python 入门-introduction ¶

什么是python

python是一种动态的高级语言，有一些特性可能初学的时候没有感觉，但是如果和其他的语言（比如C）相比会有明显的区别：

- 不用声明类型，不管是变量，对象还是函数
- 语言弹性好，接近自然语言
- 没有{}，严格缩进

python可以从官方网站下载： python3 windows (https://www.python.org/downloads/release/python-364/) python2.7 windows (https://www.python.org/downloads/release/python-2714/)

python下载安装以后是自带编辑器/解释器的，也就是说你不用另外准备编程环境直接就可以开始工作了。打开一个叫做idle的程序，或者也可以在你的命令行模式下面输入python，接着就可以开始输入命令了：

但是，如果你愿意再多花一些时间来设置编程环境，我很推荐你一步到位使用anaconda。anaconda是数据科学中非常常用的一个python环境大礼包，里面包括了jupyter，numpy，pandas等等很有用的工具，当然也包括了python。其他的不说，至少jupyter对于学习python是很有帮助的。

anaconda的下载地址在这里：

anaconda清华大学镜像说明 (https://mirrors.tuna.tsinghua.edu.cn/help/anaconda/)

anaconda清华大学镜像 (https://mirrors.tuna.tsinghua.edu.cn/anaconda/archive/)

anaconda3 17年10月更新，windows 64位 (https://mirrors.tuna.tsinghua.edu.cn/anaconda/archive/Anaconda3-5.0.1-Windows-x86_64.exe)

anaconda3 17年10月更新，windows 32位 (https://mirrors.tuna.tsinghua.edu.cn/anaconda/archive/Anaconda3-5.0.1-Windows-x86.exe)

这些都是国内的源，下载速度应该是ok的。实际上下载完以后基本都够你用的了，但是如果你想进一步做一些生信的分析，可能还要装别的包，这时候你就需要配置一下国内的channel保证下载速度：

在命令行输入下列命令：

```
conda config --add channels https://mirrors.tuna.tsinghua.edu.
cn/anaconda/pkgs/main/
conda config --add channels https://mirrors.tuna.tsinghua.edu.
cn/anaconda/pkgs/free/
conda config --add channels https://mirrors.tuna.tsinghua.edu.
cn/anaconda/cloud/bioconda/
conda config --set show_channel_urls yes
```

jupyter是很好用的一个工具，可以在命令行里面用`jupyter notebook`调用。

符号说明：

命令行提示符，表示后面的内容要由你输入到命令行里面，本身不用输入

表示后面的内容是注释，也就是不被运行的程序

$ python

```
Python 3.5.2 |Anaconda custom (x86_64)| (default, Jul  2 2016,
17:52:12)
[GCC 4.2.1 Compatible Apple LLVM 4.2 (clang-425.0.28)] on darw
in
Type "help", "copyright", "credits" or "license" for more info
rmation.
```

根据你安装的python平台有所不同，上面的返回结果会有所不同，这个不要紧

In [13]:
```
a = 6
a
```

Out[13]: 6

上面的程序先用a=6做了一个**赋值**。=的意义是把右边的值交给左边的符号保存。在将6交给a保存以后，a返回的值当然就是6了

In [14]:
```
a+2
```

Out[14]: 8

上面的表达式a+2表示取a的值，和2相加，返回的结果就是8。

> 现在a的值是多少?

In [15]:
```
a = 'hi'
a
```

Out[15]: 'hi'

注意上面的程序，刚刚a还是代表一个数值6，现在呢？根据a返回的结果，a表示的是'hi'这个**字符串**

> 关于字符串我们晚一些会仔细的讲

In [16]:
```
len(a)
```

Out[16]: 2

len()是python自带的一个**函数**。晚一些我们还会提到另外一个东西叫做**方法**。方法和函数在很多情况下是差不多的意思，我也会互换着说。 len()这个函数会返回*参数*的*长度*。参数是括号里面的东西，在len(a)里面，a作为参数交给了len()，len会**返回**a的长度。现在a是一个字符串，字符串的长度就是里面字符的个数，所以是2。

> 为什么len()可以返回字符串的长度呢？在现在我们可以不用管这个细节的问题。高级程序语言最大的好处就是把很多细节的东西隐藏起来，而是提供给我们一些常用的接口，这样你在编程的时候只需要关注你想要实现的功能，而不需要对底层的技术细节伤脑筋。

In [18]:
```
a + len(a)
```

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-18-994567101b6f> in <module>()
----> 1 a + len(a)

TypeError: Can't convert 'int' object to str implicitly
```

好的，现在我们遇到了一个出错信息。在你编程的时候会经常遇到这样的信息，开始你看到这样的错误信息的时候会很不知所措，不过见多了就见惯不怪了。 我们来看一下上面的错误信息，现阶段需要你去关注的东西并不多，实际上你只要看到这一段就ok了：

```
----> 1 a + len(a)
```

这个说明出问题的程序是这一行（其中1是行号，后面的a+len(a)是这行的代码)。再往下还有这一行程序出错的详细说明，这些说明对于查错（Debug）有帮助，但不会总是明确指出错误到底出在哪里。

所以上面的程序到底哪里有问题呢？len(a)是2，a是'hi'，两个东西一个是数值，另外一个是字符串，是没办法做加法的。所以如果我想获得一个hi2这样的结果要怎么办呢？

In [19]:
```
a+str(len(a))
```

Out[19]:
```
'hi2'
```

str()又是一个内置的函数，他会把参数转化成字符串。也就是2==>'2'。现在就可以用+作运算了

> 字符串的+表示把两个字符串连在一起

In [20]:
```
foo
```

```
---------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-20-d3b07384d113> in <module>()
----> 1 foo

NameError: name 'foo' is not defined
```

随着你上网查代码，你会遇到很多foo,bar这样的伪代码常用的名称。上面的代码里面我们引用了一个不存在的变量，所以程序会报错

从上面的代码可以看到，python里面想对代码进行测试是很容易的。另外我推荐大家在容易找到的地方建立一个文件夹作为工作文件夹，使用英文命名，放在d盘根目录里面。把自己的代码放在这个文件夹里面。这样有几个好处：

- 代码可以逐渐积累
- 涉及文件操作的时候，可以把相关文件直接拷贝进工作文件夹里面，这样就不用写很长串的绝对路径了
- 因为是英文文件夹名称，又放在d盘根目录下面，不会由于中文路径名称出现兼容性问题

看一段python代码

刚才我们都是一行一行的输入命令，获得反馈。实际编程的时候，我们很多时候是在一个文本文档里面把程序全部写好然后才开始运行的。就像文本文档的扩展名是txt，python源代码也有自己的扩展名：**py**

看一下下面的代码：

In [21]:
```
#!/usr/bin/env python

# import modules used here -- sys is a very standard one
import sys

# Gather our code in a main() function
def main():
    print ('Hello there', sys.argv[1])
    # Command line args are in sys.argv[1], sys.argv[2] ...
    # sys.argv[0] is the script name itself and can be ignored

# Standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()
```

```
Hello there -f
```

因为我是在jupyter里面运行的上面的代码，所以sys.argv[1]返回的结果有点怪，这不重要。你可以自己试试把上面的代码存成一个hello.py放到工作文件夹里面，然后打开命令行，cd到你的工作文件夹，输入：

```
$ python hello.py Guido
Hello there Guido
$ ./hello.py Alice
Hello there Alice
```

有时候第二种方式不见得能行（需要把executable属性打开），但是第一种方式`python hello.py Guido`是没有问题的。

在上面这个命令里面，python是python这个解释器，hello.py是你刚才写的代码的文件名，而Guido则是交给这个程序的一个参数，而在程序里面，我们可以通过sys.argv获取参数**列表**。

因为sys不是python默认加载的模块，所以在程序的最开头我们要import一下sys

关于程序的结构

上面的程序除了有很多的注释(#开头的那些行)，还有个特点：实际干活的的东西是放在main()这个**函数**里面的。

函数的调用格式很简单：

```
def 函数名():
    函数内容
```

前面说过，python对于缩进的要求非常严格。因为python没有{}这样的符号表示代码块，所以代码块完全是按照缩进来判断层级的。现代的代码文本编辑器基本上都可以自动帮你缩进，所以一般不用你手动进行，但是脑子里面要有这个意识：代码缩进对不对?

In [22]:
```
def main():
    print ('Hello there', sys.argv[1])
```

可以看到上面的代码直接运行的话是不会有任何返回结果的。这是因为你还没有作任何的调用。一个函数在定义完以后要调用才行

In [23]:
```
main()
```

```
Hello there -f
```

那么在hello.py里面，main()的调用前面还有一个条件：

```
if __name__ == '__main__':
```

ok, **name**又是什么鬼，'**main**'又是什么鬼？其实先不用管这么多，这是很常规的一种判断形势，如果你是直接运行这个代码，比如通过`python hello.py alice`，那么就是符合这个条件的需求的，但是，如果你是通过另外一个python代码调用了这个hello.py文件，那么这个条件就不会被满足，main()也就不会被调用了

重要的事情说三遍!!! 定义一个你自己的函数

前面大概提了一下怎么定义函数，下面这个就是一个自定义函数，看看代码先：

In [24]:
```
# Defines a "repeat" function that takes 2 arguments.
def repeat(s, exclaim):
    """
    把s重复三遍返回
    如果exclaim是True的话，后面还要加上!!!
    """

    result = s + s + s # can also use "s * 3" which is faster (Why?)
    if exclaim:
        result = result + '!!!'
    return result
```

到现在我们可以看到代码里面都有很多的#。这个是表示注释的符号，写注释是一个很好的习惯，就算你现在再怎么确定你是知道这段程序是干嘛的，但是也应该写一下注释，毕竟3个月以后再来看代码的你会感谢现在写注释的自己。

回头看看上面的代码，有几个地方注意一下：

- 参数以及return
- 缩进
- +和*在字符串中的应用

先看一下上面代码的第一行

```
def repeat(s, exclaim):
```

def表示声明一个函数，repeat是这个函数的名字，括号里面的是这个函数参数的名字。这个函数有两个参数s和exclaim。

然后是函数里面的这些：

```
"""
    把s重复三遍返回
    如果exclaim是True的话，后面还要加上!!!
"""
```

这些是一个function里面的docstring，文档字符串。用来说明一个函数的功能。一般都是一个函数的第一行，但是如果你想多打一些回车的话，那可以用"""三重引号来表示*这一大段文字都是字符串*

> 三重引号是python独有的特性，js里面也有类似的一个三重撇号（es2015引入的？）

再往下：

```
result = s+s+s
```

这里+表示字符的连接。实际上对于字符串*也有意义，表示重复。所以再这个函数里面，用*\更合适一些。

> 如果我希望这个函数多一个参数times，表示重复s的次数，程序应该怎么改写？*是肯定要用的了

如果要引用刚才我们定义的函数，把刚才写的main()函数修改成：

In [25]:
```
def main():
    print (repeat('Yay', False))
    print (repeat('Woo Hoo', True))
```

再说说缩进

缩进对于python是有非常特殊的意义的。同属于一个层级的代码（比如一个if下面的一段，一个def里面的一段）需要具有同样的缩进。我以前写了一点点js，代码可能像这个样子：

```
function foo(){
 var a = 10
 var b = 'temp'
 var c = 'cast'
 if(a>5){
   console.log(b)
 }else{
   console.log(c)
 }
}
```

那么写成python的话是下面这个样子

In [26]:
```
def foo():
    a = 10
    b = 'temp'
    c = 'cast'
    if a>5 :
        print(b)
    else:
        print(c)
```

其实对比一下就可以看出来，python除了把function替换成def，console.log替换成print以外，其他的和js都差不多。最大的区别就是没了{}，少了一些()

但是，**缩进**大家都是差不多的。

当然，在js里面你可以不像上面这样缩进，程序不会说你有错，但是这种代码风格肯定是不OK的。python无非就是把这个代码风格上的要求变成了自己用来识别代码块的标准。所以用习惯了以后不会有什么反直觉的地方。

一般来说，这个缩进啊，最好用空格键，不要用TAB键。另外有些地方代码是用两个空格，有些地方代码是用4个空格表示缩进，所以有时候你拷贝别人代码过来的时候会在这种地方出错，自己注意一下就好。

所以到底是4个空格好还是2个空格好呢？PEP8标准建议4个，google内部用2个，所以你说咋整。

代码的实时审查

先看看下面的代码：

In [27]:
```
def main():
    if name == 'Guido':
        print repeeeet(name)+'!!!'
    else:
        print repeat(name)
```

```
  File "<ipython-input-27-d4ad95284f58>", line 3
    print repeeeet(name)+'!!!'
                  ^
SyntaxError: invalid syntax
```

上面的代码是有错的，我们定义的函数叫做repeat()，不是repeeeet()。但是呢，如果你运行的话程序并不会报错。

为什么呢？因为python是在运行的时候实时审查的。也就是说只要那么这个变量不是'Guido'这个值的话，print repeeeet(name)+'!!!'这一支代码根本就不会被运行，所以也就不会发现错误。

如果你只是写一下'hello world!'这样无关痛痒的小代码玩玩就算了，如果这是出错以后很要命的代码的话呢（比如用在自动驾驶或者医疗器械上面的代码），这就有点可怕了。因为如果没有很好的测试，你的代码可能写的时候完全ok，等到交付用户使用了以后才出问题。

像C#，java这样的静态检查代码就不会有这样的问题。当然了，这类语言的编写也有更多条条框框，毕竟有得必有失。

代码的命名

给代码命名的时候，不要像前面这样a啊，b啊之类的，这种变量命名完全没有任何意义，别人不容易看懂，自己也看不懂。有意义的变量名可以帮你更好的表达程序的意义。

如果变量名用一个单词不能完全表达意思的话，你可以在单词之间用_（下划线)连接，也可以用骆驼命名法来命名，比如camelName，也就是每个单词的第一个字母大写。一般来说用下划线连接更常用这种方法更常用。

当然了，像print, def, if, else之类这些东西是不应该作为变量名的。这类词统称保留字。一般的编程书籍都会单独开辟出一个区域来列一大堆的保留字，搞得你不背都不好意思一样。但是其实没什么可背的，首先，如果你用的是代码文本编辑器（比如sublime，atom）或者python的ide（比如pycharm），那么保留字会有特殊的颜色显示。此外代码只要写上一段时间都会对常用的保留字有所了解的了。

模块以及命名空间

比如你在一个binky.py的代码文件里面定义了一个函数'def foo()'。这个foo的全称实际上应该是binky.foo。这样的一个命名空间法则保证了不同文件里面的同名函数、对象不会彼此冲突。

之前遇到了sys.argv[1]，这里面的sys就是一个模块，argv是这个模块下面的一个list。通过命名空间我们拿到了它。

```
import sys
print(sys.argv)
```

除了单纯的import，还有`from sys import argv, exit`。这个代码的作用是把argv, exit这两个东西直接放到当前代码下面，前面不用再跟sys了。如果要引入的对象很常用，`from ... import ...`是不错的一个方式，但是平时还是单纯用import更好一些。毕竟sys.argv比起argv更能说明argv是个什么东西。

获取帮助

python自己是有文档的，你可以使用help()或者dir()获取一个函数的帮助说明。help的说明比dir要详细一些。

除了自带的说明，你还可以去网上查文档，官方的文档地址 (http://docs.python.org/)

然后更好的寻找答案的地方其实是在线编程社区，比如stackoverflow (http://stackoverflow.com/questions/tagged/python)和quora (http://quora.com/Python-programming-language)

但是，最重要的，还是自己敲代码。

python入门 - 字符串的操作

python自带了一个str类用来处理字符串。字符串可以用单引号（''）或者双引号（""）来表示。

> 类(class)是什么？well……总之你把它当成一个东西（object），有很多功能的东西就行了

所以如果字符串里面有'或者"怎么办呢？比如这么个字符串

In [28]:
```
print('He said:"thanks"')
```

```
He said:"thanks"
```

In [29]:
```
print("I didn't do it")
```

```
I didn't do it
```

In [30]:
```
print('I didn't do it')
#你看，颜色都不对了
```

```
  File "<ipython-input-30-53584c91997f>", line 1
    print('I didn't do it')
                  ^
SyntaxError: invalid syntax
```

In [31]:
```
print('I didn\'t do it')
```

```
I didn't do it
```

可以看到''里面可以放双引号，""里面可以放单引号。如果一定要在单引号里面放单引号，也可以用\来做escape。

之前我们还提到了docstring这么个东西，在一个函数的开头放上docstring用来说明这个函数是干嘛的。docstring可以用"""实现多行字符串的输入。三重引号除了可以换行，还可以在里面随便输入：

In [32]:
```
print("""
testing
testing
''
""
""")
```

```
testing
testing
''
""
```

不再受到双引号单引号引号的限制了。

不可变的字符串

在python里面，字符串是不可变的。看一下下面的代码：

In [33]:
```
string1 = "the old one"
string2 = string1
string1 = string1+" changed"
print(string1)
print(string2)
```

```
the old one changed
the old one
```

粗略一点说的话，字符串的不可变性决定了我们在赋值的时候，做的是拷贝，而不是引用。（如果是引用的话，那当我们在更改string1的时候，因为之前string1已经赋值给string2了，string2也应该跟着变化。）

In [34]:
```
list1 = [1,2,3,4]
list2 = list1
list1.append(5)
print(list1)
print(list2)
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

像list这样的，才是mutable的，list1变了，list2也会跟着变。关于list晚一些再说。

字符串里面某个位置的字符可以通过[]来获取。和java或者c++一样，python的序号也是从0开始。所以'hello'[1]是e而不是h。如果序号超出了字符串返回就会报错

In [35]:
```
'hello'[1]
```

Out[35]:
```
'e'
```

In [36]:
```
'hello'[5]
```

```
---------------------------------------------------------------------
-------
IndexError                                Traceback (most recent cal
l last)
<ipython-input-36-5cdae376624e> in <module>()
----> 1 'hello'[5]

IndexError: string index out of range
```

在**字符串**里面，+表示字符串的连接。所以+的两边都要是字符串

In [37]:
```
pi = 3.14
# text = 'value of pi is ' + pi
# this is wrong
text = 'value of pi is' + str(pi)
```

像前面所说的，*在字符串操作中表示重复：

In [1]:
```
print("abcd "*3)
```

```
abcd abcd abcd
```

python还有一种raw模式的字符串：在字符串的前面加一个r表示这个字符串是raw模式。看一下下面的代码，比较raw和普通模式字符串的区别

In [2]:
```
print('this\t\n and that')
```

```
this
 and that
```

In [3]:
```
print(r'this\t\n and that')
```

```
this\t\n and that
```

字符串的方法

字符串类有自己的方法，这些方法可以用来做常用的字符串处理，比如：

- s.lower()和s.upper()分别会把字母转成大些和小写

In [39]:
```
print('Python String'.lower())
```

```
python string
```

In [40]:
```
print('Python String'.upper())
```

```
PYTHON STRING
```

方法和函数很相似。不过方法是依托于对象，或者在我们这里，依托于类。一个字符串就是字符串类的一个instance（实例）。一个实例怎么调用类的方法呢？就像上面的代码一样：

```
实例.方法(参数)
```

在一个类的实例上面使用一个方法，获得一个返回结果，这是面向对象编程的基础思想。如果现在你不理解也没关系，记住这种用法。也就是说，如果你要想把一个字符串的所有字符转换成大些字母，你不能像获取字符串长度一样：

```
len('python')
```

而是要写成`'python'.upper()`。

- s.isalpha(), s.isdigit(), s.isspace()分别返回是否是字母，是否是数字，是否是空格

In [41]:
```
print('a is alpha: %s' %'a'.isalpha())
print('1 is alpha: %s' %'1'.isalpha())
print('  is alpha: %s' %' '.isalpha())
```

```
a is alpha: True
1 is alpha: False
  is alpha: False
```

In [42]:
```
print('a is digit: %s' %'a'.isdigit())
print('1 is digit: %s' %'1'.isdigit())
print('  is digit: %s' %' '.isdigit())
```

```
a is digit: False
1 is digit: True
  is digit: False
```

In [43]:
```
print('a is space: %s' %'a'.isspace())
print('1 is space: %s' %'1'.isspace())
print('  is space: %s' %' '.isspace())
```

```
a is space: False
1 is space: False
  is space: True
```

- s.strip()去除开头和结尾的空白符号

In [5]:
```
print(' strip the white spaces ')
print(' strip the white spaces '.strip())
```

```
 strip the white spaces
strip the white spaces
```

- s.startswith('other'):判断字符串s是不是以参数给出的字符串作为开始

In [7]:
```
print('alphabet start with '.startswith('alpha'))
```

```
True
```

- s.find('other'): 判断字符串是否包含参数的字符串，如果有，返回所在位置；如果没有，返回-1

In [8]:
```
print('there is an apple here'.find('apple'))
```

```
12
```

In [13]:
```
print('there is an apple here'[12:])
```

```
apple here
```

In [14]:
```
print('there is an apple here'.find('banana'))
```

```
-1
```

- s.replace('old','new'): 有find自然就会有replace。replace方法把字符串s里面找到的字符（'old')替换成新的字符('new')，并返回替换以后的字符。看一下代码：

In [15]:
```
print('there is an apple here'.replace('apple','banana'))
```

```
there is an banana here
```

- s.split('delim'): 依据分隔符把字符串拆分成列表并返回，我们看一下例子：

In [18]:
```
print('alpha beta gamma'.split(" "))
```

```
['alpha', 'beta', 'gamma']
```

In [19]:
```
print('one,two,three'.split(','))
```

```
['one', 'two', 'three']
```

In [20]:
```
print('alpha beta gamma'.split())
# split()的参数为空的时候，默认使用空格作为分隔符
```

```
['alpha', 'beta', 'gamma']
```

- s.join(list): 把列表进行连接，返回一个字符串，我们看一下例子：

In [24]:
```
print('--'.join(['a','b','c']))
```

```
a--b--c
```

In [25]:
```
print(','.join(['1','2','3']))
```

```
1,2,3
```

In [27]:
```
print(','.join[1,2,3])
```

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-27-a01060b8b24b> in <module>()
----> 1 print(','.join[1,2,3])

TypeError: 'builtin_function_or_method' object is not subscriptable
```

第三个为什么会报错呢?

字符串的切分

在前面我们的代码里面出现了这样一行：

```
print('there is an apple here'[12:])
```

这个就是对字符串做了一个切分。接下来仔细讲一下。切分指的是获取一个序列的一部分，这是列表或者字符串经常需要进行的操作。s[开始:结束]这个写法还有一些细节需要额外提一下。让我们以'hello'为例，用几行程序讲解一下。

In [29]:
```
str = 'Hello'
#  H  e  l  l  o
#  0  1  2  3  4
# -5 -4 -3 -2 -1
print(str[1:4])
#包头不包尾
```

```
ell
```

In [30]:
```
print(str[1:])
#不给出结束的时候就一直截取到序列的结尾
```

```
ello
```

In [31]:
```
print(str[:])
#开始也不给出的时候就获取序列的全部
```

```
Hello
```

In [32]:
```
print(str[1:100])
#超出范围的时候，不会报错，只会按照序列实际情况进行截取
```

```
ello
```

In [39]:
```
print(str[100])
#不过根据序号获取特定位置元素的时候，还是要注意别超出范围才好
```

```
---------------------------------------------------------------------
-------
IndexError                                Traceback (most recent cal
l last)
<ipython-input-39-2ef59b0d2401> in <module>()
----> 1 print(str[100])
      2 #不过根据序号获取特定位置元素的时候，还是要注意别超出范围才好

IndexError: string index out of range
```

从0开始的序列计数可以方便我们从序列的开头进行切分，但是如果你想拿到字符串后面几个字符怎么办呢？python除了0开始的序列，还有一个从最后往前倒数的负数序列。还是看几个例子:

In [33]:
```
print(str[-1])
#最后一个字符
```

```
o
```

In [40]:
```
print(str[-4])
#倒数第四个字符
```

```
e
```

In [41]:
```
print(str[:-3])
#从第一个字符开始，到倒数第三个字符
```

```
He
```

In [42]:
```
print(str[-3:])
#从倒数第三个字符开始，到最后一个字符
```

```
llo
```

切分包头不包尾的特性决定了一个很好的特点：s[:n]+s[n:]==s。不管你的n是正数，负数或者是超出了字符串范围的一个数字，这个等式都成立。在后面list部分可以看到，这个特点对于list也是成立的。

字符串的%

首先说一下，%这个符号是c语言的老物了，当然python不像是要放弃支持这个特性的样子，不过其实format()这个方法也是不错的。

%在字符串是一个有特定意义的**运算符**。先看几个例子：

In [45]:
```
text = 'I have %d apples' % 3
print(text)
```

```
I have 3 apples
```

In [46]:
```
text = '%s was fined %d times this week' % ('john',5)
print(text)
```

```
john was fined 5 times this week
```

在第一个例子里面，3替换了字符串里面的%d。而在第二个例子里面，有两个地方都有%，第一个%s被'john'替代，%d被5替代。%s在%d前面出现，所以会从()里面取第一个元素，%d相应的就会取第二个元素。()将两个元素括起来，组成了一个tuple。

if

if判定是所有程序都会具备的功能。一个if判定一般具有下面这样的结构：

```
if 条件 :
    语句块1
elif 条件2:
    语句块2
else:
    语句块3
```

下面的代码，根据星期几输出文本：

In [51]:
```
weekDay = 5
if weekDay<1 or weekDay>7:
    print('weekDay should be 1 to 7')
elif weekDay < 6:
    print('%d more days of work before weekend!' % (6-weekDay))
else:
    print('weekend!')
```

```
1 more days of work before weekend
```

上面的代码里，首先是判断给出的weekDay变量是不是1到7的数字，如果不是的话就报错。如果确实是1到7的数字（既不小于1也不大于7）那么就做判断，这个数字是小于6的么？如果是那么就表示现在是周一到周五，所以输出当前距离周末还有多少天。最后在else给出上面的判断都为否的时候执行的语句：输出weekend!

列表

有些程序也把这个东西叫做数组。列表由[]括起来

In [54]:
```
colors = ['red','blue','green']
print (colors[0])
print (colors[2])
print (len(colors))
```

```
red
green
3
```

colors这个数组标号和内容的对应情况如下表所示

标号	内容
0	red
1	blue
2	green

和string不一样，使用list的赋值不是复制，而是引用。再重新看一下前面提到的例子：

In [56]:
```
#list的赋值
list1 = [1,2,3]
list2 = list1
list1[2]=5
print(list1,list2)
```

```
[1, 2, 5] [1, 2, 5]
```

In [59]:
```
#字符串的赋值
str1 = 'abc'
str2 = str1
str1 = str1.upper()
print(str1,str2)
```

```
ABC abc
```

在上面的例子里面，我们利用赋值在list2和list1之间划了等号，和字符串的赋值不同，list的赋值传递的不是值，而是引用。简单的说，当你把list1赋值给list2以后，对list1的所有修改也会导致list2出现同样的变化。但是有的时候我们只希望list2获得list1当前的列表，但是并不希望list1的变化对list2产生影响，这个时侯可以考虑使用slice。

In [60]:
```
list1 = [1,2,3]
list2 = list1[:]
list1[1]=0
print(list1,list2)
```

```
[1, 0, 3] [1, 2, 3]
```

现在list1的变化就不会影响list2了。

for 和 in

介绍编程肯定要讲到循环。常用的循环方式包括了for和while。我们先说一下for...in...循环。

至少在python里面，for...in...循环是极其重要的。我们看一下下面的例程，程序的作用是把列表做一个累加求和：

In [62]:
```
squares = [1,4,9,16]
sum = 0
for num in squares:
    sum += num
    print (sum)
```

```
1
5
14
30
```

先看一下为什么会出来4个结果呢？提示：缩进

python里面是通过缩进来定义语句块的，我们把print(sum)放在了和sum+=num一个层次，就表示两个语句都是隶属于for...in...循环的。一般来说使用具有编程功能的文本编辑器都可以帮你自动缩进，但是如果你要退出当前的层次，到上一个层次的话，还是需要手动操作的。修改一下程序编程下面的样子就不会输出四次了：

In [63]:
```
squares = [1,4,9,16]
sum = 0
for num in squares:
    sum += num
print(sum)
```

```
30
```

接下来具体讲一下上面的程序。首先是`for num in squares:`，和if以及def一样，后面有个:，这个不要忘了。之所以写num，是因为我们知道squares里面的元素是一个个数字（number）起这个名字可以方便我们读懂程序。`for a in b`的结构里面，b是一个序列，a是这个序列里面的一个个元素。for循环会从序列的最开始往后一个个的遍历。

In [64]:
```
for e in [1,2,3,4,5]:
    print(e)
```

```
1
2
3
4
5
```

in这个词在python里面也是很重要的一个工具。在前面我们提到str.find('other')可以查找一个字符串中是否含有某段文字，并返回这段文字所在的位置。但是如果我们只想知道一个字符串是不是含有某段文字，或者一个列表是不是含有某个元素，使用in是更快的办法：

In [65]:
```
print(1 in [1,2,3])
```

```
True
```

In [66]:
```
print('alpha' in 'alpha dog')
```

```
True
```

In [67]:
```
print('1' in [1,2,3])
```

```
False
```

In [68]:
```
list = ['larry','curly','moe']
if 'curly' in list:
    print ('yay')
```

```
yay
```

for ... in ... 循环是非常常用的一个工具，除了可以遍历list，还可以遍历string：

In [70]:
```
for c in 'string':
    print (c)
```

```
s
t
r
i
n
g
```

Range

刚才我们算了1到4的平方和（1+4+9+16)。那如果我们要算1到100的和怎么办？难道先写一个1到100的数组么？如果是js或者c#这样的程序会有类似下面这样的写法：

```
var sum=0
for(var num = 1;num<100;num++){
    sum += num+1
}
console.log(sum)
```

不过python里面没有这么个工具，怎么办?

In [78]:
```
for n in range(10):
    print(n,end=' ')
```

```
0 1 2 3 4 5 6 7 8 9
```

先不用管print里面的end=' '，那个只是为了让输出不要那么占地方所以做的一个小修正。从上面的例子可以看到，我们能使用range()生成一个由序列数字组成的列表。再用几个例子看看range的用法：

In [79]:
```
for n in range(1,10):
    print(n,end=' ')
#设置序列开始和结束
```

```
1 2 3 4 5 6 7 8 9
```

In [84]:
```
for n in range(-8,10,2):
    print(n,end=' ')
#设置序列开始，结束，间隔
```

```
-8 -6 -4 -2 0 2 4 6 8
```

同样，range也是包头不包尾。

所以我们的1到100求和就可以用下面的程序来实现了：

In [86]:
```
sum = 0
for num in range(1,101):
    sum += num
print (sum)
```

```
5050
```

while循环

for循环可以应对绝大多数的情况，但是while循环可以给你更多的控制权。比如下面的程序可以按照3的间隔输出a序列的内容：

In [1]:
```
a = range(1,20)
i = 0
while i< len(a):
    print (a[i],end=' ')
    i = i+3
```

```
1 4 7 10 13 16 19
```

In [3]:
```
a = [1,2,3,4,5]
i = 0
while i<len(a):
    i+=1
    if a[i-1]==3:
        continue
    else:
        print(a[i-1])
```

```
1
2
4
5
```

In [4]:
```
a = [1,2,3,4,5]
i = 0
while i<len(a):
    i+=1
    if a[i-1]==3:
        break
    else:
        print(a[i-1])
```

```
1
2
```

在上面的代码里面，一旦你漏了i=i+1，那程序就会陷入死循环。这也是为啥我不那么喜欢用while。

上面的代码我们演示了continue的用法，continue表示跳过**当次**循环，break表示**结束**当前循环。比较一下上面两段程序有什么不同。

实际上在循环里面还有一些细节，比如如果你的计数变量，比如上面的i，是在循环内部进行累加的，那你把累加的这一步放在continue的后面就会导致循环可能一直停在某个位置。

所以，说实话，用for...in...吧

列表的一些方法

- list.append(elem) 将elem元素添加到列表的结尾。这个是很常用的一个办法。但是有个地方需要注意一下，看看下面的例程：

In [5]:
```
list1 = [1,2,3]
list1 = list1.append(4)
print(list1)
```

```
None
```

在上面的程序里面，我们先把[1,2,3]交给了list1，然后用append(4)把4这个元素添加到了list1的后面，照理说这个时候list1应该是[1,2,3,4]，为什么输出的结果会是None呢?

这是使用append方法经常出现的错误。append方法**不返回**新的列表，而是直接修改原先的list。简单的说，上面的代码应该这么写。

In [6]:
```
list1 = [1,2,3]
list1.append(4)
print(list1)
```

```
[1, 2, 3, 4]
```

在这一点上，string和list之间的区别是很明显的，务必注意。list的方法基本都是这个样子，不返回结果，直接修改原list。

接下来再说明一下其他的常用方法：

- list.insert(index,elem): 在index的位置插入elem

In [14]:
```
list1 = ['apple','pen']
list1.insert(1,'appleben')
print(list1)
```

```
['apple', 'appleben', 'pen']
```

- list.extend(list2): 类似javascript里面的concat，将list1和list2连接起来：

In [19]:
```
list1 = ['apple','pen']
list2 = ['apple pen','pen apple']
list1.extend(list2)
print(list1)
```

```
['apple', 'pen', 'apple pen', 'pen apple']
```

- list.index(elem): 和string里面的find有些相似，不过如果list里面不包含elem的话不会给出-1，而是报错。顺带一提，这个方法是有返回值的。

In [20]:
```
list1 = [1,2,3,4]
print(list1.index(1))
```

```
0
```

In [21]:
```
print(list1.index(5))
```

```
---------------------------------------------------------------------
-------
ValueError                                Traceback (most recent cal
l last)
<ipython-input-21-2c1e02054a43> in <module>()
----> 1 print(list1.index(5))

ValueError: 5 is not in list
```

同样，如果只是想判断list里面是否包含了elem，用in是最好的：

In [23]:
```
print(1 in list1)
```

```
True
```

In [25]:
```
print(5 in list1)
```

```
False
```

- list.remove(elem): 在list中找到第一次出现的elem，并且删除。如果找不到的话就会报错

In [26]:
```
list1 = [1,2,3,2,4,5]
list1.remove(2)
print(list1)
```

```
[1, 3, 2, 4, 5]
```

In [27]:
```
list1.remove(10)
```

```
---------------------------------------------------------------------
-------
ValueError                                Traceback (most recent cal
l last)
<ipython-input-27-f0eba5f26a1f> in <module>()
----> 1 list1.remove(10)

ValueError: list.remove(x): x not in list
```

- list.sort(): 对列表进行排序，无返回值
- sorted(list): 返回一个排序过了的列表

In [29]:
```
list1 = [1,4,5,7,2,0]
print(sorted(list1))
print(list1)
```

```
[0, 1, 2, 4, 5, 7]
[1, 4, 5, 7, 2, 0]
```

sorted()函数不会修改作为参数传入的list1，而是返回一个新的，排好序的list。实际使用中更加常用一些。list.sort()的例子如下：

In [30]:
```
list1 = [1,4,5,7,2,0]
list1.sort()
print(list1)
```

```
[0, 1, 2, 4, 5, 7]
```

- list.reverse(): 将序列反序，无返回值
- list.pop(index): 将index处的元素删除，并返回删除了的元素。

In [31]:
```
list1 = [1,2,3,4]
list1.reverse()
print(list1)
```

```
[4, 3, 2, 1]
```

In [32]:
```
list1 = [1,2,3,4]
print(list1.pop(1))
print(list1)
```

```
2
[1, 3, 4]
```

几个额外的问题

1. 如何把一个字符串反序?

In [35]:
```
string = 'abcd'
string.reverse()
print(string)
```

```
---------------------------------------------------------------------
-------
AttributeError                            Traceback (most recent cal
l last)
<ipython-input-35-a1c9f55beae6> in <module>()
      1 string = 'abcd'
----> 2 string.reverse()
      3 print(string)

AttributeError: 'str' object has no attribute 'reverse'
```

看到了吧，字符串是没有reverse这个方法的，毕竟人家是immutable的。那要怎么办才好呢?

可以先转换成list，反序以后再转换回string

In [36]:
```
string = 'abcd'
list1 = list(string)
list1.reverse()
string = "".join(list1)
print(string)
```

```
dcba
```

1. 如何对['1','21','3','45']进行排序？要根据数值大小进行排序哦。

In [38]:
```
l = ['1','21','3','45']
print(sorted(l))
```

```
['1', '21', '3', '45']
```

可以看到直接作sort是不行的，因为程序会使用字符串的排序法，也就是比较每个元素第一个字符在字符表上的顺序，所以虽然3比21小，但是因为3比2大所以3还会被排在21后面。

sort和sorted都可以接受一个叫做key的参数，这个参数指向一个函数。还是看一下例子吧：

In [39]:
```
l = ['1','21','3','45']
print(sorted(l,key=int))
```

```
['1', '3', '21', '45']
```

key=int是什么意思呢？key表示这个参数的名字，这个在每个sort都是一样的，关键在int。

int实际上代表的是`int(s)`这么一个函数。但是在作为参数的时候我们不写括号以及里面的参数。在将int(s)作为参数交给sorted以后，python会把l的每一个元素都作为参数传入int，获得一个返回值做一个转化：

元素	转化	转化结果
'1'	int('1')	1
'21'	int('21')	21
'3'	int('3')	3
'45'	int('45')	45

sorted之后就会根据转化以后的结果做一个排序。

用函数作为另外一个函数或方法的参数是不大直观的事情，需要自己多去想一想才能弄明白。目前不明白也不要紧，只要知道大致的写法就可以了。