Python学习教程

Python是一种解释型、面向对象、动态数据类型的高级程序设计语言。和PHP一样，它是后端开发语言。如果有C语言、PHP语言、JAVA语言等其中一种语言的基础，学习Python入门很容易。本教程基于Python3。

目　录

致谢

Python学习—01入门

Python学习—01入门

Python是一种解释型、面向对象、动态数据类型的高级程序设计语言。和PHP一样，它是后端开发语言。

如果有C语言、PHP语言、JAVA语言等其中一种语言的基础，学习Python入门很容易。

Hello World!

python文件以 `.py` 结尾。
`hello.py`

```
#!/usr/bin/python
print("Hello, World!");
```

在命令行里运行(直接输入文件名即可):

```
$ chmod +x hello.py
$ ./hello.py
```

Windows:

```
hello.py
```

或者IDE里运行 `hello.py` 文件，将输出:

```
Hello, World!
```

这里注意:
1、单双引号都可以表示字符串，没有区别;
2、 `print` 在python3里是函数，要加括号;在2.6版本之前是语句，没有括号;2.6/2.7作为一个过渡版本，兼容两种写法。

环境搭建

在学习Python语言前我们需要本地搭建Python开发环境。

一般Mac系统和Linux服务器自带Python2.7版本。在命令行输入 `python` 可以看到相关信息：

```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

Python最新版是3.x系列，与2.x系列差别很大。其中2.6/2.7作为过渡版本，建议新学者使用，这样既可以兼容旧版本程序，也可以体验新版本特色。实际开发中，请合理选择。

Python下载：https://www.python.org/

Python安装：

1. Windows版本：官网下载python-2.7.11.msi安装即可。
2. Mac版本：详情见https://www.python.org/downloads/mac-osx/。
3. Linux版本：请选择源码自行编译或者使用其它安装方式。示例：

```
wget https://www.python.org/ftp/python/2.7.14/Python-2.7.14.tgz
tar zxf Python-2.7.14.tgz
cd Python-2.7.14
./configure
make && make install
```

安装好后按需配置环境变量：
在 **Unix/Linux** 设置环境变量

- 在 `csh shell` ：输入 `setenv PATH "$PATH:/usr/local/bin/python"` ，按下”Enter”。
- 在 `bash shell` ：输入 `export PATH="$PATH:/usr/local/bin/python"` ，按下”Enter”。
- 在 `sh` 或者 `ksh shell` ：输入 `PATH="$PATH:/usr/local/bin/python"` ，按下”Enter”。

注意： `/usr/local/bin/python` 是Python的安装目录。

在 **Windows** 设置环境变量
在环境变量中添加Python目录：
在命令提示框中(cmd) ： 输入 `path=%path%;C:\Python` 按下”Enter”。

注意： `C:\Python` 是Python的安装目录。

也可以通过以下方式设置：
1、右键点击”计算机”，然后点击”属性”

2、然后点击”高级系统设置”
3、选择”系统变量”窗口下面的”Path”,双击即可！
4、然后在”Path”行，添加python安装路径即可(我的 `D:\Python27`)，所以在后面，添加该路径即可。 ps：记住，路径直接用分号”；”隔开！
5、最后设置成功以后，在cmd命令行，输入命令”python”，就可以有相关显示。

运行Python

命令行交互方式

默认的，我们在命令行里运行 `python` 命令，变会进入Python命令行交互界面，以 `>>>` 开头：

```
1. >>> 5 + 5
2. 10
3. >>> print("hello World");
4. hello World
5. >>>
```

执行 `.py` 文件

我们还可以使用 `python test.py` 这样的方式直接运行文件。以最前面的 `hello.py` 为例，我们只需在命令行输入：

```
1. python hello.py
```

将输出：

```
1. Hello, World!
```

集成开发环境(IDE)

常见的有 `IDLE` 、 `PythonWin` 等。其实我们常见的 `Subline Text` 也可以通过插件打造成一款轻量级的python IDE。

支持中文

python2.x系列不支持在代码里含有中文，注释里也不例外。

Python中默认的编码格式是 ASCII 格式，在没修改编码格式时无法正确打印汉字，所以在读取中文时会报错。

解决方法为只要在文件开头加入 `# -*- coding: UTF-8 -*-` 或者 `#coding=utf-8` 就行了。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

print "你好，世界";
```

注意：Python3.X 源码文件默认使用utf-8编码，所以可以正常解析中文，无需指定 UTF-8 编码。

Python注释

python中单行注释采用 # 开头。

python 中多行注释使用三个单引号(’’’)或三个双引号(“””)。

```
'''
这是多行注释
'''
```

Python 标识符

- 在python里，标识符有 字母、数字、下划线 组成。
- 在python中，所有标识符可以包括英文、数字以及下划线（_），但不能以数字开头。
- python中的标识符是 区分大小写 的。
- 以下划线开头的标识符是有特殊意义的。
  1）以单下划线开头（ `_foo` ）的代表不能直接访问的类属性，需通过类提供的接口进行访问，不能用 `from xxx import *` 而导入；
  2）以双下划线开头的（ `__foo` ）代表类的私有成员；以双下划线开头和结尾的（ `__foo__` ）代表python里特殊方法专用的标识，如 `__init__()` 代表类的构造函数。

Python保留字符

Python中和C语言一样也预留了很多保留字。这些保留字不能用作常数或变数，或任何其他标识符名称。

```
and    exec    not
assert    finally    or
break    for    pass
class    from    print
```

```
continue    global    raise
def    if    return
del    import    try
elif    in    while
else    is    with
except    lambda    yield
```

行和缩进

学习**Python**与其他语言最大的区别就是，**Python**的代码块不使用大括号 `{}` 来控制类，函数以及其他逻辑判断。**python**最具特色的就是用缩进来写模块。

缩进的空白数量是可变的，但是所有代码块语句必须包含相同的缩进空白数量，这个必须严格执行。如下所示：

```
if True:
    print "True"
else:
  print "False"
```

建议你在每个缩进层次使用 单个制表符 或 两个空格 或 四个空格 ， 切记不能混用。

Python 引号

Python 接收单引号 `'` ，双引号 `"` ，三引号 `''' """` 来表示字符串，引号的开始与结束必须的相同类型的。

其中三引号可以由多行组成，编写多行文本的快捷语法，常用于文档字符串，在文件的特定地点，被当做注释。

```
#/usr/bin/python

print('''line1
line2
line3''')
```

多行语句

Python语句中一般以新行作为为语句的结束符。
但是我们可以使用斜杠（ \）将一行的语句分为多行显示，如下所示：

```
total = item_one + \
        item_two + \
        item_three
```

语句中包含[], {} 或 () 括号就不需要使用多行连接符。如下实例：

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```

Python空行

空行的作用在于分隔两段不同功能或含义的代码，便于日后代码的维护或重构。空行与代码缩进不同，空行并不是Python语法的一部分。

同一行显示多条语句

Python可以在同一行中使用多条语句，语句之间使用分号(;)分割，以下是一个简单的实例：

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

Python学习—02输入和输出、运算符

命令行输入

```
1.  x = input("Please input x:")
2.  y = raw_input("Please input x:")
```

使用 `input` 和 `raw_input` 都可以读取控制台的输入，但是input和raw_input在处理数字时是有区别的。 `raw_input()` 将所有输入作为字符串看待，返回字符串类型；而 `input()` 在对待纯数字输入时具有自己的特性，它返回所输入的数字的类型（int, float），`input()` 可接受合法的 python 表达式。

看python input的文档，可以看到 `input()` 本质上还是使用 `raw_input()` 来实现的，只是调用完 `raw_input()` 之后再调用 `eval()` 函数，所以，你甚至可以将表达式作为 `input()` 的参数，并且它会计算表达式的值并返回它。

```
1.  def input(prompt):
2.      return (eval(raw_input(prompt)))
```

除非对 `input()` 有特别需要，否则一般情况下我们都是推荐使用 `raw_input()` 来与用户交互。

输出

Python两种输出值的方式： 表达式语句和 `print()` 函数。（第三种方式是使用文件对象的 `write()` 方法； 标准输出文件可以用 `sys.stdout` 引用。）

print

示例：

```
1.  print "Hello, Python!";
2.  print ("Hello, Python!"); #新版本的Python
```

输出的 **print** 函数总结：

1. 字符串和数值类型
   可以直接输出：

```
>>> print(1)
1
>>> print("Hello World")
Hello World
```

2.变量

无论什么类型，数值，布尔，列表，字典…都可以直接输出

```
>>> x = 12
>>> print(x)
12
>>> s = 'Hello'
>>> print(s)
Hello
>>> L = [1,2,'a']
>>> print(L)
[1, 2, 'a']
>>> t = (1,2,'a')
>>> print(t)
(1, 2, 'a')
>>> d = {'a':1, 'b':2}
>>> print(d)
{'a': 1, 'b': 2}
```

3.格式化输出

类似于C中的 printf

```
>>> s
'Hello'
>>> x = len(s)
>>> print("The length of %s is %d" % (s,x))
The length of Hello is 5
```

Python中格式化输出的总结:

(1) `%` 字符: 标记转换说明符的开始

(2) 转换标志: `-` 表示左对齐; `+` 表示在转换值之前要加上正负号; `""（空白字符）` 表示正数之前保留空格; `0` 表示转换值若位数不够则用0填充。示例:

```
# 指定占位符宽度（左对齐）
>>> print ("Name:%-10s Age:%-8d Height:%-8.2f"%("Aviad",25,1.83))
```

```
Name:Aviad      Age:25       Height:1.83

# 指定占位符（若位数不够则用0填充）
>>> print ("Name:%-10s Age:%08d Height:%08.2f"%("Aviad",25,1.83))
Name:Aviad      Age:00000025 Height:00001.83
```

(3) 最小字段宽度：转换后的字符串至少应该具有该值指定的宽度。如果是*，则宽度会从值元组中读出。

```
# 指定占位符宽度
>>> print ("Name:%10s Age:%8d Height:%8.2f"%("Aviad",25,1.83))
Name:     Aviad Age:      25 Height:    1.83
```

(4) 点(.)后跟精度值：如果转换的是实数，精度值就表示出现在小数点后的位数。如果转换的是字符串，那么该数字就表示最大字段宽度。如果是*，则从后面的元组中读取字段宽度或精度。

```
>>> print ("His height is %f m"%(1.83))
His height is 1.830000 m

>>> print ("His height is %.2f m"%(1.83))
His height is 1.83 m

>>> print ("The String is %.2s"%("abcd"))
The String is ab

# 用*从后面的元组中读取字段宽度或精度，第1个参数是精度
>>> print ("His height is %.*f m"%(2,1.83))
His height is 1.83 m
```

(5) 字符串格式化转换类型

```
转换类型          含义
d,i                带符号的十进制整数
o                  不带符号的八进制
u                  不带符号的十进制
x                   不带符号的十六进制（小写）
X                  不带符号的十六进制（大写）
e                  科学计数法表示的浮点数（小写）
E                  科学计数法表示的浮点数（大写）
f,F                十进制浮点数
g                  如果指数大于-4或者小于精度值则和e相同，其他情况和f相同
```

```
G                   如果指数大于-4或者小于精度值则和E相同，其他情况和F相同
C                   单字符（接受整数或者单字符字符串）
r                     字符串（使用repr转换任意python对象）
s                    字符串（使用str转换任意python对象）
```

拼接字符串

```
a = 'hello '
b = 'world'

>>> a+b
'hello world'
```

查看变量类型

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

部分函数

math开头需要 `import math` 。

```
str(object) 把值转换为字符串
repr(object) 返回值的字符串标示形式

abs(number) 返回数字的绝对值
cmath.sqrt(number) 返回平方根，也可以应用于负数
float(object) 把字符串和数字转换为浮点数
help() 提供交互式帮助
input(prompt) 获取用户输入
int(object) 把字符串和数字转换为整数
math.ceil(number) 返回数的上入整数，返回值的类型为浮点数
math.floor(number) 返回数的下舍整数，返回值的类型为浮点数
math.sqrt(number) 返回平方根不适用于负数
pow(x,y[.z]) 返回X的y次幂（有z则对z取模）

round(number[.ndigits]) 根据给定的精度对数字进行四舍五入
```

str.format() 的基本使用如下：

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

括号及其里面的字符 (称作格式化字段) 将会被 format() 中的参数替换。

自定义打印对象函数：

```
def prn_obj(obj):
print ', '.join(['%s:%s' % item for item in obj.__dict__.items()])
```

JSON转换

json类里提供

```
json.dumps(param) #list转json
json.loads(param) #json转list
```

示例：

```
>>> import json
>>> json.dumps(['math','english'])
'["math", "english"]'

>>> json.loads('["math", "english"]')
[u'math', u'english']
```

json主要用在PHP的array对象 和 python的list对象上。

PHP和Python3能将同样的json还原成 各自的object 且 在各自的语言环境下代表的意义是同样的。

但是 PHP和python将object生成json的时候，却不太一样了，PHP生成的json中多了反斜线。

打开文件

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

```

```
# 打开文件
fo = open("runoob.txt", "r+")
print "文件名为: ", fo.name

line = fo.read(10)
print "读取的字符串: %s" % (line)

# 关闭文件
fo.close()
```

运算符

Python支持:

- 算数运算符
- 关系运算符
- 赋值运算符
- 逻辑运算符
- 位运算符

除了以上的一些运算符之外，Python还支持成员运算符，身份运算符:

- 成员运算符
- 身份运算符

算术运算符

以下假设变量a为10，变量b为20:

运算符	描述	实例
+	加 - 两个对象相加	a + b 输出结果 30
-	减 - 得到负数或是一个数减去另一个数	a - b 输出结果 -10
	乘 - 两个数相乘或是返回一个被重复若干次的字符串	a b 输出结果 200
/	除 - x除以y	b / a 输出结果 2
%	取模 - 返回除法的余数	b % a 输出结果 0
	幂 - 返回x的y次幂	ab 为10的20次方， 输出结果 100000000000000000000
//	取整除 - 返回商的整数部分	9//2 输出结果 4 , 9.0//2.0 输出结果 4.0

Python算术运算符没有C语言里的自增(`++`)自减(`--`)运算符。

关系运算符

以下假设变量a为10，变量b为20:

运算符	描述	实例
==	等于 - 比较对象是否相等	(a == b) 返回 False。
!=	不等于 - 比较两个对象是否不相等	(a != b) 返回 true.
<>	不等于 - 比较两个对象是否不相等	(a <> b) 返回 true。这个运算符类似 != 。
>	大于 - 返回x是否大于y	(a > b) 返回 False。
<	小于 - 返回x是否小于y。所有比较运算符返回1表示真，返回0表示假。这分别与特殊的变量True和False等价。注意，这些变量名的大写。	(a < b) 返回 true。
>=	大于等于 - 返回x是否大于等于y。	(a >= b) 返回 False。
<=	小于等于 - 返回x是否小于等于y。	(a <= b) 返回 true。

赋值运算符

以下假设变量a为10，变量b为20:

运算符	描述	实例
=	简单的赋值运算符	c = a + b 将 a + b 的运算结果赋值为 c
+=	加法赋值运算符	c += a 等效于 c = c + a
-=	减法赋值运算符	c -= a 等效于 c = c - a
=	乘法赋值运算符	c = a 等效于 c = c *a*
/=	除法赋值运算符	c /= a 等效于 c = c / a
%=	取模赋值运算符	c %= a 等效于 c = c % a
=	幂赋值运算符	c = a 等效于 c = c * a
//=	取整除赋值运算符	c //= a 等效于 c = c // a

逻辑运算符

Python语言支持逻辑运算符。

在Python中是没有 `&&` 、 `||` 、 `!` 这三个运算符的，取而代之的是英文 `and` 、 `or` 、 `not` 。

以下假设变量 a 为 10, b为 20:

运算符	逻辑表达式	描述	实例
and	x and y	布尔"与" - 如果 x 为 False，x and y 返回 False，否则它返回 y 的计算值。	(a and b) 返回 20
or	x or y	布尔"或" - 如果 x 是非 0，它返回 x 的值，否则它返回 y 的计算值。	(a or b) 返回 10。
not	not x	布尔"非" - 如果 x 为 True，返回 False 。如果 x 为 False，它返回 True。	not(a and b) 返回 False

位运算符

按位运算符是把数字看作二进制来进行计算的。Python中的按位运算法则如下：
下表中变量 a 为 60，b 为 13，二进制格式如下：

```
a = 0011 1100

b = 0000 1101

-----------------

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a  = 1100 0011
```

运算符	描述	实例
&	按位与运算符：参与运算的两个值,如果两个相应位都为1,则该位的结果为1,否则为0	(a & b) 输出结果 12 ，二进制解释: 0000 1100
|	按位或运算符：只要对应的二个二进位有一个为1时，结果位就为1。	(a | b) 输出结果 61 ，二进制解释: 0011 1101
^	按位异或运算符：当两对应的二进位相异时，结果为1	(a ^ b) 输出结果 49 ，二进制解释: 0011 0001
~	按位取反运算符：对数据的每个二进制位取反,即把1变为0,把0变为1	(~a) 输出结果 -61 ，二进制解释: 1100 0011， 在一个有符号二进制数的补码形式。
<<	左移动运算符：运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0。	a << 2 输出结果 240 ，二进制解释: 1111 0000
>>	右移动运算符：把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数	a >> 2 输出结果 15 ，二进制解释: 0000 1111

成员运算符

以下假设变量 a 为 1，b为 20，c为 `[1, 2, 3, 4, 5 ]` ：

运算符	描述	实例
in	如果在指定的序列中找到值返回 True，否则返回 False。	`(a in c)`，返回 True。
not in	如果在指定的序列中没有找到值返回 True，否则返回 False。	`(b not in c)` ，返回 True。

身份运算符

身份运算符用于比较两个对象的存储单元。

运算符	描述	实例
is	is是判断两个标识符是不是引用自一个对象	x is y, 如果 id(x) 等于 id(y) , is 返回结果 1
is not	is not是判断两个标识符是不是引用自不同对象	x is not y, 如果 id(x) 不等于 id(y). is not 返回结果 1

运算符优先级

运算符	描述
	指数 (最高优先级)
`~ + -`	按位翻转, 一元加号和减号 (最后两个的方法名为 +@ 和 -@)
`* / % //`	乘，除，取模和取整除
`+ -`	加法减法
`>> <<`	右移，左移运算符
&	位 ‘AND’
`^ |`	位运算符
<= < > >=	比较运算符
<> == !=	等于运算符
`= %= /= //= -= += = =`	赋值运算符
is is not	身份运算符
in not in	成员运算符
not or and	逻辑运算符

> 参考:
> 1、Python 3 语法小记（一）入门 (print 函数用法总结) - Just Coding！ - 博客频道 - CSDN.NET
> http://blog.csdn.net/jcjc918/article/details/9354815
> 2、Python格式化输出 - MindProbe - 博客园

> http://www.cnblogs.com/plwang1990/p/3757549.html
> 3、Python3 的json 和 PHP的json - 宁静的天空 - 博客园
> http://www.cnblogs.com/ribavnu/p/4850413.html
>
> 作者： 飞鸿影
>
> 出处：https://www.cnblogs.com/52fhy/p/6235133.html

Python学习—03变量类型

变量赋值

- Python中的变量不需要声明，变量的赋值操作既是变量声明和定义的过程。
- 每个变量在内存中创建，都包括变量的标识，名称和数据这些信息。
- 每个变量在使用前都必须赋值，变量赋值以后该变量才会被创建。
- 等号（=）用来给变量赋值。
- 等号（=）运算符左边是一个变量名,等号（=）运算符右边是存储在变量中的值。

例如:

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

counter = 100 # 赋值整型变量
miles = 1000.0 # 浮点型
name = "John" # 字符串

print(counter)
print(miles)
print(name)
```

多个变量赋值

Python允许你同时为多个变量赋值。例如:

```
a = b = c = 1
```

以上实例，创建一个整型对象，值为1，三个变量被分配到相同的内存空间上。
您也可以为多个对象指定多个变量。例如:

```
a, b, c = 1, 2, "john"
```

以上实例，两个整型对象1和2的分配给变量a和b，字符串对象”john”分配给变量c。

常量

所谓常量就是不能变的变量，比如常用的数学常数 `π` 就是一个常量。通常用大写字母表示常量。示例：

```
PI = 3.14159265359
```

与PHP、JAVA的常量不一样，Python本身并没有提供常量这一机制。也就是说，在Python里，用全部大写的变量名表示常量只是一个习惯上的用法，如果你一定要改变变量 `PI` 的值，也没人能拦住你。

数据类型

Python提供的基本数据类型主要有：布尔类型、整型、浮点型、字符串、列表、元组、集合、字典等等。

Python有五个标准的数据类型：

- Numbers（数字）
- String（字符串）
- List（列表）
- Tuple（元组）
- Dictionary（字典）

数字(number)

Python数字类型是不可改变的数据类型。

Python支持四种不同的数字类型：

- int（有符号整型）
- long（长整型[也可以代表八进制和十六进制]）
- float（浮点型）
- complex（复数）

```
# 整数
int=20;

# 浮点数
float=2.3;

print(int)
print(float)
print(pow(2,5))
```

```
print(2**5)
```

输出：

```
20
2.3
32
32
```

字符串(str)

a、使用单引号(‘)
用单引号括起来表示字符串，例如：

```
str='this is string';
print(str);
```

b、使用双引号(“)
双引号中的字符串与单引号中的字符串用法完全相同，例如：

```
str="this is string";
print(str);
```

c、使用三引号(‘’’)
利用三引号，表示多行的字符串，可以在三引号中自由的使用单引号和双引号，例如：

```
str='''this is string
this is python string
this is string'''

print(str);
```

当使用以冒号分隔的字符串，python返回一个新的对象，结果包含了以这对偏移标识的连续的内容，左边的开始是包含了下边界。

加号（+）是字符串连接运算符，星号（*）是重复操作。如下实例：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

```

```
str = 'Hello World!'

print(str)           # 输出完整字符串
print(str[0])        # 输出字符串中的第一个字符
print(str[2:5])      # 输出字符串中第三个至第五个之间的字符串
print(str[2:])       # 输出从第三个字符开始的字符串
print(str * 2)       # 输出字符串两次
print(str + "TEST")  # 输出连接的字符串
```

以上实例输出结果：

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

转义字符 `\` 可以转义很多字符，比如 `\n` 表示换行， `\t` 表示制表符，字符 `\` 本身也要转义，所以 `\\` 表示的字符就是 `\` ，可以在Python的交互式命令行用 `print()` 打印字符串看看：

```
>>> print('I\'m ok.')
I'm ok.
```

如果字符串里面有很多字符都需要转义，就需要加很多 `\` ，为了简化，Python还允许用 `r''` 表示 `''` 内部的字符串默认不转义，可以自己试试：

```
>>> print('\\\t\\')
\       \
>>> print(r'\\\t\\')
\\\t\\
```

字符串通过 `encode()` 方法可以编码为指定的bytes，例如：

```
>>> 'ABC'.encode('ascii')
b'ABC'
>>> '中文'.encode('utf-8')
b'\xe4\xb8\xad\xe6\x96\x87'
>>> '中文'.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
    UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1:
8.  ordinal not in range(128)
```

纯英文的str可以用ASCII编码为bytes，内容是一样的，含有中文的str可以用UTF-8编码为bytes。含有中文的str无法用ASCII编码，因为中文编码的范围超过了ASCII编码的范围，Python会报错。

在bytes中，无法显示为ASCII字符的字节，用 `\x##` 显示。

反过来，如果我们从网络或磁盘上读取了字节流，那么读到的数据就是bytes。要把bytes变为str，就需要用 `decode()` 方法：

```
1.  >>> b'ABC'.decode('ascii')
2.  'ABC'
3.  >>> b'\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')
4.  '中文'
```

可以使用 `str()` 将其它类型转为字符串类型。

列表(list)

List（列表） 是 Python 中使用最频繁的数据类型。
列表可以完成大多数集合类的数据结构实现。它支持字符，数字，字符串甚至可以包含列表（所谓嵌套）。
列表用 `[ ]` 标识。

列表中使用切片操作符 `[头下标:尾下标]` ，就可以截取相应的列表，从左到右索引默认0开始的，从右到左索引默认-1开始，下标可以为空表示取到头或尾。

加号（+）是列表连接运算符，星号（*）是重复操作。如下实例：

```
1.  #!/usr/bin/python
2.  # -*- coding: UTF-8 -*-
3.
4.  list = [ 'Hello', 786 , 2.23, 'john', 70.2 ]
5.  tinylist = [123, 'john']
6.
7.  print(list)             # 输出完整列表
8.  print(list[0])            # 输出列表的第一个元素
9.  print(list[1:3])          # 输出第二个至第三个的元素
10. print(list[2:])           # 输出从第三个开始至列表末尾的所有元素
11. print(tinylist * 2)       # 输出列表两次
```

```
print(list + tinylist)    # 打印组合的列表
```

以上实例输出结果：

```
['Hello', 786, 2.23, 'john', 70.2]
Hello
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['Hello', 786, 2.23, 'john', 70.2, 123, 'john']
```

删除列表元素

```
L=['spam', 'Spam', 'SPAM!'];
del L[0]
```

列表函数&方法

```
len(list) 列表长度
list.append(obj) 在列表末尾添加新的对象，相当于其它语言里的push
list.count(obj) 统计某个元素在列表中出现的次数
list.extend(seq) 在列表末尾一次性追加另一个序列中的多个值(用新列表扩展原来的列表)
list.index(obj) 从列表中找出某个值第一个匹配项的索引位置，索引从0开始
list.insert(index, obj) 将对象插入列表
list.pop(obj=list[-1]) 移除列表中的一个元素(默认最后一个元素)，并且返回该元素的值
list.remove(obj) 移除列表中某个值的第一个匹配项
list.reverse() 反向列表中元素，倒转
list.sort([func]) 对原列表进行排序
```

元组(tuple)

元组是另一个数据类型，类似于List（列表）。
元组用 `()` 标识。内部元素用逗号隔开。但是元组不能二次赋值，相当于只读列表。

```
tup = ();
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = "a", "b", "c", "d";
```

元组中只有一个元素时，需要在元素后面添加逗号，例如： `tup1 = (50,);`

元组与字符串类似，下标索引从0开始，可以进行截取，组合等。

元组中的元素值是不允许修改的，但我们可以对元组进行连接组合，例如：

```
tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');

# 以下修改元组元素操作是非法的。
# tup1[0] = 100;
```

元组中的元素值是不允许删除的，可以使用del语句来删除整个元组。

元组内置函数

```
cmp(tuple1, tuple2) 比较两个元组元素。
len(tuple) 计算元组元素个数。
max(tuple) 返回元组中元素最大值。
min(tuple) 返回元组中元素最小值。
tuple(seq) 将列表转换为元组。
```

字典(dict)

字典是一种无序存储结构，包括关键字（key）和关键字对应的值（value）。字典的格式为：`dictionary = {key:value}` 。key为不可变类型，如字符串、整数、只包含不可变对象的元组，列表等不可作为关键字。字典也被称作关联数组或哈希表。

示例：

```
>>> dict = {'name': 'Zara', 'age': 7, 'class': 'First'};
```

字典操作：

```
# 访问
>>> dict['name']
'Zara'
>>> dict['age']
7

# 修改
>>> dict['name'] = 'yjc'
>>> dict['name']
```

```
'yjc'
>>> dict["school"]="wutong"
>>> dict["school"]
'wutong'

# 删除
del dict['name'];  # 删除键是'name'的条目
dict.clear();  # 清空词典所有条目
del dict ;  # 删除词典
```

注意：字典不存在，`del` 会引发一个异常。

字典内置函数方法：

```
cmp(dict1, dict2) 比较两个字典元素。
len(dict) 计算字典元素个数，即键的总数。
str(dict) 输出字典可打印的字符串表示。
type(variable) 返回输入的变量类型，如果变量是字典就返回字典类型。
dict.clear() 删除字典内所有元素
dict.copy() 返回一个字典的浅复制
dict.fromkeys() 创建一个新字典，以序列seq中元素做字典的键，val为字典所有键对应的初始值
dict.get(key, default=None) 返回指定键的值，如果值不在字典中返回default值
dict.has_key(key) 如果键在字典dict里返回true，否则返回false
dict.items() 以列表返回可遍历的(键, 值) 元组数组
dict.keys() 以列表返回一个字典所有的键
dict.setdefault(key, default=None) 和get()类似, 但如果键不已经存在于字典中，将会添加键并将值设为default
dict.update(dict2) 把字典dict2的键/值对更新到dict里
dict.values() 以列表返回字典中的所有值
```

集合(set)

集合(set)和字典(dict)类似，也是一组key的集合，但不存储**value**。由于key不能重复，所以，在set中，没有重复的key。

要创建一个set，需要提供一个list作为输入集合：

```
>>> s = set([1, 2, 3])
>>> s
{1, 2, 3}
```

注意，传入的参数 `[1, 2, 3]` 是一个list，而显示的 `{1, 2, 3}` 只是告诉你这个set内部有 `1, 2, 3` 这3个元素，显示的顺序也不表示set是有序的。。

重复元素在set中自动被过滤：

```
>>> s = set([1, 1, 2, 2, 3, 3])
>>> s
{1, 2, 3}
```

set可以看成数学意义上的无序和无重复元素的集合，因此，两个set可以做数学意义上的交集、并集等操作：

```
>>> s1 = set([1, 2, 3])
>>> s2 = set([2, 3, 4])
>>> s1 & s2
{2, 3}
>>> s1 | s2
{1, 2, 3, 4}
```

set 常用方法：

```
s.add(key)
s.remove(key)
```

布尔类型(bool)

在Python中，None、任何数值类型中的 `0` 、空字符串 `""` 、空元组 `()` 、空列表 `[]` 、空字典 `{}` 都被当作 `False` ，还有自定义类型，如果实现了 `__nonzero__()` 或 `__len__()` 方法且方法返回 `0` 或 `False` ，则其实例也被当作 `False` ，其他对象均为 `True` 。

布尔值和布尔代数的表示完全一致，一个布尔值只有 `True` 、 `False` 两种值，要么是 `True` ， `False` ，在Python中，可以直接用 `True` 、 `False` （请注意大小写），也可以通过布尔运算计算出来：

```
>>> True
True
>>> False
False
>>> 3 > 2
True
>>> 3 > 5
```

```
False
```

布尔值可以用and(与)、or(或)和not(非)运算（逻辑运算符，并不是位运算符）：

```
a = 10
b = 20
a and b  # 20
a or b # 10
not a # False
not b # False
```

空值(None)

表示该值是一个空对象，空值是Python里一个特殊的值，用 `None` 表示。 `None` 不能理解为0，因为0是有意义的，而 `None` 是一个特殊的空值。

参考

1、python数据类型详解 - Ruthless - 博客园
http://www.cnblogs.com/linjiqin/p/3608541.html
2、Python 变量类型 | 菜鸟教程
http://www.runoob.com/python/python-variable-types.html
3、数据类型和变量
http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/001431658624177ea4f8fcb06bc4d0e8aab2fd7aa65dd95000
4、使用list和tuple
http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/00143167247729045211421 96b74a3f8abf93d8e97c6ee6000

Python学习—04条件控制与循环结构

Python学习—04条件控制与循环结构

支持

- `if` , `if...else` , `if...elif ...if`
- `while`
- `for ... in...`
- `continue` , `break`
- `pass`

没有 `switch-case` ；没有普通的 `for x;y;z` 条件循环。

条件控制

在Python程序中，用if语句实现条件控制。

语法格式：

```
if <条件判断1>:
    <执行1>
elif <条件判断2>:
    <执行2>
elif <条件判断3>:
    <执行3>
else:
    <执行4>
```

注意语句后面的冒号 `:` 。像经典的C、Java都是以花括号来区分代码块，但是Python没有使用花括号表示，而是缩进，所以一定需要了解它们的语法区别。

示例：

```
age = 3
if age >= 18:
    print('adult')
elif age >= 6:
    print('teenager')
else:
```

```
7.      print('kid')
```

循环控制

Python里有2种循环结构：

1、for…in

2、while

注意Python里没有C语言里经典的for循环结构，也没有PHP里的foreach结构。

for…in

for…in循环会依次把list或tuple中的每个元素迭代出来，示例：

```
1.  names = ['Michael', 'Bob', 'Tracy']
2.  for name in names:
3.      print(name)
```

输出：

```
1.  Michael
2.  Bob
3.  Tracy
```

注意for语句后面的冒号 `:` 。

再看个求和的例子：

```
1.  sum = 0
2.  for x in range(101):
3.      sum = sum + x
4.  print(sum)
```

输出：

```
1.  5050
```

注意的是， `range(101)` 生成的是0-100的整数序列，不是到101。

对于字典(dict)，for…in循环迭代的是key，而不是value：

```
dict = {"name":"yjc", "age":18}
for x in dict:
    print(x, dict[x])
```

输出:

```
name yjc
age 18
```

while

while循环是其它语言里很经典的循环结构，Pyhton里同样支持。

```
sum = 0
n = 0
while n < 101:
    sum = sum + n
    n = n + 1
print(sum)
```

while循环里只要条件满足，就不断循环，条件不满足时退出循环。需要注意while语句后面的冒号 `:` 。

循环控制语句

循环里如果我们想终止本次循环，可以使用 `continue` ；如果想终止整个循环，则使用 `break` 。

看看下面这个例子:

```
sum = 0
n = 0
while n < 5:
    n = n + 1
    if n == 3:
        break #试试替换成continue
    sum = sum + n
print(sum)
```

输出:

```
# 使用break:
3

# 使用continue:
12
```

空语句

Python里使用 `pass` 表示空语句，即啥也不做。

```
if age >= 18:
    pass
```

在C语言里等同于:

```
if( age>=18 ){

}
```

pass语句什么都不做，那有什么用？实际上pass可以用来作为占位符，比如现在还没想好怎么写函数的代码，就可以先放一个pass，让代码能运行起来。

因为在其它语言里有花括号，如果花括号里面为空，代表啥也不做，但Python没有花括号，缺少了pass，代码运行就会有语法错误。

Switch/Case模拟

Python没有switch-case,过去写C习惯用Switch/Case语句，官方文档说通过if-elif实现。所以不妨自己来实现Switch/Case功能。

1、通过字典实现

```
def foo(var):
    return {
            'a': 1,
            'b': 2,
            'c': 3,
    }.get(var,'error')    #'error'为默认返回值，可自设置
```

2、通过匿名函数实现

```
def foo(var,x):
    return {
            'a': lambda x: x+1,
            'b': lambda x: x+2,
            'c': lambda x: x+3,
    }[var](x)
```

参考

1、循环
http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/001431676242561226b32a9ec624505bb8f723d0027b3e7000
2、python中Switch/Case实现 - gerrydeng - 博客园
https://www.cnblogs.com/gerrydeng/p/7191927.html

Python学习—05函数

Python函数

函数是组织好的，可重复使用的，用来实现单一，或相关联功能的代码段。
函数能提高应用的模块性，和代码的重复利用率。我们已经知道Python提供了许多内建函数，比如 `print()` 。但我们也可以自己创建函数，这被叫做用户自定义函数。

定义一个函数

我们可以定义一个由自己想要功能的函数，以下是简单的规则：

- 函数代码块以 `def` 关键词开头，后接 `函数标识符名称` 和 `圆括号()` 。
- 任何传入参数和自变量必须放在圆括号中间。圆括号之间可以用于 `定义参数` 。
- 函数的 `第一行语句` 可以选择性地使用文档字符串—用于存放 `函数说明` 。
- 函数内容以 `冒号` 起始，并且缩进。
- `Return[expression]` 结束函数，选择性地返回一个值给调用方。不带表达式的return相当于返回 `None` 。

语法

```
def functionname( parameters ):
   "函数_文档字符串"
   function_suite
   return [expression]
```

默认情况下，参数值和参数名称是按函数声明中定义的的顺序匹配起来的。

实例

以下为一个简单的Python函数，它将一个字符串作为传入参数，再打印到标准显示设备上。

```
def printme( str ):
   "打印传入的字符串到标准显示设备上"
   print str
   return
```

函数调用

定义一个函数只给了函数一个名称，指定了函数里包含的参数，和代码块结构。

这个函数的基本结构完成以后，你可以通过另一个函数调用执行，也可以直接从Python提示符执行。

Python里不用像C语言里需要先申明函数再使用。

如下实例调用了 `printme()` 函数：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 定义函数
def printme( str ):
   "打印任何传入的字符串"
   print(str);
   return;

# 调用函数
printme("我要调用用户自定义函数!");
printme("再次调用同一函数");
```

以上实例输出结果：

```
我要调用用户自定义函数!
再次调用同一函数
```

参数

以下是调用函数时可使用的正式参数类型：

- 必选参数：`fun(name, age)`
- 默认参数：`fun(name='yjc', age=18)`
- 可变参数：`fun(*args)`
- 关键字参数：`fun(name, age, **kw)`
- 命名关键字参数：`fun(name, age, *, city)`

必选参数

必选参数也称位置参数。必选参数须以正确的顺序传入函数。调用时的数量必须和声明时的一样。
调用 `printme()` 函数，你必须传入一个参数，不然会出现语法错误：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

#可写函数说明
def printme( str ):
   "打印任何传入的字符串"
   print(str);
   return;

#调用printme函数
printme();
```

以上实例输出结果：

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

默认参数

调用函数时，默认参数的值如果没有传入，则被认为是默认值。下例会打印默认的age，如果age没有被传入：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

def printinfo( name, age = 35 ):
    "打印任何传入的字符串"
    print ("Name: ", name);
    print ("Age ", age);
    return;

# 调用printinfo函数
printinfo( "miki" )
printinfo( "miki", 50)

# 当不按顺序提供部分默认参数时，需要把参数名写上：
printinfo( name="miki", age=50 )
```

以上实例输出结果：

```
Name:  miki
Age  35
Name:  miki
Age  50
Name:  miki
Age  50
```

可变参数

在Python函数中，还可以定义可变参数。顾名思义，可变参数就是传入的参数个数是可变的，可以是1个、2个到任意个，还可以是0个。基本语法如下：

```
def functionname(*var_args_tuple ):
   "函数_文档字符串"
   function_suite
   return [expression]
```

加了星号（*）的变量名会存放所有未命名的变量参数。如下实例：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

def calc(*numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

在函数内部，参数numbers接收到的是一个tuple。调用该函数时，可以传入任意个参数，包括0个参数：

```
>>> calc(1, 2)
5
>>> calc()
0
```

如果已经有一个list或者tuple，要调用一个可变参数怎么办？Python允许你在list或tuple前面加一个 * 号，把list或tuple的元素变成可变参数传进去：

```
>>> nums = [1, 2, 3]
```

```
>>> calc(*nums)
14
```

`*nums` 表示把 `nums` 这个list的所有元素作为可变参数传进去。

关键字参数

关键字参数允许我们在传入必选参数外，还可以接受关键字参数kw：

```
def person(name, age, **kw):
    print('name:', name, 'age:', age, 'other:', kw)
```

这里 `name, age` 是必须的， `kw` 可选，意味着第三个参数开始我们可以传入任意个数的关键字参数：

```
>>> person('Bob', 35, city='Beijing')
name: Bob age: 35 other: {'city': 'Beijing'}

>>> person('Adam', 45, gender='M', job='Engineer')
name: Adam age: 45 other: {'gender': 'M', 'job': 'Engineer'}
```

这个例子里，关键字参数让我们保证能接收到 `name` 和 `age` 这两个参数，但是，如果提供更多的参数，我们也能收到。

实际上，关键字参数 `kw` 是个dict，如果我们已经准备好了dict，只需要在前面加 `**` 就可以转换为参数传入：

```
param = {'gender': 'M', 'job': 'Engineer'}
>>> person('Adam', 45, **param)
name: Adam age: 45 other: {'gender': 'M', 'job': 'Engineer'}
```

注关键字参数 `kw` 获得的dict是param的一份拷贝，对kw的改动不会影响到函数外的param。

命名关键字参数

如果要限制关键字参数的名字，就可以用命名关键字参数，例如，只接收city和job作为关键字参数。这种方式定义的函数如下：

```
def person(name, age, *, city, job):
    print(name, age, city, job)

```

```
# 调用：
person('yjc', 22, city='Beijing', job='IT')
```

输出：

```
yjc 22 Beijing IT
```

和关键字参数 `**kw` 不同，命名关键字参数需要一个特殊分隔符 `*` ， `*` 后面的参数被视为 `命名关键字参数` 。

命名关键字参数必须传入参数名，这和位置参数不同。如果没有传入参数名，调用将报错。如果调用时缺少参数名city和job，Python解释器把这4个参数均视为位置参数，但 `person()` 函数仅接受2个位置参数。

如果函数定义中已经有了一个可变参数，后面跟着的命名关键字参数就不再需要一个特殊分隔符 `*` 了：

```
def person(name, age, *args, city, job):
    print(name, age, args, city, job)
```

命名关键字参数可以有缺省值，从而简化调用：

```
def person(name, age, *, city='Beijing', job):
    print(name, age, city, job)
```

由于命名关键字参数city具有默认值，调用时，可不传入city参数。

返回多个值

Python里函数可以返回多个值：

```
def updPoint(x, y):
    x+=5
    y+=10
    return x,y

x,y = updPoint(1,2)
print(x,y)
```

输出：

```
6 12
```

但其实这只是一种假象，Python函数返回的仍然是单一值：

```
r = updPoint(1,2)
print(r)
```

输出：

```
(6, 12)
```

返回值是一个tuple！但是，在语法上，返回一个tuple可以省略括号，而多个变量可以同时接收一个tuple，按位置赋给对应的值，所以，Python的函数返回多值其实就是返回一个tuple，但写起来更方便。

匿名函数

python 使用 `lambda` 来创建匿名函数。

`lambda` 只是一个表达式，函数体比 `def` 简单很多。

lambda的主体是一个表达式，而不是一个代码块。仅仅能在lambda表达式中封装有限的逻辑进去。

lambda函数拥有自己的名字空间，且不能访问自有参数列表之外或全局名字空间里的参数。

虽然lambda函数看起来只能写一行，却不等同于C或C++的内联函数，后者的目的是调用小函数时不占用栈内存从而增加运行效率。

语法
lambda函数的语法只包含一个语句，如下：

```
lambda [arg1 [,arg2,.....argn]]:expression
```

如下实例：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 可写函数说明
sum = lambda arg1, arg2: arg1 + arg2;

```

```
# 调用sum函数
print "相加后的值为 : ", sum( 10, 20 )
print "相加后的值为 : ", sum( 20, 20 )
```

以上实例输出结果：

```
相加后的值为 :  30
相加后的值为 :  40
```

return语句

return语句用于退出函数，选择性地向 调用方 返回一个表达式。不带参数值的return语句返回None。之前的例子都没有示范如何返回数值，下例便告诉你怎么做：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 可写函数说明
def sum( arg1, arg2 ):
   # 返回2个参数的和."
   total = arg1 + arg2
   print("函数内 : ", total)
   return total;

# 调用sum函数
total = sum( 10, 20 );
print("函数外 : ", total )
以上实例输出结果：
函数内 :  30
函数外 :  30
```

变量作用域

一个程序的所有的变量并不是在哪个位置都可以访问的。访问权限决定于这个变量是在哪里赋值的。

变量的作用域决定了在哪一部分程序你可以访问哪个特定的变量名称。两种最基本的变量作用域如下：

- 全局变量
- 局部变量

变量和局部变量

定义在函数内部的变量拥有一个局部作用域，定义在函数外的拥有全局作用域。

局部变量只能在其被声明的函数内部访问，而全局变量可以在整个程序范围内（包括函数里面）访问。如下实例：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

total = 0; # 这是一个全局变量
# 可写函数说明
def sum( arg1, arg2 ):
   #返回2个参数的和."
   total = arg1 + arg2; # total在这里是局部变量.
   print "函数内是局部变量 : ", total
   return total;

#调用sum函数
sum( 10, 20 );
print "函数外是全局变量 : ", total
```

以上实例输出结果：

```
函数内是局部变量 :  30
函数外是全局变量 :  0
```

但是如果在函数中定义的局部变量如果和全局变量同名，则它会隐藏该全局变量。例如：

```
#!/usr/bin/python
a = 10;

def test():
    print(a);
    #a = 11;

test();
```

输出：10
如果把上面代码的注释去掉，运行会报错：UnboundLocalError: local variable ‘a’ referenced before assignment。这点非常值得注意。

如果要给全局变量在一个函数里赋值，必须使用**global**语句。

`global VarName` 表达式会告诉Python， VarName是一个全局变量，这样Python就不会在局部命名空间里寻找这个变量了。

例如，我们在全局命名空间里定义一个变量money。我们再在函数内给变量money赋值，然后Python会假定money是一个局部变量。然而，我们并没有在访问前声明一个局部变量money，结果就是会出现一个 `UnboundLocalError` 的错误。取消global语句的注释就能解决这个问题。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

Money = 2000
def AddMoney():
    # 想改正代码就取消以下注释:
    # global Money
    Money = Money + 1

print Money
AddMoney()
print Money
```

在这里我对比c、php、js进行说明:

- php里函数外定义的变量在函数内部是不可见的。如果想使用或者修改函数外的变量，需要函数内使用global语句。
- pyhon里函数外定义的变量在函数内部是可见的，但是不能修改。且函数里存在和函数外变量同名，函数外变量不可用。如果想修改函数外的变量，需要函数内使用global语句。
- c语言里函数外定义的变量在函数内部是可见可修改的。但函数内部定义的变量(含同名)作用域是局部的。
- js语言里函数外定义的变量在函数内部是可见可修改的。但函数内部定义的变量(含同名)作用域是局部的。另外js变量在函数内部还有变量提升的特性。

按值传递参数和按引用传递参数

和其他语言不一样，传递参数的时候，python不允许程序员选择采用传值还是传引用。

Python参数传递采用的肯定是“传对象引用”的方式。实际上，这种方式相当于传值和传引用的一种综合。

如果函数收到的是一个可变对象（比如字典或者列表）的引用，就能修改对象的原始值—相当于通过“传引用”来传递对象。如果函数收到的是一个不可变对象（比如数字、字符或者元组）的引用，就不能直接修改原始对象—相当于通过“传值’来传递对象。

例如：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 可写函数说明
def changeme( mylist ):
   "修改传入的列表"
   mylist.append([1,2,3,4]);
   print("函数内取值: ", mylist)
   return

# 调用changeme函数
mylist = [10,20,30];
changeme( mylist );
print("函数外取值: ", mylist)

# 不可变对象
def changeFixed(age):
    age += 5
    print(age)
age = 10;
changeFixed(age);
print(age)
```

输出结果如下：

```
函数内取值:  [10, 20, 30, [1, 2, 3, 4]]
函数外取值:  [10, 20, 30, [1, 2, 3, 4]]

15
10
```

有时候，我们并不想函数修改了原列表，有没有办法呢？有的。既然列表互相赋值是地址引用，引入新的变量名也是不行的，那么，可以复制一个列表出来，这样内存里用的就不是同一个地址，也就不会改变原数组了：

```
def changeme( mylist ):
   "修改传入的列表"
   mylist.append([1,2,3,4]);
   print("函数内取值: ", mylist)
   return
```

```
6.
7.  # 调用changeme函数
8.  mylist = [10,20,30];
9.  # tmp = mylist; # 这样还是引用同一地址，达不到不修改原列表目的
10. tmp = mylist[:] # 将原列表复制一份，地址将不再是相同的
11. changeme( tmp );
12. print("函数外取值: ", mylist)
```

输出：

```
1.  函数内取值:  [10, 20, 30, [1, 2, 3, 4]]
2.  函数外取值:  [10, 20, 30]
```

这里使用了切片相关知识，后续会详细讲解。

空函数

如果想定义一个什么事也不做的空函数，可以用pass语句：

```
1.  def nop():
2.      pass
```

常用系统函数

> 参考:
> 1、Python 函数
> http://www.runoob.com/python/python-functions.html
> 2、函数的参数
> http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/00143175294503eb82ac80a3e64b9bb4929b16eeed1eb9000
>
> 作者: 飞鸿影
> 版权: 本作品采用「署名-非商业性使用-相同方式共享 4.0 国际」许可协议进行许可。
> 出处: https://www.cnblogs.com/52fhy/p/6254553.html

Python学习—06切片

Python里提供了切片（Slice）操作符获取列表里的元素。

示例：

```
>>> L = [1,2,3,4,5]

# 取前2个元素，传统方法
>>> [L[0],L[1]]
[1,2]

# 取前2个元素，使用切片
>>> L[0:2]
[1,2]
```

`L[0:2]` 表示，从索引0开始取，直到索引2为止，但不包括索引2。

如果第一个索引是0，还可以省略：

```
>>> L[:2]
[1,2]
```

也可以倒数取元素：

```
>>> L[-2:]
[4,5]
```

`L[-2:]` 表示倒数第2个开始直到结束。记住倒数第一个元素的索引是-1。

如果不指定开始和结束，只写 `[:]` 就可以原样复制一个list：

```
>>> L[:]
[1,2,3,4,5]
```

这个技巧很有用，在函数里如果我们不希望改变原列表，就可以使用该技巧复制出一个列表，传给函数。

切片还支持第三个参数，表示每隔几个元素操作：

```
>>> L[::2]
```

```
2. [1,3,5]
```

tuple也是一种list，唯一区别是tuple不可变。因此，tuple也可以用切片操作，只是操作的结果仍是tuple:

```
1. >>> (0, 1, 2, 3, 4, 5)[:3]
2. (0, 1, 2)
```

字符串'xxx'也可以看成是一种list，每个元素就是一个字符。因此，字符串也可以用切片操作，只是操作结果仍是字符串:

```
1. >>> 'ABCDEFG'[:3]
2. 'ABC'
3. >>> 'ABCDEFG'[::2]
4. 'ACEG'
```

很多编程语言针对字符串会提供很多字符串截取函数，例如substr。Python使用简单的切片操作即可完成同样的功能。

Python学习—07迭代器、生成器

迭代

如果给定一个list或tuple，我们可以通过for循环来遍历这个list或tuple，这种遍历我们称为迭代（Iteration）。

Python里使用 `for...in` 来迭代。

常用可迭代对象有list、tuple、dict、字符串等。示例：
list:

```
for x in [1,2]:
    print(x)

for x,y in [(1,2),(3,4)]:
    print(x,y)
```

输出：

```
1
2
1 2
3 4
```

上面的for循环里，同时引用了两个变量，在Python里是很常见的。

tuple:

```
for x in (1,2):
    print(x)
```

输出：

```
1
2
```

dict:

```
dict = {"name":"yjc", "age":18}
```

```
for v in dict:
    print(v, dict[v])

for v in dict.values():
    print(v)

for k,v in dict.items():
    print(k,v)
```

输出：

```
name yjc
age 18

yjc
18

age 18
name yjc
```

dict默认迭代的key，可以使用 `dict.values()` 获取value；还可以使用 `dict.items()` 同时获取key和value。

字符串也是可迭代对象：

```
for ch in 'abcd':
    print(ch)
```

输出：

```
a
b
c
d
```

如果要对list实现类似Java那样的下标循环可以使用内置的enumerate函数—可以把一个list变成索引-元素对，这样就可以在for循环中同时迭代索引和元素本身：

```
for k,v in enumerate([1,2]):
    print(k,v)
```

输出：

```
0 1
1 2
```

那么，如何判断一个对象是否可迭代呢？可以使用 `collections` 模块里的 `Iterable` 判断：

```
from collections import Iterable

print(isinstance([1,2], Iterable));
```

输出：

```
True
```

任何可迭代对象都可以作用于for循环，包括我们自定义的数据类型，只要符合迭代条件，就可以使用for循环。

列表生成式

列表生成式(List Comprehensions)是Python特有的用来创建list的生成式。

示例：

```
list = [x*x for x in range(1,10)]
print(list)
```

输出：

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

以上代码相当于：

```
list = []
for x in range(1,10):
    list.append(x*x)
print(list)
```

看到这里大家应该理解Python列表生成式的含义了。运用列表生成式，可以写出非常简洁的代码。

我们再看几个示例：

1)输出偶数:

```
1.  list = [x for x in range(1,10) if x % 2 == 0 ]
2.  print(list)
```

输出:

```
1.  [2, 4, 6, 8]
```

2)笛卡尔积

```
1.  a = "AB"
2.  b = "XYZ"
3.  list = [m+n for m in a for n in b]
4.  print(list)
```

输出:

```
1.  ['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ']
```

3)大写转小写

```
1.  list = [s.lower() for s in ['I', 'Love', 'Python', 100] if isinstance(s,str)]
2.  print(list)
```

输出:

```
1.  ['i', 'love', 'python']
```

这里使用 `isinstance()` 判断类型，因为非字符串类型没有 `lower()` 方法，Python会报错，所以这里加了个判断:

```
1.  >>> isinstance('love', str)
2.  True
3.  >>> isinstance(100, str)
4.  False
```

生成器

前面我们使用列表生成式可以很方便的生成一个我们需要的列表。但是如果生成一个很大的列表，会比

较占内存，例如 `range(1,10000000)` ，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都浪费了。

生成器(generator)不同于列表，它根据写好的算法，能够推算出下一个元素。生成器不属于list类型，属于 `generator` 类型。要创建一个生成器，第一种方法就是把列表生成式最外面的 `[]` 的改成 `()` 就行了：

```
L = [x for x in range(1,10) if x % 2 == 0 ]
print(type(L))
print(L)

g = (x for x in range(1,10) if x % 2 == 0 )
print(type(g))
print(g)
```

输出：

```
<class 'list'>
[2, 4, 6, 8]

<class 'generator'>
<generator object <genexpr> at 0x01EAEE90>
```

我们不能像list那样直接打印出generator。如果要一个一个打印出来，可以通过 `next()` 函数获得generator的下一个返回值：

```
g = (x for x in range(1,10) if x % 2 == 0 )
print(next(g))
print(next(g))
print(next(g))
print(next(g))
print(next(g))
```

输出：

```
2
4
6
8
Traceback (most recent call last):
  File "/1.py", line 6, in <module>
```

```
7.      print(next(g))
```

generator保存的是算法，每次调用 `next(g)` ，就计算出g的下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出 `StopIteration` 的错误。

我们可以使用for循环迭代generator:

```
1.  g = (x for x in range(1,10) if x % 2 == 0 )
2.  for x in g:
3.      print(x)
```

输出:

```
1.  2
2.  4
3.  6
4.  8
```

for循环遇到StopIteration会停止迭代，不会扔出异常。

下面，我们引入生成器的另外一种创建方法，使用 `yield` 关键字。函数里如果有 `yield` 关键字，那么它将不再是一个函数，而是一个生成器，遇到 `yield` 中断，下次又从上次中断的地方继续执行。示例:

```
1.  def test():
2.      print('step 1:')
3.      yield 11
4.      print('step 2:')
5.      yield 22
6.      return 'ok'
7.
8.  g = test()
9.  print(g)
10. print(next(g))
11. print(next(g))
12. print(next(g))
```

输出:

```
1.  <generator object test at 0x005BEC38>
2.  step 1:
3.  11
```

```
step 2:
22
Traceback (most recent call last):
  File "D:\Users\Desktop\1.py", line 12, in <module>
    print(next(g))
StopIteration: ok
```

可以看到，test不是普通函数，而是generator，在执行过程中，遇到yield就中断，下次又继续执行。执行2次yield后，已经没有yield可以执行了，所以，第3次调用 `next(g)` 就报错。

函数改成generator后，同样可以使用for循环迭代：

```
for x in test():
    print(x)
```

下面是斐波拉契数列生成的函数，大家自行看有啥区别：

```
def fib(n):
    a,b,i=0,1,1
    while i <= n:
        a,b = b,a+b
        i+=1
        print(b)

def gfib(n):
    a,b,i=0,1,1
    while i <= n:
        a,b = b,a+b
        i+=1
        yield b

fib(6)
gfib(6)
```

用for循环调用generator时，发现拿不到 `generator` 的return语句的返回值。如果想要拿到返回值，必须捕获 `StopIteration` 错误，返回值包含在 `StopIteration` 的value中：

```
>>> g = fib(6)
>>> while True:
...     try:
...         x = next(g)
...         print('g:', x)
```

```
...     except StopIteration as e:
...         print('Generator return value:', e.value)
...         break
...
g: 1
g: 1
g: 2
g: 3
g: 5
g: 8
Generator return value: done
```

迭代器

可以直接作用于for循环的一类是list、tuple、dict、set、str，另一类是generator。这些对象统称为可迭代对象： `Iterable` 。

可以直接作用于 `next()` 的对象称为迭代器： `Iterator` 。迭代器既可以作用于for循环，还可以被 `next()` 函数不断调用并返回下一个值，直到最后抛出 `StopIteration` 错误表示无法继续返回下一个值了。

使用isinstance()判断一个对象是否是Iterable对象或者Iterator对象：

```
>>> from collections import Iterable
>>> isinstance([], Iterable)
True

>>> from collections import Iterator
>>> isinstance((x for x in range(10)), Iterator)
True
```

生成器都是Iterator对象，但 `list` 、 `dict` 、 `str` 虽然是 `Iterable` ，却不是 `Iterator` 。

把list、dict、str等Iterable变成Iterator可以使用 `iter()` 函数：

```
>>> isinstance(iter([]), Iterator)
True
>>> isinstance(iter('abc'), Iterator)
True
```

Python的for循环本质上就是通过不断调用next()函数实现的，例如：

```
for x in [1, 2, 3, 4, 5]:
    pass
```

实际上完全等价于：

```
# 首先获得Iterator对象:
it = iter([1, 2, 3, 4, 5])
# 循环:
while True:
    try:
        # 获得下一个值:
        x = next(it)
    except StopIteration:
        # 遇到StopIteration就退出循环
        break
```

Python学习—08函数式编程

函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数。

高阶函数

Python支持高阶函数**(Higher-order function)**。

什么是高阶函数呢？把函数作为参数传入，这样的函数称为高阶函数。

高阶函数的特点：
1、变量可以指向函数

```
>>> abs(10)
10
>>> abs
<built-in function abs>
>>> x = abs
>>> x
<built-in function abs>
```

这个例子告诉我们： `abs` 是函数， `abs(10)` 是函数调用；可以将函数赋给一个变量，这个变量也指向了函数。

2、函数名也是变量

```
>>> abs = 10
>>> abs
10
```

例子中，当我们把一个函数名重新赋值后，该函数名不再指向函数，而是指向整数 `10` ，说明函数名本身也是变量，是指向函数的变量。

3、函数参数可以传入函数

```
def test(x,y,f):
    return f(x) + f(y)

print(test(1, -2, abs))
```

输出：

```
3
```

map函数

```
map(function f, Iterable iterable)
# 返回: Iterable
```

`map()` 函数接收两个参数，一个是函数，一个是Iterable，map将传入的函数依次作用到序列的每个元素，并把结果作为新的Iterator返回。

示例：

```
def f(x):
    return x*x

iterator = map(f, [1,2,3,4])
print(list(iterator))
```

输出：

```
[1, 4, 9, 16]
```

map返回的结果是个迭代器(Iterator)。我们可以使用for循环获取结果或者直接使用 `list()` 函数将结果转变为list：

```
iterator = map(f, [1,2,3,4])
for x in iterator:
    print(x)
```

输出：

```
1
4
9
16
```

reduce函数

```
reduce(function f, Iterable iterable)
```

`reduce()` 函数同样接收两个参数，一个是函数，一个是Iterable，reduce把结果继续和序列的下一个元素做累积计算，返回的是累计结果，非迭代器。注意的是 `reduce()` 传入的函数 `函数f` 必须接收两个参数。

```
import functools

def f(x,y):
    return x*10+y

iterator = functools.reduce(f, [1,2,3,4])
print(iterator)
```

输出：

```
1234
```

reduce在Python3.3+已经移到functools里了，需要先导入functools。

字符串转整数示例：

```
from functools import reduce

def str2int(s):
    def fn(x, y):
        return x * 10 + y
    def char2num(s):
        return {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7,
'8': 8, '9': 9}[s]
    return reduce(fn, map(char2num, s))
```

调用：

```
>>> str2int('135')
135
```

reduce()还可以接收第3个可选参数，作为计算的初始值。如果把初始值设为100，计算：

```
def f(x, y):
    return x + y
```

```
reduce(f, [1, 3, 5, 7, 9], 100)
```

结果将变为125。

filter函数

```
filter(function f, Iterable iterable)
# 返回: Iterable
```

和 `map()` 类似， `filter()` 也接收一个函数和一个序列。和 `map()` 不同的是， `filter()` 把传入的函数依次作用于每个元素，然后根据返回值是True还是False决定保留还是丢弃该元素。True保留，False丢弃。

```
def f(x):
    if x % 2 == 0:
        return True
r = filter(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])
print(list(r))
```

输出:

```
[2, 4, 6, 8]
```

sorted函数

```
sorted(Iterable iterable,  key=None)
```

Python内置的 `sorted()` 函数就可以对list进行排序，按从小到大。

它还可以接收一个key函数来实现自定义的排序，key指定的函数将作用于list的每一个元素上，并根据key函数返回的结果进行排序。

```
l = [10,5,2,7]
s = sorted(l)
print(s)
```

输出:

```
[2, 5, 7, 10]
```

```
def opposite(x):
    return -x
l = [10,5,2,7]
s = sorted(l, key=opposite)
print(s)
```

输出：

```
[10, 7, 5, 2]
```

装饰器

装饰器（Decorator）是一种在代码运行期间动态增加功能的方式。例如，我们有个写日志的函数 `log()` ，我们希望在不修改该函数的情况取增强该函数的功能，比如日志前打印一行时间和该函数名：

```
import time

def writelog(func):
    def wrapper(*args, **kw):
        print('['+ time.strftime('%Y-%m-%d %H:%M:%S') + '] :' + func.__name__)
        return func(*args, **kw)
    return wrapper

@writelog
def log(msg):
    print(msg)

log('This is a test msg.')
```

输出：

```
[2017-01-11 21:10:53] :log
This is a test msg.
```

这个例子里我们编写了装饰器 `writelog()` 用来增加函数 `log()` 的功能，只需要在函数 `log()` 前加 `@writelog` 即可。其中 `func.__name__` 代表函数的函数名。内部函数 `wrapper()` 名字非固定的，可以自定义名称。

wrapper()函数的参数定义是 `(*args, **kw)` ，因此， `wrapper()` 函数可以接受任意参数的调

用。

偏函数

大家不要被这个名字给吓到，其实很简单的一个东西：用于固定函数的行为。例如，函数 `int(x, base=10)` 默认只需要传第一个参数，因为第二个参数默认是10，代表10进制。

如果我们想默认使用2进制呢，一般是这样：

```
1.  def int2(x):
2.      return int(x, 2)
```

而偏函数就是专门干这事情的。使用偏函数表示上面自定义方法就是：

```
1.  import functools
2.  int2 = functools.partial(int, base=2)
3.
4.  print(int2('10'))
5.  print(int('10'))
```

输出：

```
1.  2
2.  10
```

是不是很简单？

所以，简单总结偏函数 `functools.partial` 的作用就是，把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单。

Python学习—09 模块

模块让我们能够有逻辑地组织Python代码段。把相关的代码分配到一个 模块里能让我们的代码更好用，更易懂。

导入模块

Python使用 `import` 语句导入模块。语法：

```
# 形式一：导入模块
import module1[, module2[,... moduleN]
## 示例
import sys

# 形式二：从模块中导入一个指定的部分到当前命名空间中
from modname import name1[, name2[, ... nameN]]
## 示例
from fib import fibonacci

# 形式三：把一个模块的所有内容全都导入到当前的命名空间，不建议使用
from modname import *
```

示例：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

' a test module '

__author__ = 'Michael Liao'

import sys

def test():
    args = sys.argv
    if len(args)==1:
        print('Hello, world!')
    elif len(args)==2:
        print('Hello, %s!' % args[1])
    else:
```

```
17.         print('Too many arguments!')
18.
19. if __name__=='__main__':
20.     test()
```

第1行和第2行：标准注释，第1行注释可以让这个hello.py文件直接在Unix/Linux/Mac上运行，第2行注释表示.py文件本身使用标准UTF-8编码；
第4行是模块注释；
第6行是模块作者；
以上是模块标准模板，当然也可以全部删掉不写，但是建议按标准编写。
第8行导入 `sys` 模块；导入模块后，就可以使用该模块的所有功能了。
第11行 `sys.argv` 表示命令行参数，第一个参数永远是该 `.py` 文件的名称。
第19行需要注意：在命令行里打开啊py文件，Python解释器会把特殊变量 `__name__` 置为 `__main__` ，而在其他地方导入py文件，则判断不会成立。

定位模块

当我们导入一个模块，Python解析器对模块位置的搜索顺序是：

```
1. 1. 当前目录
2. 2. 如果不在当前目录，Python 则搜索在 shell 变量 PYTHONPATH 下的每个目录。
3. 3. 如果都找不到，Python会察看默认路径。UNIX下，默认路径一般为/usr/local/lib/python/。
```

模块搜索路径存储在system模块的 `sys.path` 变量中。

如果我们要添加自己的搜索目录，有两种方法：

一是直接修改 `sys.path` ，添加要搜索的目录：

```
1. >>> import sys
2. >>> sys.path.append('/Users/michael/my_py_scripts')
```

这种方法是在运行时修改，运行结束后失效。

第二种方法是设置环境变量 `PYTHONPATH` ，该环境变量的内容会被自动添加到模块搜索路径中。设置方式与设置 `Path` 环境变量类似。注意只需要添加你自己的搜索路径，Python自己本身的搜索路径不受影响。

查看模块

通过 `dir()` 函数可以查看指定模块的所有功能：

```
import math

print(dir(math))
```

输出：

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos',
'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial',
'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf',
'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'pi',
'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

在这里，特殊字符串变量 `__name__` 指向模块的名字， `__file__` 指向该模块的导入文件名。

作用域

在一个模块中，我们可能会定义很多函数和变量，但有的函数和变量我们希望给别人使用，有的函数和变量我们希望仅仅在模块内部使用。在Python中，是通过 `_` 前缀来实现的。

正常的模块都是公开的，可以直接使用，我们称之为public；而以 `_x` 或者 `__x` 开头的是非公开的，称为private，不应该被直接访问。其中类似 `__xxx__` 这样的变量是特殊变量，有着特殊意义，例如 `__author__` ， `__name__` 。

但是Python不会强制我们不能使用private变量，直接访问是可以的，因为Python并没有一种方法可以完全限制访问private函数或变量，但是，从编程习惯上不应该引用private函数或变量。

```
def _fun1():
    pass

def _fun2():
    pass

def myfunc(type):
    if(type == 1):
        return _fun1()
    else:
        return _fun2()
```

作用域总结：外部不需要引用的函数全部定义成private，只有外部需要引用的函数才定义为public。

编写自己的模块

Python的模块有一定的目录结构，只要按照约定的格式编写，很容易编写自己的模块。

python模块组成：

```
mymodule
 |-- __init__.py
 |-- x.py
 |-- y.py
 |-- submodule
     |--__init__.py
     |-- x.py
     |-- y.py
```

Python模块下必须有 `__init__.py` 文件，表明这是一个模块；如果有子模块，那么子模块也必须有 `__init__.py` 文件。该文件可以为空。

如果 `mymodule/__init__.py` 里有 `add()` 方法，外部调用方法：

```
import mymodule
mymodule.add()
```

如果 `mymodule/x.py` 里有 `add()` 方法，外部调用方法：

```
from mymodule import x
x.add()
```

如果 `mymodule/submodule/__init__.py` 里有 `add()` 方法，外部调用方法：

```
import mymodule.submodule
mymodule.submodule.add()
```

如果 `mymodule/submodule/x.py` 里有 `add()` 方法，外部调用方法：

```
from mymodule.submodule import x
x.add()
```

第三方模块

Python社区有大量的第三方模块供我们使用。例如网站：https://pypi.python.org/pypi。

安装pip

我们一般是通过包管理工具pip安装第三方模块的。在安装Python后，系统一般会带有该工具。安装windows版本的时候注意：如果Windows提示未找到命令，可以重新运行安装程序添加pip。

下面是Linux版本安装方法
(1)ubuntu:

```
1.  sudo apt-get install python-pip
```

(2)Fedora、centos:

```
1.  yum install python-pip
```

(3)Linux, Mac OSX, Windows 下都可用 get-pip.py 来安装
pip: https://pip.pypa.io/en/latest/installing.html

或者直接下载：get-pip.py ，然后运行在终端运行 `python get-pip.py` 就可以安装 pip。

Note: 也可以下载 pip 源码包，运行 `python setup.py install` 进行安装。

安装好后设置环境变量。windows下是：

```
1.  PATH=%PATH%;D:\Python34;D:\Python34\Scripts;
```

分别是Python和Scripts的所在目录。

如果提示pip版本过低，通过下面命令更新：

```
1.  pip install --upgrade pip
```

示例

一般来说，第三方库都会在Python官方的http://pypi.python.org 网站注册，要安装一个第三方库，必须先知道该库的名称，可以在官网或者pypi上搜索，比如Pillow的名称叫Pillow，因此，安装Pillow的命令就是：

```
1.  $ pip install Pillow
2.
3.  Collecting Pillow
```

```
  Downloading Pillow-4.0.0-cp34-cp34m-win32.whl (1.2MB)
Successfully installed Pillow-4.0.0
```

耐心等待下载并安装后，就可以使用Pillow了。

PIL: Python Imaging Library，Python平台图像处理库。PIL功能非常强大，但API却非常简单易用。

图像缩放：

```
# coding: utf-8
from PIL import Image
im = Image.open('test.jpg')
print(im.format, im.size, im.mode)
im.thumbnail((200, 100))
im.save('thumb.jpg', 'JPEG')
```

模糊效果：

```
# coding: utf-8
from PIL import Image,ImageFilter
im = Image.open('test.jpg')
im2 = im.filter(ImageFilter.BLUR)
im2.save('blur.jpg', 'jpeg')
```

验证码：

```
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')
```

注意示例里的字体文件必须是绝对路径。

Python学习—10 面向对象编程

面向对象编程——Object Oriented Programming，简称OOP，是一种程序设计思想。OOP把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。

本节对于面向对象的概念不做展开说明。本节主要内容是Python里如何使用面向对象编程。

分下面几部分：
1、类的格式
2、类的实例
3、类的封装
4、继承和多态

类的格式

下面是一个示例：
student.py

```
class Student(object):

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

    def print_score(self):
        print('%s: %s' % (self.name, self.score))
```

通过例子可以发现：
1、类由 `class` 开头，类名首字母大写，类名后面的括号表示继承自基类 `object` ，所有类都会继承这个类；
2、类的构造方法是 `__init__` ，第一个参数是固定的，永远是 `self` ,类里面通过 `self` 调用属性和方法，不同于JAVA里使用 `this` 。

现在来实例化类：

```
stu = Student('yjc', 22)
print(stu)
print(stu.name)
stu.print_score()
```

输出：

```
<__main__.Student object at 0x028811F0>
yjc
yjc: 22
```

Python里实例化类不用 `new` 关键字，而是像使用函数那样即可。调用类里的方法的时候不用传 `self` 参数。

类的封装

默认的，我们实例化了类后，可以直接访问对象里的属性，也可以去修改对象里的属性：

```
stu = Student('yjc', 22)
print(stu.name)
stu.print_score()
stu.name = 'Allen'
stu.print_score()
```

输出：

```
yjc
yjc: 22
Allen: 22
```

如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线 `__` ，在Python中，实例的变量名如果以 `__` 开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问（子类里也不能继承），所以，我们把Student类改一改：

```
class Student(object):

    def __init__(self, name, score):
        self.__name = name
        self.__score = score

    def print_score(self):
        print('%s: %s' % (self.__name, self.__score))


stu = Student('yjc', 22)
print(stu.name)
```

输出：

```
Traceback (most recent call last):
  File "/Projects/python/code/class/Student.py", line 12, in <module>
    print(stu.__name)
AttributeError: 'Student' object has no attribute '__name'
```

这时候如果直接访问，就会报错。

继承和多态

在OOP程序设计中，当我们定义一个class的时候，可以从某个现有的class继承，新的class称为 `子类` （Subclass），而被继承的class称为 `基类` 、 `父类` 或 `超类` （Base class、Super class）。

继承示例：

```
class Animal(object):
    def run(self):
        print('runing')

class Dog(Animal):
    pass
```

以上 `Dog` 类继承了 `Animal` 类，从而获得父类的方法 `run()` ：

```
dog = Dog()
dog.run()
```

输出：

```
runing
```

这里子类虽然没有写 `run()` 方法，但由于父类已经拥有，所以可以直接继承过来。

需要注意的是父类的私有变量(`__` 开头的)子类是不能访问，也不能继承的。

再看看类的多态：

```
class Dog(Animal):
```

```
    def run(self):
        print('Dog is runing')

class Cat(Animal):
    def run(self):
        print('Cat is runing')

def runClass(obj):
    obj.run()

runClass(Animal())
runClass(Dog())
runClass(Cat())
```

输出：

```
runing
Dog is runing
Cat is runing
```

通过例子我们可以看到：
1、子类如果定义了和父类一样的方法，子类的会覆盖父类；
2、方法 `runClass` 接收一个 `Animal` 类，只要含有 `run()` 方法，均可正常运行，原因就在于多态。

类属性和实例属性

由于Python是动态语言，根据类创建的实例可以任意绑定属性。对于上面的 `Animal` 类，我们可以动态绑定新的属性：

```
a = Animal()
a.name = 'lala'
```

但这个绑定的属性属于实例，不属于类，其它 `Animal` 的实例是不能访问的。

在类里直接定义的属性，称为类属性，所有实例均可访问。

获取对象信息

type

最简单的，我们可以使用 `type()` 获取对象的类型：

```
>>> type(1)
<class 'int'>
>>> type('1')
<class 'str'>
>>> type(True)
<class 'bool'>
>>> type(1.2)
<class 'float'>
>>> type(None)
<class 'NoneType'>
>>> type(abs)
<class 'builtin_function_or_method'>
>>> type(lambda x:x*2)
<class 'function'>
>>> type([1,2])
<class 'list'>
>>> type((1,2))
<class 'tuple'>
>>> type({"name":"yjc"})
<class 'dict'>
>>> type((x for x in range(10)))
<class 'generator'>
```

还可以使用 `types` 模块进行判断：

```
>>> import types
>>> def f():pass
...
>>>
>>> type(f)==types.FunctionType
True
>>> type(f)==types.BuiltinFunctionType
False
>>> type(lambda x: x)==types.LambdaType
True
>>> type((x for x in range(10)))==types.GeneratorType
True
```

isinstance

对于类的继承关系，使用 `isinstance()` 会更方便：

```
class Animal(object):
    def run(self):
        print('runing')

class Dog(Animal):
    pass

dog = Dog()
isinstance(dog, Dog)
isinstance(dog,object)
```

dir

如果要获得一个对象的所有属性和方法，可以使用 `dir()` 函数：

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos',
'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial',
'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite',
'isinf','isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf',
'pi', 'pow', 'radians', 'sin','sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

类似 `__xxx__` 的属性和方法在Python中都是有特殊用途的，比如 `__len__` 方法返回长度。

getattr/setattr/hasattr

`getattr()` 可以获取类的某个属性， `setattr()` 可以设置类的某个属性， `getattr()` 可以判断类是否拥有某个属性：

```
class Animal(object):
    def __init__(self):
        self.type = 'animal'
        self.area = 'China'

    def run(self):
        print('runing')

class Dog(Animal):
```

```
    pass

dog = Dog()

print(hasattr(dog,'type'))
print(hasattr(dog,'run'))
print(hasattr(dog,'name'))

print(getattr(dog,'area'))

# print(getattr(dog,'name'))

setattr(dog, 'name', 'Aia')
print(getattr(dog,'name'))
```

输出：

```
True
True
False

China

Aia
```

使用 `getattr()` 尝试获取对象的一个不存在属性会报错。可以通过设置默认值避免：

```
print(getattr(dog,'city', 'beijing')) #beijing
```

Python学习—11 面向对象高级编程

多重继承

Python里允许多重继承，即一个类可以同时继承多个类：

```
class Mammal(Animal):
    pass

class Runnable(object):
    def run(self):
        print('Running...')

class Dog(Mammal, Runnable):
    pass
```

这样， `Dog` 同时拥有 `Mammal` 、 `Runnable` 的属性和方法。

__slots__限制实例的属性

由于类的实例可以动态绑定新的属性，有时候我们不希望这样，可以通过 `__slots__` 进行限制：

```
class Student(object):
    __slots__ = ('name', 'age') # 用tuple定义允许绑定的属性名称
```

然后，我们试试：

```
>>> s = Student() # 创建新的实例
>>> s.name = 'yjc' # 绑定属性'name'
>>> s.age = 25 # 绑定属性'age'
>>> s.score = 99 # 绑定属性'score'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'score'
```

由于 `score` 没有被放到 `__slots__` 中，所以不能绑定score属性，试图绑定score将得到 `AttributeError` 的错误。

使用 `__slots__` 要注意， `__slots__` 定义的属性仅对当前类实例起作用，对继承的子类是不起作

用的：

```
>>> class SubStudent(Student):
...     pass
...
>>> g = SubStudent()
>>> g.score = 99
```

除非在子类中也定义 `__slots__` ，这样，子类实例允许定义的属性就是自身的 `__slots__` 加上父类的 `__slots__` 。

@property装饰器

当我们通过实例使用类的属性时，通常不希望直接访问，而是处理之后再暴露出来。例如：

```
class Student(object):
    def setScore(self, value):
        if(value > 100):
            value = 100
        if(value < 0):
            value = 0
        self.__score = value

    def getScore(self):
        return self.__score

s = Student()
s.setScore(199)
print(s.getScore())
```

输出：

```
100
```

这里， `__score` 属性我们通过 `setScore` 先设置，然后使用 `getScore` 获得，并对不合理值进行了处理。

上面我们通过类里的方法实现了类属性的设置和访问。那么，有没有既能检查参数，又可以用类似属性这样简单的方式来访问类的变量呢？Python里的 `@property` 装饰器就是做这个的：

```
class Student(object):
```

```
2.
3.      @property
4.      def score(self):
5.          return self.__score
6.
7.      @score.setter
8.      def score(self, value):
9.          if(value > 100):
10.             value = 100
11.         if(value < 0):
12.             value = 0
13.         self.__score = value
14.
15. s = Student()
16. s.score = 199
17. print(s.score)
```

输出：

```
1.  100
```

Python内置的 `@property` 装饰器就是负责把一个方法变成属性调用。此时， `@property` 本身又创建了另一个装饰器 `@score.setter` ，负责把一个 `setter` 方法变成属性赋值。

`@property` 广泛应用在类的定义中，可以让调用者写出简短的代码，同时保证对参数进行必要的检查，这样，程序运行时就减少了出错的可能性。

定制类

我们可以使用类似 `__slots__` 这种变量或者函数名来定制类，这些在Python里是有特殊作用的。

通过自定义下面这些属性或方法，我们可以对类做自定义处理：

`__slots__` ：限制实例的属性
`__len__()` ：自定义返回长度

`__str__()` ：当尝试使用print打印类的时候，自定义返回类的内容。因为默认打印出一堆 `<__main__.Student object at 0x109afb190>` ，不好看。

```
1.  class Student(object):
2.      def __init__(self, name):
3.          self.name = name
```

```
4.
5.      def __str__(self):
6.          return 'Student object (name: %s)' % self.name
7.
8.  print(Student('yjc'))
```

输出：

```
1.  Student object (name: yjc)
```

这样打印出来的实例，不但好看，而且容易看出实例内部重要的数据。

`__repr__()`：与`__str__()`类似，当直接敲变量`Student('yjc')`不用print的时候，会自动调用该方法。

`__getattr__()`：默认调用类里不存在的属性时，会报错。通过该方法，可以动态返回一个属性。

```
1.  class Student(object):
2.
3.      def __init__(self):
4.          self.name = 'yjc'
5.
6.      def __getattr__(self, attr):
7.          if attr=='score':
8.              return 80
```

这时候调用score属性，不会报错了：

```
1.  >>> s = Student()
2.  >>> s.name
3.  'yjc'
4.  >>> s.score
5.  80
```

`__call__()`：通过覆写该方法，可以将实例像方法那样直接调用：

```
1.  class Student(object):
2.      def __init__(self, name):
3.          self.name = name
4.
5.      def __call__(self):
6.          print('My name is %s.' % self.name)
```

调用方式如下：

```
>>> s = Student('yjc')
>>> s() # self参数不要传入
My name is yjc.
```

`__call__()` 还可以定义参数。对实例进行直接调用就好比对一个函数进行调用一样，所以你完全可以把对象看成函数，把函数看成对象，因为这两者之间本来就没啥根本的区别。

通过 `callable()` 函数，我们就可以判断一个对象是否是可调用对象：

```
>>> callable(Student())
True
>>> callable(max)
True
```

枚举类

Python提供 `Enum` 类来实现枚举功能：

```
# coding: utf-8
from enum import Enum

Month = Enum('Month', ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug',
'Sep', 'Oct', 'Nov', 'Dec'))

# 可以直接使用Month.Jan来引用一个常量：
print(Month.Jan.value)

# 枚举所有成员：
for name,member in Month.__members__.items():
    print(name, '=>', member, ',', member.value)
```

输出：

```
1
Jan => Month.Jan , 1
Feb => Month.Feb , 2
Mar => Month.Mar , 3
Apr => Month.Apr , 4
```

```
May => Month.May , 5
Jun => Month.Jun , 6
Jul => Month.Jul , 7
Aug => Month.Aug , 8
Sep => Month.Sep , 9
Oct => Month.Oct , 10
Nov => Month.Nov , 11
Dec => Month.Dec , 12
```

`value` 属性则是自动赋给成员的 `int` 常量，默认从1开始计数。

如果想自定义value值：

```
# coding: utf-8
from enum import Enum,unique

@unique
class Month(Enum):
    Jan = 0
    Feb = 1
    Mar = 2

print(Month.Jan.value)

for name,member in Month.__members__.items():
    print(name, '=>', member, ',', member.value)
```

输出：

```
0
Jan => Month.Jan , 0
Feb => Month.Feb , 1
Mar => Month.Mar , 2
```

`@unique` 装饰器可以帮助我们检查保证没有重复值。如果重复了，会报ValueError错误：

```
ValueError: duplicate values found in <enum 'Month'>
```

元类

> 动态语言和静态语言最大的不同，就是函数和类的定义，不是编译时定义的，而是运行时动态创建的。

type()

`type()` 可以查看一个类型或变量的类型：

```
# coding:utf-8

class Hello(object):
    pass

h = Hello()

print(type(h))
print(type(Hello))
print(type(object))
```

输出：

```
<class '__main__.Hello'>
<class 'type'>
<class 'type'>
```

通过打印我们发现，类 `Hello` 的类型是 `type` ，但它的实例类型是class `Hello` 类型。

> Python里class的定义是运行时动态创建的，而创建class的方法就是使用 `type()` 函数。

`type()` 函数既可以返回一个对象的类型，又可以创建出新的类型，比如，我们可以通过 `type()` 函数创建出 `Hello` 类：

```
# 定义成员方法：减
def sub(self, x, y):
    return x-y

# 生成类
Hello = type('Hello', (object,), {"add":add, "mysub":sub})

h = Hello()
print(h.add(1, 2))
print(h.mysub(1, 2))

print(type(h))
print(type(Hello))
```

输出：

```
3
-1
<class '__main__.Hello'>
<class 'type'>
```

要创建一个class对象，`type()`函数依次传入3个参数：

1. class的名称；
2. 继承的父类集合，注意Python支持多重继承，如果只有一个父类，别忘了tuple的单元素写法；
3. class的方法名称与函数绑定。

通过`type()`函数创建的类和直接写class是完全一样的，因为Python解释器遇到class定义时，仅仅是扫描一下class定义的语法，然后调用`type()`函数创建出class。

metaclass

`metaclass`，直译为元类，可以理解为类的模板。通过`metaclass`也可以动态创建类。

`metaclass`是Python面向对象里最难理解，也是最难使用的魔术代码。正常情况下，我们不会碰到需要使用`metaclass`的情况。

通过`metaclass`创建出类，需要：先定义`metaclass`，然后创建类。下面的示例是给自定义的`MyList`增加一个`add`方法：

示例：

```
# coding: utf-8

# metaclass是类的模板，所以必须从`type`类型派生：
class ListMetaclass(type):
    def __new__(cls, name, base, attrs):
        attrs['add'] = lambda self,value: self.append(value)

        #打印参数信息
        print(cls, '\n', name, '\n', base, '\n', attrs, '\n')

        return type.__new__(cls, name, base, attrs)

# 根据metaclass产生类
class MyList(list, metaclass=ListMetaclass):
    pass
```

```
16.
17. # 类继承
18. class OtherList(MyList):
19.     pass
20.
21. L = MyList()
22. L.add('3')
23.
24. print(L)
```

输出：

```
1. <class '__main__.ListMetaclass'>
2.  MyList
3.  (<class 'list'>,)
    {'add': <function ListMetaclass.__new__.<locals>.<lambda> at 0x02424540>,
4. '__module__': '__main__', '__qualname__': 'MyList'}
5.
6. <class '__main__.ListMetaclass'>
7.  OtherList
8.  (<class '__main__.MyList'>,)
    {'add': <function ListMetaclass.__new__.<locals>.<lambda> at 0x02424588>,
9. '__module__': '__main__', '__qualname__': 'OtherList'}
10.
11. ['3']
```

以上通过 `metaclass` 动态生成了 `MyList` 类，并增加了成员方法 `add()` 。

通过分析输出，我们可以发现： `__new__()` 方法接收到的参数依次是：

1. 当前准备创建的类的对象，例如 `ListMetaclass` ；
2. 类的名字，例如 `MyList` ；
3. 类继承的父类集合，例如 `list` ；
4. 类的方法集合，例如 `add` 、 `__module__` 、 `__qualname__` 。

什么时候需要用到 `metaclass` 呢？ORM就是一个典型的例子。

Python学习—12 异常处理、调试

异常捕获

语法格式：

```
try:
    pass
except xxx as e:
    pass
except xxx as e:
    pass
...
else:
    pass

finally:
    pass
```

except用来捕获异常类型，常见的有ValueError、ZeroDivisionError，都继承基类BaseException。如果没有错误发生，则执行else。不管有没有错误发生，都会执行finally。

注意的是，只要一处except的捕获到了，不会继续捕获。

`except xxx as e` 里的 `as e` 可以省略。

示例：

```
#!/usr/bin/python
# coding: utf-8

try:
    r = 100 / 0
    print('result is %s'% r)
except ValueError as e:
    print('ValueError: ', e)
except ZeroDivisionError as e:
    print('ZeroDivisionError: ', e)
except BaseException as e:
    print('BaseException: ', e)
```

```
13. finally:
14.     pass
```

输出：

```
    ('ZeroDivisionError: ', ZeroDivisionError('integer division or modulo by
 1. zero',))
```

抛出异常

Python里使用raise语句抛出一个异常的实例：

```
 1. #!/usr/bin/python
 2. # coding: utf-8
 3.
 4. def cal(m, n):
 5.     if n == 0:
 6.         raise ValueError('Illegal value: %d' % n)
 7.     return  m * n
 8.
 9. try:
10.     r = cal(6, 0)
11.     print(r)
12. except Exception as e:
13.     print(e)
```

输出：

```
1. Illegal value: 0
```

使用logging类记录错误

我们可以使用 `print()` 来调试程序，但如果到处是 `print()` ，想关闭又得一个个去修改。使用logging类，我们可以记录各种级别的错误，通过配置参数，可以控制显示哪些错误记录。

错误级别：

```
1. CRITICAL = 50
2. FATAL = CRITICAL
3. ERROR = 40
```

```
WARNING = 30
WARN = WARNING
INFO = 20
DEBUG = 10
NOTSET = 0
```

对应的方法：

```
logging.critical()
logging.fatal()
logging.error()
logging.warning()
logging.warn()
logging.info()
logging.debug()
```

示例：

```
#!/usr/bin/python
# coding: utf-8

import logging

logging.basicConfig(level=logging.INFO)

def cal(m, n):
    if n == 0:
        # raise ValueError('Illegal value: %d' % n)
        logging.info('Illegal value: %d' % n)
    return  m * n

try:
    r = cal(6, 0)
    print(r)
except Exception as e:
    print(e)
```

输出：

```
0
INFO:root:Illegal value: 0
```

这里设置错误级别是 `INFO` ，那么将会显示 `WARN` 、 `ERROR` 、 `FATAL` 级别的错误， `DEBUG` 、 `NOTSET` 则不会显示。

Python标准异常

异常名称	描述
BaseException	所有异常的基类
SystemExit	解释器请求退出
KeyboardInterrupt	用户中断执行(通常是输入^C)
Exception	常规错误的基类
StopIteration	迭代器没有更多的值
GeneratorExit	生成器(generator)发生异常来通知退出
StandardError	所有的内建标准异常的基类
ArithmeticError	所有数值计算错误的基类
FloatingPointError	浮点计算错误
OverflowError	数值运算超出最大限制
ZeroDivisionError	除(或取模)零 (所有数据类型)
AssertionError	断言语句失败
AttributeError	对象没有这个属性
EOFError	没有内建输入,到达EOF 标记
EnvironmentError	操作系统错误的基类
IOError	输入/输出操作失败
OSError	操作系统错误
WindowsError	系统调用失败
ImportError	导入模块/对象失败
LookupError	无效数据查询的基类
IndexError	序列中没有此索引(index)
KeyError	映射中没有这个键
MemoryError	内存溢出错误(对于Python 解释器不是致命的)
NameError	未声明/初始化对象 (没有属性)
UnboundLocalError	访问未初始化的本地变量
ReferenceError	弱引用(Weak reference)试图访问已经垃圾回收了的对象
RuntimeError	一般的运行时错误
NotImplementedError	尚未实现的方法
SyntaxError	Python 语法错误
IndentationError	缩进错误
TabError	Tab 和空格混用

SystemError	一般的解释器系统错误
TypeError	对类型无效的操作
ValueError	传入无效的参数
UnicodeError	Unicode 相关的错误
UnicodeDecodeError	Unicode 解码时的错误
UnicodeEncodeError	Unicode 编码时错误
UnicodeTranslateError	Unicode 转换时错误
Warning	警告的基类
DeprecationWarning	关于被弃用的特征的警告
FutureWarning	关于构造将来语义会有改变的警告
OverflowWarning	旧的关于自动提升为长整型(long)的警告
PendingDeprecationWarning	关于特性将会被废弃的警告
RuntimeWarning	可疑的运行时行为(runtime behavior)的警告
SyntaxWarning	可疑的语法的警告
UserWarning	用户代码生成的警告

Python学习—13 文件I/O

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）。

读取文件内容

```
# coding: utf-8

f = open('test.txt', 'r')
print(f.read())
f.close()
```

输出：

```
Hello world!
Python
```

标示符'r'表示读。

如果打开的文件不存在，会直接报错：

```
Traceback (most recent call last):
  File "/Projects/python/code/file.py", line 3, in <module>
    f = open('test.txt1', 'r')
IOError: [Errno 2] No such file or directory: 'test.txt1'
```

报错后，后面的 `f.close()` 不会调用。一般我们会使用 `try ... finally` 来操作文件：

```
try:
    f = open('test.txt', 'r')
    print(f.read())
finally:
    if f:
        f.close()
```

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

```
with open('test.txt', '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。

如果文件很大，使用 `read()` 方法一下子读到内存，内存会占满。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：

```
with open('test.txt', 'r') as f:
    for line in f.readlines():
        print(line)
```

打开二进制文件

要读取二进制文件（例如图片、音频、视频）内容，使用 `rb` 模式打开：

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

读取非utf-8编码的文本

如果要读取非utf-8编码的文本，需要给 `open()` 传入编码参数 `encoding` ，默认是utf-8：

```
>>> f = open('gbk.txt', 'r', encoding = 'gbk')
>>> f.read()
'测试GBK'
```

如果不传入，读取的内容是乱码的。

`open()` 还支持第4个参数 `errors` ，用于当读取的内容编码不规范（会返回UnicodeDecodeError错误），示例：

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

这样将忽略错误。

写入内容到文件

```
with open('test.txt', 'w') as f:
    f.write('test\n')
```

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。

读取大文件方法

方法一：将文件切分成小段，每次处理完小段，释放内存

```
def read_in_block(file_path):
    BLOCK_SIZE=1024
    with open(file_path,"r") as f:
        while True:
            block =f.read(BLOCK_SIZE) #每次读取固定长度到内存缓冲区
            if block:
                yield block
            else:
                return #如果读取到文件末尾，则退出

for block in read_in_block(file_path):
    print block
```

这个方法，速度很快，但有个问题，若满足了1024时，会将正好在1024位置的数据切开。

方法二：利用 `open()` 方法生成的迭代对象：

```
with open(file_path) as f:
    for line in f:
        print line
```

这种用法是把文件对象f当作迭代对象，系统将自动处理IO缓存和内存管理。

耗时较第1种方法慢一点。 但每个Line, type都是str, 都是自己需要的一行数据（一行是一个id)。

文件打开模式

模式	描述
r	以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。
rb	以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。
r+	打开一个文件用于读写。文件指针将会放在文件的开头。
rb+	以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。
w	打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
wb	以二进制格式打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
w+	打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
wb+	以二进制格式打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
a	打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
ab	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
a+	打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会是追加模式。如果该文件不存在，创建新文件用于读写。
ab+	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文件不存在，创建新文件用于读写。

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。

StringIO

StringIO就是在内存中读写str。

读取:

```
# coding : utf-8

from io import StringIO
with StringIO('hello\nworld\n') as f:
    while True:
```

```
        s = f.readline()
        if s == '':
            break
        print(s.strip())
```

输出：

```
hello
world
```

写入：

```
from io import StringIO

with StringIO() as f:
    f.write('hello\nworld\n')
    print(f.getvalue())
```

输出：

```
hello
world
```

`getvalue()` 方法用于获得写入后的str。

BytesIO

BytesIO就是在内存中读写二进制数据。

写入：

```
# coding:utf-8
from io import BytesIO

with BytesIO() as f:
    f.write('测试Bytes'.encode('utf-8'))
    print(f.getvalue())
```

输出：

```
b'\xe6\xb5\x8b\xe8\xaf\x95'
```

注意这里写入的是经过UTF-8编码的bytes。

读取：

```
# coding:utf-8
from io import BytesIO

with BytesIO(b'\xe6\xb5\x8b\xe8\xaf\x95') as f:
    print(f.read())
```

输出：

```
b'\xe6\xb5\x8b\xe8\xaf\x95'
```

`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。

操作文件和目录

Python内置的 `os` 模块可以直接调用操作系统提供的接口函数来操作文件和目录。

查看系统信息

```
>>> import os
>>> os.name
'nt'
>>> os.environ
environ({'PATHEXT':
'.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC;.wlua;.lexe;.PY', ...})
```

要获取某个环境变量的值，可以调用 `os.environ.get('key')` 。
unix操作系统提供 `os.uname()` 来获取详细的系统信息。

操作文件和目录

```
>>> import os

# 查看当前目录的绝对路径:
>>> os.path.abspath('.')
'/Projects/python/code'

```

```
    # 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
7.  操作系统的路径分隔符:
8.  >>> os.path.join('/Projects', 'python')
9.  '/Projects/python'
10. 
11. # 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:
12. >>> os.path.split('/Projects/python/code/os.py')
13. ('/Projects/python/code', 'os.py')
14. >>> os.path.split('/Projects/python/code/')
15. ('/Projects/python/code', '')
16. >>> os.path.split('/Projects/python/code')
17. ('/Projects/python', 'code')
18. 
19. # 获取文件扩展名：
20. >>> os.path.splitext('/Projects/python/code/os.py')
21. ('/Projects/python/code/os', '.py')
22. 
23. # 然后创建一个目录:
24. >>> os.mkdir('/Projects/python/code/testdir')
25. 
26. # 删掉一个目录:
27. >>> os.rmdir('/Projects/python/code/testdir')
28. 
29. # 对文件重命名:
30. >>> os.rename('test.txt', 'test.py')
31. 
32. # 删掉文件:
33. >>> os.remove('test.py')
```

复制文件

`os` 模块中并没有复制文件的方法。原因是复制文件并非由操作系统提供的系统调用。

幸运的是 `shutil` 模块提供了 `copyfile()` 的函数，我们还可以在 `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。

```
1.  # coding : utf-8
2.  import shutil
3.  
4.  shutil.copyfile('os.py', 'os2.py')
```

列出目录内容

```
# coding : utf-8
import os

dirs = os.listdir('.')
L = []
for d in dirs:
    if os.path.isdir(d):
        L.append(d)
print(L)
```

输出：

```
['.idea', 'class', 'module']
```

当然，利用Python的列表生成式，可以很简单：

```
L = [x for x in os.listdir('.') if os.path.isdir(x)]
print(L)
```

要列出所有的 `.py` 文件：

```
L = [x for x in os.listdir('.') if os.path.splitext(x)[1] == '.py']
print(L)
```

输出：

```
['ostest.py', 'ostest2.py']
```

> 参考：
> 1、python 如何读取大文件 - guohuino2 - 博客园
> http://www.cnblogs.com/guohuino2/p/6043196.html
>
> 作者： 飞鸿影
>
> 出处：https://www.cnblogs.com/52fhy/p/6366539.html

Python学习—14 序列化

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，marshalling，flattening等等。

pickle

pickle是Python语言特定的序列化模块，序列化的内容只能是Python才能反序列化。

```
1. pickle.dumps(obj) #把任意对象序列化成一个bytes
2. pickle.dump(obj, fp) #序列化到file-like Object（例如文件）里
3.
4. pickle.loads(bytes_obj) #反序列化
5. pickle.load(fp) #从file-like Object（例如文件）里反序列化
```

示例：

```
1. # coding: utf-8
2. import pickle
3.
4. d = dict(name='Bob', age=20, score=88)
5. print(pickle.dumps(d))
6.
7. # 将序列化的bytes内容保存到文件中：
8. fp = open('pickle.data', 'wb')
9. pickle.dump(d, fp)
```

输出：

```
1. b'\x80\x03}q\x00(X\x05\x00\x00\x00scoreq\x01KXX\x04\x00\x00\x00nameq\x02X\x03\x00
```

反序列化：

```
1. # coding: utf-8
2. import pickle
3.
4. fp = open('pickle.data', 'rb')
5. print(pickle.load(fp))
```

输出：

```
{'age': 20, 'score': 88, 'name': 'Bob'}
```

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

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。

```
json.dumps(obj) #序列化
json.dump(obj, fp) #序列化到file-like Object（例如文件）里

json.loads(str) #反序列化
json.load(fp) #从file-like Object（例如文件）里反序列化
```

示例：

```
# coding: utf-8
import json

d = dict(name='Bob', age=20, score=88)
print(json.dumps(d))
```

输出：

```
{"age": 20, "name": "Bob", "score": 88}
```

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

> 出处: https://www.cnblogs.com/52fhy/p/6368104.html

Python学习—15 日期和时间

方法预览：

```
1. datetime.now() # 当前时间, datetime类型
2. datetime.timestamp() # 时间戳, 浮点类型
3. datetime.strftime('%Y-%m-%d %H:%M:%S') # 格式化日期对象datetime, 字符串类型
4. datetime.strptime('2017-2-6 23:22:13', '%Y-%m-%d %H:%M:%S') # 字符串转日期对象
5. datetime.fromtimestamp(ts) # 获取本地时间, datetime类型
6. datetime.utcfromtimestamp(ts) # 获取UTC时间, datetime类型
```

获取当前时间

```
1. # coding: utf-8
2.
3. from datetime import datetime
4.
5. now = datetime.now()
6. print(now)
7. print(now.strftime('%Y-%m-%d %H:%M:%S'))
8.
9. print(type(now))
```

输出：

```
1. 2017-02-06 23:18:29.624698
2. 2017-02-06 23:18:29
3. <class 'datetime.datetime'>
```

`strftime()` 用于格式化日期对象datetime。另外一个方法 `strptime()` 则负责把一个字符串str转为 `datetime` 对象：

```
1. from datetime import datetime
2.
3. # 注意时间字符串与格式化字符串位置一一对应, 否则报错
4. odate = datetime.strptime('2017-2-6 23:22:13', '%Y-%m-%d %H:%M:%S')
5. print(odate)
6. print(type(odate))
```

输出：

```
2017-02-06 23:22:13
<class 'datetime.datetime'>
```

获取时间戳

```
# coding: utf-8

from datetime import datetime
import time

now = datetime.now()
print(now)

# datetime模块提供
print(now.timestamp())

# time模块提供
print(time.time())
```

输出：

```
2017-02-06 23:26:54.631582
1486394814.631582
1486394814.631582
```

小数位表示毫秒数。

自定义时间转换为时间戳：

```
from datetime import datetime

# 方式1：
odate = datetime.strptime('2017-2-6 23:29:20', '%Y-%m-%d %H:%M:%S')
print(odate.timestamp())

# 方式2：
odate = datetime(2017, 2, 6, 23, 29, 20)
print(odate.timestamp())
```

输出：

```
1486394960.0
1486394960.0
```

注意：timestamp的值是与时区无关的。datetime是有时区的。

下面演示如何把timestamp转换为datetime。

时间戳转日期：

```
# coding: utf-8

from datetime import datetime

now = datetime.now()
ts = now.timestamp()

print(datetime.fromtimestamp(ts))   # 本地时间
print(datetime.utcfromtimestamp(ts))    # UTC时间
```

输出：

```
2017-02-06 23:38:05.213937
2017-02-06 15:38:05.213937
```

datetime加减

可以直接导入timedelta类实现日期加减：

```
# coding: utf-8

from datetime import datetime, timedelta
import time

now = datetime.now()
# now += timedelta(hours=10)
# now += timedelta(minutes=10)
# now += timedelta(weeks=1)
now += timedelta(days=-1, hours=1, minutes=1, seconds=10)

```

```
print(now)
```

输出:

```
2017-02-06 00:47:12.395231
```

python中时间日期格式化符号

```
%y 两位数的年份表示（00-99）
%Y 四位数的年份表示（000-9999）
%m 月份（01-12）
%d 月内中的一天（0-31）
%H 24小时制小时数（0-23）
%I 12小时制小时数（01-12）
%M 分钟数（00=59）
%S 秒（00-59）
%a 本地简化星期名称
%A 本地完整星期名称
%b 本地简化的月份名称
%B 本地完整的月份名称
%c 本地相应的日期表示和时间表示
%j 年内的一天（001-366）
%p 本地A.M.或P.M.的等价符
%U 一年中的星期数（00-53）星期天为星期的开始
%w 星期（0-6），星期天为星期的开始
%W 一年中的星期数（00-53）星期一为星期的开始
%x 本地相应的日期表示
%X 本地相应的时间表示
%Z 当前时区的名称
%% %号本身
```

time模块

下面示例演示time模块里的常用方法:

```
# coding:utf-8
import time

# 获取时间戳
timestamp = time.time()
```

```
print(timestamp)

# 格式时间
print(time.strftime('%Y-%m-%d %H:%M:%S', time.localtime()))

# 返回当地时间下的时间元组t
print(time.localtime())

# 将时间元组转换为时间戳
print(time.mktime(time.localtime()))
t = (2017, 2, 11, 15, 3, 38, 1, 48, 0)
print(time.mktime(t))

# 字符串转时间元组：注意时间字符串与格式化字符串位置一一对应
print(time.strptime('2017 02 11', '%Y %m %d'))

# 睡眠
print('sleeping...')
time.sleep(2) # 睡眠2s
print('sleeping end.')
```

输出：

```
1486797515.78742

2017-02-11 15:18:35

time.struct_time(tm_year=2017, tm_mon=2, tm_mday=11, tm_hour=15, tm_min=18,
tm_sec=35, tm_wday=5, tm_yday=42, tm_isdst=0)

1486797515.0
1486796618.0

time.struct_time(tm_year=2017, tm_mon=2, tm_mday=11, tm_hour=0, tm_min=0,
tm_sec=0, tm_wday=5, tm_yday=42, tm_isdst=-1)

sleeping...
sleeping end.
```

时间元组

下面是Python时间元组：

序号	属性	值
0	tm_year	2008
1	tm_mon	1 到 12
2	tm_mday	1 到 31
3	tm_hour	0 到 23
4	tm_min	0 到 59
5	tm_sec	0 到 61 (60或61 是闰秒)
6	tm_wday	0到6 (0是周一)
7	tm_yday	1 到 366(儒略历)
8	tm_isdst	-1, 0, 1, -1是决定是否为夏令时的旗帜

time模块内置函数

Time 模块包含了以下内置函数，既有时间处理相的，也有转换时间格式的:

序号	函数	描述
1	time.altzone	返回格林威治西部的夏令时地区的偏移秒数。如果该地区在格林威治东部会返回负值（如西欧，包括英国）。对夏令时启用地区才能使用。
2	time.asctime([tupletime])	接受时间元组并返回一个可读的形式为"Tue Dec 11 18:07:14 2008"（2008年12月11日 周二18时07分14秒）的24个字符的字符串。
3	time.clock( )	用以浮点数计算的秒数返回当前的CPU时间。用来衡量不同程序的耗时，比time.time()更有用。
4	time.ctime([secs])	作用相当于asctime(localtime(secs))，未给参数相当于asctime()
5	time.gmtime([secs])	接收时间辍（1970纪元后经过的浮点秒数）并返回格林威治天文时间下的时间元组t。注：t.tm_isdst始终为0
6	time.localtime([secs])	接收时间辍（1970纪元后经过的浮点秒数）并返回当地时间下的时间元组t（t.tm_isdst可取0或1，取决于当地当时是不是夏令时）。
7	time.mktime(tupletime)	接受时间元组并返回时间辍（1970纪元后经过的浮点秒数）。
8	time.sleep(secs)	推迟调用线程的运行，secs指秒数。
9	time.strftime(fmt[,tupletime])	接收以时间元组，并返回以可读字符串表示的当地时间，格式由fmt决定。
10	time.strptime(str,fmt='%a %b %d %H:%M:%S %Y')	根据fmt的格式把一个时间字符串解析为时间元组。
11	time.time( )	返回当前时间的时间戳（1970纪元后经过的浮点秒数）。
12	time.tzset()	根据环境变量TZ重新初始化时间相关设置。

> 出处: https://www.cnblogs.com/52fhy/p/6372194.html

Python学习—16 正则表达式

正则表达式是一种描述性的语言，用来匹配字符串。凡是符合规则的字符串，我们认为就是匹配了。

正则表达式并非Python独有的，它与语言无关。很多语言都支持正则表达式。

我们经常用正则表达式来匹配电子邮件、手机号码、url等等。

来看一个简单的正则表达式，用于匹配手机号码：

```
1. ^1[35789]\d{9}$
```

表示匹配以1开头，第二位是3或5或7或8或9，后面9位是数字，且后面必须以9位数字结尾。满足该规则的手机号就说明匹配该正则了。

Python里 `re` 模块包含所有正则表达式的功能。

注意：由于Python的字符串本身也用 `\` 转义，所以要特别注意：

```
1. s = 'ABC\\'
2. # 对应的正则表达式字符串变成：'ABC\'
```

使用Python的 `r` 前缀，就不用考虑转义的问题了：

```
1. s = r'ABC\'
2. # 对应的正则表达式字符串不变：'ABC\'
```

上面的正则用Python写则是：

```
1. import re
2.
3. m = re.match(r'^1[35789]\d{9}$', '13271222223')
4. print(m)
5.
6. m = re.match(r'^1[35789]\d{9}$', '23271222223')
7. print(m)
```

输出：

```
1. <_sre.SRE_Match object; span=(0, 11), match='13271222223'>
2. None
```

发现第二个例子匹配结果是 `None` 。Python中 `re` 模块 `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` 。

所以我们可以写如下判断代码：

```
import re

if re.match(r'^1[35789]\d{9}$', '13271222223'):
    print('ok')
else:
    print('not match')
```

输出：

```
ok
```

元字符

像 `\d` 属于元字符。

常用的元字符：

代码	说明
.	匹配除换行符以外的任意字符
\w	匹配字母或数字或下划线或汉字
\s	匹配任意的空白符
\d	匹配数字
\b	匹配单词的开始或结束
^	匹配字符串的开始
$	匹配字符串的结束

重复

像 `{9}` 属于重复限定符。

常用的限定符：

代码/语法	说明
*	重复0次或更多次
+	重复1次或更多次

?	重复0次或1次
{n}	重复n次
{n,}	重复n次或更多次
{n,m}	重复n到m次

字符类

`[35789]` 表示匹配3或5或7或8或9中的某一个。像 `[aeiou]` 就匹配任何一个英文元音字母，`[.?!]` 匹配标点符号(.或?或!)。

像 `[0-9]` 代表的含意与 `\d` 就是完全一致的：一位数字；同理 `[a-z0-9A-Z_]` 也完全等同于 `\w` （如果只考虑英文的话）。

使用正则切分字符串

`re` 模块里的 `split` 可以代替常规的 `split` 。示例：

```
1. # coding: utf-8
2.
3. import re
4.
5. string = 'abc d  e'
6. print(string.split(' '))
7.
8. print(re.split(r'\s+', string))
```

输出：

```
1. ['abc', 'd', '', 'e']
2. ['abc', 'd', 'e']
```

我们发现常规的切分字符串无法识别连续的空格，但正则可以。

分组

正则表达式还可以提取子串。用 `()` 表示的就是要提取的分组。

`(\d{1,3}\.){3}\d{1,3}` 是一个简单的IP地址匹配表达式。要理解这个表达式，可以按顺序分析：`\d{1,3}` 匹配1到3位的数字，`(\d{1,3}\.){3}` 匹配三位数字加上一个英文句号(这个整体也就是这个分组)重复3次，最后再加上一个一到三位的数字 `(\d{1,3})` 。

```
import re

m = re.match(r'(\d{1,3}\.){3}\d{1,3}', '11.22.33.44')
print(m.group(0))
print(m.group(1))

print(m.groups())
```

输出：

```
11.22.33.44
33.
('33.',)
```

`group(0)` 永远是原始字符串，`group(1)` 、`group(2)` ……表示第1、2、……个子串。`groups()` 返回所有子串的tuple。

这里由于只有一个分组，所以打印 `group(2)` 会报错。

贪婪匹配

当正则表达式中包含能接受重复的限定符时，通常的行为是（在使整个表达式能得到匹配的前提下）匹配尽可能多的字符。

比如我们要匹配数字102300后面的 `0` ：

```
import re

m = re.match(r'^(\d+)(0*)$', '102300')
print(m.groups())
```

输出：

```
('102300', '')
```

由于贪婪匹配，`\d+` 会一直匹配到末尾，把整个数字都匹配了，`0*` 就只能匹配空字符串了。我们改改：

```
import re

m = re.match(r'^(\d+?)(0*)$', '102300')
```

```
4.  print(m.groups())
```

输出：

```
1.  ('1023', '00')
```

加个 `?` 就可以让 `\d+` 采用非贪婪匹配。

懒惰限定符：

代码/语法	说明
*?	重复任意次，但尽可能少重复
+?	重复1次或更多次，但尽可能少重复
??	重复0次或1次，但尽可能少重复
{n,m}?	重复n到m次，但尽可能少重复
{n,}?	重复n次以上，但尽可能少重复

正则表达式非常强大，本节只是讲解了基础。更多关于正则的知识，大家可以找资料学习。

> 参考：
> 1、正则表达式30分钟入门教程
> http://deerchao.net/tutorials/regex/regex.htm
>
>

Python学习—17 访问数据库

实际开发中，我们会经常用到数据库。

Python里对数据库的操作API都很统一。

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内置了sqlite3。

```
1.  #coding:utf-8
2.  import sqlite3
3.
4.  conn = sqlite3.connect('test.db')
5.  cursor = conn.cursor()
6.
7.  # sqlite创建表时，若id为INTEGER类型且为主键，可以自动递增，在插入数据时id填NULL即可
    # cursor.execute('create table user(id integer primary key, name varchar(25))')
8.  #执行一次
9.
10. # 插入一条数据
11. cursor.execute('insert into user(id,name)values(NULL,"yjc")')
12.
13. # 返回影响的行数
14. print(cursor.rowcount)
15.
16. #提交事务，否则上述SQL不会提交执行
17. conn.commit()
18.
19. # 执行查询
20. cursor.execute('select * from user')
21.
22. # 获取查询结果
23. print(cursor.fetchall())
24.
25. # 关闭游标和连接
26. cursor.close()
27. conn.close()
```

输出：

```
1
[(1, 'yjc'), (2, 'yjc')]
```

我们发现Python里封装的数据库操作很简单：
1、获取连接 `conn` ；
2、获取游标 `cursor` ；
3、使用 `cursor.execute()` 执行SQL语句；
4、使用 `cursor.rowcount` 返回执行insert，update，delete语句受影响的行数；
5、使用 `cursor.fetchall()` 获取查询的结果集。结果集是一个list，每个元素都是一个tuple，对应一行记录；
6、关闭游标和连接。

如果SQL语句带有参数，那么需要把参数按照位置传递给 `cursor.execute()` 方法，有几个 `?` 占位符就必须对应几个参数，示例：

```
cursor.execute('select * from user where name=? ', ['abc'])
```

为了能在出错的情况下也关闭掉 `Connection` 对象和 `Cursor` 对象，建议实际项目里使用 `try:...except:...finally:...` 结构。

MySQL

MySQL是最流行的关系数据库。

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。

使用需要先安装MySQL：https://dev.mysql.com/downloads/mysql/

Windows版本安装时注意选择 `UTF-8` 编码，以便正确地处理中文。

MySQL的配置文件是 `my.ini` ，Linux一般位于 `/etc/my.cnf` 。配置里需要设置编码为utf-8。配置示例：

```
[client]
default-character-set = utf8

[mysqld]
default-storage-engine = INNODB
```

```
character-set-server = utf8
collation-server = utf8_general_ci
```

Python并未内置MySQL的驱动。需要先安装：

```
$ pip3 install mysql-connector

Collecting mysql-connector
  Downloading mysql-connector-2.1.4.zip (355kB)
    100% |████████████████████████████████| 358kB 355kB/s
Building wheels for collected packages: mysql-connector
  Running setup.py bdist_wheel for mysql-connector ... done
Successfully built mysql-connector
Installing collected packages: mysql-connector
Successfully installed mysql-connector-2.1.4
```

Python使用MySQL示例：

```
# coding: utf-8
import mysql.connector

conn = mysql.connector.connect(user='root', password='123456', database='test')

cursor = conn.cursor()

cursor.execute("insert into user(id,name,age)values(null,'python', 20)")
print(cursor.rowcount)
conn.commit()

cursor.execute("select * from user order by id desc limit 3")
print(cursor.fetchall())

cursor.close()
conn.close
```

输出：

```
1
[(25, 'python', 1, 20, 1), (24, 'python', 1, 20, 1), (23, 't2', 2, 23, 1)]
```

如果SQL语句带有参数，那么需要把参数按照位置传递给 `cursor.execute()` 方法，MySQL的占位符

是 `%s` ，示例：

```
1.  cursor.execute('select * from user where name=%s and age=%s', ['python', 20])
```

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。

Python学习—18 进程和线程

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。

进程

fork调用

通过 `fork()` 系统调用，就可以生成一个子进程。

下面先了解下关于 `fork()` 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里的 `os` 模块封装了常见的系统调用，包括 `fork()` ：

```
1. # coding: utf-8
2. import os
3.
4. print('Process (%s) start...' % os.getpid())
5. pid = os.fork()
6. if pid == 0:
       print('I am child Process %s , my parent is %s ' % (os.getpid(),
7. os.getppid()))
8. else:
9.     print('I am parent Process %s , my child is %s ' % (os.getpid(), pid))
```

上面代码无法运行在Windows系统上（没有fork调用），需要运行在Unix/Linux操作系统。输出：

```
1. # ./user_process.py
2. Process (25464) start...
3. I am parent Process 25464 , my child is 25465
4. I am child Process 25465 , my parent is 25464
```

multiprocessing

虽然 `fork()` 调用无法在Windows调用，但Python也提供了跨平台的多进程支持。使用 `multiprocessing` 即可创建跨平台多进程：

```
1.  # coding: utf-8
2.  from multiprocessing import Process
3.  import os
4.
5.  def run_proc(name):
6.      print('Child Process %s %s is running...' % (name, os.getpid()))
7.
8.  if __name__ == '__main__':
9.      print('Parent Process %s is running...' % os.getpid() )
10.     p = Process(target=run_proc, args=('testProcess', ))
11.     p.start()
12.     p.join()
```

输出：

```
1.  Parent Process 10488 is running...
2.  Child Process testProcess 3356 is running...
```

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。 `jonin()` 方法用于等待子进程结束后再继续往下运行，相当于阻塞了进程的异步执行。

这里的 `if __name__ == '__main__'` 用于仅允许在命令行下直接运行。如果是另一个模块引入该文件，里面的代码不会执行。

> 在python中，当一个module作为整体被执行时， `moduel.__name__` 的值将是 `__main__` ；而当一个module被其它module引用时， `module.__name__` 将是module自己的名字，当然一个module被其它module引用时，其本身并不需要一个可执行的入口 `main` 了。

上面的多进程创建代码风格与后文讲的创建多线程很相似。

进程池

如果要启动大量的子进程，可以用进程池(Pool)的方式批量创建子进程：

> user_process_pool.py

```
1.  # coding: utf-8
2.  from multiprocessing import Pool
```

```
import os,time,random

def run_task(name):
    print('Run task %s' % name)
    s_start = time.time()
    time.sleep(random.random())
    s_end = time.time()
    print('Task %s run %.2f sec' % (name, s_end - s_start))

if __name__ == '__main__':
    print('Parent Process %s is running...' % os.getpid())

    p = Pool(5)
    for i in range(5):
        p.apply_async(run_task, args=(i,))
    print('all subProcess will running...')
    p.close()
    p.join()
    print('all subProcess running ok')
```

输出：

```
Parent Process 10576 is running...
all subProcess will running...
Run task 0
Run task 1
Run task 2
Run task 3
Run task 4
Task 3 run 0.16 sec
Task 1 run 0.31 sec
Task 4 run 0.38 sec
Task 2 run 0.44 sec
Task 0 run 0.89 sec
all subProcess running ok
```

如果修改 `Pool()` 的参数为1，看下输出：

```
Parent Process 6252 is running...
all subProcess will running...
Run task 0
Task 0 run 0.77 sec
```

```
Run task 1
Task 1 run 0.22 sec
Run task 2
Task 2 run 0.73 sec
Run task 3
Task 3 run 0.54 sec
Run task 4
Task 4 run 0.02 sec
all subProcess running ok
```

说明有Pool进程池的数量有多少，就可以最多同时运行多少个进程。如果不设置参数，Pool的默认大小是CPU的核数。

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。

外部子进程

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：

```
# coding:utf-8
import subprocess

r = subprocess.call(['ping', 'www.python.org'])
print(r)
```

然后运行：

```
$ python user_subprocess.py

正在 Ping www.python.org [151.101.72.223] 具有 32 字节的数据:
来自 151.101.72.223 的回复: 字节=32 时间=189ms TTL=53
来自 151.101.72.223 的回复: 字节=32 时间=183ms TTL=53
来自 151.101.72.223 的回复: 字节=32 时间=192ms TTL=53
来自 151.101.72.223 的回复: 字节=32 时间=192ms TTL=53

151.101.72.223 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
```

```
    最短 = 183ms, 最长 = 192ms, 平均 = 189ms
0
```

相当于命令行运行：

```
ping www.python.org
```

进程间通信

进程间肯定需要互相通信的。这里我们使用队列来实现一个简单的例子：

```
# coding: utf-8
from multiprocessing import Process,Queue
import os,time

def write(q):
    print('write Process %s is running... ' % os.getpid())
    for x in ['python', 'c', 'java']:
        q.put(x)
        print('write to Queue : %s' % x)

def read(q):
    print('read Process %s is running... ' % os.getpid())
    while True:
        r = q.get(True)
        print('read from Queue : %s' % r)
    pass

if __name__ == '__main__':
    print('MainProcess %s is running...' % os.getpid())
    q = Queue()
    p1 = Process(target=write, args=(q,))
    p2 = Process(target=read, args=(q,))

    p1.start()
    p2.start()
    p1.join()
    p2.join()
    # p2.terminate()
```

输出：

```
MainProcess 9680 is running...
write Process 10232 is running...
write to Queue : python
write to Queue : c
write to Queue : java
read Process 8024 is running...
read from Queue : python
read from Queue : c
read from Queue : java
```

由于p2进程里是死循环，默认执行完毕后程序不会退出，可以使用 `p2.terminate()` 进程强行退出。

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。

创建线程

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

```
# coding: utf-8
import threading,time

def test():
    print('Thread %s is running... ' % threading.current_thread().name)
    print('waiting 3 seconds... ')
    time.sleep(3)
    print('Hello Thread!')
    print('Thread %s is end. ' % threading.current_thread().name)

print('Thread %s is running... ' % threading.current_thread().name)
t = threading.Thread(target = test, name = 'TestThread')
t.start()
t.join()
print('Thread %s is end. ' % threading.current_thread().name)
```

输出：

```
Thread MainThread is running...
Thread TestThread is running...
waiting 3 seconds...
Hello Thread!
Thread TestThread is end.
Thread MainThread is end.
```

`t.start()` 用于启动线程。 `t.join()` 的作用是等待线程执行完毕，否则会不等待线程执行完毕就执行下面的代码了，因为线程执行是异步的。

主线程实例的名字叫 `MainThread` ，子线程的名字在创建时指定，这里我们用 `TestThread` 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。

线程锁

多线程与多进程最大的不同就是：对于同一个变量，对于多个线程是共享的，但多进程里每个进程各自会复制一份。

所以，在多线程里，最大的一个隐患就是多个线程同时改一个变量，可能就把变量内容改乱了。看下面例子如何改乱一个变量：

```
# coding:utf-8
import threading

amount = 0
def changeValue(x):
    global amount
    amount = amount + x
    amount = amount - x

# 批量运行改值
def batchRunThread(x):
    for i in range(100000):
        changeValue(x)

# 创建2个线程
t1 = threading.Thread(target=batchRunThread, args=(5,), name = 'Thread1')
t2 = threading.Thread(target=batchRunThread, args=(15,), name = 'Thread2')

t1.start()
t2.start()
t1.join()
```

```
t2.join()

print(amount)
```

正常情况下会输出0。但是运行多次，发现结果不一定是0。由于线程的调度是由操作系统决定的，当t1、t2线程交替执行时，只要循环次数足够多，结果就可能被改乱了。

原因是高级语言的一条语句执行在CPU执行是多个语句，即使是一个简单的计算：

```
amount = amount + x
```

CPU会执行下列运算：
1、计算amount + x，存入临时变量中；
2、将临时变量的值赋给amount。

想要解决这个问题，就要使用线程锁：线程执行 `changeValue()` 时先获取锁，这时候其它线程想要执行 `changeValue()` ，想要等待之前那个线程释放锁。Python里通过 `threading.Lock()` 来实现。

```
# coding:utf-8
import threading

lock = threading.Lock()

amount = 0
def changeValue(x):
    global amount
    amount = amount + x
    amount = amount - x

# 批量运行改值
def batchRunThread(x):
    for i in range(100000):
        lock.acquire() # 获得锁
        try:
            changeValue(x)
        finally:
            lock.release() # 释放锁

# 创建2个线程
t1 = threading.Thread(target=batchRunThread, args=(5,), name = 'Thread1')
t2 = threading.Thread(target=batchRunThread, args=(15,), name = 'Thread2')
```

```
24.
25.  t1.start()
26.  t2.start()
27.  t1.join()
28.  t2.join()
29.
30.  print(amount)
```

这时候不管循环多少次，输出的结果永远是0。

想要注意的是，获得锁后一定要记得释放，否则其它线程一直在等待，就成了死锁。这里使用 `try...finally...` 保证最后一定会释放锁。

> 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。

如果锁使用不当，也可能造成死锁，程序无法正常运行。

> 1、说说进程与线程的区别与联系 - oyzway - 博客园
> http://www.cnblogs.com/way_testlife/archive/2011/04/16/2018312.html
> 2、进程与线程的一个简单解释 - 阮一峰的网络日志
> http://www.ruanyifeng.com/blog/2013/04/processes_and_threads.html
>
>

Python学习—19 网络编程

TCP编程

Client

创建一个基于TCP连接的Socket:

```
# coding: utf-8
import socket

# 创建一个TCP连接:
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# 建立连接:
s.connect(('52fhy.com', 80))

# 发送HTTP请求
s.send(b'GET / HTTP/1.1\r\nHost: 52fhy.com\r\nConnection: close\r\n\r\n')

# 读取响应
buffer = []
while True:
    d = s.recv(1024)
    if d:
        buffer.append(d)
    else:
        break

data = b''.join(buffer)

# 关闭连接
s.close()

# 解析响应
header, body = data.split(b'\r\n\r\n', 1)
print(header.decode('utf-8'))

with open('52fhy.html', 'wb') as f:
    f.write(body)
```

输出：

```
HTTP/1.1 200 OK
Server: nginx/1.4.4
Date: Sat, 11 Feb 2017 08:16:43 GMT
Content-Type: text/html
Content-Length: 8368
Last-Modified: Mon, 19 Sep 2016 07:04:02 GMT
Connection: close
Vary: Accept-Encoding
ETag: "57df8de2-20b0"
Accept-Ranges: bytes
```

代码说明：

1、创建socket连接的时候使用 `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。

2、建立连接的 `connect()` 接受一个tuple，包含地址和端口号。

3、发送请求是模拟浏览器发送一个Request，实际浏览器请求时包含更多项：

```
GET / HTTP/1.1
Host: 52fhy.com
Connection: keep-alive
Cache-Control: max-age=0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/50.0.2661.102 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6
```

换行使用 `\r\n` ， `\r\n\r\n` 则表示该段内容的结束，用于分隔请求的header和请求的body。

4、读取远程服务器响应则使用 `recv()` 每次接受1024byte内容。注意接收的是字节内容，不是字符串，所以拼接以 `b` 开头。

5、读取响应完毕，关闭socket连接。

6、通过 `\r\n\r\n` 可以区分响应头和返回的body内容（即网页），这里将网页内容保存到了文件里。

Server

上节例子说明了如何编写客户端，这节讲如何创建一个基于TCP的Server端。

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。

```
# coding: utf-8
import socket, threading

# 创建一个TCP连接
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# 绑定IP和端口
s.bind(('127.0.0.1', 9999))

# 开始监听客户端连接
# 传入的参数指定等待连接的最大数量
s.listen(3)

# 输出提示语
print('Server is running on %s:%s' % ('127.0.0.1', 9999))
print('Waiting for connection...')

def tcp_link(sock, addr):
    print('Accept new connection from %s:%s...' % addr) # 参数addr是一个tuple
    sock.send(b'Welcome!') # 发送问候语给客户端

    while True:
        r = sock.recv(1024)
        if not r or r.decode('utf-8') == 'exit': # 结束连接指令
            break

        # 输出客户端发来的信息，注意元组拼接
        msg = addr + (r.decode('utf-8'),)
        print('%s:%s : %s' % msg)

        # 回复客户端
        sock.send( ('you say: %s' % r.decode('utf-8') ).encode('utf-8') )
    sock.close()
    print('Client %s:%s is closed.' % addr)

while True:
```

```
    # 接受一个新连接:
    sock, addr = s.accept()
    print(addr) # ('127.0.0.1', 62090)

    # 创建新线程来处理TCP连接:
    # 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
    t = threading.Thread(target=tcp_link, args=(sock, addr))
    t.start()
```

还需要个客户端发送消息:

```
# coding: utf-8
import socket

# 创建一个TCP连接
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# 连接服务端
s.connect(('127.0.0.1', 9999))

# 接收来自服务端的信息
print(s.recv(1024).decode('utf-8'))

# 发送数据给服务端
for x in [b'Python', b'PHP']:
    s.send(x)
    print(s.recv(1024).decode('utf-8'))

# 发送断开连接命令
s.send(b'exit')

s.close()
```

先运行服务端:

```
erver is running on 127.0.0.1:9999
Waiting for connection...
```

这时候运行一个客户端:

```
Welcome!
```

```
you say: Python
you say: PHP
```

服务端输出：

```
('127.0.0.1', 62428)
Accept new connection from 127.0.0.1:62428...
127.0.0.1:62428 : Python
127.0.0.1:62428 : PHP
Client 127.0.0.1:62428 is closed.
```

编写代码需要注意：
1、服务端和客户端互相发送和接收到的是字节Bytes，需要使用 `encode()` 或者 `decode()` 转码；
2、服务端使用 `accept()` 接收到的是一个元组，例如： `('127.0.0.1', 62090)` ；
3、服务端必须使用多线程以处理来自多个客户端的连接；
4、服务端同一个端口，被一个Socket绑定了以后，就不能被别的Socket绑定了。

> 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。

UDP编程

TCP建立的是可靠的连接，而使用UDP，无需建立连接，只要知道对方的IP和端口，就可以发送数据。

UDP是一个非连接的协议，传输数据之前源端和终端不建立连接。UDP使用尽最大努力交付，即不保证可靠交付。

使用UDP虽然传输数据不可靠，但比TCP要快。

user_udp_server.py:

```
# coding: utf-8
import socket

# 建立UDP连接：SOCK_DGRAM表示UDP
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# 绑定地址和端口，之后无需监听
s.bind(('127.0.0.1', 9999))

```

```
print('UDP server is running on %s:%s' % ('127.0.0.1', 9999))

while True:
    # recvfrom()返回客户端发过来的数据及地址端口信息
    data, addr = s.recvfrom(1024)
    msg = addr + (data.decode('utf-8'),)
    print('Received from %s:%s : %s' % msg)

    # 使用sendto()发送消息，注意第二个参数是addr
    s.sendto(('you say : %s ' % data.decode('utf-8')).encode('utf-8') , addr)
```

user_udp_client.py

```
# coding: utf-8
import socket

# 建立UDP连接：SOCK_DGRAM表示UDP
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# 服务端地址
serv_addr = ('127.0.0.1', 9999)

#无需使用connect()连接

# 发送数据给服务端
for x in [b'Python', b'PHP']:
    s.sendto(x, serv_addr)
    print(s.recv(1024).decode('utf-8'))

# s.close()
```

先运行服务端：

```
UDP server is running on 127.0.0.1:9999
```

再运行客户端：

```
you say : Python
you say : PHP
```

服务端输出：

```
Received from 127.0.0.1:57827 : Python
Received from 127.0.0.1:57827 : PHP
```

与TCP不同的是：

1、服务端和客户端建立连接时选择UDP：`SOCK_DGRAM`；

2、服务端无需进行监听`listen()`；

3、服务端使用`recvfrom()`返回客户端发过来的数据及地址端口信息，而不是`recv()`或者`accept()`；

4、服务端和客户端使用`sendto()`发送数据，第二个参数必填，是addr元组；

5、客户端创建连接后无需使用`connect()`连接服务端；

这里服务端我们没有使用多线程，因为比较简单。需要注意的是客户端发送数据如果加入一句`s.close()`，本次连接关闭，服务端也会退出。

> 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。
>
> 作者： 飞鸿影
> 版权：本作品采用「署名-非商业性使用-相同方式共享 4.0 国际」许可协议进行许可。
> 出处：https://www.cnblogs.com/52fhy/p/6389634.html

Python学习—20 Web开发

HTTP格式

HTTP协议是基于TCP和IP协议的。HTTP协议是一种文本协议。

每个HTTP请求和响应都遵循相同的格式，一个HTTP包含Header和Body两部分，其中Body是可选的。

HTTP请求格式：

GET：

```
GET /path HTTP/1.1
Header1: Value1
Header2: Value2
Header3: Value3
```

POST：

```
POST /path HTTP/1.1
Header1: Value1
Header2: Value2
Header3: Value3

body data goes here...
```

Header部分每行用 `\r\n` 换行，每行里键名和键值之间以 `:` 分割，注意冒号后有个空格。

当遇到 `\r\n\r\n` 时，Header部分结束，后面的数据全部是Body。

HTTP响应格式：

```
200 OK
Header1: Value1
Header2: Value2
Header3: Value3

body data goes here...
```

HTTP响应如果包含body，也是通过 `\r\n\r\n` 来分隔的。

> 请再次注意，Body的数据类型由 `Content-Type` 头来确定，如果是网页，Body就是文本，如果是图片，Body就是图片的二进制数据。

Body数据是可以被压缩的，如果看到 `Content-Encoding` ，说明网站使用了压缩。最常见的压缩方式是gzip。

WSGI接口

了解了HTTP协议的格式后，我们可以理解一个Web应用的本质：
1、浏览器发送HTTP请求给服务器；
2、服务器接收请求后，生成HTML；
3、服务器把生成的HTML作为HTTP响应的body返回给浏览器；
4、浏览器接收到HTTP响应后，解析HTTP里body并显示。

接受HTTP请求、解析HTTP请求、发送HTTP响应实现起来比较复杂，有专门的服务器软件来实现，例如Nginx，Apache。我们要做的就是专注于生成HTML文档。

Python里也提供了一个比较底层的 `WSGI` （Web Server Gateway Interface）接口来实现TCP连接、HTTP原始请求和响应格式。实现了该接口定义的内容，就可以实现类似Nginx、Apache等服务器的功能。

`WSGI` 接口定义要求Web开发者实现一个函数，就可以响应HTTP请求，示例：

```
1.  def application(environ, start_response):
2.      start_response('200 OK', [('Content-Type', 'text/html')])
3.      return [b'<h1>Hello, web!</h1>']
```

这是一个简单的文本版本的 `Hello, web!` 。

上面的 `application()` 函数就是符合 `WSGI` 标准的一个HTTP处理函数，它接收两个参数：

```
1.  environ：一个包含所有HTTP请求信息的dict对象；
2.  start_response：一个发送HTTP响应的函数。
```

有了WSGI，我们关心的就是如何从 `environ` 这个dict对象拿到HTTP请求信息，然后构造HTML，通过 `start_response()` 发送Header，最后返回Body。

整个 `application()` 函数本身没有涉及到任何解析HTTP的部分，即底层代码不需要自己编写，只负责在更高层次上考虑如何响应请求就可以了。

但是， `application()` 函数由谁来调用呢？因为这里的参数 `environ` 、 `start_response` 我们没法提供，返回的bytes也没法发给浏览器。

`application()` 函数必须由 `WSGI` 服务器来调用。

有很多符合WSGI规范的服务器，Python提供了一个最简单的 `WSGI` 服务器，可以把我们的Web应用程序跑起来。这个模块叫 `wsgiref` ，它是用纯Python编写的 `WSGI` 服务器的参考实现。所谓“参考实现”是指该实现完全符合 `WSGI` 标准，但是不考虑任何运行效率，仅供开发和测试使用。

运行WSGI服务

有了 `wsgiref` ，我们可以非常快的实现一个简单的web服务器：

```
1.  # coding: utf-8
2.
3.  from wsgiref.simple_server import make_server
4.
5.  def application(environ, start_response):
6.      print(environ)
7.      start_response('200 OK', [('Content-Type', 'text/html')])
8.      return [b'<h1>Hello web!</h1>']
9.
10. print('HTTP server is running on http://127.0.0.1:9999')
11.
12. # 创建一个服务器，IP地址可以为空，端口是9999，处理函数是application:
13. httpd = make_server('', 9999, application)
14. httpd.serve_forever()
```

运行后访问 `http://127.0.0.1:9999/` ，会看到：

```
1.  Hello web!
```

> 扩展知识：
> `make_server()` 里第一个参数如果为空，实际等效于 `0.0.0.0` ，表示监听本地所有ip地址（包括 `127.0.0.1` ）。

通过Chrome浏览器的控制台，我们可以查看到浏览器请求和服务器响应信息：

```
1.  # 请求信息：
2.  GET / HTTP/1.1
3.  Host: 127.0.0.1:9999
4.  Connection: keep-alive
5.  Cache-Control: max-age=0
    Accept:
6.  text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
```

```
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/50.0.2661.102 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6
Cookie: _ga=GA1.1.948200530.1463673425

# 响应信息：
HTTP/1.0 200 OK
Date: Sun, 12 Feb 2017 05:20:31 GMT
Server: WSGIServer/0.2 CPython/3.4.3
Content-Type: text/html
Content-Length: 19

<h1>Hello web!</h1>
```

我们再看终端的输出信息：

```
$ python user_wsgiref_server.py
HTTP server is running on http://127.0.0.1:9999
127.0.0.1 - - [12/Feb/2017 13:18:38] "GET / HTTP/1.1" 200 19
127.0.0.1 - - [12/Feb/2017 13:18:39] "GET /favicon.ico HTTP/1.1" 200 19
```

如果我们打印 `environ` 参数信息，会看到如下值：

```
{
    "SERVER_SOFTWARE": "WSGIServer/0.1 Python/2.7.5",
    "SCRIPT_NAME": "",
    "REQUEST_METHOD": "GET",
    "SERVER_PROTOCOL": "HTTP/1.1",
    "HOME": "/root",
    "LANG": "en_US.UTF-8",
    "SHELL": "/bin/bash",
    "SERVER_PORT": "9999",
    "HTTP_HOST": "dev.banyar.cn:9999",
    "HTTP_UPGRADE_INSECURE_REQUESTS": "1",
    "XDG_SESSION_ID": "64266",
    "HTTP_ACCEPT":
"text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8",
    "wsgi.version": "0",
    "wsgi.errors": "",
    "HOSTNAME": "localhost",
```

```
    "HTTP_ACCEPT_LANGUAGE": "zh-CN,zh;q=0.8,en;q=0.6",
    "PATH_INFO": "/",
    "USER": "root",
    "QUERY_STRING": "",
    "PATH":
"/usr/local/php/bin:/usr/local/php/sbin:/usr/local/sbin:/usr/local/bin:/usr/sbin:
    "HTTP_USER_AGENT": "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/50.0.2661.102 Safari/537.36",
    "HTTP_CONNECTION": "keep-alive",
    "SERVER_NAME": "localhost",
    "REMOTE_ADDR": "192.168.0.101",
    "wsgi.url_scheme": "http",
    "CONTENT_LENGTH": "",
    "GATEWAY_INTERFACE": "CGI/1.1",
    "CONTENT_TYPE": "text/plain",
    "REMOTE_HOST": "",
    "HTTP_ACCEPT_ENCODING": "gzip, deflate, sdch"
}
```

为显示方便，已精简部分信息。有了环境变量信息，我们可以对程序做些修改，可以动态显示内容：

```
def application(environ, start_response):
    print(environ['PATH_INFO'])
    start_response('200 OK', [('Content-Type', 'text/html')])
    body = '<h1>Hello %s!</h1>'  % (environ['PATH_INFO'][1:] or 'web' )
    return [body.encode('utf-8')]
```

以上使用了 `environ` 里的 `PATH_INFO` 的值。我们在浏览器输入 `http://127.0.0.1:9999/python` ，浏览器会显示：

```
Hello python!
```

终端的输出信息：

```
$ python user_wsgiref_server.py
HTTP server is running on http://127.0.0.1:9999
/python
127.0.0.1 - - [12/Feb/2017 13:54:57] "GET /python HTTP/1.1" 200 22
/favicon.ico
127.0.0.1 - - [12/Feb/2017 13:54:58] "GET /favicon.ico HTTP/1.1" 200 27
```

web框架

实际项目开发中，我们不可能使用 `swgiref` 来实现服务器，因为WSGI提供的接口虽然比HTTP接口高级了不少，但和Web App的处理逻辑比，还是比较低级。我们需要使用成熟的web框架。

由于用Python开发一个Web框架十分容易，所以Python有上百个开源的Web框架。部分流行框架：

```
1. Flask：轻量级Web应用框架；
2. Django：全能型Web框架；
3. web.py：一个小巧的Web框架；
4. Bottle：和Flask类似的Web框架；
5. Tornado：Facebook的开源异步Web框架
```

Flask

Flask是一个使用 Python 编写的轻量级 Web 应用框架。其 WSGI 工具箱采用 Werkzeug ，模板引擎则使用 Jinja2 。

安装非常简单：

```
1. pip install flask
```

控制台输出：

```
1. Collecting flask
2.   Downloading Flask-0.12-py2.py3-none-any.whl (82kB)
3.     100% |████████████████████████████████| 92kB 163kB/s
4. Collecting itsdangerous>=0.21 (from flask)
5.   Downloading itsdangerous-0.24.tar.gz (46kB)
6.     100% |████████████████████████████████| 51kB 365kB/s
7. Collecting click>=2.0 (from flask)
8.   Downloading click-6.7-py2.py3-none-any.whl (71kB)
9.     100% |████████████████████████████████| 71kB 349kB/s
10. Collecting Jinja2>=2.4 (from flask)
11.   Downloading Jinja2-2.9.5-py2.py3-none-any.whl (340kB)
12.     100% |████████████████████████████████| 348kB 342kB/s
13. Collecting Werkzeug>=0.7 (from flask)
14.   Downloading Werkzeug-0.11.15-py2.py3-none-any.whl (307kB)
15.     100% |████████████████████████████████| 317kB 194kB/s
16. Collecting MarkupSafe>=0.23 (from Jinja2>=2.4->flask)
17.   Downloading MarkupSafe-0.23.tar.gz
18. Building wheels for collected packages: itsdangerous, MarkupSafe
```

```
  Running setup.py bdist_wheel for itsdangerous ... done
Successfully built itsdangerous MarkupSafe
Installing collected packages: itsdangerous, click, MarkupSafe, Jinja2,
Werkzeug, flask
Successfully installed Jinja2-2.9.5 MarkupSafe-0.23 Werkzeug-0.11.15 click-6.7
flask-0.12 itsdangerous-0.24
```

安装完flask会同时安装依赖模块: `itsdangerous` , `click` , `MarkupSafe` , `Jinja2` , `Werkzeug` 。

现在我们来写个简单的登录功能，主要是三个页面:

- 首页，显示 `home` 字样;
- 登录页，地址 `/login` ，有登录表单;
- 登录后的欢迎页面，如果登录成功，提示欢迎语，否则提示用户名不正确。

那么一共有3个URL:

- GET /: 首页，返回Home;
- GET /login: 登录页，显示登录表单;
- POST /login: 处理登录表单，显示登录结果。

user_flask_app.py

```
# coding: utf-8

from flask import Flask
from flask import request

app = Flask(__name__)

# 首页
@app.route('/', methods=['GET', 'POST'])
def home():
    return '<h1>Home</h1><p><a href="/login">去登录</a></p>'

# 登录页
@app.route('/login', methods=['get'])
def login():
    return '''<form action="/login" method="post">
              <p>用户名：<input name="username"></p>
              <p>密码：<input name="password" type="password"></p>
              <p><button type="submit">登录</button></p>
```

```
            </form>'''

# 登录页处理
@app.route('/login', methods=['post'])
def do_login():
    # 从request对象读取表单内容：
    param = request.form
    if(param['username'] == 'yjc' and param['password'] == 'yjc'):
        return '欢迎您 %s !' % param['username']
    else:
        return '用户名或密码不正确。'
    pass

if __name__ == '__main__':
    # run()方法参数可以都为空，使用默认值
    app.run('', 5000)
```

我们可以打开：http://localhost:5000/ 看效果。实际的Web App应该拿到用户名和口令后，去数据库查询再比对，来判断用户是否能登录成功。

通过代码我们可以发现，Flask通过Python的装饰器在内部自动地把URL和函数给关联起来。

注意代码里同一个 `URL/login` 分别有 `GET` 和 `POST` 两种请求，可以映射到两个处理函数中。

使用模板

Web框架让我们从编写底层WSGI接口拯救出来了，极大的提高了我们编写程序的效率。

但代码里嵌套太多的html让整个代码易读性变差，使程序变得复杂。我们需要将后端代码逻辑与前端html分离出来。这就是传说中的 `MVC` ：Model-View-Controller，中文名“模型-视图-控制器”。

`Controlle` r负责业务逻辑，比如检查用户名是否存在，取出用户信息等等；

`View` 负责显示逻辑，通过简单地替换一些变量，View最终输出的就是用户看到的HTML。

‘Model’负责数据的获取，如从数据库查询用户信息等。Model简单可以理解为数据。

那么就是： `Model` 获取数据， `Controlle` 处理业务逻辑， `View` 显示数据。

现在，我们把上次直接输出字符串作为HTML的例子用MVC模式改写一下：

```
# coding: utf-8

```

```
from flask import Flask,request,render_template

app = Flask(__name__)

# 首页
@app.route('/', methods=['GET', 'POST'])
def home():
    return render_template('home.html')

# 登录页
@app.route('/login', methods=['get'])
def login():
    return render_template('login.html', param = [])

# 登录页处理
@app.route('/login', methods=['post'])
def do_login():
    param = request.form
    if(param['username'] == 'yjc' and param['password'] == 'yjc'):
        return render_template('welcome.html', username = param['username'])
    else:
        return render_template('login.html', msg = '用户名或密码不正确。', param = param)
    pass

if __name__ == '__main__':
    app.run('', 5000)
```

Flask通过 `render_template()` 函数来实现模板的渲染。和Web框架类似，Python的模板也有很多种。Flask默认支持的模板是 `jinja2` 。

模板页面：

home.html

```
<h1>Home</h1><p><a href="/login">去登录</a></p>
```

login.html

```
{% if msg %}
<p style="color:red;">{{ msg }}</p>
{% endif %}
<form action="/login" method="post">
```

```
    <p>用户名：<input name="username" value="{{ param.username }}"></p>
    <p>密码：<input name="password" type="password"></p>
    <p><button type="submit">登录</button></p>
</form>
```

welcome.html

```
<p>欢迎您， {{ username }} !</p>
```

项目目录：

```
user_flask_app
    |-- templates
        |-- home.html
        |-- login.html
        |-- welcome.html
    |-- user_flask_app.py
```

`render_template()` 函数第一个参数是模板名，默认是 `templates` 目录下。后面的参数是传给模板的变量。变量的值可以是数字、字符串、列表等等。

在Jinja2模板中，我们用 `{{ name }}` 表示一个需要替换的变量。很多时候，还需要循环、条件判断等指令语句，在Jinja2中，用 `{% ... %}` 表示指令。

比如循环输出页码：

```
{% for i in page_list %}
    <a href="/page/{{ i }}">{{ i }}</a>
{% endfor %}
```

除了 `Jinja2` ，常见的模板还有：

```
Mako：用<% ... %>和${xxx}的一个模板；
Cheetah：也是用<% ... %>和${xxx}的一个模板；
Django：Django是一站式框架，内置一个用{% ... %}和{{ xxx }}的模板。
```

Python学习—21 电子邮件

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。

发送简单邮件

下面是最简单的发邮件的例子：

```
1.  # coding: utf-8
2.  import smtplib
3.  from email.mime.text import MIMEText
4.  from email.header import Header
5.
6.  # 连接邮件服务器
7.  mail_host = 'smtp.163.com'
8.  mail_port = '25'
9.  mail_from = 'xxx@163.com'
10. smtp = smtplib.SMTP()
11. # smtp.set_debuglevel(1) # 打印出和SMTP服务器交互的所有信息
12. smtp.connect(mail_host, mail_port)
13. smtp.login(mail_from, 'xxx')
14.
15. # 构造邮件对象
16. msg = MIMEText('Python测试', 'plain', 'utf-8')
17. msg['From'] = mail_from
18. msg['To'] = 'xxx@qq.com'
19. msg['Subject'] = Header('测试邮件', 'utf-8')
20.
21. # 发送邮件
22. smtp.sendmail(mail_from, 'xxx@qq.com', msg.as_string())
23. smtp.quit()
```

注意默认 `msg['From']` 和 `msg['To']` 都是邮箱格式。 `msg['From']` 可以和发件邮箱不一致，即使是不存在的邮箱，例如 `'yjc@test.com'` ，会显示是代发的。

如果想要将 `msg['From']` 改为中文的，需要编码：

```
# coding: utf-8
import smtplib
from email.mime.text import MIMEText
from email.header import Header
from email.utils import parseaddr, formataddr

def _format_addr(s):
    name, addr = parseaddr(s)
    return formataddr((Header(name, 'utf-8').encode(), addr))

# 连接邮件服务器
mail_host = 'smtp.163.com'
mail_port = '25'
mail_from = 'xxx@163.com'
smtp = smtplib.SMTP()
smtp.connect(mail_host, mail_port)
smtp.login(mail_from, 'xxx')

# 构造邮件对象
msg = MIMEText('Python测试', 'plain', 'utf-8')
msg['From'] = _format_addr('Python爱好者 <%s>' % mail_from)
msg['To'] = _format_addr('管理员 <%s>' % 'xxx@qq.com')
msg['Subject'] = Header('测试邮件', 'utf-8')

# 发送邮件
smtp.sendmail(mail_from, 'xxx@qq.com', msg.as_string())
smtp.quit()
```

发送HTML格式的邮件

Python发送HTML格式的邮件与发送纯文本消息的邮件不同之处就是将MIMEText中_subtype设置为html。

很简单，只需要把实例化邮件对象那句简单修改：

```
msg = MIMEText('<h1 style="color:red;">Python测试</h1>', 'html', 'utf-8')
```

再次发送就可以看到效果了。支持内联CSS。

发送附件

发送附件就需要把真个邮件当做复合型的内容：文本和各个附件本身。通过构造一个 `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，再继续往里面加上附件部分：

```
# coding: utf-8
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from email.header import Header

# 连接邮件服务器
mail_host = 'smtp.163.com'
mail_port = '25'
mail_from = 'xxx@163.com'
smtp = smtplib.SMTP()
smtp.connect(mail_host, mail_port)
smtp.login(mail_from, 'xxx')

# 构造邮件对象
msg = MIMEMultipart()
msg['From'] = mail_from
msg['To'] = 'xxx@qq.com'
msg['Subject'] = Header('测试邮件', 'utf-8')
msg.attach(MIMEText('Python测试', 'plain', 'utf-8')) #邮件正文

## 附件
att1 = MIMEText(open('test.txt', 'rb').read(), 'base64', 'utf-8')
att1["Content-Type"] = 'application/octet-stream' # 二进制流，不知道下载文件类型
att1["Content-Disposition"] = 'attachment; filename="test.txt"' # 这里的filename可以任意写，写什么名字，邮件中显示什么名字
msg.attach(att1)

att2 = MIMEText(open('test.jpg', 'rb').read(), 'base64', 'utf-8')
att2["Content-Type"] = 'image/jpeg'
att2["Content-Disposition"] = 'attachment; filename="test.jpg"'
msg.attach(att2)

smtp.sendmail(mail_from, 'xxx@qq.com', msg.as_string())
smtp.quit()
```

这里将二进制文件读入并转成base64编码，添加到邮件中。需要注意的是， `Content-Type` 指文件的Mime-Type，例如纯文本是 `text/plain` ，jpg是 `image/jpeg` ，如果不知道类型，则统称

为 `application/octet-stream` 。

最后需要注意的是，发送邮件部分最好使用 `try...except...` 语句：

```
try:
    smtp = smtplib.SMTP('localhost')
    smtp.sendmail(sender, receivers, msg.as_string())
    print "邮件发送成功"
except smtplib.SMTPException:
    print "Error: 无法发送邮件"
```

参考:
1、SMTP发送邮件 - 廖雪峰的官方网站
http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/001432005226355aadb8d4b2f3f42f6b1d6f2c5bd8d5263000
2、Python SMTP发送邮件 | 菜鸟教程
http://www.runoob.com/python/python-email.html
3、HTTP Content-type 对照表
http://tool.oschina.net/commons

Python学习—22 异步I/O

> 在同步IO中，线程启动一个IO操作然后就立即进入等待状态，直到IO操作完成后才醒来继续执行。而异步IO方式中，线程发送一个IO请求到内核，然后继续处理其他的事情，内核完成IO请求后，将会通知线程IO操作完成了。
>
> 如果IO请求需要大量时间执行的话，异步IO方式可以显著提高效率，因为在线程等待的这段时间内，CPU将会调度其他线程进行执行，如果没有其他线程需要执行的话，这段时间将会浪费掉。

协程

协程（Coroutine），又称微线程。

我们平常使用的函数又称子程序，是层级调用的，即A函数调用B，B函数又调用C，那么需要等C执行完毕返回，然后B程序执行完毕返回，最后A执行完毕。

协程看上去也是子程序，但是执行顺序和子程序不同：协程执行过程可以中断，同样是A函数调用B，但B可以执行一部分继续去执行A，然后继续执行B未执行完的部分。

协程看起来很像多线程。但协程最大的优势是极高的执行效率。因为线程需要互相切换，切换需要开销。且线程直接共享变量需要使用锁机制，因为协程只有一个线程，不存在同时写变量冲突。

Python对协程的支持是通过generator实现的。

在generator中，我们不但可以通过for循环来迭代，还可以不断调用 `next()` 函数获取由 `yield` 语句返回的下一个值。

但是Python的 `yield` 不但可以返回一个值，它还可以接收调用者发出的参数。下面是一个典型的 `生产者-消费者` 模型：

```
1.  # coding: utf-8
2.
3.  def consumer():
4.      r = ''
5.      while True:
6.          n = yield r
7.          print('[Consumer] Consuming %s' % n)
8.          r = '200 OK'
9.
10.
11. def produce(c):
12.     c.send(None) #启动生成器
```

```
    i = 0
    while i < 5:
        i = i + 1
        print('[Produce] Start produce %s' % i)
        r = c.send(i)
        print('[Produce] Consumer return  %s' % r)


c = consumer() #生成器
produce(c)
```

输出：

```
[Produce] Start produce 1
[Consumer] Consuming 1
[Produce] Consumer return  200 OK
[Produce] Start produce 2
[Consumer] Consuming 2
[Produce] Consumer return  200 OK
[Produce] Start produce 3
[Consumer] Consuming 3
[Produce] Consumer return  200 OK
[Produce] Start produce 4
[Consumer] Consuming 4
[Produce] Consumer return  200 OK
[Produce] Start produce 5
[Consumer] Consuming 5
[Produce] Consumer return  200 OK
```

> 执行顺序:
> 1、发送None启动生成器consumer()，运行yield r，返回''，程序中断;
> 2、第2次发送1，从上次运行结束的地方开始，先通过n = yield r接收到1，继续运行打印语句，再次运行到yield r，返回'200 OK';
> 3、第3次发送2，从上次运行结束的地方开始，先通过n = yield r接收到2，继续运行打印语句，再次运行到yield r，返回'200 OK';
> 4、…

大家可以使用 Intellij IDEA调试功能 进行单步运行观察执行流程。

整个流程由一个线程执行，produce和consumer协作完成任务，所以称为“协程”，而非线程的抢占式多任务。

asyncio

`asyncio` 是Python 3.4版本引入的标准库，直接内置了对异步IO的支持。使用 `asyncio` 可以实现单线程并发IO操作。

`@asyncio.coroutine` 把一个generator标记为coroutine类型：

```
# coding: utf-8
import asyncio

@asyncio.coroutine
def helloWorld(n):
    print('Hello world! %s' % n)
    r = yield from asyncio.sleep(3)
    print('Hello %s %s ' % (r, n))

loop = asyncio.get_event_loop()
tasks = [helloWorld(1), helloWorld(2), helloWorld(3), helloWorld(4)]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

输出：

```
Hello world! 2
Hello world! 3
Hello world! 1
Hello world! 4

#(等待3秒左右)

Hello None 2
Hello None 1
Hello None 3
Hello None 4
```

程序先运行 `Hello world!` ，然后由于 `asyncio.sleep()` 也是一个coroutine，线程不会等待 `asyncio.sleep()` ，而是直接中断并执行下一个消息循环。当 `asyncio.sleep()` 返回时，线程就可以从 `yield from` 拿到返回值（此处是 `None` ），然后接着执行下一行语句。

> `asyncio` 的编程模型就是一个消息循环。我们从 `asyncio` 模块中直接获取一个 `EventLoop` 的引用，然后把需要执行的协程扔到 `EventLoop` 中执行，就实现了异步IO。

async/await

用asyncio提供的 `@asyncio.coroutine` 可以把一个 `generator` 标记为 `coroutine` 类型，然后在 `coroutine` 内部用 `yield from` 调用另一个 `coroutine` 实现异步操作。

为了简化并更好地标识异步IO，从Python 3.5开始引入了新的语法 `async` 和 `await` ，可以让 `coroutine` 的代码更简洁易读。

请注意， `async` 和 `await` 是针对 `coroutine` 的新语法，要使用新的语法，只需要做两步简单的替换：

1. 把 `@asyncio.coroutine` 替换为 `async` ；
2. 把 `yield from` 替换为 `await` 。

上节的代码用Python3.5写：

```
# coding: utf-8
import asyncio

async def helloWorld(n):
    print('Hello world! %s' % n)
    r = await asyncio.sleep(3)
    print('Hello %s %s ' % (r, n))

loop = asyncio.get_event_loop()
tasks = [helloWorld(1), helloWorld(2), helloWorld(3), helloWorld(4)]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

aiohttp

`aiohttp` 是基于 `asyncio` 实现的HTTP框架。

需要先安装：

```
$ pip install aiohttp
```

控制台输出：

```
Collecting aiohttp
  Downloading aiohttp-1.3.1-cp34-cp34m-win32.whl (147kB)
    100% |████████████████████████████████| 153kB 820kB/s
```

```
Collecting async-timeout>=1.1.0 (from aiohttp)
  Downloading async_timeout-1.1.0-py3-none-any.whl
Collecting yarl>=0.8.1 (from aiohttp)
  Downloading yarl-0.9.6-cp34-cp34m-win32.whl (77kB)
    100% |████████████████████████████████| 81kB 1.8MB/s
Collecting chardet (from aiohttp)
  Downloading chardet-2.3.0-py2.py3-none-any.whl (180kB)
    100% |████████████████████████████████| 184kB 890kB/s
Collecting multidict>=2.1.4 (from aiohttp)
  Downloading multidict-2.1.4-cp34-cp34m-win32.whl (133kB)
    100% |████████████████████████████████| 143kB 3.3MB/s
Installing collected packages: async-timeout, multidict, yarl, chardet, aiohttp
Successfully installed aiohttp-1.3.1 async-timeout-1.1.0 chardet-2.3.0
multidict-2.1.4 yarl-0.9.6
```

说明安装完成。

示例：

```
# coding: utf-8
import asyncio
from aiohttp import web

@asyncio.coroutine
def index(request):
    return web.Response(body=b'Hello aiohttp!', content_type='text/html')

@asyncio.coroutine
def user(request):
    name = request.match_info['name']
    body = 'Hello %s' % name
    return web.Response(body=body.encode('utf-8'), content_type='text/html')
    pass

@asyncio.coroutine
def init(loop):
    # 创建app
    app = web.Application(loop=loop)

    #添加路由
    app.router.add_route('GET', '/', index)
    app.router.add_route('GET', '/user/{name}', user)

```

```
25.     #运行server
        server = yield from loop.create_server(app.make_handler(), '127.0.0.1',
26. 9999)
27.     print('Server is running at http://127.0.0.1:9999 ...')
28. 
29. loop = asyncio.get_event_loop()
30. loop.run_until_complete(init(loop))
31. loop.run_forever()
```

运行程序:

```
1. $ python user_aiohttp.py
2. 
3. Server is running at http://127.0.0.1:9999 ...
```

浏览器依次输入查看效果:

http://127.0.0.1:9999/

http://127.0.0.1:9999/user/aiohttp

更多知识可以查看aiohttp文档:

http://aiohttp.readthedocs.io/en/stable/

> 参考:
> 1、Python asyncio库的学习和使用
> http://www.cnblogs.com/rockwall/p/5750900.html
> 2、异步IO
> http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/0014320857348055808fa77514407cb23834c78c6c7309000
>
> 作者: 飞鸿影
>
> 出处: https://www.cnblogs.com/52fhy/p/6407121.html

Python学习—23 内建模块及第三方库

本文将介绍python里常用的模块。如未特殊说明，所有示例均以python3.4为例：

```
$ python -V
Python 3.4.3
```

网络请求

urllib

urllib提供了一系列用于操作URL的功能。通过urllib我们可以很方便的抓取网页内容。

抓取网页内容

```
# coding: utf-8

import urllib.request

url = 'https://api.douban.com/v2/book/2129650'

with urllib.request.urlopen(url) as f:
    headers = f.getheaders() # 报文头部
    body = f.read() # 报文内容

    print(f.status, f.reason) # 打印状态码、原因语句
    for k,v in headers:
        print(k + ': ' + v)

    print(body.decode('utf-8'))
```

抓取百度搜索图片

```
import urllib.request
import os
import re
import time
```

```
    url=r'http://image.baidu.com/search/index?
    tn=baiduimage&ipn=r&ct=201326592&cl=2&lm=-1&st=-1&fm=result&fr=&sf=1&fmq=14887223
5.  8&word=%E5%A3%81%E7%BA%B8%E5%B0%8F%E6%B8%85%E6%96%B0&f=3&oq=bizhi%E5%B0%8F%E6%B8%
6.
7.  imgPath=r'E:\img'
8.
9.  if not os.path.isdir(imgPath):
10.     os.mkdir(imgPath)
11.
12. imgHtml=urllib.request.urlopen(url).read().decode('utf-8')
13. #test html
14. #print(imgHtml)
15. urls=re.findall(r'"objURL":"(.*?)"',imgHtml)
16.
17. index=1
18. for url in urls:
19.     print("下载:",url)
20.
21.     #未能正确获得网页 就进行异常处理
22.     try:
23.         res=urllib.request.urlopen(url)
24.
25.         if str(res.status)!='200':
26.             print('未下载成功：',url)
27.             continue
28.     except Exception as e:
29.         print('未下载成功：',url)
30.
31.     filename=os.path.join(imgPath,str(time.time()) + '_' + str(index)+'.jpg')
32.     with open(filename,'wb') as f:
33.         f.write(res.read())
34.         print('下载完成\n')
35.         index+=1
36. print("下载结束，一共下载了 %s 张图片"% (index-1))
```

python2.7的用户需要把 `urllib.request` 替换成 `urllib` 。

批量下载图片

```
1.  # coding: utf-8
2.  import os,urllib.request
3.
```

```
url_path = 'http://www.ruanyifeng.com/images_pub/'

imgPath=r'E:\img'
if not os.path.isdir(imgPath):
    os.mkdir(imgPath)

index=1
for i in range(1,355):
    url = url_path + 'pub_' + str(i) + '.jpg'
    print("下载:",url)

    try:
        res = urllib.request.urlopen(url)
        if(str(res.status) != '200'):
            print("下载失败:", url)
            continue
    except:
        print('未下载成功：',url)

    filename=os.path.join(imgPath,str(i)+'.jpg')
    with open(filename,'wb') as f:
        f.write(res.read())
        print('下载完成\n')
        index+=1
print("下载结束，一共下载了 %s 张图片"% (index-1))
```

模拟GET请求附带头信息

`urllib.request.Request` 实例化后有个 `add_header()` 方法可以添加头信息。

```
# coding: utf-8
import urllib.request

url = 'http://www.douban.com/'

req = urllib.request.Request(url)
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 urllib.request.urlopen(req) as f:
    headers = f.getheaders()
```

```
    body = f.read()

    print(f.status, f.reason)
    for k,v in headers:
        print(k + ': ' + v)

    print(body.decode('utf-8'))
```

这样会返回适合iPhone的移动版网页。

发送POST请求

`urllib.request.urlopen()` 第二个参数可以传入需要post的数据。

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import json
from urllib import request
from urllib.parse import urlencode

#---------------------------------
# 手机号码归属地调用示例代码 - 聚合数据
# 在线接口文档：http://www.juhe.cn/docs/11
#---------------------------------

def main():

    #配置您申请的APPKey
    appkey = "**********************"

    #1.手机归属地查询
    request1(appkey,"GET")

#手机归属地查询
def request1(appkey, m="GET"):
    url = "http://apis.juhe.cn/mobile/get"
    params = {
        "phone" : "", #需要查询的手机号码或手机号码前7位
        "key" : appkey, #应用APPKEY(应用详细页查询)
        "dtype" : "", #返回数据的格式,xml或json, 默认json

    }
```

```
    params = urlencode(params).encode('utf-8')

    if m =="GET":
        f = request.urlopen("%s?%s" % (url, params))
    else:
        f = request.urlopen(url, params)

    content = f.read()
    res = json.loads(content.decode('utf-8'))
    if res:
        error_code = res["error_code"]
        if error_code == 0:
            #成功请求
            print(res["result"])
        else:
            print("%s:%s" % (res["error_code"],res["reason"]) )
    else:
        print("request api error")

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

Requests

虽然Python的标准库中 `urllib2` 模块已经包含了平常我们使用的大多数功能，但是它的API使用起来让人实在感觉不好。它已经不适合现在的时代，不适合现代的互联网了。而 `Requests` 的诞生让我们有了更好的选择。

正像它的名称所说的， `HTTP for Humans` ,给人类使用的HTTP库！在Python的世界中，一切都应该简单。 `Requests` 使用的是 `urllib3` ，拥有了它的所有特性， `Requests` 支持 HTTP 连接保持和连接池，支持使用 cookie 保持会话，支持文件上传，支持自动确定响应内容的编码，支持国际化的 URL 和 POST 数据自动编码。现代、国际化、人性化。

官网: http://python-requests.org/
文档: http://cn.python-requests.org/zh_CN/latest/
Github主页: https://github.com/kennethreitz/requests

需要先安装:

```
$ pip3 install requests
Collecting requests
  Downloading requests-2.13.0-py2.py3-none-any.whl (584kB)
```

```
    100% |████████████████████████████████| 593kB 455kB/s
Installing collected packages: requests
Successfully installed requests-2.13.0
```

请求示例

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import requests

url = 'https://api.github.com/user'
# r = requests.request('get', url, auth=('52fhy', ''))
r = requests.get(url, auth=('', ''))
print('Status: %s' % r.status_code) # 状态码
# 头信息
for k,v in r.headers.items():
    print(k + ': ' + v)

print('encoding: ' , r.encoding)
print('body: ' , r.text)
print('json body: ' , r.json())
```

> 默认情况下，dict迭代的是key。如果要迭代value，可以用 `for value in d.values()` ，如果要同时迭代key和value，可以用 `for k, v in d.items()` 。

POST请求

基于表单的:

```
# coding: utf-8
import requests

payload = {'name': 'python', 'age': '11'}
r = requests.post("http://httpbin.org/post", data=payload)
print(r.text)
```

基于text的:

```
# coding: utf-8
import requests,json

```

```
payload = {'name': 'python', 'age': '11'}
r = requests.post("https://api.github.com/some/endpoint",
data=json.dumps(payload))
print(r.text)
```

还可以使用 `json` 参数直接传递，然后它就会被自动编码。这是 `2.4.2` 版的新加功能：

```
r = requests.post("https://api.github.com/some/endpoint", json=payload)
```

hashlib

md5

```
import hashlib

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

结果如下：

```
d26a53750bc40b38b65a520292f69306
```

`update()` ，用于将内容分块进行处理，适用于大文件的情况。示例：

```
import hashlib

def get_file_md5(f):
    m = hashlib.md5()

    while True:
        data = f.read(10240)
        if not data:
            break

        m.update(data)
    return m.hexdigest()


with open(YOUR_FILE, 'rb') as f:
```

```
16.     file_md5 = get_file_md5(f)
```

对于普通字符串的md5，可以封装成函数：

```
1.  def md5(string):
2.      import hashlib
3.      return hashlib.md5(string.encode('utf-8')).hexdigest()
```

SHA1

```
1.  import hashlib
2.
3.  sha1 = hashlib.sha1()
4.  sha1.update('py'.encode('utf-8'))
5.  sha1.update('thon'.encode('utf-8'))
6.  print(sha1.hexdigest())
```

等效于：

```
1.  hashlib.sha1('python'.encode('utf-8')).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。

此外，hashlib还支持 `sha224` ， `sha256` ， `sha384` ， `sha512` 。

base64

Base64是一种用64个字符来表示任意二进制数据的方法。Python内置的base64可以直接进行base64的编解码：

```
1.  >>> import base64
2.  >>> base64.b64encode(b'123')
3.  b'MTIz'
4.  >>> base64.b64decode(b'MTIz')
5.  b'123'
```

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 `_` ：

```
1.  >>> base64.b64encode(b'i\xb7\x1d\xfb\xef\xff')
```

```
b'abcd++//'
>>> base64.urlsafe_b64encode(b'i\xb7\x1d\xfb\xef\xff')
b'abcd--__'
>>> base64.urlsafe_b64decode('abcd--__')
b'i\xb7\x1d\xfb\xef\xff'
```

时间日期

该部分在前面的笔记里已做详细介绍：http://www.cnblogs.com/52fhy/p/6372194.html。本节仅作简单回顾。

time

```python
# coding:utf-8
import time

# 获取时间戳
timestamp = time.time()
print(timestamp)

# 格式时间
print(time.strftime('%Y-%m-%d %H:%M:%S', time.localtime()))

# 返回当地时间下的时间元组t
print(time.localtime())

# 将时间元组转换为时间戳
print(time.mktime(time.localtime()))
t = (2017, 2, 11, 15, 3, 38, 1, 48, 0)
print(time.mktime(t))

# 字符串转时间元组：注意时间字符串与格式化字符串位置一一对应
print(time.strptime('2017 02 11', '%Y %m %d'))

# 睡眠
print('sleeping...')
time.sleep(2) # 睡眠2s
print('sleeping end.')
```

输出：

```
1486797515.78742

2017-02-11 15:18:35

time.struct_time(tm_year=2017, tm_mon=2, tm_mday=11, tm_hour=15, tm_min=18,
tm_sec=35, tm_wday=5, tm_yday=42, tm_isdst=0)

1486797515.0
1486796618.0

time.struct_time(tm_year=2017, tm_mon=2, tm_mday=11, tm_hour=0, tm_min=0,
tm_sec=0, tm_wday=5, tm_yday=42, tm_isdst=-1)

sleeping...
sleeping end.
```

datetime

方法概览:

```
datetime.now() # 当前时间, datetime类型
datetime.timestamp() # 时间戳, 浮点类型
datetime.strftime('%Y-%m-%d %H:%M:%S') # 格式化日期对象datetime, 字符串类型
datetime.strptime('2017-2-6 23:22:13', '%Y-%m-%d %H:%M:%S') # 字符串转日期对象
datetime.fromtimestamp(ts) # 获取本地时间, datetime类型
datetime.utcfromtimestamp(ts) # 获取UTC时间, datetime类型
```

示例:

```
# coding: utf-8

from datetime import datetime
import time

now = datetime.now()
print(now)

# datetime模块提供
print(now.timestamp())
```

输出:

```
2017-02-06 23:26:54.631582
1486394814.631582
```

小数位表示毫秒数。

图片处理

PIL

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。

安装:

```
$ pip install Pillow

Collecting Pillow
  Downloading Pillow-4.0.0-cp34-cp34m-win32.whl (1.2MB)
Successfully installed Pillow-4.0.0
```

图像缩放:

```
# coding: utf-8
from PIL import Image
im = Image.open('test.jpg')
print(im.format, im.size, im.mode)
im.thumbnail((200, 100))
im.save('thumb.jpg', 'JPEG')
```

模糊效果:

```
# coding: utf-8
from PIL import Image,ImageFilter
im = Image.open('test.jpg')
im2 = im.filter(ImageFilter.BLUR)
im2.save('blur.jpg', 'jpeg')
```

验证码:

```
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')
```

注意示例里的字体文件必须是绝对路径。

下载

you-get

安卓you-get

```
pin install you-get
```

需要有ffmpeg的支持。windows下载地址：
https://ffmpeg.zeranoe.com/builds/win64/static/ffmpeg-3.4.1-win64-static.zip
解压后添加环境变量，例如：

```
C:\Program Files\ffmpeg-3.4.1-win64-static\bin
```

然后就可以下载各大视频网站的视频了。示例：

```
you-get v.youku.com/v_show/id_XNTA5MDQ1OTM2.html
```

如果提示201:客户端未授权，可以安装Adobe Flash Player 28 PPAPI试试。

如果你在使用Mac，可以下载客户端：
https://github.com/coslyk/moonplayer

> 参考：
> 1、Python资源
> http://hao.jobbole.com/?catid=144
>
> 作者： 飞鸿影
> 版权：本作品采用「署名-非商业性使用-相同方式共享 4.0 国际」许可协议进行许可。
> 出处：https://www.cnblogs.com/52fhy/p/6507378.html