Hao WANG

Carestream  [Company address]

本文来源于本人blog：https://i.cnblogs.com/posts?categoryid=1430366

Python 基础知识

目 录

[**Python 学习笔记1 安装和IDE** 2](#_Toc25870191)

[Python 学习笔记2 变量 3](#_Toc25870192)

[Python 学习笔记3 变量-数字 4](#_Toc25870193)

[Python 学习笔记4 变量-字符串 7](#_Toc25870194)

[**Python 学习笔记5 变量-列表** 12](#_Toc25870195)

[**Python 学习笔记6 变量-元组** 17](#_Toc25870196)

[**Python 学习笔记7 变量-字典** 19](#_Toc25870197)

[**Python 学习笔记8 条件语句 If** 23](#_Toc25870198)

[**Python 学习笔记9 循环语句 while** 25](#_Toc25870199)

[**Python 学习笔记10 循环语句 For in** 29](#_Toc25870200)

[**Python 学习笔记11 函数** 32](#_Toc25870201)

[**Python 学习笔记12 函数模块** 37](#_Toc25870202)

[Python 学习笔记13 类 - 创建和简单使用 40](#_Toc25870203)

[Python 学习笔记14 类 - 使用类和实例 43](#_Toc25870204)

[Python 学习笔记15 类 - 继承 48](#_Toc25870205)

[Python 学习笔记16 类 - 导入 55](#_Toc25870206)

[Python 学习笔记17 文本 - 读写 58](#_Toc25870207)

[Python 学习笔记18 异常处理 63](#_Toc25870208)

[Python 学习笔记19 安装robot Framework 67](#_Toc25870209)

[Python 学习笔记20 自定义robot Framework 关键字 73](#_Toc25870210)

[方法一： 74](#_Toc25870211)

[方法二 77](#_Toc25870212)

[Python 学习笔记21 CMD执行测试用例 83](#_Toc25870213)

[**Python 学习笔记1 安装和IDE**](https://www.cnblogs.com/wanghao4023030/p/10610587.html)

前面的话

现在随着互联网的快速发展，对测试人员的代码要求也越来越高。有种逐步往全栈开发人员发展的趋势。

越来越多的手工测试被自动化取代。 对于测试人员，学习一门开发语言迫在眉睫。 C#， JAVA， Go， Python，都是不错的选择。

Python近年来异军突起，结构简单，类库较多，可以作为一门入门的语言进行学习。

以下是我在学习的一些笔记，作为读书练习后的一些小结，加深学习印记。

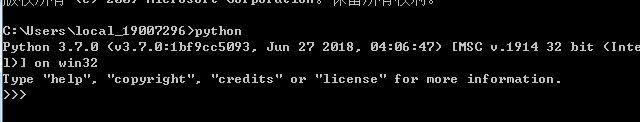
1. 获取安装文件：

对于linux用户，许多系统自动集成了python环境， 也可以通过自身安装库进行下载。这里不做进一步描述了。

对于windows我们可以在python社区获取最新的安装包， <https://www.python.org/downloads/>

下载后，根据提示默认安装即可。

最新的安装包自动添加环境变量到系统中，我们打开CMD窗口，输入 “python”，如果系统显示python安装版本，以及出现python提示符，表示安装成功：



2. IDE 工具

Python 自带一个IDE但是不是非常好用。

我们当然也可以使用notepad或者notepad++， 编写脚本并使用python进行编译后运行，但是这种方式效率不发哦。

网上有很多工具可以使用，提高我们的开发效率：

比如该网址中介绍的几款： <http://www.runoob.com/w3cnote/best-python-ide-for-developers.html>

我个人比较推荐 PyCharm， 现在该公司提供免费的社区版，供大家使用：

<http://www.jetbrains.com/pycharm/download/#section=windows>

# [Python 学习笔记2 变量](https://www.cnblogs.com/wanghao4023030/p/10610769.html)

每种编程语言都需要变量， 这些变量的命名，我们一般会遵守一些公认的规则。 已达到方便自己，他人阅读的好处。

1. 变量只能包含字母、数字和下划线。变量可以以字母和下划线打头，但是不能以数字打头， 比如 message\_warning\_1是可以的， 1\_message\_warning是不被允许的。推荐大家使用小写字母加下划线的方式， 命名变量。
2. 变量名不能包含空格，可以使用下划线分隔过长的名称。
3. 不要使用python预留的关键字作为变量名。
4. 变量名应该既短小又具有描述性， 比如 name， student\_name
5. 做好不要使用l和O，有概率被看成数字。

有关其他比如类名，包，函数等命名规则我们后续在进一步了解。

python中的变量是不需要声明类型的，但是必须赋值，赋值后变量才会被创建。而我们通常所说的类型，也只是其保存在内存中对象的类型

Python有以下几种变量类型：

* Number（数字）
* String（字符串）
* List（列表）
* Tuple（元组）
* Set（集合）
* Dictionary（字典）

Number：

　　Python中的数字包含**int、float、bool、complex复数。**

　　int： 　　　　通常我们所说的整数， 比如 1， 2 ，3 ，100， 3000 等等。

　　float：　  　　即浮点型， 比如1.2，2.0001， 3.1415926等等。

　　bool：　  　　python3里面将其定义成为整数， 在python2里面是没有bool型的， 我们通常使用1,0来表示。

　　complex：　   是指数学意义上的复数，比如 2+7i。

String：

　　python里面的字符串是指用单引号 ‘ 或者双引号 “” 括起来一段字符和数字，同时一些特殊字符需要使用 \ 来进行转译。

　　string = "1AAaa11\\"

List：

　　List在python中使用会比较频繁，他和数组很大的相似性。可以实现许多集合类的实现。

　　list = ["1", "a", 1, 1.5, ["a", 1]]

Tuple：

　　元组是一组不能修改的数据集合，使用（）和“，” 将数据进行分隔。

　　tuple = ( 'abc', 123 , 3.14, 'ralf', 70.2 )

Set：

　　一个用“{}”包含起来一组或多个个体组成的集合，通常是进行成员关系测试和删除重复元素使用的。

　　注意空的set，要使用set（）来实现， { }是用来表示一个空的字典。

  　　set\_list = 'a', 'ab', 'c'} 或者 set（‘abcdef’）

 Dictionary：

　　字典是一种数据映射类型，用{ } 进行标识，它包含了一组由 key:value组成的集合。

　　key（键）和 value（值）一一对应，在同一个字典中，key值必须是唯一的，不能够重复。

　　list是有序的， Dictionary昰无序的。

以上就是Python中变量类型，后续我们逐步深入学习他们的定义和使用方法。

# [Python 学习笔记3 变量-数字](https://www.cnblogs.com/wanghao4023030/p/10617737.html)

我们来具体了解下有关 number类型的变量的使用方式和含义。

在Python中的Number类型的变量包含以下几种：

**int：** 　　　　通常我们所说的整数， 比如 1， 2 ，3 ，100， 3000 等等。

**float：**  　　即浮点型， 比如1.2，2.0001， 3.1415926等等。

**bool：**　  　　python3里面将其定义成为整数， 在python2里面是没有bool型的， 我们通常使用1,0来表示。

**complex**：　   是指数学意义上的复数，比如 2+7i。

接下来我们一个个来认识他们。

* **Int（整型）：**

　　通常被称为是整型或整数，他可以是正数，也可以是复数。不能包含小数点。Python3 整型是没有限制大小的， 理论上可以无穷大，只要计算机的内存可以支撑存储和运算。

[复制代码](javascript:void(0);)

score = 99

print(99)

long\_int = 123456789012345678900

print(long\_int)

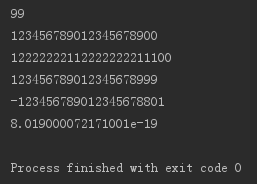
print(score \* long\_int)

print(score + long\_int)

print(score - long\_int)

print(score / long\_int)

[复制代码](javascript:void(0);)



在python中也可以使用 “int（）” 来申明一个整形的变量。如 score = int（100）

**float（浮点型）：**

浮点型由整数部分与小数部分组成，其实就是我们通常所说的小数型。浮点型也可以使用科学计数法表示（3.14e2 = 3.14 x 102）

浮点型的数据不是一个精确的值，Python 3.X对于浮点数默认的是提供17位数字的精度。

[复制代码](javascript:void(0);)

a = 33.0

print(type(a))

b = float(88)

print(b)

c = float('678.99')

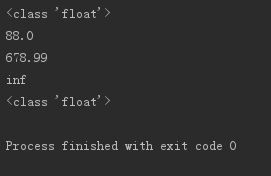
print(c)

d = float('Infinity')

print(d)

print(type(d))

[复制代码](javascript:void(0);)



**bool：**

　　python3里面将其定义成为整数， 在python2里面是没有bool型的， 我们通常使用1,0来表示。

[复制代码](javascript:void(0);)

Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)] on win32

>>> 2<1

False

>>> 1<2

True

>>> bool(1)

True

>>> bool(0)

False

>>> bool('a')

True

>>> bool(1<2)

True

[复制代码](javascript:void(0);)

**complex**：

　　是指数学意义上的复数，比如 2+7i。注意的是，该复数的实部和虚部都是浮点型的数据（float）

[复制代码](javascript:void(0);)

a = 1 + 2j

b = 2 + 3j

a+b

(3+5j)

complex(2,5)

(2+5j)

[复制代码](javascript:void(0);)

数字性的变量可以使用自身的构造函数进行相互转换，比如使用int(), float(), bool, complex()

[复制代码](javascript:void(0);)

>>> int(10.5)

10

>>> float(10)

10.0

>>> bool(1)

True

>>> bool(0)

False

>>> complex(1)

(1+0j)

>>> complex(1,2)

(1+2j)

>>> a = 10.5

>>> int(a)

10

>>> float(a)

10.5

>>> bool(a)

True

>>> complex(a)

(10.5+0j)

[复制代码](javascript:void(0);)

这些数字型的变量实际上指向的是一个对象，其本身会有很多的内置方法和属性。我们要学着多多使用。

# [Python 学习笔记4 变量-字符串](https://www.cnblogs.com/wanghao4023030/p/10646140.html)

Python中的字符串，我们可以简单的认为是一组用单引号，双引号，三引号包含的一组字符，数字或者特殊字符。在Python3中，所有的字符串都是Unicode字符串。

* 变量定义

[复制代码](javascript:void(0);)

#单引号

string1 = 'abc'

#双引号

string2 = "def"

#三引号，文字可以换行

string3 = ''' 我可以

随便

换行

'''

print(string1)

print(string2)

print(string3)

[复制代码](javascript:void(0);)

* 常规操作符

　　　　对与字符串我们可以进行拼接，截取，重复，判断，格式化等操作。python中常规的操作符有如下几种：

|  |  |
| --- | --- |
| **操作符** | **描述** |
| + | 字符串连接，将多个字符串连接成一个字符串 |
| \* | 重复输出字符串内容 |
| [] | 通过索引获取字符串中的部分内容 |
| [ : ] | 使用start：end的方式截取字符串中的一部分内容，遵循**左闭右开**原则，str[0,5] 是不包含第 5 个字符的，取前四个内容。 |
| in | 成员运算符 - 如果字符串中包含给定的字符返回 True |
| not in | 成员运算符 - 如果字符串中不包含给定的字符返回 True |
| r/R | 原始字符串 - 所有的字符串都是定义的值来显示，转译符和特殊字符不会被转译。 |
| % | 格式字符串，根据定义的格式输出字符串 |

　　字符串连接：使用"+"，将多个字符串连接成一个字符串.

[复制代码](javascript:void(0);)

string1 = 'abc'

string2 = "def"

string3 = '''gh

ijk

'''

print(string1 + string2 + string3)

#输出 abcdefgh

#ijk

[复制代码](javascript:void(0);)

 　　重复输出：重复输出字符串内容。

[复制代码](javascript:void(0);)

string1 = 'abc'

string2 = "def"

string3 = '''gh

ijk

'''

print(string1 \* 2)

#输出:abcabc

print(string1 \* len(string2))

#输出: abcabcabc

[复制代码](javascript:void(0);)

　　索引取值：使用[]字符，通过索引获取字符串中的部分内容。注意索引正向是从 0 开始的，如果取值超过索引范围会报系统错误。索引也可以是是负数，表示反向取值。

[复制代码](javascript:void(0);)

string1 = 'abc'

string2 = "def"

string3 = '''gh

ijk

'''

print(string1[2])

#输出:c

print(string1[-3])

#输出:a

print(string1[3])

#输出: IndexError : string index out of range

[复制代码](javascript:void(0);)

　　范围取值： 使用[start:end]的形式，取字符串的部分内容，遵循**左闭右开**原则，str[0,5] 是不包含第 5 个字符的，取前四个内容。还可以使用[start:end:step]的形式获取内容，意思是从start到end位置，每隔step个取一个字符。

[复制代码](javascript:void(0);)

string1 = 'abc'

string2 = "123456789"

string3 = '''gh

ijk

'''

print(string1[0:2])

#输出:ab

print(string1[0:10])

#输出:abc

print(string1[0:-2])

#输出: a , -2表示倒数第二个字符，即 'b'

print(string2[0:10:2])

#输出:13579

[复制代码](javascript:void(0);)

 　　成员运算符： 判断主字符串是否包含子字符串，返回True or False。

[复制代码](javascript:void(0);)

string1 = 'abc'

string2 = "123456789"

string3 = '''gh

ijk

'''

print('a' in string1)

#输出:True

print('d' not in string1)

#输出:True

print(string1 in string2)

#输出: False

print("d" in string2)

#输出:False

[复制代码](javascript:void(0);)

　　原始字符串： r/R + “” 用来显示原始定义的字符，转译符也将作为普通字符来显示。

[复制代码](javascript:void(0);)

string1 = 'abc'

string2 = "123456789"

string3 = '''gh

ijk

'''

string4 = r"display original chanter \n"

string5 = R"\t \\"

print(string4)

#输出:display original chanter \n

print(string5)

#输出:\t \\

[复制代码](javascript:void(0);)

　　格式字符串：使用%来格式化输出想要的字符类型：

|  |  |
| --- | --- |
| **符   号** | **描述** |
| %c | 格式化字符及其ASCII码 |
| %s | 格式化字符串 |
| %d | 格式化整数 |
| %u | 格式化无符号整型 |
| %o | 格式化无符号八进制数 |
| %x | 格式化无符号十六进制数 |
| %X | 格式化无符号十六进制数（大写） |
| %f | 格式化浮点数字，可指定小数点后的精度 |
| %e | 用科学计数法格式化浮点数 |
| %E | 作用同%e，用科学计数法格式化浮点数 |
| %g | %f和%e的简写 |
| %G | %f 和 %E 的简写 |
| %p | 用十六进制数格式化变量的地址 |

[复制代码](javascript:void(0);)

string1 = 'abc'

string2 = "123456789"

int\_num = 10

print("string4 is %s" % string1)

#输出:string4 is abc

print("string4 is %d" % int\_num)

#输出:string4 is abc

[复制代码](javascript:void(0);)

 　　我们也可以使用 format关键字进行格式化，注意点“.”不要遗漏了

name = 'ralf'

age = 35

print("name is {var\_name} and age is {var\_age}".format(var\_name = name, var\_age = age))

#输出: name is ralf and age is 35

[**Python 学习笔记5 变量-列表**](https://www.cnblogs.com/wanghao4023030/p/10659996.html)

列表是python常用的一种变量。 是由一些列按照特定顺序排列的元素组成的。你可以创建包含字母表中的所有字母，数字。可以将任何东西都加入到列表中。

通常情况下，列表中都包含多个元素，所以建议变量的名称以复数的形式来表示，比如 foods， names， cars。

在Python中，我们用方括号 [] 来表示列表，使用逗号来进行元素分隔：

names = ["Ralf", "Rachel", "clark"]

print(names)

# 输出： ['Ralf', 'Rachel', 'clark']

和字符串类型变量相类似，我们可以使用索引来访问list中的变量，索引从0 开始计数：

[复制代码](javascript:void(0);)

names = ['Ralf', 'Rachel', 'clark']

print(names[0])

print(names[2])

print(names[-1])

print(names[3])

# 输出： ralf

# 输出： clark

# 输出： clark

# 输出： IndexError: list index out of range

[复制代码](javascript:void(0);)

大多数定义的列表都是动态的，这意味着列表创建后，将随着程序的运行增减元素。

可以根据索引定位直接修改列表元素：

[复制代码](javascript:void(0);)

names = ['Ralf', 'Rachel', 'clark']

print(names[1])

names[1] = 'Terry'

print(names[1])

# 输出： Rachel

# 输出： Terry

[复制代码](javascript:void(0);)

添加元素到list中，我们可以使用list的内置函数来实现：

[复制代码](javascript:void(0);)

names = ['Ralf', 'Rachel', 'clark']

""" 在List末尾插入一个数值"""

names.append('Terry')

print(names)

# 输出： ['Ralf', 'Rachel', 'clark', 'Terry']

""" 在首位置插入一个数值"""

names.insert(0, 'LEONA')

print(names)

# 输出： ['LEONA', 'Ralf', 'Rachel', 'clark', 'Terry']

[复制代码](javascript:void(0);)

删除列表中的元素主要有以下几种方法： del，pop(), remove

[复制代码](javascript:void(0);)

names = ['Ralf', 'Rachel', 'Clark', 'Leona', 'Terry']

""" 删除List中的第二个值 Rachel"""

del names[1]

print(names)

# 输出： ['Ralf', 'Clark', 'Leona', 'Terry']

""" pop()会从列表中取出最后的一个值，并将该元素从list中删除"""

pop\_name = names.pop()

print(pop\_name)

print(names)

# 输出：

# Terry

# ['Ralf', 'Clark', 'Leona']

""" pop()也可以从指定位置处取值和删除元素"""

pop\_name = names.pop(2)

print(pop\_name)

print(names)

# 输出：

# Leona

# ['Ralf', 'Clark']

""" 使用remove()根据值来删除元素"""

names.remove('Clark')

print(names)

# 输出：['Ralf']

""" 注意remove()只会删除一个元素，如果list存在重复的值，那么只会删除第一个匹配的值"""

names2 = ['Ralf', 'Clark', 'Clark', 'Leona', 'Terry']

names2.remove('Clark')

print(names2)

# 输出：['Ralf', 'Clark', 'Leona', 'Terry']

[复制代码](javascript:void(0);)

列表排序，我们可以永久或者临时性的对list进行排序，使用sort和sorted方法：

[复制代码](javascript:void(0);)

names = ['Ralf', 'Rachel', 'Clark', 'Leona', 'Terry']

""" 使用sorted（）函数临时排序list中的元素"""

print(sorted(names))

print(sorted(names, reverse = True))

print(names)

#输出：

#['Clark', 'Leona', 'Rachel', 'Ralf', 'Terry']

#['Terry', 'Ralf', 'Rachel', 'Leona', 'Clark']

#['Ralf', 'Rachel', 'Clark', 'Leona', 'Terry']

"""使用list内置函数sort（）实现元素的排序的永久化"""

print(names)

names.sort()

print(names)

#输出：

#['Ralf', 'Rachel', 'Clark', 'Leona', 'Terry']

#['Clark', 'Leona', 'Rachel', 'Ralf', 'Terry']

[复制代码](javascript:void(0);)

确定列表的长度，我们可以使用len()函数来实现

[复制代码](javascript:void(0);)

names = ['Ralf', 'Rachel', 'Clark', 'Leona', 'Terry']

""" 使用len()函数来得到list中的元素的个数"""

print(len(names))

#输出： 5

[复制代码](javascript:void(0);)

 遍历整个列表，我们可以使用循环的形式遍历整个列表中的元素：

[复制代码](javascript:void(0);)

""" 使用For循环遍历整个List中的元素"""

for name in names:

print(name)

#输出： Ralf

#Rachel

#Clark

#Leona

#Terry

[复制代码](javascript:void(0);)

列表切片， List和string相类似，都支持索引和切片方式的取值。我们可以使用[start:end:step]的方式对列表中的元素进行读取操作。注意start从0开始，第end个元素不包含在操作中。

[复制代码](javascript:void(0);)

names = ['Ralf', 'Rachel', 'Clark', 'Leona', 'Terry']

"""取出list中下标1到2的元素"""

print(names[1:3])

#输出： ['Rachel', 'Clark']

"""取出list中下表从开始到3的元素"""

print(names[:4])

#输出： ['Ralf', 'Rachel', 'Clark', 'Leona']

"""从下标0开始每格两个取一次值"""

print(names[::2])

#输出： ['Ralf', 'Clark', 'Terry']

[复制代码](javascript:void(0);)

List列表也支持复数的取值，表示从后往前取值：

[复制代码](javascript:void(0);)

names = ['Ralf', 'Rachel', 'Clark', 'Leona', 'Terry']

"""取出list中倒数第三个到末尾的所有元素"""

print(names[-3:])

#输出： ['Clark', 'Leona', 'Terry']

"""从List末尾取出所有元素"""

print(names[::-1])

#输出： ['Terry', 'Leona', 'Clark', 'Rachel', 'Ralf']

"""取出从下标0开始到倒数第三个元素前的所有元素"""

print(names[:-3:])

#输出： ['Ralf', 'Rachel']

[复制代码](javascript:void(0);)

注意： list切片返回的结果任然是一个列表（List）。

列表的复制。我们尝试使用赋值的形式复制一张新的列表，发现其不能正确的工作：

[复制代码](javascript:void(0);)

names1 = ['Ralf', 'Rachel', 'Clark', 'Leona', 'Terry']

names2 = names1

names1.append('Heidern')

print(names1)

print(names2)

#输出：

['Ralf', 'Rachel', 'Clark', 'Leona', 'Terry', 'Heidern']

['Ralf', 'Rachel', 'Clark', 'Leona', 'Terry', 'Heidern']

[复制代码](javascript:void(0);)

我们发现，两个list发生了相同的变化。这是由变量特性决定， python是一种地址指向。names定义了变量的值后，这些数值存储到内存中，names作为变量，其指针指向了存储这些值得地址。而names2指向了names1指向的地址。因此两个实际上指向的存储地址是一样的。

我们如果想要复制列表，需要使用切片或者遍历的方式来实现：

[复制代码](javascript:void(0);)

names1 = ['Ralf', 'Rachel', 'Clark', 'Leona', 'Terry']

names2 = names1[:]

names1.append('Heidern')

print(names1)

print(names2)

#输出：

['Ralf', 'Rachel', 'Clark', 'Leona', 'Terry', 'Heidern']

['Ralf', 'Rachel', 'Clark', 'Leona', 'Terry']

[复制代码](javascript:void(0);)

[**Python 学习笔记6 变量-元组**](https://www.cnblogs.com/wanghao4023030/p/10679650.html)

我们在上一篇中了解了变量list（列表）， 今天我们来介绍下元组。元组是由括号和逗号，组织起来的一个元素的集合。和list不同的是，它其中的元素是不能被修改的，和其他语言中的常量相类似。

需要注意的是，如果元组中只包含一个元素，那么在元组的最后需要加一个逗号，来申明他是一个元组，而不是一个被运算符括号保护起来的一个整数。

[复制代码](javascript:void(0);)

names = ('Ralf', 'Rachel', 'Clark', 'Leona', 'Terry')

print(names)

numbers1 = (50)

print(type(numbers1))

numbers2 = (50,)

print(numbers2)

#输出：

#('Ralf', 'Rachel', 'Clark', 'Leona', 'Terry')

#<class 'int'>

#(50,)

[复制代码](javascript:void(0);)

 我们可以像列表list那样使用循环来访问和遍历元组：

[复制代码](javascript:void(0);)

names = ('Ralf', 'Rachel', 'Clark', 'Leona', 'Terry')

for name in names:

print(name)

#输出：

#Ralf

#Rachel

#Clark

#Leona

#Terry

[复制代码](javascript:void(0);)

也可以使用[:]切片的方式来访问元组：

[复制代码](javascript:void(0);)

names = ('Ralf', 'Rachel', 'Clark', 'Leona', 'Terry')

print(names[1])

#输出：Rachel

print(names[1:3])

#输出：('Rachel', 'Clark')

print(names[:-1])

#输出： ('Ralf', 'Rachel', 'Clark', 'Leona')

print(names[-1])

#输出： Terry

print(names[::2])

#输出： ('Ralf', 'Clark', 'Terry')

[复制代码](javascript:void(0);)

元组中的元素的值是不能被修改的， 否则会爆出系统错误：

names = ('Ralf', 'Rachel', 'Clark', 'Leona', 'Terry')

names[2] = 'NewMan'

#输出： TypeError: 'tuple' object does not support item assignment

但是我们可以对元组进行删除和重新定义：

[复制代码](javascript:void(0);)

names = ('Ralf', 'Rachel', 'Clark', 'Leona', 'Terry')

print(names)

names = ('Ralf', 'Rachel', 'Clark', 'Leona')

print(names)

del names

print(names)

#输出： ('Ralf', 'Rachel', 'Clark', 'Leona', 'Terry')

#输出： ('Ralf', 'Rachel', 'Clark', 'Leona')

#输出： NameError: name 'names' is not defined

[复制代码](javascript:void(0);)

多个元组可以进行合并，一个元组中也可以包含另一个元组：

[复制代码](javascript:void(0);)

names1 = ('Ralf', 'Rachel', 'Clark', 'Leona', 'Terry')

names2 = ('Ralf', 'Rachel', 'Clark', 'Leona')

names3 = names1 + names2

print(names3)

#输出： ('Ralf', 'Rachel', 'Clark', 'Leona', 'Terry', 'Ralf', 'Rachel', 'Clark', 'Leona')

names4 = (1, 2, ('a','b'))

print(names4)

print(names4[2])

print((names4[2])[0])

#输出： (1, 2, ('a', 'b'))

#输出： ('a', 'b')

#输出： a

[复制代码](javascript:void(0);)

由于元组的不可修改的相关特性，所以它不支持排序、替换、添加等操作。 只保留了计算元素个数，最大值，最小值等方法。

也正是因为这种特性，我们在做一些方法传递或者参数传递的时候，可以使用元组，保证传出去的数值，是不可以修改的，确保安全性。

[**Python 学习笔记7 变量-字典**](https://www.cnblogs.com/wanghao4023030/p/10686025.html)

字典是python中一个十分重要的变量，他是一个可变的容器对象。通过一组key（键）和value（值）对组成一个元素。

组成形式为{'key':'value', 'key':'value'}。整个字典使用大括号{}， 括起来， key和value使用冒号：进行分隔。

 键的值可以是数字、字符串、列表甚至是字典。

ralf = {'name': 'Ralf', 'sex': 'male', 'height': '188'}

print(type(ralf))

#输出 ： <class 'dict'>

通过键值来访问字典中的值：

[复制代码](javascript:void(0);)

ralf = {'name': 'Ralf', 'sex': 'male', 'height': '188'}

print(ralf['name'])

print(ralf['height'])

#输出 ：Ralf

#输出 ：188

[复制代码](javascript:void(0);)

添加键-值对：我们可以通过指定字典名，键和值来添加键值对。

[复制代码](javascript:void(0);)

ralf = {'name': 'Ralf', 'sex': 'male', 'height': '188'}

print(ralf)

ralf['hobby'] = 'knife'

print(ralf)

#输出 ：{'name': 'Ralf', 'sex': 'male', 'height': '188'}

#输出 ：{'name': 'Ralf', 'sex': 'male', 'height': '188', 'hobby': 'knife'}

[复制代码](javascript:void(0);)

修改字典中的键值对：

[复制代码](javascript:void(0);)

ralf = {'name': 'Ralf', 'sex': 'male', 'height': '188'}

print(ralf)

ralf['height'] = '186'

print(ralf)

#输出 ：{'name': 'Ralf', 'sex': 'male', 'height': '188'}

#输出 ：{'name': 'Ralf', 'sex': 'male', 'height': '186'}

[复制代码](javascript:void(0);)

删除键值对，可以使用del 函数进行删除：

[复制代码](javascript:void(0);)

ralf = {'name': 'Ralf', 'sex': 'male', 'height': '188'}

print(ralf)

del ralf['height']

print(ralf)

#输出 ：{'name': 'Ralf', 'sex': 'male', 'height': '188'}

#输出 ：{'name': 'Ralf', 'sex': 'male'}

[复制代码](javascript:void(0);)

 使用For循环遍历所有键值对：

[复制代码](javascript:void(0);)

ralf = {'name': 'Ralf', 'sex': 'male', 'height': '188'}

for key, value in ralf.items():

print(key + ":" + value)

#输出 ：name:Ralf

#sex:male

#height:188

[复制代码](javascript:void(0);)

遍历字典中的所有键：

[复制代码](javascript:void(0);)

ralf = {'name': 'Ralf', 'sex': 'male', 'height': '188'}

for key in ralf.keys():

print(key)

#输出 ：name

#sex

#height

[复制代码](javascript:void(0);)

遍历字典中的所有值：

[复制代码](javascript:void(0);)

ralf = {'name': 'Ralf', 'sex': 'male', 'height': '188'}

for value in ralf.values():

print(value)

#输出 ：Ralf

#male

#188

[复制代码](javascript:void(0);)

根据PEP8法则，我们在定义字典的时候通常使用以下格式：

ralf = {

'name': 'Ralf',

'sex': 'male',

'height': '188'

}

List和dict混合使用：

在list中包含字典：

[复制代码](javascript:void(0);)

ralf = {

'name': 'Ralf',

'sex': 'male',

}

clark = {

'name': 'Clark',

'sex': 'male',

}

leona = {

'name': 'Leona',

'sex': 'female',

}

team = [ralf, clark, leona]

print(type(team))

print(team)

#输出： <class 'list'>

#输出： [{'name': 'Ralf', 'sex': 'male'}, {'name': 'Clark', 'sex': 'male'}, {'name': 'Leona', 'sex': 'female'}]

for fighter in team:

for key, value in fighter.items():

print(key + ":" + value)

#输出：

#name:Ralf

#sex:male

#name:Clark

#sex:male

#name:Leona

#sex:female

[复制代码](javascript:void(0);)

在字典中存储列表：

[复制代码](javascript:void(0);)

team = {

'ralf': ['male', 188],

'clark': ['male', 187],

'leona': ['female', 174]

}

print(type(team))

print(team)

#输出： <class 'dict'>

#输出：{'ralf': ['male', 188], 'clark': ['male', 187], 'leona': ['female', 174]}

for key, value in team.items():

print(key)

for element in value:

print(element)

#输出：ralf

#male

#188

#clark

#male

#187

#leona

#female

#174

[复制代码](javascript:void(0);)

字典中嵌套字典：

[复制代码](javascript:void(0);)

team = {

'ralf': {'sex': 'male', 'height': '188'},

'clark': {'sex': 'male', 'height': '187'},

'leona': {'sex': 'female', 'height': '174'}

}

for keys, values in team.items():

print(keys)

for key, value in values.items():

print(key + ":" + value)

#输出：ralf

#sex:male

#height:188

#clark

#sex:male

#height:187

#leona

#sex:female

#height:174

[复制代码](javascript:void(0);)

[**Python 学习笔记8 条件语句 If**](https://www.cnblogs.com/wanghao4023030/p/10699090.html)

Python中条件语句if 是通过一条或者多条的执行语句的结果，来判断是否执行其包含的代码块。

通常会配合else、elif一起使用，达到根据条件进行多个代码块的执行操作。

简单的if

[复制代码](javascript:void(0);)

score = 90

if score >= 95:

print("优秀")

#没有输出

if 95 > score >= 80:

print("良")

#输出： 良

[复制代码](javascript:void(0);)

和else 配合使用：

[复制代码](javascript:void(0);)

score = 90

if score >= 60:

print("合格")

else:

print("不合格")

#输出： 合格

[复制代码](javascript:void(0);)

使用if -elif-else判断结构：

[复制代码](javascript:void(0);)

score = 80

if score < 60:

print("不合格")

elif 80 > score >= 60:

print("中")

else:

print("良")

#输出： 良

[复制代码](javascript:void(0);)

 在if语句中嵌套条件语句：

[复制代码](javascript:void(0);)

score = 80

if score >= 60:

print("通过")

if 80 > score >= 60:

print("中")

if 90 > score >= 80:

print("良")

if 100 >= score >= 90:

print("优")

else:

print("不合格")

#输出：通过

# 良

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

score = 80

if score < 60:

print("不合格")

elif score < 70:

print("合格")

elif score < 80:

print("中")

elif score < 90:

print("良")

else:

print("优")

#输出：良

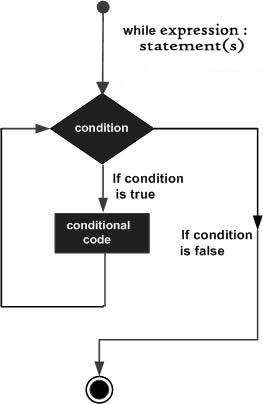
[复制代码](javascript:void(0);)

目前python3.0中没有 switch case结构，所以我们需要灵活的使用if-elif-else结构来进行条件判断。

[**Python 学习笔记9 循环语句 while**](https://www.cnblogs.com/wanghao4023030/p/10726478.html)

While循环是哟中利用条件语句，不断的执行某一段代码块，达到批量操作输出等一系列的操作，直到条件不满足或者被强制退出为止。

其工作流程如下： （图片来源菜鸟教程：<http://www.runoob.com/python/python-while-loop.html>  ）



我们来看一个例子：

[复制代码](javascript:void(0);)

current\_number = 10

while current\_number <= 20:

print("Current number is : " + str(current\_number))

current\_number += 1

print("Final number: " + str(current\_number))

'''

输出：

Current number is : 10

Current number is : 11

Current number is : 12

Current number is : 13

Current number is : 14

Current number is : 15

Current number is : 16

Current number is : 17

Current number is : 18

Current number is : 19

Current number is : 20

Final number: 21

'''

[复制代码](javascript:void(0);)

我们可以看到 变量current\_number 的初始值为10， 在小于等于20的情况下，不断的被打印，被增长，直至增长到21的时候，跳出了循环。

我们也可以使用标志，比如True,False 等布尔值来进行循环：

[复制代码](javascript:void(0);)

flag = True

while flag:

number = int(input("Input a number less than 10:\n"))

if number < 10:

flag = True

else:

flag = False

print("The game is end...")

'''

输出：

Input a number less than 10:

2

Input a number less than 10:

11

The game is end...

Final number: 21

'''

[复制代码](javascript:void(0);)

使用 break 退出循环。在while循环中，你可以通过特定的条件语句，选择终止循环，不再继续运行余下的代码。

[复制代码](javascript:void(0);)

flag = True

while flag:

city = input("Input a city which you like: \n" + "Input 'Q' to quit...\n")

if city == 'Q':

break

else:

print("You like " + city)

print("Thanks for your information.")

'''

输出：

Input a city which you like:

Input 'Q' to quit...

ShangHai

You like ShangHai

Input a city which you like:

Input 'Q' to quit...

HangZhou

You like HangZhou

Input a city which you like:

Input 'Q' to quit...

Q

Thanks for your information.

'''

[复制代码](javascript:void(0);)

使用 continue 不再继续执行余下的代码块，返回到循环开头，继续下一轮循环。

[复制代码](javascript:void(0);)

number = 0

while number <= 10:

if number % 2 == 0:

print("The number is even: " + str(number))

number += 1

continue

print("Next...")

else:

number += 1

'''

输出：

The number is even: 0

The number is even: 2

The number is even: 4

The number is even: 6

The number is even: 8

The number is even: 10

'''

[复制代码](javascript:void(0);)

 循环也可以和else 一起使用：

[复制代码](javascript:void(0);)

number = 0

while number <= 10:

if number % 2 == 0:

print("The number is even: " + str(number))

number += 1

continue

print("Next...")

else:

number += 1

else:

print("The number is equal or more than 10, stop loop.")

'''

输出：

The number is even: 0

The number is even: 2

The number is even: 4

The number is even: 6

The number is even: 8

The number is even: 10

The number is equal or more than 10, stop loop.

'''

[复制代码](javascript:void(0);)

使用while 操作列表 List

[复制代码](javascript:void(0);)

peoples = ['Ralf', 'Clark', 'Leon']

while peoples:

people = peoples.pop()

print(people)

'''

输出：

Leon

Clark

Ralf

'''

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

peoples = ['Ralf', 'Clark', 'Leon', 'Terry']

while 'Terry' in peoples:

peoples.remove('Terry')

print(peoples)

'''

输出：

['Ralf', 'Clark', 'Leon']

'''

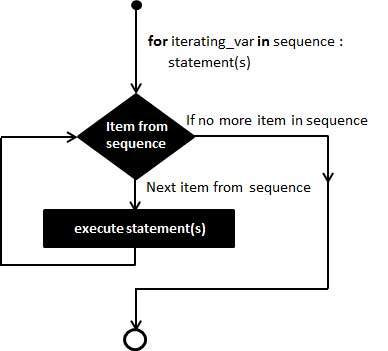
[复制代码](javascript:void(0);)

while循环语句可以解决程序中需要重复执行的操作。其循环执行的次数由循环条件确定，当循环条件满足时，重复执行某程序段，直到循环条件不成立为止。反复执行的程序段称为循环体，循环条件必须要在循环体中改变，否则可能会出现无限循环的结果

[**Python 学习笔记10 循环语句 for in**](https://www.cnblogs.com/wanghao4023030/p/10732381.html)

For in 循环主要适用于遍历一个对象中的所有元素。我们可以使用它遍历列表，元组和字典等等。

其主要的流程如下：（图片来源于： <https://www.yiibai.com/python/python_for_loop.html>）



 使用For遍历一个列表：

[复制代码](javascript:void(0);)

peoples = ['Ralf', 'Clark', 'Leon', 'Terry']

for people in peoples:

print(people)

'''

输出：

Ralf

Clark

Leon

Terry

'''

[复制代码](javascript:void(0);)

使用For in 遍历一个字典：

[复制代码](javascript:void(0);)

ralf = {'name': 'Ralf', 'sex': 'male', 'height': '188'}

for key, value in ralf.items():

print(key + ":" + value)

'''

输出：

name:Ralf

sex:male

height:188

'''

[复制代码](javascript:void(0);)

在For 循环中，我们可以使用 break， 在遇到特殊条件时，中断循环操作：

[复制代码](javascript:void(0);)

peoples = ['Ralf', 'Clark', 'Leon', 'Terry', 'Mary']

for people in peoples:

if people == 'Terry':

break

print(people)

'''

输出：

Ralf

Clark

Leon

'''

[复制代码](javascript:void(0);)

使用continue在for中继后继续下一轮的循环。

[复制代码](javascript:void(0);)

peoples = ['Ralf', 'Clark', 'Leon', 'Terry', 'Mary']

for people in peoples:

if people == 'Terry':

continue

print(people)

'''

输出：

Ralf

Clark

Leon

Mary

'''

[复制代码](javascript:void(0);)

For 循环中也可以使用else结构，当循环结束时执行特定语句，但是break中断时，else里面数据不会被执行：

[复制代码](javascript:void(0);)

peoples = ['Ralf', 'Clark', 'Leon', 'Terry', 'Mary']

for people in peoples:

print(people)

else:

print('Loop is end')

'''

输出：

Ralf

Clark

Leon

Terry

Mary

Loop is end

'''

peoples = ['Ralf', 'Clark', 'Leon', 'Terry', 'Mary']

for people in peoples:

if people == 'Terry':

break

print(people)

else:

print('Loop is end')

'''

输出：

Ralf

Clark

Leon

'''

[复制代码](javascript:void(0);)

[**Python 学习笔记11 函数**](https://www.cnblogs.com/wanghao4023030/p/10747072.html)

函数其实一段带名字的代码段，我们可以根据代码段，重复执行某一段代码段，或者有条件的执行某一段代码段。

将一段代码定义成函数后，我们可以很方便的根据自己的需求，随时调用该代码段。遇到需求变化的时候，只需要修改该函数，就可以满足需求，不需要到处修改特定的代码。

比如我们定义一个print hello的函数：

[复制代码](javascript:void(0);)

def say\_hello():

print("Hello!")

say\_hello()

'''

Hello!

'''

[复制代码](javascript:void(0);)

我们可以看到，我们使用def来定义函数， 以冒号结尾。 输入回车后，ide会自动缩进，缩进后的代码就是函数的定义体。定义完函数体后，我们通常空一行表示定义完函数。

我们直接输入函数名，即可调用函数。

我们还可以向函数传递“消息”，在编程语言中我们称之为参数。比如我们修改函数，可以向其传递一个人名，并输出想要语句。

[复制代码](javascript:void(0);)

def say\_hello(name):

print(name + ", Hello!")

say\_hello('Ralf')

'''

Ralf, Hello!

'''

[复制代码](javascript:void(0);)

通过上面代码，我们可以看到，我们重新修改了函数，增加了一个参数 name， 修改了函数体。这样我们可以根据自己的需要，向函数传递消息，并输出想要的结果。

函数定义中的参数，我们通常称之为 形参，比如上述例子中的变量name， 这个变量只在函数定义中使用，并不具有实际的值。我们在调用函数时，传递的消息或者说变量，我们称之为实参，是我们“实际“上想要使用的参数或者数值，变量。

通常情况下，我们必须按照事先定义好的参数，依次传递给函数体，才能正确得出想要的结果。

[复制代码](javascript:void(0);)

def say\_hello(name, city):

print(name + ", Welcome to " + city + "!" )

say\_hello('Ralf', "Shanghai")

'''

Ralf, Welcome to Shanghai!

'''

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

def say\_hello(name, city):

print(name + ", Welcome to " + city + "!" )

say\_hello('Shanghai', "Ralf")

'''

Shanghai, Welcome to Ralf!

'''

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

def say\_hello(name, city):

print(name + ", Welcome to " + city + "!" )

say\_hello('Ralf')

'''

Traceback (most recent call last):

File "D:/PythonStudy/test.py", line 31, in <module>

say\_hello('Ralf')

TypeError: say\_hello() missing 1 required positional argument: 'city'

'''

[复制代码](javascript:void(0);)

我们可以使用关键字的形式来规避上面例子中出现的错误，即在调用函数时，使用 形参 = value的方式来调用函数：

[复制代码](javascript:void(0);)

def say\_hello(name, city):

print(name + ", Welcome to " + city + "!" )

say\_hello(city = "Shanghai", name = 'Ralf')

'''  
输出：

Ralf, Welcome to Shanghai!

'''

[复制代码](javascript:void(0);)

在定义函数时，我们也可以定义一些参数默认值，这样在调用，如果给了实参，函数就使用实参，没有给值，就是用默认值。

[复制代码](javascript:void(0);)

def say\_hello(name, city = "Shanghai"):

print(name + ", Welcome to " + city + "!" )

# 使用默认值

say\_hello("Ralf")

#使用实际值

say\_hello("Rachel", "Beijing")

'''

输出：

Ralf, Welcome to Shanghai!

Rachel, Welcome to Beijing!

'''

[复制代码](javascript:void(0);)

返回值：

函数不仅可以接受外部传送的变量参数，也可以输出一个返回值给调用者。

[复制代码](javascript:void(0);)

def say\_hello(name, city = "Shanghai"):

return name + ", Welcome to " + city + "!"

out\_message = say\_hello("Rachel", "Beijing")

print(out\_message)

'''

输出：

Rachel, Welcome to Beijing!

'''

[复制代码](javascript:void(0);)

 返回一个列表：

[复制代码](javascript:void(0);)

def born\_city(name, city = "Shanghai"):

return [name, city]

out\_message = born\_city("Rachel", "Beijing")

print(type(out\_message))

print(out\_message)

'''

输出：

<class 'list'>

['Rachel', 'Beijing']

'''

[复制代码](javascript:void(0);)

返回一个字典：

[复制代码](javascript:void(0);)

def born\_city(name, city = "Shanghai"):

return {'name': name, 'city': city}

out\_message = born\_city("Rachel", "Beijing")

print(type(out\_message))

print(out\_message)

'''

输出：

<class 'dict'>

{'name': 'Rachel', 'city': 'Beijing'}

'''

[复制代码](javascript:void(0);)

传递任意数量的实参，有时候我们不确定，实际参数有几个，可以能是一个也可能是多个，要怎么定义哪。我们可以在形参前面加一个星号，表示这个参数可以是多个：

[复制代码](javascript:void(0);)

def say\_hello(\*names):

print(names)

out\_message = say\_hello("Rachel", "Ralf", "Terry")

print(out\_message)

'''

输出：

('Rachel', 'Ralf', 'Terry')

'''

[复制代码](javascript:void(0);)

注意函数返回了一个元组，即便函数只输入一个参数，返回的也是一个元组。

结合实参和任意数量的实参：

[复制代码](javascript:void(0);)

def say\_hello(words, \*names):

for name in names:

print(words + ", " + name)

out\_message = say\_hello("hello", "Rachel", "Ralf", "Terry")

print(out\_message)

'''

输出：

hello, Rachel

hello, Ralf

hello, Terry

'''

[复制代码](javascript:void(0);)

使用任意数量的关键字实参。有时候我们需要接受任意数量的实参，但是不知传递给函数的会是什么样的信息。在这种情况下，我们可以将函数编写成接受任意数量的 键值对，即字典的形式。方式为加两个冒号 \*\*dict：

[复制代码](javascript:void(0);)

def pepole\_info(name, sex, \*\*others):

print(name + ": "+ sex )

for key, value in others.items():

print(key + ": " + value)

out\_message = pepole\_info("Ralf", "male", height = "175", hobby = "football")

print(out\_message)

'''

输出：

Ralf: male

height: 175

hobby: football

'''

[复制代码](javascript:void(0);)

[**Python 学习笔记12 函数模块**](https://www.cnblogs.com/wanghao4023030/p/10793313.html)

　　函数的优点之一，使用它们可将代码块与主程序分离。通过给函数指定描述性的名称。可以让主程序非常好理解。但是如果将过多的函数和主程序放置在一起，会让文件显得非常凌乱。太多的代码混杂在一起，不方便管理。我们可以将相关的、类似的、一个对象的先关函数编写在一个py文件里。主程序根据需要通过导入这些文件，由选择的调用一些函数。这样既方便管理，也使主程序更加简洁明了。

　　通过将程序存储在独立的文件中，我们可以隐藏程序代码的相关细节，将注意力放在主程序的逻辑实现上。而且我们也可以在许多程序中重用这些函数。独立的文件，也方便我们以其他人员分享这些文件，而不是主程序。我们还可以使用其他程序员编写的模块简历函数库。

　　Python提供了许多种方式来导入模块。

　　首先我们要先创建模块。模块的扩展名是.py，其实就是一个python文件。我们在文件中建立各种函数，为其他程序调用做准备。比如我们建立一个叫greet的模块，一个打招呼的模块：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

'''

Define a module to greet others with parmater:

'''

def say\_hello(name):

str\_message = "hello, " + str(name) + "!"

return str\_message

def say\_bye(name):

str\_message = "Bye, " + str(name) + "!"

return str\_message

[复制代码](javascript:void(0);)

　　我们把它存为greet.py文件，以此同时我们在同一文件目录下新加一个主程序文件，就可以导入和使用模块了。方式有多种，我们来一一介绍下：

1. 导入整个模块

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

import greet

s\_message = greet.say\_hello('Ralf')

print(s\_message)

'''

输出：hello, Ralf!

'''

[复制代码](javascript:void(0);)

　　我们使用import + 模块名 直接导入整个模块。python会打开整个模块，并将整个函数代码复制到主程序中。我们并不会看到这些代码。但是实际上这些代码会在后台编译时运行。而我们在调用函数时，需要使用 【模块名称】.【函数名称】的形式来使用函数体。

　　我们也可以使用\*来一次性导入模块中的所有函数：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

from greet import \*

s\_message = say\_hello('Ralf')

print(s\_message)

s\_message = say\_bye('Ralf')

print(s\_message)

'''

输出：

hello, Ralf!

Bye, Ralf!

'''

[复制代码](javascript:void(0);)

　　使用这种方式，不需要指定模块名称，直接调用相关函数即可。

　　但是一次性导入整个模块，会让编译后的程序显得臃肿不堪，我们也可以根据需要，只导入模块中的部分函数给我们使用：

2. 导入部分函数：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

from greet import say\_hello

s\_message = say\_hello('Ralf')

print(s\_message)

'''

输出：hello, Ralf!

'''

[复制代码](javascript:void(0);)

　　我们可以看到我们只导入了模块 greet中的say\_hello()的函数，并且函数可以正确的运行。需要注意的是因为我们是显示的导入了函数，则不需要在函数名称前面再加模块的名称，直接调用函数即可。

　　我们也可以导入多个函数：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

from greet import say\_hello, say\_bye

s\_message = say\_hello('Ralf')

print(s\_message)

s\_message = say\_bye('Ralf')

print(s\_message)

'''

输出：

hello, Ralf!

Bye, Ralf!

'''

[复制代码](javascript:void(0);)

　　我们在导入模块和调用函数的时候，有时候会遇到模块名称或者函数名称和代码中的变量，函数名称相冲突，或者函数、模块名称过于复杂。我们可以将模块、函数名称用as指定别名，来区别。

1. 模块别名：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

import greet as gt

s\_message = gt.say\_hello('Ralf')

print(s\_message)

s\_message = gt.say\_bye('Ralf')

print(s\_message)

'''

输出：

hello, Ralf!

Bye, Ralf!

'''

[复制代码](javascript:void(0);)

2. 函数别名

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

from greet import say\_bye as bye, say\_hello as hello

s\_message = hello('Ralf')

print(s\_message)

s\_message = bye('Ralf')

print(s\_message)

'''

输出：

hello, Ralf!

Bye, Ralf!

'''

[复制代码](javascript:void(0);)

# [Python 学习笔记13 类 - 创建和简单使用](https://www.cnblogs.com/wanghao4023030/p/10887008.html)

###### 介绍：

面向对象编程是一种非常有效的软件编写方法之一，在面向对象编程中，我们会编写表示现实世界中的事物或者情景的类，并基于类来创建对象。

在编写类的的时候，这些类对象一般都有通用的行为或者属性。基于类创建对象时，每个对象都自动具备这些属性和行为，然后给予需要给与这些对象一些特殊的属性或者行为。

我们几乎可以使用面向对象的方式来模拟现实生活中的所有情景。

根据类来创建队形被称为实例化，这让你能够使用类的实例。

###### 创建类：

我们使用一个例子来说明怎么创建一个类。

我们假设创建一个名叫dog的类，我们知道狗狗一般会跑，叫，有些还会坐下，这些都是这个类的一些行为，我们也可以定义它的一些属性，比如名字和年龄。我们可以再代码中这样来实现这个类：

[复制代码](javascript:void(0);)

class Dog():

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def sit(self):

print(self.name + " is sitting.")

def run(self):

print(self.name + " is running.")

def bark(self):

print(self.name + " is barking.")

[复制代码](javascript:void(0);)

在上面代码中我们可以看到，我们定义可一个dog类，在python的约定中，我们一般用首字母大写来表示类。

在定义好类的名字后，我们看到Python使用\_\_init\_\_来初始化类对象，就像一个构造函数一样。我们可以在这个函数方法中来初始化和定义我们需要的一些变量（属性），比如表示自身的self，表示名字的name，表示年龄的age。这些属性一旦定义好后，可以在类的方法中使用。当类被实例化成对象后，对象自动拥有这些属性。

这个类还被定义了一些方法，比如run,sit和bark，在这些方法中我们也可以看到，他可以自动的调用name，age等参数，因为他们已经是类的属性。

###### 使用类：

根据类，我们可以创建和实例化一个对象。我们可以把dog类看做是一个对象的说明，而实例化的过程，实际就是根据说明创建一个对象的过程。

[复制代码](javascript:void(0);)

class Dog():

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def sit(self):

print(self.name + " is sitting.")

def run(self):

print(self.name + " is running.")

def bark(self):

print(self.name + " is barking.")

my\_dog = Dog('WanCai', 5 )

[复制代码](javascript:void(0);)

我们可以看到，我们使用code：my\_dog = Dog('WanCai', 5 ) 来实例化了一个叫wancai， 6岁的名叫my-dog的对象。

一旦实例化了一个对象，我们就可以调用这个对象的属性和方法：

[复制代码](javascript:void(0);)

my\_dog = Dog('WanCai', 5 )

print(type(my\_dog))

print("name is : " + my\_dog.name)

print("age is : " + str(my\_dog.age))

my\_dog.sit()

my\_dog.run()

my\_dog.bark()

'''

输出：

<class '\_\_main\_\_.Dog'>

name is : WanCai

age is : 5

WanCai is sitting.

WanCai is running.

WanCai is barking.

'''

[复制代码](javascript:void(0);)

一旦我们定义好了类，我们可以根据实际的需要来创建多个实例：

[复制代码](javascript:void(0);)

my\_dog = Dog('WangCai', 5 )

print(type(my\_dog))

print("name is : " + my\_dog.name)

print("age is : " + str(my\_dog.age))

my\_dog.sit()

my\_dog.run()

my\_dog.bark()

rachel\_dog = Dog('LaiFu', 5 )

print(type(rachel\_dog))

print("name is : " + rachel\_dog.name)

print("age is : " + str(rachel\_dog.age))

rachel\_dog.sit()

rachel\_dog.run()

rachel\_dog.bark()

'''

输出：

<class '\_\_main\_\_.Dog'>

name is : WangCai

age is : 5

WangCai is sitting.

WangCai is running.

WangCai is barking.

<class '\_\_main\_\_.Dog'>

name is : LaiFu

age is : 5

LaiFu is sitting.

LaiFu is running.

LaiFu is barking.

'''

[复制代码](javascript:void(0);)

# [Python 学习笔记14 类 - 使用类和实例](https://www.cnblogs.com/wanghao4023030/p/10890767.html)

当我们熟悉和掌握了怎么样创建类和实例以后，我们编程中的大多数工作都讲关注在类的简历和实例对象使用，修改和维护上。

结合实例我们来进一步的学习类和实例的使用：

我们新建一个汽车的类：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

class Car():

def \_\_init\_\_(self, make, model, year):

self.make = make

self.model = model

self.year = year

def get\_description\_name(self):

long\_name = str(self.year) + ' ' + self.make + ' ' + self.model

return long\_name.title()

my\_car = Car("audi", "A4", 2016)

print(my\_car.get\_description\_name())

'''

输出:

2016 Audi A4

'''

[复制代码](javascript:void(0);)

通过上面的例子我们可以看到，我们创建了一个car的类，这个类有三个参数，并且有一个打印所有信息的一个方法。

我们知道其实每个属性都有初始的值，有些时候我们的属性并不是为空或者0，他需要一个默认的其他值。我们可以在类的\_\_init\_\_方法中，对一些属性进行初始化。一旦属性具有初始值后，在实例化对象时，可以不提供数值。

我们在类中添加一个里程表的属性，他的初始值应该为0，我们添加一个方法用来读取这个属性。

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

class Car():

def \_\_init\_\_(self, make, model, year):

self.make = make

self.model = model

self.year = year

self.odometer\_reading = 0

def get\_description\_name(self):

long\_name = str(self.year) + ' ' + self.make + ' ' + self.model

return long\_name.title()

def read\_odometer(self):

print("This car has " + str(self.odometer\_reading) + " miles on it.")

my\_car = Car("audi", "A4", 2016)

print(my\_car.get\_description\_name())

my\_car.read\_odometer()

'''

输出:

2016 Audi A4

This car has 0 miles on it.

'''

[复制代码](javascript:void(0);)

我们可以直接修改实例的属性，来实现我么预想的操作：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

class Car():

def \_\_init\_\_(self, make, model, year):

self.make = make

self.model = model

self.year = year

self.odometer\_reading = 0

def get\_description\_name(self):

long\_name = str(self.year) + ' ' + self.make + ' ' + self.model

return long\_name.title()

def read\_odometer(self):

print("This car has " + str(self.odometer\_reading) + " miles on it.")

my\_car = Car("audi", "A4", 2016)

print(my\_car.get\_description\_name())

my\_car.read\_odometer()

my\_car.odometer\_reading = 100

my\_car.read\_odometer()

'''

输出:

2016 Audi A4

This car has 0 miles on it.

This car has 100 miles on it.

'''

[复制代码](javascript:void(0);)

也可以通过添加特定方法来实现对实例属性的修改：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

class Car():

def \_\_init\_\_(self, make, model, year):

self.make = make

self.model = model

self.year = year

self.odometer\_reading = 0

def get\_description\_name(self):

long\_name = str(self.year) + ' ' + self.make + ' ' + self.model

return long\_name.title()

def read\_odometer(self):

print("This car has " + str(self.odometer\_reading) + " miles on it.")

def update\_odometer(self, mileage):

self.odometer\_reading = mileage

my\_car = Car("audi", "A4", 2016)

print(my\_car.get\_description\_name())

my\_car.read\_odometer()

my\_car.update\_odometer(200)

my\_car.read\_odometer()

'''

输出:

2016 Audi A4

This car has 0 miles on it.

This car has 200 miles on it.

'''

[复制代码](javascript:void(0);)

我么知道里程数一般都是递增的，正常情况下我们是无法将其往回调整的，所以我们需要对update函数进一步的优化。输入的参数必须大于等于当前值，小于则必须报错：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

class Car():

def \_\_init\_\_(self, make, model, year):

self.make = make

self.model = model

self.year = year

self.odometer\_reading = 0

def get\_description\_name(self):

long\_name = str(self.year) + ' ' + self.make + ' ' + self.model

return long\_name.title()

def read\_odometer(self):

print("This car has " + str(self.odometer\_reading) + " miles on it.")

def update\_odometer(self, mileage):

if mileage >= self.odometer\_reading:

self.odometer\_reading = mileage

else:

print("You cannot do that.")

my\_car = Car("audi", "A4", 2016)

print(my\_car.get\_description\_name())

my\_car.read\_odometer()

my\_car.update\_odometer(200)

my\_car.read\_odometer()

my\_car.update\_odometer(100)

my\_car.read\_odometer()

'''

输出:

2016 Audi A4

This car has 0 miles on it.

This car has 200 miles on it.

You cannot do that.

This car has 200 miles on it.

'''

[复制代码](javascript:void(0);)

根据现实中的情况，其实我们定义一个增量方法更为合适，该方法每次都加上参数给定的值，并且该值应该大于等于0：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

class Car():

def \_\_init\_\_(self, make, model, year):

self.make = make

self.model = model

self.year = year

self.odometer\_reading = 0

def get\_description\_name(self):

long\_name = str(self.year) + ' ' + self.make + ' ' + self.model

return long\_name.title()

def read\_odometer(self):

print("This car has " + str(self.odometer\_reading) + " miles on it.")

def update\_odometer(self, mileage):

if mileage >= self.odometer\_reading:

self.odometer\_reading = mileage

else:

print("You cannot do that.")

def increase\_odometer(self, miles):

if miles >= 0:

self.odometer\_reading += miles

else:

print("The value is invalid, please input the number which should more than zero.")

my\_car = Car("audi", "A4", 2016)

print(my\_car.get\_description\_name())

my\_car.read\_odometer()

my\_car.update\_odometer(200)

my\_car.read\_odometer()

my\_car.increase\_odometer(100)

my\_car.read\_odometer()

my\_car.increase\_odometer(-1)

my\_car.read\_odometer()

'''

输出:

2016 Audi A4

This car has 0 miles on it.

This car has 200 miles on it.

This car has 300 miles on it.

The value is invalid, please input the number which should more than zero.

This car has 300 miles on it.

'''

[复制代码](javascript:void(0);)

# [Python 学习笔记15 类 - 继承](https://www.cnblogs.com/wanghao4023030/p/10896836.html)

我们在编程的过程中，并非都是要重头开始。比如其他人已经有现成的类，我们可以使用其他找人编写的类。术语称之为： 继承。

当一个类继承例外一个类时，它可以获得这个类的所有属性和方法：原有的类称之为 父类，新的类称之为子类。子类可以继承父类的所有方法和属性，还可以自定一些自己的方法和属性。

比如我们已经有了一个叫汽车的父类，我们可以继承这个类，生成一个电动车的子类：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

class Car():

def \_\_init\_\_(self, make, model, year):

self.make = make

self.model = model

self.year = year

self.odometer\_reading = 0

def get\_description\_name(self):

long\_name = str(self.year) + ' ' + self.make + ' ' + self.model

return long\_name.title()

def read\_odometer(self):

print("This car has " + str(self.odometer\_reading) + " miles on it.")

def update\_odometer(self, mileage):

if mileage >= self.odometer\_reading:

self.odometer\_reading = mileage

else:

print("You cannot do that.")

def increase\_odometer(self, miles):

if miles >= 0:

self.odometer\_reading += miles

else:

print("The value is invalid, please input the number which should more than zero.")

'''继承car，生成一个新类'''

class ElectricCar(Car):

def \_\_init\_\_(self, make, model, year):

super().\_\_init\_\_(make, model, year)

my\_BYD = ElectricCar("BYD", "Tang", 2019)

print(my\_BYD.get\_description\_name())

'''

输出:

2019 Byd Tang

'''

[复制代码](javascript:void(0);)

通过上面的代码，我们看到，我们基于一个car的父类，生成了一个ElectricCar的子类。

在类定义是，在括号里面包含父类的名称，来表示继承这个类： class NewClass（SupperClass）。

而真正继承父类的方法和属性的，则是在\_\_init\_\_方法中的super（）方法的使用，该方法告诉Python使用父类的\_\_init\_\_方法，来重新构造一个类。

通过上面的例子，我们可以看到，子类可以正确的调用父类的方法，实际上这时已经是子类的方法了。

我们也可以根据累的特性，给子类定义自己特有的属性和方法：

比如电动车有一个电瓶，并且有方法可以实时的显示当前的电量。

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

class Car():

def \_\_init\_\_(self, make, model, year):

self.make = make

self.model = model

self.year = year

self.odometer\_reading = 0

def get\_description\_name(self):

long\_name = str(self.year) + ' ' + self.make + ' ' + self.model

return long\_name.title()

def read\_odometer(self):

print("This car has " + str(self.odometer\_reading) + " miles on it.")

def update\_odometer(self, mileage):

if mileage >= self.odometer\_reading:

self.odometer\_reading = mileage

else:

print("You cannot do that.")

def increase\_odometer(self, miles):

if miles >= 0:

self.odometer\_reading += miles

else:

print("The value is invalid, please input the number which should more than zero.")

'''继承car，生成一个新类'''

class ElectricCar(Car):

def \_\_init\_\_(self, make, model, year):

super().\_\_init\_\_(make, model, year)

self.battery\_size = 100

def describe\_battery(self):

print("Catr has " + str(self.battery\_size) + "-kwh battery. " )

my\_BYD = ElectricCar("BYD", "Tang", 2019)

print(my\_BYD.get\_description\_name())

my\_BYD.describe\_battery()

'''

输出:

2019 Byd Tang

Catr has 100-kwh battery.

'''

[复制代码](javascript:void(0);)

在上述代码中，我们可以看到，我们在\_\_init\_\_方法中，添加了一个电瓶容量的属性，

self.battery\_size = 100

并且添加了一个电动车特有的显示电量的方法。

def describe\_battery(self):

print("Catr has " + str(self.battery\_size) + "-kwh battery. " )

这些方法是属于子类（ElectricCar）的，它能够正确的被运行。

当父类中的某些方法，并不适用子类的时候怎么办呐？我们可以在子类中重新定义该方法。

比如Car类中有加汽油的方法，而这对电动车并不适用，我们可以在子类中对这个方法进行覆盖重写。子类在调用这个方法时，将采用子类的定义：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

class Car():

def \_\_init\_\_(self, make, model, year):

self.make = make

self.model = model

self.year = year

self.odometer\_reading = 0

def get\_description\_name(self):

long\_name = str(self.year) + ' ' + self.make + ' ' + self.model

return long\_name.title()

def read\_odometer(self):

print("This car has " + str(self.odometer\_reading) + " miles on it.")

def update\_odometer(self, mileage):

if mileage >= self.odometer\_reading:

self.odometer\_reading = mileage

else:

print("You cannot do that.")

def increase\_odometer(self, miles):

if miles >= 0:

self.odometer\_reading += miles

else:

print("The value is invalid, please input the number which should more than zero.")

def fill\_gas(self):

print("Car is filling gas.")

'''继承car，生成一个新类'''

class ElectricCar(Car):

def \_\_init\_\_(self, make, model, year):

super().\_\_init\_\_(make, model, year)

self.battery\_size = 100

def describe\_battery(self):

print("Catr has " + str(self.battery\_size) + "-kwh battery. " )

def fill\_gas(self):

print("Electric car no gas tank.")

my\_BYD = ElectricCar("BYD", "Tang", 2019)

my\_BYD.fill\_gas()

'''

输出:

Electric car no gas tank.

'''

[复制代码](javascript:void(0);)

我们在编写代码时候，需要灵活的对类进行定义。在编程思想中，现实生活中的所有对象，都可以被定义成类。

我们尽可能多订一些类，以简化我们的代码长度，同时也变成程序代码的维护和修改。

比如在上述例子中，我们对电动车类增加了一个电池的属性和相关的方法。其实我们也可以新建一个电池的类，将电池特有的属性和方法独立开来。这样我们可以根据这个类生成各式各样的实例：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

class Car():

def \_\_init\_\_(self, make, model, year):

self.make = make

self.model = model

self.year = year

self.odometer\_reading = 0

def get\_description\_name(self):

long\_name = str(self.year) + ' ' + self.make + ' ' + self.model

return long\_name.title()

def read\_odometer(self):

print("This car has " + str(self.odometer\_reading) + " miles on it.")

def update\_odometer(self, mileage):

if mileage >= self.odometer\_reading:

self.odometer\_reading = mileage

else:

print("You cannot do that.")

def increase\_odometer(self, miles):

if miles >= 0:

self.odometer\_reading += miles

else:

print("The value is invalid, please input the number which should more than zero.")

def fill\_gas(self):

print("Car is filling gas.")

'''生成一个电池类'''

class Battery():

def \_\_init\_\_(self, size = 100):

self.size = size

def describe\_battery(self):

print("Battery has " + str(self.size) + "-kwh battery. " )

'''继承car，生成一个新类'''

class ElectricCar(Car):

def \_\_init\_\_(self, make, model, year):

super().\_\_init\_\_(make, model, year)

self.battery = Battery()

def fill\_gas(self):

print("Electric car no gas tank.")

my\_BYD = ElectricCar("BYD", "Tang", 2019)

my\_BYD.battery.describe\_battery()

'''

输出:

Battery has 100-kwh battery.

'''

[复制代码](javascript:void(0);)

我么可以看到我们增加了一个电池类Battery()，该类有自己属性 size和方法describe\_battery。我们在定义电动车时，增加了一个battery的属性，这个属性是一个baterry的实例，我们可以认为该属性实际上是一个对象 object，我们可以操作和使用它的属性和方法。

这样做的好处就是，有关电池的属性和方法的修改，可以放在battery类中进行处理。EelctricCar类中，只关注与其相关的属性和方法。比如我们可以添加一个电池能跑多少里程的方法，该方法与电池的容量相关：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

class Car():

def \_\_init\_\_(self, make, model, year):

self.make = make

self.model = model

self.year = year

self.odometer\_reading = 0

def get\_description\_name(self):

long\_name = str(self.year) + ' ' + self.make + ' ' + self.model

return long\_name.title()

def read\_odometer(self):

print("This car has " + str(self.odometer\_reading) + " miles on it.")

def update\_odometer(self, mileage):

if mileage >= self.odometer\_reading:

self.odometer\_reading = mileage

else:

print("You cannot do that.")

def increase\_odometer(self, miles):

if miles >= 0:

self.odometer\_reading += miles

else:

print("The value is invalid, please input the number which should more than zero.")

def fill\_gas(self):

print("Car is filling gas.")

'''生成一个电池类'''

class Battery():

def \_\_init\_\_(self, size = 100):

self.size = size

def describe\_battery(self):

print("Battery has " + str(self.size) + "-kwh battery. " )

def show\_range(self):

print("Battery has " + str(self.size \* 3) + " killmaters on full charge")

'''继承car，生成一个新类'''

class ElectricCar(Car):

def \_\_init\_\_(self, make, model, year):

super().\_\_init\_\_(make, model, year)

self.battery = Battery()

def fill\_gas(self):

print("Electric car no gas tank.")

my\_BYD = ElectricCar("BYD", "Tang", 2019)

my\_BYD.battery.describe\_battery()

my\_BYD.battery.show\_range()

my\_BYD.battery.size = 200

my\_BYD.battery.describe\_battery()

my\_BYD.battery.show\_range()

'''

输出:

Battery has 100-kwh battery.

Battery has 300 killmaters on full charge

Battery has 200-kwh battery.

Battery has 600 killmaters on full charge

'''

[复制代码](javascript:void(0);)

# [Python 学习笔记16 类 - 导入](https://www.cnblogs.com/wanghao4023030/p/11221069.html)

我们在编码的过程中，可能会给对象添加越来越多的功能，即使我们使用了继承，也不可避免的使文件越来越臃肿。

为了避免这种情况， Python允许将对象存储在模块中，并且可以在其他模块中进行导入。

其实这和C#中的命名空间相类似。

我们首先准备了一个叫car.py的模块，其中包含了多个对象：

[复制代码](javascript:void(0);)

class Car():

def \_\_init\_\_(self, make, model, year):

self.make = make

self.model = model

self.year = year

self.odometer\_reading = 0

def get\_description\_name(self):

long\_name = str(self.year) + ' ' + self.make + ' ' + self.model

return long\_name.title()

def read\_odometer(self):

print("This car has " + str(self.odometer\_reading) + " miles on it.")

def update\_odometer(self, mileage):

if mileage >= self.odometer\_reading:

self.odometer\_reading = mileage

else:

print("You cannot do that.")

def increase\_odometer(self, miles):

if miles >= 0:

self.odometer\_reading += miles

else:

print("The value is invalid, please input the number which should more than zero.")

def fill\_gas(self):

print("Car is filling gas.")

'''生成一个电池类'''

class Battery():

def \_\_init\_\_(self, size = 100):

self.size = size

def describe\_battery(self):

print("Battery has " + str(self.size) + "-kwh battery. " )

def show\_range(self):

print("Battery has " + str(self.size \* 3) + " killmaters on full charge")

'''继承car，生成一个新类'''

class ElectricCar(Car):

def \_\_init\_\_(self, make, model, year):

super().\_\_init\_\_(make, model, year)

self.battery = Battery()

def fill\_gas(self):

print("Electric car no gas tank.")

[复制代码](javascript:void(0);)

接下来我们新建一个my\_car的模块，并导入Car类并创建实例，并且调用一些方法：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

from car import Car

my\_new\_car = Car('Volvo', 'V60', '2020')

print(my\_new\_car.get\_description\_name())

my\_new\_car.odometer\_reading = 200

my\_new\_car.read\_odometer()

'''

输出：

2020 Volvo V60

This car has 200 miles on it.

'''

[复制代码](javascript:void(0);)

同样，我们也可以在一个模块里面存储、导入多个类，进行操作：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

from car import Car, ElectricCar

my\_new\_car = Car('Volvo', 'V60', '2020')

print(my\_new\_car.get\_description\_name())

my\_new\_car.odometer\_reading = 200

my\_new\_car.read\_odometer()

my\_byd\_tang = ElectricCar('BYD', 'Tang', '2020')

my\_byd\_tang.battery.show\_range()

my\_byd\_tang.fill\_gas()

'''

输出：

2020 Volvo V60

This car has 200 miles on it.

Battery has 300 killmaters on full charge

Electric car no gas tank.

'''

[复制代码](javascript:void(0);)

当然你可以导入模块中的所有的类：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

from car import \*

my\_new\_car = Car('Volvo', 'V60', '2020')

print(my\_new\_car.get\_description\_name())

my\_new\_car.odometer\_reading = 200

my\_new\_car.read\_odometer()

my\_byd\_tang = ElectricCar('BYD', 'Tang', '2020')

my\_byd\_tang.battery.show\_range()

my\_byd\_tang.fill\_gas()

my\_battery = Battery()

my\_battery.show\_range()

'''

输出：

2020 Volvo V60

This car has 200 miles on it.

Battery has 300 killmaters on full charge

Electric car no gas tank.

Battery has 300 killmaters on full charge

'''

[复制代码](javascript:void(0);)

也可以直接导入整个模块，不过这种方式需要在类的名字前加上模块名：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

import car

my\_new\_car = car.Car('Volvo', 'V60', '2020')

print(my\_new\_car.get\_description\_name())

my\_new\_car.odometer\_reading = 200

my\_new\_car.read\_odometer()

my\_byd\_tang = car.ElectricCar('BYD', 'Tang', '2020')

my\_byd\_tang.battery.show\_range()

my\_byd\_tang.fill\_gas()

my\_battery = car.Battery()

my\_battery.show\_range()

'''

输出：

2020 Volvo V60

This car has 200 miles on it.

Battery has 300 killmaters on full charge

Electric car no gas tank.

Battery has 300 killmaters on full charge

'''

[复制代码](javascript:void(0);)

我们在实际的编码过程中，还是比较推荐一个类只定义在单一的模块中，这样模块的大小不会太大，也方便其他模块进行调用。

也方便进行维护。

# [Python 学习笔记17 文本 - 读写](https://www.cnblogs.com/wanghao4023030/p/11221385.html)

在我们的编程过程中，我们经常需要对文件进行读写操作。

在Python中，对文本的读写非常的方便，只需要简单的几行代码就可以实现。

我们首先新建一个文本文件"Text.txt", 里面有很多的内容，我们尝试将其全部读出来

[复制代码](javascript:void(0);)

with open('Text.txt') as file\_object:

contents = file\_object.read()

print(contents)

'''

输出:

查询出系统中最慢的SQL语句

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT TOP 20

CAST(qs.total\_elapsed\_time / 1000000.0 AS DECIMAL(28, 2))

AS [Total Elapsed Duration(s)]

, qs.execution\_count

, SUBSTRING (qt.text, (qs.statement\_start\_offset/2) + 1,

.....

.....

'''

[复制代码](javascript:void(0);)

我们看到，使用open的方式，我们可以打开一个文本对象，并通过read函数，将文本中的内容全部读出来了。和其他的语言不同的是，代码中我们没有使用close将文本关闭。

Python会自动进行管理，在文本不再使用的时候或者合适的时候将其关闭，当然你也可以人工将其关闭，以节约资源。

with open('Text.txt') as file\_object:

contents = file\_object.read()

print(contents)

file\_object.close()

逐行读取文本：

[复制代码](javascript:void(0);)

with open('Text.txt') as file\_object:

for line in file\_object:

print(line)

file\_object.close()

'''

输出:

查询出系统中最慢的SQL语句

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT TOP 20

CAST(qs.total\_elapsed\_time / 1000000.0 AS DECIMAL(28, 2))

AS [Total Elapsed Duration(s)]

, qs.execution\_count

, SUBSTRING (qt.text, (qs.statement\_start\_offset/2) + 1,

.....

.....

'''

[复制代码](javascript:void(0);)

使用列表存储文本中的内容，我们可以使用readlines将文本按行读出，并赋值给一个列表，为其他代码使用：

[复制代码](javascript:void(0);)

with open('Text.txt') as file\_object:

lineslist = file\_object.readlines()

file\_object.close()

print(len(lineslist))

print(lineslist[1])

'''

输出:

187

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

'''

[复制代码](javascript:void(0);)

通过以上的方式我们可以简单的对文件进行读操作。

接下来我们可以尝试写一个文本：

[复制代码](javascript:void(0);)

with open('write.txt', 'w') as file\_object:

file\_object.write('hello world!')

file\_object.close()

'''

输出:

hello world!

'''

[复制代码](javascript:void(0);)

上述代码中，我们尝试对一个python相同目录下的名叫 ‘write.txt’的文本中写入一段文字 ‘hello world!’。如果该文件不存在，python会默认新建一个。

我们也可以向文件中写入多行的文件，注意使用换行符：

[复制代码](javascript:void(0);)

with open('write.txt', 'w') as file\_object:

file\_object.write('hello world!\n')

file\_object.write('hello ralf!\n')

file\_object.write('hello rachel!\n')

file\_object.write('hello wowo!\n')

file\_object.close()

'''

输出:

hello world!

hello ralf!

hello rachel!

hello wowo!

'''

[复制代码](javascript:void(0);)

需要注意的是使用 'w'模式，会对文本进行覆盖写入，所以我们可以使用 ‘a’附加模式，对文件的内容进行附加：

[复制代码](javascript:void(0);)

with open('write.txt', 'w') as file\_object:

file\_object.write('hello world!\n')

file\_object.write('hello ralf!\n')

file\_object.write('hello rachel!\n')

file\_object.write('hello wowo!\n')

file\_object.close()

with open('write.txt', 'a') as file\_object:

file\_object.write('hello Python!\n')

file\_object.close()

'''

输出:

hello world!

hello ralf!

hello rachel!

hello wowo!

hello Python!

'''

[复制代码](javascript:void(0);)

通过上述的代码我们可以看出，实际上我们对文本的操作就是通过 open创建一个文本对象，根据不同的模式选择读，写，附加等相关的操作：

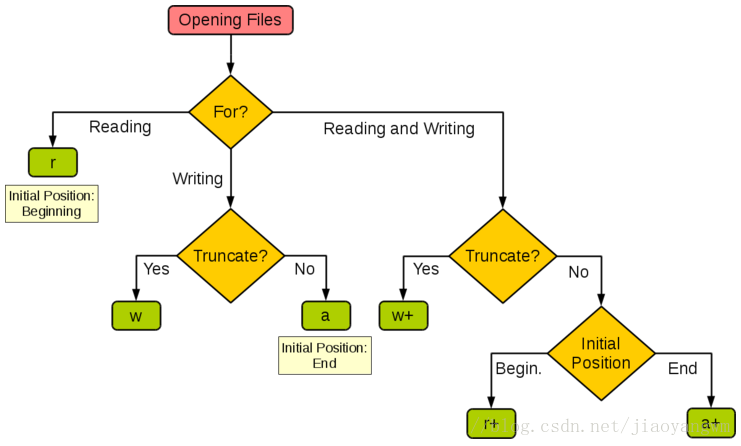
file object = open(file\_name [, access\_mode][, buffering])

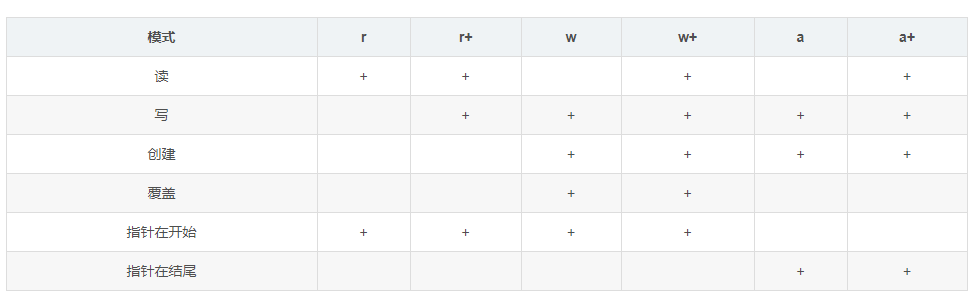
file\_name：file\_name变量是一个包含了你要访问的文件名称的字符串值。  
access\_mode：access\_mode决定了打开文件的模式：只读，写入，追加等。所有可取值见如下的完全列表。这个参数是非强制的，默认文件访问模式为只读(r)。  
buffering:如果buffering的值被设为0，就不会有寄存。如果buffering的值取1，访问文件时会寄存行。如果将buffering的值设为大于1的整数，表明了这就是的寄存区的缓冲大小。如果取负值，寄存区的缓冲大小则为系统默认。

其他的相关的参数和流程如下：

以下内容来自于：  
作者：呆呆的猫  
来源：CSDN  
原文：https://blog.csdn.net/jiaoyangwm/article/details/79635271  
版权声明：本文为博主原创文章，转载请附上博文链接！

* 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。一般用于非文本文件如图片等。
* 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。一般用于非文本文件如图片等。
* 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。一般用于非文本文件如图片等。
* 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。一般用于非文本文件如图片等。
* 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。





# [Python 学习笔记18 异常处理](https://www.cnblogs.com/wanghao4023030/p/11221801.html)

我们在编码的过程中，难免会遇到一些错误和异常， 这时候程序会异常退出，并且会抛出错误信息：

比如：

[复制代码](javascript:void(0);)

print(1/0)

'''

输出：

Traceback (most recent call last):

File "D:/PythonStudy/errors.py", line 3, in <module>

print(1/0)

ZeroDivisionError: division by zero

'''

[复制代码](javascript:void(0);)

我们尝试让1 去除0，结果系统报错了异常  ‘ZeroDivisionError: division by zero’， 表示不能被零除。

Python内置许多异常，当出现这些问题时，会提示我们并中断运行，python3.7中主要有以下的异常：

BaseException

+-- SystemExit

+-- KeyboardInterrupt

+-- GeneratorExit

+-- Exception

+-- StopIteration

+-- StopAsyncIteration

+-- ArithmeticError

| +-- FloatingPointError

| +-- OverflowError

| +-- ZeroDivisionError

+-- AssertionError

+-- AttributeError

+-- BufferError

+-- EOFError

+-- ImportError

| +-- ModuleNotFoundError

+-- LookupError

| +-- IndexError

| +-- KeyError

+-- MemoryError

+-- NameError

| +-- UnboundLocalError

+-- OSError

| +-- BlockingIOError

| +-- ChildProcessError

| +-- ConnectionError

| | +-- BrokenPipeError

| | +-- ConnectionAbortedError

| | +-- ConnectionRefusedError

| | +-- ConnectionResetError

| +-- FileExistsError

| +-- FileNotFoundError

| +-- InterruptedError

| +-- IsADirectoryError

| +-- NotADirectoryError

| +-- PermissionError

| +-- ProcessLookupError

| +-- TimeoutError

+-- ReferenceError

+-- RuntimeError

| +-- NotImplementedError

| +-- RecursionError

+-- SyntaxError

| +-- IndentationError

| +-- TabError

+-- SystemError

+-- TypeError

+-- ValueError

| +-- UnicodeError

| +-- UnicodeDecodeError

| +-- UnicodeEncodeError

| +-- UnicodeTranslateError

+-- Warning

+-- DeprecationWarning

+-- PendingDeprecationWarning

+-- RuntimeWarning

+-- SyntaxWarning

+-- UserWarning

+-- FutureWarning

+-- ImportWarning

+-- UnicodeWarning

+-- BytesWarning

+-- ResourceWarning

 有关内置异常的详细描述可以参考python的相关文档，链接如下： <https://docs.python.org/zh-cn/3/library/exceptions.html#bltin-exceptions>

Python提供了一套方式，用来捕获和处理错误和异常：

使用try....except...finally来对可能存在的错误进行处理：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

try:

print(1/0)

except ZeroDivisionError:

print('Can not division 0, happen ZeroDivisionError')

finally:

print('Go to next step')

'''

输出：

Can not division 0, happen ZeroDivisionError

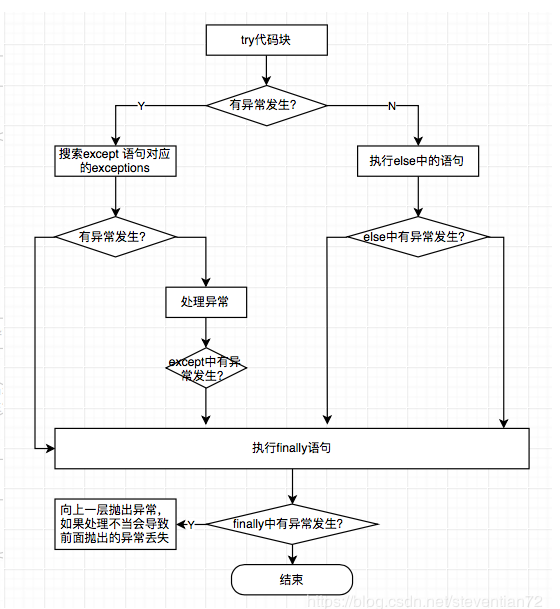
Go to next step

'''

[复制代码](javascript:void(0);)

通过上面的例子，我们可以看到，代码首先执行try里的code，如果遇到了指定的错误，那么就执行except中的代码， 处理完毕后执行finally中的代码。

在每个代码块中，支持各种判断和迭代，其工作的流程结构如下：



 除了系统内置的错误，我们也可以在代码中主动抛出一些错误：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

name = 'ralf'

if name != 'rachel':

raise Exception('The name is not ralf')

'''

输出：

Traceback (most recent call last):

File "D:/PythonStudy/errors.py", line 4, in <module>

raise Exception('The name is not ralf')

Exception: The name is not ralf

'''

[复制代码](javascript:void(0);)

我们也可以通过继承Exception base class，自定我们自己的错误类型：

[复制代码](javascript:void(0);)

#-\*- coding:utf-8 -\*-

class NameError(Exception):

def \_\_init\_\_(self, level, message):

self.level = level

self.message = message

try:

name = 'ralf'

if name != 'rachel':

raise NameError('Error', 'Name is not right')

except NameError as e:

print(e.level)

print(e.message)

'''

输出：

Error

Name is not right

'''

[复制代码](javascript:void(0);)

# [Python 学习笔记19 安装robot Framework](https://www.cnblogs.com/wanghao4023030/p/11366649.html)

因为项目组要做自动化测试，本人其实很希望能够使用 MStest + unit + C#来实现。

毕竟产品是基于.net 环境，并且使用C#环境开发的，适用性比较好，一些开发代码可以复用。

但是领导基于组内人员的开发水平，还是建议使用python+ robotframework的方式，比较方便以后代码的维护和开发。

于是，搜索如何在python环境下安装robot framework框架。返现网上很多都是基于2.7老版本的，或者步骤比较繁琐。

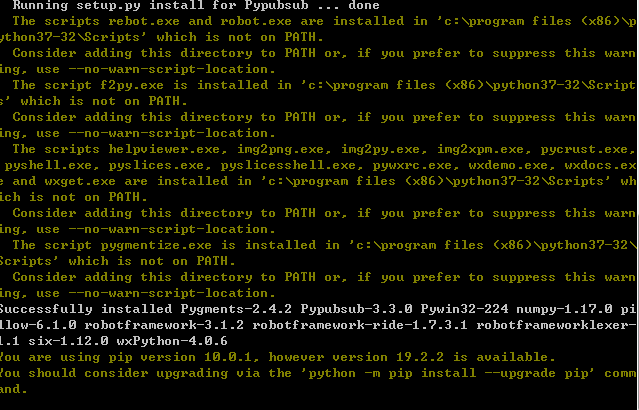
研究下来，其实没有那么复杂:

首先我的环境是python3.7, IDE 是pycharm社区版，已经安装好了PIP。

实际上我们只要打开cmd窗口或者IDE， 直接安装robot-ride就可以了，其相关的依赖组件会自动被安装好：

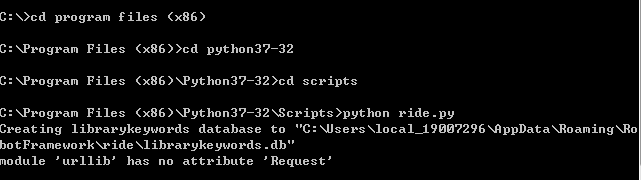
打开CMD窗口输入：

pip install robot-framework-ride



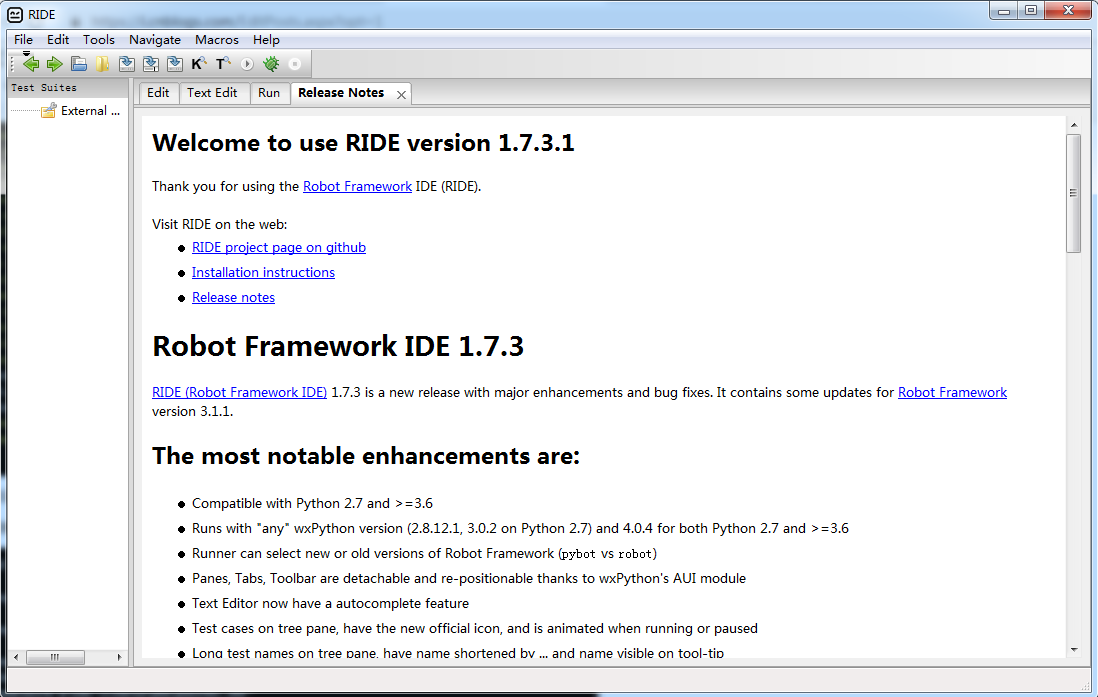
因为我没有使用pycharm的环境，安装过程中给出了一些环境变量错误的信息，可以直接忽略。

我们直接进入到给出的地址，直接运行ride.py即可：



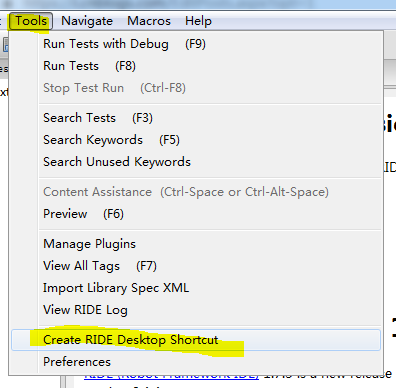
运行命令： python ride.py

ride界面就打开了：

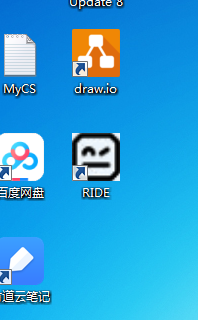


目前ride的最新版本是1.7.3.1版本。

我们tools的目录下面，可以找到建立桌面快捷方式的方式，直接选择即可：



我们可以看到桌面上就有快捷方式了：



ride安装好后，我们可能还需要一些第三方的库来支持我们进行case的设计。

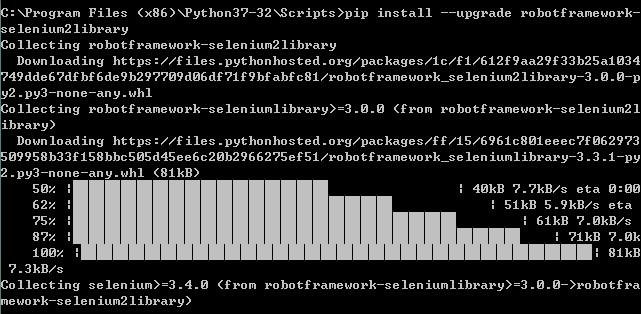
我们可以先取robotframework的官方网站上来浏览和了解有哪些已经开发好的第三方的库：

<https://robotframework.org/>

然后我们可以根据需要直接使用PIP进行安装，比如我们安装一个seleniun2Library:

输入命令：

pip install --upgrade robotframework-selenium2library



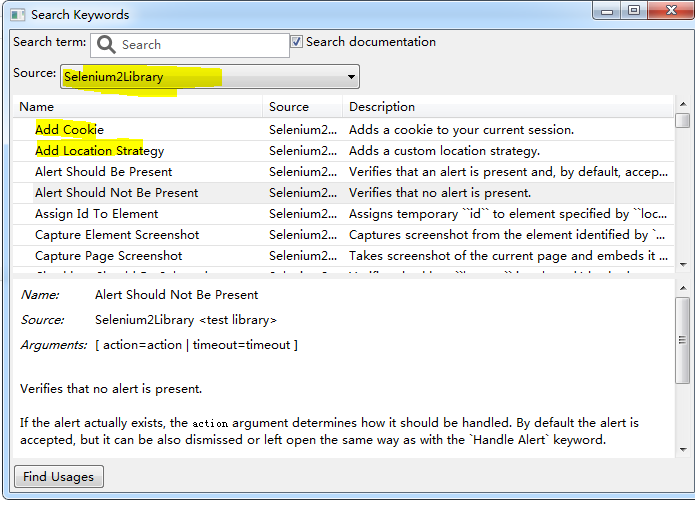
系统就会自动开始安装了。

安装完毕后，我们重新打开ride，然后新建一个project，再增加一个test suite或者testcase。

点击新建的project， 点击右侧的‘library’， 在弹出界面中输入 Selenium2Library,并确认。

如果新加入的library的字体是黑色的而不是红色的，就表示rid找到了这个库，可以正常使用了。

按F5或者右键选择project，选择search keywords， 在弹出的界面中，选择selenium2library,就可以看到所有selenium库中的关键字了。



到这里，robot的安装和库的导入我们就介绍完了，后续的内容会接续介绍有关python和robot的使用情况。

# [Python 学习笔记20 自定义robot Framework 关键字](https://www.cnblogs.com/wanghao4023030/p/11417438.html)

Robot framework 自定义了一些关键字我们可以把他们当作函数在设计测试用例的时候使用。

同时RF也提供了许多第三方的库，我们可以自己下载使用。

当时我们在实际工作中，会编写一些自己的函数来设计测试用例。

自定义关键的方法有两种：

## 

## 方法一：

如果你编写的python模块没有使用面向对象的方法，那么直接在界面中选择对应的python文件即可：

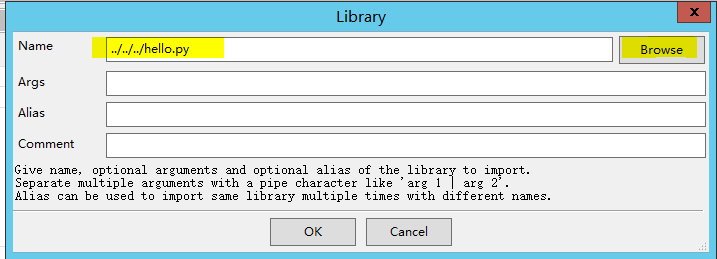
 我在D盘下，新建了一个hello.py文件，里面代码如下：

# -\*- coding : utf-8 -\*-

def hello\_world():

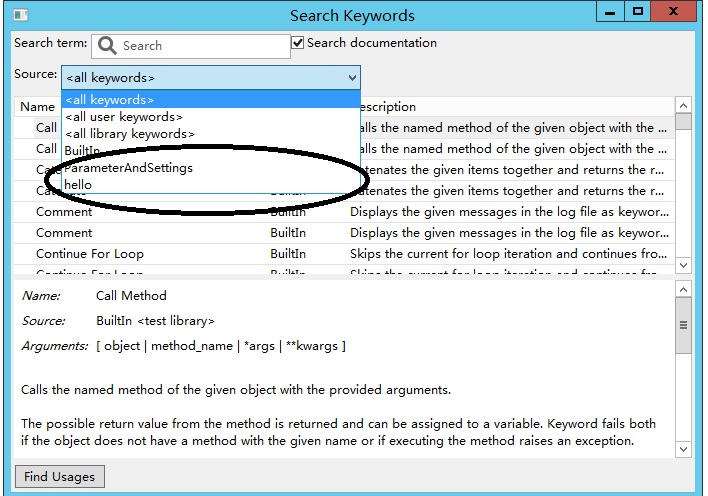
print("hello, world!")

我们打开robot framework， 选择library，在弹出的窗口中，直接选刚才的文件即可：

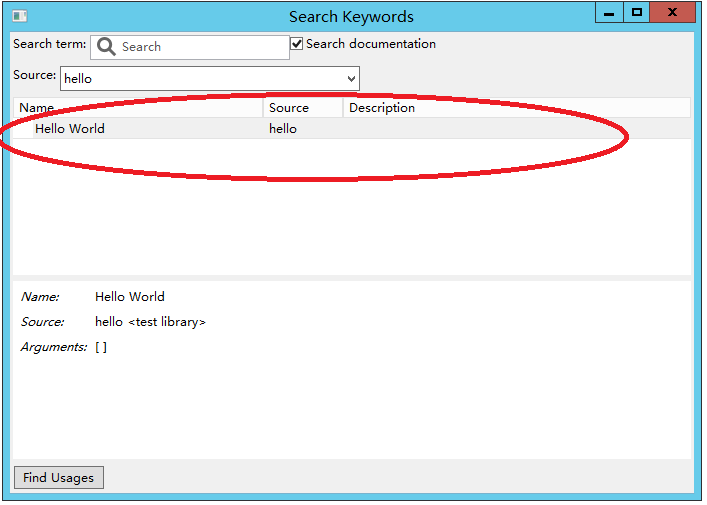


如果库文件导入正确， 库的名字是黑色的，如果是红色的就表示，有错误。

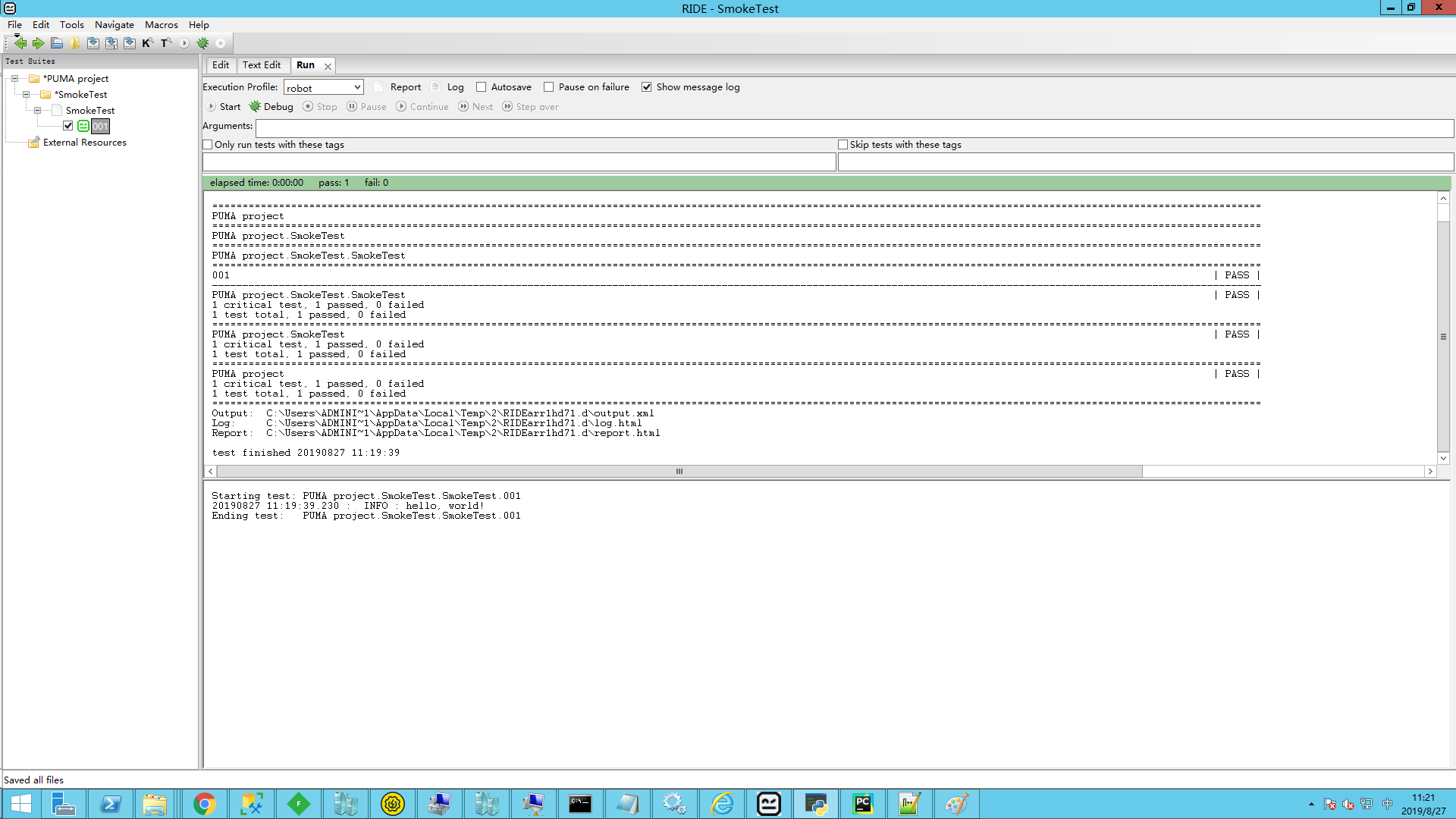
导入成功后，我们可以按下F5键，在弹出的库关键字查询中我啊们就可以看到我们刚才导入的库：



选择好source hello，我们会看到刚才文件中的函数变成了自定义关键字：



我们尝试在测试用例中，使用这个自定义关键字，发现可以正常使用：



## 方法二

 如果我们的python代码中使用了面向对象方法，那么我们就不能使用方法一了，因为RF不能调用对象，我们则需要其他方法来实现。

首先我们要进入到Python的运行目录，找到site-packages目录下，这个目录下存放的我们下载的一些python包。

1. 我们新建一个文件夹名叫： ‘ParameterAndSettings’， 这个将来会作为library进行使用。

2. 我们在该目录下新建一个python文件， Settings.py

代码如下：

[复制代码](javascript:void(0);)

import json

import os

from collections import namedtuple

import random

from datetime import datetime, timedelta

class Settings(object):

def \_\_init\_\_(self):

file\_path = 'C:\Program Files (x86)\Python37\Lib\site-packages\ParameterAndSettings\configuration.json'

try:

if os.path.isfile(file\_path):

with open(file\_path, 'r') as fileobject:

data = fileobject.read()

try:

json\_obj = json.loads(data, object\_hook=lambda d: namedtuple('json\_obj', d.keys())(\*d.values()))

except Exception as e:

raise Exception("Settings: Load the configuration to Json failed.\n " )

raise Exception("Settings: File is: %s" % (file\_path))

self.server = json\_obj.server

self.modality = eval(json\_obj.modality)

self.bodypart = eval(json\_obj.bodypart)

self.grender = eval(json\_obj.grender)

self.notifyserver\_exam\_body\_content = json\_obj.notifyserver\_exam\_body\_content

self.watermark\_path = json\_obj.watermark\_path

self.db\_connectString = json\_obj.db\_connectString

self.db\_driver = json\_obj.db\_driver

self.db\_server = json\_obj.db\_server

self.db\_default\_database = json\_obj.db\_default\_database

self.db\_uid = json\_obj.db\_uid

self.db\_pwd = json\_obj.db\_pwd

self.report\_template\_file = json\_obj.report\_template\_file

self.report\_file = json\_obj.report\_file

self.report\_default\_printer = json\_obj.report\_default\_printer

self.EHDPS\_status\_url = json\_obj.EHDPS\_status\_url

self.EHDPS\_printtask\_create\_url = json\_obj.EHDPS\_printtask\_create\_url

self.EHDPS\_printtask\_print\_url = json\_obj.EHDPS\_printtask\_print\_url

self.EHDPS\_printtask\_report\_getinfo\_url = json\_obj.EHDPS\_printtask\_report\_getinfo\_url

self.EHDPS\_printtask\_report\_print\_url = json\_obj.EHDPS\_printtask\_report\_print\_url

self.EHDPS\_printtask\_status\_url = json\_obj.EHDPS\_printtask\_status\_url

self.EHDPS\_printtask\_status\_dict = eval(json\_obj.EHDPS\_printtask\_status\_dict)

self.EHDUS\_upload\_report\_upload\_url = json\_obj.EHDUS\_upload\_report\_upload\_url

self.Printmode\_dict\_mode\_value = eval(json\_obj.Printmode\_dict\_mode\_value)

self.Printmode\_dict\_value\_mode = eval(json\_obj.Printmode\_dict\_value\_mode)

self.Integration\_URL = json\_obj.Integration\_URL

self.Notify\_URL = json\_obj.Notify\_URL

self.Holdflag\_dict\_mode\_value = eval(json\_obj.Holdflag\_dict\_mode\_value)

self.Holdflag\_dict\_value\_mode = eval(json\_obj.Holdflag\_dict\_value\_mode)

else:

raise Exception("File Error", "The file %s is not exist." % (file\_path))

except Exception:

raise Exception("Settings: init failed." )

'''

Random return a modality

'''

def random\_modality(self):

modality = self.modality[random.randint(0, len(self.modality) - 1)]

return modality

'''

Random return a bodypart

'''

def random\_bodypart(self):

modality\_type = self.bodypart[random.randint(0, len(self.bodypart) - 1)]

return modality\_type

'''

Random return a gender

'''

def random\_gender(self):

grender = self.grender[random.randint(0, len(self.grender) - 1)]

return grender

'''

Random return a brithday

'''

def random\_brithday(self):

random\_number = random.randint(0, 100)

random\_days = random\_number \* 365

brithday = (datetime.now() - timedelta(days=random\_days)).strftime('%Y-%m-%d')

return brithday

'''

return the content with string type.

'''

def get\_notifyserver\_exam\_body\_content(self):

ret = self.notifyserver\_exam\_body\_content

return ret

'''

settings = Settings()

print(settings.server)

'''

[复制代码](javascript:void(0);)

3. 确保代码无误，可以正常使用

4. 在同级目录下新建一个\_\_init\_\_.py文件，该文件的目的是将该目录变成一个python的模块包。

5. 我们在文件中输入一下代码：

[复制代码](javascript:void(0);)

# -\*- coding: utf-8 -\*-

from ParameterAndSettings.Settings import Settings

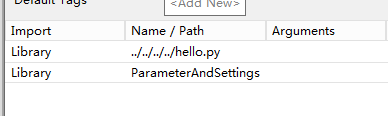
\_\_verison\_\_ = "0.1.1"

class ParameterAndSettings(Settings):

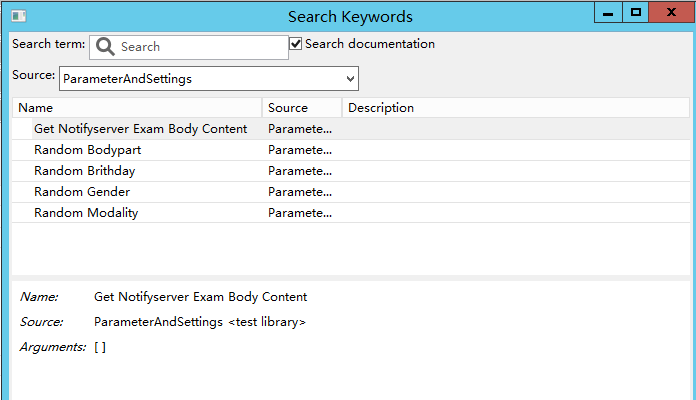
ROBOT\_LIBRARY\_SCOPE = 'GLOBAL'

[复制代码](javascript:void(0);)

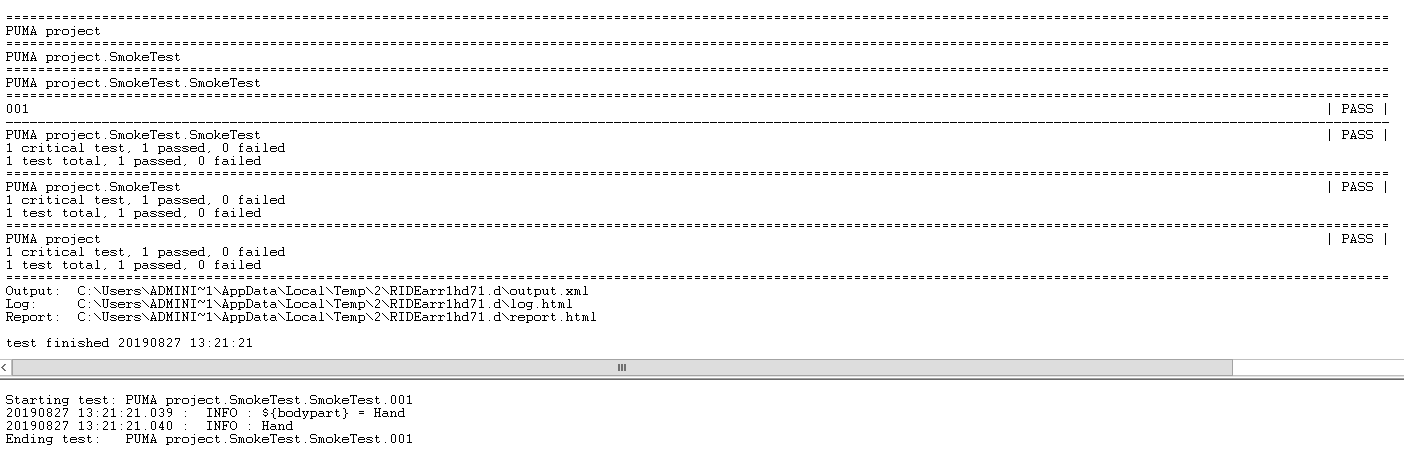
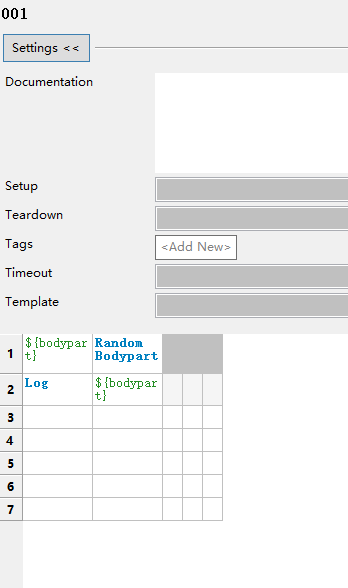
我们看到我们从python文件中，导入我们定义的Settings模块，并定义了一个名叫 ParameterAndSettings的class，这个class将成为RF的库文件。



Settings的中的函数将成为关键字：



并且能够在case中使用：



 需要注意的是：

1. 库文件里面，定义对象的文件名称要和文件中定义的对象名称要一致。就是说ParameterAndSettings文件夹下的Settings.py中定义的对象的名称必须是 【Settings】

2. 对象定义文件中不能出现相对路径否则会报错： 比如"./ path",  path = os.join(paath1, path2)等。

3. 如果库文件导入失败了，可以在Tools->View RIDE Log中查看错误原因。

# [Python 学习笔记21 CMD执行测试用例](https://www.cnblogs.com/wanghao4023030/p/11799459.html)

当我们在ride中设计好测试用例后，我们可以使用ride的界面工具来选择和运行测试用例。

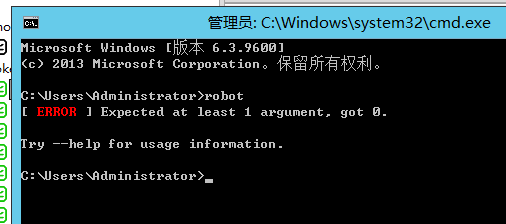
系统也会提供比较好的报告和日志的浏览功能。

但是这样的自动化，毕竟是需要手工介入的。通常，我们是希望能够和Jenkins等配置管理工具结合，实现代码发布，测试自动化运行。

Jenkins本身提供了插件来管理和运行robotframework架构，这里介绍下怎么使用robot工具，通过命令行来管理和运行相应的用例。

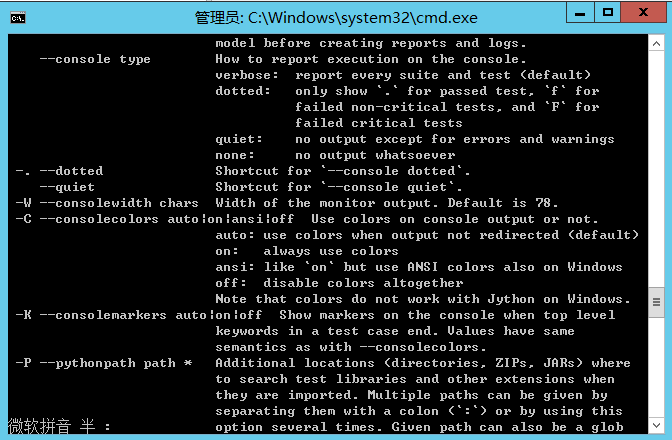
其实方法很简单，首先我们打开cmd窗口，输入 robot 命令：robot

如果robot安装正常，他会抛出下列的错误信息：



根据提示信息，我们在命令中加入 --help来获取帮助： robot --help:

系统会给出所有robot有关的帮助文档：



里面的内容很多，我们也可以将其输出到本地文件中： robot --help > D:\robothelp.txt。

这样我们在文件中预览相关内容会比较方便。

我们经常会用到的一些参数如下：

--test name \*   选择执行那些测试用例，这里的参数是用例的名字。

--suite name \*  　　选择执行那些test suite，该参数会和 --test --include --exclude配合使用，过滤出那些用例需要执行。

--include tag \*   　   选择运行那些tag的测试用例，它支持正则表达式，可以使用 \*，?来替代用例的名称。 还可以使用 AND，OR，Not等操作符。 例如 --include smoke --include BVT; --inlcude smokeANDB\*等等。

--exclude tag \* 　　选择不执行那些tag的测试用例。

--outputdir dir 　　  选择输出的日志和报告的文件夹路径。

--output file　　　  设置输出的文件路径，默认是一个XML文件。

--log file　　    　　 设置输出HTML的log文件， 可以用NONE来关闭，不推荐。  
--report file              设置输出HTML report 文件，也可以使用NONE来关闭，不推荐。 默认文件名字为 report.html。

--timestampoutputs  如果使用了该选项，他会在输出的文件名中间加上一个时间戳 格式为 YYYMMDD-hhmmss。

--maxerrorlines lines 该参数用来设置在报告中，显示多少行错误信息。默认是40行，最小是10行。使用参数NONE，则会显示所有错误信息。

--loglevel level 　　设置log的输出级别， 默认有TRACE,DEBUG, INFO (default), WARN, NONE (no logging)。 例如 --loglevel DEBUG  --loglevel DEBUG:INFO

根据文档，我们基本上可以使用以下格式来用命令启动robot执行用例： robot --options（参数） robot文件目录。

其中文件目录如果是具体到了robot文件，那么只会执行该文件中的相关用例。

还可以同时执行多个文件， 比如 robot d:\smoke.robot, e:\BVT.robot.

如果路径是个文件夹，robot会执行该目录下的所有robot文件， 比如 project下面有 1.robot, 2.robot, name语句 robot ../project 就是执行1.robot, 2.robot中所有的测试用例。

在一个demo的测试项目中使用的命令如下：

robot --include tag demo -d D:\demo -o output -l log -r report -T --timestampoutputs --maxerrorlines NONE -L DEBUG D:\PUMA\_AUTO\RF\_PUMA\PUMA\demo.robot

如果想要再进一步，我的解决方案是，新建了一个ini配置文件，写一个python脚本来解析配置文件，这样只要通过修改ini配置文件，就可以灵活的执行你想要用例或者测试套件。

当然如果你想要和CI集成， Jenkins本身就有很好的插件支持，有机会我们可以尝试一下。