python3学习

1. os 模块
   1. os 模块提供了与操作系统相关的功能。你可以使用如下语句导入它：
   2. getuid() 函数返回当前进程的有效用户 id。
   3. getpid() 函数返回当前进程的 id。getppid() 返回父进程的 id。
   4. uname() 函数返回识别操作系统的不同信息，在 Linux 中它返回的详细信息可以从 uname -a 命令得到。uname() 返回的对象是一个元组，（sysname, nodename, release, version, machine）。
   5. getcwd() 函数返回当前工作目录。chdir(path) 则是更改当前目录到 path。在例子中我们首先看到当前工作目录是 /home/shiyanlou，然后我们更改当前工作目录到 /Code 并再一次查看当前工作目录。

>>> os.getcwd()

'/home/shiyanlou'

>>> os.chdir('Code')

>>> os.getcwd()

* 1. '/home/shiyanlou/Code'
  2. os.listdir（path）以列表形式给出path下的所有文件和目录。

1. Requests 模块。是一个第三方 Python 模块。
   1. 首先要安装 pip3：

$ sudo apt-get update

$ sudo apt-get install python3-pip

然后用 pip3 安装 requests

$ sudo pip3 install requests

上面的命令会在你的系统中安装 Python3 版本的 Requests 模块。

* 1. get()获取任意一个网页。

>>> import requests

>>> req = requests.get('https://github.com')

>>> req.status\_code

200

* 1. req.text 属性存有服务器返回的 HTML 网页，由于 HTML 文本太长就不在这里贴出来了。

1. if \_\_name\_\_ == '\_\_main\_\_':它的作用是，只有在当前模块名为 \_\_main\_\_ 的时候（即作为脚本执行的时候）才会执行此 if 块内的语句。换句话说，当此文件以模块的形式导入到其它文件中时，if 块内的语句并不会执行。

list:可变可重复有序表，可存储可变量。

tuple:不可变可重复有序表，可存储可变量。

dict: 可变不可重复无序表，key存储不可变量

set: 可变不可重复无序表，key存储不可变量

int 是不可迭代对象

1. 编码风格。

4.1在不同优先级之间，考虑在更低优先级的操作符两侧插入空格。用你自己的判断力；但不要使用超过一个空格，并且在二元操作符的两侧有相同的空格数。

好的：

i = i + 1

submitted += 1

x = x\*2 - 1

hypot2 = x\*x + y\*y

c = (a+b) \* (a-b)

不好的：

i=i+1

submitted +=1

x = x \* 2 - 1

hypot2 = x \* x + y \* y

c = (a + b) \* (a - b)

4.2不要在关键值参数或默认值参数的等号两边加入空格。

好的：

def complex(real, imag=0.0):

return magic(r=real, i=imag)

不好的：

def complex(real, imag = 0.0):

return magic(r = real, i = imag)

【注：Python 3】带注释的函数定义中的等号两侧要各插入空格。此外，在冒号后用一个单独的空格，也要在表明函数返回值类型的->左右各插入一个空格。

好的：

def munge(input: AnyStr):

def munge(sep: AnyStr = None):

def munge() -> AnyStr:

def munge(input: AnyStr, sep: AnyStr = None, limit=1000):

不好的：

def munge(input: AnyStr=None):

def munge(input:AnyStr):

def munge(input: AnyStr)->PosInt:

4.3个行内注释与语句在同一行。行内注释应该至少与语句相隔两个空格。以#打头，#后接一个空格。

4.5文档字符串的结尾"""应该放在单独的一行，例如：

"""Return a foobang

Optional plotz says to frobnicate the bizbaz first.

"""

4.6模块名应该短，且全小写。如果能改善可读性，可以使用下划线。

4.7 Python 的包名也应该短，全部小写，但是不推荐使用下划线。

4.8 类名通常使用 CapWords 约定。

4.9因为异常应该是类，所以类名约定在这里适用。但是，你应该用Error作为你的异常名的后缀（异常实际上是一个错误）。

4.10函数名应该是小写的，有必要的话用下划线来分隔单词提高可读性。

4.11常量通常是模块级的定义，全部大写，单词之间以下划线分隔。例如MAX\_OVERFLOW和TOTAL

4.12像None应该用is或is not，从不使用==操作符。

4..13坚持使用return语句。函数内的return语句都应该返回一个表达式，或者None。如果一个return语句返回一个表达式，另一个没有返回值的应该用return None清晰的说明，并且在一个函数的结尾应该明确使用一个return语句（如果有返回值的话）。

4.14用''.startswith()和''.endswith()代替字符串切片来检查前缀和后缀。

startswith()和endswith()是更简洁的，不容易出错的。例如：

Yes: if foo.startswith('bar'):

No: if foo[:3] == 'bar':

4.15对象类型的比较应该始终使用isinstance()而不是直接比较。

Yes: if isinstance(obj, int):

No: if type(obj) is type(1):

4.16不要用==比较True和False。

Yes: if greeting:

No: if greeting == True:

Worse: if greeting is True:

5. 表达式和语句中的空格

5.1与括号保持紧凑（小括号、中括号、大括号）：

Yes: spam(ham[1], {eggs: 2})

No: spam( ham[ 1 ], { eggs: 2 } )

5.2与后面的逗号、分号或冒号保持紧凑：

Yes: if x == 4: print x, y; x, y = y, x

No: if x == 4 : print x , y ; x , y = y , x

5.3切片内的冒号就像二元操作符一样，任意一侧应该被等同对待（把它当做一个极低优先级的操作）。在一个可扩展的切片中，冒号两侧必须有相同的空格数量。例外：切片参数省略时，空格也省略。

好的：

ham[1:9], ham[1:9:3], ham[:9:3], ham[1::3], ham[1:9:]

ham[lower:upper], ham[lower:upper:], ham[lower::step]

ham[lower+offset : upper+offset]

ham[: upper\_fn(x) : step\_fn(x)], ham[:: step\_fn(x)]

ham[lower + offset : upper + offset]

不好的：

ham[lower + offset:upper + offset]

ham[1: 9], ham[1 :9], ham[1:9 :3]

ham[lower : : upper]

ham[ : upper]

5.4函数名与其后参数列表的左括号应该保持紧凑：

Yes: spam(1)

No: spam (1)

5.5与切片或索引的左括号保持紧凑：

Yes: dct['key'] = lst[index]

No: dct ['key'] = lst [index]

5.6在赋值操作符（或其它）的两侧保持多余一个的空格：

好的：

x = 1

y = 2

long\_variable = 3

不好的：

x = 1

y = 2

long\_variable = 3

5.7总是在这些二元操作符的两侧加入一个空格：赋值(=)，增量赋值(+=, -= etc.)，比较(==, <, >, !=, <>, <=, >=, in, not in, is, is not)，布尔运算(and, or, not)。

5.8在不同优先级之间，考虑在更低优先级的操作符两侧插入空格。用你自己的判断力；但不要使用超过一个空格，并且在二元操作符的两侧有相同的空格数。

好的：

i = i + 1

submitted += 1

x = x\*2 - 1

hypot2 = x\*x + y\*y

c = (a+b) \* (a-b)

不好的：

i=i+1

submitted +=1

x = x \* 2 - 1

hypot2 = x \* x + y \* y

c = (a + b) \* (a - b)

5.9不要在关键值参数或默认值参数的等号两边加入空格。

好的：

def complex(real, imag=0.0):

return magic(r=real, i=imag)

不好的：

def complex(real, imag = 0.0):

return magic(r = real, i = imag)

6.生成器，是更简单的创建迭代器的方法，这通过在函数中使用 yield 关键字完成：

6.1def counter\_generator(low, high):

while low <= high:

yield low

low += 1

>>> for i in counter\_generator(5,10):

... print(i, end=' ')

...

5 6 7 8 9 10

在 While 循环中，每当执行到 yield 语句时，返回变量 low 的值并且生成器状态转为挂起。在下一次调用生成器时，生成器从之前冻结的地方恢复执行然后变量 low 的值增一。生成器继续 while 循环并且再次来到 yield 语句...

当你调用生成器函数时它返回一个生成器对象。如果你把这个对象传入 dir() 函数，你会在返回的结果中找到 \_\_iter\_\_ 和 \_\_next\_\_ 两个方法名。

我们通常使用生成器进行惰性求值。这样使用生成器是处理大数据的好方法。如果你不想在内存中加载所有数据，你可以使用生成器，一次只传递给你一部分数据。

6.2如果我们回到 my\_generator() 这个例子，我们会发现生成器的一个特点：它们是不可重复使用的。

>>> def my\_generator():

... print("Inside my generator")

... yield 'a'

... yield 'b'

... yield 'c'

>>>g = my\_generator()

>>> for c in g:

... print(c)

...

Inside my generator

a

b

c

>>> for c in g:

... print(c)

...

这个时候没有输出了，因为对于g来说已经输出过一次了，不可以重复。

一个创建可重复使用生成器的方式是不保存任何状态的基于对象的生成器。任何一个生成数据的含有 \_\_iter\_\_ 方法的类都可以用作对象生成器。在下面的例子中我们重新创建了 counter 生成器。

>>> class Counter(object):

... def \_\_init\_\_(self, low, high):

... self.low = low

... self.high = high

... def \_\_iter\_\_(self):

... counter = self.low #为什么要赋值给counter，这是关键

... while self.high >= counter:

... yield counter

... counter += 1

...

>>> gobj = Counter(5, 10)

>>> for num in gobj:

... print(num, end=' ')

...

5 6 7 8 9 10

>>> for num in gobj:

... print(num, end=' ')

...

5 6 7 8 9 10

这里可重复迭代输出的原因是self.low没有发生变化，一直都是5，因此可以迭代。之前的例子不可以重复的根本就在于代码迭代到了’c’,无法回到’a’再重新迭代。

但是上面的 gobj 并不是生成器或迭代器，因为它不具有 \_\_next\_\_ 方法，只是一个可迭代对象，生成器是一定不能重复循环的。

6.3生成器表达式

在这一节我们学习生成器表达式（Generator expressions），生成器表达式是列表推导式和生成器的一个高性能，内存使用效率高的推广。

生成器表达式的语法要求其总是直接在在一对括号内，并且不能在两边有逗号。

7.函数参数。

7.1 参数类型包括形参和实参。其中实参包括位置参数和关键字参数。

7.2位置参数，包括常规参数，和以\*开头的参数（表示参数个数不定,打印出来以元祖的形式呈现）。如：

def foo(a,\*b)

print(a,b)

>>>foo（1,2）

1,(2,)

>>>foo(1,2,3)

1 (2,3)

7.3 关键字参数，包括默认参数和以\*\*开头的参数（参数个数不定，以字典形式呈现）。如：

def foo(a=1,\*\*b):

print(a,b)

>>>foo1(name=’Tom’)

1 {‘name’:’Tom’}

>>>foo1(2,d=3,e=5)

2 {‘d’:3,’e’:5}