、x、y 获取

下面通过一个示例来示范为鼠标移动事件绑定事件处理方法：

```
# 将 tkinter 写成 Tkinter 可兼容 Python 2.x
from tkinter import *
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        lb = Label(self.master, width=40, height=3)
        lb.config(bg='lightgreen', font=('Times', 20))
        # 为鼠标移动事件绑定事件处理方法
        lb.bind('<Motion>', self.motion)
        # 为按住左键时的鼠标移动事件绑定事件处理方法
        lb.bind('<B1-Motion>', self.press_motion)
        lb.pack()
        self.show = Label(self.master, width=38, height=1)
        self.show.config(bg='white', font=('Courier New', 20))
        self.show.pack()
    def motion(self, event):
        self.show['text'] = "鼠标移动到: (%s %s)" % (event.x, event.y)
```

```
20.                 return
21.             def press_motion(self, event):
22.                 self.show['text'] = "按住鼠标的位置为: (%s %s)" % (event.x,
    event.y)
23.                 return
24.         root = Tk()
25.         root.title('鼠标事件')
26.         App(root)
27.         root.mainloop()
```

上面程序中第 11 行代码为 <Motion>（鼠标移动）事件绑定了事件处理方法，因此鼠标在 lb 组件上移动时将会不断触发 motion() 方法；第 13 行代码为 <B1-Motion>（按住左键时鼠标移动）事件绑定了事件处理方法，因此按住鼠标左键在 lb 组件上移动时将会不断触发 press_motion() 方法。

运行该程序，如果让鼠标直接在第一个 Label 组件（lb）上移动，将看到如图 4 所示的运行结果。

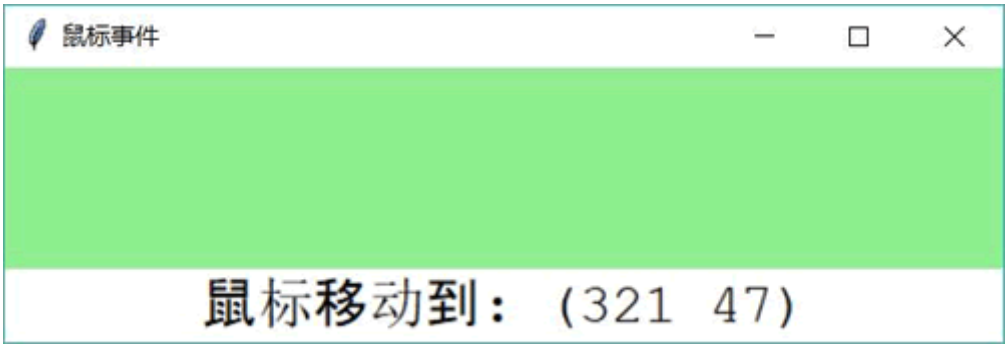


图 4 鼠标移动事件

如果按住鼠标左键并让鼠标在第一个 Label 组件（lb）上移动，将看到如图 5 所示的运行结果。

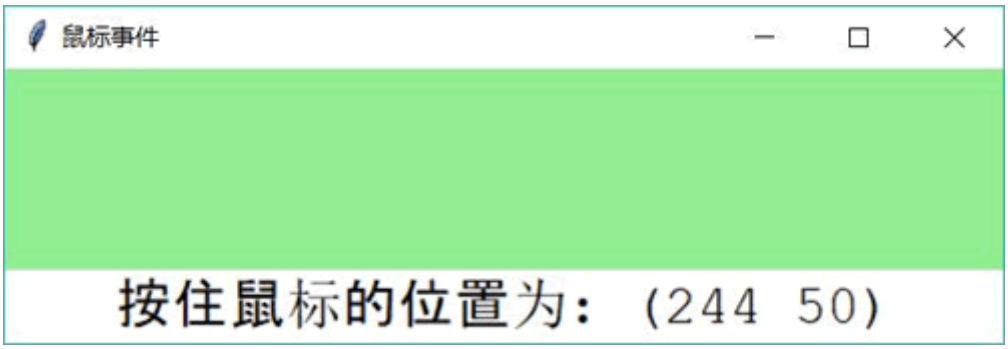


图 5 按住左键时的鼠标移动事件

读者可能对前面章节中那个徒有其表的计算器感到很失望，下面程序将会为该计算器的按钮绑定事件处理方法，从而使它变成真正可运行的计算器：

```
1.          # 将 tkinter 改为 Tkinter 兼容 Python 2.x
2.          from tkinter import *
3.
4.          class App:
5.              def __init__(self, master):
6.                  self.master = master
7.                  self.initWidgets()
8.                  self.expr = None
9.              def initWidgets(self):
10.                 # 创建一个输入组件
11.                 self.show = Label(relief=SUNKEN, font=('Courier New', 24),\
12.                     width=25, bg='white', anchor=E)
13.                 # 对该输入组件使用 Pack 布局，放在容器顶部
14.                 self.show.pack(side=TOP, pady=10)
15.                 p = Frame(self.master)
16.                 p.pack(side=TOP)
17.                 # 定义字符串的元组
18.                 names = ("0" , "1" , "2" , "3"
19.                     , "4" , "5" , "6" , "7" , "8" , "9"
20.                     , "+" , "-" , "*" , "/" , ".", "=")
21.                 # 遍历字符串元组
22.                 for i in range(len(names)):
23.                     # 创建 Button，将 Button 放入 p 组件中
24.                     b = Button(p, text=names[i], font=('Verdana', 20), width=6)
```

```
                    b.grid(row=i // 4, column=i % 4)
                    # 为鼠标左键的单击事件绑定事件处理方法
                    b.bind('<Button-1>', self.click)
                    # 为鼠标左键的双击事件绑定事件处理方法
                    if b['text'] == '=': b.bind('<Double-1>', self.clean)
            def click(self, event):
                # 如果用户单击的是数字键或点号
                if(event.widget['text'] in ('0', '1', '2', '3',\
                    '4', '5', '6', '7', '8', '9', '.')):
                    self.show['text'] = self.show['text'] + event.widget['text']
                # 如果用户单击了运算符
                elif(event.widget['text'] in ('+', '-', '*', '/')):
                    # 如果当前表达式为 None，直接用 show 组件的内容和运算符进行连接
                    if self.expr is None:
                        self.expr = self.show['text'] + event.widget['text']
                    # 如果当前表达式不为 None，用表达式、show 组件的内容和运算符进行连接
                    else:
                        self.expr = self.expr + self.show['text'] + event.widget['text']
                    self.show['text'] = ''
                elif(event.widget['text'] == '=' and self.expr is not None):
                    self.expr = self.expr + self.show['text']
                    print(self.expr)
                    # 使用 eval 函数计算表达式的值
                    self.show['text'] = str(eval(self.expr))
                    self.expr = None
            # 双击=按钮时，程序清空计算结果、将表达式设为 None
            def clean(self, event):
                self.expr = None
                self.show['text'] = ''
        root = Tk()
        root.title("计算器")
        App(root)
        root.mainloop()
```

上面程序中第 27 行代码为界面上所有按钮的单击事件绑定了处理方法，以便处理程序的计算功能；第 29 行代码则为“=”按钮的双击事件绑定了处理方法，当用户双击该按钮时，程序会清空计算结果，重新开始计算。

7.Python Tkinter ttk 组件及用法

前面程序都是直接使用 tkinter 模块下的 GUI 组件的，这些组件看上去特别“复古”，也就是丑，仿佛是从 20 年前的程序上抠出来的组件。

为了弥补这点不足，Tkinter 后来引入了一个 ttk 组件作为补充（主要就是简单包装、美化一下），并使用功能更强大的 Combobox 取代了原来的 Listbox，且新增了 LabeledScale（带标签的 Scale）、Notebook（多文档窗口）、Progressbar（进度条）、Treeview（树）等组件。

ttk 作为一个模块被放在 tkinter 包下，使用 ttk 组件与使用普通的 Tkinter 组件并没有多大的区别，只要导入 ttk 模块即可。下面程序示范了如何使用 ttk 组件：

```
1.          from tkinter import *
2.          # 导入 ttk
3.          from tkinter import ttk
4.
5.          class App:
6.                  def __init__(self, master):
7.                          self.master = master
8.                          self.initWidgets()
9.                  def initWidgets(self):
10.                         #listbox 组件部分
11.                         cb = Listbox(self.master, font=24)
12.                         # 为 Listbox 设置列表项
13.                         for s in ('Python', 'Swift', 'Kotlin'):
```

```
                cb.insert(END, s)
            cb.pack(side=LEFT, fill=X, expand=YES)
            f = Frame(self.master)
            f.pack(side=RIGHT, fill=BOTH, expand=YES)
            lab = Label(self.master, text='我的标签', font=24)
            lab.pack(side=TOP, fill=BOTH, expand=YES)
            bn = Button(self.master, text='我的按钮')
            bn.pack()
            '''
            # ttk 使用 Combobox 取代了 Listbox
            cb = ttk.Combobox(self.master, font=24)
            # 为 Combobox 设置列表项
            cb['values'] = ('Python', 'Swift', 'Kotlin')
            cb.pack(side=LEFT, fill=X, expand=YES)
            f = ttk.Frame(self.master)
            f.pack(side=RIGHT, fill=BOTH, expand=YES)
            lab = ttk.Label(self.master, text='我的标签', font=24)
            lab.pack(side=TOP, fill=BOTH, expand=YES)
            bn = ttk.Button(self.master, text='我的按钮')
            bn.pack()
            '''
root = Tk()
root.title("简单事件处理")
App(root)
root.mainloop()
```

上面程序中，被注释的代码是使用 ttk 组件的代码，未被注释的代码是直接使用 Tkinter 组件的代码。直接运行上面程序，可以看到如图 1 所示的界面。

图 1 Tkinter 组件的运行界面

如果将上面程序中被注释的代码取消注释，并注释使用 Listbox 组件的代码，改为使用 ttk 组件，再次运行上面程序，则可以看到如图 2 所示的界面。

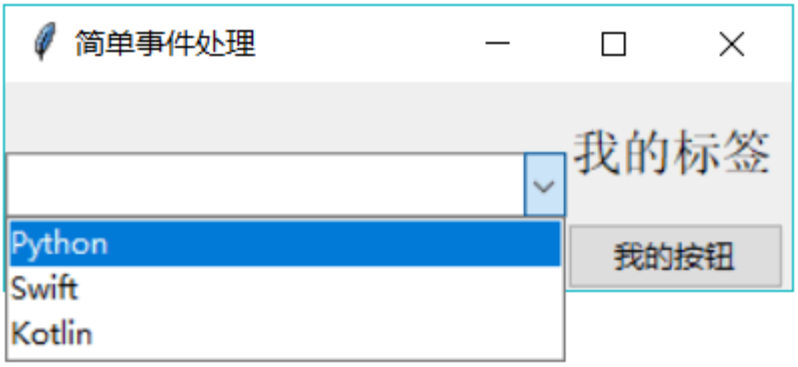


图 2 ttk 组件的运行界面

对比两个界面上 Tkinter 的 Button 和 ttk 的 Button，不难发现 ttk 下的 Button 更接近 Windows 7 本地平台的风格，显得更漂亮，这就是 ttk 组件的优势。

8.Python Tkinter Variable

Tkinter 支持将很多 GUI 组件与变量进行双向绑定，执行这种双向绑定后编程非常方便：

- 如果程序改变变量的值，GUI 组件的显示内容或值会随之改变。
- 当 GUI 组件的内容发生改变时（比如用户输入），变量的值也会随之改变。

为了让 Tkinter 组件与变量进行双向绑定，只要为这些组件指定 variable（通常绑定组件的 value）、textvariable（通常绑定组件显示的文本）等属性即可。

但这种双向绑定有一个限制，就是 Tkinter 不允许将组件和普通变量进行绑定，只能和 tkinter 包下 Variable 类的子类进行绑定。该类包含如下几个子类：

- ShringVar()：用于包装 str 值的变量。
- IntVar()：用于包装整型值的变量。
- DoubleVar()：用于包装浮点值的变量。
- BooleanVar()：用于包装 bool 值的变量。

对于 Variable 变量而言，如果要设置其保存的变量值，则使用它的 set() 方法；如果要得到其保存的变量值，则使用它的 get() 方法。

下面程序示范了将 Entry 组件与 StringVar 进行双向绑定，这样程序既可通过该

StringVar 改变 Entry 输入框显示的内容，也可通过该 StringVar 获取 Entry 输入框中的内容：

纯文本复制

```
from tkinter import *
# 导入 ttk
from tkinter import ttk

class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        self.st = StringVar()
        # 创建 Entry 组件，将其 textvariable 绑定到 self.st 变量
        ttk.Entry(self.master, textvariable=self.st,
            width=24,
            font=('StSong', 20, 'bold'),
            foreground='red').pack(fill=BOTH, expand=YES)
        # 创建 Frame 作为容器
        f = Frame(self.master)
        f.pack()
        # 创建两个按钮，将其放入 Frame 中
        ttk.Button(f, text='改变', command=self.change).pack(side=LEFT)
        ttk.Button(f, text='获取', command=self.get).pack(side=LEFT)
    def change(self):
        books = ('疯狂 Python 讲义', '疯狂 Kotlin 讲义', '疯狂 Swift 讲义')
        import random
        # 改变 self.st 变量的值，与之绑定的 Entry 的内容随之改变
        self.st.set(books[random.randint(0, 2)])
    def get(self):
        from tkinter import messagebox
        # 获取 self.st 变量的值，实际上就是获取与之绑定的 Entry 中的内容
        # 并使用消息框显示 self.st 变量的值
```

```
31.                 messagebox.showinfo(title='输入内容', message=self.st.get() )
32.         root = Tk()
33.         root.title("variable 测试")
34.         App(root)
35.         root.mainloop()
```

上面程序中第 26 行代码调用 StringVar 改变变量的值，这样与该变量绑定的 Entry 中显示的内容会随之改变；第 31 行代码获取 StringVar 变量的值，实际上就是获取与该变量绑定的 Entry 中的内容。

运行该程序，单击界面上的“改变”按钮，将可以看到输入框中的内容会随之改变；如果单击界面上的“获取”按钮，将会看到程序弹出一个消息框，显示了用户在 Entry 输入框中输入的内容。

9.Python Tkinter compound 选项使用方法

程序可以为按钮或 Label 等组件同时指定 text 和 image 两个选项，其中 text 用于指定该组件上的文本；image 用于显示该组件上的图片，当同时指定这两个选项时，通常 image 会覆盖 text。

但在某些时候，程序希望该组件能同时显示文本和图片，此时就需要通过 compound 选项进行控制。

compound 选项支持如下属性值：

None：图片覆盖文字。
LEFT 常量（值为‘left’字符串）：图片在左，文本在右。
RIGHT 常量（值为‘right’字符串）：图片在右，文本在左。
TOP 常量（值为‘top’字符串）： 图片在上，文本在下。
BOTTOM 常量（值为‘bottom’字符串）：图片在底，文本在上。
CENTER 常量（值为‘center’字符串）：文本在图片上方。

下面程序使用多个单选钮来控制 Label 的 compound 选项：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk

class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        # 创建一个位图
        bm = PhotoImage(file = 'serial.png')
        # 创建一个 Label，同时指定 text 和 image
        self.label = ttk.Label(self.master, text='学编程\n 神器',\
            image=bm, font=('StSong', 20, 'bold'), foreground='red' )
        self.label.bm = bm
        # 设置 Label 默认的 compound 为 None
        self.label['compound'] = None
        self.label.pack()
        # 创建 Frame 容器，用于装多个 Radiobutton
        f = ttk.Frame(self.master)
        f.pack(fill=BOTH, expand=YES)
        compounds = ('None', "LEFT", "RIGHT", "TOP", "BOTTOM", "CENTER")
        # 定义一个 StringVar 变量，用作绑定 Radiobutton 的变量
        self.var = StringVar()
        self.var.set('None')
        # 使用循环创建多个 Radionbutton 组件
        for val in compounds:
            rb = Radiobutton(f,
                text = val,
                padx = 20,
                variable = self.var,
                command = self.change_compound,
```

```
33.                         value = val).pack(side=LEFT, anchor=CENTER)
34.             # 实现 change_compound 方法，用于动态改变 Label 的 compound 选项
35.             def change_compound(self):
36.                 self.label['compound'] = self.var.get().lower()
37.         root = Tk()
38.         root.title("compound 测试")
39.         App(root)
40.         root.mainloop()
```

上面程序中第 16 行代码设置 Label 默认的 compound 选项为 None，这意味着该 Label 默认图片覆盖文字； 第 35 行代码会根据单选钮的值（单选钮与 self.var 绑定）来确定 Label 的 compound 选项。

运行该程序，将会看到 Label 中只显示图片，并不显示文字，这就是 compound 选项为 None 的效果。随着用户单击下面的单选钮，将可以看到 Label 上图片和文字的位置的改变，如图 1 所示。

图 1 将 compound 设为 RIGHT 让图片居右

10.Python Tkinter Entry 和 Text 控件用法(包含两者区别)

Entry 和 Text 组件都是可接收用户输入的输入框组件，区别是 Entry 是单行输入框组件，Text 是多行输入框组件，而且 Text 可以为不同的部分添加不同的格式，甚至响应事件。

不管是 Entry 还是 Text 组件，程序都提供了 get() 方法来获取文本框中的内容；但如果程序要改变文本框中的内容，则需要调用二者的 insert() 方法来实现。

如果要删除 Entry 或 Text 组件中的部分内容，则可通过 delete(self,first,last=None) 方法实现，该方法指定删除从 first 到 last 之间的内容。

关于 Entry 和 Text 支持的索引需要说明一下，由于 Entry 是单行文本框组件，因此它的索引很简单，比如要指定第 4 个字符到第 8 个字符，索引指定为 (3,8) 即可。但 Text 是多行文本框组件，因此它的索引需要同时指定行号和列号，比如 1.0 代表第 1 行、第 1 列

（行号从 1 开始， 列号从 0 开始），如果要指定第 2 行第 3 个字符到第 3 行第 7 个字符，索引应指定为 (2.2,3.6)。

此外，正如从前面程序所看到的，Entry 支持双向绑定，程序可以将 Entry 与变量绑定在一起，这样程序就可以通过该变量来改变、获取 Entry 组件中的内容。

下面程序示范了 Entry 和 Text 组件的用法：

```
1.          from tkinter import *
2.          # 导入 ttk
3.          from tkinter import ttk
4.          from tkinter import messagebox
5.          class App:
6.              def __init__(self, master):
7.                  self.master = master
8.                  self.initWidgets()
9.              def initWidgets(self):
10.                 # 创建 Entry 组件
11.                 self.entry = ttk.Entry(self.master,
12.                     width=44,
13.                     font=('StSong', 14),
14.                     foreground='green')
15.                 self.entry.pack(fill=BOTH, expand=YES)
16.                 # 创建 Entry 组件
17.                 self.text = Text(self.master,
18.                     width=44,
19.                     height=4,
20.                     font=('StSong', 14),
21.                     foreground='gray')
22.                 self.text.pack(fill=BOTH, expand=YES)
23.                 # 创建 Frame 作为容器
24.                 f = Frame(self.master)
25.                 f.pack()
26.                 # 创建五个按钮，将其放入 Frame 中
27.                 ttk.Button(f, text='开始插入',
    command=self.insert_start).pack(side=LEFT)
```

```
28.                 ttk.Button(f, text='编辑处插入',
    command=self.insert_edit).pack(side=LEFT)
29.                 ttk.Button(f, text='结尾插入',
    command=self.insert_end).pack(side=LEFT)
30.                 ttk.Button(f, text='获取 Entry',
    command=self.get_entry).pack(side=LEFT)
31.                 ttk.Button(f, text='获取 Text',
    command=self.get_text).pack(side=LEFT)
32.             def insert_start(self):
33.                 # 在 Entry 和 Text 开始处插入内容
34.                 self.entry.insert(0, 'Kotlin')
35.                 self.text.insert(1.0, 'Kotlin')
36.             def insert_edit(self):
37.                 # 在 Entry 和 Text 的编辑处插入内容
38.                 self.entry.insert(INSERT, 'Python')
39.                 self.text.insert(INSERT, 'Python')
40.             def insert_end(self):
41.                 # 在 Entry 和 Text 的结尾处插入内容
42.                 self.entry.insert(END, 'Swift')
43.                 self.text.insert(END, 'Swift')
44.             def get_entry(self):
45.                 messagebox.showinfo(title='输入内容',
    message=self.entry.get())
46.             def get_text(self):
47.                 messagebox.showinfo(title='输入内容',
    message=self.text.get(1.0, END))
48.         root = Tk()
49.         root.title("Entry 测试")
50.         App(root)
51.         root.mainloop()
```

上面程序开始创建了一个 Entry 组件和一个 Text 组件，程序中第 34、35 行代码用于在 Entry 和 Text 组件的开始部分插入指定文本内容，如果要在 Entry、Text 的指定位置插入文本内容，通过 insert() 方法的第一个参数指定位置即可。如果要在编辑处插入内容，则将第一个参数设为 INSERT 常量（值为 'insert'）； 如果要在结尾处插入内容，则将第一个参数设为 END 常量（值为 'end'）。

同时，程序第 45、47 行代码调用了 Entry 和 Text 组件的 get() 方法来获取其中的文本内容。运行上面程序，尝试向 Entry、Text 中插入一些内容，然后单击界面上的“获取 Text”按钮，则可以看到如图 1 所示的界面。

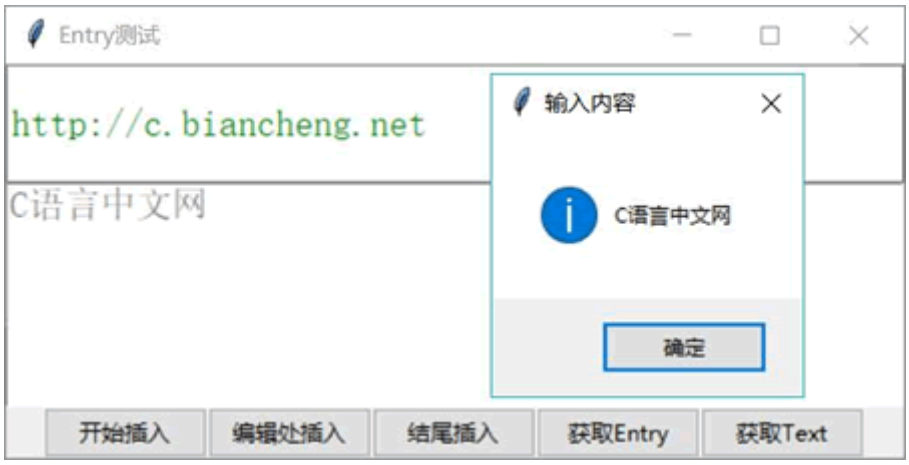


图 1 获取文本框中的内容

Text 实际上是一个功能强大的“富文本”编辑组件，这意味着使用 Text 不仅可以插入文本内容，也可以插入图片，可通过 image_create(self, index, cnf={}, **kw) 方法来插入。

Text 也可以设置被插入文本内容的格式，此时就需要为 insert(self, index, chars, *args) 方法的最后一个参数传入多个 tag 进行控制，这样就可以使用 Text 组件实现图文并茂的效果。

此外，当 Text 内容较多时就需要对该组件使用滚动条，以便该 Text 能实现被动显示。为了让滚动条控制 Text 组件内容的滚动，实际上就是将它们进行双向关联。这里需要两步操作：
将 Scrollbar 的 command 设为目标组件的 xview 或 yview，其中 xview 用于水平滚动条控制目标组件水平滚动；yview 用于垂直读动条控制目标组件垂直滚动。
将目标组件的 xscrollcommand 或 yscrollcommand 属性设为 Scrollbar 的 set 方法。

如下程序示范了使用 Text 来实现一个图文并茂的界面：

```
1.          from tkinter import *
2.          # 导入 ttk
3.          from tkinter import ttk
4.          class App:
5.              def __init__(self, master):
6.                  self.master = master
7.                  self.initWidgets()
```

```
8.              def initWidgets(self):
9.                  # 创建第一个 Text 组件
10.                 text1 = Text(self.master, height=27, width=32)
11.                 # 创建图片
12.                 book = PhotoImage(file='serial.png')
13.                 text1.bm = book
14.                 text1.insert(END,'\n')
15.                 # 在结尾处插入图片
16.                 text1.image_create(END, image=book)
17.                 text1.pack(side=LEFT, fill=BOTH, expand=YES)
18.                 # 创建第二个 Text 组件
19.                 text2 = Text(self.master, height=33, width=50)
20.                 text2.pack(side=LEFT,  fill=BOTH, expand=YES)
21.                 self.text = text2
22.                 # 创建 Scrollbar 组件，设置该组件与 text2 的纵向滚动关联
23.                 scroll = Scrollbar(self.master, command=text2.yview)
24.                 scroll.pack(side=RIGHT, fill=Y)
25.                 # 设置 text2 的纵向滚动影响 scroll 滚动条
26.                 text2.configure(yscrollcommand=scroll.set)
27.                 # 配置名为 title 的样式
28.                 text2.tag_configure('title', font=('楷体', 20, 'bold'),
29.                     foreground='red', justify=CENTER, spacing3=20)
30.                 text2.tag_configure('detail', foreground='darkgray',
31.                     font=('微软雅黑', 11, 'bold'),
32.                     spacing2=10, # 设置行间距
33.                     spacing3=15) # 设置段间距
34.                 text2.insert(END,'\n')
35.                 # 插入文本内容，设置使用 title 样式
36.                 text2.insert(END,'C 语言中文网\n', 'title')
37.                 # 创建图片
38.                 star = PhotoImage(file='serial.png')
39.                 text2.bm = star
40.                 details = ('C 语言中文网成立于 2012 年初，目前已经运营了将近 5
    年，' +\
41.                     '我们致力于分享精品教程，帮助对编程感兴趣的读者。\n',
```

```
                '我们一直都在坚持的是：认认真真、一丝不苟、以工匠的精神来打磨每一套教程，让读者感受到作者的用心，以及默默投入的时间，由衷地心动和点赞。\n',
                '这样的教程是一件作品，而不是呆板的文字！\n')
                # 采用循环插入多条介绍信息
                for de in details:
                    text2.image_create(END, image=star)
                    text2.insert(END, de, 'detail')
                url =['http://vip.biancheng.net/','http://c.biancheng.net/']
                name =['VIP 会员', 'C 语言中文网']
                m=0
                for each in name:
                    # 为每个链接创建单独的配置
                    text2.tag_configure(m, foreground='blue', underline=True,
                        font=('微软雅黑', 13, 'bold'))
                    text2.tag_bind(m, '<Enter>', self.show_arrow_cursor)
                    text2.tag_bind(m, '<Leave>', self.show_common_cursor)
                    # 使用 handlerAdaptor 包装,将当前链接参数传入事件处理函数
                    text2.tag_bind(m, '<Button-1>',
self.handlerAdaptor(self.click, x = url[m]))
                    text2.insert(END, each + '\n', m)
                    m += 1
            def show_arrow_cursor(self, event):
                # 光标移上去时变成箭头
                self.text.config(cursor='arrow')
            def show_common_cursor(self, event):
                # 光标移出去时恢复原样
                self.text.config(cursor='xterm')
            def click(self, event, x):
                import webbrowser
                # 使用默认浏览器打开链接
                webbrowser.open(x)
            def handlerAdaptor(self, fun,**kwds):
                return lambda event, fun=fun, kwds=kwds: fun(event,**kwds)
        root = Tk()
        root.title("Text 测试")
        App(root)
```

```
76.         root.mainloop()
```

上面程序中使用 Text 的 tag_configure（也写作 tag_config）方法创建了 title 和 detail 两个 tag，每个 tag 可用于控制一段文本的格式、事件等。

接下来程序使用 title tag 插入了一个标题内容，因此该标题内容的格式将受到 title tag 的控制；然后程序使用循环插入了三条受 detail tag 控制的描述信息，每次在插入描述信息之前都先插入一张图片。

不仅如此，上面程序在循环内创建了 tag，并调用 Text 组件的 tag_bind() 方法为 tag 绑定事件处理方法。与前面的描述信息不同的是，此处程序需要让每个链接打开不同的页面，因此程序为每条链接内容分别创建了不同的 tag，从而实现为每个链接打开对应的页面。

运行上面程序，将看到如图 2 所示的图文并茂的界面：

图 2 使用 Text 实现图文并茂的界面

11.Python Tkinter Radiobutton 和 Checkbutton 控件用法

Radiobutton 组件代表单选钮，该组件可以绑定一个方法或函数，当单选钮被选择时，该方法或函数将会被触发。

为了将多个 Radiobutton 编为一组，程序需要将多个 Radiobutton 绑定到同一个变量，当这组 Radiobutton 的其中一个单选钮被选中时，该变量会随之改变；反过来，当该变量发生改变时，这组 Radiobutton 也会自动选中该变量值所对应的单选钮。

下面程序示范了 Radiobutton 组件的用法：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
```

```
9.                  # 创建一个 Label 组件
10.                 ttk.Label(self.master, text='选择您喜欢的教程:')\
11.                     .pack(fill=BOTH, expand=YES)
12.                 self.intVar = IntVar()
13.                 # 定义元组
14.                 books = ('C 语言入门', 'Python 入门',
15.                     'C++入门', 'Java 入门')
16.                 i = 1
17.                 # 采用循环创建多个 Radiobutton
18.                 for book in books:
19.                     ttk.Radiobutton(self.master,
20.                         text = book,
21.                         variable = self.intVar, # 将 Radiobutton 绑定到
    self.intVar 变量
22.                         command = self.change, # 将选中事件绑定到 self.change 方
    法
23.                         value=i).pack(anchor=W)
24.                     i += 1
25.                 # 设置 Radiobutton 绑定的变量的值为 2,
26.                 # 则选中 value 为 2 的 Radiobutton
27.                 self.intVar.set(2)
28.             def change(self):
29.                 from tkinter import messagebox
30.                 # 通过 Radiobutton 绑定变量获取选中的单选框
31.                 messagebox.showinfo(title=None, message=self.intVar.get() )
32.         root = Tk()
33.         root.title("Radiobutton 测试")
34.         App(root)
35.         root.mainloop()
```

上面程序使用循环创建了多个 Radiobutton 组件，程序指定将这些 Radiobutton 绑定到 self.intVar 变量，这意味着这些 Radiobutton 位于同一组内；与此同时，程序为这组 Radiobutton 的选中事件绑定了 self.change 方法，因此每次当用户选择不同的单选钮时，总会触发该对象的 change() 方法。

运行上面程序，可以看到程序默认选中第二个单选钮，这是因为第二个单选钮的 value 为 2，而程序将这组单选钮绑定的 self.intVar 的值设置为 2；如果用户改变选中其他单选钮，程序将会弹出提示框显示用户的选择项，如图 1 所示：

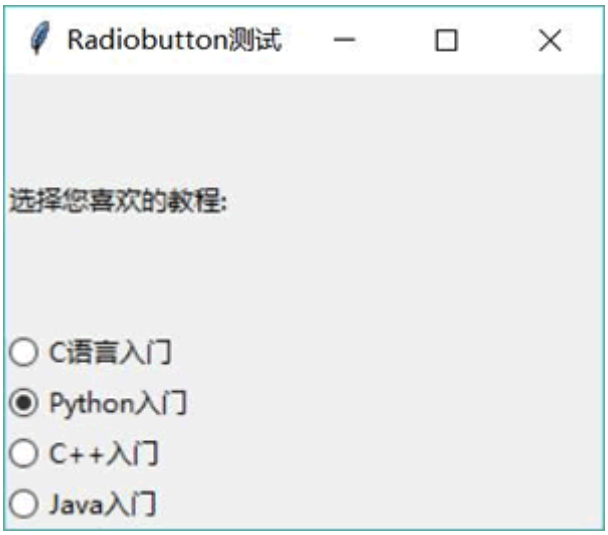


图 1 选中不同的单选钮

单边钮除了可以显示文本，也可以显示图片，只要为其指定 image 选项即可。如果希望图片和文字同时显示也是可以的，只要通过 compound 边项进行控制即可（如果不指定 compound 选项，该选项默认为 None，这意味着只显示图片）。如下程序示范了带图片的单选钮：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        # 创建一个 Label 组件
        ttk.Label(self.master, text='选择您喜欢的兵种:')\
            .pack(fill=BOTH, expand=YES)
        self.intVar = IntVar()
        # 定义元组
        races = ('z.png', 'p.png','t.png')
        raceNames = ('虫族', '神族','人族')
        i = 1
        # 采用循环创建多个 Radiobutton
```

```
                for rc in races:
                    bm = PhotoImage(file = 'images/' + rc)
                    r = ttk.Radiobutton(self.master,
                        image = bm,
                        text = raceNames[i - 1],
                        compound = RIGHT, # 图片在文字右边
                        variable = self.intVar, # 将 Radiobutton 绑定到 self.intVar 变量
                        command = self.change, # 将选中事件绑定到 self.change 方法
                        value=i)
                    r.bm = bm
                    r.pack(anchor=W)
                    i += 1
                # 设置默认选中 value 为 2 的单选按钮
                self.intVar.set(2)
            def change(self): pass
        root = Tk()
        root.title("Radiobutton 测试")
        # 改变窗口图标
        root.iconbitmap('images/fklogo.ico')
        App(root)
        root.mainloop()
```

上面程序为 RadioButton 同时指定了 image 和 text 选项，并指定 compound 选项为 RIGHT，这意味着该单选钮的图片显示在文字的右边。运行上面程序，可以看到如图 2 所示的运行界面。

图 2 带图标的单选钮

上面程序还重新设置了窗口图标，因此在运行界面上可以看到窗口图标是自定义的图标。

Checkbutton 与 Radiobutton 很相似，只是 Checkbutton 允许选择多项，而每组 Radiobutton 只能选择一项。其他功能基本相似，同样可以显示文字和图片，同样可以绑定变量，同样可以为选中事件绑定处理函数和处理方法。但由于 Checkbutton 可以同时选中多项，因此程序需要为每个 Checkbutton 都绑定一个变量。

Checkbutton 就像开关一样，它支持两个值：开关打开的值和开关关闭的值。因此，在创建 Checkbutton 时可同时设置 onvalue 和 offvalue 选项为打开和关闭分别指定值。如果不指定 onvalue 和 offvalue，则 onvalue 默认为 1，offvalue 默认为 0。

下面程序通过两组 Checkbutton 示范了 Checkbutton 的用法：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk
from tkinter import messagebox
class App:
    def __init__(self, master):
        self.master = master
```

```
8.                  self.initWidgets()
9.              def initWidgets(self):
10.                 # 创建一个 Label 组件
11.                 ttk.Label(self.master, text='选择您喜欢的人物:')\
12.                     .pack(fill=BOTH, expand=YES)
13.                 self.chars = []
14.                 # 定义元组
15.                 characters = ('孙悟空', '猪八戒','唐僧', '牛魔王')
16.                 # 采用循环创建多个 Checkbutton
17.                 for ch in characters:
18.                     intVar = IntVar()
19.                     self.chars.append(intVar)
20.                     cb = ttk.Checkbutton(self.master,
21.                         text = ch,
22.                         variable = intVar, # 将 Checkbutton 绑定到 intVar 变量
23.                         command = self.change) # 将选中事件绑定到 self.change 方
    法
24.                     cb.pack(anchor=W)
25.                 # 创建一个 Label 组件
26.                 ttk.Label(self.master, text='选择您喜欢的教程:')\
27.                     .pack(fill=BOTH, expand=YES)
28.                 # --------------下面是第二组 Checkbutton----------------
29.                 self.books = []
30.                 # 定义两个元组
31.                 books = ('C 语言基础教程', 'C++入门教程','Java 入门教程', 'Linux
    教程')
32.                 vals = ('C', 'C++','Java', 'Linux')
33.                 i = 0
34.                 # 采用循环创建多个 Checkbutton
35.                 for book in books:
36.                     strVar = StringVar()
37.                     self.books.append(strVar)
38.                     cb = ttk.Checkbutton(self.master,
39.                         text = book,
40.                         variable = strVar, # 将 Checkbutton 绑定到 strVar 变量
41.                         onvalue = vals[i],
```

```
                            offvalue = '无',
                            command = self.books_change) # 将选中事件绑定到
books_change 方法
                        cb.pack(anchor=W)
                        i += 1
            def change(self):
                # 将 self.chars 列表转换成元素为 str 的列表
                new_li = [str(e.get()) for e in self.chars]
                # 将 new_li 列表连接成字符串
                st = ', '.join(new_li)
                messagebox.showinfo(title=None, message=st)
            def books_change(self):
                # 将 self.books 列表转换成元素为 str 的列表
                new_li = [e.get() for e in self.books]
                # 将 new_li 列表连接成字符串
                st = ', '.join(new_li)
                messagebox.showinfo(title=None, message=st)
        root = Tk()
        root.title("Checkbutton 测试")
        # 改变窗口图标
        root.iconbitmap('favicon.ico')
        App(root)
        root.mainloop()
```

上面程序中第一组 Checkbutton 没有指定 onvalue 和 offvalue，因此它们的 onvalue 和 offvalue 默认分别为 1、0，所以程序将这组 Checkbutton 绑定到 IntVar 类型的变量；第二组 Checkbutton 将 onvalue 和 offvalue 都指定为字符串，因此程序将这组 Checkbutton 绑定到 StringVar 类型的变量。

运行该程序，选中“Java 入门教程”选项，可以看到如图 3 所示的运行效果。

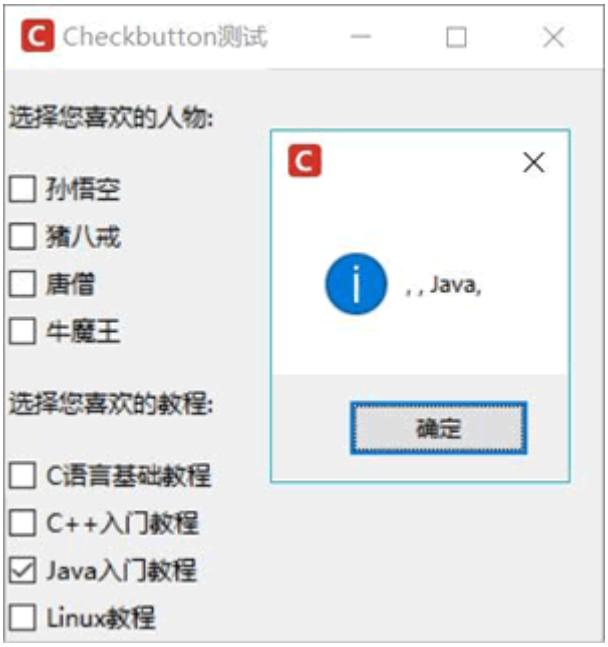


图 3 Checkbutton 组件

12.PythonTkinter Listbox 和 Combobox 控件用法

Listbox 代表一个列表框，用户可通过列表框来选择一个列表项。ttk 模块下的 Combobox 则是 Listbox 的改进版，它既提供了单行文本框让用户直接输入（就像 Entry 一样），也提供了下拉列表框供用户选择（就像 Listbox 一样），因此它被称为复合框。

程序创建 Listbox 起码需要两步：

1. 创建 Listbox 对象，并为之执行各种选项。Listbox 除支持大部分通用选项之外，还支持 selectmode 选项，用于设置 Listbox 的选择模式。
2. 调用 Listbox 的 insert(self, index, *elements) 方法来添加选项。从最后一个参数可以看出，该方法既可每次添加一个选项，也可传入多个参数，每次添加多个选项。index

参数指定选项的插入位置，它支持 END（结尾处）、ANCHOR（当前位置）和 ACTIVE（选中处）等特殊索引。

Listbox 的 selectmode 支持的选择模式有如下几种：

- 'browse'：单选模式，支持按住鼠标键拖动来改变选择。
- 'multiple'：多边模式。
- 'single'：单边模式，必须通过鼠标键单击来改变选择。
- 'extended'：扩展的多边模式，必须通过 CtrL 或 Shift 键辅助实现多选。

下面程序示范了 Listbox 的基本用法：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        topF = Frame(self.master)
        topF.pack(fill=Y, expand=YES)
        # 创建 Listbox 组件
        self.lb = Listbox(topF)
        self.lb.pack(side=LEFT, fill=Y, expand=YES)
        for item in ['Python', 'Kotlin', 'Swift', 'Ruby']:
            self.lb.insert(END, item)
        # 或直接使用多个元素来插入
        self.lb.insert(ANCHOR, 'Python', 'Kotlin', 'Swift', 'Ruby')
```

```
18.                 # 创建 Scrollbar 组件，设置该组件与 self.lb 的纵向滚动关联
19.                 scroll = Scrollbar(topF, command=self.lb.yview)
20.                 scroll.pack(side=RIGHT, fill=Y)
21.                 # 设置 self.lb 的纵向滚动影响 scroll 滚动条
22.                 self.lb.configure(yscrollcommand=scroll.set)
23.                 f = Frame(self.master)
24.                 f.pack()
25.                 Label(f, text = '选择模式:').pack(side=LEFT)
26.                 modes = ('multiple', 'browse', 'single', 'extended')
27.                 self.strVar = StringVar()
28.                 for m in modes:
29.                     rb = ttk.Radiobutton(f, text = m, value = m,
30.                         variable = self.strVar, command = self.choose_mode)
31.                     rb.pack(side=LEFT)
32.                 self.strVar.set('browse')
33.             def choose_mode(self):
34.                 print(self.strVar.get())
35.                 self.lb['selectmode'] = self.strVar.get()
36.         root = Tk()
37.         root.title("Listbox 测试")
38.         # 改变窗口图标
39.         root.iconbitmap('images/fklogo.ico')
40.         App(root)
41.         root.mainloop()
```

上面程序中，第 15 行代码表示每次插入一个选项，因此程序使用循环来控制插入多个选项；第 17 行代码则表示直接插入多个选项；第 35 行代码根据用户选择来改变 Listbox 的 Selectmode 选项，这样读者可以体会 Listbox 不同选项的差异。

运行上面程序，可以看到如图 1 所示的效果：

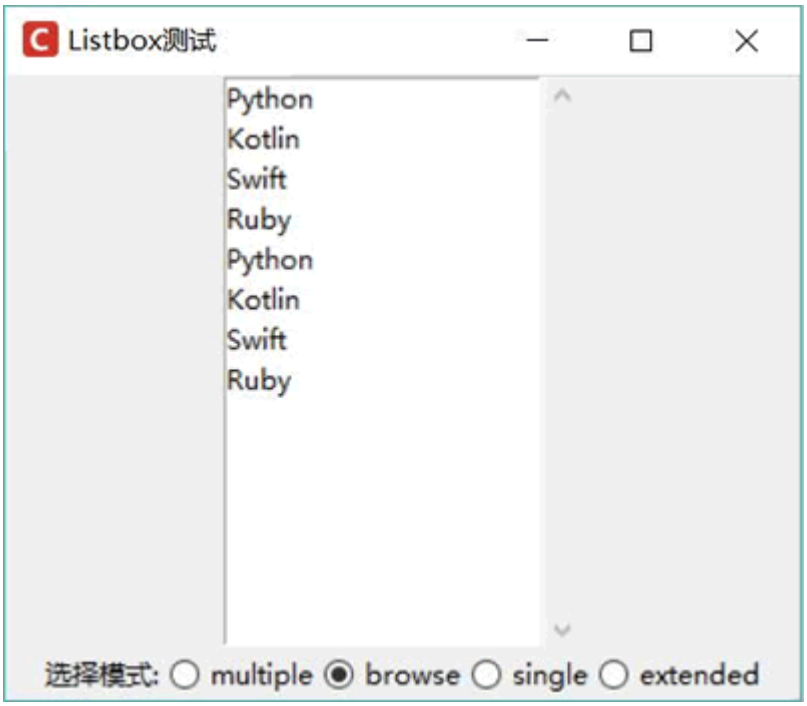


图 1 Listbox 的运行效果

除了最常见的 insert() 方法，Listbox 还支持如下常见的操作列表项的方法：

- selection_set(self, first, last=None)：选中从 first 到 last（包含）的所有列表项。如果不指定 last，则直接选中 first 列表项。
- selection_clear(self, first, last=None)：取消选中从 first 到 last（包含）的所有列表项。如果不指定 last，则只取消选中 first 列表项。
- delete(self, first, last=None)：删除从 first 到 last（包含）的所有列表项。如果不指定 last，则只删除 first 列表项。

Listbox 也支持使用 Iistvariable 选项与变量进行绑定，但这个变量并不是控制 Listbox 选中哪些项，而是控制 Listbox 包含哪些项。简单来说，如果 listvariable 选项与变量进

行了双向绑定，则无须调用 insert()、delete() 方法来添加、删除列表项，只要通过绑定变量即可改变 Listbox 中的列表项。

下面程序示范了操作 Listbox 中选项的方法：

```
1.          from tkinter import *
2.          # 导入 ttk
3.          from tkinter import ttk
4.          class App:
5.              def __init__(self, master):
6.                  self.master = master
7.                  self.initWidgets()
8.              def initWidgets(self):
9.                  topF = Frame(self.master)
10.                 topF.pack(fill=Y, expand=YES)
11.                 # 定义 StringVar 变量
12.                 self.v = StringVar()
13.                 # 创建 Listbox 组件，与 v 变量绑定
14.                 self.lb = Listbox(topF, listvariable = self.v)
15.                 self.lb.pack(side=LEFT, fill=Y, expand=YES)
16.                 for item in range(20):
17.                     self.lb.insert(END, str(item))
18.                 # 创建 Scrollbar 组件，设置该组件与 self.lb 的纵向滚动关联
19.                 scroll = Scrollbar(topF, command=self.lb.yview)
20.                 scroll.pack(side=RIGHT, fill=Y)
21.                 # 设置 self.lb 的纵向滚动影响 scroll 滚动条
22.                 self.lb.configure(yscrollcommand=scroll.set)
23.                 f = Frame(self.master)
24.                 f.pack()
25.                 Button(f, text="选中 10 项",
    command=self.select).pack(side=LEFT)
26.                 Button(f, text="清除选中 3 项",
    command=self.clear_select).pack(side=LEFT)
27.                 Button(f, text="删除 3 项", command=self.delete).pack(side=LEFT)
```

```
28.                 Button(f, text="绑定变量",
    command=self.var_select).pack(side=LEFT)
29.             def select(self):
30.                 # 选中指定项
31.                 self.lb.selection_set(0, 9)
32.             def clear_select(self):
33.                 # 取消选中指定项
34.                 self.lb.selection_clear(1,3)
35.             def delete(self):
36.                 # 删除指定项
37.                 self.lb.delete(5, 8)
38.             def var_select(self):
39.                 # 修改与 Listbox 绑定的变量
40.                 self.v.set(('12', '15'))
41.         root = Tk()
42.         root.title("Listbox 测试")
43.         # 改变窗口图标
44.         root.iconbitmap('images/fklogo.ico')
45.         App(root)
46.         root.mainloop()
```

上面程序中第 31 行代码控制选中列表项中第一个到第十个选项；第 34 行代码控制取消选中列表项中的 3 项； 第 37 行代码删除列表项中的 4 项；第 40 行代码通过绑定变量来改变 Listbox 中的列表项。运行上面程序，删除其中 4 项之后的运行效果如图 2 所示。

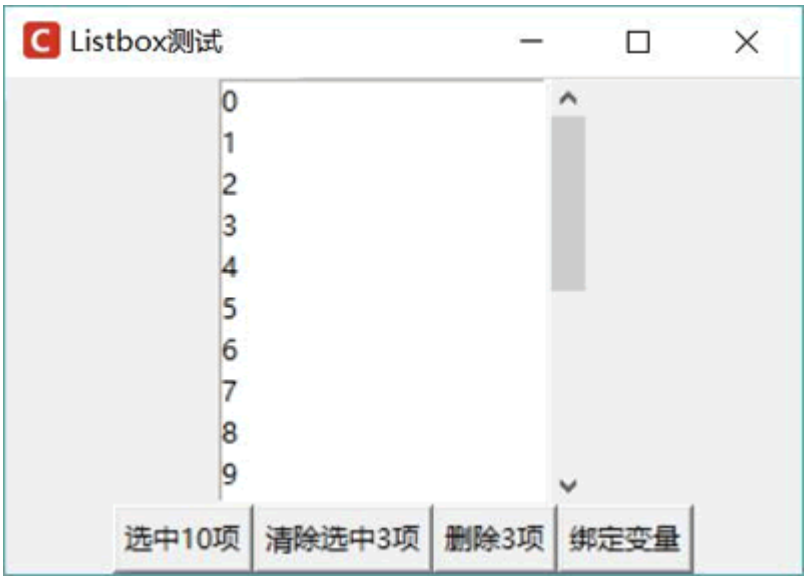


图 2 操作列表项

如果程序要获取 Listbox 当前边中的项，则可通过 curselection() 方法来实现，该方法会返回一个元组，该元组包含当前 Listbox 的所有选中项。

Listbox 并不支持使用 command 选项来绑定事件处理函数或方法，如果程序需要为 Listbox 绑定事件处理函数或方法，则可通过 bind() 方法来实现。下面程序示范了通过 bind() 方法为 Listbox 绑定事件处理方法：

```
1.          from tkinter import *
2.          # 导入 ttk
3.          from tkinter import ttk
4.          class App:
5.              def __init__(self, master):
6.                  self.master = master
7.                  self.initWidgets()
8.              def initWidgets(self):
9.                  topF = Frame(self.master)
```

```
10.                 topF.pack(fill=Y, expand=YES)
11.                 # 创建 Listbox 组件
12.                 self.lb = Listbox(topF)
13.                 self.lb.pack(side=LEFT, fill=Y, expand=YES)
14.                 for item in range(20):
15.                     self.lb.insert(END, str(item))
16.                 # 创建 Scrollbar 组件，设置该组件与 self.lb 的纵向滚动关联
17.                 scroll = Scrollbar(topF, command=self.lb.yview)
18.                 scroll.pack(side=RIGHT, fill=Y)
19.                 # 设置 self.lb 的纵向滚动影响 scroll 滚动条
20.                 self.lb.configure(yscrollcommand=scroll.set)
21.                 # 为双击事件绑定事件处理方法
22.                 self.lb.bind("<Double-1>", self.click)
23.             def click(self, event):
24.                 from tkinter import messagebox
25.                 # 获取 Listbox 当前选中项
26.                 messagebox.showinfo(title=None,
    message=str(self.lb.curselection()))
27.         root = Tk()
28.         root.title("Listbox 测试")
29.         # 改变窗口图标
30.         root.iconbitmap('images/fklogo.ico')
31.         App(root)
32.         root.mainloop()
```

上面程序中第 22 行代码为 Listbox 的左键双击事件（<Double-1> 绑定了事件处理方法，当用户双击 Listbox 时，程序将会触发该对象的 click 方法；在第 26 行代码则调用了 Listbox 的 curselection() 方法来获取当前选中项。

运行上面程序，双击某个列表项，将可以看到如图 3 所示的运行效果。

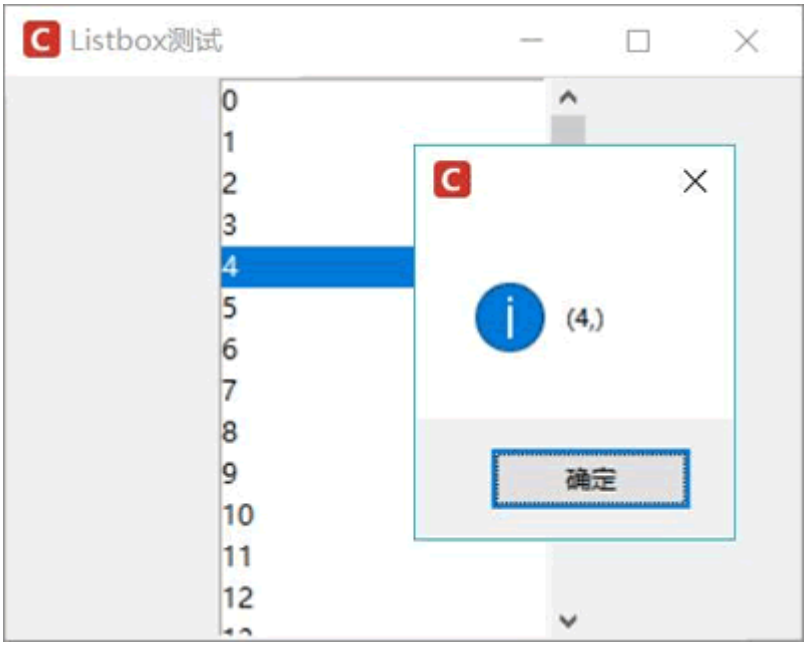


图 3 为双击事件绑定事件处理方法

Combobox 的用法更加简单，程序可通过 values 选项直接为它设置多个选项。该组件的 state 选项支持‘readonly’状态，该状态代表 Combobox 的文本框不允许编辑，只能通过下拉列表框的列表项来改变。

Combobox 同样可通过 textvariable 选项将它与指定变量绑定，这样程序可通过该变量来获取或修改 Combobox 组件的值。

Combobox 还可通过 postcommand 选项指定事件处理函数或方法，当用户单击 Combobox 的下拉箭头时，程序就会触发 postcomrnand 选项指定的事件处理函数或方法。

下面程序示范了 Combobox 组件的用法：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
```

```
8.              def initWidgets(self):
9.                  self.strVar = StringVar()
10.                 # 创建 Combobox 组件
11.                 self.cb = ttk.Combobox(self.master,
12.                     textvariable=self.strVar, # 绑定到 self.strVar 变量
13.                     postcommand=self.choose) # 当用户单击下拉箭头时触发
    self.choose 方法
14.                 self.cb.pack(side=TOP)
15.                 # 为 Combobox 配置多个选项
16.                 self.cb['values'] = ['Python', 'Ruby', 'Kotlin', 'Swift']
17.                 f = Frame(self.master)
18.                 f.pack()
19.                 self.isreadonly = IntVar()
20.                 # 创建 Checkbutton，绑定到 self.isreadonly 变量
21.                 Checkbutton(f, text = '是否只读:',
22.                     variable=self.isreadonly,
23.                     command=self.change).pack(side=LEFT)
24.                 # 创建 Button，单击该按钮激发 setvalue 方法
25.                 Button(f, text = '绑定变量设置',
26.                     command=self.setvalue).pack(side=LEFT)
27.             def choose(self):
28.                 from tkinter import messagebox
29.                 # 获取 Combbox 的当前值
30.                 messagebox.showinfo(title=None, message=str(self.cb.get()))
31.             def change(self):
32.                 self.cb['state'] = 'readonly' if self.isreadonly.get() else
    'enable'
33.             def setvalue(self):
34.                 self.strVar.set('我爱 Python')
35.         root = Tk()
36.         root.title("Combobox 测试")
37.         # 改变窗口图标
38.         root.iconbitmap('images/fklogo.ico')
39.         App(root)
40.         root.mainloop()
```

上面程序中
第 12 行代码将 Combobox 组件绑定到 self.strVar 变量；
第 13 行代码为 Combobox 的 command 绑定了事件处理方法；
第 32 行代码根据列表框的值来确定 Combobox 是否允许编辑。

运行上面程序，可以看到如图 4 所示的运行界面：

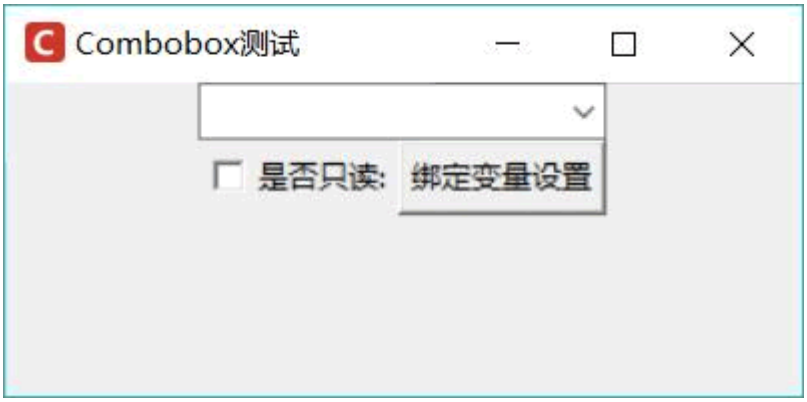


图 4 Combobox 组件

13.Python Tkinter Spinbox

Spinbox 控件是一个带有两个小箭头的文本框，用户既可以通过两个小箭头上下调整该组件内的值，也可以直接在文本框内输入内容作为该组件的值。

Spinbox 本质上也相当于持有一个列表框，这一点类似于 Combobox，但 Spinbox 不会展开下拉列表供用户选择。Spinbox 只能通过向上、向下箭头来选择不同的选项。

在使用 Spinbox 组件时，既可通过 from（由于 from 是关键字，实际使用时写成 from_）、to、increment 选项来指定选项列表，也可通过 values 选项来指定多个列表项，该选项的值可以是 list 或 tuple。

Spinbox 同样可通过 textvariable 选项将它与指定变量绑定，这样程序即可通过该变量来获取或修改 Spinbox 组件的值。

Spinbox 还可通过 command 选项指定事件处理函数或方法，即当用户单击 Spinbox 的向上、向下箭头时，程序就会触发 command 选项指定的事件处理函数或方法。

下面程序示范了 Spinbox 组件的用法：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        ttk.Label(self.master, text='指定 from、to、increment').pack()
        # 通过指定 from_、to、increament 选项创建 Spinbox
        sb1 = Spinbox(self.master, from_ = 20,
            to = 100,
            increment = 5)
        sb1.pack(fill=X, expand=YES)
        ttk.Label(self.master, text='指定 values').pack()
        # 通过指定 values 选项创建 Spinbox
        self.sb2 = Spinbox(self.master,
            values=('Python', 'Swift', 'Kotlin', 'Ruby'),
            command = self.press) # 通过 command 绑定事件处理方法
        self.sb2.pack(fill=X, expand=YES)
        ttk.Label(self.master, text='绑定变量').pack()
        self.intVar = IntVar()
        # 通过指定 values 选项创建 Spinbox，并为之绑定变量
        sb3 = Spinbox(self.master,
            values=list(range(20, 100, 4)),
            textvariable = self.intVar, # 绑定变量
            command = self.press)
        sb3.pack(fill=X, expand=YES)
        self.intVar.set(33) # 通过变量改变 Spinbox 的值
    def press(self):
        print(self.sb2.get())
root = Tk()
root.title("Spinbox 测试")
# 改变窗口图标
root.iconbitmap('images/fklogo.ico')
App(root)
```

```
37.         root.mainloop()
```

上面程序中第 11~13 行代码使用 from_、to、increment 选项创建了 Spinbox 组件；第 17~19 行代码使用 values 选项创建了 Spinbox 组件，并为该组件的 command 选项指定了事件处理方法，因此当单击 Spinbox 的向上、向下箭头调整值时，该选项指定的事件处理方法就会被触发；第 24-27 行代码在创建 Spinbox 时使用 textvariable 选项绑定了变量，这样程序完全可通过绑定变量来访问或修改该 Spinbox 组件的值。

运行上面程序，可以看到如图 1 所示的运行界面：

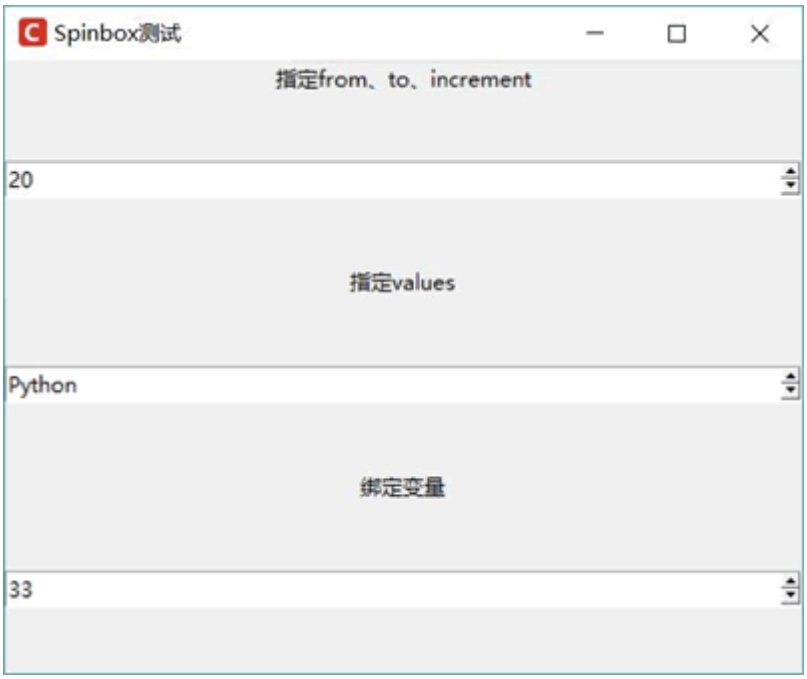


图 1 Spinbox 组件

14.Python Tkinter Scale 和 LabeledScale

Scale 组件代表一个滑动条，可以为该滑动条设置最小值和最大值，也可以设置滑动条每次调节的步长。

Scale 组件支持如下选项：

- from：设置该 Scale 的最小值。

- to：设置该 Scale 的最大值。
- resolution：设置该 Scale 滑动时的步长。
- label：为 Scale 组件设置标签内容。
- length：设置轨道的长度。
- width：设置轨道的宽度。
- troughcolor：设置轨道的背景色。
- sliderlength：设置轨道的长度。
- sliderrelief：设置滑块的立体样式。
- showvalue：设置是否显示当前值。
- orient：设置方向。该选项支持 VERTICAL 和 HORIZONTAL 两个值。
- digits：设置有效数字至少要有几位。
- variable：用于与变量进行绑定。
- command：用于为该 Scale 组件绑定事件处理，函数或方法。

> 如果使用 ttk.Scale 组件，则更接近操作系统本地的效果，但允许定制的选项少。

下面以一个示例程序来介绍 Scale 组件的选项的功能和用法：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk

class App:
    def __init__(self, master):
        self.master = master
```

```
8.                  self.initWidgets()
9.              def initWidgets(self):
10.                 self.scale = Scale(self.master,
11.                     from_ = -100,  # 设置最大值
12.                     to = 100,  # 设置最小值
13.                     resolution = 5, # 设置步长
14.                     label = '示范 Sacle', # 设置标签内容
15.                     length = 400, # 设置轨道的长度
16.                     width = 30, # 设置轨道的宽度
17.                     troughcolor='lightblue', # 设置轨道的背景色
18.                     sliderlength=20, # 设置滑块的长度
19.                     sliderrelief=SUNKEN, # 设置滑块的立体样式
20.                     showvalue=YES, # 设置显示当前值
21.                     orient = HORIZONTAL  #设置水平方向
22.                 )
23.                 self.scale.pack()
24.                 # 创建一个 Frame 作为容器
25.                 f = Frame(self.master)
26.                 f.pack(fill=X, expand=YES, padx=10)
27.                 Label(f, text='是否显示值:').pack(side=LEFT)
28.                 i = 0
29.                 self.showVar = IntVar()
30.                 self.showVar.set(1)
31.                 # 创建两个 Radiobutton 控制 Scale 是否显示值
32.                 for s in ('不显示', '显示'):
33.                     Radiobutton(f, text=s, value=i,variable=self.showVar,
34.                     command=self.switch_show).pack(side=LEFT)
35.                     i += 1
36.                 # 创建一个 Frame 作为容器
37.                 f = Frame(self.master)
38.                 f.pack(fill=X, expand=YES, padx=10)
39.                 Label(f, text='方向:').pack(side=LEFT)
40.                 i = 0
41.                 self.orientVar = IntVar()
42.                 self.orientVar.set(0)
43.                 # 创建两个 Radiobutton 控制 Scale 的方向
```

```
                for s in ('水平', '垂直'):
                    Radiobutton(f, text=s, value=i,variable=self.orientVar,
                    command=self.switch_orient).pack(side=LEFT)
                    i += 1
            def switch_show(self):
                self.scale['showvalue'] = self.showVar.get()
            def switch_orient(self):
                # 根据单选框的选择设置 orient 选项的值
                self.scale['orient'] = VERTICAL if self.orientVar.get() else
HORIZONTAL
        root = Tk()
        root.title("Scale 测试")
        App(root)
        root.mainloop()
```

上面程序创建了 Scale 组件，并为该 Scale 组件指定了前面所介绍的选项。此外，程序还根据单选钮的选中状态来设置 Scale 组件的 showvalue 选项，该选项将会控制该 Scale 组件是否显示当前值。此程序会根据单选钮的选中状态设置 Scale 是水平的还是垂直的(根据 orient 选项进行设置) 。

运行上面程序，如果将 Scale 组件默认显示为水平滑动条，则效果如图 1 所示。

图 1 水平滑动条

如果将上面的 Scale 设置为垂直渭动条，并选中“不显示”单选钮，将会看到如图 2 所示的效果。

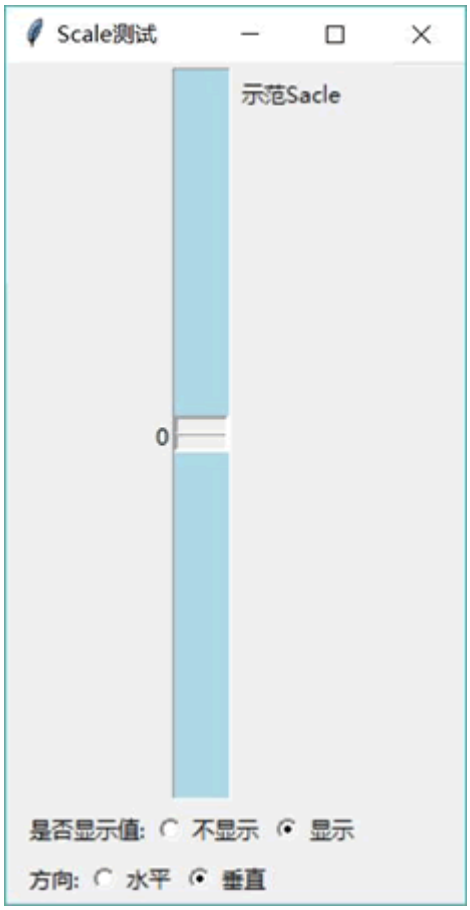


图 2 垂直滑动条

Scale 组件同样支持 variable 进行变量绑定，也支持使用 command 选项绑定事件处理函数或方法，这样每当用户拖动滑动条上的滑块时，都会触发 command 绑定的事件处理方法，不过 Scale 的事件处理方法比较奇葩，它可以额外定义一个参数，用于获取 Scale 的当前值。例如如下程序：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk

class App:
```

```
6.              def __init__(self, master):
7.                  self.master = master
8.                  self.initWidgets()
9.              def initWidgets(self):
10.                 # 定义变量
11.                 self.doubleVar = DoubleVar()
12.                 self.scale = Scale(self.master,
13.                     from_ = -100,  # 设置最大值
14.                     to = 100,  # 设置最小值
15.                     resolution = 5, # 设置步长
16.                     label = '示范 Sacle', # 设置标签内容
17.                     length = 400, # 设置轨道的长度
18.                     width = 30, # 设置轨道的宽度
19.                     orient = HORIZONTAL,  #设置水平方向
20.                     digits = 10, # 设置十位有效数字
21.                     command = self.change, # 绑定事件处理函数
22.                     variable = self.doubleVar # 绑定变量
23.                 )
24.                 self.scale.pack()
25.                 # 设置 Scale 的当前值
26.                 self.scale.set(20)
27.             # 这个事件处理函数比较奇葩，它可以接收到 Scale 的值
28.             def change(self, value):
29.                 print(value, self.scale.get(), self.doubleVar.get())
30.         root = Tk()
31.         root.title("Scale 测试")
32.         App(root)
33.         root.mainloop()
```

上面程序示范了通过三种方式来获取 Scale 组件的值：
通过事件处理方法的参数来获取。
通过 Scale 组件提供的 get() 方法来获取。
通过 Scale 组件绑定的变量来获取。

通过上面三种方式获取的变量值都是一样的，但由于 Scale 组件指定了 digits 选项（该选项指定 Scale 的值的有效数字至少保留几位）为 10，因此程序通过事件处理方法获取的值将有 10 位有效数字，如 -35.0000000。

ttk.LabeledScale 是平台化的滑动条，因此它允许设置的选项很少，只能设置 from、to 和 compound 等有限的几个选项，而且它总是生成一个水平滑动条（不能变成垂直的），其中 compound 选项控制滑动条的数值标签是显示在滑动条的上方，还是滑动条的下方。

下面程序示范了 LabeledScale 组件的功能和用法：

纯文本复制

```
from tkinter import *
# 导入 ttk
from tkinter import ttk

class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        self.scale = ttk.LabeledScale(self.master,
            from_ = -100,  # 设置最大值
            to = 100,  # 设置最小值
#             compound = BOTTOM # 设置显示数值的 Label 在下方
        )
        self.scale.value = -20
        self.scale.pack(fill=X, expand=YES)
root = Tk()
root.title("LabeledScale 测试")
App(root)
root.mainloop()
```

上面程序创建了一个 LabeledScale 组件，该组件会生成一个水平滑动条，并且滑动条的数值标签默认会显示在滑动条的上方；如果取消程序中被注释代码的注释，也就是将 compound 选项设为 BOTTOM，则意味着滑动条的数值标签默认会显示在滑动条的下方。

15.Python Tkinter LabelFrame

Labelframe 是 Frame 容器的改进版，它允许为容器添加一个标签，该标签既可以是普通的文字标签，也可以将任意 GUI 组件作为标签。

> 为了让 ttk.Labelframe 与 tkinter.LabelFrame 保持名字上的兼容，ttk 为 ttk.Labelframe 起了一个别名 ttk.LabelFrame（注意 f 的大小写），因此在程序中既可使用 ttk.Labelframe，也可使用 ttk.LabelFrame，它们二者完全相同。

为了给 Labelframe 设置文字标签，只要为它指定 text 选项即可。如下程序示范了 Labelframe 组件的用法：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        # 创建 Labelframe 容器
        lf = ttk.Labelframe(self.master, text='请选择教程',
            padding=20)
        lf.pack(fill=BOTH, expand=YES, padx=10, pady=10)
        books = ['C++', 'Python', 'Linux', 'Java']
        i = 0
        self.intVar = IntVar()
        # 使用循环创建多个 Radiobutton，并放入 Labelframe 中
        for book in books:
            Radiobutton(lf, text=book + '教程',
            value=i,
```

```
                    variable=self.intVar).pack(side=LEFT)
                    i += 1
        root = Tk()
        root.title("Labelframe 测试")
        # 改变窗口图标
        root.iconbitmap('1.ico')
        App(root)
        root.mainloop()
```

上面程序首先创建了一个简单的 Labelframe 组件，并为它指定了 text 选项，该选项的内容将会作为该容器的标签。接下来程序向 Labelframe 容器中添加了 4 个 Radiobutton。运行该程序，可以看到如图 1 所示的效果。

图 1 Labelframe 组件

Labelframe 允许通过如下选项对标签进行定制：

- labelwidget：设置可以将任意 GUI 组件作为标签。
- labelanchor：设置标签的位置。该选项支持 'e'、's'、'w'、'n'、'es'、'ws'、'en'、'wn'、'ne'、'nw'、'se'、'sw' 这 12 个边项值，用于控制标签的位置。

如下程序示范了对 Labelframe 的标签进行定制：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        # 创建 Labelframe 容器
        self.lf = ttk.Labelframe(self.master, padding=20)
        self.lf.pack(fill=BOTH, expand=YES, padx=10, pady=10)
        # 创建一个显示图片的 Label
        bm = PhotoImage(file='images/z.png')
        lb = Label(self.lf, image=bm)
        lb.bm = bm
        # 将 Labelframe 的标题设为显示图片的 Label
        self.lf['labelwidget'] = lb
        # 定义代表 Labelframe 的标题位置的 12 个常量
        self.books = ['e', 's', 'w', 'n', 'es', 'ws', 'en', 'wn',
            'ne', 'nw', 'se', 'sw']
        i = 0
        self.intVar = IntVar()
        # 使用循环创建多个 Radiobutton，并放入 Labelframe 中
        for book in self.books:
            Radiobutton(self.lf, text= book,
            value=i,
            command=self.change,
            variable=self.intVar).pack(side=LEFT)
            i += 1
        self.intVar.set(9)
    def change(self):
        # 通过 labelanchor 选项改变 Labelframe 的标题的位置
        self.lf['labelanchor'] = self.books[self.intVar.get()]
root = Tk()
root.title("Labelframe 测试")
# 改变窗口图标
```

```
37.         root.iconbitmap('images/fklogo.ico')
38.         App(root)
39.         root.mainloop()
```

上面程序通过 labelwidget 选项定制了该 Labelframe 的标签，该选项值指定为一个显示图片的 Label，因此该 Labelframe 的标签就是一张图片。与此同时，程序会根据单选钮的选中状态设置 Labelframe 的标签的位置。

运行该程序，改变 Labelframe 的标签的位置到右下角（se）处，将看到如图 2 所示的界面。

图 2 定制 Labelframe 的标签

16.Python Tkinter Panedwindow

Panedwindow 是一个管理窗口布局的容器，它允许添加多个子组件（不需要使用 Pack、Grid 或 Place 布局）并为每个子组件划分一个区域，用户可用鼠标移动各区域的分隔线来改变各子组件的大小（如果没有显式指定大小，子细件总是自动占满整个区域）。

ttk.Panedwindow 继承了 tkinter.PanedWindow，为了让 ttk.Panedwindow 与 tkinter.PanedWindow 保持名字上的兼容，ttk 为 ttk.Panedwindow 起了一个别名 ttk.PanedWindow（注意 w 的大小写），因此在程序中既可使用 ttk.Panedwindow，也可使用 ttk.PanedWindow，它们二者完全相同。

Panedwindow 是一个非常有特色的容器，它自带布局管理功能，它允许通过 orient 选项指定水平或垂直方向，让容器中的各组件按水平或垂直方向排列。

在创建 Panedwindow 之后，程序可通过如下方法操作 Panedwindow 容器中的子组件：

- add(self, child, **kw)：添加一个子组件。
- insert(self, pos, child,**kw)：在 pos 位置插入一个子组件。
- remove(self, child)：删除一个子组件，该子组件所在区域也被删除。

下面程序示范了为 Panedwindow 添加、插入、删除子组件：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk

class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        # 创建 Style
        style = ttk.Style()
```

```
12.                 style.configure("fkit.TPanedwindow", background='darkgray',
    relief=RAISED)
13.                 # 创建 Panedwindow 组件，通过 style 属性配置分隔线
14.                 pwindow = ttk.Panedwindow(self.master,
15.                     orient=VERTICAL, style="fkit.TPanedwindow")
16.                 pwindow.pack(fill=BOTH, expand=1)
17.                 first = ttk.Label(pwindow, text="第一个标签")
18.                 # 调用 add 方法添加组件，每个组件一个区域
19.                 pwindow.add(first)
20.                 okBn = ttk.Button(pwindow, text="第二个按钮",
21.                     # 调用 remove()方法删除组件，该组件所在区域消失
22.                     command=lambda : pwindow.remove(okBn))
23.                 # 调用 add 方法添加组件，每个组件一个区域
24.                 pwindow.add(okBn)
25.                 entry = ttk.Entry(pwindow, width=30)
26.                 # 调用 add 方法添加组件，每个组件一个区域
27.                 pwindow.add(entry)
28.                 # 调用 insert 方法插入组件
29.                 pwindow.insert(1, Label(pwindow, text='插入的标签'))
30.         root = Tk()
31.         root.title("Panedwindow 测试")
32.         App(root)
33.         root.mainloop()
```

上面程序中，第 11、12 行代码创建了一个 ttk.Style 对象，该对象专门用于管理他组件的样式，这样 ttk 组件即可通过 style 选项复用 ttk.Style 管理的样式。此处使用 ttk.Style 为 ttk.Panedwindow 指定样式，这样才能看到 ttk.Panedwindow 容器内的分隔线（默认是看不到的）。

与此同时，程序第 19 行代码调用了 add() 方法为 Panedwindow 容器添加子组件；第 22 行代码调用了 remove() 方法删除 Panedwindow 容器中的子组件；第 29 行代码调

用了 insert() 方法向 Panedwindow 容器中添加了子组件。

运行上面程序，将看到如图 1 所示的运行界面。

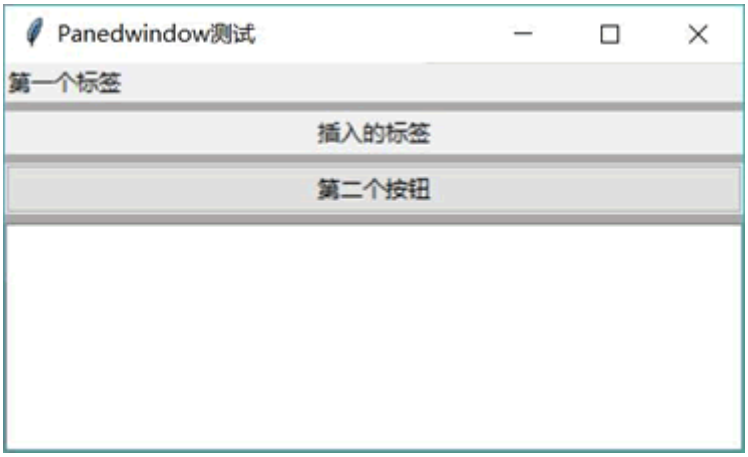


图 1 Panedwindow 组件

如果单击该界面上的“第二个按钮”，将会删除 Panedwindow 组件中的该按钮，该按钮所占的区域也会消失。此时将看到如图 2 所示的界面。

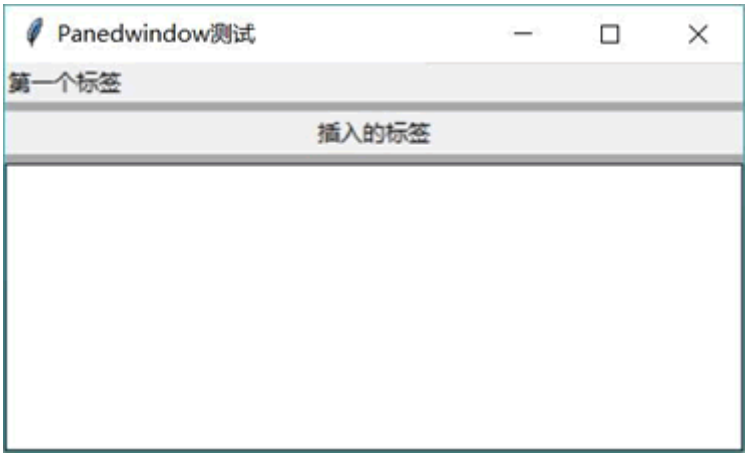


图 2 删除 Panedwindow 组件中的子组件

看到上面例子，可能有读者会想，Panedwindow 要么水平排列组件，要么垂直排列组件，这样功能不是太局限了吗？

请别忘记了，Panedwindow 组件同样也是可以嵌套的，以实现功能更丰富的界面。例如，如下程序在水平 Panedwindow 中嵌套了垂直 Panedwindow：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk

class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        # 创建 Style
        style = ttk.Style()
        style.configure("fkit.TPanedwindow",
            background='darkgray', relief=RAISED)
        # 创建 Panedwindow 组件，通过 style 属性配置分隔线
        pwindow = ttk.Panedwindow(self.master,
            orient=HORIZONTAL, style="fkit.TPanedwindow")
        pwindow.pack(fill=BOTH, expand=YES)
        left = ttk.Label(pwindow, text="左边标签", background='pink')
        pwindow.add(left)
        # 创建第二个 Panedwindow 组件，该组件的方向为垂直方向
        rightwindow = PanedWindow(pwindow, orient=VERTICAL)
        pwindow.add(rightwindow)
        top = Label(rightwindow, text="右上标签",
background='lightgreen')
        rightwindow.add(top)
```

```
25.                 bottom = Label(rightwindow, text="右下标签",
background='lightblue')
26.                 rightwindow.add(bottom)
27.         root = Tk()
28.         root.title("Panedwindow 测试")
29.         App(root)
30.         root.mainloop()
```

上面程序中，第 15 行代码创建了一个水平分布的 Panedwindow 容器，在该容器中先添加了一个 Label 组件；第 21 行代码创建了一个垂直分布的 Panedwindow 容器，该容器被添加到第一个 Panedwindow 容器中，这样就形成了嵌套，从而可以实现功能更丰富的界面。

运行上面程序，可以看到如图 3 所示的界面。

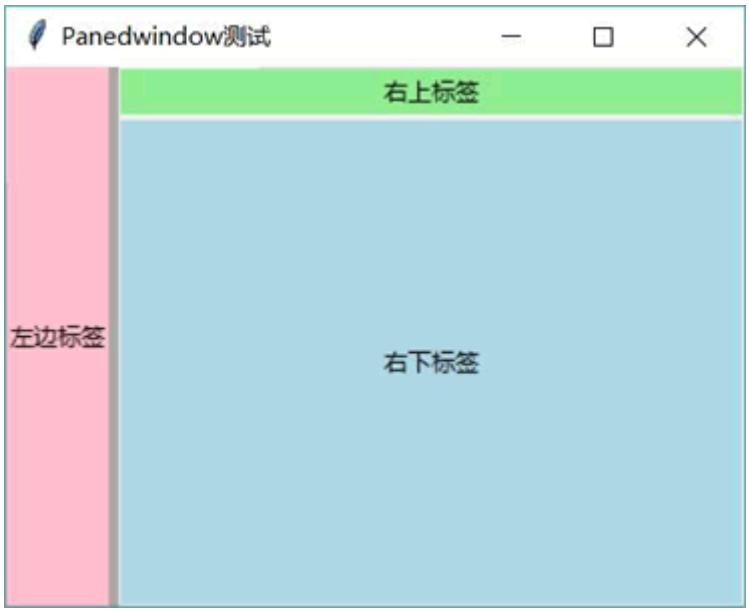


图 3 Panedwindow 嵌套

17.Python Tkinter OptionMenu

OptionMenu 组件用于构建一个带菜单的按钮，该菜单可以在按钮的四个方向上展开，展开方向可通过 direction 选项控制。

使用 OptionMenu 比较简单，直接调用它的如下构造函数即可：

```
__init__ (self, master, variable, value, *values, **kwargs)
```

其中，master 参数的作用与所有的 Tkinker 组件一样，指定将该组件放入哪个容器中。其他参数的含义如下：

- variable：指定该按钮上的菜单与哪个变量绑定。
- value：指定默认选择菜单中的哪一项。
- values：Tkinter 将收集为此参数传入的多个值，为每个值创建一个菜单项。
- kwargs：用于为 OptionMenu 配置选项。除前面介绍的常规选项之外，还可通过 direction 选项控制菜单的展开方向。

下面程序创建了一个 OptionMenu，并通过单选钮来控制 OptionMenu 中菜单的展开方向：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk
class App:
    def __init__(self, master):
        self.master = master
```

```
7.                  self.initWidgets()
8.              def initWidgets(self):
9.                  self.sv = StringVar()
10.                 # 创建一个 OptionMenu 控件
11.                 self.om = ttk.OptionMenu(root,
12.                     self.sv, # 绑定变量
13.                     'Python', # 设置初始选中值
14.                     'Kotlin', # 以下多个值用于设置菜单项
15.                     'Ruby',
16.                     'Swift',
17.                     'Java',
18.                     'Python',
19.                     'JavaScript',
20.                     'Erlang',
21.                     command = self.print_option) # 绑定事件处理方法
22.                 self.om.pack()
23.                 # 创建 Labelframe 容器
24.                 lf = ttk.Labelframe(self.master, padding=20, text='请选择菜单方
    向')
25.                 lf.pack(fill=BOTH, expand=YES, padx=10, pady=10)
26.                 # 定义代表 Labelframe 的标题位置的 12 个常量
27.                 self.directions = ['below', 'above', 'left', 'right', 'flush']
28.                 i = 0
29.                 self.intVar = IntVar()
30.                 # 使用循环创建多个 Radiobutton，并放入 Labelframe 中
31.                 for direct in self.directions:
32.                     Radiobutton(lf, text= direct,
33.                     value=i,
34.                     command=self.change,
35.                     variable=self.intVar).pack(side=LEFT)
36.                     i += 1
37.                 self.intVar.set(9)
38.             def print_option(self, val):
39.                 # 通过两种方式来获取 OptionMenu 选中的菜单项的值
40.                 print(self.sv.get(), val)
41.             def change(self):
```

```
42.                 # 通过 direction 选项改变 OptionMenu 上菜单的展开方向
43.                 self.om['direction'] = self.directions[self.intVar.get()]
44.         root = Tk()
45.         root.title("OptionMenu 测试")
46.         # 改变窗口图标
47.         root.iconbitmap('images/fklogo.ico')
48.         App(root)
49.         root.mainloop()
```

上面程序中，第 21 行代码为 OptionMenu 的 command 选项绑定了 self.print_option 方法，这意味着当用户选择菜单中的不同菜单项时，都会触发 self.print_option 方法。该事件处理方法也比较奇葩，它可以额外指定一个参数来获取目标菜单项上的值，如上面程序中第 38 行代码所示。

程序中第 43 行代码通过动态改变 OptionMenu 的 direction 选项值，就可以动态改变按钮上菜单的展开方向。

运行上面程序，选中下方的“left”单选钮，就可以看到如图 1 所示的效果。

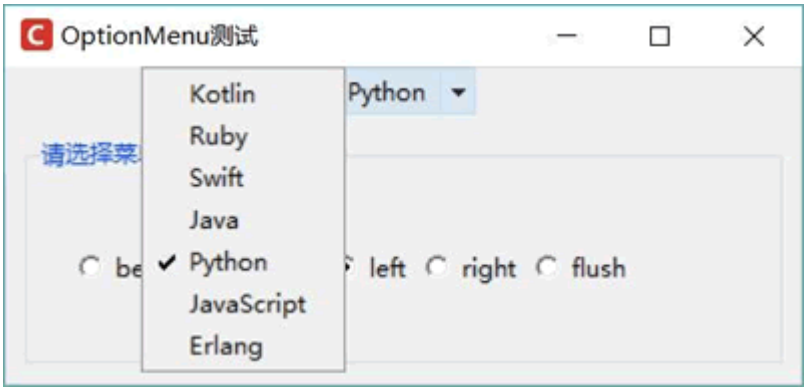


图 1 OptionMenu 组件

18.Python Tkinter 对话框

对话框也是图形界面编程中很常用的组件，通常用于向用户生成某种提示信息，或者请求用户输入某些简单的信息。

对话框看上去有点类似于顶级窗口，但对于对话框有如下两点需要注意：
对话框通常依赖其他窗口，因此程序在创建对话框时同样需要指定 master 属性（该对话框的属主窗口）。
对话框有非模式（non-modal）和模式（modal）两种，当某个模式对话框被打开之后，该模式对话框总是位于它依赖的窗口之上；在模式对话框被关闭之前，它依赖的窗口无法获得焦点。

Tkinter 在 simpledialog 和 dialog 模块下分别提供了 SimpleDialog 类和 Dialog 类，它们都可作为普通对话框使用，而且用法也差不多。

在使用 simpledialog.SimpleDialog 创建对话框时，可指定如下选项：

- title：指定该对话框的标题。
- text：指定该对话框的内容。
- button：指定该对话框下方的几个按钮。
- default：指定该对话框中默认第几个按钮得到焦点。
- cancel :指定当用户通过对话框右上角的 X 按钮关闭对话框时 ,该对话框的返回值。

如果使用 dialog.Dialog 创建对话框，除可使用 master 指定对话框的属主窗口之外，还可通过 dict 来指定如下选项：

- title：指定该对话框的标题。
- text：指定该对话框的内容。

- strings：指定该对话框下方的几个按钮。
- default：指定该对话框中默认第几个按钮得到焦点。
- bitmap：指定该对话框上的图标。

对比上面介绍不难发现，simpledialog.SimpleDialog 和 dialog.Dialog 所支持的选项大同小异，区别只是 dialog.Dialog 需要使用 dict 来传入多个选项。

如下程序分别示范了使用 SimpleDialog 和 Dialog 来创建对话框：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk
# 导入 simpledialog
from tkinter import simpledialog
# 导入 dialog
from tkinter import dialog

class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        self.msg = 'C 语言中文网成立于 2012 年初，目前已经运营了将近 5 年，我们致力于分享精品教程，帮助对编程感兴趣的读者。'
        # 创建 2 个按钮，并为之绑定事件处理函数
        ttk.Button(self.master, text='打开 SimpleDialog',
            command=self.open_simpledialog # 绑定 open_simpledialog 方法
            ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)
        ttk.Button(self.master, text='打开 Dialog',
            command=self.open_dialog # 绑定 open_dialog 方法
```

```
21.                     ).pack(side=LEFT, ipadx=5, ipady=5, padx = 10)
22.             def open_simpledialog(self):
23.                 # 使用 simpledialog.SimpleDialog 创建对话框
24.                 d = simpledialog.SimpleDialog(self.master, # 设置该对话框所属的
    窗口
25.                     title='SimpleDialog 测试', # 标题
26.                     text=self.msg,  # 内容
27.                     buttons=["是", "否", "取消"],
28.                     cancel=3,
29.                     default=0 # 设置默认是哪个按钮得到焦点
30.                 )
31.                 print(d.go())  #①
32.             def open_dialog(self):
33.                 # 使用 dialog.Dialog 创建对话框
34.                 d = dialog.Dialog(self.master # 设置该对话框所属的窗口
35.                     , {'title': 'Dialog 测试',  # 标题
36.                     'text':self.msg, # 内容
37.                     'bitmap': 'question', # 图标
38.                     'default': 0,  # 设置默认选中项
39.                     # strings 选项用于设置按钮
40.                     'strings': ('确定',
41.                         '取消',
42.                         '退出')})
43.                 print(d.num)  #②
44.
45.         root = Tk()
46.         root.title("对话框测试")
47.         App(root)
48.         root.mainloop()
```

上面程序中，第 24 行代码使用 simpledialog.SimpleDialog 创建了对话框，同时第 34 行代码使用 dialog.Dialog 创建了对话框。从代码中可以看到，创建两个对话框的代码相似，区别只是创建 dialog.Dialog 时需要使用 dict 传入选项。

运行上面程序，单击界面上的“打开 SimpleDialog”按钮，可以看到如图 1 所示的效果。

图 1 SimpleDialog 对话框

程序中 ① 号代码打印了该对话框 go() 方法的返回值，该返回值会获取用户单击了对话框的哪个按钮。如果用户通过对话框右上角的 X 按钮关闭对话框，则返回 cancel 选项指定的值。

如果单击界面上的“打开 Dialog”按钮，可以看到如图 2 所示的效果。

图 2 Dialog 对话框

程序中 ② 号代码打印了该对话框 num 属性的值，该返回值会获取用户单击了对话框的哪个按钮。

在图 2 所示对话框的左边还显示了一个问号图标，这是 Python 内置的 10 个位图之一，可以直接使用。共有如下几个常量可用于设置位图：

“error”
“gray75”
“gray50”

“gray25”
“gray12”
“howglass”
“info”
“questhead”
“question”
“warning”

19.Python Tkinter 自定义对话框

从前面章节的介绍可以看到，不管是使用 SimpleDialog 还是 Dialog，整个对话框的布局都是比较固定的，开发者只能为其指定 title、text 等选项，如果希望在对话框中添加其他组件，这就很难实现了。另外，SimpleDialog 和 Dialog 都是模式的。

如果开发者需要使用自定义的对话框，包括定制模式和非模式行为，则可通过继承 Toplevel 来实现。如果打算通过这种方式来实现自定义对话框，有两点要注意：

1. 继承 Toplevel 来实现自定义对话框同样需要为对话框指定 master。
2. 程序可调用 Toplevel 的 grab_set() 方法让该对话框变成模式对话框，否则就是非模式对话框。

下面程序示范了自定义模式和非模式对话框：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk
from tkinter import messagebox

# 自定义对话框类，继承 Toplevel
```

```
7.          class MyDialog(Toplevel):
8.              # 定义构造方法
9.              def __init__(self, parent, title = None, modal=True):
10.                 Toplevel.__init__(self, parent)
11.                 self.transient(parent)
12.                 # 设置标题
13.                 if title: self.title(title)
14.                 self.parent = parent
15.                 self.result = None
16.                 # 创建对话框的主体内容
17.                 frame = Frame(self)
18.                 # 调用 init_widgets 方法来初始化对话框界面
19.                 self.initial_focus = self.init_widgets(frame)
20.                 frame.pack(padx=5, pady=5)
21.                 # 调用 init_buttons 方法初始化对话框下方的按钮
22.                 self.init_buttons()
23.                 # 根据 modal 选项设置是否为模式对话框
24.                 if modal: self.grab_set()
25.                 if not self.initial_focus:
26.                     self.initial_focus = self
27.                 # 为"WM_DELETE_WINDOW"协议使用 self.cancel_click 事件处理方法
28.                 self.protocol("WM_DELETE_WINDOW", self.cancel_click)
29.                 # 根据父窗口来设置对话框的位置
30.                 self.geometry("+%d+%d" % (parent.winfo_rootx()+50,
31.                     parent.winfo_rooty()+50))
32.                 print( self.initial_focus)
33.                 # 让对话框获取焦点
34.                 self.initial_focus.focus_set()
35.                 self.wait_window(self)
36.             # 通过该方法来创建自定义对话框的内容
37.             def init_widgets(self, master):
38.                 # 创建并添加 Label
39.                 Label(master, text='用户名', font=12,width=10).grid(row=1,
    column=0)
40.                 # 创建并添加 Entry,用于接受用户输入的用户名
41.                 self.name_entry = Entry(master, font=16)
```

```
42.                 self.name_entry.grid(row=1, column=1)
43.                 # 创建并添加 Label
44.                 Label(master, text='密  码', font=12,width=10).grid(row=2,
    column=0)
45.                 # 创建并添加 Entry,用于接受用户输入的密码
46.                 self.pass_entry = Entry(master, font=16)
47.                 self.pass_entry.grid(row=2, column=1)
48.             # 通过该方法来创建对话框下方的按钮框
49.             def init_buttons(self):
50.                 f = Frame(self)
51.                 # 创建"确定"按钮,位置绑定 self.ok_click 处理方法
52.                 w = Button(f, text="确定", width=10, command=self.ok_click,
    default=ACTIVE)
53.                 w.pack(side=LEFT, padx=5, pady=5)
54.                 # 创建"确定"按钮,位置绑定 self.cancel_click 处理方法
55.                 w = Button(f, text="取消", width=10, command=self.cancel_click)
56.                 w.pack(side=LEFT, padx=5, pady=5)
57.                 self.bind("<Return>", self.ok_click)
58.                 self.bind("<Escape>", self.cancel_click)
59.                 f.pack()
60.             # 该方法可对用户输入的数据进行校验
61.             def validate(self):
62.                 # 可重写该方法
63.                 return True
64.             # 该方法可处理用户输入的数据
65.             def process_input(self):
66.                 user_name = self.name_entry.get()
67.                 user_pass = self.pass_entry.get()
68.                 messagebox.showinfo(message='用户输入的用户名: %s, 密码: %s'
69.                     % (user_name , user_pass))
70.             def ok_click(self, event=None):
71.                 print('确定')
72.                 # 如果不能通过校验，让用户重新输入
73.                 if not self.validate():
74.                     self.initial_focus.focus_set()
75.                     return
```

```
76.                 self.withdraw()
77.                 self.update_idletasks()
78.                 # 获取用户输入数据
79.                 self.process_input()
80.                 # 将焦点返回给父窗口
81.                 self.parent.focus_set()
82.                 # 销毁自己
83.                 self.destroy()
84.             def cancel_click(self, event=None):
85.                 print('取消')
86.                 # 将焦点返回给父窗口
87.                 self.parent.focus_set()
88.                 # 销毁自己
89.                 self.destroy()
90.         class App:
91.             def __init__(self, master):
92.                 self.master = master
93.                 self.initWidgets()
94.             def initWidgets(self):
95.                 # 创建 2 个按钮，并为之绑定事件处理函数
96.                 ttk.Button(self.master, text='模式对话框',
97.                     command=self.open_modal # 绑定 open_modal 方法
98.                     ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)
99.                 ttk.Button(self.master, text='非模式对话框',
100.                     command=self.open_none_modal # 绑定 open_none_modal 方法
101.                     ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)
102.             def open_modal(self):
103.                 d = MyDialog(self.master, title='模式对话框') # 默认是模式对话
    框
104.             def open_none_modal(self):
105.                 d = MyDialog(self.master, title='非模式对话框', modal=False)
106.         root = Tk()
107.         root.title("颜色对话框测试")
108.         App(root)
109.         root.mainloop()
```

上面程序定义了一个父类为 Toplevel 的 MyDialog 类，该类就是一个自定义对话框类，读者以后完全可以复用这个 MyDialog 类。该对话框的主体包含两个方法：

- init_widgets：该方法用于初始化对话框的主体界面组件。
- init_buttons：该方法用于初始化对话框下方的多个按钮组件。

至于 MyDialog 类中的 ok_click、cancel_click，则是程序为对话框按钮所绑定的事件处理方法。这不是固定的，程序下方有几个按钮，通常就需要为几个按钮绑定事件处理方法。

上面程序通过 MyDialog 创建了模式对话框和非模式对话框。运行该程序，单击界面上的“模式对话框”按钮，程序弹出如图 1 所示的自定义对话框。

图 1 模式对话框

在该模式对话框没有关闭的情况下，该程序的主窗口将无法与用户交互，主窗口无法获得焦点。

如果单击界面上的“非模式对话框”按钮，程序也弹出类似于图 1 所示的自定义对活框，

但这个对话框是非模式对话框，即使在该对话框没有关闭的情况下，用户也依然可以与程序的主窗口交互，主窗口可以获得焦点。

20.Python Tkinter 输入对话框

在 simpledialog 模块下还有如下便捷的工具函数，通过这些工具函数可以更方便地生成各种输入对话框：

- askinteger：生成一个让用户输入整数的对话框。
- askfloat：生成一个让用户输入浮点数的对话框。
- askstring：生成一个让用户输入字符串的对话框。

上面三个工具函数的前两个参数分别指定对话框的标题和提示信息，后面还可以通过选项来设置对话框的初始值、最大值和最小值。

下面程序示范了 simpledialog 模块下三个工具函数的用法：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk
# 导入 simpledialog
from tkinter import simpledialog
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        # 创建 3 个按钮，并为之绑定事件处理函数
```

```
12.                 ttk.Button(self.master, text='输入整数对话框',
13.                     command=self.open_integer # 绑定 open_integer 方法
14.                     ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)
15.                 ttk.Button(self.master, text='输入浮点数对话框',
16.                     command=self.open_float # 绑定 open_integer 方法
17.                     ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)
18.                 ttk.Button(self.master, text='输入字符串对话框',
19.                     command=self.open_string # 绑定 open_integer 方法
20.                     ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)
21.             def open_integer(self):
22.                  # 调用 askinteger 函数生成一个让用户输入整数的对话框
23.                 print(simpledialog.askinteger("猜糖果", "你猜我手上有几个糖
    果:",
24.                     initialvalue=3, minvalue=1, maxvalue=10))
25.             def open_float(self):
26.                 # 调用 askfloat 函数生成一个让用户输入浮点数的对话框
27.                 print(simpledialog.askfloat("猜体重", "你猜我我体重多少公斤:",
28.                     initialvalue=27.3, minvalue=10, maxvalue=50))
29.             def open_string(self):
30.                 # 调用 askstring 函数生成一个让用户输入字符串的对话框
31.                 print(simpledialog.askstring("猜名字", "你猜我叫什么名字:",
32.                     initialvalue='Charlie'))
33.         root = Tk()
34.         root.title("输入对话框测试")
35.         App(root)
36.         root.mainloop()
```

上面程序中第 23 行代码生成让用户输入整数的对话框；第 27 行代码生成让用户输入浮点数的对话框；第 31 行代码生成让用户输入字符串的对话框。

askinteger()、askfloat 和 askstring 这三个函数会返回用户输入的数据，因此上面三行代码打印了这三个函数的返回值，这样就可以打印出用户输入的内容。

运行该程序，单击界面上的“输入整数对话框”按钮，可以看到如图 1 所示的对话框。

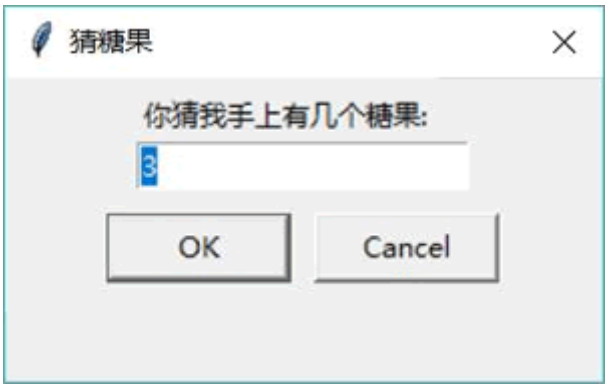


图 1 输入整数的对话框

在图 1 所示的对话框中，用户只能输入整数，而且输入的整数必须在指定范围内；否则，系统会生成错误提示。当用户输入所允许范围内的整数并单击“OK”按钮后，可以看到控制台打印了用户输入的整数。

单击界面上的“输入浮点数对话框” 按钮，可以看到如图 2 所示的对话框。

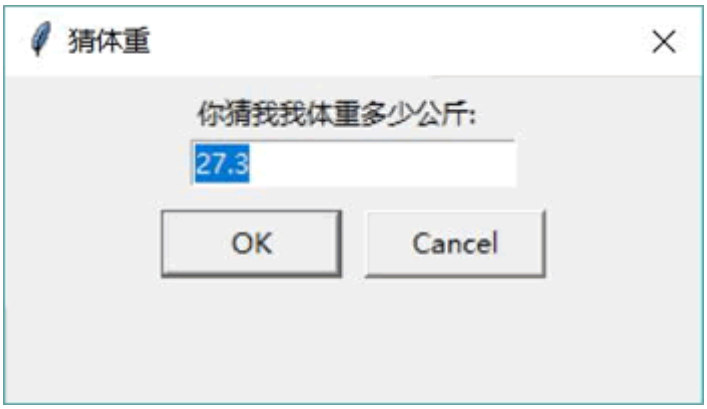


图 2 输入浮点数的对话框

在图 2 所示的对话框中，用户只能输入浮点数，而且输入的浮点数必须在指定范围内； 否则，系统会生成错误提示。当用户输入所允许范围内的浮点数并单击“OK”按钮后，可以看到控制台打印了用户输入的浮点数。

单击界面上的“输入字符串对话框”按钮，可以看到如图 3 所示的对话框。

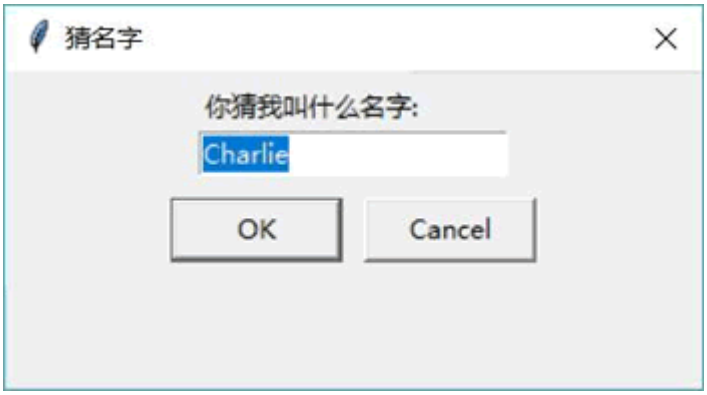


图 3 输入字符串的对话框

在图 3 所示的对话框中，用户只能输入字符串。当用户输入合适的字符串并单击“OK”按钮后，可以看到控制台打印了用户输入的字符串

21.Python Tkinter 文件对话框

在 filedialog 模块下提供了各种用于生成文件对话框的工具函数，如下所示。这些工具函数有些返回用户所选择文件的路径，有些直接返回用户所选择文件的输入/输出流：

askopenfile()：生成打开单个文件的对话框，返回所选择文件的文件流，程序可通过该文件流来读取文件内容。
askopenfiles()：生成打开多个文件的对话框，返回多个所选择文件的文件流组成的列表，程序可通过这些文件流来读取文件内容。
askopenfilename()：生成打开单个文件的对话框，返回所选择文件的文件路径。

askopenfilenames()：生成打开多个文件的对话框，返回多个所选择文件的文件路径组成的元组。
asksaveasfile()：生成保存文件的对话框，返回所选择文件的文件输出流，程序可通过该文件输出流向文件写入数据。
asksaveasfilename()：生成保存文件的对话框，返回所选择文件的文件路径。
askdirectory()：生成打开目录的对话框。

上面的用于生成打开文件的对话框的工具函数支持如下选项：
defaultextension：指定默认扩展名。当用户没有输入扩展名时，系统会默认添加该选项指定的扩展名。
filetypes：指定在该文件对话框中能查看的文件类型。该选项值是一个序列，可指定多个文件类型。可以通过“*”指定浏览所有文件。
initialdir：指定初始打开的目录。
initialfile：指定所选择的文件。
parent：指定该对话框的属主窗口。
title：指定对话框的标题。
multiple：指定是否允许多选。

对于打开目录的对话框，还额外支持一个 mustexist 选项，该选项指定是否只允许打开已存在的目录。

下面程序示范了文件对话框的各工具函数的用法：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk
# 导入 filedialog
from tkinter import filedialog
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        # 创建 7 个按钮，并为之绑定事件处理函数
        ttk.Button(self.master, text='打开单个文件',
            command=self.open_file # 绑定 open_file 方法
            ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)
        ttk.Button(self.master, text='打开多个文件',
            command=self.open_files # 绑定 open_files 方法
            ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)
```

```
18.                 ttk.Button(self.master, text='获取单个打开文件的文件名',
19.                     command=self.open_filename # 绑定 open_filename 方法
20.                     ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)
21.                 ttk.Button(self.master, text='获取多个打开文件的文件名',
22.                     command=self.open_filenames # 绑定 open_filenames 方法
23.                     ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)
24.                 ttk.Button(self.master, text='获取保存文件',
25.                     command=self.save_file # 绑定 save_file 方法
26.                     ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)
27.                 ttk.Button(self.master, text='获取保存文件的文件名',
28.                     command=self.save_filename # 绑定 save_filename 方法
29.                     ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)
30.                 ttk.Button(self.master, text='打开路径',
31.                     command=self.open_dir # 绑定 open_dir 方法
32.                     ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)
33.             def open_file(self):
34.                 # 调用 askopenfile 方法获取单个打开的文件
35.                 print(filedialog.askopenfile(title='打开单个文件',
36.                     filetypes=[("文本文件", "*.txt"), ('Python 源文件', '*.py')],
    # 只处理的文件类型
37.                     initialdir='g:/')) # 初始目录
38.             def open_files(self):
39.                 # 调用 askopenfile 方法获取多个打开的文件
40.                 print(filedialog.askopenfiles(title='打开多个文件',
41.                     filetypes=[("文本文件", "*.txt"), ('Python 源文件', '*.py')],
    # 只处理的文件类型
42.                     initialdir='g:/')) # 初始目录
43.             def open_filename(self):
44.                 # 调用 askopenfilename 方法获取单个文件的文件名
45.                 print(filedialog.askopenfilename(title='打开单个文件',
46.                     filetypes=[("文本文件", "*.txt"), ('Python 源文件', '*.py')],
    # 只处理的文件类型
47.                     initialdir='g:/')) # 初始目录
48.             def open_filenames(self):
49.                 # 调用 askopenfilenames 方法获取多个文件的文件名
50.                 print(filedialog.askopenfilenames(title='打开多个文件',
```

```
                    filetypes=[("文本文件", "*.txt"), ('Python 源文件', '*.py')],
# 只处理的文件类型
                    initialdir='g:/')) # 初始目录
            def save_file(self):
                # 调用 asksaveasfile 方法保存文件
                print(filedialog.asksaveasfile(title='保存文件',
                    filetypes=[("文本文件", "*.txt"), ('Python 源文件', '*.py')],
# 只处理的文件类型
                    initialdir='g:/')) # 初始目录
            def save_filename(self):
                # 调用 asksaveasfilename 方法获取保存文件的文件名
                print(filedialog.asksaveasfilename(title='保存文件',
                    filetypes=[("文本文件", "*.txt"), ('Python 源文件', '*.py')],
# 只处理的文件类型
                    initialdir='g:/')) # 初始目录
            def open_dir(self):
                # 调用 askdirectory 方法打开目录
                print(filedialog.askdirectory(title='打开目录',
                    initialdir='g:/')) # 初始目录
        root = Tk()
        root.title("文件对话框测试")
        App(root)
        root.mainloop()
```

上面程序就是 filedialog 模块下不同函数的示范代码。运行上面程序，单击界面上的“打开单个文件”按钮，将会看到如图 1 所示的文件对话框：

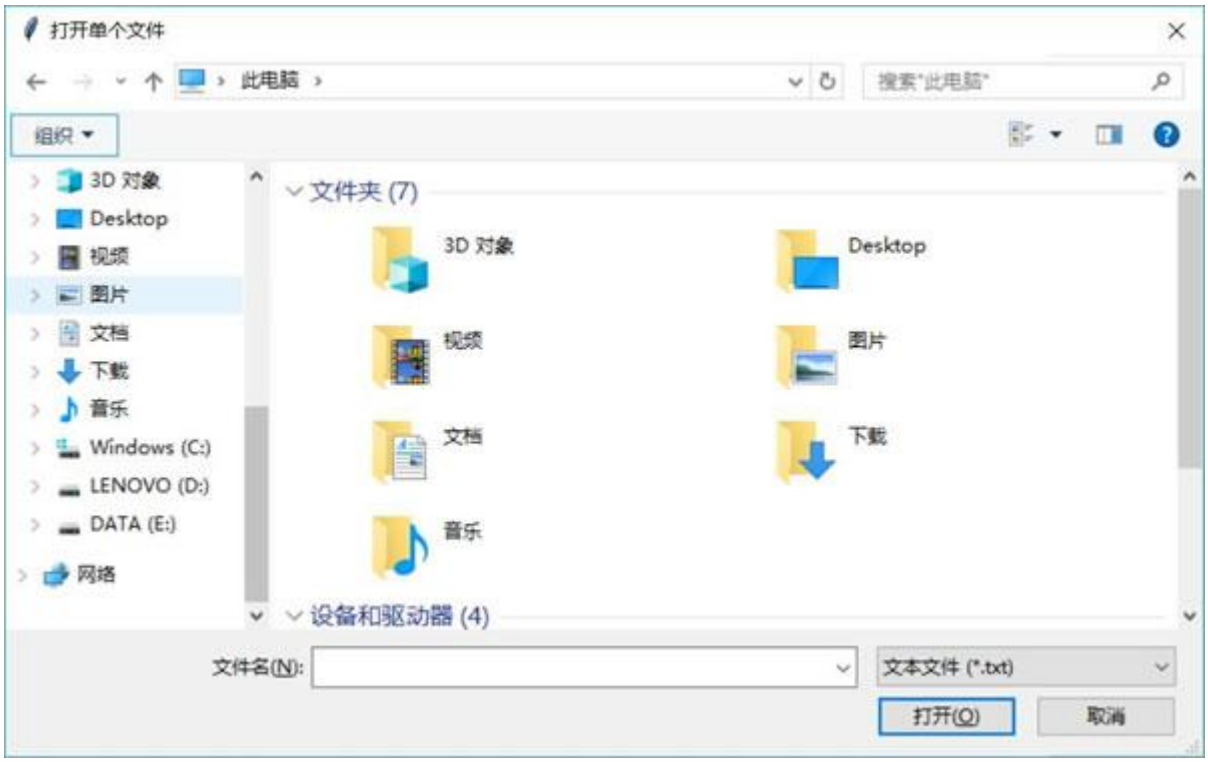


图 1 打开单个文件的对话框

从图 1 所示的对话框可以看到，在该对话框中只能浏览文本文件和 Python 源文件，这正是程序通过 filetypes 选项设置的效果；该对话框默认的打开目录是 G:/ 根目录，这是程序通过 initialdir 选项设置的效果。

通过 filedialog 模块下的工具函数打开的文件对话框依赖所在的平台，因此在不同的平台上看到的文件对话框是不同的。

当用户选择指定文件后，可以在控制台看到打印出类似于“<_io.TextIOWrapper name='xxx' mode='r' encoding='cp936'>”的输出信息，这就是被打开文件的文件流；如果用户单击“xxx 的文件名”按钮，通过文件对话框选择文件之后，将会在控制台看到只打印出所选择文件的文件路径。

22.Python Tkinter askcolor 颜色选择对话框用法

在 colorchooser 模块下提供了用于生成颜色选择对话框的 askcolor() 工具函数，为该工

具函数可指定如下选项：

- parent：指定该对话框的属主窗口。
- title：指定该对话框的标题。
- color：指定该对话框初始选择的颜色。

下面程序示范了颜色选择对话框的功能和用法：

```
1.          from tkinter import *
2.          # 导入 ttk
3.          from tkinter import ttk
4.          # 导入 colorchooser
5.          from tkinter import colorchooser
6.          class App:
7.              def __init__(self, master):
8.                  self.master = master
9.                  self.initWidgets()
10.             def initWidgets(self):
11.                 # 创建 1 个按钮，并为之绑定事件处理函数
12.                 ttk.Button(self.master, text='选择颜色',
13.                     command=self.choose_color # 绑定 choose_color 方法
14.                     ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)
15.             def choose_color(self):
16.                 # 调用 askcolor 函数获取选中的颜色
17.                 print(colorchooser.askcolor(parent=self.master, title='选择画
    笔颜色',
18.                     color = 'blue')) # 初始颜色
19.         root = Tk()
20.         root.title("颜色对话框测试")
21.         App(root)
22.         root.mainloop()
```

上面程序中第 18 行代码就是调用 askcolor() 函数生成颜色选择对话框的关键代码。运行该程序，单击界面上的“选择颜色”按钮，将可以看到如图 1 所示的对话框。

图 1 颜色选择对话框

通过 colorchooser 模块下的工具函数打开的颜色选择对话框依赖所在的平台，因此在不同的平台上看到的颜色选择对话框是不同的。

当用户选择指定颜色，并单击颜色选择对话框中的“确定”按钮后，askcolor() 函数会返回用户所选择的颜色，因此可以在控制台看到用户所选择的颜色。

23.Python Tkinter 消息框

在 messagebox 模块下提供了大量工具函数来生成各种消息框 ,这些消息框的结构大致如图 1 所示。

图 1 消息框的结构

在默认情况下，开发者在调用 messagebox 的工具函数时只要设置提示区的字符串即可，图标区的图标、按钮区的按钮都有默认设置。

如果有必要，则完全可通过如下两个选项来定制图标和按钮：

- icon :定制图标的选 I 页。该选项支持“error”、“info”、“question”、“warning”这几个选项值。
- type：定制按钮的选项。该选项支持“abortretryignore”（取消、重试、忽略）、“ok”（确定）、“okcancel”（确定、取消）、“retrycancel”（重试、取消）、“yesno”（是、否）、“yesnocancel”（是、否、取消）这些选项值。

下面的示例程序不仅示范了 messagebox 的各工具函数的用法，而且还通过两组单选钮让用户动态选择不同的 icon 和 type 选项的效果。

```
1.          from tkinter import *
2.          # 导入 ttk
```

```
3.          from tkinter import ttk
4.          # 导入 messagebox
5.          from tkinter import messagebox as msgbox
6.          class App:
7.              def __init__(self, master):
8.                  self.master = master
9.                  self.initWidgets()
10.             def initWidgets(self):
11.                 #-----------创建第 1 个 Labelframe，用于选择图标类型-----------
12.                 topF = Frame(self.master)
13.                 topF.pack(fill=BOTH)
14.                 lf1 = ttk.Labelframe(topF, text='请选择图标类型')
15.                 lf1.pack(side=LEFT, fill=BOTH, expand=YES, padx=10, pady=5)
16.                 i = 0
17.                 self.iconVar = IntVar()
18.                 self.icons = [None, "error", "info", "question", "warning"]
19.                 # 使用循环创建多个 Radiobutton，并放入 Labelframe 中
20.                 for icon in self.icons:
21.                     Radiobutton(lf1, text = icon if icon is not None else '默认
    ',
22.                     value=i,
23.                     variable=self.iconVar).pack(side=TOP, anchor=W)
24.                     i += 1
25.                 self.iconVar.set(0)
26.                 #-----------创建第二个 Labelframe，用于选择按钮类型-----------
27.                 lf2 = ttk.Labelframe(topF, text='请选择按钮类型')
28.                 lf2.pack(side=LEFT,fill=BOTH, expand=YES, padx=10, pady=5)
29.                 i = 0
30.                 self.typeVar = IntVar()
31.                 # 定义所有按钮类型
32.                 self.types = [None, "abortretryignore", "ok", "okcancel",
33.                     "retrycancel", "yesno", "yesnocancel"]
34.                 # 使用循环创建多个 Radiobutton，并放入 Labelframe 中
35.                 for tp in self.types:
36.                     Radiobutton(lf2, text= tp if tp is not None else '默认',
37.                     value=i,
```

```
38.                     variable=self.typeVar).pack(side=TOP, anchor=W)
39.                     i += 1
40.                 self.typeVar.set(0)
41.                 #-----------创建 Frame,用于包含多个按钮来生成不同的消息框
    -----------
42.                 bottomF = Frame(self.master)
43.                 bottomF.pack(fill=BOTH)
44.                 # 创建 8 个按钮，并为之绑定事件处理函数
45.                 btn1 = ttk.Button(bottomF, text="showinfo",
46.                     command=self.showinfo_clicked)
47.                 btn1.pack(side=LEFT, fill=X, ipadx=5, ipady=5,
48.                     pady=5, padx=5)
49.                 btn2 = ttk.Button(bottomF, text="showwarning",
50.                     command=self.showwarning_clicked)
51.                 btn2.pack(side=LEFT, fill=X, ipadx=5, ipady=5,
52.                     pady=5, padx=5)
53.                 btn3 = ttk.Button(bottomF, text="showerror",
54.                     command=self.showerror_clicked)
55.                 btn3.pack(side=LEFT, fill=X, ipadx=5, ipady=5,
56.                     pady=5, padx=5)
57.                 btn4 = ttk.Button(bottomF, text="askquestion",
58.                     command=self.askquestion_clicked)
59.                 btn4.pack(side=LEFT, fill=X, ipadx=5, ipady=5,
60.                     pady=5, padx=5)
61.                 btn5 = ttk.Button(bottomF, text="askokcancel",
62.                     command=self.askokcancel_clicked)
63.                 btn5.pack(side=LEFT, fill=X, ipadx=5, ipady=5,
64.                     pady=5, padx=5)
65.                 btn6 = ttk.Button(bottomF, text="askyesno",
66.                     command=self.askyesno_clicked)
67.                 btn6.pack(side=LEFT, fill=X, ipadx=5, ipady=5,
68.                     pady=5, padx=5)
69.                 btn7 = ttk.Button(bottomF, text="askyesnocancel",
70.                     command=self.askyesnocancel_clicked)
71.                 btn7.pack(side=LEFT, fill=X, ipadx=5, ipady=5,
72.                     pady=5, padx=5)
```

```
73.                 btn8 = ttk.Button(bottomF, text="askretrycancel",
74.                     command=self.askretrycancel_clicked)
75.                 btn8.pack(side=LEFT, fill=X, ipadx=5, ipady=5,
76.                     pady=5, padx=5)
77.             def showinfo_clicked(self):
78.                 print(msgbox.showinfo("Info", "showinfo 测试.",
79.                     icon=self.icons[self.iconVar.get()],
80.                     type=self.types[self.typeVar.get()]))
81.             def showwarning_clicked(self):
82.                 print(msgbox.showwarning("Warning", "showwarning 测试.",
83.                     icon=self.icons[self.iconVar.get()],
84.                     type=self.types[self.typeVar.get()]))
85.             def showerror_clicked(self):
86.                 print(msgbox.showerror("Error", "showerror 测试.",
87.                     icon=self.icons[self.iconVar.get()],
88.                     type=self.types[self.typeVar.get()]))
89.             def askquestion_clicked(self):
90.                 print(msgbox.askquestion("Question", "askquestion 测试.",
91.                     icon=self.icons[self.iconVar.get()],
92.                     type=self.types[self.typeVar.get()]))
93.             def askokcancel_clicked(self):
94.                 print(msgbox.askokcancel("OkCancel", "askokcancel 测试.",
95.                     icon=self.icons[self.iconVar.get()],
96.                     type=self.types[self.typeVar.get()]))
97.             def askyesno_clicked(self):
98.                 print(msgbox.askyesno("YesNo", "askyesno 测试.",
99.                     icon=self.icons[self.iconVar.get()],
100.                    type=self.types[self.typeVar.get()]))
101.            def askyesnocancel_clicked(self):
102.                print(msgbox.askyesnocancel("YesNoCancel", "askyesnocancel 测
    试.",
103.                    icon=self.icons[self.iconVar.get()],
104.                    type=self.types[self.typeVar.get()]))
105.            def askretrycancel_clicked(self):
106.                print(msgbox.askretrycancel("RetryCancel", "askretrycancel 测
    试.",
```

```
                    icon=self.icons[self.iconVar.get()],
                    type=self.types[self.typeVar.get()]))
        root = Tk()
        root.title("消息框测试")
        App(root)
        root.mainloop()
```

上面程序先创建了两组单选钮来让用户选择图标类型（通过 icon 选项改变）和按钮类型（通过 type 选项改变）。接下来的几行代码就是调用函数生成不同消息框的关键代码。

运行上面程序，可以看到如图 2 所示的界面。

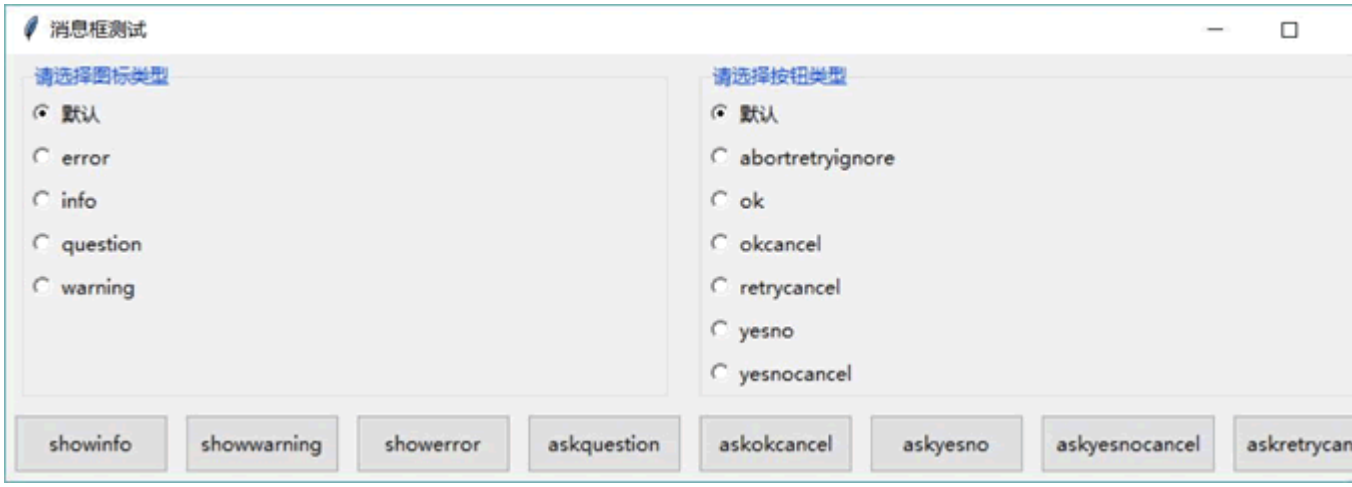


图 2 生成消息框的程序界面

读者可通过左边的单选钮选择图标类型，通过右边的单选钮选择按钮类型。比如在左边选择“error”，在右边选择“abortretryignore”，然后单击“showinfo”按钮，将可以看到如图 3 所示的消息框。

图 3 定制的消息框

showinfo() 函数默认生成的消息框的图标应该是一个感叹号，下方也只有一个按钮（读者可通过两组单边钮都选择“默认”来看默认效果）：但从图 3 中看到通过 showinfo() 函数生成的消息框被改变了，这就是因为指定了 icon 和 type 选项的缘故。

上面程序打印出消息框返回的结果，这些消息框到底返回什么呢？。消息框返回的是用户单击的按钮，比如用户单击“中止”按钮，消息框就返回 'abort' 字符串；用户单击“重试”按钮，消息框就返回 'retry' 字符串……

24.Python Tkinter Menu 菜单

Tkinter 为菜单提供了 Menu 类，该类既可代表菜单条，也可代表菜单，还可代表上下文菜单（右键菜单）。简单来说，Menu 类就可以搞定所有菜单相关内容。

程序可调用 Menu 的构造方法来创建菜单，在创建菜单之后可通过如下方法添加菜单项：

- add_command()：添加菜单项。
- add_checkbutton()：添加复选框菜单项。

- add_radiobutton()：添加单选钮菜单项。
- add_separator()：添加菜单分隔条。

上面的前三个方法都用于添加菜单项，因此都支持如下常用选项：

- label：指定菜单项的文本。
- command：指定为菜单项绑定的事件处理方法。
- image：指定菜单项的图标。
- compound：指定在菜单项中图标位于文字的哪个方位。

有了菜单之后，接下来就是如何使用菜单了。菜单有两种用法：

1. 在窗口上方通过菜单条管理菜单。
2. 通过鼠标右键触发右键菜单（上下文菜单）。

Menu 窗口菜单

在创建菜单之后，如果要将菜单设置为窗口的菜单条（Menu 对象可被当成菜单条使用），则只要将应菜单设为窗口的 menu 选项即可。例如如下代码：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk
from tkinter import messagebox as msgbox

class App:
```

```
7.              def __init__(self, master):
8.                  self.master = master
9.                  self.init_menu()
10.             # 创建菜单
11.             def init_menu(self):
12.                 # 创建 menubar，它被放入 self.master 中
13.                 menubar = Menu(self.master)
14.                 self.master.filenew_icon =
    PhotoImage(file='images/filenew.png')
15.                 self.master.fileopen_icon =
    PhotoImage(file='images/fileopen.png')
16.                 # 添加菜单条
17.                 self.master['menu'] = menubar
18.                 # 创建 file_menu 菜单，它被放入 menubar 中
19.                 file_menu = Menu(menubar, tearoff=0)
20.                 # 使用 add_cascade 方法添加 file_menu 菜单
21.                 menubar.add_cascade(label='文件', menu=file_menu)
22.                 # 创建 lang_menu 菜单，它被放入 menubar 中
23.                 lang_menu = Menu(menubar, tearoff=0)
24.                 # 使用 add_cascade 方法添加 lang_menu 菜单
25.                 menubar.add_cascade(label='选择语言', menu=lang_menu)
26.                 # 使用 add_command 方法为 file_menu 添加菜单项
27.                 file_menu.add_command(label="新建", command = None,
28.                     image=self.master.filenew_icon, compound=LEFT)
29.                 file_menu.add_command(label="打开", command = None,
30.                     image=self.master.fileopen_icon, compound=LEFT)
31.                 # 使用 add_command 方法为 file_menu 添加分隔条
32.                 file_menu.add_separator()
33.                 # 为 file_menu 创建子菜单
34.                 sub_menu = Menu(file_menu, tearoff=0)
35.                 # 使用 add_cascade 方法添加 sub_menu 子菜单
36.                 file_menu.add_cascade(label='选择性别', menu=sub_menu)
37.                 self.genderVar = IntVar()
38.                 # 使用循环为 sub_menu 子菜单添加菜单项
39.                 for i, im in enumerate(['男', '女', '保密']):
40.                     # 使用 add_radiobutton 方法为 sub_menu 子菜单添加单选菜单项
```

```
41.                         # 绑定同一个变量，说明它们是一组
42.                         sub_menu.add_radiobutton(label=im,
    command=self.choose_gender,
43.                             variable=self.genderVar, value=i)
44.                 self.langVars = [StringVar(), StringVar(), StringVar(),
    StringVar()]
45.                 # 使用循环为 lang_menu 菜单添加菜单项
46.                 for i, im in enumerate(('Python', 'Kotlin','Swift', 'Java')):
47.                     # 使用 add_add_checkbutton 方法为 lang_menu 菜单添加多选菜单
    项
48.                         lang_menu.add_checkbutton(label=im,
    command=self.choose_lang,
49.                             onvalue=im, variable=self.langVars[i])
50.             def choose_gender(self):
51.                 msgbox.showinfo(message='选择的性别为: %s' %
    self.genderVar.get())
52.             def choose_lang(self):
53.                 rt_list = [e.get() for e in self.langVars]
54.                 msgbox.showinfo(message='选择的语言为: %s' %
    ','.join(rt_list))
55.         root = Tk()
56.         root.title("菜单测试")
57.         root.geometry('400x200')
58.         # 禁止改变窗口大小
59.         root.resizable(width=False, height=False)
60.         App(root)
61.         root.mainloop()
```

上面程序中第 17 行代码将 Menu 设置为窗口的 menu 选项，这意味着该菜单变成了菜单条；第 21 行、25 行代码调用 add_cascade() 方法添加菜单，这意味着为菜单条添加了两个菜单。

接下来程序调用 add_command 方法为自 file_menu 添加多个菜单项，直到第 36 行代

码调用 file_menu 的 add_cascade() 方法再次为 file_menu 添加子菜单。

第 42 行代码位于循环中，这样程序调用 add_radiobutton() 方法添加多个单选菜单项，这些单选菜单项都绑定了一个变量，因此它们就是一组的：第 48 行代码位于循环中，这样程序调用 add_checkbutton() 方法添加多个多边菜单项，每个多选菜单项都有单独的值，因此它们都需要绑定一个变量。

运行上面程序，可以看到如图 1 所示的效果。

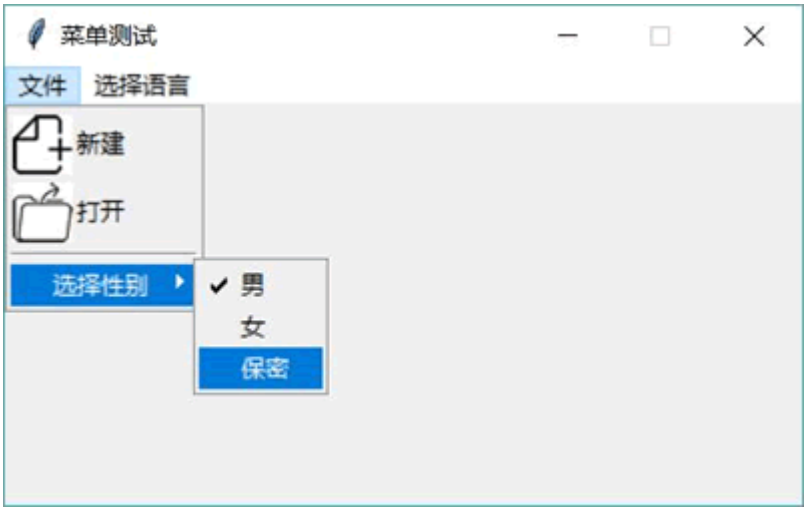


图 1 生成菜单

由于程序为单选菜单项、多选菜单项都绑定了事件处理方法，因此单击这些菜单项，程序将会弹出消息框提示用户的选择。

下面将会实现一个功能更全面的菜单示例，而且该示例程序还会添加一个工具条（实际上 Tkinter 并未提供工具条组件，因此本教程将以 Frame 来实现工具条，以 Button 实现工

具条上的按钮）。

```
from tkinter import *
# 导入 ttk
from tkinter import ttk
from collections import OrderedDict

class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        # 初始化菜单、工具条用到的图标
        self.init_icons()
        # 调用 init_menu 初始化菜单
        self.init_menu()
        # 调用 init_toolbar 初始化工具条
        self.init_toolbar()
        #--------------------------------
        # 创建、添加左边的 Frame 容器
        leftframe = ttk.Frame(self.master, width=40)
        leftframe.pack(side=LEFT, fill=Y)
        # 在左边窗口放一个 Listbox
        lb = Listbox(leftframe, font=('Courier New', 20))
        lb.pack(fill=Y, expand=YES)
        for s in ('Python', 'Ruby', 'Swift', 'Kotlin', 'Java'):
            lb.insert(END, s)
        # 创建、添加右边的 Frame 容器
        mainframe = ttk.Frame(self.master)
        mainframe.pack(side=LEFT, fill=BOTH)
        text = Text(mainframe, width=40, font=('Courier New', 16))
        text.pack(side=LEFT, fill=BOTH)
        scroll = ttk.Scrollbar(mainframe)
        scroll.pack(side=LEFT,fill=Y)
        # 设置滚动条与 text 组件关联
```

```
34.                 scroll['command'] = text.yview
35.                 text.configure(yscrollcommand=scroll.set)
36.             # 创建菜单
37.             def init_menu(self):
38.                 '初始化菜单的方法'
39.                 # 定义菜单条所包含的 3 个菜单
40.                 menus = ('文件', '编辑', '帮助')
41.                 # 定义菜单数据
42.                 items = (OrderedDict([
43.                         # 每项对应一个菜单项，后面元组第一个元素是菜单图标，
44.                         # 第二个元素是菜单对应的事件处理函数
45.                         ('新建', (self.master.filenew_icon, None)),
46.                         ('打开', (self.master.fileopen_icon, None)),
47.                         ('保存', (self.master.save_icon, None)),
48.                         ('另存为...', (self.master.saveas_icon, None)),
49.                         ('-1', (None, None)),
50.                         ('退出', (self.master.signout_icon, None)),
51.                         ]),
52.                     OrderedDict([('撤销',(None, None)),
53.                         ('重做',(None, None)),
54.                         ('-1',(None, None)),
55.                         ('剪切',(None, None)),
56.                         ('复制',(None, None)),
57.                         ('粘贴',(None, None)),
58.                         ('删除',(None, None)),
59.                         ('选择',(None, None)),
60.                         ('-2',(None, None)),
61.                         # 二级菜单
62.                         ('更多', OrderedDict([
63.                             ('显示数据',(None, None)),
64.                             ('显示统计',(None, None)),
65.                             ('显示图表',(None, None))
66.                             ]))
67.                         ]),
68.                     OrderedDict([('帮助主题',(None, None)),
69.                         ('-1',(None, None)),
```

```
70.                         ('关于', (None, None))]))
71.                 # 使用 Menu 创建菜单条
72.                 menubar = Menu(self.master)
73.                 # 为窗口配置菜单条，也就是添加菜单条
74.                 self.master['menu'] = menubar
75.                 # 遍历 menus 元组
76.                 for i, m_title in enumerate(menus):
77.                     # 创建菜单
78.                     m = Menu(menubar, tearoff=0)
79.                     # 添加菜单
80.                     menubar.add_cascade(label=m_title, menu=m)
81.                     # 将当前正在处理的菜单数据赋值给 tm
82.                     tm = items[i]
83.                     # 遍历 OrderedDict,默认只遍历它的 key
84.                     for label in tm:
85.                         print(label)
86.                         # 如果 value 又是 OrderedDict，说明是二级菜单
87.                         if isinstance(tm[label], OrderedDict):
88.                             # 创建子菜单、并添加子菜单
89.                             sm = Menu(m, tearoff=0)
90.                             m.add_cascade(label=label, menu=sm)
91.                             sub_dict = tm[label]
92.                             # 再次遍历子菜单对应的 OrderedDict，默认只遍历它的
    key
93.                             for sub_label in sub_dict:
94.                                 if sub_label.startswith('-'):
95.                                     # 添加分隔条
96.                                     sm.add_separator()
97.                                 else:
98.                                     # 添加菜单项
99.
    sm.add_command(label=sub_label,image=sub_dict[sub_label][0],
100.                                            command=sub_dict[sub_label][1],
    compound=LEFT)
101.                        elif label.startswith('-'):
102.                            # 添加分隔条
```

```
103.                                 m.add_separator()
104.                             else:
105.                                 # 添加菜单项
106.                                 m.add_command(label=label,image=tm[label][0],
107.                                     command=tm[label][1], compound=LEFT)
108.             # 生成所有需要的图标
109.             def init_icons(self):
110.                 self.master.filenew_icon =
    PhotoImage(file='images/filenew.png')
111.                 self.master.fileopen_icon =
    PhotoImage(file='images/fileopen.png')
112.                 self.master.save_icon = PhotoImage(file='images/save.png')
113.                 self.master.saveas_icon = PhotoImage(file='images/saveas.png')
114.                 self.master.signout_icon =
    PhotoImage(file='images/signout.png')
115.             # 生成工具条
116.             def init_toolbar(self):
117.                 # 创建并添加一个 Frame 作为工具条的容器
118.                 toolframe = Frame(self.master, height=20, bg='lightgray')
119.                 toolframe.pack(fill=X) # 该 Frame 容器放在窗口顶部
120.                 # 再次创建并添加一个 Frame 作为工具按钮的容器
121.                 frame = ttk.Frame(toolframe)
122.                 frame.pack(side=LEFT) # 该 Frame 容器放在容器左边
123.                 # 遍历 self.master 的全部数据，根据系统图标来创建工具栏按钮
124.                 for i, e in enumerate(dir(self.master)):
125.                     # 只处理属性名以_icon 结尾的属性（这些属性都是图标）
126.                     if e.endswith('_icon'):
127.                         ttk.Button(frame, width=20, image=getattr(self.master,
    e),
128.                                    command=None).grid(row=0, column=i, padx=1, pady=1,
    sticky=E)
129.         root = Tk()
130.         root.title("菜单测试")
131.         # 禁止改变窗口大小
132.         root.resizable(width=False, height=True)
133.         App(root)
```

```
134.         root.mainloop()
```

该程序比较实用，整个程序界面中的菜单并不是写死的，而是根据程序第 42~70 行代码自动生成的，这些代码中的每个 OrderdDict 代表一个菜单，它的每个 key-value 对代表一个菜单项，其中 key 是菜单文本，value 是一个元组，元组的第一个元素是菜单图标，第二个元素是为菜单绑定的事件处理函数。

因此，当开发者需要改变程序界面中的菜单时，并不需要修改后面的代码，只需修改第 42~70 行代码即可。

> 由于代码功能的限制，上面的第 42~70 行代码只能支持二级菜单，并不支持三级菜单。实际上 Tkinter 菜单完全支持三级，只是该程序没有做进一步处理。

上面程序也会自动生成工具条，只要为 self.master 添加了以 _icon 结尾的属性，程序就会自动把它们添加为工具条上的按钮。

运行上面程序，可以看到如图 2 所示的效果。

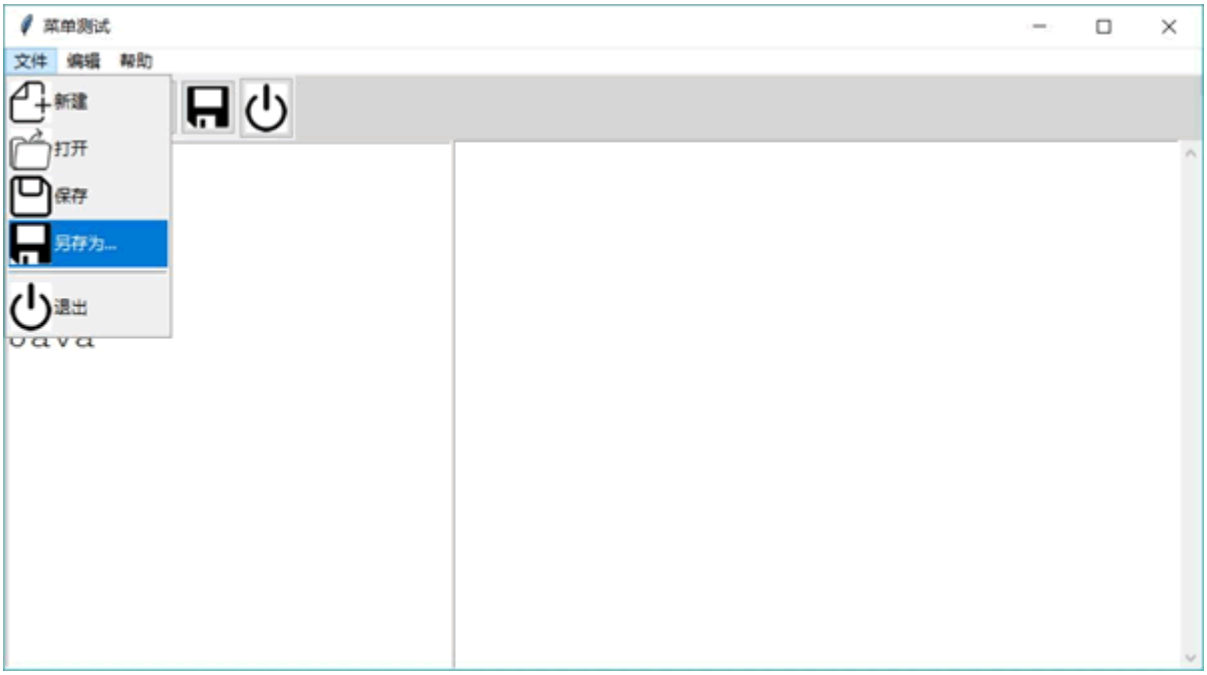


图 2 菜单和工具条

Menu 右键菜单

实现右键菜单很简单 程序只要先创建菜单 然后为目标组件的右键单击事件绑定处理函数，

当用户单击鼠标右键时，调用菜单的 post() 方法即可在指定位置弹出右键菜单。

如下程序示范了创建并添加右键菜单：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk
from collections import OrderedDict
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
```

```
                self.text = Text(self.master, height=12, width=60,
                    foreground='darkgray',
                    font=('微软雅黑', 12),
                    spacing2=8, # 设置行间距
                    spacing3=12) # 设置段间距
                self.text.pack()
                st = 'C 语言中文网成立于 2012 年初,' +\
                    '目前已经运营了将近 5 年,' +\
                    '我们致力于分享精品教程，帮助对编程感兴趣的读者\n'
                self.text.insert(END, st)
                # 为 text 组件的右键单击事件绑定处理方法
                self.text.bind('<Button-3>',self.popup)
                # 创建 Menu 对象，准备作为右键菜单
                self.popup_menu = Menu(self.master,tearoff = 0)
                self.my_items = (OrderedDict([('超大', 16), ('大',14), ('中',12),
                    ('小',10), ('超小',8)]),
                    OrderedDict([('红色','red'), ('绿色','green'), ('蓝色','blue')]))
                i = 0
                for k in ['字体大小','颜色']:
                    m = Menu(self.popup_menu, tearoff = 0)
                    # 添加子菜单
                    self.popup_menu.add_cascade(label=k ,menu = m)
                    # 遍历 OrderedDict 的 key（默认就是遍历 key）
                    for im in self.my_items[i]:
                        m.add_command(label=im,
command=self.handlerAdaptor(self.choose, x=im))
                    i += 1
            def popup(self, event):
                # 在指定位置显示菜单
                self.popup_menu.post(event.x_root,event.y_root)  #①
            def choose(self, x):
                # 如果用户选择修改字体大小的子菜单项
                if x in self.my_items[0].keys():
                    # 改变字体大小
```

```
                    self.text['font'] = ('微软雅黑', self.my_items[0][x])
                # 如果用户选择修改颜色的子菜单项
                if x in self.my_items[1].keys():
                    # 改变颜色
                    self.text['foreground'] = self.my_items[1][x]
            def handlerAdaptor(self, fun,**kwds):
                return lambda fun=fun, kwds=kwds: fun(**kwds)
        root = Tk()
        root.title("右键菜单测试")
        App(root)
        root.mainloop()
```

上面程序中，第 23-35 行代码用于创建一个 Menu，并为之添加菜单项。这段代码与前面介绍的创建菜单、添加菜单项的代码并没有区别。

程序中 ① 号代码位于 Text 组件的右键单击事件的处理函数内，这行代码调用 Menu 对象的 post() 方法弹出右键菜单，这意味着当用户在 Text 组件内单击鼠标右键时，Text 组件就会弹出右键菜单。

运行该程序，在界面上的 Text 组件内单击鼠标右键，将可以看到如图 3 所示的右键菜单。

图 3 右键菜单

25.Python tkinter Canvas 画布完全攻略

Tkinter 提供了 Canvas 组件来实现绘图。程序既可在 Canvas 中绘制直线、矩形、椭圆等各种几何图形，也可绘制图片、文字、UI 组件（如 Button）等。Canvas 允许重新改变这些图形项（Tkinter 将程序绘制的所有东西统称为 item）的属性，比如改变其坐标、外观等。

Canvas 组件的用法与其他 GUI 组件一样简单，程序只要创建并添加 Canvas 组件，然后调用该组件的方法来绘制图形即可。如下程序示范了最简单的 Canvas 绘图：

```
1.          from tkinter import *
2.
```

```
3.          # 创建窗口
4.          root = Tk()
5.          # 创建并添加 Canvas
6.          cv = Canvas(root, background='white')
7.          cv.pack(fill=BOTH, expand=YES)
8.          cv.create_rectangle(30, 30, 200, 200,
9.              outline='red', # 边框颜色
10.             stipple = 'question', # 填充的位图
11.             fill="red", # 填充颜色
12.             width=5 # 边框宽度
13.             )
14.         cv.create_oval(240, 30, 330, 200,
15.             outline='yellow', # 边框颜色
16.             fill='pink', # 填充颜色
17.             width=4 # 边框宽度
18.             )
19.         root.mainloop()
```

上面程序先创建并添加了 Canvas 组件，分别绘制了矩形和椭圆。运行上面程序，可以看到如图 1 所示的效果。

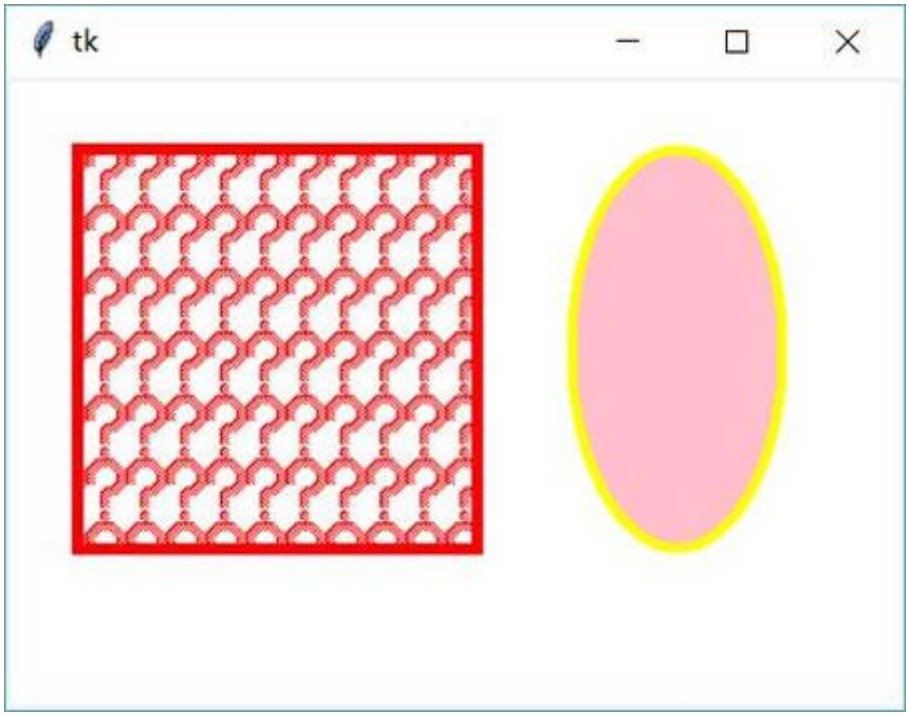


图 1 最简单的 Canvas 绘图

从上面程序可以看到，Canvas 提供了 create_rectangle() 方法绘制矩形和 create_oval() 方法绘制椭圆（包括圆，圆是椭圆的特例）。实际上，Canvas 还提供了如下方法来绘制各种图形：

- create_arc：绘制弧。
- create_bitmap：绘制位图。
- create_image：绘制图片。
- create_line()：绘制直线。
- create_polygon：绘制多边形。
- create_text：绘制文字。
- create_window：绘制组件。

Canvas 的坐标系统是绘图的基础，其中点 (0,0) 位于 Canvas 组件的左上角，X 轴水平向右延伸，Y 轴垂直向下延伸。

绘制上面这些图形时需要简单的几何基础：

- 在使用 create_line() 绘制直线时，需要指定两个点的坐标，分别作为直线的起点和终点。
- 在使用 create_rectangle() 绘制矩形时，需要指定两个点的坐标，分别作为矩形左上角点和右下角点的坐标。
- 在使用 create_oval() 绘制椭圆时，需要指定两个点的坐标，分别作为左上角点和右下角点的坐标来确定一个矩形，而该方法则负责绘制该矩形的内切椭圆，如图 2 所示。

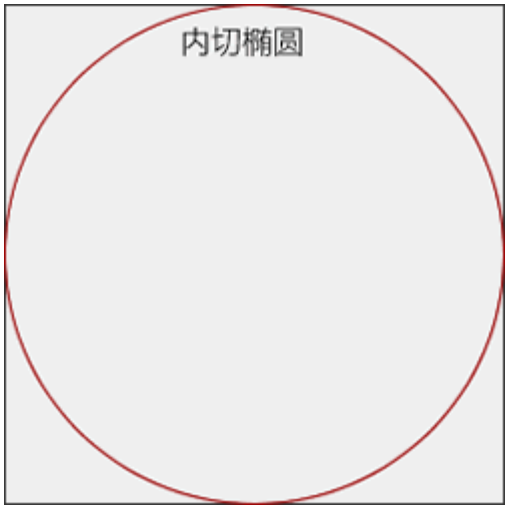


图 2 内切椭圆

从图 2 可以看出，只要矩形确定下来，该矩形的内切椭圆就能确定下来，而 create_oval() 方法所需要的两个坐标正是用于指定该矩形的左上角点和右下角点的坐标。

在使用 create_arc 绘制弧时，和 create_oval 的用法相似，因为弧是椭圆的一部分，因此同样也是指定左上角和右下角两个点的坐标，默认总是绘制从 3 点（0）开始，逆时针旋转 90° 的那一段弧。程序可通过 start 改变起始角度，也可通过 extent 改变转过的角度。

在使用 create_polygon 绘制多边形时，需要指定多个点的坐标来作为多边形的多个定点；在使用 create_bitmap、create_image、create_text、create_window 等方法时，只要指定一个坐标点，用于指定目标元素的绘制位置即可。

在绘制这些图形时可指定如下选项：

- fill：指定填充颜色。如果不指定该选项，默认不填充。
- outline：指定边框颜色。
- width：指定边框宽度。如果不指定该选项，边框宽度默认为 1。
- dash：指定边框使用虚线。该属性值既可为单独的整数，用于指定虚线中线段的长度；也可为形如（5,2,3）格式的元素，此时 5 指定虚线中线段的长度，2 指定间隔长度，3 指定虚线长度……依此类推。
- stipple：使用位图平铺进行填充。该选项可与 fill 选项结合使用，fill 选项用于指定位图的颜色。
- style：指定绘制弧的样式。该选项仅对 create_arc 方法起作用。该选项支持 PIESLICE（扇形）、CHORD（弓形）、ARC（仅绘制弧）选项值。
- start：指定绘制弧的起始角度。该选项仅对 create_arc 方法起作用。

- extent：指定绘制弧的角度。该选项仅对 create_arc 方法起作用。
- arrow :指定绘制直线时两端是否有箭头。该选项支持 NONE(两端无箭头)、FIRST（开始端有箭头）、LAST（结束端有箭头）、BOTH（两端都有箭头）选项值。
- arrowshape：指定箭头形状。该选项是一个形如 "20 20 10" 的字符串，字符串中的三个整数依次指定填充长度、箭头长度、箭头宽度。
- joinstyle：指定直接连接点的风格。仅对绘制直线和多向形有效。该选项支持 METTER、ROUND、BEVEL 选项值。
- anchor：指定绘制文字、GUI 组件的位置。该选项仅对 create_text()、create_window() 方法有效。
- justify：指定文字的对齐方式。该选项支持 CENTER、LEFT、RIGHT 常量值，该选项仅对 create_text 方法有效。

下面程序示范了通过不同的方法来绘制不同的图形，这些图形分别使用不同的边框、不同的填充效果：

```
from tkinter import *

# 创建窗口
root = Tk()
root.title('绘制图形项')
# 创建并添加 Canvas
cv = Canvas(root, background='white', width=830, height=830)
cv.pack(fill=BOTH, expand=YES)
columnFont = ('微软雅黑', 18)
titleFont = ('微软雅黑', 20, 'bold')
# 使用循环绘制文字
```

```
        for i, st in enumerate(['默认', '指定边宽', '指定填充', '边框颜色', '位图填充']):
            cv.create_text((130 + i * 140, 20),text = st,
            font = columnFont,
            fill='gray',
            anchor = W,
            justify = LEFT)
        # 绘制文字
        cv.create_text(10, 60, text = '绘制矩形',
            font = titleFont,
            fill='magenta',
            anchor = W,
            justify = LEFT)
        # 定义列表，每个元素的 4 个值分别指定边框宽度、填充色、边框颜色、位图填充
        options = [(None, None, None, None),
            (4, None, None, None),
            (4, 'pink', None, None),
            (4, 'pink', 'blue', None),
            (4, 'pink', 'blue', 'error')]
        # 采用循环绘制 5 个矩形
        for i, op in enumerate(options):
            cv.create_rectangle(130 + i * 140, 50, 240 + i * 140, 120,
                width = op[0], # 边框宽度
                fill = op[1], # 填充颜色
                outline = op[2], # 边框颜色
                stipple = op[3]) # 使用位图填充
        # 绘制文字
        cv.create_text(10, 160, text = '绘制椭圆',
            font = titleFont,
            fill='magenta',
            anchor = W,
            justify = LEFT)
        # 定义列表，每个元素的 4 个值分别指定边框宽度、填充色、边框颜色、位图填充
        options = [(None, None, None, None),
```

```
            (4, None, None, None),
            (4, 'pink', None, None),
            (4, 'pink', 'blue', None),
            (4, 'pink', 'blue', 'error')]
        # 采用循环绘制 5 个椭圆
        for i, op in enumerate(options):
            cv.create_oval(130 + i * 140, 150, 240 + i * 140, 220,
                width = op[0], # 边框宽度
                fill = op[1], # 填充颜色
                outline = op[2], # 边框颜色
                stipple = op[3]) # 使用位图填充
        # 绘制文字
        cv.create_text(10, 260, text = '绘制多边形',
            font = titleFont,
            fill='magenta',
            anchor = W,
            justify = LEFT)
        # 定义列表，每个元素的 4 个值分别指定边框宽度、填充色、边框颜色、位图填充
        options = [(None, "", 'black', None),
            (4, "", 'black', None),
            (4, 'pink', 'black', None),
            (4, 'pink', 'blue', None),
            (4, 'pink', 'blue', 'error')]
        # 采用循环绘制 5 个多边形
        for i, op in enumerate(options):
            cv.create_polygon(130 + i * 140, 320, 185 + i * 140, 250, 240 + i * 140, 320,
                width = op[0], # 边框宽度
                fill = op[1], # 填充颜色
                outline = op[2], # 边框颜色
                stipple = op[3]) # 使用位图填充
        # 绘制文字
        cv.create_text(10, 360, text = '绘制扇形',
            font = titleFont,
            fill='magenta',
```

```
79.             anchor = W,
80.             justify = LEFT)
81.         # 定义列表，每个元素的 4 个值分别指定边框宽度、填充色、边框颜色、位图填充
82.         options = [(None, None, None, None),
83.             (4, None, None, None),
84.             (4, 'pink', None, None),
85.             (4, 'pink', 'blue', None),
86.             (4, 'pink', 'blue', 'error')]
87.         # 采用循环绘制 5 个扇形
88.         for i, op in enumerate(options):
89.             cv.create_arc(130 + i * 140, 350, 240 + i * 140, 420,
90.                 width = op[0], # 边框宽度
91.                 fill = op[1], # 填充颜色
92.                 outline = op[2], # 边框颜色
93.                 stipple = op[3]) # 使用位图填充
94.         # 绘制文字
95.         cv.create_text(10, 460, text = '绘制弓形',
96.             font = titleFont,
97.             fill='magenta',
98.             anchor = W,
99.             justify = LEFT)
100.        # 定义列表，每个元素的 4 个值分别指定边框宽度、填充色、边框颜色、位图填充
101.        options = [(None, None, None, None),
102.            (4, None, None, None),
103.            (4, 'pink', None, None),
104.            (4, 'pink', 'blue', None),
105.            (4, 'pink', 'blue', 'error')]
106.        # 采用循环绘制 5 个弓形
107.        for i, op in enumerate(options):
108.            cv.create_arc(130 + i * 140, 450, 240 + i * 140, 520,
109.                width = op[0], # 边框宽度
110.                fill = op[1], # 填充颜色
111.                outline = op[2], # 边框颜色
112.                stipple = op[3], # 使用位图填充
```

```
                start = 30, # 指定起始角度
                extent = 60, # 指定逆时针转过角度
                style = CHORD) # CHORD 指定绘制弓
        # 绘制文字
        cv.create_text(10, 560, text = '仅绘弧',
            font = titleFont,
            fill='magenta',
            anchor = W,
            justify = LEFT)
        # 定义列表，每个元素的 4 个值分别指定边框宽度、填充色、边框颜色、位图填充
        options = [(None, None, None, None),
            (4, None, None, None),
            (4, 'pink', None, None),
            (4, 'pink', 'blue', None),
            (4, 'pink', 'blue', 'error')]
        # 采用循环绘制 5 个弧
        for i, op in enumerate(options):
            cv.create_arc(130 + i * 140, 550, 240 + i * 140, 620,
                width = op[0], # 边框宽度
                fill = op[1], # 填充颜色
                outline = op[2], # 边框颜色
                stipple = op[3], # 使用位图填充
                start = 30, # 指定起始角度
                extent = 60, # 指定逆时针转过角度
                style = ARC) # ARC 指定仅绘制弧
        # 绘制文字
        cv.create_text(10, 660, text = '绘制直线',
            font = titleFont,
            fill='magenta',
            anchor = W,
            justify = LEFT)
        # 定义列表，每个元素的 5 个值分别指定边框宽度、线条颜色、位图填充、箭头风格，箭头形状
        options = [(None, None, None, None, None),
            (6, None, None, BOTH, (20, 40, 10)),
```

```
            (6, 'pink', None, FIRST, (40, 40, 10)),
            (6, 'pink', None, LAST, (60, 50, 10)),
            (8, 'pink', 'error', None, None)]
        # 采用循环绘制 5 个弧
        for i, op in enumerate(options):
            cv.create_line(130 + i * 140, 650, 240 + i * 140, 720,
                width = op[0], # 边框宽度
                fill = op[1], # 填充颜色
                stipple = op[2], # 使用位图填充
                arrow = op[3], # 箭头风格
                arrowshape = op[4]) # 箭头形状
        # 绘制文字
        cv.create_text(10, 760, text = '绘制位图\n 图片、组件',
            font = titleFont,
            fill='magenta',
            anchor = W,
            justify = LEFT)
        # 定义包括 create_bitmap, create_image, create_window 三个方法的数组
        funcs = [Canvas.create_bitmap, Canvas.create_image,
    Canvas.create_window]
        # 为上面 3 个方法定义选项
        items = [{'bitmap' : 'questhead'},
    {'image':PhotoImage(file='images/fklogo.gif')},
            {'window':Button(cv,text = '单击我', padx=10, pady=5,
                command = lambda :print('按钮单击')),'anchor': W}]
        for i, func in enumerate(funcs):
            func(cv, 230 + i * 140, 780, **items[i])
        root.mainloop()
```

上面程序示范了 Canvas 中不同的 create_xxx 方法的功能和用法，它们可用于创建矩形、椭圆、多边形、扇形、弓形、弧、直线、位图、图片和组件等。在绘制不同的图形时可指定不同的选项，从而实现丰富的绘制效果。

运行上面程序，可以看到如图 3 所示的效果。

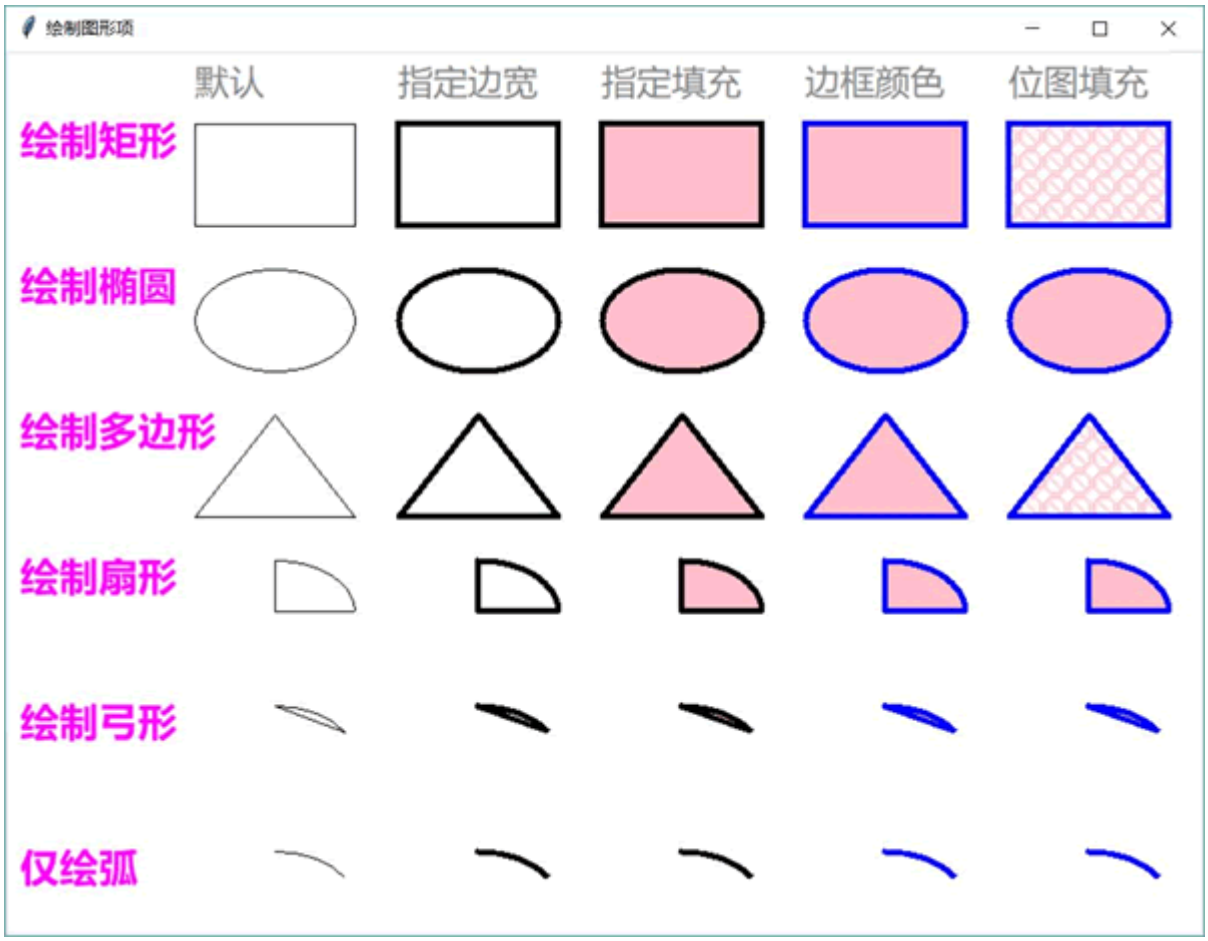


图 3 使用 Canvas 绘制图形

掌握了上面的绘制方法之后，实际上已经可以实现一些简单的游戏了。比如前面介绍的控制台五子棋，之前程序是在控制台打印游戏状态的，实际上程序完全可以在界面上绘制游戏状态，这样就能看到图形界面的五子棋了。

此外，该五子棋还需要根据用户的鼠标动作来确定下棋坐标，因此程序会为游戏界面的

<Button-1>（左键单击）、<Motion>（鼠标移动）、<Leave>（鼠标移出）事件绑定事件处理函数。下面程序示范了实现图形界面的五子棋：

```
from tkinter import *
import random

BOARD_WIDTH = 535
BOARD_HEIGHT = 536
BOARD_SIZE = 15
# 定义棋盘坐标的像素值和棋盘数组之间的偏移距。
X_OFFSET = 21
Y_OFFSET = 23
# 定义棋盘坐标的像素值和棋盘数组之间的比率。
X_RATE = (BOARD_WIDTH - X_OFFSET * 2) / (BOARD_SIZE - 1)
Y_RATE = (BOARD_HEIGHT - Y_OFFSET * 2) / (BOARD_SIZE - 1)
BLACK_CHESS = "●"
WHITE_CHESS = "○"
board = []
# 把每个元素赋为"╋"，代表无棋
for i in range(BOARD_SIZE) :
    row = ["╋"] * BOARD_SIZE
    board.append(row)
# 创建窗口
root = Tk()
# 禁止改变窗口大小
root.resizable(width=False, height=False)
# 修改图标
root.iconbitmap('images/fklogo.ico')
# 设置窗口标题
root.title('五子棋')
# 创建并添加 Canvas
cv = Canvas(root, background='white',
    width=BOARD_WIDTH, height=BOARD_HEIGHT)
cv.pack()
bm = PhotoImage(file="images/board.png")
```

```
33.         cv.create_image(BOARD_HEIGHT/2 + 1, BOARD_HEIGHT/2 + 1, image=bm)
34.         selectedbm = PhotoImage(file="images/selected.gif")
35.         # 创建选中框图片，但该图片默认不在棋盘中
36.         selected = cv.create_image(-100, -100, image=selectedbm)
37.         def move_handler(event):
38.             # 计算用户当前的选中点，并保证该选中点在 0～14 之间
39.             selectedX = max(0, min(round((event.x - X_OFFSET) / X_RATE), 14))
40.             selectedY = max(0, min(round((event.y - Y_OFFSET) / Y_RATE), 14))
41.             # 移动红色选择框
42.             cv.coords(selected,(selectedX * X_RATE + X_OFFSET,
43.                 selectedY * Y_RATE + Y_OFFSET))
44.         black = PhotoImage(file="images/black.gif")
45.         white = PhotoImage(file="images/white.gif")
46.         def click_handler(event):
47.             # 计算用户的下棋点，并保证该下棋点在 0～14 之间
48.             userX = max(0, min(round((event.x - X_OFFSET) / X_RATE), 14))
49.             userY = max(0, min(round((event.y - Y_OFFSET) / Y_RATE), 14))
50.             # 当下棋点没有棋子时，才能下棋子，用户才能下棋子
51.             if board[userY][userX] == "╋":
52.                 cv.create_image(userX * X_RATE + X_OFFSET, userY * Y_RATE +
    Y_OFFSET,
53.                     image=black)
54.                 board[userY][userX] = "●"
55.                 while(True):
56.                     comX = random.randint(0, BOARD_SIZE - 1)
57.                     comY = random.randint(0, BOARD_SIZE - 1)
58.                     # 如果电脑要下棋的点没有棋子时，才能让电脑下棋
59.                     if board[comY][comX] == "╋": break
60.                 cv.create_image(comX * X_RATE + X_OFFSET, comY * Y_RATE +
    Y_OFFSET,
61.                     image=white)
62.                 board[comY][comX] = "○"
63.         def leave_handler(event):
64.             # 将红色选中框移出界面
65.             cv.coords(selected, -100, -100)
66.         # 为鼠标移动事件绑定事件处理函数
```

```
67.         cv.bind('<Motion>', move_handler)
68.         # 为鼠标点击事件绑定事件处理函数
69.         cv.bind('<Button-1>', click_handler)
70.         # 为鼠标移出事件绑定事件处理函数
71.         cv.bind('<Leave>', leave_handler)
72.         root.mainloop()
```

上面程序先绘制了五子棋的棋盘，该棋盘就是一张预先准备好的图片，然后绘制选择框，当用户鼠标在棋盘上移动时，该选择框显示用户鼠标当前停留在哪个下棋点上。

随后程序调用了 Canvas 的 coords() 方法，该方法负责重设选择框的坐标。这是 Tkinter 绘图的特别之处，绘制好的每一个图形项都不是固定的，程序后面完全可以修改它们。因此，程序将会控制选择框图片随着用户鼠标的移动而改变位置。

程序根据用户鼠标单击来绘制黑色棋子（也就是下黑棋）和白色棋子（也就是下白棋）。在绘制黑色棋子和白色棋子的同时，也改变了底层代表棋盘状态的 board 列表的数据，这样即可记录下棋状态，从而让程序在后面可以根据 board[] 列表来判断胜负（本来这个功能在 Charlie 的程序中是有的，此处为了突出绘图的主题，作者删除了这部分）。另外，也可以加入人工智能，根据 board[] 列表来决定电脑的下棋点。

运行该程序，可以看到如图 4 所示的效果：

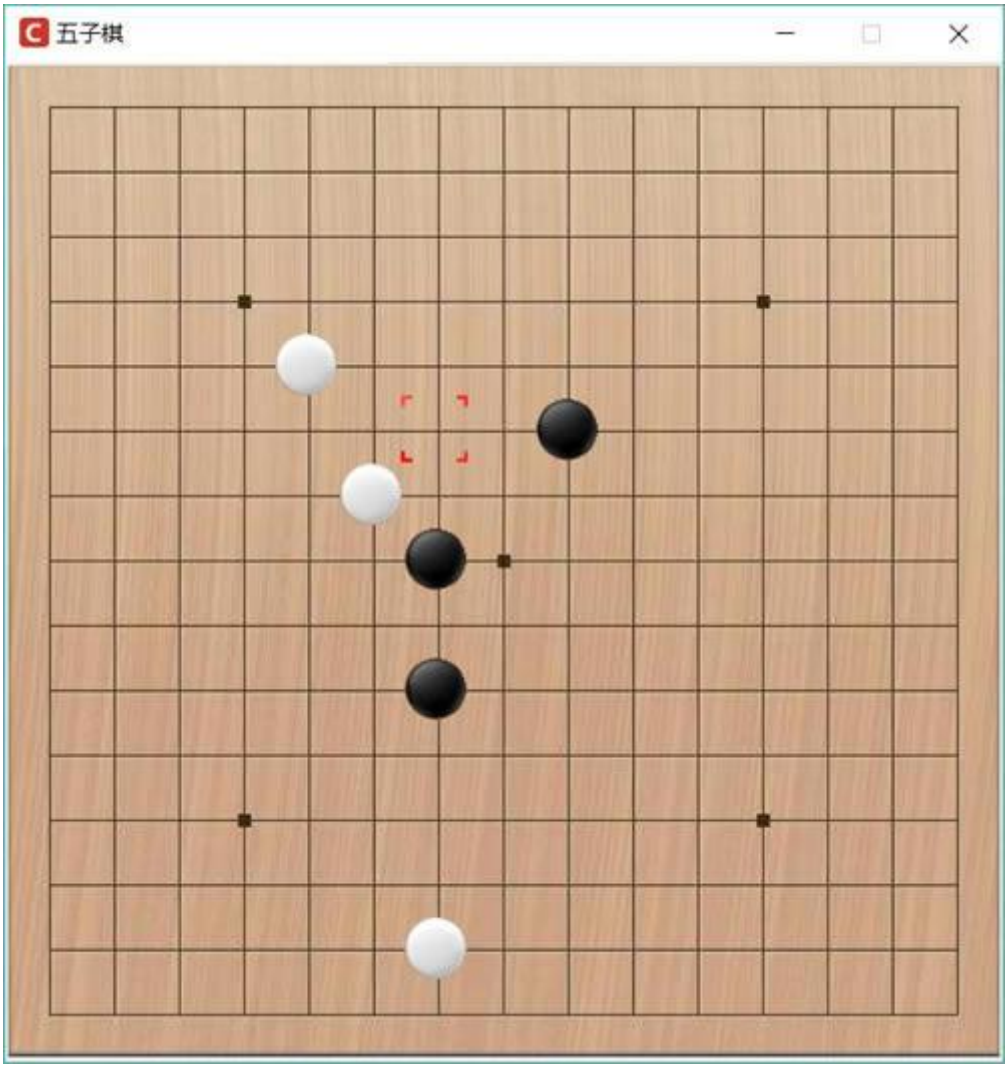


图 4 五子棋

在上面这个程序中，电脑下棋采用的方式是随机下棋，因此下得比较“凌乱”。如果要让电脑下棋更加智能，则可通过简单的人工智能来实现，本教程暂不涉及。

Canvas 操作图形项的标签

在 Canvas 中通过 create_xxx 方法绘制图形项之后，这些图形项并不是完全静态的图形，每个图形项都是一个独立的对象，程序完全可以动态地修改、删除这些图形项。

Canvas 以“堆叠”的形式来管理这些图形项，先绘制的图形项位于“堆叠”的下面，后绘制的图形项位于“堆叠”的上面。因此，如果两个图形项有重叠的部分，那么后绘制的图形项（位于上面）会遮挡先绘制的图形项。

为了修改、删除这些图形项，程序需要先获得这些图形项的引用。获得这些图形项的引用有两种方式：

1. 通过图形项的 id，也就是 Canvas 执行 create_xxx() 方法的返回值。一般来说，create_xxx() 会依次返回 1、2、3 等整数作为图形项的 id。
2. 通过图形项的 tag（标签）。

在 Canvas 中调用 create_xxx() 方法绘图时，还可传入一个 tags 选项，该选项可以为所绘制的图形项（比如矩形、椭圆、多边形等）添加一个或多个 tag（标签）。此外，Canvas 还允许调用方法为图形项添加 tag、删除 tag 等，这些 tag 也相当于该图形项的标识，程序完全可以根据 tag 来获取图形项。

总结来说，Canvas 提供了如下方法来为图形项添加 tag：

- addtag_aboove(self, newtag, tagOrId)：为 tagOrId 对应图形项的上一个图形项添加新 tag。
- addtag_all(self, newtag)：为所有图形项添加新 tag。
- addtag_below(self, newtag, tagOrId)：为 tagOrId 对应图形项的下一个图形项添加新 tag。

- addtag_closest(self, newtag, x, y)：为和 x、y 点最接近的图形项添加新 tag。
- addtag_enclosed(self, newtag, x1, y1, x2, y2)：为指定矩形区域内最上面的图形项添加新 tag。其中 x1、y1 确定矩形区域的左上角坐标；x2、y2 确定矩形区域的右下角坐标。
- addtag_overlapping(self, newtag, x1, y1, x2, y2)：为与指定矩形区域重叠的最上面的图形项添加 tag。
- addtag_withtag(self, newtag, tagOrId)：为 tagOrId 对应图形项添加新 tag。

Canvas 提供了如下方法来删除图形项的 tag：

- dtag(self, *args)：删除指定图形项的 tag。

Canvas 提供了如下方法来获取图形项的所有 tag：

- gettags(self, *args)：获取指定图形项的所有 tag。

Canvas 提供了如下方法根据 tag 来获取其对应的所有图形项：

- find_withtag(self, tagOrId)：获取 tagOrId 对应的所有图形项。

为了更好地理解上面方法的作用，下面用一个程序来进行演示：

```
1.          from tkinter import *
```

```
2.
3.          # 创建窗口
4.          root = Tk()
5.          root.title('操作标签')
6.          # 创建并添加 Canvas
7.          cv = Canvas(root, background='white', width=620, height=250)
8.          cv.pack(fill=BOTH, expand=YES)
9.          # 绘制一个矩形框
10.         rt = cv.create_rectangle(40, 40, 300, 220,
11.             outline='blue', width=2,
12.             tag = ('t1', 't2', 't3', 'tag4'))  # 为该图形项指定标签
13.         # 访问图形项的 id，也就是编号
14.         print(rt) # 1
15.         # 绘制一个椭圆
16.         oval = cv.create_oval(350, 50, 580, 200,
17.             fill='yellow', width=0,
18.             tag = ('g1', 'g2', 'g3', 'tag4'))  # 为该图形项指定标签
19.         # 访问图形项的 id，也就是编号
20.         print(oval) # 2
21.         # 根据指定 tag 该 tag 对应的所有图形项
22.         print(cv.find_withtag('tag4')) # (1, 2)
23.         # 获取指定图形项的所有 tag
24.         print(cv.gettags(rt))  # ('t1', 't2', 't3', 'tag4')
25.         print(cv.gettags(2))  # ('g1', 'g2', 'g3', 'tag4')
26.         cv.dtag(1, 't1') # 删除 id 为 1 的图形项上名为 t1 的 tag
27.         cv.dtag(oval, 'g1') # 删除 id 为 oval 的图形项上名为 g1 的 tag
28.         # 获取指定图形项的所有 tag
29.         print(cv.gettags(rt))  # ('tag4', 't2', 't3')
30.         print(cv.gettags(2))  # ('tag4', 'g2', 'g3')
31.         # 为所有图形项添加 tag
32.         cv.addtag_all('t5')
33.         print(cv.gettags(1))  # ('tag4', 't2', 't3', 't5')
34.         print(cv.gettags(oval))  # ('tag4', 'g2', 'g3', 't5')
35.         # 为指定图形项添加 tag
36.         cv.addtag_withtag('t6', 'g2')
37.         # 获取指定图形项的所有 tag
```

```
        print(cv.gettags(1))  # ('tag4', 't2', 't3', 't5')
        print(cv.gettags(oval))  # ('tag4', 'g2', 'g3', 't5', 't6')
        # 为指定图形项上面的图形项添加 tag, t2 上面的就是 oval 图形项
        cv.addtag_above('t7', 't2')
        print(cv.gettags(1))  # ('tag4', 't2', 't3', 't5')
        print(cv.gettags(oval))  # ('tag4', 'g2', 'g3', 't5', 't6', 't7')
        # 为指定图形项下面的图形项添加 tag, g2 下面的就是 rt 图形项
        cv.addtag_below('t8', 'g2')
        print(cv.gettags(1))  # ('tag4', 't2', 't3', 't5', 't8')
        print(cv.gettags(oval))  # ('tag4', 'g2', 'g3', 't5', 't6', 't7')
        # 为最接近指定点的图形项添加 tag，最接近 360、90 的图形项是 oval
        cv.addtag_closest('t9', 360, 90)
        print(cv.gettags(1))  # ('tag4', 't2', 't3', 't5', 't8')
        print(cv.gettags(oval))  # ('tag4', 'g2', 'g3', 't5', 't6', 't7', 't9')
        # 为位于指定区域内（几乎覆盖整个图形区）的最上面的图形项添加 tag
        cv.addtag_closest('t10', 30, 30, 600, 240)
        print(cv.gettags(1))  # ('tag4', 't2', 't3', 't5', 't8')
        print(cv.gettags(oval))  # ('tag4', 'g2', 'g3', 't5', 't6', 't7', 't9',
't10')
        # 为与指定区域内重合的最上面的图形项添加 tag
        cv.addtag_closest('t11', 250, 30, 400, 240)
        print(cv.gettags(1))  # ('tag4', 't2', 't3', 't5', 't8')
        print(cv.gettags(oval))  # ('tag4', 'g2', 'g3', 't5', 't6', 't7', 't9',
't10', 't11')
        root.mainloop()
```

上面程序示范了操作图形项的 tag 的方法，而且列出了每次操作之后的输出结果。因此，读者可以结合程序的运行结果来理解 Canvas 是如何管理图形项的 tag 的。

操作图形项

在 Canvas 中获取图形项之后，接下来可通过 Canvas 提供的大量方法来操作图形项。总结起来，Canvas 提供了如下方法在图形项“堆叠”中查找图形项：

find_above(self, tagOrId)：返回 tagOrId 对应图形项的上一个图形项。
find_all(self)：返回全部图形项。
find_below(self, tagOrId)：返回 tagOrId 对应图形项的下一个图形项。
find_closest(self, x, y)：返回和 x 、y 点最接近的图形项。
find_enclosed(self, x1, y1, x2, y2)：返回位于指定矩形区域内最上面的图形项。
find_overlapping(self, x1, y1, x2, y2)：返回与指定矩形区域重叠的最上面的图形项。
find_withtag(self, tagOrId)：返回 tagOrId 对应的全部图形项。

Canvas 提供了如下方法在图形项“堆叠”中移动图形项：
tag_lower(self, *args)|lower：将 args 的第一个参数对应的图形项移到“堆叠”的最下面。也可额外指定一个参数，代表移动到指定图形项的下面。
tag_raise(self, *args)|lift：将 args 的第一个参数对应的图形项移到“堆叠”的最上面。也可额外指定一个参数，代表移动到指定图形项的上面。

如果程序希望获取或修改图形项的选项，则可通过 Canvas 的如下方法来操作：
itemcget(self, tagOrId, option)：获取 tagOrId 对应图形项的 option 选项值。
itemconfig(self, tagOrId, cnf=None, **kw)：为 tagOrId 对应图形项配置选项。
itemconfigure：该方法与上一个方法完全相同。

Canvas 提供了如下方法来改变图形项的大小和位置：
coords(self, *args)：重设图形项的大小和位置。
move(self, *args)：移动图形项，但不能改变大小。简单来说，就是在图形项的 x、y 基础上加上新的 mx、my 参数。
scale(self, *args)：缩放图形项。该方法的 args 参数要传入 4 个值，其中前两个值指定缩放中心；后两个值指定 x、y 方向的缩放比。

此外，Canvas 还提供了如下方法来删除图形项或文字图形项（由 create_text 方法创建）中间的部分文字：
delete(self, *args)：删除指定 id 或 tag 对应的全部图形项。
dchars(self, *args)：删除文字图形项中间的部分文字。

下面程序示范了操作图形项的方法：

纯文本复制

```
from tkinter import *
from tkinter import colorchooser
import threading

# 创建窗口
root = Tk()
```

```
root.title('操作图形项')
# 创建并添加 Canvas
cv = Canvas(root, background='white', width=400, height=350)
cv.pack(fill=BOTH, expand=YES)
# 该变量用于保存当前选中的图形项
current = None
# 该变量用于保存当前选中的图形项的边框颜色
current_outline = None
# 该变量用于保存当前选中的图形项的边框宽度
current_width = None
# 该函数用于高亮显示选中图形项（边框颜色会 red、yellow 之间切换）
def show_current():
    # 如果当前选中项不为 None
    if current is not None:
        # 如果当前选中图形项的边框色为 red，将它改为 yellow
        if cv.itemcget(current, 'outline') == 'red':
            cv.itemconfig(current, width=2,
                outline='yellow')
        # 否则，将颜色改为 red
        else:
            cv.itemconfig(current, width=2,
                outline='red')
    global t
    # 通过定时器指定 0.2 秒之后执行 show_current 函数
    t = threading.Timer(0.2, show_current)
    t.start()
# 通过定时器指定 0.2 秒之后执行 show_current 函数
t = threading.Timer(0.2, show_current)
t.start()
# 分别创建矩形、椭圆、和圆
rect = cv.create_rectangle(30, 30, 250, 200,
    fill='magenta', width='0')
oval = cv.create_oval(180, 50, 380, 180,
    fill='yellow', width='0')
circle = cv.create_oval(120, 150, 300, 330,
    fill='pink', width='0')
```

```
43.         bottomF = Frame(root)
44.         bottomF.pack(fill=X,expand=True)
45.         liftbn = Button(bottomF, text='向上',
46.             # 将椭圆移动到它上面的 item 之上
47.             command=lambda : cv.tag_raise(oval, cv.find_above(oval)))
48.         liftbn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)
49.         lowerbn = Button(bottomF, text='向下',
50.             # 将椭圆移动到它下面的 item 之下
51.             command=lambda : cv.tag_lower(oval, cv.find_below(oval)))
52.         lowerbn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)
53.         def change_fill():
54.             # 弹出颜色选择框,让用户选择颜色
55.             fill_color = colorchooser.askcolor(parent=root,
56.                 title='选择填充颜色',
57.                 # 初始颜色设置为椭圆当前的填充色（fill 选项值）
58.                 color =  cv.itemcget(oval, 'fill'))
59.             if fill_color is not None:
60.                 cv.itemconfig(oval, fill=fill_color[1])
61.         fillbn = Button(bottomF, text='改变填充色',
62.             # 该按钮触发 change_fill 函数
63.             command=change_fill)
64.         fillbn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)
65.         def change_outline():
66.             # 弹出颜色选择框,让用户选择颜色
67.             outline_color = colorchooser.askcolor(parent=root,
68.                 title='选择边框颜色',
69.                 # 初始颜色设置为椭圆当前的边框色（outline 选项值）
70.                 color = cv.itemcget(oval, 'outline'))
71.             if outline_color is not None:
72.                 cv.itemconfig(oval, outline=outline_color[1],
73.                     width=4)
74.         outlinebn = Button(bottomF, text='改变边框色',
75.             # 该按钮触发 change_outline 函数
76.             command=change_outline)
77.         outlinebn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)
78.         movebn = Button(bottomF, text='右下移动',
```

```
79.             # 调用 move 方法移动图形项
80.             command=lambda : cv.move(oval, 15, 10))
81.         movebn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)
82.         coordsbn = Button(bottomF, text='位置复位',
83.             # 调用 coords 方法重设图形项的大小和位置
84.             command=lambda : cv.coords(oval, 180, 50, 380, 180))
85.         coordsbn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)
86.         # 再次添加 Frame 容器
87.         bottomF = Frame(root)
88.         bottomF.pack(fill=X,expand=True)
89.         zoomoutbn = Button(bottomF, text='缩小',
90.             # 调用 scale 方法对图形项进行缩放
91.             # 前面两个坐标指定缩放中心，后面两个参数指定横向、纵向的缩放比
92.             command=lambda : cv.scale(oval, 180, 50, 0.8, 0.8))
93.         zoomoutbn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)
94.         zoominbn = Button(bottomF, text='放大',
95.             # 调用 scale 方法对图形项进行缩放
96.             # 前面两个坐标指定缩放中心，后面两个参数指定横向、纵向的缩放比
97.             command=lambda : cv.scale(oval, 180, 50, 1.2, 1.2))
98.         zoominbn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)
99.         def select_handler(ct):
100.            global current, current_outline, current_width
101.            # 如果 ct 元组包含了选中项
102.            if ct is not None and len(ct) > 0:
103.                ct = ct[0]
104.                # 如果 current 对应的图形项不为空
105.                if current is not None:
106.                    # 恢复 current 对应的图形项的边框
107.                    cv.itemconfig(current, outline=current_outline,
108.                        width = current_width)
109.                # 获取当前选中图形项的边框信息
110.                current_outline = cv.itemcget(ct, 'outline')
111.                current_width = cv.itemcget(ct, 'width')
112.                # 使用 current 保存当前选中项
113.                current = ct
114.        def click_handler(event):
```

```
115.             # 获取当前选中的图形项
116.             ct = cv.find_closest(event.x, event.y)
117.             # 调用 select _handler 处理选中图形项
118.             select_handler(ct)
119.         def click_select():
120.             # 取消为“框选”绑定的两个事件处理函数
121.             cv.unbind('<B1-Motion>')
122.             cv.unbind('<ButtonRelease-1>')
123.             # 为“点选”绑定鼠标点击的事件处理函数
124.             cv.bind('<Button-1>', click_handler)
125.         clickbn = Button(bottomF, text='点选图形项',
126.             # 该按钮触发 click_select 函数
127.             command=click_select)
128.         clickbn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)
129.         # 记录鼠标拖动的第一个点的 x、y 坐标
130.         firstx = firsty = None
131.         # 记录前一次绘制的、代表选择区的虚线框
132.         prev_select = None
133.         def drag_handler(event):
134.             global firstx, firsty, prev_select
135.             # 刚开始拖动时，用鼠标位置为 firstx、firsty 赋值
136.             if firstx is None and firsty is None:
137.                 firstx, firsty = event.x, event.y
138.             leftx, lefty = min(firstx, event.x), min(firsty, event.y)
139.             rightx, righty = max(firstx, event.x), max(firsty, event.y)
140.             # 删除上一次绘制的虚线选择框
141.             if prev_select is not None:
142.                 cv.delete(prev_select)
143.             # 重新绘制虚线选择框
144.             prev_select = cv.create_rectangle(leftx, lefty, rightx, righty,
145.                 dash=2)
146.         def release_handler(event):
147.             global firstx, firsty
148.             if prev_select is not None:
149.                 cv.delete(prev_select)
150.             if firstx is not None and firsty is not None:
```

```
                leftx, lefty = min(firstx, event.x), min(firsty, event.y)
                rightx, righty = max(firstx, event.x), max(firsty, event.y)
                firstx = firsty = None
                # 获取当前选中的图形项
                ct = cv.find_enclosed(leftx, lefty, rightx, righty)
                # 调用 select _handler 处理选中图形项
                select_handler(ct)
        def rect_select():
            # 取消为“点选”绑定的事件处理函数
            cv.unbind('<Button-1>')
            # 为“框选”绑定鼠标拖动、鼠标释放的事件处理函数
            cv.bind('<B1-Motion>', drag_handler)
            cv.bind('<ButtonRelease-1>', release_handler)
        rectbn = Button(bottomF, text='框选图形项',
            # 该按钮触发 rect_select 函数
            command=rect_select)
        rectbn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)
        deletebn = Button(bottomF, text='删除',
            # 删除图形项
            command=lambda : cv.delete(oval))
        deletebn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)
        root.mainloop()
```

读者可自定运行此程序，查看运行效果。

26.tkinter Canvas tag_bind：指定图形项绑定事件处理函数或方法

Canvas 提供了一个 tag_bind() 方法 ,该方法用于为指定图形项绑定事件处理函数或方法，这样图形项就可以响应用户动作了。

下面程序示范了为矩形的单击事件绑定事件处理函数：

```
1.          from tkinter import *
2.
3.          root = Tk()
4.          # 创建一个 Canvas，设置其背景色为白色
5.          cv = Canvas(root,bg = 'white')
6.          cv.pack()
7.          # 创建一个 rectangle
8.          cv.create_rectangle(30, 30, 220, 150,
9.              width = 8,
10.             tags = ('r1','r2','r3'))
11.         def first(event):
12.             print('第一次的函数')
13.         def second(event):
14.             print('第二次的函数')
15.         # 为指定图形项的左键单击事件绑定处理函数
16.         cv.tag_bind('r1','<Button-1>', first)
17.         # 为指定图形项的左键单击事件绑定处理函数
18.         cv.tag_bind('r1','<Button-1>', second, add=True) # add 为 True 是添加，
    否则是替代
19.         root.mainloop()
```

上面程序中第 16 行代码为 r1 对应的图形项的左键单击事件绑定事件处理函数，第 18 行代码依然为 r1 对应的图形项的左键单击事件绑定事件处理函数，其中的 add 选项为 True，表示为该图形项再次添加一个事件处理函数（即为该图形项的单击事件绑定两个事件处理函数），如果将 add 选项设为 False，则表示第二次添加的事件处理函数会取代第一次添加的事件处理函数。

下面将开发一个功能相对完善的绘图程序，该程序可以让用户绘制直线、矩形、椭圆、多边形，用户还可以通过鼠标左键单击来选中所绘制的图形，也可以通过鼠标右键拖动来移动图形项。程序代码如下：

```
from tkinter import *
# 导入 ttk
from tkinter import ttk
from tkinter import colorchooser
import threading

class App:
    def __init__(self, master):
        self.master = master
        # 保存设置初始的边框宽度
        self.width = IntVar()
        self.width.set(1)
        # 保存设置初始的边框颜色
        self.outline = 'black'
        # 保存设置初始的填充颜色
        self.fill = None
        # 记录拖动时前一个点的 x、y 坐标
        self.prevx = self.prevy = -10
        # 记录拖动开始的第一个点的 x、y 坐标
        self.firstx = self.firsty = -10
        # 记录拖动右键来移动图形时前一个点的 x、y 坐标
        self.mv_prevx = self.mv_prevy = -10
        # item_type 记录要绘制哪种图形
        self.item_type = 0
        self.points = []
        self.init_widgets()
        self.temp_item = None
        self.temp_items = []
        # 初始化选中的图形项
        self.choose_item = None
    # 创建界面组件
    def init_widgets(self):
        self.cv = Canvas(root, background='white')
        self.cv.pack(fill=BOTH, expand=True)
        # 为鼠标左键拖动事件、鼠标左键释放事件绑定处理函数
        self.cv.bind('<B1-Motion>', self.drag_handler)
```

```
37.                 self.cv.bind('<ButtonRelease-1>', self.release_handler)
38.                 # 为鼠标左键双击事件绑定处理函数
39.                 self.cv.bind('<Double-1>', self.double_handler)
40.                 f = ttk.Frame(self.master)
41.                 f.pack(fill=X)
42.                 self.bns = []
43.                 # 采用循环创建多个按钮，用于绘制不同的图形
44.                 for i, lb in enumerate(('直线', '矩形', '椭圆', '多边形', '铅笔
    ')):
45.                     bn = Button(f, text=lb, command=lambda i=i:
    self.choose_type(i))
46.                     bn.pack(side=LEFT, ipadx=8,ipady=5, padx=5)
47.                     self.bns.append(bn)
48.                 # 默认选中直线
49.                 self.bns[self.item_type]['relief'] = SUNKEN
50.                 ttk.Button(f, text='边框颜色',
51.                     command=self.choose_outline).pack(side=LEFT,
    ipadx=8,ipady=5, padx=5)
52.                 ttk.Button(f, text='填充颜色',
53.                     command=self.choose_fill).pack(side=LEFT, ipadx=8,ipady=5,
    padx=5)
54.                 om = ttk.OptionMenu(f,
55.                     self.width, # 绑定变量
56.                     '1', # 设置初始选中值
57.                     '0', # 以下多个值用于设置菜单项
58.                     '1',
59.                     '2',
60.                     '3',
61.                     '4',
62.                     '5',
63.                     '6',
64.                     '7',
65.                     '8',
66.                     command = None)
67.                 om.pack(side=LEFT, ipadx=8,ipady=5, padx=5)
68.             def choose_type(self, i):
```

```
69.                     # 将所有按钮恢复默认状态
70.                     for b in self.bns: b['relief'] = RAISED
71.                     # 将当前按钮设置选中样式
72.                     self.bns[i]['relief'] = SUNKEN
73.                     # 设置要绘制的图形
74.                     self.item_type = i
75.                 # 处理选择边框颜色的方法
76.                 def choose_outline(self):
77.                     # 弹出颜色选择对话框
78.                     select_color = colorchooser.askcolor(parent=self.master,
79.                         title="请选择边框颜色", color=self.outline)
80.                     if select_color is not None:
81.                         self.outline = select_color[1]
82.                 # 处理选择填充颜色的方法
83.                 def choose_fill(self):
84.                     # 弹出颜色选择对话框
85.                     select_color = colorchooser.askcolor(parent=self.master,
86.                         title="请选择填充颜色", color=self.fill)
87.                     if select_color is not None:
88.                         self.fill = select_color[1]
89.                     else:
90.                         self.fill = None
91.                 def drag_handler(self, event):
92.                     # 如果是绘制直线
93.                     if self.item_type == 0:
94.                         # 如果第一个点不存在（self.firstx 和 self.firsty 都小于 0）
95.                         if self.firstx < -1 and self.firsty < -1:
96.                             self.firstx, self.firsty = event.x, event.y
97.                         # 删除上一次绘制的虚线图形
98.                         if self.temp_item is not None:
99.                             self.cv.delete(self.temp_item)
100.                        # 重新绘制虚线
101.                        self.temp_item = self.cv.create_line(self.firstx,
    self.firsty,
102.                            event.x, event.y, dash=2)
103.                    # 如果是绘制矩形或椭圆
```

```
104.                 if self.item_type == 1 or self.item_type == 2:
105.                     # 如果第一个点不存在（self.firstx 和 self.firsty 都小于 0）
106.                     if self.firstx < -1 and self.firsty < -1:
107.                         self.firstx, self.firsty = event.x, event.y
108.                     # 删除上一次绘制的虚线图形
109.                     if self.temp_item is not None:
110.                         self.cv.delete(self.temp_item)
111.                     leftx, lefty = min(self.firstx, event.x), min(self.firsty,
    event.y)
112.                     rightx, righty = max(self.firstx, event.x), max(self.firsty,
    event.y)
113.                     # 重新绘制虚线选择框
114.                     self.temp_item = self.cv.create_rectangle(leftx, lefty,
    rightx, righty,
115.                         dash=2)
116.                 if self.item_type == 3:
117.                     self.draw_polygon = True
118.                     # 如果第一个点不存在（self.firstx 和 self.firsty 都小于 0）
119.                     if self.firstx < -1 and self.firsty < -1:
120.                         self.firstx, self.firsty = event.x, event.y
121.                     # 删除上一次绘制的虚线图形
122.                     if self.temp_item is not None:
123.                         self.cv.delete(self.temp_item)
124.                     # 重新绘制虚线
125.                     self.temp_item = self.cv.create_line(self.firstx,
    self.firsty,
126.                         event.x, event.y, dash=2)
127.                 if self.item_type == 4:
128.                     # 如果前一个点存在（self.prevx 和 self.prevy 都大于 0）
129.                     if self.prevx > 0 and self.prevy > 0:
130.                         self.cv.create_line(self.prevx, self.prevy, event.x,
    event.y,
131.                             fill=self.outline, width=self.width.get())
132.                     self.prevx, self.prevy = event.x, event.y
133.             def item_bind(self, t):
134.                 # 为鼠标右键拖动事件绑定处理函数
```

```
135.                 self.cv.tag_bind(t, '<B3-Motion>', self.move)
136.                 # 为鼠标右键释放事件绑定处理函数
137.                 self.cv.tag_bind(t, '<ButtonRelease-3>', self.move_end)
138.             def release_handler(self, event):
139.                 # 删除临时绘制的虚线图形项
140.                 if self.temp_item is not None:
141.                     # 如果不是绘制多边形
142.                     if self.item_type != 3:
143.                         self.cv.delete(self.temp_item)
144.                     # 如果绘制多边形，将之前绘制的虚线先保存下来，以便后面删除
    它们
145.                     else:
146.                         self.temp_items.append(self.temp_item)
147.                 self.temp_item = None
148.                 # 如果是绘制直线
149.                 if self.item_type == 0:
150.                     # 如果第一个点存在（self.firstx 和 self.firsty 都大于 0）
151.                     if self.firstx > 0 and self.firsty > 0:
152.                         # 绘制实际的直线
153.                         t = self.cv.create_line(self.firstx, self.firsty,
154.                             event.x, event.y, fill=self.outline,
    width=self.width.get())
155.                         # 为鼠标左键单击事件绑定处理函数，用于选择被单击的图形
    项
156.                         self.cv.tag_bind(t, '<Button-1>',
157.                             lambda event=event, t=t:
    self.choose_item_handler(event,t))
158.                         self.item_bind(t)
159.                 # 如果是绘制矩形或椭圆
160.                 if self.item_type == 1 or self.item_type == 2:
161.                     # 如果第一个点存在（self.firstx 和 self.firsty 都大于 0）
162.                     if self.firstx > 0 and self.firsty > 0:
163.                         leftx, lefty = min(self.firstx, event.x),
    min(self.firsty, event.y)
164.                         rightx, righty = max(self.firstx, event.x),
    max(self.firsty, event.y)
```

```
165.                         if self.item_type == 1:
166.                             # 绘制实际的矩形
167.                             t = self.cv.create_rectangle(leftx, lefty, rightx,
    righty,
168.                                 outline=self.outline, fill=self.fill,
    width=self.width.get())
169.                         if self.item_type == 2:
170.                             # 绘制实际的椭圆
171.                             t = self.cv.create_oval(leftx, lefty, rightx,
    righty,
172.                                 outline=self.outline, fill=self.fill,
    width=self.width.get())
173.                         # 为鼠标左键单击事件绑定处理函数，用于选择被单击的图形
    项
174.                         self.cv.tag_bind(t, '<Button-1>',
175.                             lambda event=event, t=t:
    self.choose_item_handler(event,t))
176.                         self.item_bind(t)
177.                 if self.item_type != 3:
178.                     self.prevx = self.prevy = -10
179.                     self.firstx = self.firsty = -10
180.                 # 如果正在绘制多边形
181.                 elif(self.draw_polygon):
182.                     # 将第一个点添加到列表中
183.                     self.points.append((self.firstx, self.firsty))
184.                     self.firstx, self.firsty = event.x, event.y
185.             def double_handler(self, event):
186.                 # 只处理绘制多边形的情形
187.                 if self.item_type == 3:
188.                     t = self.cv.create_polygon(*self.points,
189.                         outline=self.outline, fill="" if self.fill is None else
    self.fill,
190.                         width=self.width.get())
191.                     # 为鼠标左键单击事件绑定处理函数，用于选择被单击的图形项
192.                     self.cv.tag_bind(t, '<Button-1>',
```

```
                        lambda event=event, t=t:
    self.choose_item_handler(event,t))
                    self.item_bind(t)
                    # 清空所有保存的点数据
                    self.points.clear()
                    # 将 self.firstx = self.firsty 设置为-10，停止绘制
                    self.firstx = self.firsty = -10
                    # 删除所有临时的虚线框
                    for it in self.temp_items: self.cv.delete(it)
                    self.temp_items.clear()
                    self.draw_polygon = False
            # 根据传入的参数 t 来选中对应的图形项
            def choose_item_handler(self, event, t):
                # 使用 self.choose_item 保存当前选中项
                self.choose_item = t
            # 定义移动图形项的方法
            def move(self, event):
                # 如果被选中图形项不为空，才可以执行移动
                if self.choose_item is not None:
                    # 如果前一个点存在(self.mv_prevx 和 self.mv_prevy 都大于 0)
                    if self.mv_prevx > 0 and self.mv_prevy > 0:
                        # 移动选中的图形项
                        self.cv.move(self.choose_item, event.x - self.mv_prevx,
                            event.y - self.mv_prevy)
                    self.mv_prevx, self.mv_prevy = event.x, event.y
            # 结束移动的方法
            def move_end(self, event):
                self.mv_prevx = self.mv_prevy = -10
            def delete_item(self, event):
                # 如果被选中的 item 不为空，删除被选中的图形项
                if self.choose_item is not None:
                    self.cv.delete(self.choose_item)
        root = Tk()
        root.title("绘图工具")
        root.iconbitmap('images/fklogo.ico')
        root.geometry('800x680')
```

```
228.         app = App(root)
229.         root.bind('<Delete>', app.delete_item)
230.         root.mainloop()
```

上面程序稍微有点复杂，这是由于该程序并不是简单地绘制图形，而是当用户拖动鼠标时可以动态地绘制虚线图形，只有当用户松开鼠标时才真正完成绘制，因此该程序完全是一个很实用的绘图工具。

程序中第 135 行代码为鼠标右键拖动事件绑定事件处理函数，该事件处理函数会根据用户的鼠标拖动来移动图形项：第 137 行代码为鼠标右键松开事件绑定事件处理函数，该事件处理函数会结束鼠标拖动行为。

接下来程序完成一件事情，即为程序绘制的图形项的鼠标左键单击事件绑定事件处理函数，当用户在图形项上单击鼠标时，程序会选中该图形项由于程序为所有图形项的鼠标左键单击事件都绑定了事件处理函数，因此不管用户单击哪个图形项，程序都会选择该图形项，接下来就可以操作和删除该图形项了。

程序中还为 Delete 按键事件绑定了事件处理函数，当用户按 Delete 键时，程序会删除当前所选择的图形项。运行上面程序，用户完全可以按照自己的意愿进行创作，绘制各种图形。随意创作后的效果如图 1 所示。

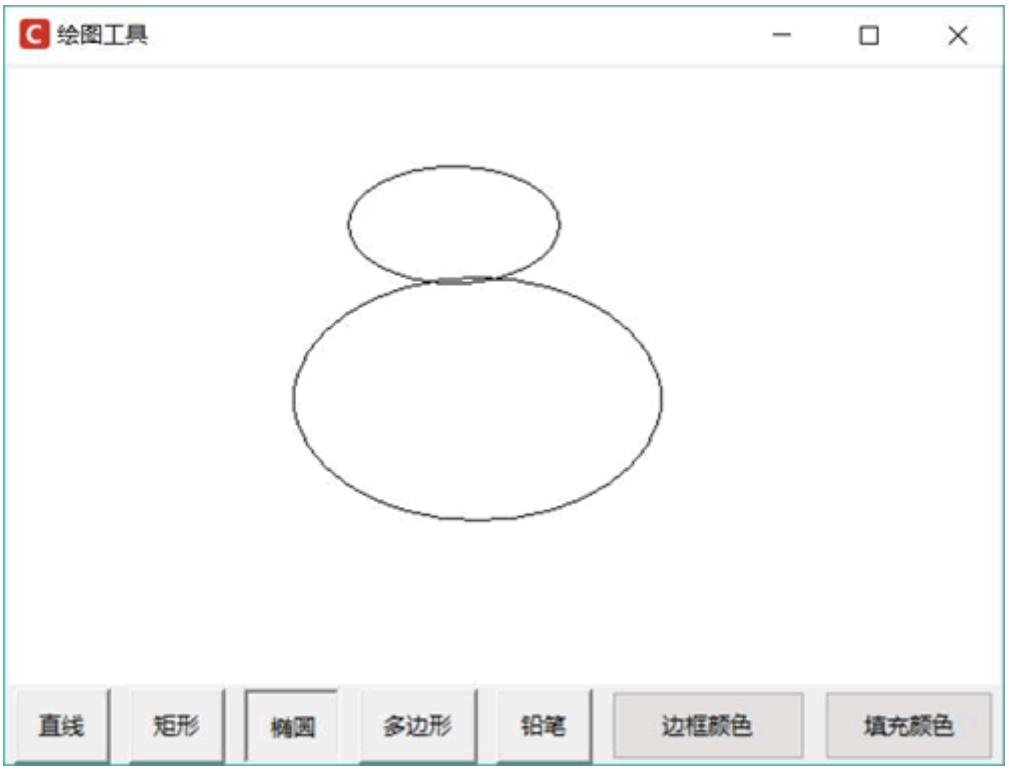


图 1 绘图工具

27.Python tkinter Canvas 绘制动画

其实前面程序中的高亮显示已经是动画效果了。程序会用红色、黄色交替显示几何图形的边框，这样看上去就是动画效果了。实现其他动画效果也是这个原理，程序只要增加一个定时器，周期性地改变界面上图形项的颜色、大小、位置等选项，用户看上去就是所谓的“动画”了。

下面以一个简单的桌面弹球游戏来介绍使用 Canvas 绘制动画。在游戏界面上会有一个小球，该小球会在界面上滚动，遇到边界或用户挡板就会反弹。该程序涉及两个动画：
小球转动：小球转动是一个“逐帧动画”，程序会循环显示多张转动的小球图片，这样用户就会看到小球转动的效果。
小球移动：只要改变小球的坐标程序就可以控制小球移动。

为了让用户控制挡板移动，程序还为 Canvas 的向左箭头、向右箭头绑定了事件处理函数。下面是桌面弹球游戏的程序：

```
1.          from tkinter import *
2.          from tkinter import messagebox
```

```
import threading
import random
GAME_WIDTH = 500
GAME_HEIGHT = 680
BOARD_X = 230
BOARD_Y = 600
BOARD_WIDTH = 80
BALL_RADIUS = 9
class App:
    def __init__(self, master):
        self.master = master
        # 记录小球动画的第几帧
        self.ball_index = 0
        # 记录游戏是否失败的旗标
        self.is_lose = False
        # 初始化记录小球位置的变量
        self.curx = 260
        self.cury = 30
        self.boardx = BOARD_X
        self.init_widgets()
        self.vx = random.randint(3, 6) # x 方向的速度
        self.vy = random.randint(5, 10) # y 方向的速度
        # 通过定时器指定 0.1 秒之后执行 moveball 函数
        self.t = threading.Timer(0.1, self.moveball)
        self.t.start()
    # 创建界面组件
    def init_widgets(self):
        self.cv = Canvas(root, background='white',
            width=GAME_WIDTH, height=GAME_HEIGHT)
        self.cv.pack()
        # 让画布得到焦点，从而可以响应按键事件
        self.cv.focus_set()
        self.cv.bms = []
        # 初始化小球的动画帧
        for i in range(8):
```

```
38.                     self.cv.bms.append(PhotoImage(file='images/ball_' + str(i+1)
    + '.gif'))
39.                 # 绘制小球
40.                 self.ball = self.cv.create_image(self.curx, self.cury,
41.                     image=self.cv.bms[self.ball_index])
42.                 self.board = self.cv.create_rectangle(BOARD_X, BOARD_Y,
43.                     BOARD_X + BOARD_WIDTH, BOARD_Y + 20, width=0,
    fill='lightblue')
44.                 # 为向左箭头按键绑定事件，挡板左移
45.                 self.cv.bind('<Left>', self.move_left)
46.                 # 为向右箭头按键绑定事件，挡板右移
47.                 self.cv.bind('<Right>', self.move_right)
48.             def move_left(self, event):
49.                 if self.boardx <= 0:
50.                     return
51.                 self.boardx -= 5
52.                 self.cv.coords(self.board, self.boardx, BOARD_Y,
53.                     self.boardx + BOARD_WIDTH, BOARD_Y + 20)
54.             def move_right(self, event):
55.                 if self.boardx + BOARD_WIDTH >= GAME_WIDTH:
56.                     return
57.                 self.boardx += 5
58.                 self.cv.coords(self.board, self.boardx, BOARD_Y,
59.                     self.boardx + BOARD_WIDTH, BOARD_Y + 20)
60.             def moveball(self):
61.                 self.curx += self.vx
62.                 self.cury += self.vy
63.                 # 小球到了右边墙壁，转向
64.                 if self.curx + BALL_RADIUS >= GAME_WIDTH:
65.                     self.vx = -self.vx
66.                 # 小球到了左边墙壁，转向
67.                 if self.curx - BALL_RADIUS <= 0:
68.                     self.vx = -self.vx
69.                 # 小球到了上边墙壁，转向
70.                 if self.cury - BALL_RADIUS <= 0:
71.                     self.vy = -self.vy
```

```
72.                     # 小球到了挡板处
73.                     if self.cury + BALL_RADIUS >= BOARD_Y:
74.                         # 如果在挡板范围内
75.                         if self.boardx <= self.curx <= (self.boardx + BOARD_WIDTH):
76.                             self.vy = -self.vy
77.                         else:
78.                             messagebox.showinfo(title='失败', message='您已经输了
')
79.                             self.is_lose = True
80.                     self.cv.coords(self.ball, self.curx, self.cury)
81.                     self.ball_index += 1
82.                     self.cv.itemconfig(self.ball,
image=self.cv.bms[self.ball_index % 8])
83.                     # 如果游戏还未失败，让定时器继续执行
84.                     if not self.is_lose:
85.                         # 通过定时器指定 0.1 秒之后执行 moveball 函数
86.                         self.t = threading.Timer(0.1, self.moveball)
87.                         self.t.start()
88.         root = Tk()
89.         root.title("弹球游戏")
90.         root.iconbitmap('images/fklogo.ico')
91.         root.geometry('%dx%d' % (GAME_WIDTH, GAME_HEIGHT))
92.         # 禁止改变窗口大小
93.         root.resizable(width=False, height=False)
94.         App(root)
95.         root.mainloop()
```

上面程序中第 26、27 行代码通过线程启动了一个定时器，该定时器控制 moveball() 方法每隔 0.1 秒执行一次，而 moveball() 方法中通过改变小球的坐标可以实现小球移动的效果；通过改变小球的图片，可以实现小球滚动的效果。

运行上面程序，可以看到如图 1 所示的游戏效果：

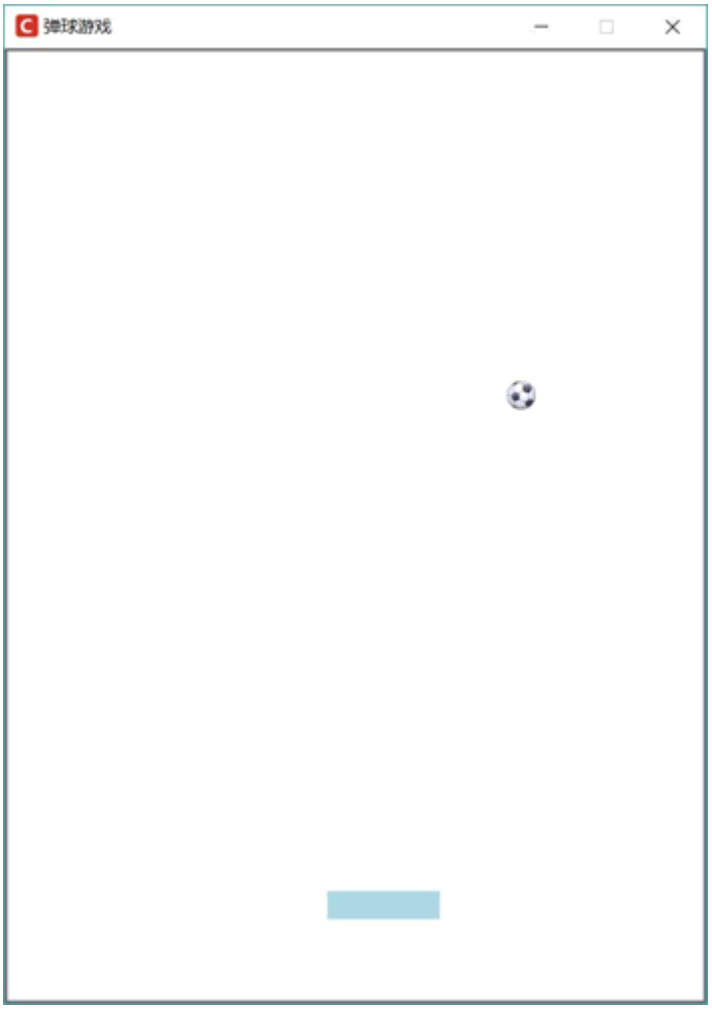


图 1 桌面弹球游戏

如果读者对使用 Python 开发游戏有很浓厚的兴趣，则不要局限于使用简单的 Tkinter 和 Canvas 画布。记住 Python 的优势是拥有大量的工具和库。如果想使用 Python 开发游戏，则尽量考虑使用专业的游戏工具库，比如 Pygame。

第 14 章 Python 文件操作（I/O）

Python 文件 I/O(输入/输出)是比较乏味的事情 ,因为看不到明显的运行效果 ,但 I/O 是所有程序都必需的部分，使用输入机制，允许程序读取外部数据 (包括来自磁盘、光盘等存

储设备的数据），用户输入数据；使用输出机制，允许程序记录运行状态，将程序数据输出到磁盘、光盘等存储设备中；

Python 提供有非常丰富的文件 I/O 支持，它既提供了 pathlib 和 os.path 来操作各种路径，也提供了全局的 open() 函数来打开文件（在打开文件之后，程序既可读取文件的内容，也可向文件输出内容）。而且 Python 提供了多种方式来读取文件内容，因此非常简单、灵活。

此外，在 Python 的 os 模块下也包含了大量进行文件 I/O 的函数，使用这些函数来读取、写入文件也很方便，因此读者可以根据需要选择不同的方式来读写文件。

Pyhon 还提供了 tempfile 模块来创建临时文件和临时目录，tempfile 模块下的高级 API 会自动管理临时文件的创建和删除；当程序不再使用临时文件和临时目录时，程序会自动删除临时文件和临时目录。

1.Python 文件基本操作

要知道，在变量、序列以及对象中存储的数据是暂时的，程序结束后就会丢失，如果想要长时间保存程序中的指定数据，最常用的方法是经其存储到磁盘文件中。Python 提供了内置的文件对象，以及对文件、目录进行操作的内置模块，通过这些技术可以很方便地将数据保存到文件（如文本文件等）中。

Python 中，对文件的操作有很多种，常见的操作包括创建、删除、修改权限、读取、写入等，这些操作可大致分为以下 2 类：

1. 删除、修改权限：作用于文件本身，属于系统级操作。

2. 写入、读取：是文件最常用的操作，作用于文件的内容，属于应用级操作。

其中，对文件的系统级操作功能单一，比较容易实现，可以借助 Python 中的转用模块(os、sys 等），并调用模块中的指定函数来实现。例如，假设如下代码文件的同级目录中有一个文件“a.txt”，通过调用 os 模块中的 remove 函数，可以将该文件删除，具体实现代码如下：

```
import os
os.remove("a.txt")
```

而对于文件的应用级操作，通常需要按照固定的步骤进行操作，且实现过程相对比较复杂，同时也是本章重点要讲解的部分。

文件的应用级操作可以分为以下 3 步，每一步都需要借助对应的函数实现：

1. 打开文件：使用 open() 函数，该函数会返回一个文件对象；
2. 对已打开文件做读/写操作：读取文件内容可使用 read()、readline() 以及 readlines() 函数；向文件中写入内容，可以使用 write() 函数。
3. 关闭文件：完成对文件的读/写操作之后，最后需要关闭文件，可以使用 close() 函数。

一个文件，必须在打开之后才能对其进行操作，并且在操作结束之后，还应该将其关闭，这 3 步的顺序不能打乱。

以上操作文件的各个函数，会各自作为一节进行详细介绍。

2.Python open()函数详解：打开指定文件

在 Python 中，如果想要操作文件，首先需要创建或者打开指定的文件，并创建一个文件对象，而这些工作可以通过内置的 open() 函数实现。

open() 函数用于创建或打开指定文件，该函数的语法格式如下：

```
file = open(file_name [, mode[, buffering]])
```

此格式中，用 [] 括起来的部分，表示作为可选操作，可以使用，也可以省略。其中，各个参数所代表的含义如下：

- file：表示要创建的文件对象。
- file_mode：要创建或打开文件的文件名称，该名称要用引号（单引号或双引号都可以）括起来。需要注意的是，如果要打开的文件和当前执行的代码文件位于同一目录，则直接写文件名即可；否则，此参数需要指定打开文件所在的完整路径。
- mode：可选参数，用于指定文件的打开模式。可选的打开模式如表 1 所示。如果不写，则默认以只读（r）模式打开文件。
- buffing：可选参数，用于指定对文件做读写操作时，是否使用缓冲区（本节后续会详细介绍）。

open() 函数支持的文件打开模式如表 1 所示。

表 1 open 函数支持的文件打开模式

模式	意义	注意事项
r	只读模式打开文件，读文件内容的指针会放在文件的开头。	操作的文件必须存在。
rb	以二进制格式、采用只读模式打开文件，读文件内容的指针位于文件的开头，一般用于非文本文件，如图片文件、音频文件等。	
r+	打开文件后，既可以从头读取文件内容，也可以从开头向文件中写入新的内容，写入的新内容会覆盖文件中等长度的原有内容。	
rb+	以二进制格式、采用读写模式打开文件，读写文件的指针会放在文件的开头，通常针对非文本文件（如音频文件）。	
w	以只写模式打开文件，若该文件存在，打开时会清空文件中原有的内容。	若文件存在，会清空其原有内容(覆盖文件)；反之，则创建新文件。
wb	以二进制格式、只写模式打开文件，一般用于非文本文件（如音频文件）	
w+	打开文件后，会对原有内容进行清空，并对该文件有读写权限。	
wb+	以二进制格式、读写模式打开文件，一般用于非文本文件	

a	以追加模式打开一个文件，对文件只有写入权限，如果文件已经存在，文件指针将放在文件的末尾（即新写入内容会位于已有内容之后）；反之，则会创建新文件。	
ab	以二进制格式打开文件，并采用追加模式，对文件只有写权限。如果该文件已存在，文件指针位于文件末尾（新写入文件会位于已有内容之后）；反之，则创建新文件。	
a+	以读写模式打开文件；如果文件存在，文件指针放在文件的末尾（新写入文件会位于已有内容之后）；反之，则创建新文件。	
ab+	以二进制模式打开文件，并采用追加模式，对文件具有读写权限，如果文件存在，则文件指针位于文件的末尾（新写入文件会位于已有内容之后）；反之，则创建新文件。	

文件打开模式，直接决定了后续可以对文件做哪些操作。例如，使用 r 模式打开的文件，后续编写的代码只能读取文件，而无法改动文件内容。

图 2 中，将以上几个容易混淆的文件打开模式的功能，做了很好的对比：

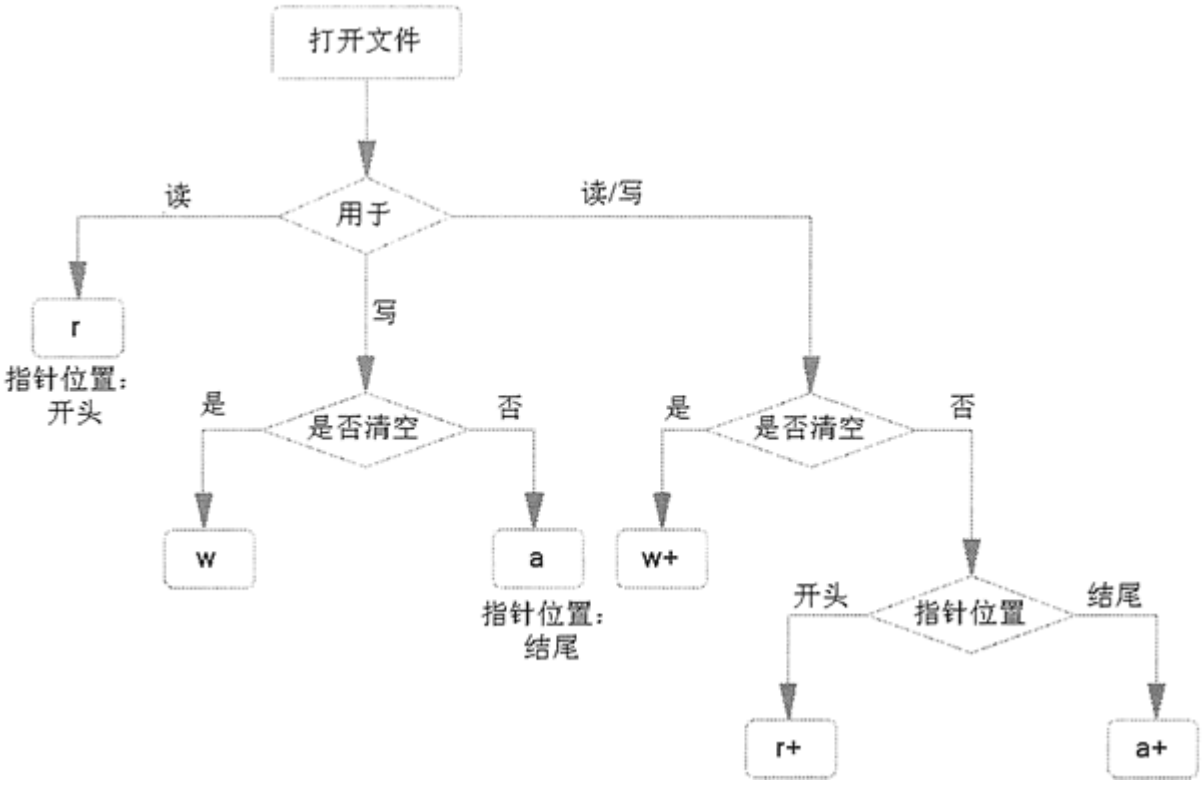


图 2 不同文件打开模式的功能

【例 1】默认打开 "a.txt" 文件。

```
#当前程序文件同目录下没有 a.txt 文件
file = open("a.txt")
print(file)
```

当以默认模式打开文件时，默认使用 r 权限，由于该权限要求打开的文件必须存在，因此运行此代码会报如下错误：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 1, in <module>
    file = open("a.txt")
FileNotFoundError: [Errno 2] No such file or directory: 'a.txt'
```

现在，在程序文件同目录下，手动创建一个 a.txt 文件，并再次运行该程序，其运行结果为：

```
<_io.TextIOWrapper name='a.txt' mode='r' encoding='cp936'>
```

可以看到，当前输出结果中，输出了 file 文件对象的相关信息，包括打开文件的名称、打开模式、打开文件时所使用的编码格式。

使用 open() 打开文件时，默认采用 GBK 编码。但当要打开的文件不是 GBK 编码格式时，可以在使用 open() 函数时，手动指定打开文件的编码格式，例如：

```
file = open("a.txt",encoding="utf-8")
```

此行代码表示，打开以 UTF-8 编码方式保存的 a.txt 文件。

open()是否需要缓冲区

众所周知，计算机外设（比如硬盘、网络）的 I/O 速度远远低于访问内存的速度，而程序执行 I/O 时要么将内存中的数据写入外设，要么将外设中的数据读取到内存，如果不使用缓冲，就必须等外设输入或输出一个字节后，内存中的程序才能输出或输入一个字节，这意味着内存中的程序大部分时间都处于等待状态。

内存中程序的读写速度很快，如果不使用缓冲，则程序必须等待外设进行同步读写。打个形象的比喻，就像在一条堵车的马路上开着跑车，必须等前面的车开一点，跑车才能前进一点。

因此，一般建议打开缓冲。在打开缓冲之后，当程序执行输出时，程序会先将数据输出到缓冲区中，而不用等待外设同步输出，当程序把所有数据都输出到缓冲区中之后，程序就可以去干其他事情了，留着缓冲区慢慢同步到外设即可；反过来，当程序执行输入时，程序会先等外设将数据读入缓冲区中，而不用等待外设同步输入。

在使用 open() 函数时，如果其第三个参数是 0（或 False），那么该函数打开的文件就是不带缓冲的；如果其第三个参数是 1（或 True），则该函数打开的文件就是带缓冲的，此时程序执行 I/O 将具有更好的性能。如果其第三个参数是大于 1 的整数，则该整数用于指定缓冲区的大小（单位是字节）；如果其第三个参数为任何负数，则代表使用默认的缓冲区大小。

open()文件对象常用的属性

成功打开文件之后，可以调用文件对象本身拥有的属性获取当前文件的部分信息，其常见的属性为：

- file.closed：判断文件是否已经关闭。
- file.mode：返回被打开文件的访问模式。
- file.name：返回文件的名称。

如下程序简单示范了如何打开文件和访问被打开文件的属性：

```
1.          # 以默认方式打开文件
2.          f = open('open_test.py')
3.          # 访问文件的编码方式
```

```
4.          print(f.encoding) # cp936
5.          # 访问文件的访问模式
6.          print(f.mode) # r
7.          # 访问文件是否已经关闭
8.          print(f.closed) # False
9.          # 访问文件对象打开的文件名
10.         print(f.name) # open_test.py
```

上面程序使用 open() 内置函数打开了 open_test.py 文件，接下来程序访问了被打开文件的各属性。运行上面程序，可以看到如下输出结果：

```
cp936

r

False

open_test.py
```

在打开文件之后，接下来就可以开始读取文件内容了。

3.以文本格式和二进制格式打开文件，到底有什么区别？

我们知道，open() 函数第二个参数是一个字符串，用于指定文件的打开方式，如果该字符串中出现 b，则表示以二进制格式打开文件；反之，则以普通的文本格式打开文件。

那么，文本文件和二进制文件有什么区别呢？

根据我们以往的经验，文本文件通常用来保存肉眼可见的字符，比如 .txt 文件、.c 文件、.dat 文件等，用文本编辑器打开这些文件，我们能够顺利看懂文件的内容。而二进制文件通常用来保存视频、图片、音频等不可阅读的内容，当用文本编辑器打开这些文件，会

看到一堆乱码，根本看不懂。

实际上，从数据存储的角度上分析，二进制文件和文本文件没有区别，它们的内容都是以二进制的形式保存在磁盘中的。

我们之所以能看懂文本文件的内容，是因为文本文件中采用的是 ASCII、UTF-8、GBK 等字符编码，文本编辑器可以识别出这些编码格式，并将编码值转换成字符展示出来。而对于二进制文件，文本编辑器无法识别这些文件的编码格式，只能按照字符编码格式胡乱解析，所以最终看到的是一堆乱码。

open()的文本格式和二进制格式

使用 open() 函数以文本格式打开文件和以二进制格式打开文件，唯一的区别是对文件中换行符的处理不同。

在 Windows 系统中，文件中用 "\r\n" 作为行末标识符（即换行符），当以文本格式读取文件时，会将 "\r\n" 转换成 "\n"；反之，以文本格式将数据写入文件时，会将 "\n" 转换成 "\r\n"。这种隐式转换换行符的行为，对用文本格式打开文本文件是没有问题的，但如果用文本格式打开二进制文件，就有可能改变文本中的数据（将 \r\n 隐式转换为 \n）。

而在 Unix/Linux 系统中，默认的文件换行符就是 \n，因此在 Unix/Linux 系统中文本格式和二进制格式并无本质的区别。

总的来说，为了保险起见，对于 Windows 平台最好用 b 打开二进制文件；对于 Unix/Linux 平台，打开二进制文件，可以用 b，也可以不用。

4.Python read()函数：按字节（字符）读取文件

文件对象（open() 函数的返回值）提供了 read() 函数可以按字节或字符读取文件内容，到底是读取字节还是字符，取决于使用 open() 函数打开文件时，是否使用了 b 模式，如果使用了 b 模式，则每次读取一个字节；反之，则每次读取一个字符。

read() 函数的基本语法格式如下：

```
file.read([size])
```

其中，file 表示打开的文件对象；size 作为一个可选参数，用于指定要读取的字符个数，如果省略，则默认一次性读取所有内容。

【例 1】采用循环读取整个文件的内容。

```
#a.txt 文件内容为：C 语言中文网
f = open("a.txt", 'r', True)
while True:
    # 每次读取一个字符
    ch = f.read(1)
    # 如果没有读到数据，跳出循环
    if not ch:
        break
    # 输出 ch
    print(ch, end='')
f.close()
```

运行结果为：

```
C 语言中文网
```

上面程序采用循环依次读取每一个字符（因为程序没有使用 b 模式），每读取到一个字符，程序就输出该字符。

正如从上面程序所看到的，当程序读写完文件之后，推荐立即调用 close() 方法来关闭文件，这样可以避免资源泄露（后续章节会详细介绍 close() 函数）。

注意，在调用 read() 函数读取文件内容时，成功读取的前提是在 open() 函数中使用 r 或 r+ 的模式打开文件，否则（比如将上面程序中 open( ) 的打开模式改为 w），程序会抛出 io.UnsupportedOperation 异常：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 4, in <module>
    ch = f.read(1)
io.UnsupportedOperation: not readable
```

【例 2】调用 read() 方法时不传入参数，该方法默认会读取全部文件内容。例如：

```
1.          f = open("a.txt", 'r', True)
2.          # 直接读取全部文件
3.          print(f.read())
4.          f.close()
```

运行结果为：

C 语言中文网

read()函数抛出 UnicodeDecodeError 异常的解决方法

当使用 open() 函数打开文本文件时，默认会使用当前操作系统的字符集，比如 Windows 平台，open() 函数默认使用 GBK 字符集。因此，上面程序读取的 a.txt 也必须使用 GBK 字符集保存；否则，程序就会出现 UnicodeDecodeError 错误。

如果要读取的文件所使用的字符集和当前操作系统的字符集不匹配，则有两种解决方式：

1. 使用二进制模式读取，然后用 bytes 的 decode() 方法恢复成字符串。
2. 利用 codecs 模块的 open() 函数来打开文件，该函数在打开文件时允许指定字符集。

例如，下面程序使用二进制模式来读取文本文件：

```
# 指定使用二进制方式读取文件内容，a.txt 以 utf-8 编码存储
f = open("a.txt", 'rb', True)
# 直接读取全部文件，并调用 bytes 的 decode 将字节内容恢复成字符串
print(f.read().decode('utf-8'))
f.close()
```

上面程序在调用 open() 函数时，传入了 rb 模式，这表明采用二进制模式读取文件，此时文件对象的 read() 方法返回的是 bytes 对象，程序可调用 bytes 对象的 decode() 方法将它恢复成字符串。由于此时读取的 a.txt 文件是以 UTF-8 的格式保存的，因此程序需要使用 decode() 方法恢复字符串时显式指定使用 UTF-8 字符集。

下面程序使用 codes 模块的 open() 函数来打开文件，此时可以显式指定字符集：

```
import codecs
#指定使用 utf-8 字符集读取文件内容
f = codecs.open("a.txt", 'r', 'utf-8', buffering=True)
while True:
    #每次读取一个字符
    ch = f.read(1)
    #如果没有读取到数据，则跳出循环
    if not ch : break
    #输出 ch
    print (ch, end='')
f.close()
```

上面程序在调用 open() 函数时显式指定使用 UTF-8 字符集，这样程序在读取文件内容时就完全没有问题了。

5.Python readline()和 readlines()函数：按行读取文件

使用 read() 函数读取文件时，如果文件过大，则一次读取全部内容到内存，容易造成内存不足，而相比每次限制读取字符（或字节）的个数，更推荐大家使用逐行读取文件的方式。

> 一般情况下，逐行读取只适用于以文本格式打开的文件，道理很简单，只有文本文件才有行的概念，二进制文件没有所谓行的概念。

文件对象提供了 readline() 和 readlines() 两个函数来逐行读取文件，其中 readline() 函数用于读取一行内容，而 readlines() 函数用于读取文件内的所有行。

readline()函数

readline() 函数用于读取文件中的一行，包含最后的换行符 “\n”。此函数的基本语法格式

为：

```
file.readline([size])
```

其中，file 为打开的文件对象；size 为可选参数，用于指定读取每一行时，一次最多读取的字符数。

和 read() 函数一样，此函数成功读取文件数据的前提是，使用 open() 函数指定打开文件的模式必须为 r（只读模式）或 r+（读写模式）。

下面程序示范了使用 readline() 方法来读取文件内容：

```
#a.txt 中有两行内容，分别是：
#C 语言中文网
#http://c.biancheng.net
f = open("a.txt", 'r', True)
while True:
        # 每次读取一行
        line = f.readline()
        # 如果没有读到数据，跳出循环
        if not line: break
        # 输出 line
        print(line)
f.close()
```

运行结果为：

```
C 语言中文网


http://c.biancheng.net
```

由于 readline() 函数在读取文件中一行的内容时，会读取最后的换行符“\n”，再加上

print() 函数输出内容时默认会换行，所以输出结果中会看到多出了一个空行。

不仅如此，在逐行读取时，还可以限制每次最多读取的字符数，例如：

```
f = open("a.txt", 'r', True)
while True:
    # 每次读取一行
    line = f.readline(3)
    # 如果没有读到数据，跳出循环
    if not line: break
    # 输出 line
    print(line)
f.close()
```

运行结果为：

```
C 语言

中文网



htt

p:/

/c.

bia

nch

eng
```

```
.ne

t
```

readlines()函数

readlines() 函数用于读取文件中的所有行，它和调用不指定 size 参数的 read() 函数类似，只不过该函数返回是一个字符串列表，其中每个元素为文件中的一行内容。

和 readline() 函数一样 ,readlines() 函数在读取每一行时 ,会连同行尾的换行符一块读取。

readlines() 函数的基本语法格式如下：

```
file.readlines()
```

其中，file 为打开的文件对象。和 read()、readline() 函数一样，它要求打开文件的模式使用 r（只读）或者 r+（读写）。

例如如下程序：

```
1.          f = open("a.txt", 'r', True)
2.          lines = f.readlines()
3.          for line in lines:
4.                  print(line)
```

运行结果为：

```
C 语言中文网


http://c.biancheng.net
```

6.Python write()和 writelines()：向文件中写入数据

前面章节中学习了如何使用 read()、readline() 和 readlines() 这 3 个函数读取文件，如果我们想把一些数据保存到文件中，又该如何实现呢？

Python 中的文件对象提供了 write() 函数，可以向文件中写入指定内容。该函数的语法格式如下：

```
file.write(string)
```

其中，file 表示已经打开的文件对象；string 表示要写入文件的字符串（或字节串，仅适用写入二进制文件中）。

注意，在使用 write() 向文件中写入数据，需保证使用 open() 函数是以 r+、w、w+、a 或 a+ 的模式打开文件，否则执行 write() 函数会抛出 io.UnsupportedOperation 错误。

例如，创建一个 a.txt 文件，该文件内容如下：

```
C 语言中文网
http://c.biancheng.net
```

然后，在和 a.txt 文件同级目录下，创建一个 Python 文件，编写如下代码：

```
1.          f = open("a.txt", 'w')
2.          f.write("写入一行新数据")
3.          f.close()
```

前面已经讲过，如果打开文件模式中包含 w（写入），那么向文件中写入内容时，会先清空原文件中的内容，然后再写入新的内容。因此运行上面程序，再次打开 a.txt 文件，只会看到新写入的内容：

```
写入一行新数据
```

而如果打开文件模式中包含 a（追加），则不会清空原有内容，而是将新写入的内容会添加到原内容后边。例如，还原 a.txt 文件中的内容，并修改上面代码为：

```
1.          f = open("a.txt", 'a')
2.          f.write("\n 写入一行新数据")
3.          f.close()
```

再次打开 a.txt，可以看到如下内容：

```
C 语言中文网
http://c.biancheng.net
写入一行新数据
```

因此，采用不同的文件打开模式，会直接影响 write() 函数向文件中写入数据的效果。

另外，在写入文件完成后，一定要调用 close() 函数将打开的文件关闭，否则写入的内容不会保存到文件中。例如，将上面程序中最后一行 f.close() 删掉，再次运行此程序并打开 a.txt，你会发现该文件是空的。这是因为，当我们在写入文件内容时，操作系统不会立刻把数据写入磁盘，而是先缓存起来，只有调用 close() 函数时，操作系统才会保证把没有写入的数据全部写入磁盘文件中。

除此之外，如果向文件写入数据后，不想马上关闭文件，也可以调用文件对象提供的 flush() 函数，它可以实现将缓冲区的数据写入文件中。例如：

```
1.          f = open("a.txt", 'w')
2.          f.write("写入一行新数据")
3.          f.flush()
```

打开 a.txt 文件，可以看到写入的新内容：

```
写入一行新数据
```

有读者可能会想到，通过设置 open() 函数的 buffering 参数可以关闭缓冲区，这样数据不就可以直接写入文件中了？对于以二进制格式打开的文件，可以不使用缓冲区，写入的数据会直接进入磁盘文件；但对于以文本格式打开的文件，必须使用缓冲区，否则 Python 解释器会 ValueError 错误。例如：

```
1.          f = open("a.txt", 'w',buffering = 0)
2.          f.write("写入一行新数据")
```

运行结果为：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 1, in <module>
    f = open("a.txt", 'w',buffering = 0)
ValueError: can't have unbuffered text I/O
```

Python writelines()函数

Python 的文件对象中，不仅提供了 write() 函数，还提供了 writelines() 函数，可以实现将字符串列表写入文件中。

注意，写入函数只有 write() 和 writelines() 函数，而没有名为 writeline 的函数。

例如，还是以 a.txt 文件为例，通过使用 writelines() 函数，可以轻松实现将 a.txt 文件中的数据复制到其它文件中，实现代码如下：

```
1.          f = open('a.txt', 'r')
2.          n = open('b.txt','w+')
3.          n.writelines(f.readlines())
4.          n.close()
5.          f.close()
```

执行此代码，在 a.txt 文件同级目录下会生成一个 b.txt 文件，且该文件中包含的数据和 a.txt 完全一样。

需要注意的是，使用 writelines() 函数向文件中写入多行数据时，不会自动给各行添加换行符。上面例子中，之所以 b.txt 文件中会逐行显示数据，是因为 readlines() 函数在读取各行数据时，读入了行尾的换行符。

7.Python close()函数：关闭文件

前面章节中，已经多次使用了 close() 函数，它用来关闭使用 open() 函数打开的文件。

close() 函数的语法格式如下：

```
file.close()
```

其中，file 表示已打开的文件对象。

文件在打开并操作完成之后，应及时关闭，否则会给程序带来很多无法预知的错误。例如，对一个没有关闭的文件做删除操作会失败：

```
1.          import os
2.          f = open("a.txt",'w')
3.          f.write("C 语言中文网")
4.          os.remove("a.txt")
```

此代码中使用了 os 模块中的 remove() 函数，其作用就是删除指定的文件，但由于代码中没有使用 close() 函数关闭 a.txt，因此删除操作会失败，运行结果为：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 4, in <module>
    os.remove("a.txt")
PermissionError: [WinError 32] 另一个程序正在使用此文件，进程无法访问。: 'a.txt'
```

不仅如此，当我们想用 write() 函数将数据写入到 a.txt 文件中，如果不使用 close() 函数关闭文件，会导致数据无法成功写入到文件中。例如：

```
1.          f = open("a.txt",'w')
2.          f.write("C 语言中文网")
```

运行程序结束后，打开 a.txt 文件，你会发现文件是空的。这是因为，在向以文本格式（而不是二进制格式）打开的文件中写入数据时，Python 出于效率的考虑，会先将数据临时存储到缓冲区中，只有使用 close() 函数关闭文件时，才会将缓冲区中的数据真正写入文件中。

因此，在上面程序的最后添加如下代码：

```
1.          f.close()
```

再次运行程序，就会看到 "C 语言中文网" 成功写入到了 a.txt 文件。

> 当然，如果想在不关闭文件的前提下将数据成功写入到文件中，可以使用文件对象提供的 flush() 函数。

8.Python seek 和 tell

在讲解 seek() 函数和 tell() 函数之前，首先来了解一下什么是文件指针。

我们知道，使用 open() 函数打开文件并读取文件中的内容时，总是会从文件的第一个字符（字节）开始读起。那么，有没有办法可以自定指定读取的起始位置呢？答案是肯定，这就需要移动文件指针的位置。

文件指针用于标明文件读写的起始位置。假如把文件看成一个水流，文件中每个数据（以 b 模式打开，每个数据就是一个字节；以普通模式打开，每个数据就是一个字符）就相当于一

个水滴，而文件指针就标明了文件将要从文件的哪个位置开始读起。图 1 简单示意了文件指针的概念。

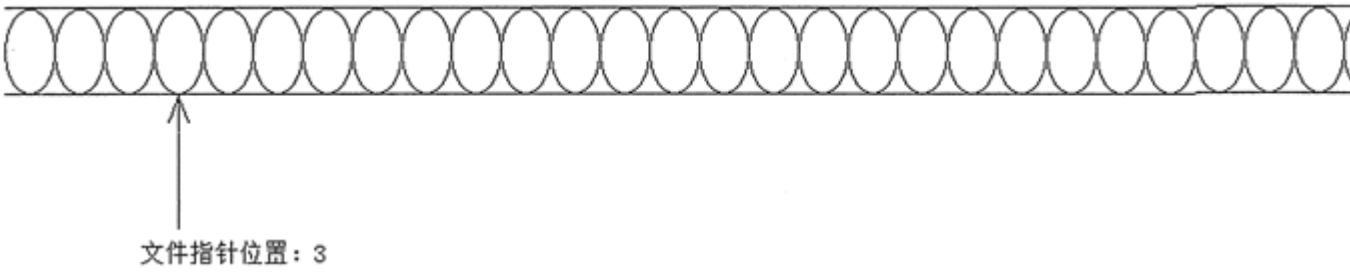


图 1 文件指针概念示意图

可以看到，通过移动文件指针的位置，再借助 read() 和 write() 函数，就可以轻松实现，读取文件中指定位置的数据（或者向文件中的指定位置写入数据）。

注意，当向文件中写入数据时，如果不是文件的尾部，写入位置的原有数据不会自行向后移动，新写入的数据会将文件中处于该位置的数据直接覆盖掉。

实现对文件指针的移动，文件对象提供了 tell() 函数和 seek() 函数。tell() 函数用于判断文件指针当前所处的位置，而 seek() 函数用于移动文件指针到文件的指定位置。

tell() 函数

tell() 函数的用法很简单，其基本语法格式如下：

```
file.tell()
```

其中，file 表示文件对象。

例如，在同一目录下，编写如下程序对 a.txt 文件做读取操作，a.txt 文件中内容为：

```
http://c.biancheng.net
```

读取 a.txt 的代码如下：

```
f = open("a.txt",'r')
print(f.tell())
print(f.read(3))
print(f.tell())
```

运行结果为：

```
0
htt
3
```

可以看到，当使用 open() 函数打开文件时，文件指针的起始位置为 0，表示位于文件的开头处，当使用 read() 函数从文件中读取 3 个字符之后，文件指针同时向后移动了 3 个字符的位置。这就表明，当程序使用文件对象读写数据时，文件指针会自动向后移动：读写了多少个数据，文件指针就自动向后移动多少个位置。

seek()函数

seek() 函数用于将文件指针移动至指定位置，该函数的语法格式如下：

```
file.seek(offset[, whence])
```

其中，各个参数的含义如下：

- file：表示文件对象；
- whence :作为可选参数 ,用于指定文件指针要放置的位置 ,该参数的参数值有 3 个选择：0 代表文件头（默认值）、1 代表当前位置、2 代表文件尾。
- offset：表示相对于 whence 位置文件指针的偏移量，正数表示向后偏移，负数表示向前偏移。例如，当 `whence == 0 &&offset == 3`（即 seek(3,0) ），表示文件指针移动至距离文件开头处 3 个字符的位置；当 `whence == 1 &&offset == 5`（即 seek(5,1) ），表示文件指针向后移动，移动至距离当前位置 5 个字符处。

> 注意，当 offset 值非 0 时，Python 要求文件必须要以二进制格式打开，否则会抛出 io.UnsupportedOperation 错误。

下面程序示范了文件指针操作：

```
f = open('a.txt', 'rb')
# 判断文件指针的位置
print(f.tell())
# 读取一个字节，文件指针自动后移 1 个数据
print(f.read(1))
print(f.tell())
# 将文件指针从文件开头，向后移动到 5 个字符的位置
f.seek(5)
print(f.tell())
print(f.read(1))
# 将文件指针从当前位置，向后移动到 5 个字符的位置
f.seek(5, 1)
print(f.tell())
print(f.read(1))
# 将文件指针从文件结尾，向前移动到距离 10 个字符的位置
f.seek(-1, 2)
print(f.tell())
print(f.read(1))
```

运行结果为：

```
0
b'h'
1
5
b'/'
11
b'a'
21
b't'
```

注意：由于程序中使用 seek() 时，使用了非 0 的偏移量，因此文件的打开方式中必须包含 b，否则就会报 io.UnsupportedOperation 错误，有兴趣的读者可自定尝试。

上面程序示范了使用 seek() 方法来移动文件指针，包括从文件开头、指针当前位置、文件结尾处开始计算。运行上面程序，结合程序输出结果可以体会文件指针移动的效果。

9.Python with as 用法详解

任何一门编程语言中，文件的输入输出、数据库的连接断开等，都是很常见的资源管理操作。但资源都是有限的，在写程序时，必须保证这些资源在使用过后得到释放，不然就容易造成资源泄露，轻者使得系统处理缓慢，严重时会使系统崩溃。

例如，前面在介绍文件操作时，一直强调打开的文件最后一定要关闭，否则会程序的运行造

成意想不到的隐患。但是，即便使用 close() 做好了关闭文件的操作，如果在打开文件或文件操作过程中抛出了异常，还是无法及时关闭文件。

为了更好地避免此类问题，不同的编程语言都引入了不同的机制。在 Python 中，对应的解决方式是使用 with as 语句操作上下文管理器（context manager），它能够帮助我们自动分配并且释放资源。

简单的理解，同时包含 __enter__() 和 __exit__() 方法的对象就是上下文管理器。常见构建上下文管理器的方式有 2 种，分别是基于类实现和基于生成器实现，在《什么是上下文管理器，深入底层了解 with as 语句》一文有详细介绍。

例如，使用 with as 操作已经打开的文件对象（本身就是上下文管理器），无论期间是否抛出异常，都能保证 with as 语句执行完毕后自动关闭已经打开的文件。

首先学习如何使用 with as 语句。with as 语句的基本语法格式为：

```
with 表达式 [as target]：
    代码块
```

此格式中，用 [] 括起来的部分可以使用，也可以省略。其中，target 参数用于指定一个变量，该语句会将 expression 指定的结果保存到该变量中。with as 语句中的代码块如果不想执行任何语句，可以直接使用 pass 语句代替。

举个例子，假设有一个 a.txt 文件，其存储内容如下：

C 语言中文网

http://c.biancheng.net

在和 a.txt 同级目录下，创建一个 .py 文件，并编写如下代码：

纯文本复制

```
with open('a.txt', 'a') as f:
    f.write("\nPython 教程")
```

运行结果为：

C 语言中文网

http://c.biancheng.net

Python 教程

可以看到，通过使用 with as 语句，即便最终没有关闭文件，修改文件内容的操作也能成功。

with as 语句实现的底层原理到底是怎样的呢？可以阅读《什么是上下文管理器，深入底层了解 with as 语句》一文做详细了解。

10.什么是上下文管理器，深入底层了解 Python with as 语句

在介绍 with as 语句时讲到，该语句操作的对象必须是上下文管理器。那么，到底什么是上下文管理器呢？

简单的理解，同时包含 __enter__() 和 __exit__() 方法的对象就是上下文管理器。也就是说，上下文管理器必须实现如下两个方法：

1. __enter__(self)：进入上下文管理器自动调用的方法，该方法会在 with as 代码块执行之前执行。如果 with 语句有 as 子句，那么该方法的返回值会被赋值给 as 子句后的变量；该方法可以返回多个值，因此在 as 子句后面也可以指定多个变量（多个变量必须由“()”括起来组成元组）。
2. __exit__（self, exc_type, exc_value, exc_traceback）：退出上下文管理器自动调用的方法。该方法会在 with as 代码块执行之后执行。如果 with as 代码块成功执行结束，程序自动调用该方法，调用该方法的三个参数都为 None：如果 with as 代码块因为异常而中止，程序也自动调用该方法，使用 sys.exc_info 得到的异常信息将作为调用该方法的参数。

当 with as 操作上下文管理器时，就会在执行语句体之前，先执行上下文管理器的 __enter__() 方法，然后再执行语句体，最后执行 __exit__() 方法。

构建上下文管理器，常见的有 2 种方式：基于类实现和基于生成器实现。

基于类的上下文管理器

通过上面的介绍不难发现，只要一个类实现了 __enter__() 和 __exit__() 这 2 个方法，程序就可以使用 with as 语句来管理它，通过 __exit__() 方法的参数，即可判断出 with 代码块执行时是否遇到了异常。其实，上面程序中的文件对象也实现了这两个方法，因此可以

接受 with as 语句的管理。

下面我们自定义一个实现上下文管理协议的类，并尝试用 with as 语句来管理它：

```
class FkResource:
    def __init__(self, tag):
        self.tag = tag
        print('构造器,初始化资源: %s' % tag)
    # 定义__enter__方法，with 体之前的执行的方法
    def __enter__(self):
        print('[__enter__ %s]: ' % self.tag)
        # 该返回值将作为 as 子句中变量的值
        return 'fkit'  # 可以返回任意类型的值
    # 定义__exit__方法，with 体之后的执行的方法
    def __exit__(self, exc_type, exc_value, exc_traceback):
        print('[__exit__ %s]: ' % self.tag)
        # exc_traceback 为 None，代表没有异常
        if exc_traceback is None:
            print('没有异常时关闭资源')
        else:
            print('遇到异常时关闭资源')
            return False   # 可以省略，默认返回 None 也被看做是 False
with FkResource('孙悟空') as dr:
    print(dr)
    print('[with 代码块] 没有异常')
print('------------------------------')
with FkResource('白骨精'):
    print('[with 代码块] 异常之前的代码')
    raise Exception
    print('[with 代码块] ~~~~~~~~异常之后的代码')
```

运行上面的程序，可以看到如下输出结果：

```
构造器,初始化资源: 孙悟空
[__enter__ 孙悟空]:
fkit
[with 代码块] 没有异常
[__exit__ 孙悟空]:
没有异常时关闭资源
------------------------------
构造器,初始化资源: 白骨精
[__enter__ 白骨精]:
[with 代码块] 异常之前的代码
[__exit__ 白骨精]:
遇到异常时关闭资源
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 26, in <module>
    raise Exception
Exception
```

上面程序定义了一个 FkResource 类，并包含了 __enter__() 和 __exit__() 两个方法，因此该类的对象可以被 with as 语句管理。

此外，程序中两次使用 with as 语句管理 FkResource 对象。第一次代码块没有出现异常，第二次代码块出现了异常。从上面的输出结果来看，使用 with as 语句管理资源，无论代

码块是否有异常，程序总可以自动执行 __exit__() 方法。

> 注意，当出现异常时，如果 __exit__ 返回 False（默认不写返回值时，即为 False），则会重新抛出异常，让 with as 之外的语句逻辑来处理异常；反之，如果返回 True，则忽略异常，不再对异常进行处理。

基于生成器的上下文管理器

除了基于类的上下文管理器，它还可以基于生成器实现。接下来先看一个例子。比如，我们可以使用装饰器 contextlib.contextmanager，来定义自己所需的基于生成器的上下文管理器，用以支持 with as 语句：

```
from contextlib import contextmanager

@contextmanager
def file_manager(name, mode):
    try:
        f = open(name, mode)
        yield f
    finally:
        f.close()

with file_manager('a.txt', 'w') as f:
    f.write('hello world')
```

这段代码中，函数 file_manager() 就是一个生成器，当我们执行 with as 语句时，便会打开文件，并返回文件对象 f；当 with 语句执行完后，finally 中的关闭文件操作便会执行。另外可以看到，使用基于生成器的上下文管理器时，不再用定义 __enter__() 和 __exit__() 方法，但需要加上装饰器 @contextmanager，这一点新手很容易疏忽。

需要强调的是，基于类的上下文管理器和基于生成器的上下文管理器，这两者在功能上是一致的。只不过，基于类的上下文管理器更加灵活，适用于大型的系统开发，而基于生成器的上下文管理器更加方便、简洁，适用于中小型程序。但是，无论使用哪一种，不用忘记在方法“__exit__()”或者是 finally 块中释放资源，这一点尤其重要。

11.Python fileinput 模块：逐行读取多个文件

fileinput 模块提供了 input 函数，可以把多个输入流合并在一起，该函数的语法格式如下：

```
fileinput.input( files="filename1, filename2, ...", inplace=False, backup='', bufsize=0, mode='r', openhook=None )
```

此函数会返回一个 FileInput 对象，其中，各个参数的含义如下：

- files：多个文件的路径列表；
- inplace：用于指定是否将标准输出的结果写回到文件，此参数默认值为 False；
- backup：用于指定备份文件的扩展名；
- bufsize：指定缓冲区的大小，默认为 0；
- mode：打开文件的格式，默认为 r（只读格式）；
- openhook：控制文件的打开方式，例如编码格式等。

当程序使用上面函数创建了 FileInput 对象之后，即可通过 for 循环来遍历文件的每一行。此外，fileinput 还提供了很多全局函数来判断正在读取的文件信息，如表 1 所示。

表 1 fileinput 模块常用函数

函数名	功能描述
fileinput.filename()	返回正在读取的文件的文件名。
fileinput.fileno()	返回当前文件的文件描述符（file descriptor），该文件描述符是一个整数。
fileinput.lineno()	返回当前读取的行号。
fileinput.filelineno()	返回当前读取的行在其文件中的行号。
fileinput.isfirstline()	返回当前读取的行在其文件中是否为第一行。
fileinput.isstdin()	返回最后一行是否从 sys.stdin 读取。程序可以使用“-”代表从 sys.stdin 读取。
fileinput.nextfile()	关闭当前文件，开始读取下一个文件。
fileinput.close()	关闭 FileInput 对象。

文件描述符是一个文件的代号，其值为一个整数。后续章节将会介绍关于文件描述符的操作。

通过上面的介绍不难发现，fileinput 也存在一个缺陷，即在创建 FileInput 对象时不能指定字符集，因此它所读取的文件的字符集必须与操作系统默认的字符集保持一致。当然，如果文本文件的内容是纯英文，则不存在字符集的问题。

假设，目前有 a.txt 和 b.txt 文件，它们位于同一目录，并各自包含如下内容：

#a.txt 包含如下内容：

C 语言中文网

http://c.biancheng.net

#b.txt 包含如下内容：

Python 基础教程

http://c.biancheng.net/python

下面程序示范了使用 fileinput 模块来读取以上 2 个文件：

```
import fileinput
# 一次读取多个文件
for line in fileinput.input(files=('a.txt', 'b.txt')):
    # 输出文件名，当前行在当前文件中的行号
    print(fileinput.filename(), fileinput.filelineno(), line)
# 关闭文件流
fileinput.close()
```

上面程序使用 fileinput.input 直接合并了 a.txt 和 b.txt 两个文件，这样程序可以直接遍历读取这两个文件的内容。程序运行结果如下：

a.txt 1 C 语言中文网

a.txt 2 http://c.biancheng.net

b.txt 1 Python 基础教程

```
b.txt 2 http://c.biancheng.net/python
```

12.Python linecache 模块用法：随机读取文件指定行

linecache 模块允许从 Python 源文件中随机读取指定行，并在内部使用缓存优化存储。由于该模块主要被设计成读取 Python 源文件，因此它会用 UTF-8 字符集来读取文本文件。

实际上，使用 linecache 模块也可以读取其他文件，只要该文件使用了 UTF-8 字符集存储。

linecache 模块包含以下常用函数：

- linecache.getline(filename, lineno, module_globals=None)：读取指定模块中指定文件的指定行。其中 filename 指定文件名，lineno 指定行号。
- linecache.clearcache()：清空缓存。
- linecache.checkcache(filename=None)：检查缓存是否有效。如果没有指定 filename 参数，则默认检查所有缓存的数据。

下面程序示范了使用 linecache 模块来随机读取指定行：

纯文本复制

```
import linecache
import random
```

```
3.
4.          # 读取 random 模块的源文件的第 3 行
5.          print(linecache.getline(random.__file__, 3))
6.          # 读取本程序的第 3 行
7.          print(linecache.getline('linecache_test.py', 3))
8.          # 读取普通文件的第 2 行
9.          print(linecache.getline('utf_text.txt', 2))
```

上面程序示范了使用 linecache 模块随机读取指定模块源文件、Python 源程序、普通文件的指定行。运行程序后，即可看到使用 linecache 模块读取指定行的效果。

13.Python pathlib 模块

pathlib 模块提供了一组面向对象的类，这些类可代表各种操作系统上的路径，程序可通过这些类操作路径。pathlib 模块下的类如图 1 所示。

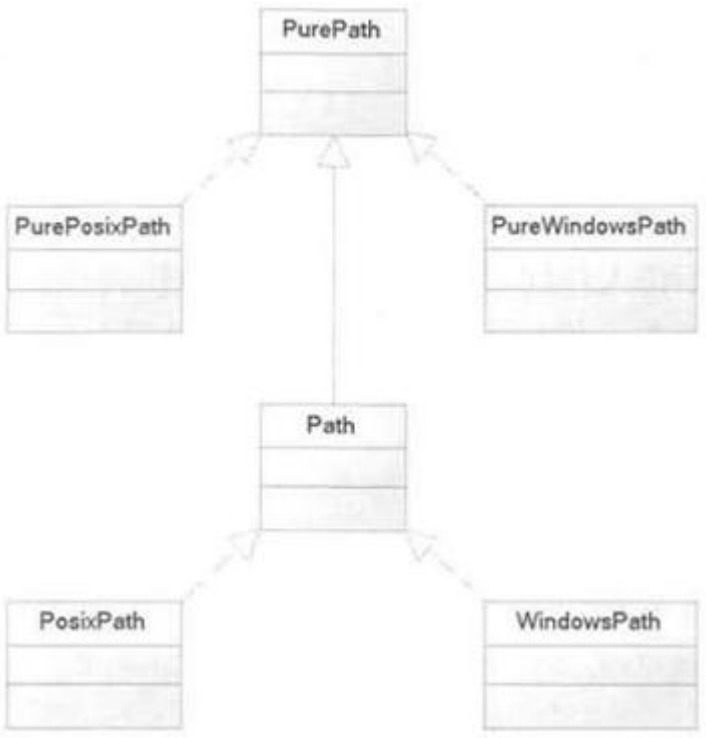


图 1 pathlib 模块下的类

PurePath 代表并不访问实际文件系统的“纯路径”。简单来说，PurePath 只是负责对路径字符串执行操作，至于该字符串是否对应实际的路径，它并不关心。

PurePath 有两个子类，即 PurePosixPath 和 PureWindowsPath，分别代表 UNIX 风格的路径（包括 Mac OS X）和 Windows 风格的路径。

> UNIX 风格的路径和 Windows 风格路径的主要区别在于根路径和路径分隔符，UNIX 风格路径的根路径是斜杠（/），而 Windows 风格路径的根路径是盘符（c:）；UNIX 风格的路径的分隔符是斜杠（/），而 Windows 风格路径的分隔符是反斜杠（\）。

Path 代表访问实际文件系统的“真正路径”。Path 对象可用于判断对应的文件是否存在、是否为文件、是否为目录等。Path 同样有两个子类，即 PosixPath 和 WindowsPath。

PurePath 类的用法

程序可使用 PurePath 或它的两个子类来创建 PurePath 对象，如果在 UNIX 或 Mac OS X 系统上使用 PurePath 创建对象，程序实际返回 PurePosixPath 对象；如果在 Windows 系统上使用 PurePath 创建对象，程序实际返回 PureWindowsPath 对象。

> 如果程序明确希望创建 PurePosixPath 或 PureWindowsPath 对象，则应该直接使用 PurePath 的子类。

程序在创建 PurePath 和 Path 时，既可传入单个路径字符串，也可传入多个路径字符串，

PurePath 会将它们拼接成一个字符串。例如如下程序：

```
from pathlib import *
# 创建 PurePath，实际上使用 PureWindowsPath
pp = PurePath('setup.py')
print(type(pp))  # <class 'pathlib.PureWindowsPath'>
pp = PurePath('crazyit', 'some/path', 'info')
# 看到输出 Windows 风格的路径
print(pp) # 'crazyit\some\path\info'
pp = PurePath(Path('crazyit'), Path('info'))
# 看到输出 Windows 风格的路径
print(pp) # 'crazyit\info'
# 明确指定创建 PurePosixPath
pp = PurePosixPath('crazyit', 'some/path', 'info')
# 看到输出 Unix 风格的路径
print(pp) # crazyit/some/path/info
```

如果在创建 PurePath 时不传入任何参数，系统默认创建代表当前路径的 PurePath，相当于传入点号（代表当前路径）作为参数。例如如下代码：

```
# 如果不传入参数，默认使用当前路径
pp = PurePath()
print(pp) # .
```

如果在创建 PurePath 时传入的参数包含多个根路径，则只有最后一个根路径及后面的子路径生效。例如如下代码：

```
# 如果传入参数包含多个根路径，则只有最后一个根路径及后面子路径生效
pp = PurePosixPath('/etc', '/usr', 'lib64')
print(pp) # /usr/lib64
pp = PureWindowsPath('c:/Windows', 'd:info')
print(pp) # d:info
```

需要说明的是，在 Windows 风格的路径中，只有盘符才能算根路径，仅有斜杠是不算的。

例如如下代码：

```
# 在 Windows 风格路径中，只有盘符才算根路径
pp = PureWindowsPath('c:/Windows', '/Program Files')
print(pp) # c:\Program Files
```

如果在创建 PurePath 时传入的路径字相串中包含多余的斜杠和点号，系统会直接忽略它们。但不会忽略两点，因为两点在路径中有实际意义（两点代表上一级路径）。例如如下代码：

```
# 路径字符串中多出来的斜杠和点号（代表当前路径）都会被忽略
pp = PurePath('crazyit//info')
print(pp) # crazyit\info
pp = PurePath('crazyit/./info')
print(pp) # crazyit\info
pp = PurePath('crazyit/../info')
print(pp) # crazyit\..\info，相当于找和 crazyit 同一级的 info 路径
```

PurePath 对象支持各种比较运算符，它们既可比较是否相等，也可比较大小（实际上就是比较它们的路径字符串）。

> PurePath 只是代表特定平台的路径字符串，读者可以把它们看做包装后的字符串（它们本质上就是字符串）。

下面程序示范了 PurePath 对象的比较运算：

```
from pathlib import *
# 比较两个 Unix 风格的路径，区分大小写
print(PurePosixPath('info') == PurePosixPath('INFO')) # False
# 比较两个 Windows 风格的路径，不区分大小写
```

```
print(PureWindowsPath('info') == PureWindowsPath('INFO')) # True
# Windows 风格的路径不区分大小写
print(PureWindowsPath('INFO') in { PureWindowsPath('info') })#True
# Unix 风格的路径区分大小写,所以'D:'小于'c:'
print(PurePosixPath('D:') < PurePosixPath('c:')) # True
# Windows 风格的路径不区分大小写,所以'd:'（D:）大于'c:'
print(PureWindowsPath('D:') > PureWindowsPath('c:')) #True
```

对于不同风格的 PurePath，它们依然可以比较是否相等（结果总是返回 False），但不能比较大小，否则会引发错误。例如如下代码：

```
# 不同风格的路径可以判断是否相等（总不相等）
print(PureWindowsPath('crazyit') == PurePosixPath('crazyit')) # False
# 不同风格的路径不能判断大小，否则会引发异常
#print(PureWindowsPath('info') < PurePosixPath('info')) # TypeError
```

PurePath 对象支持斜杠（/）作为运算符，该运算符的作用是将多个路径连接起来。不管是 UNIX 风格的路径，还是 Windows 风格的路径，都是使用斜杠作为连接运算符的：

```
pp = PureWindowsPath('abc')
# 将多个路径拼起来（Windows 风格的路径）
print(pp / 'xyz' / 'wawa') # abc\xyz\wawa
pp = PurePosixPath('abc')
# 将多个路径拼起来（Unix 风格的路径）
print(pp / 'xyz' / 'wawa') # abc/xyz/wawa
pp2 = PurePosixPath('haha', 'hehe')
# 将 pp、pp2 两个路径连接起来
print(pp / pp2) # abc/haha/hehe
```

正如从上面程序中代码所看到的，程序将使用斜杠来连接多个 Windows 路径，连接完成后可以看到 Windows 路径的分隔符依然是反斜杠。

PurePath 的本质其实就是字符串，因此程序可使用 str() 将它们恢复成字符串对象。在恢

复成字符串对象时会转换为对应平台风格的字符串。例如如下代码：

```
        pp = PureWindowsPath('abc', 'xyz', 'wawa')
        print(str(pp)) # abc\xyz\wawa
        pp = PurePosixPath('abc', 'xyz', 'wawa')
        print(str(pp)) # abc/xyz/wawa
```

上面程序在创建 PurePath 时传入的参数完全相同，但在创建 PureWindowsPath 对象后路径字符串使用反斜杠作为分隔符；在创建 PurePosixPath 对象后路径字符串使用斜杠作为分隔符。

PurePath 类的属性和方法

PurePath 提供了不少属性和方法（如表 1 所示），这些属性和方法主要还是用于操作路径字符串。由于 PurePath 并不真正执行底层的文件操作，也不理会路径字符串在底层是否有对应的路径，因此这些操作有点类似于字符串方法。

表 1 PurePath 类属性和方法

类属性和方法名	功能描述
PurePath.parts	该属性返回路径字符串中所包含的各部分。
PurePath.drive	该属性返回路径字符串中的驱动器盘符。
PurePath.root	该属性返回路径字符串中的根路径。
PurePath.anchor	该属性返回路径字符串中的盘符和根路径。
PurePath.parents	该属性返回当前路径的全部父路径。

PurPath.parent	该属性返回当前路径的上一级路径，相当于 parents[0] 的返回值。
PurePath.name	该属性返回当前路径中的文件名。
PurePath.suffixes	该属性返回当前路径中的文件所有后缀名。
PurePath.suffix	该属性返回当前路径中的文件后缀名。相当于 suffixes 属性返回的列表的最后一个元素。
PurePath.stem	该属性返回当前路径中的主文件名。
PurePath.as_posix()	将当前路径转换成 UNIX 风格的路径。
PurePath.as_uri()	将当前路径转换成 URI。只有绝对路径才能转换，否则将会引发 ValueError。
PurePath.is_absolute()	判断当前路径是否为绝对路径。
PurePath.joinpath(*other)	将多个路径连接在一起，作用类似于前面介绍的斜杠运算符。
PurePath.match(pattern)	判断当前路径是否匹配指定通配符。
PurePath.relative_to(*other)	获取当前路径中去除基准路径之后的结果。
PurePath.with_name(name)	将当前路径中的文件名替换成新文件名。如果当前路径中没有文件名，则会引发 ValueError。
PurePath.with_suffix(suffix)	将当前路径中的文件后缀名替换成新的后缀名。如果当前路

	径中没有后缀名，则会添加新的后缀名。

下面程序大致测试了上面属性和方法的用法：

```
from pathlib import *

# 访问 drive 属性
print(PureWindowsPath('c:/Program Files/').drive) # c:
print(PureWindowsPath('/Program Files/').drive) # ''
print(PurePosixPath('/etc').drive) # ''

# 访问 root 属性
print(PureWindowsPath('c:/Program Files/').root) # \
print(PureWindowsPath('c:Program Files/').root) # ''
print(PurePosixPath('/etc').root) # /

# 访问 anchor 属性
print(PureWindowsPath('c:/Program Files/').anchor) # c:\
print(PureWindowsPath('c:Program Files/').anchor) # c:
print(PurePosixPath('/etc').anchor) # /

# 访问 parents 属性
pp = PurePath('abc/xyz/wawa/haha')
print(pp.parents[0]) # abc\xyz\wawa
print(pp.parents[1]) # abc\xyz
print(pp.parents[2]) # abc
print(pp.parents[3]) # .
# 访问 parent 属性
print(pp.parent) # abc\xyz\wawa

# 访问 name 属性
print(pp.name) # haha
pp = PurePath('abc/wawa/bb.txt')
print(pp.name) # bb.txt
```

```
31.
32.         pp = PurePath('abc/wawa/bb.txt.tar.zip')
33.         # 访问 suffixes 属性
34.         print(pp.suffixes[0]) # .txt
35.         print(pp.suffixes[1]) # .tar
36.         print(pp.suffixes[2]) # .zip
37.         # 访问 suffix 属性
38.         print(pp.suffix) # .zip
39.         print(pp.stem) # bb.txt.tar
40.
41.         pp = PurePath('abc', 'xyz', 'wawa', 'haha')
42.         print(pp) # abc\xyz\wawa\haha
43.         # 转成 Unix 风格的路径
44.         print(pp.as_posix()) # abc/xyz/wawa/haha
45.         # 将相对路径转换成 Uri 引发异常
46.         #print(pp.as_uri()) # ValueError
47.         # 创建绝对路径
48.         pp = PurePath('d:/', 'Python', 'Python3.6')
49.         # 将绝对路径转换成 Uri
50.         print(pp.as_uri()) # file:///d:/Python/Python3.6
51.
52.         # 判断当前路径是否匹配指定模式
53.         print(PurePath('a/b.py').match('*.py')) # True
54.         print(PurePath('/a/b/c.py').match('b/*.py')) # True
55.         print(PurePath('/a/b/c.py').match('a/*.py')) # False
56.
57.         pp = PurePosixPath('c:/abc/xyz/wawa')
58.         # 测试 relative_to 方法
59.         print(pp.relative_to('c:/'))  # abc\xyz\wawa
60.         print(pp.relative_to('c:/abc'))  # xyz\wawa
61.         print(pp.relative_to('c:/abc/xyz'))  # wawa
62.
63.         # 测试 with_name 方法
64.         p = PureWindowsPath('e:/Downloads/pathlib.tar.gz')
65.         print(p.with_name('fkit.py')) # e:\Downloads\fkit.py
66.         p = PureWindowsPath('c:/')
```

```
67.         #print(p.with_name('fkit.py')) # ValueError
68.
69.         # 测试 with_suffix 方法
70.         p = PureWindowsPath('e:/Downloads/pathlib.tar.gz')
71.         print(p.with_suffix('.zip'))  # e:\Downloads\pathlib.tar.zip
72.         p = PureWindowsPath('README')
73.         print(p.with_suffix('.txt'))  # README.txt
```

上面程序在测试每个方法后都给出了对应的输出结果，读者可结合程序的输出结果来理解 PurePath 各属性和方法的功能。

Path 类的功能和用法

Path 是 PurePath 的子类，除了支持 PurePath 的各种操作、属性和方法之外，还会真正访问底层的文件系统，包括判断 Path 对应的路径是否存在，获取 Path 对应路径的各种属性（如是否只读、是文件还是文件夹等），甚至可以对文件进行读写。

PurePath 和 Path 最根本的区别在于，PurePath 处理的仅是字符串，而 Path 则会真正访问底层的文件路径，因此它提供了属性和方法来访问底层的文件系统。

关于 Path 的大量属性和方法本章不再详细列出，读者可参考 https://docs.python.org/3/library/pathlib.html 进行查阅。

Path 同样提供了两个子类，PosixPath 和 WindowsPath，其中前者代表 UNIX 风格的路径，后者代表 Windows 风格的路径。

Path 对象包含了大量 is_xxx() 方法，用于判断该 Path 对应的路径是否为 xxx。Path 包

含一个 exists() 方法，用于判断该 Path 对应的目录是否存在。

Path 还包含一个很常用的 iterdir() 方法，该方法可返回 Path 对应目录下的所有子目录和文件。此外，Path 还包含一个 glob() 方法，用于获取 Path 对应目录及其子目录下匹配指定模式的所有文件，借助此方法可以非常方便地查找指定文件。

下面程序示范了 Path 的简单用法：

```
1.          from pathlib import *
2.
3.          # 获取当前目录
4.          p = Path('.')
5.          # 遍历当前目录下所有文件和子目录
6.          for x in p.iterdir():
7.              print(x)
8.
9.          # 获取上一级目录
10.         p = Path('../')
11.         # 获取上级目录及其所有子目录下的的 py 文件
12.         for x in p.glob('**/*.py'):
13.             print(x)
14.
15.         # 获取 g:/publish/codes 对应的目录
16.         p = Path('g:/publish/codes')
17.         # 获取上级目录及其所有子目录下的的 py 文件
18.         for x in p.glob('**/Path_test1.py'):
19.             print(x)
```

上面程序中第 6 行代码调用了 Path 的 iterdir() 方法，该方法将会返回当前目录下的所有文件和子目录；第 12 行代码调用了 glob() 方法，获取上一级目录及其所有子目录下的

*.py 文件；第 16 行代码用于获取 g:/publish/codes 目录及其所有子目录下的 Path_test1.py 文件。

运行上面程序，可以看到如下输出结果：

```
Path_test1.py
PurePath_test1.py
PurePath_test2.py
PurePath_test3.py
..\12.1\Path_test1.py
..\12.1\PurePath_test1.py
..\12.1\PurePath_test2.py
..\12.1\PurePath_test3.py
g:\publish\codes\12\12.1\path_test1.py
```

从上面的输出结果来看，不管是遍历当前目录下的文件和子目录，还是搜索指定目录及其子目录，Path 对象都能用一个方法搞定，对于不少语言来说，Path 的 glob() 方法所实现的功能，其他语言往往要通过递归才能实现，这可能就是 Python 的魅力所在。

此外，Path 还提供了 read_bytes() 和 read_text(encoding=None, errors=None) 方法，分别用于读取该 Path 对应文件的字节数据（二进制数据）和文本数据；也提供了 write_bytes(data) 和 Path.write_text(data, encoding=None, errors=None) 方法来输出字节数据（二进制数据）和文本数据。

下面程序示范了使用 Path 来读写文件：

纯文本复制

```
from pathlib import *
p = Path('a_test.txt')
# 以 GBK 字符集输出文本内容
result = p.write_text('''有一个美丽的新世界
它在远方等我
那里有天真的孩子
还有姑娘的酒窝''',  encoding='GBK')
# 返回输出的字符数
print(result)

# 指定以 GBK 字符集读取文本内容
content = p.read_text(encoding='GBK')
# 输出读取的文本内容
print(content)

# 读取字节内容
bb = p.read_bytes()
print(bb)
```

上面程序中第 4 行代码使用 GBK 字符集调用 write_text() 方法输出字符串内容，该方法将会返回实际输出的字符个数；第 12 行代码使用 GBK 字符集读取文件的字符串内容，该方法将会返回整个文件的内容，也就是刚刚输出的内容。

14.Python os.path 模块常见函数用法（实例+详细注释）

os.path 模块下提供了一些操作目录的方法，这些函数可以操作系统的目录本身。例如，该模块提供了 exists() 函数判断该目录是否存在，也提供了 getctime()、getmtime()、getatime() 函数来获取该目录的创建时间、最后一次修改时间、最后一次访问时间，还提

供了 getsize() 函数来获取指定文件的大小。

下面程序示范了 os.path 模块下的操作目录的常见函数的功能和用法：

纯文本复制

```
import os
import time

# 获取绝对路径
print(os.path.abspath("abc.txt")) # G:\publish\codes\12\12.2\abc.txt
# 获取共同前缀
print(os.path.commonprefix(['/usr/lib', '/usr/local/lib'])) # /usr/l
# 获取共同路径
print(os.path.commonpath(['/usr/lib', '/usr/local/lib'])) # \usr
# 获取目录
print(os.path.dirname('abc/xyz/README.txt')) #abc/xyz
# 判断指定目录是否存在
print(os.path.exists('abc/xyz/README.txt')) # False
# 获取最近一次访问时间
print(time.ctime(os.path.getatime('os.path_test.py')))
# 获取最后一次修改时间
print(time.ctime(os.path.getmtime('os.path_test.py')))
# 获取创建时间
print(time.ctime(os.path.getctime('os.path_test.py')))
# 获取文件大小
print(os.path.getsize('os.path_test.py'))
# 判断是否为文件
print(os.path.isfile('os.path_test.py')) # True
# 判断是否为目录
print(os.path.isdir('os.path_test.py')) # False
# 判断是否为同一个文件
print(os.path.samefile('os.path_test.py', './os.path_test.py')) # True
```

运行上面程序，大部分函数的输出结果都通过注释给出了，程序中 getatime()、

getmtime()、getctime() 三个函数分别获取了文件的最后一次访问时间、最后一次修改时间和创建时间。读者可通过运行该程序来理解 os.path 模块下这些函数的功能。

15.Pythonfnmatch 模块：用于文件名的匹配

前面介绍的那些操作目录的函数只能进行简单的模式匹配 ,但 fnmatch 模块可以支持类似于 UNIX shell 风格的文件名匹配。

fnmatch 匹配支持如下通配符：

- *：可匹配任意个任意字符。
- ？：可匹配一个任意字符。
- [字符序列]：可匹配中括号里字符序列中的任意字符。该字符序列也支持中画线表示法。比如[a-c]可代表 a、b 和 c 字符中任意一个。
- [!字符序列]：可匹配不在中括号里字符序列中的任意字符。

在该模块下提供了如下函数：

- fnmatch.fnmatch(filename, pattern)：判断指定文件名是否匹配指定 pattern。

如下程序示范了 fnmatch() 函数的用法：

```
from pathlib import *
import fnmatch

# 遍历当前目录下所有文件和子目录
for file in Path('.').iterdir():
    # 访问所有以_test.py 结尾的文件
```

```
7.                 if fnmatch.fnmatch(file, '*_test.PY'):
8.                     print(file)
```

上面程序先遍历当前目录下的所有文件和子目录，然后第 7 行代码调用 fnmatch() 函数判断所有以 _test.py 结尾的文件，并将该文件打印出来。

- fnmatch.fnmatchcase(filename, pattern) :该函数与上一个函数的功能大致相同，只是该函数区分大小写。
- fnmatch.filter(names, pattern)：该函数对 names 列表进行过滤，返回 names 列表中匹配 pattern 的文件名组成的子集合。如下代码示范了该函数的功能：

```
1.          names = ['a.py', 'b.py', 'c.py', 'd.py']
2.          # 对 names 列表进行过滤
3.          sub = fnmatch.filter(names, '[ac].py')
4.          print(sub) # ['a.py', 'c.py']
```

上面程序定义了一个 ['a.py', 'b.py', 'c.py', 'd.py'] 集合，该集合中的 4 个元素都代表了指定文件（实际上文件是否存在，fnmatch 模块并不关心）。接下来程序中第 3 行代码调用了 filter() 函数对 names 进行过滤，过滤完成后只保留匹配 [ac].py 模式的文件名（要求文件名要么是 a.py，要么是 c.py）。

- fnmatch.translate(pattern)：该函数用于将一个 UNIX shell 风格的 pattern 转换为正则表达式 pattern。如下代码示范了该函数的功能：

```
1.          print(fnmatch.translate('?.py')) # (?s:.\.py)\Z
2.          print(fnmatch.translate('[ac].py')) # (?s:[ac]\.py)\Z
3.          print(fnmatch.translate('[a-c].py')) # (?s:[a-c]\.py)\Z
```

16.Python os 模块

除前面章节介绍的各种函数之外，os 模块还提供了大量操作文件和目录的函数，本节将介绍 os 模块下常用的函数。

> 如果读者需要查阅有关这些函数的说明，则可访问 https://docs.python.org/3/library/os.html 页面。

os 模块与目录相关的函数

与目录相关的函数如下：

- os.getcwd()：获取当前目录。
- os.chdir(path)：改变当前目录。
- os.fchdir(fd)：通过文件描述利改变当前目录。该函数与上一个函数的功能基本相似，只是该函数以文件描述符作为参数来代表目录。

下面程序测试了与目录相关的函数的用法：

```
import os

# 获取当前目录
print(os.getcwd())  # G:\publish\codes\12.7
# 改变当前目录
os.chdir('../12.6')
# 再次获取当前目录
print(os.getcwd())  # G:\publish\codes\12.6
```

上面程序示范了使用 getcwd() 来获取当前目录，也示范了使用 chdir() 来改变当前目录。

- os.chroot(path)：改变当前进程的根目录。
- os.listdir(path)：返回 path 对应目录下的所有文件和子目录。
- os.mkdir(path[, mode])：创建 path 对应的目录，其中 mode 用于指定该目录的权限。该 mode 参数代表一个 UNIX 风格的权限，比如 0o777 代表所有者可读/可写/可执行、组用户可读/可写/可执行、其他用户可读/可写/可执行。
- os.makedirs(path[, mode])：其作用类似于 mkdir()，但该函数的功能更加强大，它可以边归创建目录。比如要创建 abc/xyz/wawa 目录，如果在当前目录下没有 abc 目录，那么使用 mkdir() 函数就会报错，而使用 makedirs() 函数则会先创建 abc，然后在其中创建 xyz 子目录，最后在 xyz 子目录下创建 wawa 子目录。

如下程序示范了如何创建目录：

```
import os
path = 'my_dir'
# 直接在当前目录下创建目录
os.mkdir(path, 0o755)
path = "abc/xyz/wawa"
# 递归创建目录
os.makedirs(path, 0o755)
```

正如从上面代码所看到的，直接在当前目录下创建 mydir 子目录，因此可以使用 mkdir() 函数创建；需要程序递归创建 abc/xyz/wawa 目录，因此使用 makedirs() 函数。

- os.rmdir(path)：删除 path 对应的空目录。如果目录非空，则抛出一个 OSError 异常。程序可以先用 os.remove() 函数删除文件。
- os.removedirs(path)：边归删除目录。其功能类似于 rmdir()，但该函数可以递归删除 abc/xyz/wawa 目录，它会从 wawa 子目录开始删除，然后删除 xyz 子目录，最后

删除 abc 目录。

如下程序示范了如何删除目录：

```
1.          import os
2.
3.          path = 'my_dir'
4.          # 直接删除当前目录下的子目录
5.          os.rmdir(path)
6.          path = "abc/xyz/wawa"
7.          # 递归删除子目录
8.          os.removedirs(path)
```

上面程序中第 5 行代码使用 rmdir() 函数删除当前目录下的 my_dir 子目录，该函数不会执行递归删除；第 8 行代码使用 removedirs() 函数删除 abc/xyz/wawa 目录，该函数会执行递归删除，它会先删除 wawa 子目录，然后删除 xyz 子目录，最后才删除 abc 目录。

- os.rename(src, dst)：重命名文件或目录，将 src 重名为 dst。
- os.renames(old, new)：对文件或目录进行递归重命名。其功能类似于 rename()，但该函数可以递归重命名 abc/xyz/wawa 目录，它会从 wawa 子目录开始重命名，然后重命名 xyz 子目录，最后重命名 abc 目录。

如下程序示范了如何重命名目录：

```
1.          import os
2.
3.          path = 'my_dir'
4.          # 直接重命名当前目录下的子目录
5.          os.rename(path, 'your_dir')
```

```
path = "abc/xyz/wawa"
# 递归重命名子目录
os.renames(path, 'foo/bar/haha')
```

上面程序中第 5 行代码直接重命名当前目录下的 my_dir 子目录，程序会将该子目录重命名为 your_dir；第 8 行代码则执行递归重命名，程序会将 wawa 重命名为 haba，将 xyz 重命名为 bar，将 abc 重命名为 foo。

os 模块与权限相关的函数

与权限相关的函数如下：

- os.access(path, mode)：检查 path 对应的文件或目录是否具有指定权限。该函数的第二个参数可能是以下四个状态值的一个或多个值：
- os.F_OK：判断是否存在。
- os.R_OK：判断是否可读。
- os.W_OK：判断是否可写。
- os.X_OK：判断是否可执行。

例如如下程序：

```
import os

# 判断当前目录的权限
ret = os.access('.', os.F_OK|os.R_OK|os.W_OK|os.X_OK)
print("os.F_OK|os.R_OK|os.W_OK|os.X_OK - 返回值:", ret)
# 判断 os.access_test.py 文件的权限
ret = os.access('os.access_test.py', os.F_OK|os.R_OK|os.W_OK)
```

```
12.         print("os.F_OK|os.R_OK|os.W_OK - 返回值:", ret)
```

上面程序判断当前目录的权限和当前文件的权限，这里特意将此文件设为只读的。运行该程序，可以看到如下输出结果：

```
os.F_OK|os.R_OK|os.W_OK|os.X_OK - 返回值：True
os.F_OK|os.R_OK|os.W_OK - 返回值：False
```

- os.chrnod(path, mode)：更改权限。其中 mode 参数代表要改变的权限，该参数支持的值可以是以下一个或多个值的组合：
- stat.S_IXOTH：其他用户有执行权限。
- stat.S_IWOTH：其他用户有写权限。
- stat.S_TROTH：其他用户有读权限。
- stat.S_IRWXO：其他用户有全部权限。
- stat.S_IXGRP：组用户有执行权限。
- stat.S_IWGRP：组用户有写权限。
- stat.S_IRGRP：组用户有读权限。
- stat.S_IRWXG：组用户有全部权限。
- stat.S_IXUSR：所有者有执行权限。
- stat.S_IWUSR：所有者有写权限。
- stat.S_IRUSR：所有者有读权限。
- stat.S_IRWXU：所有者有全部权限。
- stat.S_IREAD：Windows 将该文件设为只读的。
- stat.S_IWRITE：Windows 将该文件设为可写的。

> 前面的那些权限都是 UNIX 文件系统下有效的概念，UNIX 文件系统下的文件有一个所有者，跟所有者处于同一组的其他用户被称为组用户。因此在 UNIX 文件系统下允许为不同用户分配不同的权限。

例如如下程序：

```
14.         import os, stat
15.
16.         # 将 os.chmod_test.py 文件改为只读
17.         os.chmod('os.chmod_test.py', stat.S_IREAD)
18.         # 判断是否可写
19.         ret = os.access('os.chmod_test.py', os.W_OK)
20.         print("os.W_OK - 返回值:", ret)
```

运行上面程序后，os.chmod_test.py 变成只读文件。

- os.chown(path, uid, gid)：更改文件的所有者。其中 uid 代表用户 id，gid 代表组 id。该命令主要在 UNIX 文件系统下有效。
- os.fchmod(fd, mode)：改变一个文件的访问权限，该文件由文件描述符 fd 指定。该函数的功能与 os.chmod() 函数的功能相似，只是该函数使用 fd 代表文件。
- os.fchown(fd, uid, gid)：改变文件的所有者，该文件由文件描述符 fd 指定。该函数的功能与 os.chown() 函数的功能相似，只是该函数使用 fd 代表文件。

os 模块与文件访问相关的函数

与文件访问相关的函数如下：

- os.open(file, flags[, mode])：打开一个文件，并且设置打开选项，mode 参数是可选的。该函数返回文件描述符。其中 flags 代表打开文件的旗标，它支持如下一个或多个选项：
  - os.O_RDONLY：以只读的方式打开。
  - os.O_WRONLY：以只写的方式打开。
  - os.O_RDWR：以读写的方式打开。
  - os.O_NONBLOCK：打开时不阻塞。
  - os.O_APPEND：以追加的方式打开。
  - os.O_CREAT：创建并打开一个新文件。
  - os.O_TRUNC：打开一个文件并截断它的长度为 0（必须有写权限）。
  - os.O_EXCL：在创建文件时，如果指定的文件存在，则返回错误。
  - os.O_SHLOCK：自动获取共享锁。
  - os.O_EXLOCK：自动获取独立锁。
  - os.O_DIRECT：消除或减少缓存效果。
  - os.O_FSYNC：同步写入。
  - os.O_NOFOLLOW：不追踪软链接。
- os.read(fd, n)：从文件描述符 fd 中读取最多 n 个字节，返回读到的字符串。如果文件描述符副对应的文件已到达结尾，则返回一个空字节串。
- os.write(fd, str)：将字节串写入文件描述符 fd 中，返回实际写入的字节串长度。
- os.close(fd)：关闭文件描述符 fd。
- os.lseek(fd, pos, how)：该函数同样用于移动文件指针。其中 how 参数指定从哪里开始移动，如果将 how 设为 0 或 SEEK_SET，则表明从文件开头开始移动；如果将

how 设为 1 或 SEEK_CUR，则表明从文件指针当前位置开始移动；如果将 how 设为 2 或 SEEK_END，则表明从文件结束处开始移动。上面几个函数同样可用于执行文件的读写，程序通常会先通过 os.open() 打开文件，然后调用 os.read()、os.write() 来读写文件，当操作完成后通过 os.close() 关闭文件。

如下程序示范了使用上面的函数来读写文件：

```
1.  import os
2.
3.  # 以读写、创建方式打开文件
4.  f = os.open('abc.txt', os.O_RDWR|os.O_CREAT)
5.  # 写入文件内容
6.  len1 = os.write(f, '水晶潭底银鱼跃，\n'.encode('utf-8'))
7.  len2 = os.write(f, '清徐风中碧竿横。\n'.encode('utf-8'))
8.  # 将文件指针移动到开始处
9.  os.lseek(f, 0, os.SEEK_SET)
10. # 读取文件内容
11. data = os.read(f, len1 + len2)
12. # 打印读取到字节串
13. print(data)
14. # 将字节串恢复成字符串
15. print(data.decode('utf-8'))
16. os.close(f)
```

上面程序中，第 6、7 行代码用于向所打开的文件中写入数据；第 11 行代码用于读取文件内容。

- os.fdopen(fd[, mode[, bufsize]])：通过文件描述符 fd 打开文件，并返回对应的文件对象。

- os.closerange(fd_low, fd_high)：关闭从 fd_low（包含）到 fd_high（不包含）范围的所有文件描述符。
- os.dup(fd)：复制文件描述符。
- os.dup2(fd,fd2)：将一个文件描述符 fd 复制到另一个文件描述符 fd2 中。
- os.ftruncate(fd, length)：将 fd 对应的文件截断到 length 长度，因此此处传入的 length 参数不应该超过文件大小。
- os.remove(path)：删除 path 对应的文件。如果 path 是一个文件夹，则抛出 OSError 错误。如果要删除目录，则使用 os.rmdir()。
- os.link(src, dst)：创建从 src 到 dst 的硬链接。硬链接是 UNIX 系统的概念，如果在 Windows 系统中就是复制目标文件。
- os.symlink(src, dst)：创建从 src 到 dst 的符号链接，对应于 Windows 的快捷方式。

> 由于 Windows 权限的缘故，因此必须以管理员身份执行 os.symlink() 函数来创建快捷方式。

下面程序示范了在 Windows 系统中使用 os.symlink(src, dst) 函数来创建快捷方式：

```
import os

# 为 os.link_test.py 文件创建快捷方式
os.symlink('os.link_test.py', 'tt')
# 为 os.link_test.py 文件创建硬连接（Windows 上就是复制文件）
os.link('os.link_test.py', 'dst')
```

上面程序使用 symlink() 函数为指定文件创建符号链接，在 Windows 系统中就是创建快

捷方式；使用 link() 函数创建硬链接，在 Windows 系统中就是复制文件。

运行上面程序，将会看到程序在当前目录下创建了一个名为 “tt” 的快捷方式，并将 os.link_test.py 文件复制为 dst 文件。

17.Python tempfile 模块：生成临时文件和临时目录

tempfile 模块专门用于创建临时文件和临时目录，它既可以在 UNIX 平台上运行良好，也可以在 Windows 平台上运行良好。

tempfile 模块中常用的函数，如表 1 所示。

表 1 tempfile 模块常用函数及功能

tempfile 模块函数	功能描述
tempfile.TemporaryFile(mode='w+b', buffering=None, encoding=None, newline=None, suffix=None, prefix=None, dir=None)	创建临时文件。该函数返回一个类文件对象，也就是支持文件 I/O。
tempfile.NamedTemporaryFile(mode='w+b', buffering=None, encoding=None, newline=None, suffix=None, prefix=None, dir=None, delete=True)	创建临时文件。该函数的功能与上一个函数的功能大致相同，只是它生成的临时文件在文件系统中有文件名。
tempfile.SpooledTemporaryFile(max_size=0,	创建临时文件。与

mode='w+b', buffering=None, encoding=None, newline=None, suffix=None, prefix=None, dir=None)	TemporaryFile 函数相比，当程序向该临时文件输出数据时，会先输出到内存中，直到超过 max_size 才会真正输出到物理磁盘中。
tempfile.TemporaryDirectory(suffix=None, prefix=None, dir=None)	生成临时目录。
tempfile.gettempdir()	获取系统的临时目录。
tempfile.gettempdirb()	与 gettempdir() 相同，只是该函数返回字节串。
tempfile.gettempprefix()	返回用于生成临时文件的前缀名。
tempfile.gettempprefixb()	与 gettempprefix() 相同，只是该函数返回字节串。

提示：表中有些函数包含很多参数，但这些参数都具有自己的默认值，因此如果没有特殊要求，可以不对其传参。

tempfile 模块还提供了 tempfile.mkstemp() 和 tempfile.mkdtemp() 两个低级别的函数。上面介绍的 4 个用于创建临时文件和临时目录的函数都是高级别的函数，高级别的函数支持自动清理，而且可以与 with 语句一起使用，而这两个低级别的函数则不支持，因此一般推荐使用高级别的函数来创建临时文件和临时目录。

此外，tempfile 模块还提供了 tempfile.tempdir 属性，通过对该属性赋值可以改变系统的临时目录。

下面程序示范了如何使用临时文件和临时目录：

纯文本复制

```
import tempfile

# 创建临时文件
fp = tempfile.TemporaryFile()
print(fp.name)
fp.write('两情若是久长时，'.encode('utf-8'))
fp.write('又岂在朝朝暮暮。'.encode('utf-8'))
# 将文件指针移到开始处，准备读取文件
fp.seek(0)
print(fp.read().decode('utf-8')) # 输出刚才写入的内容
# 关闭文件，该文件将会被自动删除
fp.close()

# 通过 with 语句创建临时文件，with 会自动关闭临时文件
with tempfile.TemporaryFile() as fp:
    # 写入内容
    fp.write(b'I Love Python!')
    # 将文件指针移到开始处，准备读取文件
    fp.seek(0)
    # 读取文件内容
    print(fp.read()) # b'I Love Python!'

# 通过 with 语句创建临时目录
with tempfile.TemporaryDirectory() as tmpdirname:
    print('创建临时目录', tmpdirname)
```

上面程序以两种方式来创建临时文件：

1. 第一种方式是手动创建临时文件，读写临时文件后需要主动关闭它，当程序关闭该临时文件时，该文件会被自动删除。
2. 第二种方式则是使用 with 语句创建临时文件，这样 with 语句会自动关闭临时文件。

上面程序最后还创建了临时目录。由于程序使用 with 语句来管理临时目录，因此程序也会自动删除该临时目录。

运行上面程序，可以看到如下输出结果：

```
C:\Users\admin\AppData\Local\Temp\tmphvehw9z1
两情若是久长时，又岂在朝朝暮暮。
b'I Love Python!'
创建临时目录 C:\Users\admin\AppData\Local\Temp\tmp3sjbnwob
```

上面第一行输出结果就是程序生成的临时文件的文件名，最后一行输出结果就是程序生成的临时目录的目录名。需要注意的是，不要去找临时文件或临时文件夹，因为程序退出时该临时文件和临时文件夹都会被删除。

第 15 章 Python 数据库编程

数据库适用于保存数据量巨大且具有复杂关系的数据，Python 为操作不同的数据库提供了不同的模块，但各位读者不用担心，这些模块内 API 的设计大同小异，因此掌握 Python 的一个数据库模块之后，再看其他数据库模块时就会有似曾相识的感觉。

为了让读者体会使用 Python 操作不同数据库的相似性，本章会分别介绍如何使用 Python 操作 SQLite 内置数据库和开源的 MySQL 数据库。

Python3.6 默认内置了操作 SQLite 数据库的模块，但如果 Python 程序需要操作 MySQL 数据库，则需要自行下载操作 MySQL 数据库的 Python 模块。

1.Python 数据库 API

支持 SQL 标准的数据库类型有很多（SQListe、MySQL、SQL Server 等），其中多数在 Python 中都有对应的客户端模块（有些数据库甚至对应多个模块），并且所有数据库的绝大多数功能基本上都是相同的，因为它们基本都遵守 Python 制订的 DB API 协议（目前该协议的最新版本是 2.0）。

DB API 是一个规范，它定义了一系列必须的对象和数据库存取方式，以便为各种各样的底层数据库系统和多种多样的数据库接口程序提供一致的访问接口。正式由于 DB API 为不同数据库提供了一致的访问接口，使得在不同数据库之间移植代码变得很轻松。

下面给大家介绍一下这些不同数据库模块之间的通用内容。

全局变量

Python 中，任何支持 2.0 版本 DB API 的数据库模块都必须定义 3 个描述模块特性的全局变量。这样做的原因是 API 设计得很灵活，以支持许多不同的底层机制，避免过多包装。

例如，如果想让程序同时应用于几个数据库，程序中就需要考虑各种可能出现的状况，通常的做法就是检查这些全局变量，判断给定的数据库模块是否能被程序接受，如果不能，则显示合适的提示信息（例如抛出一些异常）并退出。

这 3 个全局变量分别是：

1. apilevel：用于显示数据库模块的 API 版本号。对于支持 DB API 2.0 版本的数据库模块来说，该变量值可能是 1.0 或 2.0。如果这个变量不存在，则可能该数据库模块暂时不支持 2.0 版本的 API，读者就应该考虑选择使用支持该数据库的其他数据库模块。
2. threadsafety：指定数据库模块的线程安全等级，该等级值为 0～3 ，其中 3 代表该模块完全是线程安全的；1 表示该模块具有部分线程安全性，线程可以共享该模块，但不能共享连接；0 则表示线程完全不能共享该模块。
3. paramstyle :该全局变量指定当 SQL 语句需要参数时，可以使用哪种风格的参数。该变量可能返回如下变量值：

- format :表示在 SQL 语句中使用 Python 标准的格式化字符串代表参数。例如，在程序中需要参数的地方使用 %s，接下来程序即可为这些参数指定参数值。

- pyformat：表示在 SQL 语句中使用扩展的格式代码代表参数。比如使用 %(name)，这样即可使用包含 key 为 name 的字典为该参数指定参数值。
- qmark：表示在 SQL 语句中使用问号（？）代表参数。在 SQL 语句中有几个参数，全部用问号代替。
- numeric :表示在 SQL 语句中使用数字占位符(:N)代表参数。例如 :1 代表一个参数，:2 也表示一个参数，这些数字相当于参数名，因此它们不一定需要连续。
- named：表示在 SQL 语句中使用命名占位符（:name）代表参数。例如 :name 代表一个参数，:age 也表示一个参数。

通过查阅这些全局变量，即可大致了解该数据库 API 模块的对外的编程风格，至于该模块内部的实现细节，完全由该模块实现者负责提供，通常不需要开发者关心。

数据库 API 的核心类

遵守 DB API 2.0 协议的数据库模块通常会提供一个 connect() 函数，该函数用于连接数据库，并返回数据库连接对象。

数据库连接对象通常会具有表 1 所示的方法和属性。

表 1 数据库连接对象的方法和属性

方法或属性	功能描述
cursor()	打开游标。

commit()	提交事务。
rollback()	回滚事务。
close()	关闭数据库连接。
isolation_level	返回或设置数据库连接中事务的隔离级别。
in_transaction	判断当前是否处于事务中。

表 1 中，第一个方法可以返回一个游标对象，游标对象是 Python DB API 的核心对象，该对象主要用于执行各种 SQL 语句，包括 DDL、DML、select 查询语句等。使用游标执行不同的 SQL 语句返回不同的数据。

游标对象通常会具有表 2 所示的方法和属性。

表 2 数据库游标对象拥有的属性和方法

属性或方法	**功能描述**
execute(sql[, parameters])	执行 SQL 语句。parameters 参数用于为 SQL 语句中的参数指定值。
executemany(sql, seq_of_parameters)	重复执行 SQL 语句。可以通过 seq_of_parameters 序列为 SQL 语句中的参数指定值，该序列有多少个元素，SQL 语句被执行多少次。

executescript(sql_script)	这不是 DB API 2.0 的标准方法。该方法可以直接执行包含多条 SQL 语句的 SQL 脚本。
fetchone()	获取查询结果集的下一行。如果没有下一行，则返回 None。
fetchmany(size=cursor.arraysize)	返回查询结果集的下 N 行组成的列表。如果没有更多的数据行，则返回空列表。
fetchall()	返回查询结果集的全部行组成的列表。
close()	关闭游标。
rowcount	该只读属性返回受 SQL 语句影响的行数。对于 executemany() 方法，该方法所修改的记录条数也可通过该属性获取。
lastrowid	该只读属性可获取最后修改行的 rowid。
arraysize	用于设置或获取 fetchmany() 默认获取的记录条数，该属性默认为 1。有些数据库模块没有该属性。
description	该只读属性可获取最后一次查询返回的所有列的信息。
connection	该只读属性返回创建游标的数据库连接对象。有些数据库模块没有该属性。

总结来看，Python 的 DB API 2.0 由一个 connect() 开始，一共涉及数据库连接和游标两个核心 API。它们的分工如下：

- 数据库连接：用于获取游标、控制事务。
- 游标：执行各种 SQL 语句。

掌握了上面这些 API 之后，接下来可以大致归纳出 Python DB API 2.0 的编程步骤。

操作数据库的基本流程

使用 Python DB API 2.0 操作数据库的基本流程如下：

1. 调用 connect() 方法打开数据库连接，该方法返回数据库连接对象。
2. 通过数据库连接对象打开游标。
3. 使用游标执行 SQL 语句（包括 DDL、DML、select 查询语句等）。如果执行的是查询语句，则处理查询数据。
4. 关闭游标。
5. 关闭数据库连接。

图 1 显示了使用 Python DB API 2.0 操作数据库的基本流程。

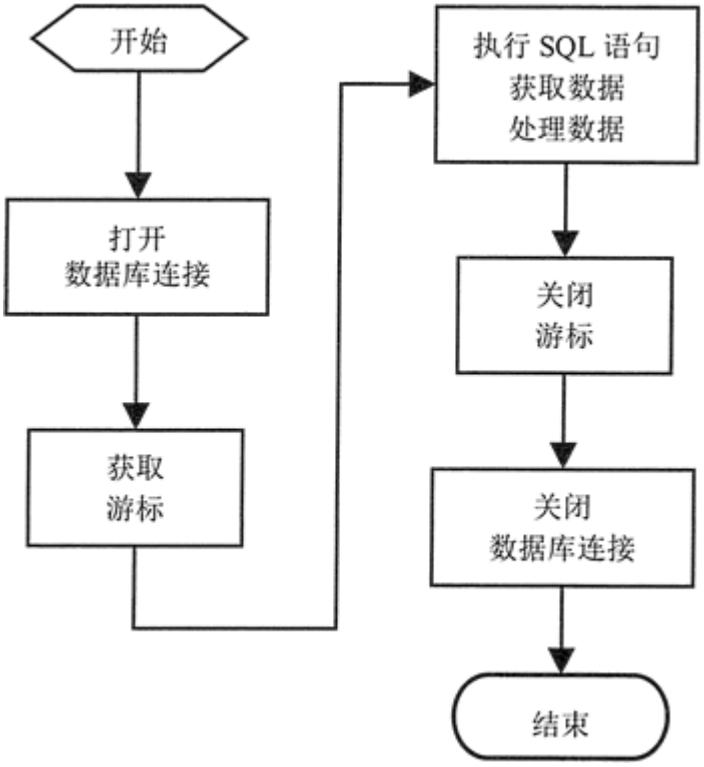


图 1 使用 Python DB API 2.0 操作数据库的基本流程

2.Python SQLite 创建数据表

和很多其他数据库管理系统不同，SQLite 不是一个客户端 / 服务器结构的数据库引擎，而是一种嵌入式数据库，它的数据库其实就是一个文件。SQLite 将整个数据库（包括定义、表、索引以及数据本身）作为一个单独的、可跨平台使用的文件存储在主机中。

由于 SQLite 本身是用 C 写的，体积很小，所以经常被集成到各种应用程序中，Python 就内置了 SQLite 3。因此，在 Python 中使用 SQLite，不需要安装任何模块，使用 import 语句引入即可使用。

我们知道，程序只要通过数据库连接对象打开游标，接下来就可以用游标来执行 DDL 语句

（创建表、修改表或删除表）。那么，如何获取连接对象呢？这就需要使用 connect() 函数。

使用 connect() 函数可以打开或创建一个数据库，例如如下代码：

```
conn = sqlite3.connect('first.db')
```

上面代码就用于打开或创建一个 SQLite 数据库。如果 first.db 文件（该文件就是一个数据库）存在，那么程序就是打开该数据库；反之，如果该文件不存在，则会在当前目录下创建相应的文件。

因此，初次执行上面代码，将会在当前目录下创建一个 first.db 文件，如果程序希望创建内存中的数据库，则只需将 first.db 改为特殊名称 :memory: 即可。

如下程序示范了如何创建数据表：

```
# 导入访问 SQLite 的模块
import sqlite3
# ①、打开或创建数据库
# 也可以使用特殊名：:memory:代表创建内存中的数据库
conn = sqlite3.connect('first.db')
# ②、获取游标
c = conn.cursor()
# ③、执行 DDL 语句创建数据表
c.execute('''create table user_tb(
    _id integer primary key autoincrement,
    name text,
    pass text,
    gender text)''')
```

```
        # 执行 DDL 语句创建数据表
        c.execute('''create table order_tb(
            _id integer primary key autoincrement,
            item_name text,
            item_price real,
            item_number real,
            user_id inteter,
            foreign key(user_id) references user_tb(_id) )''')
        # ④、关闭游标
        c.close()
        # ⑤、关闭连接
        conn.close()
```

上面程序使用 ①~⑤ 清晰地标出了使用 Python DB API 2.0 执行数据库访问的步骤。

上面程序中第 ③ 步执行了两次 ,每次分别执行一条 create 语句 ,因此该程序执行完成后，将会看到在当前数据库中包含两个数据表 user_tb 和 order_tb，且在 order_tb 表中有一个外键列引用 user_tb 表的 _id 主键列。

> 程序中第 ③ 步使用 execute() 方法执行的就是标准的 DDL 语句，因此只要读者拥有 SQL 语法知识，就可以使用 Python DB API 模块来执行这些 SQL 语句，非常简单。

SQLite 数据库所支持的 SQL 语句与 MySQL 大致相同，开发者完全可以把已有的 MySQL 经验“移植”到 SQLite 数据库上。当然，当 Python 程序提示某条 SQL 语句有语法错误时，最好先利用 SQLite 数据库管理工具（后续章节介绍）来测试这条语句，以保证这条 SQL 语句的语法正确。

需要指出的是，SQLite 内部只支持 NULL、INTEGER、REAL（浮点数）、TEXT（文本）

和 BLOB(大二进制对象)这 5 种数据类型 ,但实际上 SQLite 完全可以接受 varchar(n)、char(n)、decimal(p, s) 等数据类型 , 只不过 SQLite 会在运算或保存时将它们转换为上面 5 种数据类型中相应的类型。

除此之外 , SQLite 还有一个特点 , 就是它允许把各种类型的数据保存到任何类型的字段中，开发者可以不用关心声明该字段所使用的数据类型。例如，可以把字符串类型的值存入 INTEGER 类型的字段中，也可以把数值类型的值存入布尔类型的字段中……但有一种情况例外，被定义为“INTEGER PRIMARY KEY”的字段只能存储 64 位整数，当使用这种字段保存除整数以外的其他类型的数据时，SQLite 会产生错误。

由于 SQLite 允许在存入数据时忽略底层数据列实际的数据类型，因此在编写建表语句时可以省略各数据列后面的类型声明。例如 ,对于 SQLite 数据库如下 SQL 语句也是正确的：

```
create table my_test
{
    _id_integer primary key autoincrement,
    name,
    pass,
    gender
};
```

3.SQLite Expert 安装和使用

前面章节中，我们编写程序创建了一个 first.db 数据库（就是一个文件），并在该数据库中创建了两个数据表，那么怎样才能看到它们呢？此时可以通过 SQLite Expert 工具进行查看和管理。

安装 SQLite Expert 工具的步骤如下：

1. 登录 http://www.sqliteexpert.com/download.html 站点来下载 SQLite Expert，该工具提供了两个版本：免费的个人版和收费的商业版。此处选择免费的个人版。将页面滚动到下方，找到“SQLite Expert Personal 5.x”，然后单击下方的链接（64 位操作系统选择 64bit 版，32 位操作系统选择 32bit 版），如图 1 所示。

图 1 下载 SQLite Expert

2. 教程以 64 位操作系统为例，所以下载 SQLiteExpertPersSetup64.exe 文件，下载完成后单击该文件开始安装，其安装过程和安装普通的 Windows 软件完全相同。

3. 安装完成后，启动 SQLite Expert 工具，启动后可以看到如图 2 所示的程序界面。

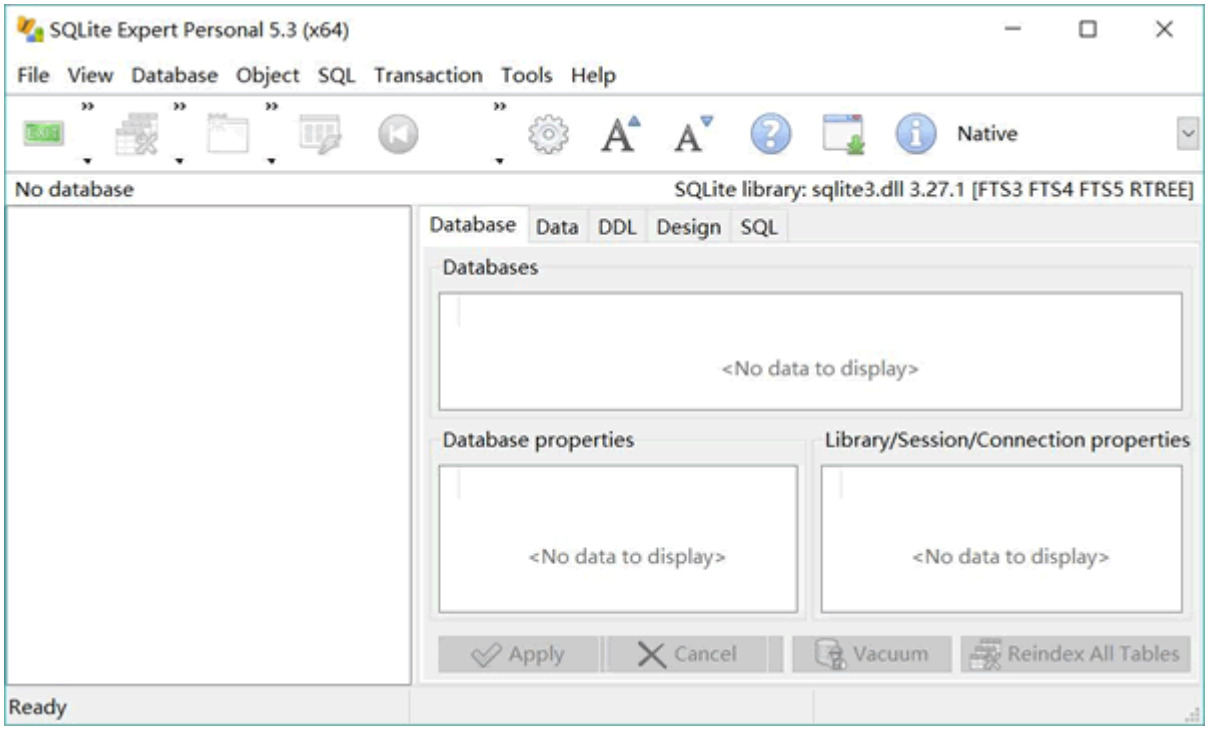


图 2 SQLite Expert 程序界面

在图 2 所示程序界面的左上角工具栏中看到 4 个工具按钮，它们的作用依次是创建数据库、创建内存中的数据库、打开数据库和关闭数据库：

- 如果要使用 SQLite Expert 新建数据库，则单击图 2 所示工具栏中的“New Database”按钮，即可创建一个新的数据库。

- 如果要使用 SQLite Expert 打开已有的数据库文件，则单击图 2 所示工具栏中的“Open Database”按钮。

4. 单击图 2 所示工具栏中的“Open Database”按钮，打开如图 3 所示的浏览数据库文件窗口。

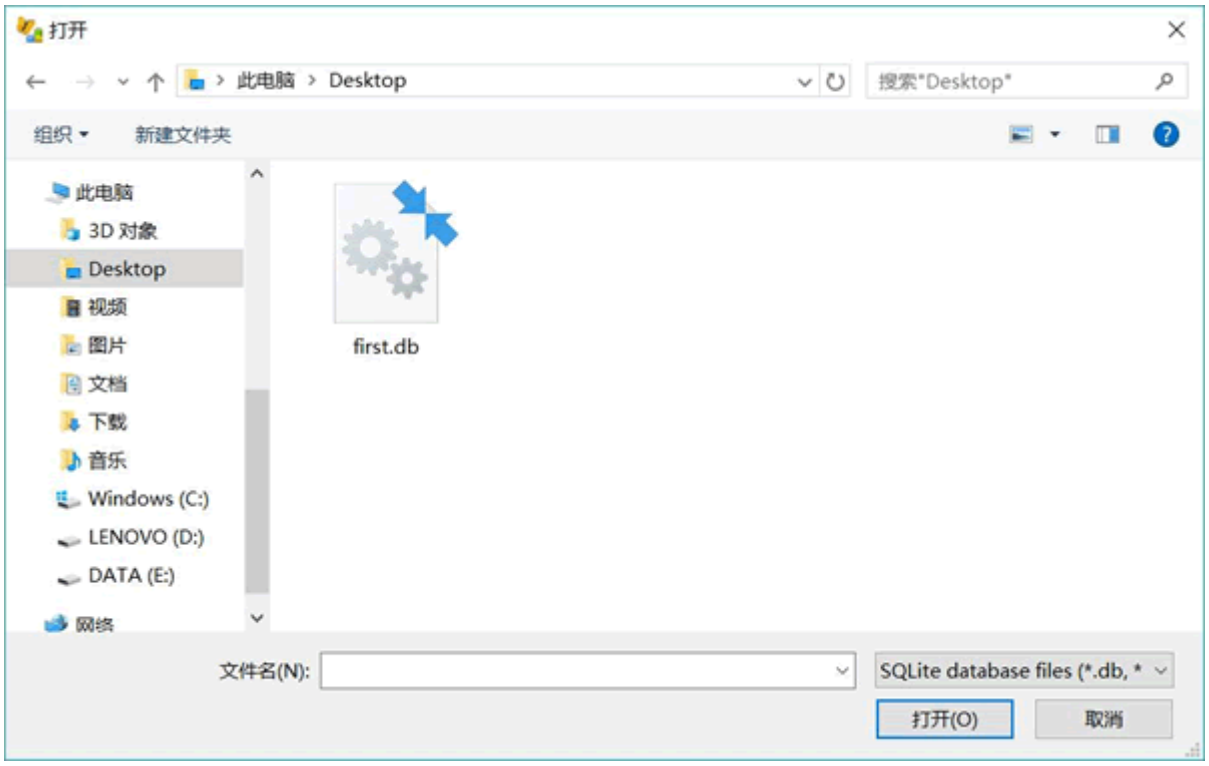


图 3 浏览数据库文件窗口

找到前面程序所创建的 first.db 文件，然后单击“打开”按钮，SQLite Expert 将会打开 first.db 文件所代表的数据库。

5. 打开 first.db 数据库之后，可以在 SQLite Expert 工具中看到该数据库包含两个数据表。随便选中一个数据表，就可以在右边看到该数据表的详细信息，包括数据列（Columns）、主键（Primary Key）、索引（Indexs）、外键（Foreign Keys）、唯一约束（Unique Constraints）等，如图 4 所示。

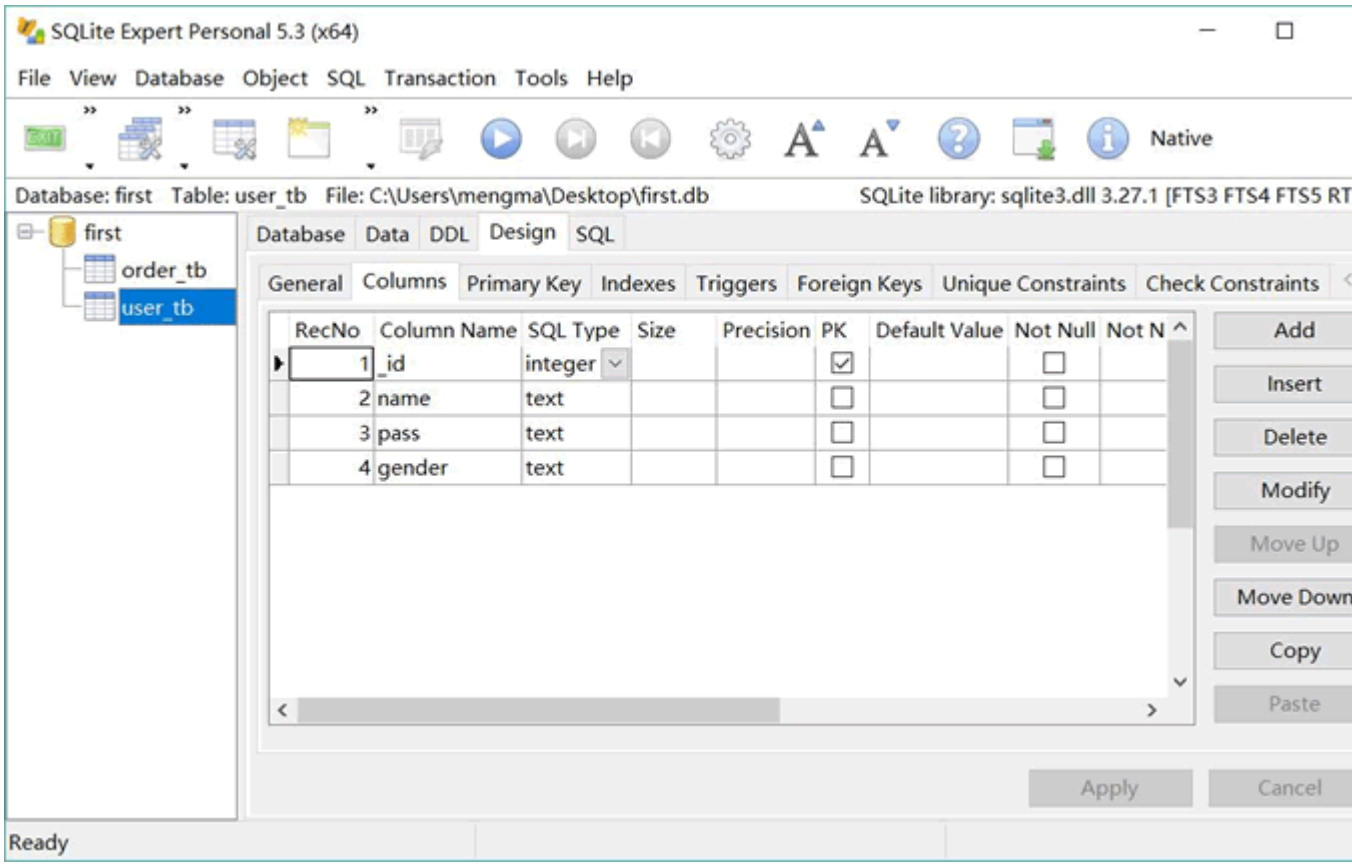


图 4 查看数据表设计

如图 4 所示的界面，就是一个非常方便的数据库管理界面，可以通过该界面执行创建数据表、删除数据表、添加数据、删除数据等操作。

4.Python execute()和 executemany()

使用游标的 execute() 方法可以执行 DML（数据库操纵语言）的 insert、update、delete 语句，即可以对数据库执行插入、修改和删除数据操作。

例如，如下程序示范了向数据库的两个数据表中分别插入一条数据：

```
1.          # 导入访问 SQLite 的模块
2.          import sqlite3
3.
4.          # ①、打开或创建数据库
5.          # 也可以使用特殊名：:memory:代表创建内存中的数据库
6.          conn = sqlite3.connect('first.db')
7.          # ②、获取游标
8.          c = conn.cursor()
9.          # ③、调用执行 insert 语句插入数据
10.         c.execute('insert into user_tb values(null, ?, ?, ?)',
11.             ('孙悟空', '123456', 'male'))
12.         c.execute('insert into order_tb values(null, ?, ?, ?, ?)',
13.             ('鼠标', '34.2', '3', 1))
14.         conn.commit()
15.         # ④、关闭游标
16.         c.close()
17.         # ⑤、关闭连接
18.         conn.close()
```

此程序中，通过调用 execute() 方法执行 insert 语句，即可向数据表中插入数据。

注意，由于 Python 的 SQLite 数据库 API 默认是开启了事务的，因此必须调用上面程序中第 14 行代码来提交事务；否则，程序对数据库所做的修改（包括插入数据、修改数据、删除数据）不会生效。

运行上面程序，将会向 first.db 数据库的两个数据表中各插入一条数据。打开 SQLite Expert，可以看到如图 1 所示的数据。

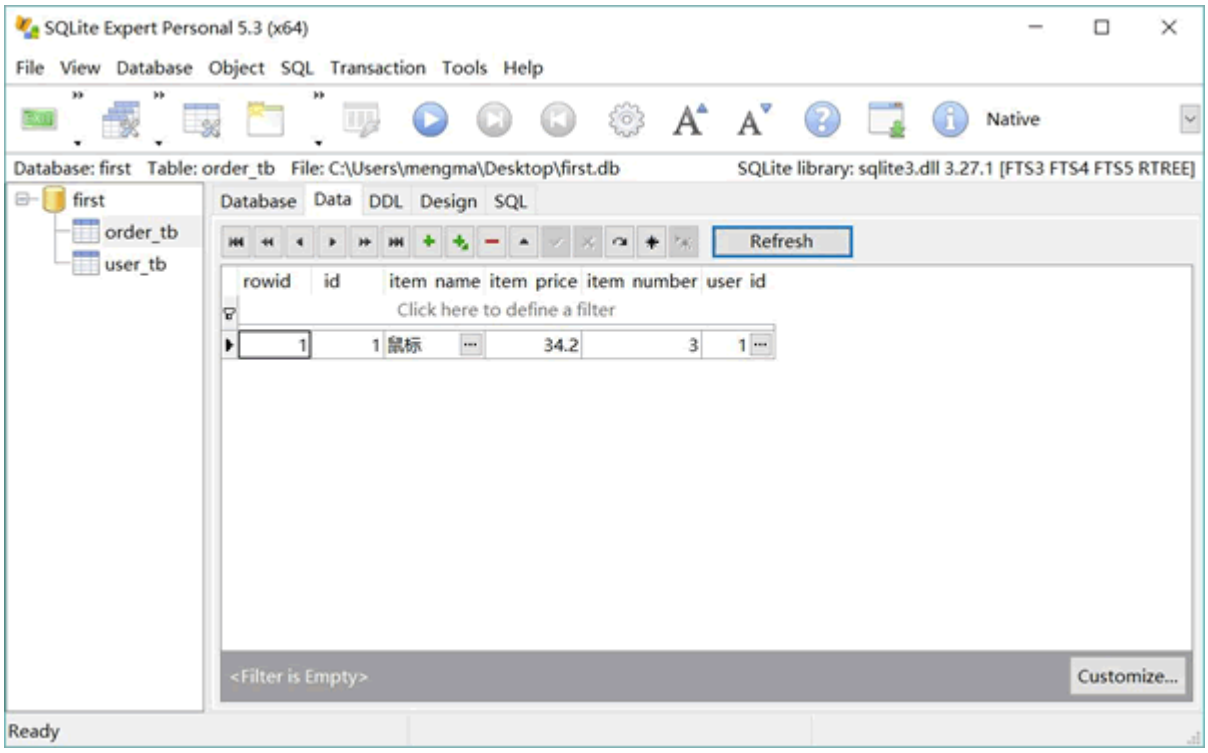


图 1 插入数据

如果程序使用 executemany() 方法，则可以多次执行同一条 SQL 语句。例如如下程序：

```
# 导入访问 SQLite 的模块
import sqlite3

# ①、打开或创建数据库
# 也可以使用特殊名：:memory:代表创建内存中的数据库
conn = sqlite3.connect('first.db')
# ②、获取游标
c = conn.cursor()
# ③、调用 executemany()方法把同一条 SQL 语句执行多次
```

```
        c.executemany('insert into user_tb values(null, ?, ?, ?)',
            (('sun', '123456', 'male'),
            ('bai', '123456', 'female'),
            ('zhu', '123456', 'male'),
            ('niu', '123456', 'male'),
            ('tang', '123456', 'male')))
        conn.commit()
        # ④、关闭游标
        c.close()
        # ⑤、关闭连接
        conn.close()
```

上面代码调用 executemany() 方法执行一条 insert 语句，但调用该方法的第二个参数是一个元组，该元组的每个元素都代表执行该 insert 语句一次，在执行 insert 语句时这些元素负责为该语句中的“?”占位符赋值。

运行上面程序，将向 user_tb 数据表中插入 5 条数据。打开 SQLite Expert，可以看到如图 2 所示的数据。

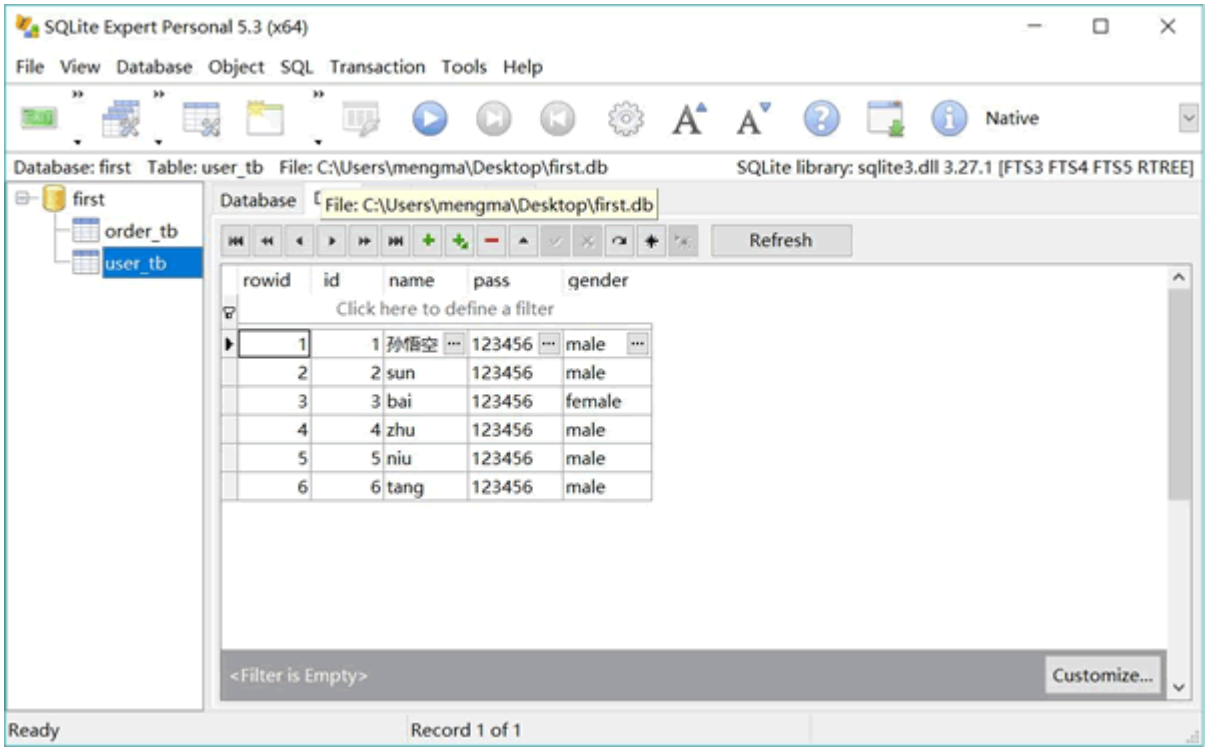


图 2 使用 executemany()重复执行 insert 语句插入 5 条数据

虽然上面程序演示的是使用 executemany() 重复执行 insert 语句，但实际上程序完全可以使用 executemany() 重复执行 update 语句或 delete 语句，只要其第二个参数是一个序列，序列的每个元素都可对被执行 SQL 语句的参数赋值即可。

如下程序示范了如何重复执行 update 语句：

```
纯文本复制
```

```
# 导入访问 SQLite 的模块
import sqlite3

# ①、打开或创建数据库
# 也可以使用特殊名：:memory:代表创建内存中的数据库
```

```
        conn = sqlite3.connect('first.db')
        # ②、获取游标
        c = conn.cursor()
        # ③、调用 executemany()方法把同一条 SQL 语句执行多次
        c.executemany('update user_tb set name=? where _id=?',
            (('小孙', 2),
            ('小白', 3),
            ('小猪', 4),
            ('小牛', 5),
            ('小唐', 6)))
        # 通过 rowcount 获取被修改的记录条数
        print('修改的记录条数：', c.rowcount)
        conn.commit()
        # ④、关闭游标
        c.close()
        # ⑤、关闭连接
        conn.close()
```

正如代码中所看到的，此时使用 executemany() 执行的 update 语句中包含两个参数，因此调用 executemany() 方法的第二个参数是一个元组，该元组中的每个元素只包含两个元素，这两个元素就用于为 update 语句中的两个“?”占位符赋值。

上面程序还使用游标的 rowcount 属性来获取 update 语句所修改的记录条数。运行上面程序，可以看到 user_tb 表中 _id 为 2~6 记录的 name 都被修改了。

5.Python fetchone()、fetchmany()和 fetchall()

执行查询依然按照前面介绍的步骤进行，只是改为执行 select 语句。由于 select 语句执行完成后可以得到查询结果，因此程序可通过游标的 fetchone()、fetchmany(n)、fetchall()来获取查询结果。正如它们的名字所暗示的，fetchone() 用于获取一条记录，fetchmany(n)

用于获取 n 条记录，fetchall() 用于获取全部记录。

如下程序示范了如何执行查询语句，并输出查询结果。

```
# 导入访问 SQLite 的模块
import sqlite3

# ①、打开或创建数据库
# 也可以使用特殊名:memory:代表创建内存中的数据库
conn = sqlite3.connect('first.db')
# ②、获取游标
c = conn.cursor()
# ③、调用执行 select 语句查询数据
c.execute('select * from user_tb where _id > ?', (2,))
print('查询返回的记录数:', c.rowcount)
# 通过游标的 description 属性获取列信息
for col in (c.description):
    print(col[0], end='\t')
print('\n-------------------------------')
while True:
    # 获取一行记录，每行数据都是一个元组
    row = c.fetchone()
    # 如果抓取的 row 为 None，退出循环
    if not row :
        break
    print(row)
    print(row[1] + '-->' + row[2])
# ④、关闭游标
c.close()
# ⑤、关闭连接
conn.close()
```

上面程序使用 execute() 方法执行了一条 select 语句，接下来程序使用循环，通过游标的 description 属性获取查询的列信息，也可以通过游标来获取查询结果，如上面程序所

示。

运行上面程序，可以看到如下运行结果：

```
_id name pass gender
--------------------------------
(3, 'bai', '123456', 'female')
bai-->123456
(4, 'zhu', '123456', 'male')
zhu-->123456
(5, 'niu', '123456', 'male')
niu-->123456
(6, 'tang', '123456', 'male')
tang-->123456
```

从上面的运行结果来看，程序返回了所有 _id 大于 2 的记录，这就是上面程序查询所返回的结果。

由于每条 Select 语句都可能返回多个查询结果，因此不能使用 executemany() 执行查询语句，远没什么意义。

注意，不要试图使用 executemany()方法执行 select 语句，否则程序将会报错：

ProgrammingError:executemany() can only execute DML statements。

上面程序使用 fetchone() 方法每次获取一条记录，这是比较常见的做法。实际上，程序也可以使用 fetchmany(n) 或 fetchall() 方法一次获取多条记录。

例如，可将上面程序中第 12~23 行代码改为如下形式：

```
        # 通过游标的 description 属性获取列信息
        for col in (c.description):
            print(col[0], end='\t')
        print('\n----------------------------------')
        while True:
            # 每次抓取 3 条记录，该方法返回一个由 3 条记录组成的列表
            rows = c.fetchmany(3)
            # 如果抓取的 rows 为 None，退出循环
            if not rows :
                break
            # 再次使用循环遍历获取的列表
            for r in rows:
                print(r)
```

上面程序使用 fetchmany(3) 每次获取 3 条记录，该方法返回由 3 条记录组成的列表，因此程序还需要遍历该列表才能取出每条记录。

一般来说，在程序中应该尽量避免使用 fetchall() 来获取查询返回的全部记录。这是因为程序可能并不清楚实际查询会返回多少条记录，如果查询返回的记录数量太多，那么调用 fetchall() 一次获取全部记录可能会导致内存开销过大，情况严重时可能导致系统崩溃。

6.Python executescript()

SQLite 数据库模块的游标对象还包含了一个 executescript() 方法，这不是一个标准的 API 方法，这意味着在其他数据库 API 模块中可能没有这个方法。但是这个方法却很实用，它可以执行一段 SQL 脚本。

例如，如下程序使用 executescript() 方法执行一段 SQL 脚本：

```
1.          # 导入访问 SQLite 的模块
2.          import sqlite3
3.
4.          # ①、打开或创建数据库
5.          # 也可以使用特殊名：:memory:代表创建内存中的数据库
6.          conn = sqlite3.connect('first.db')
7.          # ②、获取游标
8.          c = conn.cursor()
9.          # ③、调用 executescript()方法执行一段 SQL 脚本
10.         c.executescript('''
11.             insert into user_tb values(null, '武松', '3444', 'male');
12.             insert into user_tb values(null, '林冲', '44444', 'male');
13.             create table item_tb(_id integer primary key autoincrement, name,
    price);
14.             ''')
15.         conn.commit()
16.         # ④、关闭游标
17.         c.close()
18.         # ⑤、关闭连接
19.         conn.close()
```

上面程序调用 executescript() 方法执行一段复杂的 SQL 脚本，在这段 SQL 脚本中包含了两条 insert 语句，该语句负责向 user_tb 表中插入记录，还使用 create 语句创建了

一个数据表。

运行上面程序，可以看到 first.db 数据库中多了一个 item_tb 数据表，user_tb 数据表被插入了两条记录。

此外，为了简化编程，SQLite 数据库模块还为数据库连接对象提供了如下 3 个方法：

1. execute(sql[, parameters])：执行一条 SQL 语句。
2. executemany(sql[, parameters])：根据序列重复执行 SQL 语句。
3. executescript(sql_script)：执行 SQL 脚本。

读者可能会发现，这 3 个方法与游标对象所包含的 3 个方法完全相同。事实正是如此，数据库连接对象的这 3 个方法都不是 DB API 2.0 的标准方法，它们只是游标对象的 3 个方法的快捷方式，因此在用法上与游标对象的 3 个方法完全相同。

7.Python SQLite create_function()

数据库连接对象还提供了一个 create_function(name, num_params, func) 方法，该方法用于注册一个自定义函数，接下来程序就可以在 SQL 语句中使用该自定义函数：

create_function 方法包含 3 个参数：

- name 参数：指定注册的自定义函数的名字。

- num_params：指定自定义函数所需参数的个数。
- func：指定自定义函数对应的函数。

下面程序使用 create_function() 方法为 SQL 语句注册一个自定义函数，然后程序就可以在 SQL 语句中使用该自定义函数：

```
# 导入访问 SQLite 的模块
import sqlite3

# 先定义一个普通函数，准备注册为 SQL 中的自定义函数
def reverse_ext(st):
    # 对字符串反转，前后加方括号
    return '[' + st[::-1] + ']'
# ①、打开或创建数据库
# 也可以使用特殊名：:memory:代表创建内存中的数据库
conn = sqlite3.connect('first.db')
# 调用 create_function 注册自定义函数：enc
conn.create_function('enc', 1, reverse_ext)
# ②、获取游标
c = conn.cursor()
# ③、在 SQL 语句中使用 enc 自定义函数
c.execute('insert into user_tb values(null, ?, enc(?), ?)',
    ('贾宝玉', '123456', 'male'))
conn.commit()
# ④、关闭游标
c.close()
# ⑤、关闭连接
conn.close()
```

上面程序中，第 12 行代码将 reverse_ext() 函数注册为自定义函数 enc，该函数用于模拟一个简单的加密功能：程序会对字符串反转，并在字符串前后添加方括号。

> 提示：此时使用的加密功能只是一个简单的模拟，如果需要真正对密码进行加密，则建议使用更高强度的加密算法，比如加盐 MD5 加密。

程序中第 16 行代码在执行的 SQL 语句中使用了 enc 自定义函数，该自定义函数用于对插入的密码进行加密。因此，当使用上面程序插入数据时，程序会自动对插入的密码进行加密；程序会对密码进行反转，并在密码前后添加方括号。

8.Python SQLite create_aggregate()

标准的 SQL 语句提供了如下 5 个标准的聚集函数：

- sum()：统计总和。
- avg()：统计平均值。
- count()：统计记录条数。
- max()：统计最大值。
- min()：统计最小值。

如果程序需要在 SQL 语句中使用与其他业务相关的聚集函数，则可使用数据库连接对象所提供的 create_aggregate(name, num_params, aggregate_class)方法，该方法用于注册一个自定义的聚集函数。

create_aggregate 方法包含 3 个参数：

- name：指定自定义聚集函数的名字。
- num_params：指定聚集函数所需的参数。
- aggregate_class :指定聚集函数的实现类。该类必须实现 step(self, params...) 和 finalize(self) 方法 ,其中 step() 方法对于查询所返回的每条记录各执行一次 ;finalize(self) 方法只在最后执行一次，该方法的返回值将作为聚集函数最后的返回值。

假设需要查询 user_tb 表中长度最短的密码，此时就需要用到自定义的聚集函数。下面程序使用 create_aggregate() 方法为 SQL 语句注册一个自定义的聚集函数 ,然后程序就可以在 SQL 语句中使用该自定义的聚集函数：

```
1.          # 导入访问 SQLite 的模块
2.          import sqlite3
3.
4.          # 先定义一个普通类，准备注册为 SQL 中的自定义聚集函数
5.          class MinLen:
6.              def __init__(self):
7.                  self.min_len = None
8.              def step(self, value):
9.                  # 如果 self.min_len 还未赋值,直接将当前 value 赋值给 self.min_lin
10.                 if self.min_len is None :
11.                     self.min_len = value
12.                     return
13.                 # 找到一个长度更短的 value，用 value 代替 self.min_len
14.                 if len(self.min_len) > len(value):
15.                     self.min_len = value
16.             def finalize(self):
```

```
17.                 return self.min_len
18.         # ①、打开或创建数据库
19.         # 也可以使用特殊名：:memory:代表创建内存中的数据库
20.         conn = sqlite3.connect('first.db')
21.         # 调用 create_aggregate 注册自定义聚集函数：min_len
22.         conn.create_aggregate('min_len', 1, MinLen)
23.         # ②、获取游标
24.         c = conn.cursor()
25.         # ③、在 SQL 语句中使用 min_len 自定义聚集函数
26.         c.execute('select min_len(pass) from user_tb')
27.         print(c.fetchone()[0])
28.         conn.commit()
29.         # ④、关闭游标
30.         c.close()
31.         # ⑤、关闭连接
32.         conn.close()
```

程序中第 22 行代码使用 create_aggregate() 创建了一个自定义聚集函数，该函数用于将 MinLen 类注册成 min_len 自定义聚集函数，其中 MinLen 类中的 step() 方法负责对每个传入的参数进行比较，选出长度最短的字符串，而 finalize() 方法则负责返回长度最短的字符串，该方法的返回值将作为聚集函数的返回值；第 26 行代码在 select 语句中使用自定义聚集函数，通过该函数就可以选出长度最短的密码。

运行上面的程序，将可以看到 use_tb 表中长度最短的密码被选出来。

9.Python SQLite create_collation()方法 :创建自定义比较函数

在标准的 SQL 语句中提供了一个 order by 子句，该子句用于对查询结果进行排序，但这种排序只会按默认的排序规则进行，如果程序需要按业务相关规则进行排序，则需要创建自定义的比较函数。

如果程序需要在 SQL 语句中使用与业务相关的比较函数，则可使用数据库连接对象所提供的 create_collation(name, callable) 方法，该方法用于注册一个自定义的比较函数。

create_collation 方法包含两个参数：

1. name：指定自定义比较函数的名字。
2. callable：指定自定义比较函数对应的函数。该函数包含两个参数，并对这两个参数进行大小比较，如果该方法返回正整数，系统认为第一个参数更大；如果返回负整数，系统认为第二个参数更大；如果返回 0，系统认为两个参数相等。

> 需要注意的是，callable 函数的参数以 Python（bytes）字节串的形式传入，因此系统默认会以 UTF-8 字符集将字符串编码成字节串后传入 callable 函数。

假设要求对 user_tb 表中的 pass 进行排序，但考虑到 pass 列前面采用了加密：第一个字符和最后一个字符都是方括号，因此程序会对 pass 列去掉前后两个方括号之后再进行排序。所以，程序需要自定义比较函数，该函数将会把字符串的第一个字符和最后一个字符

去掉后比较大小。

下面程序创建了一个自定义比较函数，然后在 SQL 语句中使用该自定义比较函数进行排序：

纯文本复制

```
# 导入访问 SQLite 的模块
import sqlite3

# 去掉字符串第一个、最后一个字符后比较大小
def my_collate(st1, st2):
    if st1[1: -1] == st2[1: -1]:
        return 0
    elif st1[1: -1] > st2[1: -1]:
        return 1
    else:
        return -1
# ①、打开或创建数据库
# 也可以使用特殊名：:memory:代表创建内存中的数据库
conn = sqlite3.connect('first.db')
# 调用 create_collation 注册自定义比较函数：sub_cmp
conn.create_collation('sub_cmp', my_collate)
# ②、获取游标
c = conn.cursor()
# ③、在 SQL 语句中使用 sub_cmp 自定义的比较函数
c.execute('select * from user_tb order by pass collate sub_cmp')
# 采用 for 循环遍历游标
for row in c:
    print(row)
conn.commit()
# ④、关闭游标
c.close()
# ⑤、关闭连接
conn.close()
```

上面程序中，第 5 行代码定义了一个大小比较函数：my_collate()，该函数会去掉字节串的第一个字符和最后一个字符后再比较大小。第 16 行代码调用 create_collation() 方法将 my_collate() 函数注册为 sub_cmp 自定义比较函数，接下来就可以在 SQL 语句中使用该自定义比较函数，如第 20 行代码所示。

运行上面的程序，将可以看到查询结果是按 pass 列去掉前后各一个字符之后进行排序的。

> 上面程序还用到游标的一个特点，游标本身是可迭代对象，因此程序不需要使用 fetchone() 来逐行获取查询结果，而是直接使用 for 循环来遍历游标获取查询结果集。

10.MySQL 数据库下载和安装

MySQL 是一款开源的数据库软件，由于其免费特性得到了全世界用户的喜爱，是目前使用人数最多的数据库。本节就给大家详细讲解如何下载和安装 MySQL 库。

> 使用 Python 的 DB API 2.0 来操作 MySQL 数据库与操作 SQLite 数据库并没有太大的区别，因为不管是 SQLite 数据库模块，还是 MySQL 数据库模块，它们遵循的是相同的 DB API 2.0 规范。

安装 MySQL 数据库与安装普通程序并没有太大的区别，关键是在配置 MySQL 数据库时需要注意选择支持中文的编码集。下面简要介绍在 Windows 平台上下载和安装 MySQL 数据库的步骤：

1. 登录 http://dev.mysql.com/downloads/mysql/ 站点，下载 MySQL 数据库社区版（Community）的最新版本。建议下载该版本的 MySQL 安装文件。读者可根据自己所用的 Windows 平台，选择下载相应的 MSI Installer 安装文件。

2. 下载完成后，得到一个 mysql-installer-web-community-8.0.15.0.msi 文件（笔者下载的版本）。双击该文件，开始安装 MySQL 数据库。

3. 在出现的“License Agreement”对话框界面，该界面要求用户必须接受该协议才能安装 MySQL 数据库。勾选该界面下方的“I accept the license terms”复选框，然后单击“Next”按钮，显示如图 1 所示的安装选项对话框。

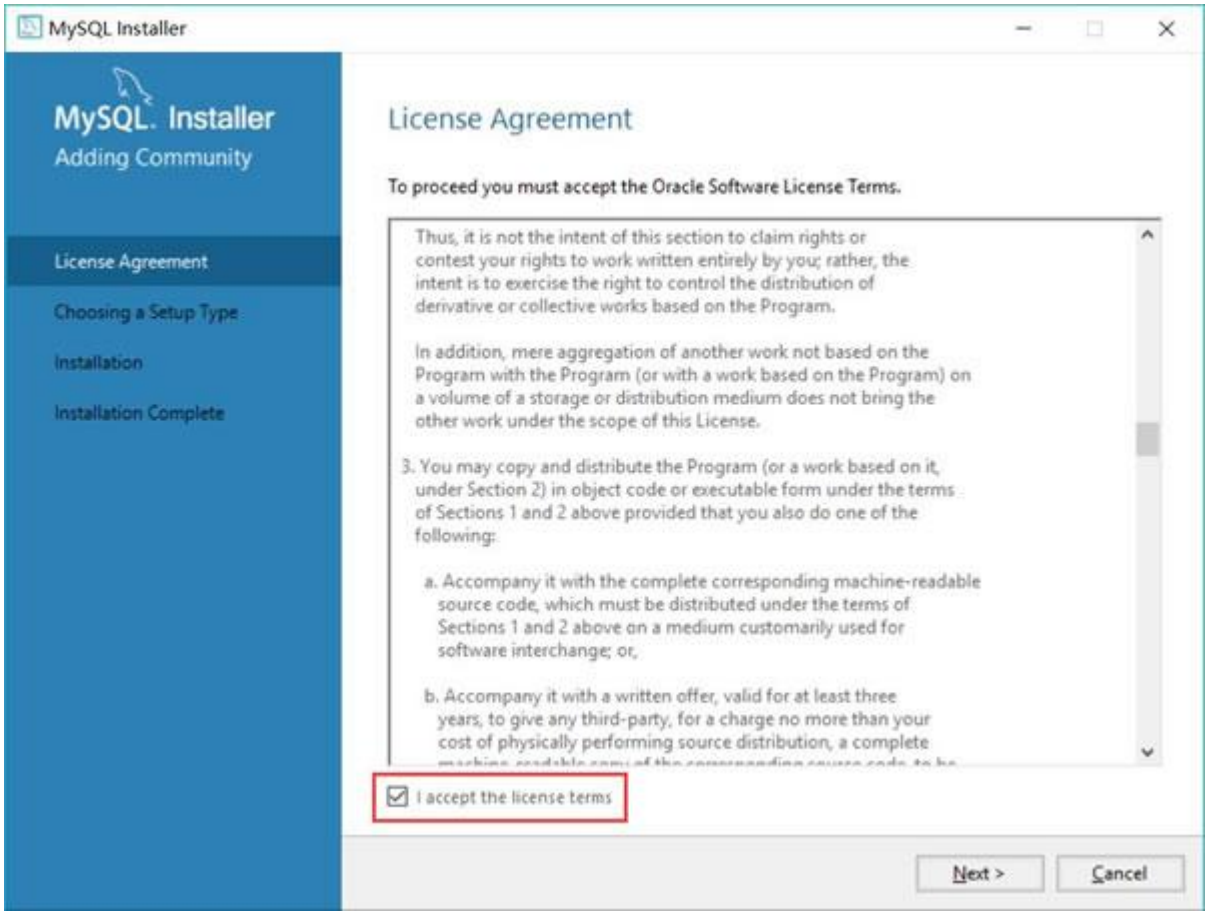


图 1 选择自定义安装

4. 勾选“Custom”单选钮，然后单击“Next”按钮。在接下来的界面中可以选择安装 MySQL 所需的组件，并选择 MySQL 数据库及数据文件的安装路径，本教程选择将 MySQL 数据库和数据文件都安装在 D 盘下。选择安装 MySQL 服务器、文档和 Connector/Python（这就是 Python 连接 MySQL 的模块），如图 2 所示。

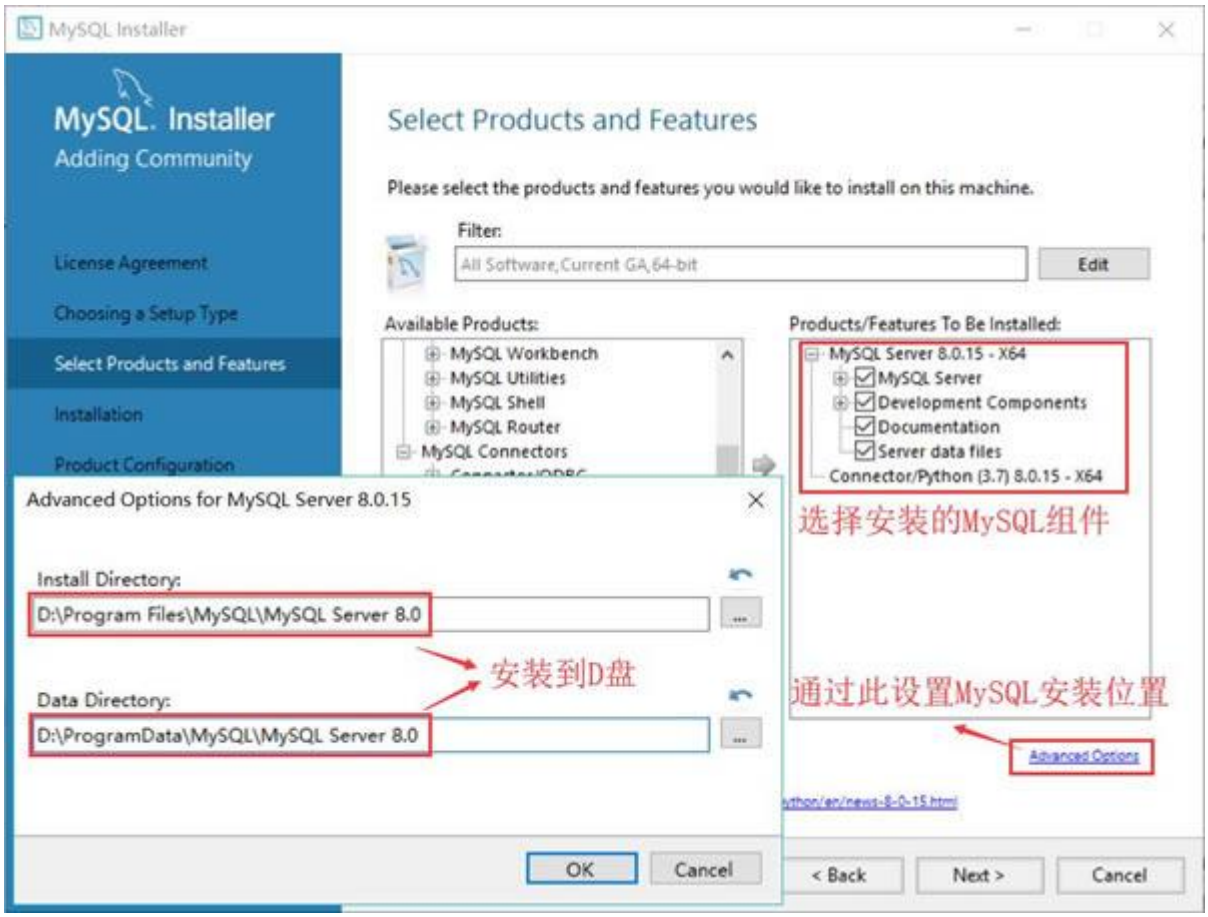


图 2 选择安装组件并设置数据库和数据文件的安装位置

5. 单击“Next”按钮，MySQL Installer 会检查系统环境是否满足安装 MySQL 数据库的要求。如果满足要求，则可以直接单击“Next”按钮开始安装；如果不符合条件，请根据 MySQL 提示先安装相应的系统组件，然后再重新安装 MySQL 数据库。

6. 开始安装 MySQL 数据库，安装完成后，会看到如图 3 所示的成功安装对话框。

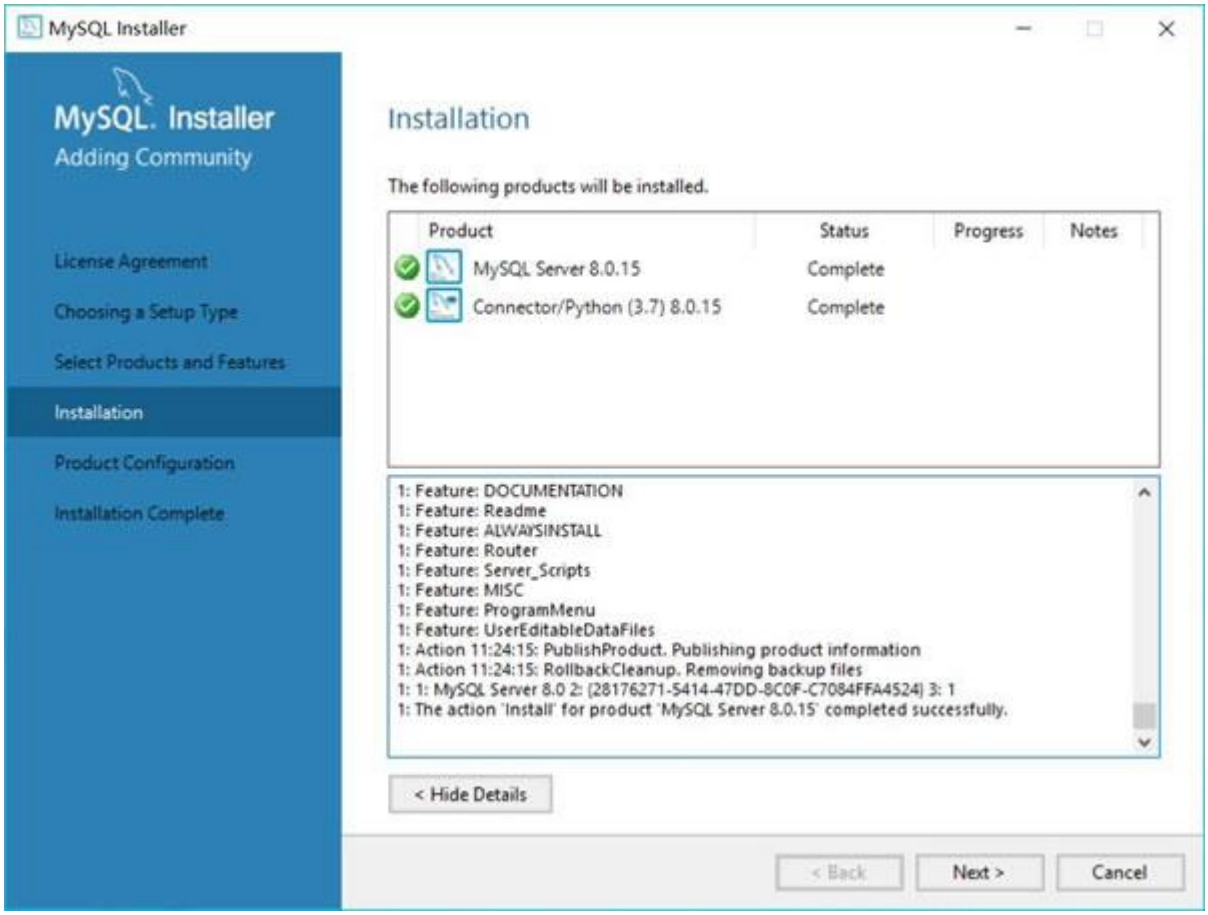


图 3 成功安装 MySQL 数据库

从图 3 可以看出，MySQL Server（数据库服务器）安装成功，Connector/Python 模块

也安装成功，通过下方的 Details 信息可以看到 Python 的 MySQL 模块的安装位置。

> 注意，MySQL8.0 需要 Visual C++ 2015 Redistributable，而且不管操作系统是 32 位还是 64 位的，它始终需要 32 位的 Visual C++ 2015 Redistributable，否则会安装失败。

MySQL 数据库程序安装成功后，系统还要求配置 MySQL 数据库。单击如图 3 所示对话框下方的“Next”按钮，开始配置 MySQL 数据库。在如图 4 所示的对话框中，选中“Standalone MySQL Server/Classic MySQL Replication”单选钮，即可进行更详细的配置。

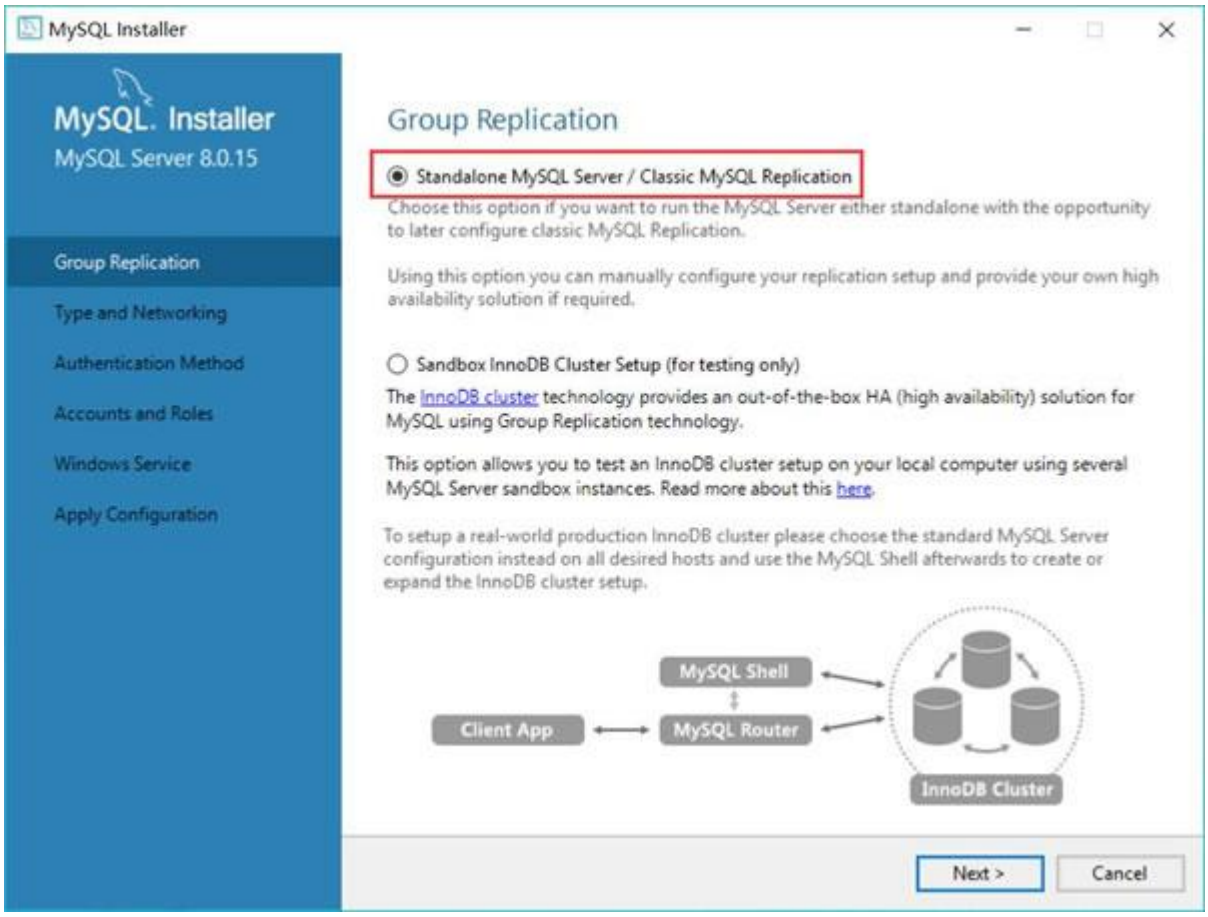


图 4 选择 Standalone 模式进行详细配置

两次单击“Next”按钮，将出现如图 5 所示的对话框，允许用户设置 MySQL 的 root 账户密码，也允许添加更多的用户。

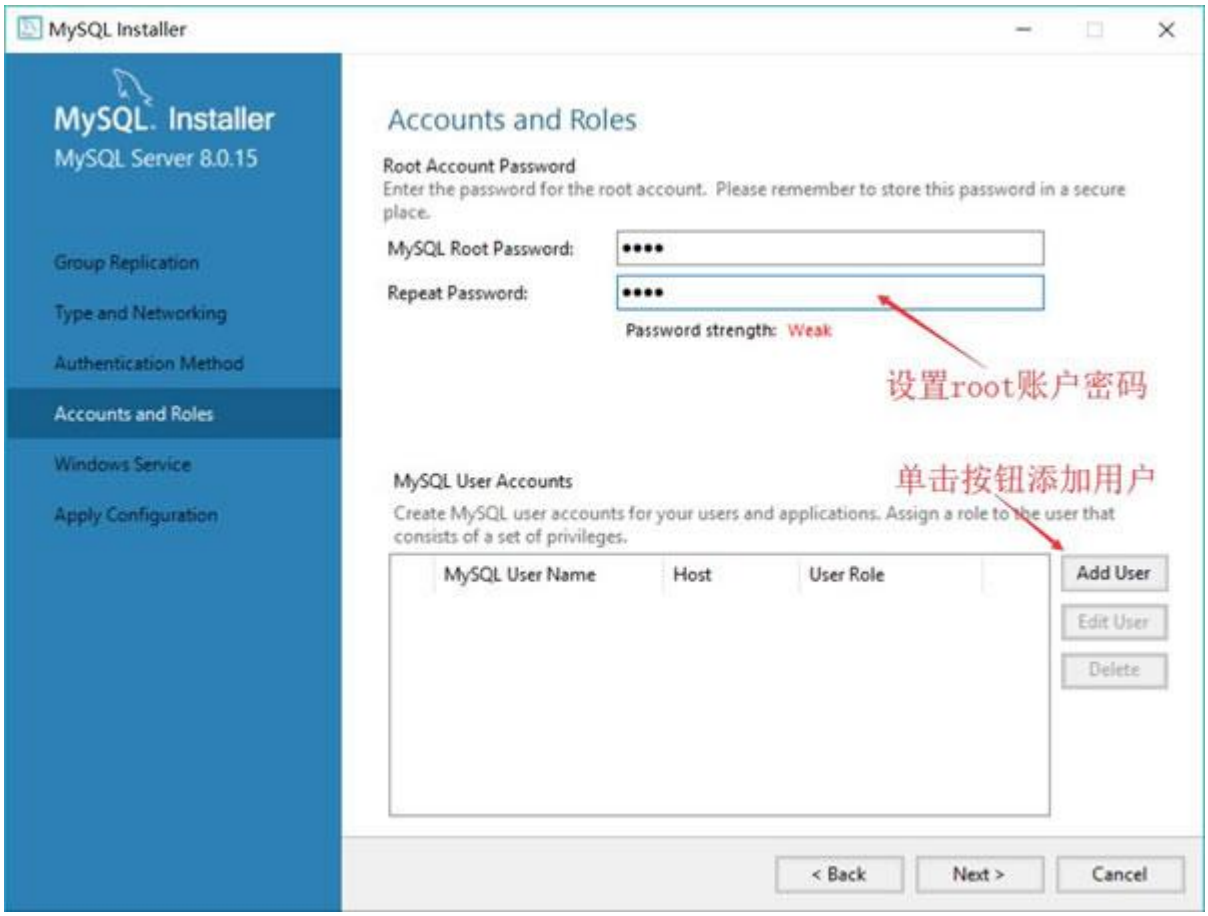


图 5 设置 root 账户密码和添加新用户

如果需要为 MySQL 数据库添加更多的用户，则可单击“Add User”按钮。设置完成后，单击“Next”按钮，将依次出现一系列对话框，但这些对话框对配置影响不大，直接单击

“Next”按钮，直至 MySQL 数据库配置成功。

MySQL 可通过命令行客户端来管理 MySQL 数据库及数据库中的数据。经过上面几个步骤之后，应该在 Windows 的“开始”菜单中看到“MySQL”→“MySQL Server 8.0”→“MySQL 8.0 Command Line Client - Unicode”菜单项，单击该菜单项将启动 MySQL 的命令行客户端窗口，进入该窗口将会提示输入 root 账户密码。

由于 MySQL 默认使用 UTF-8 字符串，因此应该通过“MySQL 8.0 Command Line Client - Unicode”菜单项启动命令行工具，该工具将会使用 UTF-8 字符集。市面上有一个名为 SQLyog 的工具程序提供了较好的图形用户界面来管理 MySQL 数据库中的数据。除此之外，MySQL 也提供了 MySQLAdministrator 工具来管理 MySQL 数据库。读者可以自行下载这两个工具，并使用这两个工具来管理 MySQL 数据库。但本教程依然推荐读者使用命令行工具，因为这种“恶劣”的工具会强制读者记住 SQL 命令。

在命令行客户端中输入在图 5 中为 root 账户设置的密码，进入 MySQL 数据库系统中，通过执行 SQL 命令就可以管理 MySQL 数据库了。

11. Python pip（管理模块工具）基础用法

前面章节在介绍安装 MySQL 服务器时，已经选择安装 MySQL 提供的 Python 数据库模块，而且根据最后的安装提示，显示已经安装成功。

现在的问题是，我们有没有办法来查看是否已经安装成功了呢？或者说，如果在安装 MySQL 服务器时忘记了选择 Connector/Python 模块，现在是否还有补救措施？

答案是肯定的。Python 自带了一个 pip 工具用来查看、管理所安装的各种模块。

查看已安装的模块

查看已安装的模块，使用如下命令：

```
pip show packagename
```

启动命令行窗口，在窗口中输入如下命令：

```
pip show mysql-connector-python
```

在上面的命令中，mysql-connector-python 就是该模块的名字。运行该命令，可以看到如下输出结果：

```
Name: mysql-connector-python

Version: 8.0.11

Summary: MySQL driver written in Python

Home-page: http://dev.mysql.com/doc/connector-python/en/index.html

Author: Oracle and/or its affiliates

Author-email: UNKNOWN

License: GNU GPLv2 (with FOSS License Exception)

Location: d:\python\python36\lib\site-packages
```

```
Requires: protobuf
Required-by:
```

从上面的输出结果可以看到，已经成功安装了 mysql-connector-python 8.0.11，以及该模块的官方网址和安装路径等有用的信息。

卸载已安装的模块

卸载已安装的模块，使用如下命令：

```
pip uninstall packagename
```

在命令行窗口中输入如下命令：

```
pip uninstall mysql-connector-python
```

运行该命令，可以看到如下输出结果：

```
Uninstalling mysql-connector-python 8.0.11:
  Would remove:
    d:\python\python36\lib\site-packages\mysql
    d:\python\python36\lib\site-packages\mysql_connector_python-8.0.11-py3.6.egg-info
    d:\python\python36\lib\site-packages\mysqlx
Proceed (y/n)?
```

上面的提示信息询问是否要删除 mysql-connector-python 模块，如果删除该模块，将会删除 3 个目录。如果希望删除，则可以在输入“y”之后按回车键。接下来将看到系统提示如下信息：

```
Successfully uninstalled mysql connector-python-8.0.11
```

该信息显示 mysql-connector-python-8.0.11 被删除成功。

执行该删除命令后，Python 将不再包含 mysql-connector-python 模块，相当于在安装 MySQL 服务器时没有选择 Connector/Python 模块。

如果要查看已安装的所有模块，可以使用如下命令：

```
pip list
```

安装模块

安装模块，使用如下命令：

```
pip install packagename
```

在命令行窗口中输入如下命令：

```
pip install mysql-connectar-python
```

运行该命令，将看到程序下载并安装 mysql-connector-python 模块的过程，最后会生成如下提示信息：

```
Successfully installed mysql-connector-python-8.0.11
```

上面的信息提示该模块安装成功。

如果希望安装不同版本的模块，则可指定版本号。例如：

```
pip install packagename ==1.0.4 #安装指定版本
```

除使用 MySQL 官方提供的 Python 模块来连接 MySQL 数据库之外，还有一个使用广泛的连接 MySQL 数据库的模块 MySQL-python，其官方站点为 https://pypi.org/project/MySQL-python/。

12.Python MySQL 数据库执行 DDL 语句

在使用 mysql-connector-python 模块操作 MySQL 数据库之前，同样先检查一下该模块的全局属性。

```
>>> import mysql.connectar
>>> mysql.connector.apilevel
'2.0'
>>> mysql.connector.paramstyle
'pyformat'
>>>
```

从上面的输出信息可以看到，mysql-connector-python 数据库模块同样遵守 DB API 2.0 规范，且该模块允许在 SQL 语句中使用扩展的格式代码（pyformat）来代表参数。

使用 MySQL 模块对 MySQL 数据库执行 DDL 语句，与使用 SQLite 模块对 SQLite 数据库执行 DDL 语句并没有太大的区别，只是 MySQL 数据库有服务器进程，默认通过 3306 端口对外提供服务。因此，Python 程序在连接 MySQL 数据库时可指定远程服务器 IP 地址和端口，如果不指定服务器 IP 地址和端口，则使用默认的服务器 IP 地址 localhost 和默认端口 3306。

下面程序示范了如何连接 MySQL 数据库，并通过 DDL 语句来创建两个数据表：

纯文本复制

```
# 导入访问 MySQL 的模块
import mysql.connector

# ①、连接数据库
conn = conn = mysql.connector.connect(user='root', password='32147',
    host='localhost', port='3306',
    database='python', use_unicode=True)
# ②、获取游标
c = conn.cursor()
# ③、执行 DDL 语句创建数据表
c.execute('''create table user_tb(
    user_id int primary key auto_increment,
    name varchar(255),
    pass varchar(255),
    gender varchar(255))''')
# 执行 DDL 语句创建数据表
c.execute('''create table order_tb(
    order_id integer primary key auto_increment,
```

```
            item_name varchar(255),
            item_price double,
            item_number double,
            user_id int,
            foreign key(user_id) references user_tb(user_id) )''')
        # ④、关闭游标
        c.close()
        # ⑤、关闭连接
        conn.close()
```

与连接 SQLite 的程序相比，上面程序最大的区别就在于第 5 行代码，程序要连接 localhost 主机上 3306 端口服务的 python 数据库 ,必须先在本机的 MySQL 数据库服务器中创建一个 python 数据库。

通过“开始”菜单中的“MySQL→MySQL Server 8.0->MySQL 8.0 Command Line Client-Unicode”启动 MySQL 的命令行客户端，输入在 root 账户设置的密码，即可进入 MySQL 的命令行客户端，然后输入如下命令来创建 python 数据库：

```
create database python;
```

运行上面的程序，当程序运行结束后，将可以看到 python 数据库中多了两个数据表。

上面程序中的 ①~⑤ 步与前面并无区别，程序在第 ③ 步执行了两次，每次分别执行一条 create 语句。因此，该程序执行完成后，将会看到当前数据库中包含两个数据表，user_tb 和 order_tb，且在 order_tb 表中有一个外键列引用了 user_tb 表的 user_id 主键列。

需要指出的是，此处程序使用 execute() 方法执行的 create 语句与前面操作 SQLite 数

据库时所使用的 create 语句略有差异，但这个差异是由两个数据库本身所引起的，与 Python 程序并没有任何关系。

如果 Python 程序提示某条 SQL 语句有语法错误，则最好先利用此处介绍的 MySQL 的命令行客户端测试这条 SQL 语句，以保证该语句的语法正确。

> 同一条 SQL 语句，在有的数据库上可能是成功的，而在其他数据库上可能会失败，这是由于不同数据库所支持的 SQL 虽然大体是相同的，但在实现细节上略有差异。

13.Python MySQL 数据库执行 DML 语句

与使用 SQLite 数据库模块类似，MySQL 数据库模块同样可以使用游标的 execute() 方法执行 DML 的 insert、update、delete 语句，对数据库进行插入、修改和删除数据操作。

例如，如下程序示范了向数据库的两个数据表中分别插入一条数据：

```
1.          # 导入访问 MySQL 的模块
2.          import mysql.connector
3.
4.          # ①、连接数据库
5.          conn = conn = mysql.connector.connect(user='root', password='32147',
6.              host='localhost', port='3306',
7.              database='python', use_unicode=True)
8.          # ②、获取游标
9.          c = conn.cursor()
10.         # ③、调用执行 insert 语句插入数据
```

```
11.         c.execute('insert into user_tb values(null, %s, %s, %s)',
12.             ('孙悟空', '123456', 'male'))
13.         c.execute('insert into order_tb values(null, %s, %s, %s, %s)',
14.             ('鼠标', '34.2', '3', 1))
15.         conn.commit()
16.         # ④、关闭游标
17.         c.close()
18.         # ⑤、关闭连接
19.         conn.close()
```

上面程序中第 11、13 行代码分别用于向 user_tb 和 order_tb 表中插入数据。注意该程序的 SQL 语句中的占位符：%s，这正如 mysql.connector.paramstyle 属性所标识的 pyformat，它指定在 SQL 语句中使用扩展的格式代码来作为占位符。程序运行完成之后，就会向 python 数据库的两个数据表中各插入一条记录。

与 SQLite 数据库模块类似的是 ,MySQL 数据库模块同样支持使用 executemany() 方法重复执行一条 SQL 语句。例如如下程序：

```
1.          # 导入访问 MySQL 的模块
2.          import mysql.connector
3.
4.          # ①、连接数据库
5.          conn = conn = mysql.connector.connect(user='root', password='32147',
6.              host='localhost', port='3306',
7.              database='python', use_unicode=True)
8.          # ②、获取游标
9.          c = conn.cursor()
10.         # ③、调用 executemany()方法把同一条 SQL 语句执行多次
11.         c.executemany('insert into user_tb values(null, %s, %s, %s)',
12.             (('sun', '123456', 'male'),
13.             ('bai', '123456', 'female'),
14.             ('zhu', '123456', 'male'),
```

```
            ('niu', '123456', 'male'),
            ('tang', '123456', 'male')))
        conn.commit()
        # ④、关闭游标
        c.close()
        # ⑤、关闭连接
        conn.close()
```

该程序与前面使用 SQLite 数据库模块重复执行 SQL 语句的程序基本相同，只是该程序在 SQL 语句中使用了 %s 作为占位符。

使用 MySQL 数据库模块中游标的 executemany() 方法同样可重复执行 update、delete 语句：

```
        # 导入访问 MySQL 的模块
        import mysql.connector

        # ①、连接数据库
        conn = conn = mysql.connector.connect(user='root', password='32147',
            host='localhost', port='3306',
            database='python', use_unicode=True)
        # ②、获取游标
        c = conn.cursor()
        # ③、调用 executemany()方法把同一条 SQL 语句执行多次
        c.executemany('update user_tb set name=%s where user_id=%s',
            (('小孙', 2),
            ('小白', 3),
            ('小猪', 4),
            ('小牛', 5),
            ('小唐', 6)))
        # 通过 rowcount 获取被修改的记录条数
        print('修改的记录条数：', c.rowcount)
        conn.commit()
```

```
20.         # ④、关闭游标
21.         c.close()
22.         # ⑤、关闭连接
23.         conn.close()
```

需要说明的是，MySQL 数据库模块的连接对象有一个 autoconunit 属性，如果将该属性设为 True ,则意味着关闭该连接的事务支持 ,程序每次执行 DML 语句之后都会自动提交，这样程序就无须调用连接对象的 commit() 方法来提交事务了。例如如下程序：

纯文本复制

```
1.          # 导入访问 MySQL 的模块
2.          import mysql.connector
3.
4.          # ①、连接数据库
5.          conn = conn = mysql.connector.connect(user='root', password='32147',
6.              host='localhost', port='3306',
7.              database='python', use_unicode=True)
8.          # 将 autocommit 设置 True，关闭事务
9.          conn.autocommit = True
10.         #下面执行的 DML 语句会自动提交
11.         ...
12.         # ④、关闭游标
13.         c.close()
14.         # ⑤、关闭连接
15.         conn.close()
```

在上面程序中，将连接对象的 autocommit 设为 True，这意味着该连接将会自动提交每条 DML 语句，相当于关闭了事务，所以程序也不需要调用连接对象的 commit() 方法来提交事务。

14.Python MySQL 数据库执行查询语句

使用 MySQL 数据库模块执行查询语句，与使用 SQLite 数据库模块执行查询语句基本相似，只需注意 SQL 语句中的占位符的差别即可。

例如，如下程序示范了查询 MySQL 数据库中的数据：

```
1.          # 导入访问 MySQL 的模块
2.          import mysql.connector
3.
4.          # ①、连接数据库
5.          conn = conn = mysql.connector.connect(user='root', password='32147',
6.              host='localhost', port='3306',
7.              database='python', use_unicode=True)
8.          # ②、获取游标
9.          c = conn.cursor()
10.         # ③、调用执行 select 语句查询数据
11.         c.execute('select * from user_tb where user_id > %s', (2,))
12.         # 通过游标的 description 属性获取列信息
13.         for col in (c.description):
14.             print(col[0], end='\t')
15.         print('\n----------------------------------')
16.         # 直接使用 for 循环来遍历游标中的结果集
17.         for row in c:
18.             print(row)
19.             print(row[1] + '-->' + row[2])
20.         # ④、关闭游标
21.         c.close()
22.         # ⑤、关闭连接
23.         conn.close()
```

上面程序中，第 11 行代码调用 execute() 方法执行 select 语句查询数据，在该 SQL 语句中同样使用了 %s 作为占位符，这就是与 SQLite 数据库模块的差别。

该程序直接使用 for 循环来遍历游标所包含的查询数据，这完全是可以的，因为游标本身就是可遍历对象。运行上面的程序，可以看到如下查询结果：

```
user_id name pass gender

------------------------------

(3, '小白'，'123456'，'female')

小白-->123456

(4, '小猪', '123456', 'male')

小猪-->123456

(5, '小牛', '123456', 'male')

小牛-->123456

(6, '小唐', '123456', 'male')

小唐-->123456

(7, '孙悟空', '123456', 'male')

孙悟空-->123456
```

MySQL 数据库模块的游标对象同样支持 fetchone()、fetchmany()、fetchall() 方法，例如如下程序使用 fetchmany() 方法每次获取 3 条记录：

纯文本复制

```
# 导入访问 MySQL 的模块
import mysql.connector

```

```
        # ①、连接数据库
        conn = conn = mysql.connector.connect(user='root', password='32147',
            host='localhost', port='3306',
            database='python', use_unicode=True)
        # ②、获取游标
        c = conn.cursor()
        # ③、调用执行 select 语句查询数据
        c.execute('select * from user_tb where user_id > %s', (2,))
        # 通过游标的 description 属性获取列信息
        for col in (c.description):
            print(col[0], end='\t')
        print('\n---------------------------------')
        while True:
            # 每次抓取 3 条记录，该方法返回一个多个元组组成的列表
            rows = c.fetchmany(3)
            # 如果抓取的 row 为 None，退出循环
            if not rows :
                break
            # 再次使用循环遍历获取的列表
            for r in rows:
                print(r)
        # ④、关闭游标
        c.close()
        # ⑤、关闭连接
        conn.close()
```

15.Python MySQL callproc()方法：调用数据库存储过程

MySQL 数据库模块为游标对象提供了一个非标准的 callproc(self, procname, args=()) 方法，该方法用于调用数据库存储过程。

callproc 方法的 procname 参数代表存储过程的名字，而 args 参数则用于为存储过程传入参数。

下面的 SQL 脚本可以在 MySQL 数据库中创建一个简单的存储过程。打开 MySQL 的命令行客户端，连接 python 数据库之后，输入如下 SQL 脚本来创建存储过程：

```
delimiter //
create procedure add pro(a int, b int, out sum int)
begin
set sum = a + b;
end;
//
```

下面程序示范了使用 MySQL 数据库模块来调用存储过程：

纯文本复制

```
# 导入访问 MySQL 的模块
import mysql.connector

# ①、连接数据库
conn = conn = mysql.connector.connect(user='root', password='32147',
    host='localhost', port='3306',
    database='python', use_unicode=True)
# ②、获取游标
c = conn.cursor()
# ③、调用 callproc()方法执行存储过程
# 虽然 add_pro 存储过程需要 3 个参数，但最后一个参数是传出参数，
# 因此程序不会用它的值
```

```
13.         result_args = c.callproc('add_pro', (5, 6, 0))
14.         # 返回的 result_args 既包含了传入参数的值，也包含了传出参数的值
15.         print(result_args)
16.         # 如果只想访问传出参数的值，可直接访问 result_args 的第 3 个元素，如下代
    码
17.         print(result_args[2])
18.         # ④、关闭游标
19.         c.close()
20.         # ⑤、关闭连接
21.         conn.close()
```

上面程序中，第 13 行代码就是调用存储过程的关键代码。使用 MySQL 数据库模块调用存储过程非常简单，存储过程需要几个参数，程序通过 callproc() 方法调用存储过程时就传入一个包含几个元素的元组；对于存储过程的传入参数，该参数对应的元组元素负责为传入参数传值；对于存储过程的传出参数，该参数对应的元组元素随便定义即可。

运行上面的程序，可以看到如下输出结果：

```
(5, 6, 11)

11
```

从上面的输出结果来看，当程序使用 Python 调用存储过程后，程序会返回传入参数和传出参数组成的元组，如第一行输出结果所示。如果程序只需要获取传出参数的值，则通过返回的结果元组取出对应的值即可。

16.Python PyMySQL 模块下载和安装

由于 MySQL 服务器以独立的进程运行，并通过网络对外服务。所以，需要支持 Python 的 MySQL 驱动来连接到 MySQL 服务器。前面在安装 MySQL 时，要大家安装

Connector/Python 模块，这是 Python 连接 MySQL 的模块，但除此之外，还可以使用 PyMySQL 模块。

PyMySQL 是支持 Python 操作数据库的模块，它也可以实现 Python 和 MySQL 之间的连接。

注意，MySQLdb 模块不再支持 Python 3.x 版本。

在实际编程中，会遇到很多类似 Connector/Python、PyMySQL 这样的模块，它们也被称为接口程序，即通过该程序，可以对另外一个对象进行操作。

接口程序就好比钥匙，如果我们要开锁，必须要借助其他的工具，其中要属和锁配对的钥匙是最佳选择（也可以使用锤子，只不多不便利），打开锁的工具就是接口程序。PyMySQL 就可以作为打开 MySQL 数据库的钥匙。

本节就来讲解如何下载并安装 PyMySQL。PyMySQL 的安装方式主要有 2 种，一种是先下载源码，然后再安装；另一种是使用 pip 自动下载并安装。PyMySQL 源码的下载地址为 https://github.com/PyMySQL/PyMySQL，读者可以下载源码进行安装。本节选择用第 2 种方式安装。

使用 pip 安装 PyMySQL 非常简单，只需要在命令行中执行一行命令：

```
$sudo pip install PyMySQL
```

安装完成之后，在 Python 交互模式下运行如下命令：

```
>>>import pymysql
>>>
```

如上所示，如果 Python 解释器不报错，则证明 PyMySQL 安装成功。

第 16 章 Python 并发编程

相比单线程的编程，多线程的程序可包含多个顺序执行流，这些顺序执行流之间互不干扰。也可以这样理解，单线程的程序如同只雇佣一个服务员的餐厅，他必须做完一件事情后才可以做下一件事情，而多线程的程序则如同雇佣多个服务员的餐厅，他们可以同时做多件事情。Python 语言提供了非常优秀的多线程支持，程序可以通过非常简单的方式来启动多线程。本章将会详细介绍 Python 多线程编程的相关知识，包括创建、启动线程，控制线程，以及多线程的同步操作，并会介绍如何利用 Python 内建支持的线程池来提高多线程的性能。

1.Python 进程和线程

几乎所有的操作系统都支持同时运行多个任务，每个任务通常是一个程序，每一个运行中的程序就是一个进程，即进程是应用程序的执行实例。现代的操作系统几乎都支持多进程并发执行。

注意，并发和并行是两个概念，并行指在同一时刻有多条指令在多个处理器上同时执行；并发是指在同一时刻只能有一条指令执行，但多个进程指令被快速轮换执行，使得在宏观上具有多个进程同时执行的效果。

例如，程序员一边开着开发工具在写程序，一边开着参考手册备查，同时还使用电脑播放音乐……除此之外，每台电脑运行时还有大量底层的支撑性程序在运行……这些进程看上去像

是在同时工作。

但事实的真相是，对于一个 CPU 而言，在某个时间点它只能执行一个程序。也就是说，只能运行一个进程，CPU 不断地在这些进程之间轮换执行。那么，为什么用户感觉不到任何中断呢？

这是因为相对人的感觉来说，CPU 的执行速度太快了（如果启动的程序足够多，则用户依然可以感觉到程序的运行速度下降了）。所以，虽然 CPU 在多个进程之间轮换执行，但用户感觉到好像有多个进程在同时执行。

线程是进程的组成部分，一个进程可以拥有多个线程。在多线程中，会有一个主线程来完成整个进程从开始到结束的全部操作，而其他的线程会在主线程的运行过程中被创建或退出。

当进程被初始化后，主线程就被创建了，对于绝大多数的应用程序来说，通常仅要求有一个主线程，但也可以在进程内创建多个顺序执行流，这些顺序执行流就是线程。

当一个进程里只有一个线程时，叫作单线程。超过一个线程就叫作多线程。

每个线程必须有自己的父进程，且它可以拥有自己的堆栈、程序计数器和局部变量，但不拥有系统资源，因为它和父进程的其他线程共享该进程所拥有的全部资源。线程可以完成一定的任务，可以与其他线程共享父进程中的共享变量及部分环境，相互之间协同完成进程所要完成的任务。

多个线程共享父进程里的全部资源，会使得编程更加方便，需要注意的是，要确保线程不会妨碍同一进程中的其他线程。

线程是独立运行的，它并不知道进程中是否还有其他线程存在。线程的运行是抢占式的，也就是说，当前运行的线程在任何时候都可能被挂起，以便另外一个线程可以运行。

多线程也是并发执行的，即同一时刻，Python 主程序只允许有一个线程执行，这和全局解释器锁有关系，后续会做详细介绍。

一个线程可以创建和撤销另一个线程，同一个进程中的多个线程之间可以并发运行。

从逻辑的角度来看，多线程存在于一个应用程序中，让一个应用程序可以有多个执行部分同时执行，但操作系统无须将多个线程看作多个独立的应用，对多线程实现调度和管理以及资源分配，线程的调度和管理由进程本身负责完成。

简而言之，进程和线程的关系是这样的：操作系统可以同时执行多个任务，每一个任务就是一个进程，进程可以同时执行多个任务，每一个任务就是一个线程。

2.Python 创建线程

Python 中，有关线程开发的部分被单独封装到了模块中，和线程相关的模块有以下 2 个：

- _thread：是 Python 3 以前版本中 thread 模块的重命名，此模块仅提供了低级别的、原始的线程支持，以及一个简单的锁。功能比较有限。正如它的名字所暗示的（以 _ 开头），一般不建议使用 thread 模块；
- threading：Python 3 之后的线程模块，提供了功能丰富的多线程支持，推荐使用。

本节就以 threading 模块为例进行讲解。Python 主要通过两种方式来创建线程：

1. 使用 threading 模块中 Thread 类的构造器创建线程。即直接对类 threading.Thread 进行实例化，并调用实例化对象的 start 方法创建线程。
2. 继承 threading 模块中的 Thread 类创建线程类。即用 threading.Thread 派生出一个新的子类，将新建类实例化，并调用其 start 方法创建线程。

调用 Thread 类的构造器创建线程

调用 Thread 类的构造器创建线程很简单，直接调用 threading.Thread 类的如下构造器创建线程：

```
__init__(self, group=None, target=None, name=None, args=(), kwargs=None,
*,daemon=None)
```

上面的构造器涉及如下几个参数：

- group：指定该线程所属的线程组。目前该参数还未实现，因此它只能设为 None。
- target：指定该线程要调度的目标方法。
- args：指定一个元组，以位置参数的形式为 target 指定的函数传入参数。元组的第一个元素传给 target 函数的第一个参数，元组的第二个元素传给 target 函数的第二个参数……依此类推。
- kwargs：指定一个字典，以关键字参数的形式为 target 指定的函数传入参数。
- daemon：指定所构建的线程是否为后代线程。

通过 Thread 类的构造器创建并启动多线程的步骤如下：

1. 调用 Thread 类的构造器创建线程对象。在创建线程对象时，target 参数指定的函数将作为线程执行体。
2. 调用线程对象的 start() 方法启动该线程。

下面程序示范了通过 Thread 类的构造器来创建线程对象：

```
import threading

# 定义一个普通的 action 函数，该函数准备作为线程执行体
def action(max):
    for i in range(max):
        # 调用 threading 模块 current_thread()函数获取当前线程
        # 线程对象的 getName()方法获取当前线程的名字
        print(threading.current_thread().getName() +  " " + str(i))
# 下面是主程序（也就是主线程的执行体）
for i in range(100):
    # 调用 threading 模块 current_thread()函数获取当前线程
    print(threading.current_thread().getName() +  " " + str(i))
    if i == 20:
        # 创建并启动第一个线程
        t1 =threading.Thread(target=action,args=(100,))
        t1.start()
        # 创建并启动第二个线程
        t2 =threading.Thread(target=action,args=(100,))
        t2.start()
print('主线程执行完成!')
```

上面程序中的主程序包含一个循环，当循环变量 i 等于 20 时创建并启动两个新线程：

1. 创建了一个 Thread 对象，该线程的 target 为 action，这意味着它会将 action 函数作为线程执行体。接下来程序调用 start() 方法来启动 t1 线程。
2. 再次创建了一个线程，其创建和启动方式与第一个线程完全相同。

运行上面程序，将会看到如图 1 所示的界面。

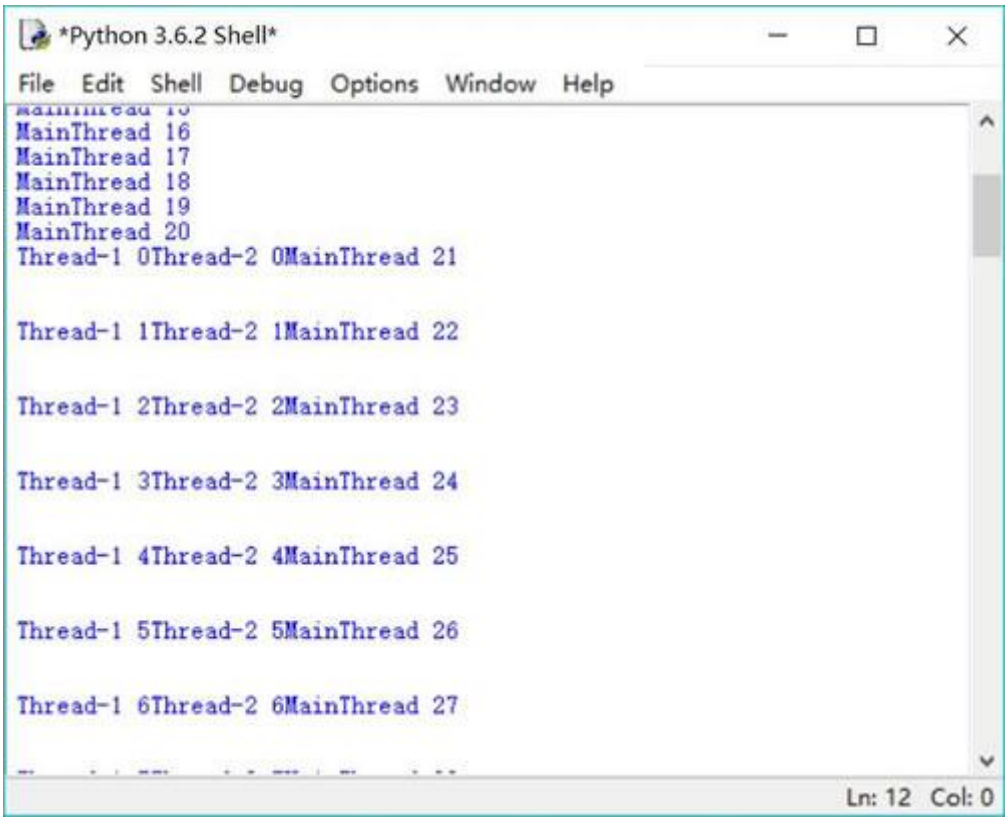


图 1 多线程的运行效果

> 默认情况下，主线程的名字为 MainThread，用户启动的多个线程的名字依次为 Thread-1、Thread-2、Thread-3、...、Thread-n 等。

虽然上面程序只显式创建并启动了两个线程，但实际上程序有三个线程，即程序显式创建的两个子线程和主线程。前面已经提到，当 Python 程序开始运行后，程序至少会创建一个

主线程，主线程的线程执行体就是程序中的主程序（没有放在任何函数中的代码）。

> 在进行多线程编程时，不要忘记 Python 程序运行时默认的主线程，主程序部分（没有放在任何函数中的代码）就是主线程的线程执行体。

从图 1 可以看出，此时程序中共包含三个线程，这三个线程的执行没有先后顺序，它们以并发方式执行 :Thread-1 执行一段时间 ,然后可能 Thread-2 或 MainThread 获得 CPU 执行一段时间，接下来又换其他线程执行，这就是典型的线程并发执行，CPU 以快速轮换的方式在多个线程之间切换，从而给用户一种错觉，即多个线程似乎同时在执行。

通过上面介绍不难看出多线程的意义，如果不使用多线程，主程序直接调用两次 action() 函数，那么程序必须等第一次调用的 action() 函数执行完成，才会执行第二次调用的 action() 函数；必须等第二次调用的 action() 函数执行完成，才会继续向下执行主程序。

而使用多线程之后，程序可以让两个 action() 函数、主程序以并发方式执行，给用户一种错觉，两个 action() 函数和主程序似乎同时在执行。

> 说穿了很简单，多线程就是让多个函数能并发执行，让普通用户感觉到多个函数似乎同时在执行。

除此之外，上面程序还用到了如下函数和方法：

- threading.current_thread()：它是 threading 模块的函数，该函数总是返回当前正在执行的线程对象。

- getName()：它是 Thread 类的实例方法，该方法返回调用它的线程名字。

在 Threading 模块中，除了 current_thread() 函数外，还经常使用如下 2 个函数：

- threading.enumerate() :返回一个正运行线程的 list。“正运行”是指线程处于“启动后，且在结束前”状态，不包括“启动前”和“终止后”状态。
- threading.activeCount()：返回正在运行的线程数量。与 len(threading.enumerate()) 有相同的结果。

> 程序通过 geName() 方法返回指定线程的名字，还可以通过 setName(name) 方法为线程设置名字，这两个方法可通过 name 属性来代替。

继承 Thread 类创建线程类

通过继承 Thread 类来创建并启动线程的步骤如下：

1. 定义 Thread 类的子类，并重写该类的 run() 方法。run() 方法的方法体就代表了线程需要完成的任务，因此把 run() 方法称为线程执行体。
2. 创建 Thread 子类的实例，即创建线程对象。
3. 调用线程对象的 start() 方法来启动线程。

下面程序示范了通过继承 Thread 类来创建并启动线程：

```
1.          import threading
2.
```

```
3.          # 通过继承 threading.Thread 类来创建线程类
4.          class FkThread(threading.Thread):
5.              def __init__(self):
6.                  threading.Thread.__init__(self)
7.                  self.i = 0
8.              # 重写 run()方法作为线程执行体
9.              def run(self):
10.                 while self.i < 100:
11.                     # 调用 threading 模块 current_thread()函数获取当前线程
12.                     # 线程对象的 getName()方法获取当前线程的名字
13.                     print(threading.current_thread().getName() +  " " +
    str(self.i))
14.                     self.i += 1
15.         # 下面是主程序（也就是主线程的执行体）
16.         for i in range(100):
17.             # 调用 threading 模块 current_thread()函数获取当前线程
18.             print(threading.current_thread().getName() +  " " + str(i))
19.             if i == 20:
20.                 # 创建并启动第一个线程
21.                 ft1 = FkThread()
22.                 ft1.start()
23.                 # 创建并启动第二个线程
24.                 ft2 = FkThread()
25.                 ft2.start()
26.         print('主线程执行完成!')
```

上面程序中的 FkThread 类继承了 threading.Thread 类，并实现了 run() 方法。run() 方法中的代码执行流就是该线程所需要完成的任务。

运行上面程序，将会看到如图 2 所示的界面。

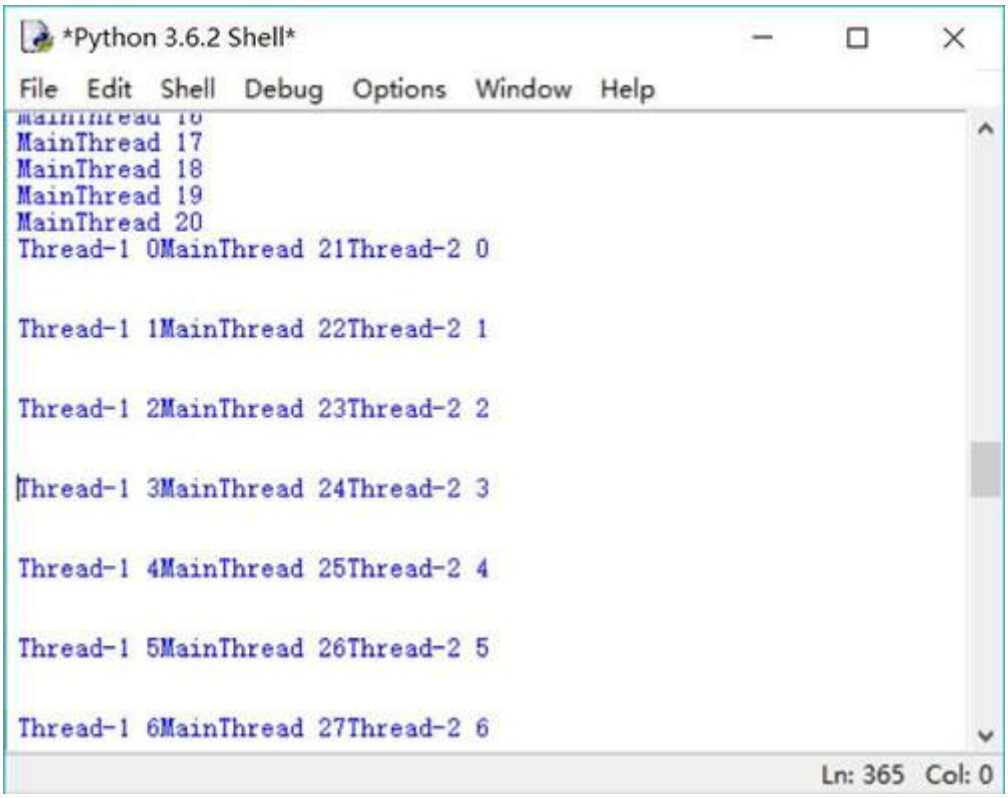


图 2 继承 Thread 类创建并启动线程

从图 2 可以看到，此时程序中同样有主线程、Thread-1 和 Thread-2 三个线程，它们以快速轮换的方式在执行，这就是三个线程并发执行的效果。

通常来说，推荐使用第一种方式来创建线程，因为这种方式不仅编程简单，而且线程直接包装 target 函数，具有更清晰的逻辑结构。

3.Python 线程的生命周期

Python 线程的生命周期（新建、就绪、运行、阻塞和死亡）

当线程被创建并启动后，并不会直接进入执行状态，也不会一直处于执行状态，线程的生命周期中，它会经历新建（new）、就绪（Ready）、运行（Running）、阻塞（Blocked）和死亡（Dead）5 种状态。

尤其是当线程启动以后，它不可能一直“霸占”着 CPU 独自运行。所以 CPU 需要在多个线程之间切换，于是线程的状态也会多次在运行、就绪之间转换。

线程的新建和就绪状态

当程序创建了一个 Thread 对象或 Thread 子类的对象之后，该线程就处于新建状态，和其他的 Python 对象一样，此时的线程对象并没有表现出任何线程的动态特征，程序也不会执行线程执行体。

当线程对象调用 start() 方法之后，该线程处于就绪状态，Python 解释器会为其创建方法调用栈和程序计数器，处于这种状态中的线程并没有开始运行，只是表示该线程可以运行了。至于该线程何时开始运行，取决于 Python 解释器中线程调度器的调度。

> 注意，启动线程使用 start() 方法，而不是 run() 方法。调用 start() 方法来启动线程，系统会把该 run() 方法当成线程执行体来处理；但如果直接调用线程对象的 run() 方法，则 run() 方法立即就会被执行，而且在该方法返回之前其他线程无法并发执行。也就是说，如果直接调用线程对象的 run() 方法，则系统会把线程对象当成一个普通对象，而 run() 方法是一个普通方法，不是线程执行体。

```
1.          import threading
2.
3.          # 定义准备作为线程执行体的 action 函数
4.          def action(max):
5.              for i in range(max):
```

```
                # 直接调用 run()方法时，Thread 的 name 属性返回的是该对象的名字
                # 而不是当前线程的名字
                # 使用 threading.current_thread().name 总是获取当前线程的名字
                print(threading.current_thread().name + " " + str(i))  # ①
        for i in range(100):
            # 调用 Thread 的 currentThread()方法获取当前线程
            print(threading.current_thread().name + " " + str(i))
            if i == 20:
                # 直接调用线程对象的 run()方法
                # 系统会把线程对象当成普通对象，把 run()方法当成普通方法
                # 所以下面两行代码并不会启动两个线程，而是依次执行两个 run()方法
                threading.Thread(target=action,args=(100,)).run()
                threading.Thread(target=action,args=(100,)).run()
```

上面程序在创建线程对象后，直接调用了线程对象的 run() 方法，程序运行的结果是整个程序只有一个主线程。还有一点需要指出，如果直接调用线程对象的 run() 方法，则在 run() 方法中不能直接通过 name 属性（getName() 方法）来获取当前执行线程的名字，而是需要使用 threading.current_thread() 函数先获取当前线程，然后再调用线程对象的 name 属性来获取线程的名字。

通过上面程序不难看出，启动线程的正确方法是调用 Thread 对象的 start() 方法，而不是直接调用 run() 方法，否则就变成单线程程序了。

需要指出的是，在调用线程对象的 run() 方法之后，该线程已经不再处于新建状态，不要再次调用线程对象的 start() 方法。

注意，只能对处于新建状态的线程调用 start() 方法。也就是说，如果程序对同一个线程重复调用 start() 方法，将引发 RuntimeError 异常。

在调用线程对象的 start() 方法之后，该线程立即进入就绪状态（相当于“等待执行”），但该线程并未真正进入运行状态。

线程的运行和阻塞状态

如果处于就绪状态的线程获得了 CPU，开始执行 run() 方法的线程执行体，则该线程处于运行状态。

如果计算机只有一个 CPU，那么在任何时刻只有一个线程处于运行状态。当然，在一个具有多处理器的机器上，将会有多个线程并行（Parallel）执行；当线程数大于处理器数时，依然会存在多个线程在同一个 CPU 上轮换的情况。

当一个线程开始运行后，它不可能一直处于运行状态（除非它的线程执行体足够短，瞬间就执行结束了），线程在运行过程中需要被中断，目的是使其他线程获得执行的机会，线程调度的细节取决于底层平台所采用的策略。对于采用抢占式调度策略的系统而言，系统会给每一个可执行的线程一个小时间段来处理任务；当该时间段用完后，系统就会剥夺该线程所占用的资源，让其他线程获得执行的机会。在选择下一个线程时，系统会考虑线程的优先级。

所有现代的桌面和服务器操作系统都采用抢占式调度策略，但一些小型设备如手机等则可能采用协作式调度策略，在这样的系统中，只有当一个线程调用了它的 sleep() 或 yield() 方

法后才会放弃其所占用的资源（也就是必须由该线程主动放弃其所占用的资源）。

当发生如下情况时，线程将会进入阻塞状态：

- 线程调用 sleep() 方法主动放弃其所占用的处理器资源。
- 线程调用了一个阻塞式 I/O 方法，在该方法返回之前，该线程被阻塞。
- 线程试图获得一个锁对象，但该锁对象正被其他线程所持有。关于锁对象的知识，后面将有更深入的介绍。
- 线程在等待某个通知（Notify）。

当前正在执行的线程被阻塞之后，其他线程就可以获得执行的机会。被阻塞的线程会在合适的时候重新进入就绪状态，注意是就绪状态，而不是运行状态。也就是说，被阻塞线程的阻塞解除后，必须重新等待线程调度器再次调度它。

针对上面几种情况，当发生如下特定的情况时可以解除阻塞，让该线程重新进入就绪状态：

- 调用 sleep() 方法的线程经过了指定的时间。
- 线程调用的阻塞式 I/O 方法已经返回。
- 线程成功地获得了试图获取的锁对象。
- 线程正在等待某个通知时，其他线程发出了一个通知。

图 1 显示了线程状态转换图。

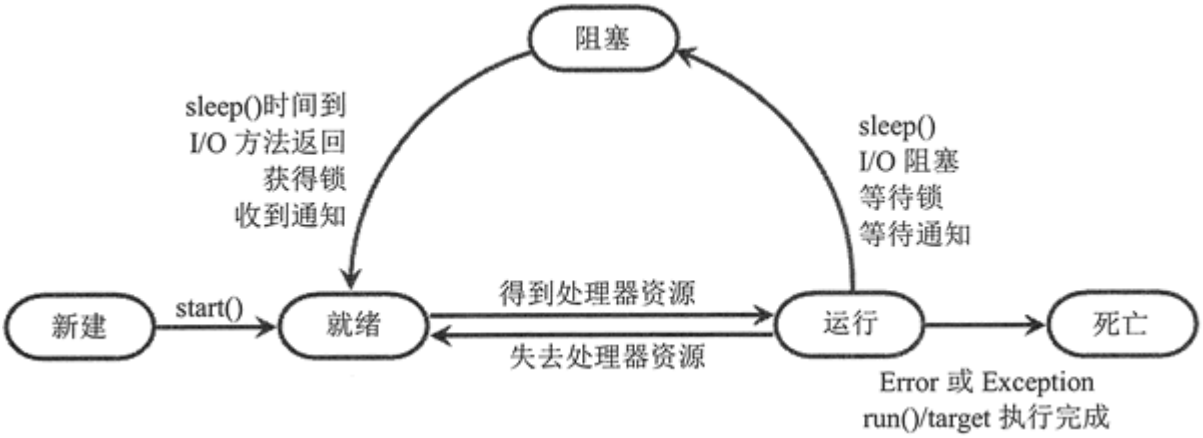


图 1 线程状态转换图

从图 1 中可以看出，线程从阻塞状态只能进入就绪状态，无法直接进入运行状态。就绪和运行状态之间的转换通常不受程序控制，而是由系统线程调度所决定的，当处于就绪状态的线程获得处理器资源时，该线程进入运行状态；当处于运行状态的线程失去处理器资源时，该线程进入就绪状态。

线程死亡

线程会以如下方式结束，结束后就处于死亡状态：

- run() 方法或代表线程执行体的 target 函数执行完成，线程正常结束。
- 线程抛出一个未捕获的 Exception 或 Error。

> 注意，当主线程结束时，其他线程不受任何影响，并不会随之结束。一旦子线程启动起来后，它就拥有和主线程相同的地位，不会受主线程的影响。

为了测试某个线程是否已经死亡，可以调用线程对象的 is_alive() 方法，当线程处于就绪、

运行、阻塞三种状态时，该方法将返回 True；当线程处于新建、死亡两种状态时，该方法将返回 False。

不要试图对一个已经死亡的线程调用 start() 方法使它重新启动，死亡就是死亡，该线程将不可再次作为线程运行。下面程序尝试对处于死亡状态的线程再次调用 start() 方法:

```
import threading

# 定义 action 函数准备作为线程执行体使用
def action(max):
    for  i in range(100):
        print(threading.current_thread().name +  " " + str(i))
# 创建线程对象
sd = threading.Thread(target=action, args=(100,))
for i in range(300):
    # 调用 threading.current_thread()函数获取当前线程
    print(threading.current_thread().name +  " " + str(i))
    if i == 20:
        # 启动线程
        sd.start()
        # 判断启动后线程的 is_alive()值，输出 True
        print(sd.is_alive())
    # 当线程处于新建、死亡两种状态时，is_alive()方法返回 False
    # 当 i > 20 时，该线程肯定已经启动过了，如果 sd.is_alive()为 False 时
    # 那就是死亡状态了
    if i > 20 and not(sd.is_alive()):
        # 试图再次启动该线程
        sd.start()
```

程序试图在线程已死亡的情况下再次调用 start() 方法来启动该线程。运行程序，将引发 RuntimeError 异常，这表明处于死亡状态的线程无法再次运行。

注意，不要对处于死亡状态的线程调用 start() 方法，程序只能对处于新建状态的线程调用 start() 方法，对于处于新建状态的线程两次调用 start() 方法也是错误的。它们都会引发 RuntimeError 异常。

看到这里，可能有读者感觉 Python 的多线程编程有些似曾相识，有点类似于 Java 中关于多线程的介绍，的确如此。实际上，Python 的多线程模型完全是借用 Java 的。因此，如果有很好的 Java 多线程编程基础，那么学习 Python 多线程编程基本上毫无压力，因为它们大致是相同的。

4.Python Thread join()

Thread 提供了让一个线程等待另一个线程完成的 join() 方法。当在某个程序执行流中调用其他线程的 join() 方法时，调用线程将被阻塞，直到被 join() 方法加入的 join 线程执行完成。

join() 方法通常由使用线程的程序调用，以将大问题划分成许多小问题，并为每个小问题分配一个线程。当所有的小问题都得到处理后，再调用主线程来进一步操作。

```
1.          import threading
2.
3.          # 定义 action 函数准备作为线程执行体使用
4.          def action(max):
5.              for i in range(max):
6.                  print(threading.current_thread().name + " " + str(i))
7.
8.          # 启动子线程
9.          threading.Thread(target=action, args=(100,), name="新线程").start()
10.         for i in range(100):
```

```
11.             if i == 20:
12.                 jt = threading.Thread(target=action, args=(100,), name="被 Join
    的线程")
13.                 jt.start()
14.                 # 主线程调用了 jt 线程的 join()方法，主线程
15.                 # 必须等 jt 执行结束才会向下执行
16.                 jt.join()
17.             print(threading.current_thread().name + " " + str(i))
```

上面程序中一共有三个线程，主程序开始时就启动了名为“新线程”的子线程，该子线程将会和主线程并发执行。当主线程的循环变量 i 等于 20 时，启动了名为“被 Join 的线程”的线程，该线程不会和主线程并发执行，主线程必须等该线程执行结束后才可以向下执行。

在名为“被 Join 的线程”的线程执行时，实际上只有两个子线程并发执行，而主线程处于等待状态。运行上面程序，将会看到如图 1 所示的运行效果。

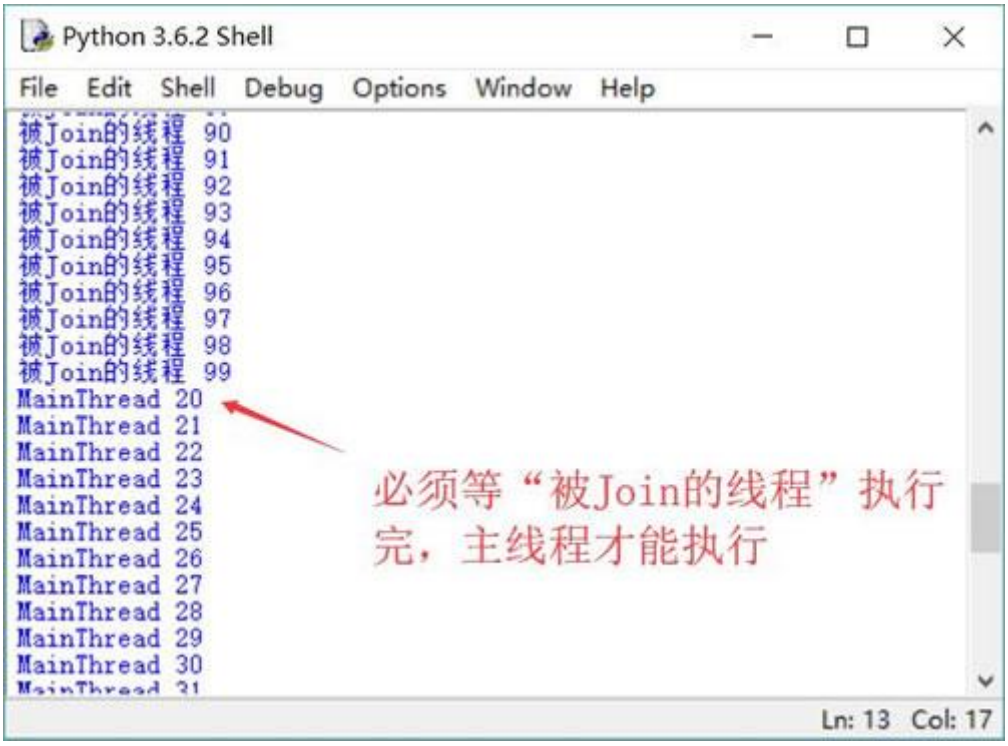


图 1 主线程等待 join 线程的效果

从图 1 中可以看出，主线程执行到 i==20 时，程序启动并 join 了名为“被 Join 的线程”的线程，所以主线程将一直处于阻塞状态，直到名为“被 Join 的线程”的钱程执行完成。

join(timeout=None)方法可以指定一个 timeout 参数，该参数指定等待被 join 的线程的时间最长为 timeout 秒。如果在 timeout 秒内被 join 的线程还没有执行结束，则不再等待。

5.Python 守护线程

有一种线程，它是在后台运行的，它的任务是为其他线程提供服务，这种线程被称为“后台线程（Daemon Thread）”，又称为“守护线程”或“精灵线程”。Python 解释器的垃圾回收线程就是典型的后台线程。

后台线程有一个特征，如果所有的前台线程都死亡了，那么后台线程会自动死亡。

调用 Thread 对象的 daemon 属性可以将指定线程设置成后台线程。下面程序将指定线程设置成后台线程，可以看到当所有的前台线程都死亡后，后台线程随之死亡。当在整个虚拟机中只剩下后台线程时，程序就没有继续运行的必要了，所以程序也就退出了。

```
1.          import threading
2.
3.          # 定义后台线程的线程执行体与普通线程没有任何区别
4.          def action(max):
5.              for i in range(max):
6.                  print(threading.current_thread().name + "  " + str(i))
```

```
7.          t = threading.Thread(target=action, args=(100,), name='后台线程')
8.          # 将此线程设置成后台线程
9.          # 也可在创建 Thread 对象时通过 daemon 参数将其设为后台线程
10.         t.daemon = True
11.         # 启动后台线程
12.         t.start()
13.         for i in range(10):
14.             print(threading.current_thread().name + "  " + str(i))
15.         # -----程序执行到此处，前台线程（主线程）结束------
16.         # 后台线程也应该随之结束
```

上面程序中先将 t 线程设置成后台线程（第 10 行代码），然后启动该线程。本来该线程应该执行到 i 等于 99 时才会结束，但在运行程序时不难发现，该后台线程无法运行到 99，因为当主线程也就是程序中唯一的前台线程运行结束后，程序会主动退出，所以后台线程也就被结束了。

从上面的程序可以看出，主线程默认是前台线程，t 线程默认也是前台线程。但并不是所有的线程默认都是前台线程，有些线程默认就是后台线程。前台线程创建的子线程默认是前台线程，后台线程创建的子线程默认是后台线程。

可见，创建后台线程有两种方式：

1. 主动将线程的 daemon 属性设置为 True。
2. 后台线程启动的线程默认是后台线程。

注意，当前台线程死亡后，Python 解释器会通知后台线程死亡，但是从它接收指令到做出响应需要一定的时间。如果要将某个线程设置为后台线程，则必须在该线程启动之前进行设

置。也就是说，将 daemon 属性设为 True，必须在 start() 方法调用之前进行，否则会引发 RuntimeError 异常。

6.Python sleep()函数用法：线程睡眠

如果需要让当前正在执行的线程暂停一段时间，并进入阻塞状态，则可以通过调用 time 模块的 sleep(secs) 函数来实现。该函数可指定一个 secs 参数，用于指定线程阻塞多少秒。

当前线程调用 sleep() 函数进入阻塞状态后，在其睡眠时间段内，该线程不会获得执行的机会，即使系统中没有其他可执行的线程，处于 sleep() 中的线程也不会执行，因此 sleep() 函数常用来暂停程序的运行。

下面程序调用 sleep() 函数来暂停主线程的执行，因为该程序只有一个主线程，当主线程进入睡眠后，系统没有可执行的线程，所以可以看到程序在 sleep() 函数处暂停。

```
import time

for i in range(10):
    print("当前时间: %s" % time.ctime())
    # 调用 sleep()函数让当前线程暂停 1s
    time.sleep(1)
```

上面程序将当前执行的线程暂停 1s。运行上面的程序，将看到程序依次输出 10 个字符串，输出两个字符串的时间间隔为 1s。

7. Python 互斥锁（Lock）：解决多线程安全问题

多线程的优势在于并发性，即可以同时运行多个任务。但是当线程需要使用共享数据时，也可能会由于数据不同步产生“错误情况”，这是由系统的线程调度具有一定的随机性造成的。

互斥锁的作用就是解决数据不同步问题。关于互斥锁，有一个经典的“银行取钱”问题。银行取钱的基本流程可以分为如下几个步骤：

1. 用户输入账户、密码，系统判断用户的账户、密码是否匹配。
2. 用户输入取款金额。
3. 系统判断账户余额是否大于取款金额。
4. 如果余额大于取款金额，则取款成功；如果余额小于取款金额，则取款失败。

乍一看上去，这确实就是日常生活中的取款流程，这个流程没有任何问题。但一旦将这个流程放在多线程并发的场景下，就有可能出现问题。注意，此处说的是有可能，并不是一定。也许你的程序运行了一百万次都没有出现问题，但没有出现问题并不等于没有问题！

按照上面的流程编写取款程序，并使用两个线程分别模拟两个人使用同一个账户做并发取钱操作。此处忽略检查账户和密码的操作，仅仅模拟后面三步操作。下面先定义一个账户类，该账户类封装了账户编号和余额两个成员变量。

```
class Account:
    # 定义构造器
    def __init__(self, account_no, balance):
        # 封装账户编号、账户余额的两个成员变量
```

```
5.                  self.account_no = account_no
6.                  self.balance = balance
```

接下来程序会定义一个模拟取钱的函数，该函数根据执行账户、取钱数量进行取钱操作，取钱的逻辑是当账户余额不足时无法提取现金，当余额足够时系统吐出钞票，余额减少。

程序的主程序非常简单，仅仅是创建一个账户，并启动两个线程从该账户中取钱。程序如下：

```
1.          import threading
2.          import time
3.          import Account
4.
5.          # 定义一个函数来模拟取钱操作
6.          def draw(account, draw_amount):
7.              # 账户余额大于取钱数目
8.              if account.balance >= draw_amount:
9.                  # 吐出钞票
10.                 print(threading.current_thread().name\
11.                     + "取钱成功！吐出钞票:" + str(draw_amount))
12.         #         time.sleep(0.001)
13.                 # 修改余额
14.                 account.balance -= draw_amount
15.                 print("\t 余额为: " + str(account.balance))
16.             else:
17.                 print(threading.current_thread().name\
18.                     + "取钱失败！余额不足！")
19.         # 创建一个账户
20.         acct = Account.Account("1234567" , 1000)
21.         # 模拟两个线程对同一个账户取钱
22.         threading.Thread(name='甲', target=draw , args=(acct , 800)).start()
23.         threading.Thread(name='乙', target=draw , args=(acct , 800)).start()
```

先不要管程序中第 12 行被注释掉的代码，上面程序是一个非常简单的取钱逻辑，这个取

钱逻辑与实际的取钱操作也很相似。

多次运行上面程序，很有可能都会看到如图 1 所示的错误结果。

图 1 线程安全问题

如图 1 所示的运行结果并不是银行所期望的结果（不过有可能看到正确的运行结果），这正是多线程编程突然出现的“偶然” 错误因为线程调度的不确定性。

假设系统线程调度器在注释代码处暂停，让另一个线程执行（为了强制暂停，只要取消程序中注释代码前的注释即可）。取消注释后，再次运行程序，将总可以看到如图 1 所示的错误结果。

问题出现了，账户余额只有 1000 元时取出了 1600 元，而且账户余额出现了负值，远不是银行所期望的结果。虽然上面程序是人为地使用 `time.sleep(0.001)` 来强制线程调度切换，但这种切换也是完全可能发生的（100000 次操作只要有 1 次出现了错误，那就是由编程错误引起的）。

Python 互斥锁同步线程

之所以出现如图 1 所示的错误结果，是因为 run() 方法的方法体不具有线程安全性，程序

中有两个并发线程在修改 Account 对象，而且系统恰好在注释代码处执行线程切换，切换到另一个修改 Account 对象的线程，所以就出现了问题。

为了解决这个问题，Python 的 threading 模块引入了互斥锁（Lock）。threading 模块提供了 Lock 和 RLock 两个类，它们都提供了如下两个方法来加互斥锁和释放互斥锁：

1. acquire(blocking=True, timeout=-1)：请求对 Lock 或 RLock 加锁，其中 timeout 参数指定加锁多少秒。
2. release()：释放锁。

Lock 和 RLock 的区别如下：

- threading.Lock：它是一个基本的锁对象，每次只能锁定一次，其余的锁请求，需等待锁释放后才能获取。
- threading.RLock：它代表可重入锁（Reentrant Lock）。对于可重入锁，在同一个线程中可以对它进行多次锁定，也可以多次释放。如果使用 RLock，那么 acquire() 和 release() 方法必须成对出现。如果调用了 n 次 acquire() 加锁，则必须调用 n 次 release() 才能释放锁。

由此可见，RLock 锁具有可重入性。也就是说，同一个线程可以对已被加锁的 RLock 锁再次加锁，RLock 对象会维持一个计数器来追踪 acquire() 方法的嵌套调用，线程在每次调用 acquire() 加锁后，都必须显式调用 release() 方法来释放锁。所以，一段被锁保护

的方法可以调用另一个被相同锁保护的方法。

Lock 是控制多个线程对共享资源进行访问的工具。通常，锁提供了对共享资源的独占访问，每次只能有一个线程对 Lock 对象加锁，线程在开始访问共享资源之前应先请求获得 Lock 对象。当对共享资源访问完成后，程序释放对 Lock 对象的锁定。

在实现线程安全的控制中，比较常用的是 RLock。通常使用 RLock 的代码格式如下：

```
class X:
    #定义需要保证线程安全的方法
    def m () :
        #加锁
        self.lock.acquire()
        try :
            #需要保证线程安全的代码
            #...方法体
        #使用 finally 块来保证释放锁
        finally :
            #修改完成，释放锁
            self.lock.release()
```

使用 RLock 对象来控制线程安全，当加锁和释放锁出现在不同的作用范围内时，通常建议使用 finally 块来确保在必要时释放锁。

通过使用 Lock 对象可以非常方便地实现线程安全的类，线程安全的类具有如下特征：

- 该类的对象可以被多个线程安全地访问。
- 每个线程在调用该对象的任意方法之后，都将得到正确的结果。
- 每个线程在调用该对象的任意方法之后，该对象都依然保持合理的状态。

总的来说，不可变类总是线程安全的，因为它的对象状态不可改变；但可变对象需要额外的方法来保证其线程安全。例如，上面的 Account 就是一个可变类，它的 self.account_no 和 self._balance（为了更好地封装，将 balance 改名为 _balance）两个成员变量都可以被改变，当两个钱程同时修改 Account 对象的 self._balance 成员变量的值时，程序就出现了异常。下面将 Account 类对 self.balance 的访问设置成线程安全的，那么只需对修改 self.balance 的方法增加线程安全的控制即可。

将 Account 类改为如下形式，它就是线程安全的：

```
import threading
import time

class Account:
    # 定义构造器
    def __init__(self, account_no, balance):
        # 封装账户编号、账户余额的两个成员变量
        self.account_no = account_no
        self._balance = balance
        self.lock = threading.RLock()

    # 因为账户余额不允许随便修改，所以只为 self._balance 提供 getter 方法
    def getBalance(self):
        return self._balance
    # 提供一个线程安全的 draw()方法来完成取钱操作
```

```
16.             def draw(self, draw_amount):
17.                 # 加锁
18.                 self.lock.acquire()
19.                 try:
20.                     # 账户余额大于取钱数目
21.                     if self._balance >= draw_amount:
22.                         # 吐出钞票
23.                         print(threading.current_thread().name\
24.                             + "取钱成功！吐出钞票:" + str(draw_amount))
25.                         time.sleep(0.001)
26.                         # 修改余额
27.                         self._balance -= draw_amount
28.                         print("\t 余额为: " + str(self._balance))
29.                     else:
30.                         print(threading.current_thread().name\
31.                             + "取钱失败！余额不足！")
32.                 finally:
33.                     # 修改完成，释放锁
34.                     self.lock.release()
```

上面程序中的定义了一个 RLock 对象。在程序中实现 draw() 方法时，进入该方法开始执行后立即请求对 RLock 对象加锁，当执行完 draw() 方法的取钱逻辑之后，程序使用 finally 块来确保释放锁。

程序中 RLock 对象作为同步锁，线程每次开始执行 draw() 方法修改 self.balance 时，都必须先对 RLock 对象加锁。当该线程完成对 self._balance 的修改，将要退出 draw() 方法时，则释放对 RLock 对象的锁定。这样的做法完全符合“加锁→修改→释放锁”的安全访问逻辑。

当一个线程在 draw() 方法中对 RLock 对象加锁之后，其他线程由于无法获取对 RLock

对象的锁定，因此它们同时执行 draw() 方法对 self._balance 进行修改。这意味着，并发线程在任意时刻只有一个线程可以进入修改共享资源的代码区（也被称为临界区），所以在同一时刻最多只有一个线程处于临界区内，从而保证了线程安全。

为了保证 Lock 对象能真正“锁定”它所管理的 Account 对象，程序会被编写成每个 Account 对象有一个对应的 Lock（就像一个房间有一个锁一样）。

上面的 Account 类增加了一个代表取钱的 draw() 方法，并使用 Lock 对象保证该 draw() 方法的线程安全，而且取消了 setBalance() 方法（避免程序直接修改 self._balance 成员变量）,因此线程执行体只需调用 Account 对象的 draw() 方法即可执行取钱操作。

下面程序创建并启动了两个取钱线程：

```
1.          import threading
2.          import Account
3.
4.          # 定义一个函数来模拟取钱操作
5.          def draw(account, draw_amount):
6.              # 直接调用 account 对象的 draw()方法来执行取钱操作
7.              account.draw(draw_amount)
8.          # 创建一个账户
9.          acct = Account.Account("1234567", 1000)
10.         # 模拟两个线程对同一个账户取钱
11.         threading.Thread(name='甲', target=draw , args=(acct , 800)).start()
12.         threading.Thread(name='乙', target=draw , args=(acct , 800)).start()
```

上面程序中代表线程执行体的 draw() 函数无须自己实现取钱操作，而是直接调用

account 的 draw() 方法来执行取钱操作。由于 draw() 方法已经使用 RLock 对象实现了线程安全，因此上面程序就不会导致线程安全问题。

多次重复运行上面程序，总可以看到如图 2 所示的运行结果。

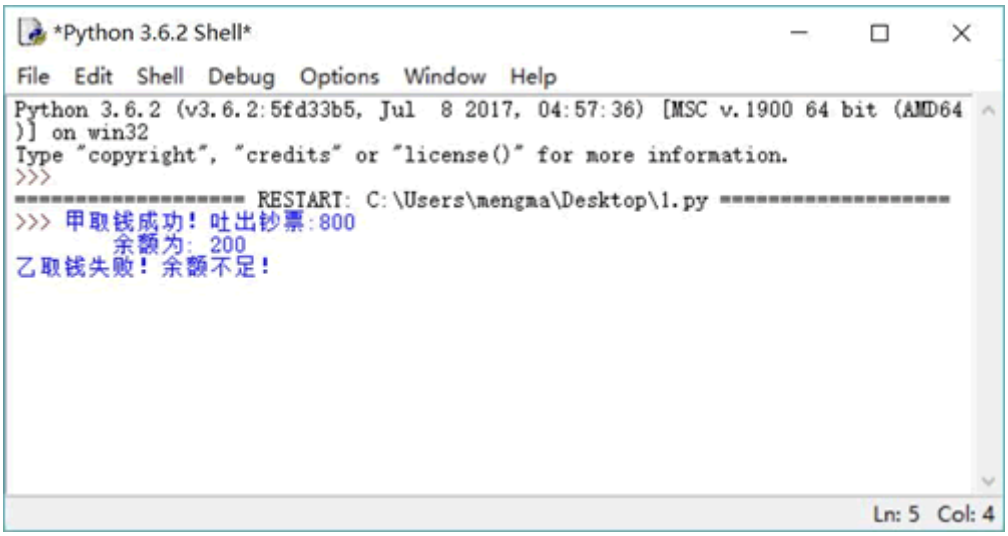


图 2 使用互斥锁保证线程安全

可变类的线程安全是以降低程序的运行效率作为代价的，为了减少线程安全所带来的负面影响，程序可以采用如下策略：

- 不要对线程安全类的所有方法都进行同步，只对那些会改变竞争资源（竞争资源也就是共享资源）的方法进行同步。例如，上面 Account 类中的 account_no 实例变量就无须同步，所以程序只对 draw() 方法进行了同步控制。
- 如果可变类有两种运行环境，单线程环境和多线程环境，则应该为该可变类提供两种版本，即线程不安全版本和线程安全版本。在单线程环境中使用钱程不安全版本以保证性能，在多线程环境中使用线程安全版本。

8.Python 线程死锁

什么是死锁，如何避免死锁（4 种方法）

当两个线程相互等待对方释放同步监视器时就会发生死锁。Python 解释器没有监测， 也没有采取措施来处理死锁情况，所以在进行多线程编程时应该采取措施避免出现死锁。

> 一旦出现死锁，整个程序既不会发生任何异常，也不会给出任何提示，只是所有线程都处于阻塞状态，无法继续。

死锁是很容易发生的，尤其是在系统中出现多个同步监视器的情况下，如下程序将会出现死锁：

```
import threading
import time

class A:
    def __init__(self):
        self.lock = threading.RLock()
    def foo(self, b):
        try:
            self.lock.acquire()
            print("当前线程名: " + threading.current_thread().name\
                + " 进入了 A 实例的 foo()方法" )      # ①
            time.sleep(0.2)
            print("当前线程名: " + threading.current_thread().name\
                + " 企图调用 B 实例的 last()方法")    # ③
            b.last()
        finally:
            self.lock.release()
    def last(self):
        try:
```

```
                    self.lock.acquire()
                    print("进入了 A 类的 last()方法内部")
                finally:
                    self.lock.release()
        class B:
            def __init__(self):
                self.lock = threading.RLock()
            def bar(self, a):
                try:
                    self.lock.acquire()
                    print("当前线程名: " + threading.current_thread().name\
                        + " 进入了 B 实例的 bar()方法" )    # ②
                    time.sleep(0.2)
                    print("当前线程名: " + threading.current_thread().name\
                        + " 企图调用 A 实例的 last()方法")  # ④
                    a.last()
                finally:
                    self.lock.release()
            def last(self):
                try:
                    self.lock.acquire()
                    print("进入了 B 类的 last()方法内部")
                finally:
                    self.lock.release()
        a = A()
        b = B()
        def init():
            threading.current_thread().name = "主线程"
            # 调用 a 对象的 foo()方法
            a.foo(b)
            print("进入了主线程之后")
        def action():
            threading.current_thread().name = "副线程"
            # 调用 b 对象的 bar()方法
            b.bar(a)
            print("进入了副线程之后")
```

```
56.         # 以 action 为 target 启动新线程
57.         threading.Thread(target=action).start()
58.         # 调用 init()函数
59.         init()
```

运行上面程序，将会看到如图 1 所示的效果。

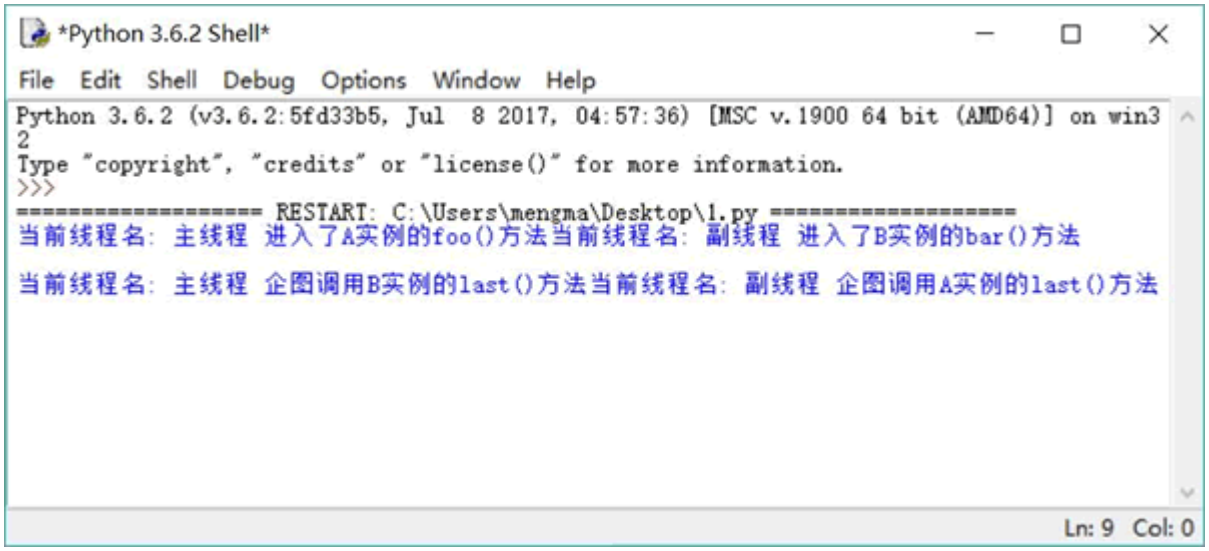


图 1 死锁效果

从图 1 中可以看出，程序既无法向下执行，也不会抛出任何异常，就一直“僵持”着。究其原因，是因为上面程序中 A 对象和 B 对象的方法都是线程安全的方法。

程序中有两个线程执行，副线程的线程执行体是 action() 函数，主线程的线程执行体是 init() 函数(主程序调用了 init() 函数)。其中在 action() 函数中让 B 对象调用 bar() 方法，而在 init() 函数中让 A 对象调用 foo() 方法。

图 1 显示 action() 函数先执行，调用了 B 对象的 bar() 方法，在进入 bar() 方法之前，该线程对 B 对象的 Lock 加锁(当程序执行到 ② 号代码时，副线程暂停 0.2s ）;CPU 切换到执行另一个线程，让 A 对象执行 foo() 方法，所以看到主线程开始执行 A 实例的 foo() 方法，在进入 foo() 方法之前，该线程对 A 对象的 Lock 加锁(当程序执行到 ① 号代码时，主线程也暂停 0.2s ）。

接下来副线程会先醒过来，继续向下执行，直到执行到 ④ 号代码处希望调用 A 对象的 last() 方法（在执行该方法之前，必须先对 A 对象的 Lock 加锁），但此时主线程正保持着 A 对象的 Lock 的锁定，所以副线程被阻塞。

接下来主线程应该也醒过来了，继续向下执行，直到执行到 ③ 号代码处希望调用 B 对象的 last() 方法（在执行该方法之前，必须先对 B 对象的 Lock 加锁），但此时副线程没有释放对 B 对象的 Lock 的锁定。

至此，就出现了主线程保持着 A 对象的锁，等待对 B 对象加锁，而副线程保持着 B 对象的锁，等待对 A 对象加锁，两个线程互相等待对方先释放锁，所以就出现了死锁。

死锁是不应该在程序中出现的，在编写程序时应该尽量避免出现死锁。下面有几种常见的方式用来解决死锁问题：

1. 避免多次锁定。尽量避免同一个线程对多个 Lock 进行锁定。例如上面的死锁程序，主线程要对 A、B 两个对象的 Lock 进行锁定，副线程也要对 A、B 两个对象的 Lock 进行锁定，这就埋下了导致死锁的隐患。

2. 具有相同的加锁顺序。如果多个线程需要对多个 Lock 进行锁定，则应该保证它们以相同的顺序请求加锁。比如上面的死锁程序，主线程先对 A 对象的 Lock 加锁，再对 B 对象的 Lock 加锁；而副线程则先对 B 对象的 Lock 加锁，再对 A 对象的 Lock 加锁。这种加锁顺序很容易形成嵌套锁定，进而导致死锁。如果让主线程、副线程按照相同的顺序加锁，就可以避免这个问题。
3. 使用定时锁。程序在调用 acquire() 方法加锁时可指定 timeout 参数，该参数指定超过 timeout 秒后会自动释放对 Lock 的锁定，这样就可以解开死锁了。
4. 死锁检测。死锁检测是一种依靠算法机制来实现的死锁预防机制，它主要是针对那些不可能实现按序加锁，也不能使用定时锁的场景的。

9.Python condition 实现线程通信

当线程在系统中运行时，线程的调度具有一定的透明性，通常程序无法准确控制线程的轮换执行，如果有需要，Python 可通过线程通信来保证线程协调运行。

假设系统中有两个线程，这两个线程分别代表存款者和取钱者，现在假设系统有一种特殊的要求，即要求存款者和取钱者不断地重复存款、取钱的动作，而且要求每当存款者将钱存入指定账户后，取钱者就立即取出该笔钱。不允许存款者连续两次存钱，也不允许取钱者连续两次取钱。

为了实现这种功能，可以借助于 Condition 对象来保持协调。使用 Condition 可以让那些已经得到 Lock 对象却无法继续执行的线程释放 Lock 对象，Condition 对象也可以唤醒

其他处于等待状态的线程。

将 Condition 对象与 Lock 对象组合使用，可以为每个对象提供多个等待集(wait-set)。因此，Condition 对象总是需要有对应的 Lock 对象。从 Condition 的构造器 __init__(self, lock=None) 可以看出，程序在创建 Condition 时可通过 lock 参数传入要绑定的 Lock 对象；如果不指定 lock 参数，在创建 Condition 时它会自动创建一个与之绑定的 Lock 对象。

Condition 类提供了如下几个方法：

- acquire([timeout])/release()：调用 Condition 关联的 Lock 的 acquire() 或 release() 方法。
- wait([timeout])：导致当前线程进入 Condition 的等待池等待通知并释放锁，直到其他线程调用该 Condition 的 notify() 或 notify_all() 方法来唤醒该线程。在调用该 wait() 方法时可传入一个 timeout 参数，指定该线程最多等待多少秒。
- notify()：唤醒在该 Condition 等待池中的单个线程并通知它，收到通知的线程将自动调用 acquire() 方法尝试加锁。如果所有线程都在该 Condition 等待池中等待，则会选择唤醒其中一个线程，选择是任意性的。
- notify_all()：唤醒在该 Condition 等待池中等待的所有线程并通知它们。

本例程序中，可以通过一个旗标来标识账户中是否已有存款，当旗标为 False 时，表明账户中没有存款，存款者线程可以向下执行，当存款者把钱存入账户中后，将旗标设为 True，

并调用 Condition 的 notify() 或 notify_all() 方法来唤醒其他线程。

当存款者线程进入线程体后，如果旗标为 True，就调用 Condition 的 wait() 方法让该线程等待。当旗标为 True 时，表明账户中已经存入了钱，取钱者线程可以向下执行，当取钱者把钱从账户中取出后，将旗标设为 False，并调用 Condition 的 notify() 或 notify_all() 方法来唤醒其他线程；当取钱者线程进入线程体后，如果旗标为 False，就调用 wait() 方法让该线程等待。

本程序为 Account 类提供了 draw() 和 deposit() 两个方法，分别对应于该账户的取钱和存款操作。因为这两个方法可能需要并发修改 Account 类的 self.balance 成员变量的值，所以它们都使用 Lock 来控制线程安全。除此之外，这两个方法还使用了 Condition 的 wait() 和 notify_all() 来控制线程通信。

```
import threading

class Account:
    # 定义构造器
    def __init__(self, account_no, balance):
        # 封装账户编号、账户余额的两个成员变量
        self.account_no = account_no
        self._balance = balance
        self.cond = threading.Condition()
        # 定义代表是否已经存钱的旗标
        self._flag = False

    # 因为账户余额不允许随便修改，所以只为 self._balance 提供 getter 方法
    def getBalance(self):
        return self._balance
    # 提供一个线程安全的 draw()方法来完成取钱操作
```

```
17.             def draw(self, draw_amount):
18.                 # 加锁,相当于调用 Condition 绑定的 Lock 的 acquire()
19.                 self.cond.acquire()
20.                 try:
21.                     # 如果 self._flag 为假，表明账户中还没有人存钱进去，取钱方法
    阻塞
22.                     if not self._flag:
23.                         self.cond.wait()
24.                     else:
25.                         # 执行取钱操作
26.                         print(threading.current_thread().name
27.                             + " 取钱:" +  str(draw_amount))
28.                         self._balance -= draw_amount
29.                         print("账户余额为: " + str(self._balance))
30.                         # 将标识账户是否已有存款的旗标设为 False
31.                         self._flag = False
32.                         # 唤醒其他线程
33.                         self.cond.notify_all()
34.                 # 使用 finally 块来释放锁
35.                 finally:
36.                     self.cond.release()
37.             def deposit(self, deposit_amount):
38.                 # 加锁,相当于调用 Condition 绑定的 Lock 的 acquire()
39.                 self.cond.acquire()
40.                 try:
41.                     # 如果 self._flag 为真，表明账户中已有人存钱进去，存钱方法阻
    塞
42.                     if self._flag:              # ①
43.                         self.cond.wait()
44.                     else:
45.                         # 执行存款操作
46.                         print(threading.current_thread().name\
47.                             + " 存款:" +  str(deposit_amount))
48.                         self._balance += deposit_amount
49.                         print("账户余额为: " + str(self._balance))
50.                         # 将表示账户是否已有存款的旗标设为 True
```

```
                    self._flag = True
                    # 唤醒其他线程
                    self.cond.notify_all()
                # 使用 finally 块来释放锁
                finally:
                    self.cond.release()
```

上面程序使用 Condition 的 wait() 和 notify_all() 方法进行控制，对存款者线程而言，当程序进入 deposit() 方法后，如果 self._flag 为 True，则表明账户中已有存款，程序调用 Condition 的 wait() 方法被阻塞；否则，程序向下执行存款操作，当存款操作执行完成后，系统将 self._flag 设为 True，然后调用 notify_all() 来唤醒其他被阻塞的线程。如果系统中有存款者线程，存款者线程也会被唤醒，但该存款者线程执行到 ① 号代码处时再次进入阻塞状态，只有执行 draw() 方法的取钱者线程才可以向下执行。同理，取钱者线程的运行流程也是如此。

程序中的存款者线程循环 100 次重复存款，而取钱者线程则循环 100 次重复取钱，存款者线程和取钱者线程分别调用 Account 对象的 deposit()、draw() 方法来实现。主程序可以启动任意多个“存款”线程和“取钱”线程，可以看到所有的“取钱”线程必须等“存款”线程存钱后才可以向下执行，而“存款”线程也必须等“取钱”线程取钱后才可以向下执行。主程序代码如下：

```
import threading
import Account

#  定义一个函数，模拟重复 max 次执行取钱操作
def draw_many(account, draw_amount, max):
    for i in range(max):
        account.draw(draw_amount)
```

```
        #  定义一个函数，模拟重复 max 次执行存款操作
        def deposit_many(account, deposit_amount, max):
            for i in range(max):
                account.deposit(deposit_amount)
        # 创建一个账户
        acct = Account.Account("1234567" , 0)
        # 创建、并启动一个“取钱”线程
        threading.Thread(name="取钱者", target=draw_many,
            args=(acct, 800, 100)).start()
        # 创建、并启动一个“存款”线程
        threading.Thread(name="存款者甲", target=deposit_many,
            args=(acct , 800, 100)).start();
        threading.Thread(name="存款者乙", target=deposit_many,
            args=(acct , 800, 100)).start()
        threading.Thread(name="存款者丙", target=deposit_many,
            args=(acct , 800, 100)).start()
```

运行该程序，可以看到存款者线程、取钱者线程交替执行的情形，每当存款者向账户中存入 800 元之后，取钱者线程就立即从账户中取出这笔钱。存款完成后账户余额总是 800 元，取钱结束后账户余额总是 0 元。

运行该程序，将会看到如图 1 所示的结果。

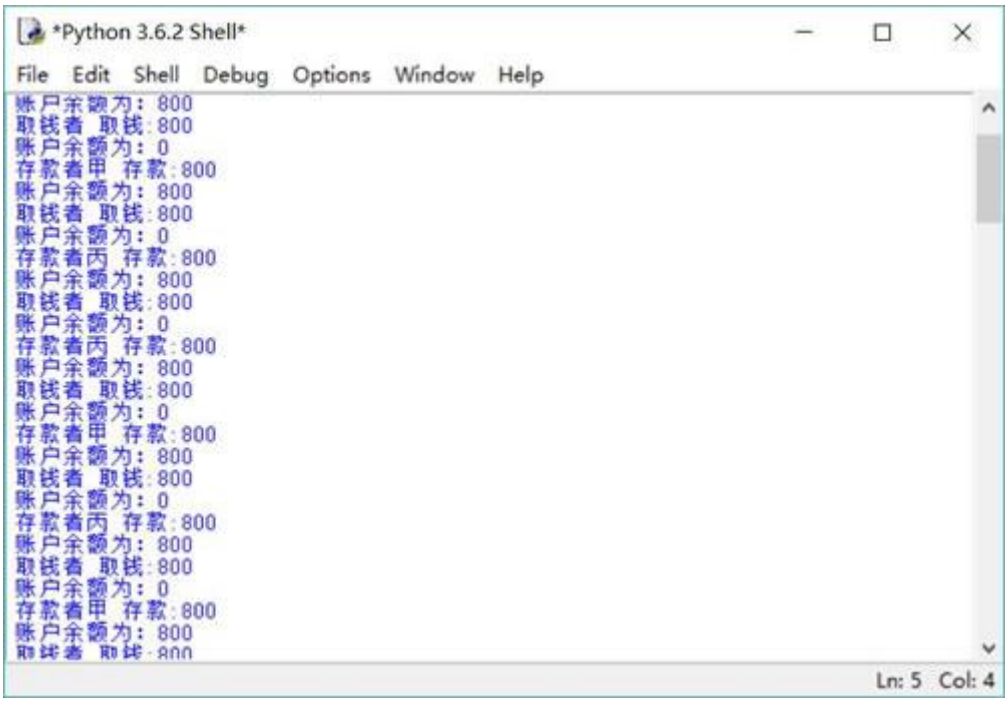


图 1 线程通信的效果

从图 1 中可以看出，3 个存款者线程随机地向账户中存钱，只有 1 个取钱者线程执行取钱操作。只有当取钱者线程取钱后，存款者线程才可以存钱；同理，只有等存款者线程存钱后，取钱者线程才可以取钱。

图 1 显示程序最后被阻塞无法继续向下执行。这是因为 3 个存款者线程共有 300 次尝试存钱操作，但 1 个取钱者线程只有 100 次尝试取钱操作，所以程序最后被阻塞。

如图 1 所示的阻塞并不是死锁，对于这种情况，取钱者线程已经执行结束，而存款者线程只是在等待其他线程来取钱而已，并不是等待其他线程释放同步监视器。不要把死锁和程序阻塞等同起来。

10.Python Queue 队列实现线程通信

queue 模块下提供了几个阻塞队列，这些队列主要用于实现线程通信。在 queue 模块下主要提供了三个类，分别代表三种队列，它们的主要区别就在于进队列、出队列的不同。

关于这三个队列类的简单介绍如下：

1. queue.Queue(maxsize=0)：代表 FIFO（先进先出）的常规队列，maxsize 可以限制队列的大小。如果队列的大小达到队列的上限，就会加锁，再次加入元素时就会被阻塞，直到队列中的元素被消费。如果将 maxsize 设置为 0 或负数，则该队列的大小就是无限制的。
2. queue.LifoQueue(maxsize=0)：代表 LIFO（后进先出）的队列，与 Queue 的区别就是出队列的顺序不同。
3. PriorityQueue(maxsize=0)：代表优先级队列，优先级最小的元素先出队列。

这三个队列类的属性和方法基本相同， 它们都提供了如下属性和方法：

- Queue.qsize()：返回队列的实际大小，也就是该队列中包含几个元素。
- Queue.empty()：判断队列是否为空。
- Queue.full()：判断队列是否已满。
- Queue.put(item, block=True, timeout=None)：向队列中放入元素。如果队列已满，且 block 参数为 True（阻塞），当前线程被阻塞，timeout 指定阻塞时间，如果将

timeout 设置为 None，则代表一直阻塞，直到该队列的元素被消费；如果队列已满，且 block 参数为 False（不阻塞），则直接引发 queue.FULL 异常。

- Queue.put_nowait(item)：向队列中放入元素，不阻塞。相当于在上一个方法中将 block 参数设置为 False。
- Queue.get(item, block=True, timeout=None)：从队列中取出元素（消费元素）。如果队列已满，且 block 参数为 True（阻塞），当前线程被阻塞，timeout 指定阻塞时间，如果将 timeout 设置为 None，则代表一直阻塞，直到有元素被放入队列中； 如果队列已空，且 block 参数为 False（不阻塞），则直接引发 queue.EMPTY 异常。
- Queue.get_nowait(item)：从队列中取出元素，不阻塞。相当于在上一个方法中将 block 参数设置为 False。

下面以普通的 Queue 为例介绍阻塞队列的功能和用法。首先用一个最简单的程序来测试 Queue 的 put() 和 get() 方法。

```
import queue

# 定义一个长度为 2 的阻塞队列
bq = queue.Queue(2)
bq.put("Python")
bq.put("Python")
print("1111111111")
bq.put("Python")  # ① 阻塞线程
print("2222222222")
```

上面程序先定义了一个大小为 2 的 Queue，程序先向该队列中放入两个元素，此时队列还没有满，两个元素都可以被放入。当程序试图放入第三个元素时，如果使用 put() 方法

尝试放入元素将会阻塞线程，如上面程序中 ① 号代码所示。

与此类似的是，在 Queue 已空的情况下，程序使用 get() 方法尝试取出元素将会阻塞线程。

在掌握了 Queue 阻塞队列的特性之后，在下面程序中就可以利用 Queue 来实现线程通信了。

```
1.          import threading
2.          import time
3.          import queue
4.
5.          def product(bq):
6.              str_tuple = ("Python", "Kotlin", "Swift")
7.              for i in range(99999):
8.                  print(threading.current_thread().name + "生产者准备生产元组元
    素！")
9.                  time.sleep(0.2);
10.                 # 尝试放入元素，如果队列已满，则线程被阻塞
11.                 bq.put(str_tuple[i % 3])
12.                 print(threading.current_thread().name \
13.                     + "生产者生产元组元素完成！")
14.         def consume(bq):
15.             while True:
16.                 print(threading.current_thread().name + "消费者准备消费元组元
    素！")
17.                 time.sleep(0.2)
18.                 # 尝试取出元素，如果队列已空，则线程被阻塞
19.                 t = bq.get()
20.                 print(threading.current_thread().name \
21.                     + "消费者消费[ %s ]元素完成！" % t)
22.         # 创建一个容量为 1 的 Queue
```

```
23.         bq = queue.Queue(maxsize=1)
24.         # 启动 3 个生产者线程
25.         threading.Thread(target=product, args=(bq, )).start()
26.         threading.Thread(target=product, args=(bq, )).start()
27.         threading.Thread(target=product, args=(bq, )).start()
28.         # 启动一个消费者线程
29.         threading.Thread(target=consume, args=(bq, )).start()
```

上面程序启动了三个生产者线程向 Queue 队列中放入元素，启动了三个消费者线程从 Queue 队列中取出元素。本程序中 Queue 队列的大小为 1，因此三个生产者线程无法连续放入元素，必须等待消费者线程取出一个元素后，其中的一个生产者线程才能放入一个元素。

运行该程序，将会看到如图 1 所示的结果。

图 1 使用 Queue 控制线程通信

从图 1 可以看出，三个生产者线程都想向 Queue 中放入元素，但只要其中一个生产者线

程向该队列中放入元素之后，其他生产者线程就必须等待，等待消费者线程取出 Queue 队列中的元素

11.Python Event 实现线程通信

Event 是一种非常简单的线程通信机制，一个线程发出一个 Event，另一个线程可通过该 Event 被触发。

Event 本身管理一个内部旗标，程序可以通过 Event 的 set() 方法将该旗标设置为 True，也可以调用 clear() 方法将该旗标设置为 False。程序可以调用 wait() 方法来阻塞当前线程，直到 Event 的内部旗标被设置为 True。

Event 提供了如下方法：

- is_set()：该方法返回 Event 的内部旗标是否为 True。
- set()：该方法将会把 Event 的内部旗标设置为 True，并唤醒所有处于等待状态的线程。
- clear()：该方法将 Event 的内部旗标设置为 False，通常接下来会调用 wait() 方法来阻塞当前线程。
- wait(timeout=None)：该方法会阻塞当前线程。

下面程序示范了 Event 最简单的用法：

```
1.          import threading
```

```
import time

event = threading.Event()
def cal(name):
    # 等待事件，进入等待阻塞状态
    print('%s 启动' % threading.currentThread().getName())
    print('%s 准备开始计算状态' % name)
    event.wait()    # ①
    # 收到事件后进入运行状态
    print('%s 收到通知了.' % threading.currentThread().getName())
    print('%s 正式开始计算！'% name)
# 创建并启动两条，它们都会①号代码处等待
threading.Thread(target=cal, args=('甲', )).start()
threading.Thread(target=cal, args=("乙", )).start()
time.sleep(2)    #②
print('------------------')
# 发出事件
print('主线程发出事件')
event.set()
```

上面程序以 cal() 函数为 target，创建并启动了两个线程。由于 cal() 函数在 ① 号代码处调用了 Event 的 wait()，因此两个线程执行到 ① 号代码处都会进入阻塞状态；即使主线程在 ② 号代码处被阻塞，两个子线程也不会向下执行。

直到主程序执行到最后一行，程序调用了 Event 的 set() 方法将 Event 的内部旗标设直为 True，并唤醒所有等待的线程，这两个线程才能向下执行。

运行上面程序，将看到如下输出结果：

```
Thread-1 启动
甲 准备开始计算状态
Thread-2 启动
乙 准备开始计算状态
------------------
主线程发出事件
Thread-1 收到通知了.
Thread-2 收到通知了.
甲 正式开始计算！
乙 正式开始计算！
```

上面程序还没有使用 Event 的内部旗标，如果结合 Event 的内部旗标，同样可实现前面的 Account 的生产者-消费者效果：存钱线程（生产者）存钱之后，必须等取钱线程（消费者）取钱之后才能继续向下执行。

Event 实际上优点类似于 Condition 和旗标的结合体，但 Event 本身并不带 Lock 对象，因此如果要实现线程同步，还需要额外的 Lock 对象。

下面是使用 Event 改写后的 Account：

纯文本复制

```
import threading

class Account:
    # 定义构造器
    def __init__(self, account_no, balance):
```

```
6.                  # 封装账户编号、账户余额的两个成员变量
7.                  self.account_no = account_no
8.                  self._balance = balance
9.                  self.lock = threading.Lock()
10.                 self.event = threading.Event()
11.             # 因为账户余额不允许随便修改，所以只为 self._balance 提供 getter 方法
12.             def getBalance(self):
13.                 return self._balance
14.             # 提供一个线程安全的 draw()方法来完成取钱操作
15.             def draw(self, draw_amount):
16.                 # 加锁
17.                 self.lock.acquire()
18.                 # 如果 Event 内部旗标为 True，表明账户中已有人存钱进去
19.                 if self.event.is_set():
20.                     # 执行取钱操作
21.                     print(threading.current_thread().name
22.                         + " 取钱:" +  str(draw_amount))
23.                     self._balance -= draw_amount
24.                     print("账户余额为: " + str(self._balance))
25.                     # 将 Event 内部旗标设为 False
26.                     self.event.clear()
27.                     # 释放加锁
28.                     self.lock.release()
29.                     # 阻塞当前线程阻塞
30.                     self.event.wait()
31.                 else:
32.                     # 释放加锁
33.                     self.lock.release()
34.                     # 阻塞当前线程阻塞
35.                     self.event.wait()
36.             def deposit(self, deposit_amount):
37.                 # 加锁
38.                 self.lock.acquire()
39.                 # 如果 Event 内部旗标为 False，表明账户中还没有人存钱进去
40.                 if not self.event.is_set():
41.                     # 执行存款操作
```

```
                print(threading.current_thread().name\
                    + " 存款:" +  str(deposit_amount))
                self._balance += deposit_amount
                print("账户余额为: " + str(self._balance))
                # 将 Event 内部旗标设为 True
                self.event.set()
                # 释放加锁
                self.lock.release()
                # 阻塞当前线程阻塞
                self.event.wait()
            else:
                # 释放加锁
                self.lock.release()
                # 阻塞当前线程阻塞
                self.event.wait()
```

12.Python 线程池

系统启动一个新线程的成本是比较高的，因为它涉及与操作系统的交互。在这种情形下，使用线程池可以很好地提升性能，尤其是当程序中需要创建大量生存期很短暂的线程时，更应该考虑使用线程池。

线程池在系统启动时即创建大量空闲的线程，程序只要将一个函数提交给线程池，线程池就会启动一个空闲的线程来执行它。当该函数执行结束后，该线程并不会死亡，而是再次返回到线程池中变成空闲状态，等待执行下一个函数。

此外，使用线程池可以有效地控制系统中并发线程的数量。当系统中包含有大量的并发线程时，会导致系统性能急剧下降，甚至导致 Python 解释器崩溃，而线程池的最大线程数参数

可以控制系统中并发线程的数量不超过此数。

线程池的使用

线程池的基类是 concurrent.futures 模块中的 Executor，Executor 提供了两个子类，即 ThreadPoolExecutor 和 ProcessPoolExecutor，其中 ThreadPoolExecutor 用于创建线程池，而 ProcessPoolExecutor 用于创建进程池。

如果使用线程池/进程池来管理并发编程，那么只要将相应的 task 函数提交给线程池/进程池，剩下的事情就由线程池/进程池来搞定。

Exectuor 提供了如下常用方法：

- submit(fn, *args, **kwargs)：将 fn 函数提交给线程池。*args 代表传给 fn 函数的参数，*kwargs 代表以关键字参数的形式为 fn 函数传入参数。
- map(func, *iterables, timeout=None, chunksize=1)：该函数类似于全局函数 map(func, *iterables)，只是该函数将会启动多个线程，以异步方式立即对 iterables 执行 map 处理。
- shutdown(wait=True)：关闭线程池。

程序将 task 函数提交（submit）给线程池后，submit 方法会返回一个 Future 对象，Future 类主要用于获取线程任务函数的返回值。由于线程任务会在新线程中以异步方式执行，因此，线程执行的函数相当于一个“将来完成”的任务，所以 Python 使用 Future 来

代表。

> 实际上，在 Java 的多线程编程中同样有 Future，此处的 Future 与 Java 的 Future 大同小异。

Future 提供了如下方法：

- cancel()：取消该 Future 代表的线程任务。如果该任务正在执行，不可取消，则该方法返回 False；否则，程序会取消该任务，并返回 True。
- cancelled()：返回 Future 代表的线程任务是否被成功取消。
- running()：如果该 Future 代表的线程任务正在执行、不可被取消，该方法返回 True。
- done()：如果该 Funture 代表的线程任务被成功取消或执行完成，则该方法返回 True。
- result(timeout=None)：获取该 Future 代表的线程任务最后返回的结果。如果 Future 代表的线程任务还未完成，该方法将会阻塞当前线程，其中 timeout 参数指定最多阻塞多少秒。
- exception(timeout=None)：获取该 Future 代表的线程任务所引发的异常。如果该任务成功完成，没有异常，则该方法返回 None。
- add_done_callback(fn)：为该 Future 代表的线程任务注册一个“回调函数”，当该任务成功完成时，程序会自动触发该 fn 函数。

在用完一个线程池后，应该调用该线程池的 shutdown() 方法，该方法将启动线程池的关

闭序列。调用 shutdown() 方法后的线程池不再接收新任务，但会将以前所有的已提交任务执行完成。当线程池中的所有任务都执行完成后，该线程池中的所有线程都会死亡。

使用线程池来执行线程任务的步骤如下：

1. 调用 ThreadPoolExecutor 类的构造器创建一个线程池。
2. 定义一个普通函数作为线程任务。
3. 调用 ThreadPoolExecutor 对象的 submit() 方法来提交线程任务。
4. 当不想提交任何任务时，调用 ThreadPoolExecutor 对象的 shutdown() 方法来关闭线程池。

下面程序示范了如何使用线程池来执行线程任务：

```
from concurrent.futures import ThreadPoolExecutor
import threading
import time

# 定义一个准备作为线程任务的函数
def action(max):
    my_sum = 0
    for i in range(max):
        print(threading.current_thread().name + '  ' + str(i))
        my_sum += i
    return my_sum
# 创建一个包含 2 条线程的线程池
pool = ThreadPoolExecutor(max_workers=2)
# 向线程池提交一个 task, 50 会作为 action()函数的参数
future1 = pool.submit(action, 50)
# 向线程池再提交一个 task, 100 会作为 action()函数的参数
```

```
17.         future2 = pool.submit(action, 100)
18.         # 判断 future1 代表的任务是否结束
19.         print(future1.done())
20.         time.sleep(3)
21.         # 判断 future2 代表的任务是否结束
22.         print(future2.done())
23.         # 查看 future1 代表的任务返回的结果
24.         print(future1.result())
25.         # 查看 future2 代表的任务返回的结果
26.         print(future2.result())
27.         # 关闭线程池
28.         pool.shutdown()
```

上面程序中，第 13 行代码创建了一个包含两个线程的线程池，接下来的两行代码只要将 action() 函数提交(submit)给线程池，该线程池就会负责启动线程来执行 action() 函数。这种启动线程的方法既优雅，又具有更高的效率。

当程序把 action() 函数提交给线程池时，submit() 方法会返回该任务所对应的 Future 对象，程序立即判断 futurel 的 done() 方法，该方法将会返回 False (表明此时该任务还未完成)。接下来主程序暂停 3 秒，然后判断 future2 的 done() 方法，如果此时该任务已经完成，那么该方法将会返回 True。

程序最后通过 Future 的 result() 方法来获取两个异步任务返回的结果。

> 读者可以自己运行此代码查看运行结果，这里不再演示。

当程序使用 Future 的 result() 方法来获取结果时，该方法会阻塞当前线程，如果没有指定 timeout 参数，当前线程将一直处于阻塞状态，直到 Future 代表的任务返回。

获取执行结果

前面程序调用了 Future 的 result() 方法来获取线程任务的运回值，但该方法会阻塞当前主线程，只有等到钱程任务完成后，result() 方法的阻塞才会被解除。

如果程序不希望直接调用 result() 方法阻塞线程，则可通过 Future 的 add_done_callback() 方法来添加回调函数，该回调函数形如 fn(future)。当线程任务完成后，程序会自动触发该回调函数，并将对应的 Future 对象作为参数传给该回调函数。

下面程序使用 add_done_callback() 方法来获取线程任务的返回值：

```
from concurrent.futures import ThreadPoolExecutor
import threading
import time

# 定义一个准备作为线程任务的函数
def action(max):
    my_sum = 0
    for i in range(max):
        print(threading.current_thread().name + '  ' + str(i))
        my_sum += i
    return my_sum
# 创建一个包含 2 条线程的线程池
with ThreadPoolExecutor(max_workers=2) as pool:
    # 向线程池提交一个 task, 50 会作为 action()函数的参数
    future1 = pool.submit(action, 50)
    # 向线程池再提交一个 task, 100 会作为 action()函数的参数
    future2 = pool.submit(action, 100)
    def get_result(future):
```

```
            print(future.result())
        # 为future1添加线程完成的回调函数
        future1.add_done_callback(get_result)
        # 为future2添加线程完成的回调函数
        future2.add_done_callback(get_result)
        print('--------------')
```

上面主程序分别为 future1、future2 添加了同一个回调函数，该回调函数会在线程任务结束时获取其返回值。

主程序的最后一行代码打印了一条横线。由于程序并未直接调用 future1、future2 的 result() 方法，因此主线程不会被阻塞，可以立即看到输出主线程打印出的横线。接下来将会看到两个新线程并发执行，当线程任务执行完成后，get_result() 函数被触发，输出线程任务的返回值。

另外，由于线程池实现了上下文管理协议（Context Manage Protocol），因此，程序可以使用 with 语句来管理线程池，这样即可避免手动关闭线程池，如上面的程序所示。

此外，Exectuor 还提供了一个 `map(func, *iterables, timeout=None, chunksize=1)` 方法，该方法的功能类似于全局函数 map()，区别在于线程池的 map() 方法会为 iterables 的每个元素启动一个线程，以并发方式来执行 func 函数。这种方式相当于启动 len(iterables) 个线程，并收集每个线程的执行结果。

例如，如下程序使用 Executor 的 map() 方法来启动线程，并收集线程任务的返回值：

```
from concurrent.futures import ThreadPoolExecutor
```

```
import threading
import time

# 定义一个准备作为线程任务的函数
def action(max):
    my_sum = 0
    for i in range(max):
        print(threading.current_thread().name + '  ' + str(i))
        my_sum += i
    return my_sum
# 创建一个包含 4 条线程的线程池
with ThreadPoolExecutor(max_workers=4) as pool:
    # 使用线程执行 map 计算
    # 后面元组有 3 个元素，因此程序启动 3 条线程来执行 action 函数
    results = pool.map(action, (50, 100, 150))
    print('--------------')
    for r in results:
        print(r)
```

上面程序使用 map() 方法来启动 3 个线程(该程序的线程池包含 4 个线程,如果继续使用只包含两个线程的线程池，此时将有一个任务处于等待状态，必须等其中一个任务完成，线程空闲出来才会获得执行的机会),map() 方法的返回值将会收集每个线程任务的返回结果。

运行上面程序,同样可以看到 3 个线程并发执行的结果,最后通过 results 可以看到 3 个线程任务的返回结果。

通过上面程序可以看出，使用 map() 方法来启动线程，并收集线程的执行结果，不仅具有代码简单的优点,而且虽然程序会以并发方式来执行 action() 函数,但最后收集的 action() 函数的执行结果，依然与传入参数的结果保持一致。也就是说，上面 results 的第一个元

素是 action(50) 的结果，第二个元素是 action(100) 的结果，第三个元素是 action(150) 的结果。

13.Python threading Local()函数用法：返回线程局部变量

Python 在 threading 模块下提供了一个 local() 函数，该函数可以返回一个线程局部变量，通过使用线程局部变量可以很简捷地隔离多线程访问的竞争资源，从而简化多线程并发访问的编程处理。

线程局部变量（Thread Local Variable）的功用其实非常简单，就是为每一个使用该变量的线程都提供一个变量的副本，使每一个线程都可以独立地改变自己的副本，而不会和其他线程的副本冲突。从线程的角度看，就好像每一个线程都完全拥有该变量一样。

下面程序示范了线程局部变量的作用：

```
1.          import threading
2.          from concurrent.futures import ThreadPoolExecutor
3.
4.          # 定义线程局部变量
5.          mydata = threading.local()
6.          # 定义准备作为线程执行体使用的函数
7.          def action (max):
8.              for i in range(max):
9.                  try:
10.                     mydata.x += i
11.                 except:
12.                     mydata.x = i
13.                 # 访问 mydata 的 x 的值
14.                 print('%s mydata.x 的值为: %d' %
```

```
15.                     (threading.current_thread().name, mydata.x))
16.         # 使用线程池启动两个子线程
17.         with ThreadPoolExecutor(max_workers=2) as pool:
18.             pool.submit(action , 10)
19.             pool.submit(action , 10)
```

上面程序中，第 5 行代码定义了一个 threading.local 变量，程序将会为每个线程各创建一个该变量的副本。

程序中作为线程执行体的 action 函数使用 mydata.x 记录 0~10 的累加值，如果两个线程共享同一个 mydata 变量，将会看到 mydata.x 最后会累加到 90（0~9 的累加值是 45，但两次累加会得到 90）。但由于 mydata 是 threading.local 变量，因此程序会为每个线程都创建一个该变量的副本，所以将会看到两个线程的 mydata.x 最后都累加到 45。

运行上面程序，将看到如图 1 所示的输出信息。

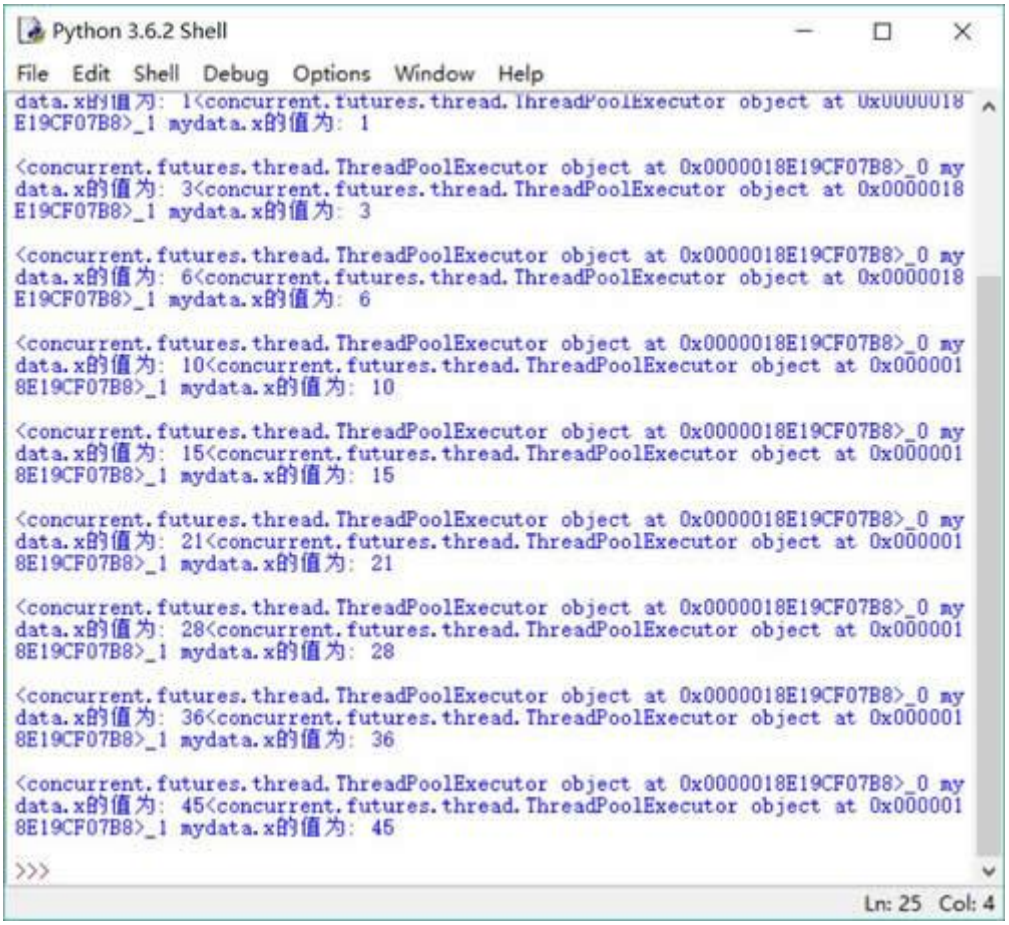


图 1 使用线程局部变量

线程局部变量和其他同步机制一样，都是为了解决多线程中对共享资源的访问冲突的。在普通的同步机制中，是通过为对象加锁来实现多个线程对共享资源的安全访问的。由于共享资源是多个线程共享的，所以要使用这种同步机制，就需要很细致地分析什么时候对共享资源进行读写，什么时候需要锁定该资源，什么时候释放对该资源的锁定等。在这种情况下，系统并没有将这份资源复制多份，只是采用安全机制来控制对这份资源的的访问而已。

线程局部变量从另一个角度来解决多线程的并发访问问题。线程局部变量将需要并发访问的资源复制多份，每个线程都拥有自己的资源副本，从而也就没有必要对该资源进行同步了。

线程局部变量提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的整个变量放到线程局部变量中，或者把该对象中与线程相关的状态放入线程局部变量中保存。

线程局部变量并不能替代同步机制，两者面向的问题领域不同。同步机制是为了同步多个线程对共享资源的并发访问，是多个线程之间进行通信的有效方式；而线程局部变量是为了隔离多个线程的数据共享，从根本上避免多个钱程之间对共享资源（变量）的竞争，也就不需要对多个线程进行同步了。

通常建议，如果多个线程之间需要共享资源，以实现线程通信，则使用同步机制；如果仅仅需要隔离多个线程之间的共享冲突，则可以使用线程局部变量。

14.Python Timer 定时器：控制函数在特定时间执行

Thread 类有一个 Timer 子类，该子类可用于控制指定函数在特定时间内执行一次。例如如下程序：

```
1.          from threading import Timer
2.
3.          def hello():
4.              print("hello, world")
5.          # 指定 10 秒后执行 hello 函数
6.          t = Timer(10.0, hello)
7.          t.start()
```

上面程序使用 Timer 控制 10s 后执行 hello 函数。

需要说明的是，Timer 只能控制函数在指定时间内执行一次，如果要使用 Timer 控制函数

多次重复执行，则需要再执行下一次调度。

如果程序想取消 Timer 的调度，则可调用 Timer 对象的 cancel() 函数。例如，如下程序每 1s 输出一次当前时间：

纯文本复制

```
from threading import Timer
import time

# 定义总共输出几次的计数器
count = 0
def print_time():
    print("当前时间：%s" % time.ctime())
    global t, count
    count += 1
    # 如果 count 小于 10，开始下一次调度
    if count < 10:
        t = Timer(1, print_time)
        t.start()
# 指定 1 秒后执行 print_time 函数
t = Timer(1, print_time)
t.start()
```

上面程序开始运行后，程序控制 1s 后执行 print_time() 函数。print_time() 函数中的代码会进行判断，如果 count 小于 10，程序再次使用 Timer 调度 1s 后执行 print_time() 函数，这样就可以控制 print_time() 函数多次重复执行。

在上面程序中，由于只有当 count 小于 10 时才会使用 Timer 调度 1s 后执行 print_time() 函数，因此该函数只会重复执行 10 次。

15.Python schedule 任务调度及其用法

如果需要执行更复杂的任务调度，则可使用 Python 提供的 sched 模块。该模块提供了 sched.scheduler 类，该类代表一个任务调度器。

sched.scheduler(timefunc=time.monotonic, delayfunc=time.sleep) 构造器支持两个参数：

1. timefunc：该参数指定生成时间戳的时间函数，默认使用 time.monotonic 来生成时间戳。
2. delayfunc：该参数指定阻塞程序的函数，默认使用 time.sleep 函数来阻塞程序。

sched.scheduler 调度器支持如下常用属性和方法：

- scheduler.enterabs(time, priority, action, argument=(), kwargs={})：指定在 time 时间点执行 action 函数，argument 和 kwargs 都用于向 action 函数传入参数，其中 argument 使用位置参数的形式传入参数，kwargs 使用关键字参数的形式传入参数。该方法返回一个 event，它可作为 cancel() 方法的参数用于取消该调度。priority 参数指定该任务的优先级，当在同一个时间点有多个任务需要执行时，优先级高（值越小代表优先级越高）的任务会优先执行。
- scheduler.enter(delay, priority, action, argument=(),kwargs={})：该方法与上一个方法基本相同，只是 delay 参数用于指定多少秒之后执行 action 任务。

- scheduler.cancel(event)：取消任务。如果传入的 event 参数不是当前调度队列中的 event，程序将会引发 ValueError 异常。
- scheduler.empty()：判断当前该调度器的调度队列是否为空。
- scheduler.run(blocking=True)：运行所有需要调度的任务。如果调用该方法的 blocking 参数为 True，该方法将会阻塞线程，直到所有被调度的任务都执行完成。
- scheduler.queue：该只读属性返回该调度器的调度队列。

下面程序示范了使用 sched.scheduler 来执行任务调度：

```
import sched, time
import threading

# 定义线程调度器
s = sched.scheduler()

# 定义被调度的函数
def print_time(name='default'):
    print("%s 的时间: %s" % (name, time.ctime()))
print('主线程: ', time.ctime())
# 指定 10 秒之后执行 print_time 函数
s.enter(10, 1, print_time)
# 指定 5 秒之后执行 print_time 函数，优先级为 2
s.enter(5, 2, print_time, argument=('位置参数',))
# 指定 5 秒之后执行 print_time 函数，优先级为 1
s.enter(5, 1, print_time, kwargs={'name': '关键字参数'})
# 执行调度的任务
s.run()
print('主线程: ', time.ctime())
```

上面程序中，第 12 行代码指定 10s 后执行 print_time() 函数，本次调度没有为该函数传入参数；第 14 行代码指定 5s 后调度 print_time() 函数，本次调度使用位置参数的形式为该函数传入参数；第 16 行代码指定 5s 后调度 print_time() 函数，本次调度使用关键字参数的形式为该函数传入参数。

上面程序运行后，将会看到程序在 5s 后执行两次 print_time() 函数，其中传入关键字参数的函数先执行（它的优先级更高），10s 后执行一次 print_time() 函数。运行上面程序，将看到如下输出结果：

```
主线程： Tue Mar  5 16:30:51 2019
关键字参数 的时间: Tue Mar  5 16:30:56 2019
位置参数 的时间: Tue Mar  5 16:30:56 2019
default 的时间: Tue Mar  5 16:31:01 2019
主线程： Tue Mar  5 16:31:01 2019
```

16.Python os.fork()方法：创建新进程

除可以进行多线程编程之外，Python 还支持使用多进程来实现并发编程。

Python 的 os 模块提供了一个 fork() 方法，该方法可以 fork 出来一个子进程。简单来说，fork() 方法的作用在于，程序会启动两个进程（一个是父进程，一个是 fork 出来的子进程）来执行从 os.fork() 开始的所有代码。

fork() 方法不需要参数，它有一个返回值，该返回值表明是哪个进程在执行：

- 如果 fork() 方法返回 0，则表明是 fork 出来的子进程在执行。
- 如果 fork() 方法返回非 0，则表明是父进程在执行，该方法返回 fork() 出来的子进程的进程 ID。

下面程序示范了使用 fork() 方法创建新进程的过程：

```
1.          import os
2.
3.          print('父进程（%s）开始执行' % os.getpid())
4.          # 开始 fork 一个子进程
5.          # 从这行代码开始，下面代码都会被两个进程执行
6.          pid = os.fork()
7.          print('进程进入：%s' % os.getpid())
8.          # 如果 pid 为 0，表明子进程
9.          if pid == 0:
10.             print('子进程，其 ID 为 (%s)， 父进程 ID 为 (%s)' % (os.getpid(),
    os.getppid()))
11.         else:
12.             print('我 (%s) 创建的子进程 ID 为 (%s).' % (os.getpid(), pid))
13.         print('进程结束：%s' % os.getpid())
```

上面程序 fork 出来一个子进程，这意味着程序会分别使用父进程和子进程来执行从此行代码开始的代码。

在 Linux 或 Mac OS X 系统上运行上面程序（Windows 不支持 fork() 方法，因此在

Windows 系统上运行上面程序会报锚），可以看到如下运行结果：

```
父进程(1795)开始执行
进程进入：1795
我(1795)创建的子进程 ID 为(1796).
进程结束：1795
进程进入：1796
子进程，其 ID 为(1796)，父进程 ID 为(1795)
进程结束： 1796
```

从上面的运行结果可以看到，此时程序分别使用两个进程执行从“进程进入”到“进程结束”之间的代码，这就是 os.fork() 方法的作用。

在实际编程中，程序可通过 fork() 方法来创建一个子进程，然后通过判断 fork() 方法的返回值来确定程序是否正在执行子进程，也就是把需要并发执行的任务放在 if pid==0: 的条件执行体中，这样就可以启动多个子进程来执行并发任务。

需要注意的是，os.fork() 方法在 Windows 系统上无效，只在 UNIX 及类 UNIX 系统上有效，UNIX 及类 UNIX 系统包括 UNIX、Linux 和 Mac OS X。

17.PythonProcess 创建进程（2 种方法）详解

虽然使用 os.fork() 方法可以启动多个进程，但这种方式显然不适合 Windows，而 Python 是跨平台的语言，所以 Python 绝不能仅仅局限于 Windows 系统，因此

Python 也提供了其他方式在 Windows 下创建新进程。

Python 在 multiprocessing 模块下提供了 Process 来创建新进程。与 Thread 类似的是，使用 Process 创建新进程也有两种方式：

1. 以指定函数作为 target，创建 Process 对象即可创建新进程。
2. 继承 Process 类，并重写它的 run() 方法来创建进程类，程序创建 Process 子类的实例作为进程。

Process 类也有如下类似的方法和属性：

- run()：重写该方法可实现进程的执行体。
- start()：该方法用于启动进程。
- join([timeout])：该方法类似于线程的 join() 方法，当前进程必须等待被 join 的进程执行完成才能向下执行。
- name：该属性用于设置或访问进程的名字。
- is_alive()：判断进程是否还活着。
- daemon：该属性用于判断或设置进程的后台状态。
- pid：返回进程的 ID。
- authkey：返回进程的授权 key。
- terminate()：中断该进程。

以指定函数作为 target 创建新进程

下面先介绍以指定函数作为 target 来创建新进程。

```
1.          import multiprocessing
2.          import os
3.
4.          # 定义一个普通的 action 函数，该函数准备作为进程执行体
5.          def action(max):
6.              for i in range(max):
7.                  print("(%s)子进程（父进程:(%s)）: %d" %
8.                      (os.getpid(), os.getppid(), i))
9.          if __name__ == '__main__':
10.             # 下面是主程序（也就是主进程）
11.             for i in range(100):
12.                 print("(%s)主进程: %d" % (os.getpid(), i))
13.                 if i == 20:
14.                     # 创建并启动第一个进程
15.                     mp1 = multiprocessing.Process(target=action,args=(100,))
16.                     mp1.start()
17.                     # 创建并启动第一个进程
18.                     mp2 = multiprocessing.Process(target=action,args=(100,))
19.                     mp2.start()
20.                     mp2.join()
21.             print('主进程执行完成!')
```

上面程序中第 15、16 两行代码就是程序创建并启动新进程的关键代码，不难发现这两行代码和创建并启动新线程的代码几乎一样，只是此处创建的是 multiprocessing.Process 对象。

需要说明的是，通过 multiprocessing.Process 来创建并启动进程时，程序必须先判断 `if __name__=='__main__':`，否则可能引发异常。

运行上面程序，可以看到程序中运行了三个进程，一个主进程和程序启动的两个子进程。由于程序中调用了 mp2.join()，因此主进程必须等 mp2 进程完成后才能向下执行。

继承 Process 类创建子进程

继承 Process 类创建子进程的步骤如下：

1. 定义继承 Process 的子类，重写其 run() 方法准备作为进程执行体。
2. 创建 Process 子类的实例。
3. 调用 Process 子类的实例的 start() 方法来启动进程。

下面程序通过继承 Process 类来创建子进程：

纯文本复制

```
import multiprocessing
import os

class MyProcess(multiprocessing.Process):
    def __init__(self, max):
        self.max = max
        super().__init__()
    # 重写 run()方法作为进程执行体
    def run(self):
        for i in range(self.max):
            print("(%s)子进程（父进程:(%s)）: %d" %
                (os.getpid(), os.getppid(), i))
if __name__ == '__main__':
    # 下面是主程序（也就是主进程）
    for i in range(100):
        print("(%s)主进程: %d" % (os.getpid(), i))
```

```
            if i == 20:
                # 创建并启动第一个进程
                mp1 = MyProcess(100)
                mp1.start()
                # 创建并启动第一个进程
                mp2 = MyProcess(100)
                mp2.start()
                mp2.join()
        print('主进程执行完成!')
```

该程序的运行结果与上一个程序的运行结果大致相同，它们只是创建进程的方式略有不同而已。

通常，推荐使用第一种方式来创建进程，因为这种方式不仅编程简单，而且进程直接包装 target 函数，具有更清晰的逻辑结构。

18.Python 设置进程启动的 3 种方式

根据平台的支持，Python 支持 3 种启动进程的方式：

1) spawn

父进程会启动一个全新的 Python 解释器进程。在这种方式下，子进程只能继承那些处理 run() 方法所必需的资源。典型的，那些不必要的文件描述器和 handle 都不会被继承。使用这种方式来启动进程，其效率比使用 fork 或 forkserver 方式要低得多。

> Windows 只支持 spawn 方式来启动进程，因此在 Windows 平台上默认使用这种方式来启动进程。

2) fork

父进程使用 os.fork() 来启动一个 Python 解释器进程。在这种方式下，子进程会继承父

进程的所有资源，因此子进程基本等效于父进程。这种方式只在 UNIX 平台上有效，UNIX 平台默认使用这种方式来启动进程。

3) forkserver

如果使用这种方式来启动进程，程序将会启动一个服务器进程。在以后的时间内，当程序再次请求启动新进程时，父进程都会连接到该服务器进程，请求由服务器进程来 fork 新进程。通过这种方式启动的进程不需要从父进程继承资源。这种方式只在 UNIX 平台上有效。

从上面介绍可以看出，如果程序使用 UNIX 平台（包括 Linux 和 Mac OS X），Python 支持三种启动进程的方式；但如果使用 Windows 平台，则只能使用效率最低的 spawn 方式。

multiprocessing 模块提供了一个 `set_start_method()` 函数，该函数可用于设置启动进程的方式（必须将这行设置代码放在所有与多进程有关的代码之前）。下面程序示范了显式设置启动进程的方式：

```
1.          import multiprocessing
2.          import os
3.
4.          def foo(q):
5.              print('被启动的新进程: (%s)' % os.getpid())
6.              q.put('Python')
7.          if __name__ == '__main__':
8.              # 设置使用 fork 方式启动进程
9.              multiprocessing.set_start_method('spawn')
10.             q = multiprocessing.Queue()
11.             # 创建进程
```

```
12.             mp = multiprocessing.Process(target=foo, args=(q, ))
13.             # 启动进程
14.             mp.start()
15.             # 获取队列中的消息
16.             print(q.get())
17.             mp.join()
```

上面程序显式指定必须使用 fork 方式来启动进程，因此该程序只能在 UNIX 平台（包括 Linux、Mac OS X）上运行。上面代码实际上就相当于前面介绍的使用 os.fork() 方法来启动新进程。

在 Mac OS X 上运行上面程序，可以看到如下输出结果：

```
被启动的新进程：(1868)
Python
```

上面程序的新进程向 multiprocessing.Queue 中放入一个数据（Python 字符串），主进程取出该 Queue 中的数据，并输出该数据。

还有一种设置进程启动方式的方法，就是利用 get_context() 方法来获取 Context 对象，调用该方法时可传入 spawn、fork 或 forkserver 字符串。Context 拥有和 multiprocessing 相同的 API，因此程序可通过 Context 来创建并启动进程。例如如下程序：

```
1.          import multiprocessing
2.          import os
3.
```

```
4.          def foo(q):
5.              print('被启动的新进程: (%s)' % os.getpid())
6.              q.put('Python')
7.          if __name__ == '__main__':
8.              # 设置使用 fork 方式启动进程，并获取 Context 对象
9.              ctx = multiprocessing.get_context('fork')
10.             # 接下来就可用 Context 对象来代替 mutliprocessing 模块了
11.             q = ctx.Queue()
12.             # 创建进程
13.             mp = ctx.Process(target=foo, args=(q, ))
14.             # 启动进程
15.             mp.start()
16.             # 获取队列中的消息
17.             print(q.get())
18.             mp.join()
```

上面程序中，第 9 行代码设置以 fork 方式启动进程，并获取 Context 对象，这样程序后面就可以使用 Context 对象来代替 multiprocessing 模块了。

19.多进程编程和多线程编程的优缺点

多进程编程和多钱程编程，都可以使用并行机制来提升系统的运行效率。二者的区别在于运行时所占的内存分布不同，多钱程是共用一套内存的代码块区间；而多进程是各用一套独立的内存区间。

多进程的优点是稳定性好，一个子进程崩溃了，不会影响主进程以及其余进程。基于这个特性，常常会用多进程来实现守护服务器的功能。

多进程编程也有不足，即创建进程的代价非常大，因为操作系统要给每个进程分配固定的资源，并且操作系统对进程的总数会有一定的限制，若进程过多，操作系统调度都会存在问题，

会造成假死状态。

多线程编程的优点是效率较高一些，适用于批处理任务等功能；不足之处在于，任何一个线程崩溃都可能造成整个进程的崩溃，因为它们共享了进程的内存资源池。

既然多线程编程和多进程编程各有优缺点，因此它们分别适用于不同的场景。比如说，对于计算密集型的任务，多进程效率会更高一下；而对于 IO 密集型的任务（比如文件操作，网络爬虫），采用多线程编程效率更高。为什么是这样呢？

其实也不难理解。对于 IO 密集型操作，大部分消耗时间其实是等待时间，在等待时间中，Python 会释放 GIL 供新的线程使用，实现了线程间的切换；相反对于 CPU 密集型代码，2 个 CPU 干活肯定比一个 CPU 快很多。

在大型的计算机集群系统中，通常都会将多进程程序分布运行在不同的计算机上协同工作。而每一台计算机上的进程内部，又会由多个线程来并行工作。

注意，对于任务数来说，无论是多进程编程或者多线程编程，其进程数或线程数都不能太多：

- 对于多进程编程来说，操作系统在切换任务时，会有一系列的保护现场措施，这要花费相当多的系统资源，若任务过多，则大部分资源都被用做干这些了，结果就是所有任务都做不好；
- 多线程编程也不是线程个数越多效率越高，通过下面的公式可以计算出线程数量最优的一个参考值。

$$最佳线程数量 = \frac{线程等待时间 + 线程CPU时间}{线程CPU时间} * CPU数量$$

20.Python 使用进程池管理进程

与线程池类似的是，如果程序需要启动多个进程，也可以使用进程池来管理进程。程序可以通过 multiprocessing 模块的 Pool() 函数创建进程池，进程池实际上是 multiprocessing.pool.Pool 类。

进程池具有如下常用方法：

- apply(func[, args[, kwds]])：将 func 函数提交给进程池处理。其中 args 代表传给 func 的位置参数，kwds 代表传给 func 的关键字参数。该方法会被阻塞直到 func 函数执行完成。
- apply_async(func[, args[, kwds[, callback[, error_callback]]]])：这是 apply() 方法的异步版本，该方法不会被阻塞。其中 callback 指定 func 函数完成后的回调函数，error_callback 指定 func 函数出错后的回调函数。
- map(func, iterable[, chunksize])：类似于 Python 的 map() 全局函数，只不过此处使用新进程对 iterable 的每一个元素执行 func 函数。
- map_async(func, iterable[, chunksize[, callback[, error_callback]]])：这是 map() 方法的异步版本，该方法不会被阻塞。其中 callback 指定 func 函数完成后的回调函数，error_callback 指定 func 函数出错后的回调函数。
- imap(func, iterable[, chunksize])：这是 map() 方法的延迟版本。

- imap_unordered(func, iterable[, chunksize])：功能类似于 imap() 方法，但该方法不能保证所生成的结果（包含多个元素）与原 iterable 中的元素顺序一致。
- starmap(func, iterable[,chunksize])：功能类似于 map() 方法，但该方法要求 iterable 的元素也是 iterable 对象，程序会将每一个元素解包之后作为 func 函数的参数。
- close()：关闭进程池。在调用该方法之后，该进程池不能再接收新任务，它会把当前进程池中的所有任务执行完成后再关闭自己。
- terminate()：立即中止进程池。
- join()：等待所有进程完成。

从上面介绍不难看出，如果程序只是想将任务提交给进程池执行，则可调用 apply() 或 apply_async() 方法；如果程序需要使用指定函数将 iterable 转换成其他 iterable，则可使用 map() 或 imap() 方法。

下面程序示范了使用 apply_async() 方法启动进程：

```
1.          import multiprocessing
2.          import time
3.          import os
4.
5.          def action(name='default'):
6.              print('(%s)进程正在执行，参数为: %s' % (os.getpid(), name))
7.              time.sleep(3)
8.          if __name__ == '__main__':
9.              # 创建包含 4 条进程的进程池
10.             pool = multiprocessing.Pool(processes=4)
```

```
11.             # 将 action 分 3 次提交给进程池
12.             pool.apply_async(action)
13.             pool.apply_async(action, args=('位置参数', ))
14.             pool.apply_async(action, kwds={'name': '关键字参数'})
15.             pool.close()
16.             pool.join()
```

上面程序创建了一个进程池，其中第 12-14 行代码都负责将 action 提交给进程池，只是每次提交时指定参数的方式不同。

运行上面程序，可以看到如下输出结果：

```
(14304)进程正在执行，参数为：default
(9344)进程正在执行，参数为：位置参数
(13796)进程正在执行，参数为：关键字参数
```

从上面的输出结果可以看到，程序分别使用 3 个进程来执行 action 任务。

从上面程序可以看出，进程池同样实现了上下文管理协议，因此程序可以使用 with 子句来管理进程池，这样就可以避免程序主动关闭进程池。

下面程序示范了使用 map() 方法来启动进程：

```
1.          import multiprocessing
2.          import time
3.          import os
4.
5.          # 定义一个准备作为进程任务的函数
6.          def action(max):
```

```
            my_sum = 0
            for i in range(max):
                print('(%s)进程正在执行: %d' % (os.getpid(), i))
                my_sum += i
            return my_sum
    if __name__ == '__main__':
        # 创建一个包含 4 条进程的进程池
        with multiprocessing.Pool(processes=4) as pool:
            # 使用进程执行 map 计算
            # 后面元组有 3 个元素，因此程序启动 3 条进程来执行 action 函数
            results = pool.map(action, (50, 100, 150))
            print('--------------')
            for r in results:
                print(r)
```

运行上面程序，可以看到程序启动 3 个进程来执行 action 函数，程序最后输出 0~50、0~100、0~150 的累加结果。

可能读者已经发现了，其实该程序与前面介绍线程池的 map() 方法时所用的示例程序几乎一样。事实就是如此，只不过前面程序使用了更轻量级的线程来实现并发，而此处则使用进程来实现并发。这两种方式殊途同归，但相比之下，使用线程会有更好的性能，因此一般推荐使用多线程来实现并发。

21.Python 进程间通信 2 种实现方法（Queue 和 Pipe）

Python 为进程通信提供了两种机制：

1. Queue：一个进程向 Queue 中放入数据，另一个进程从 Queue 中读取数据。

2. Pipe：Pipe 代表连接两个进程的管道。程序在调用 Pipe() 函数时会产生两个连接端，分别交给通信的两个进程，接下来进程既可从该连接端读取数据，也可向该连接端写入数据。

使用 Queue 实现进程间通信

multiprocessing 模块下的 Queue 和 queue 模块下的 Queue 基本类似，它们都提供了 qsize()、empty()、full()、put()、put_nowait()、get()、get_nowait() 等方法。区别只是 multiprocessing 模块下的 Queue 为进程提供服务，而 queue 模块下的 Queue 为线程提供服务。

下面程序使用 Queue 来实现进程之间的通信：

```
1.          import multiprocessing
2.
3.          def f(q):
4.              print('(%s) 进程开始放入数据...' %
multiprocessing.current_process().pid)
5.              q.put('Python')
6.          if __name__ == '__main__':
7.              # 创建进程通信的 Queue
8.              q = multiprocessing.Queue()
9.              # 创建子进程
10.             p = multiprocessing.Process(target=f, args=(q,))
11.             # 启动子进程
12.             p.start()
13.             print('(%s) 进程开始取出数据...' %
multiprocessing.current_process().pid)
14.             # 取出数据
15.             print(q.get())  # Python
```

```
16.                 p.join()
```

上面程序中，第 5 行代码（子进程）负责向 Queue 中放入一个数据，第 15 行代码（父进程）负责从 Queue 中读取一个数据，这样就实现了父、子两个进程之间的通信。

运行上面程序，可以看到如下输出结果：

```
(14180）进程开始取出数据...
(14700）进程开始放入数据...
Python
```

使用 Pipe 实现进程间通信

使用 Pipe 实现进程通信，程序会调用 multiprocessing.Pipe() 函数来创建一个管道，该函数会返回两个 PipeConnection 对象，代表管道的两个连接端（一个管道有两个连接端，分别用于连接通信的两个进程）。

PipeConnection 对象包含如下常用方法：

- send(obj)：发送一个 obj 给管道的另一端，另一端使用 recv() 方法接收。需要说明的是，该 obj 必须是可 picklable 的（Python 的序列化机制），如果该对象序列化之后超过 32MB，则很可能会引发 ValueError 异常。
- recv()：接收另一端通过 send() 方法发送过来的数据。
- fileno()：关于连接所使用的文件描述器。
- close()：关闭连接。

- poll([timeout])：返回连接中是否还有数据可以读取。
- send_bytes(buffer[, offset[, size]]) :发送字节数据。如果没有指定 offset、size 参数，则默认发送 buffer 字节串的全部数据；如果指定了 offset 和 size 参数，则只发送 buffer 字节串中从 offset 开始、长度为 size 的字节数据。通过该方法发送的数据，应该使用 recv_bytes() 或 recv_bytes_into 方法接收。
- recv_bytes([maxlength])：接收通过 send_bytes() 方法发送的数据，maxlength 指定最多接收的字节数。该方法返回接收到的字节数据。
- recv_bytes_into(buffer[, offset])：功能与 recv_bytes() 方法类似，只是该方法将接收到的数据放在 buffer 中。

下面程序将会示范如何使用 Pipe 来实现两个进程之间的通信：

```
1.          import multiprocessing
2.
3.          def f(conn):
4.              print('(%s) 进程开始发送数据...' %
multiprocessing.current_process().pid)
5.              # 使用 conn 发送数据
6.              conn.send('Python')
7.          if __name__ == '__main__':
8.              # 创建 Pipe，该函数返回两个 PipeConnection 对象
9.              parent_conn, child_conn = multiprocessing.Pipe()
10.             # 创建子进程
11.             p = multiprocessing.Process(target=f, args=(child_conn, ))
12.             # 启动子进程
13.             p.start()
14.             print('(%s) 进程开始接收数据...' %
multiprocessing.current_process().pid)
15.             # 通过 conn 读取数据
```

```
16.             print(parent_conn.recv())  # Python
17.             p.join()
```

上面程序中第 6 行代码（子进程）通过 PipeConnection 向管道发送数据，数据将会被发送给管道另一端的父进程。第 16 行代码（父进程）通过 PipeConnection 从管道读取数据，程序就可以读取到另一端子进程写入的数据，这样就实现了父、子两个进程之间的通信。

运行上面程序，可以看到如下输出结果：

```
(15560)进程开始接收数据...
(15580)进程开始发送数据...
Python
```

22.Python Futures 并发编程

无论哪门编程语言，并发编程都是一项很常用很重要的技巧。例如，爬虫就被广泛应用在工业界的各个领域，我们每天在各个网站、各个 App 上获取的新闻信息，很大一部分便是通过并发编程版的爬虫获得。

正确合理地使用并发编程，无疑会给程序带来极大的性能提升。因此，本节就带领大家一起学习 Python 中的 Futures 并发编程。首先，先带领大家从代码的角度来理解并发编程中的 Futures，并进一步来比较其与单线程的性能区别。

假设有这样一个任务，要下载一些网站的内容并打印，如果用单线程的方式，它的代码实现如下所示（为了突出主题，对代码做了简化，忽略了异常处理）：

```
1.          import requests
2.          import time
3.
4.          def download_one(url):
5.              resp = requests.get(url)
6.              print('Read {} from {}'.format(len(resp.content), url))
7.
8.          def download_all(sites):
9.              for site in sites:
10.                 download_one(site)
11.
12.         def main():
13.             sites = [
14.                 'http://c.biancheng.net',
15.                 'http://c.biancheng.net/c',
16.                 'http://c.biancheng.net/python'
17.             ]
18.             start_time = time.perf_counter()
19.             download_all(sites)
20.             end_time = time.perf_counter()
21.             print('Download {} sites in {} seconds'.format(len(sites), end_time
    - start_time))
22.
23.         if __name__ == '__main__':
24.             main()
```

输出结果为：

```
Read 52053 from http://c.biancheng.net

Read 30718 from http://c.biancheng.net/c
```

```
Read 34470 from http://c.biancheng.net/python

Download 3 sites in 0.3537296 seconds
```

注意，此程序中，requests 模块需单独安装，可通过执行 pip install requests 命令进行安装。

这种方式应该是最直接也最简单的：

- 先是遍历存储网站的列表；
- 然后对当前网站执行下载操作；
- 等到当前操作完成后，再对下一个网站进行同样的操作，一直到结束。

可以看到，总共耗时约 0.35s。单线程的优点是简单明了，但是明显效率低下，因为上述程序的绝大多数时间都浪费在了 I/O 等待上。程序每次对一个网站执行下载操作，都必须等到前一个网站下载完成后才能开始。如果放在实际生产环境中，我们需要下载的网站数量至少是以万为单位的，不难想象，这种方案根本行不通。

接着再来看多线程版本的代码实现：

```
1.          import concurrent.futures
2.          import requests
3.          import threading
4.          import time
5.
6.          def download_one(url):
7.              resp = requests.get(url)
8.              print('Read {} from {}'.format(len(resp.content), url))
9.
```

```
10.         def download_all(sites):
11.             with concurrent.futures.ThreadPoolExecutor(max_workers=5) as
    executor:
12.                 executor.map(download_one, sites)
13.
14.         def main():
15.             sites = [
16.                 'http://c.biancheng.net',
17.                 'http://c.biancheng.net/c',
18.                 'http://c.biancheng.net/python'
19.             ]
20.             start_time = time.perf_counter()
21.             download_all(sites)
22.             end_time = time.perf_counter()
23.             print('Download {} sites in {} seconds'.format(len(sites), end_time
    - start_time))
24.
25.         if __name__ == '__main__':
26.             main()
```

运行结果为：

```
Read 52053 from http://c.biancheng.net

Read 30718 from http://c.biancheng.net/c

Read 34470 from http://c.biancheng.net/python



Download 3 sites in 0.1606366 seconds
```

可以看到，总耗时是 0.2s 左右，效率一下子提升了很多。

注意，虽然线程的数量可以自己定义，但是线程数并不是越多越好，因为线程的创建、维护和删除也会有一定的开销，所以如果设置的很大，反而可能会导致速度变慢。我们往往需要根据实际的需求做一些测试，来寻找最优的线程数量。

上面两段代码中，多线程版本和单线程版的主要区别在于如下代码：

```
1.          with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
2.              executor.map(download_one, sites)
```

这里创建了一个线程池，总共有 5 个线程可以分配使用。executer.map() 与前面所讲的 Python 内置的 map() 函数类似，表示对 sites 中的每一个元素并发地调用函数 download_one()。

在 download_one() 函数中使用的 requests.get() 方法是线程安全的，在多线程的环境下也可以安全使用，不会出现条件竞争（多个线程同时竞争使用同一资源）的情况。

当然，也可以用并行的方式去提高程序运行效率，只需要在 download_all() 函数中做出下面的变化即可：

```
1.          with futures.ThreadPoolExecutor(workers) as executor
2.          #=>
3.          with futures.ProcessPoolExecutor() as executor:
```

这部分代码中，函数 ProcessPoolExecutor() 表示创建进程池，使用多个进程并行的执行程序。不过，这里通常省略参数 workers，因为系统会自动返回 CPU 的数量作为可以调用的进程数。

但是，并行的方式一般用在 CPU heavy 的场景中，因为对于 I/O heavy 的操作，多数时间都会用于等待，相比于多线程，使用多进程并不会提升效率。反而很多时候，因为 CPU 数量的限制，会导致其执行效率不如多线程版本。

什么是 Futures？

Python Futures 模块，位于 concurrent.futures 和 asyncio 中，它们都表示带有延迟的操作。Futures 会将处于等待状态的操作包裹起来放到队列中，这些操作的状态随时可以查询，当然它们的结果（或是异常）也能够在操作完成后被获取。

通常来说，用户不用考虑如何去创建 Futures，这些 Futures 底层都会帮我们处理好，唯一要做的只是去设定这些 Futures 的执行。比如，Futures 中的 Executor 类，当执行 executor.submit(func) 时，它便会安排里面的 func() 函数执行，并返回创建好的 future 实例，以便之后查询调用。

这里再介绍一些常用的函数。比如 Futures 中的方法 done()，表示相对应的操作是否完成，返回 True 表示完成；返回 False 表示没有完成。不过要注意的是，done() 是非阻塞的，会立即返回结果。相对应的 add_done_callback(fn)，则表示 Futures 完成后，相对应的参数函数 fn 会被通知并执行调用。

Futures 中还有一个重要的函数 result()，它表示当 future 完成后，返回其对应的结果或异常。而 as_completed(fs)，则是针对给定的 future 迭代器 fs，在其完成后返回完成后的迭代器。

所以，上述例子也可以写成下面的形式：

```
1.          import concurrent.futures
2.          import requests
3.          import time
4.
5.          def download_one(url):
6.              resp = requests.get(url)
7.              print('Read {} from {}'.format(len(resp.content), url))
8.
9.          def download_all(sites):
10.             with concurrent.futures.ThreadPoolExecutor(max_workers=5) as
    executor:
11.                 to_do = []
12.                 for site in sites:
13.                     future = executor.submit(download_one, site)
14.                     to_do.append(future)
15.
16.                 for future in concurrent.futures.as_completed(to_do):
17.                     future.result()
18.         def main():
19.             sites = [
20.                 'http://c.biancheng.net',
21.                 'http://c.biancheng.net/c',
22.                 'http://c.biancheng.net/python'
23.             ]
24.             start_time = time.perf_counter()
25.             download_all(sites)
26.             end_time = time.perf_counter()
27.             print('Download {} sites in {} seconds'.format(len(sites), end_time
    - start_time))
28.
29.         if __name__ == '__main__':
30.             main()
```

运行结果为：

```
Read 52053 from http://c.biancheng.net

Read 34470 from http://c.biancheng.net/python

Read 30718 from http://c.biancheng.net/c


Download 3 sites in 0.2275894 seconds
```

此程序中，首先调用 executor.submit()，将下载每一个网站的内容都放进 future 队列 to_do 等待执行。然后是 as_completed() 函数在 future 完成后便输出结果。

不过，这里要注意，future 列表中每个 future 完成的顺序和它在列表中的顺序并不一定完全一致。到底哪个先完成、哪个后完成，取决于系统的调度和每个 future 的执行时间。

23.Python Asyncio 并发编程

本节继续学习 Python 并发编程的另一种实现方式，也就是 Asyncio 并发编程。

我们知道，使用多线程和普通的单线程相比，其运行效率会有极大的提高。但不得不说，多线程虽然有诸多优势，也存在一定的局限性：

- 多线程运行过程中容易被打断，还可能出现多个线程同时竞争同一资源的情况；
- 多线程切换本身存在一定的损耗，线程数不能无线增加，因此如果 I\O 操作非常频繁，多线程很有可能满足不了高效率、高质量的需求。

为了解决这些问题，Asyncio 并发编程应运而生。

在详细介绍 Asyncio 之前，要先搞清楚什么是同步，什么是异步。所谓同步，是指操作一个接一个地执行，下一个操作必须等上一个操作执行完成之后才能开始执行；而异步是指不同操作间可以相互交替执行，如果其中地某个操作被堵塞，程序并不会等待，而是会找出可执行的操作继续执行。

为了更好地区分同步和异步，这里举个例子，假设公司要我们做一份报表，并以邮件的方式提交，则分别以同步和异步的方式完成的过程如下：

- 如果按照同步的方式，应先向软件中输入各项数据，接下来等报表生成，再写邮件提交；
- 如果按照异步的方式，向软件中输出各项数据后，会先写邮件，等待报表生成后，暂停写邮件的工作去查看生成的报表，确认无误后在写邮件直到发送完毕。

了解了同步和异步（以及它们之间的区别）之后，接下来正式开始介绍 Asyncio。

什么是 Asyncio

事实上，Asyncio 和其他 Python 程序一样，是单线程的，它只有一个主线程，但可以进行多个不同的任务。这里的任务，指的就是特殊的 future 对象，我们可以把它类比成多线程版本里的多个线程。

这些不同的任务，被一个叫做事件循环（Event Loop）的对象所控制。所谓事件循环，是指主线程每次将执行序列中的任务清空后，就去事件队列中检查是否有等待执行的任务，如果有则每次取出一个推到执行序列中执行，这个过程是循环往复的。

为了简化讲解这个问题，可以假设任务只有两个状态：，分别是预备状态和等待状态：

- 预备状态是指任务目前空闲，但随时待命准备运行；
- 等待状态是指任务已经运行，但正在等待外部的操作完成，比如 I/O 操作。

在这种情况下，事件循环会维护两个任务列表，分别对应这两种状态，并且选取预备状态的一个任务（具体选取哪个任务，和其等待的时间长短、占用的资源等等相关）使其运行，一直到这个任务把控制权交还给事件循环为止。

当任务把控制权交还给事件循环对象时，它会根据其是否完成把任务放到预备或等待状态的列表，然后遍历等待状态列表的任务，查看他们是否完成：如果完成，则将其放到预备状态的列表；反之，则继续放在等待状态的列表。而原先在预备状态列表的任务位置仍旧不变，因为它们还未运行。

这样，当所有任务被重新放置在合适的列表后，新一轮的循环又开始了，事件循环对象继续从预备状态的列表中选取一个任务使其执行...如此周而复始，直到所有任务完成。

值得一提的是，对于 Asyncio 来说，它的任务在运行时不会被外部的一些因素打断，因此

Asyncio 内的操作不会出现竞争资源（多个线程同时使用同一资源）的情况，也就不需要担心线程安全的问题了。

如何使用 Asyncio

讲完了 Asyncio 的原理，下面结合具体的代码来看一下它的用法。还是以下载网站内容为例，用 Asyncio 的实现代码（省略了异常处理的一些操作）如下：

```
import asyncio
import aiohttp
import time

async def download_one(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as resp:
            print('Read {} from {}'.format(resp.content_length, url))

async def download_all(sites):
    tasks = [asyncio.ensure_future(download_one(site)) for site in sites]
    await asyncio.gather(*tasks)

def main():
    sites = [
        'http://c.biancheng.net',
        'http://c.biancheng.net/c',
        'http://c.biancheng.net/python'
    ]
    start_time = time.perf_counter()

    loop = asyncio.get_event_loop()
    try:
        loop.run_until_complete(download_all(sites))
    finally:
```

```
26.                 loop.close()
27.
28.             end_time = time.perf_counter()
29.             print('Download {} sites in {} seconds'.format(len(sites), end_time
    - start_time))
30.
31.         if __name__ == '__main__':
32.             main()
```

运行结果为：

```
Read 52053 from http://c.biancheng.net

Read 30718 from http://c.biancheng.net/c

Read 34470 from http://c.biancheng.net/python

Download 3 sites in 0.12174049999999999 seconds
```

注意，此程序运行前，需确保已安装好 aiohttp 模块，此模块可直接执行 pip install aiohttp 命令安装。

上面程序中，Async 和 await 关键字是 Asyncio 的最新写法，表示这个语句（函数）是非阻塞的，正好对应前面所讲的事件循环的概念，即如果任务执行的过程需要等待，则将其放入等待状态的列表中，然后继续执行预备状态列表里的任务。

另外在主函数中，第 22-26 行代码表示拿到事件循环对象，并运行 download_all() 函数，直到其结束，最后关闭这个事件循环对象。

值得一提的，如果读者使用 Python 3.7 及以上版本，则 22-26 行代码可以直接用 asyncio.run(download_all(sites)) 来代替。

至于 Asyncio 版本的函数 download_all()，和之前多线程版本有很大的区别：

1. 这里的 asyncio.ensure_future(coro) 表示对输入的协程 coro 创建一个任务，安排它的执行，并返回此任务对象。可以看到，这里对每一个网站的下载，都创建了一个对应的任务。

> 注意，Python 3.7+ 版本之后，可以使用 asyncio.create_task(coro) 等效替代 asyncio.ensure_future(coro)。

2. asyncio.gather() 表示在事件循环对象中运行 aws 序列的所有任务。

可以看到，其输出结果显示用时只有 0.12s，比之前的多线程版本效率更高，充分体现其优势。

> 当然，除了例子中用到的这几个函数，Asyncio 还提供了很多其他的用法，你可以查看 Python 事件循环官方文档进行了解。

Asyncio 有缺陷吗？

通过以上的学习，明显看到了 Asyncio 的强大。但是，任何一种方案都不是完美的，都存在一定的局限性，Asyncio 同样如此。

实际工作中，想用好 Asyncio，特别是发挥其强大的功能，很多情况下必须得有相应的 Python 库支持。前面章节在学习多线程编程中使用的是 requests 库，但本节使用的是

aiohttp 库，原因在于 requests 库并不兼容 Asyncio，而 aiohttp 库兼容。Asyncio 软件库的兼容性问题，在 Python3 的早期一直是个大问题，但是随着技术的发展，这个问题正逐步得到解决。

另外，使用 Asyncio 时，因为在任务调度方面有了更大的自主权，写代码时就得更加注意，不然很容易出错。

24.深入底层了解 Python GIL 全局解释器锁

通过前面的学习，我们了解了 Pyton 并发编程的特性以及什么是多线程编程。其实除此之外，Python 多线程还有一个很重要的知识点，就是本节要讲的 GIL。

GIL，中文译为全局解释器锁。在讲解 GIL 之前，首先通过一个例子来直观感受一下 GIL 在 Python 多线程程序运行的影响。

首先运行如下程序：

```
1.          import time
2.          start = time.clock()
3.          def CountDown(n):
4.              while n > 0:
5.                  n -= 1
6.          CountDown(100000)
7.          print("Time used:",(time.clock() - start))
```

运行结果为：

```
Time used: 0.0039529000000000005
```

在我们的印象中，使用多个（适量）线程是可以加快程序运行效率的，因此可以尝试将上面程序改成如下方式：

```
1.          import time
2.          from threading import Thread
3.          start = time.clock()
4.          def CountDown(n):
5.              while n > 0:
6.                  n -= 1
7.          t1 = Thread(target=CountDown, args=[100000 // 2])
8.          t2 = Thread(target=CountDown, args=[100000 // 2])
9.          t1.start()
10.         t2.start()
11.         t1.join()
12.         t2.join()
13.         print("Time used:",(time.clock() - start))
```

运行结果为：

```
Time used: 0.006673
```

可以看到，此程序中使用了 2 个线程来执行和上面代码相同的工作，但从输出结果中可以看到，运行效率非但没有提高，反而降低了。

如果使用更多线程进行尝试，会发现其运行效率和 2 个线程效率几乎一样（本机器测试使用 4 个线程，其执行效率约为 0.005）。这里不再给出具体测试代码，有兴趣的读者可自行测试。

是不是和你猜想的结果不一样？事实上，得到这样的结果是肯定的，因为 GIL 限制了 Python 多线程的性能不会像我们预期的那样。

那么，什么是 GIL 呢？GIL 是最流程的 CPython 解释器（平常称为 Python）中的一个技术术语，中文译为全局解释器锁，其本质上类似操作系统的 Mutex。GIL 的功能是：在 CPython 解释器中执行的每一个 Python 线程，都会先锁住自己，以阻止别的线程执行。

当然，CPython 不可能容忍一个线程一直独占解释器，它会轮流执行 Python 线程。这样一来，用户看到的就是“伪”并行，即 Python 线程在交替执行，来模拟真正并行的线程。

有读者可能会问，既然 CPython 能控制线程伪并行，为什么还需要 GIL 呢？其实，这和 CPython 的底层内存管理有关。

CPython 使用引用计数来管理内容，所有 Python 脚本中创建的实例，都会配备一个引用计数，来记录有多少个指针来指向它。当实例的引用计数的值为 0 时，会自动释放其所占的内存。

举个例子，看如下代码：

```
>>> import sys
>>> a = []
>>> b = a
```

```
>>> sys.getrefcount(a)

3
```

可以看到，a 的引用计数值为 3，因为有 a、b 和作为参数传递的 getrefcount 都引用了一个空列表。

假设有两个 Python 线程同时引用 a，那么双方就都会尝试操作该数据，很有可能造成引用计数的条件竞争，导致引用计数只增加 1（实际应增加 2），这造成的后果是，当第一个线程结束时，会把引用计数减少 1，此时可能已经达到释放内存的条件（引用计数为 0），当第 2 个线程再次视图访问 a 时，就无法找到有效的内存了。

所以，CPython 引进 GIL，可以最大程度上规避类似内存管理这样复杂的竞争风险问题。

Python GIL 底层实现原理

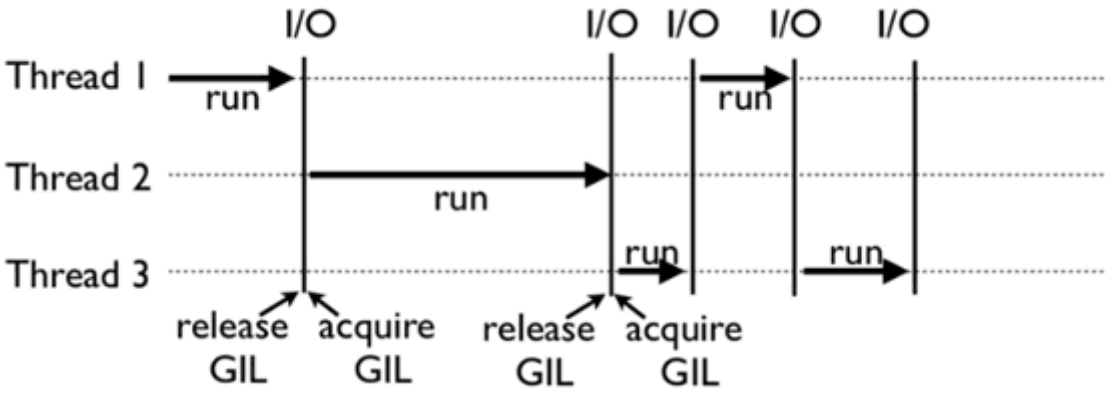


图 1 GIL 工作流程示意图

上面这张图，就是 GIL 在 Python 程序的工作示例。其中，Thread 1、2、3 轮流执行，

每一个线程在开始执行时，都会锁住 GIL，以阻止别的线程执行；同样的，每一个线程执行完一段后，会释放 GIL，以允许别的线程开始利用资源。

读者可能会问，为什么 Python 线程会去主动释放 GIL 呢？毕竟，如果仅仅要求 Python 线程在开始执行时锁住 GIL，且永远不去释放 GIL，那别的线程就都没有运行的机会。其实，CPython 中还有另一个机制，叫做间隔式检查(check_interval)，意思是 CPython 解释器会去轮询检查线程 GIL 的锁住情况，每隔一段时间，Python 解释器就会强制当前线程去释放 GIL，这样别的线程才能有执行的机会。

注意 不同版本的 Python 其间隔式检查的实现方式并不一样。早期的 Python 是 100 个刻度（大致对应了 1000 个字节码）；而 Python 3 以后，间隔时间大致为 15 毫秒。当然，我们不必细究具体多久会强制释放 GIL，读者只需要明白，CPython 解释器会在一个“合理”的时间范围内释放 GIL 就可以了。

整体来说，每一个 Python 线程都是类似这样循环的封装，来看下面这段代码：

```
1.          for (;;) {
2.              if (--ticker < 0) {
3.                  ticker = check_interval;
4.                  /* Give another thread a chance */
5.                  PyThread_release_lock(interpreter_lock);
6.                  /* Other threads may run now */
7.                  PyThread_acquire_lock(interpreter_lock, 1);
8.              }
9.              bytecode = *next_instr++;
10.             switch (bytecode) {
11.                 /* execute the next instruction ... */
```

```
            }
        }
```

从这段代码中可以看出，每个 Python 线程都会先检查 ticker 计数。只有在 ticker 大于 0 的情况下，线程才会去执行自己的代码。

Python GIL 不能绝对保证线程安全

注意，有了 GIL，并不意味着 Python 程序员就不用去考虑线程安全了，因为即便 GIL 仅允许一个 Python 线程执行，但别忘了 Python 还有 check interval 这样的抢占机制。

比如，运行如下代码：

```
import threading
n = 0
def foo():
    global n
    n += 1
threads = []
for i in range(100):
    t = threading.Thread(target=foo)
    threads.append(t)
for t in threads:
    t.start()
for t in threads:
    t.join()
print(n)
```

执行此代码会发现，其大部分时候会打印 100，但有时也会打印 99 或者 98，原因在于 n+=1 这一句代码让线程并不安全。如果去翻译 foo 这个函数的字节码就会发现，它实际上是由下面四行字节码组成：

```
>>> import dis
>>> dis.dis(foo)
LOAD_GLOBAL              0 (n)
LOAD_CONST               1 (1)
INPLACE_ADD
STORE_GLOBAL             0 (n)
```

而这四行字节码中间都是有可能被打断的！所以，千万别以为有了 GIL 程序就不会产生线程问题，我们仍然需要注意线程安全。

25.深度解析 Python 垃圾回收机制

我们知道，目前的计算机都采用的是图灵机架构，其本质就是用一条无限长的纸带，对应今天的存储器。随后在工程学的推演中，逐渐出现了寄存器、易失性存储器（内存）以及永久性存储器（硬盘）等产品。由于不同的存储器，其速度越快，单位价格也就越昂贵，因此，妥善利用好每一寸告诉存储器的空间，永远是系统设计的一个核心。

Python 程序在运行时，需要在内存中开辟出一块空间，用于存放运行时产生的临时变量，计算完成后，再将结果输出到永久性存储器中。但是当数据量过大，或者内存空间管理不善，就很容易出现内存溢出的情况，程序可能会被操作系统终止。

而对于服务器这种用于永不中断的系统来说，内存管理就显得更为重要了，不然很容易引发内存泄漏。

这里的内存泄漏是指程序本身没有设计好，导致程序未能释放已不再使用的内存，或者直接失去了对某段内存的控制，造成了内存的浪费。

那么，对于不会再用到的内存空间，Python 是通过什么机制来管理的呢？其实在前面章节已大致接触过，就是引用计数机制。

Python 引用计数机制

在学习 Python 的整个过程中，我们一直在强调，Python 中一切皆对象，也就是说，在 Python 中你用到的一切变量，本质上都是类对象。

那么，如何知道一个对象永远都不能再使用了呢？很简单，就是当这个对象的引用计数值为 0 时，说明这个对象永不再用，自然它就变成了垃圾，需要被回收。

举个例子：

```
1.          import os
2.          import psutil
3.
4.          # 显示当前 python 程序占用的内存大小
5.          def show_memory_info(hint):
6.              pid = os.getpid()
7.              p = psutil.Process(pid)
8.
9.              info = p.memory_full_info()
10.             memory = info.uss / 1024. / 1024
11.             print('{} memory used: {} MB'.format(hint, memory))
12.         def func():
13.             show_memory_info('initial')
14.             a = [i for i in range(10000000)]
```

```
15.             show_memory_info('after a created')
16.
17.         func()
18.         show_memory_info('finished')
```

输出结果为：

```
initial memory used: 47.19140625 MB

after a created memory used: 433.91015625 MB

finished memory used: 48.109375 MB
```

注意，运行此程序之前，需安装 psutil 模块（获取系统信息的模块），可使用 pip 命令直接安装，执行命令为 $pip install psutil，如果遇到 Permission denied 安装失败，请加上 sudo 重试。

可以看到，当调用函数 func() 且列表 a 被创建之后，内存占用迅速增加到了 433 MB，而在函数调用结束后，内存则返回正常。这是因为，函数内部声明的列表 a 是局部变量，在函数返回后，局部变量的引用会注销掉，此时列表 a 所指代对象的引用计数为 0，Python 便会执行垃圾回收，因此之前占用的大量内存就又回来了。

明白了这个原理后，稍微修改上面的代码，如下所示：

```
1.         def func():
2.             show_memory_info('initial')
3.             global a
4.             a = [i for i in range(10000000)]
5.             show_memory_info('after a created')
6.
```

```
func()
show_memory_info('finished')
```

输出结果为：

```
initial memory used: 48.88671875 MB

after a created memory used: 433.94921875 MB

finished memory used: 433.94921875 MB
```

上面这段代码中，global a 表示将 a 声明为全局变量，则即使函数返回后，列表的引用依然存在，于是 a 对象就不会被当做垃圾回收掉，依然占用大量内存。

同样，如果把生成的列表返回，然后在主程序中接收，那么引用依然存在，垃圾回收也不会被触发，大量内存仍然被占用着：

```
def func():
    show_memory_info('initial')
    a = [i for i in derange(10000000)]
    show_memory_info('after a created')
    return a

a = func()
show_memory_info('finished')
```

输出结果为：

```
initial memory used: 47.96484375 MB

after a created memory used: 434.515625 MB

finished memory used: 434.515625 MB
```

以上最常见的几种情况，下面由表及里，深入看一下 Python 内部的引用计数机制。先来分析一段代码：

```
1.          import sys
2.          a = []
3.          # 两次引用，一次来自 a，一次来自 getrefcount
4.          print(sys.getrefcount(a))
5.          def func(a):
6.              # 四次引用，a，python 的函数调用栈，函数参数，和 getrefcount
7.              print(sys.getrefcount(a))
8.          func(a)
9.          # 两次引用，一次来自 a，一次来自 getrefcount，函数 func 调用已经不存在
10.         print(sys.getrefcount(a))
```

输出结果为：

```
2

4

2
```

注意 ,sys.getrefcount() 函数用于查看一个变量的引用次数 ,不过别忘了 ,getrefcount 本身也会引入一次计数。

另一个要注意的是，在函数调用发生的时候，会产生额外的两次引用，一次来自函数栈，另一个是函数参数。

```
1.          import sys
2.          a = []
3.          print(sys.getrefcount(a)) # 两次
4.          b = a
```

```
5.          print(sys.getrefcount(a)) # 三次
6.          c = b
7.          d = b
8.          e = c
9.          f = e
10.         g = d
11.         print(sys.getrefcount(a)) # 八次
```

输出结果为：

```
2

3

8
```

分析一下这段代码，a、b、c、d、e、f、g 这些变量全部指代的是同一个对象，而 sys.getrefcount() 函数并不是统计一个指针，而是要统计一个对象被引用的次数，所以最后一共会有 8 次引用。

理解引用这个概念后，引用释放是一种非常自然和清晰的思想。相比 C 语言中需要使用 free 去手动释放内存，Python 的垃圾回收在这里可以说是省心省力了。

不过，有读者还是会好奇，如果想手动释放内存，应该怎么做呢？方法同样很简单，只需要先调用 del a 来删除一个对象，然后强制调用 gc.collect() 即可手动启动垃圾回收。例如：

```
1.          import gc
2.
3.          show_memory_info('initial')
4.          a = [i for i in range(10000000)]
5.          show_memory_info('after a created')
```

```
6.          del a
7.          gc.collect()
8.          show_memory_info('finish')
9.          print(a)
```

输出结果为：

```
initial memory used: 48.1015625 MB

after a created memory used: 434.3828125 MB

finish memory used: 48.33203125 MB

NameError Traceback (most recent call last)

<ipython-input-12-153e15063d8a> in <module>

    11

    12 show_memory_info('finish')

---> 13 print(a)

NameError: name 'a' is not defined
```

是不是觉得垃圾回收非常简单呢？这里再问大家一个问题：引用次数为 0 是垃圾回收启动的充要条件吗？还有没有其他可能性呢？

其实，引用计数是其中最简单的实现，引用计数并非充要条件，它只能算作充分非必要条件，至于其他的可能性，下面所讲的循环引用正是其中一种。

循环引用

首先思考一个问题，如果有两个对象，之间互相引用，且不再被别的对象所引用，那么它们

应该被垃圾回收吗？

举个例子：

```
        def func():
            show_memory_info('initial')
            a = [i for i in range(10000000)]
            b = [i for i in range(10000000)]
            show_memory_info('after a, b created')
            a.append(b)
            b.append(a)

        func()
        show_memory_info('finished')
```

输出结果为：

```
initial memory used: 47.984375 MB
after a, b created memory used: 822.73828125 MB
finished memory used: 821.73046875 MB
```

程序中，a 和 b 互相引用，并且作为局部变量在函数 func 调用结束后，a 和 b 这两个指针从程序意义上已经不存在，但从输出结果中看到，依然有内存占用，这是为什么呢？因为互相引用导致它们的引用数都不为 0。

试想一下，如果这段代码出现在生产环境中，哪怕 a 和 b 一开始占用的空间不是很大，但经过长时间运行后，Python 所占用的内存一定会变得越来越大，最终撑爆服务器，后果不堪设想。

有读者可能会说，互相引用还是很容易被发现的呀，问题不大。可是，更隐蔽的情况是出现一个引用环，在工程代码比较复杂的情况下，引用环真不一定能被轻易发现。那么应该怎么做呢？

事实上，Python 本身能够处理这种情况，前面刚刚讲过，可以显式调用 gc.collect() 来启动垃圾回收，例如：

```
import gc

def func():
    show_memory_info('initial')
    a = [i for i in range(10000000)]
    b = [i for i in range(10000000)]
    show_memory_info('after a, b created')
    a.append(b)
    b.append(a)

func()
gc.collect()
show_memory_info('finished')
```

输出结果为：

```
initial memory used: 49.51171875 MB

after a, b created memory used: 824.1328125 MB

finished memory used: 49.98046875 MB
```

事实上，Python 使用标记清除（mark-sweep）算法和分代收集（generational），来启用针对循环引用的自动垃圾回收。

先来看标记清除算法。我们先用图论来理解不可达的概念。对于一个有向图，如果从一个节点出发进行遍历，并标记其经过的所有节点；那么，在遍历结束后，所有没有被标记的节点，我们就称之为不可达节点。显而易见，这些节点的存在是没有任何意义的，自然的，我们就需要对它们进行垃圾回收。

当然，每次都遍历全图，对于 Python 而言是一种巨大的性能浪费。所以，在 Python 的垃圾回收实现中，标记清除算法使用双向链表维护了一个数据结构，并且只考虑容器类的对象（只有容器类对象才有可能产生循环引用）。

而分代收集算法，则是将 Python 中的所有对象分为三代。刚刚创立的对象是第 0 代；经过一次垃圾回收后，依然存在的对象，便会依次从上一代挪到下一代。而每一代启动自动垃圾回收的阈值，则是可以单独指定的。当垃圾回收器中新增对象减去删除对象达到相应的阈值时，就会对这一代对象启动垃圾回收。

事实上，分代收集基于的思想是，新生的对象更有可能被垃圾回收，而存活更久的对象也有更高的概率继续存活。因此，通过这种做法，可以节约不少计算量，从而提高 Python 的性能。

由于篇幅有限，这里不再对标记清除算法和分代收集算法的具体实现所详细介绍，有兴趣的读者可自行百度搜索。

第 17 章 Python 网络编程

本章将由浅入深地介绍 Python 网络编程相关的知识。首先给读者简要介绍计算机网络的基础知识，比如 TCP/IP 协议、端口等，这是进行网络编程的基础。

同时，本章将详细介绍基于 TCP 协议和 UDP 协议的 socket 通信，这是基于传输层协议的编程，属于较底层的、真正的网络编程。

本章最后还将介绍两个应用层协议的网络编程，smtplib 模块和 poplib 模块的使用，它们是 Python 实际网络编程中应用非常广泛的功能。

1.什么是计算机网络，TCP/IP 协议和端口号

时至今日，计算机网络缩短了人们之间的距离，把“地球村”变成现实，网络应用已经成为计算机领域最广泛的应用。

所谓计算机网络 ,就是把分布在不同地理区域的计算机与专门的外部设备用通信线路互联成一个规模大、功能强的网络系统 ,从而使众多的计算机可以方便地互相传递信息 ,共享硬件、软件、数据信息等资源。

计算机网络是现代通信技术与计算机技术相结合的产物 ,计算机网络可以提供如下一些主要功能：

- 资源共享。
- 信息传输与集中处理。
- 均衡负荷与分布处理。

- 综合信息服务。

通过计算机网络可以向全社会提供各种经济信息、科研情报和咨询服务等。其中，国际互联网 Internet 上的全球信息网（WWW, World Wide Web）服务就是一个最典型的、最成功的例子。实际上，今天的网络承载了绝大部分大型企业的运转，一个大型的、全球性的企业或组织的日常工作流程都是建立在互联网基础之上的。

计算机网络有很多种类型，根据不同的分类原则，可以得到不同类型的计算机网络。通常计算机网络是按照规模大小和延伸范围来分类的，常见的类型有：局域网（LAN）、城域网（MAN）和广域网（WAN）。Internet 可以被视为世界上最大的广域网。

在计算机网络中实现通信必须有一些约定，这些约定被称为通信协议。通信协议负责对传输速率、传输代码、代码结构、传输控制步骤、出错控制等制定处理标准。为了让两个节点能进行对话，必须在它们之间建立通信工具，使彼此之间能进行信息交换。

通信协议通常由三部分组成：

1. 一是语义部分，用于决定双方对话的类型；
2. 二是语法部分，用于决定双方对话的格式；
3. 三是变换规则，用于决定通信双方的应答关系；

国际标准化组织(ISO)于 1978 年提出了“开放系统互连参考模型”，即著名的 OSI(Open

System Interconnection）参考模型。OSI 参考模型力求将网络简化，并以模块化的方式来设计网络。

OSI 参考模型把计算机网络分成物理层、数据链路层、网络层、传输层、会话层、表示层、应用层七层，受到计算机界和通信业的极大关注。经过十多年的发展和推进，OSI 模式已成为各种计算机网络结构的参考标准。图 1 显示了 OSI 参考模型的推荐分层。

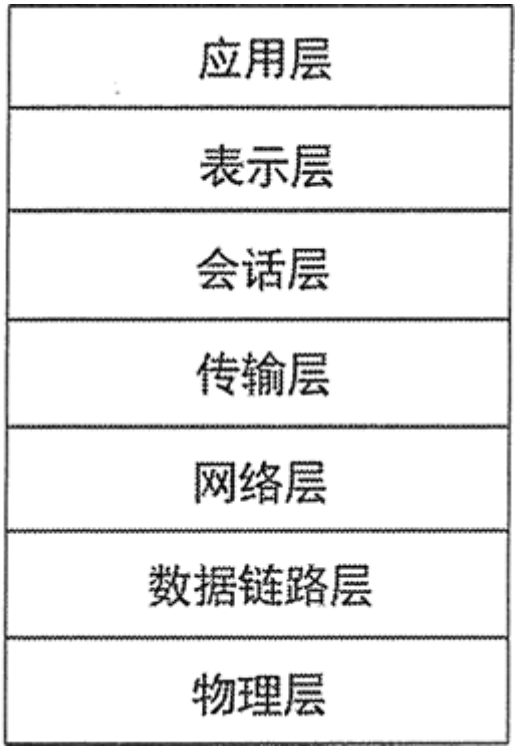


图 1 OSI 参考模型的推荐分层

通信协议是网络通信的基础，IP 协议则是一种非常重要的通信协议。IP(Internet Protocol) 又称网际协议，是支持网间互联的数据报协议。IP 协议提供了网间连接的完善功能，包括 IP 数据报规定的互联网络范围内的地址格式。

经常与 IP 协议放在一起的还有 TCP(Transmission Control Protocol),即传输控制协议，它规定了一种可靠的数据信息传递服务。虽然 IP 和 TCP 这两个协议的功能不尽相同，也可以分开单独使用 ,但它们是在同一个时期作为一个协议来设计的 ,并且在功能上是互补的，因此，在实际使用中常常把这两个协议统称为 TCP/IP 协议。

TCP/IP 协议最早出现在 UNIX 操作系统中，现在几乎所有的操作系统都支持 TCP/IP 协议，因此 TCP/IP 协议也是 Internet 中最常用的基础协议。

按照 TCP/IP 协议模型，网络模型通常被分为四层。OSI 参考模型和 TCP/IP 分层模型的大致对应关系如图 2 所示。

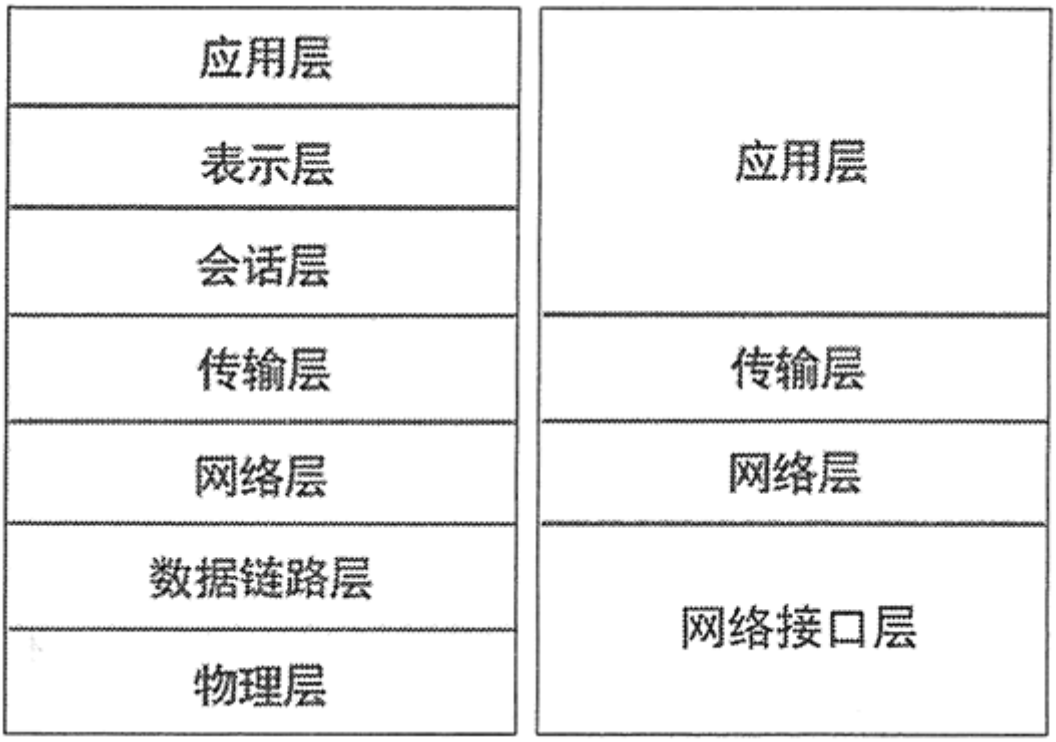


图 2 OSI 参考模型和 TCP/IP 分层模型的大致对应关系

IP 地址和端口号

IP 地址用于唯一标识网络中的一个通信实体，这个通信实体既可以是一个主机，也可以是

一台打印机，或者是路由器的某一个端口。而在基于 IP 协议的网络中传输的数据包，都必须使用 IP 地址来进行标识。

就像写一封信，要标明收信人的地址和寄信人的地址，而邮政工作人员则通过该地址来决定信件的去向。类似的过程也发生在计算机网络中，被传输的每一个数据包也要包括一个源 IP 地址和一个目的 IP 地址。当该数据包在网络中进行传输时，这两个地址要保持不变，以确保网络设备总能根据确定的 IP 地址，将数据包从源通信实体送往指定的目的通信实体。

IP 地址是数字型的，它是一个 32 位(32 bit)整数。但为了便于记忆，通常把它分成 4 个 8 位的二进制数，每 8 位之间用圆点隔开，每个 8 位整数都可以转换成一个 0~255 的十进制整数，因此日常看到的 IP 地址常常是这种形式：202.9.128.88。

NIC (Internet Network Information Center) 统一负责全球 Internet IP 地址的规划和管理，而 InterNIC、APNIC、RIPE 三大网络信息中心则具体负责美国及其他地区的 IP 地址分配。其中 APNIC 负责亚太地区的 IP 地址管理，我国申请 IP 地址也要通过 APNIC，APNIC 的总部设在日本东京大学。

IP 地址被分成 A、B、C、D、E 五类，每个类别的网络标识和主机标识各有规则：

- A 类：10.0.0.0~10.255.255.255
- B 类：172.16.0.0~172.31.255.255
- C 类：192.168.0.0~192.168.255.255

IP 地址用于唯一标识网络上的一个通信实体，但一个通信实体可以有多个通信程序同时提供网络服务，此时还需要使用端口。

端口是一个 16 位的整数，用于表示将数据交给哪个通信程序处理。因此，端口就是应用程序与外界交流的出入口，它是一种抽象的软件结构，包括一些数据结构和 I/O（输入/输出）缓冲区。

不同的应用程序处理不同端口上的数据，在同一台机器中不能有两个程序使用同一个端口。端口号可以为 0~65535，通常将端口分为如下三类：

1. 公认端口（Well Known Port）：端口号为 0~1023，它们紧密地绑定（Binding）一些特定的服务。
2. 注册端口（Registered Port）：端口号为 1024~49151，它们松散地绑定一些服务。应用程序通常应该使用这个范围内的端口。
3. 动态和/或私有端口（Dynamic and/or Private Port）：端口号为 49152～65535，这些端口是应用程序使用的动态端口，应用程序一般不会主动使用这些端口。

如果把应用程序比作人，把计算机网络比作类似于邮递员的角色，把 IP 地址理解为某个人所在地方的地址（包括街道和门牌号），但仅有地址是找不到这个人的，还需要知道这个人所在的房间号才可以找到他，这个房间号就相当于端口号。因此，当一个程序需要发送数据时，需要指定目的地的 IP 地址和端口号，只有指定了正确的 IP 地址和端口号，计算机网

络才可以将数据发送给该 IP 地址和端口号所对应的程序。

2.Python 网络编程模块

根据前面对网络分层棋型的介绍，我们知道实际的网络模型大致分为四层，这四层各有对应的网络协议提供支持，如图 1 所示。

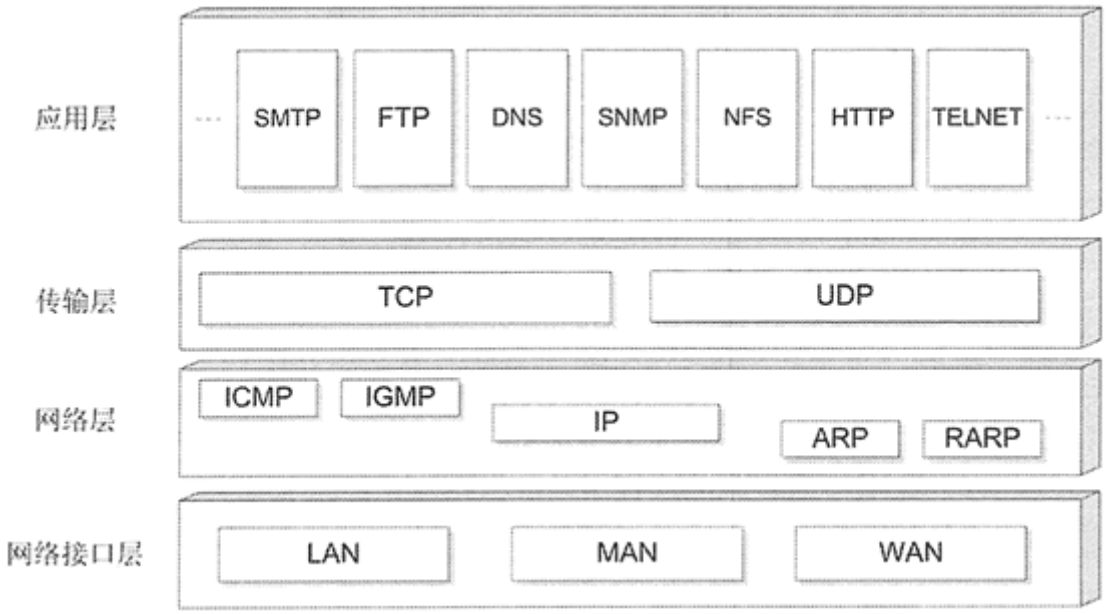


图 1 四层网络模型及对应的协议

网络层协议主要是 IP，它是所有互联网协议的基础，其中 ICMP（Internet Control Message Protocol）、IGMP（Internet Group Manage Protocol）、ARP（Address Resolution Protocol）、RARP（Reverse Address Resolution Protocol）等协议都可认为是 IP 协议族的子协议。通常来说，很少会直接基于网络层进行应用程序编程。

传输层协议主要是 TCP 和 UDP，Python 提供了 socket 等模块针对传输层协议进行编程。

应用层协议就更多了，正如图 1 所示的，FTP、HTTP、TELNET 等协议都属于应用层协议，Python 同样为基于应用层协议的编程提供了丰富的支持。

虽然 Python 自带的标准库已经提供了很多与网络有关的模块，但如果在使用时觉得不够方便，则不要忘记了 Python 的优势，即大量的第三方模块随时可用于增强 Python 的功能。

表 2 显示了 Python 标准库中的网络相关模块。

表 2 Python 标准库中的网络相关模块

模块	描述
socket	基于传输层 TCP、UDP 协议进行网络编程的模块
asyncore	socket 模块的异步版，支持基于传输层协议的异步通信
asynchat	asyncore 的增强版
cgi	基本的 CGI（Common Gateway Interface，早期开发动态网站的技术）支持
email	E-mail 和 MLME 消息处理模块
ftplib	支持 FTP 协议的客户端模块
httplib、http.client	支持 HTTP 协议以及 HTTP 客户揣的模块

imaplib	支持 IMAP4 协议的客户端模块
mailbox	操作不同格式邮箱的模块
mailcap	支持 Mailcap 文件处理的模块
nntplib	支持 NTTP 协议的客户端模块
smtplib	支持 SMTP 协议（发送邮件）的客户端模块
poplib	支持 POP3 协议的客户端模块
telnetlib	支持 TELNET 协议的客户端模块
urllib 及其子模块	支持 URL 处理的模块
xmlrpc、xmlrpc.server、 xmlrpc.client	支持 XML-RPC 协议的服务器端和客户端模块

3.Python urllib.parseurllib.parse 模块用法详解

URL（Uniform Resource Locator）对象代表统一资源定位器，它是指向互联网“资源”的指针。资源可以是简单的文件或目录，也可以是对复杂对象的引用，例如对数据库或搜索引擎的查询。在通常情况下，URL 可以由协议名、主机、端口和资源路径组成，即满足如下格式：

```
protocol://host:port/path
```

例如如下的 URL 地址：

```
http://www.crazyit.org/index.php
```

urllib 模块则包含了多个用于处理 URL 的子模块：

- urllib.request：这是最核心的子模块，它包含了打开和读取 URL 的各种函数。
- urllib.error：主要包含由 urllib.request 子模块所引发的各种异常。
- urllib.parse：用于解析 URL。
- urllib.robotparser：主要用于解析 robots.txt 文件。

通过使用 urllib 模块可以打开任意 URL 所指向的资源，就像打开本地文件一样，这样程序就能完整地下载远程页面。如果再与 re 模块结合使用，那么程序完全可以提取页面中各种信息，这就是所谓的“网络爬虫”的初步原理。

> 在 Python2.x 中，urllib 模块被分为 urllib 和 urllib2 两个模块，其中 urllib 主要用于简单的下载，而 urllib2 则可实现 HTTP 验证，cookie 管理。

下面先介绍 urllib.parse 子模块中用于解析 URL 地址和查询字符串的函数：

urllib.parse.urlparse(urlstring, scheme='', allow_fragments=True)：该函数用于解析 URL 字符串。程序返回一个 ParseResult 对象，可以获取解析出来的数据。
urllib.parse.urlunparse(parts)：该函数是上一个函数的反向操作，用于将解析结果反向拼接成 URL 地址。
urllib.parse.parse_qs(qs, keep_blank_values=False, strict_parsing=False, encoding='utf-8', errors='replace')：该该函数用于解析查询字符串（application/x-www-form-urlencoded 类型的数据），并以 dict 形式返回解析结果。

urllib.parse.parse_qsl(qs, keep_blank_values=False, strict_parsing=False, encoding='utf-8', errors='replace')：该函数用于解析查询字符串（application/x-www-form-urlencoded 类型的数据），并以列表形式返回解析结果。
urllib.parse.urlencode(query, doseq=False, safe='', encoding=None, errors=None, quote_via=quote_plus)：将字典形式或列表形式的请求参数恢复成请求字符串。该函数相当于 parse_qs()、parse_qsl() 的逆函数。
urllib.parse.urljoin(base, url, allow_fragments=True)：该函数用于将一个 base_URL 和另一个资源 URL 连接成代表绝对地址的 URL。

例如，如下程序使用 urlparse() 函数来解析 URL 字符串：

```
1.          from urllib.parse import *
2.
3.          # 解析 URL 字符串
4.          result =
    urlparse('http://www.crazyit.org:80/index.php;yeeku?name=fkit#frag')
5.          print(result)
6.          # 通过属性名和索引来获取 URL 的各部分
7.          print('scheme:', result.scheme, result[0])
8.          print('主机和端口:', result.netloc, result[1])
9.          print('主机:', result.hostname)
10.         print('端口:', result.port)
11.         print('资源路径:', result.path, result[2])
12.         print('参数:', result.params, result[3])
13.         print('查询字符串:', result.query, result[4])
14.         print('fragment:', result.fragment, result[5])
15.         print(result.geturl())
```

上面程序中使用 urlparse() 函数解析 URL 字符串，解析结果是一个 ParseResult 对象，该对象实际上是 tuple 的子类。因此，程序既可通过属性名来获取 URL 的各部分，也可通过索引来获取 URL 的各部分。

表 1 显示了 ParseResult 各属性与元组索引的对应关系。

表 1 ParseResult 各属性与元组索引的对应关系

属性名	元组索引	返回值	默认值
scheme	0	返回 URL 的 scheme	scheme 参数
netloc	1	网络位置部分（主机名 + 端口）	空字符串
path	2	资源路径	空字符串
params	3	资源路径的附加参数	空字符串
query	4	查询字符串	空字符串
fragment	5	Fragment 标识符	空字符串
username		用户名	None
password		密码	None
hostname		主机名	None
port		端口	None

上面程序使用 urlparse() 函数解析 URL 字符串之后，分别使用了属性名和索引来获取 URL 的各部分。运行上面程序，将看到如下输出结果：

```
ParseResult(scheme='http', netloc='www.crazyit.org:80', path='/index.php',
params='yeeku', query='name=fkit', fragment='frag')
scheme: http http
```

```
主机和端口: www.crazyit.org:80 www.crazyit.org:80
主机: www.crazyit.org
端口: 80
资源路径: /index.php /index.php
参数: yeeku yeeku
查询字符串: name=fkit name=fkit
fragment: frag frag
http://www.crazyit.org:80/index.php;yeeku?name=fkit#frag
```

如果使用 urlunparse() 函数，则可以把一个 ParseResult 对象或元组恢复成 URL 字符串。例如如下代码：

```
result = urlunparse(('http', 'www.crazyit.org:80', 'index.php',
    'yeeku', 'name=fkit', 'frag'))
print('URL 为:', result)
```

运行上面程序，将看到如下输出结果：

```
URL 为: http://www.crazyit.org:80/index.php;yeeku?name=fkit#frag
```

如果被解析的 URL 以双斜线（//）开头，那么 urlparse() 函数可以识别出主机，只是缺少 scheme 部分。但如果被解析的 URL 既没有 scheme，也没有以双斜线（//）开头，那么 urlparse() 函数将会把这些 URL 都当成资源路径。例如如下代码：

```
        # 解析以//开头的 URL
        result = urlparse('//www.crazyit.org:80/index.php')
        print('scheme:', result.scheme, result[0])
        print('主机和端口:', result.netloc, result[1])
        print('资源路径:', result.path, result[2])
        print('------------------')
        # 解析没有 scheme，也没有以双斜线（//）开头的 URL
        # 从开头部分开始就会被当成资源路径
        result = urlparse('www.crazyit.org/index.php')
        print('scheme:', result.scheme, result[0])
        print('主机和端口:', result.netloc, result[1])
        print('资源路径:', result.path, result[2])
```

运行上面程序， 可以看到如下输出结果：

```
scheme:
主机和端口: www.crazyit.org:80 www.crazyit.org:80
资源路径: /index.php /index.php
----------
scheme:
主机和端口:
资源路径: www.crazyit.org/index.php www.crazyit.org/index.php
```

parse_qs() 和 parse_qsl()（这个 l 代表 list）两个函数都用于解析查询字符串，只不过返回值不同而已，parse_qsl() 函数的返回值是 list（正如该函数名所暗示的）。urlencode() 则是它们的逆函数。例如如下代码：

```
        # 解析查询字符串，返回 dict
```

```
result = parse_qs('name=fkit&name=%E7%96%AF%E7%8B%82java&age=12')
print(result)
# 解析查询字符串，返回 list
result = parse_qsl('name=fkit&name=%E7%96%AF%E7%8B%82java&age=12')
print(result)
# 将列表格式的请求参数恢复成请求参数字符串
print(urlencode(result))
```

运行上面程序，将看到如下输出结果：

```
{'name': ['fkit', '疯狂 java'], 'age': ['12']}
[('name', 'fkit'), ('name', '疯狂 java'), ('age', '12')]
name=fkit&name=%E7%96%AF%E7%8B%82java&age=12
```

从上面的输出结果可以看到，parse_qs() 函数返回了一个 dict，其中 key 是参数名，value 是参数值。而 parse_qsl() 函数返回了一个 list，每个元素代表一个查向参数。

urljoin() 函数负责将两个 URL 拼接在一起，返回代表绝对地址的 URL。这里主要可能出现 3 种情况：

1. 被拼接的 URL 只是一个相对路径 path（不以斜线开头），那么该 URL 将会被拼接到 base 之后，如果 base 本身包含 path 部分，则用被拼接的 URL 替换 base 所包含的 path 部分。
2. 被拼接的 URL 是一个根路径 path（以单斜线开头），那么该 URL 将会被拼接到 base 的域名之后。
3. 被拼接的 URL 是一个绝对路径 path（以双斜线开头），那么该 URL 将会被拼接到 base 的 scheme 之后。

例如， 如下代码示范了 urljoin() 函数的功能和用法：

```
纯文本复制
```

```
1.          # 被拼接 URL 不以斜线开头
2.          result = urljoin('http://www.crazyit.org/users/login.html',
'help.html')
3.          print(result) # http://www.crazyit.org/users/help.html
4.          result = urljoin('http://www.crazyit.org/users/login.html',
'book/list.html')
5.          print(result) # http://www.crazyit.org/users/book/list.html
6.          # 被拼接 URL 以斜线（代表根路径 path）开头
7.          result = urljoin('http://www.crazyit.org/users/login.html',
'/help.html')
8.          print(result) # http://www.crazyit.org/help.html
9.          # 被拼接 URL 以双斜线（代表绝对 URL）开头
10.         result = urljoin('http://www.crazyit.org/users/login.html',
'//help.html')
11.         print(result) # http://help.html
```

上面代码已经给出了程序运行结果，通过运行结果可以看到 urljoin() 函数的功能。

4.Python urllib.request

urllib.request 模块读取资源用法详解

在 urllib.request 子模块下包含了一个非常实用的 urllib.request.urlopen(url, data=None) 方法，该方法用于打开 url 指定的资源，并从中读取数据。根据请求 url 的不同，该方法的返回值会发生动态改变。如果 url 是一个 HTTP 地址，那么该方法返回一个 http.client.HTTPResponse 对象。

例如如下程序：

```
from urllib.request import *

# 打开 URL 对应的资源
result = urlopen('http://www.crazyit.org/index.php')
# 按字节读取数据
data = result.read(326)
# 将字节数据恢复成字符串
print(data.decode('utf-8'))

# 用 context manager 来管理打开的 URL 资源
with urlopen('http://www.crazyit.org/index.php') as f:
    # 按字节读取数据
    data = f.read(326)
    # 将字节数据恢复成字符串
    print(data.decode('utf-8'))
```

上面程序都是向 http://www.crazyit.org/index.php 发送请求，并请求下载该页面的内容。只不过第一个示例程序是直接获取 http://www.crazyit.org/index.php 资源的数据；第二个示例程序则使用 context manager 来管理通过 urlopen 打开的资源。

运行上面程序，可以看到程序分两次获取并输出了 http://www.crazyit.org/index.php 页面内容。

在使用 urlopen() 函数时，可以通过 data 属性向被请求的 URL 发送数据。例如如下程序：

```
from urllib.request import *

# 向 https://localhost/cgi-bin/test.cgi 发送请求数据
#with urlopen(url='https://localhost/cgi-bin/test.cgi',
with urlopen(url='http://localhost:8888/test/test',     #①
```

```
6.              data='测试数据'.encode('utf-8')) as f:
7.              # 读取服务器全部响应
8.              print(f.read().decode('utf-8'))
```

上面程序为 data 属性指定了一个 bytes 字节数据，该字节数据会以原始二进制流的方式提交给服务器。

上面程序需要在本地（localhost）部署一个 Web 应用，该程序对应的服务器端所使用的 CGI 代码为：

```
#!/usr/bin/env python

import sys

data = sys.stdin.read()

print('Content-type: text/plain\n\nGot Data:"%s"' % data)
```

运行上面程序，可以看到如下输出结果：

```
Got Data:"测试数据"
```

如果使用 urlopen() 函数向服务器页面发送 GET 请求参数，则无须使用 data 属性，直接把请求参数附加在 URL 之后即可。例如如下代码：

```
1.          import urllib.parse
2.          params = urllib.parse.urlencode({'name': 'fkit', 'password': '123888'})
3.          # 将请求参数添加到 URL 的后面
4.          url = 'http://localhost:8888/test/get.jsp?%s' % params
5.          with urlopen(url=url) as f:
```

```
6.              # 读取服务器全部响应
7.              print(f.read().decode('utf-8'))
```

正如程序中第 4 行代码所看到的，程序在发送 GET 请求参数时，只要将请求参数附加到 URL 的后面即可（上面程序同样需要服务器的支持）。

运行上面程序，可以看到如下输出结果：

```
name 参数：fkit
password 参数：123888
```

如果想通过 urlopen() 函数发送 POST 请求参数，则同样可通过 data 属性来实现。例如如下代码：

```
1.          import urllib.parse
2.          params = urllib.parse.urlencode({'name': '疯狂软件', 'password':
'123888'})
3.          params = params.encode('utf-8')
4.          # 使用 data 指定请求参数
5.          with urlopen("http://localhost:8888/test/post.jsp", data=params) as f:
6.              print(f.read().decode('utf-8'))
```

从上面第 5 行代码可以看到，如果要向指定地址发送 POST 请求，那么通过 data 指定请求参数即可。

实际上，使用 data 属性不仅可以发送 POST 请求，还可以发送 PUT、PATCH、DELETE 等请求，此时需要使用 urllib.request.Request 来构建请求参数。程序使用 urlopen() 函

数打开远程资源时，第一个 url 参数既可以是 URL 字符串，也可以使用 urllib.request.Request 对象。

urllib.request.Request 对象的构造器如下：

```
urllib.request.Request(url, data=None, headers={}, origin_req_host=None,
unverifiable=False, method=None)
```

从该构造器可以看出，使用 Request 可以通过 method 指定请求方法，也可以通过 data 指定请求参数，还可以通过 headers 指定请求头。

下面代码示范了如何通过 Request 对象来发送 PUT 请求：

```
from urllib.request import *

params = 'put 请求数据'.encode('utf-8')
# 创建 Request 对象，设置使用 PUT 请求
req = Request(url='http://localhost:8888/test/put',
    data=params, method='PUT')
with urlopen(req) as f:
    print(f.status)
    print(f.read().decode('utf-8'))
```

正如从代码中所看到的，程序在创建 Request 对象时通过 method 指定使用 PUT 请求方式，这意味着程序会发送 PUT 请求。

正如刚刚所提到的，程序也可以使用 Request 对象来添加请求头。例如如下代码：

```
        # 创建 Request 对象
        req = Request('http://localhost:8888/test/header.jsp')
        # 添加请求头
        req.add_header('Referer', 'http://www.crazyit.org/')
        with urlopen(req) as f:
            print(f.status)
            print(f.read().decode('utf-8'))
```

程序通过 Request 的 add_header() 方法添加了一个 Referer 请求头，服务器端的处理程序将可以读到此处添加的请求头。

通过 urlopen() 函数打开远程资源之后，也可以非常方便地读取远程资源（甚至实现多线程下载）。如下程序实现了一个多线程下载的工具类：

```
        from urllib.request import *
        import threading

        class DownUtil:
            def __init__(self, path, target_file, thread_num):
                # 定义下载资源的路径
                self.path = path
                # 定义需要使用多少个线程下载资源
                self.thread_num = thread_num
                # 指定所下载的文件的保存位置
                self.target_file = target_file
                # 初始化 threads 数组
                self.threads = []
            def download(self):
                # 创建 Request 对象
                req = Request(url=self.path, method='GET')
                # 添加请求头
                req.add_header('Accept', '*/*')
                req.add_header('Charset', 'UTF-8')
```

```
20.                 req.add_header('Connection', 'Keep-Alive')
21.                 # 打开要下载的资源
22.                 f = urlopen(req)
23.                 # 获取要下载的文件大小
24.                 self.file_size = int(dict(f.headers).get('Content-Length', 0))
25.                 f.close()
26.                 # 计算每个线程要下载的资源大小
27.                 current_part_size = self.file_size // self.thread_num + 1
28.                 for i in range(self.thread_num):
29.                     # 计算每个线程下载的开始位置
30.                     start_pos = i * current_part_size
31.                     # 每个线程使用一个 wb 模式打开的文件进行下载
32.                     t = open(self.target_file, 'wb')
33.                     # 定位该线程的下载位置
34.                     t.seek(start_pos, 0);
35.                     # 创建下载线程
36.                     td = DownThread(self.path, start_pos, current_part_size, t)
37.                     self.threads.append(td)
38.                     # 启动下载线程
39.                     td.start()
40.             # 获取下载的完成百分比
41.             def get_complete_rate(self):
42.                 # 统计多个线程已经下载的总大小
43.                 sum_size = 0
44.                 for i in range(self.thread_num):
45.                     sum_size += self.threads[i].length
46.                 # 返回已经完成的百分比
47.                 return sum_size / self.file_size
48.         class DownThread(threading.Thread):
49.             def __init__(self, path, start_pos, current_part_size,
    current_part):
50.                 super().__init__()
51.                 self.path = path
52.                 # 当前线程的下载位置
53.                 self.start_pos = start_pos
54.                 # 定义当前线程负责下载的文件大小
```

```
55.                 self.current_part_size = current_part_size
56.                 # 当前线程需要下载的文件块
57.                 self.current_part = current_part
58.                 # 定义该线程已下载的字节数
59.                 self.length = 0
60.             def run(self):
61.                 # 创建 Request 对象
62.                 req = Request(url = self.path, method='GET')
63.                 # 添加请求头
64.                 req.add_header('Accept', '*/*')
65.                 req.add_header('Charset', 'UTF-8')
66.                 req.add_header('Connection', 'Keep-Alive')
67.                 # 打开要下载的资源
68.                 f = urlopen(req)
69.                 # 跳过 self.start_pos 个字节，表明该线程只下载自己负责的那部分内
    容
70.                 for i in range(self.start_pos):
71.                     f.read(1)
72.                 # 读取网络数据，并写入本地文件
73.                 while self.length < self.current_part_size:
74.                     data = f.read(1024)
75.                     if data is None or len(data) <= 0:
76.                         break
77.                     self.current_part.write(data)
78.                     # 累计该线程下载的总大小
79.                     self.length += len(data)
80.                 self.current_part.close()
81.                 f.close()
```

程序中定义了 DownThread 线程类，该线程类负责读取从 start_pos 开始、长度为 current_part_size 的所有字节数据，并写入本地文件对象中。DownThread 线程类的 run() 方法就是一个简单的输入/输出实现。

程序中 DownUtils 类的 download() 方法负责按如下步骤来实现多线程下载：

1. 使用 urlopen() 方法打开远程资源。
2. 获取指定的 URL 对象所指向资源的大小（通过 Content-Length 响应头获取）。
3. 计算每个线程应该下载网络资源的哪个部分（从哪个字节开始，到哪个字节结束）。
4. 依次创建并启动多个线程来下载网络资源的指定部分。

上面程序已经实现了多线程下载的核心代码，如果要实现断点下载，则需要额外增加一个配直文件（读者可以发现，所有的断点下载工具都会在下载开始时生成两个文件：一个是与网络资源具有相同大小的空文件；一个是配置文件），该配置文件分别记录每个线程已经下载到哪个字节，当网络断开后再次开始下载时，每个线程根据配置文件中记录的位置向后下载即可。

有了上面的 DownUtil 工具类之后，接下来就可以在主程序中调用该工具类的 download() 方法执行下载，如下面的程序所示：

```
from DownUtil import *

du = DownUtil("http://www.crazyit.org/data/attachment/" \
    + "forum/201801/19/121212ituj1s9gj8g880jr.png", 'a.png', 3)
du.download()
def show_process():
    print("已完成：%.2f" % du.get_complete_rate())
    global t
    if du.get_complete_rate() < 1:
        # 通过定时器启动 0.1 之后执行 show_process 函数
```

```
                t = threading.Timer(0.1, show_process)
                t.start()
        # 通过定时器启动 0.1 之后执行 show_process 函数
        t = threading.Timer(0.1, show_process)
        t.start()
```

运行上面程序，即可看到程序从 www.crazyit.org 下载得到一个名为 a.png 的图片文件。

5.Python http.cookiejar 模块：管理 cookie

通过前面章节的学习，使用 urlopen() 既可发送 GET 请求，也可发送 POST、PUT、DELETE、PATCH 等请求。因此，在绝大部分时候，完全可以使用 urllib.request 模块代替 http.client 模块。

有时候，用户可能需要访问 Web 应用中的被保护页面，如果使用浏览器则十分简单，通过系统提供的登录页面登录系统，浏览器会负责维护与服务器之间的 session，如果用户登录的用户名、密码符合要求，就可以访问被保护资源了。

如果使用 urllib.request 模块来访问被保护页面，则同样需要维护与服务器之间的 session，此时就需要借助于 cookie 管理器来实现。

HTTP 是一种“请求-响应”式协议：客户端向服务器发送请求，服务器向客户揣生成响应数据。这就涉及一个问题，服务器如何辨别两次请求的客户端是同一个客户端呢？答案是 session id。当客户端第一次向服务器发送请求时，服务器会为该客户端分配一个 session id 作为其标识，服务器在生成响应数据时，也会把该 session id 作为响应数据发送给客户端。当客户端第二次向服务器发送请求时，如果客户端把自己的 session id 也发送给服

> 务器，且服务器端的 session id 还未过期，服务器就知道该客户端与前一次发送请求的客户端是同一个。

如果程序直接使用 urlopen() 发送请求，并且并未管理与服务器之间的 session，那么服务器就无法识别两次请求是否是同一个客户端发出的。为了有效地管理 session，程序可引入 http.cookiejar 模块。

此外，程序还需要使用 OpenerDirector 对象来发送请求。为了使用 urllib.request 模块通过 cookie 来管理 session，可按如下步骤进行操作：

1. 创建 http.cookiejar.CookieJar 对象或其子类的对象。
2. 以 CookieJar 对象为参数，创建 urllib.request.HTTPCookieProcessor 对象，该对象负责调用 CookieJar 来管理 cookie。
3. 以 HTTPCookieProcessor 对象为参数，调用 urllib.request.build_opener() 函数创建 OpenerDirector 对象。
4. 使用 OpenerDirector 对象来发送请求，该对象将会通过 HTTPCookieProcessor 调用 CookieJar 来管理 cookie。

下面程序示范了先登录 Web 应用，然后访问 Web 应用中的被保护页面：

```
from urllib.request import *
import http.cookiejar, urllib.parse

# 以指定文件创建 CookieJar 对象，对象将可以把 cookie 保存在文件中
cookie_jar = http.cookiejar.MozillaCookieJar('a.txt')
```

```
6.          # 创建 HTTPCookieProcessor 对象
7.          cookie_processor = HTTPCookieProcessor(cookie_jar)
8.          # 创建 OpenerDirector 对象
9.          opener = build_opener(cookie_processor)
10.
11.         # 定义模拟 Chrome 浏览器的 user_agent
12.         user_agent = r'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36' \
13.             r' (KHTML, like Gecko) Chrome/56.0.2924.87 Safari/537.36'
14.         # 定义请求头
15.         headers = {'User-Agent':user_agent, 'Connection':'keep-alive'}
16.
17.         #-------------下面代码发送登录的 POST 请求----------------
18.         # 定义登录系统的请求参数
19.         params = {'name':'crazyit.org', 'pass':'leegang'}
20.         postdata = urllib.parse.urlencode(params).encode()
21.         # 创建向登录页面发送 POST 请求的 Request
22.         request = Request('http://localhost:8888/test/login.jsp',
23.             data = postdata, headers = headers)
24.         # 使用 OpenerDirector 发送 POST 请求
25.         response = opener.open(request)
26.         print(response.read().decode('utf-8'))
27.
28.         # 将 cookie 信息写入磁盘文件
29.         cookie_jar.save(ignore_discard=True, ignore_expires=True)  # ①
30.
31.         #-------------下面代码发送访问被保护资源的 GET 请求----------------
32.         # 创建向"受保护页面"发送 GET 请求的 Request
33.         request = Request('http://localhost:8888/test/secret.jsp',
34.             headers=headers)
35.         response = opener.open(request)
36.         print(response.read().decode())
```

上面程序中，第 5 行代码先创建了一个 CookieJar 对象，此处使用它的子类 MozillaCookieJar，该对象负责把 cookie 信息保存在文件中。第 7 行代码负责创建 HTTPCookieProcessor 对象；第 9 行代码调用 build_opener() 函数创建

OpenerDirector 对象。接下来程序就会通过该对象来发送请求，而底层的 CookieJar 对象就负责处理 cookie。

程序发送 POST 请求和 GET 请求的代码与前面章节中的示例代码并没有太大的区别，只不过此处额外设置了一个 User-Agent 请求头，该请求头用于模拟 Chrome 浏览器。

正如从上面代码所看到的，该程序同样需要设置并启动 Tomcat 服务器。运行上面程序，可以看到如下输出结果：

```
恭喜您，登录成功!
<!DOCTYPE html>
<html>
<head>
    <meta name = "author" content = "Yeeku.H.Lee(Crazyit.org)"/>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <title>安全资源</title>
</head>
<body>
    安全资源，只有登录用户<br/>
    且用户名是 crazyit.org 才可访问该资源
</body>
</html>
```

上面第一行输出信息就是发送 POST 登录请求的响应数据，运行响应数据提示“登录成功”。后面的输出信息则显示该程序成功访问了“安全资源”（这份资源要求用户必须登录才能访问），如果没有登录，直接访问是看不到该输出信息的。

如果将上面程序中 ① 号代码的注释取消，程序就会把 cookie 信息写入 a.txt 文件中。这意味着该程序将会把服务器响应的 session id 等 cookie 持久化保存在 a.txt 文件中，后面程序可以读取该 cookie 文件信息，这样程序就可以模拟前面登录过的客户端，从而直接访问被保护页面了。

例如如下程序：

```
from urllib.request import *
import http.cookiejar, urllib.parse

# 以指定文件创建 CookieJar 对象，对象将可以把 cookie 保存在文件中
cookie_jar = http.cookiejar.MozillaCookieJar('a.txt')
# 直接加载 a.txt 中的 Cookie 信息
cookie_jar.load('a.txt',ignore_discard=True,ignore_expires=True)
# 遍历 a.txt 中保存的 cookie 信息
for item in cookie_jar:
    print('Name ='+ item.name)
    print('Value ='+ item.value)
# 创建 HTTPCookieProcessor 对象
cookie_processor = HTTPCookieProcessor(cookie_jar)
# 创建 OpenerDirector 对象
opener = build_opener(cookie_processor)

# 定义模拟 Chrome 浏览器的 user_agent
user_agent = r'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36' \
    r' (KHTML, like Gecko) Chrome/56.0.2924.87 Safari/537.36'
# 定义请求头
headers = {'User-Agent':user_agent, 'Connection':'keep-alive'}

#-------------下面代码发送访问被保护资源的 GET 请求----------------
# 创建向"受保护页面"发送 GET 请求的 Request
```

```
25.         request = Request('http://localhost:8888/test/secret.jsp',
26.             headers=headers)
27.         response = opener.open(request)
28.         print(response.read().decode())
```

该程序的前面部分同样用于创建 CookieJar、HTTPCookieProcessor、OpenerDirector 对象，但它多了第 7 行代码，运行代码用于从 a.txt 文件中读取 cookie 信息。接下来的第 9~11 行代码只是用于显示 a.txt 文件中所保存的 cookie 信息，这些代码不影响程序本身。

该程序并未向服务器发送登录请求，但由于该 CookieJar 会把登录成功的 session id 发送给服务器，因此服务器就会认为该程序与前面那个登录成功的程序是同一个客户端。运行上面程序，也可以访问到 Web 应用中的被保护页面，输出结果如下：

```
Name =JSESSIONID
Value =CA772F2EB65808A43A09A452005C2F55


<!DOCTYPE html>
<html>
<head>
    <meta name="author" content="Yeeku.H.Lee(Crazyit.org)"/>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <title>安全资源</title>
</head>
<body>
    安全资源，只有登录用户<br/>
    且用户名是 crazyit.org 才可访问该资源
</body>
```

```
</html>
```

上面前两行输出内容就是 a.txt 文件中保存的 cookie 信息。从该输出结果可以看到，该 cookie 信息只是保存了服务器发送给客户端的 session id。因此，程序使用 OpenerDirector 向服务器发送请求时，就会将该 session id 发送给服务器，这样服务器就会把程序当成前一个已经登录成功的程序。所以，该程序也可以访问到被保护页面。

6. 什么是 TCP 协议，TCP 协议和 IP 协议的关系

TCP/IP 通信协议是一种可靠的网络协议，它在通信的两端各建立一个 socket，从而形成虚拟的网络链路。一旦建立了虚拟的网络链路，两端的程序就可以通过该链路进行通信了。Python 的 socket 模块为基于 TCP 协议的网络通信提供了良好的封装，Python 使用 socket 对象来代表两端的通信端口，并通过 socket 进行网络通信。

IP 是 Internet 上使用的一个关键协议，它的全称是 Internet Protocol，即 Internet 协议，通常简称 IP 协议。通过使用 IP 协议，使 Internet 成为一个允许连接不同类型的计算机和不同操作系统的网络。

要使两台计算机彼此能进行通信，必须使这两台计算机使用同一种“语言”，IP 协议只保证计算机能发送和接收分组数据。IP 协议负责将消息从一个主机传送到另一个主机，消息在传送的过程中被分割成一个个小包。

尽管通过安装 IP 软件，保证了计算机之间可以发送和接收数据，但 IP 协议还不能解决数据分组在传输过程中可能出现的问题。因此，连接 Internet 的计算机还需要安装 TCP 协议来提供可靠且无差错的通信服务。

TCP 被称作端对端协议，这是因为它在两台计算机的连接中起了重要作用。当一台计算机需要与另一台远程计算机连接时，TCP 协议会让它们之间建立一个虚拟链路，用于发送和接收数据。

TCP 协议负责收集这些数据包，并将其按照适当的顺序传送，接收端接收到数据包后再将其正确地还原。TCP 协议保证数据包在传送过程中准确无误。TCP 协议采用重发机制，即当一个通信实体发送一个消息给另一个通信实体后，需要接收到另一个通信实体的确认信息，如果没有接收到该确认信息，则会重发信息。

通过重发机制，TCP 协议向应用程序提供了可靠的通信连接，使其能够自动适应网络上的各种变化。即使在 Internet 暂时出现堵塞的情况下，TCP 协议也能够保证通信的可靠性。

图 1 显示了 TCP 协议控制两个通信实体互相通信的示意图。

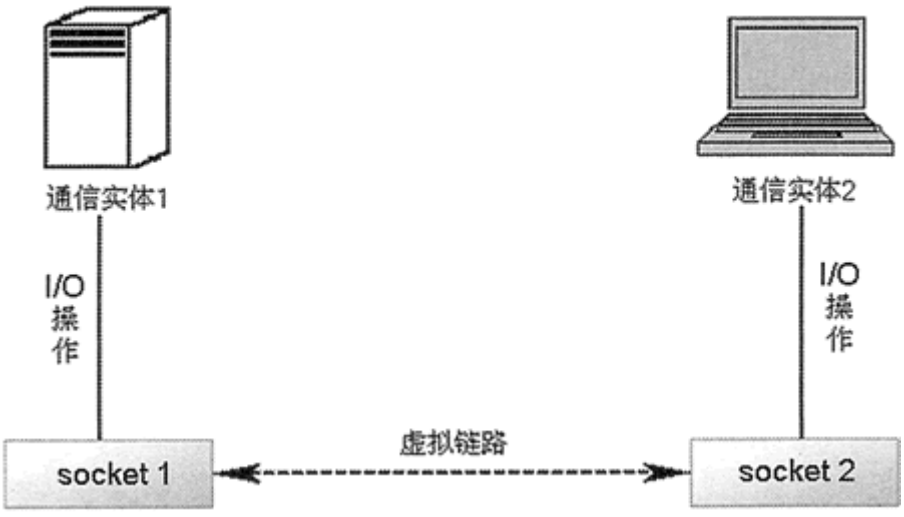


图 1 TCP 协议控制两个通信实体互相通信的示意图

只有把 TCP 和 IP 两个协议结合起来，才能保证 Internet 在复杂的环境下正常运行凡是要连接到 Internet 的计算机，都必须同时安装和使用 TCP/IP 协议。

> 读者想深入了解 TCP 协议，可猛击《socket 编程入门：1 天玩转 socket 通信技术》一章作详细了解。

7.Python socket 建立 TCP 连接

程序在使用 socket 之前，必须先创建 socket 对象，可通过该类的如下构造器来创建 socket 实例：

```
socket.socket(family=AF_INET, type=SOCK_STREAM, proto=0, fileno=None)
```

上面构造器的前三个参数比较重要，其中：

1. family 参数用于指定网络类型。该参数支持 socket.AF_UNIX（UNIX 网络）、socket.AF_INET（基于 IPv4 协议的网络）和 socket.AF_INET6（基于 IPv6 协议的网络）这三个常量。
2. type 参数用于指定网络 Sock 类型。该参数可支持 SOCK_STREAM（默认值，创建基于 TCP 协议的 socket）、SOCK_DGRAM（创建基于 UDP 协议的 socket）和 SOCK_RAW（创建原始 socket）。一般常用的是 SOCK_STREAM 和 SOCK_DGRAM。
3. proto 参数用于指定协议号，如果没有特殊要求，该参数默认为 0 ，并可以忽略。

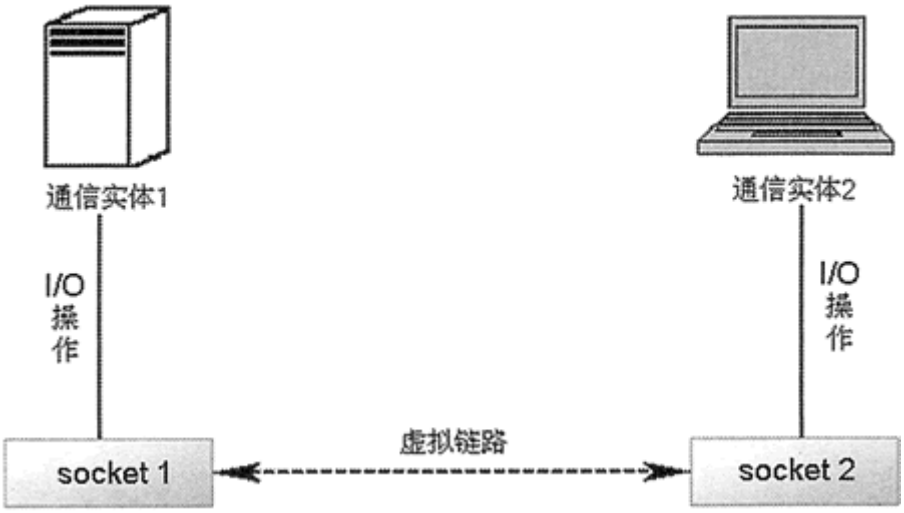


图 1 TCP 协议控制两个通信实体互相通信的示意图

在创建了 socket 之后，接下来需要将两个 socket 连接起来。从图 1 中并没有看出 TCP 协议控制的两个通信实体之间有服务器端和客户端之分，这是因为此图是两个通信实体之间已经建立虚拟链路之后的示意图。在两个通信实体之间没有建立虚拟链路时，必须有一个通信实体先做出“主动姿态”，主动接收来自其他通信实体的连接请求。

作为服务器端使用的 socket 必须被绑定到指定 IP 地址和端口，并在该 IP 地址和端口进行监听，接收来自客户端的连接。

socket 对象提供了如下常用方法：

- socket.accept()：作为服务器端使用的 socket 调用该方法接收来自客户端的连接。
- socket.bind(address)：作为服务器端使用的 socket 调用该方法，将该 socket 绑定到指定 address，该 address 可以是一个元组，包含 IP 地址和端口。

- socket.close()：关闭连接，回收资源。
- socket.connect(address)：作为客户端使用的 socket 调用该方法连接远程服务器。
- socket.connect_ex(address)：该方法与上一个方法的功能大致相同，只是当程序出错时，该方法不会抛出异常，而是返回一个错误标识符。
- socket.listen([backlog])：作为服务器端使用的 socket 调用该方法进行监听。
- socket.makefile(mode='r', buffering=None, *, encoding=None, errors=None, newline=None)：创建一个和该 socket 关联的文件对象。
- socket.recv(bufsize[, flags])：接收 socket 中的数据。该方法返回 bytes 对象代表接收到的数据。
- socket.recvfrom(bufsize[,flags])：该方法与上一个方法的功能大致相同，只是该方法的返回值是 (bytes, address) 元组。
- socket.recvmsg(bufsize[, ancbufsize[, flags]])：该方法不仅接收来自 socket 的数据，还接收来自 socket 的辅助数据，因此该方法的返回值是一个长度为 4 的元组 (data, ancdata, msg_flags, address)，其中 ancdata 代表辅助数据。
- socket.recvmsg_into(buffers[, ancbufsize[, flags]])：类似于 socket.recvmsg() 方法，但该方法将接收到的数据放入 buffers 中。
- socket.recvfrom_into(buffer[, nbytes[, flags]])：类似于 socket.recvfrom() 方法，但该方法将接收到的数据放入 buffer 中。
- socket.recv_into(buffer[, nbytes[, flags]])：类似于 recv() 方法，但该方法将接收到的数据放入 buffer 中。

- socket.send(bytes[, flags]) :向 socket 发送数据 ,该 socket 必须与远程 socket 建立了连接。该方法通常用于在基于 TCP 协议的网络中发送数据。
- socket.sendto(bytes, address)：向 socket 发送数据，该 socket 应该没有与远程 socket 建立连接。该方法通常用于在基于 UDP 协议的网络中发送数据。
- socket.sendfile(file, offset=0, count=None)：将整个文件内容都发送出去，直到遇到文件的 EOF。
- socket.shutdown(how)：关闭连接。其中 how 用于设置关闭方式。

掌握了这些常用的方法之后，可以大致归纳出 TCP 通信的服务器端编程的基本步骤：

1. 服务器端先创建一个 socket 对象。
2. 服务器端 socket 将自己绑定到指定 IP 地址和端口。
3. 服务器端 socket 调用 listen() 方法监听网络。
4. 程序采用循环不断调用 socket 的 accept() 方法接收来自客户端的连接。

代码片段如下：

```
#创建 socket 对象
s = socket.socket()
#将 socket 绑定到本机 IP 地址和端口
s.bind (('192.168.1.88', 30000))
#服务器端开始监听来自客户端的连接
s.listen()
while True:
```

```
8.              #每当接收到客户端 socket 的请求时，该方法就返回对应的 socket 和远程
    地址
9.              c, addr = s.accept()
10.             ...
```

上面程序先创建了一个 socket 对象，接下来将该 socket 绑定到 192.168.1.88 的 30000 端口，其中 192.168.1.88 是程序所在计算机的 IP 地址。

> 提示，上面程序使用 30000 作为该 socket 的监听端口，通常推荐使用 1024 以上的端口，主要是为了避免与其他应用程序的通用端口发生冲突。

同样，客户端也是先创建一个 socket 对象，然后调用 socket 的 connect() 方法建立与服务器端的连接，这样就可以建立一个基于 TCP 协议的网络连接。

TCP 通信的客户端编程的基本步骤大致归纳如下：

1. 客户端先创建一个 socket 对象。
2. 客户端 socket 调用 connect() 方法连接远程服务器。

代码片段如下：

```
1.          #创建 socket 对象
2.          s = socket.socket ()
3.          #连接远程服务器
4.          s.connect({'192.168.1.88', 30000))
5.          #下面就可以使用 socket 进行通信了
6.          ...
```

当执行上面程序中第 4 行代码时，将会连接到指定服务器，让服务器端 socket 的 accept() 方法向下执行，于是服务器端和客户端就产生一对互相连接的 socket。

当服务器端和客户端产生了对应的 socket 之后，就得到了如图 1 所示的通信示意图，程序无须再区分服务器端和客户端，而是通过各自的 socket 进行通信。

通过前面介绍我们知道，socket 提供了大量方法来发送和接收数据：

- 发送数据：使用 send() 方法。注意，sendto() 方法用于 UDP 协议的通信。
- 接收数据：使用 recv_xxx() 方法。

下面的服务器端程序非常简单，它仅仅建立 socket，并监听来自客户端的连接，只要客户端连接进来，程序就会向 socket 发送一条简单的信息：

```
1.          # 导入 socket 模块
2.          import socket
3.
4.          # 创建 socket 对象
5.          s = socket.socket()
6.          # 将 socket 绑定到本机 IP 和端口
7.          s.bind(('192.168.1.88', 30000))
8.          # 服务端开始监听来自客户端的连接
9.          s.listen()
10.         while True:
11.             # 每当接收到客户端 socket 的请求时，该方法返回对应的 socket 和远程地
    址
12.             c, addr = s.accept()
13.             print(c)
```

```
14.             print('连接地址：', addr)
15.             c.send('您好，您收到了服务器的新年祝福！'.encode('utf-8'))
16.             # 关闭连接
17.             c.close()
```

下面的客户端程序也非常简单，它仅仅使用 socket 建立与指定 IP 地址和端口的连接，并从 socket 中获取服务器端发送的数据：

```
1.          # 导入 socket 模块
2.          import socket
3.
4.          # 创建 socket 对象
5.          s = socket.socket()
6.          # 连接远程主机
7.          s.connect(('192.168.1.88', 30000))     # ①
8.          print('--%s--' % s.recv(1024).decode('utf-8'))
9.          s.close()
```

上面程序中 ① 号代码使用 socket 建立与服务器端的连接，接下来调用 socket 的 recv() 方法来接收网络数据。

先运行服务器端程序，将看到服务器一直处于等待状态，因为服务器使用了死循环来接收来自客户端的请求；再运行客户端程序，将看到程序输出："--您好，您收到了服务器的新年祝福！--"，这表明客户端和服务器端通信成功。

8.Python 多线程实现 socket 通信

前面章节对服务器端和客户端只是进行了简单的通信操作，即服务器端接受客户端的连接之后，向客户端输出一个字符串，而客户端也只是读取服务器端的字符串后就退出了。

在实际应用中，客户端则可能需要和服务器端保持长时间通信，即服务器端需要不断地读取客户端数据，并向客户端写入数据；客户端也需要不断地读取服务器端数据，并向服务器端写入数据。

由于 socket 的 recv() 方法在成功读取到数据之前，线程会被阻塞，程序无法继续执行。考虑到这个原因，服务器端应该为每个 socket 都单独启动一个线程，每个线程负责与一个客户端进行通信。

客户端读取服务器端数据的线程同样会被阻塞，所以系统应该单独启动一个线程，该线程专门负责读取服务器端数据。

现在考虑实现一个命令行界面的 C/S 聊天室应用，服务器端应该包含多个线程，每个 socket 对应一个线程，该线程负责从 socket 中读取数据（从客户端发送过来的数据），并将所读取到的数据向每个 socket 发送一次（将一个客户端发送过来的数据“广播”给其他客户端），因此需要在服务器端使用 list 来保存所有的 socket。

下面是服务器端的实现代码。该服务器端代码定义了一个 server_target() 函数，该函数将会作为线程执行的 target，负责处理每个 socket 的数据通信。

```
import socket
import threading

# 定义保存所有 socket 的列表
socket_list = []
# 创建 socket 对象
```

```
ss = socket.socket()
# 将 socket 绑定到本机 IP 和端口
ss.bind(('192.168.1.88', 30000))
# 服务端开始监听来自客户端的连接
ss.listen()
def read_from_client(s):
    try:
        return s.recv(2048).decode('utf-8')
    # 如果捕获到异常，则表明该 socket 对应的客户端已经关闭
    except:
        # 删除该 socket
        socket_list.remove(s);      # ①
def server_target(s):
    try:
        # 采用循环不断地从 socket 中读取客户端发送过来的数据
        while True:
            content = read_from_client(s)
            print(content)
            if content is None:
                break
            for client_s in socket_list:
                client_s.send(content.encode('utf-8'))
    except e:
        print(e.strerror)
while True:
    # 此行代码会阻塞，将一直等待别人的连接
    s, addr = ss.accept()
    socket_list.append(s)
    # 每当客户端连接后启动一个线程为该客户端服务
    threading.Thread(target=server_target, args=(s, )).start()
```

上面实现的服务器端主程序只负责接收客户端 socket 的连接请求，每当客户端 socket 连接进来之后，程序都将对应的 socket 加入 socket_list 列表中保存，并为该 socket 启动一个线程，该线程负责处理该 socket 所有的通信任务，如程序中最后 3 行代码所示。

代表服务器端线程执行体的 server_target() 函数则不断地读取客户端数据。程序使用 read_from_client() 函数来读取客户端数据，如果在读取数据过程中捕获到异常，则表明该 socket 对应的客户端 socket 出现了问题（到底什么问题不用深究，反正不正常），程序就将该 socket 从 socket_list 列表中删除，如 read_from_client() 函数中的 ① 号代码所示。

当服务器端线程读取到客户端数据之后，程序遍历 socket_list 列表，并将该数据向 socket_list 列表中的每个 socket 发送一次（该服务器端线程把从 socket 中读取到的数据向 socket_list 列表中的每个 socket 转发一次），如 server_target() 函数中的代码所示。

每个客户端都应该包含两个线程，其中一个负责读取用户的键盘输入内容，并将用户输入的数据输出到 socket 中，另一个负责读取 socket 中的数据（从服务器端发送过来的数据），并将这些数据打印输出。由程序的主线程负责读取用户的键盘输入内容，由新线程负责读取 socket 数据。

```
1.          import socket
2.          import threading
3.
4.          # 创建 socket 对象
5.          s = socket.socket()
6.          # 连接远程主机
7.          s.connect(('192.168.1.88', 30000))
8.          def read_from_server(s):
9.              while True:
```

```
                print(s.recv(2048).decode('utf-8'))
        # 客户端启动线程不断地读取来自服务器的数据
        threading.Thread(target=read_from_server, args=(s, )).start()    # ①
        while True:
            line = input('')
            if line is None or line == 'exit':
                break
            # 将用户的键盘输入内容写入 socket
            s.send(line.encode('utf-8'))
```

上面程序中的主线程读取到用户的键盘输入内容后，将该内容发送到 socket 中（实际上就是把数据发送给服务器端）。

此外，当主线程的 socket 连接到服务器端之后，以 read_from_server() 函数为 target 启动了新线程来处理 socket 通信，如程序中 ① 号代码所示。read_from_server() 函数使用死循环读取 socket 中的数据（就是来自服务器端的数据），并将这些内容在控制台打印出来，如 read_from_server() 函数中的代码所示。

先运行上面的 MyServer 程序，该程序运行后只是作为服务器，看不到任何输出信息。再运行多个 MyClient 程序（相当于启动多个聊天室客户端登录该服务器），接下来可以在任何一个客户端通过键盘输入一些内容，然后按回车键，即可在所有客户端（包括自己）的控制台中收到刚刚输入的内容，这就粗略地实现了一个 C/S 结构的聊天室应用。

9.Python socket shutdown 方法详解

前面在介绍服务器端和客户端通信时，总是以一个 bytes 对象作为通信的最小数据单位的，服务器端在处理信息时也是针对每个 bytes 进行的。在一些协议中，通信的数据单位

可能需要多个 bytes 对象，在这种情况下，就需要解决一个问题，socket 如何表示输出数据已经结束？

如果要表示输出数据已经结束，则可以通过关闭 socket 来实现。但如果彻底关闭了 socket，则会导致程序无法再从该 socket 中读取数据。

在这种情况下，socket 提供了一个 shutdown(how) 关闭方法，该方法可以只关闭 socket 的输入或输出部分，用以表示输出数据已经发送完成。

shutdown 方法的 how 参数接受如下参数值：

- SHUT_RD：关闭 socket 的输入部分，程序还可通过该 socket 输出数据。
- SHUT_WR： 关闭该 socket 的输出部分，程序还可通过该 socket 读取数据。
- SHUT_RDWR：全关闭。该 socket 既不能读取数据，也不能写入数据。

当调用 shutdown() 方法关闭 socket 的输入或输出部分之后，该 socket 处于“半关闭”状态。

> 即使一个 socket 实例在调用 shutdown() 方法时专入了 SHUT_RDWR 参数，该 socket 也依然没有被彻底清理（与 close() 方法不同），只是该 socket 既不能输出数据，也不能读取数据了。

下面程序示范了 shutdown() 方法的用法。在该程序中服务器端先向客户端发送多条数据，

当数据发送完成后，该 socket 对象调用 shutdown() 方法来关闭输出部分，表明数据发送结束在关闭输出部分之后，依然可以从 socket 中读取数据。

```
1.          import socket
2.
3.          # 创建 socket 对象
4.          s = socket.socket()
5.          # 将 socket 绑定到本机 IP 和端口
6.          s.bind(('192.168.1.88', 30000))
7.          # 服务端开始监听来自客户端的连接
8.          s.listen()
9.          # 每当接收到客户端 socket 的请求时，该方法返回对应的 socket 和远程地址
10.         skt, addr = s.accept()
11.         skt.send("服务器的第一行数据".encode('utf-8'))
12.         skt.send("服务器的第二行数据".encode('utf-8'))
13.         # 关闭 socket 的输出，表明输出数据已经结束
14.         skt.shutdown(socket.SHUT_WR)
15.         while True:
16.             # 从 socket 读取数据
17.             line = skt.recv(2048).decode('utf-8')
18.             if line is None or line == '':
19.                 break
20.             print(line)
21.         skt.close()
22.         s.close()
```

上面程序中，第 14 行代码关闭了 socket 的输出部分，此时该 socket 并未被彻底关闭，程序只是不能向该 socket 中写入数据了，但依然可以从该 socket 中读取数据。

本程序的客户端代码比较简单，它先读取服务器端返回的数据，然后向服务器端输出一些内容，故此处不再赘述。

当调用 socket 的 shutdown() 方法关闭了输入或输出部分之后，该 socket 无法再次打开输

入或输出部分，因此这种做法通常不适合保持持久通信状态的交互式应用，只适用于一站式的通信协议，例如 HTTP 协议，即客户端连接到服务器端，开始发送请求数据，当发送完成后无须再次发送数据，只需要读取服务器端的响应数据即可，当读取响应数据完成后，该 socket 连接就被完全关闭了。

10.Python selectors 模块用法：实现非阻塞式编程

前面介绍的 socket 都是采用阻塞方式进行通信的，当程序调用 recv() 方法从 socket 中读取数据时，如果没有读取到有效的数据，当前线程就会被阻塞。为了解决这个问题，上面程序采用了多线程并发编程，即服务器端为每个客户端连接都启动一个单独的线程，不同的线程负责对应的 socket 的通信工作。

通过 selectors 模块允许 socket 以非阻塞方式进行通信，selectors 相当于一个事件注册中心，程序只要将 socket 的所有事件注册给 selectors 管理，当 selectors 检测到 socket 中的特定事件之后，程序就调用相应的监听方法进行处理。

selectors 主要支持两种事件：

- selectors.EVENT_READ：当 socket 有数据可读时触发该事件。当有客户端连接进来时也会触发该事件。
- selectors.EVENT_WRITE：当 socket 将要写数据时触发该事件。

使用 selectors 实现非阻塞式编程的步骤大致如下：

1. 创建 selectors 对象。

2. 通过 selectors 对象为 socket 的 selectors.EVENT_READ 或 selectors.EVENT_WRITE 事件注册监听器函数。每当 socket 有数据需要读写时，系统负责触发所注册的监听器函数。

3. 在监听器函数中处理 socket 通信。

下面程序使用 selectors 模块实现非阻塞式通信的服务器端：

```
import selectors, socket

# 创建默认的 selectors 对象
sel = selectors.DefaultSelector()
# 负责监听“有数据可读”事件的函数
def read(skt, mask):
    try:
        # 读取数据
        data = skt.recv(1024)
        if data:
            # 将读取的数据采用循环向每个 socket 发送一次
            for s in socket_list:
                s.send(data)  # Hope it won't block
        else:
            # 如果该 socket 已被对方关闭，关闭该 socket，
            # 并从 socket_list 列表中删除
            print('关闭', skt)
            sel.unregister(skt)
            skt.close()
            socket_list.remove(skt)
    # 如果捕捉到异常，将该 socket 关闭，并从 socket_list 列表中删除
    except:
        print('关闭', skt)
        sel.unregister(skt)
        skt.close()
```

```
                socket_list.remove(skt)
        socket_list = []
        # 负责监听“客户端连接进来”事件的函数
        def accept(sock, mask):
            conn, addr = sock.accept()
            # 使用 socket_list 保存代表客户端的 socket
            socket_list.append(conn)
            conn.setblocking(False)
            # 使用 sel 为 conn 的 EVENT_READ 事件注册 read 监听函数
            sel.register(conn, selectors.EVENT_READ, read)     #②
        sock = socket.socket()
        sock.bind(('192.168.1.88', 30000))
        sock.listen()
        # 设置该 socket 是非阻塞的
        sock.setblocking(False)
        # 使用 sel 为 sock 的 EVENT_READ 事件注册 accept 监听函数
        sel.register(sock, selectors.EVENT_READ, accept)     #①
        # 采用死循环不断提取 sel 的事件
        while True:
            events = sel.select()
            for key, mask in events:
                # key 的 data 属性获取为该事件注册的监听函数
                callback = key.data
                # 调用监听函数, key 的 fileobj 属性获取被监听的 socket 对象
                callback(key.fileobj, mask)
```

上面程序中定义了两个监听器函数 accept() 和 read()，其中 accept() 函数作为“有客户端连接进来”事件的监听函数，主程序中的 ① 号代码负责为 socket 的 selectors.EVENT_READ 事件注册该函数；read() 函数则作为“有数据可读”事件的监听函数，如 accept() 函数中的 ② 号代码所示。

通过上面这种方式，程序避免了采用死循环不断地调用 socket 的 accept() 方法来接受客

户端连接，也避免了采用死循环不断地调用 socket 的 recv() 方法来接收数据。socket 的 accept()、recv() 方法调用都是写在事件监听函数中的，只有当事件（如“有客户端连接进来”事件、“有数据可读”事件）发生时，accept() 和 recv() 方法才会被调用，这样就避免了阻塞式编程。

为了不断地提取 selectors 中的事件，程序最后使用一个死循环不断地调用 selectors 的 select() 方法“监测”事件，每当监测到相应的事件之后，程序就会调用对应的事件监听函数。

下面是该示例的客户端程序。该客户端程序更加简单，客户端程序只需要读取 socket 中的数据，因此只要使用 selectors 为 socket 注册“有数据可读”事件的监听函数即可。

```
import selectors, socket, threading

# 创建默认的 selectors 对象
sel = selectors.DefaultSelector()
# 负责监听“有数据可读”事件的函数
def read(conn, mask):
    data = conn.recv(1024)  # Should be ready
    if data:
        print(data.decode('utf-8'))
    else:
        print('closing', conn)
        sel.unregister(conn)
        conn.close()
# 创建 socket 对象
s = socket.socket()
# 连接远程主机
s.connect(('192.168.1.88', 30000))
```

```
        # 设置该 socket 是非阻塞的
        s.setblocking(False)
        # 使用 sel 为 s 的 EVENT_READ 事件注册 read 监听函数
        sel.register(s, selectors.EVENT_READ, read)     # ①
        # 定义不断读取用户键盘输入的函数
        def keyboard_input(s):
            while True:
                line = input('')
                if line is None or line == 'exit':
                    break
                # 将用户的键盘输入内容写入 socket
                s.send(line.encode('utf-8'))
        # 采用线程不断读取用户的键盘输入
        threading.Thread(target=keyboard_input, args=(s, )).start()
        while True:
            # 获取事件
            events = sel.select()
            for key, mask in events:
                # key 的 data 属性获取为该事件注册的监听函数
                callback = key.data
                # 调用监听函数, key 的 fileobj 属性获取被监听的 socket 对象
                callback(key.fileobj, mask)
```

上面程序中的 ① 号代码为 socket 的 EVENT_READ 事件注册了 read() 监听函数，这样每当 socket 中有数据可读时，程序就会触发 read() 函数来读取 socket 中的数据。

程序最后也采用死循环不断地调用 selectors 的 select() 方法“监测”事件，每当监测到相应的事件之后，程序就会调用对应的事件监听函数。

先运行上面的服务器端程序，该程序运行后只是作为服务器，看不到任何输出信息。再运行多个客户端程序（相当于启动多个聊天室客户端登录该服务器）。接下来可以在任何一个客户端通过键盘输入一些内容，然后按回车键，即可在所有客户端（包括自己）的控制台上接收到刚刚输入的内容。这也是一个粗略的 C/S 结构的聊天室应用。

11.UDP 是什么，UDP 协议及优缺点（包含和 TCP 协议的区别）

UDP，全称 User Datagram Protocol，中文名称为用户数据报协议，主要用来支持那些需要在计算机之间传输数据的网络连接。

UDP 协议从问世至今已经被使用了很多年，虽然目前 UDP 协议的应用不如 TCP 协议广泛，但 UDP 依然是一种非常实用和可行的网络传输层协议。尤其是在一些实时性很强的应用场景中，比如网络游戏、视频会议等，UDP 协议的快速能力更具有独特的魅力。

UDP 是一种面向非连接的协议，面向非连接指的是在正式通信前不必与对方先建立连接，不管对方状态就直接发送数据。至于对方是否可以接收到这些数据，UDP 协议无法控制，所以说 UDP 是一种不可靠的协议。

> UDP 协议适用于一次只传送少量数据、对可靠性要求不高的应用环境。

与前面介绍的 TCP 协议一样，UDP 协议直接位于 IP 协议之上。实际上，IP 协议属于 OSI 参考模型的网络层协议，而 UDP 协议和 TCP 协议都属于传输层协议。

因为 UDP 是面向非连接的协议，没有建立连接的过程，因此它的通信效率很高，但也正因为如此，它的可靠性不如 TCP 协议。

UDP 协议的主要作用是完成网络数据流和数据报之间的转换在信息的发送端，UDP 协议

将网络数据流封装成数据报，然后将数据报发送出去；在信息的接收端，UDP 协议将数据报转换成实际数据内容。

> 可以认为 UDP 协议的 socket 类似于码头，数据报则类似于集装箱。码头的作用就是负责友送、接收集装箱，而 socket 的作用则是发送、接收数据报。因此，对于基于 UDP 协议的通信双方而言，没有所谓的客户端和服务器端的概念。

UDP 协议和 TCP 协议简单对比如下：

- TCP 协议：可靠，传输大小无限制，但是需要连接建立时间，差错控制开销大。
- UDP 协议：不可靠，差错控制开销较小，传输大小限制在 64 KB 以下，不需要建立连接。

相比较 TCP，UDP 是一种不可靠的网络协议，它在通信实例的两端各建立一个 socket，但这两个 socket 之间并没有虚拟链路，它们只是发送、接收数据报的对象。

12.socket 发送和接受数据（基于 UDP 协议）

程序在创建 socket 时，可通过 type 参数指定该 socket 的类型，如果将该参数指定为 SOCK_DGRAM，则意味着创建基于 UDP 协议的 socket。

在创建了基于 UDP 协议的 socket 之后，程序可以通过如下两个方法来发送和接收数据：

- socket.sendto(bytes, address)：将 bytes 数据发送到 address 地址。

- socket.recvfrom(bufsize[, flags])：接收数据。该方法可以同时返回 socket 中的数据和数据来源地址。

从这两个方法的介绍可以看出，使用 UDP 协议的 socket 在发送数据时必须使用 sendto() 方法，这是因为程序必须指定发送数据的目标地址（通过 address 参数指定）；使用 UDP 协议的 socket 在接收数据时，既可使用普通的 recv() 方法，也可使用 recvfrom() 方法。如果程序需要得到数据报的来源，则应该使用 recvfrom() 方法。

从上面的介绍不难看出，由于 UDP 协议没有建立虚拟链路，因此程序使用 socket 发送数据报时，scoket 并不知道将该数据报发送到哪里，必须通过 sendto() 方法的 address 参数来指定数据报的目的地，这个目的地的地址会被附加在所发送的数据报上。

> 就像码头并不知道每个集装箱的目的地一样，码头只是将这些集装箱发送出去，而集装箱本身包含了该集装箱的目的地。

程序在使用 UDP 协议进行网络通信时，实际上并没有明显的服务器端和客户端，因为双方都需要先建立一个 socket 对象，用来接收或发送数据报。但在实际编程中，通常具有固定 IP 地址和端口的 socket 对象所在的程序被称为服务器，因此该 socket 应该调用 bind() 方法被绑定到指定 IP 地址和端口，这样其他 socket（客户端 socket）才可向服务器端 socket（绑定了固定 IP 地址和端口的 socket）发送数据报，而服务器端 socket 就可以接收这些客户端数据报。

当服务器端（也可以是客户端）接收到一个数据报后，如果想向该数据报的发送者“反馈”一些信息，此时就必须获取数据报的“来源信息”，这就到了 recvfrom() 方法“闪亮登场”的时候，该方法不仅可以获取 socket 中的数据，也可以获取数据的来源地址，程序就可以通过该来源地址来“反馈”信息。

一般来说，服务器端 socket 的 IP 地址和端口应该是固定的，因此客户端程序可以直接向服务器端 socket 发送数据，但服务器端无法预先知道各客户端 socket 的 IP 地址和端口，因此必须调用 recvfrom() 方法来获取客户端 socket 的 IP 地址和端口。

下面程序使用 UDP 协议的 socket 实现了 C/S 结构的网络通信。本程序的服务器端通过循环 1000 次来读取 socket 中的数据报，每当读取到内容之后，便向该数据报的发送者发送一条信息。服务器端程序的代码如下：

```
import socket

PORT = 30000;
# 定义每个数据报的大小最大为 4KB
DATA_LEN = 4096;
# 定义一个字符串数组，服务器端发送该数组的元素
books = ("疯狂 Python 讲义",
    "疯狂 Kotlin 讲义",
    "疯狂 Android 讲义",
    "疯狂 Swift 讲义")
# 通过 type 属性指定创建基于 UDP 协议的 socket
s = socket.socket(type=socket.SOCK_DGRAM)
# 将该 socket 绑定到本机的指定 IP 和端口
s.bind(('192.168.1.88', PORT))
# 采用循环接收数据
for i in range(1000):
```

```
            # 读取 s 中的数据的数据的发送地址
            data, addr = s.recvfrom(DATA_LEN)
            # 将接收到的内容转换成字符串后输出
            print(data.decode('utf-8'))
            # 从字符串数组中取出一个元素作为发送数据
            send_data = books[i % 4].encode('utf-8')
            # 将数据报发送给 addr 地址
            s.sendto(send_data, addr)
        s.close()
```

上面程序中的代码就是使用 UDP 协议的 socket 发送和接收数据报的关键代码，该程序可以接收 1000 个客户端发送过来的数据。

客户端程序的代码与此类似，客户端采用循环不断地读取用户的键盘输入内容，每当读取到用户输入的内容后，就将该内容通过数据报发送出去；接下来再读取来自 socket 中的信息（也就是来自服务器端的数据）。客户端程序的代码如下：

```
import socket

PORT = 30000;
# 定义每个数据报的大小最大为 4KB
DATA_LEN = 4096;
DEST_IP = "192.168.1.88";
# 通过 type 属性指定创建基于 UDP 协议的 socket
s = socket.socket(type=socket.SOCK_DGRAM)
# 不断地读取键盘输入
while True:
    line = input('')
    if line is None or line == 'exit':
        break
    data = line.encode('utf-8')
    # 发送数据报
    s.sendto(data, (DEST_IP, PORT))
```

```
17.             # 读取 socket 中的数据
18.             data = s.recv(DATA_LEN)
19.             print(data.decode('utf-8'))
20.         s.close()
```

上面程序中的代码就是使用 UDP 协议的 socket 发送和接收数据报的关键代码，这些代码与服务器端程序的代码基本相似。而客户端与服务器端的唯一区别在于，服务器端的 IP 地址和端口是固定的，所以客户端可以直接将数据报发送给服务器端；而服务器端则需要根据所接收到的数据报来决定“反馈”数据报的目的地。

读者可能会发现，在使用 UDP 协议进行网络通信时，服务器端无须也无法保存每个客户端的状态，客户端把数据报发送到服务器端后，完全有可能立即退出。但不管客户端是否退出，服务器端都无法知道客户端的状态。

当使用 UDP 协议时，如果想让一个客户端发送的聊天信息被转发到其他所有的客户端也是可以的，程序可以考虑在服务器端使用 list 列表来保存所有的客户端信息，每当接收到一个客户端的数据报之后，程序检查该数据报的来源地址是否在 list 列表中，如果不在就将该来源地址添加到 list 列表中。这样又涉及一个问题，可能有些客户端发送一个数据报之后，永久地退出了程序，但服务器端依然会将该客户端的 IP 地址和端口保存在 list 列表中。

因此，程序还需要定义一个定时器，定期检查每个客户端有多长时间没向服务器端发送数据了，如果超过一定的时间（比如 10 分钟）该客户端还没有发送数据，则服务器端就将该客户端的 IP 地址和端口从 list 列表中删除……总之，这种方式需要处理的问题比较多，编

程比较烦琐。幸好 UDP 协议还支持多点广播，Python 也为 UDP 协议的多点广播提供了支持。

13.UDP 多点广播

通过多点广播，可以将数据报以广播方式式发送到多个客户端。

若要使用多点广播，则需要将数据报发送到一个组目标地址，当数据报发出后，整个组的所有主机都能接收到该数据报。IP 多点广播（或多点发送）实现了将单一信息发送给多个接收者的广播，其思想是设置一组特殊的网络地址作为多点广播地址，每一个多点广播地址都被看作一个组，当客户端需要发送和接收广播信息时，加入该组即可。

IP 协议为多点广播提供了特殊的 IP 地址，这些 IP 地址的范围是 224.0.0.0~239.255.255.255。多点广播示意图如图 1 所示。

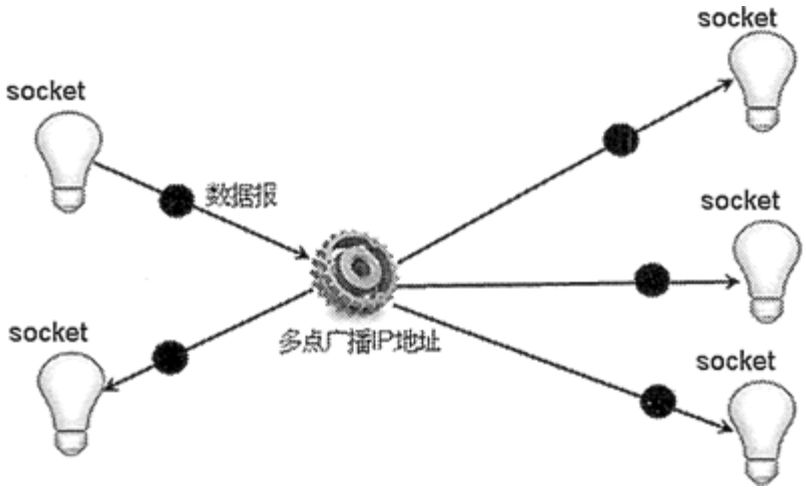


图 1 多点广播示意图

从图 1 中可以看出，当 socket 把一个数据报发送到多点广播 IP 地址时，该数据报将被自动广播到加入该地址的所有 socket。该 socket 既可以将数据报发送到多点广播地址，也可以接收其他主机的广播信息。

在创建了 socket 对象后，还需要将该 socket 加入指定的多点广播地址中，socket 使用 setsockopt() 方法加入指定组。

如果创建仅用于发送数据报的 socket 对象，则使用默认地址、随机端口即可。但如果创建接收数据报的 socket 对象，则需要将该 socket 对象绑定到指定端口；否则，发送方无法确定发送数据报的目标端口。

支持多点广播的 socket 还可设置广播信息的 TTL（Time-To-Live），该 TTL 参数用于设置数据报最多可以跨过多少个网络：

- 当 TTL 的值为 0 时，指定数据报应停留在本地主机中；
- 当 TTL 的值为 1 时，指定将数据报发送到本地局域网中；
- 当 TTL 的值为 32 时，意味着只能将数据报发送到本站点的网络上；
- 当 TTL 的值为 64 时，意味着数据报应被保留在本地区；
- 当 TTL 的值为 128 时，意味着数据报应被保留在本大洲；
- 当 TTL 的值为 255 时，意味着数据报可被发送到所有地方；

> 在默认情况下，TTL 的值为 1。

从图 1 中可以看出，使用 socket 进行多点广播时所有的通信实体都是平等的，它们都将自己的数据报发送到多点广播 IP 地址，并使用 socket 接收其他人发送的广播数据报。

下面程序使用 socket 实现了一个基于广播的多人聊天室。程序只需要一个 socket、两个线程，其中 socket 既用于发送数据，也用于接收数据；主线程负责读取用户的键盘输入内容，并向 socket 发送数据，子线程则负责从 socket 中读取数据。

```
import time, socket, threading, os

# 定义本机 IP 地址
SENDERIP = '192.168.1.88'
# 定义本地端口
SENDERPORT = 30000
# 定义本程序的多点广播 IP 地址
MYGROUP = '230.0.0.1'
# 通过 type 属性指定创建基于 UDP 协议的 socket
s = socket.socket(type=socket.SOCK_DGRAM)
# 将该 socket 绑定到 0.0.0.0 的虚拟 IP
s.bind(('0.0.0.0', SENDERPORT))      # ①
# 设置广播消息的 TTL（Time-To-Live）
s.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, 64)
# 设置允许多点广播使用相同的端口
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
# 将 socket 进入广播组
status = s.setsockopt(socket.IPPROTO_IP,
    socket.IP_ADD_MEMBERSHIP,
    socket.inet_aton(MYGROUP) + socket.inet_aton(SENDERIP))
# 定义从 socket 读取数据的方法
def read_socket(sock):
    while True:
        data = sock.recv(2048)
        print("信息: ", data.decode('utf-8'))
```

```
        # 以 read_socket 作为 target 启动多线程
        threading.Thread(target=read_socket, args=(s, )).start()
        # 采用循环不断读取键盘输入，并输出到 socket 中
        while True:
            line = input('')
            if line is None or line == 'exit':
                break
                os._exit(0)
            # 将 line 输出到 socket 中
            s.sendto(line.encode('utf-8'), (MYGROUP, SENDERPORT))
```

上面主程序中，第 10 行代码先创建了一个基于 UDP 协议的 socket 对象，由于需要使用该 socket 对象接收数据报，所以将该 socket 绑定到固定端口（由于只需要绑定到固定端口，因此程序中 ① 号代码使用了 0.0.0.0 这个虚拟 IP 地址）。

第 18 行代码将该 socket 对象添加到指定的多点广播 IP 地址。至于程序中使用 socket 发送和接收数据报的代码，与前面的程序并没有区别，故此处不再赘述。

14.Python smtplib 模块详解：发送邮件

使用 Python 的 smtplib 模块来发送邮件非常简单，大部分底层的处理都由 smtplib 进行了封装，开发者只需要按照如下 3 步来发送邮件即可：

1. 连接 SMTP 服务器，并使用用户名、密码登录服务器。
2. 创建 EmailMessage 对象，该对象代表邮件本身。
3. 调用代表与 SMTP 服务器连接的对象的 sendmail() 方法发送邮件。

下面程序按照上面步骤示范了如何发送邮件：

```
1.          import smtplib
2.          from email.message import EmailMessage
3.
4.          # 定义 SMTP 服务器地址:
5.          smtp_server = 'smtp.qq.com'
6.          # 定义发件人地址
7.          from_addr = 'kongyeeku@qq.com'
8.          # 定义登录邮箱的密码
9.          password = '123456'
10.         # 定义收件人地址:
11.         to_addr = 'kongyeeku@163.com'
12.
13.         # 创建 SMTP 连接
14.         #conn = smtplib.SMTP(smtp_server, 25)
15.         conn = smtplib.SMTP_SSL(smtp_server,465)
16.         conn.set_debuglevel(1)
17.         conn.login(from_addr, password)             #①
18.         # 创建邮件对象
19.         msg = EmailMessage()
20.         # 设置邮件内容
21.         msg.set_content('您好，这是一封来自 Python 的邮件', 'plain', 'utf-8')
22.         # 发送邮件
23.         conn.sendmail(from_addr, [to_addr], msg.as_string())
24.         # 退出连接
25.         conn.quit()
```

上面程序中，第 15 行、19 行和 23 行代码基本代表了使用 Python 的 smtp 模块发送邮件的 3 大核心步骤，其中 ① 号代码使用了发件人的地址和密码来登录邮箱。

关于该程序有以下几点需要说明：

- 程序中提供的邮箱密码是错误的，不用尝试。读者必须改为使用自己的邮箱地址和密码。

- 早期 SMTP 服务器都采用普通的网络连接，因此默认端口是 25。但现在绝大部分 SMTP 都是基于 SSL（Secure Socket Layer）的，这样保证网络上传输的信息都是加密过的，从而使得信息更加安全。这种基于 SSL 的 SMTP 服务器的默认端口是 465。上面程序中连接的是 QQ 邮箱的基于 SSL 的 SMTP 服务器，QQ 邮箱服务器不支持普通的 SMTP。
- 国内有些公司的免费邮箱（比如 QQ 邮箱）默认是关闭了 SMTP 的，因此需要读者登录邮箱进行设置。
- 由于该程序发送的邮件太简单，邮件没有主题，而且程序在测试过程中可能会发送很多邮件，因此有些邮箱服务商会将该程序发送的邮件当成垃圾邮件。

> 早期 Python 2.x 提供了 email.mime、email.header、email.charset、email.encoders、email.iterators 等库来处理邮件，这些库设计得过于烦琐，用起来极为不便，因此读者应该尽快改为使用最新的 Python 库。本教程不会介绍这些过时的库。具体可参考 https://docs.python.org/3/library/email.html 页面的说明。

由于程序打开了 smtplib 调试模式（将 debuglevel 设置为 1），因此在运行该程序时，可以看到 SMTP 发送邮件的详细过程。当程序运行结束后，将可以在收件人邮箱中看到一封新邮件（可能在垃圾邮件内），如图 1 所示。

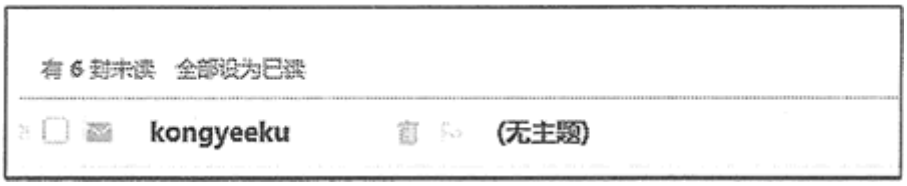


图 1 在收件人邮箱内收到了 smtplib 模块发送的邮件

上面这封邮件是最简单的，没有为该邮件设置主题、发件人名字和收件人名字，邮件内容也只是简单的文本。如果要为邮件设置主题、发件人名字和收件人名字，那么只需设置 EmailMessage 对象的相应属性即可。如果程序要将邮件内容改为 HTML 内容，那么只需将调用 EmailMessage 的 set_content() 方法的第二个参数设置为 html 即可。

例如，如下程序只是对 EmailMessage 进行了修改：

```
import smtplib
from email.message import EmailMessage

# 定义 SMTP 服务器地址:
smtp_server = 'smtp.qq.com'
# 定义发件人地址
from_addr = 'kongyeeku@qq.com'
# 定义登录邮箱的密码
password = '123456'
# 定义收件人地址:
to_addr = 'kongyeeku@163.com'

# 创建 SMTP 连接
#conn = smtplib.SMTP(smtp_server, 25)
conn = smtplib.SMTP_SSL(smtp_server,465)
conn.set_debuglevel(1)
conn.login(from_addr, password)              #①
# 创建邮件对象
msg = EmailMessage()
# 设置邮件内容，指定邮件内容为 HTML
msg.set_content('<h2>邮件内容</h2>' +
    '<p>您好，这是一封来自 Python 的邮件<p>' +
    '来自<a href="http://c.biancheng.net">C 语言中文网</a>', 'html',
'utf-8')
```

```
24.         msg['subject'] = '一封 HTML 邮件'
25.         msg['from'] = '李刚 <%s>' % from_addr
26.         msg['to'] = '新用户 <%s>' % to_addr
27.         # 发送邮件
28.         conn.sendmail(from_addr, [to_addr], msg.as_string())
29.         # 退出连接
30.         conn.quit()
```

该程序与上一个程序基本相似，只是在调用 set_content() 方法时将第二个参数改为了“html”。

此外，程序增加了上面 3 行代码（第 24~26 行），分别用于设置邮件主题、发件人名字和收件人名字。运行上面程序，在目标邮箱内可以看到如图 2 所示的邮件。

图 2 设置了邮件主题、发件人名字和收件人名字的 HTML 邮件

打开该邮件，将可以看到如图 3 所示的邮件内容。

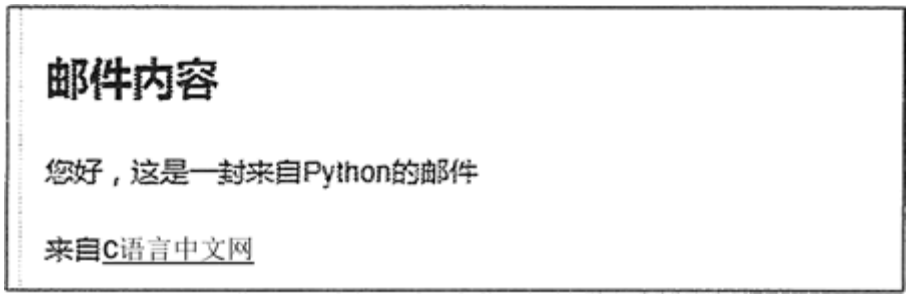


图 3 HTML 邮件内容

如果希望实现图文并茂的邮件，也就是在邮件中插入图片，则首先要给邮件添加附件（不要直接在邮件中嵌入外链的图片，很多邮箱出于安全考虑，都会禁用邮件中外链的资源）。因此，如果直接在 HTML 右键中外链其他图片，那么该图片很有可能显示不出来。

为了给邮件添加附件，只需调用 EmailMessage 的 add_attachment() 方法即可。该方法支持很多参数，最常见的参数如下：

- maintype：指定附件的主类型。比如指定 image 代表附件是图片。
- subtype：指定附件的子类型。比如指定为 png，代表附件是 PNG 图片。一般来说，子类型受主类型的限制。
- filename：指定附件的文件名。
- cid=img：指定附件的资源 ID，邮件正文可通过资源 ID 来引用该资源。

例如，如下程序为邮件添加了 3 个附件：

```
import smtplib, email.utils
from email.message import EmailMessage

# 定义 SMTP 服务器地址:
smtp_server = 'smtp.qq.com'
# 定义发件人地址
from_addr = 'kongyeeku@qq.com'
# 定义登录邮箱的密码
password = '123456'
# 定义收件人地址:
to_addr = 'kongyeeku@163.com'
```

```
12.
13.         # 创建 SMTP 连接
14.         #conn = smtplib.SMTP(smtp_server, 25)
15.         conn = smtplib.SMTP_SSL(smtp_server,465)
16.         conn.set_debuglevel(1)
17.         conn.login(from_addr, password)              #①
18.         # 创建邮件对象
19.         msg = EmailMessage()
20.         # 随机生成两个图片 id
21.         first_id, second_id = email.utils.make_msgid(), email.utils.make_msgid()
22.         # 设置邮件内容，指定邮件内容为 HTML
23.         msg.set_content('<h2>邮件内容</h2>' +
24.             '<p>您好，这是一封来自 Python 的邮件' +
25.             '<img src="cid:' + second_id[1:-1] +'"><p>' +
26.             '来自<a href="http://c.biancheng.net">C 语言中文网</a>')
27.         msg['subject'] = '一封 HTML 邮件'
28.         msg['from'] = '李刚 <%s>' % from_addr
29.         msg['to'] = '新用户 <%s>' % to_addr
30.         with open('E:/logo.jpg', 'rb') as f:
31.             # 添加第一个附件
32.             msg.add_attachment(f.read(), maintype='image',
33.                 subtype='jpeg', filename='test.png', cid=first_id)
34.         with open('E:/fklogo.gif', 'rb') as f:
35.             # 添加第二个附件
36.             msg.add_attachment(f.read(), maintype='image',
37.                 subtype='gif', filename='test.gif', cid=second_id)
38.         with open('E:/fkit.pdf', 'rb') as f:
39.             # 添加第三个附件，邮件正文不需引用该附件，因此不指定 cid
40.             msg.add_attachment(f.read(), maintype='application',
41.                 subtype='pdf', filename='test.pdf',)
42.         # 发送邮件
43.         conn.sendmail(from_addr, [to_addr], msg.as_string())
44.         # 退出连接
45.         conn.quit()
```

该程序与上一个程序的最大区别在于增加了一段代码（31 行代码起），它们为邮件添加了三个附件。

由于邮件正文不需要引用第三个附件，因此程序添加第三个附件时没有指定 cid 属性。

在添加附件时指定 cid 属性之后，程序就可以在邮件正文中通过该 cid 来引用附件，如上面程序中邮件正文代码所示。运行上面程序，可以在目标邮箱内看到一封新邮件。打开该邮件，可看到邮件内容。

通过上面三个示例，可以发现使用 smtplib 模块发送邮件很简单，基本只需要连接服务器、创建邮件和发送邮件三个步骤。如果要构建复杂的邮件内容，则主要通过 EmailMessage 对象来进行设置。EmailMessage 也是 Python 3.x 对邮件处理的巨大简化，它把对邮件内容的各种处理都封装在 EmailMessage 类中，因此使得编程变得轻松、简单。

15.Pythonpoplib 模块：收取邮件

使用 poplib 模块收取邮件也很简单，该模块提供了 poplib.POP3 和 poplib.POP3_SSL 两个类，分别用于连接普通的 POP 服务器和基于 SSL 的 POP 服务器。

一旦使用 poplib.POP3 或 poplib.POP3_SSL 连接到服务器之后，接下来基本就按照 POP3 协议与服务器进行交互。为了更好地理解 poplib 模块的运行机制，下面先简单介绍 POP3 协议内容。

POP3 协议也属于请求，响应式交互协议，当客户端连接到服务器之后，客户端向 POP 服

务器发送请求，而 POP 服务器则对客户端生成响应数据，客户端可通过响应数据下载得到邮件内容。当下载完成后，邮件客户端可以删除或修改任意邮件，而无须与电子邮件服务器进行进一步交互。

POP3 的命令和响应数据都是基于 ASCII 文本的，并以 CR 和 LF(/r/n) 作为行结束符，响应数据包括一个表示返回状态的符号（ +/ ）和描述信息。

请求和响应的标准格式如下：

```
请求标准格式：命令 [参数] CRLF

响应标准格式：+OK /[-ERR] description CRLF
```

POP3 协议客户端的命令和服务器端对应的响应数据如下：

- user name：向 POP 服务器发送登录的用户名。
- pass string：向 POP 服务器发送登录的密码。
- quit：退出 POP 服务器。
- stat：统计邮件服务器状态，包括邮件数和总大小。
- list [msg_no]：列出全部邮件或指定邮件。返回邮件编号和对应大小。
- retr msg_no：获取指定邮件的内容（根据邮件编号来获取，编号从 1 开始）。
- dele msg_no：删除指定邮件（根据邮件编号来删除，编号从 1 开始）。
- noop：空操作。仅用于与服务器保持连接。
- rset：用于撤销 dele 命令。

poplib 模块完全模拟了上面命令，poplib.POP3 或 poplib.POP3_SSL 为上面命令提供了相应的方法，开发者只要依次使用上面命令即可从服务器端下载对应的邮件。

使用 poplib 收取邮件可分为两步：

1. 使用 poplib.POP3 或 poplib.POP3_SSL 按 POP3 协议从服务器端下载邮件。
2. 使用 email.parser.Parser 或 email.parser.BytesParser 解析邮件内容，得到 EmailMessage 对象，从 EmailMessage 对象中读取邮件内容。

下面程序示范了如何使用 poplib 模块来收取邮件：

```
import poplib, os.path, mimetypes
from email.parser import BytesParser, Parser
from email.policy import default

# 输入邮件地址, 口令和 POP3 服务器地址:
email = 'kongyeeku@qq.com'
password = '123456'
pop3_server = 'pop.qq.com'

# 连接到 POP 3 服务器:
#conn = poplib.POP3(pop3_server, 110)
conn = poplib.POP3_SSL(pop3_server, 995)
# 可以打开或关闭调试信息:
conn.set_debuglevel(1)
# 可选:打印 POP 3 服务器的欢迎文字:
print(conn.getwelcome().decode('utf-8'))
# 输入用户名、密码信息
```

```
18.         # 相当于发送 POP 3 的 user 命令
19.         conn.user(email)
20.         # 相当于发送 POP 3 的 pass 命令
21.         conn.pass_(password)
22.         # 获取邮件统计信息，相当于发送 POP 3 的 stat 命令
23.         message_num, total_size = conn.stat()
24.         print('邮件数: %s. 总大小: %s' % (message_num, total_size))
25.         # 获取服务器上的邮件列表，相当于发送 POP 3 的 list 命令
26.         # resp 保存服务器的响应码
27.         # mails 列表保存每封邮件的编号、大小
28.         resp, mails, octets = conn.list()
29.         print(resp, mails)
30.         # 获取指定邮件的内容（此处传入总长度，也就是获取最后一封邮件）
31.         # 相当于发送 POP 3 的 retr 命令
32.         # resp 保存服务器的响应码
33.         # data 保存该邮件的内容
34.         resp, data, octets  = conn.retr(len(mails))
35.         # 将 data 的所有数据（原本是一个字节列表）拼接在一起
36.         msg_data = b'\r\n'.join(data)
37.         # 将字符串内容解析成邮件，此处一定要指定 policy=default
38.         msg = BytesParser(policy=default).parsebytes(msg_data)        #①
39.         print(type(msg))
40.         print('发件人:' + msg['from'])
41.         print('收件人:' + msg['to'])
42.         print('主题:' + msg['subject'])
43.         print('第一个收件人名字:' + msg['to'].addresses[0].username)
44.         print('第一个发件人名字:' + msg['from'].addresses[0].username)
45.         for part in msg.walk():
46.             counter = 1
47.             # 如果 maintype 是 multipart，说明是容器（用于包含正文、附件等）
48.             if part.get_content_maintype() == 'multipart' :
49.                 continue
50.             # 如果 maintype 是 multipart，说明是邮件正文部分
51.             elif part.get_content_maintype() == 'text':
52.                 print(part.get_content())
53.             # 处理附件
```

```
            else :
                # 获取附件的文件名
                filename = part.get_filename()
                # 如果没有文件名，程序要负责为附件生成文件名
                if not filename:
                    # 根据附件的 contnet_type 来推测它的后缀名
                    ext = mimetypes.guess_extension(part.get_content_type())
                    # 如果推测不出后缀名
                    if not ext:
                        # 使用.bin 作为后缀名
                        ext = '.bin'
                    # 程序为附件来生成文件名
                    filename = 'part-%03d%s' % (counter, ext)
                counter += 1
                # 将附件写入的本地文件
                with open(os.path.join('.', filename), 'wb') as fp:
                    fp.write(part.get_payload(decode=True))
        # 退出服务器，相当于发送 POP 3 的 quit 命令
        conn.quit()
```

上面程序通过 poplib 模块使用 POP3 命令从服务器端下载邮件，其实就是依次发送 user、pass、stat、list、retr 命令的过程。当 retr 命令执行完成后，将得到最后一封邮件的数据 data，该 data 是一个 list 列表，因此程序需要先将这些数据拼接成一个整体，然后使用 ① 号代码将邮件数据恢复成 EmailMessage 对象。

这里有一点需要指出，程序在创建 BytesParser（解析字节串格式的邮件数据）或 Parser（解析字符串格式的邮件数据）时，必须指定 policy=default，否则，BytesParse 或 Parser 解析邮件数据得到的就是过时的 Message 对象，处理起来非常不方便。

在创建 BytesParse 或 Parser 解析器时，一定要指定 policy=default；否则，解析出来的对象就是 Message，而不是新的 EmailMessage。

程序在 ① 号代码之后特意输出了解析得到的 msg 类型，此时应该看到的是 EmailMesssage，而不是过时的 Message 对象。

在 ① 号代码之前，就是完成 poplib 模块收取邮件的第一步，即从服务器端下载邮件；在 ① 号代码之后，就是完成 poplib 模块收取邮件的第二步，也就是解析邮件内容。

如果程序要获取邮件的发件人、收件人和主题，直接通过 EmailMessage 的相应属性来获取即可，与前面为 EmailMessage 设置发件人、收件人和主题的方式是对应的。

如果程序要读取 EmailMessage 的各部分，则需要调用该对象的 walk() 方法，该方法返回一个可迭代对象，程序使用 for 循环遍历 walk() 方法的返回值，对邮件内容进行逐项处理：

- 如果邮件某项的 maintype 是 'multipart'，则说明这一项是容器，用于包含邮件内容、附件等其他项。
- 如果邮件某项的 maintype 是 'text'，则说明这一项的内容是文本，通常就是邮件正文或文本附件。对于这种文本内容，程序直接将其输出到控制台中。
- 如果邮件某项的 maintype 是其他，则说明这一项的内容是附件，程序将附件内容保存在本地文件中。

运行上面程序，可以看到程序收取了指定邮件的最后一封邮件，并将邮件内容输出到控制台中，将邮件附件保存在本地文件中。

第 18 章 Python 文档和测试

文档和测试是与 Python 开发相关的两大块附属知识。

对于实际的企业开发来说，有效的说明文档是非常重要的一环，否则其他人员就不能有效地使用该程序。Python 提供了 pydoc 工具用来查看、生成文档。

与此同时，Python 软件测试也是本章的重点内容，将重点介绍 Python 的两种测试：文档测试和单元测试，另外 unittest 是目前最流行的单元测试工具，需要读者重点掌握。

1.编写 Python 开发文档的重要性

为自己的代码做的最重要一件事，就是告诉别人如何使用它，这有可能像只需要说明一点稍微复杂的逻辑那样简单，也有可能像编写 3000 行长的程序介绍那么复杂。我们把这些说明叫做文档。

文档既可能是说明如何使用程序的文件，也可能会内嵌在代码中，它可以包含示例项目、教程或者每个函数的清单。编写文档看上去像是一项占用了编写代码的时间的活动，然而，这却是大多数程序员必须要做的工作。

占用编写代码的时间难道不是一件糟糕的事吗？事实上，文档为每个参与者所节省的时间之多，常常令人难以置信。如果很好地为代码编写了文档，今后接手工作的所有程序员，都能

更容易地搞明白代码是做什么的，以及他们可以如何使用代码，他们就能扩展代码来做更多的事情，或者修改所出现的 Bug。

不仅如此，文档在将来可能还会对自己有所帮助。今天看上去很直观的一些代码，明天可能就会令人很困惑。你可能真的忘记了为什么总是需要给出一个参数，或者为什么要用一个 `try/except` 子句把函数包围起来。

好的文档应该告诉用户想要使用一段代码所需要知道的信息，而且没有废话。试图介绍每种可能的状态或说明所编写的每行代码，会很容易把代码淹没在一个文档中。

过多的文档就像使用新数码相机那 320 页的用户手册一样。你可能不会痛下决心去阅读它。事实却是如此，你可能只会在使用遇到困难时才会从文档中寻找解决方案，但是并不会太在意自己可能错过了数百个其他功能。

好的文档应该是有组织的。其他的程序员会希望在特定的位置有某种特定类型的文档。关于函数能够做什么的说明，应该包含在函数中，并且通常要使用某种特定的格式。安装说明不应该在代码中，而应该位于一个单独的文件中。对于需要些技巧的代码的详细说明，应该在该代码的附近。

既然文档这么有用，那么应该如何编写呢？怎样才能编写出“高大上”的 Python 软件开发文档呢？请大家带着问题继续阅读《如何编写 Python 软件开发文档（7 个技巧）》一节。

2.读完本文，你也能编写高大上的 Python 软件开发文档

开发文档是经常被程序员忽略的工作，有时也会被管理者忽略。这往往是由于在项目生命周期结束的后期缺乏时间，以及人们认为自己不擅长写作，其中一些人确实写不好，但他们中的大多数能够完成一个良好的文档。

在任何情况下，匆忙编写文档的结果是文档会变得一团糟。大多时候，开发人员讨厌做这种工作，尤其当现有文档需要更新时，情况会更糟。因为经理不知道如何处理更新，许多项目只是提供简陋而又过时的文档。

编写良好的文档在许多方面比编写代码更容易，大多数开发人员认为这很难，但通过遵循一组简单的规则，它会变得非常容易。

本节给初学者介绍了 7 条应该遵循的规则，可以应用在所有情况下，让编写开发文档事半功倍。

1) 专注于想法，然后审查和塑造你的文本

要知道，几乎没有人在第一次就写出完美的开发文档，因为多数人在编写文档时都尝试一次性将文档编写到完美，为此他们每写两句就停止写作，然后阅读他们并做一些修正。这意味着，他们将重点都放在文本的内容和风格上。

这种编写方式的结果往往没有预期的那么好，原因很简单，人们在还没有想清楚要写的内容

之前，就将大量的时间和精力花在修正文件的风格和形状上。

我认为，正确的编写方式应分为 2 步，第一步无需考虑文本的风格和组织，专注于编写文档的内容。注意，最好能够将所有的想法都写在纸上，至于怎么写先不考虑。

也就是说，第一步的重点就是打造内容，只要不是关于内容的问题（例如语法错误等），都不需要关心。

通过这种方式，开发者能够专注于他的想法，并且可能会从想到更多的内容，而不只局限于其最初的想法。注意，在编写过程中，很容易在头脑中一闪而过一些想法，当它们出现时，比较好的的做法是将它们写到纸上，写完后可以继续回到当前的写作中。

第二步就是回读整个文本并对其进行修正，以便每个人都能理解。修正文本意味着要增强其风格，纠正其错误，重新组织它，并删除任何冗余的内容。

总之，当编写开发文档的时间有限时，比较好的做法就是将可利用的时间一分两半，一半用于写作，一半用于清理和组织文本。

2) 准确定位读者

编写开发文档时，首先应该考虑清楚，谁会读这个文档？

千万不要认为定位读者很简单，因为开发文档解释了一个软件如何工作，并且通常为每个可能获取和使用代码的人而写，读者可能是正在寻找问题的合适技术解决方案的研究员，也可能是需要利用其实现特性的开发者，甚至架构师，也可以从架构的角度来看它是否符合需求。

因此，好的开发文档应该遵循一个简单的规划，即每个文本应该只针对一种类型的读者。而这也使编写文档更容易，因为我们可以准确地知道面对的是什么样的读者。

另外，开发文档中最好提供一个简短的介绍性文本，用来解释文档是什么，并知道读者去读他感兴趣的部分。

3) 读者更喜欢简短的句子

对于绝大多数读者来说，往往更喜欢短而简单的句子，而不是长篇大论的段落。

使句子短而简单的方法有如下几个：

- 每个句子不应超过 2 行；
- 每一段应只表达一个主要思想，最多包含 3 到 4 个句子；
- 每个想法的解释不要重复太多，避免像新闻那样一再重复，只要保证读者能够理解即可。
- 如果你不是一个真正优秀的作家，避免在文档中讲笑话，因为在技术文档中添加滑稽的语言是很难的，很少有人掌握它。如果你想添加一些幽默，可以将它们放到代码示例中。

4) 撰写贴合内容的标题

不好的软件开发文档几乎都存在一个问题，即我们知道文档中包含自己要找的信息，但却要花很长时间去找。当作者没有将它们的文本内容合理地组织到标题中时，就会出现这种情况。

因此，建议大家在编写开发文档时，将段落聚集在给定章节的有意义的标题下；整个文档的标题可以用短语来组织；文档的目录可以由所有章节的标题组成。

总之，撰写标题最简单的做法就是问自己："在百度中输入什么短语才能找到此部分内容"？

5) 文档中应使用项目中的代码作为示例

当读者试图通过一个使用示例来理解一段代码如何运行时，不切实际的示例代码往往更让人难以理解。

举个例子，假设我们想要展示如何使用 parse() 函数：

```
1.          from atomisator.parser import parse
2.          #用法如下:
3.          stuff = parse("some-feed.xml")
4.          stuff.next()
```

输出结果为：

```
{'title':'foo','content':'blable'}
```

更好的例子应该是，让解析器知道如何使用解析函数返回一个 feed 的内容，可作为一个顶级函数使用，如下所示：

```
1.          from atomisator.parser import parse
2.          #用法如下:
3.          my_feed = parse("http://tarekziade.wordpress.com/feed")
4.          my_feed.next()
```

输出结果为：

```
{'title':'eight tips to start with python','content':'The first tip is ..., ...'}
```

这个微小的差距可能有点过分夸张，但实际上它能够使文档更有用，读者可以将这些代码行复制到一个 shell 程序中，理解 parse 将使用一个 URL 作为参数，并且返回包含博客条目的一个枚举型变量。

总之，文档中使用的代码示例应该在实际的程序中直接可用。

6) 文档内容以实用为主

其实，软件开发中所用的大部分方法，是不需要用文档进行说明的，相比更详尽的文档，使软件正常工作无疑更重要。因此，一个好的做法是只编写真正需要的文档，而不是编写一个详尽的文档集。

举个例子，对于管理员来说，只需在文档中说明软件是如何工作的就足够了，他们除了要知道这个工具的配置和运行方法之外，没有其他的需求。所以，这个文档应将范围限制"如何在自己的服务器上运行软件"。

可以运行的软件，远远胜过面面俱到的文档。

7) 使用模板

以维基百科为例，你会发现它每个页面的布局都是相似的，一侧有用于总结日期或事实的文本框，文档开头总有一个目录（包含该页面中的所有标题的链接），文档结尾处总有一个参考部分，等等。

使用维基百科的用户习惯了这个页面的布局，例如他们就可以快速查看每个页面的目录部分，还可以快速查看参考文献等等，固定的结构布局会提高用户的效率。

所以，使用模板强制文档安装固定的模式，可以使用户更高效地使用它们。

3.Python docstring 添加注释

docstring（文档字符串）：给代码添加注释

dcostring 是一种特殊类型的注释，可以把它放在一个函数或类定义之后，或者一个文件的开头，其功能是说明函数、类或者模块是做什么的。

docstring 可以用三个引号、单引号或者双引号括起来，如下所示：

```
1.          class Test:
2.              '''测试 docstring 的类
3.              '''
4.              def __init__(self):
5.                  '''Test 类初始化对象,不需要任何参数
6.                  '''
7.                  self.num = 10
8.          t = Test()
9.          help(t)
```

有读者可能会问，为什么不直接使用注释？因为 docstring 在 Python 中很特殊，有一些 Python 内建的函数，可以使用 docstring 来帮助其他程序员来使用你的代码。

运行上面程序，可以看到如下输出结果：

```
Help on Test in module __main__ object:


class Test(builtins.object)
 |  测试 docstring 的类
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |      Test 类初始化对象,不需要任何参数
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

可以看到，help() 函数找到了所有的函数和 docstring，并且自动创建了一个格式漂亮的帮助页面。由此，用户不需要深入你的代码，就可以知道哪些函数可用，函数需要接受什么参数。

这里给初学者总结了适合写在 docstring 中的内容：

- 函数或者类、模板是干什么的；
- 对于类来说，它是否可以直接使用，或者它只是基类，不能单独使用；

- 用户期望从一个函数中返回什么；
- 在使用过程中，可能会给出的警告类型；
- 如果包含参数，则参数的值应该是什么数据类型。

> 注意，PEP 257 中包含了编写 docstring 的指南，感兴趣的读者可前往 Python PEP-0257 进行阅读。

4.Python pydoc

pydoc 模块详解：查看、生成帮助文档

前面已经介绍了为函数、类、方法等编写文档（只要在函数、类、方法定义后定义一个字符串即可）。前面也介绍了使用 help() 函数和 __doc__ 属性来查看函数、类、方法的文档，但这种方式总是在控制器中查看，有时候难免不太方便。

借助于 Python 自带的 pydoc 模块，可以非常方便地查看、生成帮助文档，该文档是 HTML 格式的，因此查看、使用起来非常方便。

这里先提供如下 Python 源文件（文件名为 fkmodule.py）：

```
MY_NAME = 'C 语言中文网'

def say_hi(name):
    '''
    定义一个打招呼的函数
    返回对指定用户打招呼的字符串
    '''
    print("执行 say_hi 函数")
    return name + '您好！'
```

```
    def print_rect(height, width):
        '''
        定义一个打印矩形的函数
        height - 代表矩形的高
        width - 代表矩形的宽
        '''
        print(('*' * width + '\n') * height)
    class User:
        NATIONAL = 'China'
        '''
        定义一个代表用户的类
        该类包括 name、age 两个变量
        '''
        def __init__(self, name, age):
            '''
            name 初始化该用户的 name
            age 初始化该用户的 age
            '''
            self.name = name
            self.age = age
        def eat (food):
            '''
            定义用户吃东西的方法
            food - 代表用户正在吃的东西
            '''
            print('%s 正在吃%s' % (self.name, food))
```

上面代码定义了一个 fkmodule.py 源文件，也就是定义了一个 fkmodule 模块，该模块为函数、类和方法都提供了文档说明。下面将会示范如何使用 pydoc 来查看、生成该模块的文档。

pydoc 在控制台中查看文档

先看如何使用 pydoc 模块在控制台中查看 HTML 文档。使用 pydoc 模块在控制台中查看帮助文档的命令如下：

```
python -m pydoc 模块名
```

上面命令中的 -m 是 python 命令的一个选项，表示运行指定模块，此处表示运行 pydoc 模块。后面的“模块名”参数代表程序要查看的模块。

例如，在 fkmodule.py 文件所在目录下运行如下命令：

```
python -m pydoc fkmodule
```

上面命令表示使用 pydoc 查看 fkmodule 模块的命令。运行该命令，将看到如图 1 所示的输出结果。

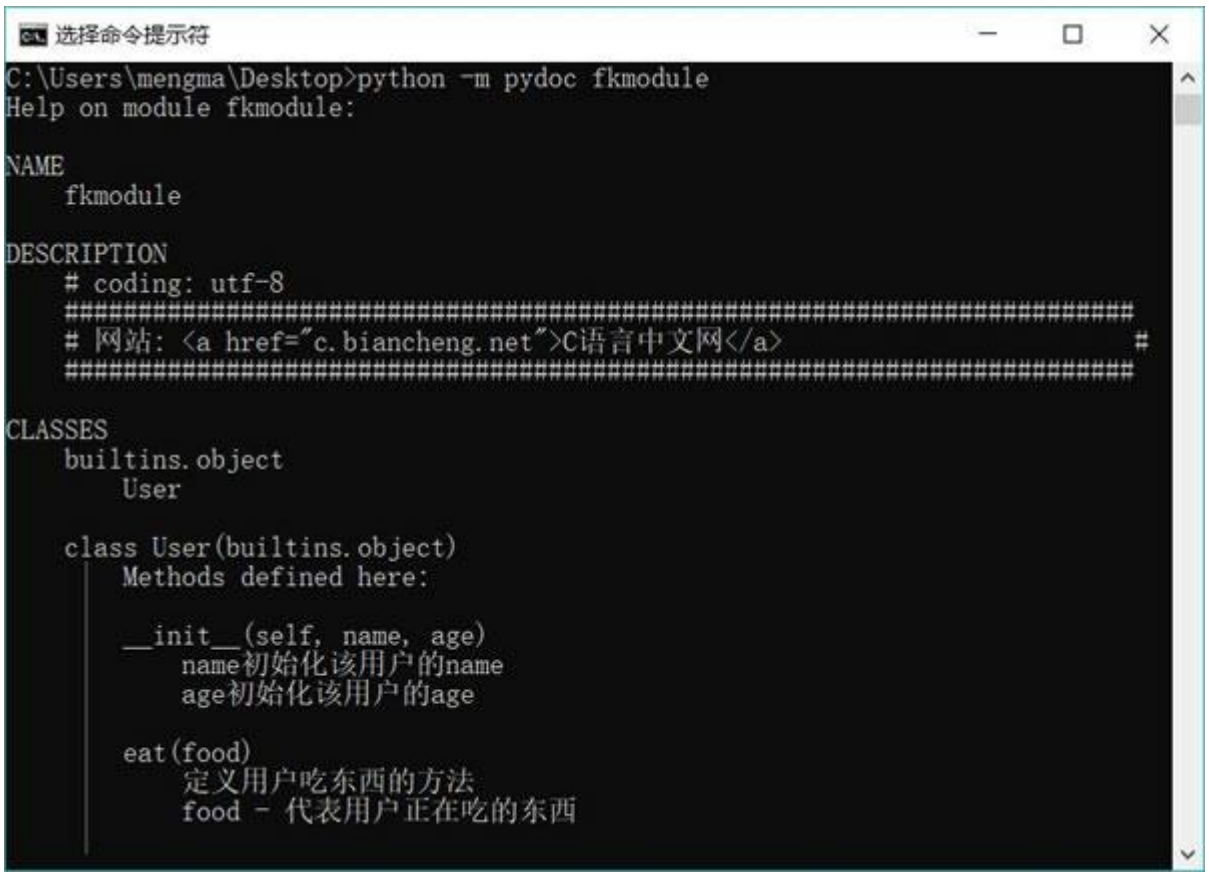


图 1 使用 pydoc 模块在控制台中查看文档

按下空格键，pydoc 将会使用第二屏来显示文档信息，如图 2 所示。

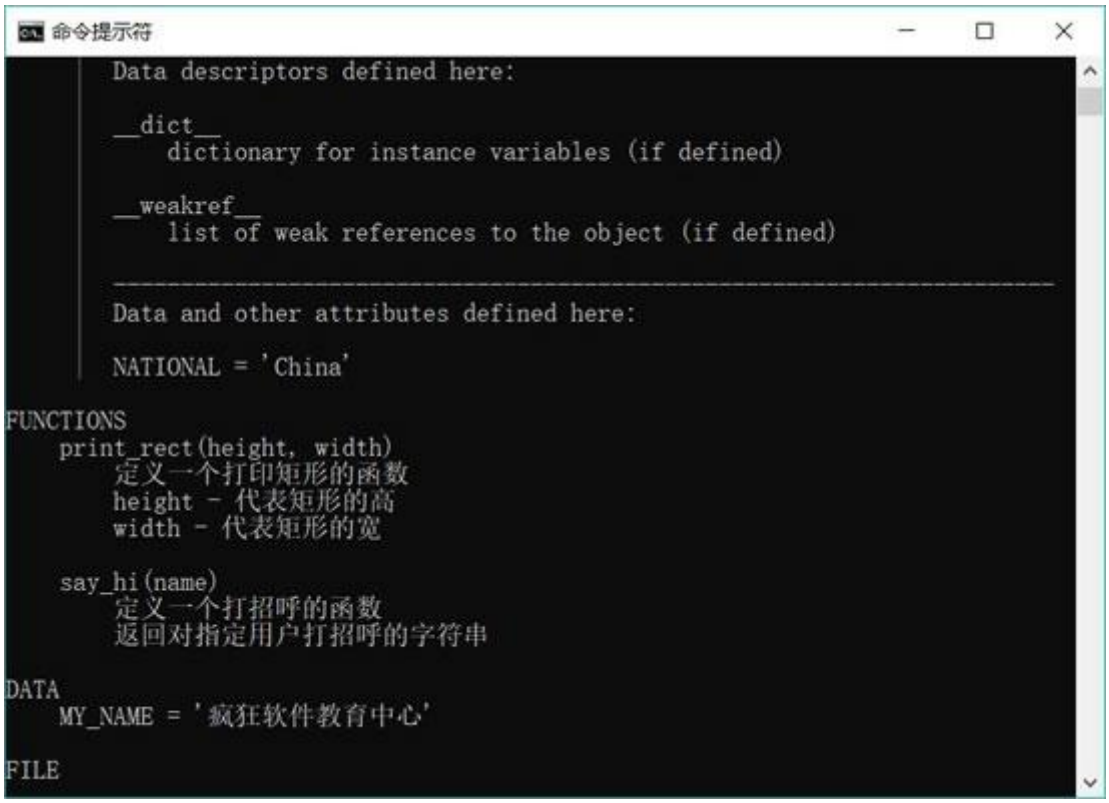


图 2 使用 pydoc 查看文档的第二屏信息

从图 1 可以看出，使用 pydoc 在控制台中查看文档时，由于一屏无法显示所有的文档信息，因此同样需要以分屏的形式来显示，这样查看其实并不方便，与使用 help() 命令查看帮助信息的差别并不大。

当然，在使用 pydoc 查看帮助信息时，它会有自己的组织方式，它总是按如下顺序来显示模块中的全部内容：

- 模块的文档说明：就是 * .py 文件顶部的注释信息，这部分信息会被提取成模块的文档说明。
- CLASSES 部分：这部分会列出该模块所包含的全部类。
- FUNCTIONS 部分：这部分会列出该模块所包含的全部函数。

- DATA 部分：这部分会列出该模块所包含的全部成员变量。
- FILE 部分：这部分会显示该模块对应的源文件。

不管怎么样，直接在控制台中查看指定模块的帮助信息依然不太方便，下面将会使用 pydoc 来为指定模块生成 HTML 文档。

pydoc 生成 HTML 文档

使用 pydoc 模块在控制台中查看帮助文档的命令如下：

```
python -m pydoc -w 模块名
```

上面命令主要就是为 pydoc 模块额外指定了 -w 选项，该选项代表 write，表明输出 HTML 文档。

例如，在 fkmodule.py 所在当前目录下运行如下命令：

```
python -m pydoc -w fkmodule
```

运行上面命令，可以看到系统生成“wrote fkmodule.html” 提示信息。接下来可以在该目录下发现额外生成了一个 fkmodule.html 文件，使用浏览器打开该文件，可以看到如图 3 所示的页面。

图 3 使用 pydoc 生成的 HTML 文档

将图 3 所示的页面拉到下面，可以看到如图 4 所示的页面。

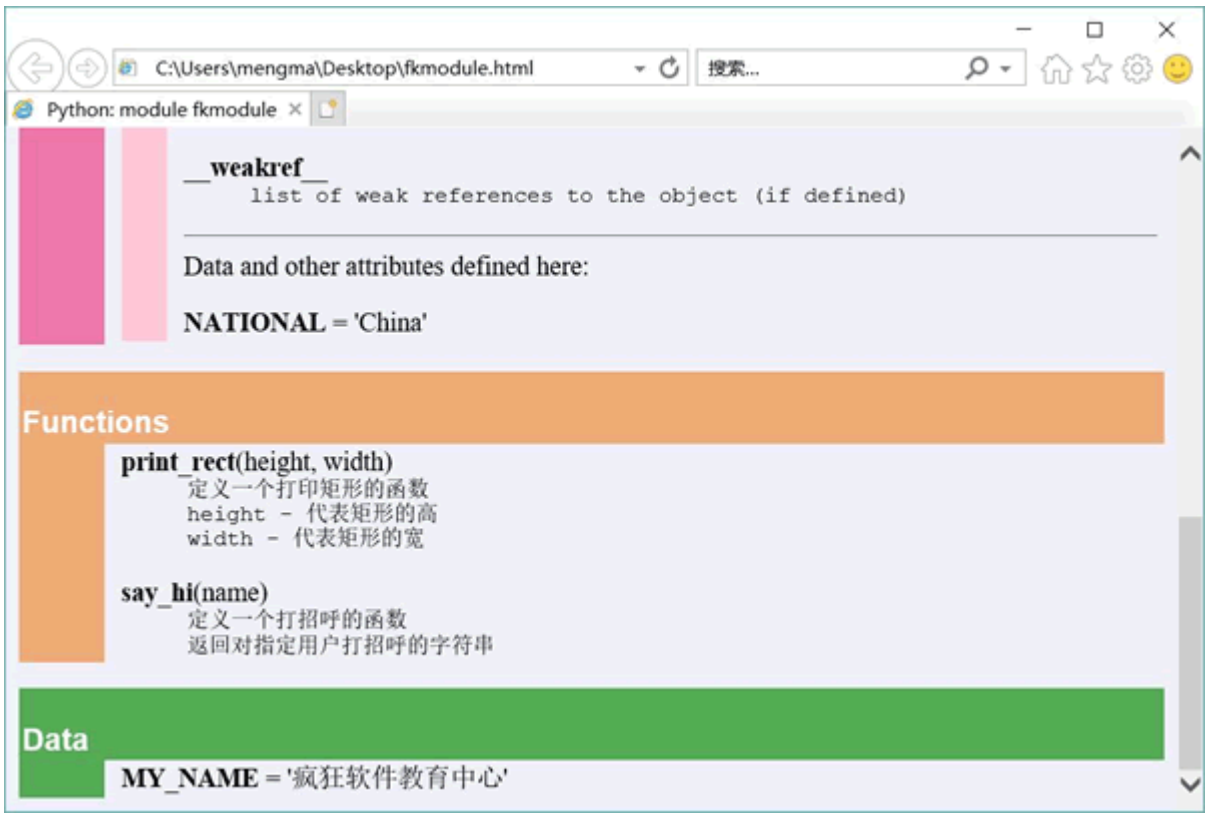


图 4 使用 pydoc 生成的文挡的第二屏信息

从图 3 和图 4 所示的页面来看，该 HTML 页面与在控制台中查看的文档信息基本相同，区别在于，由于这是一个 HTML 页面，因此用户可以拖动滑块来上、下滚动屏幕，以方便查看。

需要说明的是，pydoc 还可用于为指定目录生成 HTML 文档。例如，通过如下命令为指定目录下的所有模块生成 HTML 文档：

```
python3 -m pydoc -w 目录名
```

但上面命令有一个缺陷，那就是当该命令工具要展示目录下子文件的说明时，会去子目录下

找对应的 .html 文件，如果文件不存在，就会显示 404 错误。

如果真的要查看指定目录下所有子目录中的文档信息，则建议启动本地服务器来查看。

启动本地服务器来查看文档信息

启动本地服务器来查看文档信息，可以使用如下两个命令：

```
python3 -m pydoc -p 端口号
```

在指定端口启动 HTTP 服务器 ,接下来用户可以通过浏览器来查看 Python 的所有模块的文档信息：

```
python3 -m pydoc -b
```

在任意一个未占用的端口启动 HTTP 服务器，接下来用户同样可以通过浏览器来查看 Python 的所有模块的文档信息。

例如，在文件当前所在目录下运行如下命令：

```
python -m pydoc -p 8899
```

该命令工具将会显示如下输出信息：

```
Server ready at http://localhost:8899/

Server commands: [b]rowser, [q]uit
```

上面的输出信息提示 HTTP 服务器正在 8899 端口提供服务器，用户可以输入 b 命令来启动浏览器（实际上用户可以自行启动浏览器），也可以输入 q 命令来停止服务器。

打开浏览器访问 http://localhost:8899/，将会看到如图 5 所示的页面。

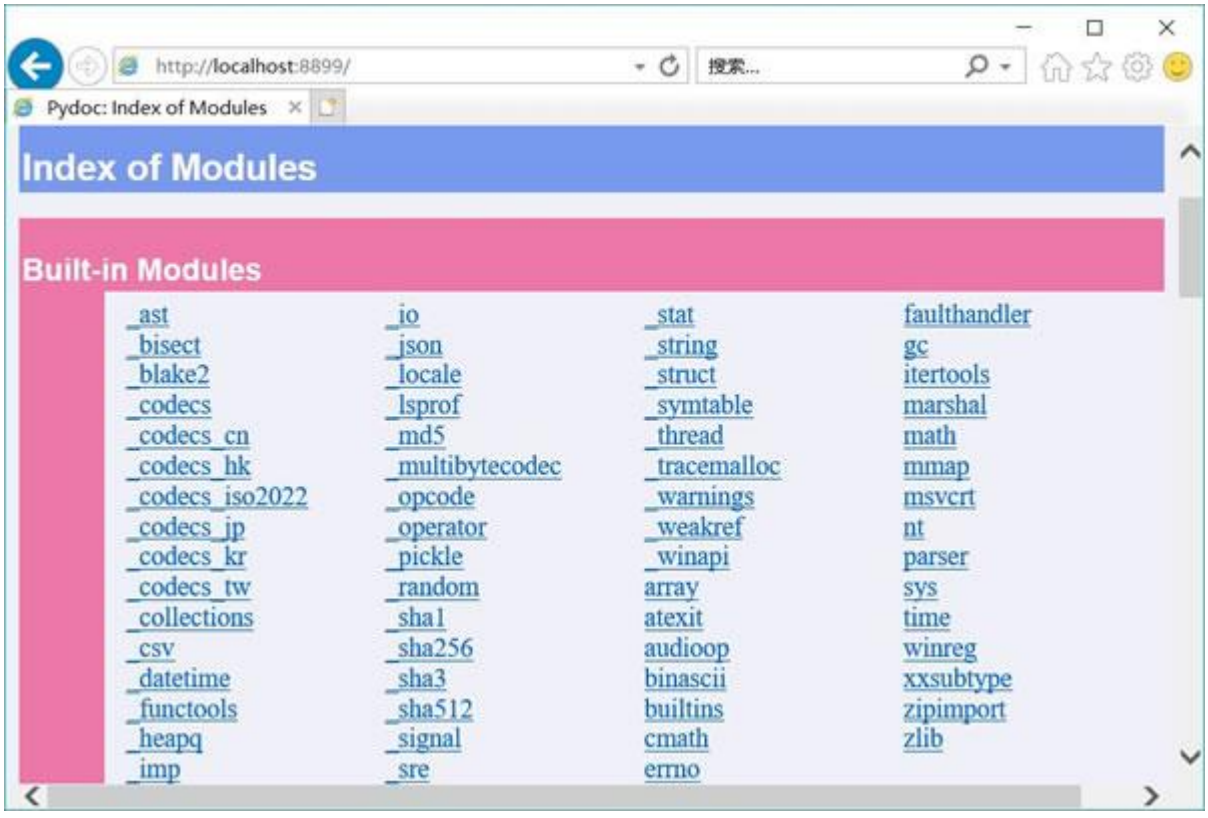


图 5 模块列表

从图 5 可以看出，该页面默认显示了当前 Python 的所有模块。其中：

- 第一部分显示 Python 内置的核心模块。
- 第二部分显示当前目录下的所有模块，此处显示的就是 fkmodule 模块。
- 第三部分显示 d:\python_module 目录下的所有模块，此时在该目录下并未包含任何模块。pydoc 之所以显示该目录，是因为本机配置了 PYTHONPATH 环境变量，其值为 .;d:\python_module，因此 pydoc 会自动显示该目录下的所有模块。换而言之，第三部分用于显示 PYTHONPATH 环境变量所指定路径下的模块。

如果读者将图 5 所示的页面向下拉，将会依次看到 Python 系统在 D:\Python\Python36\DLLs 、 D:\Python\Python36\lib 、 D:\Python\Python36\lib\site-packages 路径下的所有模块。

如果要查看指定模块，只要单击图 5 所示页面中的模块链接即可。例如，单击图 5 所示页面中的“fkmodule”模块链接，将会看到如图 6 所示的页面。

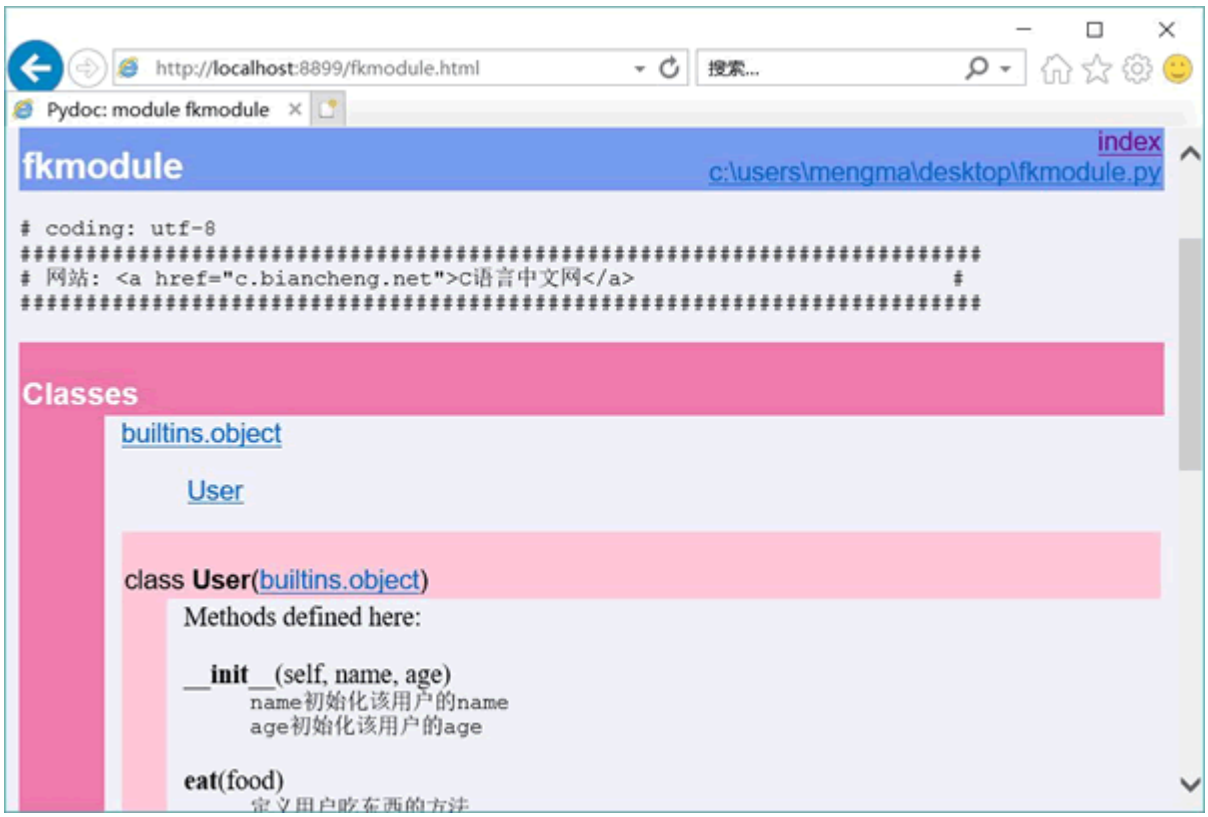


图 6 通过服务器查看 fkmodule 模块的信息

对比图 4 与图 6 所示的页面不难发现，它们显示的内容是一样的。因此，无论是生成 HTML 页面，还是直接启动 HTTP 服务器，都能看到相同的文档页面。

pydoc 查找模块

此外，pydoc 还提供了一个 -k 选项，该选项用于查找模块。该选项的语法格式如下：

```
python -m pydoc -k 被搜索模块的部分内容
```

例如，在 fkmodule.py 所在目录下运行如下命令：

```
python -m pydoc -k fk
```

可以看到如下输出信息：

```
fkmodule
```

从上面的输出信息可以看到，pydoc 找到了包含“fk”的 fkmodule。

5.Python doctest

所谓文档测试，指的是通过 doctest 模块运行 Python 源文件的说明文档中的测试用例，从而生成测试报告。

前面在查看 Python 的模块文档时，经常会看到如下信息：

```
>>>os.path.commonprefix (['/usr/lib','/usr/local/lib'])
'/usr/1'
>>>os.path.commonpath(['/usr/lib', '/usr/local/lib'])
'/usr'
```

用户完全可以将这些代码拷贝到 Python 的交互式解释器中运行，然后可以看到运行结果与文档中示例代码的输出结果完全一致。

上面的说明就是文档测试的注释，文档测试工具可以提取说明文档中的测试用例，其中“>>>”之后的内容表示测试用例，接下来的一行则代表该测试用例的输出结果。文档测试工具会判断测试用例的运行结果与输出结果是否一致，如果不一致就会显示错误信息。

现在定义一个简单的模块，该模块包含一个函数和一个类，程序为该函数和该类提供了说明文档，该文档中包含了测试用例。程序代码如下：

```
1.          def square (x):
2.              '''
3.              一个用于计算平方的函数
4.
5.              例如
6.              >>> square(2)
7.              4
8.              >>> square(3)
9.              9
10.             >>> square(-3)
11.             9
12.             >>> square(0)
13.             0
14.             '''
15.             return x * 2 # ①、故意写错的
16.
17.         class User:
18.             '''
19.             定义一个代表用户的类，该类包含如下两个属性：
20.             name - 代表用户的名字
21.             age - 代表用户的年龄
22.
23.             例如
24.             >>> u = User('fkjava', 9)
25.             >>> u.name
26.             'fkjava'
27.             >>> u.age
28.             9
29.             >>> u.say('i love python')
30.             'fkjava 说: i love python'
```

```
            '''
            def __init__(self, name, age):
                self.name = 'fkit' # ②、故意写错的
                self.age = age
            def say(self, content):
                return self.name + '说: ' + content
        if __name__=='__main__':
            import doctest
            doctest.testmod()
```

上面第一段注释代码就是程序为 square() 函数提供的测试用例，在文档中一共为该函数提供了 4 个测试用例；第二段注释代码是程序为 User 类提供的测试用例，在文档中一共为该类提供了 3 个测试用例，分别用于测试用户的 name 、age 和 say() 方法。

程序最后几行代码进行了判断，如果是直接使用 python 命令来运行该程序(__name__ 等于 __main__)，程序将导入 doctest 模块，并调用该模块的 testmod() 函数。

从上面程序可以看到，Python 为文档测试提供了 doctest 模块，该模块的用法非常简单，程序只要调用该模块的 testmod() 函数即可。

运行上面程序，可以看到如下输出结果：

```
**********************************************************************
File "C:\Users\mengma\Desktop\1.py", line 25, in __main__.User
Failed example:
    u.name
```

```
Expected:
    'fkjava'
Got:
    'fkit'
**********************************************************************
File "C:\Users\mengma\Desktop\1.py", line 29, in __main__.User
Failed example:
    u.say('i love python')
Expected:
    'fkjava 说: i love python'
Got:
    'fkit 说: i love python'
**********************************************************************
File "C:\Users\mengma\Desktop\1.py", line 8, in __main__.square
Failed example:
    square(3)
Expected:
    9
Got:
    6
**********************************************************************
File "C:\Users\mengma\Desktop\1.py", line 10, in __main__.square
```

```
Failed example:
    square(-3)
Expected:
    9
Got:
    -6
**********************************************************************
2 items had failures:
   2 of   4 in __main__.User
   2 of   4 in __main__.square
***Test Failed*** 4 failures.
```

从上面的输出结果可以看出，一共有 4 个测试没有通过，在 User 类中有两个测试没有通过，在 square() 函数中有两个测试没有通过。这是因为上面程序中 ① ② 两行代码故意写错了，其中 ① 号代码用于计算 x 的平方，应该写成 x**2，但漏写了一个星号；② 号代码则应该用传入的 name 参数对 self.name 赋值。

上面显示的测试输出结果也很清晰，每个测试用例结果都包含如下 4 部分：

1. 第一部分：显示在哪个源文件的哪一行。
2. 第二部分：Failed example，显示是哪个测试用例出错了。
3. 第三部分：Expected，显示程序期望的输出结果。也就是在“>>>命令”的下一行给出的运行结果，它就是期望结果。

4. 第四部分：Got，显示程序实际运行产生的输出结果。只有当实际运行产生的输出结果与期望结果一致时，才表明该测试用例通过。

将上面程序中 ① ② 两行代码修改正确，再次使用 python 命令来运行程序，将看不到任何输出结果，说明文档测试中的所有测试用例都通过了。

由此可见，Python 为文档注释提供了 doctest 模块，该模块的用法非常简单，程序只要导入该模块，并调用该模块的 testmod() 函数即可。testmod() 函数会自动提取该模块的说明文档中的测试用例，并执行这些测试用例，最终生成测试报告。如果存在没有通过的测试用例，程序就会显示有多少个测试用例没有通过；如果所有测试用例都能通过测试，则不生成任何输出结果。

6.Python unittest（PyUnit）

PyUnit(unittest) 是 Python 自带的单元测试框架 ,用于编写和运行可重复的测试。PyUnit 是 xUnit 体系的一个成员，xUnit 是众多测试框架的总称，PyUnit 主要用于进行白盒测试和回归测试。

> 如果你使用的是 2.1 或更早版本的 Python，则可能需要自行下载和安装 PyUnit，现在的开发者通常不需要操心这些事情。

通过 PyUnit 可以让测试具有持久性，测试与开发同步进行，测试代码与开发代码一同发布。使用 PyUnit 具有如下好处：

- 可以使测试代码与产品代码分离。
- 针对某一个类的测试代码只需要进行较少的改动，便可以应用于另一个类的测试。
- PyUnit 开放源代码，可以进行二次开发，方便对 PyUnit 的扩展。

PyUnit 是一个简单、易用的测试框架，其具有如下特征：

- 使用断言方法判断期望值和实际值的差异，返回 bool 值。
- 测试驱动设备可使用共同的初始化变量或实例。
- 测试包结构便于组织和集成运行。

PyUnit (unittest) 的用法

所有测试的本质其实都是一样的，都是通过给定参数来执行函数，然后判断函数的实际输出结果和期望输出结果是否一致。

PyUnit 测试与其他 xUnit 的套路一样，基于断言机制来判断函数或方法的实际输出结果和期望输出结果是否一致，测试用例提供参数来执行函数或方法，获取它们的执行结果，然后使用断言方法来判断该函数或方法的输出结果与期望输出结果是否一致，如果一致则说明测试通过；如果不一致则说明测试不通过。

目前还有一种流行的开发方式叫作测试驱动开发，这种方式强调先编写测试用例，然后再编写函数和方法。假如程序要开发满足 A 功能的 fun_a() 函数，采用测试驱动开发的步骤如下：

1. 为 fun_a() 函数编写测试用例，根据业务要求，使用大量不同的参数组合来执行 fun_a() 函数，并断言该函数的执行结果与业务期望的执行结果匹配。
2. 编写、修改 fun_a() 函数。
3. 运行 fun_a() 函数的测试用例 ,如果测试用例不能完全通过 ;则重复第 2 步和第 3 步，直到 fun_a() 的所有测试用例全部通过。

测试驱动开发强调结果导向，也就是在开发某个功能之前，先定义好该功能的最终结果（测试用例关注函数的执行结果），然后再去开发该功能。就像建筑工人在砌墙之前，要先拉好一根笔直的绳子（作用相当于测试用例），然后再开始砌墙，这样砌出来的墙就会符合标准。所以说测试驱动开发确实是一种不错的开发方式。

下面开发一个简单的 fk_math.py 程序，该程序包含两个函数，分别用于计算一元一次方程的解和二元一次方程的解。

```
1.          def one_equation(a , b):
2.              '''
3.              求一元一次方程 a * x + b = 0 的解
4.              参数 a - 方程中变量的系数
5.              参数 b - 方程中的常量
6.              返回 方程的解
7.              '''
8.              # 如果 a = 0，则方程无法求解
9.              if a == 0:
10.                 raise ValueError("参数错误")
11.             # 返回方程的解
12.             else:
13.         #         return -b / a  # ①
```

```
            return b / a
        def two_equation(a , b , c):
            '''
            求一元二次方程 a * x * x + b * x + c = 0 的解
            参数 a - 方程中变量二次幂的系数
            参数 b - 方程中变量的系数
            参数 c - 方程中的常量
            返回 方程的根
            '''
            # 如果 a == 0，变成一元一次方程
            if a == 0:
                raise ValueError("参数错误")
            # 有理数范围内无解
            elif b * b - 4 * a * c < 0:
                raise ValueError("方程在有理数范围内无解")
            # 方程有唯一的解
            elif b * b - 4 * a * c == 0:
                # 使用数组返回方程的解
                return -b / (2 * a)
            # 方程有两个解
            else:
                r1 = (-b + (b * b - 4 * a * c) ** 0.5) / 2 / a
                r2 = (-b - (b * b - 4 * a * c) ** 0.5) / 2 / a
                # 方程的两个解
                return r1, r2
```

在定义好上面的 tk_math.py 程序之后，该程序就相当于一个模块，接下来为该模块编写单元测试代码。

unittest 要求单元测试类必须继承 unittest.TestCase，该类中的测试方法需要满足如下要求：

- 测试方法应该没有返回值。

- 测试方法不应该有任何参数。
- 测试方法应以 test 开头。

下面是测试用例的代码：

```
import unittest

from fk_math import *

class TestFkMath(unittest.TestCase):
    # 测试一元一次方程的求解
    def test_one_equation(self):
        # 断言该方程求解应该为-1.8
        self.assertEqual(one_equation(5 , 9) , -1.8)
        # 断言该方程求解应该为-2.5
        self.assertTrue(one_equation(4 , 10) == -2.5 , .00001)
        # 断言该方程求解应该为 27/4
        self.assertTrue(one_equation(4 , -27) == 27 / 4)
        # 断言当 a == 0 时的情况，断言引发 ValueError
        with self.assertRaises(ValueError):
            one_equation(0 , 9)
    # 测试一元二次方程的求解
    def test_two_equation(self):
        r1, r2 = two_equation(1 , -3 , 2)
        self.assertCountEqual((r1, r2), (1.0, 2.0), '求解出错')
        r1, r2 = two_equation(2 , -7 , 6)
        self.assertCountEqual((r1, r2), (1.5, 2.0), '求解出错')
        # 断言只有一个解的情形
        r = two_equation(1 , -4 , 4)
        self.assertEqual(r, 2.0, '求解出错')
        # 断言当 a == 0 时的情况，断言引发 ValueError
        with self.assertRaises(ValueError):
            two_equation(0, 9, 3)
```

```
29.                 # 断言引发 ValueError
30.                 with self.assertRaises(ValueError):
31.                     two_equation(4, 2, 3)
```

上面测试用例中使用断言方法判断函数的实际输出结果与期望输出结果是否一致，如果一致则表明测试通过，否则表明测试失败。

在上面的测试用例中，在测试 one_equation() 方法时传入了四组参数。至于此处到底需要传入几组参数进行测试，关键取决于测试者要求达到怎样的逻辑覆盖程度，随着测试要求的提高，此处可能需要传入更多的测试参数。当然，此处只是介绍 PyUnit 的用法示例，并未刻意去达到怎样的逻辑覆盖，这一点请务必留意。

> 在测试某个方法时，如果实际测试要求达到某种覆盖程度，那么在编写测试用例时必须传入多组参数来进行测试，使得测试用例能达到指定的逻辑覆盖。

unittest.TestCase 内置了大量 assertXxx 方法来执行断言，其中最常用的断言方法如表 1 所示。

表 1 TestCase 中最常用的断言方法

断言方法	检查条件
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True

assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

除了上面这些断言方法，如果程序要对异常、错误、警告和日志进行断言判断，TestCase 提供了如表 2 所示的断言方法。

表 2 TestCase 包含的与异常、错误、警告和日志相关的断言方法

断言方法	检查条件
assertRaises(exc, fun, *args, **kwds)	fun(*args, **kwds) 引发 exc 异常
assertRaisesRegex(exc, r, fun, *args, **kwds)	fun(*args, **kwds) 引发 exc 异常，且异常信息匹配 r 正则表达式

assertWarns(warn, fun, *args, **kwds)	fun(*args, **kwds) 引发 warn 警告
assertWamsRegex(warn, r, fun, *args, **kwds)	fun(*args, **kwds) 引发 warn 警告 ,且警告信息匹配 r 正则表达式
assertLogs(logger, level)	With 语句块使用日志器生成 level 级别的日志

TestCase 还包含了如表 3 所示的断言方法用于完成某种特定检查。

表 3 TestCase 包含的用于完成某种特定检查的断言方法

断言方法	检查条件
assertAlmostEqual(a, b)	round(a-b, 7) == 0
assertNotAlmostEqual(a, b)	round(a-b, 7) != 0
assertGreater(a, b)	a > b
assertGreaterEqual(a, b)	a >= b
assertLess(a, b)	a < b
assertLessEqual(a, b)	a <= b
assertRegex(s, r)	r.search(s)
assertNotRegex(s, r)	not r.search(s)
assertCountEqual(a, b)	a、b 两个序列包含的元素相同，不管元素出现的顺序如何

当测试用例使用 assertEqual() 判断两个对象是否相等时，如果被判断的类型是字符串、序列、列表、元组、集合、字典，则程序会自动改为使用如表 4 所示的断言方法进行判断。换而言之，如表 4 所示的断言方法其实没有必要使用，unittest 模块会自动应用它们。

表 4 TestCase 包含的针对特定类型的断言方法

断言方法	用于比较的类型
assertMultiLineEqual(a, b)	字符串(string)
assertSequenceEqual(a, b)	序列(sequence)
assertListEqual(a, b)	列表(list)
assertTupleEqual(a, b)	元组(tuple)
assertSetEqual(a, b)	集合(set 或 frozenset)
assertDictEqual(a, b)	字典(dict)

运行测试

在编写完测试用例之后，可以使用如下两种方式来运行它们：

1. 通过代码调用测试用例。程序可以通过调用 unittest.main() 来运行当前源文件中的所有测试用例。例如，在上面的测试用例中增加如下代码：

```
if __name__ == '__main__':
    unittest.main()
```

2. 使用 unittest 模块运行测试用例。使用该模块的语法格式如下：

```
python -m unittest 测试文件
```

在使用 python -m unittest 命令运行测试用例时，如果没有指定测试用例，该命令将自动查找并运行当前目录下的所有测试用例。因此，程序也可直接使用如下命令来运行所有测试用例：

```
py -m unittest
```

采用上面任意一种方式来运行测试用例，均可以看到如下输出结果：

```
..
Ran 2 tests in 0.000s


OK
```

在上面输出结果的第一行可以看到两个点，这里的每个点都代表一个测试用例（每个以 test_ 开头的方法都是一个真正独立的测试用例）的结果。由于上面测试类中包含了两个测试用例，因此此处看到两个点，其中点代表测试用例通过。此处可能出现如下字符：

- .：代表测试通过。
- F：代表测试失败，F 代表 failure。
- E：代表测试出错，E 代表 error。

- s：代表跳过该测试，s 代表 skip。

在上面输出结果的横线下面看到了“Ran 2 tests in O.OOOs”提示信息，这行提示信息说明本次测试运行了多少个测试用例。如果看到下面提示 OK，则表明所有测试用例均通过。

上面的测试用例都可通过，是因为 fk_math.py 程序没有错误。如果将 fk_math.py 程序中的 ① 号代码故意修改为出错，假如将 ① 号代码修改为 return b/a，再次运行上面的测试用例，将会看到如下输出结果：

```
F.
======================================================================
==================
FAIL: test_one_equation (__main__.TestFkMath)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "E:\test_fk_math.py", line 24, in test_one_equation
    self.assertEqual(one_equation(5 , 9) , -1.8)
AssertionError: 1.8 != -1.8
----------------------------------------------------------------------
Ran 2 tests in 0.001s
```

```
FAILED (failures=1)
```

此时看到第一行的输出信息为 F，这表明第一个测试用例失败，第二个测试用例成功。

接下来在两条横线之间可以看到断言错误的 Traceback 信息，以及函数运行的实际输出结果和期望输出结果的差异。信息提示该函数运行的实际输出结果是 1.8，但期望输出结果是 -1.8。

7.Python TestSuite

测试包（TestSuite）可以组织多个测试用例，测试包还可以嵌套测试包。在使用测试包组织多个测试用例和测试包之后，程序可以使用测试运行器（TestRunner）来运行该测试包所包含的所有测试用例。

为了示范测试包的功能，下面再开发一个程序（hello.py）：

```
1.          # 该方法简单地返回字符串
2.          def say_hello():
3.              return "Hello World."
4.          # 计算两个整数的和
5.          def add(nA, nB):
6.              return nA + nB
```

接下来为上面程序提供如下测试类(test_hello.py)：

```
1.          import unittest
2.
3.          from hello import *
```

```
4.
5.          class TestHello(unittest.TestCase):
6.              # 测试 say_hello 函数
7.              def test_say_hello(self):
8.                  self.assertEqual(say_hello() , "Hello World.")
9.              # 测试 add 函数
10.             def test_add(self):
11.                 self.assertEqual(add(3, 4) , 7)
12.                 self.assertEqual(add(0, 4) , 4)
13.                 self.assertEqual(add(-3, 0) , -3)
```

将 test_fk_math.py 和 test_hello.py 文件放在同一目录，此时程序就可以通过 TestSuite 将它们组织在一起，然后使用 TestRunner 来运行该测试包。

```
1.          import unittest
2.          from test_fk_math import TestFkMath
3.          from test_hello import TestHello
4.
5.          test_cases = (TestHello, TestFkMath)
6.          def whole_suite():
7.              # 创建测试加载器
8.              loader = unittest.TestLoader()
9.              # 创建测试包
10.             suite = unittest.TestSuite()
11.             # 遍历所有测试类
12.             for test_class in test_cases:
13.                 # 从测试类中加载测试用例
14.                 tests = loader.loadTestsFromTestCase(test_class)
15.                 # 将测试用例添加到测试包中
16.                 suite.addTests(tests)
17.             return suite
18.         if __name__ == '__main__':
19.             # 创建测试运行器（TestRunner）
20.             runner = unittest.TextTestRunner(verbosity=2) //①
21.             runner.run(whole_suite())
```

上面程序调用 TestSuite 的 addTests() 方法来添加测试用例，这样就实现了使用 TestSuite 来组织多个测试用例。

上面程序还使用 TestLoader 来加载测试用例，该对象提供了一个 loadTestsFromTestCase() 方法，从指定类加载测试用例。

上面程序中的 ① 号代码创建了 TextTestRunner，它是一个测试运行器，专门用于运行测试用例和测试包。其实前面使用的 unittest.main() 函数，同样也是通过 TextTestRunner 来运行测试用例的。

程序中的 ① 号代码在创建 TextTestRunner 时还指定了 verbosity=2，这样可以生成更详细的测试信息。

> 在调用 unittest.main() 函数时同样可指定 verbosity=2，用来生成更详细的测试信息。

运行上面程序，可以看到生成如下测试报告：

```
test_add (test_hello.TestHello) ... ok
test_say_hello (test_hello.TestHello) ... ok
test_one_equation (test_fk_math.TestFkMath) ... FAIL
test_two_equation (test_fk_math.TestFkMath) ... ok

======================================================================
```

```
==================

FAIL: test_one_equation (test_fk_math.TestFkMath)

----------------------------------------------------------------------

Traceback (most recent call last):

  File "E:\BaiduNetdiskDownload\《疯狂 Python 讲义》光盘文件

\codes\16\16.5\test_fk_math.py", line 24, in test_one_equation

    self.assertEqual(one_equation(5 , 9) , -1.8)

AssertionError: 1.8 != -1.8



----------------------------------------------------------------------

Ran 4 tests in 0.017s



FAILED (failures=1)
```

从上面的运行结果可以看到，测试报告通过更详细的信息来提示每个测试用例的运行结果，此时同样可以看到 test_one_equation() 测试失败。

如果不希望仅能在控制台中看到测试报告，而是希望直接生成文件格式的测试报告，则可以在 ① 号代码创建 TextTestRunner 对象时指定 stream 属性，该属性是一个打开的类文件对象，这样程序就会把测试报告输出到该类文件对象中。

例如，将上面的 __main__ 部分改为如下形式：

```
if __name__ == '__main__':
    with open('fk_test_report.txt', 'a') as f:
        # 创建测试运行器（TestRunner），将测试报告输出到文件中
        runner = unittest.TextTestRunner(verbosity=2, stream=f)
        runner.run(whole_suite())
```

再次运行该程序，此时在控制台中将看不到任何信息，测试报告将会被输出到 fk_test_report.txt 文件中。

8.Python setUp 和 tearDown

单元测试 setUp 和 tearDown 用法详解

到目前为止，针对 unittest 已经介绍了测试用例类（TestCase 的子类）、测试包（TestSuite）和测试运行器（TestRunner）。此外，unittest 还有测试固件（Test Fixture）的概念：

- 测试用例类：测试用例类就是单个的测试单元，其负责检查特定输入和对应的输出是否匹配。unittest 提供了一个 TestCase 基类用于创建测试用例类。
- 测试包：用于组合多个测试用例，测试包也可以嵌套测试包。
- 测试运行器：负责组织、运行测试用例，并向用户呈现测试结果。
- 测试固件：代表执行一个或多个测试用例所需的准备工作，以及相关联的准备操作，准备工作可能包括创建临时数据库、创建目录、开启服务器进程等。

unittest.TestCase 包含了 setUp() 和 tearDown() 两个方法，其中 setUp() 方法用于初始化测试固件；而 tearDown() 方法用于销毁测试固件。程序会在运行每个测试用例（以 test_ 开头的方法）之前自动执行 setUp() 方法来初始化测试固件，并在每个测试用例（以 test_ 开头的方法）运行完成之后自动执行 tearDown() 方法来销毁测试固件。

由此可见，如果希望程序为测试用例初始化、销毁测试固件，那么只要重写 TestCase 的 setUp() 和 tearDown() 方法即可。例如如下测试程序（fixture_test.py）：

```
import unittest

from hello import *

class TestHello(unittest.TestCase):
    # 测试 say_hello 函数
    def test_say_hello(self):
        self.assertEqual(say_hello() , "Hello World.")
    # 测试 add 函数
    def test_add(self):
        self.assertEqual(add(3, 4) , 7)
        self.assertEqual(add(0, 4) , 4)
        self.assertEqual(add(-3, 0) , -3)
    def setUp(self):
        print('\n====执行 setUp 模拟初始化固件====')
    def tearDown(self):
        print('\n====调用 tearDown 模拟销毁固件====')
```

使用如下命令来运行该测试程序：

```
python -m unittest -v fixture_test.py
```

为该命令添加了 -v 选项，该选项用于告诉 unittest 生成更详细的输出信息。此时可以看到如下输出结果：

```
test_add (fixture_test1.TestHello) ...
====执行 setUp 模拟初始化固件====

====调用 tearDown 模拟销毁固件====
ok
test_say_hello (fixture_test1.TestHello) ...
====执行 setUp 模拟初始化固件====

====调用 tearDown 模拟销毁固件====
ok

----------------------------------------------------------------------
Ran 2 tests in 0.002s

OK
```

从上面的输出结果可以看出，unittest 在运行每个测试用例（以 test_ 开头的方法）之前都执行了 setUp() 方法，在每个测试用例（以 test_ 开头的方法）运行完成之后都执行了 tearDown() 方法。

如果希望程序在该类的所有测试用例执行之前都用一个方法来初始化测试固件，在该类的所有测试用例执行之后都用一个方法来销毁测试固件，则可通过重写 setUpClass() 和

tearDownClass() 类方法来实现。

例如如下测试程序（fixture_test1.py）：

```
import unittest

from hello import *

class TestHello(unittest.TestCase):
    # 测试 say_hello 函数
    def test_say_hello(self):
        self.assertEqual(say_hello() , "Hello World.")
    # 测试 add 函数
    def test_add(self):
        self.assertEqual(add(3, 4) , 7)
        self.assertEqual(add(0, 4) , 4)
        self.assertEqual(add(-3, 0) , -3)
    @classmethod
    def setUpClass(cls):
        print('\n====执行 setUpClass 在类级别模拟初始化固件====')
    @classmethod
    def tearDownClass(cls):
        print('\n====调用 tearDownClass 在类级别模拟销毁固件====')
```

上面程序中定义的 setUpClass() 和 tearDownClass() 两个类方法也是用于初始化测试固件和销毁测试固件的方法，但它们会在该类的所有测试用例执行之前和执行之后执行。

使用如下命令来运行该测试程序：

```
python -m unittest -v fixture_test1.py
```

可以看到如下输出结果：

```
====执行 setUpClass 在类级别模拟初始化固件====

test_add (fixture_test2.TestHello) ... ok

test_say_hello (fixture_test2.TestHello) ... ok



====调用 tearDownClass 在类级别模拟销毁固件====



----------------------------------------------------------------------

Ran 2 tests in 0.001s



OK
```

9.Python unittest 跳过测试用例

unittest 跳过测试用例（2 种方法，附带实例分析）

在默认情况下，unittest 会自动测试每一个测试用例（以 test 开头的方法），但如果希望临时跳过某个测试用例，则可以通过如下两种方式来实现：

1. 使用 skipXxx 装饰器来跳过测试用例。unittest 一共提供了 3 个装饰器，分别是 @unittest.skip(reason)、@unittest.skipIf(condition, reason) 和 @

unittest.skipUnless(condition, reason)。其中 skip 代表无条件跳过，skipIf 代表当 condition 为 True 时跳过；skipUnless 代表当 condition 为 False 时跳过。

2. 使用 TestCase 的 skipTest() 方法来跳过测试用例。

下面程序示范了使用 @unittest.skip 装饰器来跳过测试用例（skip_test.py）：

```
1.  import unittest
2.
3.  from hello import *
4.
5.  class TestHello(unittest.TestCase):
6.      # 测试 say_hello 函数
7.      def test_say_hello(self):
8.          self.assertEqual(say_hello() , "Hello World.")
9.      # 测试 add 函数
10.     @unittest.skip('临时跳过 test_add')
11.     def test_add(self):
12.         self.assertEqual(add(3, 4) , 7)
13.         self.assertEqual(add(0, 4) , 4)
14.         self.assertEqual(add(-3, 0) , -3)
```

第 10 行代码使用 @unittest.skip 装饰器跳过了 test_add() 测试方法。使用如下命令来运行该测试程序：

```
python -m unittest skip_test.py
```

可以看到如下输出结果：

```
s.
----------------------------------------------------------------------
Ran 2 tests in 0.000s


OK (skipped=1)
```

在上面输出结果的第一行可以看到 s.，这表明程序运行了两个测试用例，s 代表跳过了第一个测试用例，点（.）代表第二个测试用例通过。

此外，程序也可以使用 TestCase 的 skipTest() 方法跳过测试用例。例如，如下程序示范了使用 skipTest() 方法来跳过测试用例（skip_test1.py）：

```
1.          import unittest
2.
3.          from hello import *
4.
5.          class TestHello(unittest.TestCase):
6.              # 测试 say_hello 函数
7.              def test_say_hello(self):
8.                  self.assertEqual(say_hello() , "Hello World.")
9.              # 测试 add 函数
10.             def test_add(self):
11.                 self.skipTest('临时跳过 test_add')
12.                 self.assertEqual(add(3, 4) , 7)
13.                 self.assertEqual(add(0, 4) , 4)
14.                 self.assertEqual(add(-3, 0) , -3)
```

第 11 行代码使用 self.skipTest() 方法跳过了测试方法（test_add()）。使用如下命令来运行该测试程序：

```
python -m unittest -v skip_test1.py
```

上面命令使用了 -v 选项来生成更详细的测试报告。运行上面命令 ,可以看到如下输出结果：

```
test_add (skip_test2.TestHello) ... skipped '临时跳过 test_add'

test_say_hello (skip_test2.TestHello) ... ok



----------------------------------------------------------------------

Ran 2 tests in 0.000s



OK (skipped=1)
```

从上面的输出结果可以看到 ,unittest 测试跳过了 test_add() 方法 ,并显示了跳过的原因：'临时跳过 test_add'（如果不使用 -v 选项，将不会输出该原因）。

第 19 章 Python 打包和发布

1.Python 打包的发展历程

本章将介绍两个常用的 Python 打包和发布工具，分别是「zipapp」和「PyInstaller」。zipapp 模块主要用于将 Python 应用打包成一个 .pyz 文件。同时 ,无论开发 Python 应用时有多少源文件和依赖包，使用 zipapp 都可以将它们打包成一个 .pyz 文件，不足之处是该文件依然需要 Python 环境来执行。

PyInstaller 工具则更强大，它可以直接将 Python 程序打包成可执行程序，前该工具跨平台，使用非常方便。使用 PyInstaller 打包出来的程序，完全可以被分发到对应平台的的目标机器上直接运行，无须在目标机器上安装 Python 解释器环境。

Python 打包一开始可能有些难以理解，其主要原因是不了解创建 Python 包的正确工具。但不管怎样，一旦创建好第一个包，就会发现其实它并没有想象的那么难。

即便你对将代码开源分发不感兴趣，也应该知道如何创建一个完整的包，因为这可以让你深入了解打包生态系统，有助于使用 PyPI 上可用的第三方代码。此外，将闭源项目或者其组件编程源代码发行包，有助于你在不同的环境中部署代码。

Python 打包的混乱状态

Python 打包曾经在很长一段时间内处于混乱不堪的状态，人们花了很多年才使其重新变得有组织。

一切都从 1998 年引入的 distutils 包开始，随后在 2003 年对 setuptools 进行改进，这两个项目开启了一段漫长而又纠结的故事，包括派生（fork），替代项目与完全重新编写，都想要彻底修复 Python 的打包生态系统。

然而不幸的是，大部分尝试非但没有成功，每个想要取代 setuptools 或 distutils 的新项目都只是给打包工具十分混乱的状态添乱而已。其中，有些派生被合并回它们的祖先中（例如 setuptools 派生的 distribute），有些则直接被弃用（distutils2）。

这种混乱的状态直到成立了被称为 Python Packaging Authority（简称 PyPA）的组织，将秩序和组织性带回到打包生态系统中。PyPA 维护的 Python 打包用户指南（Python Packaging User Guide）是关于最新打包工具和最佳实践的权威信息来源。

这份指南包含详细的历史变化以及与打包相关的新项目，因此，如果你已经了解一些内容，但想要确保使用的是正确的工具，那么这份指南是很有用的。

Python 打包的现状

PyPA 除了提供一份权威的打包指南之外，还维护着打包项目与新的官方打包的标准化过程，前面章节中已经提到过其中一些项目，最有名的例如 pip、virtualenv、twine 和 warehouseo。

有兴趣的读者可参阅 PyPA 官方网站做详细了解。

注意，其中大部分项目都是在这个组织之外开始的，只是作为一个成熟且广泛使用的解决方案迁移到 PyPA 下而已。

由于 PyPA 的参与，构建发行版已经正在逐步弃用 egg 格式，而是支持使用 wheel 格式，未来可能会带来全新的方法。另外，PyPA 正在积极开发 warehouse，其目的是完全替代当前的 PyPI 实现，这将是打包历史上迈出的一大步，因为 PyPI 是如此古老且被忽视的项目。

PyPA 打包工具推荐

Python 打包用户指南有关使用包的推荐工具给出了一些建议。这些工具大体可分为两组，分别是用于安装包的工具和用于包的创建与分发的工具。

PyPA 推荐用于安装包的工具：

- 使用 pip 安装来自 PyPI 的包。
- 将 virtualenv 或 venv 用于 Python 环境的应用级隔离。

PyPA 推荐包的创建与分发的工具如下：

- 使用 setuptools 来定义项目并创建源代码发行版。
- 使用 wheel 而不是 egg 来构建发行版。
- 使用 twine 向 PyPI 上传包的发行版。

2.如何创建一个程序包？

我们知道，组织大型应用的代码的最简单方法，通常就是将其分成几个包，这使得代码更加简单，也更容易理解、维护和修改，同时还可以使每个包的可复用性最大化，它们的作用就像组件一样。

setup.py 脚本文件

对于一个需要被分发的包来说，其根目录包含一个 setup.py 脚本，它定义了 distutils 模块中描述的所有元数据，并将其合并为标准的 setup() 函数调用的参数。

> 虽然 distutils 是一个标准库模块，但建议读者使用 setuptools 包来代替，因为它对标准的 distutils 做了一些改进。

因此，这个文件的最少内容如下：

```
1.          from setuptools import setup
2.          setup(
3.              name='mypackage',
4.          )
```

其中，name 给出了包的全名。

另外，该脚本提供了一些命令，可以用 --help -commands 选项列出这些命令：

```
$Python setup.py --help-commands
tandard commands:
  build             build everything needed to install
  clean             clean up temporary files from 'build' command
  install           install everything from build directory
  sdist             create a source distribution (tarball, zip file)
  register          register the distribution with the PyP
  bdist             create a built (binary) distribution
```

```
  check             perform some checks on the package
  upload            upload binary package to PyPI

Extra commands:
  develop           install package in 'development mode'
  alias             define a shortcut to invoke one or more commands
  test              run unit t#sts after in-place build
  bdist_wheel       create a wheel distribution

usage: setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts】...]
  or: setup.py --help [cmd1 end2 ...]
  or: setup.py --help-commands
  or: setup.py cmd --help
```

实际的命令列表更长，而且会根据 setuptools 的可用扩展而变化。这里所列举的都是相对来说比较重要的，且和本节相关的命令。

Standard commands（标准命令）是 distutils 提供的内置命令，而 Extra commands(额外命令 则是由诸如 setuptools 这样的第三方包或任何其他定义并注册一个新命令的包所创建的。由另一个包注册的一个额外命令就是 wheel 包提供的 bdist_wheel。

setup.cfg 文件

setup.cfg 文件包含 setup.py 脚本命令的默认选项。如果构建和分发包的过程更加复杂，并且需要向 setup.py 命令中传入许多可选参数，那么这个文件非常有用。

读者可以按项目将这些默认参数保存在代码中，这将使整个分发流程独立于项目之外，也能够让包的构建方式与向用户和其他团队成员的分发方式变得透明。

setup.cfg 文件的语法与内置 configparser 模块提供的语法相同，因此它类似于常见的 Microsoft Windows INI 文件。下面是安装配置文件的示例，提供了 global、sdist 和 bdist_wheel 命令的默认值，代码如下：

```
[global]
quiet=1


[sdist]
formats=zip,tar


[bdist_wheel】
universal=1
```

这个配置示例可以确保源代码发行版总是以两种格式创建(ZIP 和 TAR),并且构建 wheel 发行版将被创建为通用 wheel (与 Python 版本无关)。此外，由于全局 quiet 开关，每个命令的大部分输出都将被阻止。

注意，这只是为了便于说明，默认阻止每个命令的输出可能并不是一个合理的选择。

MANIFEST.in

使用 sdist 命令构建发行版时，distutils 将浏览包的目录，查找需要包含在存档中的文件。distutils 将包含：

- py_modules、packages 和 scripts 选项隐含的所有 Python 源文件；
- ext_modules 选项列出的所有 C 源文件。

匹配 glob 模式 test/test*.py 的文件包括：README、README.txt、setup.py 和 setup.cfg。

此外，如果你的包是由 subversion 或 CVS 管理，那么 sdist 将浏览诸如 .svn 之类的文件夹，查找需要包含的文件。利用扩展也可以与其他版本控制系统集成。sdist 将构建一个 MANIFEST 文件，列出所有文件并将它们包含在存档中。

假设你不使用这些版本控制系统，并且需要包含更多的文件，那么在与 setup.py 相同的目录中，可以为 MANIFEST 文件定义一个名为 MANIFEST.in 的模板，在其中可以指定 sdist 要包含哪些文件。

这个模板的每一行都定义一条包含或排除规则，例如：

```
include HISTORY.txt

include README.txt

include CHANGES.txt

include CONTRIBUTORS.txt

include LICENSE

recursive-include *.txt *.py
```

MANIFEST.in 命令的完整列表可以在 distutils 官方文档中找到。

最重要的元数据

除了被分发包的名称和版本之外，setup 可以接受的最重要的参数包括：

- descriptions：包含描述包的几句话。
- long_description：包含完整说明，可以使用 reStructuredText 格式。
- keywords：定义包的关键字列表。
- authors：作者的姓名或组织。
- author_email：联系人电子邮件地址。
- url：项目的 URL。
- license：许可证（GPL、LGPL 等）
- packages :包中所有名称的列表 ,setuptools 提供了一个名为 find_packages 的小函数来计算它。
- namespace_packages：命令空间包的列表。

trove 分类器

PyPI 和 distutils 为应用程序分类提供了一种解决方案，就是使用一套 trove 分类器。所有 trove 分类器都形成一个树状结构，每个分类器都是字符 串形式，其中用 :: 字符串分隔每个命名空间。分类器列表在包定义中是作为 setup() 函数的 classifiers 参数。

下面是 PyPI 上某个项目的分类器列表示例（这里是 solrq 项目）：

```
        from setuptools import setup
        setup(
            name="solrq",
            # (...)
            classifiers=[
                'Development Status :: 4 - Beta',
                'Intended Audience :: Developers',
                'License :: OSI Approved :: BSD License',
                'Operating System :: OS Independent',
                'Programming Language : : Python',
                'Programming Language :: Python :: 2',
                'Programming Language :: Python :: 2.6',
                'Programming Language :: Python :: 2.7',
                'Programming Language :: Python :: 3',
                'Programming Language :: Python :: 3.2',
                'Programming Language :: Python :: 3.3',
                'Programming Language :: Python :: 3.4',
                'Programming Language :: Python :: Implementation :: PyPy',
                'Topic :: Internet :: WWW/HTTP :: Indexing/Search',
            ],
        )
```

它们在包定义中是完全可选的，但可以对 setup() 接口中可用的基本元数据提供有用的扩展。

此外，trove 分类器还可以提供支持的 Python 版本或系统、项目的开发阶段或发布代码所使用的许可证等信息。许多 PyPI 用户按类别对可用的包进行搜索和浏览，因此正确的分类可以让 Python 包找到目标客户。

trove 分类器在整个打包生态系统中发挥重要作用，不应该被忽略。没有一个组织来验证包的分类，所以我们有责任为自己的包提供正确的分类器，并且不要为整个包索引带来混乱。

在编写本教程时，PyPI 上共有 608 个可用的分类器，分为以下 9 类：

1. 开发状态（DevelopmentStatus）
2. 环境（Environment）
3. 框架（Framework）
4. 目标受众（IntendedAudience）
5. 许可证（License）
6. 自然语言（Natural Language）
7. 操作系统（Operating System）
8. 编程语言（Programming Language）
9. 话题（Topic）

由于不时会添加新的分类器，所以在阅读本教程时，这些数字可能会有所不同。当前可用的 trove 分类器的完整列表可以用 setup.py register --list-classifiers 命令来查看。

常见模式

对于没有经验的开发者来说，创建一个用于分发的包可能是一项乏味的任务。如果不考虑元数据可能在项目其他部分找到的事实，setuptools 或 distuitls 在 setup() 函数调用中接受的大多数元数据都可以手动输入，代码如下：

```
1.          from setuptools import setup
2.
3.          setup(
4.              name="myproject",
5.              version="0.0.1",
6.              description="mypackage project short description",
7.              long_description="""
8.                  Longer description of mypackage project possibly with some
    documentation and/or usage examples
9.              """
10.             install_requires=[
11.                 'dependency1',
12.                 'dependency2',
13.                 'etc1',
14.             ]
15.         )
```

这么做当然可行，但从长远来看很难维护，并且未来可能会出现错误和不一致。setuptools 和 distuitls 都不能从项目源代码中自动提取各种元数据信息，因此需要自己提供这些信息。

在 Python 社区中，有一些常见模式可以解决最常见的问题。例如依赖管理、包含版本/自述文件等。至少应该知道其中一些模式，因为它们非常流行，已经被看作一种打包惯例。

1）自动包含包中的版本字符串

PEP440（版本标识和依赖规范）文档规定了版本和依赖规范的标准。这是一份很长的文档，包含已接受的版本规范方案和 Python 打包工具中应该如何做版本匹配和比较。

如果你正在使用或打算使用一种复杂的项目版本编号方案，那么一定要阅读这份文档；如果你使用的是一种简单方案，其中包含用点分开的一个、两个、三个或更多的数字，那么可以不必阅读 PEP 440。

另一个问题是将包或模块的版本标识符包含在什么位置。PEP 396（模块版本号）正好解决这个问题。注意，这份文档只是信息性的，并且状态为延期，所以它并不是标准路径的一部分。不管怎样，它描述的内容现在似乎成了事实上的标准。

根据 PEP 396，如果一个包或模块要指定一个版本，那么应该将其包含在包的根目录（__init__.py）或模块文件的 __version__ 属性中。另一个事实上的标准是，也要将包括版本元组的 VERSION 属性包含其中，这有助于用户编写兼容代码，因为如果版本方案足够简单的话，这样的版本元组很容易比较。

因此，PyPI 上的很多包都进循这两个标准。它们的 __init__.py 文件包含如下所示的版本属性，如下所示：

```
1.          #用元组表示版本，可以简单比较
2.          VERSION = (0, 1, 1)
3.          #利用元组创建字符串，以避免出现不一致
4.          __version__ = ".".join([str(x) for x in VERSION])
```

延期的 PEP 396 的另一个建议是，在 distutils 的 setup() 函数中提供的版本应该从 __version__ 派生，反之亦然。Python 打包用户指南为单一来源的项目版本提供了多种模式，每一种都有自己的优点和局限性。

我个人最喜欢相当长的，并没有包含在 PyPA 的指南中，但它的优点是仅限制 setup.py 脚本的复杂度。这个样板假定，版本标识符由包的 __init__ 模块的 VERSION 属性给出，并且提取这一数据包含在 setup () 调用中。

下面是某个虚构的包的 setup.py 脚本中的片段，其中使用了以下这种方法：

```
from setuptools import setup
import os
def get_version(version_tuple):
    #additional handling of a,b,rc tags, this can
    #be simpler depending on your versioning scheme
    if not isinstance(version_tuple[-1], int):
        return '.'.join(
            map(str, version.tuple[:-1])
        )+ version.tuple[-1]
    return '.'.join(map(str, version_tuple))
#path to the packages __init__ module in project
#source tree
init = os.path.join(
    os.path.dirname(__file__), 'src', 'some_package','__init__.py'
)

version_line = list(
    filter(lambda l: l.startswith('VERSION'), open(init))
)[0]

```

```
        # VERSION is a tuple so we need to eval its line of code.
        # We could simply import it from the package but we
        # cannot be sure that this package is importable before
        # finishing its installation

        VERSION = get_version(eval (version.line.split('=') [-1]))

        setup(
            name='some-package',
            version=VERSION,
            #...
        )
```

2) README 文件

Python 包索引可以在 PyPI 门户的包页面中显示一个项目的 readme 或者 long_description 的值。你可以用 reStructuredText 标记来编写这个说明，它在上传时会转换为 HTML 格式。

不幸的是，目前 PyPI 上的文档标记只能使用 reStructuredText，这在短期内也不太可能改变。更有可能的是，如果 warehouse 项目完全取代了当前的 PyPI 实现，那么将会支持其他标记语言。但是，我们仍然不知道 warehouse 的最终发布时间。

但是，许多开发者想要使用不同的标记语言，原因有很多。最常见的选择是 Markdown。它是 GitHub 上默认的标记语言，目前大多数开源的 Python 开发都是在 GitHub 上。

因此，GitHub 和 Markdown 的粉丝通常要么忽略这个问题，要么就提供两份独立的文档文本。提供给 PyPI 的说明要么是项目 GitHub 页面上说明的简短版本，要么是在 PyPI 上

无法正常显示的普通的无格式 Markdown。

如果你想使用除了 rcStructurcdText 之外的标记语言来编写项目的 README，你仍然可以用可读的形式将它作为 PyPI 页面上的项目说明。诀窍是在将包上传到 Python 包索引时使用 pypandoc 包将你使用的其他脚本语言转换成 reStructuredText，同时准备 readme 文件的简单内容作为备用（fallback）也很重要，这样即使用户没有安装 pypandoc，安装也不会失败，代码如下：

```
        try:
            from pypandoc import convert
            def read_md(f):
                return convert(f, 'rst')
        except ImportError:
            convert = None
            print(
                "warning: pypandoc module not found, could not convert Markdown to RST"
            )

            def readjnd(f):
                return open(f, 'r').read() #noqa
        README = os.path.join (os.path.dirname(__file__), 'README.md')

        setup(
            name='some-package',
            long_description=read_md(README),
            #...
        )
```

3）管理依赖

许多项目需要安装和（或）使用一些外部包。如果依赖列表很长的话，就会出现一个问题，

即如何管理依赖？在大多数情况下答案很简单，不要过度设计问题。保持简单，并在 setup.py 脚本中明确提供依赖列表，代码如下：

```
from setuptools import setup
setup(
    name = 'some-package',
    install_requires=['falcon', 'requests', 'delorean']
    #...
)
```

有些 Python 开发者喜欢使用 requirements.txt 文件来追踪包的依赖列表。在某些情况下，你可能会找到这么做的原因，但在大多数情况下，这是项目代码没有正确打包的时代遗留的问题。

无论如何，即使像 Celery 这样著名的项目也仍然坚持使用这一约定。因此，如果你不愿意改变习惯或者不知何故被迫使用 requirements.txt 文件，那么至少要将其做对。下面是从 requirements.txt 文件读取依赖列表的常见做法之一：

纯文本复制

```
from setuptools import setup
import os
def strip_comments(l):
    return l.split ('#', 1)[0].strip()

def reqs(*f):
    return list(filter(None, [strip_comments(l) for l in open(os.path.join(os.getcwd(), *f)).readlines()]))

setup(
    name='some-package',
    install.requires = reqs('requirements.txt')
```

```
12.                 #...
13.             )
```

3.如何在开发期间安装测试 Python 程序包？

使用 setuptools 主要是用于构建并分发包，但是我们仍然需要知道如何使用它们直接从项目源代码安装包，因为在向 PyPI 提交包之前，我们需要测试一下自己的打包代码是否正常工作，而最简单的测试方法就是安装它。

如果你将坏的软件包发送到仓库中，那么需要增加版本号才能重新上传。因此，在最终发行版之前测试打包代码是否被正确打包，可以避免不必要的版本号增加，当然也可以节省时间。

此外，在同时处理多个相关的包时，使用 setuptools 直接从自己的源代码安装有时也是必要的。

Python 安装和卸载程序包

通过执行 `Python setup.py install` 命令（执行此命令时，所在目录需位于包目录下），可以将包安装到 Python 环境中。如果之前没有构建过的话，它会尝试构建包，然后将结果注入到 Python 树中。

如果提供了源代码发行版，那么可以在临时文件夹中将其解压，然后用这个命令安装。除此之外，install 命令还将安装在 install_requires 元数据中定义的依赖，这是通过查看

Python 包索引上的包来完成的。

除了使用 setup.py 脚本安装包，还可以使用 pip 命令，它是 PyPA 推荐的工具，即使在本地环境为了开发而安装时，也可以使用它。

使用 pip 命令从本地源代码中安装一个包，可以运行下面这个命令：

```
pip install 包路径
```

值得一提的是，setuptools 和 distutils 都没有 uninstall (卸载) 命令，而使用 pip 可以卸载任何 Python 包，命令格式如下：

```
pip uninstall 包名
```

注意，在系统级的包上尝试卸载操作可能很危险，这也是为什么使用虚拟环境进行开发很重要的另一个原因。

Python 开发过程中安装程序包

使用 setup.py 安装的包会被复制到当前环境的 site-packages 目录下，这意味着，一旦我们修改了包的源代码，就需要重新安装它。

这常常是集中开发过程中的一个问题，因为很容易忘记需要再次执行安装，这就是为什么

setuptools 提供了一个额外的 develop 命令的重要原因。

develop 命令的完整格式为 `Python setup.py develop`，该命令允许我们在开发模式下安装包，它会在部署目录（site-packages）时创建一个指向项目源代码的特殊链接，而不是将整个包复制过去，从而实现在编辑包的源代码之后无需再重新安装，并且它在 sys.path 中可用，就像正常安装一样。

pip 也支持用这种模式来安装包，这个安装选项叫作可编辑模式，可 以使用 install 命令的 -e 参数来启用，代码格式如下：

```
pip install -e 包路径
```

4.Python 命名空间包

提到命名空间，其实包含 2 层含义，一种是语言上下文的命名空间，其实在编写 Python 程序时，无时无刻都在使用命名空间，比如模块使用的全局命名空间、函数或方法调用的本地命名空间等；另一种命名空间可以在包的层面提供，称为命名空间包。

简单的理解，命名空间包是对相关的包或模块进行分组的一种方法，通常是一个被忽略的功能，它对于在较大的项目中构建打包生态系统非常有用。

如果你的应用组件的开发、打包和版本化都是独立的，但仍然希望从同一个命名空间访问它

们，那么命名空间包特别有用，它有利于明确每个包所属的组织或项目。

例如，假设有一个 Acme 公司，该公司中使用共同的 acme 命名空间，同时创建通用的 acme 命名空间包作为该组织的其他包的容器。如果 Acme 公司中的某人想要向这个命名空间贡献一个与 SQL 相关的库，那么他需要在 acme 中注册自己新的 acme.sql 包，整个文件结构如下所示：

```
$tree acme/
acme/
├──────acme
|       ├──────__init__.py
|       └──────sql
|               └────__init__.py
└──────setup.py

2 directories, 3 files
```

在此基础上，如果想添加一个新的子包，例如添加 templating，则需要将其包含在 acme 的源代码树中，如下所示：

```
$tree acme/
acme/
├──────acme
|       ├──────__init__.py
|       ├──────sql
|       |       └────__init__.py
|       └──────templating
```

```
|               └──── __init__.py
└──────setup.py


3 directories, 4 files
```

仔细观察就会发现，采用这种方式几乎不可能单独开发 acme.sql 和 acme.templating。且 setup.py 脚本还必须指定每个子包的所有依赖，所以不可能（至少非常困难）选择性地安装 acme 中的部分组件。此外，如果某些子包的需求文件有冲突，是一个无法解决的问题。

通过利用命名空间包，我们可以单独保存每个子包的源代码树，如下所示：

```
$tree acme.sql/
acme.sql/
├──────acme
|       └──────sql
|               └──── __init__.py
└──────setup.py


2 directories, 2 files


$tree acme.templating/
acme.templating/
├──────acme
|       └──────templating
|                   └──── __init__.py
└──────setup.py
```

```
2 directories, 2 files
```

由此，就可以在 PyPI 或者使用的任何包索引中单独注册它们，用户还可以从 acme 命名空间中选择想要安装的子包，而无需安装通用的 acme 包，执行代码如下：

```
$pip install acme.sql acme.templating
```

注意，独立的源代码树不足以在 Python 中创建命名空间包，如果不想让包之间相互覆盖，就需要做一些额外的工作，此外，正确的处理方式也会随着 Python 版本的不同而有所不同。

隐式命名空间包

如果你只使用 Python 3.x，也只面向 Python 3.x 的用户，则可以使用 PEP 420 引入的定义命令空间的新方法，即隐式命名空间包。它是标准路径的一部分，并从 Python 3.3 版本开始成为语言官方内容的一部分。

简单来说，对于每一个包含 Python 包或模块（也包括命名空间包）的目录来说，如果其不包含 __init__().py 文件，那么它就被看做是命名空间包。

例如，前面所说的 acme 在 Python 3.3 以及更高版本中，就是一个命名空间包。使用安装工具的最小 setup.py 脚本文件如下所示：

```
1.          from setuptools import setup
2.
3.          setup(
4.              name = 'acme.templating',
```

```
5.                 packages = ['acme.templating'],
6.             )
```

但是，直到发表本节时，setuptools.find_packages() 还不支持 PEP 420，但这在未来很可能会改变。此外，要想实现命名空间包的简单继承，显示地定义包列表是值得的。

以前 Python 版本中的命名空间包

Python 3.3 之前的版本中，虽无法使用 PEP 420 布局中的命名空间包，但仍可以使用它。旧版 Python 中，有几种方法可以将包定义成命名空间。

最简单的方法就是为每个组件创建一个文件结构，类似于没有命名空间包的普通包布局，并将所有事情都留给 setuptools。

因此，acme.sql 和 acme.templating 的布局示例可能如下所示：

```
$tree acme.sql/
acme.sql/
├──────acme
|       ├──────__init__.py
|       └──────sql
|                └──────__init__.py
└──────setup.py

2 directories, 3 files

$tree acme.templating/
acme.templating/
```

```
├──────acme
|       ├──────__init__.py
|       └──────templating
|                  └──────__init__.py
└──────setup.py


2 directories, 3 files
```

注意，acme.sql 和 acme.templating 都有一个额外的源代码文件 acme/__init__.py，这个文件必须是空的。

如果我们提供 acme 作为 setuptools.setup() 函数 namespace_package 关键字参数的值，那么将会创建如下的 acme 命名空间包：

```
1.          from setuptools import setup
2.
3.          setup(
4.              name = 'acme.templating',
5.              packages = ['acme.templating'],
6.              namespace_package = ['acme'],
7.          )
```

当然，最简单的方法不一定是最好的，为了注册一个新的命名空间，setuptools 将会在 __init__.py 文件中调用 pkg_resources.declare_namespace() 函数，即便 __init__.py 文件是空的也会调用。

无论如何，正如官方文档所说，你自己负责在 __init__.py 文件中声明命名空间，并且未来可能会删除 setuptools 的这个隐式行为。为了保证安全，也为了未来依然可用

（future-proof），需要将下面这行代码添加到 acme/__init__.py 文件中：

```
__import__('pkg_resources').declare_namespace(__name__)
```

5.如何向 PyPI 上传 Python 程序包？

对于 Python 包而言，如果没有有组织的保存、上传和下载方式，那么它是没有用的。Python 包索引（一个搜索下载包的平台）是 Python 社区开源包的主要来源，任何人都可以免费上传新的包，唯一的要求就是在 PyPI 网站上进行注册。

当然，我们不必局限于这个索引，而且所有打包工具都支持使用其他包仓库。对于在内部组织分发或为了开发目的而分发的闭源代码来说，这一点特别有用。

本节，主要解决如何向 PyPI 上传程序包，同时还会简单介绍一下指定其他仓库的方法。

如前所述，PyPI 是开源包发行版的官方来源。从 PyPI 下载不需要任何账号或者权限，唯一需要的是一个包管理器，可以从 PyPI 下载新的发行版，而首选应该是 pip。

上传到 PyPI 或其他包索引

任何人都可以注册并向 PyPI 上传包，只要有注册账号就行。包与用户绑定，默认情况下只有注册了包名称的用户是它的管理员，并且可以上传新的发行版。但对于大型项目来说，有一个选项可以指定其他用户作为包的维护者，以便他们能够上传新的发行版。

上传包的最简单方法就是使用 setup.py 脚本的 upload 命令：

```
$ python setup.py <dist-commands> upload
```

这里的 <dist-commands> 是创建要上传的发行版的命令列表，只有在相同的 setup.py 执行期间创建的发行版才会被上传到仓库中。

因此，如果想要同时上传源代码发行版、构建发行版和 wheel 包，可以使用下列命令：

```
$python setup.py sdist bdist bdist_wheel upload
```

注意，使用 setup uplod 的方式有以下 2 个问题：

1. 使用 setup.py 进行上传时，不能重复使用已经构建的发行版，每次上传时都必须重新构建，这对于大型项目或复杂项目来说很不方便，因为创建发行版需要相当长的时间。
2. 在某些 Python 版本中 ,它可以使用纯文本 HTTP 连接或未验证的 HTTPS 连接。

这里推荐大家使用 twine 作为 setup.py upload 的安全替代。twine 是与 PyPI 交互的实用程序，目前只有一个作用，就是将包安全地上传到仓库中。twine 支持任何打包格式，并始终确保连接安全，它还允许上传已经创建的文件，这样能够在发布之前对发行版进行测试。

twine 的一个示例用法仍然需要调用 setup.py 来构建发行版，如下所示：

```
$ python setup.py sdist bdist_wheel

$ twine upload diat/*
```

注意，在上传之前一定要注册账号，否则上传会失败。当然，也可以使用 twine 来完成注册，如下所示：

```
$ twine register dist/*
```

.pypirc 配置文件

.pypirc 是一个配置文件，其中保存有关 Python 包仓库的信息，应位于包的主目录中。该文件的格式如下所示：

```
[distutils]

index-servers =

pypi

other



[pypi]

repository: <repository-url>

username: <username>

password: <password>



[other]
```

```
repository: https://example.com/pypi

username: <username>

password: <password>
```

其中，distutils 字段应该包含 index-servers 变量，其中列出描述所有可用仓库及其证书的所有区段。每个仓库区段中只能修改下面这 3 个变量：

1. repository：包仓库的 URL，默认是 https://www.python.org/pypi；
2. username：给定仓库中授权的用户名；
3. password：明文的授权用户密码。

注意，明文保存仓库密码可能并不安全，我们可以一直将其空这，必要时再提示输入。

所有为 Python 构建的打包工具都应该遵守 .pypirc 文件。虽然不是每个打包相关的实用程序都满足这一要求，但大多数重要的实用程序都支持这一点，例如 pip、twine、distutils 和 setuptools。

6.Python 源码包和二进制包

通常来说，Python 程序包有两种类型的发行版，分别是源代码发行版和二进制（构建）发行版。

源代码发行版是最简单的，也是最不依赖于平台的，对于纯 Python 包，无需动脑选择它就行，这种发行版只包含 Python 源代码，具有高度的可移植性。

但当包中引入了用其他语言（例如 C 语言）编写的一些扩展，如果包用户的环境中有合适的开发工具链（主要包括编译器和正确的 C 头文件）的话，那么源代码发行版也是可行的。相比较而言，二进制发行版的格式可能更适合，因为它可以为特定平台提供已经构建好的扩展。

sdist 发行版

sdist 命令是最简单的命令，它创建一棵分发树，其中复制了运行一个包所需要的全部内容，这棵树被归档到一个或多个存档文件中（通常只创建一个 tar 文件），这个存档基本上是源代码树的副本。

这个命令是从目标系统独立地分发一个包的最简单方法，它将创建一个 dist 文件夹，里面包含可被分发的存档。为了能够使用它，必须向 setup 传递一个额外参数以提供版本号，如果没有提供 version 值，那它默认为 version = 0.0.0，代码如下：

```
1.          from setuptools import setup
2.          setup(
3.              name='acme.sql',
4.              version='0.1.1'
5.          )
```

这个版本号在升级安装时非常有用，因为每次发布包时版本号都会增加，这样目标系统就知道它发生了变化。

我们运行带有这个额外参数的 sdist 命令，代码如下：

```
$ python setup.py sdist

running sdist

...

creating dist

tar -cf dist/acme.sql-0.1.1.tar acme.sql-0.1.1

gzip -f9 dist/acme.sql-0.1.1.tar

removing 'acme.sql-0.1.1' (and everything under it)

$ ls dist/

acme.sql-0.1.1.tar.gz
```

在 Windows 中，存档其实就是一个 ZIP 文档。

版本被用于标记存档名称，这个存档可以在任何拥有 Python 的系统上分发并安装。在 sdist 发行版中，如果包里面包含 C 库或扩展，那么目标系统将负责编译它们。

这在基于 Linux 的系统或 Mac OS 中很常见，因为这些系统通常都会提供编译器，但在 Windows 下却并不常见。因此，如果一个包打算在多个平台中运行，那么分发时应该同时提供预构建的发行版。

bdist 和 wheels 发行版

为了能够分发预构建的发行版，distutils 提供了 build 命令，可以通过 4 个步骤来编译包，分别是：

1. build_py：通过字节编译并将其复制到构建文件夹中，来构建纯 Python 模块。

2. build_clib：如果包中包含任何 C 库，它会利用 C 编译器在构建文件夹中创建一个静态库来构建 C 库。
3. build_ext：构建 C 扩展，并像 build_clib 一样将结果放在构建文件夹中。
4. build_scripts：构建被标记为脚本的模块。如果第一行被设为 !# 的话，它还会修改解释器路径并修改文件模式使其变为可执行文件。

上面每个步骤都是可以被单独调用的命令，编译过程的结果是构建一个文件夹，里面包含要安装的包所需要的全部内容。

需要注意的是，distutils 包中还没有提供交叉编译器的选项，也就是说，这些命令的结果总是针对构建时所使用的操作系统。

如果必须创建一些 C 扩展，构建过程将使用系统编译器和 Python 头文件（Python.h）。Python 从源代码构建完成之后这个包含（include）文件就是可用的了。对于打包的发行版，可能需要针对系统发行版的额外包，至少在流行的 Linux 发行版中，它通常被命名为 python-dev，其包含构建 Python 扩展所有必要的头文件。

所使用的 C 编译器是系统编译器，对于基于 Linux 系统或 Mac OS X 而言，它分别是 gcc 或 clang。对于 Windows 而言，可以使用 Microsoft Visual C++，也可以使用开源项目 MinGW，可以在 distutils 中进行相应的配置。

bdist 命令使用 build 命令来构建二进制发行版。它调用 build 和所有依赖的命令，然后用和 sdist 相同的方式创建一份存档。

我们在 Mac OS X 系统中为 acme.sql 创建一个二进制发行版，如下所示：

```
$ python setup.py bdist

running bdist

running bdist_dumb

running build

...

running insta1l_scripts

tar -cf dist/acme.sql-0.1.1.macosx-10.3-fat.tar

gzip -f9 acme.sql-0.1.1.macosx-10.3-fat.tar

removing 'build/bdist.macosx-10.3-fat/dumb' (and everything under it)



$ ls dist/

acme.sql-0.1.1.macosx-10.3-fat.tar.gz acme.sql-0.1.1.tar.gz
```

注意，新创建的存档名称中包含系统名称及其发行版本（Mac OS X 10.3）。

在 Windows 中，调用相同的命令将会创建一个特定的发行版存档，如下所示：

```
C:\acme.sql> python.exe setup.py bdist

...

C:\acme.sql> dir dist

25/02/2008 08:18 <DIR>          .

25/02/2008 08:18 <DIR>          ..
```

```
25/02/2008 08:24            16 055 acme.sql-0.1.Win32.zip
               1 File(s)             16 055 bytes
               2 Dir(s)      22 239 752 192 bytes free
```

如果一个包里包含 C 代码，那么除了源代码发行版之外，发布尽可能多的不同的二进制发行版也很重要。至少对于那些没存安装 C 编译器的人是很重要的。

二进制版本中包含一棵可以汽接复制到 Python 树中的树，它主要包含一个文件夹，将被复制到 Python 的 site-packages 文件夹中。同时，它还可能包含缓存字节码文件（在 Python 2.x 中是 *.pyc 文件，在 Python 3.x 中是 __pycache__/*.pyc）。

另一种构建发行版是 wheel 包提供的“wheel”，安装完 wheel 后（例如使用 pip），它会向 distutils 中添加一个新的 bdist_wheel 命令。wheel 允许创建特定平台的发行版（目前仅适用于 Windows 和 Mac OS X），作为普通 bdist 发行版的替代。

设计 wheel 是为了替代早先 setuptools 引入的另一种 egg 版本，egg 现在已经过时了，所以这里不再详细介绍它。

使用 wheel 的优点相当多，在 Python Wheels 中提到的优点如下所示：

- 更快速地安装纯 Python 包和本地 C 扩展包；
- 避免安装任意代码执行（避免 setup.py）；
- 安装 C 扩展不需要 Windows 或 OS X 上的编译器。
- 允许更好的缓存，用于测试和持续集成。
- 创建 .pyc 文件作为安装的一部分，以确保它们匹配所使用的 Python 解释器；

- 在跨平台和跨机器上更一致的安装。

根据 PyPA 的推荐，wheel 应该成为默认的分发格式。但 Linux 平台特定的 wheel 还不可用，因此如果必须分发带有 C 扩展的包，那么需要为 Linux 用户创建 sdist 发行版。

7.Python zipapp 打包

经过复杂的开发、调试之后，终于得到一个 Python 程序，这个程序或许精巧，或许有些古拙，但它是我们心血的结晶，我们当然希望将这个程序发布出来。

Python 提供了一个 zipapp 模块，通过该模块可以将一个 Python 模块（可能包含很多个源程序）打包成一个 Python 应用，甚至发布成一个 Windows 的可执行程序。

生成可执行的 Python 档案包

zipapp 是一个可以直接运行的模块，该模块用于将单个 Python 文件或整个目录下的所有文件打包成可执行的档案包。

zipapp 模块的命令行语法如下：

```
python -m zipapp source [options]
```

在上面命令中，source 参数代表要打包的 Python 源程序或目录，该参数既可以是单个的 Python 文件，也可以是文件夹。如果 source 参数是文件夹，那么 zipapp 模块会打包该文件夹中的所有 Python 文件。

该命令的 options 支持如下选项：

- -o <output>，--output=<output>：应选项指定输出档案包的文件名。如果不指定该选项，所生成的档案包的文件名默认为 source 参数值，并加上 .pyz 后缀。
- -p <interpreter>，--python=<interpreter>：改选项用于指定 Python 解释器。
- -m <mainfn>，--main=<mainfn>：该选项用于指定 Python 程序的入口函数。该选项应该为 pkg.mod:fn 形式，其中 pkg.mod 是一个档案包中的包或模块，fn 是指定模块中的函数。如果不指定该选项，则默认从模块中的 __main__.py 文件开始执行。
- -c，--compress：从 Python 3.7 开始支持该选项。该选项用于指定是否对档案包进行压缩来减小文件的大小，默认是不压缩。
- --info：该选项用于在诊断时显示档案包中的解释器。
- -h，--help：该选项用于显示 zipapp 模块的帮助信息。

下面在某目录下建立一个 app 子目录，该子目录用于包含多个 Python 程序。首先在该目录下开发一个 say_hello.py 程序：

```
1.          def say_hello(name):
2.              return name + ", 您好！"
```

然后在该目录下开发一个 app.py 程序来使用 say_hello 模块：

```
1.          from say_hello import *
2.
3.          def main():
4.              print('程序开始执行')
```

```
5.                 print(say_hello('孙悟空'))
```

在命令行工具中进入该目录（app 目录的父目录），然后执行如下命令：

```
python -m zipapp app -o first.pyz -m "app:main"
```

上面命令指定将当前目录下的 app 子目录下的所有 Python 源文件打包成一个档案包，并通过 -o 选项指定所生成的档案包的文件名为 first.pyz；-m 选项指定使用 app.py 模块中的 main 函数作为程序入口。

运行上面命令，将会生成一个 first.pyz 文件。接下来可以使用 python 命令来运行 first.pyz 文件：

```
python first.pyz
程序开始执行
孙悟空，您好！
```

通过命令行工具在 app 目录的父目录下执行如下命令：

```
python -m zipapp app -m "app:main"
```

上面命令没有指定 -o 选项，这意味着该命令将会使用默认的输出文件名：source 参数值加 .pyz 后缀。运行上面命令，将会在当前目录下生成一个 app.pyz 文件。

zipapp 创建独立应用

通过上面介绍的方式打包得到的档案包中只有当前项目的 Python 文件，如果 Python 应

用还需要使用第三方模块和包（比如前面介绍的需要连接 MySQL 的应用），那么仅打包该应用的 Python 程序是不够的。

为了创建能独立启动的应用（自带依赖模块和包），需要执行两步操作：

1. 将应用依赖的模块和包下载到应用目录中。
2. 使用 zipapp 将应用和依赖模块一起打包成档案包。

下面在 app 所在目录下再创建一个 dbapp 子目录，该子目录将会作为本应用的目录。接下来在 dbapp 目录下新建一个 __main__.py 文件作为程序入口，这样程序在打包档案包时就不需要指定程序入口了。

下面是 __main__.py 文件的代码：

```
from exec_select import *

# 执行 query_db()函数
query_db()
```

其中，exec_select.py 文件需要自己添加，包含代码如下：

```
# 导入访问 MySQL 的模块
import mysql.connector

def query_db():
    # ①、连接数据库
    conn = conn = mysql.connector.connect(user='root', password='32147',
        host='localhost', port='3306',
```

```
                database='python', use_unicode=True)
            # ②、获取游标
            c = conn.cursor()
            # ③、调用执行 select 语句查询数据
            c.execute('select * from user_tb where user_id > %s', (2,))
            # 通过游标的 description 属性获取列信息
            for col in (c.description):
                print(col[0], end='\t')
            print('\n------------------------------')
            # 直接使用 for 循环来遍历游标中的结果集
            for row in c:
                print(row)
                print(row[1] + '-->' + row[2])
            # ④、关闭游标
            c.close()
            # ⑤、关闭连接
            conn.close()
```

最后按照如下步骤将 dbapp 子目录下的应用打包成独立应用：

1. 通过命令行工具在 dbapp 当前所在目录执行如下命令：

```
python -m pip install -r requirements.txt --target dbapp
```

上面命令实际上就是使用 pip 模块来安装模块，其中 python -m pip install 表示要安装模块。--target 选项指定将模块安装到指定目录下，此处指定将依赖模块安装到 dbapp 子目录下。-r 选项指定要安装哪些模块，此处使用 requirements.txt 文件列出要安装的模块和包。-r 选项支持两个值：

- 直接指定要安装的模块或包。
- 使用清单文件指定要安装的模块和包。

当应用依赖的模块较多时，建议使用清单文件来列出所依赖的模块。

如果直接运行上面命令，pip 模块会提示找不到 requirements.txt 文件，因此需要在当前目录下添加一个 requirements.txt 文件，并在该文件中增加如下一行。

```
mysql-connector-python
```

如果项目需要依赖多个模块，则可以在 requirements.txt 文件中定义多行，每行定义一个模块。

重新运行上面命令，将可以看到 pip 开始下载 mysql-connector-python 模块，下载完成后将可以在 dbapp 子目录下看到大量有关 mysql-connector-python 模块的文件。

2. 如果 pip 在 dbapp 子目录下生成了 .dist-info 目录，则建议删除该目录。
3. 使用 zipapp 模块执行打包操作。由于本例的 dbapp 子目录下包含了 __main__.py 文件，该文件将会作为程序入口，因此打包时不需要指定 -m 选项。使用如下命令来打包：

```
python -m zipapp dbapp
```

与上一节所使用的命令相比，该命令没有使用 -m 边项来指定程序入口，该程序将会使用档案包中的 __main__.py 文件作为程序入口。运行上面命令，将会得到一个大约为 18MB 的档案包。因为该档案包自包含了 mysql-connector-python 模块，所以比较大。

在创建了独立应用之后，只要目标机器上安装了合适版本的 Python 解释器，即可运行该独立应用。我们可以先使用如下命令卸载在 Python 目录下安装的 mysql-connector-python 模块：

```
pip uninstall mysql-connector-python
```

此时在本机的 Python 目录下不再包含 mysql-connector-python 模块，但 dbapp.pyz 程序依然可以正常运行，因为它自包含了 mysql-connector-python 模块。

8.Python PyInstaller

在创建了独立应用（自包含该应用的依赖包）之后，还可以使用 PyInstaller 将 Python 程序生成可直接运行的程序，这个程序就可以被分发到对应的 Windows 或 Mac OS X 平台上运行。

安装 PyInstalle

Python 默认并不包含 PyInstaller 模块，因此需要自行安装 PyInstaller 模块。

安装 PyInstaller 模块与安装其他 Python 模块一样，使用 pip 命令安装即可。在命令行输入如下命令：

```
pip install pyinstaller
```

强烈建议使用 pip 在线安装的方式来安装 PyInstaller 模块，不要使用离线包的方式来安装，因为 PyInstaller 模块还依赖其他模块，pip 在安装 PyInstaller 模块时会先安装它的依赖模块。

运行上面命令，应该看到如下输出结果：

```
Successfully installed pyinstaller-x.x.x
```

其中的 x.x.x 代表 PyInstaller 的版本。

在 PyInstaller 模块安装成功之后，在 Python 的安装目录下的 Scripts(D:\Python\Python36\Scripts) 目录下会增加一个 pyinstaller.exe 程序，接下来就可以使用该工具将 Python 程序生成 EXE 程序了。

PyInstaller 生成可执行程序

PyInstaller 工具的命令语法如下：

```
pyinstaller 选项 Python 源文件
```

不管这个 Python 应用是单文件的应用，还是多文件的应用，只要在使用 pyinstaller 命令时编译作为程序入口的 Python 程序即可。

PyInstaller 工具是跨平台的，它既可以在 Windows 平台上使用，也可以在 Mac OS X 平台上运行。在不同的平台上使用 PyInstaller 工具的方法是一样的，它们支持的选项也是一样的。

下面先创建一个 app 目录，在该目录下创建一个 app.py 文件，文件中包含如下代码：

```
from say_hello import *

def main():
    print('程序开始执行')
    print(say_hello('孙悟空'))
# 增加调用 main()函数
if __name__ == '__main__':
    main()
```

接下来使用命令行工具进入到此 app 目录下，执行如下命令：

```
pyinstaller -F app.py
```

执行上面命令，将看到详细的生成过程。当生成完成后，将会在此 app 目录下看到多了一个 dist 目录，并在该目录下看到有一个 app.exe 文件，这就是使用 PyInstaller 工具生成的 EXE 程序。

在命令行窗口中进入 dist 目录下，在该目录执行 app.exe ，将会看到该程序生成如下输出结果：

```
程序开始执行
孙悟空，您好！
```

由于该程序没有图形用户界面，因此如果读者试图通过双击来运行该程序，则只能看到程序窗口一闪就消失了，这样将无法看到该程序的输出结果。

在上面命令中使用了-F 选项，该选项指定生成单独的 EXE 文件，因此，在 dist 目录下生成了一个单独的大约为 6MB 的 app.exe 文件（在 Mac OS X 平台上生成的文件就叫 app，没有后缀）；与 -F 选项对应的是 -D 选项（默认选项），该选项指定生成一个目录（包含多个文件）来作为程序。

下面先将 PyInstaller 工具在 app 目录下生成的 build、dist 目录删除，并将 app.spec 文件也删除，然后使用如下命令来生成 EXE 文件。

```
pyinstaller -D app.py
```

执行上面命令，将看到详细的生成过程。当生成完成后，将会在 app 目录下看到多了一个 dist 目录，并在该目录下看到有一个 app 子目录，在该子目录下包含了大量 .dll 文件和 .pyz 文件，它们都是 app.exe 程序的支撑文件。在命令行窗口中运行该 app.exe 程序，同样可以看到与前一个 app.exe 程序相同的输出结果。

PyInstaller 不仅支持 -F、-D 选项，而且也支持如表 1 所示的常用选项。

表 1 PyInstaller 支持的常用选项

-h，--help	查看该模块的帮助信息
-F，-onefile	产生单个的可执行文件
-D，--onedir	产生一个目录（包含多个文件）作为可执行程序
-a，--ascii	不包含 Unicode 字符集支持
-d，--debug	产生 debug 版本的可执行文件
-w ， --windowed ，--noconsolc	指定程序运行时不显示命令行窗口（仅对 Windows 有效）
-c ，--nowindowed ，--console	指定使用命令行窗口运行程序（仅对 Windows 有效）
-o DIR，--out=DIR	指定 spec 文件的生成目录。如果没有指定，则默认使用当前目录来生成 spec 文件
-p DIR，--path=DIR	设置 Python 导入模块的路径（和设置 PYTHONPATH 环境变量的作用相似）。也可使用路径分隔符（Windows 使用分号，Linux 使用冒号）来分隔多个路径
-n NAME ，--name=NAME	指定项目（产生的 spec）名字。如果省略该选项，那么第一个脚本的主文件名将作为 spec 的名字

在表 1 中列出的只是 PyInstaller 模块所支持的常用选项，如果需要了解 PyInstaller 选项的详细信息，则可通过 pyinstaller -h 来查看。

下面再创建一个带图形用户界面，可以访问 MySQL 数据库的应用程序。

在 app 当前所在目录再创建一个 dbapp 目录，并在该目录下创建 Python 程序，其中 exec_select.py 程序负责查询数据 ,main.py 程序负责创建图形用户界面来显示查询结果。

exec_select.py 文件包含的代码如下：

```
1.          # 导入访问 MySQL 的模块
2.          import mysql.connector
3.
4.          def query_db():
5.              # ①、连接数据库
6.              conn = conn = mysql.connector.connect(user='root', password='32147',
7.                  host='localhost', port='3306',
8.                  database='python', use_unicode=True)
9.              # ②、获取游标
10.             c = conn.cursor()
11.             # ③、调用执行 select 语句查询数据
12.             c.execute('select * from user_tb where user_id > %s', (2,))
13.             # 通过游标的 description 属性获取列信息
14.             description = c.description
15.             # 使用 fetchall 获取游标中的所有结果集
16.             rows = c.fetchall()
17.             # ④、关闭游标
18.             c.close()
19.             # ⑤、关闭连接
```

```
            conn.close()
            return description, rows
```

mian.py 文件包含的代码如下：

```
from exec_select import *
from tkinter import *

def main():
    description, rows = query_db()
    # 创建窗口
    win = Tk()
    win.title('数据库查询')
    # 通过 description 获取列信息
    for i, col in enumerate(description):
        lb = Button(win, text=col[0], padx=50, pady=6)
        lb.grid(row=0, column=i)
    # 直接使用 for 循环查询得到的结果集
    for i, row in enumerate(rows):
        for j in range(len(row)):
            en = Label(win, text=row[j])
            en.grid(row=i+1, column=j)
    win.mainloop()
if __name__ == '__main__':
    main()
```

通过命令行工具进入 dbapp 目录下，在该目录下执行如下命令：

```
Pyinstaller -F -w main.py
```

上面命令中的 -F 选项指定生成单个的可执行程序，-w 选项指定生成图形用户界面程序（不需要命令行界面）。运行上面命令，该工具同样在 dbapp 目录下生成了一个 dist 子目录，并在该子目录下生成了一个 main.exe 文件。

直接双击运行 main.exe 程序（该程序有图形用户界面，因此可以双击运行），读者可自行查看运行结果。

9.怎样阻止别人反编译我们的程序包？

要知道，独立的可执行文件决不会让应用代码变得安全，从这样的可执行文件中反编译嵌入代码虽然不易，但确是可行的。更重要的是，这种反编译的结果（如果使用适当的工具）可能与原始源代码非常相似。

基于这一事实，对于泄露应用代码会对组织带来危害的闭源项目来说，独立的 Python 可执行文件并不是一个可行的解决方案。因此，如果仅复制应用的源代码就可以复制你的整个业务，那么你应该考虑使用其他方法来分发应用，比如，仅提供软件作为服务可能是更好的选择。

使反编译更难

目前，并没有可靠的方法能够保护应用不被可用的工具反编译，但是仍有一些方法可以使这个过程变得更加困难。

更加困难并不意味着被反编译的可能性更小，对于某些人来说，最吸引人的挑战正是最难的那些，并且挑战的最终奖励是非常大的，就是你设法保护的代码。

通常来说，反编译过程包括以下几个步骤：

1. 从独立可执行文件中提取项目字节码的二进制表示；

2. 将二进制表示映射到特定 Python 版本的字节码。
3. 将字节码转换成 AST。
4. 从 AST 直接重新创建源代码。

虽然无法阻止开发者对独立可执行文件进行这种逆向工程，但是这里给出一些想法，可以阻止反编译过程或者使其结果变得没有价值，比如：

- 运行时删除所有可用的代码元数据(文档字符串),从而稍微降低最终结果的可读性；
- 修改 CPython 解释器使用的字节码，这样从二进制转换成字节码、随后再转换成 AST 需要花费更多精力；
- 用复杂的方式修改 CPython 源代码版本，这样即使得到了应用的反编译源代码，没有对修改过的 CPython 库反编译也是没有用的；
- 在将源代码打包成可执行文件之前，对源代码使用混淆脚本，这将会使反编译后的源代码变得没有价值。

需要注意的是，这些解决方案都会使开发过程变得更加困难，其中的某些想法要求开发者对 Python 运行有非常深入的理解，而且每一个想法都有许多陷阱和缺点。

大多数情况下，这些方案只是将反编译的过程推迟了而已，一旦你的把戏被识破，所有额外的努力都将变成时间和资源的浪费。

要想不让闭源代码泄露到应用之外，唯一可靠的方法是就是不以任何形式将其直接发送给用户，只有你的组织其他方面的安全性都做得很好时，这种方法才有效。

第 20 章 Python 数据可视化

1.Python Matplotlib 安装

Matplotlib 是一个非常优秀的 Python 2D 绘图库，只要给出符合格式的数据，通过 Matplotlib 就可以方便地制作折线图、柱状图、散点图等各种高质量的数据图。

安装 Matplotlib 包与安装其他 Python 包没有区别，同样可以使用 pip 来安装。

启动命令行窗口，在命令行窗口中输入如下命令：

```
pip install matplotlib
```

上面命令将会自动安装 Matplotlib 包的最新版本。运行上面命令，可以看到程序先下载 Matplotlib 包，然后提示 Matplotlib 包安装成功：

```
Installing collected packages: matplotlib
Successfully installed matplotlib-2.2.3
```

如果在命令行窗口中提示找不到 pip 命令，则也可以通过 python 命令运行 pip 模块来安装 Matplotlib 包。

例如，通过如下命令来安装 Matplotlib 包：

```
python -m pip install matplotlib
```

在成功安装 Matplotlib 包之后，可以通过 pydoc 来查看 Matplotlib 包的文挡。在命令行窗口中输入如下命令：

```
python -m pydoc -p 8899
```

运行上面命令之后，打开浏览器查看 http://localhost:8899/ 页面，可以在 Python 安装目录的 lib\site-packages 下看到 Matplotlib 包的文档，如图 1 所示。

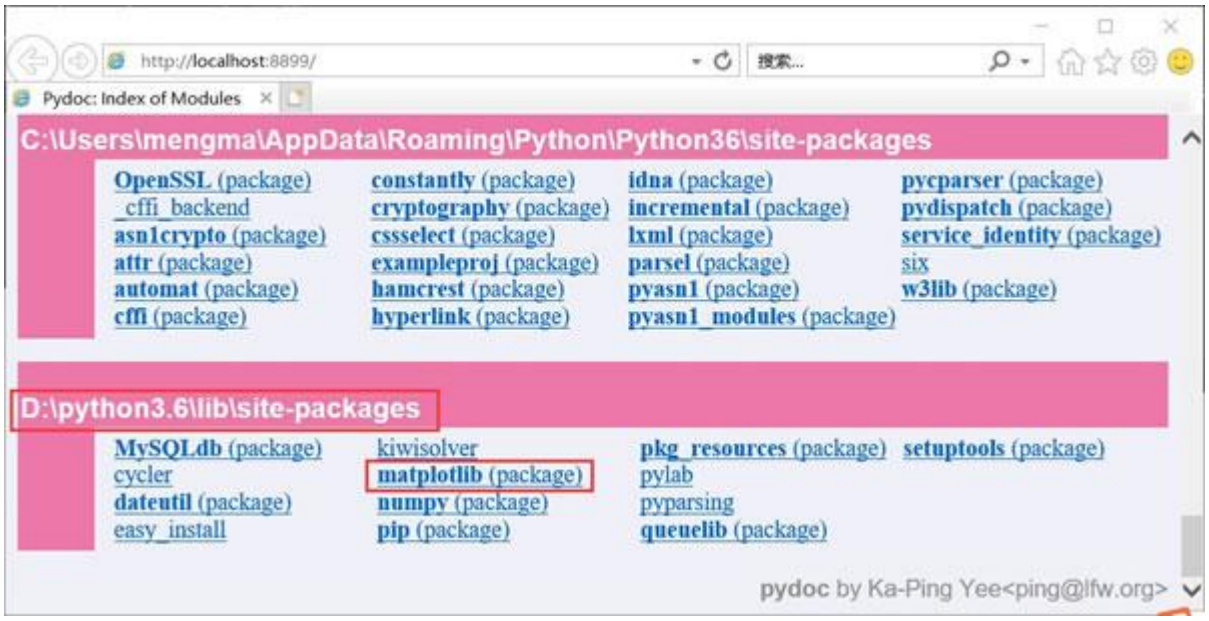


图 1 Matplotlib 包的文档

单击图 1 所示页面上的 “matplotlib(package)” 链接，将可以看到如图 2 所示的 API 页面。

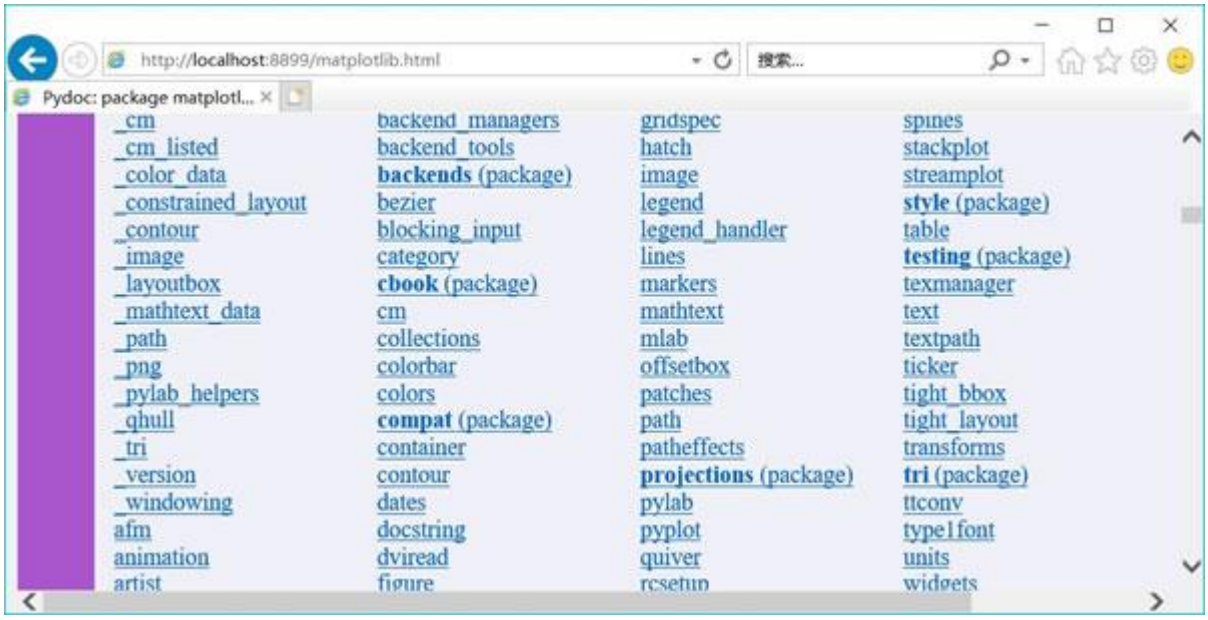


图 2 Matplotlib 包的 API 页面

通过图 2 所示的页面，即可查看 Matplotlib 包下的函数和类。

2.Python Matplotlib plot

plot 函数用法：生成折线图

Matplotlib 的用法非常简单，对于最简单的折线图来说，程序只需根据需要给出对应的 X 轴、Y 轴数据，调用 pyplot 子模块下的 plot() 函数即可生成简单的折线图。

假设分析《C 语言基础》这本教程从 2013 年到 2019 年的销售数据，此时可考虑将年份作为 X 轴数据，将图书各年销量作为 Y 轴数据。程序只要将 2013~2019 年定义成 list 列表作为 X 轴数据，并将对应年份的销量作为 Y 轴数据即可。

例如，使用如下简单的入门程序来展示这套教程从 2013 年到 2019 年的销售数据。

```
1.          import matplotlib.pyplot as plt
2.
3.          # 定义 2 个列表分别作为 X 轴、Y 轴数据
4.          x_data = ['2013', '2014', '2015', '2016', '2017', '2018', '2019']
5.          y_data = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
6.          # 第一个列表代表横坐标的值，第二个代表纵坐标的值
7.          plt.plot(x_data, y_data)
8.          # 调用 show()函数显示图形
9.          plt.show()
```

上面程序中，第 7 行代码调用 plot() 函数根据 X 轴、Y 轴数据来生成折线图，第 9 行代码则调用 show() 函数将折线图显示出来。

运行上面程序，可以看到生成如图 1 所示的简单折线图。

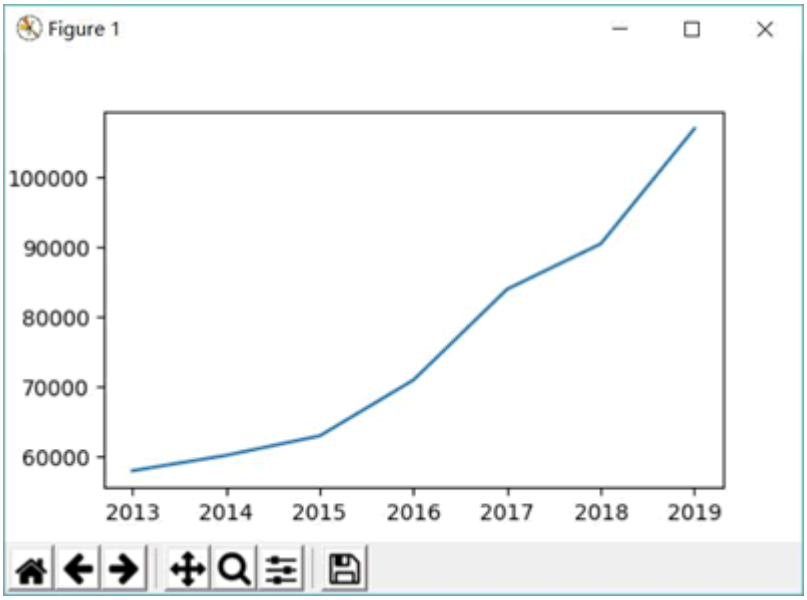


图 1 简单折线图

如果在调用 plot() 函数时只传入一个 list 列表，该 list 列表表的数据将作为 Y 轴数据，那么 Matplotlib 会自动使用 0、1、2、3 作为 X 轴数据。例如，将上面程序中的第 7 行代码改为如下形式：

```
plt.plot(y_data)
```

再次运行该程序，将看到如图 2 所示的结果。

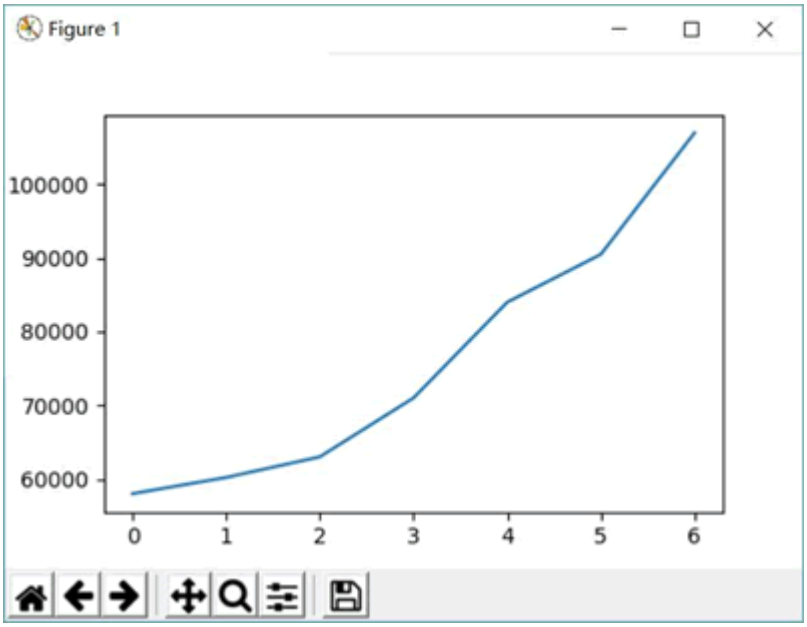


图 2 使用默认的 X 轴数据

plot() 函数除了支持创建具有单条折线的折线图，也支持创建包含多条折线的复式折线图，只要在调用 plot() 函数时传入多个分别代表 X 轴和 Y 轴数据的 list 列表即可。例如如下程序：

```
1.          import matplotlib.pyplot as plt
2.
3.          x_data = ['2013', '2014', '2015', '2016', '2017', '2018', '2019']
4.          # 定义 2 个列表分别作为两条折线的 Y 轴数据
```

```
y_data = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
# 传入 2 组分别代表 X 轴、Y 轴的数据
plt.plot(x_data, y_data, x_data, y_data2)
# 调用 show()函数显示图形
plt.show()
```

上面程序在调用 plot() 函数时，传入了两组分别代表 X 轴、Y 轴数据的 list 列表，因此该程序可以显示两条折线，如图 3 所示。

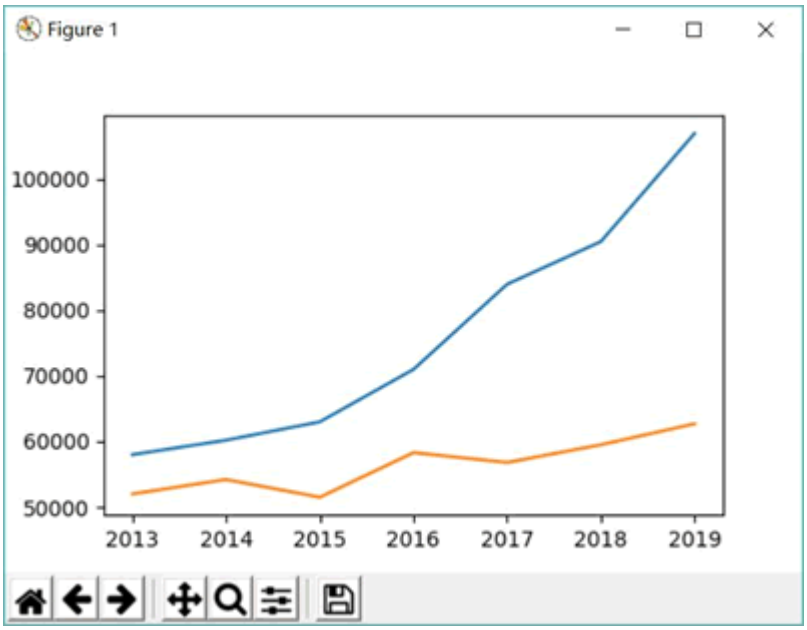


图 3 包含多条折线的复式折线图

也可以通过多次调用 plot() 函数来生成多条折线。例如，将上面程序中第 8 行代码改为如下两行代码，程序同样会生成包含两条折线的复式折线图：

```
plt.plot(x_data, y_data)
plt.plot(x_data, y_data2)
```

在调用 plot() 函数时还可以传入额外的参数来指定折线的样子，如线宽、颜色、样式等。

例如如下程序：

```
1.          import matplotlib.pyplot as plt
2.
3.          x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']
4.          # 定义 2 个列表分别作为两条折线的 Y 轴数据
5.          y_data = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
6.          y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
7.          # 指定折线的颜色、线宽和样式
8.          plt.plot(x_data, y_data, color ='red', linewidth = 2.0, linestyle ='--')
9.          plt.plot(x_data, y_data2, color = 'blue', linewidth = 3.0, linestyle =
'-.')
10.         # 调用 show()函数显示图形
11.         plt.show()
```

上面第 8、9 两行代码分别绘制了两条折线，并通过 color 指定折线的颜色，linewidth 指定线宽，linestyle 指定折线样式。

在使用 linestyle 指定折线样式时，该参数支持如下字符串参数值：

- -：代表实线，这是默认值。
- --：代表虚线。
- ·: 代表点钱。
- -.：代表短线、点相间的虚钱。

运行上面程序，可以看到如图 4 所示的折线图。

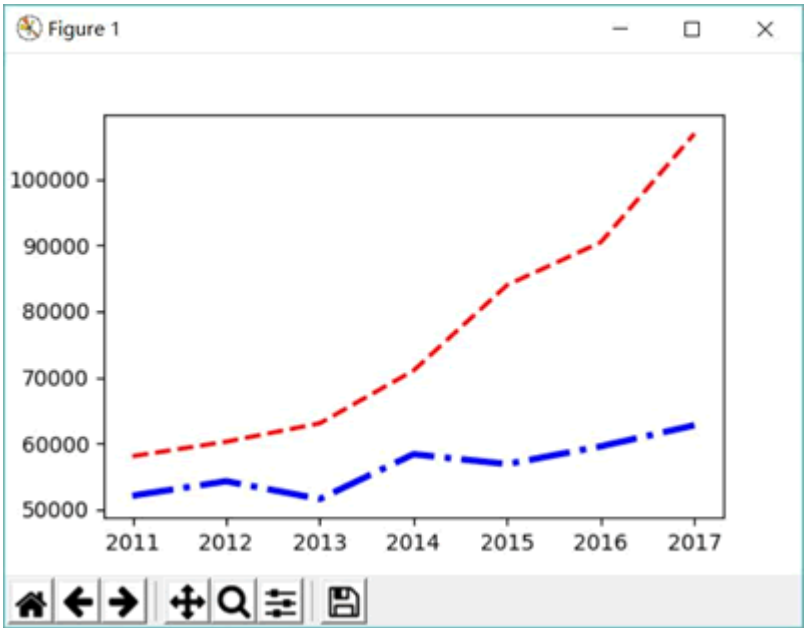


图 4 设置了折线的折线图

3.Python Matplotlib legend

legend 函数：为每条折线添加图例

对于复式折线图来说，应该为每条折线都添加图例，此时可以通过 legend() 函数来实现。

对于该函数可传入两个 list 参数，其中第一个 list 参数（handles 参数）用于引用折线图上的每条折线；第二个 list 参数（labels）代表为每条折线所添加的图例。

下面程序示范了为两条折线添加图例：

```
1.          import matplotlib.pyplot as plt
2.
3.          x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']
4.          # 定义 2 个列表分别作为两条折线的 Y 轴数据
5.          y_data = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
6.          y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
7.          # 指定折线的颜色、线宽和样式
```

```
8.          ln1, = plt.plot(x_data, y_data, color = 'red', linewidth = 2.0, linestyle
    = '--')
9.          ln2, = plt.plot(x_data, y_data2, color = 'blue', linewidth = 3.0, linestyle
    = '-.')
10.         # 调用 legend 函数设置图例
11.         plt.legend(handles=[ln2, ln1], labels=['Android 基础', 'Java 基础'],
12.             loc='lower right')
13.         # 调用 show()函数显示图形
14.         plt.show()
```

上面程序在调用 plot() 函数绘制折线图时，获取了该函数的返回值。由于该函数的返回值是一个列表，而此处只需要获取它返回的列表的第一个元素（第一个元素才代表该函数所绘制的折线图），因此程序利用返回值的序列解包来获取。

上面程序中，第 11 行代码用于为 ln2、ln1 所代表的折线添加图例（按传入该函数的两个列表的元素顺序一一对应），其中 loc 参数指定图例的添加位置，该参数支持如下参数值：

- 'best'：自动选择最佳位置。
- 'upper right'：将图例放在右上角。
- 'upper left'：将图例放在左上角。
- 'lower left'：将图例放在左下角。
- 'lower right'：将图例放在右下角。
- 'right'：将图例放在右边。
- 'center left'：将图例放在左边居中的位置。
- 'center right'：将图例放在右边居中的位置。
- 'lower center'：将图例放在底部居中的位置。

- 'upper center'：将图例放在顶部居中的位置。
- 'center'：将图例放在中心。

运行上面程序，将会发现应程序并没有绘制图例，这是因为 Matplotlib 默认不支持中文字体。如果希望在程序中修改 Matplotlib 的默认字体，则可按如下步骤进行：

1. 使用 matplotlib.font_manager 子模块下的 FontProperties 类加载中文字体。
2. 在调用 legend() 函数时通过 prop 属性指定使用中文字体。

将上面程序中第 11 行代码改为如下几行代码：

```
import matplotlib.font_manager as fm
# 使用 Matplotlib 的字体管理器加载中文字体
my_font=fm.FontProperties(fname="C:\Windows\Fonts\simkai.ttf")
# 调用 legend 函数设置图例
plt.legend(handles=[ln2, ln1], labels=['Android 基础', 'Java 基础'],
    loc='lower right', prop=my_font)
```

上面程序使用 FontProperties 类来加载 C:Windows\Fonts\simkai.ttf 文件所对应的中文字体，因此需要保证系统能找到该路径下的中文字体。

再次运行上面程序，将看到如图 1 所示的效果。

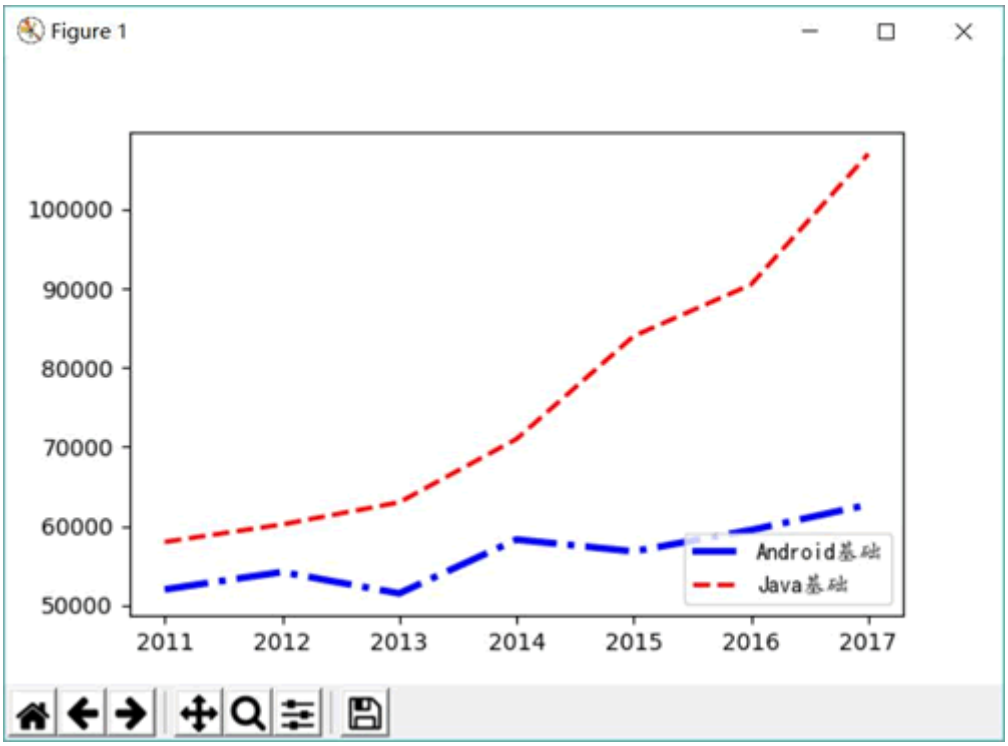


图 1 使用 legend() 函数指定图例

在使用 legend() 函数时可以不指定 handles 参数，只传入 labels 参数，这样该 labels 参数将按顺序为折线图中的多条折线添加图例。因此，可以将上面第 5 行代码改为如下形式：

```
1.          plt.legend(labels=['Java 基础', 'Android 基础'],
2.              loc='lower right', prop=my_font)
```

上面代码只指定了 labels 参数，该参数传入的列表包含两个字符串，其中第一个字符串将作为第一条折线（虚线）的图例，第二个字符串将作为第二条折线（短线、点相间的虚线）的图例。

Matplotlib 也允许在调用 plot() 函数时为每条折线分别传入 label 参数，这样程序在调用 legend() 函数时就无须传入 labels、handles 参数了。例如如下程序：

```
import matplotlib.pyplot as plt

x_data = ['2013', '2014', '2015', '2016', '2017', '2018', '2019']
# 定义 2 个列表分别作为两条折线的 Y 轴数据
y_data = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
# 指定折线的颜色、线宽和样式
plt.plot(x_data, y_data, color = 'red', linewidth = 2.0,
    linestyle = '--', label='Java 基础')
plt.plot(x_data, y_data2, color = 'blue', linewidth = 3.0,
    linestyle = '-.', label='Android 基础')
import matplotlib.font_manager as fm
# 使用 Matplotlib 的字体管理器加载中文字体
my_font=fm.FontProperties(fname="C:\Windows\Fonts\simkai.ttf")
# 调用 legend 函数设置图例
plt.legend(loc='best')
# 调用 show()函数显示图形
plt.show()
```

上面程序在调用 plot() 函数时传入了 label 参数，这样每条折线本身已经具有图例了，因此程序在调用 legend() 函数生成图例时无须传入 labels 参数。

正如从上面程序中所看到的，每次绘制中文内容时都需要设置字体，那么是否能改变 Matplotlib 的默认字体呢？答案是肯定的。在 Python 的交互式解释器中输入如下两行命令：

```
>>> import matplotlib
>>> matplotlib.matplotlib_fname()
'D:\\Python\\Python36\\lib\\site-packages\\matplotlib\\mp1-data\\matplotlibrc'
```

其中 matplotlib_fname() 函数会显示 Matplotlib 配置文件的保存位置，此处显示该文件的存储路径为 D:\Python\Python36\lib\site-packages\matplotlib\mpl-data\matplotlibrc。打开该文件，找到如下这行代码：

```
#font.family：sans-serif
```

上面这行代码用于配置 Matplotlib 的默认字体，取消运行配置代码之前的注释符号（ # ），并将后面的 sans-serif 修改为本地已有的中文字体。例如使用微软雅黑字体，只要将上面的配置代码修改为如下形式即可：

```
font.family： Microsoft YaHei
```

通过上面设置，即可改变 Matplotlib 的默认字体，这样即可避免每次调用 legend() 函数时都需要额外指定字体。

4.Python Matplotlib time、xlabel、ylabel、xticks、yticks

可以调用 xlable() 和 ylabel() 函数分别设置 X 轴、Y 轴的名称，也可以通过 title() 函数设置整个数据图的标题，还可以调用 xticks()、yticks() 函数分别改变 X 轴、Y 轴的刻度值（允许使用文本作为刻度值）。

例如，如下程序为数据图添加了名称、标题和坐标轴刻度值：

```
import matplotlib.pyplot as plt

x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']
# 定义 2 个列表分别作为两条折线的 Y 轴数据
y_data = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
# 指定折线的颜色、线宽和样式
plt.plot(x_data, y_data, color = 'red', linewidth = 2.0,
    linestyle = '--', label='Java 基础')
plt.plot(x_data, y_data2, color = 'blue', linewidth = 3.0,
    linestyle = '-.', label='C 基础')
import matplotlib.font_manager as fm
# 使用 Matplotlib 的字体管理器加载中文字体
my_font=fm.FontProperties(fname="C:\Windows\Fonts\simkai.ttf")
# 调用 legend 函数设置图例
plt.legend(loc='best')
# 设置两条坐标轴的名字
plt.xlabel("年份")
plt.ylabel("教程销量")
# 设置数据图的标题
plt.title('C 语言中文网的历年销量')
# 设置 Y 轴上的刻度值
# 第一个参数是点的位置，第二个参数是点的文字提示
plt.yticks([50000, 70000, 100000],
    [r'挺好', r'优秀', r'火爆'])
# 调用 show()函数显示图形
plt.show()
```

运行上面程序，可以看到如图 1 所示的效果。

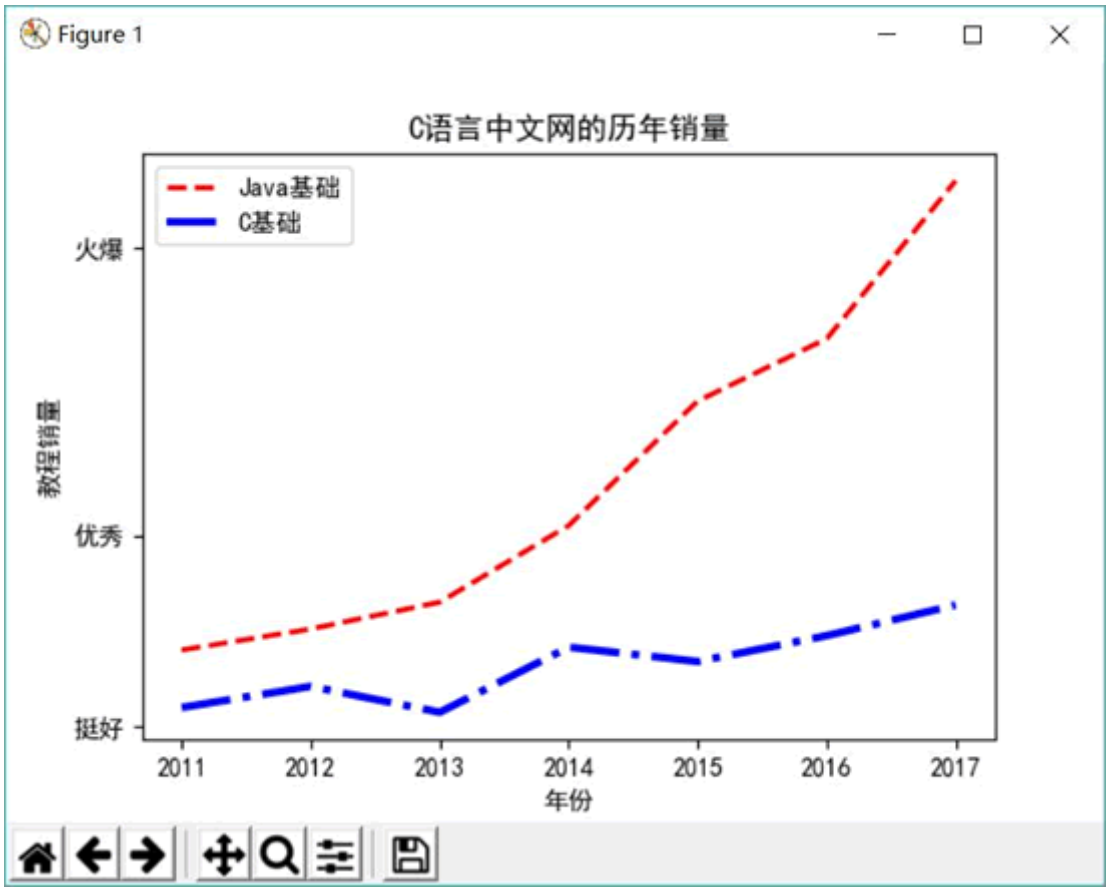


图 1 为数据图添加名称、标题和坐标轴刻度值

上面程序中的前两行粗体字代码分别设置了 X 轴、Y 轴的 label，因此可以看到图 1 中的 X 轴和 Y 轴的标签发生了改变。

如果要对 X 轴、Y 轴进行更细致的控制，则可调用 gca() 函数来获取坐标轴信息对象，然后对坐标轴进行控制。比如控制坐标轴上刻度值的位置和坐标轴的位置等。

下面程序示范了对坐标轴的详细控制：

```
1.          import matplotlib.pyplot as plt
```

```
2.
3.          x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']
4.          # 定义 2 个列表分别作为两条折线的 Y 轴数据
5.          y_data = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
6.          y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
7.          # 指定折线的颜色、线宽和样式
8.          plt.plot(x_data, y_data, color = 'red', linewidth = 2.0,
9.              linestyle = '--', label='Java 基础')
10.         plt.plot(x_data, y_data2, color = 'blue', linewidth = 3.0,
11.             linestyle = '-.', label='C 语言基础')
12.         import matplotlib.font_manager as fm
13.         # 使用 Matplotlib 的字体管理器加载中文字体
14.         my_font=fm.FontProperties(fname="C:\Windows\Fonts\simkai.ttf")
15.         # 调用 legend 函数设置图例
16.         plt.legend(loc='best')
17.         # 设置两条坐标轴的名字
18.         plt.xlabel("年份")
19.         plt.ylabel("教程销量")
20.         # 设置数据图的标题
21.         plt.title('C 语言中文网的历年销量')
22.         # 设置 Y 轴上的刻度值
23.         # 第一个参数是点的位置，第二个参数是点的文字提示
24.         plt.yticks([50000, 70000, 100000],
25.             [r'挺好', r'优秀', r'火爆'])
26.         ax = plt.gca()
27.         # 设置将 X 轴的刻度值放在底部 X 轴上
28.         ax.xaxis.set_ticks_position('bottom')
29.         # 设置将 Y 轴的刻度值放在底部 X 轴上
30.         ax.yaxis.set_ticks_position('left')
31.         # 设置右边坐标轴线的颜色（设置为 none 表示不显示）
32.         ax.spines['right'].set_color('none')
33.         # 设置顶部坐标轴线的颜色（设置为 none 表示不显示）
34.         ax.spines['top'].set_color('none')
35.         # 定义底部坐标轴线的位置（放在 70000 数值处）
36.         ax.spines['bottom'].set_position(('data', 70000))
37.         # 调用 show()函数显示图形
```

```
38.         plt.show()
```

上面程序获取了数据图上的坐标轴对象，它是一个 AxesSubplot 对象。接下来程序调用 AxesSubplot 的 xaxis 属性的 set_ticks_position() 方法设置 X 轴刻度值的位置；与之对应的是，调用 yaxis 属性的 set_ticks_position() 方法设置 Y 轴刻度值的位置。

通过 AxesSubplot 对象的 spines 属性可以访问数据图四周的坐标轴线（Spine 对象），通过 Spine 对象可设置坐标轴线的颜色、位置等。例如，程序将数据图右边和顶部的坐标轴线设为 none，表示隐藏这两条坐标轴线。程序还将底部坐标轴线放在数值 70000 处。

运行上面程序，可以看到如图 2 所示的效果。

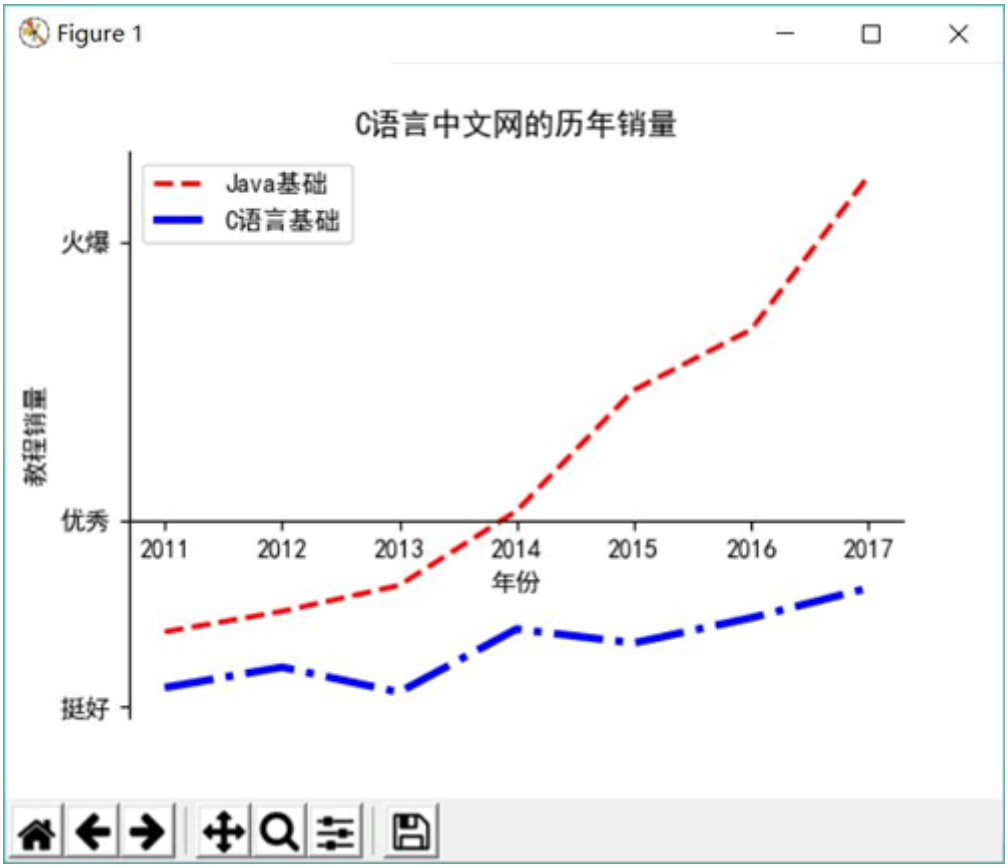


图 2 控制坐标轴

5.Python Matplotlib subplot

使用 Matplotlib 除可以生成包含多条折线的复式折线图之外，它还允许在一张数据图上包含多个子图。

调用 subplot() 函数可以创建一个子图，然后程序就可以在子图上进行绘制。

subplot(nrows, ncols, index, **kwargs) 函数的 nrows 参数指定将数据图区域分成多少行；ncols 参数指定将数据图区域分成多少列；index 参数指定获取第几个区域。

subplot() 函数也支持直接传入一个三位数的参数，其中第一位数将作为 nrows 参数；第二位数将作为 ncols 参数；第三位数将作为 index 参数。

下面程序示范了生成多个子图：

```
1.          import matplotlib.pyplot as plt
2.          import numpy as np
3.
4.          plt.figure()
5.          # 定义从-pi 到 pi 之间的数据，平均取 64 个数据点
6.          x_data = np.linspace(-np.pi, np.pi, 64, endpoint=True)  # ①
7.          # 将整个 figure 分成两行两列，第三个参数表示该图形放在第 1 个网格
8.          plt.subplot(2, 2, 1)
9.          # 绘制正弦曲线
10.         plt.plot(x_data, np.sin(x_data))
11.         plt.gca().spines['right'].set_color('none')
12.         plt.gca().spines['top'].set_color('none')
13.         plt.gca().spines['bottom'].set_position(('data', 0))
```

```
14.         plt.gca().spines['left'].set_position(('data', 0))
15.         plt.title('正弦曲线')
16.
17.         # 将整个 figure 分成两行两列，并将该图形放在第 2 个网格
18.         plt.subplot(222)
19.         # 绘制余弦曲线
20.         plt.plot(x_data, np.cos(x_data))
21.         plt.gca().spines['right'].set_color('none')
22.         plt.gca().spines['top'].set_color('none')
23.         plt.gca().spines['bottom'].set_position(('data', 0))
24.         plt.gca().spines['left'].set_position(('data', 0))
25.         plt.title('余弦曲线')
26.
27.         # 将整个 figure 分成两行两列，并该图形放在第 3 个网格
28.         plt.subplot(223)
29.         # 绘制正切曲线
30.         plt.plot(x_data, np.tan(x_data))
31.         plt.gca().spines['right'].set_color('none')
32.         plt.gca().spines['top'].set_color('none')
33.         plt.gca().spines['bottom'].set_position(('data', 0))
34.         plt.gca().spines['left'].set_position(('data', 0))
35.         plt.title('正切曲线')
36.
37.         plt.show()
```

上面程序多次调用 subplot() 函数来生成子图，每次调用 subplot() 函数之后的代码表示在该子图区域绘图。上面程序将整个数据图区域分成 2×2 的网格，程序分别在第 1 个网格中绘制正弦曲线，在第 2 个网格中绘制余弦曲线，在第 3 个网格中绘制正切曲线。

可能有读者感到疑问，plot() 函数不是用于绘制折线图的吗？怎么此处还可用于绘制正弦曲线、余弦曲线呢？

其实此处绘制的依然是折线图。看程序中的 ① 号代码调用 numpy 的 linspace() 函数生成了一个包含多个数值的列表 ,该数值列表的范围是从 -pi 到 pi ,平均分成 64 个数据点，程序中用到的 numpy.sin()、numpy.cos()、numpy.tan() 等函数也返回一个列表：传入这些函数的列表包含多少个值，这些函数返问的列表也包含多少个值。

这意味着上面程序所给制的折线图会包含 64 个转折点，由于这些转折点非常密集，看上去显得比较光滑，因此就变成了曲线。

如果读者将程序中 x_data=np.linespace(-np.pi,np.pi,64,endpoint = True)代码的 64 改为 4、6 等较小的数，将会看到程序绘制的依然是折线图。

运行上面程序，可以看到如图 1 所示的效果。

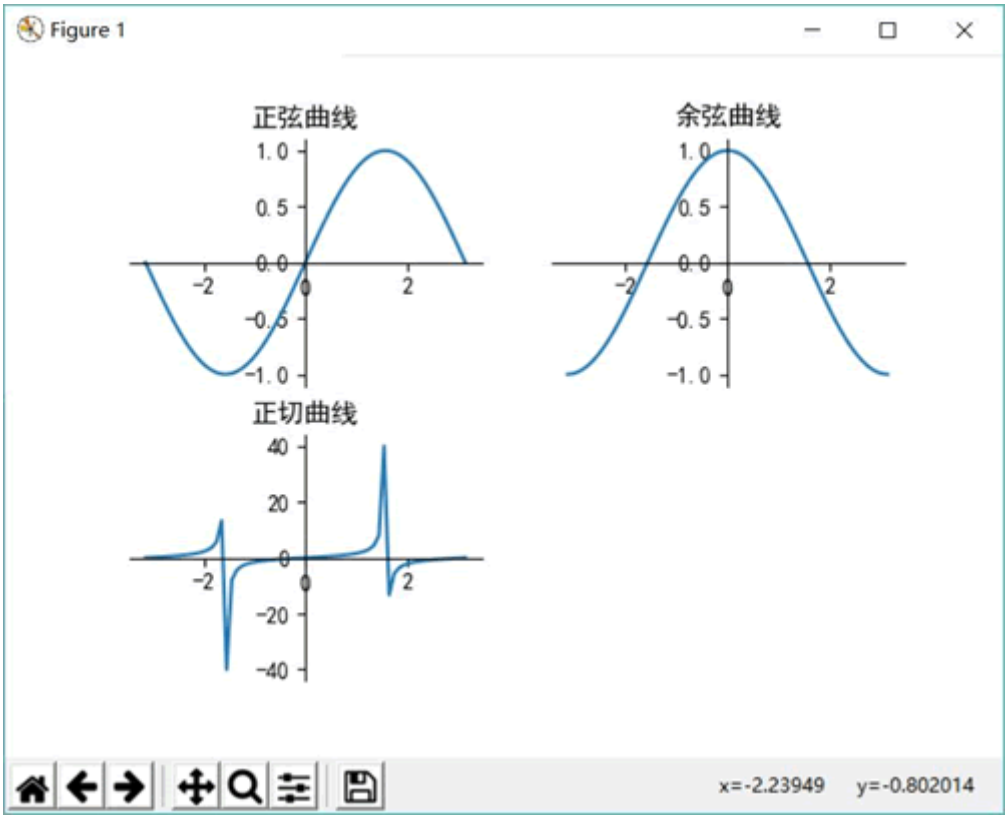


图 1 生成多个子图

如图 1 所示的显示效果比较差，程序明明只要显示 3 个子图，但第 4 个位置被空出来了，能不能让某个子图占多个网格呢？答案是肯定的，程序做好控制即可。例如，将上面程序改为如下形式：

```
import matplotlib.pyplot as plt
import numpy as np

plt.figure()
# 定义从-pi 到 pi 之间的数据，平均取 64 个数据点
x_data = np.linspace(-np.pi, np.pi, 64, endpoint=True)  # ①
# 将整个 figure 分成两行一列，第三个参数表示该图形放在第 1 个网格
plt.subplot(2, 1, 1)
# 省略绘制正弦曲线
```

```
10.         ...
11.
12.         # 将整个 figure 分成两行两列，并将该图形放在第 4 个网格
13.         plt.subplot(223)
14.         # 省略绘制余弦曲线
15.         ...
16.
17.         # 将整个 figure 分成两行两列，并该图形放在第 4 个网格
18.         plt.subplot(224)
19.         #省略绘制正切曲线
20.         ...
21.
22.         plt.show()
```

上面程序中第 8 行代码将整个区域分成两行一列，并指定子图占用第 1 个网格，也就是整个区域的第一行：第 13 行代码将整个区域分成两行两列，并指定子图占用第 3 个网格（注意不是第 2 个网络，因为第一个子图已经占用了第一行，对于两行两列的网格来说，第一个子图已经占用了两个网格，因此此处指定子图占用第 3 个网格，这意味着该子图在第二行第一格）；第 18 行代码将整个区域分成两行两列，并指定子图占用第 4 个网格，这意味着该子图会在第二行第二格。

运行上面程序，可以看到如图 2 所示的效果。

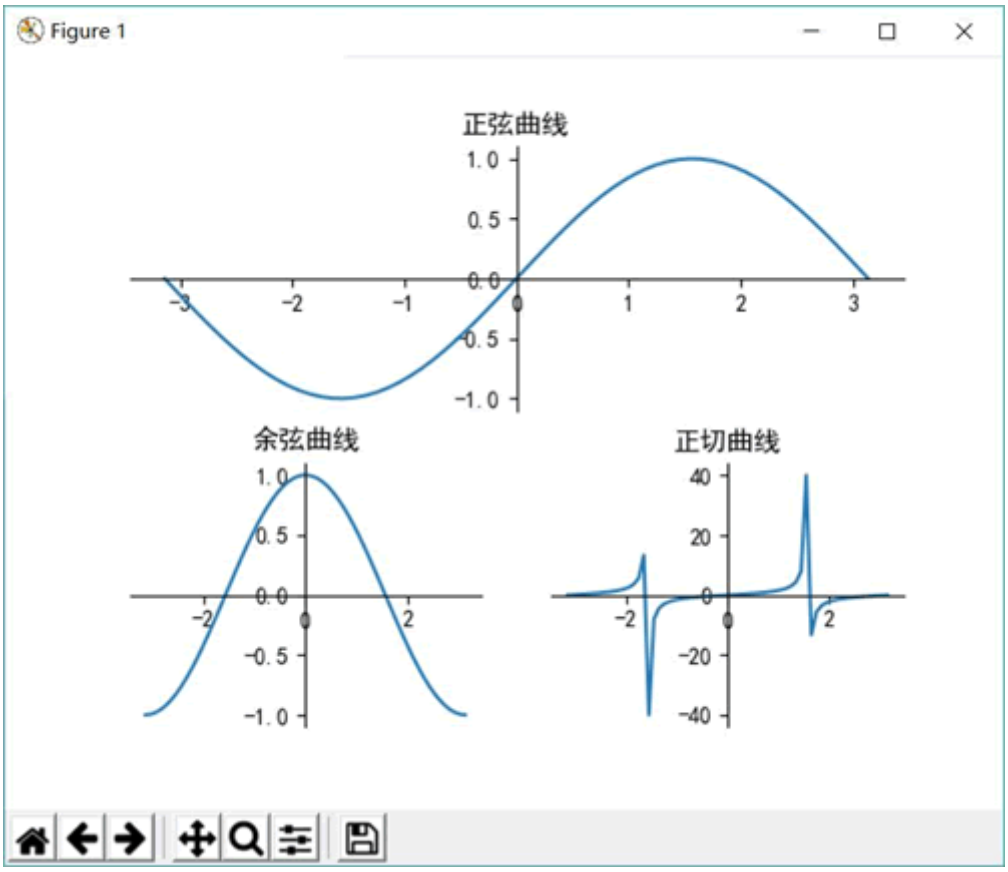


图 2 控制多个子图的分布

如果读者不想这么费劲来计算行、列，将上面程序改为如下形式：

```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.gridspec as gridspec

plt.figure()
# 定义从-pi 到 pi 之间的数据，平均取 64 个数据点
x_data = np.linspace(-np.pi, np.pi, 64, endpoint=True)  # ①

# 将绘图区域分成 2 行 3 列
gs = gridspec.GridSpec(2, 3)
# 指定 ax1 占用第一行（0）整行
ax1 = plt.subplot(gs[0, :])
```

```
13.         # 指定 ax1 占用第二行（1）的第一格（第二个参数 0 代表）
14.         ax2 = plt.subplot(gs[1, 0])
15.         # 指定 ax1 占用第二行（1）的第二、三格（第二个参数 0 代表）
16.         ax3 = plt.subplot(gs[1, 1:3])
17.
18.         # 绘制正弦曲线
19.         ax1.plot(x_data, np.sin(x_data))
20.         ax1.spines['right'].set_color('none')
21.         ax1.spines['top'].set_color('none')
22.         ax1.spines['top'].set_color('none')
23.         ax1.spines['bottom'].set_position(('data', 0))
24.         ax1.spines['left'].set_position(('data', 0))
25.         ax1.set_title('正弦曲线')
26.
27.         # 绘制余弦曲线
28.         ax2.plot(x_data, np.cos(x_data))
29.         ax2.spines['right'].set_color('none')
30.         ax2.spines['top'].set_color('none')
31.         ax2.spines['bottom'].set_position(('data', 0))
32.         ax2.spines['left'].set_position(('data', 0))
33.         ax2.set_title('余弦曲线')
34.
35.         # 绘制正切曲线
36.         ax3.plot(x_data, np.tan(x_data))
37.         ax3.spines['right'].set_color('none')
38.         ax3.spines['top'].set_color('none')
39.         ax3.spines['bottom'].set_position(('data', 0))
40.         ax3.spines['left'].set_position(('data', 0))
41.         ax3.set_title('正切曲线')
42.
43.         plt.show()
```

上面程序中的第 10 行代码将绘图区域分成两行三列；第 12 行代码调用 subplot(gs[0,:])，指定 ax1 子图区域占用第一行整行，其中第一个参数 0 代表行号，没有指定列范围，因此该子图在整个第一行；第 14 行代码调用 subplot(gs[1, 0])，指定 ax2

子图区域占用第二行的第一格，其中第一个参数 1 代表第二行，第二个参数 0 代表第一格，因此该子图在第二行的第一格；第 16 行代码调用 subplot(gs [1，1：3])，指定 ax3 子图区域占用第二行的第二格到第三格，其中第一个参数 1 代表第二行，第二个参数 1:3 代表第二格到第三格，因此该子图在第二行的第二格到第三格。

定义完 ax1、ax2、ax3 这 3 个子图所占用的区域之后，接下来程序就可以通过 ax1、ax2、ax3 的方法在各自的子图区域绘图了。运行上面程序，可以看到如图 3 所示的效果。

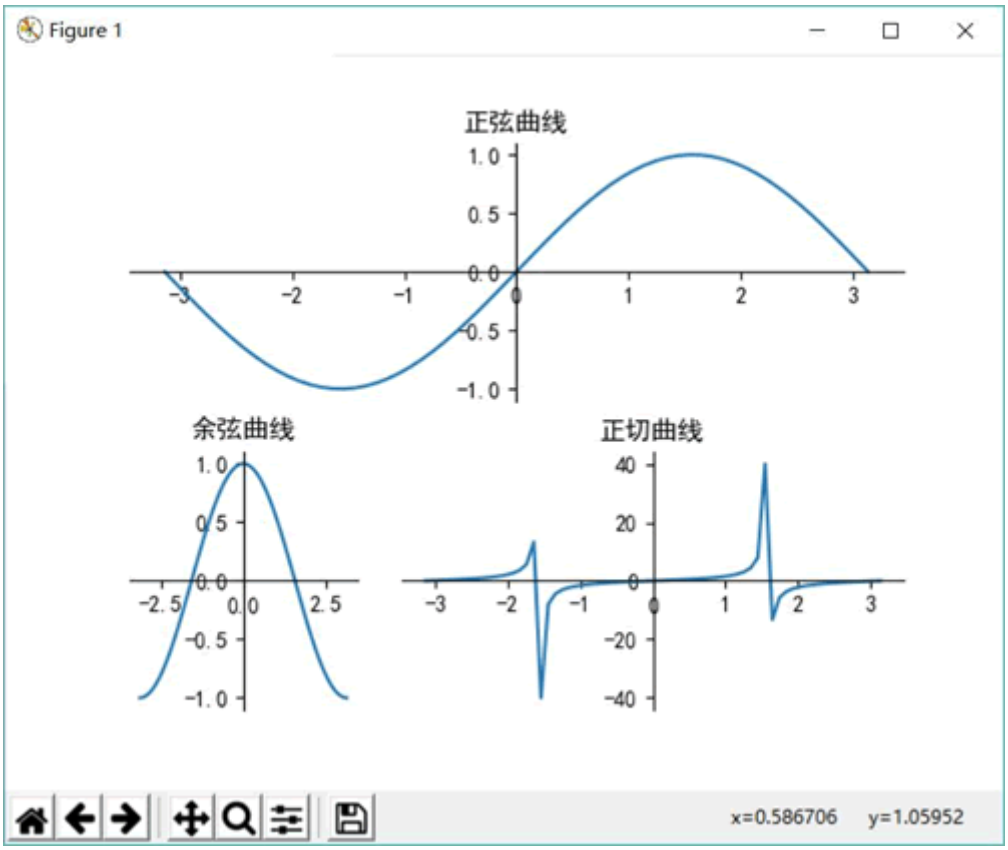


图 3 使用 GridSpec 管理子图的分布

6.Python Matplotlib pie

使用 Matplotlib 提供的 pie() 函数来绘制饼图。

下面是 TIOBE 2018 年 8 月的编程语言指数排行榜的前 10 名及其他：

Java：16.881%

C：14.966%

C++：7.471%

Python：6.992%

Visual Basic.NET：4.762%

C#：3.541%

PHP：2.925%

JavaScript：2.411%

SQL：2.316%

Assembly language：1.409%

其他：36.326%

下面程序将使用饼图来直观地展示这个编程语言指数排行榜：

纯文本复制

```
import matplotlib.pyplot as plt

# 准备数据
data = [0.16881, 0.14966, 0.07471, 0.06992,
    0.04762, 0.03541, 0.02925, 0.02411, 0.02316, 0.01409, 0.36326]
# 准备标签
```

```
7.          labels = ['Java', 'C', 'C++', 'Python',
8.              'Visual Basic .NET', 'C#', 'PHP', 'JavaScript',
9.              'SQL', 'Assembly langugage', '其他']
10.         # 将第 4 个语言（Python）分离出来
11.         explode = [0, 0, 0, 0.3, 0, 0, 0, 0, 0, 0, 0]
12.         # 使用自定义颜色
13.         colors=['red', 'pink', 'magenta','purple','orange']
14.         # 将横、纵坐标轴标准化处理，保证饼图是一个正圆，否则为椭圆
15.         plt.axes(aspect='equal')
16.         # 控制 X 轴和 Y 轴的范围（用于控制饼图的圆心，半径）
17.         plt.xlim(0,8)
18.         plt.ylim(0,8)
19.
20.         # 绘制饼图
21.         plt.pie(x = data, # 绘图数据
22.             labels=labels, # 添加编程语言标签
23.             explode=explode, # 突出显示 Python
24.             colors=colors, # 设置饼图的自定义填充色
25.             autopct='%.3f%%', # 设置百分比的格式，此处保留 3 位小数
26.             pctdistance=0.8,  # 设置百分比标签与圆心的距离
27.             labeldistance = 1.15, # 设置标签与圆心的距离
28.             startangle = 180, # 设置饼图的初始角度
29.             center = (4, 4), # 设置饼图的圆心（相当于 X 轴和 Y 轴的范围）
30.             radius = 3.8, # 设置饼图的半径（相当于 X 轴和 Y 轴的范围）
31.             counterclock = False, # 是否逆时针，这里设置为顺时针方向
32.             wedgeprops = {'linewidth': 1, 'edgecolor':'green'},# 设置饼图内外边
    界的属性值
33.             textprops = {'fontsize':12, 'color':'black'}, # 设置文本标签的属性
    值
34.             frame = 1) # 是否显示饼图的圆圈，此处设为显示
35.         # 不显示 X 轴和 Y 轴的刻度值
36.         plt.xticks(())
37.         plt.yticks(())
38.         # 添加图标题
39.         plt.title('2018 年 8 月的编程语言指数排行榜')
40.         # 显示图形
```

```
41.         plt.show()
```

上面程序调用 pie() 函数来生成饼图。创建饼图最重要的两个参数就是 x 和 labels，其中 x 指定饼图各部分的数值，labels 则指定各部分对应的标签。

运行上面程序，可以看到如图 1 所示的效果。

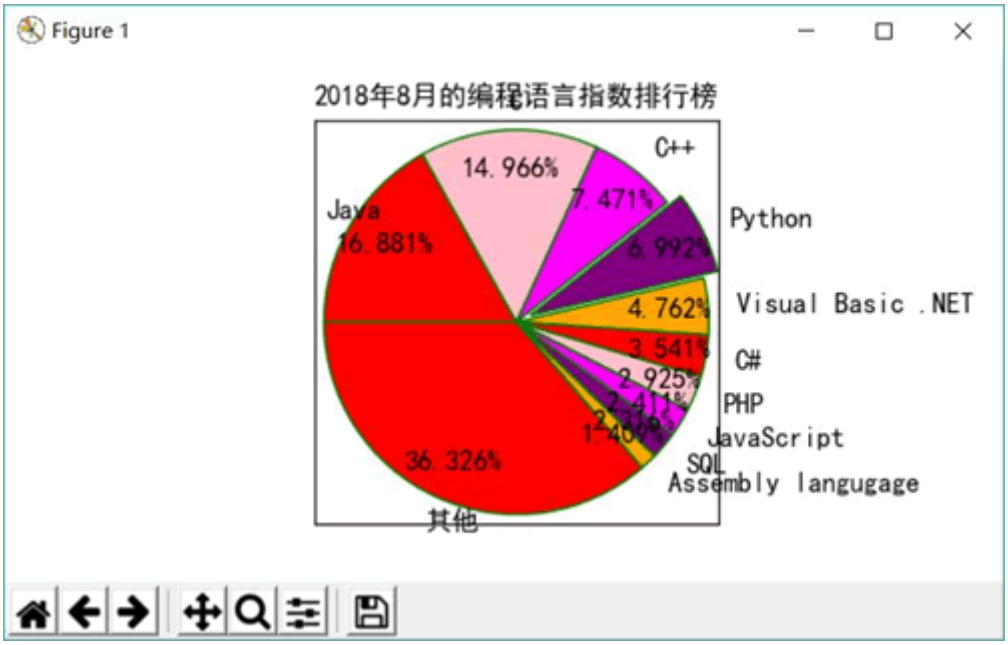


图 1 pie 绘制饼图

7.Python Matplotlib 绘制柱状图（bar 和 barh 函数）

使用 Matplotlib 提供的 bar() 函数来绘制柱状图。与前面介绍的 plot() 函数类似，程序每次调用 bar() 函数时都会生成一组柱状图， 如果希望生成多组柱状图，则可通过多次调用 bar() 函数来实现。

下面程序使用柱状图来展示《C 语言基础》和《Java 基础》两套教程历年的销量数据。

```
1.          import matplotlib.pyplot as plt
2.          import numpy as np
3.
4.          # 构建数据
5.          x_data = ['2012', '2013', '2014', '2015', '2016', '2017', '2018']
6.          y_data = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
7.          y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
8.          # 绘图
9.          plt.bar(x=x_data, height=y_data, label='C 语言基础', color='steelblue',
    alpha=0.8)
10.         plt.bar(x=x_data, height=y_data2, label='Java 基础', color='indianred',
    alpha=0.8)
11.         # 在柱状图上显示具体数值, ha 参数控制水平对齐方式, va 控制垂直对齐方式
12.         for x, y in enumerate(y_data):
13.             plt.text(x, y + 100, '%s' % y, ha='center', va='bottom')
14.         for x, y in enumerate(y_data2):
15.             plt.text(x, y + 100, '%s' % y, ha='center', va='top')
16.         # 设置标题
17.         plt.title("Java 与 Android 图书对比")
18.         # 为两条坐标轴设置名称
19.         plt.xlabel("年份")
20.         plt.ylabel("销量")
21.         # 显示图例
22.         plt.legend()
23.         plt.show()
```

上面程序中，第 9、10 两行代码用于在数据图上生成两组柱状图，程序设置了这两组柱状图的颜色和透明度。

在使用 bar() 函数绘制柱状图时，默认不会在柱状图上显示具体的数值。为了能在柱状图上显示具体的数值，程序可以调用 text() 函数在数据图上输出文字，如上面程序中第 10 行代码所示。

在使用 text() 函数输出文字时，该函数的前两个参数控制输出文字的 X、Y 坐标，第三个参数则控制输出的内容。其中 va 参数控制文字的垂直对齐方式，ha 参数控制文字的水平对齐方式。

对于上面的程序来说，由于 X 轴数据是一个字符串列表，因此 X 轴实际上是以列表元素的索引作为刻度值的。因此，当程序指定输出文字的 X 坐标为 0 时，表明将该文字输出到第一个条柱处；对于 Y 坐标而言，条柱的数值正好在条柱高度所在处，如果指定 Y 坐标为条柱的数值 +100，就是控制将文字输出到条柱略上一点的位置。

运行上面程序，可以看到如图 1 所示的效果。

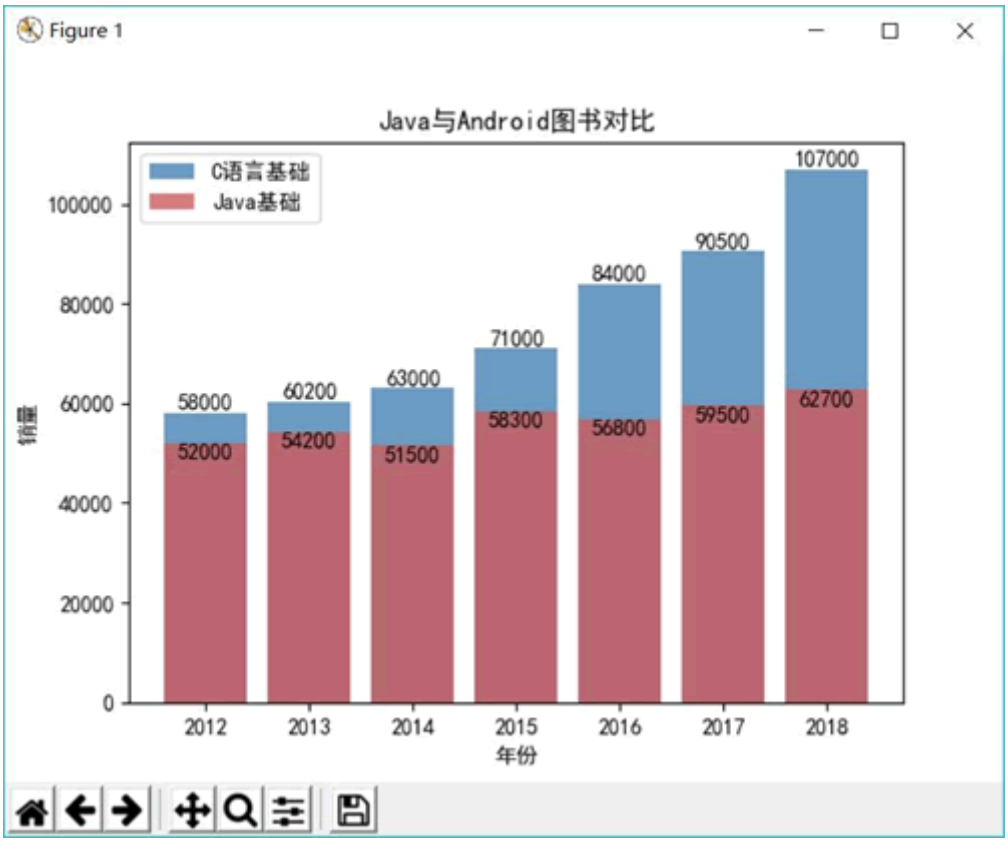


图 1 两组柱状图

从图 1 所示的显示效果来看，第二次绘制的性状图完全与第一次绘制的柱状图重叠，这并不是我们期望的结果，我们希望每组数据的条柱能并列显示。

为了实现条柱并列显示的效果，首先分析条柱重叠在一起的原因。使用 Matplotlib 绘制柱状图时同样也需要 X 轴数据，本程序的 X 轴数据是元素为字符串的 list 列表，因此程序实际上使用各字符串的索引作为 X 轴数据。比如 '2011' 字符串位于列表的第一个位置，因此代表该条柱的数据就被绘制在 X 轴的刻度值 1 处（由于两个柱状图使用了相同的 X 轴数据，因此它们的条柱完全重合在一起）。

为了将多个柱状图的条柱并列显示，程序需要为这些柱状图重新计算不同的 X 轴数据。为了精确控制条柱的宽度，程序可以在调用 bar() 函数时传入 width 参数，这样可以更好地计算条柱的并列方式。将上面程序改为如下形式：

```
1.          import matplotlib.pyplot as plt
2.          import numpy as np
3.
4.          # 构建数据
5.          x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']
6.          y_data = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
7.          y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
8.          bar_width=0.3
9.          # 将 X 轴数据改为使用 range(len(x_data), 就是 0、1、2...
10.         plt.bar(x=range(len(x_data)), height=y_data, label='C 语言基础',
11.             color='steelblue', alpha=0.8, width=bar_width)
12.         # 将 X 轴数据改为使用 np.arange(len(x_data))+bar_width,
```

```
        # 就是 bar_width、1+bar_width、2+bar_width...这样就和第一个柱状图并列了
        plt.bar(x=np.arange(len(x_data))+bar_width, height=y_data2,
            label='Java 基础', color='indianred', alpha=0.8, width=bar_width)
        # 在柱状图上显示具体数值, ha 参数控制水平对齐方式, va 控制垂直对齐方式
        for x, y in enumerate(y_data):
            plt.text(x, y + 100, '%s' % y, ha='center', va='bottom')
        for x, y in enumerate(y_data2):
            plt.text(x+bar_width, y + 100, '%s' % y, ha='center', va='top')
        # 设置标题
        plt.title("C 与 Java 对比")
        # 为两条坐标轴设置名称
        plt.xlabel("年份")
        plt.ylabel("销量")
        # 显示图例
        plt.legend()
        plt.show()
```

该程序与前一个程序的区别就在于第 10、14 两行代码，这两行代码使用了不同的 x 参数，其中第一个柱状图的 X 轴数据为 range(len(x_data))，也就是 0、1、2...，这样第一个柱状图的各条柱恰好位于 0、1、2... 刻度值处；第二个柱状图的 X 轴数据为 `np.arange(len(x_data))+bar_width`，也就是 bar_width、1+bar_width、2+bar_width·，这样第二个柱状图的各条柱位于 0、1、2...刻度值的偏右一点 bar_width 处，这样就恰好与第一个柱状图的各条柱并列了。

运行上面程序，将会发现该柱状图的 X 轴的刻度值变成 0、1、2 等值，不再显示年份。为了让柱状图的 X 轴的刻度值显示年份，程序可以调用 xticks() 函数重新设置 X 轴的刻度值。

例如，在程序中添加如下代码：

```
# 为 X 轴设置刻度值
plt.xticks(np.arange(len(x_data))+bar_width/2, x_data)
```

上面代码使用 x_data 为 X 轴设置刻度值，第一个参数用于控制各刻度值的位置，该参数是 `np.arange(len(x_data))+bar_width/2`，也就是 bar_width/2、1+bar_width/2、2+bar_width/2 等，这样这些刻度值将被恰好添加在两个条柱之间。

运行上面程序可看到如图 2 所示的运行结果：

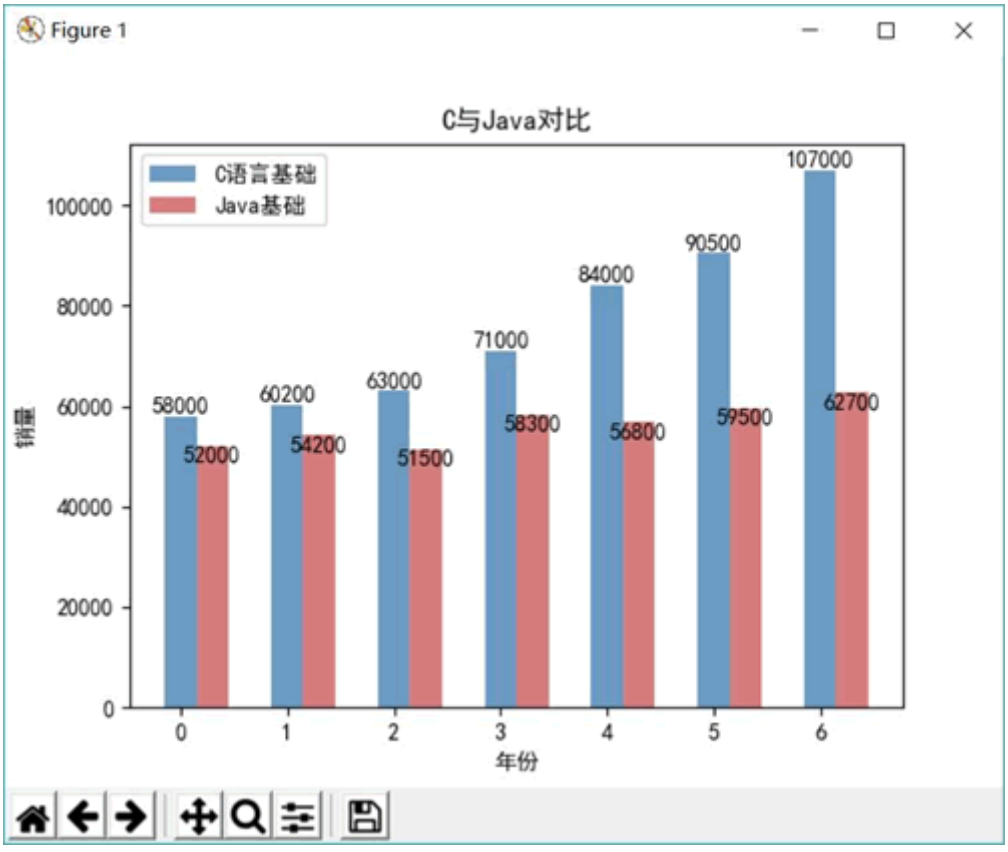


图 2 并列的柱状图

有些时候，可能希望两个条柱之间有一点缝隙，那么程序只要对第二个条柱的 X 轴数据略

做修改即可。例如，将上面程序中第 14 行代码改为如下形式：

```
plt.bar(x=np.arange(len(x_data))+bar_width+0.05, height=y_data2,
    label='Java 基础', color='indianred', alpha=0.8, width=bar_width)
```

上面代码重新计算了 X 轴数据，使用 `np.arange(len(x_data))+bar_width+0.05` 作为 X 轴数据，因此两组柱状图的条柱之间会有 0.05 的距离。

Matplotlib 绘制水平柱状图

调用 Matplotlib 的 barh() 函数可以生成水平柱状图。barh() 函数的用法与 bar() 函数的用法基本一样，只是在调用 barh() 函数时使用 y 参数传入 Y 轴数据，使用 width 参数传入代表条柱宽度的数据。

例如，如下程序调用 barh() 函数生成两组并列的水平柱状图，来展示两套教程历年的销量统计数据：

```
1.  import matplotlib.pyplot as plt
2.  import numpy as np
3.
4.  # 构建数据
5.  x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']
6.  y_data = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
7.  y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
8.  bar_width=0.3
9.  # Y 轴数据使用 range(len(x_data), 就是 0、1、2...
10. plt.barh(y=range(len(x_data)), width=y_data, label='Java 基础教程',
11.     color='steelblue', alpha=0.8, height=bar_width)
12. # Y 轴数据使用 np.arange(len(x_data))+bar_width,
13. # 就是 bar_width、1+bar_width、2+bar_width...这样就和第一个柱状图并列了
14. plt.barh(y=np.arange(len(x_data))+bar_width, width=y_data2,
```

```
15.             label='C 语言基础', color='indianred', alpha=0.8, height=bar_width)
16.
17.         # 在柱状图上显示具体数值, ha 参数控制水平对齐方式, va 控制垂直对齐方式
18.         for y, x in enumerate(y_data):
19.             plt.text(x+5000, y-bar_width/2, '%s' % x, ha='center', va='bottom')
20.         for y, x in enumerate(y_data2):
21.             plt.text(x+5000, y+bar_width/2, '%s' % x, ha='center', va='bottom')
22.         # 为 Y 轴设置刻度值
23.         plt.yticks(np.arange(len(x_data))+bar_width/2, x_data)
24.         # 设置标题
25.         plt.title("Java 与 C 对比")
26.         # 为两条坐标轴设置名称
27.         plt.xlabel("销量")
28.         plt.ylabel("年份")
29.         # 显示图例
30.         plt.legend()
31.         plt.show()
```

上面程序中，第 10 行代码使用 barh() 函数来创建水平柱状图，其中 y 参数为 range(len(x_data))，这意味着这些条柱将会沿着 Y 轴均匀分布；而 width 参数为 y_data，这意味着 y_data 列表所包含的数值会决定各条柱的宽度。第 14 行代码的控制方式与此类似。

运行上面程序，可以看到如图 3 所示的效果。

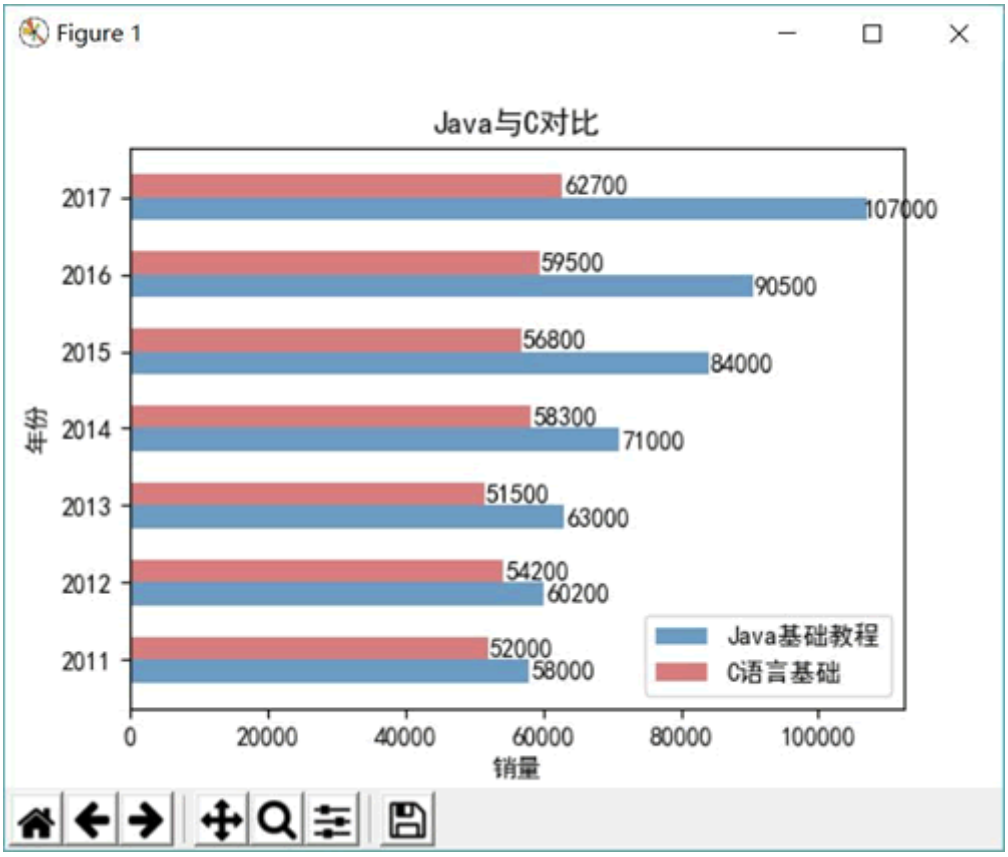


图 3 水平柱状图

8.Python Matplotlib scatter

散点图和折线图需要的数组非常相似，区别是折线图会将各数据点连接起来；而散点图则只是描绘各数据点，并不会将这些数据点连接起来。

调用 Matplotlib 的 scatter() 函数来绘制散点图，该函数支持如下常用参数：

- x：指定 X 轴数据。
- y：指定 Y 轴数据。
- s：指定散点的大小。

- c：指定散点的颜色。
- alpha：指定散点的透明度。
- linewidths：指定散点边框线的宽度。
- edgecolors：指定散点边框的颜色。
- marker：指定散点的图形样式。应参数支持'.'（点标记）、','（像素标记）、'o'（圆形标记）、'v'（向下三角形标记）、'^'（向上三角形标记）、'<'（向左三角形标记）、'>'（向右三角形标记）、'1'（向下三叉标记）、'2'（向上三叉标记）、'3'（向左三叉标记）、'4'（向右三叉标记）、's'（正方形标记）、'p'（五地形标记）、'*'（星形标记）、'h'（八边形标记）、'H'（另一种八边形标记）、'+'（加号标记）、'x'（x 标记）、'D'（菱形标记）、'd'（尖菱形标记）、'|'（竖线标记）、'_'（横线标记）等值。
- cmap：指定散点的颜色映射，会使用不同的颜色来区分散点的值。

下面程序示范了如何使用 scatter() 函数来绘制散点图：

```
import matplotlib.pyplot as plt
import numpy as np

plt.figure()
# 定义从-pi 到 pi 之间的数据，平均取 64 个数据点
x_data = np.linspace(-np.pi, np.pi, 64, endpoint=True)  # ①
# 将整个 figure 分成两行两列，第三个参数表示该图形放在第 1 个网格
# 沿着正弦曲线绘制散点图
plt.scatter(x_data, np.sin(x_data), c='purple', # 设置点的颜色
    s=50, # 设置点半径
    alpha = 0.5, # 设置透明度
    marker='p', # 设置使用五边形标记
    linewidths=1, # 设置边框的线宽
```

```
14.             edgecolors=['green', 'yellow']) # 设置边框的颜色
15.         # 绘制第二个散点图（只包含一个起点），突出起点
16.         plt.scatter(x_data[0], np.sin(x_data)[0], c='red', # 设置点的颜色
17.             s=150, # 设置点半径
18.             alpha = 1) # 设置透明度
19.         # 绘制第三个散点图（只包含一个结束点），突出结束点
20.         plt.scatter(x_data[63], np.sin(x_data)[63], c='black', # 设置点的颜色
21.             s=150, # 设置点半径
22.             alpha = 1) # 设置透明度
23.         plt.gca().spines['right'].set_color('none')
24.         plt.gca().spines['top'].set_color('none')
25.         plt.gca().spines['bottom'].set_position(('data', 0))
26.         plt.gca().spines['left'].set_position(('data', 0))
27.         plt.title('正弦曲线的散点图')
28.         plt.show()
```

上面程序使用 numpy 中的 linespace() 函数创建了一个列表作为 X 轴数据，程序使用 np.sin() 函数计算一系列 sin 值作为 Y 轴数据。程序中第 9 行代码负责生成一个散点图，该散点图包含 64 个数据点。

此外，程序在第 9 行代码之后还调用了两次 scatter() 函数，这意味将会叠加两个散点图。后面两次绘制散点图的代码分别用于绘制 x_data、sin(x+data) 的第一个点和最后一个点，这样即可突出显示散点图的起点和结束点。

运行上面程序，可以看到如图 1 所示的效果。

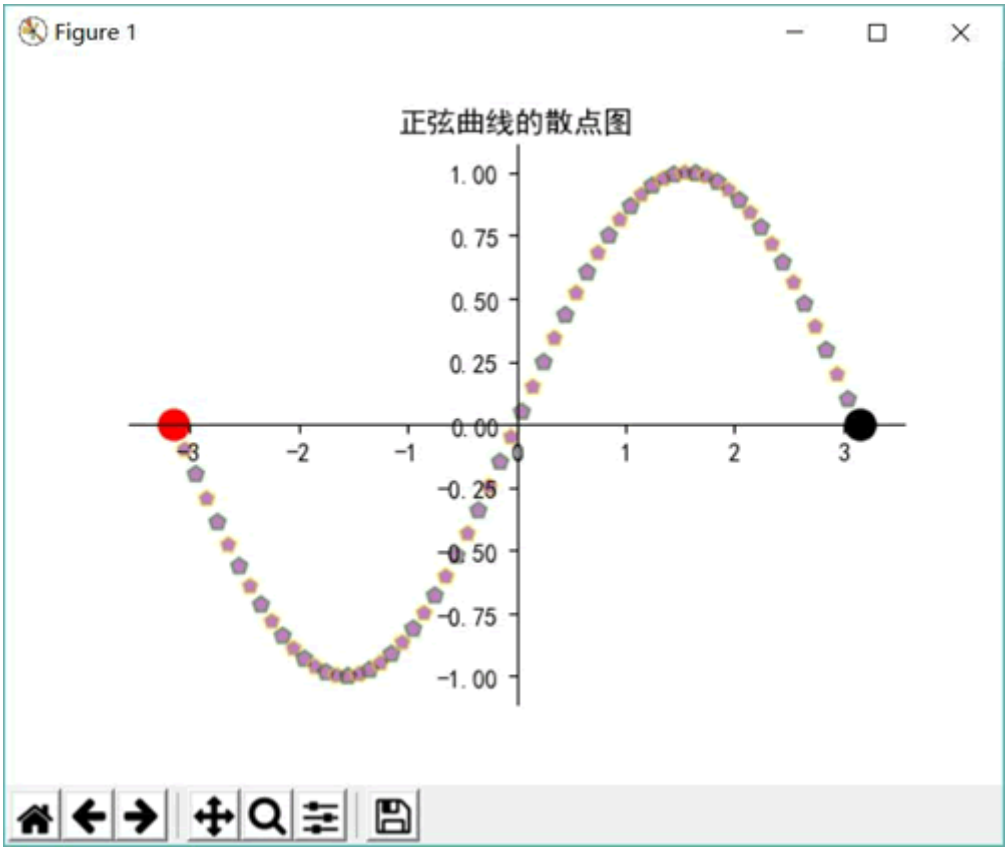


图 1 Matplotlib scatter 绘制散点图

9.Python Matplotlib contour 和 contourf

等高线图需要的是三维数据，其中 X、Y 轴数据决定坐标点，还需要对应的高度数据（相当于 Z 轴数据）来决定不同坐标点的高度。

有了合适的数据之后，程序调用 contour() 函数绘制等高线，调用 contourf() 函数为等高线图填充颜色。

在调用 contour()、contourf() 函数时可以指定如下常用参数：

- x：指定 X 轴数据。
- y：指定 Y 轴数据。
- z：指定 X、Y 坐标对应点的高度数据。
- colors：指定不同高度的等高线的颜色。
- alpha：指定等高线的透明度。
- cmap：指定等高线的颜色映射，即自动使用不同的颜色来区分不同的高度区域。
- linewidths：指定等高线的宽度。
- linestyles：指定等高线的样式。

下面程序使用 contour()、contourf() 函数来绘制等高线图：

```
import matplotlib.pyplot as plt
import numpy as np

delta = 0.025
# 生成代表 X 轴数据的列表
x = np.arange(-3.0, 3.0, delta)
# 生成代表 Y 轴数据的列表
y = np.arange(-2.0, 2.0, delta)
# 对 x、y 数据执行网格化
X, Y = np.meshgrid(x, y)
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
# 计算 Z 轴数据（高度数据）
Z = (Z1 - Z2) * 2
# 为等高线图填充颜色，16 指定将等高线分为几部分
plt.contourf(x, y, Z, 16, alpha = 0.75,
    cmap='rainbow')  # 使用颜色映射来区分不同高度的区域
# 绘制等高线
```

```
19.         C = plt.contour(x, y, Z, 16,
20.             colors = 'black', # 指定等高线的颜色
21.             linewidth = 0.5) # 指定等高线的线宽
22.         # 绘制等高线数据
23.         plt.clabel(C, inline = True, fontsize = 10)
24.         # 去除坐标轴
25.         plt.xticks(())
26.         plt.yticks(())
27.         # 设置标题
28.         plt.title("等高线图")
29.         # 为两条坐标轴设置名称
30.         plt.xlabel("纬度")
31.         plt.ylabel("经度")
32.         plt.show()
```

上面程序中，第 16 行代码用于为等高线图填充颜色，此处指定了 cmap 参数，这意味着程序将会使用不同的颜色映射来区分不同高度的区域。

程序中第 19 行代码调用 contour() 函数来绘制等高线。运行上面程序，可以看到如图 1 所示的效果。

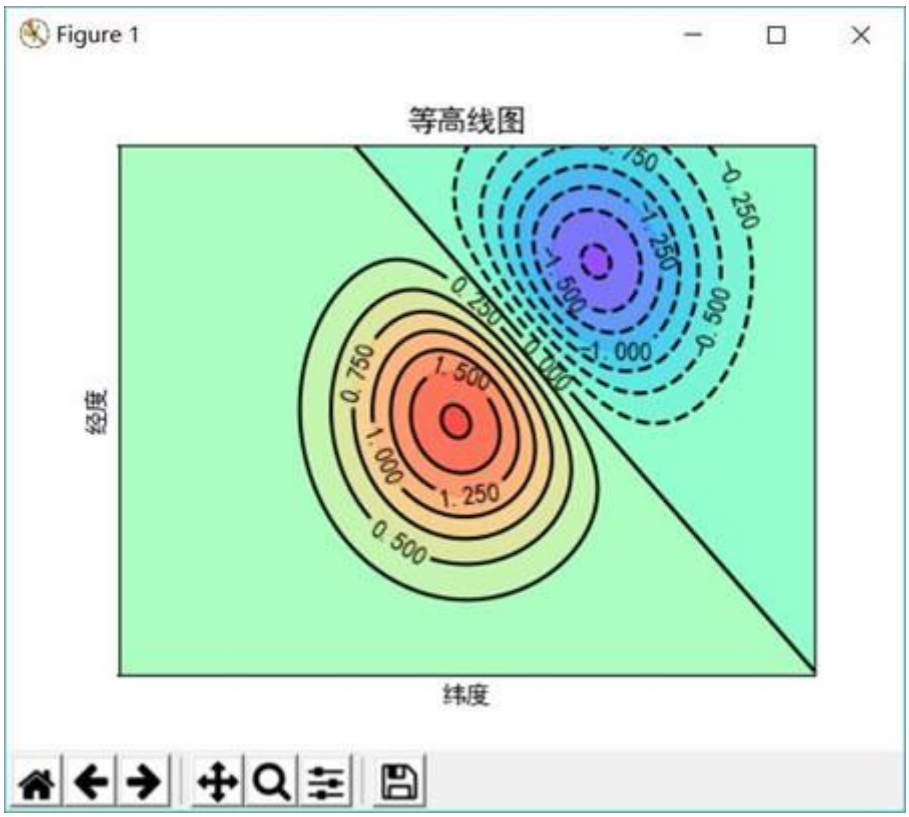


图 1 contour 和 contourf 绘制等高线图

10.Python plot_surface

3D 图形需要的数据与等高线图基本相同：X、Y 数据决定坐标点，Z 轴数据决定 X、Y 坐标点对应的高度。与等高线图使用等高线来代表高度不同，3D 图形将会以更直观的形式来表示高度。

为了绘制 3D 图形，需要调用 Axes3D 对象的 plot_surface() 方法来完成。

下面程序将使用与前面等高线图相同的数据来绘制 3D 图形，此时将看到程序会以更直观的形式来显示高度。

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(12, 8))
ax = Axes3D(fig)

delta = 0.125
# 生成代表 X 轴数据的列表
x = np.arange(-3.0, 3.0, delta)
# 生成代表 Y 轴数据的列表
y = np.arange(-2.0, 2.0, delta)
# 对 x、y 数据执行网格化
X, Y = np.meshgrid(x, y)
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
# 计算 Z 轴数据（高度数据）
Z = (Z1 - Z2) * 2
# 绘制 3D 图形
ax.plot_surface(X, Y, Z,
    rstride=1,  # rstride（row）指定行的跨度
    cstride=1,  # cstride(column)指定列的跨度
    cmap=plt.get_cmap('rainbow'))  # 设置颜色映射
# 设置 Z 轴范围
ax.set_zlim(-2, 2)
# 设置标题
plt.title("3D 图")
plt.show()
```

上面程序开始准备了和前一个程序相同的数据，只是该程序将 delta 设置为 0.125，这样可以避免生成太多的数据点（在绘制 3D 图形时，计算开销较大，如果数据点太多，Matplotlib 将会很卡）。

程序中第 20 行代码调用 Axes3D 对象的 plot_surface() 方法来绘制 3D 图形，其中 X、Y 参数负责确定坐标点，Z 参数决定 X、Y 坐标点的高度数据。

运行上面程序，可以看到如图 1 所示的 3D 图形。

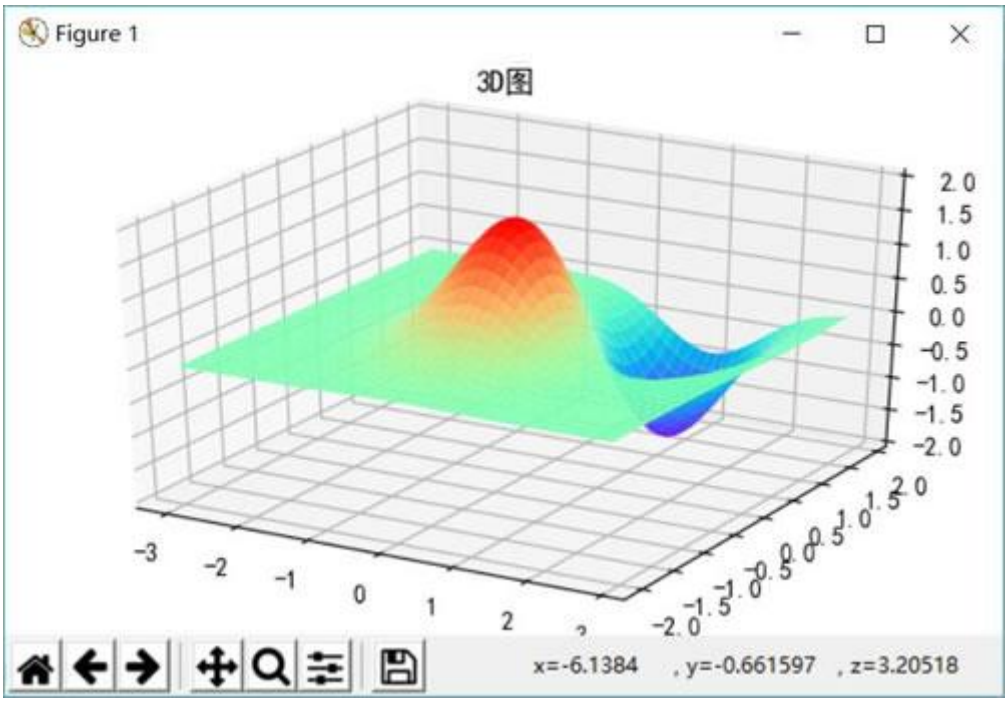


图 1 plot_surface 绘制 3D 图形

11.Python Pygal

Pygal 是另一个简单易用的数据图库，它以面向对象的方式来创建各种数据图，而且使用 Pygal 可以非常方便地生成各种格式的数据图，包括 PNG、SVG 等。使用 Pygal 也可以生成 XML etree、HTML 表格（这些都需要安装其他包）。

Pygal 模块安装

安装 Pygal 包与安装其他 Python 包基本相同，同样可以使用 pip 来安装。

启动命令行窗口，在命令行窗口中输入如下命令：

```
pip install pygal
```

上面命令将会自动安装 Pygal 包的最新版本。运行上面命令，可以看到程序先下载 Pygal 包，然后提示 Pygal 包安装成功：

```
Installing collected packages:pygal
Successfully installed pygal-2.4.0
```

如果在命令行窗口中提示找不到 pip 命令，则也可以通过 python 命令运行 pip 模块来安装 Pygal。例如，通过如下命令来安装 Pygal 包：

```
python -m pip install pygal
```

在成功安装 Pygal 包之后，可以通过 pydoc 来查看 Pygal 包的文档。在命令行窗口中输入如下命令：

```
python -m pydoc -p 8899
```

运行上面命令之后，打开浏览器查看 http://localhost:8899/ 页面，可以在 Python 安装目录的 `lib\site-packages` 下看到 Pygal 包的文档，如图 1 所示。

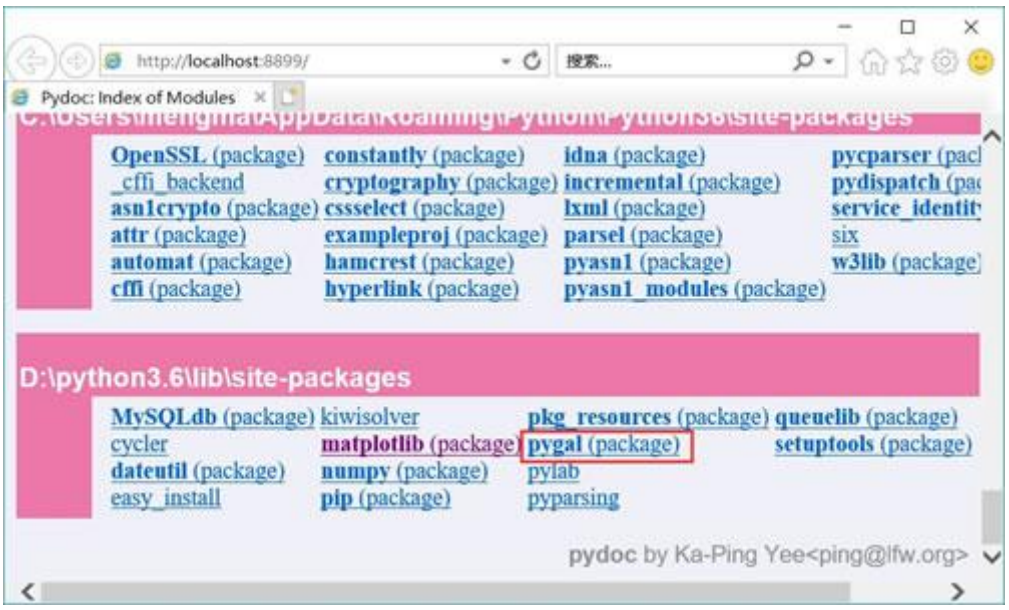


图 1 Pygal 包的文档

单击图 1 所示页面上的“pygal(package)”链接，将可以看到如图 2 所示的 API 页面。

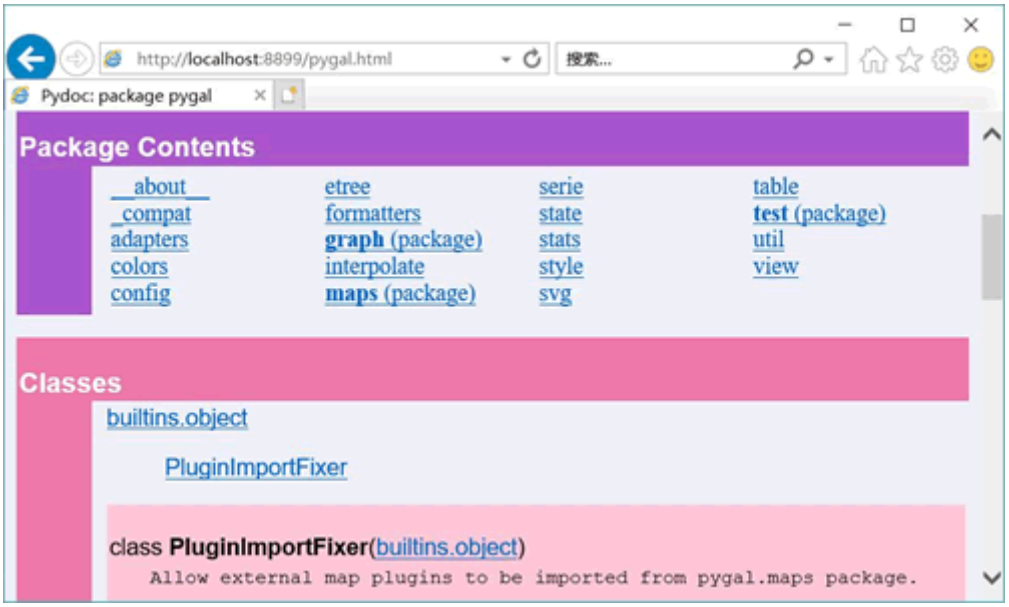


图 2 Pygal 包的 API 页面

通过图 2 所示的页面，即可查看 Pygal 包下的子模块和类。

Pygal 数据图入门

Pygal 使用面向对象的方式来生成数据图。使用 Pygal 生成数据图的步骤大致如下：

1、创建 Pygal 数据图对象。Pygal 为不同的数据图提供了不同的类，比如柱状图使用 pygal.Bar 类，饼图使用 pygal.Pie 类，折线图使用 pygal.Line 类，等等。

2、调用数据图对象的 add() 方法添加数据。

3、调用 Config 对象的属性配置数据图。

4、调用数据图对象的 render_to_xxx() 方法将数据图渲染到指定的输出节点（此处的输出节点可以是 PNG 图片、SVG 文件，也可以是其他节点）。

下面通过生成简单的柱状图来演示如何使用 Pygal 生成数据图，该柱状图展示了两种教程从 2011 年到 2017 年的销量统计数据：

```
1.          import pygal
2.
3.          x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']
4.          # 定义 2 个列表分别作为两组柱状图的 Y 轴数据
5.          y_data = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
6.          y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
7.          # 创建 pygal.Bar 对象（柱状图）
8.          bar = pygal.Bar()
9.          # 添加两组代表条柱的数据
10.         bar.add('C 语言基础', y_data)
11.         bar.add('Python 语言基础', y_data2)
12.         # 设置 X 轴的刻度值
13.         bar.x_labels = x_data
14.         bar.title = '编程教程的历年销量'
15.         # 设置 X、Y 轴的标题
16.         bar.x_title = '年份'
```

```
17.         bar.y_title = '销量'
18.         # 指定将数据图输出到 SVG 文件中
19.         bar.render_to_file('fk_books.svg')
```

上面程序中第 8 行代码创建了 pygal.Bar 对象，该对象就代表一个柱状图。接下来的两行代码为 pygal.Bar 对象添加了两组柱状图数据。

通过上面程序，实际上已经可以生成简单的柱状图了。如果注释掉后面对 pygal.Bar 对象的属性赋值的代码，运行该程序，将可以看到在程序当前目录下生成了一个 fk_books.svg 文件，使用浏览器查看该文件，可以看到如图 3 所示的柱状图。

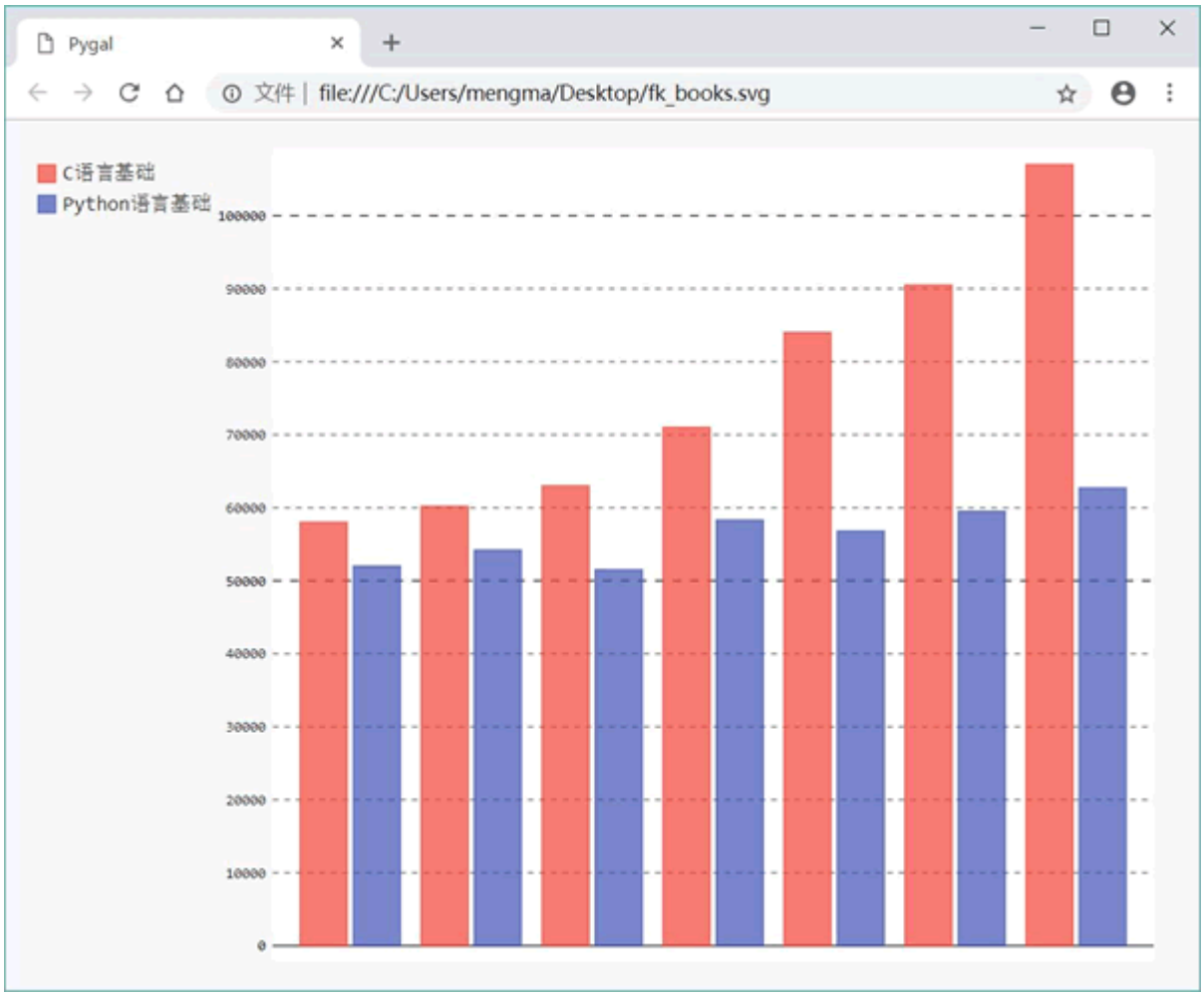


图 3 简单的柱状图

从图 3 所示的柱状图可以看到，这个数据图的 X 轴没有刻度值，X 轴、Y 轴没有名称，它们都可以通过 pygal.Bar 对象来配置。接下来程序为 pygal.Bar 对象的 title、x_labels、x_title、y_title 属性赋值，已经属于配置数据图的部分了，其分别配置了数据图的标题、X 轴的刻度值、X 轴的名称、Y 轴的名称。

在增加上面的配置代码之后，再次运行该程序，程序会再次生成一个 SVG 文件。由于 SVG 文件支持交互，因此，当用户把鼠标指针移到某个条柱上时，将可以看到关于该条柱的信息，如图 4 所示。

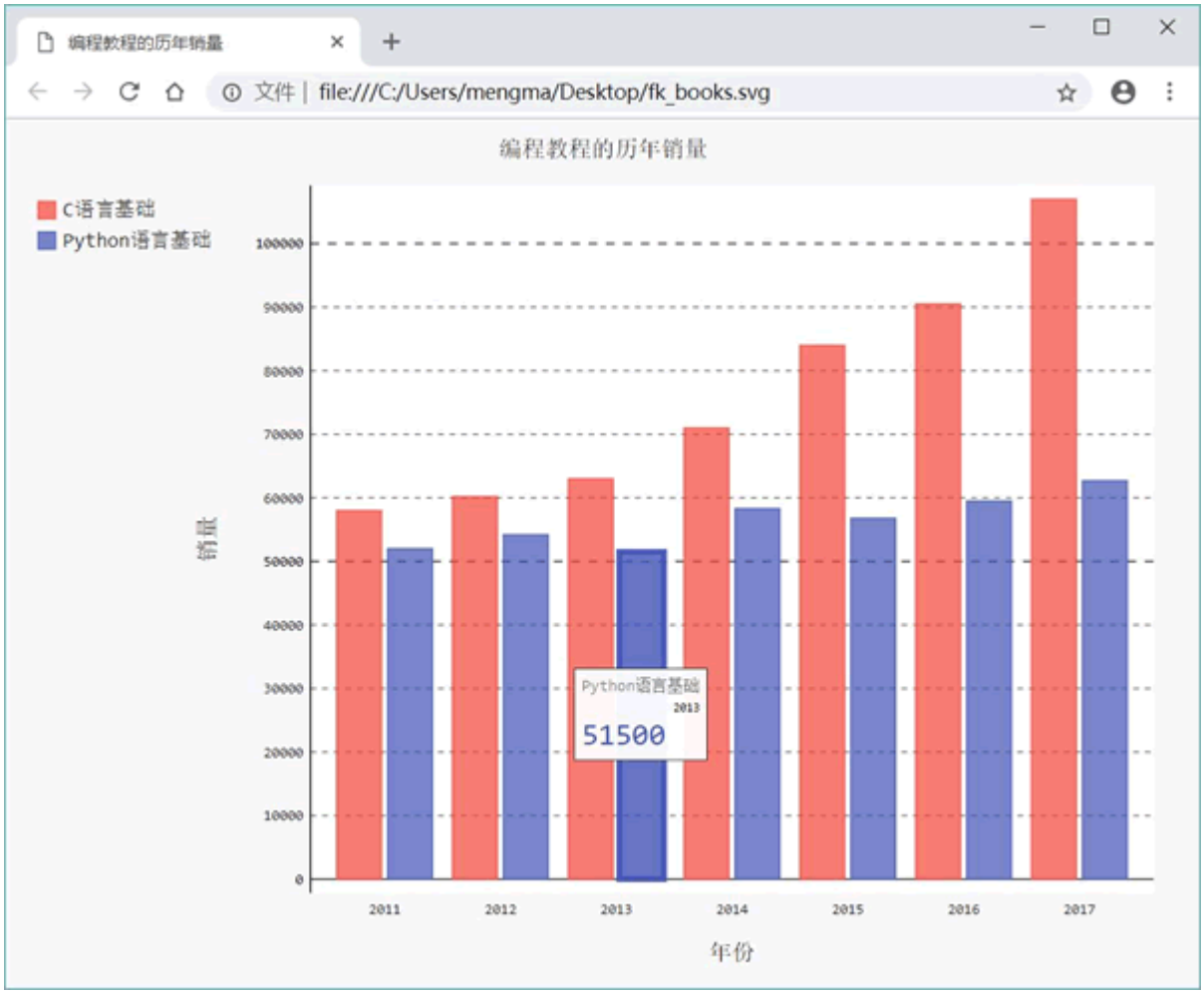


图 4 支持交互的 SVG 数据图

配置 Pygal 数据图

正如从前面程序所看到的，使用 pygal.Bar 生成数据图之后，程序可以通过对该对象的属性赋值来配置数据图。那么，除设置上面这些简单的属性之外，是否还可以设置其他属性呢？

答案是肯定的，查阅 http://localhost:8899/pygal.config.html 页面（其中 8899 是运行 pydoc 的端口），可以看到 config 模块的相关说明，该模块包含了 BaseConfig、CommonConfig、Config、SerieConfig 等配置类，这些类所包含的属性正是用于配置 Pygal 数据图的。

下面程序示范了该页面中部分配置属性的作用：

```
import pygal

x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']
# 定义 2 个列表分别作为两组柱状图的 Y 轴数据
y_data = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
# 创建 pygal.Bar 对象（柱状图）
bar = pygal.Bar()
# 添加两组代表条柱的数据
bar.add('C 语言基础', y_data)
bar.add('Python 基础', y_data2)
# 设置 X 轴的刻度值
bar.x_labels = x_data
bar.title = '编程教程的历年销量'
# 设置 X、Y 轴的标题
bar.x_title = '年份'
bar.y_title = '销量'
# 设置 X 轴的刻度值旋转 45 度
bar.x_label_rotation = 45
# 设置将图例放在底部
bar.legend_at_bottom = True
# 设置数据图四周的页边距
# 也可通过 margin_bottom、margin_left、margin_right、margin_top 只设置单独一边的页边距
bar.margin = 35
# 隐藏 X 轴上的网格线
bar.show_y_guides=False
# 显示 X 轴上的网格线
bar.show_x_guides=True
# 指定将数据图输出到 SVG 文件中
bar.render_to_file('fk_books.svg')
```

运行上面程序，将会生成如图 5 所示的数据图。

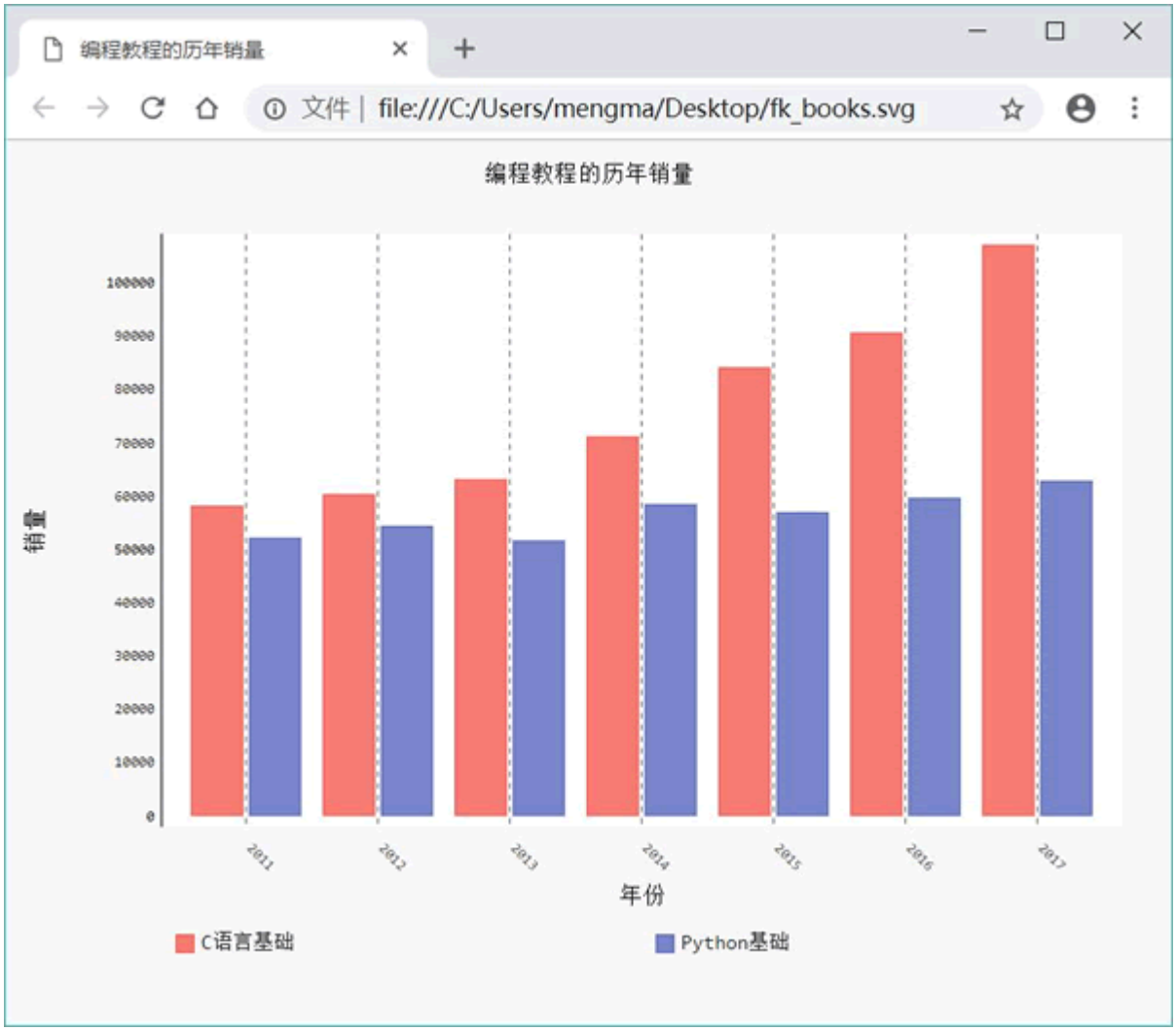


图 5 配置数据图

对比图 4 和图 5 所示的数据图，可以发现图 5 所示的数据图的 X 轴刻度值旋转了 45°，这是 x_label_rotation 属性的作用；数据图的图例被显示在底部，这是 legend_at_bottom 属性的作用；数据图不再显示水平方向的网格钱，这是 show_y_guides 属性的作用；数据图显示垂直方向的网格线，这是 show_x_guides 属性的作用。

对于不同的数据图，Pygal 支持大量对应的配置，具体可结合 http://localhost:8899/pygal.config.html 页面给出的属性进行设置、测试，此处不再一一讲解。

12.Python Pygal 常见数据图

Python Pygal 常见数据图（折线图、柱状图、饼图、点图、仪表图和雷达图）详解

Pygal 同样支持各种不同的数据图，比如饼图、折线图等。Pygal 的设计很好，不管是创建哪种数据图，Pygal 的创建方式基本是一样的，都是先创建对应的数据图对象，然后添加数据，最后对数据图进行配置。因此，使用 Pygal 生成数据图是比较简单的。

折线图

折线图与柱状图很像，它们只是表现数据的方式不同，柱状图使用条柱代表数据，而折线图则使用折线点来代表数据。因此，生成折线图的方式与生成柱状图的方式基本相同。

使用 pygal.Line 类来表示折线图，程序创建 pygal.Line 对象就是创建折线图。下面程序示范了利用折线图来展示两套教程销量统计数据的方法。

```
1.          import pygal
2.
3.          x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']
4.          # 构造数据
5.          y_data = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
6.          y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
7.          # 创建 pygal.Line 对象（折线图）
```

```
8.          line = pygal.Line()
9.          # 添加两组代表折线的数据
10.         line.add('C 语言教程', y_data)
11.         line.add('Python 教程', y_data2)
12.         # 设置 X 轴的刻度值
13.         line.x_labels = x_data
14.         # 重新设置 Y 轴的刻度值
15.         line.y_labels = [20000, 40000, 60000, 80000, 100000]
16.         line.title = '编程教程的历年销量'
17.         # 设置 X、Y 轴的标题
18.         line.x_title = '年份'
19.         line.y_title = '销量'
20.         # 设置将图例放在底部
21.         line.legend_at_bottom = True
22.         # 指定将数据图输出到 SVG 文件中
23.         line.render_to_file('fk_books.svg')
```

上面程序中的第 8 行代码创建了 pygal.Line 对象，该对象代表折线图。接下来程序调用 pygal.Line 对象的 add() 方法添加统计数据，然后对数据图进行配置。

运行上面程序，将会生成如图 1 所示的折线图。

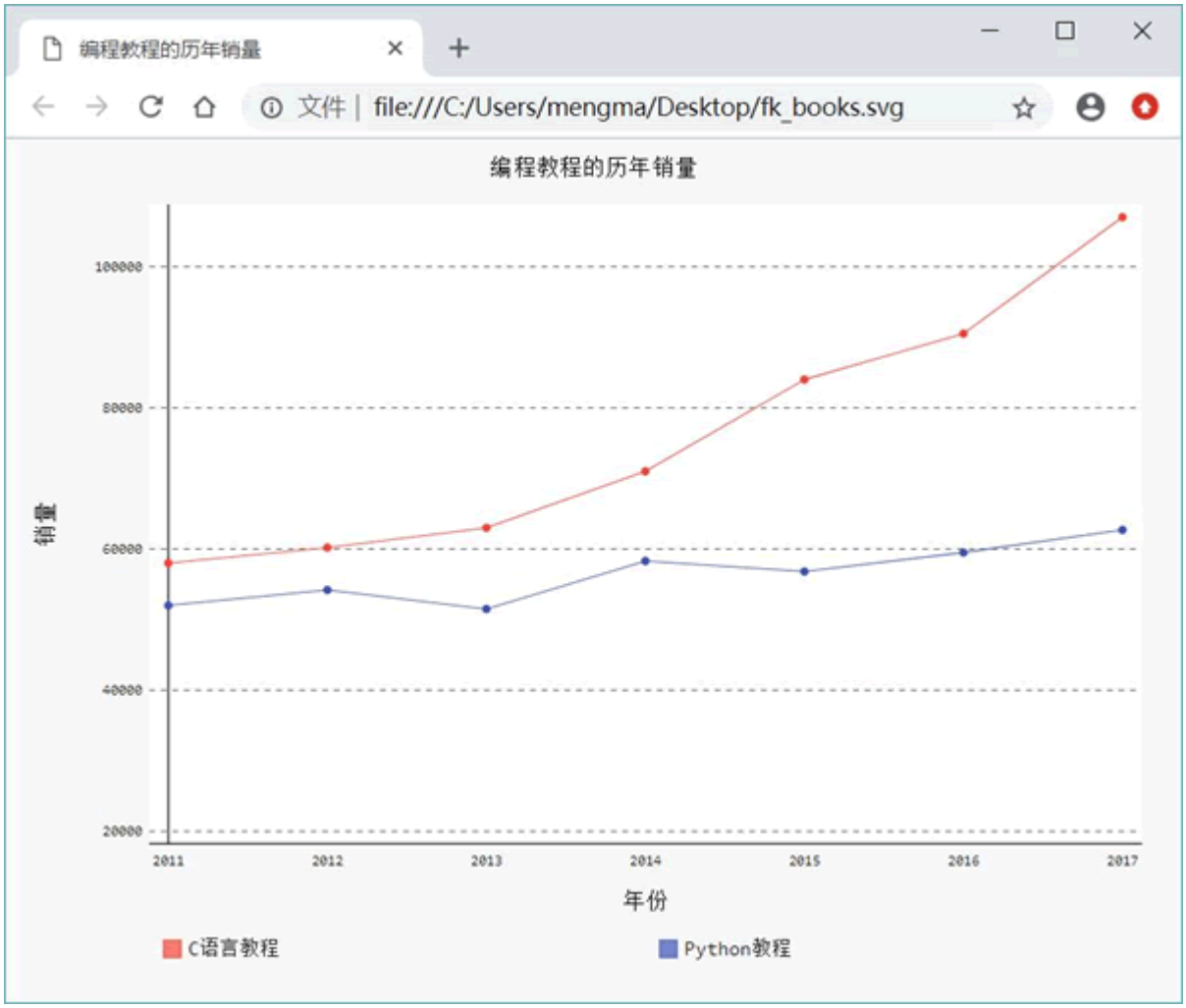


图 1 折线图

水平柱状图和水平折线图

使用 pygal.HorizontalBar 类来表示水平柱状图。使用 pygal.HorizontalBar 生成水平柱状图的步骤与创建普通柱状图的步骤基本相同。下面程序示范了使用 pygal.HorizontalBar 生成水平柱状图来展示两套教程历年销量统计数据的方法：

```
import pygal

x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']
# 构造数据
```

```
5.          y_data = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
6.          y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
7.          # 创建 pygal.HorizontalBar 对象（水平柱状图）
8.          horizontal_bar = pygal.HorizontalBar()
9.          # 添加两组数据
10.         horizontal_bar.add('C 语言教程', y_data)
11.         horizontal_bar.add('Python 教程', y_data2)
12.         # 设置 Y 轴（确实如此）的刻度值
13.         horizontal_bar.x_labels = x_data
14.         # 重新设置 X 轴（确实如此）的刻度值
15.         horizontal_bar.y_labels = [20000, 40000, 60000, 80000, 100000]
16.         horizontal_bar.title = '编程教程的历年销量'
17.         # 设置 X、Y 轴的标题
18.         horizontal_bar.x_title = '销量'
19.         horizontal_bar.y_title = '年份'
20.         # 设置将图例放在底部
21.         horizontal_bar.legend_at_bottom = True
22.         # 指定将数据图输出到 SVG 文件中
23.         horizontal_bar.render_to_file('fk_books.svg')
```

上面程序中第 8 行代码创建 pygal.HorizontalBar 对象作为水平柱状图，这与前面创建普通柱状图并无差别。在设置 pygal.HorizontalBar 对象时有一点需要注意，x_labels 属性用于设置 Y 轴的刻度值，而 y_labels 属性用于设置 X 轴的刻度值。

运行上面程序，将会生成如图 2 所示的水平柱状图。

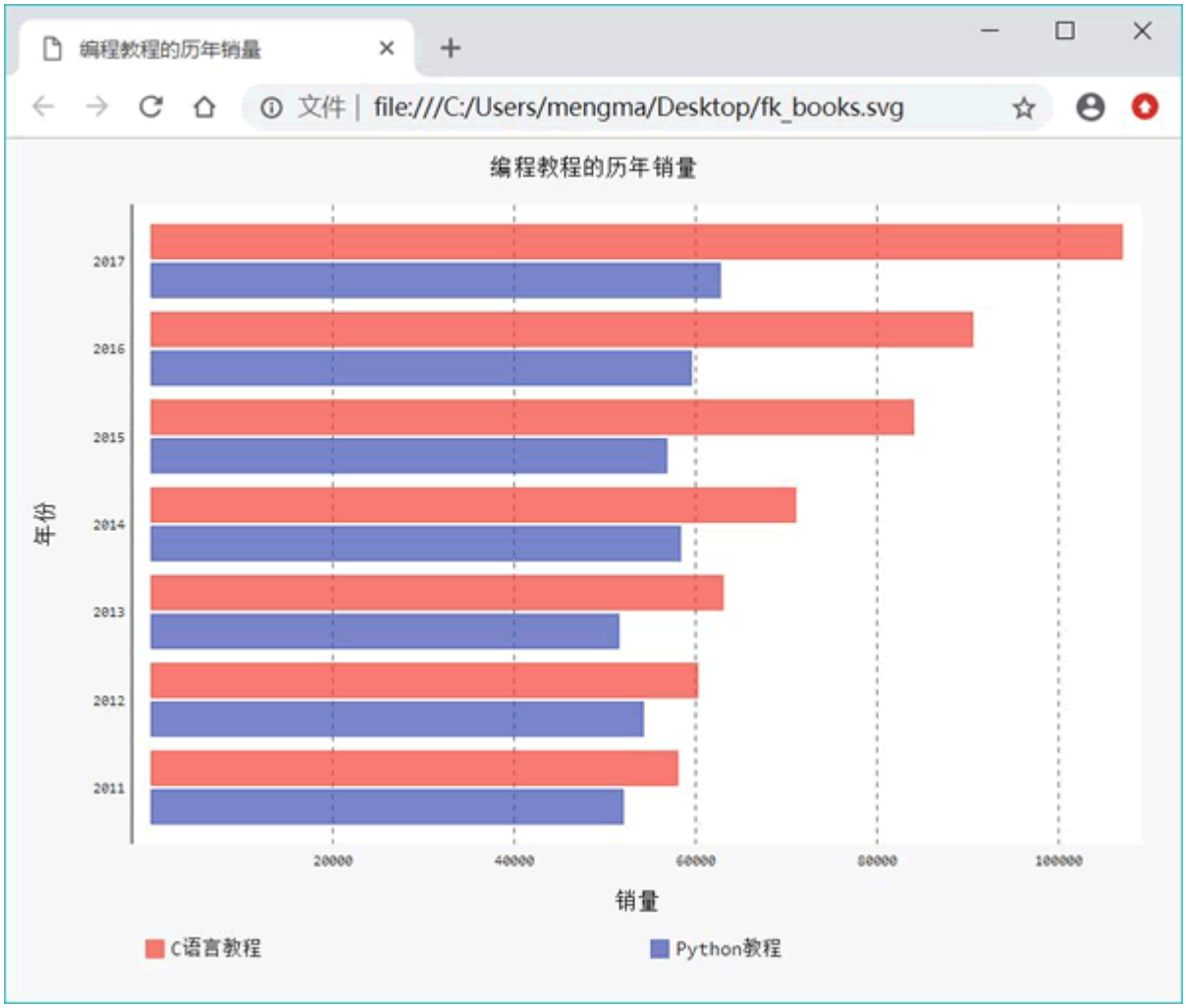


图 2 水平柱状图

与水平柱状图类似的还有水平折线图，水平折线图使用 pygal.HorizontalLine 类来表示，水平折线图的 X 轴刻度值同样使用 y_labels 属性来设置，而 Y 轴刻度值才使用 x_labels 属性来设置。

例如如下程序所示：

```
1.          import pygal
2.
```

```
x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']
# 构造数据
y_data = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
# 创建 pygal.HorizontalLine 对象（水平折线图）
horizontal_line = pygal.HorizontalLine()
# 添加两组代表折线的数据
horizontal_line.add('C 语言教程', y_data)
horizontal_line.add('Python 教程', y_data2)
# 设置 Y 轴（确实如此）的刻度值
horizontal_line.x_labels = x_data
# 重新设置 X 轴（确实如此）的刻度值
horizontal_line.y_labels = [20000, 40000, 60000, 80000, 100000]
horizontal_line.title = '编程教程的历年销量'
# 设置 X、Y 轴的标题
horizontal_line.x_title = '销量'
horizontal_line.y_title = '年份'
# 设置将图例放在底部
horizontal_line.legend_at_bottom = True
# 指定将数据图输出到 SVG 文件中
horizontal_line.render_to_file('fk_books.svg')
```

叠加柱状图和叠加折线图

有些时候，客户重点关心的不是两个产品在同一年的销量对比（应该使用普通柱状图），而是两个产品的累计销量，此时应该使用叠加柱状图或叠加折线图。

对于叠加柱状图而言，代表第二组数据的条性会叠加在代表第一组数据的条柱上，这样可以更方便地看到两组数据的累加结果。叠加柱状图使用 pygal.StackedBar 类来表示，程序使用 pygal.StackedBar 创建叠加柱状图的步骤与创建普通柱状图的步骤基本相同。

下面程序示范了使用 pygal.StackedBar 创建叠加柱状图来展示两种图书销量数据汇总的方法：

```
1.          import pygal
2.
3.          x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']
4.          # 构造数据
5.          y_data = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
6.          y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
7.          # 创建 pygal.StackedBar 对象（叠加柱状图）
8.          stacked_bar = pygal.StackedBar()
9.          # 添加两组数据
10.         stacked_bar.add('C 语言教程', y_data)
11.         stacked_bar.add('Python 教程', y_data2)
12.         # 设置 X 轴的刻度值
13.         stacked_bar.x_labels = x_data
14.         # 重新设置 Y 轴的刻度值
15.         stacked_bar.y_labels = [20000, 40000, 60000, 80000, 100000]
16.         stacked_bar.title = '编程教程的历年销量'
17.         # 设置 X、Y 轴的标题
18.         stacked_bar.x_title = '销量'
19.         stacked_bar.y_title = '年份'
20.         # 设置将图例放在底部
21.         stacked_bar.legend_at_bottom = True
22.         # 指定将数据图输出到 SVG 文件中
23.         stacked_bar.render_to_file('fk_books.svg')
```

上面程序中，第 8 行代码创建了 pygal.StackedBar 对象，该对象就代表一个叠加柱状图。

接下来程序同样先为叠加柱状图添加数据，然后配置叠加柱状图。

运行上面程序，将会生成如图 3 所示的叠加柱状图。

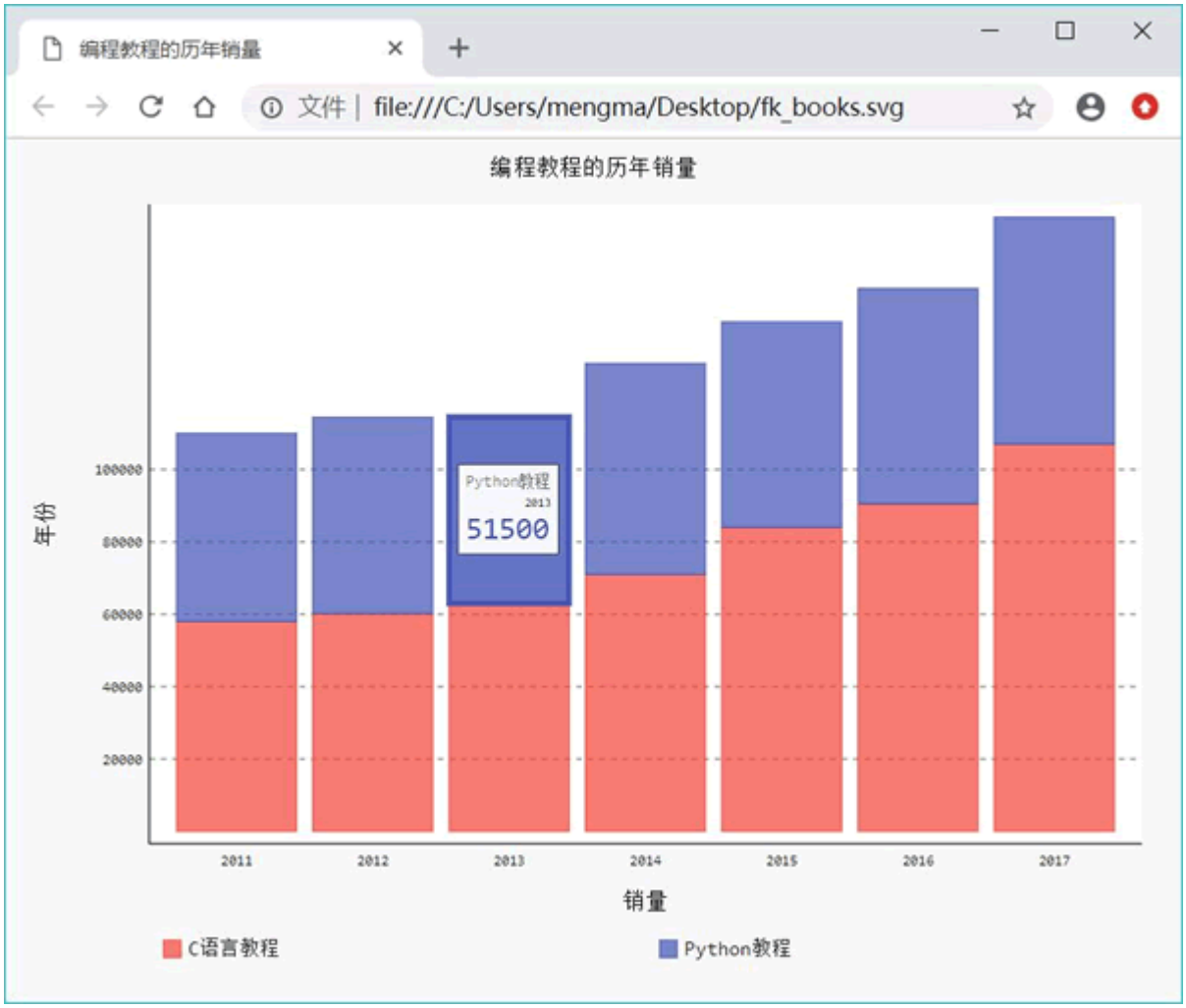


图 3 叠加柱状图

从图 3 可以看到，代表第二组数据的条柱叠加在代表第一组数据的条柱上。

与叠加柱状图类似的还有叠加折线图，叠加折线图使用 pygal.StackedLine 类来表示，叠加折线图的第二组折线的数据点同样叠加在第一组折线的数据点上。

关于叠加折线图，可参考如下程序：

```
import pygal

x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']
# 构造数据
y_data = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
# 创建 pygal.StackedBar 对象（叠加折线图）
stacked_line = pygal.StackedLine()
# 添加两组数据
stacked_line.add('C 语言教程', y_data)
stacked_line.add('Python 教程', y_data2)
# 设置 X 轴的刻度值
stacked_line.x_labels = x_data
# 重新设置 Y 轴的刻度值
stacked_line.y_labels = [20000, 40000, 60000, 80000, 100000]
stacked_line.title = '编程教程的历年销量'
# 设置 X、Y 轴的标题
stacked_line.x_title = '销量'
stacked_line.y_title = '年份'
# 设置将图例放在底部
stacked_line.legend_at_bottom = True
# 指定将数据图输出到 SVG 文件中
stacked_line.render_to_file('fk_books.svg')
```

对应的是， 如果客户需要让叠加柱状图和叠加折线图以水平方式显示，则 Pygal 提供了 pygal.HorizontalStackedBar 和 pygal.HorizontalStackedLine 类来生成水平叠加柱状图和水平叠加折线图。

饼图

Pygal 提供了 pygal.Pie 类来支持饼图，程序在创建 pygal.Pie 对象之后，同样需要调用 add() 方法来添加统计数据。

pygal.Pie 对象支持如下两个特有的属性：

- inner_radius：设置饼图内圈的半径。通过设置该属性可实现环形数据图。
- half_pie：将该属性设置为 True，可实现半圆的饼图。

下面程序示范了使用饼图来展示 2018 年 8 月编程语言的统计数据：

```
import pygal

# 准备数据
data = [0.16881, 0.14966, 0.07471, 0.06992,
    0.04762, 0.03541, 0.02925, 0.02411, 0.02316, 0.01409, 0.36326]
# 准备标签
labels = ['Java', 'C', 'C++', 'Python',
    'Visual Basic .NET', 'C#', 'PHP', 'JavaScript',
    'SQL', 'Assembly langugage', '其他']
# 创建 pygal.Pie 对象（饼图）
pie = pygal.Pie()
# 采用循环为饼图添加数据
for i, per in enumerate(data):
    pie.add(labels[i], per)
pie.title = '2018 年 8 月编程语言'
# 设置将图例放在底部
pie.legend_at_bottom = True
# 设置内圈的半径长度
pie.inner_radius = 0.4
# 创建半圆数据图
pie.half_pie = True
# 指定将数据图输出到 SVG 文件中
pie.render_to_file('language_percent.svg')
```

上面程序中，第 11 行代码创建了一个 pygal.Pie 对象，该对象就表示一个饼图。接下来程序使用循环为饼图添加了数据。程序中第 19 行代码设置 pygal.Pie 的 inner_radius 半径为 0.4，这表明将该饼图设为空心环；第 21 行代码设置 pygal.Pie 的 half_pie 为 True，这表明将该饼图设为半圆。

如果将上面程序中第 19、21 两行代码注释掉，运行该程序，将会生成如图 4 所示的传统饼图。

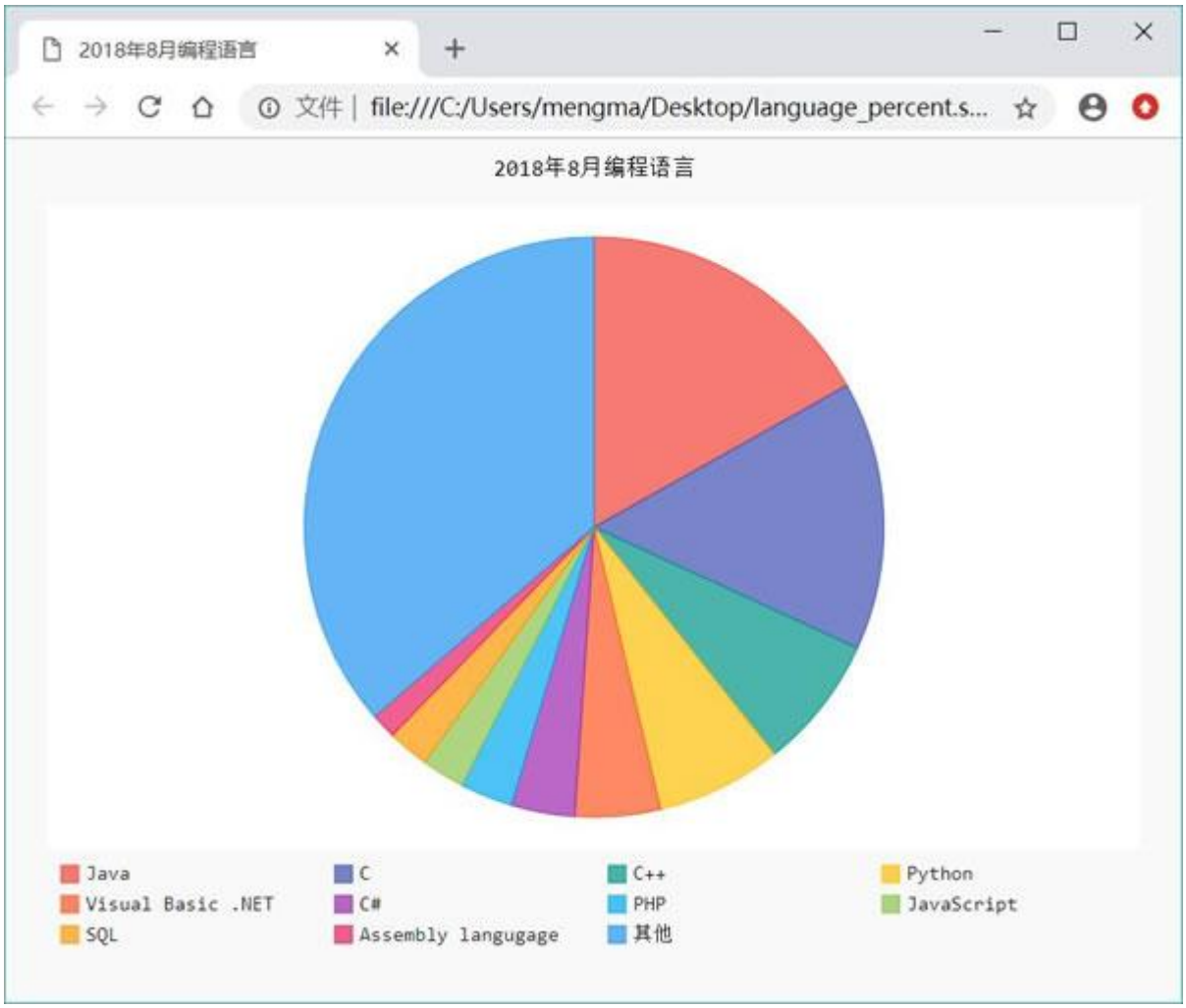


图 4 传统饼图图

如果取消这两行代码的注释，程序将会生成空心的半圆饼图，如图 5 所示。

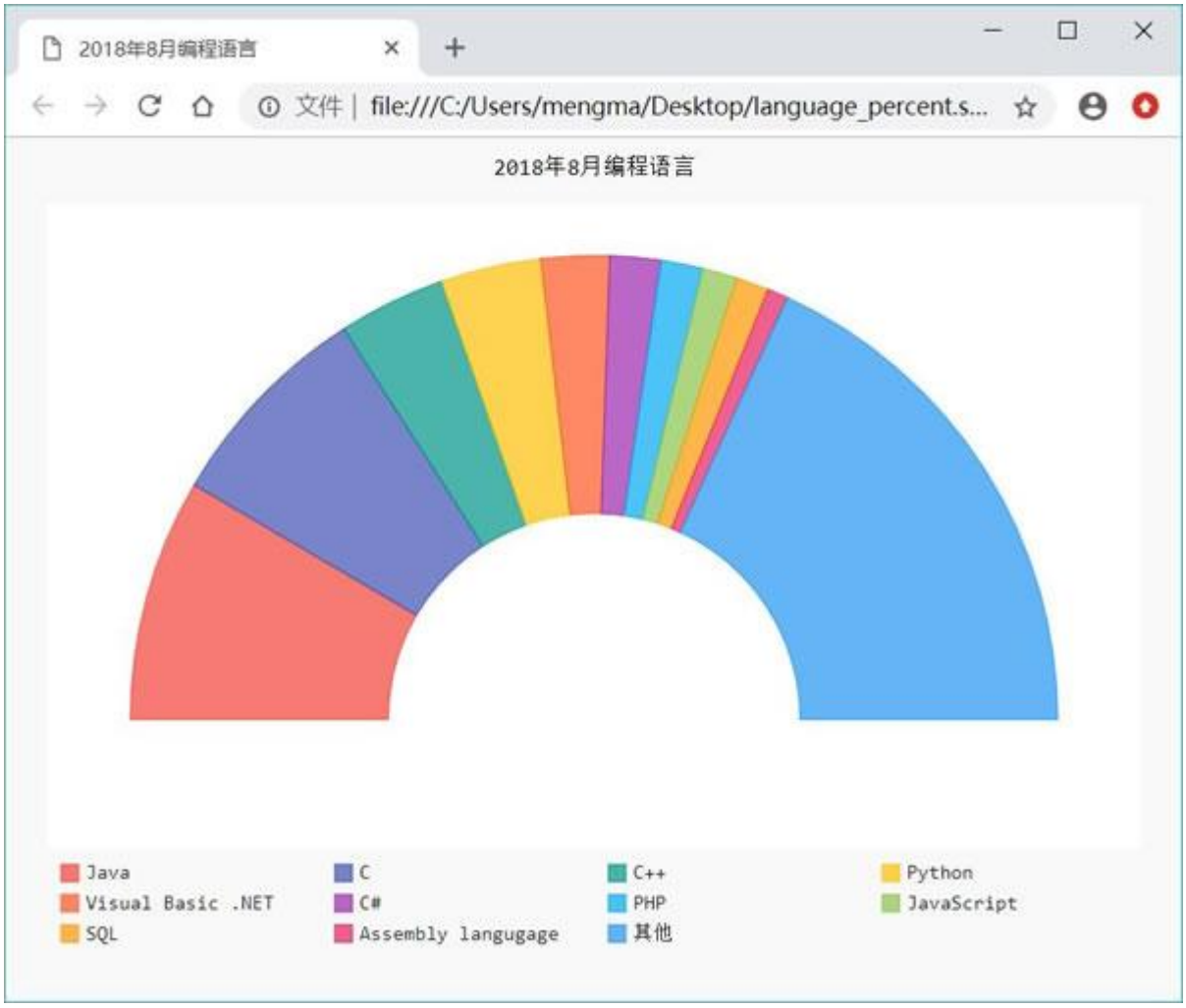


图 5 空心的半圆饼图

点图

与柱状图使用条柱高度来代表数值的大小不同，点图使用点(圆)的大小来表示数值的大小。

Pygal 使用 pygal.Dot 类表示点图，创建点图的方式与创建柱状图的方式基本相同。

下面程序示范了使用点图来展示图书销量的统计数据：

```
1.          import pygal
2.
3.          x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']
4.          # 构造数据
5.          y_data = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
6.          y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
7.          # 创建 pygal.Dot 对象（点图）
8.          dot = pygal.Dot()
9.          dot.dots_size = 5
10.         # 添加两组数据
11.         dot.add('C 语言教程', y_data)
12.         dot.add('Python 教程', y_data2)
13.         # 设置 X 轴的刻度值
14.         dot.x_labels = x_data
15.         # 重新设置 Y 轴的刻度值
16.         dot.y_labels = ['C 语言教程', 'Python 教程']
17.         # 设置 Y 轴刻度值的旋转角度
18.         dot.y_label_rotation = 45
19.         dot.title = '编程教程的历年销量'
20.         # 设置 X 轴的标题
21.         dot.x_title = '年份'
22.         # 设置将图例放在底部
23.         dot.legend_at_bottom = True
24.         # 指定将数据图输出到 SVG 文件中
25.         dot.render_to_file('fk_books.svg')
```

上面程序中，第 8 行代码创建了 pygal.Dot 对象，该对象代表点图。在创建了 pygal.Dot 对象之后，程序为该对象添加要展示的数据，然后配置该点图。

运行该程序，将会生成如图 6 所示的点图。

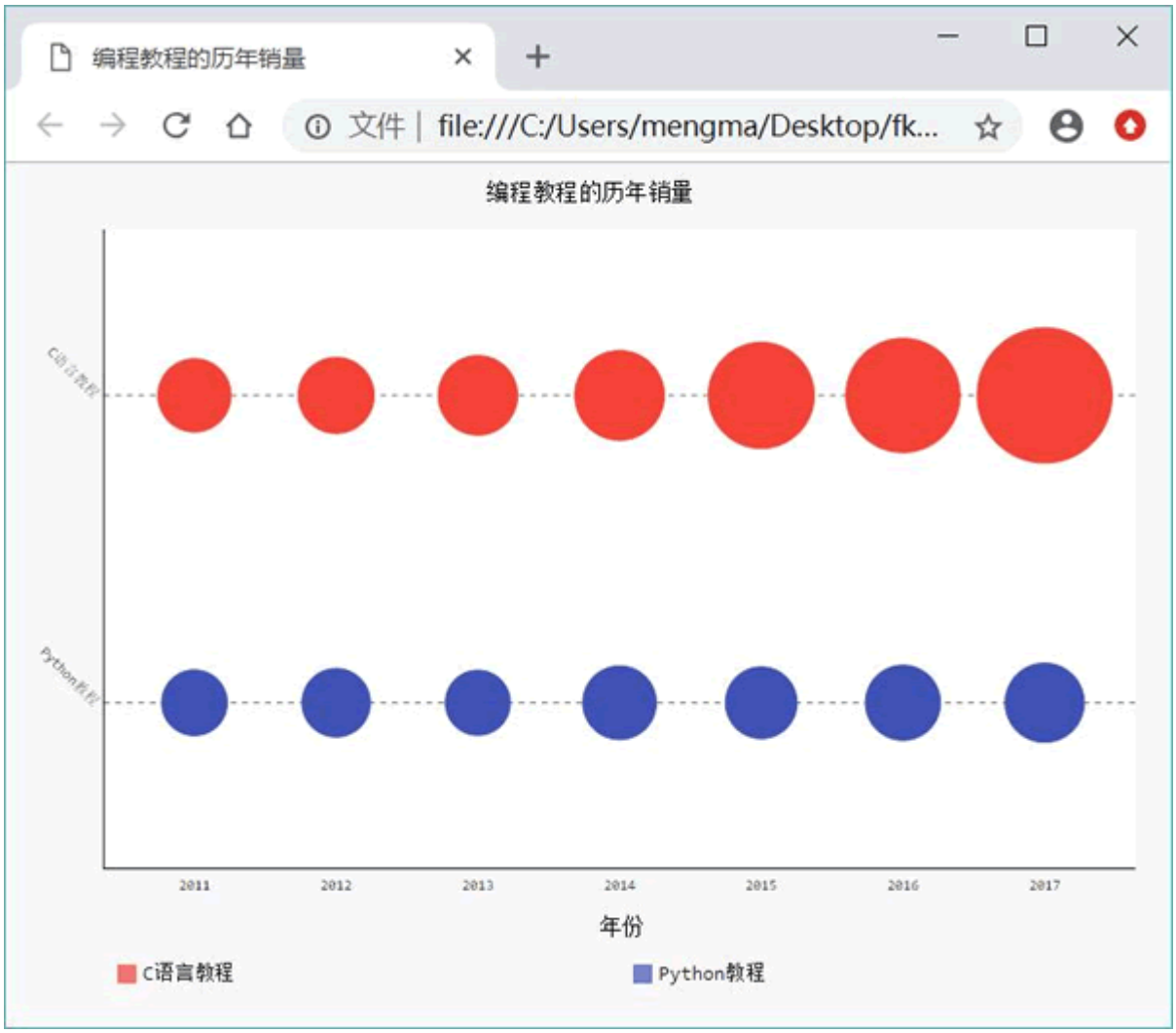


图 6 点图

仪表（Gauge）图

仪表图类似于一个仪表盘，在仪表盘内使用不同的指针代表不同的数据。Pygal 使用 pygal.Gauge 类表示仪表图。程序在创建 pygal.Gauge 对象之后，为 pygal.Gauge 对象添加数据的方式与为 pygal.Pie 对象添加数据的方式相似。

pygal.Gauge 对象有一个特别的属性：range，该属性用于指定仪表图的最小值和最大值。

下面程序示范了使用仪表图来展示各编程语言所占的市场比例：

```
1.          import pygal
2.
3.          # 准备数据
4.          data = [0.16881, 0.14966, 0.07471, 0.06992,
5.              0.04762, 0.03541, 0.02925, 0.02411, 0.02316, 0.01409, 0.36326]
6.          # 准备标签
7.          labels = ['Java', 'C', 'C++', 'Python',
8.              'Visual Basic .NET', 'C#', 'PHP', 'JavaScript',
9.              'SQL', 'Assembly langugage', '其他']
10.         # 创建 pygal.Gauge 对象（仪表图）
11.         gauge = pygal.Gauge()
12.         gauge.range = [0, 1]
13.         # 采用循环为仪表图添加数据
14.         for i, per in enumerate(data):
15.             gauge.add(labels[i], per)
16.         gauge.title = '2018 年 8 月编程语言'
17.         # 设置将图例放在底部
18.         gauge.legend_at_bottom = True
19.         # 指定将数据图输出到 SVG 文件中
20.         gauge.render_to_file('language_percent.svg')
```

上面程序中第 11 行代码创建了 pygal.Gauge 对象，接下来第 12 行代码对该对象的 range 属性赋值，将该仪表图的最大值赋值为 1，最小值赋值为 0。

运行该程序，将会生成如图 7 所示的仪表图。

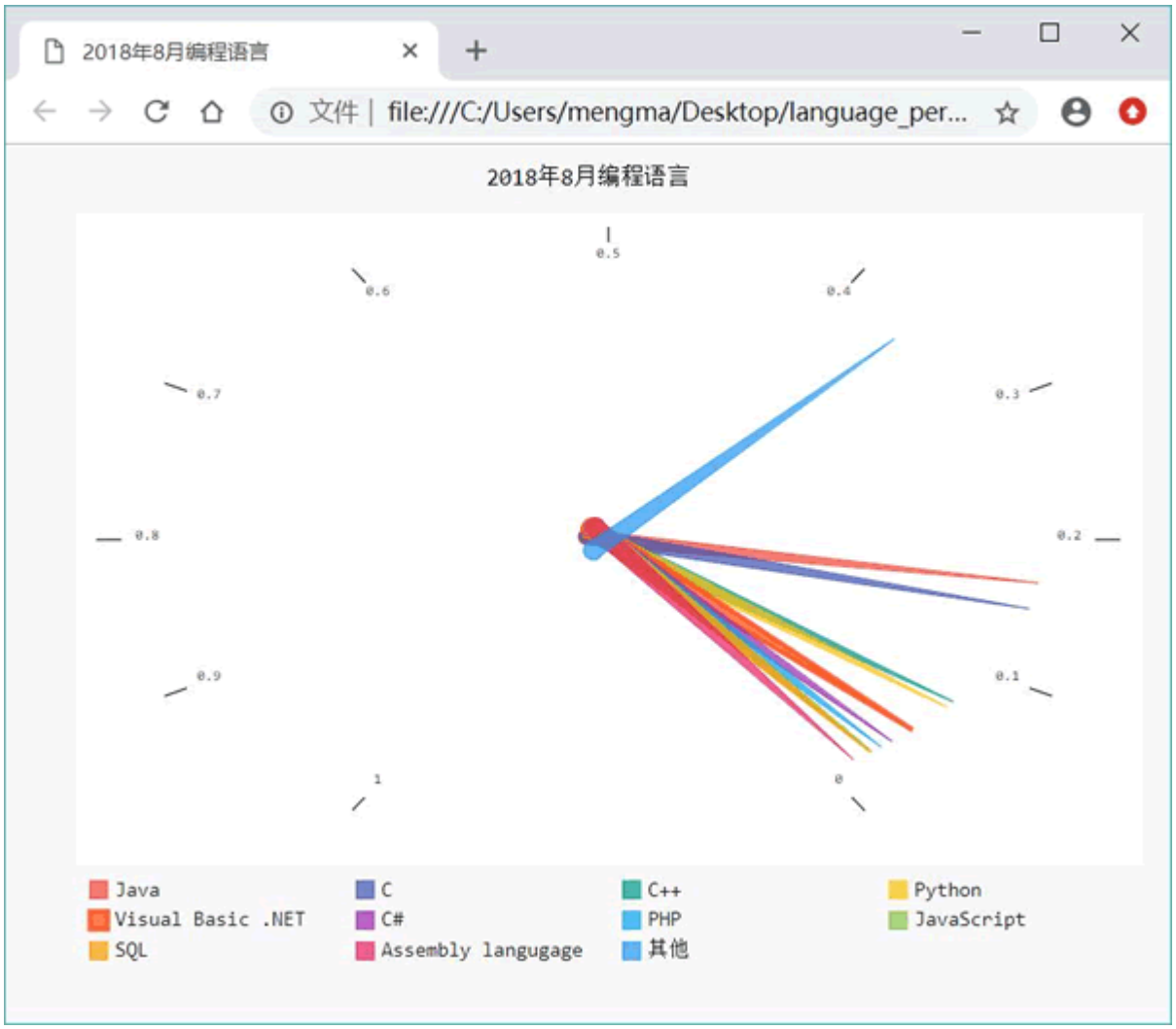


图 7 仪表图

雷达图

雷达图适合用于分析各对象在不同维度的优势和劣势，通过雷达图可对比每个对象在不同维度的得分。假如我们从表 8 所示的 5 个方面（平台健壮性、语法易用性、社区活跃度、市场份额和未来趋势）的得分来评价各编程语言的优势。

表 8 编程语言对比

	平台健壮性	语法易用性	社区活跃度	市场份额	未来趋势
Java	5	4.0	5	5	5
C	4.8	2.8	4.8	4.8	4.9
C++	4.5	2.9	4.6	4.0	4.9
Python	4.0	4.8	4.9	4.0	5
C#	3.0	4.2	2.3	3.5	2
PHP	4.8	4.3	3.9	3.0	4.5

对于表 8 所示的对比数据，我们可以使用雷达图来展示各编程语言在不同维度的优势。

```
import pygal

# 准备数据
data = [[5, 4.0, 5, 5, 5],
    [4.8, 2.8, 4.8, 4.8, 4.9],
    [4.5, 2.9, 4.6, 4.0, 4.9],
    [4.0, 4.8, 4.9, 4.0, 5],
    [3.0, 4.2, 2.3, 3.5, 2],
    [4.8, 4.3, 3.9, 3.0, 4.5]]
# 准备标签
labels = ['Java', 'C', 'C++', 'Python',
    'C#', 'PHP']
# 创建 pygal.Radar 对象（雷达图）
rader = pygal.Radar()
# 采用循环为雷达图添加数据
for i, per in enumerate(labels):
    rader.add(labels[i], data[i])
```

```
18.         rader.x_labels = ['平台健壮性', '语法易用性', '社区活跃度',
19.             '市场份额', '未来趋势']
20.         rader.title = '编程语言对比图'
21.         # 控制各数据点的大小
22.         rader.dots_size = 8
23.         # 设置将图例放在底部
24.         rader.legend_at_bottom = True
25.         # 指定将数据图输出到 SVG 文件中
26.         rader.render_to_file('language_compare.svg')
```

上面程序中第 14 行代码创建了 pygal.Rader 对象，接下来程序使用循环为雷达图添加数据。在雷达图上会显示各编程语言在不同维度的得分点，程序中第 22 行代码设置了得分点的大小。

运行该程序，将会生成如图 9 所示的雷达图。

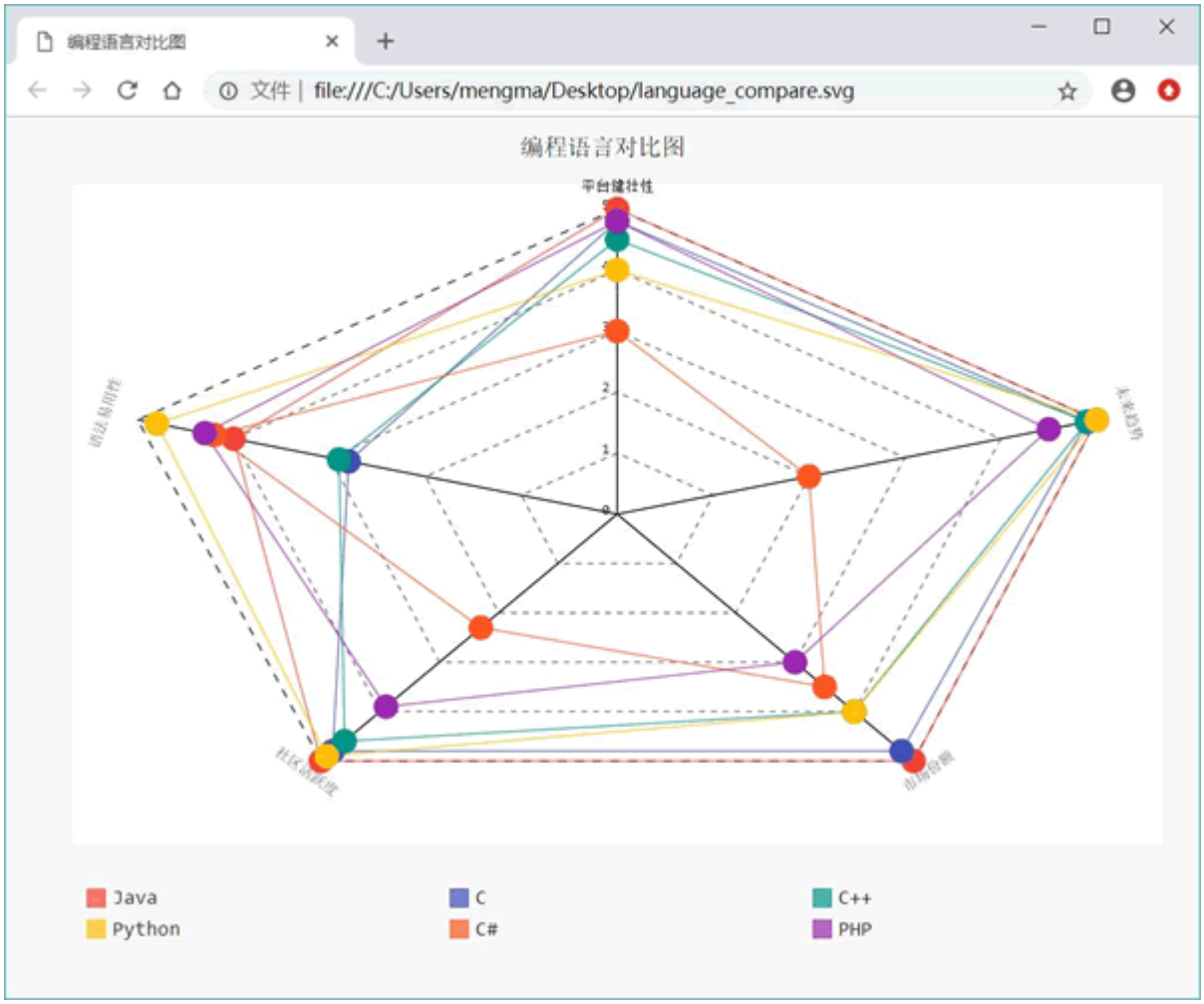


图 9 雷达图

13.Python 读取 csv 文件

前面程序展示的数据都是直接通过程序给出的，但实际应用可能需要展示不同来源（比如文件、网络）、不同格式（比如 csv、JSON）的数据，这些数据可能有部分是损坏的，因此程序需要对这些数据进行处理。

csv 文件格式的本质是一种以文本存储的表格数据（使用 Excel 工具即可读写 csv 文件）。

csv 文件的每行代表一行数据，每行数据中每个单元格内的数据以逗号隔开。

Python 提供了 csv 模块来读写 csv 文件。由于 csv 文件的格式本身比较简单（通常第一行是表头，用于说明每列数据的含义，接下来每行代表一行数据），因此使用 csv 模块读取 csv 文件也非常简单：

- 创建 csv 模块的读取器。
- 循环调用 csv 读取器的 next() 方法逐行读取 csv 文件内容即可。next() 方法返回一个 list 列表代表一行数据，list 列表的每个元素代表一个单元格数据。

本节使用的是 2017 年广州天气数据的 csv 文件（数据来源于 http://lishi.tianqi.com/ 网站。下面程序示范了使用 csv 读取器来读取 csv 文件的两行内容。

```
import csv

filename = 'guangzhou-2017.csv'
# 打开文件
with open(filename) as f:
    # 创建 cvs 文件读取器
    reader = csv.reader(f)
    # 读取第一行，这行是表头数据。
    header_row = next(reader)
    print(header_row)
    # 读取第二行，这行是真正的数据。
    first_row = next(reader)
    print(first_row)
```

上面程序中第 7 行代码创建了 CSV 读取器，第 9 行、第 12 行代码各读取文件的一行，其中第 7 行代码会返回 csv 文件的表头数据；第 9 行代码会返回真正的数据。

运行上面程序，可以看到如下输出结果：

```
['Date', 'Max TemperatureC', 'Min TemperatureC', 'Description', 'WindDir',
'WindForce']
['2017-1-1', '24', '13', '晴', '西南风', '1 级']
```

从上面的输出结果可以看到，该文件的每行包含 6 个数据，分别是日期、最高温度、最低温度、天气情况、风向、风力。

掌握了 csv 读取器的用法之后，下面程序将会使用 Matplotlib 来展示 2017 年 7 月广州的最高气温和最低气温。

```
1.          import csv
2.          from datetime import datetime
3.          from matplotlib import pyplot as plt
4.
5.          filename = 'guangzhou-2017.csv'
6.          # 打开文件
7.          with open(filename) as f:
8.              # 创建 cvs 文件读取器
9.              reader = csv.reader(f)
10.             # 读取第一行，这行是表头数据。
11.             header_row = next(reader)
12.             print(header_row)
13.             # 定义读取起始日期
14.             start_date = datetime(2017, 6, 30)
```

```
15.             # 定义结束日期
16.             end_date = datetime(2017, 8, 1)
17.             # 定义 3 个 list 列表作为展示的数据
18.             dates, highs, lows = [], [], []
19.             for row in reader:
20.                 # 将第一列的值格式化为日期
21.                 d = datetime.strptime(row[0], '%Y-%m-%d')
22.                 # 只展示 2017 年 7 月的数据
23.                 if start_date < d < end_date:
24.                     dates.append(d)
25.                     highs.append(int(row[1]))
26.                     lows.append(int(row[2]))
27.
28.         # 配置图形
29.         fig = plt.figure(dpi=128, figsize=(12, 9))
30.         # 绘制最高气温的折线
31.         plt.plot(dates, highs, c='red', label='最高气温',
32.             alpha=0.5, linewidth = 2.0, linestyle = '-', marker='v')
33.         # 再绘制一条折线
34.         plt.plot(dates, lows, c='blue', label='最低气温',
35.             alpha=0.5, linewidth = 3.0, linestyle = '-.', marker='o')
36.         # 为两个数据的绘图区域填充颜色
37.         plt.fill_between(dates, highs, lows, facecolor='blue', alpha=0.1)
38.         # 设置标题
39.         plt.title("广州 2017 年 7 月最高气温和最低气温")
40.         # 为两条坐标轴设置名称
41.         plt.xlabel("日期")
42.         # 该方法绘制斜着的日期标签
43.         fig.autofmt_xdate()
44.         plt.ylabel("气温（℃）")
45.         # 显示图例
46.         plt.legend()
47.         ax = plt.gca()
48.         # 设置右边坐标轴线的颜色（设置为 none 表示不显示）
49.         ax.spines['right'].set_color('none')
50.         # 设置顶部坐标轴线的颜色（设置为 none 表示不显示）
```

```
51.         ax.spines['top'].set_color('none')
52.         plt.show()
```

上面程序的前半部分代码用于从 csv 文件中读取 2017 年 7 月广州的气温数据，程序分别使用了 dates、highs 和 lows 三个 list 列表来保存日期、最高气温、最低气温。

程序的后半部分代码绘制了两条折线来显示最高气温和最低气温，其中第 31 行代码用于绘制最高气温，第 34 行代码用于绘制最低气温；第 37 行代码控制在两条折线之间填充颜色。程序也对坐标轴、图例进行了简单的设置。

运行上面程序，可以看到如图 1 所示的折线图。

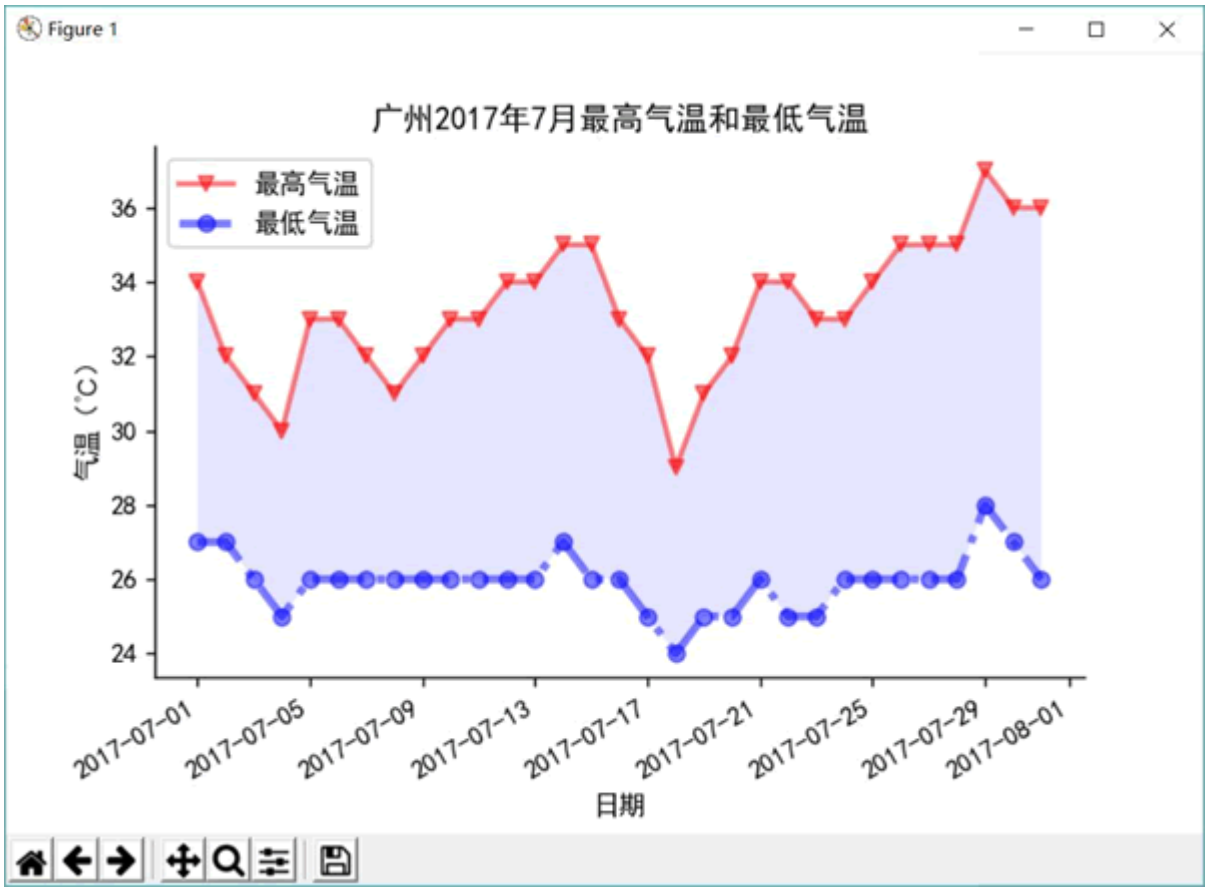


图 1 2017 年 7 月广州的气温折线图

程序也可以使用 Pygal 来统计 2017 年广州的天气汇总情况，比如统计出阴天、晴天、多云天和雨天各占多少天。程序会使用 csv 读取器读取 2017 年广州阴天、晴天、多云天和雨天共有多少天，然后将这些数据添加到 pygal.Pie 对象中即可绘制饼图。该程序的代码如下：

```
import csv
import pygal

filename = 'guangzhou-2017.csv'
# 打开文件
with open(filename) as f:
    # 创建 cvs 文件读取器
    reader = csv.reader(f)
    # 读取第一行，这行是表头数据。
    header_row = next(reader)
    print(header_row)
    # 准备展示的数据
    shades, sunnys, cloudys, rainys = 0, 0, 0, 0
    for row in reader:
        if '阴' in row[3]:
            shades += 1
        elif '晴' in row[3]:
            sunnys += 1
        elif '云' in row[3]:
            cloudys += 1
        elif '雨' in row[3]:
            rainys += 1
        else:
            print(rows[3])
```

```
25.         # 创建 pygal.Pie 对象（饼图）
26.         pie = pygal.Pie()
27.         # 为饼图添加数据
28.         pie.add("阴", shades)
29.         pie.add("晴", sunnys)
30.         pie.add("多云", cloudys)
31.         pie.add("雨", rainys)
32.         pie.title = '2017 年广州天气汇总'
33.         # 设置将图例放在底部
34.         pie.legend_at_bottom = True
35.         # 指定将数据图输出到 SVG 文件中
36.         pie.render_to_file('guangzhou_weather.svg')
```

上面程序的前半部分代码也是用于从 csv 文件中读取 2017 年广州的天气数据，该程序只读取 csv 文件的数据行的第四列数据（天气描述），并使用 shades、sunnys、cloudys、rainys 分别保存阴天、晴天、多云天和雨天的数据。

上面程序中建了一个 pygal.Pie 对象，该对象就表示一个饼图。接下来向 pygal.Pie 对象添加数据。运行上面程序，可以生成如图 2 所示的饼图。

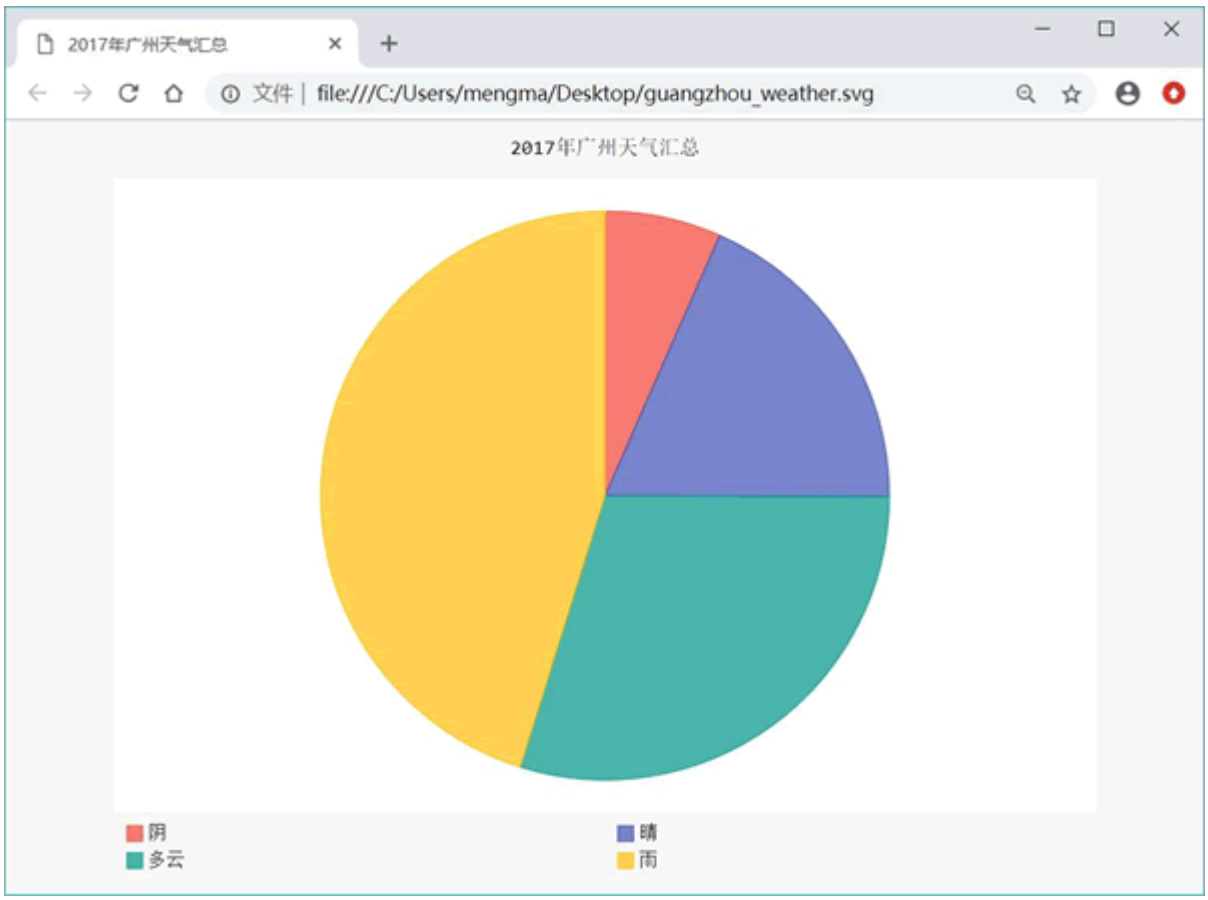


图 2 统计 2017 年广州天气情况的饼图

14.Python 读取 JSON 文件

教程前面章节曾介绍过 JSON 格式的数据，这种格式的数据通常会被转换为 Python 的 list 列表或 dict 字典。本节展示的是世界各国历年 GDP 总和，数据来源于 https://datahub.io 网站。数据格式如下：

```
[{"Country Code":"ARB","Country Name":"Arab World",
"Value":25760683041.0857,"Year":1968},
```

```
{"Country Code：ARB", "country Name":"Arab World",

"Value":28434203615.4829, "Year":1969},

...

]
```

上面的 JSON 格式数据被保存在方括号内，这些数据将会被转换为 Python 的 list 列表，而 list 列表的每个元素将会是一个 dict 对象。

使用 Python 的 json 模块读取 JSON 数据非常简单，只要使用 load() 函数加载 JSON 数据即可。下面程序示范了读取 2016 年中国的 GDP 值：

```
1.          import json
2.
3.          filename = 'gdp_json.json'
4.
5.          with open(filename) as f:
6.              gpd_list = json.load(f)
7.          # 遍历列表的每个元素，每个元素是一个 GDP 数据项
8.          for gpd_dict in gpd_list:
9.              # 只显示中国、2016 年的 GDP
10.             if gpd_dict['Year'] == 2016 and gpd_dict['Country Code'] == 'CHN':
11.                 print(gpd_dict['Country Name'], gpd_dict['Value'])
```

上面程序中，第 6 行代码调用 json 模块的 load() 函数加载 JSON 数据，该函数将会返回一个 list 列表，接下来程序遍历该 list 列表即可访问到指定年份、指定国家的 GDP 值。

运行上面程序，可以看到如下输出结果：

```
China 11199145157649.2
```

在掌握了使用 json 模块读取这份 JSON 数据的方法之后，接下来我们将会从中读取从 2001 年到 2016 年中国、美国、日本、俄罗斯、加拿大这 5 个国家的 GDP 数据，并使用柱状图进行对比。

下面程序将会使用 Matplotlib 生成柱状图来展示这 5 个国家的 GDP 数据：

```
import json
from matplotlib import pyplot as plt
import numpy as np

filename = 'gdp_json.json'
# 读取 JSON 格式的 GDP 数据
with open(filename) as f:
    gpd_list = json.load(f)
# 使用 list 列表依次保存中国、美国、日本、俄罗斯、加拿大的 GDP 值
country_gdps = [{}, {}, {}, {}, {}]
country_codes = ['CHN', 'USA', 'JPN', 'RUS', 'CAN']
# 遍历列表的每个元素，每个元素是一个 GDP 数据项
for gpd_dict in gpd_list:
    for i, country_code in enumerate(country_codes):
        # 只读取指定国家的数据
        if gpd_dict['Country Code'] == country_code:
            year = gpd_dict['Year']
            # 只读取 2001 年到 2016
            if 2017 > year > 2000:
                country_gdps[i][year] = gpd_dict['Value']
# 使用 list 列表依次保存中国、美国、日本、俄罗斯、加拿大的 GDP 值
country_gdp_list = [[], [], [], [], []]
# 构建时间数据
x_data = range(2001, 2017)
```

```
        for i in range(len(country_gdp_list)):
            for year in x_data:
                # 除以 1e8，让数值变成以亿为单位
                country_gdp_list[i].append(country_gdps[i][year] / 1e8)
        bar_width=0.15
        fig = plt.figure(dpi=128, figsize=(15, 8))
        colors = ['indianred', 'steelblue', 'gold', 'lightpink', 'seagreen']
        # 定义国家名称列表
        countries = ['中国', '美国', '日本', '俄罗斯', '加拿大']
        # 采用循环绘制 5 组柱状图
        for i in range(len(colors)):
            # 使用自定义 X 坐标将数据分开
            plt.bar(x=np.arange(len(x_data))+bar_width*i,
height=country_gdp_list[i],
                label=countries[i], color=colors[i], alpha=0.8,
width=bar_width)
            # 仅为中国、美国的条柱上绘制 GDP 数值
            if i < 2:
                for x, y in enumerate(country_gdp_list[i]):
                    plt.text(x, y + 100, '%.0f' % y, ha='center', va='bottom')
        # 为 X 轴设置刻度值
        plt.xticks(np.arange(len(x_data))+bar_width*2, x_data)
        # 设置标题
        plt.title("2001 到 2016 年各国 GDP 对比")
        # 为两条坐标轴设置名称
        plt.xlabel("年份")
        plt.ylabel("GDP(亿美元)")
        # 显示图例
        plt.legend()
        plt.show()
```

本程序的重点其实在于前半部分代码，这部分代码控制程序从 JSON 数据中只读取中国、美国、日本、俄罗斯、加拿大这 5 个国家的数据，且只读取从 2001 年到 2016 年的 GDP 数据，因此程序处理起来稍微有点麻烦（程序先以年份为 key 的 dict（如程序中

country_gdps 列表的元素所示）来保存各国的 GDP 数据。

但由于 Matplotlib 要求被展示数据是 list 列表，因此上面程序中的第 26、28 两行代码使用循环依次读取从 2001 年到 2016 年的 GDP 数据，并将这些数据添加到 country_gdp_list 列表的元素中。这样就把 dict 形式的 GDP 数据转换成 list 形式的 GDP 数据。

上面程序中的第 35、37 两行代码采用循环添加了 5 组柱状图，接下来程序还在中国、美国的条柱上绘制了 GDP 值。

运行上面程序，可以看到如图 1 所示的柱状图。

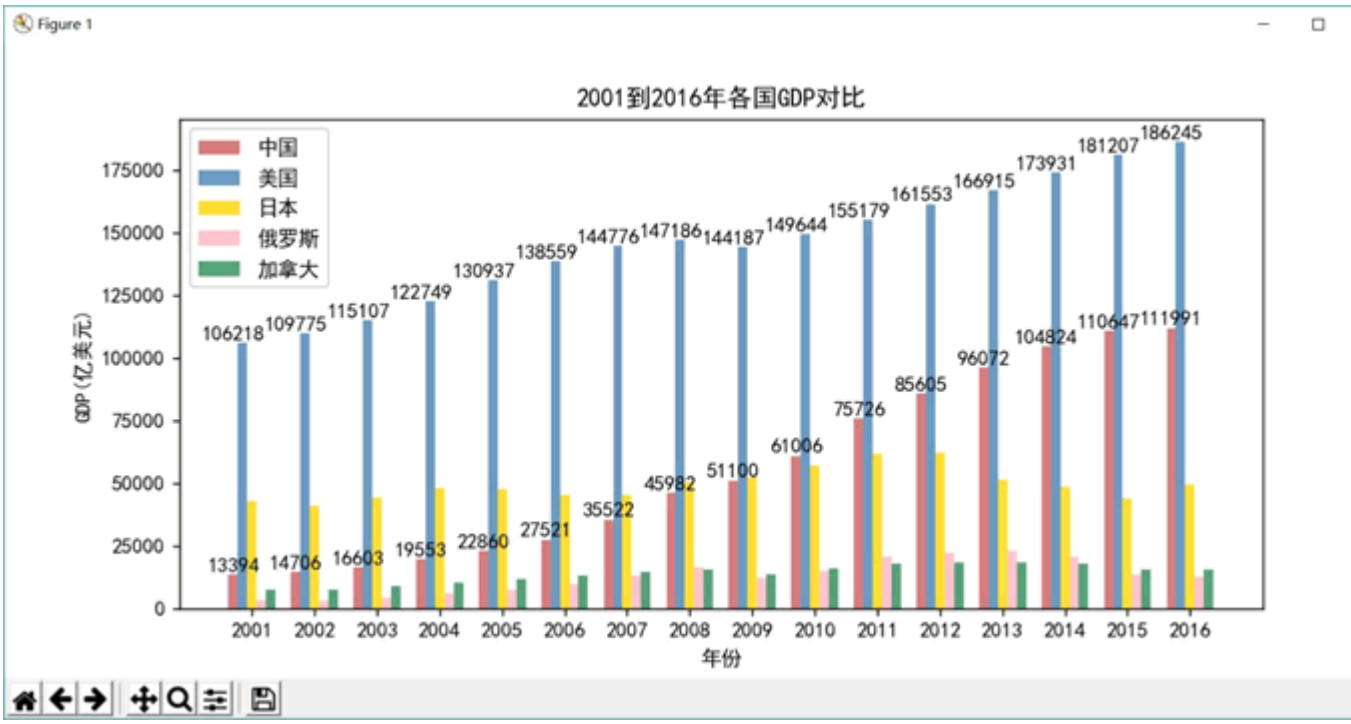


图 1 从 2001 年到 2016 年各国 GDP 对比柱状图

如果通过 https://datahub.io 网站下载了世界各国人口数据，就可以计算出以上各国的人均 GDP。下面程序会使用 Pygal 来展示世界各国的人均 GDP 数据。

```
import json
import pygal

filename = 'gdp_json.json'
# 读取 JSON 格式的 GDP 数据
with open(filename) as f:
    gpd_list = json.load(f)
pop_filename = 'population-figures-by-country.json'
# 读取 JSON 格式的人口数据
with open(pop_filename) as f:
    pop_list = json.load(f)

# 使用 list 列表依次保存美国、日本、俄罗斯、加拿大的人均 GDP 值
country_mean_gdps = [{}, {}, {}, {}]
country_codes = ['USA', 'JPN', 'RUS', 'CAN']
# 遍历列表的每个元素，每个元素是一个 GDP 数据项
for gpd_dict in gpd_list:
    for i, country_code in enumerate(country_codes):
        # 只读取指定国家的数据
        if gpd_dict['Country Code'] == country_code:
            year = gpd_dict['Year']
            # 只读取 2001 年到 2016
            if 2017 > year > 2000:
                for pop_dict in pop_list:
                    # 获取指定国家的人口数据
                    if pop_dict['Country_Code'] == country_code:
                        # 使用该国 GDP 总值除以人口数量，得到人均 GDP
```

```
28.                                     country_mean_gdps[i][year] =
round(gpd_dict['Value']
29.                                         / pop_dict['Population_in_%d' % year])
30.         # 使用 list 列表依次保存美国、日本、俄罗斯、加拿大的人均 GDP 值
31.         country_mean_gdp_list = [[], [], [], []]
32.         # 构建时间数据
33.         x_data = range(2001, 2017)
34.         for i in range(len(country_mean_gdp_list)):
35.             for year in x_data:
36.                 country_mean_gdp_list[i].append(country_mean_gdps[i][year])
37.         # 定义国家名称列表
38.         countries = ['美国', '日本', '俄罗斯', '加拿大']
39.         # 创建 pygal.Bar 对象（柱状图）
40.         bar = pygal.Bar()
41.         # 采用循环添加代表条柱的数据
42.         for i in range(len(countries)):
43.             bar.add(countries[i], country_mean_gdp_list[i])
44.         bar.width=1100
45.         # 设置 X 轴的刻度值
46.         bar.x_labels = x_data
47.         bar.title = '2001 到 2016 年各国人均 GDP 对比'
48.         # 设置 X、Y 轴的标题
49.         bar.x_title = '年份'
50.         bar.y_title = '人均 GDP(美元)'
51.         # 设置 X 轴的刻度值旋转 45 度
52.         bar.x_label_rotation = 45
53.         # 设置将图例放在底部
54.         bar.legend_at_bottom = True
55.         # 指定将数据图输出到 SVG 文件中
56.         bar.render_to_file('mean_gdp.svg')
```

上面程序中，第 11 行代码加载了一份新的关于人口数据的 JSON 文件，这样程序即可通过该文件获取世界各国历史的人口数据。第 28 行代码使用 GDP 总值除以该国的人口数量，这样就可以得到该国的人均 GDP。

此程序的后半部分代码创建了 pygal.Bar 对象 ,并使用循环为该对象添加了各国人均 GDP 数据，这样该柱状图就可以展示各国的人均 GDP 值。

运行上面程序，可以看到如图 2 所示的柱状图。

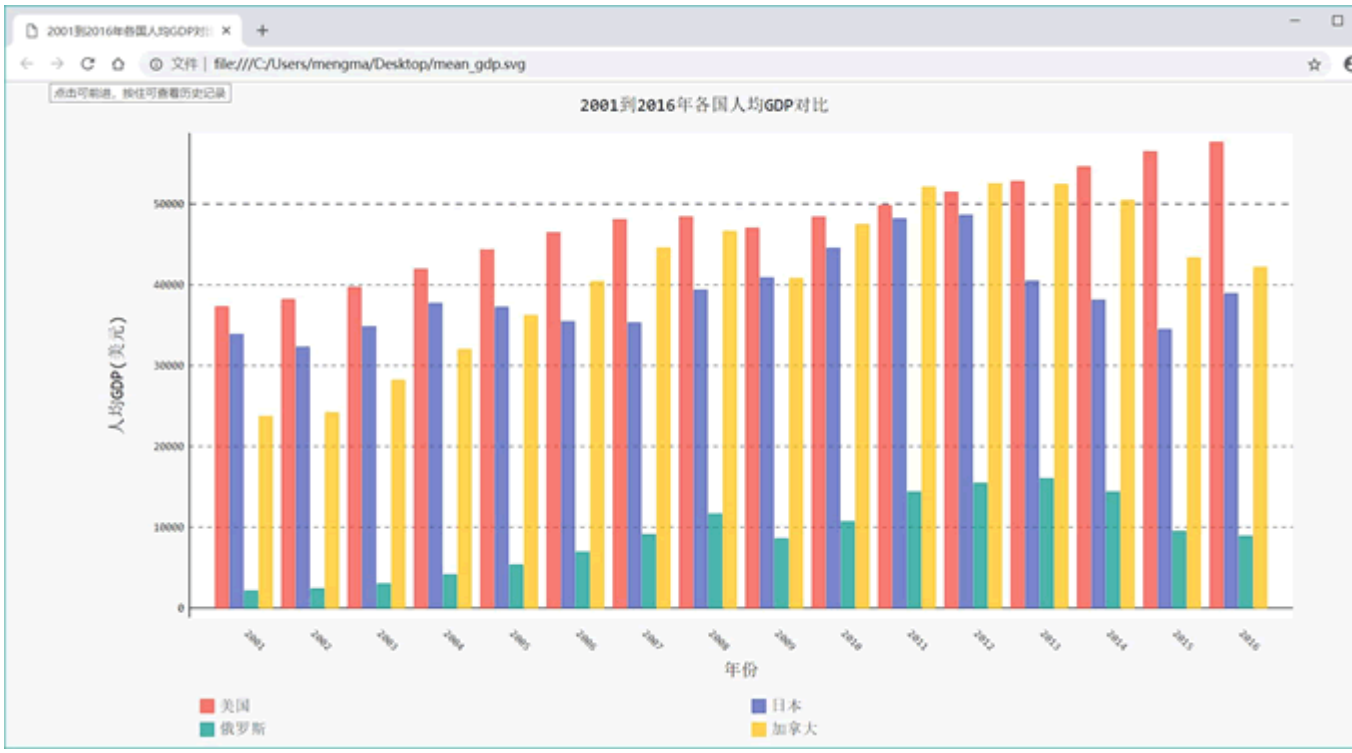


图 2 从 2001 年到 2016 年各国人均 GDP 对比柱状图（点此查看高清大图）

15.Python 读取网络数据

Python 读取网络数据（request 库和 re 模块）

很多时候，程序并不能直接展示本地文件中的数据，此时需要程序读取网络数据，并展示它们。

比如前面介绍的 http://lishi.tianqi.com 站点的数据，它并未提供下载数据的链接（前面程序所展示的 csv 文件本身就是使用程序抓取下来的）。在这种情况下，程序完全可以直接解析网络数据，然后将数据展示出来。

前面已经介绍了 Python 的网络支持库 urllib，通过该库下的 request 模块可以非常方便地向远程发送 HTTP 请求，获取服务器响应。因此，本程序的思路是使用 urllib.request 向 lishi.tianqi.com 发送请求，获取该网站的响应，然后使用 Python 的 re 模块来解析服务器响应，从中提取天气数据。

本程序将会通过网络读取 http://lishi.tianqi.com 站点的数据，并展示 2017 年广州的最高气温和最低气温。

```
1.          import re
2.          from datetime import datetime
3.          from datetime import timedelta
4.          from matplotlib import pyplot as plt
5.          from urllib.request import *
6.
7.          # 定义一个函数读取 lishi.tianqi.com 的数据
8.          def get_html(city, year, month):  #①
9.              url = 'http://lishi.tianqi.com/' + city + '/' + str(year) + str(month)
    + '.html'
10.             # 创建请求
11.             request = Request(url)
12.             # 添加请求头
13.             request.add_header('User-Agent', 'Mozilla/5.0 (Windows NT 10.0;
    WOW64)' +
```

```
14.                 'AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.99
    Safari/537.36')
15.             response = urlopen(request)
16.             # 获取服务器响应
17.             return response.read().decode('gbk')
18.
19.         # 定义 3 个 list 列表作为展示的数据
20.         dates, highs, lows = [], [], []
21.         city = 'guangzhou'
22.         year = '2017'
23.         months = ['01', '02', '03', '04', '05', '06', '07',
24.             '08', '09', '10', '11', '12']
25.         prev_day = datetime(2016, 12, 31)
26.         # 循环读取每个月的天气数据
27.         for month in months:
28.             html = get_html(city, year, month)
29.             # 将 html 响应拼起来
30.             text = "".join(html.split())
31.             # 定义包含天气信息的 div 的正则表达式
32.             patten =
    re.compile('<divclass="tqtongji2">(.*?)</div><divstyle="clear:both">')
33.             table = re.findall(patten, text)
34.             patten1 = re.compile('<ul>(.*?)</ul>')
35.             uls = re.findall(patten1, table[0])
36.             for ul in uls:
37.                 # 定义解析天气信息的正则表达式
38.                 patten2 = re.compile('<li>(.*?)</li>')
39.                 lis = re.findall(patten2, ul)
40.                 # 解析得到日期数据
41.                 d_str = re.findall('>(.*?)</a>', lis[0])[0]
42.                 try:
43.                     # 将日期字符串格式化为日期
44.                     cur_day = datetime.strptime(d_str, '%Y-%m-%d')
45.                     # 解析得到最高气温和最低气温
46.                     high = int(lis[1])
47.                     low = int(lis[2])
```

```
                except ValueError:
                    print(cur_day, '数据出现错误')
                else:
                    # 计算前、后两天数据的时间差
                    diff = cur_day - prev_day
                    # 如果前、后两天数据的时间差不是相差一天，说明数据有问题
                    if diff != timedelta(days=1):
                        print('%s 之前少了%d 天的数据' % (cur_day, diff.days - 1))
                    dates.append(cur_day)
                    highs.append(high)
                    lows.append(low)
                    prev_day = cur_day
        # 配置图形
        fig = plt.figure(dpi=128, figsize=(12, 9))
        # 绘制最高气温的折线
        plt.plot(dates, highs, c='red', label='最高气温',
            alpha=0.5, linewidth = 2.0)
        # 再绘制一条折线
        plt.plot(dates, lows, c='blue', label='最低气温',
            alpha=0.5, linewidth = 2.0)
        # 为两个数据的绘图区域填充颜色
        plt.fill_between(dates, highs, lows, facecolor='blue', alpha=0.1)
        # 设置标题
        plt.title("广州%s 年最高气温和最低气温" % year)
        # 为两条坐标轴设置名称
        plt.xlabel("日期")
        # 该方法绘制斜着的日期标签
        fig.autofmt_xdate()
        plt.ylabel("气温（℃）")
        # 显示图例
        plt.legend()
        ax = plt.gca()
        # 设置右边坐标轴线的颜色（设置为 none 表示不显示）
        ax.spines['right'].set_color('none')
        # 设置顶部坐标轴线的颜色（设置为 none 表示不显示）
        ax.spines['top'].set_color('none')
```

```
84.          plt.show()
```

这个程序后半部分的绘图代码与前面程序并没有太大的区别 ,该程序的最大改变在于前半部分代码，该程序不再使用 csv 模块来读取本地 csv 文件的内容，而是使用 urllib.request 来读取 lishi.tianqi.com 站点的天气数据 ,程序中 ① 号代码定义了一个 get_html() 函数来读取指定站点的 HTML 内容。

接下来程序使用循环依次读取 01~12 每个月的响应页面，程序读取到每个响应页面的 HTML 内容，这份 HTML 页面内容中包含天气信息的源代码如图 1 所示。

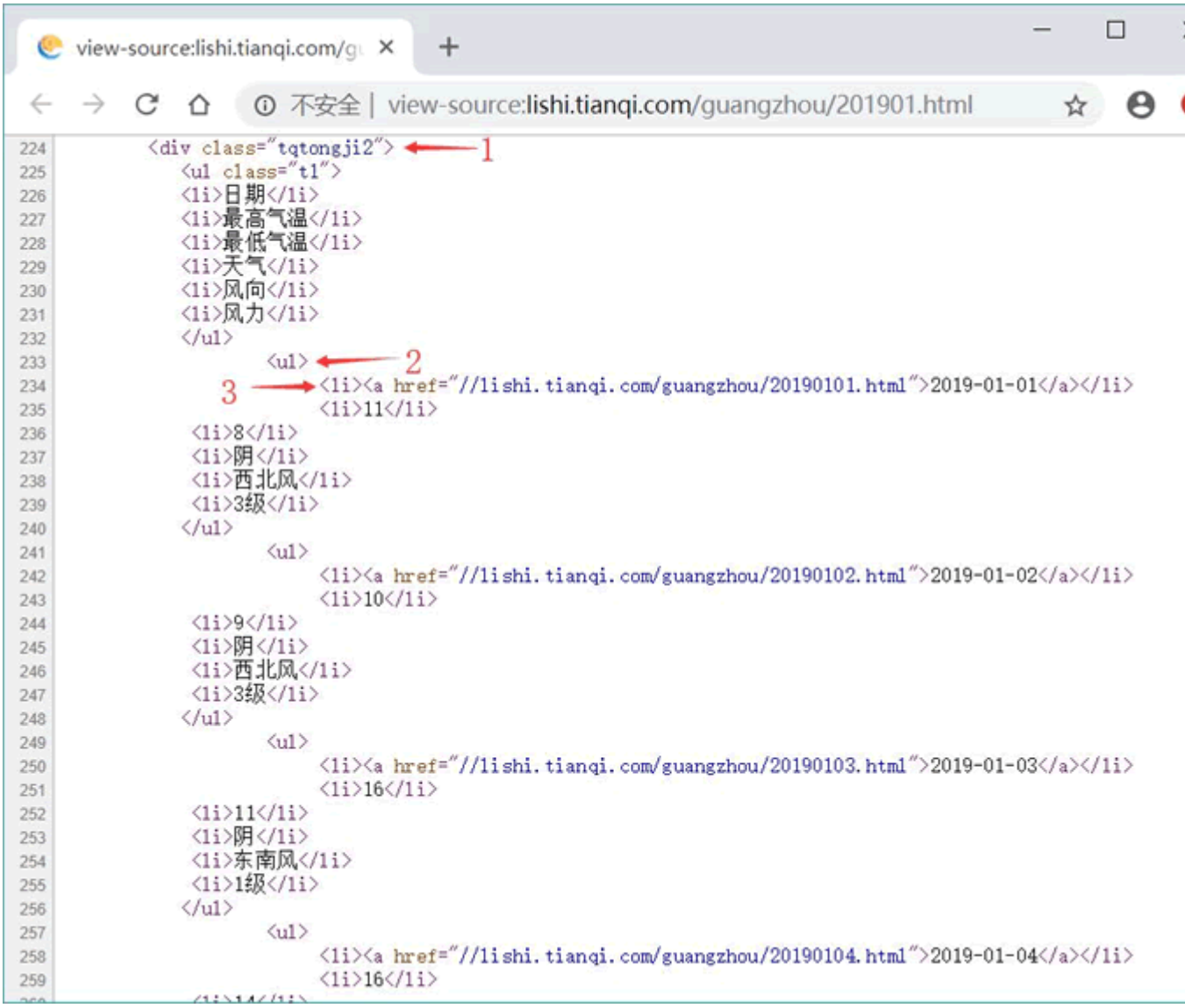


图 1 包含天气信息的 HTML 源代码

程序中第 32 行代码使用正则表达式来获取包含全部天气信息的 <div.../> 元素，即图 1 中数字 1 所标识的 <div.../> 元素。

程序中第 34 行代码使用正则表达式来匹配天气 <div.../> 中没有属性的 <ul.../> 元素，

即图 1 中数字 2 所标识的 <ul.../> 元素。这样的 <ul.../> 元素有很多个，每个 <ul.../> 元素代表一天的天气信息，因此，上面程序使用了循环来遍历每个 <ul.../> 元素。

程序中第 38 行代码使用正则表达式来匹配每日天气 <ul...> 中的 <li.../> 元素，即图 1 中数字 3 所标识的 <li.../> 元素。在每个 <ul.../> 元素内可匹配到 6 个 <li.../> 元素，但程序只获取日期、最高气温和最低气温，因此，程序只使用前三个 <li.../> 元素的数据。

通过网络、正则表达式获取了数据之后，程序使用 Matplotlib 来展示它们。运行上面程序，即可看到表示 2017 年广州气温变化的数据图。

第 21 章 Python Scrapy 网络爬虫

1.网络爬虫是什么

在介绍数据可视化分析时，最后一个示例示范了展示从网络上获取的天气信息。从广义上说，那个例子也属于网络爬虫，程序可以自动获取多个页面中的所有天气信息。如果使用某种技术（如正则表达式、XPath 等）来提取页面中所有的链接（<a.../> 元素），然后顺着这些链接递归打开对应的页面，最后提取页面中的信息，这就是网络爬虫。

既然前面已经介绍了网络爬虫的知识，那么本章还有存在的意义吗？在回答这个问题之前，我们来分析网络爬虫具体要做哪些核心工作：

- 通过网络向指定的 URL 发送请求，获取服务器响应内容。
- 使用某种技术（如正则表达式、XPath 等）提取页面中我们感兴趣的信息。

- 高效地识别响应页面中的链接信息，顺着这些链接递归执行此处介绍的第 1、2、3 步；
- 使用多线程有效地管理网络通信交互。

如果直接使用 Python 内置的 urllib 和 re 模块是否能写出自己的网络爬虫呢？答案是肯定的，只是比较复杂。就像我们要从广州去韶关，走路可以去吗？答案是肯定的，只是比较麻烦。

下面继续分析网络爬虫的核心工作：

- 向 URL 发送请求，获取服务器响应内容。这个核心工作其实是所有网络爬虫都需要做的通用工作。一般来说，通用工作应该由爬虫框架来实现，这样可以提供更稳定的性能，开发效率更高。
- 提取页面中我们感兴趣的信息。这个核心工作不是通用的！每个项目感兴趣的信息都可能有所不同，但使用正则表达式提取信息是非常低效的，原因是正则表达式的设计初衷主要是处理文本信息，而 HTML 文档不仅是文本文档，而且是结构化文档，因此使用正则表达式来处理 HTML 文档并不合适。使用 XPath 提取信息的效率要高得多。
- 识别响应页面中的链接信息。使用正则表达式可以实现这个核心工作，但是效率太低，使用 XPath 会更高效。
- 多线程管理：这个核心工作是通用的，应该由框架来完成。

现在来回答上面提出的问题，本章有存在的意义吗？当然有，本章并不介绍使用 urllib、re

模块这种简陋的工具来实现正则表达式，而是通过专业的爬虫框架 Scrapy 来实现爬虫。

Scrapy 是一个专业的、高效的爬虫框架，它使用专业的 Twisted 包（基于事件驱动的网络引擎包）高效地处理网络通信，使用 lxml（专业的 XML 处理包）、cssselect 高效地提取 HTML 页面的有效信息，同时它也提供了有效的线程管理。

一言以蔽之，上面列出的网络爬虫的核心工作，Scrapy 全部提供了实现，开发者只要使用 XPath 或 css 选择器定义自己感兴趣的信息即可。

2.Python Scrapy 安装

安装 Scrapy 与安装其他 Python 包没有区别，同样使用如下命令来安装：

```
pip install scrapy
```

如果在命令行窗口中运行该命令，将会看到程序并不立即下载、安装 Scrapy，而是不断地下载大量第三方包。

如果在命令行窗口中提示找不到 pip 命令，则也可以通过 python 命令运行 pip 模块来安装 Scrapy，例如 python -m pip install scrapy。

这是因为 Scrapy 需要依赖、大量第三方包。典型的，Scrapy 需要依赖如下第三方包：

- pyOpenSSL：Python 用于支持 SSL（Security Socket Layer）的包。
- cryptography：Python 用于加密的库。
- CFFI：Python 用于调用 C 的接口库。

- zope.interface：为 Python 缺少接口而提供扩展的库。
- lxml：一个处理 XML、HTML 文档的库，比 Python 内置的 xml 模块更好用。
- cssselect：Python 用于处理 CSS 选择器的扩展包。
- Twisted：为 Python 提供的基于事件驱动的网络引擎包。
- ……

如果在 Python 环境下没有这些第三方包，那么 Python 会根据依赖自动下载并安装它们。这个过程原本没啥好讲的 ,pip 通常会自动完成整个过程 ,我们只需要等待即可。但 pip 在自动下载、安装 Twisted 时会提示以下错误：

```
error:Microsoft Visual C++ 14.0 is required. Get it with "Microsoft Visual C++ Build Tools":http://landinghub.visualstudio.com/visual-cpp-build-tools
```

按照上面的错误提示，我们需要先下载和安装 Microsoft Visual C++ Build Tools 工具，然后才能安装 Twisted。为了安装一个小小的 Twisted 包，难道就需要安装一个庞大的 Microsoft Visual C++ Build Tools?

答案是否定的，提示上面的错误只是因为 pip 自动下载的 Twisted 安装包有一些缺陷，因此可以先自行下载 Twisted 安装包。登录 www.lfd.uci.edu/~gohlke/pythonlibs/ 站点，在该页面中间查找“Twisted”项目，可以看到如图 1 所示的下载链接。

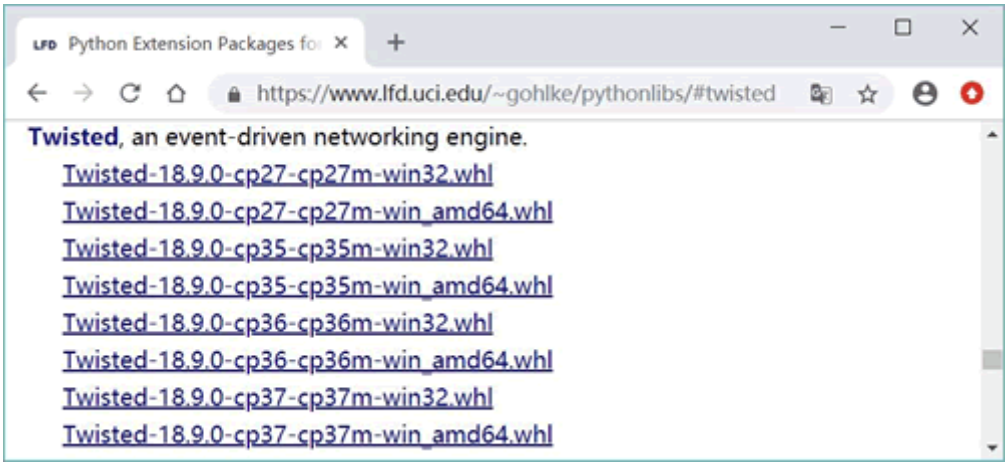


图 1 Twisted 包的下载链接

从图 1 所示的链接可以看到，当前 Twisted 的最新版是 18.9.0，Twisted 为 2.7、3.5、3.6、3.7 等不同版本的 Python 提供了对应的安装包。由于本教程内容主要以 Python 3.6 为主，因此应该下载 Twisted 的 Python 3.6 版本，其中文件名带 win32 的是 32 位版本，而带 win_amd64 的则是 64 位版本，此处还需要根据操作系统的位数选择对应的版本。

在下载了合适的 Twisted 安装包后，会得到一个 Twisted-18.9.0-cp36-cp36m-win_amd64.whl 文件（针对 64 位系统的），该文件就是 Twisted 安装包。

运行如下命令来安装 Twisted 包。

```
pip install Twisted-18.9.0-cp36-cp36m-win_amd64.whl
```

在安装过程中会自动检查，如有必要，会自动下载并安装 Twisted 所依赖的第三方包，如 zope.interface、Automat、incremental 等。

在安装完成后，会提示如下安装成功的信息：

```
Successfully installed Twisted-18.9.0
```

在成功安装 Twisted 包之后，再次执行 pip install scrapy 命令，即可成功安装 Scrapy。在安装成功后，会显示如下提示信息：

```
Successfully installed Scrapy-1.6.1
```

在成功安装 Scrapy 之后，可以通过 pydoc 来查看 Scrapy 的文档。在命令行窗口中输入如下命令：

```
python -m pydoc -p 8899
```

运行上面命令之后，打开浏览器查看 http://localhost:8899/ 页面，可以在 Python 安装目录的 lib\site-packages 下看到 Scrapy 的文档，如图 2 所示。

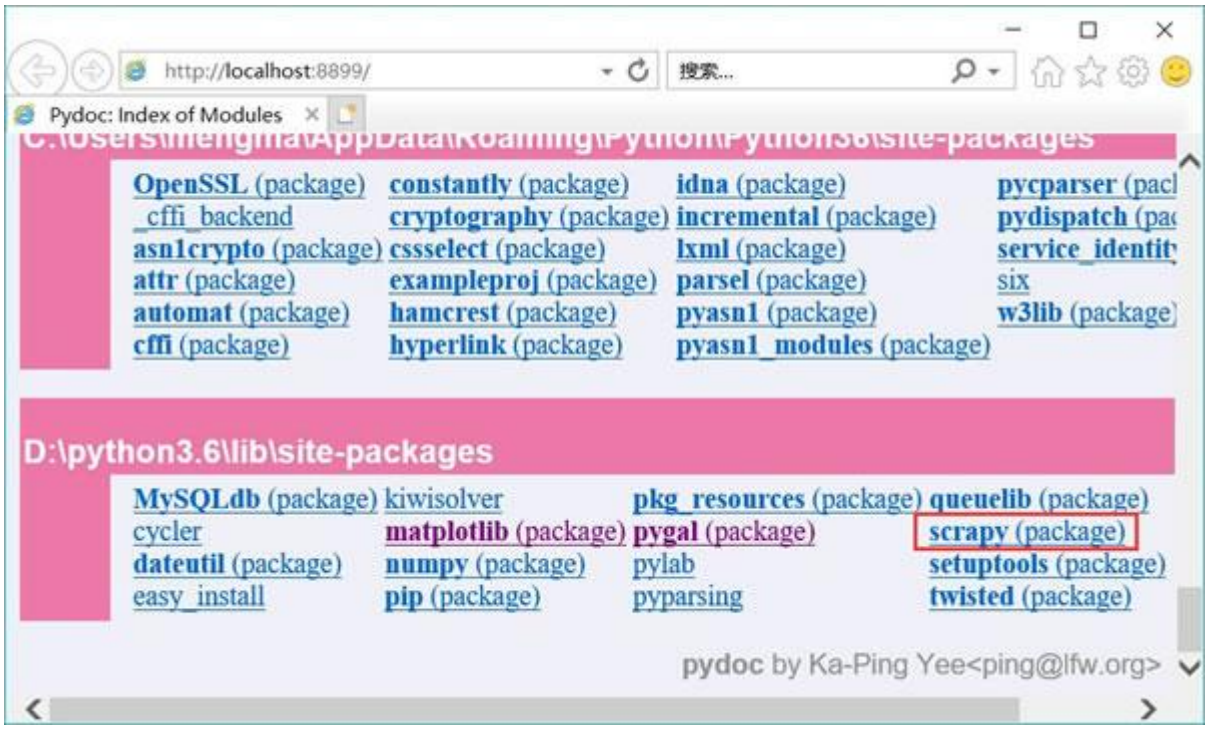


图 2 Scrapy 的文档

3.Python Scrapy 项目创建

在使用Scrapy开发爬虫时，通常需要创建一个Scrapy项目。通过如下命令即可创建 Scrapy 项目：

```
scrapy startproject ZhipinSpider
```

在上面命令中，scrapy 是 Scrapy 框架提供的命令；startproject 是 scrapy 的子命令，专门用于创建项目；ZhipinSpider 就是要创建的项目名。

scrapy 除提供 startproject 子命令之外，它还提供了 fetch（从指定 URL 获取响应）、genspider（生成蜘蛛）、shell（启动交互式控制台）、version（查看 Scrapy 版本）等常用的子命令。可以直接输入 scrapy 来查看该命令所支持的全部子命令。

运行上面命令，将会看到如下输出结果：

```
New Scrapy project 'ZhipinSpider', using template directory
'd:\python3.6\lib\site-packages\scrapy\templates\project', created in:
    C:\Users\mengma\ZhipinSpider

You can start your first spider with:
    cd ZhipinSpider
    scrapy genspider example example.com
```

上面信息显示 Scrapy 在当前目录下创建了一个 ZhipinSpider 项目，此时在当前目录下就可以看到一个 ZhipinSpider 目录，该目录就代表 ZhipinSpider 项目。

查看 ZhipinSpider 项目，可以看到如下文件结构：

```
ZhipinSpider
    |   scrapy.cfg
    |
    └───ZhipinSpider
        |   item.py
        |   middlewares.py
```

```
        |   pipelines.py
        |   setting.py
        |
        ├── spiders
        |       |   __init__.py
        |       |
        |       └── __pycache__
        └── __pycache__
```

下面大致介绍这些目录和文件的作用：

- scrapy.cfg：项目的总配置文件，通常无须修改。
- ZhipinSpider：项目的 Python 模块，程序将从此处导入 Python 代码。
- ZhipinSpider/items.py：用于定义项目用到的 Item 类。Item 类就是一个 DTO（数据传输对象），通常就是定义 N 个属性，该类需要由开发者来定义。
- ZhipinSpider/pipelines.py：项目的管道文件，它负责处理爬取到的信息。该文件需要由开发者编写。
- ZhipinSpider/settings.py：项目的配置文件，在该文件中进行项目相关配置。
- ZhipinSpider/spiders：在该目录下存放项目所需的蜘蛛，蜘蛛负责抓取项目感兴趣的信息。

为了更好地理解 Scrapy 项目中各组件的作用，下面给出 Scrapy 概览图，如图 1 所示。

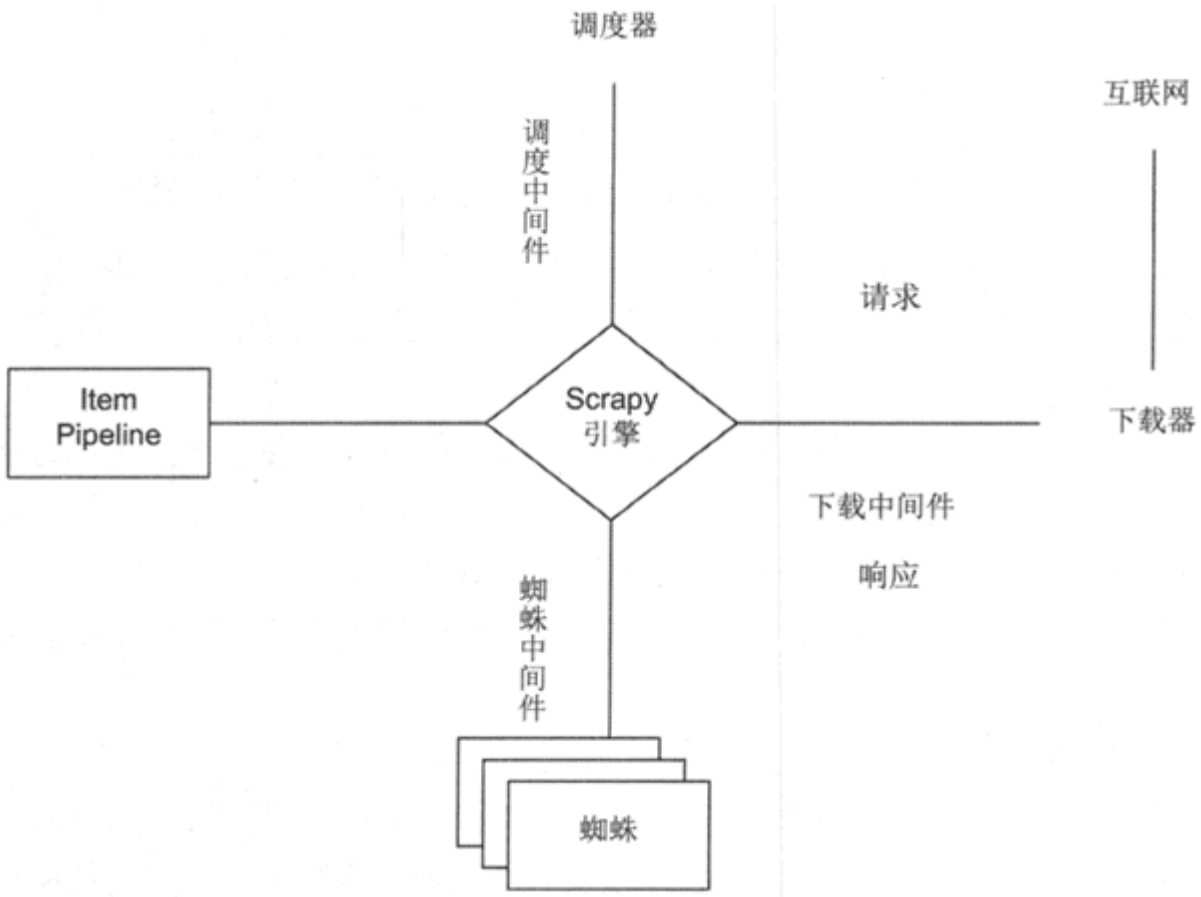


图 1 Scrapy 概览图

在图 1 中可以看到，Scrapy 包含如下核心组件：

- 调度器：该组件由 Scrapy 框架实现，它负责调用下载中间件从网络上下载资源。
- 下载器：该组件由 Scrapy 框架实现，它负责从网络上下载数据，下载得到的数据会由 Scrapy 引擎自动交给蜘蛛。
- 蜘蛛：该组件由开发者实现，蜘蛛负责从下载数据中提取有效信息。蜘蛛提取到的信息会由 Scrapy 引擎以 Item 对象的形式转交给 Pipeline。

- Pipeline：该组件由开发者实现，该组件接收到 Item 对象（包含蜘蛛提取的信息）后，可以将这些信息写入文件或数据库中。

经过上面分析可知，使用 Scrapy 开发网络爬虫主要就是开发两个组件，蜘蛛和 Pipeline。

4.Python Scrapy shell

本节示例将会爬取 BOSS 直聘网上广州地区的热门职位进行分析。首先使用浏览器访问 https://www.zhipin.com/c101280100/h_101280100/ 页面，即可看到广州地区的热门职位。

这里我们要使用爬虫来爬取该页面中的信息，因此需要查看该页面的源代码。可以看到，该页面中包含工作信息的源代码如图 1 所示。

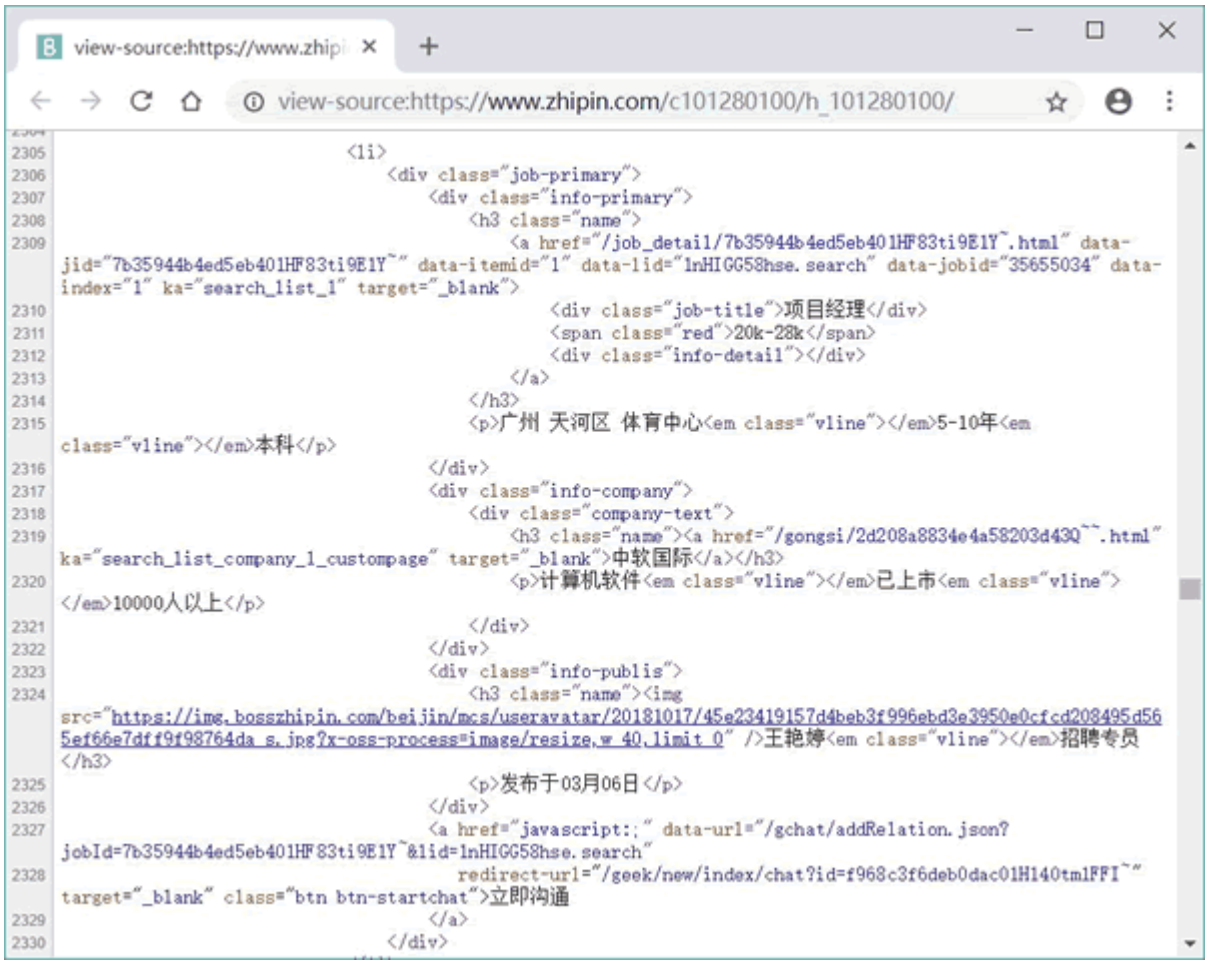


图 1 页面中包含工作信息的源代码（<div.../>元素）（点此查看高清大图）

下面将会使用 Scrapy 提供的 shell 调试工具来抓取该页面中的信息。使用如下命令来开启 shell 调试：

```
scrapy shell https://www.zhipin.com/c101280100/h_101280100/
```

运行上面命令，将会看到 Scrapy 并未抓取到页面数据，页面返回了 403 错误(如图 2 所示），这表明目标网站开启了“防爬虫”，不允许使用 Scrapy“爬取”数据。为了解决这个问题，我们需要让 Scrapy 伪装成浏览器。

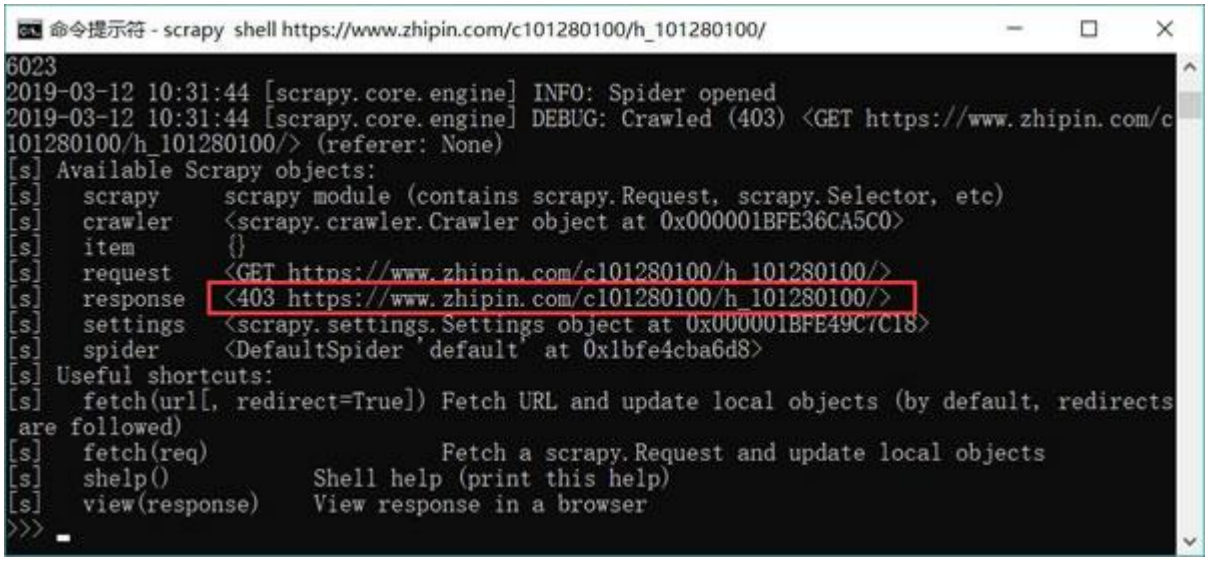


图 2 使用 shell 调试工具抓取页面信息时出现 403 错误

为了让 Scrapy 伪装成浏览器，需要在发送请求时设置 User-Agent 头，将 User-Agent 的值设置为真实浏览器发送请求的 User-Agent。

查看浏览器的 User-Agent，可按如下步骤进行操作（以 Firefox 为例）：

- 启动 Firefox 浏览器，然后按下“Ctrl+Shift+I”快捷键打开浏览器的调试控制台，选择“网络”Tab 页。
- 通过该浏览器可以正常浏览任意页面。
- 在浏览器下方的调试控制台中，将会显示浏览器向哪些资源发送了请求。

- 在调试控制台中选择浏览器所请求的任意一个资源，即可在右边看到浏览器发送请求的各种请求头，如图 3 所示。

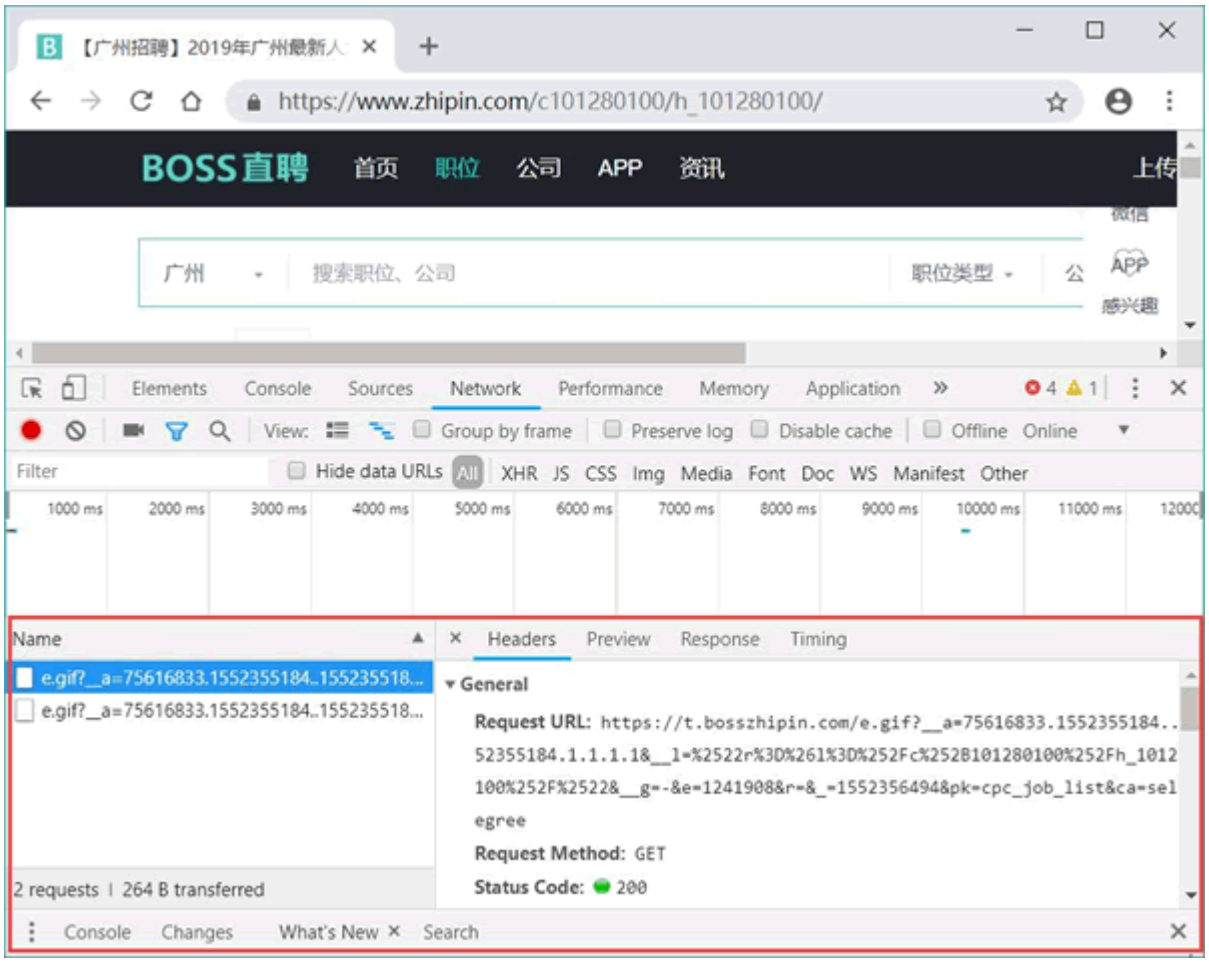


图 3 查看 Firefox 发送请求的请求头

因此，可以使用如下命令让 Scrapy 伪装成 Firefox 来开启 shell 调试。

```
scrapy shell -s USER_AGENT='Mozilla/5.0'

https://www.zhipin.com/c101280100/h_101280100
```

执行上面命令，将可以看到使用 shell 调试工具成功抓取页面的提示信息，如图 4 所示。

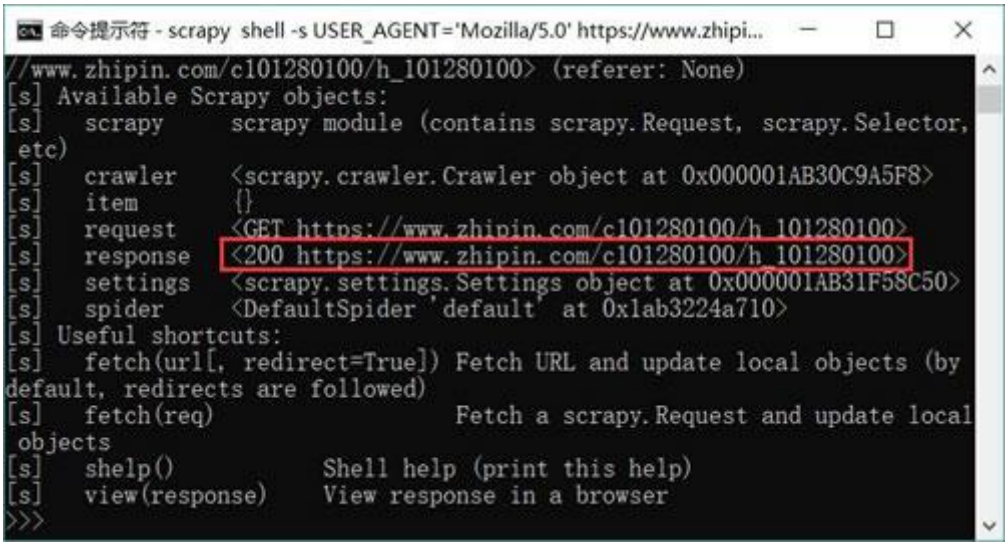


图 4 使用 shell 调试工具成功抓取页面的提示信息

接下来就可以使用 XPath 或 css 选择器来提取我们感兴趣的信息了。

为了让读者能看懂后面的代码，这里简单补充一点 XPath 的必要知识。表 5 中列出了 XPath 最实用的简化写法。

表 5 XPath 最实用的简化写法

表达式	作用
nodename	匹配此节点的所有内容
/	匹配根节点
//	匹配任意位置的节点

.	匹配当前节点
..	匹配父节点
@	匹配属性

典型的，比如可以使用 //div 来匹配页面中任意位置处的 <div.../> 元素，也可以使用 //div/span 来匹配页面中任意位置处的 <div....> 元素内的 <span.../> 子元素。

XPath 还支持“谓词”，就是在节点后增加一个方括号，在方括号内放一个限制表达式对该节点进行限制。

典型的，我们可以使用 //div[@class] 来匹配页面中任意位置处、有 class 属性的 <div.../> 元素 ,也可以使用 //div/span[1] 来匹配页面中任意位置处的 <div.../> 元素内的第一个 <span.../> 子元素；使用 //div/span[last()] 来匹配页面中任意位置处的 <div.../> 元素内的最后一个 <span.../> 子元素；使用 //div/span[last()-1] 来匹配页面中任意位置处的 <div.../> 元素内的倒数第二个 <span.../> 子元素……

例如，想获取上面页面中的第一条工作信息的工作名称，从图 1 中可以看到，所有工作信息都位于 元素内，因此该 XPath 的开始应该写成：

```
//div[@class="job-primary"]
```

接下来可以看到工作信息还处于 元素内，因此该 XPath 应该写成（此处不加谓词也可以）：

```
//div[@class="job-primary"]/div
```

接下来可以看到工作信息还处于 <h3 class="name"> 元素内，因此该 XPath 应该写成（此处不加谓词也可以）：

```
//div [@class = "job-primary"]/div/h3
```

依此类推，可以看到工作名称对应的 XPath 写成：

```
//div[@class = "job-primary"]/div/h3/a/div/text()
```

在掌握了 XPath 的写法之后，即可在 Scrapy 的 shell 控制台调用 response 的 xpath() 方法来获取 XPath 匹配的节点。执行如下命令：

```
response.xpath ('//div[@class="job-primary"]/div/h3/a/div/text()').extract()
```

上面的 extract() 方法用于提取节点的内容。运行上面命令，可以看到如图 6 所示的输出信息。

图 6 使用 css 选择器抓取感兴趣的信息

相比之下，XPath 比 css 选择器的匹配能力更强，因此本章的项目会使用 XPath 来匹配、抓取感兴趣的信息。

5.Python Scrapy 爬虫项目开发过程

通过前面的 Scrapy shell 调试，已经演示了使用 XPath 从 HTML 文档中提取信息的方法，只要将这些调试的测试代码放在 Spider 中，即可实现真正的 Scrapy 爬虫。

基于 Scrapy 项目开发爬虫大致需要如下几个步骤：

1. 定义 Item 类。该类仅仅用于定义项目需要爬取的 N 个属性。比如该项目需要爬取工作名称、工资、招聘公司等信息，则可以在 items.py 中增加如下类定义：

```
import scrapy

class ZhipinspiderItem(scrapy.Item):
    # 工作名称
```

```
5.              title = scrapy.Field()
6.              # 工资
7.              salary = scrapy.Field()
8.              # 招聘公司
9.              company = scrapy.Field()
10.             # 工作详细链接
11.             url = scrapy.Field()
12.             # 工作地点
13.             work_addr = scrapy.Field()
14.             # 行业
15.             industry = scrapy.Field()
16.             # 公司规模
17.             company_size = scrapy.Field()
18.             # 招聘人
19.             recruiter = scrapy.Field()
20.             # 发布时间
21.             publish_date = scrapy.Field()
```

上面程序中，第 3 行代码表明所有的 Item 类都需要继承 scrapy.Item 类，接下来就为所有需要爬取的信息定义对应的属性，每个属性都是一个 scrapy.Field 对象。

该 Item 类只是一个作为数据传输对象（DTO）的类，因此定义该类非常简单。

2. 编写 Spider 类。应该将该 Spider 类文件放在 spiders 目录下。这一步是爬虫开发的关键，需要使用 XPath 或 CSS 选择器来提取 HTML 页面中感兴趣的信息。

Scrapy 为创建 Spider 提供了 scrapy genspider 命令，该命令的语法格式如下：

```
scrapy genspider [options] <name> <domain>
```

在命令行窗口中进入 ZhipinSpider 目录下，然后执行如下命令即可创建一个 Spider：

```
scrapy genspider job_position "zhipin.com"
```

运行上面命令，即可在 ZhipinSpider 项目的 ZhipinSpider/spider 目录下找到一个 job_position.py 文件，打开该文件可以看到如下内容：

```
import scrapy

class JobPositionSpider(scrapy.Spider):
    # 定义该 Spider 的名字
    name = 'job_position'
    # 定义该 Spider 允许爬取的域名
    allowed_domains = ['zhipin.com']
    # 定义该 Spider 爬取的首页列表
    start_urls = ['https://www.zhipin.com/c101280100/h_101280100/']

    # 该方法负责提取 response 所包含的信息
    def parse(self, response):
        pass
```

上面程序就是 Spider 类的模板，该类的 name 属性用于指定该 Spider 的名字；allow_domains 用于限制该 Spider 所爬取的域名；start_urls 指定该 Spider 会自动爬取的页面 URL。

Spider 需要继承 scrapy.Spider，并重写它的 parse(self, response) 方法（如上面程序所示）。从该类来看，我们看不到发送请求、获取响应的代码，这也正是 Scrapy 的魅力所在，即只要把所有需要爬取的页面 URL 定义在 start_urls 列表中，Scrapy 的下载中间件就会负责从网络上下载数据，并将所有数据传给 parse(self, response) 方法的

response 参数。

> 注意，如果在 Windows 上使用 genspider 命令来生成爬虫类，则容易引发 SyntaxError:(unicode error)'utf-8' codec can’ t decode byte 0xb9 in position 0: invalid start byte 错误，这是由于 Windows 采用了 GBK 字符集。因此，需要手动将该 Spider 类保存为 UTF-8 字符集。

一言以蔽之，开发者只要在 start_urls 列表中列出所有需要 Spider 爬取的页面 URL，这些页面的数据就会“自动”传给 parse(self, response) 方法的 response 参数。

因此，开发者主要就是做两件事情：

- 将要爬取的各页面 URL 定义在 start_urls 列表中。
- 在 parse(self, response) 方法中通过 XPath 或 CSS 选择器提取项目感兴趣的信息。

下面将 job_position.py 文件改为如下形式：

```
16.         import scrapy
17.         from ZhipinSpider.items import ZhipinspiderItem
18.
19.         class JobPositionSpider(scrapy.Spider):
20.             # 定义该 Spider 的名字
21.             name = 'job_position'
22.             # 定义该 Spider 允许爬取的域名
```

```
23.             allowed_domains = ['zhipin.com']
24.             # 定义该 Spider 爬取的首页列表
25.             start_urls = ['https://www.zhipin.com/c101280100/h_101280100/']
26.
27.             # 该方法负责提取 response 所包含的信息
28.             # response 代表下载器从 start_urls 中每个 URL 下载得到的响应
29.             def parse(self, response):
30.                 # 遍历页面上所有//div[@class="job-primary"]节点
31.                 for job_primary in
    response.xpath('//div[@class="job-primary"]'):
32.                     item = ZhipinspiderItem()
33.                     # 匹配//div[@class="job-primary"]节点下
    /div[@class="info-primary"]节点
34.                     # 也就是匹配到包含工作信息的<div.../>元素
35.                     info_primary =
    job_primary.xpath('./div[@class="info-primary"]')
36.                     item['title'] =
    info_primary.xpath('./h3/a/div[@class="job-title"]/text()').extract_first()
37.                     item['salary'] =
    info_primary.xpath('./h3/a/span[@class="red"]/text()').extract_first()
38.                     item['work_addr'] =
    info_primary.xpath('./p/text()').extract_first()
39.                     item['url'] =
    info_primary.xpath('./h3/a/@href').extract_first()
40.                     # 匹配//div[@class="job-primary"]节点
    下./div[@class="info-company"]节点下
41.                     # 的/div[@class="company-text"]的节点
42.                     # 也就是匹配到包含公司信息的<div.../>元素
43.                     company_text =
    job_primary.xpath('./div[@class="info-company"]' +
44.                         '/div[@class="company-text"]')
45.                     item['company'] =
    company_text.xpath('./h3/a/text()').extract_first()
46.                     company_info = company_text.xpath('./p/text()').extract()
47.                     if company_info and len(company_info) > 0:
48.                         item['industry'] = company_info[0]
```

```
49.                     if company_info and len(company_info) > 2:
50.                         item['company_size'] = company_info[2]
51.                     # 匹配//div[@class="job-primary"]节点
    下./div[@class="info-publis"]节点下
52.                     # 也就是匹配到包含发布人信息的<div.../>元素
53.                     info_publis =
    job_primary.xpath('./div[@class="info-publis"]')
54.                     item['recruiter'] =
    info_publis.xpath('./h3/text()').extract_first()
55.                     item['publish_date'] =
    info_publis.xpath('./p/text()').extract_first()
56.                     yield item
```

上面程序中，第 10 行代码修改了 start_urls 列表，重新定义了该 Spider 需要爬取的首页；接下来程序重写了 Spider 的 parse(self, response) 方法。

程序第 16 行代码使用 XPath 匹配所有的 '//div[@class="job-primary"]' 节点(每个节点都包含一份招聘信息)。因此 程序使用循环遍历每个 '//div[@class="job-primary"]' 节点，为每个节点都建立一个 ZhipinspiderItem 对象，并从该节点中提取项目感兴趣的信息存入 ZhipinspiderItem 对象中。

程序最后一行代码使用 yield 语句将 item 对象返回给 Scrapy 引擎。此处不能使用 return，因为 return 会导致整个方法返回，循环不能继续执行，而 yield 将会创建一个生成器。

Spider 使用 yield 将 item 返回给 Scrapy 引擎之后 ,Scrapy 引擎将这些 item 收集起来传给项目的 Pipeline，因此自然就到了使用 Scrapy 开发爬虫的第二步。

3. 编写 pipelines.py 文件，该文件负责将所爬取的数据写入文件或数据库中。

现在开始修改 pipelines.py 文件。为了简化开发，只在控制台打印 item 数据。下面是修改后的 pipelines.py 文件的内容。

```
0.  class ZhipinspiderPipeline(object):
1.      def process_item(self, item, spider):
2.          print("工作:" , item['title'])
3.          print("工资:" , item['salary'])
4.          print("工作地点:" , item['work_addr'])
5.          print("详情链接:" , item['url'])
6.
7.          print("公司:" , item['company'])
8.          print("行业:" , item['industry'])
9.          print("公司规模:" , item['company_size'])
10.
11.         print("招聘人:" , item['recruiter'])
12.         print("发布日期:" , item['publish_date'])
```

从上面第 2 行代码可以看到，ZhipinspiderPipeline 主要就是实现 process_item(self, item, spider) 方法，该方法的 item、spider 参数都由 Scrapy 引擎传入，Scrapy 引擎会自动将 Spider "捕获" 的所有 item 逐个传给 process_item(self, item, spider) 方法，因此该方法只需处理单个的 item 即可（不管爬虫总共爬取了多少个 item，process_item(self, item, spider) 方法只处理一个即可）。

经过上面三个步骤，基于 Scrapy 的爬虫基本开发完成。下面还需要修改 settings.py 文件进行一些简单的配置，比如增加 User-Agent 头。取消 settings.py 文件中如下代码行的注释，并将这些代码行改为如下形式：

```
BOT_NAME = 'ZhipinSpider'


SPIDER_MODULES = ['ZhipinSpider.spiders']
NEWSPIDER_MODULE = 'ZhipinSpider.spiders'
ROBOTSTXT_OBEY = True
# 配置默认的请求头
DEFAULT_REQUEST_HEADERS = {
    "User-Agent" : "Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:61.0) Gecko/20100101 Firefox/61.0",
    'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8'
}
# 配置使用 Pipeline
ITEM_PIPELINES = {
    'ZhipinSpider.pipelines.ZhipinspiderPipeline': 300,
}
```

回顾一下上面的开发过程，使用 Scrapy 开发爬虫的核心工作就是三步：

- 定义 Item 类。由于 Item 只是一个 DTO 对象，因此定义 Item 类很简单。
- 开发 Spider 类，这一步是核心。Spider 使用 XPath 从页面中提取项目所需的信息，并用这些信息来封装 Item 对象。
- 开发 Pipeline。Pipeline 负责处理 Spider 获取的 Item 对象。

> 注意，所有修改过的源文件（尤其是添加了中文注释的文件）都需要手动保存为 UTF-8 字符集，否则在运行爬虫时会出现错误。

经过上面步骤，这个基于 Scrapy 的 Spider 已经开发完成。在命令行窗口中进入 ZhipinSpider 项目目录下（不是进入 ZhipinSpider/ZhipinSpider 目录下），执行如下

命令来启动 Spider：

```
scrapy crawl job_position
```

上面 scrapy crawl 命令中的 job_position 就是前面定义的 Spider 名称（通过 JobPositionSpider 类的 name 属性指定）。运行上面命令，可以看到如图 1 所示的输出结果。

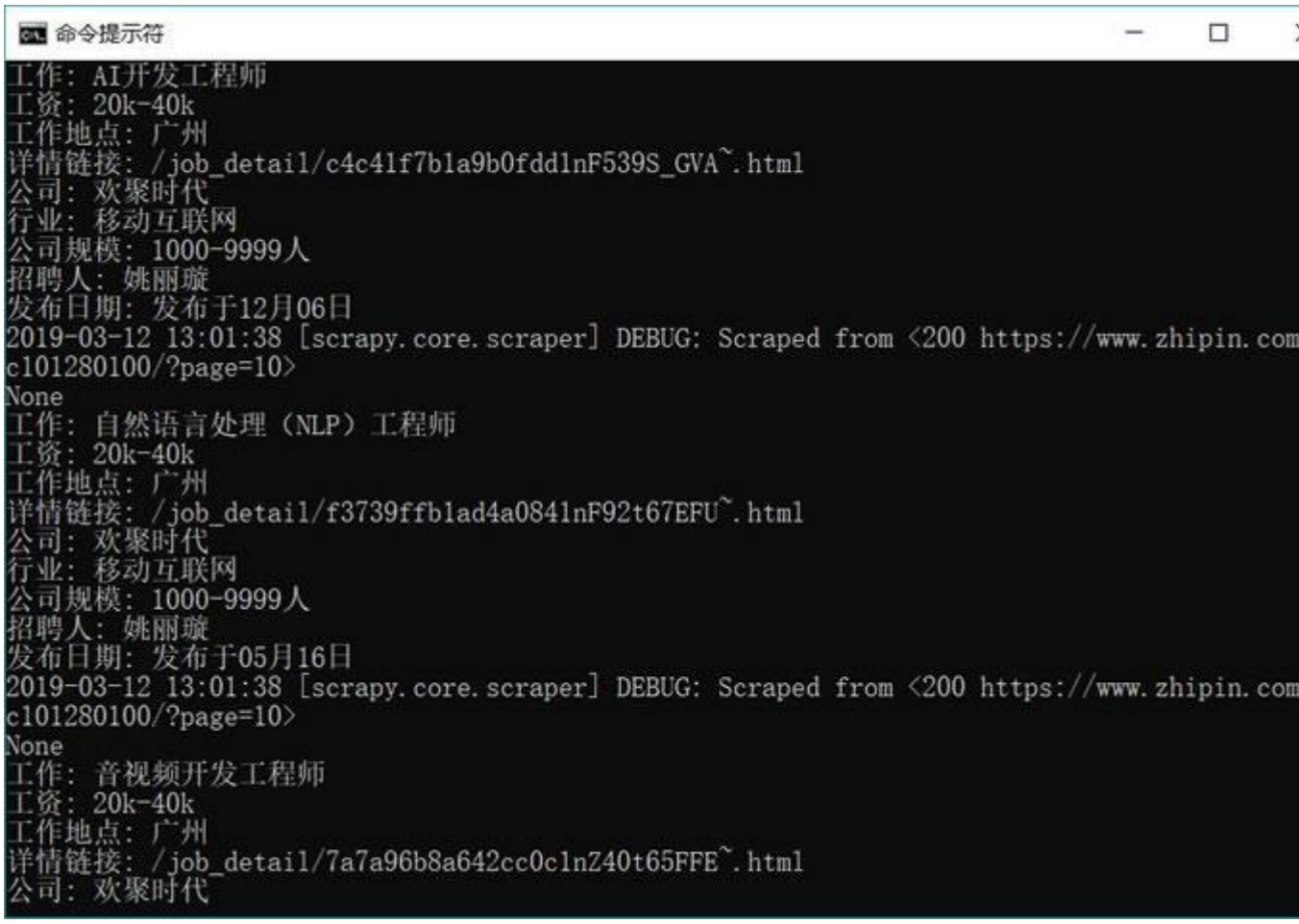


图 1 爬取第一页数据

从图 1 可以看出，该爬虫已经顺利爬取了 start_urls 列表所给出页面中的工作信息。但问题又出现了，该爬虫并未继续爬取下一页的工作信息。

爬虫可以自动爬取下一页信息吗？答案是肯定的。在提取完 response 中的所有“工作信息”之后，Spider 可以使用 XPath 找到页面中代表“下一页”的链接，然后使用 Request 发送请求即可。

通过浏览器查看源代码，可以看到 https://www.zhipin.com/c101280100/h_101280100/ 页面中包含分页信息的源代码，如图 2 所示。

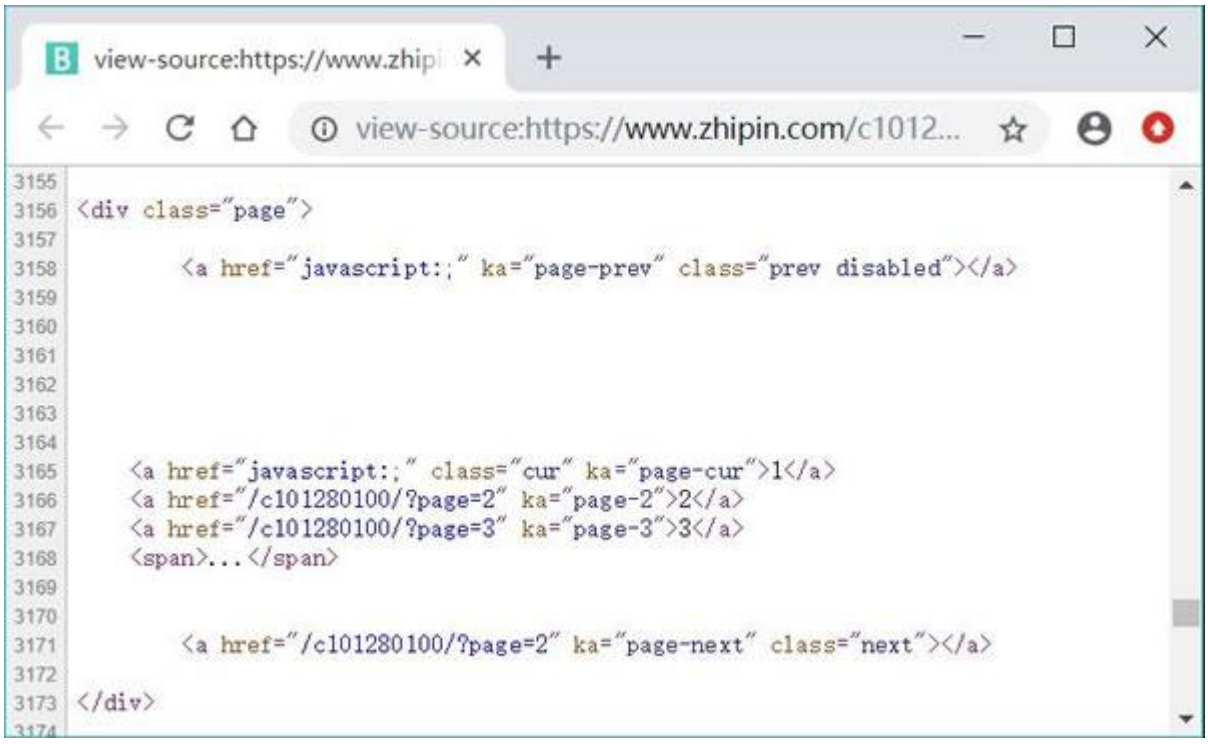


图 2 页面中包含分页信息的源代码

从图 2 可以看出，页面中“下一页”元素的 XPath 为 //div[@class='page']/a[@class='next']，因此程序需要在 JobPositionSpider 类的 parse(self, response) 方法的后面添加如下代码：

纯文本复制

```
1.          # 解析下一页的链接
2.          new_links = response.xpath('//div[@class="page"]/a[@class="next"]/@href').extract()
3.          if new_links and len(new_links) > 0:
4.              # 获取下一页的链接
5.              new_link = new_links[0]
6.              # 再次发送请求获取下一页数据
7.              yield scrapy.Request("https://www.zhipin.com" + new_link, callback=self.parse)
```

应该将上面这段代码放在 parse(self, response) 方法的后面，这样可以保证 Spider 在爬取页面中所有项目感兴趣的工作信息之后，才会向下一个页面发送请求。

上面程序中第 2 行代码解析页面中的“下一页”链接；第 7 行代码显式使用 scrapy.Request 来发送请求，并指定使用 self.parse 方法来解析服务器响应数据。需要说明的是，这是一个递归操作，即每当 Spider 解析完页面中项目感兴趣的工作信息之后，它总会再次请求“下一页”数据，通过这种方式即可爬取广州地区所有的热门职位信息。

再次运行 scrapy crawl job_position 命令来启动爬虫，即可看到该爬虫成功爬取了 10 页

职位信息，这是因为该页面默认只提供 10 页职位信息。

6.Python scrapy 爬虫数据保存到 MySQL 数据库

除将爬取到的信息写入文件中之外，程序也可通过修改 Pipeline 文件将数据保存到数据库中。为了使用数据库来保存爬取到的信息，在 MySQL 的 python 数据库中执行如下 SQL 语句来创建 job_inf 数据表：

```
CREATE TABLE job inf (
  id INT (11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
  title VARCHAR (255),
  salary VARCHAR (255),
  company VARCHAR (255),
  url VARCHAR(500),
  work_addr VARCHAR (255),
  industry VARCHAR (255),
  company_size VARCHAR(255),
  recruiter VARCHAR(255),
  publish_date VARCHAR (255)
)
```

然后将 Pipeline 文件改为如下形式，即可将爬取到的信息保存到 MySQL 数据库中：

纯文本复制

```
1.          # 导入访问 MySQL 的模块
2.          import mysql.connector
3.
4.          class ZhipinspiderPipeline(object):
5.              # 定义构造器，初始化要写入的文件
6.              def __init__(self):
```

```
7.                     self.conn = mysql.connector.connect(user='root',
password='32147',
8.                         host='localhost', port='3306',
9.                         database='python', use_unicode=True)
10.                    self.cur = self.conn.cursor()
11.                # 重写 close_spider 回调方法，用于关闭数据库资源
12.                def close_spider(self, spider):
13.                    print('----------关闭数据库资源----------')
14.                    # 关闭游标
15.                    self.cur.close()
16.                    # 关闭连接
17.                    self.conn.close()
18.                def process_item(self, item, spider):
19.                    self.cur.execute("INSERT INTO job_inf
VALUES(null, %s, %s, %s, %s, %s, \
20.                        %s, %s, %s, %s)", (item['title'], item['salary'],
item['company'],
21.                        item['url'], item['work_addr'], item['industry'],
22.                        item.get('company_size'), item['recruiter'],
item['publish_date']))
23.                    self.conn.commit()
```

上面程序中第 19 行代码使用 execute() 方法将 item 对象中的信息插入数据库中。

程序为该 Pipeline 类定义了构造器，该构造器可用于初始化数据库连接、游标；程序还为该 Pipeline 类重写了 close_spider() 方法，该方法负责关闭构造器中初始化的数据库资源。

使用 scrapy crawl job_position 命令启动爬虫，当程序运行结束之后，将会在 python 数据库的 job_inf 表中看到多了 300 条招聘信息。

7.Python Scrapy 突破反爬虫机制

对于 BOSS 直聘这种网站，当程序请求网页后，服务器响应内容包含了整个页面的 HTML 源代码，这样就可以使用爬虫来爬取数据。但有些网站做了一些“反爬虫”处理，其网页内容不是静态的，而是使用 JavaScript 动态加载的，此时的爬虫程序也需要做相应的改进。

使用 shell 调试工具分析目标站点

本项目爬取的目标站点是 https://unsplash.com/，该网站包含了大量高清、优美的图片。本项目的目标是爬虫程序能自动识别并下载该网站上的所有图片。

在开发该项目之前，依然先使用 Firefox 浏览该网站，然后查看该网站的源代码，将会看到页面的 <body.../> 元素几乎是空的，并没有包含任何图片。

现在使用 Scrapy 的 shell 调试工具来看看该页面的内容。在控制台输入如下命令，启动 shell 调试：

```
scrapy shell https://unsplash.com/
```

执行上面命令，可以看到 Scrapy 成功下载了服务器响应数据。接下来，通过如下命令来尝试获取所有图片的 src 属性（图片都是 img 元素，src 属性指定了图片的 URL）：

```
response.xpath('//img/@src').extract()
```

执行上面命令，将会看到返回一系列图片的 URL，但它们都不是高清图片的 URL。

还是通过 "Ctrl+Shift+I" 快捷键打开 Firefox 的调试控制台，再次向

https://unsplash.com/ 网站发送请求，接下来可以在 Firefox 的调试控制台中看到如图 1 所示的请求。

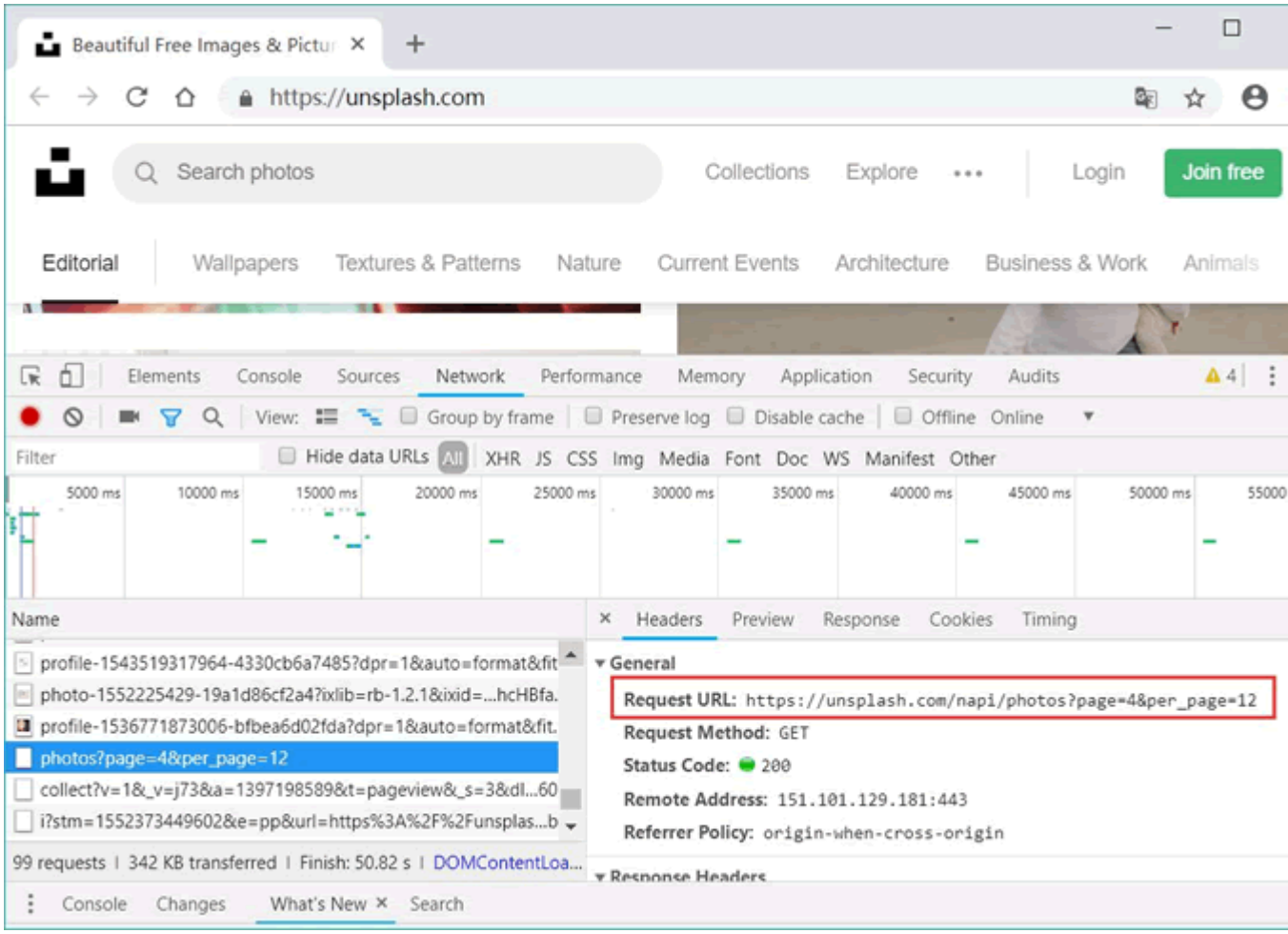


图 1 动态获取图片的请求

可见，该网页动态请求图片的 URL 如下：

```
https://unsplash.com/napi/photos?page=4&per_page=12
```

上面 URL 中的 page 代表第几页，per_page 代表每页加载的图片数。使用 Scrapy 的

shell 调试工具来调试该网址，输入如下命令：

```
scrapy shell https://unsplash.com/napi/photos?page=1&per_page=10
```

上面命令代表请求第 1 页，每页显示 10 张图片的响应数据。执行上面命令，服务器响应内容是一段 JSON 数据，接下来在 shell 调试工具中输入如下命令：

```
>>> import json
>>> len(json.loads(response.text))
10
```

从上面的调试结果可以看到，服务器响应内容是一个 JSON 数组（转换之后对应于 Python 的 list 列表），且该数组中包含 10 个元素。

使用 Firefox 直接请求 https://unsplash.com/napi/photos?page=1&per_page=12 地址（如果希望使用更专业的工具，则可选择 Postman），可以看到服务器响应内容如图 2 所示。

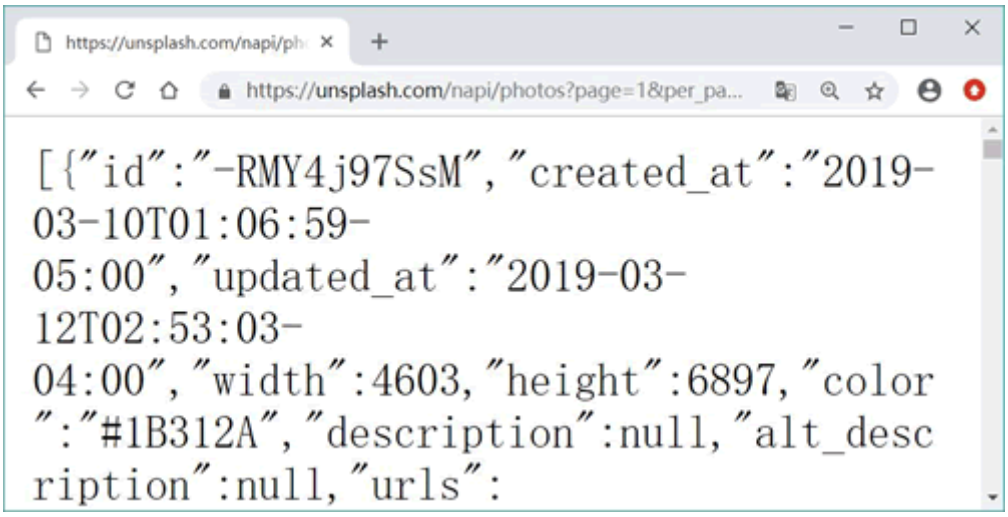


图 2 服务器响应的 JSON 数据

在图 2 所示为所有图片的 JSON 数据，每张图片数据都包含 id、created_at（创建时间）、updated_at（更新时间）、width（图片宽度）、height（图片高度）等基本信息和一个 links 属性，该属性值是一个对象（转换之后对应于 Python 的 dict），它包含了 self、html、download、download_location 属性，其中 self 代表浏览网页时的图片的 URL；而 download 才是要下载的高清图片的 URL。

网络爬虫毕竟是针对别人的网站“爬取” 数据的，而目标网站的结构随时可能发生改变，读者应该学习这种分析方法，而不是“生搬硬套”照抄本节的分析结果。

尝试在 shell 调试工具中查看第一张图片的下载 URL，应该在 shell 调试工具中输入如下命令：

```
>>> json.loads(response.text)[0]['links']['download']
'https://unsplash.com/photos/-RMY4j97SsM/download'
```

与图 2 中 JSON 数据对比不难看出，shell 调试工具输出的第一张图片的下载 URL 与图 2 所显示的第一张图片的下载 URL 完全相同。

由此得到一个结论，该网页加载时会自动向 https://unsplash.com/napi/photos?age=N&per_page=N 发送请求，然后根据服务器响应的 JSON 数据来动态加载图片。

> 由于该网页是“瀑布流”设计（所谓“瀑布流”设计，就是网页没有传统的分页按钮，而是让用户通过滚动条来实现分页，当用户向下拖动滚动条时，程序会动态载入新的分页），当我们在 Firefox 中拖动滚动条时，可以在 Firefox 的调试控制台中看到再次向 https://unsplash.com/napi/photos?page=N&per_page=N 发送了请求，只是 page 参数发生了改变。可见，为了不断地加载新的图片，程序只要不断地向该 URL 发送请求，并改变 page 参数即可。

经过以上分析，下面我们开始正式使用 Scrapy 来实现爬取高清图片。

使用 Scrapy 爬取高清图片

按照惯例，使用如下命令来创建一个 Scrapy 项目：

```
scrapy startproject UnsplashimageSpider
```

然后在命令行窗口中进入 UnsplashImageSpider 所在的目录下（不要进入 UnsplashImageSpider\UnsplashImageSpider 目录下），执行如下命令来生成 Spider 类：

```
scrapy genspider unsplash_image 'unsplash.com
```

上面两个命令执行完成之后，一个简单的 Scrapy 项目就创建好了。

接下来需要修改 UnsplashImageSpider\items.py 、UnsplashImageSpider\pipelines.py 、UnsplashImageSpider\spiders\unsplash_image.py 、UnsplashImageSpider\settings.py 文件，将它们全部改为使用 UTF-8 字符集来保存。

现在按照如下步骤来开发该爬虫项目：

1. 定义 Item 类。由于本项目的目标是爬取高清图片，因此其所使用的 Item 类比较简单，只要保存图片 id 和图片下载地址即可。

下面是该项目的 Item 类的代码：

```
import scrapy

class ImageItem(scrapy.Item):
    # 保存图片 id
    image_id = scrapy.Field()
    # 保存图片下载地址
    download = scrapy.Field()
```

上面程序为 Item 类定义了两个变量，分别用于保存图片 id 和图片下载地址。

2. 开发 Spider。开发 Spider 就是指定 Scrapy 发送请求的 URL，并实现 parse(self, response) 方法来解析服务器响应数据。

下面是该项目的 Spider 程序：

```
1.          import scrapy, json
2.          from UnsplashImageSpider.items import ImageItem
3.
4.          class UnsplashImageSpider(scrapy.Spider):
5.              # 定义 Spider 的名称
6.              name = 'unsplash_image'
7.              allowed_domains = ['unsplash.com']
8.              # 定义起始页面
9.              start_urls =
    ['https://unsplash.com/napi/photos?page=1&per_page=12']
10.             def __init__ (self):
11.                 self.page_index = 1
12.
13.             def parse(self, response):
14.                 # 解析服务器响应的 JSON 字符串
15.                 photo_list = json.loads(response.text) # ①
16.                 # 遍历每张图片
17.                 for photo in photo_list:
18.                     item = ImageItem()
19.                     item['image_id'] = photo['id']
20.                     item['download'] = photo['links']['download']
21.                     yield item
22.
23.                 self.page_index += 1
24.                 # 获取下一页的链接
25.                 next_link = 'https://unsplash.com/napi/photos?page='\
26.                     + str(self.page_index) + '&per_page=12'
27.                 # 继续获取下一页的图片
28.                 yield scrapy.Request(next_link, callback=self.parse)
```

上面程序中第 9 行代码指定的 URL 是本项目爬取的第一个页面，由于该页面的响应是一个 JSON 数据，因此程序无须使用 XPath 或 CSS 选择器来“提取”数据，而是直接使

用 json 模块的 loads() 函数来加载该响应数据即可。

在获取 JSON 响应数据之后 程序同样将 JSON 数据封装成 Item 对象后返回给 Scrapy 引擎。

> Spider 到底应该使用 XPath 或 CSS 选择器来提取响应数据，还是使用 JSON，完全取决于目标网站的响应内容，怎么方便怎么来！总之，提取到数据之后，将数据封装成 Item 对象后返回给 Scrapy 引擎就对了。

上面程序中倒数第 2 行代码定义了加载下一页数据的 URL，接下来使用 scrapy.Request 向该 URL 发送请求，并指定使用 self.parse 方法来处理服务器响应内容，这样程序就可以不断地请求下一页的图片数据。

3. 开发 Pipeline。Pipeline 负责保存 Spider 返回的 Item 对象（封装了爬取到的数据）。本项目爬取的目标是图片，因此程序得到图片的 URL 之后，既可将这些 URL 地址导入专门的下载工具中批量下载，也可在 Python 程序中直接下载。

本项目的 Pipeline 将使用 urllib.request 包直接下载。下面是该项目的 Pipeline 程序：

```
1.          from urllib.request import *
2.
3.          class UnsplashimagespiderPipeline(object):
4.              def process_item(self, item, spider):
5.                  # 每个 item 代表一个要下载的图片
6.                  print('----------' + item['image_id'])
7.                  real_url = item['download'] + "?force=true"
```

```
8.                  try:
9.                      pass
10.                     # 打开 URL 对应的资源
11.                     with urlopen(real_url) as result:
12.                         # 读取图片数据
13.                         data = result.read()
14.                         # 打开图片文件
15.                         with open("images/" + item['image_id'] + '.jpg', 'wb+')
    as f:
16.                             # 写入读取的数据
17.                             f.write(data)
18.                 except:
19.                     print('下载图片出现错误' % item['image_id'])
```

上面程序中第 7 行代码用于拼接下载图片的完整地址。可能有读者会问，为何要在图片下载地址的后面追加“?force=true” ？这并不是本项目所能决定的，读者可以把鼠标指针移动到 https://unsplash.com 网站中各图片右下角的下载按钮上，即可看到各图片的下载地址都会在 download 后追加“?force=true”，此处只是模拟这种行为而已。

程序中第 11 行代码使用 urlopen() 函数获取目标 URL 的数据，接下来即可读取图片数据，并将图片数据写入下载的目标文件中。

经过上面 3 步，基于 Scrapy 开发的高清图片爬取程序基本完成。接下来依然需要对 settings.py 文件进行修改，即增加一些自定义请求头（用于模拟浏览器），设置启用指定的 Pipeline。下面是本项目修改后的 settings.py 文件：

```
1.          BOT_NAME = 'UnsplashImageSpider'
2.
3.          SPIDER_MODULES = ['UnsplashImageSpider.spiders']
```

```
4.          NEWSPIDER_MODULE = 'UnsplashImageSpider.spiders'
5.          ROBOTSTXT_OBEY = True
6.
7.          # 配置默认的请求头
8.          DEFAULT_REQUEST_HEADERS = {
9.              "User-Agent" : "Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:61.0)
Gecko/20100101 Firefox/61.0",
10.             'Accept':
'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8'
11.         }
12.
13.         # 配置使用 Pipeline
14.         ITEM_PIPELINES = {
15.             'UnsplashImageSpider.pipelines.UnsplashimagespiderPipeline': 300,
16.         }
```

至此 ,这个可以爬取高清图片的爬虫项目开发完成 ,读者可以在 UnsplashImageSpider 目录下执行如下命令来启动爬虫。

```
scrapy crawl unsplash_image
```

运行该爬虫程序之后，可以看到在项目的 images 目录下不断地增加新的高清图片（对图片的爬取速度在很大程度上取决于网络下载速度），这些高清图片正是 https://unsplash.com 网站中所展示的图片。

8.Python Scrapy 反爬虫解决方案

爬虫的本质就是“抓取”第二方网站中有价值的数据，因此，每个网站都会或多或少地采用一些反爬虫技术来防范爬虫。比如前面介绍的通过 User-Agent 请求头验证是否为浏览器、使用 JavaScript 动态加载资源等，这些都是常规的反爬虫手段。

下面针对更强的反爬虫技术提供一些解决方案。

IP 地址验证

有些网站会使用 IP 地址验证进行反爬虫处理，程序会检查客户端的 IP 地址，如果发现同一个 IP 地址的客户端频繁地请求数据， 该网站就会判断该客户端是爬虫程序。

针对这种情况，我们可以让 Scrapy 不断地随机更换代理服务器的 IP 地址，这样就可以欺骗目标网站了。

为了让 Scrapy 能随机更换代理服务器，可以自定义一个下载中间件，让该下载中间件随机更换代理服务器即可。

Scrapy 随机更换代理服务器只要两步：

1. 打开 Scrapy 项目下的 middlewares.py 文件，在该文件中增加定义如下类：

```
class RandomProxyMiddleware (object) :
#动态设置代理服务器的 IP 地址
    def process request (self, request, spider):
        # get_random_proxy() 函数随机返回代理服务器的 IP 地址和端口
        request.meta["proxy"] = get_random_proxy()
```

上面程序通过自定义的下载中间件为 Scrapy 设置了代理服务器。程序中的 get_random_proxy() 函数需要能随机返回代理服务器的 IP 地址和端口 ,这就需要开发者事先准备好一系列代理服务器，该函数能随机从这些代理服务器中选择一个。

2. 通过 settings.py 文件设置启用自定义的下载中间件。在 settings.py 文件中增加如下配置代码：

```
#配置自定义的下载中间件
DOWNLOADER MIDDLEWARES = {
    'ZhipinSpider.middlewares.RandomProxyMiddleware': 543,
}
```

禁用 Cookie

有些网站可以通过跟踪 Cookie 来识别是否是同一个客户端。Scrapy 默认开启了 Cookie，这样目标网站就可以根据 Cookie 来识别爬虫程序是同一个客户端。

目标网站可以判断，如果同一个客户端在单位时间内的请求过于频繁，则基本可以断定这个客户端不是正常用户，很有可能是程序操作（比如爬虫），此时目标网站就可以禁用该客户端的访问。

针对这种情况，可以让 Scrapy 禁用 Cookie(Scrapy 不需要登录时才可禁用 Cookie)。在 settings.py 文件中取消如下代码的注释即可禁用 Cookie：

```
COOKIES_ENABLED = False
```

违反爬虫规则文件

在很多 Web 站点目录下都会提供一个 robots.txt 文件，在该文件中制定了一系列爬虫规则。例如，Weibo.com 网站下的 robots.txt 文件的内容如下：

Sitemap: http://weibo.com/sitemap.xml User-Agent: Baiduspider Disallow : User-agent : 360Spider Disallow : User-agent : Googlebot Disallow : User-agent : Sogou web spider Disallow : User-agent:bingbot Disallow : User-agent : smspider Disallow : User-agent : HaosouSpider Disallow : User-agent : YisouSpider Disallow : User-agent : * Disallow : /

该规则文件指定该站点只接受 Baidu 的网络爬虫，不接受其他爬虫程序。

为了让爬虫程序违反爬虫规则文件的限制，强行爬取站点信息，可以在 settings 文件中取消如下代码的注释来违反站点制定的爬虫规则：

```
#指定不遵守爬虫规则
ROBOTSTXT OBEY = False
```

限制访问频率

正如前面所提到的，当同一个 IP 地址、同一个客户端访问目标网站过于频繁时（正常用户的访问速度没那么快），其很可能会被当成机器程序（比如爬虫）禁止访问。

为了更好地模拟正常用户的访问速度，可以限制 Scrapy 的访问频率。在 settings 文件中取消如下代码的注释即可限制 Scrapy 的访问频率：

```
#开启访问频率限制
AUTOTHROTTLE ENABLED = True
#设置访问开始的延迟
AUTOTHROTTLE START DELAY = 5
#设置访问之间的最大延迟
```

```
AUTOTHROTTLE MAX DELAY = 60
#设置 Scrapy 并行发给每台远程服务器的请求数量
AUTOTHROTTLE TARGET CONCURRENCY= 1.0
#设置下载之后的自动延迟
DOWNLOAD DELAY = 3
```

图形验证码

有些网站为了防止机器程序访问，会做一些很“变态”的设计，它会记录同一个客户端、同一个 IP 地址的访问次数，只要达到一定的访问次数（不管你是正常用户，还是机器程序），目标网站就会弹出一个图形验证码让你输入，只有成功输入了图形验证码才能继续访问。

为了让机器识别这些图形验证码，通常有两种解决方式：

1. 使用 PIL、Libsvrn 等库自己开发程序来识别图形验证码。这种方式具有最大的灵活性，只是需要开发人员自己编码实现。
2. 通过第三方识别。有不少图形验证码的在线识别网站，它们的识别率基本可以做到 90% 以上。但是识别率高的在线识别网站通常都要收费，而免费的往往识别率不高，还不如自己写程序来识别。

9.Python Scrapy 整合 Selenium

某些网站要求用户必须先登录 ,然后才能获取网络数据 ,这样爬虫程序将无法随意爬取数据。

为了登录该网站，通常有两种做法：

1. 直接用爬虫程序向网站的登录处理程序提交请求，将用户名、密码、验证码等作为请求参数，登录成功后记录登录后的 Cookie 数据。
2. 使用真正的浏览器来模拟登录，然后记录浏览器登录之后的 Cookie 数据。

上面两种方式的目的是一样的，都是为了登录目标网站，记录登录后的 Cookie 数据。但这两种方式各有优缺点：

- 第一种方式需要爬虫开发人员自己来处理网站登录、Cookie 管理等复杂行为。这种方式的优点是完全由自己来控制程序，因此爬虫效率高、灵活性好；缺点是编程麻烦，尤其是当目标网站有非常强的反爬虫机制时，爬虫开发人员要花费大量的时间来处理。
- 第二种方式则完全使用真正的浏览器（比如 Firefox、Chrome 等）来模拟登录。这种方式的优点是简单、易用，而且几乎可以轻松登录所有网站（因为本来就是用浏览器登录的，正常用户怎么访问，爬虫启动的浏览器也怎么访问）；缺点是需要启动浏览器，用浏览器加载页面，因此效率较低。

在使用 Scrapy 开发爬虫程序时，经常会整合 Selenium 来启动浏览器登录。

需要指出的是，Selenium 本身与爬虫并没有多大的关系，Selenium 开始主要是作为 Web 应用的自动化测试工具来使用的，广大 Java 开发人员对 Selenium（开始是用 Java 写成的）应该非常熟悉。Selenium 可以驱动浏览器对 Web 应用进行测试，就像真正的用户在使用浏览器测试 Web 应用一样。后来的爬虫程序正是借助于 Selenium 的这个功能来驱动浏览器登录 Web 应用的。

为了在 Python 程序中使用 Selenium，需要以下 3 步：

1. 为 Python 安装 Selenium 库。运行如下命令，即可安装 Selenium：

```
pip install selenium
```

运行上面命令，安装成功后将会看到如下提示信息：

```
Installing collected packages: selenium
Successfully installed selenium-3.14.0
```

2. 为 Selenium 下载对应的浏览器驱动。Selenium 支持 Chrome、Firefox、Edge、Safari 等各种主流的浏览器，登录 https://selenium-python.readthedocs.io/installation.html#drivers 即可看到各浏览器驱动的下载链接。

本章我们将驱动 Firefox 来模拟登录，因此，通过其页面的链接来下载 Firefox 对应的驱动(对于 32 位操作系统，下载 32 位的驱动；对于 64 位操作系统，下载 64 位的驱动)。下载完成后将得到一个压缩包，解压该压缩包将得到一个 geckodriver.exe 文件，可以将该文件放在任意目录下，本项目将该驱动文件直接放在项目目录下。

3. 安装目标浏览器。比如本项目需要启动 Firefox 浏览器，那么就需要在目标机器上安装 Firefox 浏览器。

除安装 Firefox 浏览器之外，还应该将 Firefox 浏览器的可执行程序（firefox.exe）所在的目录添加到 PATH 环境变量中，以便 Selenium 能找到该浏览器。

经过上面 3 步，Python 程序即可使用 Selenium 来启动 Firefox 浏览器，并驱动 Firefox 浏览目标网站。

此处使用如下简单的程序进行测试。

```
1.          from selenium import webdriver
2.          import time
3.
4.          # 通过 executable_path 指定浏览器驱动的路径
5.          browser =
webdriver.Firefox(executable_path="WeiboSpider/geckodriver.exe")
6.          # 等待 3 秒，用于等待浏览器启动完成
7.          time.sleep(3)
8.          # 浏览指定网页
9.          browser.get("http://c.biancheng.net")
10.         # 暂停 5 秒
11.         time.sleep(5)
```

如果成功安装了 Selenium，并成功加载了 Firefox 浏览器驱动，且 Firefox 的可执行程序所在的目录位于 PATH 环境变量中，运行上面程序，Firefox 浏览器将会被启动，并自动访问 http://c.biancheng.net 站点。

在成功安装了 Selenium、驱动及目标浏览器之后，接下来我们在 Scrapy 项目中整合 Selenium，通过 Scrapy+Selenium 来登录 weibo.com。

按照惯例，首先创建一个 Scrapy 项目。在命令行窗口中执行如下命令：

```
scrapy startproject WeiboSpider
```

然后在命令行窗口中进入 WeiboSpider 所在的目录下（不要进入 WeiboSpider\WeiboSpider 目录下），执行如下命令来生成 Spider 类：

```
scrapy genspider weibo_post "weibo.com"
```

上面两个命令执行完成后，一个简单的 Scrapy 项目就创建好了。

接下来需要修改 WeiboSpider\items.py、WeiboSpider\pipelines.py、WeiboSpider\spiders\weibo_post.py、WeiboSpider\settings.py 文件，将它们全部改为使用 UTF-8 字符集来保存。

本项目不再重复介绍使用 Scrapy 爬取普通文本内容的方法，而是重点介绍在 Scrapy 项目中整合 Selenium 的方法，因此不需要修改 items.py 和 pipelines.py 文件。

本项目直接修改 weibo_post.py 文件，在 Spider 类中整合 Selenium 调用 Firefox 登录 weibo.com，接下来爬虫程序即可利用登录后的 Cookie 数据来访问 weibo 内容。

使用 Selenium 调用 Firefox 登录 weibo.com，首先肯定要对 weibo.com 的登录页面进行分析，不过前面两个项目已经详细介绍了这种分析过程，故此处直接给出分析结果：

- weibo 的登录页面是：https://weibo.com/login/。
- 在登录页面中输入用户名的文本框是：//input[@id="loginname"] 节点。
- 在登录页面中输入密码的文本框是：//input[@type="password"] 节点。

- 在登录页面中登录按钮是：//a[@node-type="submitBtn"] 节点。

通过分析得到以上内容之后，接下来可以在 Spider 类中额外定义一个方法来使用 Selenimn 调用 Firefox 登录 weibo.com。该 Spider 类的代码如下：

```
1.          import scrapy
2.          from selenium import webdriver
3.          import time
4.
5.          class WeiboPostSpider(scrapy.Spider):
6.              name = 'weibo_post'
7.              allowed_domains = ['weibo.com']
8.              start_urls = ['http://weibo.com/']
9.              def __init__(self):
10.                 # 定义保存登录成功之后的 cookie 的变量
11.                 self.login_cookies = []
12.             # 定义发送请求的请求头
13.             headers = {
14.                 "Referer": "https://weibo.com/login/",
15.                 'User-Agent': "Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:61.0)
    Gecko/20100101 Firefox/61.0"
16.             }
17.             def get_cookies(self):
18.                 '''使用 Selenium 模拟浏览器登录并获取 cookies'''
19.                 cookies = []
20.                 browser = webdriver.Firefox(executable_path="geckodriver.exe")
21.                 # 等待 3 秒，用于等待浏览器启动完成，否则可能报错
22.                 time.sleep(3)
23.                 browser.get("https://weibo.com/login/")  #①
24.                 # 获取输入用户名的文本框
25.                 elem_user =
    browser.find_element_by_xpath('//input[@id="loginname"]')
26.                 # 模拟输入用户名
```

```
27.                 elem_user.send_keys('xxxxxx@sina.com') #②
28.                 # 获取输入密码的文本框
29.                 elem_pwd =
    browser.find_element_by_xpath('//input[@type="password"]')
30.                 # 模拟输入密码
31.                 elem_pwd.send_keys('yyyyyy')  #③
32.                 # 获取提交按钮
33.                 commit =
    browser.find_element_by_xpath('//a[@node-type="submitBtn"]')
34.                 # 模拟单击提交按钮
35.                 commit.click()  #④
36.                 # 暂停 10 秒，等待浏览器登录完成
37.                 time.sleep(10)
38.                 #登录成功后获取 cookie
39.                 if "微博-随时随地发现新鲜事" in browser.title:
40.                     self.login_cookies = browser.get_cookies()
41.                 else:
42.                     print("登录失败！")
43.             # start_requests 方法会在 parse 方法之前执行，该方法可用于处理登录逻
    辑。
44.             def start_requests(self):
45.                 self.get_cookies()
46.                 print('====================', self.login_cookies)
47.                 # 开始访问登录后的内容
48.                 return
    [scrapy.Request('https://weibo.com/cyuyanzhongwenwang/',
49.                     headers=self.headers,
50.                     cookies=self.login_cookies,
51.                     callback=self.parse)]
52.
53.             # 解析服务器相应的内容
54.             def parse(self, response):
55.                 print('~~~~~~~parse~~~~~')
56.                 print("是否解析成功:", in response.text)
```

上面程序中 ① 号代码控制 Firefox 打开 weibo.com 的登录页面：https://weibo.com/login/ ;② 号代码控制 Firefox 在登录页面的用户名文本框中输入用户名；③ 号代码控制 Firefox 在登录页面的密码文本框中输入密码：④ 号代码模拟用户单击登录页面中的“登录”按钮。

上面 Spider 程序重写了两个方法，start_requests(self) 和 parse(self, response)，其中 start_request(self) 方法会在 Scrapy 发送请求之前执行，该方法中调用 self.get_cookies() 方法来登录 weibo.com，并保存登录之后的 Cookie 数据，这样该爬虫程序即可成功访问登录之后的 https://weibo.com/cyuyanzhongwenwang/ 页面（这是我的 weibo 主页，读者在测试时应换成登录账户对应的主页）。

本项目的 parse(self, response) 方法并未 yield item，只是简单地判断了 response 中是否包含登录账号信息，因为本项目只是示范在 Scrapy 项目中如何整合 Selenium 进行登录，至于登录之后如何提取信息，前面两个项目已多次介绍，故本项目不再重复讲解。

接下来依然需要对 settings.py 文件进行修改 ,增加一些自定义请求头(用于模拟浏览器)，设置启用指定的 Pipeline。下面是本项目修改后的 settings.py 文件：

```
BOT_NAME = 'WeiboSpider'


SPIDER_MODULES = ['WeiboSpider.spiders']
NEWSPIDER_MODULE = 'WeiboSpider.spiders'
```

```
ROBOTSTXT_OBEY = False


# 配置默认的请求头
DEFAULT_REQUEST_HEADERS = {
    "User-Agent" : "Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:61.0)
Gecko/20100101 Firefox/61.0",
    'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8'
}


# 配置使用 Pipeline
ITEM_PIPELINES = {
    'WeiboSpider.pipelines.WeibospiderPipeline': 300,
}
```

留心上面的 ROBOTSTXT_OBEY 配置，这行配置代码指定该爬虫程序不遵守该站点下的 robot.txt 规则文件（Scrapy 默认遵守 robot.txt 规则文件），强行爬取该站点的内容。

经过上面配置，接下来在 WeiboSpider 目录下执行如下命令来启动爬虫。

```
scrapy crawl weibo_post
```

运行该爬虫程序，即可看到它会自动启动 Firefox 来登录 weibo.com ，并可以在命令行窗口中看到对应解析成功的输出信息。

从该爬虫程序的运行过程来看，整合 Selenium 之后 Scrapy 的运行速度明显慢了很多。因此，Scrapy 通常只使用 Selenium 控制浏览器执行登录，不会使用 Selenium 控制浏览器执行普通下载，普通下载使用 Scrapy 自己的下载中间件即可（效率更高）。

一句话，只要技术到位，网络上没有爬取不到的数据。当然，如果有些网站的数据属于机密数据，并且这些网站也已经采取种种措施来阻止非法访问，但是你非要越过层层限制去访问这些数据，这就涉嫌触犯法律了。因此，爬虫也要适可而止。