Python基础教程

Beginning Python From Novice to Professional, 3rd Edition

Stephen CUI[1]

October 30, 2022

[1]cuixuanStephen@gmail.com

Contents

1 开箱即用 **4**

1.0.1 包 . . . 4

1.1 探索模块 . . . 4

1.1.1 模块包含什么 . . . 5

1.1.2 变量__all__ . . . 5

1.1.3 使用 help 获取帮助 . . . 5

1.1.4 文档 . . . 6

1.1.5 使用源代码 . . . 6

1.2 标准库：一些深受欢迎的模块 . . . 6

1.2.1 sys . . . 7

1.3 os . . . 7

1.3.1 fileinput . . . 8

1.3.2 集合、堆和双端队列 . . . 10

1.3.3 time . . . 13

1.3.4 random . . . 14

1.3.5 shelve 和 json . . . 16

Table 1: 复看重点

知识点　章节（位置）	是否复看
生成器	×

Chapter 1

开箱即用

1.0.1 包

为组织模块，可将其编组为包（package）。包其实就是另一种模块，但有趣的是它们可包含其他模块。模块存储在扩展名为.py的文件中，而包则是一个目录。要被Python视为包，目录必须包含文件`__init__.py`。如果像普通模块一样导入包，文件`__init__.py`的内容就将是包的内容。

要将模块加入包中，只需将模块文件放在包目录中即可。你还可以在包中嵌套其他包。例如，要创建一个名为drawing的包，其中包含模块shapes和colors，需要创建如Table 1.1所示的文件和目录（UNIX路径名）。

完成这些准备工作后，下面的语句都是合法的：

```
1  import drawing
2  import drawing.colors
3  from drawing import shapes
```

执行第1条语句后，便可使用目录drawing中文件`__init__.py`的内容，但不能使用模块shapes和colors的内容。执行第2条语句后，便可使用模块colors，但只能通过全限定名drawing.colors来使用。执行第3条语句后，便可使用简化名（即shapes）来使用模块shapes。请注意，这些语句只是示例，并不用像这里做的那样，先导入包再导入其中的模块。换而言之，完全可以只使用第2条语句，第3条语句亦如此。

1.1 探索模块

这是一种很有用的技能，因为你编写程序可能会遇到很多很有用的模块。

Table 1.1: 一种简单的包布局

文件/目录	描述
`~/python/`	PYTHONPATH中的目录
`~/python/drawing/`	包目录（包drawing）
`~/python/drawing/__init__.py`	包代码（模块drawing）
`~/python/drawing/colors.py`	模块colors
`~/python/drawing/shapes.py`	模块shapes

1.1.1 模块包含什么

要探索模块，最直接的方式是使用Python解释器进行研究。假设导入copy标准模块`import copy`。

使用dir

要查明模块包含哪些东西，可使用函数dir，它列出对象的所有属性（对于模块，它列出所有的函数、类、变量等）。如果将dir(copy)的结果打印出来，将是一个很长的名称列表（请试试看）。在这些名称中，有几个以下划线打头。根据约定，这意味着它们并非供外部使用。有鉴于此，使用一个简单的列表推导将这些名称过滤掉。

```
[n for n in dir(copy) if not n.startswith('_')]
# ['Error', 'copy', 'deepcopy', 'dispatch_table', 'error']
```

1.1.2 变量__all__

可以使用列表推导来猜测可在模块copy中看到哪些内容，然而可直接咨询这个模块来获得正确的答案。你可能注意到了，在dir(copy)返回的完整清单中，包含名称`__all__`。这个变量包含一个列表，它与前面使用列表推导创建的列表类似，但是在模块内部设置的。下面来看看这个列表包含的内容：

```
copy.__all__  # ['Error', 'copy', 'deepcopy']
```

这个`__all__`列表是怎么来的呢？为何要提供它？第一个问题很容易回答：它是在模块copy中像下面这样设置的（这些代码是直接从copy.py复制而来的）：

```
__all__ = ['Error', 'copy', 'deepcopy']
```

为何要提供它呢？旨在定义模块的公有接口。具体地说，它告诉解释器从这个模块导入所有的名称意味着什么。因此，如果你使用`from copy import *`。将只能得到变量`__all__`中列出的3个函数。要导入`dispatch_table`，必须显式地：导入copy并使用`copy.dispatch_table`；或者使用

```
from copy import dispatch_table
```

。

编写模块时，像这样设置`__all__`也很有用。因为模块可能包含大量其他程序不需要的变量、函数和类，比较周全的做法是将它们过滤掉。如果不设置`__all__`，则会在以import *方式导入时，导入所有不以下划线打头的全局名称。

1.1.3 使用 help 获取帮助

前面一直在巧妙地利用你熟悉的各种Python函数和特殊属性来探索模块copy。对这种探索来说，交互式解释器是一个强大的工具，因为使用它来探测模块时，探测的深度仅受限于你对Python语言的掌握程度。然而，有一个标准函数可提供你通常需要的所有信息，它就是help。

```
help(copy.copy)
# Help on function copy in module copy:

# copy(x)
#     Shallow copy operation on arbitrary Python objects.

#     See the module's __doc__ string for more info.
```

上述帮助信息指出，函数copy只接受一个参数x，且执行的是浅复制。文档字符串就是在函数开头编写的字符串，用于对函数进行说明，而函数的属性`__doc__`可能包含这个字符串。从前面的帮助信息可知，模块也可能有文档字符串（它们位于模块的开头），而类也可能如此（位于类的开头）。

实际上，前面的帮助信息是从函数copy的文档字符串中提取的：

```
print(copy.copy.__doc__)
# Shallow copy operation on arbitrary Python objects.

#     See the module's __doc__ string for more info.
```

相比于直接查看文档字符串，使用help的优点是可获取更多的信息，如函数的特征标（即它接受的参数）。请尝试对模块copy本身调用help，看看将显示哪些信息。这将打印大量的信息，包括对copy和deepcopy之间差别的详细讨论（大致而言，deepcopy(x)创建x的属性的副本并依此类推；而copy(x)只复制x，并将副本的属性关联到x的属性值）。

1.1.4 文档

显然，文档是有关模块信息的自然来源。

```
print(range.__doc__)
# range(stop) -> range object
# range(start, stop[, step]) -> range object

# Return an object that produces a sequence of integers from start (inclusive)
# to stop (exclusive) by step.  range(i, j) produces i, i+1, i+2, ..., j-1.
# start defaults to 0, and stop is omitted!  range(4) produces 0, 1, 2, 3.
# These are exactly the valid indices for a list of 4 elements.
# When step is given, it specifies the increment (or decrement).
```

这样就获得了函数range的准确描述。

1.1.5 使用源代码

在大多数情况下，前面讨论的探索技巧都够用了。但要真正理解Python语言，可能需要了解一些不阅读源代码就无法了解的事情。事实上，要学习Python，阅读源代码是除动手编写代码外的最佳方式。

实际阅读源代码应该不成问题，但源代码在哪里呢？假设你要阅读标准模块copy的代码，可以在什么地方找到呢？一种办法是像解释器那样通过sys.path来查找，但更快捷的方式是查看模块的特性`__file__`。

```
print(copy.__file__)
# d:\Python3.11.1\Lib\copy.py
```

如果列出的文件名以.pyc结尾，可打开以.py结尾的相应文件。

> 在文本编辑器中打开标准库文件时，存在不小心修改它的风险。这可能会破坏文件。因此关闭文件时，千万不要保存你可能对其所做的修改。

1.2 标准库：一些深受欢迎的模块

在Python中，短语“开箱即用”（batteries included）最初是由Frank Stajano提出的，指的是Python丰富的标准库。

Table 1.2: 模块sys中一些重要的函数和变量

函数/变量	描述
argv	命令行参数，包括脚本名
exit([arg])	退出当前程序，可通过可选参数指定返回值或错误消息
modules	一个字典，将模块名映射到加载的模块
path	一个列表，包含要在其中查找模块的目录的名称
platform	一个平台标识符，如sunos5或win32
stdin	标准输入流——一个类似于文件的对象
stdout	标准输出流——一个类似于文件的对象
stderr	标准错误流——一个类似于文件的对象

1.2.1 sys

模块sys让你能够访问与Python解释器紧密相关的变量和函数，Table 1.2列出了其中的一些。

变量sys.argv包含传递给Python解释器的参数，其中包括脚本名。

函数sys.exit退出当前程序。（在第8章讨论的try/finally块中调用它时，finally子句依然会执行。）你可向它提供一个整数，指出程序是否成功，这是一种UNIX约定。在大多数情况下，使用该参数的默认值（0，表示成功）即可。也可向它提供一个字符串，这个字符串将成为错误消息，对用户找出程序终止的原因很有帮助。在这种情况下，程序退出时将显示指定的错误消息以及一个表示失败的编码。

映射sys.modules将模块名映射到模块（仅限于当前已导入的模块）。

变量sys.path在本章前面讨论过，它是一个字符串列表，其中的每个字符串都是一个目录名，执行import语句时将在这些目录中查找模块。

变量sys.platform（一个字符串）是运行解释器的“平台”名称。这可能是表示操作系统的名称（如sunos5或win32），也可能是表示其他平台类型（如Java虚拟机）的名称（如java1.4.0）——如果你运行的是Jython。

变量sys.stdin、sys.stdout和sys.stderr是类似于文件的流对象，表示标准的UNIX概念：标准输入、标准输出和标准错误。简单地说，Python从sys.stdin获取输入（例如，用于input中），并将输出打印到sys.stdout。

举个例子，来看看按相反顺序打印参数的问题。从命令行调用Python脚本时，你可能指定一些参数，也就是所谓的命令行参数。这些参数将放在列表sys.argv中，其中sys.argv[0]为Python脚本名。

```
# reverseargs.py
import sys
args = sys.argv[1:]
args.reverse()
print(' '.join(args))
```

创建了一个sys.argv的副本。也可修改sys.argv，但一般而言，不这样做更安全，因为程序的其他部分可能依赖于包含原始参数的sys.argv。另外，注意到我跳过了sys.argv的第一个元素，即脚本的名称。

在命令行中输入`python reverseargs.py this is a test`，就能得到`test a is this`。

1.3 os

模块os让你能够访问多个操作系统服务。它包含的内容很多，表10-3只描述了其中几个最有用的函数和变量。除此之外，os及其子模块os.path还包含多个查看、创建和删除目录及文件的函数，以及一些

Table 1.3: 模块os中一些重要的函数和变量

函数/变量	描述
environ	包含环境变量的映射
system(command)	在子shell中执行操作系统命令
sep	路径中使用的分隔符
pathsep	分隔不同路径的分隔符
linesep	行分隔符（'\n'、'\r'或'\r\n'）
urandom(n)	返回n个字节的强加密随机数据

操作路径的函数。

映射os.environ包含本章前面介绍的环境变量。例如，要访问环境变量PYTHONPATH，可使用表达式os.environ['PYTHONPATH']。这个映射也可用于修改环境变量，但并非所有的平台都支持这样做。

函数os.system用于运行外部程序。还有其他用于执行外部程序的函数，如execv和popen。前者退出Python解释器，并将控制权交给被执行的程序，而后者创建一个到程序的连接（这个连接类似于文件）。

变量os.sep是用于路径名中的分隔符。在UNIX（以及macOS的命令行Python版本）中，标准分隔符为/。在Windows中，标准分隔符为\\（这种Python语法表示单个反斜杠）。

可使用os.pathsep来组合多条路径，就像PYTHONPATH中那样。pathsep用于分隔不同的路径名：在UNIX/macOS中为:，而在Windows中为;。

变量os.linesep是用于文本文件中的行分隔符：在UNIX/OS X中为单个换行符（\n），在Windows中为回车和换行符（\r\n）。

函数urandom使用随系统而异的“真正”（至少是强加密）随机源。如果平台没有提供这样的随机源，将引发NotImplementedError异常。

webbrowser

函数os.system可用于完成很多任务，但就启动Web浏览器这项任务而言，有一种更佳的解决方案：使用模块webbrowser。这个模块包含一个名为open的函数，让你能够启动启动Web浏览器并打开指定的URL。例如，要让程序在Web浏览器中打开Python网站（启动浏览器或使用正在运行的浏览器，只需像下面这样做：

```
import webbrowser
webbrowser.open('http://www.python.org')
```

这将弹出指定的网页。

1.3.1 fileinput

模块fileinput让你能够轻松地迭代一系列文本文件中的所有行。如果你这样调用脚本（假设是在UNIX命令行中）：

```
python some_script.py file1.txt file2.txt file3.txt
```

就能够依次迭代文件file1.txt到file3.txt中的行。

Table 1.4描述了模块fileinput中最重要的函数。

fileinput.input是其中最重要的函数，它返回一个可在for循环中进行迭代的对象。如果要覆盖默认行

Table 1.4: 模块fileinput中一些重要的函数和变量

函数	描述
input([files[, inplace[, backup]]])	帮助迭代多个输入流中的行
filename()	返回当前文件的名称
lineno()	返回（累计的）当前行号
filelineno()	返回在当前文件中的行号
isfirstline()	检查当前行是否是文件中的第一行
isstdin()	检查最后一行是否来自sys.stdin
nextfile()	关闭当前文件并移到下一个文件
close()	关闭序列

为（确定要迭代哪些文件），可以序列的方式向这个函数提供一个或多个文件名。还可将参数inplace设置为True（inplace=True），这样将就地进行处理。对于你访问的每一行，都需打印出替代内容，这些内容将被写回到当前输入文件中。就地进行处理时，可选参数backup用于给从原始文件创建的备份文件指定扩展名。

函数fileinput.filename返回当前文件（即当前处理的行所属文件）的文件名。

函数fileinput.lineno返回当前行的编号。这个值是累计的，因此处理完一个文件并接着处理下一个文件时，不会重置行号，而是从前一个文件最后一行的行号加1开始。

函数fileinput.filelineno返回当前行在当前文件中的行号。每次处理完一个文件并接着处理下一个文件时，将重置这个行号并从1重新开始。

函数fileinput.isfirstline在当前行为当前文件中的第一行时返回True，否则返回False。

函数fileinput.isstdin在当前文件为sys.stdin时返回True，否则返回False。

函数fileinput.nextfile关闭当前文件并跳到下一个文件，且计数时忽略跳过的行。这在你知道无需继续处理当前文件时很有用。例如，如果每个文件包含的单词都是按顺序排列的，而你要查找特定的单词，则过了这个单词所在的位置后，就可放心地跳到下一个文件。

函数fileinput.close关闭整个文件链并结束迭代。

fileinput小案例

假设你编写了一个Python脚本，并想给其中的代码行加上行号。鉴于你希望这样处理后程序依然能够正常运行，因此必须在每行末尾以注释的方式添加行号。为让这些行号对齐，可使用字符串格式设置功能。假设只允许每行代码最多包含50个字符，并在第51个字符处开始添加注释。

```
# numberlines.py
import fileinput

for line in fileinput.input(inplace=True):
    line = line.rstrip()
    num = fileinput.lineno()
    print('{:<50} # {:2d}'.format(line, num))
```

如果像下面这样运行这个程序，并将其作为参数传入：

```
$ python numberlines.py numberlines.py
```

注意到，程序本身也被修改了，如果像上面这样运行它多次，每行都将包含多个行号。

```
# numberlines.py                                    #  1
import fileinput                                    #  2
                                                    #  3
for line in fileinput.input(inplace=True):          #  4
    line = line.rstrip()                            #  5
    num = fileinput.lineno()                        #  6
    print('{:<50} # {:2d}'.format(line, num))       #  7
```

警告

务必慎用参数inplace，因为这很容易破坏文件。你应在不设置inplace的情况下仔细测试程序（这样将只打印结果），确保程序能够正确运行后再让它修改文件。

1.3.2 集合、堆和双端队列

集合

很久以前，集合是由模块sets中的Set类实现的。在较新的版本中，集合是由内置类set实现的，这意味着你可直接创建集合，而无需导入模块sets。

可使用序列（或其他可迭代对象）来创建集合，也可使用花括号显式地指定。请注意，不能仅使用花括号来创建空集合，因为这将创建一个空字典。

必须在不提供任何参数的情况下调用set。集合主要用于成员资格检查，因此将忽略重复的元素。与字典一样，集合中元素的排列顺序是不确定的，因此不能依赖于这一点。

```
set(range(10))  # {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

type({})  # dict

{0, 1, 2, 3, 0, 1, 2, 3, 4, 5}
# {0, 1, 2, 3, 4, 5}
```

除成员资格检查外，还可执行各种标准集合操作，如并集和交集，为此可使用对整数执行按位操作的运算符。

```
a = {1, 2, 3}
b = {2, 3, 4}
a.union(b)  # {1, 2, 3, 4}
a | b  # {1, 2, 3, 4}

c = a & b
c.issubset(a)  # True
c <= a  # True

c.issuperset(a)  # False
c >= a  # False

a.intersection(b)  # {2, 3}
a & b  # {2, 3}
```

Table 1.5: 模块heapq中一些重要的函数和变量

函数	描述
heappush(heap, x)	将x压入堆中
heappop(heap)	从堆中弹出最小的元素
heapify(heap)	让列表具备堆特征
heapreplace(heap, x)	弹出最小的元素，并将x压入堆中
nlargest(n, iter)	返回iter中n个最大的元素
nsmallest(n, iter)	返回iter中n个最小的元素

```
a.difference(b)  # {1}
a - b  # {1}

# A | B - A & B
a.symmetric_difference(b)  # {1, 4}
a ^ b  # {1, 4}
a.copy()  # {1, 2, 3}
a.copy() is a  # False
```

另外，还有对应于各种就地操作的方法以及基本方法add和remove。

集合是可变的，因此不能用作字典中的键。另一个问题是，集合只能包含不可变（可散列）的值，因此不能包含其他集合。由于在现实世界中经常会遇到集合的集合，因此这可能是个问题。所幸还有frozenset类型，它表示不可变（可散列）的集合。

```
a = set()
b = set()
a.add(b)  # TypeError: unhashable type: 'set'
a.add(frozenset(b))
a  # {frozenset()}
```

构造函数frozenset创建给定集合的副本。在需要将集合作为另一个集合的成员或字典中的键时，frozenset很有用。

堆

另一种著名的数据结构是堆（heap），它是一种优先队列。优先队列让你能够以任意顺序添加对象，并随时（可能是在两次添加对象之间）找出（并删除）最小的元素。相比于列表方法min，这样做的效率要高得多。

Python没有独立的堆类型，而只有一个包含一些堆操作函数的模块。这个模块名为heapq（其中的q表示队列），它包含6个函数（如Table 1.5所示），其中前4个与堆操作直接相关。必须使用列表来表示堆对象本身。

函数heappush用于在堆中添加一个元素。请注意，不能将它用于普通列表，而只能用于使用各种堆函数创建的列表。原因是元素的顺序很重要（虽然元素的排列顺序看起来有点随意，并没有严格地排序）。

```
from random import shuffle
from heapq import *

data = list(range(10))
shuffle(data)
data
heap = []
for n in data:
    heappush(heap, n)

heappush(heap, .5)
heap  # [0, 0.5, 1, 5, 3, 2, 6, 8, 7, 9, 4]
```

元素的排列顺序并不像看起来那么随意。它们虽然不是严格排序的，但必须保证一点：位置i处的元素x_i总是大于位置i // 2处的元素$x_{[i//2]}$（反过来说就是小于x_{2i}和x_{2i+1}处的元素）。这是底层堆算法的基础，称为堆特征（**heap property**）。

函数heappop弹出最小的元素（总是位于索引0处），并确保剩余元素中最小的那个位于索引0处（保持堆特征）。虽然弹出列表中第一个元素的效率通常不是很高，但这不是问题，因为heappop会在幕后做些巧妙的移位操作。

```
heappop(heap)  # 0
heappop(heap)  # 0.5
heappop(heap)  # 1
heap  # [2, 4, 3, 5, 6, 7, 8, 9]
```

函数heapify通过执行尽可能少的移位操作将列表变成合法的堆（即具备堆特征）。如果你的堆并不是使用heappush创建的，应在使用heappush和heappop之前使用这个函数。

```
heap = [5, 8, 0, 3, 6, 7, 9, 1, 4, 2]
heapify(heap)
heap  # [0, 1, 5, 3, 2, 7, 9, 8, 4, 6]
```

函数heapreplace用得没有其他函数那么多。它从堆中弹出最小的元素，再压入一个新元素。相比于依次执行函数heappop和heappush，这个函数的效率更高。

```
heapreplace(heap, .5)
heap  # [0.5, 1, 5, 3, 2, 7, 9, 8, 4, 6]
heapreplace(heap, 10)
heap  # [1, 2, 5, 3, 6, 7, 9, 8, 4, 10]
```

模块heapq中还有两个函数没有介绍：nlargest(n, iter)和nsmallest(n, iter)，:分别用于找出可迭代对象iter中最大和最小的n个元素。这种任务也可通过先排序（如使用函数sorted）再切片来完成，但堆算法的速度更快，使用的内存更少（而且使用起来也更容易）。

```
data = list(range(1000))
shuffle(data)
nlargest(5, data)  # [999, 998, 997, 996, 995]
nsmallest(5, data)  # [0, 1, 2, 3, 4]
```

双端队列（及其他集合）

在需要按添加元素的顺序进行删除时，双端队列很有用。在模块collections中，包含类型deque以及其他几个集合（collection）类型。

与集合（set）一样，双端队列也是从可迭代对象创建的：

```
from collections import deque
q = deque(range(5))
q.append(5)
q.appendleft(6)
q  # deque([6, 0, 1, 2, 3, 4, 5])
q.pop()  # 5
q.popleft()  # 6
q.rotate(3)
q  # deque([2, 3, 4, 0, 1])

q.rotate(-1)
q  # deque([3, 4, 0, 1, 2])
```

双端队列很有用，因为它支持在队首（左端）高效地附加和弹出元素，而使用列表无法这样做。另外，还可高效地旋转元素（将元素向右或向左移，并在到达一端时环绕到另一端）。双端队列对象还包含方法extend和extendleft，其中extend类似于相应的列表方法，而extendleft类似于appendleft。请注意，用于extendleft的可迭代对象中的元素将按相反的顺序出现在双端队列中。

```
q.extend([1, 2])
# deque([3, 4, 0, 1, 2, 1, 2])
q.extendleft([1, 2])
q  # deque([2, 1, 3, 4, 0, 1, 2, 1, 2])
```

1.3.3 time

模块time包含用于获取当前时间、操作时间和日期、从字符串中读取日期、将日期格式化为字符串的函数。日期可表示为实数（从“新纪元”1月1日0时起过去的秒数。“新纪元”是一个随平台而异的年份，在UNIX中为1970年），也可表示为包含9个整数的元组。Table 1.6解释了这些整数。例如，元组(2008, 1, 21, 12, 2, 56, 0, 21, 0)表示2008年1月21日12时2分56秒。这一天是星期一，2008年的第21天（不考虑夏令时）。

Table 1.6: Python日期元组中的字段

索引	字段	值
0	年	如2000、2001等
1	月	范围1 12
2	日	范围1 31
3	时	范围0 23
4	分	范围0 59
5	秒	范围0 61
6	星期	范围0 6，其中0表示星期一
7	儒略日	范围1 366
8	夏令时	0、1或-1

秒的取值范围为0 61，这考虑到了闰一秒和闰两秒的情况。夏令时数字是一个布尔值（True或False），但如果你使用-1，那么mktime［将时间元组转换为时间戳（从新纪元开始后的秒数）的函数］可能得到正确的值。Table 1.7描述了模块time中一些最重要的函数。

Table 1.7: 模块time中一些重要的函数和变量

函数	描述
asctime([tuple])	将时间元组转换为字符串
localtime([secs])	将秒数转换为表示当地时间的日期元组
mktime(tuple)	将时间元组转换为当地时间
sleep(secs)	休眠（什么都不做）secs秒
strptime(string[, format])	将字符串转换为时间元组
time()	当前时间（从新纪元开始后的秒数，以UTC为准）

函数time.asctime将当前时间转换为字符串，如下所示：

```
import time
time.asctime()  # 'Sat Feb 18 17:32:01 2023'
```

如果不想使用当前时间，也可向它提供一个日期元组（如localtime创建的日期元组）。要设置更复杂的格式，可使用函数strftime，标准文档对此做了介绍。

函数time.localtime将一个实数（从新纪元开始后的秒数）转换为日期元组（本地时间）。如果要转换为国际标准时间，应使用gmtime。

函数time.mktime将日期元组转换为从新纪元后的秒数，这与localtime的功能相反。

函数time.sleep让解释器等待指定的秒数。

函数time.strptime将一个字符串（其格式与asctime所返回字符串的格式相同）转换为日期元组。（可选参数format遵循的规则与strftime相同，详情请参阅标准文档。）

函数time.time返回当前的国际标准时间，以从新纪元开始的秒数表示。虽然新纪元随平台而异，但可这样进行可靠的计时：存储事件（如函数调用）发生前后time的结果，再计算它们的差。

还有两个较新的与时间相关的模块：datetime和timeit。前者提供了日期和时间算术支持，而后者可帮助你计算代码段的执行时间。“Python库参考手册”提供了有关这两个模块的详细信息。

1.3.4 random

模块random包含生成伪随机数的函数，有助于编写模拟程序或生成随机输出的程序。请注意，虽然这些函数生成的数字好像是完全随机的，但它们背后的系统是可预测的。如果你要求真正的随机（如用于加密或实现与安全相关的功能），应考虑使用模块os中的函数urandom。模块random中的SystemRandom类基于的功能与urandom类似，可提供接近于真正随机的数据。

Table 1.8列出了这个模块中一些重要的函数。

函数random.random是最基本的随机函数之一，它返回一个0 1（含）的伪随机数。除非这正是你需要的，否则可能应使用其他提供了额外功能的函数。

函数random.getrandbits以一个整数的方式返回指定数量的二进制位。

向函数random.uniform提供了两个数字参数a和b时，它返回一个a b（含）的随机（均匀分布的）实数。

Table 1.8: 模块random中一些重要的函数和变量

函数	描述
random()	返回一个0 ~ 1（含）的随机实数
getrandbits(n)	以长整数方式返回n个随机的二进制位
uniform(a, b)	返回一个$a \sim b$（含）的随机实数
randrange([start], stop, [step])	从range(start, stop, step)中随机地选择一个数
choice(seq)	从序列seq中随机地选择一个元素
shuffle(seq[, random])	就地打乱序列seq
sample(seq, n)	从序列seq中随机地选择n个值不同的元素

函数random.randrange是生成随机整数的标准函数。为指定这个随机整数所在的范围，你可像调用range那样给这个函数提供参数。例如，生成奇数或者偶数等等。

函数random.choice从给定序列中随机（均匀）地选择一个元素。

函数random.shuffle随机地打乱一个可变序列中的元素，并确保每种可能的排列顺序出现的概率相同。

函数random.sample从给定序列中随机（均匀）地选择指定数量的元素，并确保所选择元素的值各不相同。

来看几个使用模块random的示例。

随机选择一个时间

```
from time import *
from random import *
date1 = (2023, 1, 1, 0, 0, 0, -1, -1, -1)
date2 = (2023, 12, 31, 23, 59, 59, -1, -1, -1)
time1 = mktime(date1)
time2 = mktime(date2)
random_time = uniform(time1, time2)
print(asctime(localtime(random_time)))
# Tue Oct 17 09:08:16 2023
```

询问用户要掷多少个骰子、每个骰子有多少面

```
from random import randrange
num = int(input("How many dice?"))
sides = int(input("How many sides per die?"))
sum = 0
for i in range(num):
    sum += randrange(sides) + 1
print('The result is', sum)
# How many dice? 3
# How many sides per die? 6
# The result is 13
```

用户每次按回车键时都给他发一张牌

```
from random import shuffle
values = list(range(1, 11)) + 'Jack Queen King'.split()
suits = 'diamonds clubs hearts spades'.split()
deck = ['{} of {}'.format(v, s) for v in values for s in suits]
shuffle(deck)
while deck:
    input(deck.pop())
    # ignore = input(deck.pop())
```

请注意，如果在交互式解释器中尝试运行这个while循环，那么每当你按回车键时都将打印一个空字符串。这是因为input返回你输入的内容（什么都没有），然后这些内容将被打印出来。在普通程序中，将忽略input返回的值。要在交互式解释器中也忽略input返回的值，只需将其赋给一个你不会再理会的变量，并将这个变量命名为ignore。

1.3.5 shelve 和 json

如果需要的是简单的存储方案，模块shelve可替你完成大部分工作——你只需提供一个文件名即可。对于模块shelve，你唯一感兴趣的是函数open。这个函数将一个文件名作为参数，并返回一个Shelf对象，供你用来存储数据。你可像操作普通字典那样操作它（只是键必须为字符串），操作完毕（并将所做的修改存盘）时，可调用其方法close。

一个潜在的陷阱 至关重要的一点是认识到shelve.open返回的对象并非普通映射。

```
s = shelve.open('test')
s['x'] = ['a', 'b', 'c']
s['x'].append('d')
s['x']
```

当你查看shelf对象中的元素时，将使用存储版重建该对象，而当你将一个元素赋给键时，该元素将被存储。

要正确地修改使用模块shelve存储的对象，必须将获取的副本赋给一个临时变量，并在修改这个副本后再次存储：

```
s = shelve.open('test')
s['x'] = ['a', 'b', 'c']
temp = s['x']
temp.append('d')
s['x'] = temp
s['x']
```

还有另一种避免这个问题的办法：将函数open的参数writeback设置为True。这样，从shelf对象读取或赋给它的所有数据结构都将保存到内存（缓存）中，并等到你关闭shelf对象时才将它们写入磁盘中。如果你处理的数据不多，且不想操心这些问题，将参数writeback设置为True可能是个不错的主意。在这种情况下，你必须确保在处理完毕后将shelf对象关闭。为此，一种办法是像处理打开的文件那样，将shelf对象用作上下文管理器。

```
import sys
import shelve
def store_person(db):
    """
    Query user  for data and store it in the shelf object
    """
    pid = input('Enter unique ID number:')
    person = {}
    person['name'] = input('Enter name:')
    person['age'] = input('Enter age:')
    person['phone'] = input('Enter phone:')
    db[pid] = person
def lookup_person(db):
    """
    Query user for ID and desired field, and fetch the corresponding data from the shelf object
    """
    pid = input('Enter ID number:')
    field = input('What would you like to know? (name, age, phone)')
    field = field.strip().lower()

    print(field.capitalize() + ':', db[pid][field])
def print_help():
    print('The available commands are:')
    print('store : Stores information about a person')
    print('lookup : Looks up a person from ID number')
    print('quit : Save changes and exit')
    print('? : Prints this message')
def enter_command():
    cmd = input('Enter command (? for help): ')
    cmd = cmd.strip().lower()
    return cmd
def main():
    database = shelve.open('database')
    try:
        while True:
            cmd = enter_command()
            if cmd == 'store':
                store_person(database)
            elif cmd == 'lookup':
                lookup_person(database)
            elif cmd == '?':
                print_help()
            elif cmd == 'quit':
                return
    finally:
        database.close()

if __name__ == '__main__':
    main()
```

为确保数据库得以妥善的关闭，使用了try和finally。不知道什么时候就会出现问题，进而引发异常。如果程序终止时未妥善地关闭数据库，数据库文件可能受损，变得毫无用处。通过使用try和finally，可避免这样的情况发生。

如果要以这样的格式保存数据，也就是让使用其他语言编写的程序能够轻松地读取它们，可考虑使用JSON格式。Python标准库提供了用于处理JSON字符串（在这种字符串和Python值之间进行转换）的模块json。

1.3.6 re

有些人面临问题时会想：“我知道，我将使用正则表达式来解决这个问题。”这让他们面临的问题变成了两个。

—Jamie Zawinski