Python自学监督小小班——基础语法简介

> 杜若整理编写，20220114

基础语法

前言

python和C语言不同，python语言是**面向对象的**高级动态编程语言，其不需要像C一样编译连接运行；python具有极高的灵活性，也具有极高的可拓展性。

python对于初学者来说有一些很好玩的特点，比如说它是以代码格式作为判断表达式是否结束的依据， 也就是说**python代码的格式也是代码的一部分**，该缩进的地方要缩进，该换行的地方要换行，不然格式错误也就是代码错误。 python实际上是可以利用分号作为判断一个表达式（逻辑行）是否结束的标志，但是我们极少使用分号来破坏代码块本身的格式美感。

python追求优雅简洁，所以不要把事情搞得很复杂哦！

推荐一个可以展示python内在运行机制的小工具： visual python

talk is cheap, show the code !

变量与运算符

变量

python不需要类似于C的显式变量声明；若一个新出现的变量没有在python中出现过或者被定义过，那python解释器会自动声明这样一个变量。

我们现在来试着声明变量：

以下代码框里的内容可以通过 ipython 或者 python shell书写，没有必要编写脚本文件。

```
# a 是一个变量，并且被赋值为数字6
a = 6
print(a)
print(a+6)

# b 是一个变量并且被赋值为一个字符串
b = "hello, ivy! "
print(b)
```

变量的命名有一些规则：

- 开头必须是字母或者下划线，不能使用数字开头
- 后面可以是字母、下划线、数字等， 如 _flag, flag5

- 变量的命名尽量不要和python的保留字如list、def、int、str等重复。

运算符

python中常见的运算符有 + 、 -、 *、 /、 and、 or、 is 等等。

在python之中我们不需要像C一样关心运算符两端的运算数是不是一定符合int等等类型，这些解释器会帮我们做，并且在绝大多数时候会得到我们期望的结果。

除了使用类似于C的逻辑运算符，你也可以使用 and、 or、 和 is 等来进行逻辑运算。

（后文大部分代码都是通过IPython写的，前面的`In [1]:`等字符是IPython的输入输出提示符，而非代码的一部分）。

```
In [1]: 1 and 0
Out[1]: 0

In [2]: 1 or 0
Out[2]: 1

In [3]: 1 is 0
Out[3]: False

In [4]: 1 is not 0
Out[4]: True
```

数据类型

python中所有的数据类型都是对象，遵循对象的继承等原则。

一个对象会有该对象附属的方法和属性，这一点之后你将知道；因为所有数据类型都是对象，所以我们在处理这些数据类型时，很多时候只需要调用该对象的属性或者方法就可以实现我们的目的。

数字

1. 整数型

   就是指整数啦，就像 5、 6、 13 等等，我们无需关心它是几位的，直接用就好了；
2. 小数型

   类似于5.6、 8.9、 3.1415926 等等
3. 类型转换

   有时候我们需要将整型（int）转换为小数型（float），或者将字符转换为数字。python为我们提供了几个函数来处理这些事情：

```
In [1]: x = 5           # 定义 x 为整型

In [2]: x = float(x)    # 利用 float 函数将 x 转换为小数
```

```
In [3]: x
Out[3]: 5.0

In [4]: x = "5"         # 定义 x 为一个字符

In [5]: int(x)          # 利用 int 函数将 x 转换为整数
Out[5]: 5

In [6]: float(x)        # 利用 float 函数将 x 转换为小数
Out[6]: 5.0
```

浮点数的比较可能因为电脑不同而有不同。计算机内部使用二进制来表示浮点数，因而会有一定的误差。如果要做高精度的计算，可能需要用到math.isclose()函数来判断两个浮点数是否相等。

H4 字符串

python中的字符串有三种表达的方式；

一种是以 "" （英文双引号）括起来的，一种是以 '' （英文单引号）括起来的；这两种字符串的表达都是同一个意思，在使用中我么要注意，如果字符串内本身就包含单引号，那我们外围用英文双引号括起来，如果还用单引号，则会出现解释器无法识别第二个单引号之后的内容。

```
In [1]: string_1 = "hello world! "

In [2]: string_2 = 'hello world! '

In [3]: string_3 = " i'm ur lover ~ "

In [4]: string_4 = ' i'm ur lover '
  File "<ipython-input-4-d24a732b622d>", line 1
    string_4 = ' i'm ur lover '
                   ^
SyntaxError: invalid syntax
```

就像上面的例子一样，第三个是ok的，第四个因为外围开始使用的是单引号，和 i'm 的单引号组成了一个单引号对，就导致解释器将字符串识别到 i'm 的单引号为止， 后面的内容无法识别，也就导致报错了。

第三种字符串的表示方式是利用 """ 或者 ''' 将字符串包围起来，这样的一个字符串可以有多行，也可以在其中任意使用单引号双引号。 这种类型的字符串在python代码中常常被用来表示注释或者整个代码文本的注释，即在代码的最开始，以这样的一个字符串来阐明代码的作者、功能等等。

```
In [1]: """
   ...: 这是一个超长的表示代码功能的文本，也就是我们常见的注释等等
   ...: """

In [2]: x = """ Jack says: "i’m a good man, and you are a good
   woman!" """

In [3]: x
Out[3]: ' Jack says: "i’m a good man, and you are a good woman!" '


```

表示注释的时候我们并不把这个字符串放进某个变量里，但是解释器还是会将其作为一个字符串处理，也就是说会占用我们的内存，而利用 # 注释掉的文本就不会被解释器执行。 不过也不用担心，python解释器有类似于垃圾回收处理的机制，没用的变量字符串等，过不了多久就会被清理掉。

H4 布尔类型

Boolean类型也就是 Ture、False 了，表示正确或者错误，False一般也可以用0表示，True一般也可以用非零数字表示。

```
In [4]: bool(9)          # 非零数字或者非空字符串会被 bool 函数解释为 True
Out[4]: True

In [5]: bool(0)
Out[5]: False

In [6]: int(True)        # 布尔类型的 True 会被int函数转化为1
Out[6]: 1

In [7]: int(False)       # False 被转化为0
Out[7]: 0

In [8]: bool("")
Out[8]: False

In [9]: bool('hh')
Out[9]: True

```

H4 list

list 也就是python中经常用到的列表了，你可以把它理解为C语言的数组。 其是由中括号 []括起来的，元素与元素中间以逗号(,)分隔的数据形式。

我们这里介绍list的一些基本操作，更深入的介绍我放到了数据类型一个小节。

string，list，dictionary都是需要初始化或者声明的。

list的初始化有两种方法：

```
In [1]: li = []             # 初始化一个列表并赋给变量li，列表为空

In [2]: li_1 = list()       # c初始化一个列表并赋给变量 li_1，列表为空

In [3]: li_1
Out[3]: []

In [4]: li
Out[4]: []

In [5]: li.append(5)        # 向列表 li 中增加一个元素 数字5

In [6]: li
Out[6]: [5]

In [7]: li_2 = [6,7]        # 初始化一个列表并包含元素 6，7

In [8]: li_2
Out[8]: [6, 7]

In [9]: li_1.append(1)

In [10]: li_1 += li_2       # 表示两个列表相加并赋值给li_1

In [11]: li_1
Out[11]: [1, 6, 7]
```

我们可以通过 list_name[index] 来查询某个列表指定下标的元素：

```
In [12]: li_1[0]            # 注意列表的开始下标为零！
Out[12]: 1

In [13]: li_1[2]
Out[13]: 7

In [14]: li_1[-1]           # 可以通过index = -1 查询列表最后一个元素，-n查询倒数第n个元素
Out[14]: 7
```

H3 控制流

H4 循环

python中较为常用的循环控制语句大概有两种，for语句和 while语句

while 语句的控制基本与C语言相同：

```
In [1]: i = 0                  # 需要提前定义循环的控制变量

In [2]: while i < 5:           # 注意python中while语句后面需要有一个分号，表示
   ...:     print(i)           # 这里是while语句内部的循环体，需要缩进，建议Tab缩进！！！（因为我是Tab党）
   ...:     i += 1             # python中没有 i++ 的写法，一般写为 i += 1
   ...:
0
1
2
3
4

```

for语句与C语言有较大不同，我们不再需要声明计数变量，常用的是下面这种形式：

```
In [1]: lis = [0, 1, 2, 3, 4, "hello, my lover "]

In [2]: for i in lis:
   ...:     print(i)
   ...:
0
1
2
3
4
hello, my lover

```

对于上面的例子讲解如下：对于每个属于列表lis的元素（元素被赋值给变量 i），都执行一遍print操作。对于for语句，in前面的变量会依次等于后面列表内的元素，从而实现循环的效果。这里并不限制后面列表内内容是什么，上式中列表内也有一个字符串。但是我们或许更常用for语句来计数并希望得到计数的值，那可以用下面的形式：

```
In [1]: range(5)              # 等同于range(0,5)，产生一个0到4的列表，等价于[0, 1, 2, 3, 4]
Out[1]: range(0, 5)           # 注意其生成的列表是从第一个参数开始，到小于第二个参数结束，也就是0开始到4，不包括5；若只写一个参数，默认是第二个参数，第一个参数默认为0

In [2]: for i in range(0,5):
   ...:     print(i)
   ...:
0
1
2
3
4
```

for语句还有一个值得注意的点，上面的运行机制相信你已经理解了，每一次循环结束，i会被重新赋值，赋的值就是列表的下一个元素，所以如果我们在循环体中改变了i的值，并不会影响循环的进行，这和C有很大差别。

```
In [2]: for i in [1, 2, 3, 4, 'hello']:
   ...:     if i == 1 or i == 4:            # 注意用 == 表示相等
   ...:         i += 10
   ...:     print(i)
   ...:
11
2
3
14
hello
```

循环过程中还涉及到两个保留字： break和continue

和C语言类似，break表示跳出循环，continue表示跳过此次循环体continue后面的内容。

```
In [37]: for i in range(10):
    ...:  if i == 5:
    ...:    continue
    ...:  print(i)
    ...:
0
1
2
3
4
6
7
8
9
```

```
In [38]: for i in range(10):
    ...:  if i == 5:
    ...:    break
    ...:  print(i)
    ...:
0
1
2
3
4

In [41]: for j in range(5):
    ...:  for i in range(10):
    ...:    if i == 5:
    ...:      break
    ...:    print(i, end = ' ')
    ...:
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
```

从上面的例子发现，对于多层嵌套循环，break只是跳出它所在的循环，并不结束所有循环。

条件

python并没有自带switch语句！
(python老师："但是你们自己实现一个switch不行吗？")

所以我们常用的就是简单的if-elif-else 语句：

```
In [3]: i = 3

In [4]: if i == 1:
   ...:     print('hello, pumi~ ')
   ...: elif i == 2:
   ...:     print('hello, lovely~ ')
   ...: elif i == 3:
   ...:     print('mua~ ')
   ...: else:
   ...:     print('this is in "else" ! ')
   ...:
mua~

In [5]: if i in [1, 2, 3, 'hello']:          # in 表示在列表中
   ...:     print(" it's in ")
   ...: elif i not in ['huga', 5, 6]:        # not in 表示不再列表中
   ...:     print("it's not in list_2 ")
   ...: else:
   ...:     print('in else')
   ...:
 it's in
```

```
22
23 In [6]: if i in [1, 2, 3, 'hello']:
24    ...:     print(" it's in ")
25    ...: if i not in ['huga', 5, 6]:          # 注意与上一个例子比较
26    ...:     print("it's not in list_2 ")
27    ...: else:
28    ...:     print('in else')
29    ...:
30  it's in
31 it's not in list_2
32
```

如上我们可以发现，在if-elif-else结构中，一旦某个条件成立，程序会跳出结构体，不再执行后面的内容，即使后面也有条件是符合的。而后一个例子实际上可以算是两个if结构，第一个结构只有一个if，后一个结构有if-else；所以程序执行完一个结构之后接着执行第二个if结构。

H4 **(*)try-except**

解释器有时候在处理我们的代码时会有无法理解的现象，也就是我们的代码可能格式或者syntax并不符合python的规则。

python的常见错误类型有： SyntaxError（语法错误）、 IndentationError（缩进错误）、KeyError（字典的键错误）、IndexError（下标错误）等等； 同时还会有各种非代码错误的错误原因，比如说需要从网络读入数据时，因为网络的原因导致数据无法读入或错误读入等等。

C语言是需要编译连接的，如果C语言代码有错，那么编译器会在编译的时候就告诉你，也就是编译不能通过，只有编译通过了才能生成可执行文件。但是python的解释器的运作机制并不一样，python解释器是解释一个逻辑行代码运行一行的，并不会产生C中的可执行文件（所以我们的脚本中被调用的函数一般要写在函数调用前面）。在python中，一般的错误发生时，程序会在发生错误的地方中断运行并抛出错误，这时候后面的程序不会再被执行。我们需要根据错误信息更正代码，并重新从头开始运行。

如果只是很小的脚本程序出错，那我们更正的代价或许不大，但是如果一个程序需要一直运行几个小时，类似于几十万数据量的爬虫，如果因为一个网络IO（input output）错误而导致代码崩溃，那我们的时间代价和资源代价就比较高了，这时候我们往往会利用try-except 语句来使得即使程序出错，但仍能够正确地处理错误并正常不中断的运行下去；这是其一。 其二，python给出的错误信息里包含了部分我们的源码等等信息，对于你可能的程序用户（如果有的话）来说是不容易明白的，我们可以把错误信息换成更用户友好的方式，另外对于潜在的破坏者来说，如果你直接展示python提供的错误信息，那他很有可能可以根据你的错误信息对你的程序进行攻击，所以一定的错误处理是必要的。

try-except 结构可以对某些我们认为可能会出错的代码进行 try 运行，如果真的发生了错误，那么程序会跳转到 except 代码块运行。

try-except

```
In [1]: a, b = 1, 0                    # 对a, b进行赋值，分别赋值1和0

In [2]: try:
   ...:     a/b                        # 这会产生一个除零错误
   ...:     print('this is in try block')
   ...: except:
   ...:     print('this is in except block')
   ...:
this is in except block                # 这一句是输出，可以看到 try 里面的print没有被运行，除零错误之后就跳转到except运行了
```

try-except-else-finally

这是一个比较完整的错误处理流程，工作原理是这样的：我们依靠经验把容易出错的部分放进try结构里，如果出错，程序跳转到except执行，如果没有出错，程序跳转到else执行；不管出没出错，finally结构里的东西都是会被执行的。

```
In [1]: a, b = 1, 0

In [2]: a, b = b, a                    # 交换a, b 的值

In [3]: try:
   ...:     b/a
   ...:     print('this is in try ')
   ...: except:
   ...:     print('this is in except ')
   ...: else:
   ...:     print('this is in else')
   ...: finally:
   ...:     print('this is in finally')
   ...:
this is in except
this is in finally

In [4]: try:
   ...:     a/b
   ...:     print('this is in try')
   ...: except:
   ...:     print('this is in except')
   ...: else:
   ...:     print('this is in else')
   ...: finally:
   ...:     print('this is in finally')
   ...:
this is in try
this is in else
this is in finally
```

H3 函数

H4 define

python中的函数和类是非常重要的概念，一个类所包含的函数也叫做类的方法。

我们这里介绍简单的函数定义：

```
In [5]: def function_name():
   ...:     print("-> miao~")
   ...: function_name()
   ...:
-> miao~
```

函数定义以关键字 def 开始，后面跟着函数名和括号，括号里面可以是参数。 之后是函数体；在需要调用函数的时候只需要写 函数名() 即可。

接下来我们来看一个传参的例子：

```
In [6]: def hello(name):
   ...:     print("hello " + name)
   ...:
In [7]: hello('ivy')
hello ivy

In [8]: hello('monica')
hello monica
```

在这个例子中，我们在函数定义是设置了一个参数，类似于C中的形参，name；在后面调用的时候，我们传进去实参，也就是两个字符串，'ivy', 'monica'，这就达到了我们想要的输出效果。

以上例子中的函数是没有 return 的，接下来我们来看一个有返回参数的例子：

```
In [9]: def re_func(name):
   ...:     string = name + ', i got u ~ '
   ...:     return string
   ...:
In [10]: s = re_func('ivy')

In [11]: s
Out[11]: 'ivy, i got u ~ '
```

在上面的例子中，我们定义了一个re_func() 函数，它接受一个name参数并返回一个字符串，我们只需要用一个变量 s 去接受函数的返回值即可。

最后我们来看一个小递归程序：

```
In [14]: def fib(n):
    ...:     if n == 1 or n == 2:
```

```
    ...:         return 1
    ...:     else:
    ...:         return fib( n-1 ) + fib( n-2 )
    ...:

In [15]: fib(3)
Out[15]: 2

In [16]: fib(5)
Out[16]: 5

In [17]: fib(6)
Out[17]: 8
```

以上的小程序实现了一个简单的求第n位斐波那契数列的功能，fib() 函数中最后一行出现了自身的调用；我们以fib(3)为例，n=3进入fib()函数，触发return调用fib(1)和fib(2)，这时候现在的工作现场会被压入内存堆栈中，程序执行fib(1)和fib(2)，得到返回值都是1，然后程序会从堆栈中取出之前的工作现场并进行计算得到返回值为3。

像这样的python递归调用，是有一定的层数限制的，程序不能一直在递归调用自身，因为每次调用，都需要把工作现场压入堆栈，也就会造成所需的内存空间随调用层数呈线性关系，调用层数越多，所需内存空间也越大。

```
In [18]: def fib():
    ...:     return fib()
    ...:

In [19]: fib()
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
<ipython-input-19-0a26aff926e2> in <module>()
----> 1 fib()

<ipython-input-18-a9b02fe1717f> in fib()
      1 def fib():
----> 2         return fib()

... last 1 frames repeated, from the frame below ...

<ipython-input-18-a9b02fe1717f> in fib()
      1 def fib():
----> 2         return fib()

RecursionError: maximum recursion depth exceeded
```

(*)变量作用域

一般来说我们在函数内部声明的变量和外面的变量是不在一个作用域的。

首先有一个全局的作用域，我们最开始用到的那些变量就是在全局作用域上工作的，我们可以理解为当def或lambda关键字出现时，一个新的变量域生成，函数里的变量都在这个新的作用域工作，同时这个新的函数作用域在函数内部优先于全局作用域，在函数内部，变量的检索方式是先检索函数作用域，如果没有发现我们要找的变量，那么解释器会到全局作用域中寻找。正因为这种机制，我们才有类似于闭包之类的好玩的东西，不过这里不做深入介绍。

```
In [1]: a = 4

In [2]: def exe():
   ...:     a = 5
   ...:     print(a)
   ...:

In [3]: exe()
5

In [4]: a
Out[4]: 4
```

在这个例子中，我们看到即使函数内部的变量和全局变量有相同的名字，也不会影响，全局变量a并没有因为函数内部的a的赋值而发生改变。

当我们在函数内部想要使用一个全局变量的时候，我们需要 global 关键字来声明

```
In [1]: a = 4

In [2]: def exe():
   ...:     global a
   ...:     a = 5
   ...:     print(a)
   ...:

In [3]: exe()
5

In [4]: a
Out[4]: 5
```

但是当我们在函数内部用 global 关键字声明了我们使用的是全局变量 a，那么在函数内部对a进行了修改，全局变量a也就被修改了。 如果某个变量是list、 dictionary之类的类型，那么即使我们不用global关键字，解释器依旧会根据先局部作用域再全局作用域的方检索式去查找。

```
In [1]: lis = [1,2]

In [2]: def exe():
   ...:     lis.append(4)
   ...:     print(lis)
   ...:

In [3]: exe()
[1, 2, 4]

In [4]: lis
Out[4]: [1, 2, 4]
```

或许为了避免我们搞不清楚变量属于哪个作用域，最简单的方法就是变量不要给一样的名字。

H2 数据类型

H3 组合数据类型

python中常见的数据类型有string、list、dictionary、 tuple、set。这一部分稍微深入一点讲讲这些数据类型的使用，更多专业的理解，还是推荐参阅更加有水准的tutorials。

组合数据类型为多个同类型或者不同类型的数据提供一种单一的表示，主要可以分为三类：序列类型、集合类型和映射类型。

序列类型是一个元素向量，元素之间存在先后关系，可以通过序号访问。

集合类型是一个数据的集合，元素之间无序，并且一个元素在集合中唯一存在。

映射类型是键-值对的组合，每一个元素是一个键值对，可以用(key, value)表示。

在python中，序列类型有字符串(string)、元组(tuple)、列表(list)；集合类型有集合(set)；映射类型有字典(dictionary。

H4 序列类型通用函数和方法

在开始之前先简单罗列序列类型通用函数和方法，或许可以帮助理解序列类型的一些共性。

操作	描述
x in s	如果x是序列s的一个元素，返回True，否则返回False
x not in s	如果x不是序列s的一个元素，返回True，否则返回False
s + t	连接序列s和序列t
s *n 或 n*x	将序列复制n次
s[i]	访问序列s下标为i的元素
s[i:j]	分片，返回包含序列i项到j项（不包括j项）的子序列
s[i: j: k]	步骤分片，返回包含序列i~j项中以k为步长的子序列

操作	描述
len(s)	求序列s长度
min(s)	求序列s中的最小元素
max(s)	求序列s中的最大元素
s.index(x[,i[,j]])	求序列s中i~j项中第一次出现元素x的位置
s.count(x)	求序列中出现x的总次数

H3 string

H4 string的输入

字符串的输入主要有两种方式：

input()

```
input()函数会读入终端的一行并作为字符串使用
```

eval()

```
eval()函数会将读入的字符串去掉两边的引号，直接将其作为python代码语句运行；在某些数据读入时确实很方便，但是也比较容易导致[代码注入]发生的可能性，所以我并不推荐在没有进行任何输入内容过滤的情况下粗暴地使用eval()。
```

```
In [22]: string = "[1,2,3,4,5]"

In [23]: lis = eval(string)

In [24]: lis
Out[24]: [1, 2, 3, 4, 5]
```

H4 string的访问和切片

前面已经提过了字符串的定义方式，通过引号将包裹字符形成字符串。
有些时候我们需要截取字符串的一部分，这时候我们可以通过 [] 来进行字符串中某个字符的访问和多个字符的截取：

```
In [1]: string = 'hello world !'

In [2]: string[0]     # 访问字符串string的第零位字符
Out[2]: 'h'

In [3]: string[6]
Out[3]: 'w'

In [4]: string[1:7]   # 截取字符串的第一位到第7位（不包括第7位）
Out[4]: 'ello w'
```

```
11
12 In [5]: string[:5]    # 截取字符串的第0位到第5位，不包括第5位；这里省略的第一个参数默认为0
13 Out[5]: 'hello'
14
15 In [6]: string[5:]      # 截取字符串的第5位到最后一位，包括最后一位
16 Out[6]: ' world !'
17
18 In [7]: string[-3:-1]    # 截取字符串的倒数第3位到最后一位，包括最后一位
19 Out[7]: 'lo world '
```

转义字符

最常用的转义字符主要有 \，\'， \"，\b（退格），\n（回车）等等，转义字符的做作用是在字符串中表示特定字符的含义。

```
1  In [12]: s  = 'hello\nworld!'
2
3  In [13]: print(s)
4  hello
5  world!
6
7  In [14]: s = 'hello\\world!'
8
9  In [15]: print(s)
10 hello\world!
11
```

string格式化

传统的字符格式化输出需要用到蛮复杂的表达式，从类型上来说有点类似于C语言，可以参考菜鸟教程，但是我并不推荐这种格式化方法，如果表达式过于复杂，代码的可读性会非常差。

这里推荐一个格式化字符的方法：format()

首先举个栗子：

```
1 In [1]: '{}{}'.format('hello ', 'world!')
2 Out[1]: 'hello world!'
```

format()方法的基本使用格式为 <模板字符串>.format(<逗号分隔的参数>)

模板字符串由一系列{}标记的槽组成，用来修改字符串中嵌入值出现的位置。其根本思想是将format()方法中逗号分隔的参数按照序号关系填充到模板字符串的槽中。槽用{}表示如果大括号中没有序号，则按照顺序填充，否则按照序号对其进行填充。

```
1 In [1]: '{0}的{2}不过是寻找一个{1}'.format('人生', '终点', '目的')
2 Out[1]: '人生的目的不过是寻找一个终点'
```

format()方法中模板字符串的槽除了包含参数序号，还可以包含一些格式控制信息。

:	<填充>	<对齐>	<宽度>	<,>	<.精度>	<类型>
引号符号	用于补充占位的单个字符	<左对齐 >右对齐 ^居中对齐	槽的设定输出宽度	数字的千位分隔符，适用于整数和浮点数	浮点数小数部分的精度或字符串的最大输出长度	整数类型 b,c,d,o,x,X 浮点数类型 e,E,f,%

以上的控制信息字段都是可选的。参见下面的例子：

```
In [4]: '{0:30}'.format(s)
Out[4]: 'python                        '

In [5]: '{0:>30}'.format(s)
Out[5]: '                        python'

In [6]: '{0:*^30}'.format(s)
Out[6]: '************python************'

In [7]: num = 1.2345678

In [8]: "{:,.2f}".format(num)
Out[8]: '1.23'

In [9]: '{0:*^3}'.format(s)
Out[9]: 'python'

In [10]: '{0:*^.3}'.format(s)
Out[10]: 'pyt'

In [12]: '{0:b}, {0:c}, {0:d}, {0:o}, {0:x},{0:X}'.format(12345)
Out[12]: '11000000111001, □, 12345, 30071, 3039,3039'

In [13]: '{0:b}, {0:c}, {0:d}, {0:o}, {0:x},{0:X}'.format(425)
Out[13]: '110101001, Σ, 425, 651, 1a9,1A9'

```

最后的例子中 b表示二进制方式，c表示整数对应的Unicode字符，d表示十进制，o表示八进制，x表示小写十六进制，X表示大写十六进制；同理对于浮点数，e表示浮点数对应的小写字符e的指数形式，E表示对应的大写字母E表示的指数形式，f表示标准的浮点形式，%表示浮点数的百分形式。

H4 string的方法

string类型共有43个内置的方法，我们认识几个常用的方法就好了，具体的可以查阅标准文档。

方法	描述
str.lower()	返回字符串的副本（并非直接在原字符串上修改），全部字符小写
str.upper()	返回字符串的副本，全部字符小写
str.islower()	当str所有字符都是小写时，返回True，否则返回False
str.isprintable()	当str所有字符都是可打印的，返回True，否则返回False
str.isnumeric()	当str所有字符都是数字时，返回True，否则返回False
str.isspace()	当str所有字符都是空格时，返回True，否则返回False
str.split(sep = None, maxsplit= -1)	返回一个列表，由str根据sep被分割的部分构成
str.count(sub[,start[,end]])	返回str[start:end]中sub子串出现的次数
str.replace(old, new[, count])	返回str的副本，所有old子串被替换为new，如果count给出，则只替换前count次出现的old子串
str.center(width[,fillchar])	字符串居中函数
str.strip([chars])	返回字符串str的副本，去掉了chars中列出的字符，若省略，则去掉字符串前后的空格
str.format()	返回字符串的排版格式
str.join(iterable)	返回一个新字符串，用str作为间隔串联起来的iterable

```
In [25]: string = 'hello this is an example for the tutorial of String'

In [26]: string.lower()
Out[26]: 'hello this is an example for the tutorial of string'

In [27]: string
Out[27]: 'hello this is an example for the tutorial of String'

In [28]: string.upper()
Out[28]: 'HELLO THIS IS AN EXAMPLE FOR THE TUTORIAL OF STRING'

In [29]: string.islower()
Out[29]: False

In [30]: string.split()
Out[30]:
```

```
['hello',
 'this',
 'is',
 'an',
 'example',
 'for',
 'the',
 'tutorial',
 'of',
 'String']

In [31]: string.strip()
Out[31]: 'hello this is an example for the tutorial of String'

In [32]: string.strip('h')
Out[32]: 'ello this is an example for the tutorial of String'

In [34]: string.center(60)
Out[34]: '    hello this is an example for the tutorial of String
    '

In [35]: lis = string.split()

In [36]: '*'.join(lis)   # 用*号将list中的元素连接起来
Out[36]: 'hello*this*is*an*example*for*the*tutorial*of*String'
```

string的编码

最后我们需要了解的就是string的编码方式了，我们常见的编码方式有Ascll，Unicode，utf-8等等，sublime默认使用utf-8编码方式，解释器的输出默认是cp936，IDLE的默认输出也是utf-8 。我们使用着各种不同的编码方式，为了编码的统一，一般的网页文档开头就会写明编码方式，我们的程序文件也可以在最开始声明编码，不过对于python3，并没有啥实际作用（编码取决于你的编辑器的输出），看着挺复古的可能（python2才需要声明编码，而python2即将被历史淘汰）。 我一般的文件头两行会这么写：

```
## author : Charlie
## date : 20220114
```

或者我也见过有人这么写（就是定义了两个变量，不具有太多的实际意义）：

```
__author__ = "Charlie"
__date__ = "20220114"
```

tuple

tuple是序列类型中比较特殊的一种，因为它一旦被创建就不能再被修改。它可以表达固定数据项、函数返回值、多变量同步赋值等等。

下面就结合例子来简单看看tuple有关的操作：

```
In [1]: tu = 'hello', 'monica', 12, [34, 56]     # 创建一个元组只需要用逗号将元组的元素隔开就行了

In [2]: tu
Out[2]: ('hello', 'monica', 12, [34, 56])

In [3]: tu_1 = ('hello','christina')              # 创建时也可以在最外层加上括号来表示

In [4]: tu_1
Out[4]: ('hello', 'christina')

In [5]: tu[1]                                     # 访问元组的元素方法和访问列表的相同
Out[5]: 'monica'

In [6]: tu[3][1]
Out[6]: 56

In [7]: tu_2 = ('hello', 'monica', tu)            # 元组也可以嵌套元组

In [8]: tu_2
Out[8]: ('hello', 'monica', ('hello', 'monica', 12, [34, 56]))

In [9]: tu_2[2][3][0]
Out[9]: 34
```

元组与列表不同的地方在于元组一旦被定义，则其内容是没有办法修改的；

下面介绍元组的一点点小应用，多变量同时赋值和循环的遍历：

```
In [10]: a, b = 'world', 'hello'          # 等号的两边都是元组，将一个元组的元素相应地赋值给另一个元组

In [11]: a,b = b,a                        # 原理和上面是一样的，我们把元素(b,a) 赋给了(a,b)，达到了交换a,b值的作用

In [12]: a
Out[12]: 'hello'

In [14]: b
Out[14]: 'world'

In [15]: for x,y in ((0,0),(1,1), (2,2), (3,3)):
    ...:     print(x,y)                   # 我们将元组内的元组元素赋给(x,y)元组并输出
    ...:

0 0
```

```
15 1 1
16 2 2
17 3 3
```

list

list和dictionary可能是我们最常用的数据类型。

list的长度和内容都是可变，属于可变数据类型。

list的创建与访问

我们前面已经讨论过了如何创建一个list，即可以通过 [ ] 或者 list() 创建。

当list()函数没有参数的时候，等同于[ ]，返回一个空列表，但是我们也可以利用list()将元组或者字符串转化为列表。

```
In [1]: lis = list((234, 'loveU', [12, 'mua'], 425))

In [2]: lis
Out[2]: [234, 'loveU', [12, 'mua'], 425]

In [3]: lis[2][1]
Out[3]: 'mua'

In [4]: list('谁还不是个小宝宝呢？！')
Out[4]: ['谁', '还', '不', '是', '个', '小', '宝', '宝', '呢', '？',
'！']
```

我们一般通过<列表名>[index]来访问指定列表中指定下标的元素。在上面的例子中，lis列表里还嵌套着一个小列表[12, 'mua']，我们通过访问lis[2]访问的就是lis列表的第2个元素，也就是嵌套的列表，再通过`lis[2][1]`访问小列表的第1个元素。多重嵌套的列表也可以这样访问。

list的常用方法

方法或操作	描述
ls[i] = x	用x替换第i项数据
ls[i:j] = lt	用列表lt的内容替换ls列表中的第i项到第j项，不包括第j项
ls[i: j: k]	用列表lt替换ls列表中的第i项到第j项以k为步长的数据
del ls[i]	删除列表第i项
del ls[i:j]	删除列表第i项到第j项
del ls[i: j: k]	删除列表第i项到第j项的以k为步长的数据
ls += lt 或 ls.extend(lt)	将列表lt的元素增加到ls的末尾
ls *= n	将列表的元素重复n倍并赋值给ls
ls.append(x)	将元素x添加到ls末尾

方法或操作	描述
ls.clear()	删除列表中的所有元素
ls.copy()	产生一个ls的副本
ls.insert(i,x)	在列表的第i位插入元素x
ls.pop(i)	弹出列表的第i项元素并在列表中删除该元素
ls.remove(x)	在列表中删除元素x
ls.reverse(x)	将列表中元素反转
ls.sort()	列表排序，可以设置排序的参数

list的方法有点多，我们通过例子来理解, 具体的讲解我写在了代码注释里哦

```
In [1]: lis = list('iiiiiiiiiii')   # 初始化一个列表

In [2]: len(lis)                    # len()函数求list长度，也就是元素个数
Out[2]: 11

In [3]: lis[5] = 'o'                # 将列表第5项替换为字符 'o'

In [4]: lis                         # 输出列表瞧瞧，可以看到第5项已经变成了 'o'
Out[4]: ['i', 'i', 'i', 'i', 'i', 'o', 'i', 'i', 'i', 'i', 'i']

In [5]: lis[1:4] = 'n'              # 将列表的第1项到第4项（不包括第4项）用一个字符'n'替换

In [6]: lis                         # 可以看到列表的1~4项都没了，替换成了'n'
Out[6]: ['i', 'n', 'i', 'o', 'i', 'i', 'i', 'i', 'i']

In [7]: lis[1:4] = ['m', 'm', 'm']  # 再用一个新列表替换列表的1~4项

In [8]: lis
Out[8]: ['i', 'm', 'm', 'm', 'i', 'i', 'i', 'i', 'i']

In [9]: len(lis)
Out[9]: 9

In [10]: lis[4:-1:2] = ['K', 'K']   # 以2为步长替换列表的4~-1项，也就是将4，6项（不包括-1项，也就是不包括8项）替换为‘K’，如果步长为3，则替换4，7项

In [11]: lis

Out[11]: ['i', 'm', 'm', 'm', 'K', 'i', 'K', 'i', 'i']
```

上面的例子大致可以说明如何更改列表中的元素，下面我们来看看删除列表中的元素：

```
In [1]: lis = list('0123456789')          # 创建一个列表

In [2]: lis
Out[2]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

In [3]: lis.append('10')                  # 在列表末尾添加字符 '10'

In [4]: lis
Out[4]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10']

In [5]: del lis[-1]                       # 删除列表的最后一个元素

In [6]: lis
Out[6]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

In [7]: del lis[:3]                       # 删除列表的前三个元素

In [8]: lis
Out[8]: ['3', '4', '5', '6', '7', '8', '9']

In [10]: lis.insert(0, '0')               # 在列表的第0位插入元素字符'0'

In [11]: lis.insert(1, '1')

In [12]: lis.insert(2, '2')

In [13]: lis
Out[13]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

In [14]: del lis[:: 3]                    # 以3为步长删除列表中的元素

In [15]: lis                              # lis中第0、3、6、9位的元素已经被删除
Out[15]: ['1', '2', '4', '5', '7', '8']
```

一大段一大段的代码哈哈哈哈哈哈哈哈哈哈，多读读代码好！嗯！

```
In [1]: lis = list('0123456789')

In [2]: lis += ['10', '11', '12']     # 将列表的元素添加到lis末尾

In [3]: lis
Out[3]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12']

```

```
In [4]: lis.pop()                    # 弹出列表末尾的元素并在列表中删除它
Out[4]: '12'

In [5]: lis.remove('11')             # 删除列表中的元素'11'

In [6]: lis
Out[6]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10']

In [7]: lis.clear()                  # 清空列表

In [8]: lis
Out[8]: []
```

同学们也可以在环境里敲敲代码啊，敲一敲就会了~

```
In [1]: lis = list('01234')

In [2]: li = lis.copy()              # 将lis复制一份给li

In [3]: li
Out[3]: ['0', '1', '2', '3', '4']

In [4]: lis *= 2                     # 将lis的元素重复一次

In [5]: lis
Out[5]: ['0', '1', '2', '3', '4', '0', '1', '2', '3', '4']

In [6]: lis.reverse()                # 将lis的元素反转

In [7]: lis
Out[7]: ['4', '3', '2', '1', '0', '4', '3', '2', '1', '0']

In [10]: lis.sort()                  # 对lis进行排序，这里并不是按数值排序，而是按字符的ascll码排序

In [11]: lis
Out[11]: ['0', '0', '1', '1', '2', '2', '3', '3', '4', '4']
```

H3 set

集合类型和数学中集合的类型定义一致，即包含0个或多个数据项的无序不重复序列，集合的元素类型只能是不可变数据类型，例如整数、浮点数、字符串、元组等。列表、字典、和集合类型本身都是可变数据类型，所以不能作为集合的元素出现。

```
In [16]: se = {1,2,{3,4}}        # 这里我们试图将集合作为集合的一个元素，解释器会报错；具体机制我们放在类的一节来讲。
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-b1321669d109> in <module>()
----> 1 se = {1,2,{3,4}}

TypeError: unhashable type: 'set'
```

集合是无序序列，所以没有索引的概念，也不能分片。集合中的元素可以动态增加和删除。

集合的定义

集合的定义主要使用set()函数，其可以用于生成集合，输入的参数可以是任何组合数据类型，返回结果是一个无重复的且排序任意的集合。

```
In [17]: se = set('banana ! ')     # 将字符串转换为集合

In [18]: se                        # 可以看到空格，'a'等重复的元素都只有一个了，而且排序随意
Out[18]: {' ', '!', 'a', 'b', 'n'}
```

集合的操作符

不知道大家对于数学中的集合了解多少，介绍结合的操作符需要了解一些结合的基本概念，例如子集、交集、并集、差集、补集。

或许我可以简单说明一下？ 那我们简单假设有两个集合S和T：

子集： 假如T中的元素在S中都存在，那么T是S的子集，表示为 T <= S

交集： 同时属于S和T的元素构成了S和T的交集，表示为 S & T

并集： S中的元素和T中的元素合起来称为S和T的并集，表示为 S | T

差集： 属于S但是不属于T的元素构成了S和T的差集，表示为 S - T

补集： 除去同时属于S和T的元素，S的元素和T的元素构成了集合S和T的补集，表示为 S^T

下面罗列集合的操作符(需要的时候查阅就好了)：

操作符	描述
S - T 或 S.difference(T)	返回一个新集合，包括在集合S中但是不在集合T中的元素
S -+ T 或 S.difference_update(T)	更新集合S，包括在集合S中但是不在集合T中的元素
S & T 或 S.intersection(T)	返回一个新集合，包括同时在集合S和T中的元素

操作符	描述
S &= T 或 S.intersection_update(T)	更新集合S，包括同时在集合S和T中的元素
S ^ T 或 S.symmetric_difference(T)	返回一个新集合，包括在集合S中和T中的元素，但是不包括同时在集合S和T中的元素
S ^= T 或 S.symmetric_difference_update(T)	更新集合S，包括在集合S中和T中的元素，但是不包括同时在集合S和T中的元素
S	T 或 S.union(T)
S	T 或 S.union_update(T)
S <= T 或 S.issubset(T)	若S是T的子集，返回True，否则返回False
S >= T 或 S.isuperset(T)	若T是S的子集，返回True，否则返回False

H4 集合的方法或函数

先列出集合类型的操作函数或方法如下：

操作函数或方法	描述
S.add(x)	如果数据项x不存在集合中，添加x到S
S.clear()	清空集合S
S.copy()	返回集合S的一个副本
S.pop()	随机返回S内的一个元素，如果S为空，产生KeyError异常
S.discard(x)	如果x在集合S中，删除该元素，如果不在，不报错
S.remove(x)	如果x在集合S中，删除该元素，如果不在，产生KeyError异常
S.isdisjoint(T)	如果集合S和T没有相同的元素，返回True
len(S)	返回基金和S的元素个数
x in S	检验x是不是在集合S内，是返回True，否则返回False
x not in S	检验X是不是不在集合S内

那我们来简单看看例子吧！

```
In [1]: s = set('0123456789')          # 初始化一个集合，包含0~9共十个字符

In [2]: s
Out[2]: {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'}

In [3]: s.add('10')                    # 添加字符'10'到集合中

```

```
In [4]: s
Out[4]: {'0', '1', '10', '2', '3', '4', '5', '6', '7', '8', '9'}

In [5]: s.pop()                         # 随机弹出一个集合中的元素，这里弹出了'4'
Out[5]: '4'

In [6]: s.discard('10')                 # 移除元素'10'

In [7]: s
Out[7]: {'0', '1', '2', '3', '5', '6', '7', '8', '9'}

In [8]: s.remove(10)                    # 移除数字10，因为不存在，所以报错
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-8-23b5f1bb77c8> in <module>()
----> 1 s.remove(10)

KeyError: 10

In [9]: t = set('huga')                 # 初始化另一个集合

In [10]: t
Out[10]: {'a', 'g', 'h', 'u'}

In [11]: s.isdisjoint(t)                # 看看两个集合有不有共同的元素
Out[11]: True

In [12]: '0' in s                      # 字符'0'在集合s里吗？！
Out[12]: True

```

集合一般会用于数据去重，在初学者阶段，set和tuple可能都会较少的接触，更多的是string，list，dictionary。

前边我们了解到set中的元素是无序并且不重复的，那假如说我们需要将一个list中的元素去重，但是却需要保留原来的顺序，我们能不能用set来实现呢？

我们可以将list转化为set去重，之后再将set转换为list并按照之前的顺序排序就好了，这里我们可以用到list.sort()函数来解决重新排序的问题。

```
In [1]: ids = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 10, 3, 5, 7, 3]

In [2]: ids_set = set(ids)                      # 将列表ids转换为集合ids_set

In [3]: ids_set
```

```
Out[3]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

In [4]: ids_list = list(ids_set)                    # 将集合ids_set转换为列表ids_list

In [5]: ids_list                                    # 这时列表并没有按照之前的顺序排列
Out[5]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In [6]: ids_list.sort(key = ids.index)              # 对列表ids_list排序

In [7]: ids_list
Out[7]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

上面的例子中ids_list.sort(key = ids.index) 中sort()的参数意思是：按照原列表ids的元素下标来排序，也就是说，在ids中靠前的元素在排序后的列表中也靠前。

H3 dictionary

字典也是我们经常会使用到的一种数据类型，它是属于映射类的组合数据类型。

当我们访问列表中的某个元素时，我们是通过索引来访问，但是假如说我们现在有一群人的姓名和他们的绩点成绩，已知他们的姓名，如何快速地查找他们的成绩呢？

或许我们也可以用列表来实现，每个人的姓名和成绩组成一个小列表，所有人的数据再组合成为一个大列表。

但是这样的话，似乎要知道一个人的成绩，就需要遍历整个列表，从效率上来说是很低的，而python的字典就正好解决了这样一个问题，其采用键作为检索的索引，效率非常高，存储几十万项内容依旧没有问题。

H4 字典的初始化

字典的初始化有两种方法，一种是使用 dict() 函数，一种是直接使用大括号 {} 来创建一个空字典。字典的键值对与键值对之间用逗号(,)分隔，每个键值对的键与值之间用分号(:)分隔。

```
In [1]: dic = {}

In [2]: dic
Out[2]: {}

In [3]: dic_1 = dict()

In [4]: dic_1
Out[4]: {}

```

访问字典中的元素时我们用 dictionary_name[key_name]，当我们需要添加一个键值对到字典中时，我们使用 dictionary_name[key_name] = value 即可。

```
In [1]: dic = {'charlie': 'science', 'ivy':'arts'}   #初始化一个字典

In [2]: dic
Out[2]: {'charlie': 'science', 'ivy': 'arts'}

In [3]: dic['ivy']                          # 查询字典中键为'ivy'的值
Out[3]: 'arts'

In [4]: dic['i']                            # 查询字典中键为'i'的值，不存在所以会有报错
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-4-75add8a4144f> in <module>()
----> 1 dic['i']

KeyError: 'i'

In [6]: dic['monica'] = 'my computer'  # 新增一个键值对，键为'monica'，值为'my computer'

In [7]: dic
Out[7]: {'charlie': 'science', 'ivy': 'arts', 'monica': 'my computer'}

```

H4 字典的操作

下面介绍字典常用的几种函数和方法

函数和方法	描述
d.keys()	返回所有的键的信息
d.values()	返回所有的值的信息
d.items()	返回所有的键值对
d.get(key, default)	如果键存在，就返回相应的值，否则返回default值
d.pop(key, default)	如果键存在，就返回相应值，并删除键值对，否则返回默认值
d.popitem()	随机从字典里取出一个键值对，以元组(key, value)的形式返回
d.clear()	清空字典
del d[key]	删除字典中的某一个键值对
key in d	检验key是否在字典里

Talk is cheap, show code again again again ……

```
In [1]: dic = {'charlie': 'science', 'ivy': 'arts', 'monica':'act',
'christina': 'computer'}       # 初始化一个字典

In [2]: dic.keys()             # 返回dic的键
Out[2]: dict_keys(['charlie', 'ivy', 'monica', 'christina'])

In [3]: dic.values()           # 返回dic的值
Out[3]: dict_values(['science', 'arts', 'act', 'computer'])

In [4]: dic.items()            # 以元组的形式返回所有键值对
Out[4]: dict_items([('charlie', 'science'), ('ivy', 'arts'),
('monica', 'act'), ('christina', 'computer')])

In [5]: dic.popitem()          # 随机以元组的形式弹出一个键值对
Out[5]: ('christina', 'computer')

In [6]: dic.get('charlie', 'no this key in dic')     # 获取
dic['charlie']
Out[6]: 'science'

In [7]: dic.get('c', 'no this key in dic')           # key为'c'的值并
不存在，所以返回default参数
Out[7]: 'no this key in dic'

In [8]: dic.pop('monica', 'no this key in dic')      # 弹出
dic['monica']，并删除该键值对
Out[8]: 'act'

In [9]: dic
Out[9]: {'charlie': 'science', 'ivy': 'arts'}

In [10]: 'charlie' in dic     # 检验键为'charlie'的键值对存不存在
Out[10]: True

In [12]: dic['hello'] = 'world'          # 新增一个键值对

In [13]: dic
Out[13]: {'charlie': 'science', 'ivy': 'arts', 'hello': 'world'}

In [14]: del dic['hello']                # 删除键为'hello'的键值对

In [15]: dic
Out[15]: {'charlie': 'science', 'ivy': 'arts'}

In [16]: dic.clear()                    # 清空所有键值对

In [17]:
```

```
In [17]: dic
Out[17]: {}

```

很多时候我们可能需要遍历字典，将它的键值对拿出来做一些处理

```
In [1]: dic = {1:0, 2:0, 3:0, 0:0}

In [2]: for key in dic.keys():            # 遍历访问字典的键
   ...:     print(key, end = ' ' )        # print函数后面的end参数表示输出的结尾是一个空格，默认是换行'\n'
   ...:
1 2 3 0
In [3]: for value in dic.values():        # 遍历访问字典的值
   ...:     print(value, end = " * ")
   ...:
0 * 0 * 0 * 0 *
In [4]: for key, value in dic.items():    # 遍历访问字典的键值对
   ...:     dic[key] = 1                  # 将字典的值都改为 1
   ...:     print(key, value)
   ...:
1 0
2 0
3 0
0 0

In [5]: dic                               # 字典所有的值都被改为 1
Out[5]: {1: 1, 2: 1, 3: 1, 0: 1}

```

好啦，所有的数据类型基本上都说清楚啦~ 努力 coding ~

H2 第三方库

H3 the zen of python

python 有它自己的一些编程哲学或者说理念，我们大概可以通过python之禅来理解体会它。

```
In [1]: import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.

Sparse is better than dense.
```

```
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

打开文件

打开文件一般只需要open(filename, open_methods)，并将打开的结果放入一个文件对象就可以了，常用文件打开方式有'w', 'r', 'b', 'a'； 'w'指以写入的方式打开，无论文件里打开之前有什么内容，一旦开始写入，就会清空以前所有的内容,'r'表示以只读方式打开,'b'表示以二进制方式打开,'a'表示以*追加*方式打开，新写入的内容将追加在原来内容的后面。

以下示例打开文件的基本操作：

```
file_object = open(filename, 'r'): # 以只读方式打开文件并将之赋值给文件对象
    all_text = file_object.read()   # 读取所有内容
    print(all_text)

file_object.close()                 # 关闭文件
```

值得注意的是，我们操作完文件之后一定要将之关闭，否则该文件可能一直被我们占用，其它程序可能无法使用它；有一个简单的方法来保证文件一定会被关闭，就是使用 'with' 语句。

```
with open(name, 'a') as fo:
    fo.someOperation()
```

内置库

python3内置了一些基本的库，例如os, math, time, csv等等，使用它们可以方便我们的编程。我们可以通过查看库的文档来了解其提供的相关函数等服务。

查看python库文档的方法：

1. 在cmd中输入 `python -m pydoc -p 3000`，该命令意思是 调用pydoc模块在本地的3000端口上生成文档显示。
2. 打开浏览器，在地址栏输入 `http://localhost:3000/#/` 即可访问内置库和第三方库的文档内容。

除了利用上述方式了解库，我们也可以查阅python官方文档和第三方库的官方文档。

所有的库使用的时候都需要用`import`语句导入。

math

math不能进行复数计算，但是可以胜任整数和浮点数运算；其提供了4个数学常数和44个函数。

以下我们调最常用的几个函数进行介绍，如需要用到其它函数，可以参阅文档。

1. math.pi 表示圆周率。
2. math.e 表示自然对数。
3. math.fabs(x) 返回x的绝对值。
4. math.fsun([x,y,z,……]) 表示浮点数进行精确求和，会比 sum() 函数精确很多。
5. math.pow(x,y) 返回x的y次幂。
6. math.exp() 返回e的x次幂。
7. math.sqrt(x) 返回x的平方根。
8. math.log(x [,base]) 返回以base为底数的x的对数值。
9. math.sin() 返回x的正弦函数值。
10. math.cos() 返回x的余弦函数值。
11. math.tan() 返回x的正切函数值。

time

这个库我用得较少，一般使用它的clock()函数来计算程序运行的时间或者格式化输出当前时间。

```
t1 = time.clock()
# some other python codes
t2 = time.clock()
t2 - t1   # 此处的结果就是两次clock()之间的时间差。
```

```
print(time.strftime("%Y-%m-%d %H:%M:%S"))
```

csv

csv是excel也支持的一类简单的数据组织形式，其值与值之间以简单的逗号相互隔开，行与行之间用'\n'隔开。

内置的csv库函数可以帮助我们方便快捷的读取和写入csv文件。

下面简单介绍csv文件的写入和读取：

```
import csv
with open('name.csv', 'ra', newline = '') as fo:
    read_content = csv.reader(fo)   # 读取文件内容
    print(read_content)
    writer = csv.writer(fo, dialect = 'excel') # 创建文件的writer 对象
    writer.writerow(list)                 #利用writer来写入一行内容，参数为一个列表。
```

json

json 是一种以键值对组织数据的数据交换格式。 类似于dictionary；

json的数据保存在键值对中，键值对与键值对之间以逗号分隔，大括号用于保存键值对数据组成的对象，中括号用于保存键值对数据组成的数组。sublime 的设置文档就是一种json格式。

json库提供提供了几种对于json格式的处理，以下介绍最常用的函数：

1. json.dumps(obj, sort_keys = False, indent = None) 将python的数据格式转换为json格式。
2. json.loads(string) 将json格式的字符串转换为python的数据类型。
3. json.dump(obj, fp sort_keys = False, indent = None) 将python的数据格式转换为json格式，输出到文件fp。
4. json.load(fp) 从文件fp读入json格式数据。

```
{"lovers" : [
            {
                "name":"charlie",
                "age":22,
                "height":171,
                "weight":56,
                "love":["Ivy", "chocolate"]  /*此键值对的值为一个列表*/
            },                              /*此处一个键值对对象结束*/
            {
                "name":"Ivy",
                "age":22,
                "height":156,
                "weight":50,
                "love":["charlie", "blue"]
            }           /*两个键值对对象一起组成一个列表，成为lovers键的值
*/
        ]
}
```

假如上方的json数据为一个字符串json_string，我们现在需要读取Charlie的love项并输出，我们可以怎么做呢？

```
import json
dic = json.loads(json_string)
love = []
love = dic['lovers'][1]['love']
for i in love:
    print(i)

```

需要注意的是，json.loads()函数在解析json字符串的时候，如果键值为一个字符串，那么该字符串一定得是用**双引号**包裹，如果仅仅是单引号，可能会报错。

```
In [1]: import json

```

```
In [2]: json_s = """{ 'charlie' : [1,2,3] }"""

In [3]: json.loads(json_s)
---------------------------------------------------------------------------
JSONDecodeError                           Traceback (most recent call last)
<ipython-input-3-94410645e07c> in <module>()
----> 1 json.loads(json_s)

c:\users\hhhhh\c_python\lib\json\__init__.py in loads(s, encoding, cls, object_hook, parse_float, parse_int, parse_constant, object_pairs_hook, **kw)
    346             parse_int is None and parse_float is None and
    347             parse_constant is None and object_pairs_hook is None and not kw):
--> 348         return _default_decoder.decode(s)
    349     if cls is None:
    350         cls = JSONDecoder

c:\users\hhhhh\c_python\lib\json\decoder.py in decode(self, s, _w)
    335
    336         """
--> 337         obj, end = self.raw_decode(s, idx=_w(s, 0).end())
    338         end = _w(s, end).end()
    339         if end != len(s):

c:\users\hhhhh\c_python\lib\json\decoder.py in raw_decode(self, s, idx)
    351         """
    352         try:
--> 353             obj, end = self.scan_once(s, idx)
    354         except StopIteration as err:
    355             raise JSONDecodeError("Expecting value", s, err.value) from None

JSONDecodeError: Expecting property name enclosed in double quotes: line 1 column 3 (char 2)
```

H3 第三方库

python拥有丰富的第三方库，也有活跃的社区。 可以利用pypi查询我们需要的库。 各领域有各领域经典的或者强大的第三方库，比如说自然语言处理，数据分析挖掘，机器学习等等领域，都有非常著名的库，需要的时候搜索安装使用即可。

一般较为复杂的第三方库都有自己特定的一些语句结构或者语法，虽然都是以python为基础，但是也是需要花时间精力研究的。或许有兴趣可以去看看库的源码实现，对其的认识会好很多。

类

Class

平时写写小脚本较少用到类，因此这里仅仅简单介绍下类的定义和使用，关于继承多态等等概念这里暂时不做介绍。

类的定义

我们先看一个例子：

```
#类的定义
class People:
    """ this is a Class"""
    def __init__(self, name, age, color = "blue"):
        self.name = name
        self.age = age
        self.color = color

    def say_hay(self):
        print(self.name + ": hay! ")

    def tell_age(self):
        print(self.name + ": i'm " + str(self.age))

#实例化

charlie = People('charlie', 22, 'rose_red')
ivy = People('Ivy',22)

```

上方定义了一个People类，并给出了它的初始化函数`__init__`和两个方法，`say_hay` 和`tell age`；

定义一个类需要用关键字class（建议类名字'People'的首字母大写），之后可以稍微对定义的类进行一些注解；之后紧跟着就是`__init__`函数，以这个名字命名的函数，在类每次被实例化的过程中都会被运行，并给类做一些初始的设置；

类中的函数的第一个参数都是self，表示自身的意思，在初始化函数的参数中，name、age因为没有在函数定义时写入默认参数，所以调用时是一定需要给出实参的，而color因为已经给出了默认参数为"blue"，所以调用时可以不用给出实参。

初始化函数中的`self.name`表示定义了类的一个属性name，这个属性的值被赋予为实例化这个类时给出的nama的实参。 而之后的两个方法，都没有外界的参数，输出语句中的'self.name'等都表示该类的属性。

类的实例化

类的实例化，即是通过类来定义出一个对象，类可以简单理解为对象的模板，我们可以通过类实例化出很多的具有相似性质的对象。上方的例子中我们通过People类实例化出了两个对象，分别是charlie和ivy，它们分别被赋予名字年龄等属性。

既然这些对象是通过同一个类产生的，那么它们都具有该类的方法，我们可以通过调用这些方法来对对象进行一些操作。

```
class People:
    """ this is a Class"""
    def __init__(self, name, age, color = "blue"):
        self.name = name
        self.age = age
        self.color = color

    def say_hay(self):
        print(self.name + ": hay! ")

    def tell_age(self):
        print(self.name + ": i'm " + str(self.age))

charlie = People('charlie', 22, 'rose_red')
ivy = People('Ivy',22)

charlie.say_hay()   # 调用charlie对象的方法
charlie.tell_age()
ivy.say_hay()
ivy.tell_age()

charlie.age = 6     # 更改charlie对象的一些属性
charlie.tell_age()
```

运行结果为：

```
charlie: hay!
charlie: i'm 22
Ivy: hay!
Ivy: i'm 22
charlie: i'm 6  # 更改属性之后，charlie的年龄改变了
```

要深入学习python，就需要掌握python的类等的高级用法，需要熟悉python的特性。