11. 标准库简介 —— 第二部分

第二部分涵盖了专业编程所需要的更高级的模块。这些模块很少用在小脚本中。

11.1. 格式化输出

reprlib 模块提供了一个定制化版本的 repr() 函数，用于缩略显示大型或深层嵌套的容器对象:

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
"{'a', 'c', 'd', 'e', 'f', 'g', ...}"
```

pprint 模块提供了更加复杂的打印控制，其输出的内置对象和用户自定义对象能够被解释器直接读取。当输出结果过长而需要折行时，“美化输出机制”会添加换行符和缩进，以更清楚地展示数据结构:

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...     'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

textwrap 模块能够格式化文本段落，以适应给定的屏幕宽度:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
...
>>> print(textwrap.fill(doc, width=40))
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

locale 模块处理与特定地域文化相关的数据格式。locale 模块的 format 函数包含一个 grouping 属性，可直接将数字格式化为带有组分隔符的样式:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()          # 获取语言区域设置的映射
>>> x = 1234567.8
>>> locale.format_string("%d", x, grouping=True)
```

```
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                       conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2. 模板

string 模块包含一个通用的 Template 类，具有适用于最终用户的简化语法。它允许用户在不更改应用逻辑的情况下定制自己的应用。

上述格式化操作是通过占位符实现的，占位符由 $ 加上合法的 Python 标识符（只能包含字母、数字和下划线）构成。一旦使用花括号将占位符括起来，就可以在后面直接跟上更多的字母和数字而无需空格分割。$$ 将被转义成单个字符 $:

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

如果在字典或关键字参数中未提供某个占位符的值，那么 substitute() 方法将抛出 KeyError。对于邮件合并类型的应用，用户提供的数据有可能是不完整的，此时使用 safe_substitute() 方法更加合适 —— 如果数据缺失，它会直接将占位符原样保留。

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  ...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Template 的子类可以自定义分隔符。例如，以下是某个照片浏览器的批量重命名功能，采用了百分号作为日期、照片序号和照片格式的占位符:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
...
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format):  ')
Enter rename style (%d-date %n-seqnum %f-format):  Ashley_%n%f

>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print('{0} --> {1}'.format(filename, newname))

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

模板的另一个应用是将程序逻辑与多样的格式化输出细节分离开来。这使得对 XML 文件、纯文本报表和 HTML 网络报表使用自定义模板成为可能。

11.3. 使用二进制数据记录格式

`struct` 模块提供了 `pack()` 和 `unpack()` 函数，用于处理不定长度的二进制记录格式。下面的例子展示了在不使用 `zipfile` 模块的情况下，如何循环遍历一个 ZIP 文件的所有头信息。Pack 代码 `"H"` 和 `"I"` 分别代表两字节和四字节无符号整数。`"<"` 代表它们是标准尺寸的小端字节序:

```
import struct

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3):                      # 显示前 3 个文件标头
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size     # 跳过下一个标头
```

11.4. 多线程

线程是一种对于非顺序依赖的多个任务进行解耦的技术。多线程可以提高应用的响应效率，当接收用户输入的同时，保持其他任务在后台运行。一个有关的应用场景是，将 I/O 和计算运行在两个并行的线程中。

以下代码展示了高阶的 `threading` 模块如何在后台运行任务，且不影响主程序的继续运行:

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')
```

```
background.join()    # 等待背景任务结束
print('Main program waited until background was done.')
```

多线程应用面临的主要挑战是，相互协调的多个线程之间需要共享数据或其他资源。为此，threading 模块提供了多个同步操作原语，包括线程锁、事件、条件变量和信号量。

尽管这些工具非常强大，但微小的设计错误却可以导致一些难以复现的问题。因此，实现多任务协作的首选方法是将所有对资源的请求集中到一个线程中，然后使用 queue 模块向该线程供应来自其他线程的请求。 应用程序使用 Queue 对象进行线程间通信和协调，更易于设计，更易读，更可靠。

11.5. 日志记录

logging 模块提供功能齐全且灵活的日志记录系统。在最简单的情况下，日志消息被发送到文件或 `sys.stderr`

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

这会产生以下输出:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

默认情况下，informational 和 debugging 消息被压制，输出会发送到标准错误流。其他输出选项包括将消息转发到电子邮件，数据报，套接字或 HTTP 服务器。新的过滤器可以根据消息优先级选择不同的路由方式：DEBUG，INFO，WARNING，ERROR，和 CRITICAL。

日志系统可以直接从 Python 配置，也可以从用户配置文件加载，以便自定义日志记录而无需更改应用程序。

11.6. 弱引用

Python 会自动进行内存管理（对大多数对象进行引用计数并使用 garbage collection 来清除循环引用）。 当某个对象的最后一个引用被移除后不久就会释放其所占用的内存。

此方式对大多数应用来说都适用，但偶尔也必须在对象持续被其他对象所使用时跟踪它们。 不幸的是，跟踪它们将创建一个会令其永久化的引用。 weakref 模块提供的工具可以不必创建引用就能跟踪对象。 当对象不再需要时，它将自动从一个弱引用表中被移除，并为弱引用对象触发一个回调。 典型应用包括对创建开销较大的对象进行缓存:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
```

```
...         return str(self.value)
...
>>> a = A(10)                   # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # does not create a reference
>>> d['primary']                # fetch the object if it is still alive
10
>>> del a                       # remove the one reference
>>> gc.collect()                # run garbage collection right away
0
>>> d['primary']                # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                # entry was automatically removed
  File "C:/python314/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7. 用于操作列表的工具

许多对于数据结构的需求可以通过内置列表类型来满足。 但是，有时也会需要具有不同效费比的替代实现。

array 模块提供了一种 array 对象，它类似于列表，但只能存储类型一致的数据且存储密度更高。下面的例子显示了一个由存储为双字节无符号整数的数字 (类型码 "H") 组成的元组，而不是常规 Python int 对象列表所采用的每个条目 16 字节:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

collections 模块提供了一种 deque 对象，它类似于列表，但从左端添加和弹出的速度较快而在中间查找的速度较慢。 此种对象适用于实现队列和广度优先树搜索:

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print("Handling", d.popleft())
Handling task1
```

```
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

在替代的列表实现以外，标准库也提供了其他工具，例如 bisect 模块具有用于操作有序列表的函数:

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

heapq 模块提供了基于常规列表来实现堆的函数。 最小值的条目总是保持在位置零。 这对于需要重复访问最小元素而不希望运行完整列表排序的应用来说非常有用:

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # 将列表重新调整为堆顺序
>>> heappush(data, -5)                 # 添加一个新条目
>>> [heappop(data) for i in range(3)]  # 获取三个最小的条目
[-5, 0, 1]
```

11.8. 十进制浮点运算

decimal 模块提供了一种 Decimal 数据类型用于十进制浮点运算。 相比内置的 float 二进制浮点实现，该类特别适用于

- 财务应用和其他需要精确十进制表示的用途，
- 控制精度，
- 控制四舍五入以满足法律或监管要求，
- 跟踪有效小数位，或
- 用户期望结果与手工完成的计算相匹配的应用程序。

例如，使用十进制浮点和二进制浮点数计算70美分手机和5%税的总费用，会产生的不同结果。如果结果四舍五入到最接近的分数差异会更大:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73
```

Decimal 表示的结果会保留尾部的零，并根据具有两个有效位的被乘数自动推出四个有效位。 Decimal 可以模拟手工运算来避免当二进制浮点数无法精确表示十进制数时会导致的问题。

精确表示特性使得 Decimal 类能够执行对于二进制浮点数来说不适用的模运算和相等性检测:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
False
```

decimal 模块提供了运算所需要的足够精度:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```