



廖雪峰 2018年官方最新Python3教程(二)

共三册



扫一扫获取第一册和第三册

- Python教程
 - 一、错误、调试和测试
 - 1、错误处理
 - 2、调试
 - 3、单元测试
 - 4、文档测试
 - 二、IO编程
 - 1、文件读写
 - 2、StringIO和BytesIO
 - 3、操作文件和目录
 - 4、序列化
 - 三、进程和线程
 - 1、多进程
 - 2、多线程
 - 3、ThreadLocal
 - 4、进程 vs. 线程
 - 5、分布式进程
 - 四、正则表达式
 - 五、内建模块
 - 1、datetime
 - 2、collections
 - 3、base64
 - 4、struct
 - 5、hashlib
 - 6、hmac
 - 7、itertools
 - 8、contextlib
 - 9、urllib
 - 10、XML
 - 11、HTMLParser
 - 六、常用第三方模块
 - 1、Pillow
 - 2、requests
 - 3、chardet
 - 4、psutil
 - 七、virtualenv
 - 八、图形界面

- 九、网络编程
 - 1、TCP/IP简介
 - 2、TCP编程
 - 3、UDP编程
- 十、电子邮件
 - 1、SMTP发送邮件
 - 2、POP3收取邮件
- 十一、访问数据库
 - 1、使用SQLite
 - 2、使用MySQL
 - 3、使用SQLAlchemy

Python教程

这是小白的Python新手教程，具有如下特点：

中文，免费，零起点，完整示例，基于最新的Python 3版本。

Python是一种计算机程序设计语言。你可能已经听说过很多种流行的编程语言，比如非常难学的C语言，非常流行的Java语言，适合初学者的Basic语言，适合网页编程的JavaScript语言等等。

那Python是一种什么语言？

首先，我们普及一下编程语言的基础知识。用任何编程语言来开发程序，都是为了让计算机干活，比如下载一个MP3，编写一个文档等等，而计算机干活的CPU只认识机器指令，所以，尽管不同的编程语言差异极大，最后都得“翻译”成CPU可以执行的机器指令。而不同的编程语言，干同一个活，编写的代码量，差距也很大。

比如，完成同一个任务，C语言要写1000行代码，Java只需要写100行，而Python可能只要20行。

所以Python是一种相当高级的语言。

你也许会问，代码少还不好？代码少的代价是运行速度慢，C程序运行1秒钟，Java程序可能需要2秒，而Python程序可能就需要10秒。

那是不是越低级的程序越难学，越高级的程序越简单？表面上来说，是的，但是，在非常高的抽象计算中，高级的Python程序设计也是非常难学的，所以，高级程序语言不等于简单。

但是，对于初学者和完成普通任务，Python语言是非常简单易用的。连Google都在大规模使用Python，你就不用担心学了会没用。

用Python可以做什么？可以做日常任务，比如自动备份你的MP3；可以做网站，很多著名的网站包括YouTube就是Python写的；可以做网络游戏的后台，很多在线游戏的后台都是Python开发的。总之就是能干很多很多事啦。

Python当然也有不能干的事情，比如写操作系统，这个只能用C语言写；写手机应用，只能用Swift/Objective-C（针对iPhone）和Java（针对Android）；写3D游戏，最好用C或C++。

如果你是小白用户，满足以下条件：

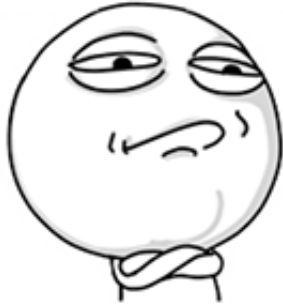
- 会使用电脑，但从来没写过程序；

- 还记得初中数学学的方程式和一点点代数知识；
- 想从编程小白变成专业的软件架构师；
- 每天能抽出半个小时学习。

不要再犹豫了，这个教程就是为你准备的！

准备好了吗？

CHALLENGE ACCEPTED !



一、错误、调试和测试

在程序运行过程中，总会遇到各种各样的错误。

有的错误是程序编写有问题造成的，比如本来应该输出整数结果输出了字符串，这种错误我们通常称之为bug，bug是必须修复的。

有的错误是用户输入造成的，比如让用户输入email地址，结果得到一个空字符串，这种错误可以通过检查用户输入来做相应的处理。

还有一类错误是完全无法在程序运行过程中预测的，比如写入文件的时候，磁盘满了，写不进去了，或者从网络抓取数据，网络突然断掉了。这类错误也称为异常，在程序中通常是必须处理的，否则，程序会因为各种问题终止并退出。

Python内置了一套异常处理机制，来帮助我们进行错误处理。

此外，我们也需要跟踪程序的执行，查看变量的值是否正确，这个过程称为调试。Python的pdb可以让我们以单步方式执行代码。

最后，编写测试也很重要。有了良好的测试，就可以在程序修改后反复运行，确保程序输出符合我们编写的测试。

1、错误处理

在程序运行的过程中，如果发生了错误，可以事先约定返回一个错误代码，这样，就可以知道是否有错，以及出错的原因。在操作系统提供的调用中，返回错误码非常常见。比如打开文件的函数 `open()`，成功时返回文件描述符（就是一个整数），出错时返回 `-1`。

用错误码来表示是否出错十分不便，因为函数本身应该返回的正常结果和错误码混在一起，造成调用者必须用大量的代码来判断是否出错：

```
def foo():
    r = some_function()
    if r==(-1):
        return (-1)
    # do something
    return r

def bar():
    r = foo()
```

```
if r==(-1):
    print('Error')
else:
    pass
```

一旦出错，还要一级一级上报，直到某个函数可以处理该错误（比如，给用户输出一个错误信息）。

所以高级语言通常都内置了一套 `try...except...finally...` 的错误处理机制，Python也不例外。

try

让我们用一个例子来看看 `try` 的机制：

```
try:
    print('try...')
    r = 10 / 0
    print('result:', r)
except ZeroDivisionError as e:
    print('except:', e)
finally:
    print('finally...')
print('END')
```

当我们认为某些代码可能会出错时，就可以用 `try` 来运行这段代码，如果执行出错，则后续代码不会继续执行，而是直接跳转至错误处理代码，即 `except` 语句块，执行完 `except` 后，如果有 `finally` 语句块，则执行 `finally` 语句块，至此，执行完毕。

上面的代码在计算 `10 / 0` 时会产生一个除法运算错误：

```
try...
except: division by zero
finally...
END
```

从输出可以看到，当错误发生时，后续语句 `print('result:', r)` 不会被执行，`except` 由于捕获到 `ZeroDivisionError`，因此被执行。最后，`finally` 语句被执行。然后，程序继续按照流程往下走。

如果把除数 `0` 改成 `2`，则执行结果如下：

```
try...
result: 5
finally...
END
```

由于没有错误发生，所以 `except` 语句块不会被执行，但是 `finally` 如果有，则一定会被执行（可以没有 `finally` 语句）。

你还可以猜测，错误应该有很多种类，如果发生了不同类型的错误，应该由不同的 `except` 语句块处理。没错，可以有多个 `except` 来捕获不同类型的错误：

```
try:
    print('try...')
    r = 10 / int('a')
    print('result:', r)
except ValueError as e:
    print('ValueError:', e)
except ZeroDivisionError as e:
    print('ZeroDivisionError:', e)
finally:
    print('finally...')
print('END')
```

`int()` 函数可能会抛出 `ValueError`，所以我们用一个 `except` 捕获 `ValueError`，用另一个 `except` 捕获 `ZeroDivisionError`。

此外，如果没有错误发生，可以在 `except` 语句块后面加一个 `else`，当没有错误发生时，会自动执行 `else` 语句：

```
try:
    print('try...')
    r = 10 / int('2')
    print('result:', r)
except ValueError as e:
    print('ValueError:', e)
except ZeroDivisionError as e:
    print('ZeroDivisionError:', e)
else:
    print('no error!')
finally:
```



```
print('finally...')
print('END')
```

Python的错误其实也是class，所有的错误类型都继承自 `BaseException`，所以在使用 `except` 时需要注意的是，它不但捕获该类型的错误，还把其子类也“一网打尽”。比如：

```
try:
    foo()
except ValueError as e:
    print('ValueError')
except UnicodeError as e:
    print('UnicodeError')
```

第二个 `except` 永远也捕获不到 `UnicodeError`，因为 `UnicodeError` 是 `ValueError` 的子类，如果有，也被第一个 `except` 给捕获了。

Python所有的错误都是从 `BaseException` 类派生的，常见的错误类型和继承关系看这里：

<https://docs.python.org/3/library/exceptions.html#exception-hierarchy>

使用 `try...except` 捕获错误还有一个巨大的好处，就是可以跨越多层调用，比如函数 `main()` 调用 `foo()`，`foo()` 调用 `bar()`，结果 `bar()` 出错了，这时，只要 `main()` 捕获到了，就可以处理：

```
def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    try:
        bar('0')
    except Exception as e:
        print('Error:', e)
    finally:
        print('finally...')
```

也就是说，不需要在每个可能出错的地方去捕获错误，只要在合适的层次去捕获错误就可以了。这样一来，就大大减少了写 `try...except...finally` 的麻烦。

调用栈

如果错误没有被捕获，它就会一直往上抛，最后被Python解释器捕获，打印一个错误信息，然后程序退出。来看看 `err.py`：

```
# err.py:
def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    bar('0')

main()
```

执行，结果如下：

```
$ python3 err.py
Traceback (most recent call last):
  File "err.py", line 11, in <module>
    main()
  File "err.py", line 9, in main
    bar('0')
  File "err.py", line 6, in bar
    return foo(s) * 2
  File "err.py", line 3, in foo
    return 10 / int(s)
ZeroDivisionError: division by zero
```

出错并不可怕，可怕的是不知道哪里出错了。解读错误信息是定位错误的关键。我们从上往下可以看到整个错误的调用函数链：

错误信息第1行：

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

告诉我们这是错误的跟踪信息。

第2~3行:

```
File "err.py", line 11, in <module>
    main()
```

调用 `main()` 出错了，在代码文件 `err.py` 的第11行代码，但原因是第9行:

```
File "err.py", line 9, in main
    bar('0')
```

调用 `bar('0')` 出错了，在代码文件 `err.py` 的第9行代码，但原因是第6行:

```
File "err.py", line 6, in bar
    return foo(s) * 2
```

原因是 `return foo(s) * 2` 这个语句出错了，但这还不是最终原因，继续往下看:

```
File "err.py", line 3, in foo
    return 10 / int(s)
```

原因是 `return 10 / int(s)` 这个语句出错了，这是错误产生的源头，因为下面打印了:

```
ZeroDivisionError: integer division or modulo by zero
```

根据错误类型 `ZeroDivisionError`，我们判断，`int(s)` 本身并没有出错，但是 `int(s)` 返回 `0`，在计算 `10 / 0` 时出错，至此，找到错误源头。

出错的时候，一定要分析错误的调用栈信息，才能定位错误的位置。



记录错误

如果不捕获错误，自然可以让Python解释器来打印出错误堆栈，但程序也被结束了。既然我们能捕获错误，就可以把错误堆栈打印出来，然后分析错误原因，同时，让程序继续执行下去。

Python内置的 `logging` 模块可以非常容易地记录错误信息：

```
# err_logging.py

import logging

def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    try:
        bar('0')
    except Exception as e:
        logging.exception(e)

main()
print('END')
```

同样是出错，但程序打印完错误信息后会继续执行，并正常退出：

```
$ python3 err_logging.py
ERROR:root:division by zero
Traceback (most recent call last):
```

```
File "err_logging.py", line 13, in main
    bar('0')
File "err_logging.py", line 9, in bar
    return foo(s) * 2
File "err_logging.py", line 6, in foo
    return 10 / int(s)
ZeroDivisionError: division by zero
END
```

通过配置，`logging` 还可以把错误记录到日志文件里，方便事后排查。

抛出错误

因为错误是class，捕获一个错误就是捕获到该class的一个实例。因此，错误并不是凭空产生的，而是有意创建并抛出的。Python的内置函数会抛出很多类型的错误，我们自己编写的函数也可以抛出错误。

如果要抛出错误，首先根据需要，可以定义一个错误的class，选择好继承关系，然后，用 `raise` 语句抛出一个错误的实例：

```
# err_raise.py
class FooError(ValueError):
    pass

def foo(s):
    n = int(s)
    if n==0:
        raise FooError('invalid value: %s' % s)
    return 10 / n

foo('0')
```

执行，可以最后跟踪到我们自己定义的错误：

```
$ python3 err_raise.py
Traceback (most recent call last):
  File "err_throw.py", line 11, in <module>
    foo('0')
  File "err_throw.py", line 8, in foo
    raise FooError('invalid value: %s' % s)
__main__.FooError: invalid value: 0
```

只有在必要的时候才定义我们自己的错误类型。如果可以选择Python已有的内置的错误类型（比如 `ValueError`，`TypeError`），尽量使用Python内置的错误类型。

最后，我们来看另一种错误处理的方式：

```
# err_reraise.py

def foo(s):
    n = int(s)
    if n==0:
        raise ValueError('invalid value: %s' % s)
    return 10 / n

def bar():
    try:
        foo('0')
    except ValueError as e:
        print('ValueError!')
        raise

bar()
```

在 `bar()` 函数中，我们明明已经捕获了错误，但是，打印一个 `ValueError!` 后，又把错误通过 `raise` 语句抛出去了，这不有病么？

其实这种错误处理方式不但没病，而且相当常见。捕获错误目的只是记录一下，便于后续追踪。但是，由于当前函数不知道应该怎么处理该错误，所以，最恰当的方式是继续往上抛，让顶层调用者去处理。好比一个员工处理不了一个问题时，就把问题抛给他的老板，如果他的老板也处理不了，就一直往上抛，最终会抛给CEO去处理。

`raise` 语句如果不带参数，就会把当前错误原样抛出。此外，在 `except` 中 `raise` 一个Error，还可以把一种类型的错误转化成另一种类型：

```
try:
    10 / 0
except ZeroDivisionError:
    raise ValueError('input error!')
```

只要是合理的转换逻辑就可以，但是，决不应该把一个 `IOError` 转换成毫不相干的 `ValueError`。

小结

Python内置的 `try...except...finally` 用来处理错误十分方便。出错时，会分析错误信息并定位错误发生的代码位置才是最关键的。

程序也可以主动抛出错误，让调用者来处理相应的错误。但是，应该在文档中写清楚可能会抛出哪些错误，以及错误产生的原因。

**** 参考源码****

[do_try.py](#)

[err.py](#)

[err_logging.py](#)

[err_raise.py](#)

[err_reraise.py](#)

2、调试

程序能一次写完并正常运行的概率很小，基本不超过1%。总会有各种各样的bug需要修正。有的bug很简单，看看错误信息就知道，有的bug很复杂，我们需要知道出错时，哪些变量的值是正确的，哪些变量的值是错误的，因此，需要一整套调试程序的手段来修复bug。

第一种方法简单直接粗暴有效，就是用 `print()` 把可能有问题的变量打印出来看看：

```
def foo(s):
    n = int(s)
    print('>>> n = %d' % n)
    return 10 / n

def main():
    foo('0')

main()
```

执行后在输出中查找打印的变量值：

```
$ python err.py
>>> n = 0
Traceback (most recent call last):
```

```
...
ZeroDivisionError: integer division or modulo by zero
```

用 `print()` 最大的坏处是将来还得删掉它，想想程序里到处都是 `print()`，运行结果也会包含很多垃圾信息。所以，我们又有第二种方法。

断言

凡是用 `print()` 来辅助查看的地方，都可以用断言（`assert`）来替代：

```
def foo(s):
    n = int(s)
    assert n != 0, 'n is zero!'
    return 10 / n

def main():
    foo('0')
```

`assert` 的意思是，表达式 `n != 0` 应该是 `True`，否则，根据程序运行的逻辑，后面的代码肯定会出错。

如果断言失败，`assert` 语句本身就会抛出 `AssertionError`：

```
$ python err.py
Traceback (most recent call last):
...
AssertionError: n is zero!
```

程序中如果到处充斥着 `assert`，和 `print()` 相比也好不到哪去。不过，启动Python解释器时可以用 `-O` 参数来关闭 `assert`：

```
$ python -O err.py
Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

关闭后，你可以把所有的 `assert` 语句当成 `pass` 来看。

logging

把 `print()` 替换为 `logging` 是第3种方式，和 `assert` 比，`logging` 不会抛出错误，而且可以输出到文件：

```
import logging

s = '0'
n = int(s)
logging.info('n = %d' % n)
print(10 / n)
```

`logging.info()` 就可以输出一段文本。运行，发现除了 `ZeroDivisionError`，没有任何信息。怎么回事？

别急，在 `import logging` 之后添加一行配置再试试：

```
import logging
logging.basicConfig(level=logging.INFO)
```

看到输出了：

```
$ python err.py
INFO:root:n = 0
Traceback (most recent call last):
  File "err.py", line 8, in <module>
    print(10 / n)
ZeroDivisionError: division by zero
```

这就是 `logging` 的好处，它允许你指定记录信息的级别，有 `debug`，`info`，`warning`，`error` 等几个级别，当我们指定 `level=INFO` 时，`logging.debug` 就不起作用了。同理，指定 `level=WARNING` 后，`debug` 和 `info` 就不起作用了。这样一来，你可以放心地输出不同级别的信息，也不用删除，最后统一控制输出哪个级别的信息。

`logging` 的另一个好处是通过简单的配置，一条语句可以同时输出到不同的地方，比如 `console` 和文件。

pdb

第4种方式是启动Python的调试器pdb，让程序以单步方式运行，可以随时查看运行状态。我们先准备好程序：

```
# err.py
s = '0'
n = int(s)
print(10 / n)
```

然后启动：

```
$ python -m pdb err.py
> /Users/michael/Github/learn-python3/samples/debug/err.py(2)<module>()
-> s = '0'
```

以参数 `-m pdb` 启动后，pdb定位到下一步要执行的代码 `-> s = '0'`。输入命令 `l` 来查看代码：

```
(Pdb) l
1      # err.py
2  -> s = '0'
3      n = int(s)
4      print(10 / n)
```

输入命令 `n` 可以单步执行代码：

```
(Pdb) n
> /Users/michael/Github/learn-python3/samples/debug/err.py(3)<module>()
-> n = int(s)
(Pdb) n
> /Users/michael/Github/learn-python3/samples/debug/err.py(4)<module>()
-> print(10 / n)
```

任何时候都可以输入命令 `p 变量名` 来查看变量：

```
(Pdb) p s
'0'
(Pdb) p n
0
```

输入命令 `q` 结束调试，退出程序：

```
(Pdb) q
```

这种通过pdb在命令行调试的方法理论上是万能的，但实在是太麻烦了，如果有一千行代码，要运行到第999行得敲多少命令啊。还好，我们还有另一种调试方法。

`pdb.set_trace()`

这个方法也是用pdb，但是不需要单步执行，我们只需要 `import pdb`，然后，在可能出错的地方放一个 `pdb.set_trace()`，就可以设置一个断点：

```
# err.py
import pdb

s = '0'
n = int(s)
pdb.set_trace() # 运行到这里会自动暂停
print(10 / n)
```

运行代码，程序会自动在 `pdb.set_trace()` 暂停并进入pdb调试环境，可以用命令 `p` 查看变量，或者用命令 `c` 继续运行：

```
$ python err.py
> /Users/michael/Github/learn-python3/samples/debug/err.py(7)<module>()
-> print(10 / n)
(Pdb) p n
0
(Pdb) c
Traceback (most recent call last):
  File "err.py", line 7, in <module>
    print(10 / n)
ZeroDivisionError: division by zero
```

这个方式比直接启动pdb单步调试效率要高很多，但也高不到哪去。

IDE

如果要比较爽地设置断点、单步执行，就需要一个支持调试功能的IDE。目前比较好的Python IDE有：

Visual Studio Code: <https://code.visualstudio.com/>，需要安装Python插件。

PyCharm: <http://www.jetbrains.com/pycharm/>

另外，Eclipse加上pydev插件也可以调试Python程序。

小结

写程序最痛苦的事情莫过于调试，程序往往会以你意想不到的流程来运行，你期待执行的语句其实根本没有执行，这时候，就需要调试了。

虽然用IDE调试起来比较方便，但是最后你会发现，logging才是终极武器。

参考源码

[do_assert.py](#)

[do_logging.py](#)

[do_pdb.py](#)

3、单元测试

如果你听说过“测试驱动开发”（TDD：Test-Driven Development），单元测试就不陌生。

单元测试是用来对一个模块、一个函数或者一个类来进行正确性检验的测试工作。

比如对函数 `abs()`，我们可以编写出以下几个测试用例：

1. 输入正数，比如 `1`、`1.2`、`0.99`，期待返回值与输入相同；
2. 输入负数，比如 `-1`、`-1.2`、`-0.99`，期待返回值与输入相反；
3. 输入 `0`，期待返回 `0`；
4. 输入非数值类型，比如 `None`、`[]`、`{}`，期待抛出 `TypeError`。

把上面的测试用例放到一个测试模块里，就是一个完整的单元测试。

如果单元测试通过，说明我们测试的这个函数能够正常工作。如果单元测试不通过，要么函数有bug，要么测试条件输入不正确，总之，需要修复使单元测试能够通过。

单元测试通过后有什么意义呢？如果我们对 `abs()` 函数代码做了修改，只需要再跑一遍单元测

试，如果通过，说明我们的修改不会对 `abs()` 函数原有的行为造成影响，如果测试不通过，说明我们的修改与原有行为不一致，要么修改代码，要么修改测试。

这种以测试为驱动的开发模式最大的好处就是确保一个程序模块的行为符合我们设计的测试用例。在将来修改的时候，可以极大程度地保证该模块行为仍然是正确的。

我们来编写一个 `Dict` 类，这个类的行为和 `dict` 一致，但是可以通过属性来访问，用起来就像下面这样：

```
>>> d = Dict(a=1, b=2)
>>> d['a']
1
>>> d.a
1
```

`mydict.py` 代码如下：

```
class Dict(dict):

    def __init__(self, **kw):
        super().__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Dict' object has no attribute '%s'" % key)

    def __setattr__(self, key, value):
        self[key] = value
```

为了编写单元测试，我们需要引入Python自带的 `unittest` 模块，编写 `mydict_test.py` 如下：

```
import unittest

from mydict import Dict

class TestDict(unittest.TestCase):

    def test_init(self):
        d = Dict(a=1, b='test')
        self.assertEqual(d.a, 1)
```

```

self.assertEqual(d.b, 'test')
self.assertTrue(isinstance(d, dict))

def test_key(self):
    d = Dict()
    d['key'] = 'value'
    self.assertEqual(d.key, 'value')

def test_attr(self):
    d = Dict()
    d.key = 'value'
    self.assertTrue('key' in d)
    self.assertEqual(d['key'], 'value')

def test_keyerror(self):
    d = Dict()
    with self.assertRaises(KeyError):
        value = d['empty']

def test_attrerror(self):
    d = Dict()
    with self.assertRaises(AttributeError):
        value = d.empty

```

编写单元测试时，我们需要编写一个测试类，从 `unittest.TestCase` 继承。

以 `test` 开头的方法就是测试方法，不以 `test` 开头的方法不被认为是测试方法，测试的时候不会被执行。

对每一类测试都需要编写一个 `test_xxx()` 方法。由于 `unittest.TestCase` 提供了很多内置的条件判断，我们只需要调用这些方法就可以断言输出是否是我们所期望的。最常用的断言就是 `assertEqual()`：

```
self.assertEqual(abs(-1), 1) # 断言函数返回的结果与1相等
```

另一种重要的断言就是期待抛出指定类型的Error，比如通过 `d['empty']` 访问不存在的key时，断言会抛出 `KeyError`：

```

with self.assertRaises(KeyError):
    value = d['empty']

```

而通过 `d.empty` 访问不存在的key时，我们期待抛出 `AttributeError`：

```
with self.assertRaises(AttributeError):
    value = d.empty
```

运行单元测试

一旦编写好单元测试，我们就可以运行单元测试。最简单的运行方式是在 `mydict_test.py` 的最后加上两行代码：

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

这样就可以把 `mydict_test.py` 当做正常的python脚本运行：

```
$ python mydict_test.py
```

另一种方法是在命令行通过参数 `-m unittest` 直接运行单元测试：

```
$ python -m unittest mydict_test
.....
-----
Ran 5 tests in 0.000s

OK
```

这是推荐的做法，因为这样可以一次批量运行很多单元测试，并且，有很多工具可以自动来运行这些单元测试。

setUp与tearDown

可以在单元测试中编写两个特殊的 `setUp()` 和 `tearDown()` 方法。这两个方法会分别在每调用一个测试方法的前后分别被执行。

`setUp()` 和 `tearDown()` 方法有什么用呢？设想你的测试需要启动一个数据库，这时，就可以在 `setUp()` 方法中连接数据库，在 `tearDown()` 方法中关闭数据库，这样，不必在每个测试方法中重复相同的代码：

```
class TestDict(unittest.TestCase):

    def setUp(self):
        print('setUp...')

    def tearDown(self):
        print('tearDown...')
```

可以再次运行测试看看每个测试方法调用前后是否会打印出 `setUp...` 和 `tearDown...`。

小结

单元测试可以有效地测试某个程序模块的行为，是未来重构代码的信心保证。

单元测试的测试用例要覆盖常用的输入组合、边界条件和异常。

单元测试代码要非常简单，如果测试代码太复杂，那么测试代码本身就可能bug。

单元测试通过了并不意味着程序就没有bug了，但是不通过程序肯定有bug。

参考源码

[mydict.py](#)

[mydict_test.py](#)

4、文档测试

如果你经常阅读Python的官方文档，可以看到很多文档都有示例代码。比如[re模块](#)就带了很多示例代码：

```
>>> import re
>>> m = re.search('(<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

可以把这些示例代码在Python的交互式环境下输入并执行，结果与文档中的示例代码显示的一致。

这些代码与其他说明可以写在注释中，然后，由一些工具来自动生成文档。既然这些代码本身就可以粘贴出来直接运行，那么，可不可以自动执行写在注释中的这些代码呢？

答案是肯定的。

当我们编写注释时，如果写上这样的注释：

```
def abs(n):
    '''
    Function to get absolute value of number.

    Example:

    >>> abs(1)
    1
    >>> abs(-1)
    1
    >>> abs(0)
    0
    '''
    return n if n >= 0 else (-n)
```

无疑更明确地告诉函数的调用者该函数的期望输入和输出。

并且，Python内置的“文档测试”（doctest）模块可以直接提取注释中的代码并执行测试。

doctest严格按照Python交互式命令行的输入和输出来判断测试结果是否正确。只有测试异常的时候，可以用 ... 表示中间一大段烦人的输出。

让我们用doctest来测试上次编写的 Dict 类：

```
# mydict2.py
class Dict(dict):
    '''
    Simple dict but also support access as x.y style.

    >>> d1 = Dict()
    >>> d1['x'] = 100
    >>> d1.x
    100
    >>> d1.y = 200
    >>> d1['y']
    200
    >>> d2 = Dict(a=1, b=2, c='3')
    >>> d2.c
    '3'
    >>> d2['empty']
    Traceback (most recent call last):
```

```

...
KeyError: 'empty'
>>> d2.empty
Traceback (most recent call last):
...
AttributeError: 'Dict' object has no attribute 'empty'
...

def __init__(self, **kw):
    super(Dict, self).__init__(**kw)

def __getattr__(self, key):
    try:
        return self[key]
    except KeyError:
        raise AttributeError(r"'Dict' object has no attribute '%s'" % key)

def __setattr__(self, key, value):
    self[key] = value

if __name__ == '__main__':
    import doctest
    doctest.testmod()

```

运行 `python mydict2.py` :

```
$ python mydict2.py
```

什么输出也没有。这说明我们编写的doctest运行都是正确的。如果程序有问题，比如把 `__getattr__()` 方法注释掉，再运行就会报错：

```

$ python mydict2.py
*****
File "/Users/michael/Github/learn-python3/samples/debug/mydict2.py", line 10, in __
main__.Dict
Failed example:
    d1.x
Exception raised:
Traceback (most recent call last):
...
AttributeError: 'Dict' object has no attribute 'x'
*****
File "/Users/michael/Github/learn-python3/samples/debug/mydict2.py", line 16, in __
main__.Dict

```

```
Failed example:
  d2.c
Exception raised:
  Traceback (most recent call last):
    ...
  AttributeError: 'Dict' object has no attribute 'c'
*****
1 items had failures:
  2 of   9 in __main__.Dict
***Test Failed*** 2 failures.
```

注意到最后3行代码。当模块正常导入时，doctest不会被执行。只有在命令行直接运行时，才执行doctest。所以，不必担心doctest会在非测试环境下执行。

小结

doctest非常有用，不但可以用来测试，还可以直接作为示例代码。通过某些文档生成工具，就可以自动把包含doctest的注释提取出来。用户看文档的时候，同时也看到了doctest。

参考源码

[mydict2.py](#)

二、IO编程

IO在计算机中指Input/Output，也就是输入和输出。由于程序和运行时数据是在内存中驻留，由CPU这个超快的计算核心来执行，涉及到数据交换的地方，通常是磁盘、网络等，就需要IO接口。

比如你打开浏览器，访问新浪首页，浏览器这个程序就需要通过网络IO获取新浪的网页。浏览器首先会发送数据给新浪服务器，告诉它我想要首页的HTML，这个动作是往外发数据，叫Output，随后新浪服务器把网页发过来，这个动作是从外面接收数据，叫Input。所以，通常，程序完成IO操作会有Input和Output两个数据流。当然也有只用一个的情况，比如，从磁盘读取文件到内存，就只有Input操作，反过来，把数据写到磁盘文件里，就只是一个Output操作。

IO编程中，Stream（流）是一个很重要的概念，可以把流想象成一个水管，数据就是水管里的水，但是只能单向流动。Input Stream就是数据从外面（磁盘、网络）流进内存，Output Stream就是数据从内存流到外面去。对于浏览网页来说，浏览器和新浪服务器之间至少需要建立两根水管，才可以既能发数据，又能收数据。

由于CPU和内存的速度远远高于外设的速度，所以，在IO编程中，就存在速度严重不匹配的问题。举个例子来说，比如要把100M的数据写入磁盘，CPU输出100M的数据只需要0.01秒，可是

磁盘要接收这100M数据可能需要10秒，怎么办呢？有两种办法：

第一种是CPU等着，也就是程序暂停执行后续代码，等100M的数据在10秒后写入磁盘，再接着往下执行，这种模式称为同步IO；

另一种方法是CPU不等待，只是告诉磁盘，“您老慢慢写，不着急，我接着干别的事去了”，于是，后续代码可以立刻接着执行，这种模式称为异步IO。

同步和异步的区别就在于是否等待IO执行的结果。好比你去麦当劳点餐，你说“来个汉堡”，服务员告诉你，对不起，汉堡要现做，需要等5分钟，于是你站在收银台前面等了5分钟，拿到汉堡再去逛商场，这是同步IO。

你说“来个汉堡”，服务员告诉你，汉堡需要等5分钟，你可以先去逛商场，等做好了，我们再通知你，这样你可以立刻去干别的事情（逛商场），这是异步IO。

很明显，使用异步IO来编写程序性能会远远高于同步IO，但是异步IO的缺点是编程模型复杂。想想看，你得知道什么时候通知你“汉堡做好了”，而通知你的方法也各不相同。如果是服务员跑过来找到你，这是回调模式，如果服务员发短信通知你，你就得不停地检查手机，这是轮询模式。总之，异步IO的复杂度远远高于同步IO。

操作IO的能力都是由操作系统提供的，每一种编程语言都会把操作系统提供的低级C接口封装起来方便使用，Python也不例外。我们后面会详细讨论Python的IO编程接口。

注意，本章的IO编程都是同步模式，异步IO由于复杂度太高，后续涉及到服务器端程序开发时我们再讨论。

1、文件读写

读写文件是最常见的IO操作。Python内置了读写文件的函数，用法和C是兼容的。

读写文件前，我们先必须了解一下，在磁盘上读写文件的功能都是由操作系统提供的，现代操作系统不允许普通的程序直接操作磁盘，所以，读写文件就是请求操作系统打开一个文件对象（通常称为文件描述符），然后，通过操作系统提供的接口从这个文件对象中读取数据（读文件），或者把数据写入这个文件对象（写文件）。

读文件

要以读文件的模式打开一个文件对象，使用Python内置的 `open()` 函数，传入文件名和标示符：

```
>>> f = open('/Users/michael/test.txt', 'r')
```

标示符'r'表示读，这样，我们就成功地打开了一个文件。

如果文件不存在，`open()` 函数就会抛出一个 `IOError` 的错误，并且给出错误码和详细的信息告诉你文件不存在：

```
>>> f=open('/Users/michael/notfound.txt', 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: '/Users/michael/notfound.tx
t'
```

如果文件打开成功，接下来，调用 `read()` 方法可以一次读取文件的全部内容，Python把内容读到内存，用一个 `str` 对象表示：

```
>>> f.read()
'Hello, world!'
```

最后一步是调用 `close()` 方法关闭文件。文件使用完毕后必须关闭，因为文件对象会占用操作系统的资源，并且操作系统同一时间能打开的文件数量也是有限的：

```
>>> f.close()
```

由于文件读写时都有可能产生 `IOError`，一旦出错，后面的 `f.close()` 就不会调用。所以，为了保证无论是否出错都能正确地关闭文件，我们可以使用 `try ... finally` 来实现：

```
try:
    f = open('/path/to/file', 'r')
    print(f.read())
finally:
    if f:
        f.close()
```

但是每次都这么写实在太繁琐，所以，Python引入了 `with` 语句来自动帮我们调用 `close()` 方法：

```
with open('/path/to/file', 'r') as f:
```

```
print(f.read())
```

这和前面的 `try ... finally` 是一样的，但是代码更佳简洁，并且不必调用 `f.close()` 方法。

调用 `read()` 会一次性读取文件的全部内容，如果文件有10G，内存就爆了，所以，要保险起见，可以反复调用 `read(size)` 方法，每次最多读取size个字节的内容。另外，调用 `readline()` 可以每次读取一行内容，调用 `readlines()` 一次读取所有内容并按行返回 `list`。因此，要根据需要决定怎么调用。

如果文件很小，`read()` 一次性读取最方便；如果不能确定文件大小，反复调用 `read(size)` 比较保险；如果是配置文件，调用 `readlines()` 最方便：

```
for line in f.readlines():
    print(line.strip()) # 把末尾的'\n'删掉
```

file-like Object

像 `open()` 函数返回的这种有个 `read()` 方法的对象，在Python中统称为file-like Object。除了file外，还可以是内存的字节流，网络流，自定义流等等。file-like Object不要求从特定类继承，只要写个 `read()` 方法就行。

`StringIO` 就是在内存中创建的file-like Object，常用作临时缓冲。

二进制文件

前面讲的默认都是读取文本文件，并且是UTF-8编码的文本文件。要读取二进制文件，比如图片、视频等等，用 `'rb'` 模式打开文件即可：

```
>>> f = open('/Users/michael/test.jpg', 'rb')
>>> f.read()
b'\xff\xd8\xff\xe1\x00\x18Exif\x00\x00...' # 十六进制表示的字节
```

字符编码

要读取非UTF-8编码的文本文件，需要给 `open()` 函数传入 `encoding` 参数，例如，读取GBK编码的文件：

```
>>> f = open('/Users/michael/gbk.txt', 'r', encoding='gbk')
>>> f.read()
```

'测试'

遇到有些编码不规范的文件，你可能会遇到 `UnicodeDecodeError`，因为在文本文件中可能夹杂了一些非法编码的字符。遇到这种情况，`open()` 函数还接收一个 `errors` 参数，表示如果遇到编码错误后如何处理。最简单的方式是直接忽略：

```
>>> f = open('/Users/michael/gbk.txt', 'r', encoding='gbk', errors='ignore')
```

写文件

写文件和读文件是一样的，唯一区别是调用 `open()` 函数时，传入标识符 `'w'` 或者 `'wb'` 表示写文本文件或写二进制文件：

```
>>> f = open('/Users/michael/test.txt', 'w')
>>> f.write('Hello, world!')
>>> f.close()
```

你可以反复调用 `write()` 来写入文件，但是务必要调用 `f.close()` 来关闭文件。当我们写文件时，操作系统往往不会立刻把数据写入磁盘，而是放到内存缓存起来，空闲的时候再慢慢写入。只有调用 `close()` 方法时，操作系统才保证把没有写入的数据全部写入磁盘。忘记调用 `close()` 的后果是数据可能只写了一部分到磁盘，剩下的丢失了。所以，还是用 `with` 语句来得保险：

```
with open('/Users/michael/test.txt', 'w') as f:
    f.write('Hello, world!')
```

要写入特定编码的文本文件，请给 `open()` 函数传入 `encoding` 参数，将字符串自动转换成指定编码。

细心的童鞋会发现，以 `'w'` 模式写入文件时，如果文件已存在，会直接覆盖（相当于删掉后新写入一个文件）。如果我们希望追加到文件末尾怎么办？可以传入 `'a'` 以追加（append）模式写入。

所有模式的定义及含义可以参考Python的[官方文档](#)。

小结

在Python中，文件读写是通过 `open()` 函数打开的文件对象完成的。使用 `with` 语句操作文件IO是个好习惯。

参考源码

[with_file.py](#)

2、StringIO和BytesIO

StringIO

很多时候，数据读写不一定是文件，也可以在内存中读写。

StringIO顾名思义就是在内存中读写str。

要把str写入StringIO，我们需要先创建一个StringIO，然后，像文件一样写入即可：

```
>>> from io import StringIO
>>> f = StringIO()
>>> f.write('hello')
5
>>> f.write(' ')
1
>>> f.write('world!')
6
>>> print(f.getvalue())
hello world!
```

`getvalue()` 方法用于获得写入后的str。

要读取StringIO，可以用一个str初始化StringIO，然后，像读文件一样读取：

```
>>> from io import StringIO
>>> f = StringIO('Hello!\nHi!\nGoodbye!')
>>> while True:
...     s = f.readline()
...     if s == '':
...         break
...     print(s.strip())
...
Hello!
Hi!
Goodbye!
```


BytesIO

StringIO操作的只能是str，如果要操作二进制数据，就需要使用BytesIO。

BytesIO实现了在内存中读写bytes，我们创建一个BytesIO，然后写入一些bytes：

```
>>> from io import BytesIO
>>> f = BytesIO()
>>> f.write('中文'.encode('utf-8'))
6
>>> print(f.getvalue())
b'\xe4\xba\xad\xe6\x96\x87'
```

请注意，写入的不是str，而是经过UTF-8编码的bytes。

和StringIO类似，可以用一个bytes初始化BytesIO，然后，像读文件一样读取：

```
>>> from io import BytesIO
>>> f = BytesIO(b'\xe4\xba\xad\xe6\x96\x87')
>>> f.read()
b'\xe4\xba\xad\xe6\x96\x87'
```

小结

StringIO和BytesIO是在内存中操作str和bytes的方法，使得和读写文件具有一致的接口。

参考源码

[do_stringio.py](#)

[do_bytesio.py](#)

3、操作文件和目录

如果我们要操作文件、目录，可以在命令行下面输入操作系统提供的各种命令来完成。比如 `dir`、`cp` 等命令。

如果要在Python程序中执行这些目录和文件的操作怎么办？其实操作系统提供的命令只是简单地调用了操作系统提供的接口函数，Python内置的 `os` 模块也可以直接调用操作系统提供的接口函数。

打开Python交互式命令行，我们来看看如何使用 `os` 模块的基本功能：

```
>>> import os
>>> os.name # 操作系统类型
'posix'
```

如果是 `posix`，说明系统是 `Linux`、`Unix` 或 `Mac OS X`，如果是 `nt`，就是 `Windows` 系统。

要获取详细的系统信息，可以调用 `uname()` 函数：

```
>>> os.uname()
posix.uname_result(sysname='Darwin', nodename='MichaelMacPro.local', release='14.3.0', version='Darwin Kernel Version 14.3.0: Mon Mar 23 11:59:05 PDT 2015; root:xnu-2782.20.48~5/RELEASE_X86_64', machine='x86_64')
```

注意 `uname()` 函数在Windows上不提供，也就是说，`os` 模块的某些函数是跟操作系统相关的。

环境变量

在操作系统中定义的环境变量，全部保存在 `os.environ` 这个变量中，可以直接查看：

```
>>> os.environ
environ({'VERSIONER_PYTHON_PREFER_32_BIT': 'no', 'TERM_PROGRAM_VERSION': '326', 'LOGNAME': 'michael', 'USER': 'michael', 'PATH': '/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/opt/X11/bin:/usr/local/mysql/bin', ...})
```

要获取某个环境变量的值，可以调用 `os.environ.get('key')`：

```
>>> os.environ.get('PATH')
'/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/opt/X11/bin:/usr/local/mysql/bin'
>>> os.environ.get('x', 'default')
'default'
```

操作文件和目录

操作文件和目录的函数一部分放在 `os` 模块中，一部分放在 `os.path` 模块中，这一点要注意一下。查看、创建和删除目录可以这么调用：

```
# 查看当前目录的绝对路径:
>>> os.path.abspath('.')
'/Users/michael'
# 在某个目录下创建一个新目录，首先把新目录的完整路径表示出来:
>>> os.path.join('/Users/michael', 'testdir')
'/Users/michael/testdir'
# 然后创建一个目录:
>>> os.mkdir('/Users/michael/testdir')
# 删掉一个目录:
>>> os.rmdir('/Users/michael/testdir')
```

把两个路径合成一个时，不要直接拼字符串，而要通过 `os.path.join()` 函数，这样可以正确处理不同操作系统的路径分隔符。在Linux/Unix/Mac下，`os.path.join()` 返回这样的字符串：

```
part-1/part-2
```

而Windows下会返回这样的字符串：

```
part-1\part-2
```

同样的道理，要拆分路径时，也不要直接去拆字符串，而要通过 `os.path.split()` 函数，这样可以 把一个路径拆分为两部分，后一部分总是最后级别的目录或文件名：

```
>>> os.path.split('/Users/michael/testdir/file.txt')
('/Users/michael/testdir', 'file.txt')
```

`os.path.splitext()` 可以直接让你得到文件扩展名，很多时候非常方便：

```
>>> os.path.splitext('/path/to/file.txt')
('/path/to/file', '.txt')
```

这些合并、拆分路径的函数并不要求目录和文件要真实存在，它们只对字符串进行操作。

文件操作使用下面的函数。假定当前目录下有一个 `test.txt` 文件：

```
# 对文件重命名:
>>> os.rename('test.txt', 'test.py')
# 删掉文件:
>>> os.remove('test.py')
```

但是复制文件的函数居然在 `os` 模块中不存在！原因是复制文件并非由操作系统提供的系统调用。理论上讲，我们通过上一节的读写文件可以完成文件复制，只不过要多写很多代码。

幸运的是 `shutil` 模块提供了 `copyfile()` 的函数，你还可以在 `shutil` 模块中找到很多实用函数，它们可以看做是 `os` 模块的补充。

最后看看如何利用Python的特性来过滤文件。比如我们要列出当前目录下的所有目录，只需要一行代码：

```
>>> [x for x in os.listdir('.') if os.path.isdir(x)]
['.lein', '.local', '.m2', '.npm', '.ssh', '.Trash', '.vim', 'Applications', 'Desktop', ...]
```

要列出所有的 `.py` 文件，也只需一行代码：

```
>>> [x for x in os.listdir('.') if os.path.isfile(x) and os.path.splitext(x)[1]=='.py']
['apis.py', 'config.py', 'models.py', 'pymonitor.py', 'test_db.py', 'urls.py', 'wsgiapp.py']
```

是不是非常简洁？

小结

Python的 `os` 模块封装了操作系统的目录和文件操作，要注意这些函数有的在 `os` 模块中，有的在 `os.path` 模块中。

练习

1. 利用 `os` 模块编写一个能实现 `dir -l` 输出的程序。
2. 编写一个程序，能在当前目录以及当前目录的所有子目录下查找文件名包含指定字符串的文件，并打印出相对路径。

参考源码

do_dir

4、序列化

在程序运行的过程中，所有的变量都是在内存中，比如，定义一个dict：

```
d = dict(name='Bob', age=20, score=88)
```

可以随时修改变量，比如把 name 改成 'Bill'，但是一旦程序结束，变量所占用的内存就被操作系统全部回收。如果没有把修改后的 'Bill' 存储到磁盘上，下次重新运行程序，变量又被初始化为 'Bob'。

我们把变量从内存中变成可存储或传输的过程称之为序列化，在Python中叫pickling，在其他语言中也被称之为serialization, marshallng, flattening等等，都是一个意思。

序列化之后，就可以把序列化后的内容写入磁盘，或者通过网络传输到别的机器上。

反过来，把变量内容从序列化的对象重新读到内存里称之为反序列化，即unpickling。

Python提供了 pickle 模块来实现序列化。

首先，我们尝试把一个对象序列化并写入文件：

```
>>> import pickle
>>> d = dict(name='Bob', age=20, score=88)
>>> pickle.dumps(d)
b'\x80\x03}q\x00(X\x03\x00\x00\x00ageq\x01K\x14X\x05\x00\x00\x00scoreq\x02KXX\x04\x00\x00\x00nameq\x03X\x03\x00\x00\x00Bobq\x04u.'
```

pickle.dumps() 方法把任意对象序列化成 bytes，然后，就可以把这个 bytes 写入文件。或者用另一个方法 pickle.dump() 直接把对象序列化后写入一个file-like Object：

```
>>> f = open('dump.txt', 'wb')
>>> pickle.dump(d, f)
>>> f.close()
```

看看写入的 dump.txt 文件，一堆乱七八糟的内容，这些都是Python保存的对象内部信息。

当我们要把对象从磁盘读到内存时，可以先把内容读到一个 `bytes`，然后用 `pickle.loads()` 方法反序列化出对象，也可以直接用 `pickle.load()` 方法从一个 `file-like Object` 中直接反序列化出对象。我们打开另一个Python命令行来反序列化刚才保存的对象：

```
>>> f = open('dump.txt', 'rb')
>>> d = pickle.load(f)
>>> f.close()
>>> d
{'age': 20, 'score': 88, 'name': 'Bob'}
```

变量的内容又回来了！

当然，这个变量和原来的变量是完全不相干的对象，它们只是内容相同而已。

Pickle的问题和所有其他编程语言特有的序列化问题一样，就是它只能用于Python，并且可能不同版本的Python彼此都不兼容，因此，只能用Pickle保存那些不重要的数据，不能成功地反序列化也没关系。

JSON

如果我们要在不同的编程语言之间传递对象，就必须把对象序列化为标准格式，比如XML，但更好的方法是序列化为JSON，因为JSON表示出来就是一个字符串，可以被所有语言读取，也可以方便地存储到磁盘或者通过网络传输。JSON不仅是标准格式，并且比XML更快，而且可以直接在Web页面中读取，非常方便。

JSON表示的对象就是标准的JavaScript语言的对象，JSON和Python内置的数据类型对应如下：

JSON类型	Python类型
{ }	dict
[]	list
"string"	str
1234.56	int或float
true/false	True/False
null	None

Python内置的 `json` 模块提供了非常完善的Python对象到JSON格式的转换。我们先看看如何把Python对象变成一个JSON：

```
>>> import json
>>> d = dict(name='Bob', age=20, score=88)
>>> json.dumps(d)
'{"age": 20, "score": 88, "name": "Bob"}'
```

`dumps()` 方法返回一个 `str`，内容就是标准的JSON。类似的，`dump()` 方法可以直接把JSON写入一个 `file-like Object`。

要把JSON反序列化为Python对象，用 `loads()` 或者对应的 `load()` 方法，前者把JSON的字符串反序列化，后者从 `file-like Object` 中读取字符串并反序列化：

```
>>> json_str = '{"age": 20, "score": 88, "name": "Bob"}'
>>> json.loads(json_str)
{'age': 20, 'score': 88, 'name': 'Bob'}
```

由于JSON标准规定JSON编码是UTF-8，所以我们总是能正确地在Python的 `str` 与JSON的字符串之间转换。

JSON进阶

Python的 `dict` 对象可以直接序列化为JSON的 `{}`，不过，很多时候，我们更喜欢用 `class` 表示对象，比如定义 `Student` 类，然后序列化：

```
import json

class Student(object):
    def __init__(self, name, age, score):
        self.name = name
        self.age = age
        self.score = score

s = Student('Bob', 20, 88)
print(json.dumps(s))
```

运行代码，毫不留情地得到一个 `TypeError`：

```
Traceback (most recent call last):
...
TypeError: <__main__.Student object at 0x10603cc50> is not JSON serializable
```

错误的原因是 `Student` 对象不是一个可序列化为JSON的对象。

如果连 `class` 的实例对象都无法序列化为JSON，这肯定不合理！

别急，我们仔细看看 `dump()` 方法的参数列表，可以发现，除了第一个必须的 `obj` 参数外，`dump()` 方法还提供了一大堆的可选参数：

<https://docs.python.org/3/library/json.html#json.dump>

这些可选参数就是让我们定制JSON序列化。前面的代码之所以无法把 `Student` 类实例序列化为JSON，是因为默认情况下，`dump()` 方法不知道如何将 `Student` 实例变为一个JSON的 `{}` 对象。

可选参数 `default` 就是把任意一个对象变成一个可序列为JSON的对象，我们只需要为 `Student` 专门写一个转换函数，再把函数传进去即可：

```
def student2dict(std):
    return {
        'name': std.name,
        'age': std.age,
        'score': std.score
    }
```

这样，`Student` 实例首先被 `student2dict()` 函数转换成 `dict`，然后再被顺利序列化为JSON：

```
>>> print(json.dump(s, default=student2dict))
{"age": 20, "name": "Bob", "score": 88}
```

不过，下次如果遇到一个 `Teacher` 类的实例，照样无法序列化为JSON。我们可以偷个懒，把任意 `class` 的实例变为 `dict`：

```
print(json.dump(s, default=lambda obj: obj.__dict__))
```

因为通常 `class` 的实例都有一个 `__dict__` 属性，它就是一个 `dict`，用来存储实例变量。也有少数例外，比如定义了 `__slots__` 的 `class`。

同样的道理，如果我们要把JSON反序列化为一个 `Student` 对象实例，`loads()` 方法首先转换出一个 `dict` 对象，然后，我们传入的 `object_hook` 函数负责把 `dict` 转换为 `Student` 实例：

```
def dict2student(d):
    return Student(d['name'], d['age'], d['score'])
```

运行结果如下：

```
>>> json_str = '{"age": 20, "score": 88, "name": "Bob"}'
>>> print(json.loads(json_str, object_hook=dict2student))
<__main__.Student object at 0x10cd3c190>
```

打印出的是反序列化的 `Student` 实例对象。

小结

Python语言特定的序列化模块是 `pickle`，但如果要把序列化搞得更通用、更符合Web标准，就可以使用 `json` 模块。

`json` 模块的 `dumps()` 和 `loads()` 函数是定义得非常好的接口的典范。当我们使用时，只需要传入一个必须的参数。但是，当默认的序列化或反序列化机制不满足我们的要求时，我们又可以传入更多的参数来定制序列化或反序列化的规则，既做到了接口简单易用，又做到了充分的扩展性和灵活性。

参考源码

[use_pickle.py](#)

[use_json.py](#)

三、进程和线程

很多同学都听说过，现代操作系统比如Mac OS X，UNIX，Linux，Windows等，都是支持“多任务”的操作系统。

什么叫“多任务”呢？简单地说，就是操作系统可以同时运行多个任务。打个比方，你一边在用浏览器上网，一边在听MP3，一边在用Word赶作业，这就是多任务，至少同时有3个任务正在运行。还有很多任务悄悄地在后台同时运行着，只是桌面上没有显示而已。

现在，多核CPU已经非常普及了，但是，即使过去的单核CPU，也可以执行多任务。由于CPU执

行代码都是顺序执行的，那么，单核CPU是怎么执行多任务的呢？

答案就是操作系统轮流让各个任务交替执行，任务1执行0.01秒，切换到任务2，任务2执行0.01秒，再切换到任务3，执行0.01秒.....这样反复执行下去。表面上看，每个任务都是交替执行的，但是，由于CPU的执行速度实在是太快了，我们感觉就像所有任务都在同时执行一样。

真正的并行执行多任务只能在多核CPU上实现，但是，由于任务数量远远多于CPU的核心数量，所以，操作系统也会自动把很多任务轮流调度到每个核心上执行。

对于操作系统来说，一个任务就是一个进程（Process），比如打开一个浏览器就是启动一个浏览器进程，打开一个记事本就启动了一个记事本进程，打开两个记事本就启动了两个记事本进程，打开一个Word就启动了一个Word进程。

有些进程还不止同时干一件事，比如Word，它可以同时进行打字、拼写检查、打印等事情。在一个进程内部，要同时干多件事，就需要同时运行多个“子任务”，我们把进程内的这些“子任务”称为线程（Thread）。

由于每个进程至少要干一件事，所以，一个进程至少有一个线程。当然，像Word这种复杂的进程可以有多个线程，多个线程可以同时执行，多线程的执行方式和多进程是一样的，也是由操作系统在多个线程之间快速切换，让每个线程都短暂地交替运行，看起来就像同时执行一样。当然，真正地同时执行多线程需要多核CPU才可能实现。

我们前面编写的所有的Python程序，都是执行单任务的进程，也就是只有一个线程。如果我们要同时执行多个任务怎么办？

有两种解决方案：

一种是启动多个进程，每个进程虽然只有一个线程，但多个进程可以一块执行多个任务。

还有一种方法是启动一个进程，在一个进程内启动多个线程，这样，多个线程也可以一块执行多个任务。

当然还有第三种方法，就是启动多个进程，每个进程再启动多个线程，这样同时执行的任务就更多了，当然这种模型更复杂，实际很少采用。

总结一下就是，多任务的实现有3种方式：

- 多进程模式；
- 多线程模式；
- 多进程+多线程模式。

同时执行多个任务通常各个任务之间并不是没有关联的，而是需要相互通信和协调，有时，任务1必须暂停等待任务2完成后才能继续执行，有时，任务3和任务4又不能同时执行，所以，多进程

和多线程的程序的复杂度要远远高于我们前面写的单进程单线程的程序。

因为复杂度高，调试困难，所以，不是迫不得已，我们也不想编写多任务。但是，有很多时候，没有多任务还真不行。想想在电脑上看电影，就必须由一个线程播放视频，另一个线程播放音频，否则，单线程实现的话就只能先把视频播放完再播放音频，或者先把音频播放完再播放视频，这显然是不行的。

Python既支持多进程，又支持多线程，我们会讨论如何编写这两种多任务程序。

小结

线程是最小的执行单元，而进程由至少一个线程组成。如何调度进程和线程，完全由操作系统决定，程序自己不能决定什么时候执行，执行多长时间。

多进程和多线程的程序涉及到同步、数据共享的问题，编写起来更复杂。

1、多进程

要让Python程序实现多进程（multiprocessing），我们先了解操作系统的相关知识。

Unix/Linux操作系统提供了一个 `fork()` 系统调用，它非常特殊。普通的函数调用，调用一次，返回一次，但是 `fork()` 调用一次，返回两次，因为操作系统自动把当前进程（称为父进程）复制了一份（称为子进程），然后，分别在父进程和子进程内返回。

子进程永远返回 0，而父进程返回子进程的ID。这样做的理由是，一个父进程可以fork出很多子进程，所以，父进程要记下每个子进程的ID，而子进程只需要调用 `getppid()` 就可以拿到父进程的ID。

Python的 `os` 模块封装了常见的系统调用，其中就包括 `fork`，可以在Python程序中轻松创建子进程：

```
import os

print('Process (%s) start...' % os.getpid())
# Only works on Unix/Linux/Mac:
pid = os.fork()
if pid == 0:
    print('I am child process (%s) and my parent is %s.' % (os.getpid(), os.getppid()))
else:
    print('I (%s) just created a child process (%s).' % (os.getpid(), pid))
```

运行结果如下：

```
Process (876) start...
I (876) just created a child process (877).
I am child process (877) and my parent is 876.
```

由于Windows没有 `fork` 调用，上面的代码在Windows上无法运行。由于Mac系统是基于BSD（Unix的一种）内核，所以，在Mac下运行是没有问题的，推荐大家用Mac学Python！

有了 `fork` 调用，一个进程在接到新任务时就可以复制出一个子进程来处理新任务，常见的Apache服务器就是由父进程监听端口，每当有新的http请求时，就fork出子进程来处理新的http请求。

multiprocessing

如果你打算编写多进程的服务程序，Unix/Linux无疑是正确的选择。由于Windows没有 `fork` 调用，难道在Windows上无法用Python编写多进程的程序？

由于Python是跨平台的，自然也应该提供一个跨平台的多进程支持。`multiprocessing` 模块就是跨平台版本的多进程模块。

`multiprocessing` 模块提供了一个 `Process` 类来代表一个进程对象，下面的例子演示了启动一个子进程并等待其结束：

```
from multiprocessing import Process
import os

# 子进程要执行的代码
def run_proc(name):
    print('Run child process %s (%s)...' % (name, os.getpid()))

if __name__ == '__main__':
    print('Parent process %s.' % os.getpid())
    p = Process(target=run_proc, args=('test',))
    print('Child process will start.')
    p.start()
    p.join()
    print('Child process end.')
```

执行结果如下：

```
Parent process 928.
```

```
Process will start.
Run child process test (929)...
Process end.
```

创建子进程时，只需要传入一个执行函数和函数的参数，创建一个 `Process` 实例，用 `start()` 方法启动，这样创建进程比 `fork()` 还要简单。

`join()` 方法可以等待子进程结束后再继续往下运行，通常用于进程间的同步。

Pool

如果要启动大量的子进程，可以用进程池的方式批量创建子进程：

```
from multiprocessing import Pool
import os, time, random

def long_time_task(name):
    print('Run task %s (%s)...' % (name, os.getpid()))
    start = time.time()
    time.sleep(random.random() * 3)
    end = time.time()
    print('Task %s runs %0.2f seconds.' % (name, (end - start)))

if __name__ == '__main__':
    print('Parent process %s.' % os.getpid())
    p = Pool(4)
    for i in range(5):
        p.apply_async(long_time_task, args=(i,))
    print('Waiting for all subprocesses done...')
    p.close()
    p.join()
    print('All subprocesses done.')
```

执行结果如下：

```
Parent process 669.
Waiting for all subprocesses done...
Run task 0 (671)...
Run task 1 (672)...
Run task 2 (673)...
Run task 3 (674)...
Task 2 runs 0.14 seconds.
Run task 4 (673)...
```

```
Task 1 runs 0.27 seconds.
Task 3 runs 0.86 seconds.
Task 0 runs 1.41 seconds.
Task 4 runs 1.91 seconds.
All subprocesses done.
```

代码解读：

对 `Pool` 对象调用 `join()` 方法会等待所有子进程执行完毕，调用 `join()` 之前必须先调用 `close()`，调用 `close()` 之后就不能继续添加新的 `Process` 了。

请注意输出的结果，task 0，1，2，3 是立刻执行的，而task 4 要等待前面某个task完成后才执行，这是因为 `Pool` 的默认大小在我的电脑上是4，因此，最多同时执行4个进程。这是 `Pool` 有意设计的限制，并不是操作系统的限制。如果改成：

```
p = Pool(5)
```

就可以同时跑5个进程。

由于 `Pool` 的默认大小是CPU的核数，如果你不幸拥有8核CPU，你要提交至少9个子进程才能看到上面的等待效果。

子进程

很多时候，子进程并不是自身，而是一个外部进程。我们创建了子进程后，还需要控制子进程的输入和输出。

`subprocess` 模块可以让我们非常方便地启动一个子进程，然后控制其输入和输出。

下面的例子演示了如何在Python代码中运行命令 `nslookup www.python.org`，这和命令行直接运行的效果是一样的：

```
import subprocess

print('$ nslookup www.python.org')
r = subprocess.call(['nslookup', 'www.python.org'])
print('Exit code:', r)
```

运行结果：

```
$ nslookup www.python.org
Server:          192.168.19.4
Address:         192.168.19.4#53

Non-authoritative answer:
www.python.org   canonical name = python.map.fastly.net.
Name:   python.map.fastly.net
Address: 199.27.79.223

Exit code: 0
```

如果子进程还需要输入，则可以通过 `communicate()` 方法输入：

```
import subprocess

print('$ nslookup')
p = subprocess.Popen(['nslookup'], stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
output, err = p.communicate(b'set q=mx\npython.org\nexit\n')
print(output.decode('utf-8'))
print('Exit code:', p.returncode)
```

上面的代码相当于在命令行执行命令 `nslookup`，然后手动输入：

```
set q=mx
python.org
exit
```

运行结果如下：

```
$ nslookup
Server:          192.168.19.4
Address:         192.168.19.4#53

Non-authoritative answer:
python.org      mail exchanger = 50 mail.python.org.

Authoritative answers can be found from:
mail.python.org internet address = 82.94.164.166
mail.python.org has AAAA address 2001:888:2000:d::a6
```

Exit code: 0

进程间通信

Process 之间肯定是需要通信的，操作系统提供了很多机制来实现进程间的通信。Python 的 multiprocessing 模块包装了底层的机制，提供了 Queue、Pipes 等多种方式来交换数据。

我们以 Queue 为例，在父进程中创建两个子进程，一个往 Queue 里写数据，一个从 Queue 里读数据：

```
from multiprocessing import Process, Queue
import os, time, random

# 写数据进程执行的代码:
def write(q):
    print('Process to write: %s' % os.getpid())
    for value in ['A', 'B', 'C']:
        print('Put %s to queue...' % value)
        q.put(value)
        time.sleep(random.random())

# 读数据进程执行的代码:
def read(q):
    print('Process to read: %s' % os.getpid())
    while True:
        value = q.get(True)
        print('Get %s from queue.' % value)

if __name__ == '__main__':
    # 父进程创建Queue，并传给各个子进程:
    q = Queue()
    pw = Process(target=write, args=(q,))
    pr = Process(target=read, args=(q,))
    # 启动子进程pw，写入:
    pw.start()
    # 启动子进程pr，读取:
    pr.start()
    # 等待pw结束:
    pw.join()
    # pr进程里是死循环，无法等待其结束，只能强行终止:
    pr.terminate()
```

运行结果如下：


```
Process to write: 50563
Put A to queue...
Process to read: 50564
Get A from queue.
Put B to queue...
Get B from queue.
Put C to queue...
Get C from queue.
```

在Unix/Linux下，`multiprocessing` 模块封装了 `fork()` 调用，使我们不需要关注 `fork()` 的细节。由于Windows没有 `fork` 调用，因此，`multiprocessing` 需要“模拟”出 `fork` 的效果，父进程所有Python对象都必须通过pickle序列化再传到子进程去，所有，如果 `multiprocessing` 在Windows下调用失败了，要先考虑是不是pickle失败了。

小结

在Unix/Linux下，可以使用 `fork()` 调用实现多进程。

要实现跨平台的多进程，可以使用 `multiprocessing` 模块。

进程间通信是通过 `Queue` 、 `Pipes` 等实现的。

参考源码

[do_folk.py](#)

[multi_processing.py](#)

[pooled_processing.py](#)

[do_subprocess.py](#)

[do_queue.py](#)

2、多线程

多任务可以由多进程完成，也可以由一个进程内的多线程完成。

我们前面提到了进程是由若干线程组成的，一个进程至少有一个线程。

由于线程是操作系统直接支持的执行单元，因此，高级语言通常都内置多线程的支持，Python也不例外，并且，Python的线程是真正的Posix Thread，而不是模拟出来的线程。

Python的标准库提供了两个模块：`_thread` 和 `threading`，`_thread` 是低级模块，`threading` 是

高级模块，对 `_thread` 进行了封装。绝大多数情况下，我们只需要使用 `threading` 这个高级模块。

启动一个线程就是把一个函数传入并创建 `Thread` 实例，然后调用 `start()` 开始执行：

```
import time, threading

# 新线程执行的代码:
def loop():
    print('thread %s is running...' % threading.current_thread().name)
    n = 0
    while n < 5:
        n = n + 1
        print('thread %s >>> %s' % (threading.current_thread().name, n))
        time.sleep(1)
    print('thread %s ended.' % threading.current_thread().name)

print('thread %s is running...' % threading.current_thread().name)
t = threading.Thread(target=loop, name='LoopThread')
t.start()
t.join()
print('thread %s ended.' % threading.current_thread().name)
```

执行结果如下：

```
thread MainThread is running...
thread LoopThread is running...
thread LoopThread >>> 1
thread LoopThread >>> 2
thread LoopThread >>> 3
thread LoopThread >>> 4
thread LoopThread >>> 5
thread LoopThread ended.
thread MainThread ended.
```

由于任何进程默认就会启动一个线程，我们把该线程称为主线程，主线程又可以启动新的线程，Python的 `threading` 模块有个 `current_thread()` 函数，它永远返回当前线程的实例。主线程实例的名字叫 `MainThread`，子线程的名字在创建时指定，我们用 `LoopThread` 命名子线程。名字仅仅在打印时用来显示，完全没有其他意义，如果不起名字Python就自动给线程命名为 `Thread-1`，`Thread-2`

Lock

多线程和多进程最大的不同在于，多进程中，同一个变量，各自有一份拷贝存在于每个进程中，互不影响，而多线程中，所有变量都由所有线程共享，所以，任何一个变量都可以被任何一个线程修改，因此，线程之间共享数据最大的危险在于多个线程同时改一个变量，把内容给改乱了。

来看看多个线程同时操作一个变量怎么把内容给改乱了：

```
import time, threading

# 假定这是你的银行存款：
balance = 0

def change_it(n):
    # 先存后取，结果应该为0：
    global balance
    balance = balance + n
    balance = balance - n

def run_thread(n):
    for i in range(100000):
        change_it(n)

t1 = threading.Thread(target=run_thread, args=(5,))
t2 = threading.Thread(target=run_thread, args=(8,))
t1.start()
t2.start()
t1.join()
t2.join()
print(balance)
```

我们定义了一个共享变量 `balance`，初始值为 `0`，并且启动两个线程，先存后取，理论上结果应该为 `0`，但是，由于线程的调度是由操作系统决定的，当 `t1`、`t2` 交替执行时，只要循环次数足够多，`balance` 的结果就不一定是 `0` 了。

原因是因为高级语言的一条语句在CPU执行时是若干条语句，即使一个简单的计算：

```
balance = balance + n
```

也分两步：

1. 计算 `balance + n`，存入临时变量中；
2. 将临时变量的值赋给 `balance`。

也就是可以看成：

```
x = balance + n
balance = x
```

由于x是局部变量，两个线程各自都有自己的x，当代码正常执行时：

```
初始值 balance = 0

t1: x1 = balance + 5 # x1 = 0 + 5 = 5
t1: balance = x1     # balance = 5
t1: x1 = balance - 5 # x1 = 5 - 5 = 0
t1: balance = x1     # balance = 0

t2: x2 = balance + 8 # x2 = 0 + 8 = 8
t2: balance = x2     # balance = 8
t2: x2 = balance - 8 # x2 = 8 - 8 = 0
t2: balance = x2     # balance = 0

结果 balance = 0
```

但是t1和t2是交替运行的，如果操作系统以下面的顺序执行t1、t2：

```
初始值 balance = 0

t1: x1 = balance + 5 # x1 = 0 + 5 = 5

t2: x2 = balance + 8 # x2 = 0 + 8 = 8
t2: balance = x2     # balance = 8

t1: balance = x1     # balance = 5
t1: x1 = balance - 5 # x1 = 5 - 5 = 0
t1: balance = x1     # balance = 0

t2: x2 = balance - 8 # x2 = 0 - 8 = -8
t2: balance = x2     # balance = -8

结果 balance = -8
```

究其原因，是因为修改 balance 需要多条语句，而执行这几条语句时，线程可能中断，从而导致多个线程把同一个对象的内容改乱了。

两个线程同时一存一取，就可能导致余额不对，你肯定不希望你的银行存款莫名其妙地变成了负数，所以，我们必须确保一个线程在修改 `balance` 的时候，别的线程一定不能改。

如果我们要确保 `balance` 计算正确，就要给 `change_it()` 上一把锁，当某个线程开始执行 `change_it()` 时，我们说，该线程因为获得了锁，因此其他线程不能同时执行 `change_it()`，只能等待，直到锁被释放后，获得该锁以后才能改。由于锁只有一个，无论多少线程，同一时刻最多只有一个线程持有该锁，所以，不会造成修改的冲突。创建一个锁就是通过 `threading.Lock()` 来实现：

```
balance = 0
lock = threading.Lock()

def run_thread(n):
    for i in range(100000):
        # 先要获取锁:
        lock.acquire()
        try:
            # 放心地改吧:
            change_it(n)
        finally:
            # 改完了一定要释放锁:
            lock.release()
```

当多个线程同时执行 `lock.acquire()` 时，只有一个线程能成功地获取锁，然后继续执行代码，其他线程就继续等待直到获得锁为止。

获得锁的线程用完后一定要释放锁，否则那些苦苦等待锁的线程将永远等待下去，成为死线程。所以我们用 `try...finally` 来确保锁一定会被释放。

锁的好处就是确保了某段关键代码只能由一个线程从头到尾完整地执行，坏处当然也很多，首先是阻止了多线程并发执行，包含锁的某段代码实际上只能以单线程模式执行，效率就大大地下降了。其次，由于可以存在多个锁，不同的线程持有不同的锁，并试图获取对方持有的锁时，可能会造成死锁，导致多个线程全部挂起，既不能执行，也无法结束，只能靠操作系统强制终止。

多核CPU

如果你不幸拥有一个多核CPU，你肯定在想，多核应该可以同时执行多个线程。

如果写一个死循环的话，会出现什么情况呢？

打开Mac OS X的Activity Monitor，或者Windows的Task Manager，都可以监控某个进程的CPU使用率。

我们可以监控到一个死循环线程会100%占用一个CPU。

如果有两个死循环线程，在多核CPU中，可以监控到会占用200%的CPU，也就是占用两个CPU核心。

要想把N核CPU的核心全部跑满，就必须启动N个死循环线程。

试试用Python写个死循环：

```
import threading, multiprocessing

def loop():
    x = 0
    while True:
        x = x ^ 1

for i in range(multiprocessing.cpu_count()):
    t = threading.Thread(target=loop)
    t.start()
```

启动与CPU核心数量相同的N个线程，在4核CPU上可以监控到CPU占用率仅有102%，也就是仅使用了一核。

但是用C、C++或Java来改写相同的死循环，直接可以把全部核心跑满，4核就跑到400%，8核就跑到800%，为什么Python不行呢？

因为Python的线程虽然是真正的线程，但解释器执行代码时，有一个GIL锁：Global Interpreter Lock，任何Python线程执行前，必须先获得GIL锁，然后，每执行100条字节码，解释器就自动释放GIL锁，让别的线程有机会执行。这个GIL全局锁实际上把所有线程的执行代码都给上了锁，所以，多线程在Python中只能交替执行，即使100个线程跑在100核CPU上，也只能用到1个核。

GIL是Python解释器设计的历史遗留问题，通常我们用的解释器是官方实现的CPython，要真正利用多核，除非重写一个不带GIL的解释器。

所以，在Python中，可以使用多线程，但不要指望能有效利用多核。如果一定要通过多线程利用多核，那只能通过C扩展来实现，不过这样就失去了Python简单易用的特点。

不过，也不用过于担心，Python虽然不能利用多线程实现多核任务，但可以通过多进程实现多核任务。多个Python进程有各自独立的GIL锁，互不影响。

小结

多线程编程，模型复杂，容易发生冲突，必须用锁加以隔离，同时，又要小心死锁的发生。

Python解释器由于设计时有GIL全局锁，导致了多线程无法利用多核。多线程的并发在Python中就是一个美丽的梦。

参考源码

[multi_threading.py](#)

[do_lock.py](#)

3、ThreadLocal

在多线程环境下，每个线程都有自己的数据。一个线程使用自己的局部变量比使用全局变量好，因为局部变量只有线程自己能看见，不会影响其他线程，而全局变量的修改必须加锁。

但是局部变量也有问题，就是在函数调用的时候，传递起来很麻烦：

```
def process_student(name):
    std = Student(name)
    # std是局部变量，但是每个函数都要用它，因此必须传进去：
    do_task_1(std)
    do_task_2(std)

def do_task_1(std):
    do_subtask_1(std)
    do_subtask_2(std)

def do_task_2(std):
    do_subtask_2(std)
    do_subtask_2(std)
```

每个函数一层一层调用都这么传参数那还得了？用全局变量？也不行，因为每个线程处理不同的 `Student` 对象，不能共享。

如果用一个全局 `dict` 存放所有的 `Student` 对象，然后以 `thread` 自身作为 `key` 获得线程对应的 `Student` 对象如何？

```
global_dict = {}

def std_thread(name):
    std = Student(name)
    # 把std放到全局变量global_dict中：
    global_dict[threading.current_thread()] = std
    do_task_1()
```

```
do_task_2()

def do_task_1():
    # 不传入std, 而是根据当前线程查找:
    std = global_dict[threading.current_thread()]
    ...

def do_task_2():
    # 任何函数都可以查找出当前线程的std变量:
    std = global_dict[threading.current_thread()]
    ...
```

这种方式理论上是可行的，它最大的优点是消除了 `std` 对象在每层函数中的传递问题，但是，每个函数获取 `std` 的代码有点丑。

有没有更简单的方式？

`ThreadLocal` 应运而生，不用查找 `dict`，`ThreadLocal` 帮你自动做这件事：

```
import threading

# 创建全局ThreadLocal对象:
local_school = threading.local()

def process_student():
    # 获取当前线程关联的student:
    std = local_school.student
    print('Hello, %s (in %s)' % (std, threading.current_thread().name))

def process_thread(name):
    # 绑定ThreadLocal的student:
    local_school.student = name
    process_student()

t1 = threading.Thread(target= process_thread, args=('Alice',), name='Thread-A')
t2 = threading.Thread(target= process_thread, args=('Bob',), name='Thread-B')
t1.start()
t2.start()
t1.join()
t2.join()
```

执行结果：


```
Hello, Alice (in Thread-A)
Hello, Bob (in Thread-B)
```

全局变量 `local_school` 就是一个 `ThreadLocal` 对象，每个 `Thread` 对它都可以读写 `student` 属性，但互不影响。你可以把 `local_school` 看成全局变量，但每个属性如 `local_school.student` 都是线程的局部变量，可以任意读写而互不干扰，也不用管理锁的问题，`ThreadLocal` 内部会处理。

可以理解为全局变量 `local_school` 是一个 `dict`，不但可以用 `local_school.student`，还可以绑定其他变量，如 `local_school.teacher` 等等。

`ThreadLocal` 最常用的地方就是为每个线程绑定一个数据库连接，HTTP请求，用户身份信息等，这样一个线程的所有调用到的处理函数都可以非常方便地访问这些资源。

小结

一个 `ThreadLocal` 变量虽然是全局变量，但每个线程都只能读写自己线程的独立副本，互不干扰。`ThreadLocal` 解决了参数在一个线程中各个函数之间互相传递的问题。

参考源码

[use_threadlocal.py](#)

4、进程 vs. 线程

我们介绍了多进程和多线程，这是实现多任务最常用的两种方式。现在，我们来讨论一下这两种方式的优缺点。

首先，要实现多任务，通常会设计Master-Worker模式，Master负责分配任务，Worker负责执行任务，因此，多任务环境下，通常是一个Master，多个Worker。

如果用多进程实现Master-Worker，主进程就是Master，其他进程就是Worker。

如果用多线程实现Master-Worker，主线程就是Master，其他线程就是Worker。

多进程模式最大的优点就是稳定性高，因为一个子进程崩溃了，不会影响主进程和其他子进程。（当然主进程挂了所有进程就全挂了，但是Master进程只负责分配任务，挂掉的概率低）著名的Apache最早就是采用多进程模式。

多进程模式的缺点是创建进程的代价大，在Unix/Linux系统下，用 `fork` 调用还行，在Windows下创建进程开销巨大。另外，操作系统能同时运行的进程数也是有限的，在内存和CPU的限制下，如果有几千个进程同时运行，操作系统连调度都会成问题。

多线程模式通常比多进程快一点，但是也快不到哪去，而且，多线程模式致命的缺点就是任何一个线程挂掉都可能直接造成整个进程崩溃，因为所有线程共享进程的内存。在Windows上，如果一个线程执行的代码出了问题，你经常可以看到这样的提示：“该程序执行了非法操作，即将关闭”，其实往往是某个线程出了问题，但是操作系统会强制结束整个进程。

在Windows下，多线程的效率比多进程要高，所以微软的IIS服务器默认采用多线程模式。由于多线程存在稳定性的问题，IIS的稳定性就不如Apache。为了缓解这个问题，IIS和Apache现在又有多进程+多线程的混合模式，真是把问题越搞越复杂。

线程切换

无论是多进程还是多线程，只要数量一多，效率肯定上不去，为什么呢？

我们打个比方，假设你不幸正在准备中考，每天晚上需要做语文、数学、英语、物理、化学这5科的作业，每项作业耗时1小时。

如果你先花1小时做语文作业，做完了，再花1小时做数学作业，这样，依次全部做完，一共花5小时，这种方式称为单任务模型，或者批处理任务模型。

假设你打算切换到多任务模型，可以先做1分钟语文，再切换到数学作业，做1分钟，再切换到英语，以此类推，只要切换速度足够快，这种方式就和单核CPU执行多任务是一样的了，以幼儿园小朋友的眼光来看，你就正在同时写5科作业。

但是，切换作业是有代价的，比如从语文切到数学，要先收拾桌子上的语文书本、钢笔（这叫保存现场），然后，打开数学课本、找出圆规直尺（这叫准备新环境），才能开始做数学作业。操作系统在切换进程或者线程时也是一样的，它需要先保存当前执行的现场环境（CPU寄存器状态、内存页等），然后，把新任务的执行环境准备好（恢复上次的寄存器状态，切换内存页等），才能开始执行。这个切换过程虽然很快，但是也需要耗费时间。如果有几千个任务同时进行，操作系统可能就主要忙着切换任务，根本没有多少时间去执行任务了，这种情况最常见的就是硬盘狂响，点窗口无反应，系统处于假死状态。

所以，多任务一旦多到一个限度，就会消耗掉系统所有的资源，结果效率急剧下降，所有任务都做不好。

计算密集型 vs. IO密集型

是否采用多任务的第二个考虑是任务的类型。我们可以把任务分为计算密集型和IO密集型。

计算密集型任务的特点是要进行大量的计算，消耗CPU资源，比如计算圆周率、对视频进行高清解码等等，全靠CPU的运算能力。这种计算密集型任务虽然也可以用多任务完成，但是任务越多，花在任务切换的时间就越多，CPU执行任务的效率就越低，所以，要最高效地利用CPU，计算密集型任务同时进行的数量应当等于CPU的核心数。

计算密集型任务由于主要消耗CPU资源，因此，代码运行效率至关重要。Python这样的脚本语言运行效率很低，完全不适合计算密集型任务。对于计算密集型任务，最好用C语言编写。

第二种任务的类型是IO密集型，涉及到网络、磁盘IO的任务都是IO密集型任务，这类任务的特点是CPU消耗很少，任务的大部分时间都在等待IO操作完成（因为IO的速度远远低于CPU和内存的速度）。对于IO密集型任务，任务越多，CPU效率越高，但也有一个限度。常见的大部分任务都是IO密集型任务，比如Web应用。

IO密集型任务执行期间，99%的时间都花在IO上，花在CPU上的时间很少，因此，用运行速度极快的C语言替换用Python这样运行速度极低的脚本语言，完全无法提升运行效率。对于IO密集型任务，最合适的语言就是开发效率最高（代码量最少）的语言，脚本语言是首选，C语言最差。

异步IO

考虑到CPU和IO之间巨大的速度差异，一个任务在执行的过程中大部分时间都在等待IO操作，单进程单线程模型会导致别的任务无法并行执行，因此，我们才需要多进程模型或者多线程模型来支持多任务并发执行。

现代操作系统对IO操作已经做了巨大的改进，最大的特点就是支持异步IO。如果充分利用操作系统提供的异步IO支持，就可以用单进程单线程模型来执行多任务，这种全新的模型称为事件驱动模型，Nginx就是支持异步IO的Web服务器，它在单核CPU上采用单进程模型就可以高效地支持多任务。在多核CPU上，可以运行多个进程（数量与CPU核心数相同），充分利用多核CPU。由于系统总的进程数量十分有限，因此操作系统调度非常高效。用异步IO编程模型来实现多任务是一个主要的趋势。

对应到Python语言，单线程的异步编程模型称为协程，有了协程的支持，就可以基于事件驱动编写高效的多任务程序。我们会在后面讨论如何编写协程。

5、分布式进程

在Thread和Process中，应当优选Process，因为Process更稳定，而且，Process可以分布到多台机器上，而Thread最多只能分布到同一台机器的多个CPU上。

Python的 `multiprocessing` 模块不但支持多进程，其中 `managers` 子模块还支持把多进程分布到多台机器上。一个服务进程可以作为调度者，将任务分布到其他多个进程中，依靠网络通信。由于 `managers` 模块封装很好，不必了解网络通信的细节，就可以很容易地编写分布式多进程程序。

举个例子：如果我们已经有一个通过 `Queue` 通信的多进程程序在同一台机器上运行，现在，由于处理任务的进程任务繁重，希望把发送任务的进程和处理任务的进程分布到两台机器上。怎么用分布式进程实现？

原有的 Queue 可以继续使用，但是，通过 managers 模块把 Queue 通过网络暴露出去，就可以让其他机器的进程访问 Queue 了。

我们先看服务进程，服务进程负责启动 Queue，把 Queue 注册到网络上，然后往 Queue 里面写入任务：

```
# task_master.py

import random, time, queue
from multiprocessing.managers import BaseManager

# 发送任务的队列:
task_queue = queue.Queue()
# 接收结果的队列:
result_queue = queue.Queue()

# 从BaseManager继承的QueueManager:
class QueueManager(BaseManager):
    pass

# 把两个Queue都注册到网络上，callable参数关联了Queue对象:
QueueManager.register('get_task_queue', callable=lambda: task_queue)
QueueManager.register('get_result_queue', callable=lambda: result_queue)
# 绑定端口5000，设置验证码'abc':
manager = QueueManager(address=('', 5000), authkey=b'abc')
# 启动Queue:
manager.start()
# 获得通过网络访问的Queue对象:
task = manager.get_task_queue()
result = manager.get_result_queue()
# 放几个任务进去:
for i in range(10):
    n = random.randint(0, 10000)
    print('Put task %d...' % n)
    task.put(n)
# 从result队列读取结果:
print('Try get results...')
for i in range(10):
    r = result.get(timeout=10)
    print('Result: %s' % r)
# 关闭:
manager.shutdown()
print('master exit.')
```

请注意，当我们在一台机器上写多进程程序时，创建的 Queue 可以直接拿来用，但是，在分布式

多进程环境下，添加任务到 Queue 不可以直接对原始的 task_queue 进行操作，那样就绕过了 QueueManager 的封装，必须通过 manager.get_task_queue() 获得的 Queue 接口添加。

然后，在另一台机器上启动任务进程（本机上启动也可以）：

```
# task_worker.py

import time, sys, queue
from multiprocessing.managers import BaseManager

# 创建类似的QueueManager:
class QueueManager(BaseManager):
    pass

# 由于这个QueueManager只从网络上获取Queue，所以注册时只提供名字:
QueueManager.register('get_task_queue')
QueueManager.register('get_result_queue')

# 连接到服务器，也就是运行task_master.py的机器:
server_addr = '127.0.0.1'
print('Connect to server %s...' % server_addr)
# 端口和验证码注意保持与task_master.py设置的完全一致:
m = QueueManager(address=(server_addr, 5000), authkey=b'abc')
# 从网络连接:
m.connect()
# 获取Queue的对象:
task = m.get_task_queue()
result = m.get_result_queue()
# 从task队列取任务,并把结果写入result队列:
for i in range(10):
    try:
        n = task.get(timeout=1)
        print('run task %d * %d...' % (n, n))
        r = '%d * %d = %d' % (n, n, n*n)
        time.sleep(1)
        result.put(r)
    except Queue.Empty:
        print('task queue is empty.')
# 处理结束:
print('worker exit.')
```

任务进程要通过网络连接到服务进程，所以要指定服务进程的IP。

现在，可以试试分布式进程的工作效果了。先启动 task_master.py 服务进程：

```
$ python3 task_master.py
Put task 3411...
Put task 1605...
Put task 1398...
Put task 4729...
Put task 5300...
Put task 7471...
Put task 68...
Put task 4219...
Put task 339...
Put task 7866...
Try get results...
```

`task_master.py` 进程发送完任务后，开始等待 `result` 队列的结果。现在启动 `task_worker.py` 进程：

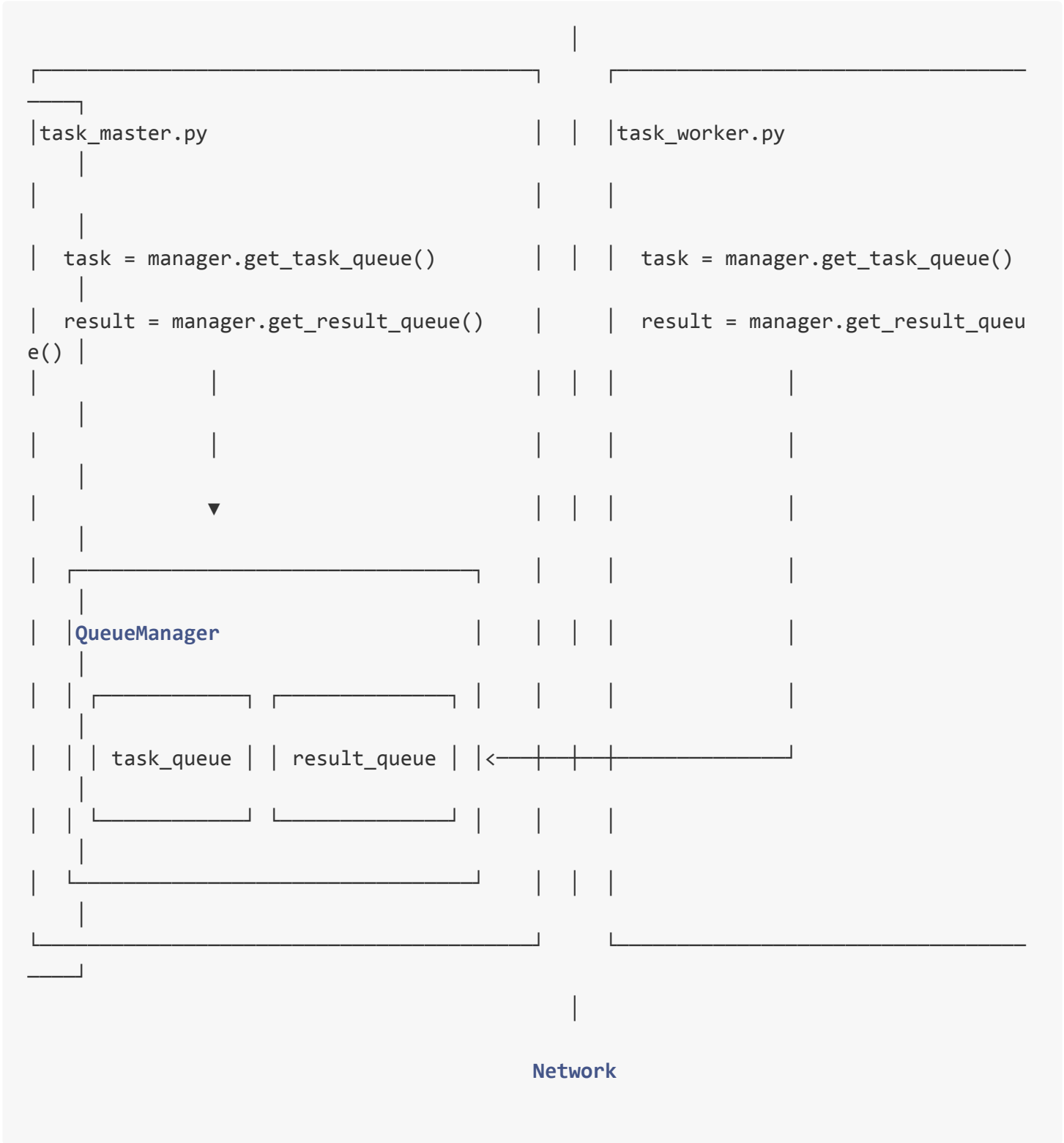
```
$ python3 task_worker.py
Connect to server 127.0.0.1...
run task 3411 * 3411...
run task 1605 * 1605...
run task 1398 * 1398...
run task 4729 * 4729...
run task 5300 * 5300...
run task 7471 * 7471...
run task 68 * 68...
run task 4219 * 4219...
run task 339 * 339...
run task 7866 * 7866...
worker exit.
```

`task_worker.py` 进程结束，在 `task_master.py` 进程中会继续打印出结果：

```
Result: 3411 * 3411 = 11634921
Result: 1605 * 1605 = 2576025
Result: 1398 * 1398 = 1954404
Result: 4729 * 4729 = 22363441
Result: 5300 * 5300 = 28090000
Result: 7471 * 7471 = 55815841
Result: 68 * 68 = 4624
Result: 4219 * 4219 = 17799961
Result: 339 * 339 = 114921
Result: 7866 * 7866 = 61873956
```

这个简单的Master/Worker模型有什么用？其实这就是一个简单但真正的分布式计算，把代码稍加改造，启动多个worker，就可以把任务分布到几台甚至几十台机器上，比如把计算 `n*n` 的代码换成发送邮件，就实现了邮件队列的异步发送。

Queue对象存储在哪？注意到 `task_worker.py` 中根本没有创建Queue的代码，所以，Queue对象存储在 `task_master.py` 进程中：



而 Queue 之所以能通过网络访问，就是通过 QueueManager 实现的。由于 QueueManager 管理的不止一个 Queue，所以，要给每个 Queue 的网络调用接口起个名字，比如 get_task_queue。

authkey 有什么用？这是为了保证两台机器正常通信，不被其他机器恶意干扰。如果 task_worker.py 的 authkey 和 task_master.py 的 authkey 不一致，肯定连接不上。

小结

Python的分布式进程接口简单，封装良好，适合需要把繁重任务分布到多台机器的环境下。

注意Queue的作用是用来传递任务和接收结果，每个任务的描述数据量要尽量小。比如发送一个处理日志文件的任务，就不要发送几百兆的日志文件本身，而是发送日志文件存放的完整路径，由Worker进程再去共享的磁盘上读取文件。

参考源码

[task_master.py](#)

[task_worker.py](#)

四、正则表达式

字符串是编程时涉及到的最多的一种数据结构，对字符串进行操作的需求几乎无处不在。比如判断一个字符串是否是合法的Email地址，虽然可以编程提取 @ 前后的子串，再分别判断是否是单词和域名，但这样做不但麻烦，而且代码难以复用。

正则表达式是一种用来匹配字符串的强有力的武器。它的设计思想是用一种描述性的语言来给字符串定义一个规则，凡是符合规则的字符串，我们就认为它“匹配”了，否则，该字符串就是不合法的。

所以我们判断一个字符串是否是合法的Email的方法是：

1. 创建一个匹配Email的正则表达式；
2. 用该正则表达式去匹配用户的输入来判断是否合法。

因为正则表达式也是用字符串表示的，所以，我们要首先了解如何用字符来描述字符。

在正则表达式中，如果直接给出字符，就是精确匹配。用 \d 可以匹配一个数字，\w 可以匹配一个字母或数字，所以：

- '00\d' 可以匹配 '007'，但无法匹配 '00A'；
- '\d\d\d' 可以匹配 '010'；

- `'\w\w\d'` 可以匹配 `'py3'` ；

. 可以匹配任意字符，所以：

- `'py.'` 可以匹配 `'pyc'` 、 `'pyo'` 、 `'py!'` 等等。

要匹配变长的字符，在正则表达式中，用 `*` 表示任意个字符（包括0个），用 `+` 表示至少一个字符，用 `?` 表示0个或1个字符，用 `{n}` 表示n个字符，用 `{n,m}` 表示n-m个字符：

来看一个复杂的例子： `\d{3}\s+\d{3,8}` 。

我们来从左到右解读一下：

1. `\d{3}` 表示匹配3个数字，例如 `'010'` ；
2. `\s` 可以匹配一个空格（也包括Tab等空白符），所以 `\s+` 表示至少有一个空格，例如匹配 `' '` ， `' '` 等；
3. `\d{3,8}` 表示3-8个数字，例如 `'1234567'` 。

综合起来，上面的正则表达式可以匹配以任意个空格隔开的带区号的电话号码。

如果要匹配 `'010-12345'` 这样的号码呢？由于 `'-'` 是特殊字符，在正则表达式中，要用 `'\'` 转义，所以，上面的正则则是 `\d{3}\-\d{3,8}` 。

但是，仍然无法匹配 `'010 - 12345'` ，因为带有空格。所以我们需要更复杂的匹配方式。

进阶

要做更精确地匹配，可以用 `[]` 表示范围，比如：

- `[0-9a-zA-Z_]` 可以匹配一个数字、字母或者下划线；
- `[0-9a-zA-Z_]+` 可以匹配至少由一个数字、字母或者下划线组成的字符串，比如 `'a100'` ， `'0_Z'` ， `'Py3000'` 等等；
- `[a-zA-Z_][0-9a-zA-Z_]*` 可以匹配由字母或下划线开头，后接任意个由一个数字、字母或者下划线组成的字符串，也就是Python合法的变量；
- `[a-zA-Z_][0-9a-zA-Z_]{0,19}` 更精确地限制了变量的长度是1-20个字符（前面1个字符+后面最多19个字符）。

`A|B` 可以匹配A或B，所以 `(P|p)ython` 可以匹配 `'Python'` 或者 `'python'` 。

`^` 表示行的开头， `^\d` 表示必须以数字开头。

re模块

有了准备知识，我们就可以在Python中使用正则表达式了。Python提供 `re` 模块，包含所有正则表达式的功能。由于Python的字符串本身也用 `\` 转义，所以要特别注意：

```
s = 'ABC\\-001' # Python的字符串
# 对应的正则表达式字符串变成：
# 'ABC\\-001'
```

因此我们强烈建议使用Python的 `r` 前缀，就不用考虑转义的问题了：

```
s = r'ABC\\-001' # Python的字符串
# 对应的正则表达式字符串不变：
# 'ABC\\-001'
```

先看看如何判断正则表达式是否匹配：

```
>>> import re
>>> re.match(r'^\d{3}\-\d{3,8}/div>', '010-12345')
<_sre.SRE_Match object; span=(0, 9), match='010-12345'>
>>> re.match(r'^\d{3}\-\d{3,8}/div>', '010 12345')
>>>
```

`match()` 方法判断是否匹配，如果匹配成功，返回一个 `Match` 对象，否则返回 `None`。常见的判断方法就是：

```
test = '用户输入的字符串'
if re.match(r'正则表达式', test):
    print('ok')
else:
    print('failed')
```

切分字符串

用正则表达式切分字符串比用固定的字符更灵活，请看正常的切分代码：

```
>>> 'a b c'.split(' ')
```

```
['a', 'b', ' ', ' ', 'c']
```

嗯，无法识别连续的空格，用正则表达式试试：

```
>>> re.split(r'\s+', 'a b c')
['a', 'b', 'c']
```

无论多少个空格都可以正常分割。加入 `,` 试试：

```
>>> re.split(r'[\s\,]+', 'a,b c d')
['a', 'b', 'c', 'd']
```

再加入 `;` 试试：

```
>>> re.split(r'[\s\,\;]+', 'a,b;; c d')
['a', 'b', 'c', 'd']
```

如果用户输入了一组标签，下次记得用正则表达式来把不规范的输入转化成正确的数组。

分组

除了简单地判断是否匹配之外，正则表达式还有提取子串的强大功能。用 `()` 表示的就是要提取的分组（Group）。比如：

```
`^(\d{3})-(\d{3,8})`, '010-12345'
>>> m
<sre.SRE_Match object; span=(0, 9), match='010-12345'>
>>> m.group(0)
'010-12345'
>>> m.group(1)
'010'
>>> m.group(2)
'12345'
```

如果正则表达式中定义了组，就可以在 `Match` 对象上用 `group()` 方法提取出子串来。

注意到 `group(0)` 永远是原始字符串，`group(1)`、`group(2)`表示第1、2、.....个子串。

提取子串非常有用。来看一个更凶残的例子：

```
>>> t = '19:05:30'
>>> m = re.match(r'^(0[0-9]|1[0-9]|2[0-3]|[0-9])\:(0[0-9]|1[0-9]|2[0-9]|3[0-9]|4[0-9]|5[0-9]|[0-9])\:(0[0-9]|1[0-9]|2[0-9]|3[0-9]|4[0-9]|5[0-9]|[0-9])/div>, t)
>>> m.groups()
('19', '05', '30')
```

这个正则表达式可以直接识别合法的时间。但是有些时候，用正则表达式也无法做到完全验证，比如识别日期：

```
'^(0[1-9]|1[0-2]|[0-9])-(0[1-9]|1[0-9]|2[0-9]|3[0-1]|[0-9])/div>
```

对于 `'2-30'`，`'4-31'` 这样的非法日期，用正则还是识别不了，或者说写出来非常困难，这时就需要程序配合识别了。

贪婪匹配

最后需要特别指出的是，正则匹配默认是贪婪匹配，也就是匹配尽可能多的字符。举例如下，匹配出数字后面的 `0`：

```
>>> re.match(r'^(\d+)(0*)/div>', '102300').groups()
('102300', '')
```

由于 `\d+` 采用贪婪匹配，直接把后面的 `0` 全部匹配了，结果 `0*` 只能匹配空字符串了。

必须让 `\d+` 采用非贪婪匹配（也就是尽可能少匹配），才能把后面的 `0` 匹配出来，加个 `?` 就可以让 `\d+` 采用非贪婪匹配：

```
>>> re.match(r'^(\d+?)(0*)/div>', '102300').groups()
('1023', '00')
```

编译

当我们在Python中使用正则表达式时，`re`模块内部会干两件事情：

1. 编译正则表达式，如果正则表达式的字符串本身不合法，会报错；
2. 用编译后的正则表达式去匹配字符串。

如果一个正则表达式要重复使用几千次，出于效率的考虑，我们可以预编译该正则表达式，接下来重复使用时就不需要编译这个步骤了，直接匹配：

```
>>> import re
编译:
>>> re_telephone = re.compile(r'^(\d{3})-(\d{3,8})/div>)
使用:
>>> re_telephone.match('010-12345').groups()
('010', '12345')
>>> re_telephone.match('010-8086').groups()
('010', '8086')
```

编译后生成Regular Expression对象，由于该对象自己包含了正则表达式，所以调用对应的方法时不用给出正则字符串。

小结

正则表达式非常强大，要在短短的一节里讲完是不可能的。要讲清楚正则的所有内容，可以写一本厚厚的书了。如果你经常遇到正则表达式的问题，你可能需要一本正则表达式的参考书。

参考源码

[regex.py](#)

五、内建模块

1、datetime

datetime是Python处理日期和时间的标准库。

获取当前日期和时间

我们先看如何获取当前日期和时间：

```
>>> from datetime import datetime
>>> now = datetime.now() # 获取当前datetime
>>> print(now)
2015-05-18 16:28:07.198690
```

```
>>> print(type(now))
<class 'datetime.datetime'>
```

注意到 `datetime` 是模块，`datetime` 模块还包含一个 `datetime` 类，通过 `from datetime import datetime` 导入的才是 `datetime` 这个类。

如果仅导入 `import datetime`，则必须引用全名 `datetime.datetime`。

`datetime.now()` 返回当前日期和时间，其类型是 `datetime`。

获取指定日期和时间

要指定某个日期和时间，我们直接用参数构造一个 `datetime`：

```
>>> from datetime import datetime
>>> dt = datetime(2015, 4, 19, 12, 20) # 用指定日期时间创建datetime
>>> print(dt)
2015-04-19 12:20:00
```

datetime转换为timestamp

在计算机中，时间实际上是用数字表示的。我们把1970年1月1日 00:00:00 UTC+00:00时区的时刻称为epoch time，记为 `0`（1970年以前的时间timestamp为负数），当前时间就是相对于epoch time的秒数，称为timestamp。

你可以认为：

```
timestamp = 0 = 1970-1-1 00:00:00 UTC+0:00
```

对应的北京时间是：

```
timestamp = 0 = 1970-1-1 08:00:00 UTC+8:00
```

可见timestamp的值与时区毫无关系，因为timestamp一旦确定，其UTC时间就确定了，转换到任意时区的时间也是完全确定的，这就是为什么计算机存储的当前时间是以timestamp表示的，因为全球各地的计算机在任意时刻的timestamp都是完全相同的（假定时间已校准）。

把一个 `datetime` 类型转换为timestamp只需要简单调用 `timestamp()` 方法：

```
>>> from datetime import datetime
>>> dt = datetime(2015, 4, 19, 12, 20) # 用指定日期时间创建datetime
>>> dt.timestamp() # 把datetime转换为timestamp
1429417200.0
```

注意Python的timestamp是一个浮点数。如果有小数位，小数位表示毫秒数。

某些编程语言（如Java和JavaScript）的timestamp使用整数表示毫秒数，这种情况下只需要把timestamp除以1000就得到Python的浮点表示方法。

timestamp转换为datetime

要把timestamp转换为 `datetime`，使用 `datetime` 提供的 `fromtimestamp()` 方法：

```
>>> from datetime import datetime
>>> t = 1429417200.0
>>> print(datetime.fromtimestamp(t))
2015-04-19 12:20:00
```

注意到timestamp是一个浮点数，它没有时区的概念，而datetime是有时区的。上述转换是在timestamp和本地时间做转换。

本地时间是指当前操作系统设定的时区。例如北京时区是东8区，则本地时间：

```
2015-04-19 12:20:00
```

实际上就是UTC+8:00时区的时间：

```
2015-04-19 12:20:00 UTC+8:00
```

而此刻的格林威治标准时间与北京时间差了8小时，也就是UTC+0:00时区的时间应该是：

```
2015-04-19 04:20:00 UTC+0:00
```

timestamp也可以直接被转换到UTC标准时区的时间：

```
>>> from datetime import datetime
>>> t = 1429417200.0
>>> print(datetime.fromtimestamp(t)) # 本地时间
2015-04-19 12:20:00
>>> print(datetime.utcfromtimestamp(t)) # UTC时间
2015-04-19 04:20:00
```

str转换为datetime

很多时候，用户输入的日期和时间是字符串，要处理日期和时间，首先必须把str转换为datetime。转换方法是通过 `datetime.strptime()` 实现，需要一个日期和时间的格式化字符串：

```
>>> from datetime import datetime
>>> cday = datetime.strptime('2015-6-1 18:19:59', '%Y-%m-%d %H:%M:%S')
>>> print(cday)
2015-06-01 18:19:59
```

字符串 `'%Y-%m-%d %H:%M:%S'` 规定了日期和时间部分的格式。详细的说明请参考[Python文档](#)。

注意转换后的datetime是没有时区信息的。

datetime转换为str

如果已经有了datetime对象，要把它格式化为字符串显示给用户，就需要转换为str，转换方法是通过 `strftime()` 实现的，同样需要一个日期和时间的格式化字符串：

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> print(now.strftime('%a, %b %d %H:%M'))
Mon, May 05 16:28
```

datetime加减

对日期和时间进行加减实际上就是把datetime往后或往前计算，得到新的datetime。加减可以直接用 `+` 和 `-` 运算符，不过需要导入 `timedelta` 这个类：

```
>>> from datetime import datetime, timedelta
```



```
>>> now = datetime.now()
>>> now
datetime.datetime(2015, 5, 18, 16, 57, 3, 540997)
>>> now + timedelta(hours=10)
datetime.datetime(2015, 5, 19, 2, 57, 3, 540997)
>>> now - timedelta(days=1)
datetime.datetime(2015, 5, 17, 16, 57, 3, 540997)
>>> now + timedelta(days=2, hours=12)
datetime.datetime(2015, 5, 21, 4, 57, 3, 540997)
```

可见，使用 `timedelta` 你可以很容易地算出前几天和后几天的时刻。

本地时间转换为UTC时间

本地时间是指系统设定时区的时间，例如北京时间是UTC+8:00时区的时间，而UTC时间指UTC+0:00时区的时间。

一个 `datetime` 类型有一个时区属性 `tzinfo`，但是默认为 `None`，所以无法区分这个 `datetime` 到底是哪个时区，除非强行给 `datetime` 设置一个时区：

```
>>> from datetime import datetime, timedelta, timezone
>>> tz_utc_8 = timezone(timedelta(hours=8)) # 创建时区UTC+8:00
>>> now = datetime.now()
>>> now
datetime.datetime(2015, 5, 18, 17, 2, 10, 871012)
>>> dt = now.replace(tzinfo=tz_utc_8) # 强制设置为UTC+8:00
>>> dt
datetime.datetime(2015, 5, 18, 17, 2, 10, 871012, tzinfo=datetime.timezone(datetime.timedelta(0, 28800)))
```

如果系统时区恰好是UTC+8:00，那么上述代码就是正确的，否则，不能强制设置为UTC+8:00时区。

时区转换

我们可以先通过 `utcnow()` 拿到当前的UTC时间，再转换为任意时区的时间：

```
# 拿到UTC时间，并强制设置时区为UTC+0:00:
>>> utc_dt = datetime.utcnow().replace(tzinfo=timezone.utc)
>>> print(utc_dt)
2015-05-18 09:05:12.377316+00:00
# astimezone()将转换时区为北京时间：
```

```
>>> bj_dt = utc_dt.astimezone(timezone(timedelta(hours=8)))
>>> print(bj_dt)
2015-05-18 17:05:12.377316+08:00
# astimezone()将转换时区为东京时间:
>>> tokyo_dt = utc_dt.astimezone(timezone(timedelta(hours=9)))
>>> print(tokyo_dt)
2015-05-18 18:05:12.377316+09:00
# astimezone()将bj_dt转换时区为东京时间:
>>> tokyo_dt2 = bj_dt.astimezone(timezone(timedelta(hours=9)))
>>> print(tokyo_dt2)
2015-05-18 18:05:12.377316+09:00
```

时区转换的关键在于，拿到一个 `datetime` 时，要获知其正确的时区，然后强制设置时区，作为基准时间。

利用带时区的 `datetime`，通过 `astimezone()` 方法，可以转换到任意时区。

注：不是必须从UTC+0:00时区转换到其他时区，任何带时区的 `datetime` 都可以正确转换，例如上述 `bj_dt` 到 `tokyo_dt` 的转换。

小结

`datetime` 表示的时间需要时区信息才能确定一个特定的时间，否则只能视为本地时间。

如果要存储 `datetime`，最佳方法是将其转换为timestamp再存储，因为timestamp的值与时区完全无关。

参考源码

[use_datetime.py](#)

2、collections

`collections`是Python内建的一个集合模块，提供了许多有用的集合类。

namedtuple

我们知道 `tuple` 可以表示不变集合，例如，一个点的二维坐标就可以表示成：

```
>>> p = (1, 2)
```

但是，看到 `(1, 2)`，很难看出这个 `tuple` 是用来表示一个坐标的。

定义一个class又小题大做了，这时，`namedtuple` 就派上了用场：

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(1, 2)
>>> p.x
1
>>> p.y
2
```

`namedtuple` 是一个函数，它用来创建一个自定义的 `tuple` 对象，并且规定了 `tuple` 元素的个数，并可以用属性而不是索引来引用 `tuple` 的某个元素。

这样一来，我们用 `namedtuple` 可以很方便地定义一种数据类型，它具备`tuple`的不变性，又可以根据属性来引用，使用十分方便。

可以验证创建的 `Point` 对象是 `tuple` 的一种子类：

```
>>> isinstance(p, Point)
True
>>> isinstance(p, tuple)
True
```

类似的，如果要用坐标和半径表示一个圆，也可以用 `namedtuple` 定义：

```
# namedtuple('名称', [属性list]):
Circle = namedtuple('Circle', ['x', 'y', 'r'])
```

deque

使用 `list` 存储数据时，按索引访问元素很快，但是插入和删除元素就很慢了，因为 `list` 是线性存储，数据量大的时候，插入和删除效率很低。

`deque`是为了高效实现插入和删除操作的双向列表，适合用于队列和栈：

```
>>> from collections import deque
>>> q = deque(['a', 'b', 'c'])
>>> q.append('x')
>>> q.appendleft('y')
```

```
>>> q
deque(['y', 'a', 'b', 'c', 'x'])
```

`deque` 除了实现`list`的 `append()` 和 `pop()` 外，还支持 `appendleft()` 和 `popleft()`，这样就可以非常高效地往头部添加或删除元素。

defaultdict

使用 `dict` 时，如果引用的Key不存在，就会抛出 `KeyError`。如果希望key不存在时，返回一个默认值，就可以用 `defaultdict`：

```
>>> from collections import defaultdict
>>> dd = defaultdict(lambda: 'N/A')
>>> dd['key1'] = 'abc'
>>> dd['key1'] # key1存在
'abc'
>>> dd['key2'] # key2不存在，返回默认值
'N/A'
```

注意默认值是调用函数返回的，而函数在创建 `defaultdict` 对象时传入。

除了在Key不存在时返回默认值，`defaultdict` 的其他行为跟 `dict` 是完全一样的。

OrderedDict

使用 `dict` 时，Key是无序的。在对 `dict` 做迭代时，我们无法确定Key的顺序。

如果要保持Key的顺序，可以用 `OrderedDict`：

```
>>> from collections import OrderedDict
>>> d = dict([('a', 1), ('b', 2), ('c', 3)])
>>> d # dict的Key是无序的
{'a': 1, 'c': 3, 'b': 2}
>>> od = OrderedDict([('a', 1), ('b', 2), ('c', 3)])
>>> od # OrderedDict的Key是有序的
OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

注意，`OrderedDict` 的Key会按照插入的顺序排列，不是Key本身排序：

```
>>> od = OrderedDict()
```

```
>>> od['z'] = 1
>>> od['y'] = 2
>>> od['x'] = 3
>>> list(od.keys()) # 按照插入的Key的顺序返回
['z', 'y', 'x']
```

`OrderedDict` 可以实现一个FIFO（先进先出）的dict，当容量超出限制时，先删除最早添加的Key：

```
from collections import OrderedDict

class LastUpdatedOrderedDict(OrderedDict):

    def __init__(self, capacity):
        super(LastUpdatedOrderedDict, self).__init__()
        self._capacity = capacity

    def __setitem__(self, key, value):
        containsKey = 1 if key in self else 0
        if len(self) - containsKey >= self._capacity:
            last = self.popitem(last=False)
            print('remove:', last)
            if containsKey:
                del self[key]
                print('set:', (key, value))
            else:
                print('add:', (key, value))
            OrderedDict.__setitem__(self, key, value)
```

Counter

`Counter` 是一个简单的计数器，例如，统计字符出现的个数：

```
>>> from collections import Counter
>>> c = Counter()
>>> for ch in 'programming':
...     c[ch] = c[ch] + 1
...
>>> c
Counter({'g': 2, 'm': 2, 'r': 2, 'a': 1, 'i': 1, 'o': 1, 'n': 1, 'p': 1})
```

Counter 实际上也是 dict 的一个子类，上面的结果可以看出，字符 'g'、'm'、'r' 各出现了两次，其他字符各出现了一次。

小结

collections 模块提供了一些有用的集合类，可以根据需要选用。

参考源码

[use_collections.py](#)

3、base64

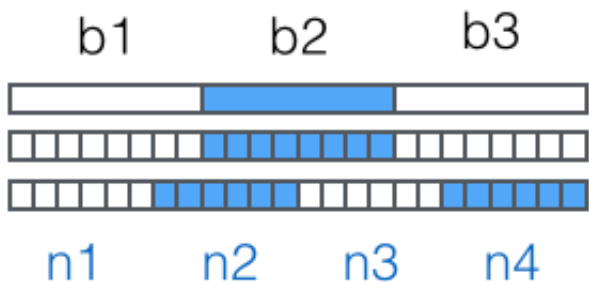
Base64是一种用64个字符来表示任意二进制数据的方法。

用记事本打开 exe、jpg、pdf 这些文件时，我们都会看到一大堆乱码，因为二进制文件包含很多无法显示和打印的字符，所以，如果能让记事本这样的文本处理软件能处理二进制数据，就需要一个二进制到字符串的转换方法。Base64是一种最常见的二进制编码方法。

Base64的原理很简单，首先，准备一个包含64个字符的数组：

```
['A', 'B', 'C', ..., 'a', 'b', 'c', ..., '0', '1', ..., '+', '/']
```

然后，对二进制数据进行处理，每3个字节一组，一共是 $3 \times 8 = 24$ bit，划为4组，每组正好6个 bit：



这样我们得到4个数字作为索引，然后查表，获得相应的4个字符，就是编码后的字符串。

所以，Base64编码会把3字节的二进制数据编码为4字节的文本数据，长度增加33%，好处是编码后的文本数据可以在邮件正文、网页等直接显示。

如果要编码的二进制数据不是3的倍数，最后会剩下1个或2个字节怎么办？Base64用 `\x00` 字节在末尾补足后，再在编码的末尾加上1个或2个 `=` 号，表示补了多少字节，解码的时候，会自动去掉。

Python内置的 `base64` 可以直接进行base64的编解码：

```
>>> import base64
>>> base64.b64encode(b'binary\x00string')
b'YmluYXJ5AHN0cm1uZw=='
>>> base64.b64decode(b'YmluYXJ5AHN0cm1uZw==')
b'binary\x00string'
```

由于标准的Base64编码后可能出现字符 `+` 和 `/`，在URL中就不能直接作为参数，所以又有一种"url safe"的base64编码，其实就是把字符 `+` 和 `/` 分别变成 `-` 和 `_`：

```
>>> base64.b64encode(b'i\x0b\x1d\xfb\xef\xff')
b'abcd++/'
>>> base64.urlsafe_b64encode(b'i\x0b\x1d\xfb\xef\xff')
b'abcd--_'
>>> base64.urlsafe_b64decode('abcd--_')
b'i\x0b\x1d\xfb\xef\xff'
```

还可以自己定义64个字符的排列顺序，这样就可以自定义Base64编码，不过，通常情况下完全没有必要。

Base64是一种通过查表的编码方法，不能用于加密，即使使用自定义的编码表也不行。

Base64适用于小段内容的编码，比如数字证书签名、Cookie的内容等。

由于 `=` 字符也可能出现在Base64编码中，但 `=` 用在URL、Cookie里面会造成歧义，所以，很多Base64编码后会把 `=` 去掉：

```
# 标准Base64:
'abcd' -> 'YWJjZA=='
# 自动去掉=:
'abcd' -> 'YWJjZA'
```

去掉 `=` 后怎么解码呢？因为Base64是把3个字节变为4个字节，所以，Base64编码的长度永远是4的倍数，因此，需要加上 `=` 把Base64字符串的长度变为4的倍数，就可以正常解码了。

小结

Base64是一种任意二进制到文本字符串的编码方法，常用于在URL、Cookie、网页中传输少量

二进制数据。

参考源码

[do_base64.py](#)

4、struct

准确地讲，Python没有专门处理字节的数据类型。但由于 `b'str'` 可以表示字节，所以，字节数组 = 二进制str。而在C语言中，我们可以很方便地用struct、union来处理字节，以及字节和int，float的转换。

在Python中，比方说要把一个32位无符号整数变成字节，也就是4个长度的 `bytes`，你得配合位运算符这么写：

```
>>> n = 10240099
>>> b1 = (n & 0xff000000) >> 24
>>> b2 = (n & 0xff0000) >> 16
>>> b3 = (n & 0xff00) >> 8
>>> b4 = n & 0xff
>>> bs = bytes([b1, b2, b3, b4])
>>> bs
b'\x00\x9c@c'
```

非常麻烦。如果换成浮点数就无能为力了。

好在Python提供了一个 `struct` 模块来解决 `bytes` 和其他二进制数据类型的转换。

`struct` 的 `pack` 函数把任意数据类型变成 `bytes`：

```
>>> import struct
>>> struct.pack('>I', 10240099)
b'\x00\x9c@c'
```

`pack` 的第一个参数是处理指令，`'>I'` 的意思是：

> 表示字节顺序是big-endian，也就是网络序，I 表示4字节无符号整数。

后面的参数个数要和处理指令一致。

`unpack` 把 `bytes` 变成相应的数据类型：


```
>>> struct.unpack('>IH', b'\xf0\xf0\xf0\xf0\x80\x80')
(4042322160, 32896)
```

根据 >IH 的说明，后面的 bytes 依次变为 I：4字节无符号整数和 H：2字节无符号整数。

所以，尽管Python不适合编写底层操作字节流的代码，但在对性能要求不高的地方，利用 struct 就方便多了。

struct 模块定义的数据类型可以参考Python官方文档：

<https://docs.python.org/3/library/struct.html#format-characters>

Windows的位图文件（.bmp）是一种非常简单的文件格式，我们用 struct 分析一下。

首先找一个bmp文件，没有的话用“画图”画一个。

读入前30个字节来分析：

```
>>> s = b'\x42\x4d\x38\x8c\x0a\x00\x00\x00\x00\x00\x36\x00\x00\x00\x28\x00\x00\x00\x80\x02\x00\x00\x68\x01\x00\x00\x01\x00\x18\x00'
```

BMP格式采用小端方式存储数据，文件头的结构按顺序如下：

两个字节：'BM' 表示Windows位图，'BA' 表示OS/2位图； 一个4字节整数：表示位图大小； 一个4字节整数：保留位，始终为0； 一个4字节整数：实际图像的偏移量； 一个4字节整数：Header的字节数； 一个4字节整数：图像宽度； 一个4字节整数：图像高度； 一个2字节整数：始终为1； 一个2字节整数：颜色数。

所以，组合起来用 unpack 读取：

```
>>> struct.unpack('<ccIIIIIIHH', s)
(b'B', b'M', 691256, 0, 54, 40, 640, 360, 1, 24)
```

结果显示，b'B'、b'M' 说明是Windows位图，位图大小为640x360，颜色数为24。

参考源码

[check_bmp.py](#)

5、hashlib

摘要算法简介

Python的hashlib提供了常见的摘要算法，如MD5，SHA1等等。

什么是摘要算法呢？摘要算法又称哈希算法、散列算法。它通过一个函数，把任意长度的数据转换为一个长度固定的数据串（通常用16进制的字符串表示）。

举个例子，你写了一篇文章，内容是一个字符串 'how to use python hashlib - by Michael'，并附上这篇文章的摘要 '2d73d4f15c0db7f5ecb321b6a65e5d6d'。如果有人篡改了你的文章，并发表为 'how to use python hashlib - by Bob'，你可以一下子指出Bob篡改了你的文章，因为根据 'how to use python hashlib - by Bob' 计算出的摘要不同于原始文章的摘要。

可见，摘要算法就是通过摘要函数 $f()$ 对任意长度的数据 `data` 计算出固定长度的摘要 `digest`，目的是为了发现原始数据是否被人篡改过。

摘要算法之所以能指出数据是否被篡改过，就是因为摘要函数是一个单向函数，计算 $f(data)$ 很容易，但通过 `digest` 反推 `data` 却非常困难。而且，对原始数据做一个bit的修改，都会导致计算出的摘要完全不同。

我们以常见的摘要算法MD5为例，计算出一个字符串的MD5值：

```
import hashlib

md5 = hashlib.md5()
md5.update('how to use md5 in python hashlib?'.encode('utf-8'))
print(md5.hexdigest())
```

计算结果如下：

```
d26a53750bc40b38b65a520292f69306
```

如果数据量很大，可以分块多次调用 `update()`，最后计算的结果是一样的：

```
import hashlib

md5 = hashlib.md5()
md5.update('how to use md5 in '.encode('utf-8'))
md5.update('python hashlib?'.encode('utf-8'))
```

```
print(md5.hexdigest())
```

试试改动一个字母，看看计算的结果是否完全不同。

MD5是最常见的摘要算法，速度很快，生成结果是固定的128 bit字节，通常用一个32位的16进制字符串表示。

另一种常见的摘要算法是SHA1，调用SHA1和调用MD5完全类似：

```
import hashlib

sha1 = hashlib.sha1()
sha1.update('how to use sha1 in '.encode('utf-8'))
sha1.update('python hashlib?'.encode('utf-8'))
print(sha1.hexdigest())
```

SHA1的结果是160 bit字节，通常用一个40位的16进制字符串表示。

比SHA1更安全的算法是SHA256和SHA512，不过越安全的算法不仅越慢，而且摘要长度更长。

有没有可能两个不同的数据通过某个摘要算法得到了相同的摘要？完全有可能，因为任何摘要算法都是把无限多的数据集映射到一个有限的集合中。这种情况称为碰撞，比如Bob试图根据你的摘要反推出一篇文章 'how to learn hashlib in python - by Bob'，并且这篇文章的摘要恰好和你的文章完全一致，这种情况也并非不可能出现，但是非常非常困难。

摘要算法应用

摘要算法能应用到什么地方？举个常用例子：

任何允许用户登录的网站都会存储用户登录的用户名和口令。如何存储用户名和口令呢？方法是存到数据库表中：

name	password
michael	123456
bob	abc999
alice	alice2008

如果以明文保存用户口令，如果数据库泄露，所有用户的口令就落入黑客的手里。此外，网站运维人员是可以访问数据库的，也就是能获取到所有用户的口令。

正确的保存口令的方式是不存储用户的明文口令，而是存储用户口令的摘要，比如MD5：

username	password
michael	e10adc3949ba59abbe56e057f20f883e
bob	878ef96e86145580c38c87f0410ad153
alice	99b1c2188db85afee403b1536010c2c9

当用户登录时，首先计算用户输入的明文口令的MD5，然后和数据库存储的MD5对比，如果一致，说明口令输入正确，如果不一致，口令肯定错误。

练习

根据用户输入的口令，计算出存储在数据库中的MD5口令：

```
def calc_md5(password):
    pass
```

存储MD5的好处是即使运维人员能访问数据库，也无法获知用户的明文口令。

设计一个验证用户登录的函数，根据用户输入的口令是否正确，返回True或False：

```
# -*- coding: utf-8 -*-
db = {
    'michael': 'e10adc3949ba59abbe56e057f20f883e',
    'bob': '878ef96e86145580c38c87f0410ad153',
    'alice': '99b1c2188db85afee403b1536010c2c9'
}

# 测试:
assert login('michael', '123456')
assert login('bob', 'abc999')
assert login('alice', 'alice2008')
assert not login('michael', '1234567')
assert not login('bob', '123456')
assert not login('alice', 'Alice2008')
print('ok')
```

采用MD5存储口令是否就一定安全呢？也不一定。假设你是一个黑客，已经拿到了存储MD5口令的数据库，如何通过MD5反推用户的明文口令呢？暴力破解费事费力，真正的黑客不会这么干。

考虑这么个情况，很多用户喜欢用 123456，888888，password 这些简单的口令，于是，黑客可以事先计算出这些常用口令的MD5值，得到一个反推表：

```
'e10adc3949ba59abbe56e057f20f883e': '123456'
'21218cca77804d2ba1922c33e0151105': '888888'
'5f4dcc3b5aa765d61d8327deb882cf99': 'password'
```

这样，无需破解，只需要对比数据库的MD5，黑客就获得了使用常用口令的用户账号。

对于用户来讲，当然不要使用过于简单的口令。但是，我们能否在程序设计上对简单口令加强保护呢？

由于常用口令的MD5值很容易被计算出来，所以，要确保存储的用户口令不是那些已经被计算出来的常用口令的MD5，这一方法通过对原始口令加一个复杂字符串来实现，俗称“加盐”：

```
def calc_md5(password):
    return get_md5(password + 'the-Salt')
```

经过Salt处理的MD5口令，只要Salt不被黑客知道，即使用户输入简单口令，也很难通过MD5反推明文口令。

但是如果有两个用户都使用了相同的简单口令比如 123456，在数据库中，将存储两条相同的MD5值，这说明这两个用户的口令是一样的。有没有办法让使用相同口令的用户存储不同的MD5呢？

如果假定用户无法修改登录名，就可以通过把登录名作为Salt的一部分来计算MD5，从而实现相同口令的用户也存储不同的MD5。

练习

根据用户输入的登录名和口令模拟用户注册，计算更安全的MD5：

```
db = {}

def register(username, password):
    db[username] = get_md5(password + username + 'the-Salt')
```

然后，根据修改后的MD5算法实现用户登录的验证：

```
# -*- coding: utf-8 -*-
import hashlib, random

def get_md5(s):
    return hashlib.md5(s.encode('utf-8')).hexdigest()

class User(object):
    def __init__(self, username, password):
        self.username = username
        self.salt = ''.join([chr(random.randint(48, 122)) for i in range(20)])
        self.password = get_md5(password + self.salt)

db = {
    'michael': User('michael', '123456'),
    'bob': User('bob', 'abc999'),
    'alice': User('alice', 'alice2008')
}

# 测试:
assert login('michael', '123456')
assert login('bob', 'abc999')
assert login('alice', 'alice2008')
assert not login('michael', '1234567')
assert not login('bob', '123456')
assert not login('alice', 'Alice2008')
print('ok')
```

小结

摘要算法在很多地方都有广泛的应用。要注意摘要算法不是加密算法，不能用于加密（因为无法通过摘要反推明文），只能用于防篡改，但是它的单向计算特性决定了可以在不存储明文口令的情况下验证用户口令。

参考源码

[use_hashlib.py](#)

6、hmac

通过哈希算法，我们可以验证一段数据是否有效，方法就是对比该数据的哈希值，例如，判断用户口令是否正确，我们用保存在数据库中的 `password_md5` 对比计算 `md5(password)` 的结果，如果一致，用户输入的口令就是正确的。

为了防止黑客通过彩虹表根据哈希值反推原始口令，在计算哈希的时候，不能仅针对原始输入计算，需要增加一个salt来使得相同的输入也能得到不同的哈希，这样，大大增加了黑客破解的难度。

如果salt是我们自己随机生成的，通常我们计算MD5时采用 `md5(message + salt)`。但实际上，把salt看做一个“口令”，加salt的哈希就是：计算一段message的哈希时，根据不通口令计算出不同的哈希。要验证哈希值，必须同时提供正确的口令。

这实际上就是Hmac算法：Keyed-Hashing for Message Authentication。它通过一个标准算法，在计算哈希的过程中，把key混入计算过程中。

和我们自定义的加salt算法不同，Hmac算法针对所有哈希算法都通用，无论是MD5还是SHA-1。采用Hmac替代我们自己的salt算法，可以使程序算法更标准化，也更安全。

Python自带的hmac模块实现了标准的Hmac算法。我们来看看如何使用hmac实现带key的哈希。

我们首先需要准备待计算的原始消息message，随机key，哈希算法，这里采用MD5，使用hmac的代码如下：

```
>>> import hmac
>>> message = b'Hello, world!'
>>> key = b'secret'
>>> h = hmac.new(key, message, digestmod='MD5')
>>> # 如果消息很长，可以多次调用h.update(msg)
>>> h.hexdigest()
'fa4ee7d173f2d97ee79022d1a7355bcf'
```

可见使用hmac和普通hash算法非常类似。hmac输出的长度和原始哈希算法的长度一致。需要注意传入的key和message都是 `bytes` 类型，`str` 类型需要首先编码为 `bytes`。

练习

将上一节的salt改为标准的hmac算法，验证用户口令：

```
# -*- coding: utf-8 -*-
import hmac, random

def hmac_md5(key, s):
```

```

    return hmac.new(key.encode('utf-8'), s.encode('utf-8'), 'MD5').hexdigest()

class User(object):
    def __init__(self, username, password):
        self.username = username
        self.key = ''.join([chr(random.randint(48, 122)) for i in range(20)])
        self.password = hmac_md5(self.key, password)

db = {
    'michael': User('michael', '123456'),
    'bob': User('bob', 'abc999'),
    'alice': User('alice', 'alice2008')
}

# 测试:
assert login('michael', '123456')
assert login('bob', 'abc999')
assert login('alice', 'alice2008')
assert not login('michael', '1234567')
assert not login('bob', '123456')
assert not login('alice', 'Alice2008')
print('ok')
```

小结

Python内置的hmac模块实现了标准的Hmac算法，它利用一个key对message计算“杂凑”后的hash，使用hmac算法比标准hash算法更安全，因为针对相同的message，不同的key会产生不同的hash。

7、itertools

Python的内建模块 `itertools` 提供了非常有用的用于操作迭代对象的函数。

首先，我们看看 `itertools` 提供的几个“无限”迭代器：

```

>>> import itertools
>>> natuals = itertools.count(1)
>>> for n in natuals:
...     print(n)
...
1
2
```



```
3
...
```

因为 `count()` 会创建一个无限的迭代器，所以上述代码会打印出自然数序列，根本停不下来，只能按 `Ctrl+C` 退出。

`cycle()` 会把传入的一个序列无限重复下去：

```
>>> import itertools
>>> cs = itertools.cycle('ABC') # 注意字符串也是序列的一种
>>> for c in cs:
...     print(c)
...
'A'
'B'
'C'
'A'
'B'
'C'
...
```

同样停不下来。

`repeat()` 负责把一个元素无限重复下去，不过如果提供第二个参数就可以限定重复次数：

```
>>> ns = itertools.repeat('A', 3)
>>> for n in ns:
...     print(n)
...
A
A
A
```

无限序列只有在 `for` 迭代时才会无限地迭代下去，如果只是创建了一个迭代对象，它不会事先把无限个元素生成出来，事实上也不可能在内存中创建无限多个元素。

无限序列虽然可以无限迭代下去，但是通常我们会通过 `takewhile()` 等函数根据条件判断来截取出一个有限的序列：

```
>>> naturals = itertools.count(1)
```

```
>>> ns = itertools.takewhile(lambda x: x <= 10, naturals)
>>> list(ns)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

`itertools` 提供的几个迭代器操作函数更加有用：

chain()

`chain()` 可以把一组迭代对象串联起来，形成一个更大的迭代器：

```
>>> for c in itertools.chain('ABC', 'XYZ'):
...     print(c)
# 迭代效果: 'A' 'B' 'C' 'X' 'Y' 'Z'
```

groupby()

`groupby()` 把迭代器中相邻的重复元素挑出来放在一起：

```
>>> for key, group in itertools.groupby('AAABBBCCAAA'):
...     print(key, list(group))
...
A ['A', 'A', 'A']
B ['B', 'B', 'B']
C ['C', 'C']
A ['A', 'A', 'A']
```

实际上挑选规则是通过函数完成的，只要作用于函数的两个元素返回的值相等，这两个元素就被认为是在一组的，而函数返回值作为组的key。如果我们要忽略大小写分组，就可以让元素 'A' 和 'a' 都返回相同的key：

```
>>> for key, group in itertools.groupby('AaaBBbcAAa', lambda c: c.upper()):
...     print(key, list(group))
...
A ['A', 'a', 'a']
B ['B', 'B', 'b']
C ['c', 'C']
A ['A', 'A', 'a']
```

练习

计算圆周率可以根据公式：

利用Python提供的itertools模块，我们来计算这个序列的前N项和：

```
# -*- coding: utf-8 -*-
import itertools

# 测试：
print(pi(10))
print(pi(100))
print(pi(1000))
print(pi(10000))
assert 3.04
```

小结

itertools 模块提供的全部是处理迭代功能的函数，它们的返回值不是list，而是 `Iterator`，只有用 `for` 循环迭代的时候才真正计算。

参考源码

[use_itertools.py](#)

8、contextlib

在Python中，读写文件这样的资源要特别注意，必须在使用完毕后正确关闭它们。正确关闭文件资源的一个方法是使用 `try...finally`：

```
try:
    f = open('/path/to/file', 'r')
    f.read()
finally:
    if f:
        f.close()
```

写 `try...finally` 非常繁琐。Python的 `with` 语句允许我们非常方便地使用资源，而不必担心资源没有关闭，所以上面的代码可以简化为：

```
with open('/path/to/file', 'r') as f:
    f.read()
```

并不是只有 `open()` 函数返回的fp对象才能使用 `with` 语句。实际上，任何对象，只要正确实现了上下文管理，就可以用于 `with` 语句。

实现上下文管理是通过 `__enter__` 和 `__exit__` 这两个方法实现的。例如，下面的class实现了这两个方法：

```
class Query(object):

    def __init__(self, name):
        self.name = name

    def __enter__(self):
        print('Begin')
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        if exc_type:
            print('Error')
        else:
            print('End')

    def query(self):
        print('Query info about %s...' % self.name)
```

这样我们就可以把自己写的资源对象用于 `with` 语句：

```
with Query('Bob') as q:
    q.query()
```

@contextmanager

编写 `__enter__` 和 `__exit__` 仍然很繁琐，因此Python的标准库 `contextlib` 提供了更简单的写法，上面的代码可以改写如下：

```
from contextlib import contextmanager
```

```
class Query(object):

    def __init__(self, name):
        self.name = name

    def query(self):
        print('Query info about %s...' % self.name)

@contextmanager
def create_query(name):
    print('Begin')
    q = Query(name)
    yield q
    print('End')
```

@contextmanager 这个decorator接受一个generator，用 yield 语句把 with ... as var 把变量输出出去，然后，with 语句就可以正常地工作了：

```
with create_query('Bob') as q:
    q.query()
```

很多时候，我们希望在某段代码执行前后自动执行特定代码，也可以用 @contextmanager 实现。例如：

```
@contextmanager
def tag(name):
    print("<%s>" % name)
    yield
    print("</%s>" % name)

with tag("h1"):
    print("hello")
    print("world")
```

上述代码执行结果为：

```
<h1>
hello
world
</h1>
```

代码的执行顺序是：

1. `with` 语句首先执行 `yield` 之前的语句，因此打印出 `<h1>` ；
2. `yield` 调用会执行 `with` 语句内部的所有语句，因此打印出 `hello` 和 `world` ；
3. 最后执行 `yield` 之后的语句，打印出 `</h1>` 。

因此，`@contextmanager` 让我们通过编写generator来简化上下文管理。

@closing

如果一个对象没有实现上下文，我们就不能把它用于 `with` 语句。这个时候，可以用 `closing()` 来把该对象变为上下文对象。例如，用 `with` 语句使用 `urlopen()` ：

```
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('https://www.python.org')) as page:
    for line in page:
        print(line)
```

`closing` 也是一个经过`@contextmanager`装饰的generator，这个generator编写起来其实非常简单：

```
@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

它的作用就是把任意对象变为上下文对象，并支持 `with` 语句。

`@contextlib` 还有一些其他decorator，便于我们编写更简洁的代码。

9、urllib

`urllib`提供了一系列用于操作URL的功能。

Get

urllib的 request 模块可以非常方便地抓取URL内容，也就是发送一个GET请求到指定的页面，然后返回HTTP的响应：

例如，对豆瓣的一个URL `https://api.douban.com/v2/book/2129650` 进行抓取，并返回响应：

```
from urllib import request

with request.urlopen('https://api.douban.com/v2/book/2129650') as f:
    data = f.read()
    print('Status:', f.status, f.reason)
    for k, v in f.getheaders():
        print('%s: %s' % (k, v))
    print('Data:', data.decode('utf-8'))
```

可以看到HTTP响应的头和JSON数据：

```
Status: 200 OK
Server: nginx
Date: Tue, 26 May 2015 10:02:27 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 2049
Connection: close
Expires: Sun, 1 Jan 2006 01:00:00 GMT
Pragma: no-cache
Cache-Control: must-revalidate, no-cache, private
X-DAE-Node: pid11
Data: {"rating":{"max":10,"numRaters":16,"average":"7.4","min":0},"subtitle":"","author":["廖雪峰编著"],"pubdate":"2007-6",...}
```

如果我们要想模拟浏览器发送GET请求，就需要使用 Request 对象，通过往 Request 对象添加HTTP头，我们就可以把请求伪装成浏览器。例如，模拟iPhone 6去请求豆瓣首页：

```
from urllib import request

req = request.Request('http://www.douban.com/')
req.add_header('User-Agent', 'Mozilla/6.0 (iPhone; CPU iPhone OS 8_0 like Mac OS X) AppleWebKit/536.26 (KHTML, like Gecko) Version/8.0 Mobile/10A5376e Safari/8536.25')
with request.urlopen(req) as f:
    print('Status:', f.status, f.reason)
    for k, v in f.getheaders():
        print('%s: %s' % (k, v))
```

```
print('Data:', f.read().decode('utf-8'))
```

这样豆瓣会返回适合iPhone的移动版网页：

```
...
<meta name="viewport" content="width=device-width, user-scalable=no, initial-scale=1.0, minimum-scale=1.0, maximum-scale=1.0">
<meta name="format-detection" content="telephone=no">
<link rel="apple-touch-icon" sizes="57x57" href="http://img4.douban.com/pics/cardkit/launcher/57.png" />
...
```

Post

如果要以POST发送一个请求，只需要把参数 `data` 以bytes形式传入。

我们模拟一个微博登录，先读取登录的邮箱和口令，然后按照weibo.cn的登录页的格式以 `username=xxx&password=xxx` 的编码传入：

```
from urllib import request, parse

print('Login to weibo.cn...')
email = input('Email: ')
passwd = input('Password: ')
login_data = parse.urlencode([
    ('username', email),
    ('password', passwd),
    ('entry', 'mweibo'),
    ('client_id', ''),
    ('savestate', '1'),
    ('ec', ''),
    ('pagerefer', 'https://passport.weibo.cn/signin/welcome?entry=mweibo&r=http%3A%2F%2Fm.weibo.cn%2F')
])

req = request.Request('https://passport.weibo.cn/sso/login')
req.add_header('Origin', 'https://passport.weibo.cn')
req.add_header('User-Agent', 'Mozilla/6.0 (iPhone; CPU iPhone OS 8_0 like Mac OS X) AppleWebKit/536.26 (KHTML, like Gecko) Version/8.0 Mobile/10A5376e Safari/8536.25')
req.add_header('Referer', 'https://passport.weibo.cn/signin/login?entry=mweibo&res=wel&wm=3349&r=http%3A%2F%2Fm.weibo.cn%2F')
```



```
with request.urlopen(req, data=login_data.encode('utf-8')) as f:
    print('Status:', f.status, f.reason)
    for k, v in f.getheaders():
        print('%s: %s' % (k, v))
    print('Data:', f.read().decode('utf-8'))
```

如果登录成功，我们获得的响应如下：

```
Status: 200 OK
Server: nginx/1.2.0
...
Set-Cookie: SSOLoginState=1432620126; path=/; domain=weibo.cn
...
Data: {"retcode":20000000,"msg":"","data":{"...","uid":"1658384301"}}
```

如果登录失败，我们获得的响应如下：

```
...
Data: {"retcode":50011015,"msg":"\u7528\u6237\u540d\u6216\u5bc6\u7801\u9519\u8bef",
"data":{"username":"example@python.org","errline":536}}
```

Handler

如果还需要更复杂的控制，比如通过一个Proxy去访问网站，我们需要利用 ProxyHandler 来处理，示例代码如下：

```
proxy_handler = urllib.request.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')
opener = urllib.request.build_opener(proxy_handler, proxy_auth_handler)
with opener.open('http://www.example.com/login.html') as f:
    pass
```

小结

urllib提供的功能就是利用程序去执行各种HTTP请求。如果要模拟浏览器完成特定功能，需要把请求伪装成浏览器。伪装的方法是先监控浏览器发出的请求，再根据浏览器的请求头来伪

装， `User-Agent` 头就是用来标识浏览器的。

练习

利用urllib读取JSON，然后将JSON解析为Python对象：

```
# -*- coding: utf-8 -*-
from urllib import request

# 测试
URL = 'https://query.yahooapis.com/v1/public/yql?q=select%20*%20from%20weather.forecast%20where%20woeid%20%3D%202151330&format=json'
data = fetch_data(URL)
print(data)
assert data['query']['results']['channel']['location']['city'] == 'Beijing'
print('ok')
```

参考源码

[use_urllib.py](#)

10、XML

XML虽然比JSON复杂，在Web中应用也不如以前多了，不过仍有很多地方在用，所以，有必要了解如何操作XML。

DOM vs SAX

操作XML有两种方法：DOM和SAX。DOM会把整个XML读入内存，解析为树，因此占用内存大，解析慢，优点是可以任意遍历树的节点。SAX是流模式，边读边解析，占用内存小，解析快，缺点是我们需要自己处理事件。

正常情况下，优先考虑SAX，因为DOM实在太占内存。

在Python中使用SAX解析XML非常简洁，通常我们关心的事件是 `start_element`，`end_element` 和 `char_data`，准备好这3个函数，然后就可以解析xml了。

举个例子，当SAX解析器读到一个节点时：

```
<a href="/">python</a>
```

会产生3个事件：

1. start_element事件，在读取 `` 时；
2. char_data事件，在读取 `python` 时；
3. end_element事件，在读取 `` 时。

用代码实验一下：

```
from xml.parsers.expat import ParserCreate

class DefaultSaxHandler(object):
    def start_element(self, name, attrs):
        print('sax:start_element: %s, attrs: %s' % (name, str(attrs)))

    def end_element(self, name):
        print('sax:end_element: %s' % name)

    def char_data(self, text):
        print('sax:char_data: %s' % text)

xml = r'''<?xml version="1.0"?>
<ol>
  <li><a href="/python">Python</a></li>
  <li><a href="/ruby">Ruby</a></li>
</ol>
'''

handler = DefaultSaxHandler()
parser = ParserCreate()
parser.StartElementHandler = handler.start_element
parser.EndElementHandler = handler.end_element
parser.CharacterDataHandler = handler.char_data
parser.Parse(xml)
```

需要注意的是读取一大段字符串时，`CharacterDataHandler` 可能被多次调用，所以需要自己保存起来，在 `EndElementHandler` 里面再合并。

除了解析XML外，如何生成XML呢？99%的情况下需要生成的XML结构都是非常简单的，因此，最简单也是最有效的生成XML的方法是拼接字符串：

```
L = []
```

```
L.append(r'<?xml version="1.0"?>')
L.append(r'<root>')
L.append(encode('some & data'))
L.append(r'</root>')
return ''.join(L)
```

如果要生成复杂的XML呢？建议你不要用XML，改成JSON。

小结

解析XML时，注意找出自己感兴趣的节点，响应事件时，把节点数据保存起来。解析完毕后，就可以处理数据。

练习

请利用SAX编写程序解析Yahoo的XML格式的天气预报，获取天气预报：

https://query.yahooapis.com/v1/public/yql?q=select%20*%20from%20weather.forecast%20where%20woeid%20%3D%202151330&format=xml

参数 `woeid` 是城市代码，要查询某个城市代码，可以在weather.yahoo.com搜索城市，浏览器地址栏的URL就包含城市代码。

```
# -*- coding:utf-8 -*-

from xml.parsers.expat import ParserCreate
from urllib import request

# 测试:
URL = 'https://query.yahooapis.com/v1/public/yql?q=select%20*%20from%20weather.forecast%20where%20woeid%20%3D%202151330&format=xml'

with request.urlopen(URL, timeout=4) as f:
    data = f.read()

result = parseXml(data.decode('utf-8'))
assert result['city'] == 'Beijing'
```

参考源码

use_sax.py

11、HTMLParser

如果我们要编写一个搜索引擎，第一步是用爬虫把目标网站的页面抓下来，第二步就是解析该HTML页面，看看里面的内容到底是新闻、图片还是视频。

假设第一步已经完成了，第二步应该如何解析HTML呢？

HTML本质上是XML的子集，但是HTML的语法没有XML那么严格，所以不能用标准的DOM或SAX来解析HTML。

好在Python提供了HTMLParser来非常方便地解析HTML，只需简单几行代码：

```
from html.parser import HTMLParser
from html.entities import name2codepoint

class MyHTMLParser(HTMLParser):

    def handle_starttag(self, tag, attrs):
        print('<%s>' % tag)

    def handle_endtag(self, tag):
        print('</%s>' % tag)

    def handle_startendtag(self, tag, attrs):
        print('<%s/>' % tag)

    def handle_data(self, data):
        print(data)

    def handle_comment(self, data):
        print('<!--', data, '-->')

    def handle_entityref(self, name):
        print('&%s;' % name)

    def handle_charref(self, name):
        print('&#s;' % name)

parser = MyHTMLParser()
parser.feed(''<html>
<head></head>
<body>
<!-- test html parser -->
<p>Some <a href="#">html</a> HTML&nbsp;tutorial...<br>END</p>')
```

```
</body></html>''')
```

`feed()` 方法可以多次调用，也就是不一定一次把整个HTML字符串都塞进去，可以一部分一部分塞进去。

特殊字符有两种，一种是英文表示的 ` `，一种是数字表示的 `Ӓ`，这两种字符都可以通过Parser解析出来。

小结

利用HTMLParser，可以把网页中的文本、图像等解析出来。

练习

找一个网页，例如<https://www.python.org/events/python-events/>，用浏览器查看源码并复制，然后尝试解析一下HTML，输出Python官网发布的会议时间、名称和地点。

参考源码

[use_htmlparser.py](#)

六、常用第三方模块

除了内建的模块外，Python还有大量的第三方模块。

基本上，所有的第三方模块都会在[PyPI - the Python Package Index](#)上注册，只要找到对应的模块名字，即可用pip安装。

此外，在[安装第三方模块](#)一节中，我们强烈推荐安装Anaconda，安装后，数十个常用的第三方模块就已经就绪，不用pip手动安装。

本章介绍常用的第三方模块。

1、Pillow

PIL：Python Imaging Library，已经是Python平台事实上的图像处理标准库了。PIL功能非常强大，但API却非常简单易用。

由于PIL仅支持到Python 2.7，加上年久失修，于是一群志愿者在PIL的基础上创建了兼容的版本，名字叫Pillow，支持最新Python 3.x，又加入了许多新特性，因此，我们可以直接安装使用Pillow。

安装Pillow

如果安装了Anaconda，Pillow就已经可用了。否则，需要在命令行下通过pip安装：

```
$ pip install pillow
```

如果遇到 Permission denied 安装失败，请加上 `sudo` 重试。

操作图像

来看看最常见的图像缩放操作，只需三四行代码：

```
from PIL import Image

# 打开一个jpg图像文件，注意是当前路径：
im = Image.open('test.jpg')
# 获得图像尺寸：
w, h = im.size
print('Original image size: %s%s' % (w, h))
# 缩放到50%:
im.thumbnail((w//2, h//2))
print('Resize image to: %s%s' % (w//2, h//2))
# 把缩放后的图像用jpeg格式保存：
im.save('thumbnail.jpg', 'jpeg')
```

其他功能如切片、旋转、滤镜、输出文字、调色板等一应俱全。

比如，模糊效果也只需几行代码：

```
from PIL import Image, ImageFilter

# 打开一个jpg图像文件，注意是当前路径：
im = Image.open('test.jpg')
# 应用模糊滤镜：
im2 = im.filter(ImageFilter.BLUR)
im2.save('blur.jpg', 'jpeg')
```

效果如下：



PIL的 `ImageDraw` 提供了一系列绘图方法，让我们可以直接绘图。比如要生成字母验证码图片：

```
from PIL import Image, ImageDraw, ImageFont, ImageFilter

import random

# 随机字母:
def rndChar():
    return chr(random.randint(65, 90))

# 随机颜色1:
def rndColor():
    return (random.randint(64, 255), random.randint(64, 255), random.randint(64, 255))

# 随机颜色2:
def rndColor2():
    return (random.randint(32, 127), random.randint(32, 127), random.randint(32, 127))

# 240 x 60:
width = 60 * 4
height = 60
image = Image.new('RGB', (width, height), (255, 255, 255))
# 创建Font对象:
font = ImageFont.truetype('Arial.ttf', 36)
# 创建Draw对象:
draw = ImageDraw.Draw(image)
# 填充每个像素:
for x in range(width):
    for y in range(height):
```



```
draw.point((x, y), fill=rndColor())
# 输出文字:
for t in range(4):
    draw.text((60 * t + 10, 10), rndChar(), font=font, fill=rndColor2())
# 模糊:
image = image.filter(ImageFilter.BLUR)
image.save('code.jpg', 'jpeg')
```

我们用随机颜色填充背景，再画上文字，最后对图像进行模糊，得到验证码图片如下：



如果运行的时候报错：

```
IOError: cannot open resource
```

这是因为PIL无法定位到字体文件的位置，可以根据操作系统提供绝对路径，比如：

```
'/Library/Fonts/Arial.ttf'
```

要详细了解PIL的强大功能，请参考Pillow官方文档：

<https://pillow.readthedocs.org/>

小结

PIL提供了操作图像的强大功能，可以通过简单的代码完成复杂的图像处理。

参考源码

https://github.com/michaelliao/learn-python3/blob/master/samples/packages/pil/use_pil_resize.py

https://github.com/michaelliao/learn-python3/blob/master/samples/packages/pil/use_pil_blur.py

https://github.com/michaelliao/learn-python3/blob/master/samples/packages/pil/use_pil_draw.py

2、requests

我们已经讲解了Python内置的urllib模块，用于访问网络资源。但是，它用起来比较麻烦，而且，缺少很多实用的高级功能。

更好的方案是使用requests。它是一个Python第三方库，处理URL资源特别方便。

安装requests

如果安装了Anaconda，requests就已经可用了。否则，需要在命令行下通过pip安装：

```
$ pip install requests
```

如果遇到Permission denied安装失败，请加上sudo重试。

使用requests

要通过GET访问一个页面，只需要几行代码：

```
>>> import requests
>>> r = requests.get('https://www.douban.com/') # 豆瓣首页
>>> r.status_code
200
>>> r.text
r.text
'<!DOCTYPE HTML>\n<html>\n<head>\n<meta name="description" content="提供图书、电影、\n音乐唱片的推荐、评论和...'
```

对于带参数的URL，传入一个dict作为 params 参数：

```
>>> r = requests.get('https://www.douban.com/search', params={'q': 'python', 'cat': '1001'})
>>> r.url # 实际请求的URL
'https://www.douban.com/search?q=python&cat=1001'
```

requests自动检测编码，可以使用 encoding 属性查看：

```
>>> r.encoding
'utf-8'
```

无论响应是文本还是二进制内容，我们都可以用 `content` 属性获得 `bytes` 对象：

```
>>> r.content
b'<!DOCTYPE html>\n<html>\n<head>\n<meta http-equiv="Content-Type" content="text/html; charset=utf-8">\n...'
```

`requests`的方便之处还在于，对于特定类型的响应，例如JSON，可以直接获取：

```
>>> r = requests.get('https://query.yahooapis.com/v1/public/yql?q=select%20*%20from%20weather.forecast%20where%20woeid%20%3D%202151330&format=json')
>>> r.json()
{'query': {'count': 1, 'created': '2017-11-17T07:14:12Z', ...}}
```

需要传入HTTP Header时，我们传入一个dict作为 `headers` 参数：

```
>>> r = requests.get('https://www.douban.com/', headers={'User-Agent': 'Mozilla/5.0 (iPhone; CPU iPhone OS 11_0 like Mac OS X) AppleWebKit'})
>>> r.text
'<!DOCTYPE html>\n<html>\n<head>\n<meta charset="UTF-8">\n <title>豆瓣(手机版)</title>...'
```

要发送POST请求，只需要把 `get()` 方法变成 `post()`，然后传入 `data` 参数作为POST请求的数据：

```
>>> r = requests.post('https://accounts.douban.com/login', data={'form_email': 'abc@example.com', 'form_password': '123456'})
```

`requests`默认使用 `application/x-www-form-urlencoded` 对POST数据编码。如果要传递JSON数据，可以直接传入`json`参数：

```
params = {'key': 'value'}
r = requests.post(url, json=params) # 内部自动序列化为JSON
```

类似的，上传文件需要更复杂的编码格式，但是requests把它简化成 `files` 参数：

```
>>> upload_files = {'file': open('report.xls', 'rb')}
>>> r = requests.post(url, files=upload_files)
```

在读取文件时，注意务必使用 `'rb'` 即二进制模式读取，这样获取的 `bytes` 长度才是文件的长度。

把 `post()` 方法替换为 `put()`，`delete()` 等，就可以以PUT或DELETE方式请求资源。

除了能轻松获取响应内容外，requests对获取HTTP响应的其他信息也非常简单。例如，获取响应头：

```
>>> r.headers
{'Content-Type': 'text/html; charset=utf-8', 'Transfer-Encoding': 'chunked', 'Content-Encoding': 'gzip', ...}
>>> r.headers['Content-Type']
'text/html; charset=utf-8'
```

requests对Cookie做了特殊处理，使得我们不必解析Cookie就可以轻松获取指定的Cookie：

```
>>> r.cookies['ts']
'example_cookie_12345'
```

要在请求中传入Cookie，只需准备一个dict传入 `cookies` 参数：

```
>>> cs = {'token': '12345', 'status': 'working'}
>>> r = requests.get(url, cookies=cs)
```

最后，要指定超时，传入以秒为单位的`timeout`参数：

```
>>> r = requests.get(url, timeout=2.5) # 2.5秒后超时
```

小结

用requests获取URL资源，就是这么简单！

3、chardet

字符串编码一直是令人非常头疼的问题，尤其是在处理一些不规范的第三方网页的时候。虽然Python提供了Unicode表示的 `str` 和 `bytes` 两种数据类型，并且可以通过 `encode()` 和 `decode()` 方法转换，但是，在不知道编码的情况下，对 `bytes` 做 `decode()` 不好做。

对于未知编码的 `bytes`，要把它转换成 `str`，需要先“猜测”编码。猜测的方式是先收集各种编码的特征字符，根据特征字符判断，就能有很大概率“猜对”。

当然，我们肯定不能从头自己写这个检测编码的功能，这样做费时费力。chardet这个第三方库正好就派上了用场。用它来检测编码，简单易用。

安装chardet

如果安装了Anaconda，chardet就已经可用了。否则，需要在命令行下通过pip安装：

```
$ pip install chardet
```

如果遇到Permission denied安装失败，请加上sudo重试。

使用chardet

当我们拿到一个 `bytes` 时，就可以对其检测编码。用chardet检测编码，只需要一行代码：

```
>>> chardet.detect(b'Hello, world!')
{'encoding': 'ascii', 'confidence': 1.0, 'language': ''}
```

检测出的编码是 `ascii`，注意到还有个 `confidence` 字段，表示检测的概率是1.0（即100%）。

我们来试试检测GBK编码的中文：

```
>>> data = '离离原上草，一岁一枯荣'.encode('gbk')
>>> chardet.detect(data)
{'encoding': 'GB2312', 'confidence': 0.7407407407407407, 'language': 'Chinese'}
```

检测的编码是 `GB2312`，注意到GBK是GB2312的超集，两者是同一种编码，检测正确的概率是

74%，`language` 字段指出的语言是 `'Chinese'` 。

对UTF-8编码进行检测：

```
>>> data = '离离原上草，一岁一枯荣'.encode('utf-8')
>>> chardet.detect(data)
{'encoding': 'utf-8', 'confidence': 0.99, 'language': ''}
```

我们再试试对日文进行检测：

```
>>> data = '最新の主要ニュース'.encode('euc-jp')
>>> chardet.detect(data)
{'encoding': 'EUC-JP', 'confidence': 0.99, 'language': 'Japanese'}
```

可见，用chardet检测编码，使用简单。获取到编码后，再转换为 `str`，就可以方便后续处理。

chardet支持检测的编码列表请参考官方文档[Supported encodings](#)。

小结

使用chardet检测编码非常容易，chardet支持检测中文、日文、韩文等多种语言。

4、psutil

用Python来编写脚本简化日常的运维工作是Python的一个重要用途。在Linux下，有许多系统命令可以让我们时刻监控系统运行的状态，如 `ps`，`top`，`free` 等等。要获取这些系统信息，Python可以通过 `subprocess` 模块调用并获取结果。但这样做显得很麻烦，尤其是要写很多解析代码。

在Python中获取系统信息的另一个好办法是使用 `psutil` 这个第三方模块。顾名思义，`psutil` = process and system utilities，它不仅可以通过一两行代码实现系统监控，还可以跨平台使用，支持Linux / UNIX / OSX / Windows等，是系统管理员和运维小伙伴不可或缺的必备模块。

安装psutil

如果安装了Anaconda，`psutil`就已经可用了。否则，需要在命令行下通过pip安装：

```
$ pip install psutil
```

如果遇到Permission denied安装失败，请加上sudo重试。

获取CPU信息

我们先来获取CPU的信息：

```
>>> import psutil
>>> psutil.cpu_count() # CPU逻辑数量
4
>>> psutil.cpu_count(logical=False) # CPU物理核心
2
# 2说明是双核超线程，4则是4核非超线程
```

统计CPU的用户 / 系统 / 空闲时间：

```
>>> psutil.cpu_times()
scputimes(user=10963.31, nice=0.0, system=5138.67, idle=356102.45)
```

再实现类似 `top` 命令的CPU使用率，每秒刷新一次，累计10次：

```
>>> for x in range(10):
...     psutil.cpu_percent(interval=1, percpu=True)
...
[14.0, 4.0, 4.0, 4.0]
[12.0, 3.0, 4.0, 3.0]
[8.0, 4.0, 3.0, 4.0]
[12.0, 3.0, 3.0, 3.0]
[18.8, 5.1, 5.9, 5.0]
[10.9, 5.0, 4.0, 3.0]
[12.0, 5.0, 4.0, 5.0]
[15.0, 5.0, 4.0, 4.0]
[19.0, 5.0, 5.0, 4.0]
[9.0, 3.0, 2.0, 3.0]
```

获取内存信息

使用psutil获取物理内存和交换内存信息，分别使用：

```
>>> psutil.virtual_memory()
svmem(total=8589934592, available=2866520064, percent=66.6, used=7201386496, free=2
```

```
16178688, active=3342192640, inactive=2650341376, wired=1208852480)
>>> psutil.swap_memory()
sswap(total=1073741824, used=150732800, free=923009024, percent=14.0, sin=107059814
40, sout=40353792)
```

返回的是字节为单位的整数，可以看到，总内存大小是 $8589934592 = 8 \text{ GB}$ ，已用 $7201386496 = 6.7 \text{ GB}$ ，使用了66.6%。

而交换区大小是 $1073741824 = 1 \text{ GB}$ 。

获取磁盘信息

可以通过psutil获取磁盘分区、磁盘使用率和磁盘IO信息：

```
>>> psutil.disk_partitions() # 磁盘分区信息
[sdiskpart(device='/dev/disk1', mountpoint='/', fstype='hfs', opts='rw,local,rootfs
,dovolfs,journaled,multilabel')]
>>> psutil.disk_usage('/') # 磁盘使用情况
sdiskusage(total=998982549504, used=390880133120, free=607840272384, percent=39.1)
>>> psutil.disk_io_counters() # 磁盘IO
sdiskio(read_count=988513, write_count=274457, read_bytes=14856830464, write_bytes=
17509420032, read_time=2228966, write_time=1618405)
```

可以看到，磁盘 `'/'` 的总容量是 $998982549504 = 930 \text{ GB}$ ，使用了39.1%。文件格式是HFS，`opts` 中包含 `rw` 表示可读写，`journaled` 表示支持日志。

获取网络信息

psutil可以获取网络接口和网络连接信息：

```
>>> psutil.net_io_counters() # 获取网络读写字节 / 包的个数
snetio(bytes_sent=3885744870, bytes_recv=10357676702, packets_sent=10613069, packet
s_recv=10423357, errin=0, errout=0, dropin=0, dropout=0)
>>> psutil.net_if_addrs() # 获取网络接口信息
{
  'lo0': [snic(family=<AddressFamily.AF_INET: 2>, address='127.0.0.1', netmask='255
.0.0.0'), ...],
  'en1': [snic(family=<AddressFamily.AF_INET: 2>, address='10.0.1.80', netmask='255
.255.255.0'), ...],
  'en0': [...],
  'en2': [...],
  'bridge0': [...]
}
```



```
>>> psutil.net_if_stats() # 获取网络接口状态
{
    'lo0': snicstats(isup=True, duplex=<NicDuplex.NIC_DUPLEX_UNKNOWN: 0>, speed=0, mtu=16384),
    'en0': snicstats(isup=True, duplex=<NicDuplex.NIC_DUPLEX_UNKNOWN: 0>, speed=0, mtu=1500),
    'en1': snicstats(...),
    'en2': snicstats(...),
    'bridge0': snicstats(...)
}
```

要获取当前网络连接信息，使用 `net_connections()`：

```
>>> psutil.net_connections()
Traceback (most recent call last):
...
PermissionError: [Errno 1] Operation not permitted

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
...
psutil.AccessDenied: psutil.AccessDenied (pid=3847)
```

你可能会得到一个 `AccessDenied` 错误，原因是psutil获取信息也是要走系统接口，而获取网络连接信息需要root权限，这种情况下，可以退出Python交互环境，用 `sudo` 重新启动：

```
$ sudo python3
Password: *****
Python 3.6.3 ... on darwin
Type "help", ... for more information.
>>> import psutil
>>> psutil.net_connections()
[
    sconn(fd=83, family=<AddressFamily.AF_INET6: 30>, type=1, laddr=addr(ip='::127.0.0.1', port=62911), raddr=addr(ip='::127.0.0.1', port=3306), status='ESTABLISHED', pid=3725),
    sconn(fd=84, family=<AddressFamily.AF_INET6: 30>, type=1, laddr=addr(ip='::127.0.0.1', port=62905), raddr=addr(ip='::127.0.0.1', port=3306), status='ESTABLISHED', pid=3725),
    sconn(fd=93, family=<AddressFamily.AF_INET6: 30>, type=1, laddr=addr(ip='::', port=8080), raddr=(), status='LISTEN', pid=3725),
    sconn(fd=103, family=<AddressFamily.AF_INET6: 30>, type=1, laddr=addr(ip='::127
```

```
.0.0.1', port=62918), raddr=addr(ip='::127.0.0.1', port=3306), status='ESTABLISHED'
, pid=3725),
    sconn(fd=105, family=<AddressFamily.AF_INET6: 30>, type=1, ..., pid=3725),
    sconn(fd=106, family=<AddressFamily.AF_INET6: 30>, type=1, ..., pid=3725),
    sconn(fd=107, family=<AddressFamily.AF_INET6: 30>, type=1, ..., pid=3725),
    ...
    sconn(fd=27, family=<AddressFamily.AF_INET: 2>, type=2, ..., pid=1)
]
```

获取进程信息

通过psutil可以获取到所有进程的详细信息：

```
>>> psutil.pids() # 所有进程ID
[3865, 3864, 3863, 3856, 3855, 3853, 3776, ..., 45, 44, 1, 0]
>>> p = psutil.Process(3776) # 获取指定进程ID=3776, 其实就是当前Python交互环境
>>> p.name() # 进程名称
'python3.6'
>>> p.exe() # 进程exe路径
'/Users/michael/anaconda3/bin/python3.6'
>>> p.cwd() # 进程工作目录
'/Users/michael'
>>> p.cmdline() # 进程启动的命令行
['python3']
>>> p.ppid() # 父进程ID
3765
>>> p.parent() # 父进程
<psutil.Process(pid=3765, name='bash') at 4503144040>
>>> p.children() # 子进程列表
[]
>>> p.status() # 进程状态
'running'
>>> p.username() # 进程用户名
'michael'
>>> p.create_time() # 进程创建时间
1511052731.120333
>>> p.terminal() # 进程终端
'/dev/ttys002'
>>> p.cpu_times() # 进程使用的CPU时间
pcputimes(user=0.081150144, system=0.053269812, children_user=0.0, children_system=
0.0)
>>> p.memory_info() # 进程使用的内存
pmem(rss=8310784, vms=2481725440, pfaults=3207, pageins=18)
>>> p.open_files() # 进程打开的文件
[]
>>> p.connections() # 进程相关网络连接
```

```
[ ]
>>> p.num_threads() # 进程的线程数量
1
>>> p.threads() # 所有线程信息
[pthread(id=1, user_time=0.090318, system_time=0.062736)]
>>> p.environ() # 进程环境变量
{'SHELL': '/bin/bash', 'PATH': '/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:...',
'PWD': '/Users/michael', 'LANG': 'zh_CN.UTF-8', ...}
>>> p.terminate() # 结束进程
Terminated: 15 <-- 自己把自己结束了
```

和获取网络连接类似，获取一个root用户的进程需要root权限，启动Python交互环境或者 .py 文件时，需要 `sudo` 权限。

psutil还提供了一个 `test()` 函数，可以模拟出 `ps` 命令的效果：

```
$ sudo python3
Password: *****
Python 3.6.3 ... on darwin
Type "help", ... for more information.
>>> import psutil
>>> psutil.test()
USER          PID %MEM    VSZ   RSS TTY      START    TIME  COMMAND
root           0  24.0 74270628 2016380 ?        Nov18    40:51  kernel_task
root           1   0.1  2494140   9484 ?        Nov18    01:39  launchd
root          44   0.4  2519872   36404 ?        Nov18    02:02  UserEventAgent
root          45    ?  2474032   1516 ?        Nov18    00:14  syslogd
root          47   0.1  2504768   8912 ?        Nov18    00:03  kextd
root          48   0.1  2505544   4720 ?        Nov18    00:19  fseventsd
_appleeven    52   0.1  2499748   5024 ?        Nov18    00:00  appleeventsd
root          53   0.1  2500592   6132 ?        Nov18    00:02  configd
...
```

小结

psutil使得Python程序获取系统信息变得易如反掌。

psutil还可以获取用户信息、Windows服务等很多有用的系统信息，具体请参考psutil的官网：<https://github.com/giampaolo/psutil>

七、virtualenv

在开发Python应用程序的时候，系统安装的Python3只有一个版本：3.4。所有第三方的包都会被 `pip` 安装到Python3的 `site-packages` 目录下。

如果我们要同时开发多个应用程序，那这些应用程序都会共用一个Python，就是安装在系统的Python 3。如果应用A需要jinja 2.7，而应用B需要jinja 2.6怎么办？

这种情况下，每个应用可能需要各自拥有一套“独立”的Python运行环境。`virtualenv`就是用来为一个应用创建一套“隔离”的Python运行环境。

首先，我们用 `pip` 安装`virtualenv`：

```
$ pip3 install virtualenv
```

然后，假定我们要开发一个新的项目，需要一套独立的Python运行环境，可以这么做：

第一步，创建目录：

```
Mac:~ michael$ mkdir myproject
Mac:~ michael$ cd myproject/
Mac:myproject michael$
```

第二步，创建一个独立的Python运行环境，命名为 `venv`：

```
Mac:myproject michael$ virtualenv --no-site-packages venv
Using base prefix '/usr/local/.../Python.framework/Versions/3.4'
New python executable in venv/bin/python3.4
Also creating executable in venv/bin/python
Installing setuptools, pip, wheel...done.
```

命令 `virtualenv` 就可以创建一个独立的Python运行环境，我们还加上了参数 `--no-site-packages`，这样，已经安装到系统Python环境中的所有第三方包都不会复制过来，这样，我们就得到了一个不带任何第三方包的“干净”的Python运行环境。

新建的Python环境被放到当前目录下的 `venv` 目录。有了 `venv` 这个Python环境，可以用 `source` 进入该环境：

```
Mac:myproject michael$ source venv/bin/activate
(venv)Mac:myproject michael$
```

注意到命令提示符变了，有个 `(venv)` 前缀，表示当前环境是一个名为 `venv` 的Python环境。

下面正常安装各种第三方包，并运行 `python` 命令：

```
(venv)Mac:myproject michael$ pip install jinja2
...
Successfully installed jinja2-2.7.3 markupsafe-0.23
(venv)Mac:myproject michael$ python myapp.py
...
```

在 `venv` 环境下，用 `pip` 安装的包都被安装到 `venv` 这个环境下，系统Python环境不受任何影响。也就是说，`venv` 环境是专门针对 `myproject` 这个应用创建的。

退出当前的 `venv` 环境，使用 `deactivate` 命令：

```
(venv)Mac:myproject michael$ deactivate
Mac:myproject michael$
```

此时就回到了正常的环境，现在 `pip` 或 `python` 均是在系统Python环境下执行。

完全可以针对每个应用创建独立的Python运行环境，这样就可以对每个应用的Python环境进行隔离。

`virtualenv`是如何创建“独立”的Python运行环境的呢？原理很简单，就是把系统Python复制一份到`virtualenv`的环境，用命令 `source venv/bin/activate` 进入一个`virtualenv`环境时，`virtualenv`会修改相关环境变量，让命令 `python` 和 `pip` 均指向当前的`virtualenv`环境。

小结

`virtualenv`为应用提供了隔离的Python运行环境，解决了不同应用间多版本的冲突问题。

八、图形界面

Python支持多种图形界面的第三方库，包括：

- Tk
- wxWidgets

- Qt
- GTK

等等。

但是Python自带的库是支持Tk的Tkinter，使用Tkinter，无需安装任何包，就可以直接使用。本章简单介绍如何使用Tkinter进行GUI编程。

Tkinter

我们来梳理一下概念：

我们编写的Python代码会调用内置的Tkinter，Tkinter封装了访问Tk的接口；

Tk是一个图形库，支持多个操作系统，使用Tcl语言开发；

Tk会调用操作系统提供的本地GUI接口，完成最终的GUI。

所以，我们的代码只需要调用Tkinter提供的接口就可以了。

第一个GUI程序

使用Tkinter十分简单，我们来编写一个GUI版本的“Hello, world!”。

第一步是导入Tkinter包的所有内容：

```
from tkinter import *
```

第二步是从 `Frame` 派生一个 `Application` 类，这是所有Widget的父容器：

```
class Application(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

    def createWidgets(self):
        self.helloLabel = Label(self, text='Hello, world!')
        self.helloLabel.pack()
        self.quitButton = Button(self, text='Quit', command=self.quit)
        self.quitButton.pack()
```

在GUI中，每个Button、Label、输入框等，都是一个Widget。Frame则是可以容纳其他Widget的Widget，所有的Widget组合起来就是一棵树。

`pack()` 方法把Widget加入到父容器中，并实现布局。`pack()` 是最简单的布局，`grid()` 可以实现更复杂的布局。

在 `createWidgets()` 方法中，我们创建一个 `Label` 和一个 `Button`，当Button被点击时，触发 `self.quit()` 使程序退出。

第三步，实例化 `Application`，并启动消息循环：

```
app = Application()
# 设置窗口标题:
app.master.title('Hello World')
# 主消息循环:
app.mainloop()
```

GUI程序的主线程负责监听来自操作系统的消息，并依次处理每一条消息。因此，如果消息处理非常耗时，就需要在新线程中处理。

运行这个GUI程序，可以看到下面的窗口：



点击“Quit”按钮或者窗口的“x”结束程序。

输入文本

我们再对这个GUI程序改进一下，加入一个文本框，让用户可以输入文本，然后点按钮后，弹出消息对话框。

```
from tkinter import *
import tkinter.messagebox as messagebox

class Application(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()
```

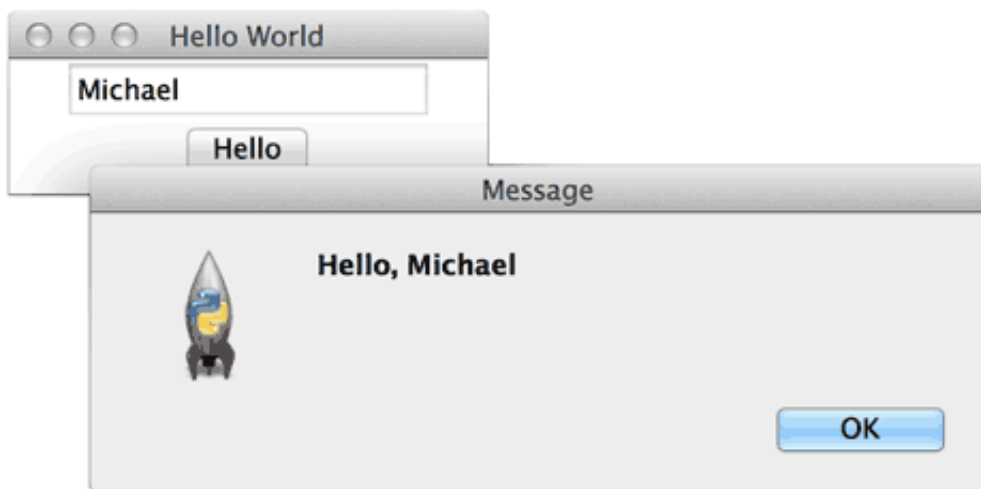
```
def createWidgets(self):
    self.nameInput = Entry(self)
    self.nameInput.pack()
    self.alertButton = Button(self, text='Hello', command=self.hello)
    self.alertButton.pack()

def hello(self):
    name = self.nameInput.get() or 'world'
    messagebox.showinfo('Message', 'Hello, %s' % name)

app = Application()
# 设置窗口标题:
app.master.title('Hello World')
# 主消息循环:
app.mainloop()
```

当用户点击按钮时，触发 `hello()`，通过 `self.nameInput.get()` 获得用户输入的文本后，使用 `tkMessageBox.showinfo()` 可以弹出消息对话框。

程序运行结果如下：



小结

Python内置的Tkinter可以满足基本的GUI程序的要求，如果是非常复杂的GUI程序，建议用操作系统原生支持的语言和库来编写。

参考源码

[hello_gui.py](#)

九、网络编程

自从互联网诞生以来，现在基本上所有的程序都是网络程序，很少有单机版的程序了。

计算机网络就是把各个计算机连接到一起，让网络中的计算机可以互相通信。网络编程就是如何在程序中实现两台计算机的通信。

举个例子，当你使用浏览器访问新浪网时，你的计算机就和新浪的某台服务器通过互联网连接起来了，然后，新浪的服务器把网页内容作为数据通过互联网传输到你的电脑上。

由于你的电脑上可能不止浏览器，还有QQ、Skype、Dropbox、邮件客户端等，不同的程序连接的别的计算机也会不同，所以，更确切地说，网络通信是两台计算机上的两个进程之间的通信。比如，浏览器进程和新浪服务器上的某个Web服务进程在通信，而QQ进程是和腾讯的某个服务器上的某个进程在通信。

原来网络通信就是两个进程之间在通信



网络编程对所有开发语言都是一样的，Python也不例外。用Python进行网络编程，就是在Python程序本身这个进程内，连接别的服务器进程的通信端口进行通信。

本章我们将详细介绍Python网络编程的概念和最主要的两种网络类型的编程。

1、TCP/IP简介

虽然大家现在对互联网很熟悉，但是计算机网络的出现比互联网要早很多。

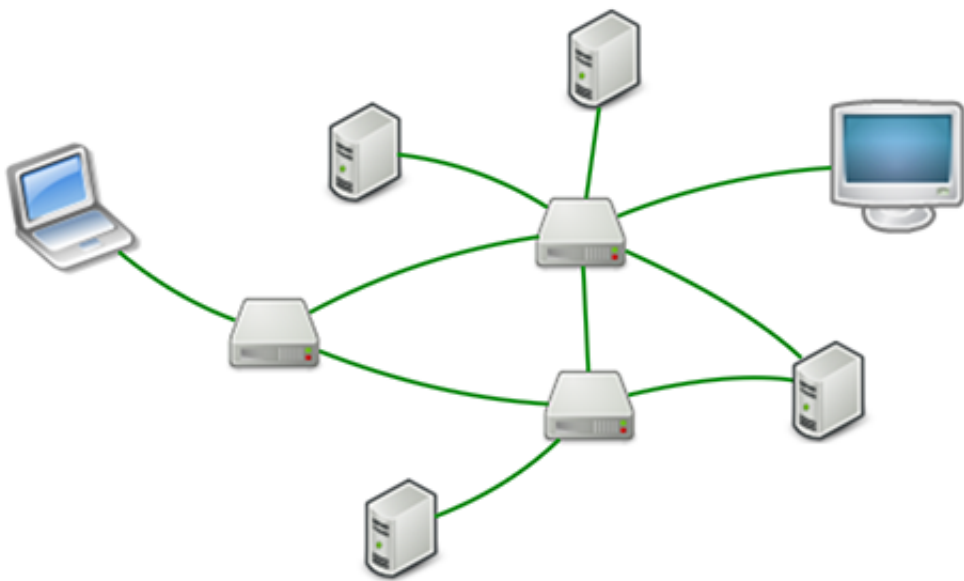
计算机为了联网，就必须规定通信协议，早期的计算机网络，都是由各厂商自己规定一套协议，IBM、Apple和Microsoft都有各自的网络协议，互不兼容，这就好比一群人有的说英语，有的说中文，有的说德语，说同一种语言的人可以交流，不同的语言之间就不行了。

为了把全世界的所有不同类型的计算机都连接起来，就必须规定一套全球通用的协议，为了实现互联网这个目标，互联网协议簇（Internet Protocol Suite）就是通用协议标准。Internet是由inter和net两个单词组合起来的，原意就是连接“网络”的网络，有了Internet，任何私有网络，只要支持这个协议，就可以联入互联网。

因为互联网协议包含了上百种协议标准，但是最重要的两个协议是TCP和IP协议，所以，大家把互联网的协议简称TCP/IP协议。

通信的时候，双方必须知道对方的标识，好比发邮件必须知道对方的邮件地址。互联网上每个计算机的唯一标识就是IP地址，类似 `123.123.123.123`。如果一台计算机同时接入到两个或更多的网络，比如路由器，它就会有两个或多个IP地址，所以，IP地址对应的实际上是计算机的网络接口，通常是网卡。

IP协议负责把数据从一台计算机通过网络发送到另一台计算机。数据被分割成一小块一小块，然后通过IP包发送出去。由于互联网链路复杂，两台计算机之间经常有多条线路，因此，路由器就负责决定如何把一个IP包转发出去。IP包的特点是按块发送，途径多个路由，但不保证能到达，也不保证顺序到达。



IP地址实际上是一个32位整数（称为IPv4），以字符串表示的IP地址如 `192.168.0.1` 实际上是把32位整数按8位分组后的数字表示，目的是便于阅读。

IPv6地址实际上是一个128位整数，它是目前使用的IPv4的升级版，以字符串表示类似于 `2001:0db8:85a3:0042:1000:8a2e:0370:7334`。

TCP协议则是建立在IP协议之上的。TCP协议负责在两台计算机之间建立可靠连接，保证数据包按顺序到达。TCP协议会通过握手建立连接，然后，对每个IP包编号，确保对方按顺序收到，如果包丢掉了，就自动重发。

许多常用的更高级的协议都是建立在TCP协议基础上的，比如用于浏览器的HTTP协议、发送邮件的SMTP协议等。

一个TCP报文除了包含要传输的数据外，还包含源IP地址和目标IP地址，源端口和目标端口。

端口有什么作用？在两台计算机通信时，只发IP地址是不够的，因为同一台计算机上跑着多个网络程序。一个TCP报文来了之后，到底是交给浏览器还是QQ，就需要端口号来区分。每个网络

程序都向操作系统申请唯一的端口号，这样，两个进程在两台计算机之间建立网络连接就需要各自的IP地址和各自的端口号。

一个进程也可能同时与多个计算机建立链接，因此它会申请很多端口。

了解了TCP/IP协议的基本概念，IP地址和端口的概念，我们就可以开始进行网络编程了。

2、TCP编程

Socket是网络编程的一个抽象概念。通常我们用一个Socket表示“打开了一个网络链接”，而打开一个Socket需要知道目标计算机的IP地址和端口号，再指定协议类型即可。

客户端

大多数连接都是可靠的TCP连接。创建TCP连接时，主动发起连接的叫客户端，被动响应连接的叫服务器。

举个例子，当我们在浏览器中访问新浪时，我们自己的计算机就是客户端，浏览器会主动向新浪的服务器发起连接。如果一切顺利，新浪的服务器接受了我们的连接，一个TCP连接就建立起来的，后面的通信就是发送网页内容了。

所以，我们要创建一个基于TCP连接的Socket，可以这样做：

```
# 导入socket库:
import socket

# 创建一个socket:
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# 建立连接:
s.connect(('www.sina.com.cn', 80))
```

创建 Socket 时，AF_INET 指定使用IPv4协议，如果要用更先进的IPv6，就指定为 AF_INET6。SOCK_STREAM 指定使用面向流的TCP协议，这样，一个 Socket 对象就创建成功，但是还没有建立连接。

客户端要主动发起TCP连接，必须知道服务器的IP地址和端口号。新浪网站的IP地址可以用域名 www.sina.com.cn 自动转换到IP地址，但是怎么知道新浪服务器的端口号呢？

答案是作为服务器，提供什么样的服务，端口号就必须固定下来。由于我们想要访问网页，因此新浪提供网页服务的服务器必须把端口号固定在 80 端口，因为 80 端口是Web服务的标准端口。其他服务都有对应的标准端口号，例如SMTP服务是 25 端口，FTP服务是 21 端口，等等。端口号小于1024的是Internet标准服务的端口，端口号大于1024的，可以任意使用。

因此，我们连接新浪服务器的代码如下：

```
s.connect(('www.sina.com.cn', 80))
```

注意参数是一个 `tuple`，包含地址和端口号。

建立TCP连接后，我们就可以向新浪服务器发送请求，要求返回首页的内容：

```
# 发送数据：
s.send(b'GET / HTTP/1.1\r\nHost: www.sina.com.cn\r\nConnection: close\r\n\r\n')
```

TCP连接创建的是双向通道，双方都可以同时给对方发数据。但是谁先发谁后发，怎么协调，要根据具体的协议来决定。例如，HTTP协议规定客户端必须先发请求给服务器，服务器收到后才发数据给客户端。

发送的文本格式必须符合HTTP标准，如果格式没问题，接下来就可以接收新浪服务器返回的数据了：

```
# 接收数据：
buffer = []
while True:
    # 每次最多接收1k字节：
    d = s.recv(1024)
    if d:
        buffer.append(d)
    else:
        break
data = b''.join(buffer)
```

接收数据时，调用 `recv(max)` 方法，一次最多接收指定的字节数，因此，在一个while循环中反复接收，直到 `recv()` 返回空数据，表示接收完毕，退出循环。

当我们接收完数据后，调用 `close()` 方法关闭Socket，这样，一次完整的网络通信就结束了：

```
# 关闭连接：
s.close()
```

接收到的数据包括HTTP头和网页本身，我们只需要把HTTP头和网页分离一下，把HTTP头打印出来，网页内容保存到文件：

```
header, html = data.split(b'\r\n\r\n', 1)
print(header.decode('utf-8'))
# 把接收的数据写入文件:
with open('sina.html', 'wb') as f:
    f.write(html)
```

现在，只需要在浏览器中打开这个 `sina.html` 文件，就可以看到新浪的首页了。

服务器

和客户端编程相比，服务器编程就要复杂一些。

服务器进程首先要绑定一个端口并监听来自其他客户端的连接。如果某个客户端连接过来了，服务器就与该客户端建立Socket连接，随后的通信就靠这个Socket连接了。

所以，服务器会打开固定端口（比如80）监听，每来一个客户端连接，就创建该Socket连接。由于服务器会有大量来自客户端的连接，所以，服务器要能够区分一个Socket连接是和哪个客户端绑定的。一个Socket依赖4项：服务器地址、服务器端口、客户端地址、客户端端口来唯一确定一个Socket。

但是服务器还需要同时响应多个客户端的请求，所以，每个连接都需要一个新的进程或者新的线程来处理，否则，服务器一次就只能服务一个客户端了。

我们来编写一个简单的服务器程序，它接收客户端连接，把客户端发过来的字符串加上 `Hello` 再发回去。

首先，创建一个基于IPv4和TCP协议的Socket：

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

然后，我们要绑定监听的地址和端口。服务器可能有多块网卡，可以绑定到某一块网卡的IP地址上，也可以用 `0.0.0.0` 绑定到所有的网络地址，还可以用 `127.0.0.1` 绑定到本机地址。`127.0.0.1` 是一个特殊的IP地址，表示本机地址，如果绑定到这个地址，客户端必须同时在本机运行才能连接，也就是说，外部的计算机无法连接进来。

端口号需要预先指定。因为我们写的这个服务不是标准服务，所以用 `9999` 这个端口号。请注意，小于 `1024` 的端口号必须要有管理员权限才能绑定：

```
# 监听端口:
s.bind(('127.0.0.1', 9999))
```

紧接着，调用 `listen()` 方法开始监听端口，传入的参数指定等待连接的最大数量：

```
s.listen(5)
print('Waiting for connection...')
```

接下来，服务器程序通过一个永久循环来接受来自客户端的连接，`accept()` 会等待并返回一个客户端的连接：

```
while True:
    # 接受一个新连接:
    sock, addr = s.accept()
    # 创建新线程来处理TCP连接:
    t = threading.Thread(target=tcplink, args=(sock, addr))
    t.start()
```

每个连接都必须创建新线程（或进程）来处理，否则，单线程在处理连接的过程中，无法接受其他客户端的连接：

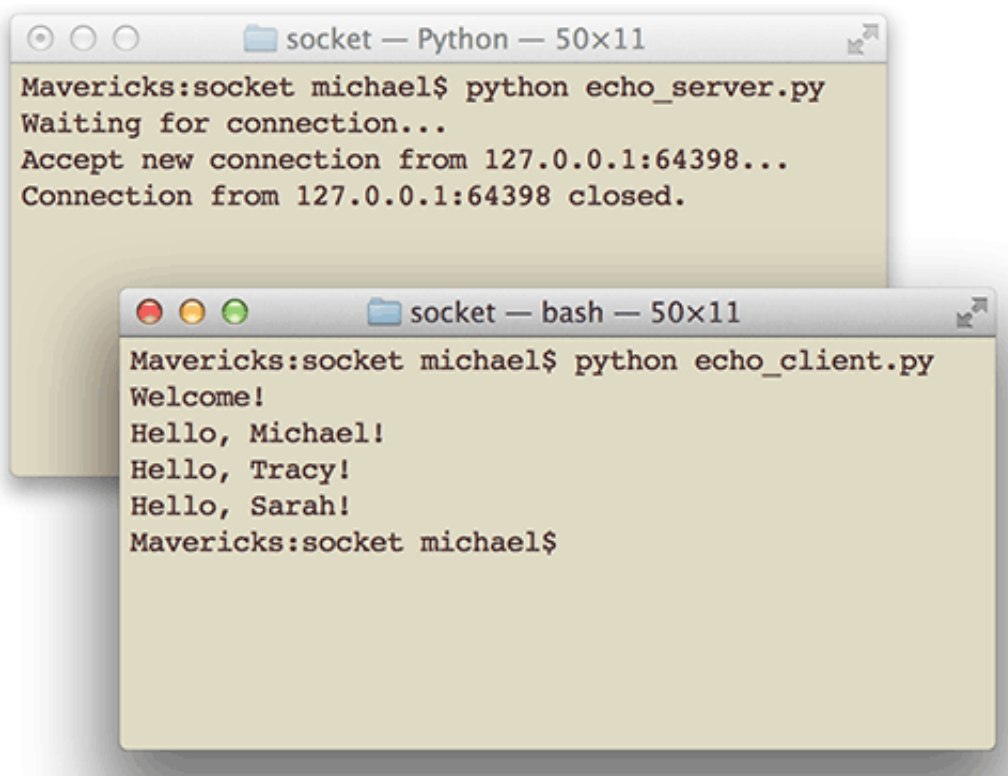
```
def tcplink(sock, addr):
    print('Accept new connection from %s:%s...' % addr)
    sock.send(b'Welcome!')
    while True:
        data = sock.recv(1024)
        time.sleep(1)
        if not data or data.decode('utf-8') == 'exit':
            break
        sock.send(('Hello, %s!' % data.decode('utf-8')).encode('utf-8'))
    sock.close()
    print('Connection from %s:%s closed.' % addr)
```

连接建立后，服务器首先发一条欢迎消息，然后等待客户端数据，并加上 `Hello` 再发送给客户端。如果客户端发送了 `exit` 字符串，就直接关闭连接。

要测试这个服务器程序，我们还需要编写一个客户端程序：


```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# 建立连接:
s.connect(('127.0.0.1', 9999))
# 接收欢迎消息:
print(s.recv(1024).decode('utf-8'))
for data in [b'Michael', b'Tracy', b'Sarah']:
    # 发送数据:
    s.send(data)
    print(s.recv(1024).decode('utf-8'))
s.send(b'exit')
s.close()
```

我们需要打开两个命令行窗口，一个运行服务器程序，另一个运行客户端程序，就可以看到效果了：



需要注意的是，客户端程序运行完毕就退出了，而服务器程序会永远运行下去，必须按Ctrl+C退出程序。

小结

用TCP协议进行Socket编程在Python中十分简单，对于客户端，要主动连接服务器的IP和指定端口，对于服务器，要首先监听指定端口，然后，对每一个新的连接，创建一个线程或进程来处理。通常，服务器程序会无限运行下去。

同一个端口，被一个Socket绑定了以后，就不能被别的Socket绑定了。

参考源码

[do_tcp.py](#)

3、UDP编程

TCP是建立可靠连接，并且通信双方都可以以流的形式发送数据。相对TCP，UDP则是面向无连接的协议。

使用UDP协议时，不需要建立连接，只需要知道对方的IP地址和端口号，就可以直接发数据包。但是，能不能到达就不知道了。

虽然用UDP传输数据不可靠，但它的优点是和TCP比，速度快，对于不要求可靠到达的数据，就可以使用UDP协议。

我们来看看如何通过UDP协议传输数据。和TCP类似，使用UDP的通信双方也分为客户端和服务端。服务端首先需要绑定端口：

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
# 绑定端口:
s.bind(('127.0.0.1', 9999))
```

创建Socket时，`SOCK_DGRAM` 指定了这个Socket的类型是UDP。绑定端口和TCP一样，但是不需要调用 `listen()` 方法，而是直接接收来自任何客户端的数据：

```
print('Bind UDP on 9999...')
while True:
    # 接收数据:
    data, addr = s.recvfrom(1024)
    print('Received from %s:%s.' % addr)
    s.sendto(b'Hello, %s!' % data, addr)
```

`recvfrom()` 方法返回数据和客户端的地址与端口，这样，服务端收到数据后，直接调用 `sendto()` 就可以把数据用UDP发给客户端。

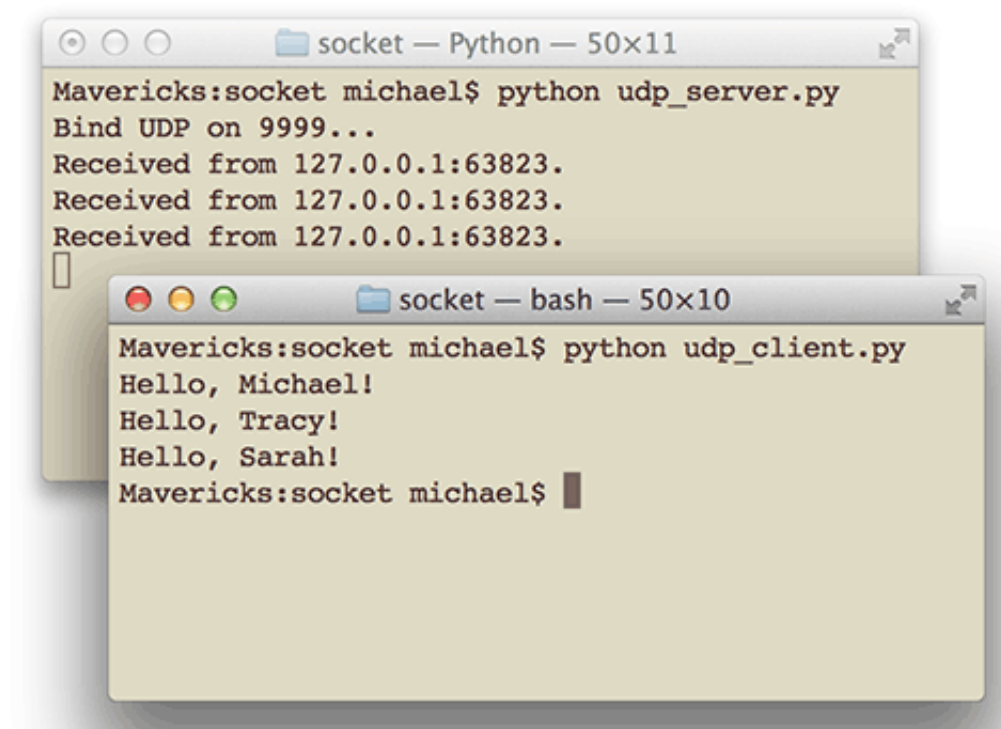
注意这里省掉了多线程，因为这个例子很简单。

客户端使用UDP时，首先仍然创建基于UDP的Socket，然后，不需要调用 `connect()`，直接通过 `sendto()` 给服务端发数据：


```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
for data in [b'Michael', b'Tracy', b'Sarah']:
    # 发送数据:
    s.sendto(data, ('127.0.0.1', 9999))
    # 接收数据:
    print(s.recv(1024).decode('utf-8'))
s.close()
```

从服务器接收数据仍然调用 `recv()` 方法。

仍然用两个命令行分别启动服务器和客户端测试，结果如下：



小结

UDP的使用与TCP类似，但是不需要建立连接。此外，服务器绑定UDP端口和TCP端口互不冲突，也就是说，UDP的9999端口与TCP的9999端口可以各自绑定。

参考源码

[udp_server.py](#)

[udp_client.py](#)

十、电子邮件

Email的历史比Web还要久远，直到现在，Email也是互联网上应用非常广泛的服务。

几乎所有的编程语言都支持发送和接收电子邮件，但是，先等等，在我们开始编写代码之前，有必要搞清楚电子邮件是如何在互联网上运作的。

我们来看看传统邮件是如何运作的。假设你现在在北京，要给一个香港的朋友发一封信，怎么做呢？

首先你得写好信，装进信封，写上地址，贴上邮票，然后就近找个邮局，把信仍进去。

信件会从就近的小邮局转运到大邮局，再从大邮局往别的城市发，比如先发天津，再走海运到达香港，也可能走京九线到香港，但是你不用关心具体路线，你只需要知道一件事，就是信件走得很慢，至少要几天时间。

信件到达香港的某个邮局，也不会直接送到朋友的家里，因为邮局的叔叔是很聪明的，他怕你的朋友不在家，一趟一趟地白跑，所以，信件会投递到你的朋友的邮箱里，邮箱可能在公寓的一层，或者家门口，直到你的朋友回家的时候检查邮箱，发现信件后，就可以取到邮件了。

电子邮件的流程基本上也是按上面的方式运作的，只不过速度不是按天算，而是按秒算。

现在我们回到电子邮件，假设我们自己的电子邮件地址是 `me@163.com`，对方的电子邮件地址是 `friend@sina.com`（注意地址都是虚构的哈），现在我们用 Outlook 或者 Foxmail 之类的软件写好邮件，填上对方的Email地址，点“发送”，电子邮件就发出去了。这些电子邮件软件被称为**MUA**：Mail User Agent——邮件用户代理。

Email从MUA发出去，不是直接到达对方电脑，而是发到**MTA**：Mail Transfer Agent——邮件传输代理，就是那些Email服务提供商，比如网易、新浪等等。由于我们自己的电子邮件是 `163.com`，所以，Email首先被投递到网易提供的MTA，再由网易的MTA发到对方服务商，也就是新浪的MTA。这个过程中间可能还会经过别的MTA，但是我们不关心具体路线，我们只关心速度。

Email到达新浪的MTA后，由于对方使用的是 `@sina.com` 的邮箱，因此，新浪的MTA会把Email投递到邮件的最终目的地**MDA**：Mail Delivery Agent——邮件投递代理。Email到达MDA后，就静静地躺在新浪的某个服务器上，存放在某个文件或特殊的数据库里，我们将这个长期保存邮件的地方称之为电子邮箱。

同普通邮件类似，Email不会直接到达对方的电脑，因为对方电脑不一定开机，开机也不一定联网。对方要取到邮件，必须通过MUA从MDA上把邮件取到自己的电脑上。

所以，一封电子邮件的旅程就是：

发件人 -> MUA -> MTA -> MTA -> 若干个MTA -> MDA <- MUA <- 收件人

有了上述基本概念，要编写程序来发送和接收邮件，本质上就是：

1. 编写MUA把邮件发到MTA；
2. 编写MUA从MDA上收邮件。

发邮件时，MUA和MTA使用的协议就是SMTP：Simple Mail Transfer Protocol，后面的MTA到另一个MTA也是用SMTP协议。

收邮件时，MUA和MDA使用的协议有两种：POP：Post Office Protocol，目前版本是3，俗称POP3；IMAP：Internet Message Access Protocol，目前版本是4，优点是不但能取邮件，还可以直接操作MDA上存储的邮件，比如从收件箱移到垃圾箱，等等。

邮件客户端软件在发邮件时，会让你先配置SMTP服务器，也就是你要发到哪个MTA上。假设你正在使用163的邮箱，你就不能直接发到新浪的MTA上，因为它只服务新浪的用户，所以，你得填163提供的SMTP服务器地址：`smtp.163.com`，为了证明你是163的用户，SMTP服务器还要求你填写邮箱地址和邮箱口令，这样，MUA才能正常地把Email通过SMTP协议发送到MTA。

类似的，从MDA收邮件时，MDA服务器也要求验证你的邮箱口令，确保不会有人冒充你收取你的邮件，所以，Outlook之类的邮件客户端会要求你填写POP3或IMAP服务器地址、邮箱地址和口令，这样，MUA才能顺利地通过POP或IMAP协议从MDA取到邮件。

在使用Python收发邮件前，请先准备好至少两个电子邮件，如 `xxx@163.com`，`xxx@sina.com`，`xxx@qq.com` 等，注意两个邮箱不要用同一家邮件服务商。

最后 **特别注意**，目前大多数邮件服务商都需要手动打开SMTP发信和POP收信的功能，否则只允许在网页登录：



1、SMTP发送邮件

SMTP是发送邮件的协议，Python内置对SMTP的支持，可以发送纯文本邮件、HTML邮件以及带附件的邮件。

Python对SMTP支持有 `smtplib` 和 `email` 两个模块，`email` 负责构造邮件，`smtplib` 负责发送邮

件。

首先，我们来构造一个最简单的纯文本邮件：

```
from email.mime.text import MIMEText
msg = MIMEText('hello, send by Python...', 'plain', 'utf-8')
```

注意到构造 `MIMEText` 对象时，第一个参数就是邮件正文，第二个参数是MIME的subtype，传入 `'plain'` 表示纯文本，最终的MIME就是 `'text/plain'`，最后一定要用 `utf-8` 编码保证多语言兼容性。

然后，通过SMTP发出去：


```
# 输入Email地址和口令：
from_addr = input('From: ')
password = input('Password: ')
# 输入收件人地址：
to_addr = input('To: ')
# 输入SMTP服务器地址：
smtp_server = input('SMTP server: ')

import smtplib
server = smtplib.SMTP(smtp_server, 25) # SMTP协议默认端口是25
server.set_debuglevel(1)
server.login(from_addr, password)
server.sendmail(from_addr, [to_addr], msg.as_string())
server.quit()
```

我们用 `set_debuglevel(1)` 就可以打印出和SMTP服务器交互的所有信息。SMTP协议就是简单的文本命令和响应。`login()` 方法用来登录SMTP服务器，`sendmail()` 方法就是发邮件，由于可以一次发给多个人，所以传入一个 `list`，邮件正文是一个 `str`，`as_string()` 把 `MIMEText` 对象变成 `str`。

如果一切顺利，就可以在收件人信箱中收到我们刚发送的Email：

(无主题) ☆

发件人: xxxxxx <xxxxxx@163.com> 

时 间: 2014年8月14日(星期四) 下午2:35

提 示: 你不在收件人里, 可能这封邮件是密送给你的。

hello, send by Python...

仔细观察, 发现如下问题:

1. 邮件没有主题;
2. 收件人的名字没有显示为友好的名字, 比如 Mr Green <green@example.com>;
3. 明明收到了邮件, 却提示不在收件人中。

这是因为邮件主题、如何显示发件人、收件人等信息并不是通过SMTP协议发给MTA, 而是包含在发给MTA的文本中的, 所以, 我们必须把 From、To 和 Subject 添加到 MIMEText 中, 才是一封完整的邮件:

```
from email import encoders
from email.header import Header
from email.mime.text import MIMEText
from email.utils import parseaddr, formataddr

import smtplib

def _format_addr(s):
    name, addr = parseaddr(s)
    return formataddr((Header(name, 'utf-8').encode(), addr))

from_addr = input('From: ')
password = input('Password: ')
to_addr = input('To: ')
smtp_server = input('SMTP server: ')

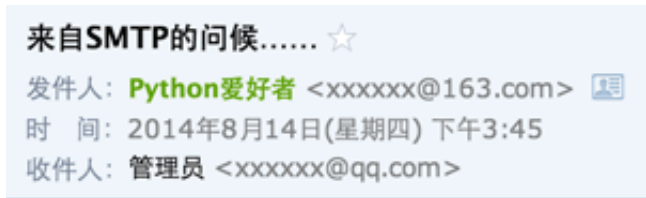
msg = MIMEText('hello, send by Python...', 'plain', 'utf-8')
msg['From'] = _format_addr('Python爱好者 <%s>' % from_addr)
msg['To'] = _format_addr('管理员 <%s>' % to_addr)
msg['Subject'] = Header('来自SMTP的问候.....', 'utf-8').encode()

server = smtplib.SMTP(smtp_server, 25)
server.set_debuglevel(1)
server.login(from_addr, password)
server.sendmail(from_addr, [to_addr], msg.as_string())
server.quit()
```

我们编写了一个函数 `_format_addr()` 来格式化一个邮件地址。注意不能简单地传入 `name <addr@example.com>`，因为如果包含中文，需要通过 `Header` 对象进行编码。

`msg['To']` 接收的是字符串而不是list，如果有多个邮件地址，用 `,` 分隔即可。

再发送一遍邮件，就可以在收件人邮箱中看到正确的标题、发件人和收件人：



hello, send by Python...

你看到的收件人的名字很可能不是我们传入的 `管理员`，因为很多邮件服务商在显示邮件时，会把收件人名字自动替换为用户注册的名字，但是其他收件人名字的显示不受影响。

如果我们查看Email的原始内容，可以看到如下经过编码的邮件头：

```
From: =?utf-8?b?UHl0aG9u54ix5aW96ICF?= <xxxxxxx@163.com>
To: =?utf-8?b?566h55CG5ZGY?= <xxxxxxx@qq.com>
Subject: =?utf-8?b?5p2l6IeqU01UUOeahOmXruWAmeKApuKApg==?=
```

这就是经过 `Header` 对象编码的文本，包含utf-8编码信息和Base64编码的文本。如果我们自己来手动构造这样的编码文本，显然比较复杂。


发送HTML邮件

如果我们要发送HTML邮件，而不是普通的纯文本文件怎么办？方法很简单，在构造 `MIMEText` 对象时，把HTML字符串传进去，再把第二个参数由 `plain` 变为 `html` 就可以了：

```
msg = MIMEText('<html><body><h1>Hello</h1>' +
               '<p>send by <a href="http://www.python.org">Python</a>...</p>' +
               '</body></html>', 'html', 'utf-8')
```

再发送一遍邮件，你将看到以HTML显示的邮件：

来自SMTP的问候..... ☆

发件人: Python爱好者 <xxxxxx@163.com> 

时 间: 2014年8月14日(星期四) 下午4:06

收件人: 管理员 <xxxxxx@qq.com>

Hello

send by Python...

发送附件

如果Email中要加上附件怎么办？带附件的邮件可以看做包含若干部分的邮件：文本和各个附件本身，所以，可以构造一个 `MIMEMultipart` 对象代表邮件本身，然后往里面加上一个 `MIMEText` 作为邮件正文，再继续往里面加上表示附件的 `MIMEBase` 对象即可：


```
# 邮件对象：
msg = MIMEMultipart()
msg['From'] = _format_addr('Python爱好者 <%s>' % from_addr)
msg['To'] = _format_addr('管理员 <%s>' % to_addr)
msg['Subject'] = Header('来自SMTP的问候.....', 'utf-8').encode()

# 邮件正文是MIMEText：
msg.attach(MIMEText('send with file...', 'plain', 'utf-8'))

# 添加附件就是加上一个MIMEBase，从本地读取一个图片：
with open('/Users/michael/Downloads/test.png', 'rb') as f:
    # 设置附件的MIME和文件名，这里是png类型：
    mime = MIMEBase('image', 'png', filename='test.png')
    # 加上必要的头信息：
    mime.add_header('Content-Disposition', 'attachment', filename='test.png')
    mime.add_header('Content-ID', '<0>')
    mime.add_header('X-Attachment-Id', '0')
    # 把附件的内容读进来：
    mime.set_payload(f.read())
    # 用Base64编码：
    encoders.encode_base64(mime)
    # 添加到MIMEMultipart：
    msg.attach(mime)
```


然后，按正常发送流程把 `msg`（注意类型已变为 `MIMEMultipart`）发送出去，就可以收到如下带附件的邮件：

来自SMTP的问候..... ☆


发件人: Python爱好者 <xxxxxxx@163.com> 

时 间: 2014年8月14日(星期四) 下午5:08

收件人: 管理员 <xxxxxxx@qq.com>

附 件: 1 个 ( test.png)

send with file...

 附件(1 个)

普通附件



test.png (80.13K)

[下载](#) [预览](#) [收藏](#) [转存](#) ▼

发送图片

如果要把一个图片嵌入到邮件正文中怎么做？直接在HTML邮件中链接图片地址行不行？答案是，大部分邮件服务商都会自动屏蔽带有外链的图片，因为不知道这些链接是否指向恶意网站。

要把图片嵌入到邮件正文中，我们只需按照发送附件的方式，先把邮件作为附件添加进去，然后，在HTML中通过引用 `src="cid:0"` 就可以把附件作为图片嵌入了。如果有多个图片，给它们依次编号，然后引用不同的 `cid:x` 即可。

把上面代码加入 `MIMEMultipart` 的 `MIMEText` 从 `plain` 改为 `html`，然后在适当的位置引用图片：

```
msg.attach(MIMEText('<html><body><h1>Hello</h1>' +
    '<p></p>' +
    '</body></html>', 'html', 'utf-8'))
```

再次发送，就可以看到图片直接嵌入到邮件正文的效果：

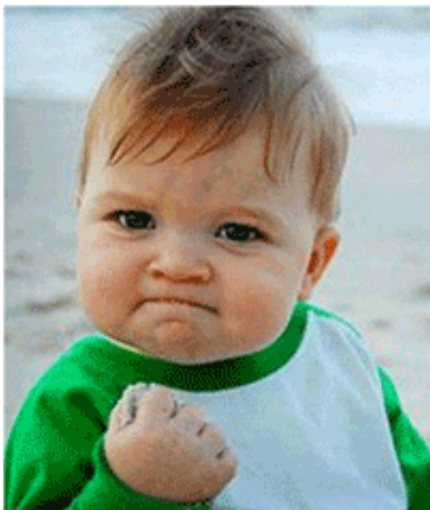
来自SMTP的问候..... ☆

发件人: Python爱好者 <asklxf@163.com> 

时 间: 2014年8月14日(星期四) 下午5:27

收件人: Xuefeng <18224514@qq.com>

Hello



同时支持HTML和Plain格式

如果我们发送HTML邮件，收件人通过浏览器或者Outlook之类的软件是可以正常浏览邮件内容的，但是，如果收件人使用的设备太古老，查看不了HTML邮件怎么办？

办法是在发送HTML的同时再附加一个纯文本，如果收件人无法查看HTML格式的邮件，就可以自动降级查看纯文本邮件。

利用 `MIMEMultipart` 就可以组合一个HTML和Plain，要注意指定subtype是 `alternative`：

```
msg = MIMEMultipart('alternative')
msg['From'] = ...
msg['To'] = ...
msg['Subject'] = ...

msg.attach(MIMEText('hello', 'plain', 'utf-8'))
msg.attach(MIMEText('<html><body><h1>Hello</h1></body></html>', 'html', 'utf-8'))
# 正常发送msg对象...
```

加密SMTP

使用标准的25端口连接SMTP服务器时，使用的是明文传输，发送邮件的整个过程可能会被窃听。要更安全地发送邮件，可以加密SMTP会话，实际上就是先创建SSL安全连接，然后再使用SMTP协议发送邮件。

某些邮件服务商，例如Gmail，提供的SMTP服务必须要加密传输。我们来看看如何通过Gmail提供的安全SMTP发送邮件。

必须知道，Gmail的SMTP端口是587，因此，修改代码如下：

```
smtp_server = 'smtp.gmail.com'
smtp_port = 587
server = smtplib.SMTP(smtp_server, smtp_port)
server.starttls()
# 剩下的代码和前面的一模一样：
server.set_debuglevel(1)
...
```

只需要在创建 SMTP 对象后，立刻调用 `starttls()` 方法，就创建了安全连接。后面的代码和前面的发送邮件代码完全一样。

如果因为网络问题无法连接Gmail的SMTP服务器，请相信我们的代码是没有问题的，你需要对你的网络设置做必要的调整。

小结

使用Python的smtplib发送邮件十分简单，只要掌握了各种邮件类型的构造方法，正确设置好邮件头，就可以顺利发出。

构造一个邮件对象就是一个 `Message` 对象，如果构造一个 `MIMEText` 对象，就表示一个文本邮件对象，如果构造一个 `MIMEImage` 对象，就表示一个作为附件的图片，要把多个对象组合起来，就用 `MIMEMultipart` 对象，而 `MIMEBase` 可以表示任何对象。它们的继承关系如下：

```
Message
+- MIMEBase
  +- MIMEMultipart
  +- MIMENonMultipart
    +- MIMEMessage
    +- MIMEText
    +- MIMEImage
```

这种嵌套关系就可以构造出任意复杂的邮件。你可以通过[email.mime文档](#)查看它们所在的包以及

详细的用法。

参考源码

[send_mail.py](#)

2、POP3收取邮件

SMTP用于发送邮件，如果要收取邮件呢？

收取邮件就是编写一个**MUA**作为客户端，从**MDA**把邮件获取到用户的电脑或者手机上。收取邮件最常用的协议是**POP**协议，目前版本号是3，俗称**POP3**。

Python内置一个 `poplib` 模块，实现了POP3协议，可以直接用来收邮件。

注意到POP3协议收取的不是一个已经可以阅读的邮件本身，而是邮件的原始文本，这和SMTP协议很像，SMTP发送的也是经过编码后的一大段文本。

要把POP3收取的文本变成可以阅读的邮件，还需要用 `email` 模块提供的各种类来解析原始文本，变成可阅读的邮件对象。

所以，收取邮件分两步：

第一步：用 `poplib` 把邮件的原始文本下载到本地；

第二部：用 `email` 解析原始文本，还原为邮件对象。

通过POP3下载邮件

POP3协议本身很简单，以下面的代码为例，我们来获取最新的一封邮件内容：

```
import poplib

# 输入邮件地址，口令和POP3服务器地址：
email = input('Email: ')
password = input('Password: ')
pop3_server = input('POP3 server: ')

# 连接到POP3服务器：
server = poplib.POP3(pop3_server)
# 可以打开或关闭调试信息：
server.set_debuglevel(1)
# 可选:打印POP3服务器的欢迎文字：
print(server.getwelcome().decode('utf-8'))

# 身份认证：
```

```
server.user(email)
server.pass_(password)

# stat()返回邮件数量和占用空间:
print('Messages: %s. Size: %s' % server.stat())
# list()返回所有邮件的编号:
resp, mails, octets = server.list()
# 可以查看返回的列表类似[b'1 82923', b'2 2184', ...]
print(mails)

# 获取最新一封邮件, 注意索引号从1开始:
index = len(mails)
resp, lines, octets = server.retr(index)

# lines存储了邮件的原始文本的每一行,
# 可以获得整个邮件的原始文本:
msg_content = b'\r\n'.join(lines).decode('utf-8')
# 稍后解析出邮件:
msg = Parser().parsestr(msg_content)

# 可以根据邮件索引号直接从服务器删除邮件:
# server.delete(index)
# 关闭连接:
server.quit()
```

用POP3获取邮件其实很简单, 要获取所有邮件, 只需要循环使用 `retr()` 把每一封邮件内容拿到即可。真正麻烦的是把邮件的原始内容解析为可以阅读的邮件对象。

解析邮件

解析邮件的过程和上一节构造邮件正好相反, 因此, 先导入必要的模块:

```
from email.parser import Parser
from email.header import decode_header
from email.utils import parseaddr

import poplib
```

只需要一行代码就可以把邮件内容解析为 `Message` 对象:

```
msg = Parser().parsestr(msg_content)
```

但是这个 `Message` 对象本身可能是一个 `MIMEMultipart` 对象，即包含嵌套的其他 `MIMEBase` 对象，嵌套可能还不止一层。

所以我们要递归地打印出 `Message` 对象的层次结构：

```
# indent用于缩进显示:
def print_info(msg, indent=0):
    if indent == 0:
        for header in ['From', 'To', 'Subject']:
            value = msg.get(header, '')
            if value:
                if header=='Subject':
                    value = decode_str(value)
                else:
                    hdr, addr = parseaddr(value)
                    name = decode_str(hdr)
                    value = u'%s <%s>' % (name, addr)
                print('%s%s: %s' % (' ' * indent, header, value))
    if (msg.is_multipart()):
        parts = msg.get_payload()
        for n, part in enumerate(parts):
            print('%spart %s' % (' ' * indent, n))
            print('%s-----' % (' ' * indent))
            print_info(part, indent + 1)
    else:
        content_type = msg.get_content_type()
        if content_type=='text/plain' or content_type=='text/html':
            content = msg.get_payload(decode=True)
            charset = guess_charset(msg)
            if charset:
                content = content.decode(charset)
            print('%sText: %s' % (' ' * indent, content + '...'))
        else:
            print('%sAttachment: %s' % (' ' * indent, content_type))
```

邮件的Subject或者Email中包含的名字都是经过编码后的str，要正常显示，就必须decode：

```
def decode_str(s):
    value, charset = decode_header(s)[0]
    if charset:
        value = value.decode(charset)
    return value
```

`decode_header()` 返回一个list，因为像 `Cc`、`Bcc` 这样的字段可能包含多个邮件地址，所以解析出来的会有多个元素。上面的代码我们偷了个懒，只取了第一个元素。

文本邮件的内容也是str，还需要检测编码，否则，非UTF-8编码的邮件都无法正常显示：

```
def guess_charset(msg):
    charset = msg.get_charset()
    if charset is None:
        content_type = msg.get('Content-Type', '').lower()
        pos = content_type.find('charset=')
        if pos >= 0:
            charset = content_type[pos + 8:].strip()
    return charset
```

把上面的代码整理好，我们就可以来试试收取一封邮件。先往自己的邮箱发一封邮件，然后用浏览器登录邮箱，看看邮件收到没，如果收到了，我们就来用Python程序把它收到本地：



Python可以使用POP3收取邮件.....

运行程序，结果如下：

```
+OK Welcome to coremail Mail Pop3 Server (163coms[...])
Messages: 126\. Size: 27228317

From: Test <xxxxxxx@qq.com>
To: Python爱好者 <xxxxxxx@163.com>
Subject: 用POP3收取邮件
part 0
-----
part 0
-----
Text: Python可以使用POP3收取邮件.....
part 1
-----
Text: Python可以<a href="...">使用POP3</a>收取邮件.....
```

```
part 1
```

```
-----
```

```
Attachment: application/octet-stream
```

我们从打印的结构可以看出，这封邮件是一个 `MIMEMultipart`，它包含两部分：第一部分又是一个 `MIMEMultipart`，第二部分是一个附件。而内嵌的 `MIMEMultipart` 是一个 `alternative` 类型，它包含一个纯文本格式的 `MIMEText` 和一个HTML格式的 `MIMEText`。

小结

用Python的 `poplib` 模块收取邮件分两步：第一步是用POP3协议把邮件获取到本地，第二步是用 `email` 模块把原始邮件解析为 `Message` 对象，然后，用适当的形式把邮件内容展示给用户即可。

参考源码

[fetch_mail.py](#)

十一、访问数据库

程序运行的时候，数据都是在内存中的。当程序终止的时候，通常都需要将数据保存到磁盘上，无论是保存到本地磁盘，还是通过网络保存到服务器上，最终都会将数据写入磁盘文件。

而如何定义数据的存储格式就是一个大问题。如果我们自己来定义存储格式，比如保存一个班级所有学生的成绩单：

```
| 名字 | 成绩 |
| Michael | 99 |
| Bob | 85 |
| Bart | 59 |
| Lisa | 87 |
```

你可以用一个文本文件保存，一行保存一个学生，用 `,` 隔开：

```
Michael,99
Bob,85
Bart,59
Lisa,87
```

你还可以用JSON格式保存，也是文本文件：

```
[
    {"name": "Michael", "score": 99},
    {"name": "Bob", "score": 85},
    {"name": "Bart", "score": 59},
    {"name": "Lisa", "score": 87}
]
```

你还可以定义各种保存格式，但是问题来了：

存储和读取需要自己实现，JSON还是标准，自己定义的格式就各式各样了；

不能做快速查询，只有把数据全部读到内存中才能自己遍历，但有时候数据的大小远远超过了内存（比如蓝光电影，40GB的数据），根本无法全部读入内存。

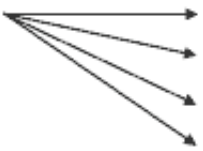
为了便于程序保存和读取数据，而且，能直接通过条件快速查询到指定的数据，就出现了数据库（Database）这种专门用于集中存储和查询的软件。

数据库软件诞生的历史非常久远，早在1950年数据库就诞生了。经历了网状数据库，层次数据库，我们现在广泛使用的关系数据库是20世纪70年代基于关系模型的基础上诞生的。

关系模型有一套复杂的数学理论，但是从概念上是十分容易理解的。举个学校的例子：

假设某个XX省YY市ZZ县第一实验小学有3个年级，要表示出这3个年级，可以在Excel中用一个表格画出来：

Grade_ID	Class_ID	Name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班
2	21	二年级一班
2	22	二年级二班
3	31	三年级一班
3	32	三年级二班
3	33	三年级三班
3	34	三年级四班



Class_ID	Num	Name	Score
11	10001	Michael	99
11	10002	Bob	85
11	10003	Bart	59
11	10004	Lisa	87
12	10005	Tracy	91
...	

每个年级又有若干个班级，要把所有班级表示出来，可以在Excel中再画一个表格：

Grade_ID	Class_ID	Name		Class_ID	Num	Name	Score
1	11	一年级一班	→	11	10001	Michael	99
1	12	一年级二班		11	10002	Bob	85
1	13	一年级三班		11	10003	Bart	59
2	21	二年级一班		11	10004	Lisa	87
2	22	二年级二班		12	10005	Tracy	91
3	31	三年级一班	
3	32	三年级二班					
3	33	三年级三班					
3	34	三年级四班					

这两个表格有个映射关系，就是根据Grade_ID可以在班级表中查找到对应的所有班级：

Grade_ID	Class_ID	Name		Class_ID	Num	Name	Score
1	11	一年级一班	→	11	10001	Michael	99
1	12	一年级二班		11	10002	Bob	85
1	13	一年级三班		11	10003	Bart	59
2	21	二年级一班		11	10004	Lisa	87
2	22	二年级二班		12	10005	Tracy	91
3	31	三年级一班	
3	32	三年级二班					
3	33	三年级三班					
3	34	三年级四班					

也就是Grade表的每一行对应Class表的多行，在关系数据库中，这种基于表（Table）的一对多的关系就是关系数据库的基础。

根据某个年级的ID就可以查找所有班级的行，这种查询语句在关系数据库中称为SQL语句，可以写成：

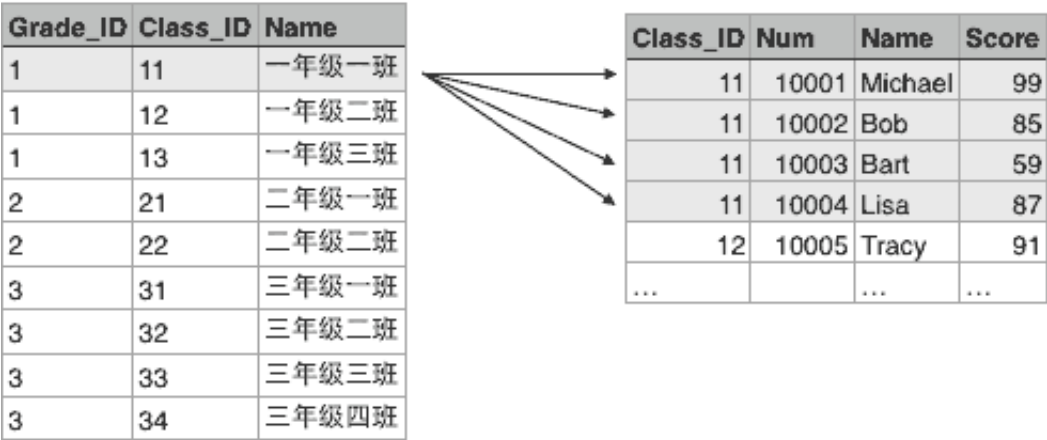
```
SELECT * FROM classes WHERE grade_id = '1';
```

结果也是一个表：

```
-----+-----+-----
grade_id | class_id | name
-----+-----+-----
1        | 11       | 一年级一班
-----+-----+-----
1        | 12       | 一年级二班
-----+-----+-----
1        | 13       | 一年级三班
```

-----+-----+-----

类似的，Class表的一行记录又可以关联到Student表的多行记录：



由于本教程不涉及到关系数据库的详细内容，如果你想从零学习关系数据库和基本的SQL语句，如果你想从零学习关系数据库和基本的SQL语句，请自行搜索相关课程。

NoSQL

你也许还听说过NoSQL数据库，很多NoSQL宣传其速度和规模远远超过关系数据库，所以很多同学觉得有了NoSQL是否就不需要SQL了呢？千万不要被他们忽悠了，连SQL都不明白怎么可能搞明白NoSQL呢？

数据库类别

既然我们要使用关系数据库，就必须选择一个关系数据库。目前广泛使用的关系数据库也就这么几种：

付费的商用数据库：

- Oracle，典型的高富帅；
- SQL Server，微软自家产品，Windows定制专款；
- DB2，IBM的产品，听起来挺高端；
- Sybase，曾经跟微软是好基友，后来关系破裂，现在家境惨淡。

这些数据库都是不开源而且付费的，最大的好处是花了钱出了问题可以找厂家解决，不过在Web的世界里，常常需要部署成千上万的数据库服务器，当然不能把大把大把的银子扔给厂家，所以，无论是Google、Facebook，还是国内的BAT，无一例外都选择了免费的开源数据库：

- MySQL，大家都在用，一般错不了；
- PostgreSQL，学术气息有点重，其实挺不错，但知名度没有MySQL高；
- sqlite，嵌入式数据库，适合桌面和移动应用。

作为Python开发工程师，选择哪个免费数据库呢？当然是MySQL。因为MySQL普及率最高，出了错，可以很容易找到解决方法。而且，围绕MySQL有一大堆监控和运维的工具，安装和使用很方便。

为了能继续后面的学习，你需要从MySQL官方网站下载并安装[MySQL Community Server 5.6](#)，这个版本是免费的，其他高级版本是要收钱的（请放心，收钱的功能我们用不上）。

1、使用SQLite

SQLite是一种嵌入式数据库，它的数据库就是一个文件。由于SQLite本身是C写的，而且体积很小，所以，经常被集成到各种应用程序中，甚至在iOS和Android的App中都可以集成。

Python就内置了SQLite3，所以，在Python中使用SQLite，不需要安装任何东西，直接使用。

在使用SQLite前，我们先要搞清楚几个概念：

表是数据库中存放关系数据的集合，一个数据库里面通常都包含多个表，比如学生的表，班级的表，学校的表，等等。表和表之间通过外键关联。

要操作关系数据库，首先需要连接到数据库，一个数据库连接称为Connection；

连接到数据库后，需要打开游标，称之为Cursor，通过Cursor执行SQL语句，然后，获得执行结果。

Python定义了一套操作数据库的API接口，任何数据库要连接到Python，只需要提供符合Python标准的数据库驱动即可。

由于SQLite的驱动内置在Python标准库中，所以我们可以直接来操作SQLite数据库。

我们在Python交互式命令行实践一下：

```
# 导入SQLite驱动:
>>> import sqlite3
# 连接到SQLite数据库
# 数据库文件是test.db
# 如果文件不存在，会自动在当前目录创建:
>>> conn = sqlite3.connect('test.db')
# 创建一个Cursor:
>>> cursor = conn.cursor()
```

```
# 执行一条SQL语句，创建user表：
>>> cursor.execute('create table user (id varchar(20) primary key, name varchar(20))')
<sqlite3.Cursor object at 0x10f8aa260>
# 继续执行一条SQL语句，插入一条记录：
>>> cursor.execute('insert into user (id, name) values (\ '1\ ', \ 'Michael\ ')')
<sqlite3.Cursor object at 0x10f8aa260>
# 通过rowcount获得插入的行数：
>>> cursor.rowcount
1
# 关闭Cursor：
>>> cursor.close()
# 提交事务：
>>> conn.commit()
# 关闭Connection：
>>> conn.close()
```

我们再试试查询记录：

```
>>> conn = sqlite3.connect('test.db')
>>> cursor = conn.cursor()
# 执行查询语句：
>>> cursor.execute('select * from user where id=?', ('1',))
<sqlite3.Cursor object at 0x10f8aa340>
# 获得查询结果集：
>>> values = cursor.fetchall()
>>> values
[('1', 'Michael')]
>>> cursor.close()
>>> conn.close()
```

使用Python的DB-API时，只要搞清楚 `Connection` 和 `Cursor` 对象，打开后一定记得关闭，就可以放心地使用。

使用 `Cursor` 对象执行 `insert`，`update`，`delete` 语句时，执行结果由 `rowcount` 返回影响的行数，就可以拿到执行结果。

使用 `Cursor` 对象执行 `select` 语句时，通过 `fetchall()` 可以拿到结果集。结果集是一个list，每个元素都是一个tuple，对应一行记录。

如果SQL语句带有参数，那么需要把参数按照位置传递给 `execute()` 方法，有几个 `?` 占位符就必须对应几个参数，例如：

```
cursor.execute('select * from user where name=? and pwd=?', ('abc', 'password'))
```

SQLite支持常见的标准SQL语句以及几种常见的数据类型。具体文档请参阅SQLite官方网站。

小结

在Python中操作数据库时，要先导入数据库对应的驱动，然后，通过 `Connection` 对象和 `Cursor` 对象操作数据。

要确保打开的 `Connection` 对象和 `Cursor` 对象都正确地被关闭，否则，资源就会泄露。

如何才能确保出错的情况下也关闭掉 `Connection` 对象和 `Cursor` 对象呢？请回忆 `try:...except:...finally:...` 的用法。

练习

请编写函数，在Sqlite中根据分数段查找指定的名字：

```
# -*- coding: utf-8 -*-

import os, sqlite3

db_file = os.path.join(os.path.dirname(__file__), 'test.db')
if os.path.isfile(db_file):
    os.remove(db_file)

# 初始数据:
conn = sqlite3.connect(db_file)
cursor = conn.cursor()
cursor.execute('create table user(id varchar(20) primary key, name varchar(20), score int)')
cursor.execute(r"insert into user values ('A-001', 'Adam', 95)")
cursor.execute(r"insert into user values ('A-002', 'Bart', 62)")
cursor.execute(r"insert into user values ('A-003', 'Lisa', 78)")
cursor.close()
conn.commit()
conn.close()

def get_score_in(low, high):
    ' 返回指定分数区间的名字，按分数从低到高排序 '
    ----
    pass
    ----
# 测试:
assert get_score_in(80, 95) == ['Adam'], get_score_in(80, 95)
```

```
assert get_score_in(60, 80) == ['Bart', 'Lisa'], get_score_in(60, 80)
assert get_score_in(60, 100) == ['Bart', 'Lisa', 'Adam'], get_score_in(60, 100)

print('Pass')
```

参考源码

[do_sqlite.py](#)

2、使用MySQL

MySQL是Web世界中使用最广泛的数据库服务器。SQLite的特点是轻量级、可嵌入，但不能承受高并发访问，适合桌面和移动应用。而MySQL是为服务器端设计的数据库，能承受高并发访问，同时占用的内存也远远大于SQLite。

此外，MySQL内部有多种数据库引擎，最常用的引擎是支持数据库事务的InnoDB。

安装MySQL

可以直接从MySQL官方网站下载最新的[Community Server 5.6.x](#)版本。MySQL是跨平台的，选择对应的平台下载安装文件，安装即可。

安装时，MySQL会提示输入 `root` 用户的口令，请务必记清楚。如果怕记不住，就把口令设置为 `password`。

在Windows上，安装时请选择 `UTF-8` 编码，以便正确地处理中文。

在Mac或Linux上，需要编辑MySQL的配置文件，把数据库默认的编码全部改为UTF-8。MySQL的配置文件默认存放在 `/etc/my.cnf` 或者 `/etc/mysql/my.cnf`：

```
[client]
default-character-set = utf8

[mysqld]
default-storage-engine = INNODB
character-set-server = utf8
collation-server = utf8_general_ci
```

重启MySQL后，可以通过MySQL的客户端命令行检查编码：

```
$ mysql -u root -p
Enter password:
```

```
Welcome to the MySQL monitor...
...

mysql> show variables like '%char%';
+-----+-----+
| Variable_name          | Value                                |
+-----+-----+
| character_set_client    | utf8                                |
| character_set_connection| utf8                                |
| character_set_database  | utf8                                |
| character_set_filesystem| binary                              |
| character_set_results   | utf8                                |
| character_set_server    | utf8                                |
| character_set_system    | utf8                                |
| character_sets_dir      | /usr/local/mysql-5.1.65-osx10.6-x86_64/share/charsets/ |
+-----+-----+
8 rows in set (0.00 sec)
```

看到 `utf8` 字样就表示编码设置正确。

注：如果MySQL的版本 $\geq 5.5.3$ ，可以把编码设置为 `utf8mb4`，`utf8mb4` 和 `utf8` 完全兼容，但它支持最新的Unicode标准，可以显示emoji字符。

安装MySQL驱动

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。MySQL官方提供了mysql-connector-python驱动，但是安装的时候需要给pip命令加上参数 `--allow-external`：

```
$ pip install mysql-connector-python --allow-external mysql-connector-python
```

如果上面的命令安装失败，可以试试另一个驱动：

```
$ pip install mysql-connector
```

我们演示如何连接到MySQL服务器的test数据库：

```
# 导入MySQL驱动:
>>> import mysql.connector
# 注意把password设为你的root口令:
>>> conn = mysql.connector.connect(user='root', password='password', database='test'
')
>>> cursor = conn.cursor()
# 创建user表:
>>> cursor.execute('create table user (id varchar(20) primary key, name varchar(20)
)')
# 插入一行记录，注意MySQL的占位符是%s:
>>> cursor.execute('insert into user (id, name) values (%s, %s)', ['1', 'Michael'])
>>> cursor.rowcount
1
# 提交事务:
>>> conn.commit()
>>> cursor.close()
# 运行查询:
>>> cursor = conn.cursor()
>>> cursor.execute('select * from user where id = %s', ('1',))
>>> values = cursor.fetchall()
>>> values
[('1', 'Michael')]
# 关闭Cursor和Connection:
>>> cursor.close()
True
>>> conn.close()
```

由于Python的DB-API定义都是通用的，所以，操作MySQL的数据库代码和SQLite类似。

小结

- 执行INSERT等操作后要调用 `commit()` 提交事务；
- MySQL的SQL占位符是 `%s` 。

参考源码

do_mysql.py

3、使用SQLAlchemy

数据库表是一个二维表，包含多行多列。把一个表的内容用Python的数据结构表示出来的话，可以用一个list表示多行，list的每一个元素是tuple，表示一行记录，比如，包含 id 和 name 的 user 表：

```
[
    ('1', 'Michael'),
    ('2', 'Bob'),
    ('3', 'Adam')
]
```

Python的DB-API返回的数据结构就是像上面这样表示的。

但是用tuple表示一行很难看出表的结构。如果把一个tuple用class实例来表示，就可以更容易地看出表的结构来：

```
class User(object):
    def __init__(self, id, name):
        self.id = id
        self.name = name

[
    User('1', 'Michael'),
    User('2', 'Bob'),
    User('3', 'Adam')
]
```

这就是传说中的ORM技术：Object-Relational Mapping，把关系数据库的表结构映射到对象上。是不是很简单？

但是由谁来做这个转换呢？所以ORM框架应运而生。

在Python中，最有名的ORM框架是SQLAlchemy。我们来看看SQLAlchemy的用法。

首先通过pip安装SQLAlchemy：

```
$ pip install sqlalchemy
```

然后，利用上次我们在MySQL的test数据库中创建的 `user` 表，用SQLAlchemy来试试：

第一步，导入SQLAlchemy，并初始化DBSession：

```
# 导入：
from sqlalchemy import Column, String, create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

# 创建对象的基类：
Base = declarative_base()

# 定义User对象：
class User(Base):
    # 表的名字：
    __tablename__ = 'user'

    # 表的结构：
    id = Column(String(20), primary_key=True)
    name = Column(String(20))

# 初始化数据库连接：
engine = create_engine('mysql+mysqlconnector://root:password@localhost:3306/test')
# 创建DBSession类型：
DBSession = sessionmaker(bind=engine)
```

以上代码完成SQLAlchemy的初始化和具体每个表的class定义。如果有多个表，就继续定义其他class，例如School：

```
class School(Base):
    __tablename__ = 'school'
    id = ...
    name = ...
```

`create_engine()` 用来初始化数据库连接。SQLAlchemy用一个字符串表示连接信息：

`'数据库类型+数据库驱动名称://用户名:口令@机器地址:端口号/数据库名'`

你只需要根据需要替换掉用户名、口令等信息即可。

下面，我们看看如何向数据库表中添加一行记录。

由于有了ORM，我们向数据库表中添加一行记录，可以视为添加一个 `User` 对象：

```
# 创建session对象：
session = DBSession()
# 创建新用户对象：
new_user = User(id='5', name='Bob')
# 添加到session：
session.add(new_user)
# 提交即保存到数据库：
session.commit()
# 关闭session：
session.close()
```

可见，关键是获取session，然后把对象添加到session，最后提交并关闭。`DBSession` 对象可视为当前数据库连接。

如何从数据库表中查询数据呢？有了ORM，查询出来的可以不再是tuple，而是 `User` 对象。SQLAlchemy提供的查询接口如下：

```
# 创建Session：
session = DBSession()
# 创建Query查询，filter是where条件，最后调用one()返回唯一行，如果调用all()则返回所有行：
user = session.query(User).filter(User.id=='5').one()
# 打印类型和对象的name属性：
print('type:', type(user))
print('name:', user.name)
# 关闭Session：
session.close()
```

运行结果如下：

```
type: <class '__main__.User'>
name: Bob
```

可见，ORM就是把数据库表的行与相应的对象建立关联，互相转换。

由于关系数据库的多个表还可以用外键实现一对多、多对多等关联，相应地，ORM框架也可以提供两个对象之间的一对多、多对多等功能。

例如，如果一个User拥有多个Book，就可以定义一对多关系如下：

```
class User(Base):
    __tablename__ = 'user'

    id = Column(String(20), primary_key=True)
    name = Column(String(20))
    # 一对多:
    books = relationship('Book')

class Book(Base):
    __tablename__ = 'book'

    id = Column(String(20), primary_key=True)
    name = Column(String(20))
    # “多”的一方的book表是通过外键关联到user表的:
    user_id = Column(String(20), ForeignKey('user.id'))
```

当我们查询一个User对象时，该对象的books属性将返回一个包含若干个Book对象的list。

小结

ORM框架的作用就是把数据库表的一行记录与一个对象互相做自动转换。

正确使用ORM的前提是了解关系数据库的原理。

参考源码

[do_sqlalchemy.py](#)