Pyhon模块、包和程序

本章学习如何写出实用的大型Python程序。

之前的独立程序

```
#!/usr/bin/env python3.6

"""
hello_world.py
"""


def main():
    print('hello, world!')


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

命令行工具

Python 作为一种脚本语言，可以非常方便地用于系统（尤其是*nix系统）命令行工具的开发。Python 自身也集成了一些标准库，专门用于处理命令行相关的问题。

标准输入输出

*nix 系统中，一切皆为文件，因此标准输入、输出可以完全可以看做是对文件的操作。标准化输入可以通过管道（pipe）或重定向（redirect）的方式传递：

```
#!/usr/bin/env python3.6

# reverse.py

import sys

for l in sys.stdin.readlines():
    sys.stdout.write(l[::-1])
```

将上面的代码保存为`reverse.py`，通过管道`|`传递：

```
chmod +x reverse.py
cat reverse.py | ./reverse.py
```

```
6.3nohtyp vne/nib/rsu/!#

yp.esrever #

sys tropmi

:)(senildaer.nidts.sys ni l rof
))l-::[l(etirw.tuodts.sys
```

通过重定向`<`传递：

```
./reverse.py < reverse.py
# 输出结果同上
```

命令行参数

一般在命令行后追加的参数可以通过`sys.argv`获取，`sys.argv`是一个列表，其中第一个元素为当前脚本的文件名：

```
#!/usr/bin/env python3.6

import sys

print(sys.argv)  #下面返回的是Jupyter运行的结果
```

运行上面的脚本：

```
chmod +x argv.py
./argv.py hello world
# ['./argv.py', 'hello', 'world']
python argv.py hello world
# ['argv.py', 'hello', 'world']
```

对于比较复杂的命令行参数，例如通过`--option`传递的选项参数，如果是对`sys.argv`逐项进行解析会很麻烦，Python 提供标准库`argparse`（旧的库为`optparse`，已经停止维护）专门解析命令行参数：

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print(args.accumulate(args.integers))
```

假设上面的代码保存为文件`prog.py`，它可以在命令行中执行并提供帮助信息。

```
$ python3 prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
  N           an integer for the accumulator

optional arguments:
  -h, --help  show this help message and exit
  --sum       sum the integers (default: find the max)
```

当通过合适的参数执行时，它将打印参数的和或最大值。

```
$ python3 prog.py 1 2 3 4
4

$ python3 prog.py 1 2 3 4 --sum
10
```

如果传入了非法参数，它将报错：

```
$ python3 prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: argument N: invalid int value: 'a'
```

执行系统命令

当 Python 能够准确地解读输入信息或参数之后，就可以通过 Python 去做任何事情了。 这里主要介绍通过 Python 调用系统命令，也就是替代 Shell 脚本完成系统管理的功能。

subprocess 模块提供简便的直接调用系统指令的`call()`方法，以及较为复杂可以让用户更加深入地与系统命令进行交互的`Popen`对象。

```
#!/usr/bin/env python3.6

# list_files.py

import subprocess as sb

res = sb.check_output("ls -lh ./*.ipynb", shell=True) # 为了安全，默认不通过系统 Shell 执行，因此需要设定 shell=True
print(res.decode())  # 默认返回值为 bytes 类型，需要进行解码操作
```

上面的代码会fork一个子进程，其中运行shell命令`ls -lh ./*.ipynb`，父进程会等待子进程完成，返回子进程向标准输出的输出结果。

如果只是简单地执行系统命令还不能满足你的需求，可以使用`subprocess.Popen`与生成的子进程进行更多交互：

```
import subprocess as sb

p = sb.Popen(['grep', 'sys'], stdin=sb.PIPE, stdout=sb.PIPE)
res, err = p.communicate(sb.check_output('cat ./argv.py', shell=True))
if not err:
    print(res.decode())
```

模块和包

模块和包是大型项目的核心，如何组织包、将大型模块分解成多个文件对于合理组织代码结构非常重要。

一个**模块**就是一个Python代码文件。把多个模块组织成文件层次，称之为**包**。

导入模块

使用`import`语句来进行模块导入，模块是不带`.py`拓展的另外一个Python文件的文件名。先来看下面的例子，这里先将代码组织成由很多分层模块构成的包。在文件系统上组织代码，并确保每一个目录都定义了一个`__init__.py`文件，这个文件可以是空的，其目的是要包含不同运行级别的包的可选的初始化代码。举个例子，如果你执行了语句`import graphics`，文件`graphics/__init__.py`将被导入，建立`graphics`命名空间的内容。像`import graphics.format.jpg`这样导入，文件`graphics/__init__.py`和文件`graphics/graphics/formats/__init__.py`将在文件`graphics/formats/jpg.py`导入之前导入。

```
graphics/
    __init__.py
    primitive/
        __init__.py
        line.py
        fill.py
        text.py
    formats/
        __init__.py
        png.py
        jpg.py
```

之后，就可以使用各种`import`语句来导入模块。

```
import graphics.primitive.line # 直接导入模块，须按照"模块名. "的用法来使用
from graphics.primitive import line # 导入模块的一部分
import graphics.formats.jpg as jpg # 使用别名导入模块
from graphics import *  # 导入模块的全部内容
```

在函数内部导入模块

```
def get_random():
    import random
    possibilities = ['a', 'b', 'c', 'd']
    return random.choice(possibilities)
```

大家都比较习惯的在函数外部导入：

```
import random
def get_random():
    possibilities = ['a', 'b', 'c', 'd']
    return random.choice(possibilities)
```

如果被导入的代码被多个地方多次使用，就应该考虑在函数外部导入；如果被导入的代码只在函数内部使用，就在函数内部导入。

模块搜索路径

Python使用存储在`sys`模块下的目录名和zip压缩文件列表作为`path`变量，这个列表可以被读取和修改。

```
import sys
for pth in sys.path:
    print(pth)
```

最开始的空白输出行是空字符串''，代表当前目录。如果空字符串是在`sys.path`的开始位置，Python会先搜索当前目录。Python会根据列表元素的位置优先导入前面目录中存在的模块，如果模块同名，后面路径中出现的模块则不会被导入。

使用相对路径导入模块

在包内，既可以使用绝对路径来导入也可以使用相对路径来导入。我们可以使用包的相对导入，使一个的模块导入同一个包的另一个模块。比如下面的例子，假设你在文件系上有`mypackage`包，组织如下：

```
mypackage/
    __init__.py
    A/
        __init__.py
        spam.py
        grok.py
    B/
        __init__.py
        bar.py
```

如果模块`mypackage.A.spam`要导入同目录下的模块`grok`，它应该包括的`import`语句如下：

```
# mypackage/A/spam.py
from . import grok
```

如果模块`mypackage.A.spam`要导入不同目录下的模块`B.bar`，它应该使用的`import`语句如下：

```
# mypackage/A/spam.py
from ..B import bar
```

两个`import`语句都没包含顶层包名，而是使用了`spam.py`的相对路径。

下面是同时使用绝对路径导入和相对路径导入的例子：

```
# mypackage/A/spam.py
from mypackage.A import grok # OK
from . import grok # OK
import grok # ImportError (not found)
```

像`mypackage.A`这样使用绝对路径名的不利之处是这将顶层包名硬编码到你的源码中。如果你想重新组织它，你的代码将更加脆弱，很难工作。举个例子，如果你改变了包名，你就必须检查所有文件来修正源码。同样，硬编码的名称会使移动代码变得困难。举个例子，也许有人想安装两个不同版本的软件包，只通过名称区分它们。如果使用相对导入，那一切都ok，然而使用绝对路径名很可能会出问题。

`import`语句中的`.`和`..`语法跟shell中的当前目录和上级目录比较类似，把它们想象成指定目录名即可。`.`意味着在当前目录中查找，而`..B`表示在`../B`目录中查找。这种语法只能用在`from xx import yy`这样的导入语句中。

```
from . import grok  # OK
import .grok  # ERROR
```

相对导入不允许跳出定义包的那个目录，即利用句点的组合形式进入一个不是Python包的目录会出现导入错误。

另外，相对导入只在特定的条件下才起作用，即，模块必须位于一个合适的包中才可以。特别的，位于脚本顶层目录的模块不能使用相对导入。

此外，如果包的某个部分是直接以脚本的形式执行的，这种情况也不能使用相对导入。

如：

```
$ python3 mypackage/A/spam.py  # relative imports fails
```

但是可以使用`-m`选项来执行上面的脚本：

```
$ python3 -m mypackage.A.spam  # relative imports work
```

Python标准库

Python具有庞大的标准库，Python的标准库和Python语言核心一起构成的Python语言。Python提供了标准库各模块的官方文档（ https://docs.python.org/3/library ）以及使用指南（ https://docs.python.org/3.6/tutorial/stdlib.html ）。另外，*Doug Hellmann* 的网站*Python 3 Module of the Week*（ https://pymotw.com/3/ ）和他的书 *The Python Standard Library by Example*（中文版《Python标准库》由机械工业出版社于2012-06-01出版，目前英文版计划出版第2版：*The Python 3 Standard Library by Example*）都是非常有帮助的指南，其针对标准库模块展示了大量的代码实例。

教材中展示的一些常用的标准库模块：

- `collections.defaultdict` 创建包含键默认值的字典，其参数是一个函数（可以是lambda函数），返回赋给缺失键的值。

```
from collections import defaultdict
food_counter = defaultdict(int)
for food in ['spam', 'spam', 'eggs', 'spam']:
    food_counter[food] += 1

for food, count in food_counter.items():
    print(food, count)
```

- `collections.Counter` 提供了计数器功能。

```
from collections import Counter
alpha_counter = Counter('aaabbbccc')
alpha_counter
```

函数`most_common()`以降序返回所有元素，可以给其指定一个数字参数，返回排名在该数字之前的元素。

```
alpha_counter.most_common(2)
```

另外，可以针对两个或多个计数器进行组合，其也支持类似集合元算的求交、并、差运算。

```
alpha_counter2 = Counter('abcddd')
alpha_counter + alpha_counter2  # Counter({'a': 4, 'b': 4, 'c': 4, 'd': 3}), 计数组合
alpha_counter - alpha_counter2  # Counter({'a': 2, 'b': 2, 'c': 2}), 计数相减
alpha_counter & alpha_counter2  # Counter({'a': 1, 'b': 1, 'c': 1}), 两者交集中取两者中较小计数
alpha_counter | alpha_counter2  # Counter({'a': 3, 'b': 3, 'c': 3, 'd': 3}), 两者并集中取两者中较大计数
```

- `collections.OrderedDict` 有序字典，记忆字典键添加的顺序，然后从一个迭代器按照相同的顺序返回。

```
from collections import OrderedDict
quotes = OrderedDict([
    ('Moe', 'A wise guy, huh?'),
    ('Larry', 'Ow!'),
    ('Curly', 'Nyuk nyuk!'),
])
for stooge in quotes:
    print(stooge)
```

```
quotes = dict([
    ('Moe', 'A wise guy, huh?'),
    ('Larry', 'Ow!'),
    ('Curly', 'Nyuk nyuk!'),
])
for stooge in quotes:
    print(stooge)
```

- `collections.deque` 双端队列，同时具有栈和队列的特征，可从序列的任何一端添加和删除项。函数 `popleft()` 去掉最左边的项并返回该项， `pop()` 去掉最右边的项并返回该项。

```
def palindrome(word):
    """检测回文"""
    from collections import deque
    dq = deque(word)
    while len(dq) > 1:
        if dq.popleft() != dq.pop():
            return False
    return True


def another_palindrome(word):
    """检测回文"""
    return word == word[::-1]
print(palindrome('racecar'))
print(another_palindrome('racecar'))
```

- `itertools` 迭代器函数，每次返回一项，并记住当前调用的状态。

```
import itertools
help(itertools)
```

```
for item in itertools.chain([1, 2], ['a', 'b']):
    print(item)
```

```
for item in itertools.cycle(['a', 'b']):
    print(item)
```

```
for item in itertools.accumulate([1, 2, 3, 4], lambda x, y: x * y):
    print(item)
```

- `pprint` 友好打印

```
quotes = OrderedDict([
    ('Moe', 'A wise guy, huh?'),
    ('Larry', 'Ow!'),
    ('Curly', 'Nyuk nyuk!'),
])
print(quotes)
```

```
from pprint import pprint
pprint(quotes)
```

获取第三方Python代码

- **Pypi** (https://pypi.python.org)
- **Github** (https://github.com)
- **ReadTheDocs** (https://readthedocs.org)
- **activestate** (http://code.activestate.com/recipes/langs/python)

鸭子类型

在程序设计中，鸭子类型（duck typing）是动态类型的一种风格。在这种风格中，一个对象有效的语义，不是由继承自特定的类或实现特定的接口，而是由"当前方法和属性的集合"决定。支持“鸭子类型”的语言的解释器/编译器会在解释或编译时推断对象的类型。在鸭子类型中，关注的不是对象的类型本身，而是它是如何使用的。例如，在不使用鸭子类型的语言中，我们可以编写一个函数，它接受一个类型为“鸭子”的对象，并调用它的“走”和“叫”方法。在使用鸭子类型的语言中，这样的一个函数可以接受一个任意类型的对象，并调用它的“走”和“叫”方法。如果这些需要被调用的方法不存在，那么将引发一个运行时错误。任何拥有这样的正确的“走”和“叫”方法的对象都可被函数接受的这种行为引出了以上表述，这种决定类型的方式因此得名。

在动态语言设计中，可以解释为无论一个对象是什么类型的，只要它具有某类型的行为（方法），则它就是这一类型的实例，而不在于它是否显示的实现或者继承。比如，如果一个对象具备迭代器所具有的所有行为特征，那它就是迭代器

了。而如何保证一个对象实现某一种类型的所有特征，则依靠**协议**。

特殊方法（魔法方法）

特殊方法是指Python类中以双下划线`__`开头和结尾的方法，比如`__init__`，根据类的定义以及传入的参数对新创建的对象进行初始化。

构造和初始化

但是当调用`x = SomeClass()`的时候，`__init__`并不是第一个被调用的方法。实际上，还有一个叫做`__new__`的方法，来构造这个实例。然后给在开始创建时候的初始化函数来传递参数。在对象生命周期的另一端，也有一个`__del__`方法（如果`__new__`和`__init__`是对象的构造器的话，那么`__del__`就是析构器），它定义的是当一个对象进行垃圾回收时候的行为。当一个对象在删除的时需要更多的清洁工作的时候此方法会很有用，比如套接字对象或者是文件对象。

```
from os.path import join

class FileObject:
    '''给文件对象进行包装从而确认在删除时关闭文件流'''

    def __init__(self, filepath='~', filename='sample.txt'):
        # 读写模式打开一个文件
        self.file = open(join(filepath, filename), 'r+')

    def __del__(self):
        self.file.close()
        del self.file
```

用于比较的魔术方法

```
__lt__(self, other)    self < other
__le__(self, other)    self <= other
__eq__(self, other)    self == other
__ne__(self, other)    self != other
__gt__(self, other)    self > other
__ge__(self, other)    self >= other
```

举一个例子，创建一个类来表示一个词语。我们也许会想要比较单词的字典序(通过字母表)，通过默认的字符串比较的方法就可以实现，但是我们也想要通过一些其他的标准来实现，比如单词长度或者音节数量。在这个例子中，我们来比较长度实现。以下是实现代码:

```
class Word(str):
    """
    存储单词的类，定义比较单词的几种方法
    """

    def __new__(cls, word):
        # 注意我们必须要用到__new__方法，因为str是不可变类型
        # 所以我们必须在创建的时候将它初始化
        if ' ' in word:
            print("Value contains spaces. Truncating to first space.")
            word = word[:word.index(' ')] #单词是第一个空格之前的所有字符
        return str.__new__(cls, word)

    def __gt__(self, other):
        return len(self) > len(other)
    def __lt__(self, other):
        return len(self) < len(other)
    def __ge__(self, other):
        return len(self) >= len(other)
    def __le__(self, other):
        return len(self) <= len(other)

foo = Word('foo')
bar = Word('bar')
foo > bar  # False
foo < bar  # False
foo >= bar  # True
foo <= bar  # True
```

普通算数操作符

- `__add__(self, other)` 实现加法`+`
- `__sub__(self, other)` 实现减法`-`
- `__mul__(self, other)` 实现乘法`*`
- `__floordiv__(self, other)` 实现`//`符号实现的整数除法
- `__truediv__(self, other)` 实现真除法`/`
- `__mod__(self, other)` 实现取模算法`%`
- `__divmod__(self, other)` 实现内置`divmod()`函数
- `__pow__(self, other)` 实现使用`**`的指数运算
- `__lshift__(self, other)` 实现使用`<<`的按位左移动

- `__rshift__(self, other)` 实现使用`>>`的按位左移动
- `__and__(self, other)` 实现使用`&`的按位与
- `__or__(self, other)` 实现使用`|`的按位或
- `__xor__(self, other)` 实现使用`^`的按位异或

其他种类的魔法方法

- `__str__(self)` 等价于`str(self)`，定义如何打印对象信息，`print()`、`str()`以及字符串格式化相关方法都会用到`__str__()`。
- `__repr__(self)` 等价于`repr(self)`，交互式解释器适用此方法输出变量。
- `__len__(self)` 等价于`len(self)`。

```
class Word():
    def __init__(self, text):
        self.text = text
    def __eq__(self, word2):
        return self.text.lower() == word2.text.lower()
    def __str__(self):
        return self.text
    def __repr__(self):
        return 'Word("' + self.text + '")'

first = Word('ha')
print(first)
first
```

更多源于魔法方法的内容可查看Python在线文档（https://docs.python.org/3/reference/datamodel.html#special-method-names）。