Python3 编程第一步

在前面的教程中我们已经学习了一些 Python3 的基本语法知识，接下来我们来尝试一些实例。

打印字符串:

实例

```
print("Hello, world!")
```

输出结果为:

```
Hello, world!
```

输出变量值:

实例

```
i = 256*256
print('i 的值为：', i)
```

输出结果为:

```
i 的值为： 65536
```

定义变量并进行简单的数学运算

实例

```
x = 3
y = 2
z = x + y
print(z)
```

输出结果为:

```
5
```

定义一个列表并打印出其中的元素:

实例

```
my_list = ['google', 'runoob', 'taobao']
print(my_list[0]) # 输出 "google"
print(my_list[1]) # 输出 "runoob"
print(my_list[2]) # 输出 "taobao"
```

输出结果为:

```
google
runoob
Taobao
```

使用 for 循环打印数字 0 到 4:

实例

```
for i in range(5):
    print(i)
```

输出结果为:

```
0
1
2
3
4
```

根据条件输出不同的结果:

实例

```
x = 6
if x > 10:
    print("x 大于 10")
else:
    print("x 小于或等于 10")
```

输出结果为：

```
x 小于或等于 10
```

下面我们尝试来写一个斐波纳契数列。

斐波那契数列是一个经典的数学问题，其中每个数字是前两个数字之和。

实例(Python 3.0+)

```
#!/usr/bin/python3

# Fibonacci series: 斐波纳契数列
# 两个元素的总和确定了下一个数
a, b = 0, 1
while b < 10:
    print(b)
    a, b = b, a+b
```

其中代码 **a, b = b, a+b** 的计算方式为先计算右边表达式，然后同时赋值给左边，等价于：

```
n=b
m=a+b
a=n
b=m
```

执行以上程序，输出结果为：

```
1
1
2
3
5
8
```

这个例子介绍了几个新特征。

第一行包含了一个复合赋值：变量 a 和 b 同时得到新值 0 和 1。最后一行再次使用了同样的方法，可以看到，右边的表达式会在赋值变动之前执行。右边表达式的执行顺序是从左往右的。

也可以使用 for 循环来实现：

实例

```
n = 10
a, b = 0, 1
for i in range(n):
    print(b)
    a, b = b, a + b
```

end 关键字

关键字 end 可以用于将结果输出到同一行，或者在输出的末尾添加不同的字符，实例如下：

实例(Python 3.0+)

```
#!/usr/bin/python3
```

```
# Fibonacci series: 斐波纳契数列
# 两个元素的总和确定了下一个数
a, b = 0, 1
while b < 1000:
    print(b, end=',')
    a, b = b, a+b
```

执行以上程序，输出结果为：

```
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

Python 推导式

Python 推导式是一种独特的数据处理方式，可以从一个数据序列构建另一个新的数据序列的结构体。

Python 推导式是一种强大且简洁的语法，适用于生成列表、字典、集合和生成器。

在使用推导式时，需要注意可读性，尽量保持表达式简洁，以免影响代码的可读性和可维护性。

Python 支持各种数据结构的推导式：

- 列表(list)推导式
- 字典(dict)推导式
- 集合(set)推导式
- 元组(tuple)推导式

列表推导式

列表推导式格式为：

[表达式 for 变量 in 列表]

[out_exp_res for out_exp in input_list]

或者

[表达式 for 变量 in 列表 if 条件]

[out_exp_res for out_exp in input_list if condition]

- out_exp_res：列表生成元素表达式，可以是有返回值的函数。
- for out_exp in input_list：迭代 input_list 将 out_exp 传入到 out_exp_res 表达式中。
- if condition：条件语句，可以过滤列表中不符合条件的值。

过滤掉长度小于或等于 3 的字符串列表，并将剩下的转换成大写字母：

实例

```
>>> names = ['Bob','Tom','alice','Jerry','Wendy','Smith']
>>> new_names = [name.upper()for name in names if len(name)>3]
>>> print(new_names)
['ALICE', 'JERRY', 'WENDY', 'SMITH']
```

计算 30 以内可以被 3 整除的整数：

实例

```
>>> multiples = [i for i in range(30) if i % 3 == 0]
>>> print(multiples)
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

字典推导式

字典推导基本格式：

{ key_expr: value_expr for value in collection }

或

{ key_expr: value_expr for value in collection if condition }

使用字符串及其长度创建字典：

实例

```
listdemo = ['Google','Runoob', 'Taobao']
# 将列表中各字符串值为键，各字符串的长度为值，组成键值对
>>> newdict = {key:len(key) for key in listdemo}
>>> newdict
{'Google': 6, 'Runoob': 6, 'Taobao': 6}
```

提供三个数字，以三个数字为键，三个数字的平方为值来创建字典：

实例

```
>>> dic = {x: x**2 for x in (2, 4, 6)}
>>> dic
{2: 4, 4: 16, 6: 36}
>>> type(dic)
<class 'dict'>
```

集合推导式

集合推导式基本格式：

```
{ expression for item in Sequence }
```

或

```
{ expression for item in Sequence if conditional }
```

计算数字 1,2,3 的平方数：

实例

```
>>> setnew = {i**2 for i in (1,2,3)}
>>> setnew
{1, 4, 9}
```

判断不是 abc 的字母并输出：

实例

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'d', 'r'}
>>> type(a)
<class 'set'>
```

元组推导式（生成器表达式）

元组推导式可以利用 range 区间、元组、列表、字典和集合等数据类型，快速生成一个满足指定需求的元组。

元组推导式基本格式：

```
(expression for item in Sequence )
```

或

```
(expression for item in Sequence if conditional )
```

元组推导式和列表推导式的用法也完全相同，只是元组推导式是用 **()** 圆括号将各部分括起来，而列表推导式用的是中括号 **[]**，另外元组推导式返回的结果是一个生成器对象。

例如，我们可以使用下面的代码生成一个包含数字 1~9 的元组：

实例

```
>>> a = (x for x in range(1,10))
>>> a
<generator object <genexpr> at 0x7faf6ee20a50>  # 返回的是生成器对象

>>> tuple(a)      # 使用 tuple() 函数，可以直接将生成器对象转换成元组
(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Python3 迭代器与生成器

迭代器

迭代是 Python 最强大的功能之一，是访问集合元素的一种方式。
迭代器是一个可以记住遍历的位置的对象。
迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。
迭代器有两个基本的方法：**iter()** 和 **next()**。

字符串，列表或元组对象都可用于创建迭代器：

实例(Python 3.0+)

```
>>> list=[1,2,3,4]
>>> it = iter(list)    # 创建迭代器对象
>>> print (next(it))   # 输出迭代器的下一个元素
1
>>> print (next(it))
2
>>>
```

迭代器对象可以使用常规 for 语句进行遍历：

实例(Python 3.0+)

```
#!/usr/bin/python3

list=[1,2,3,4]
it = iter(list) # 创建迭代器对象
for x in it:
    print (x, end=" ")
```

执行以上程序，输出结果如下：

```
1 2 3 4
```

也可以使用 next() 函数：

实例(Python 3.0+)

```
#!/usr/bin/python3

import sys # 引入 sys 模块

list=[1,2,3,4]
it = iter(list) # 创建迭代器对象

while True:
    try:
        print (next(it))
    except StopIteration:
        sys.exit()
```

执行以上程序，输出结果如下：

```
1
2
3
```

创建一个迭代器

把一个类作为一个迭代器使用需要在类中实现两个方法 __iter__() 与 __next__() 。

如果你已经了解的面向对象编程，就知道类都有一个构造函数，Python 的构造函数为 __init__(), 它会在对象初始化的时候执行。

更多内容查阅：Python3 面向对象

__iter__() 方法返回一个特殊的迭代器对象， 这个迭代器对象实现了 __next__() 方法并通过 StopIteration 异常标识迭代的完成。

__next__() 方法（Python 2 里是 next()）会返回下一个迭代器对象。

创建一个返回数字的迭代器，初始值为 1，逐步递增 1：

实例(Python 3.0+)

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

执行输出结果为：

```
1
2
3
4
5
```

StopIteration

StopIteration 异常用于标识迭代的完成，防止出现无限循环的情况，在 __next__() 方法中我们可以设置在完成指定循环次数后触发 StopIteration 异常来结束迭代。

在 20 次迭代后停止执行：

实例(Python 3.0+)

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
```

```
        return self

def __next__(self):
    if self.a <= 20:
        x = self.a
        self.a += 1
        return x
    else:
        raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

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

执行输出结果为：

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

生成器

在 Python 中，使用了 **yield** 的函数被称为生成器（generator）。

yield 是一个关键字，用于定义生成器函数，生成器函数是一种特殊的函数，可以在迭代过程中逐步产生值，而不是一次性返回所有结果。

跟普通函数不同的是，生成器是一个返回迭代器的函数，只能用于迭代操作，更简单点理解生成器就是一个迭代器。

当在生成器函数中使用 **yield** 语句时，函数的执行将会暂停，并将 **yield** 后面的表达式作为当前迭代的值返回。

然后，每次调用生成器的 **next()** 方法或使用 **for** 循环进行迭代时，函数会从上次暂停的地方继续执行，直到再次遇到 **yield** 语句。这样，生成器函数可以逐步产生值，而不需要一次性计算并返回所有结果。

调用一个生成器函数，返回的是一个迭代器对象。

下面是一个简单的示例，展示了生成器函数的使用：

实例

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

# 创建生成器对象
generator = countdown(5)

# 通过迭代生成器获取值
print(next(generator)) # 输出: 5
print(next(generator)) # 输出: 4
print(next(generator)) # 输出: 3

# 使用 for 循环迭代生成器
for value in generator:
    print(value) # 输出: 2 1
```

以上实例中，**countdown** 函数是一个生成器函数。它使用 yield 语句逐步产生从 n 到 1 的倒数数字。在每次调用 yield 语句时，函数会返回当前的倒数值，并在下一次调用时从上次暂停的地方继续执行。

通过创建生成器对象并使用 next() 函数或 for 循环迭代生成器，我们可以逐步获取生成器函数产生的值。在这个例子中，我们首先使用 next() 函数获取前三个倒数值，然后通过 for 循环获取剩下的两个倒数值。

生成器函数的优势是它们可以按需生成值，避免一次性生成大量数据并占用大量内存。此外，生成器还可以与其他迭代工具（如 for 循环）无缝配合使用，提供简洁和高效的迭代方式。

执行以上程序，输出结果如下：

```
5
4
3
2
1
```

以下实例使用 yield 实现斐波那契数列：

实例(Python 3.0+)

```
#!/usr/bin/python3

import sys

def fibonacci(n): # 生成器函数 - 斐波那契
    a, b, counter = 0, 1, 0
    while True:
        if (counter > n):
            return
        yield a
        a, b = b, a + b
        counter += 1
f = fibonacci(10) # f 是一个迭代器，由生成器返回生成
```

```
while True:
    try:
        print (next(f), end=" ")
    except StopIteration:
        sys.exit()
```

执行以上程序，输出结果如下：

```
0 1 1 2 3 5 8 13 21 34 55
```

Python3 函数

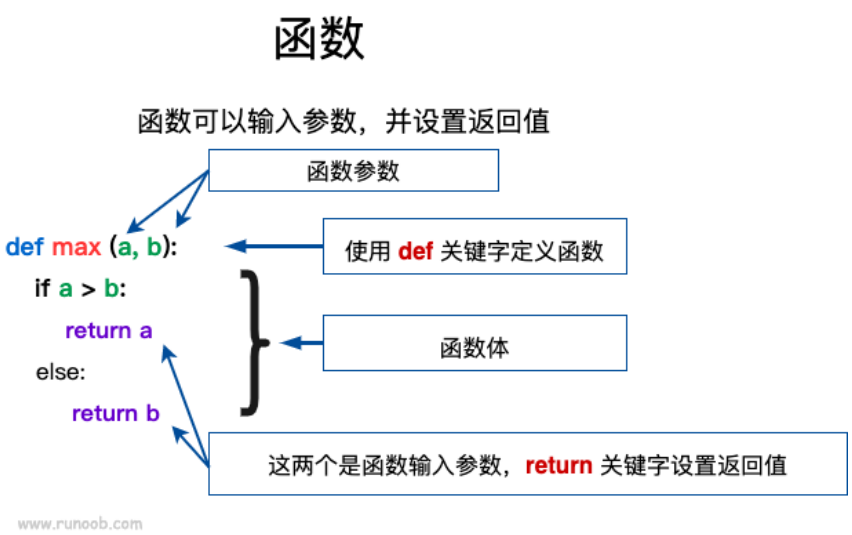
函数是组织好的，可重复使用的，用来实现单一，或相关联功能的代码段。

函数能提高应用的模块性，和代码的重复利用率。你已经知道 Python 提供了许多内建函数，比如 print()。但你也可以自己创建函数，这被叫做用户自定义函数。

定义一个函数

你可以定义一个由自己想要功能的函数，以下是简单的规则：

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



语法

Python 定义函数使用 def 关键字，一般格式如下：

```
def 函数名（参数列表）:
    函数体
```

默认情况下，参数值和参数名称是按函数声明中定义的顺序匹配起来的。

实例

让我们使用函数来输出"Hello World！"：

```
#!/usr/bin/python3

def hello() :
    print("Hello World!")

hello()
```

更复杂点的应用，函数中带上参数变量：

实例(Python 3.0+)

比较两个数，并返回较大的数：

```
#!/usr/bin/python3
def max(a, b):
    if a > b:
        return a
    else:
```

```
    return b

a = 4
b = 5
print(max(a, b))
```

以上实例输出结果：

```
5
```

实例(Python 3.0+)

计算面积函数：

```
#!/usr/bin/python3

# 计算面积函数
def area(width, height):
    return width * height

def print_welcome(name):
    print("Welcome", name)

print_welcome("Runoob")
w = 4
h = 5
print("width =", w, " height =", h, " area =", area(w, h))
```

以上实例输出结果：

```
Welcome Runoob
width = 4  height = 5  area = 20
```

函数调用

定义一个函数：给了函数一个名称，指定了函数里包含的参数，和代码块结构。

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

如下实例调用了 **printme()** 函数：

实例(Python 3.0+)

```
#!/usr/bin/python3

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

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

以上实例输出结果：

```
我要调用用户自定义函数!
```

再次调用同一函数

参数传递

在 python 中，类型属于对象，对象有不同类型的区分，变量是没有类型的：

```
a=[1,2,3]
a="Runoob"
```

以上代码中，**[1,2,3]** 是 List 类型，**"Runoob"** 是 String 类型，而变量 a 是没有类型，它仅仅是一个对象的引用（一个指针），可以是指向 List 类型对象，也可以是指向 String 类型对象。

可更改(mutable)与不可更改(immutable)对象

在 python 中，strings, tuples, 和 numbers 是不可更改的对象，而 list,dict 等则是可以修改的对象。

- **不可变类型：**变量赋值 **a=5** 后再赋值 **a=10**，这里实际是新生成一个 int 值对象 10，再让 a 指向它，而 5 被丢弃，不是改变 a 的值，相当于新生成了 a。
- **可变类型：**变量赋值 **la=[1,2,3,4]** 后再赋值 **la[2]=5** 则是将 list la 的第三个元素值更改，本身 la 没有动，只是其内部的一部分值被修改了。

python 函数的参数传递：

- **不可变类型：**类似 C++ 的值传递，如整数、字符串、元组。如 fun(a)，传递的只是 a 的值，没有影响 a 对象本身。如果在 fun(a) 内部修改 a 的值，则是新生成一个 a 的对象。
- **可变类型：**类似 C++ 的引用传递，如 列表，字典。如 fun(la)，则是将 la 真正的传过去，修改后 fun 外部的 la 也会受影响

python 中一切都是对象，严格意义我们不能说值传递还是引用传递，我们应该说传不可变对象和传可变对象。

python 传不可变对象实例

通过 **id()** 函数来查看内存地址变化：

实例(Python 3.0+)

```
def change(a):
    print(id(a)) # 指向的是同一个对象
    a=10
    print(id(a)) # 一个新对象

a=1
print(id(a))
change(a)
```

以上实例输出结果为：

```
4379369136
4379369136
4379369424
```

可以看见在调用函数前后，形参和实参指向的是同一个对象（对象 id 相同），在函数内部修改形参后，形参指向的是不同的 id。

传可变对象实例

可变对象在函数里修改了参数，那么在调用这个函数的函数里，原始的参数也被改变了。例如：

实例(Python 3.0+)

```
#!/usr/bin/python3

# 可写函数说明
def changeme( mylist ):
```

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

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

传入函数的和在末尾添加新内容的对象用的是同一个引用。故输出结果如下:

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

参数

以下是调用函数时可使用的正式参数类型:

- 必需参数
- 关键字参数
- 默认参数
- 不定长参数

必需参数

必需参数须以正确的顺序传入函数。调用时的数量必须和声明时的一样。

调用 printme() 函数，你必须传入一个参数，不然会出现语法错误:

实例(Python 3.0+)

```
#!/usr/bin/python3

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

# 调用 printme 函数，不加参数会报错
printme()
```

以上实例输出结果:

```
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    printme()
TypeError: printme() missing 1 required positional argument: 'str'
```

关键字参数

关键字参数和函数调用关系紧密，函数调用使用关键字参数来确定传入的参数值。

使用关键字参数允许函数调用时参数的顺序与声明时不一致，因为 Python 解释器能够用参数名匹配参数值。

以下实例在函数 printme() 调用时使用参数名:

实例(Python 3.0+)

```
#!/usr/bin/python3

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

#调用 printme 函数
printme( str = "菜鸟教程")
```

以上实例输出结果:

```
菜鸟教程
```

以下实例中演示了函数参数的使用不需要使用指定顺序:

实例(Python 3.0+)

```
#!/usr/bin/python3

#可写函数说明
def printinfo( name, age ):
   "打印任何传入的字符串"
   print ("名字: ", name)
   print ("年龄: ", age)
   return

#调用 printinfo 函数
printinfo( age=50, name="runoob" )
```

以上实例输出结果:

```
名字:  runoob
年龄:  50
```

默认参数

调用函数时，如果没有传递参数，则会使用默认参数。

以下实例中如果没有传入 age 参数，则使用默认值:

实例(Python 3.0+)

```
#!/usr/bin/python3

#可写函数说明
def printinfo( name, age = 35 ):
   "打印任何传入的字符串"
   print ("名字: ", name)
   print ("年龄: ", age)
   return

#调用 printinfo 函数
printinfo( age=50, name="runoob" )
print ("------------------------")
```

```
printinfo( name="runoob" )
```

以上实例输出结果:

```
名字: runoob
年龄: 50
------------------------
名字: runoob
年龄: 35
```

不定长参数

你可能需要一个函数能处理比当初声明时更多的参数。这些参数叫做不定长参数，和上述 2 种参数不同，声明时不会命名。基本语法如下:

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

加了星号 * 的参数会以元组(tuple)的形式导入，存放所有未命名的变量参数。

实例(Python 3.0+)

```
#!/usr/bin/python3

# 可写函数说明
def printinfo( arg1, *vartuple ):
    "打印任何传入的参数"
    print ("输出: ")
    print (arg1)
    print (vartuple)

# 调用 printinfo 函数
printinfo( 70, 60, 50 )
```

以上实例输出结果:

```
输出:
70
(60, 50)
```

如果在函数调用时没有指定参数，它就是一个空元组。我们也可以不向函数传递未命名的变量。如下实例:

实例(Python 3.0+)

```
#!/usr/bin/python3

# 可写函数说明
def printinfo( arg1, *vartuple ):
    "打印任何传入的参数"
    print ("输出: ")
    print (arg1)
    for var in vartuple:
        print (var)
    return
```

```
# 调用 printinfo 函数
printinfo( 10 )
printinfo( 70, 60, 50 )
```

以上实例输出结果:

```
输出:
10
输出:
70
60
50
```

还有一种就是参数带两个星号 **基本语法如下:

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

加了两个星号 ** 的参数会以字典的形式导入。

实例(Python 3.0+)

```
#!/usr/bin/python3

# 可写函数说明
def printinfo( arg1, **vardict ):
   "打印任何传入的参数"
   print ("输出: ")
   print (arg1)
   print (vardict)

# 调用 printinfo 函数
printinfo(1, a=2,b=3)
```

以上实例输出结果:

```
输出:
1
{'a': 2, 'b': 3}
```

声明函数时，参数中星号 * 可以单独出现，例如:

```
def f(a,b,*,c):
    return a+b+c
```

如果单独出现星号 *，则星号 * 后的参数必须用关键字传入:

```
>>> def f(a,b,*,c):
...     return a+b+c
...
>>> f(1,2,3)   # 报错
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: f() takes 2 positional arguments but 3 were given
>>> f(1,2,c=3) # 正常
6
>>>
```

匿名函数

Python 使用 **lambda** 来创建匿名函数。

所谓匿名，意即不再使用 **def** 语句这样标准的形式定义一个函数。

- **lambda** 只是一个表达式，函数体比 **def** 简单很多。
- lambda 的主体是一个表达式，而不是一个代码块。仅仅能在 lambda 表达式中封装有限的逻辑进去。
- lambda 函数拥有自己的命名空间，且不能访问自己参数列表之外或全局命名空间里的参数。
- 虽然 lambda 函数看起来只能写一行，却不等同于 C 或 C++ 的内联函数，内联函数的目的是调用小函数时不占用栈内存从而减少函数调用的开销，提高代码的执行速度。

语法

lambda 函数的语法只包含一个语句，如下：

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

设置参数 a 加上 10:

实例

```
x = lambda a : a + 10
print(x(5))
```

以上实例输出结果：

```
15
```

以下实例匿名函数设置两个参数：

实例(Python 3.0+)

```
#!/usr/bin/python3

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

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

以上实例输出结果：

```
相加后的值为 :  30
相加后的值为 :  40
```

我们可以将匿名函数封装在一个函数内，这样可以使用同样的代码来创建多个匿名函数。

以下实例将匿名函数封装在 myfunc 函数中，通过传入不同的参数来创建不同的匿名函数：

实例

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)
```

```
print(mydoubler(11))
print(mytripler(11))
```

以上实例输出结果:

```
22
33
```

更多匿名函数还可以参考: Python lambda（匿名函数）

---

return 语句

return [表达式] 语句用于退出函数，选择性地向调用方返回一个表达式。不带参数值的 return 语句返回 None。

之前的例子都没有示范如何返回数值，以下实例演示了 return 语句的用法:

实例(Python 3.0+)

```
#!/usr/bin/python3

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

# 调用 sum 函数
total = sum( 10, 20 )
print ("函数外 : ", total)
```

以上实例输出结果:

```
函数内 :  30
函数外 :  30
```

强制位置参数

Python3.8 新增了一个函数形参语法 / 用来指明函数形参必须使用指定位置参数，不能使用关键字参数的形式。

在以下的例子中，形参 a 和 b 必须使用指定位置参数，c 或 d 可以是位置形参或关键字形参，而 e 和 f 要求为关键字形参:

```
def f(a, b, /, c, d, *, e, f):
    print(a, b, c, d, e, f)
```

以下使用方法是正确的:

```
f(10, 20, 30, d=40, e=50, f=60)
```

以下使用方法会发生错误:

```
f(10, b=20, c=30, d=40, e=50, f=60)   # b 不能使用关键字参数的形式
f(10, 20, 30, 40, 50, f=60)           # e 必须使用关键字参数的形式
```

Python lambda（匿名函数）

Python 使用 **lambda** 来创建匿名函数。

lambda 函数是一种小型、匿名的、内联函数，它可以具有任意数量的参数，但只能有一个表达式。

匿名函数不需要使用 **def** 关键字定义完整函数。

lambda 函数通常用于编写简单的、单行的函数，通常在需要函数作为参数传递的情况下使用，例如在 map()、filter()、reduce() 等函数中。

lambda 函数特点：

- lambda 函数是匿名的，它们没有函数名称，只能通过赋值给变量或作为参数传递给其他函数来使用。
- lambda 函数通常只包含一行代码，这使得它们适用于编写简单的函数。

lambda 语法格式：

lambda arguments: expression

- lambda 是 Python 的关键字，用于定义 lambda 函数。
- arguments 是参数列表，可以包含零个或多个参数，但必须在冒号(:)前指定。
- expression 是一个表达式，用于计算并返回函数的结果。

以下的 lambda 函数没有参数：

实例

```
f = lambda: "Hello, world!"
print(f()) # 输出: Hello, world!
```

输出结果为：

```
Hello, world!
```

以下实例使用 lambda 创建匿名函数，设置一个函数参数 a，函数计算参数 a 加 10，并返回结果：

实例

```
x = lambda a : a + 10
print(x(5))
```

输出结果为：

```
15
```

lambda 函数也可以设置多个参数，参数使用逗号 **,** 隔开：

以下实例使用 lambda 创建匿名函数，函数参数 a 与 b 相乘，并返回结果：

实例

```
x = lambda a, b : a * b
print(x(5, 6))
```

输出结果为：

```
30
```

以下实例使用 lambda 创建匿名函数，函数参数 a、b 与 c 相加，并返回结果：

实例

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

输出结果为：

```
13
```

lambda 函数通常与内置函数如 map()、filter() 和 reduce() 一起使用，以便在集合上执行操作。例如：

实例

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))
```

```
print(squared) # 输出: [1, 4, 9, 16, 25]
```
输出结果为:
```
[1, 4, 9, 16, 25]
```

使用 lambda 函数与 filter() 一起，筛选偶数:

实例
```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers) # 输出：[2, 4, 6, 8]
```
输出结果为:
```
[2, 4, 6, 8]
```

下面是一个使用 reduce() 和 lambda 表达式演示如何计算一个序列的累积乘积:

实例
```
from functools import reduce
numbers = [1, 2, 3, 4, 5]
# 使用 reduce() 和 lambda 函数计算乘积
product = reduce(lambda x, y: x * y, numbers)
print(product) # 输出：120
```

输出结果为:
```
120
```

在上面的实例中，reduce() 函数通过遍历 numbers 列表，并使用 lambda 函数将累积的结果不断更新，最终得到了 **1 * 2 * 3 * 4 * 5 = 120** 的结果。

Python 装饰器

装饰器（decorators）是 Python 中的一种高级功能，它允许你动态地修改函数或类的行为。

装饰器是一种函数，它接受一个函数作为参数，并返回一个新的函数或修改原来的函数。

装饰器的语法使用 **@decorator_name** 来应用在函数或方法上。

Python 还提供了一些内置的装饰器，比如 **@staticmethod** 和 **@classmethod**，用于定义静态方法和类方法。

装饰器的应用场景：

- **日志记录**: 装饰器可用于记录函数的调用信息、参数和返回值。
- **性能分析**: 可以使用装饰器来测量函数的执行时间。
- **权限控制**: 装饰器可用于限制对某些函数的访问权限。
- **缓存**: 装饰器可用于实现函数结果的缓存，以提高性能。

基本语法

Python 装饰允许在不修改原有函数代码的基础上，动态地增加或修改函数的功能，装饰器本质上是一个接收函数作为输入并返回一个新的包装过后的函数的对象。

语法

```
def decorator_function(original_function):
    def wrapper(*args, **kwargs):
        # 这里是在调用原始函数前添加的新功能
        before_call_code()

        result = original_function(*args, **kwargs)

        # 这里是在调用原始函数后添加的新功能
        after_call_code()

        return result
    return wrapper

# 使用装饰器
@decorator_function
def target_function(arg1, arg2):
    pass  # 原始函数的实现
```

解析：decorator 是一个装饰器函数，它接受一个函数 func 作为参数，并返回一个内部函数 wrapper，在 wrapper 函数内部，你可以执行一些额外的操作，然后调用原始函数 func，并返回其结果。

- decorator_function 是装饰器，它接收一个函数 original_function 作为参数。
- wrapper 是内部函数，它是实际会被调用的新函数，它包裹了原始函数的调用，并在其前后增加了额外的行为。
- 当我们使用 @decorator_function 前缀在 target_function 定义前，Python 会自动将 target_function 作为参数传递给 decorator_function，然后将返回的 wrapper 函数替换掉原来的 target_function。

使用装饰器

装饰器通过 **@** 符号应用在函数定义之前，例如：

```
@time_logger
def target_function():
    pass
```

等同于：

```
def target_function():
    pass
target_function = time_logger(target_function)
```

这会将 target_function 函数传递给 decorator 装饰器，并将返回的函数重新赋值给 target_function。从而，每次调用 target_function 时，实际上是调用了经过装饰器处理后的函数。
通过装饰器，开发者可以在保持代码整洁的同时，灵活且高效地扩展程序的功能。

带参数的装饰器

装饰器函数也可以接受参数，例如：

实例

```
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat(3)
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

以上代码中 repeat 函数是一个带参数的装饰器，它接受一个整数参数 n，然后返回一个装饰器函数。该装饰器函数内部定义了 wrapper 函数，在调用原始函数之前重复执行 n 次。因此，greet 函数在被 @repeat(3) 装饰后，会打印三次问候语。

类装饰器

除了函数装饰器，Python 还支持类装饰器。类装饰器是包含 __call__ 方法的类，它接受一个函数作为参数，并返回一个新的函数。

实例

```
class DecoratorClass:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        # 在调用原始函数之前/之后执行的代码
        result = self.func(*args, **kwargs)
        # 在调用原始函数之后执行的代码
        return result

@DecoratorClass
def my_function():
    pass
```

Python3 数据结构

本章节我们主要结合前面所学的知识点来介绍 Python 数据结构。

列表

Python 中列表是可变的，这是它区别于字符串和元组的最重要的特点，一句话概括即：列表可以修改，而字符串和元组不能。

以下是 Python 中列表的方法：

方法	描述
list.append(x)	把一个元素添加到列表的结尾，相当于 a[len(a):] = [x]。
list.extend(L)	通过添加指定列表的所有元素来扩充列表，相当于 a[len(a):] = L。
list.insert(i, x)	在指定位置插入一个元素。第一个参数是准备插入到其前面的那个元素的索引，例如 a.insert(0, x) 会插入到整个列表之前，而 a.insert(len(a), x) 相当于 a.append(x) 。
list.remove(x)	删除列表中值为 x 的第一个元素。如果没有这样的元素，就会返回一个错误。
list.pop([i])	从列表的指定位置移除元素，并将其返回。如果没有指定索引，a.pop()返回最后一个元素。元素随即从列表中被移除。（方法中 i 两边的方括号表示这个参数是可选的，而不是要求你输入一对方括号，你会经常在 Python 库参考手册中遇到这样的标记。）
list.clear()	移除列表中的所有项，等于 del a[:]。
list.index(x)	返回列表中第一个值为 x 的元素的索引。如果没有匹配的元素就会返回一个错误。
list.count(x)	返回 x 在列表中出现的次数。
list.sort()	对列表中的元素进行排序。
list.reverse()	倒排列表中的元素。
list.copy()	返回列表的浅复制，等于 a[:]。

下面示例演示了列表的大部分方法：

实例

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
```

```
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

注意：类似 insert, remove 或 sort 等修改列表的方法没有返回值。

将列表当做栈使用

在 Python 中，可以使用列表（list）来实现栈的功能。栈是一种后进先出（LIFO, Last-In-First-Out）数据结构，意味着最后添加的元素最先被移除。列表提供了一些方法，使其非常适合用于栈操作，特别是 **append()** 和 **pop()** 方法。

用 append() 方法可以把一个元素添加到栈顶，用不指定索引的 pop() 方法可以把一个元素从栈顶释放出来。

栈操作

- **压入（Push）**：将一个元素添加到栈的顶端。
- **弹出（Pop）**：移除并返回栈顶元素。
- **查看栈顶元素（Peek/Top）**：返回栈顶元素而不移除它。
- **检查是否为空（IsEmpty）**：检查栈是否为空。
- **获取栈的大小（Size）**：获取栈中元素的数量。

以下是如何在 Python 中使用列表实现这些操作的详细说明:

1、创建一个空栈

实例

```
stack = []
```

2、压入（Push）操作

使用 append() 方法将元素添加到栈的顶端:

实例

```
stack.append(1)
stack.append(2)
stack.append(3)
print(stack)  # 输出: [1, 2, 3]
```

3、弹出（Pop）操作

使用 pop() 方法移除并返回栈顶元素:

实例

```
top_element = stack.pop()
print(top_element)  # 输出: 3
print(stack)        # 输出: [1, 2]
```

4、查看栈顶元素（Peek/Top）

直接访问列表的最后一个元素（不移除）:

实例

```
top_element = stack[-1]
print(top_element)  # 输出: 2
```

5、检查是否为空（IsEmpty）

检查列表是否为空:

实例

```
is_empty = len(stack) == 0
print(is_empty)  # 输出: False
```

6、获取栈的大小（Size）

使用 len() 函数获取栈中元素的数量：

实例

```
size = len(stack)
print(size)  # 输出: 2
```

实例

以下是一个完整的实例，展示了如何使用上述操作来实现一个简单的栈：

实例

```
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)

    def pop(self):
        if not self.is_empty():
            return self.stack.pop()
        else:
            raise IndexError("pop from empty stack")

    def peek(self):
        if not self.is_empty():
            return self.stack[-1]
        else:
            raise IndexError("peek from empty stack")

    def is_empty(self):
        return len(self.stack) == 0

    def size(self):
        return len(self.stack)

# 使用示例
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)

print("栈顶元素:", stack.peek())  # 输出: 栈顶元素: 3
print("栈大小:", stack.size())    # 输出: 栈大小: 3

print("弹出元素:", stack.pop())  # 输出: 弹出元素: 3
print("栈是否为空:", stack.is_empty())  # 输出: 栈是否为空: False
print("栈大小:", stack.size())    # 输出: 栈大小: 2
```

以上代码中，我们定义了一个 Stack 类，封装了列表作为底层数据结构，并实现了栈的基本操作。

输出结果如下：

栈顶元素: 3
栈大小: 3
弹出元素: 3
栈是否为空: False
栈大小: 2

将列表当作队列使用

在 Python 中，列表（list）可以用作队列（queue），但由于列表的特点，直接使用列表来实现队列并不是最优的选择。

队列是一种先进先出（FIFO, First-In-First-Out）的数据结构，意味着最早添加的元素最先被移除。

使用列表时，如果频繁地在列表的开头插入或删除元素，性能会受到影响，因为这些操作的时间复杂度是 O(n)。为了解决这个问题，Python 提供了 collections.deque，它是双端队列，可以在两端高效地添加和删除元素。

使用 collections.deque 实现队列

collections.deque 是 Python 标准库的一部分，非常适合用于实现队列。

以下是使用 deque 实现队列的示例：

实例

```
from collections import deque

# 创建一个空队列
queue = deque()

# 向队尾添加元素
queue.append('a')
queue.append('b')
queue.append('c')

print("队列状态:", queue)  # 输出: 队列状态: deque(['a', 'b', 'c'])

# 从队首移除元素
first_element = queue.popleft()
print("移除的元素:", first_element)  # 输出: 移除的元素: a
print("队列状态:", queue)            # 输出: 队列状态: deque(['b', 'c'])

# 查看队首元素（不移除）
front_element = queue[0]
print("队首元素:", front_element)    # 输出: 队首元素: b

# 检查队列是否为空
is_empty = len(queue) == 0
print("队列是否为空:", is_empty)     # 输出: 队列是否为空: False

# 获取队列大小
size = len(queue)
print("队列大小:", size)             # 输出: 队列大小: 2
```

使用列表实现队列

虽然 deque 更高效，但如果坚持使用列表来实现队列，也可以这么做。以下是如何使用列表实现队列的示例：

1. 创建队列

实例

```
queue = []
```

2. 向队尾添加元素

使用 append() 方法将元素添加到队尾：

实例

```
queue.append('a')
queue.append('b')
queue.append('c')
print("队列状态:", queue)  # 输出: 队列状态: ['a', 'b', 'c']
```

3. 从队首移除元素

使用 pop(0) 方法从队首移除元素：

实例

```
first_element = queue.pop(0)
print("移除的元素:", first_element)  # 输出: 移除的元素: a
print("队列状态:", queue)            # 输出: 队列状态: ['b', 'c']
```

4. 查看队首元素（不移除）

直接访问列表的第一个元素：

实例

```
front_element = queue[0]
print("队首元素:", front_element)    # 输出: 队首元素: b
```

5. 检查队列是否为空

检查列表是否为空：

实例

```
is_empty = len(queue) == 0
print("队列是否为空:", is_empty)     # 输出: 队列是否为空: False
```

6. 获取队列大小

使用 len() 函数获取队列的大小：

实例

```
size = len(queue)
print("队列大小:", size)            # 输出: 队列大小: 2
```

实例（使用列表实现队列）

```
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)
```

```
    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)
        else:
            raise IndexError("dequeue from empty queue")

    def peek(self):
        if not self.is_empty():
            return self.queue[0]
        else:
            raise IndexError("peek from empty queue")

    def is_empty(self):
        return len(self.queue) == 0

    def size(self):
        return len(self.queue)

# 使用示例
queue = Queue()
queue.enqueue('a')
queue.enqueue('b')
queue.enqueue('c')

print("队首元素:", queue.peek())  # 输出: 队首元素: a
print("队列大小:", queue.size())  # 输出: 队列大小: 3

print("移除的元素:", queue.dequeue())  # 输出: 移除的元素: a
print("队列是否为空:", queue.is_empty())  # 输出: 队列是否为空: False
print("队列大小:", queue.size())  # 输出: 队列大小: 2
```

虽然可以使用列表来实现队列，但使用 collections.deque 会更高效和简洁。它提供了 O(1) 时间复杂度的添加和删除操作，非常适合队列这种数据结构。

列表推导式

列表推导式提供了从序列创建列表的简单途径。通常应用程序将一些操作应用于某个序列的每个元素，用其获得的结果作为生成新列表的元素，或者根据确定的判定条件创建子序列。

每个列表推导式都在 for 之后跟一个表达式，然后有零到多个 for 或 if 子句。返回结果是一个根据表达从其后的 for 和 if 上下文环境中生成出来的列表。如果希望表达式推导出一个元组，就必须使用括号。

这里我们将列表中每个数值乘三，获得一个新的列表：

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
```

现在我们玩一点小花样：

```
>>> [[x, x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```

这里我们对序列里每一个元素逐个调用某方法：

实例

```
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
```

我们可以用 if 子句作为过滤器：

```
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
```

以下是一些关于循环和其它技巧的演示：

```
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

列表推导式可以使用复杂表达式或嵌套函数：

```
>>> [str(round(355/113, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

嵌套列表解析

Python 的列表还可以嵌套。

以下实例展示了 3X4 的矩阵列表：

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

以下实例将 3X4 的矩阵列表转换为 4X3 列表：

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

以上实例也可以使用以下方法来实现：

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

另外一种实现方法：

```
>>> transposed = []
>>> for i in range(4):
```

```
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

---

del 语句

使用 del 语句可以从一个列表中根据索引来删除一个元素，而不是值来删除元素。这与使用 pop() 返回一个值不同。可以用 del 语句从列表中删除一个切割，或清空整个列表（我们以前介绍的方法是给该切割赋一个空列表）。

例如：

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

也可以用 del 删除实体变量：

```
>>> del a
```

元组和序列

元组由若干逗号分隔的值组成，例如：

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

如你所见，元组在输出时总是有括号的，以便于正确表达嵌套结构。在输入时可能有或没有括号， 不过括号通常是必须的（如果元组是更大的表达式的一部分）。

集合

集合是一个无序不重复元素的集。基本功能包括关系测试和消除重复元素。

可以用大括号({})创建集合。注意：如果要创建一个空集合，你必须用 set() 而不是 {}；后者创建一个空的字典，下一节我们会介绍这个数据结构。

以下是一个简单的演示：

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # 删除重复的
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # 检测成员
True
>>> 'crabgrass' in basket
False

>>> # 以下演示了两个集合的操作
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # a 中唯一的字母
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # 在 a 中的字母，但不在 b 中
{'r', 'd', 'b'}
>>> a | b                              # 在 a 或 b 中的字母
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # 在 a 和 b 中都有的字母
{'a', 'c'}
>>> a ^ b                              # 在 a 或 b 中的字母，但不同时在 a 和 b 中
{'r', 'd', 'b', 'm', 'z', 'l'}
```

集合也支持推导式：

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

字典

另一个非常有用的 Python 内建数据类型是字典。

序列是以连续的整数为索引，与此不同的是，字典以关键字为索引，关键字可以是任意不可变类型，通常用字符串或数值。

理解字典的最佳方式是把它看做无序的键=>值对集合。在同一个字典之内，关键字必须是互不相同。

一对大括号创建一个空的字典：{}。

这是一个字典运用的简单例子：

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> list(tel.keys())
['irv', 'guido', 'jack']
```

```
>>> sorted(tel.keys())
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

构造函数 dict() 直接从键值对元组列表中构建字典。如果有固定的模式，列表推导式指定特定的键值对：

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

此外，字典推导可以用来创建任意键和值的表达式词典：

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

如果关键字只是简单的字符串，使用关键字参数指定键值对有时候更方便：

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

遍历技巧

在字典中遍历时，关键字和对应的值可以使用 items() 方法同时解读出来：

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

在序列中遍历时，索引位置和对应值可以使用 enumerate() 函数同时得到：

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

同时遍历两个或更多的序列，可以使用 zip() 组合：

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

要反向遍历一个序列，首先指定这个序列，然后调用 reversed() 函数：

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
```

...
9
7
5
3
1

要按顺序遍历一个序列，使用 sorted() 函数返回一个已排序的序列，并不修改原值：

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

Python3 模块

在前面的几个章节中我们基本上是用 python 解释器来编程，如果你从 Python 解释器退出再进入，那么你定义的所有的方法和变量就都消失了。

为此 Python 提供了一个办法，把这些定义存放在文件中，为一些脚本或者交互式的解释器实例使用，这个文件被称为模块。

模块是一个包含所有你定义的函数和变量的文件，其后缀名是.py。模块可以被别的程序引入，以使用该模块中的函数等功能。这也是使用 python 标准库的方法。

下面是一个使用 python 标准库中模块的例子。

实例(Python 3.0+)

```
#!/usr/bin/python3
# 文件名: using_sys.py

import sys

print('命令行参数如下:')
for i in sys.argv:
   print(i)

print('\n\nPython 路径为：', sys.path, '\n')
```

执行结果如下所示：

```
$ python using_sys.py 参数 1 参数 2
命令行参数如下:
using_sys.py
参数 1
参数 2


Python 路径为： ['/root', '/usr/lib/python3.4', '/usr/lib/python3.4/plat-x86_64-linux-gnu', '/usr/lib/python3.4/lib-dynload', '/usr/local/lib/python3.4/dist-packages', '/usr/lib/python3/dist-packages']
```

- 1、import sys 引入 python 标准库中的 sys.py 模块；这是引入某一模块的方法。
- 2、sys.argv 是一个包含命令行参数的列表。
- 3、sys.path 包含了一个 Python 解释器自动查找所需模块的路径的列表。

---

import 语句

想使用 Python 源文件，只需在另一个源文件里执行 import 语句，语法如下：

```
import module1[, module2[,... moduleN]
```

当解释器遇到 import 语句，如果模块在当前的搜索路径就会被导入。

搜索路径时一个解释器会先进行搜索的所有目录的列表。如想要导入模块 support，需要把命令放在脚本的顶端：

support.py 文件代码

```
#!/usr/bin/python3
# Filename: support.py

def print_func( par ):
   print ("Hello : ", par)
   return
```

test.py 引入 support 模块:

test.py 文件代码

```
#!/usr/bin/python3
# Filename: test.py

# 导入模块
import support

# 现在可以调用模块里包含的函数了
support.print_func("Runoob")
```

以上实例输出结果:

```
$ python3 test.py
Hello :  Runoob
```

下载代码

一个模块只会被导入一次，不管你执行了多少次 **import**。这样可以防止导入模块被一遍又一遍地执行。
当我们使用 import 语句的时候，Python 解释器是怎样找到对应的文件的呢?
这就涉及到 Python 的搜索路径，搜索路径是由一系列目录名组成的，Python 解释器就依次从这些目录中去寻找所引入的模块。

这看起来很像环境变量，事实上，也可以通过定义环境变量的方式来确定搜索路径。
搜索路径是在 Python 编译或安装的时候确定的，安装新的库应该也会修改。搜索路径被存储在 sys 模块中的 path 变量，做一个简单的实验，在交互式解释器中，输入以下代码:

```
>>> import sys
>>> sys.path
['',      '/usr/lib/python3.4',      '/usr/lib/python3.4/plat-x86_64-linux-gnu',      '/usr/lib/python3.4/lib-dynload',
'/usr/local/lib/python3.4/dist-packages', '/usr/lib/python3/dist-packages']
>>>
```

sys.path 输出是一个列表，其中第一项是空串 ''，代表当前目录（若是从一个脚本中打印出来的话，可以更清楚地看出是哪个目录），亦即我们执行 python 解释器的目录（对于脚本的话就是运行的脚本所在的目录）。
因此若像我一样在当前目录下存在与要引入模块同名的文件，就会把要引入的模块屏蔽掉。
了解了搜索路径的概念，就可以在脚本中修改 sys.path 来引入一些不在搜索路径中的模块。

现在，在解释器的当前目录或者 sys.path 中的一个目录里面来创建一个 fibo.py 的文件，代码如下:

实例

```
# 斐波那契(fibonacci)数列模块

def fib(n): # 定义到 n 的斐波那契数列
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n): # 返回到 n 的斐波那契数列
    result = []
```

```
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

然后进入 Python 解释器，使用下面的命令导入这个模块：
>>> import fibo
这样做并没有把直接定义在 fibo 中的函数名称写入到当前符号表里，只是把模块 fibo 的名字写到了那里。

可以使用模块名称来访问函数：
实例
```
>>>fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

如果你打算经常使用一个函数，你可以把它赋给一个本地的名称：
```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

---

from … import 语句
Python 的 from 语句让你从模块中导入一个指定的部分到当前命名空间中，语法如下：
from modname import name1[, name2[, ... nameN]]

例如，要导入模块 fibo 的 fib 函数，使用如下语句：
```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```
这个声明不会把整个 fibo 模块导入到当前的命名空间中，它只会将 fibo 里的 fib 函数引入进来。

---

from … import * 语句
把一个模块的所有内容全都导入到当前的命名空间也是可行的，只需使用如下声明：
from modname import *
这提供了一个简单的方法来导入一个模块中的所有项目。然而这种声明不该被过多地使用。

深入模块
模块除了方法定义，还可以包括可执行的代码。这些代码一般用来初始化这个模块。这些代码只有在第一次被导入时才会被执行。
每个模块有各自独立的符号表，在模块内部为所有的函数当作全局符号表来使用。
所以，模块的作者可以放心大胆的在模块内部使用这些全局变量，而不用担心把其他用户的全局变量搞混。

从另一个方面，当你确实知道你在做什么的话，你也可以通过 modname.itemname 这样的表示法来访问模块内的函数。
模块是可以导入其他模块的。在一个模块（或者脚本，或者其他地方）的最前面使用 import 来导入一个模

块，当然这只是一个惯例，而不是强制的。被导入的模块的名称将被放入当前操作的模块的符号表中。

还有一种导入的方法，可以使用 import 直接把模块内（函数，变量的）名称导入到当前操作模块。比如:
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
这种导入的方法不会把被导入的模块的名称放在当前的字符表中（所以在这个例子里面，fibo 这个名称是没有定义的）。

这还有一种方法，可以一次性的把模块中的所有（函数，变量）名称都导入到当前模块的字符表:
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
这将把所有的名字都导入进来，但是那些由单一下划线（_）开头的名字不在此例。大多数情况， Python 程序员不使用这种方法，因为引入的其它来源的命名，很可能覆盖了已有的定义。

__name__属性
一个模块被另一个程序第一次引入时，其主程序将运行。如果我们想在模块被引入时，模块中的某一程序块不执行，我们可以用__name__属性来使该程序块仅在该模块自身运行时执行。
```
#!/usr/bin/python3
# Filename: using_name.py

if __name__ == '__main__':
   print('程序自身在运行')
else:
   print('我来自另一模块')
```

运行输出如下:
```
$ python using_name.py
程序自身在运行

$ python
>>> import using_name
我来自另一模块
>>>
```

说明： 每个模块都有一个__name__属性，当其值是'__main__'时，表明该模块自身在运行，否则是被引入。
说明：**__name__** 与 **__main__** 底下是双下划线， _ _ 是这样去掉中间的那个空格。

---

dir() 函数
内置的函数 dir() 可以找到模块内定义的所有名称。以一个字符串列表的形式返回:
```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__loader__', '__name__',
 '__package__', '__stderr__', '__stdin__', '__stdout__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_getframe',
 '_home', '_mercurial', '_xoptions', 'abiflags', 'api_version', 'argv',
```

```
 'base_exec_prefix', 'base_prefix', 'builtin_module_names', 'byteorder',
 'call_tracing', 'callstats', 'copyright', 'displayhook',
 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix',
 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
 'getrefcount', 'getsizeof', 'getswitchinterval', 'gettotalrefcount',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
 'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit',
 'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout',
 'thread_info', 'version', 'version_info', 'warnoptions']
```

如果没有给定参数，那么 dir() 函数会罗列出当前定义的所有名称:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir() # 得到一个当前模块中定义的属性列表
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
>>> a = 5 # 建立一个新的变量 'a'
>>> dir()
['__builtins__', '__doc__', '__name__', 'a', 'sys']
>>>
>>> del a # 删除变量名 a
>>>
>>> dir()
['__builtins__', '__doc__', '__name__', 'sys']
>>>
```

标准模块

Python 本身带着一些标准的模块库，在 Python 库参考文档中将会介绍到（就是后面的"库参考文档"）。

有些模块直接被构建在解析器里，这些虽然不是一些语言内置的功能，但是他却能很高效的使用，甚至是系统级调用也没问题。

这些组件会根据不同的操作系统进行不同形式的配置，比如 winreg 这个模块就只会提供给 Windows 系统。

应该注意到这有一个特别的模块 sys ，它内置在每一个 Python 解析器中。变量 sys.ps1 和 sys.ps2 定义了主提示符和副提示符所对应的字符串:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Runoob!')
Runoob!
C>
```

包

包是一种管理 Python 模块命名空间的形式，采用"点模块名称"。

比如一个模块的名称是 A.B， 那么他表示一个包 A 中的子模块 B 。
就好像使用模块的时候，你不用担心不同模块之间的全局变量相互影响一样，采用点模块名称这种形式也不用担心不同库之间的模块重名的情况。
这样不同的作者都可以提供 NumPy 模块，或者是 Python 图形库。
不妨假设你想设计一套统一处理声音文件和数据的模块（或者称之为一个"包"）。
现存很多种不同的音频文件格式（基本上都是通过后缀名区分的，例如： .wav，:file:.aiff，:file:.au,），所以你需要有一组不断增加的模块，用来在不同的格式之间转换。
并且针对这些音频数据，还有很多不同的操作（比如混音，添加回声，增加均衡器功能，创建人造立体声效果），所以你还需要一组怎么也写不完的模块来处理这些操作。

这里给出了一种可能的包结构（在分层的文件系统中）：

```
sound/                          顶层包
    __init__.py                 初始化 sound 包
    formats/                    文件格式转换子包
            __init__.py
            wavread.py
            wavwrite.py
            aiffread.py
            aiffwrite.py
            auread.py
            auwrite.py
            ...
    effects/                    声音效果子包
            __init__.py
            echo.py
            surround.py
            reverse.py
            ...
    filters/                    filters 子包
            __init__.py
            equalizer.py
            vocoder.py
            karaoke.py
            ...
```

在导入一个包的时候，Python 会根据 sys.path 中的目录来寻找这个包中包含的子目录。
目录只有包含一个叫做 __init__.py 的文件才会被认作是一个包，主要是为了避免一些滥俗的名字（比如叫做 string）不小心的影响搜索路径中的有效模块。
最简单的情况，放一个空的 :file:__init__.py 就可以了。当然这个文件中也可以包含一些初始化代码或者为（将在后面介绍的） __all__变量赋值。

用户可以每次只导入一个包里面的特定模块，比如:

```
import sound.effects.echo
```

这将会导入子模块:sound.effects.echo。 他必须使用全名去访问:

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

还有一种导入子模块的方法是:

```
from sound.effects import echo
```

这同样会导入子模块: echo，并且他不需要那些冗长的前缀，所以他可以这样使用:
echo.echofilter(input, output, delay=0.7, atten=4)

还有一种变化就是直接导入一个函数或者变量:
from sound.effects.echo import echofilter

同样的，这种方法会导入子模块: echo，并且可以直接使用他的 echofilter() 函数:
echofilter(input, output, delay=0.7, atten=4)

注意当使用 **from package import item** 这种形式的时候，对应的 item 既可以是包里面的子模块（子包），或者包里面定义的其他名称，比如函数，类或者变量。
import 语法会首先把 item 当作一个包定义的名称，如果没找到，再试图按照一个模块去导入。如果还没找到，抛出一个 **:exc:ImportError** 异常。
反之，如果使用形如 **import item.subitem.subsubitem** 这种导入形式，除了最后一项，都必须是包，而最后一项则可以是模块或者是包，但是不可以是类，函数或者变量的名字。

从一个包中导入*
如果我们使用 **from sound.effects import *** 会发生什么呢?
Python 会进入文件系统，找到这个包里面所有的子模块，然后一个一个的把它们都导入进来。
但这个方法在 Windows 平台上工作的就不是非常好，因为 Windows 是一个不区分大小写的系统。
在 Windows 平台上，我们无法确定一个叫做 ECHO.py 的文件导入为模块是 echo 还是 Echo，或者是 ECHO。
为了解决这个问题，我们只需要提供一个精确包的索引。
导入语句遵循如下规则：如果包定义文件 **__init__.py** 存在一个叫做 **__all__** 的列表变量，那么在使用 **from package import *** 的时候就把这个列表中的所有名字作为包内容导入。
作为包的作者，可别忘了在更新包之后保证 **__all__** 也更新了啊。

以下实例在 file:sounds/effects/__init__.py 中包含如下代码:
__all__ = ["echo", "surround", "reverse"]
这表示当你使用 from sound.effects import *这种用法时，你只会导入包里面这三个子模块。

如果 **__all__** 真的没有定义，那么使用 **from sound.effects import *** 这种语法的时候，就不会导入包 sound.effects 里的任何子模块。他只是把包 sound.effects 和它里面定义的所有内容导入进来（可能运行 __init__.py 里定义的初始化代码）。
这会把 __init__.py 里面定义的所有名字导入进来。并且他不会破坏掉我们在这句话之前导入的所有明确指定的模块。看下这部分代码:

import sound.effects.echo
import sound.effects.surround
from sound.effects import *

这个例子中，在执行 from...import 前，包 sound.effects 中的 echo 和 surround 模块都被导入到当前的命名空间中了。（当然如果定义了 __all__ 就更没问题了）

通常我们并不主张使用 * 这种方法来导入模块，因为这种方法经常会导致代码的可读性降低。不过这样倒的确是可以省去不少敲键的功夫，而且一些模块都设计成了只能通过特定的方法导入。
记住，使用 **from Package import specific_submodule** 这种方法永远不会有错。事实上，这也是推荐的方法。除非是你要导入的子模块有可能和其他包的子模块重名。

如果在结构中包是一个子包（比如这个例子中对于包 sound 来说），而你又想导入兄弟包（同级别的包）你

就得使用导入绝对的路径来导入。比如，如果模块 sound.filters.vocoder 要使用包 sound.effects 中的模块 echo，你就要写成 from sound.effects import echo。

```
from . import echo
from .. import formats
from ..filters import equalizer
```

无论是隐式的还是显式的相对导入都是从当前模块开始的。主模块的名字永远是"__main__"，一个 Python 应用程序的主模块，应当总是使用绝对路径引用。

包还提供一个额外的属性__path__。这是一个目录列表，里面每一个包含的目录都有为这个包服务的 __init__.py，你得在其他__init__.py 被执行前定义哦。可以修改这个变量，用来影响包含在包里面的模块和子包。
这个功能并不常用，一般用来扩展包里面的模块。

Python3 输入和输出

在前面几个章节中，我们其实已经接触了 Python 的输入输出的功能。本章节我们将具体介绍 Python 的输入输出。

输出格式美化

Python 两种输出值的方式: 表达式语句和 print() 函数。

第三种方式是使用文件对象的 write() 方法，标准输出文件可以用 sys.stdout 引用。

如果你希望输出的形式更加多样，可以使用 str.format() 函数来格式化输出值。

如果你希望将输出的值转成字符串，可以使用 repr() 或 str() 函数来实现。

- **str()：** 函数返回一个用户易读的表达形式。
- **repr()：** 产生一个解释器易读的表达形式。

例如

```
>>> s = 'Hello, Runoob'
>>> str(s)
'Hello, Runoob'
>>> repr(s)
"'Hello, Runoob'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'x 的值为： ' + repr(x) + ', y 的值为： ' + repr(y) + '...'
>>> print(s)
x 的值为： 32.5, y 的值为： 40000...
>>> #  repr() 函数可以转义字符串中的特殊字符
... hello = 'hello, runoob\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, runoob\n'
>>> # repr() 的参数可以是 Python 的任何对象
... repr((x, y, ('Google', 'Runoob')))
"(32.5, 40000, ('Google', 'Runoob'))"
```

这里有两种方式输出一个平方与立方的表:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # 注意前一行 'end' 的使用
...     print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
```

```
10 100 1000

>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

注意：在第一个例子中, 每列间的空格由 print() 添加。
这个例子展示了字符串对象的 rjust() 方法, 它可以将字符串靠右, 并在左边填充空格。
还有类似的方法, 如 ljust() 和 center()。 这些方法并不会写任何东西, 它们仅仅返回新的字符串。

另一个方法 zfill(), 它会在数字的左边填充 0， 如下所示：

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

str.format() 的基本使用如下:

```
>>> print('{}网址： "{}!"'.format('菜鸟教程', 'www.runoob.com'))
菜鸟教程网址： "www.runoob.com!"
```

括号及其里面的字符 (称作格式化字段) 将会被 format() 中的参数替换。
在括号中的数字用于指向传入对象在 format() 中的位置， 如下所示：

```
>>> print('{0} 和 {1}'.format('Google', 'Runoob'))
Google 和 Runoob
>>> print('{1} 和 {0}'.format('Google', 'Runoob'))
Runoob 和 Google
```

如果在 format() 中使用了关键字参数, 那么它们的值会指向使用该名字的参数。

```
>>> print('{name}网址： {site}'.format(name='菜鸟教程', site='www.runoob.com'))
菜鸟教程网址： www.runoob.com
```

位置及关键字参数可以任意的结合:

```
>>> print('站点列表 {0}, {1}, 和 {other}。'.format('Google', 'Runoob', other='Taobao'))
站点列表 Google, Runoob, 和 Taobao。
```

!a (使用 **ascii()**), **!s** (使用 **str()**) 和 **!r** (使用 **repr()**) 可以用于在格式化某个值之前对其进行转化:

```
>>> import math
```

```
>>> print('常量 PI 的值近似为： {}。'.format(math.pi))
常量 PI 的值近似为： 3.141592653589793。
>>> print('常量 PI 的值近似为： {!r}。'.format(math.pi))
常量 PI 的值近似为： 3.141592653589793。
```

可选项 **:** 和格式标识符可以跟着字段名。 这就允许对值进行更好的格式化。
下面的例子将 Pi 保留到小数点后三位：

```
>>> import math
>>> print('常量 PI 的值近似为 {0:.3f}。'.format(math.pi))
常量 PI 的值近似为 3.142。
```

在 **:** 后传入一个整数, 可以保证该域至少有这么多的宽度。 用于美化表格时很有用。

```
>>> table = {'Google': 1, 'Runoob': 2, 'Taobao': 3}
>>> for name, number in table.items():
...     print('{0:10} ==> {1:10d}'.format(name, number))
...
Google     ==>          1
Runoob     ==>          2
Taobao     ==>          3
```

如果你有一个很长的格式化字符串, 而你不想将它们分开, 那么在格式化时通过变量名而非位置会是很好的事情。
最简单的就是传入一个字典, 然后使用方括号 **[]** 来访问键值 :

```
>>> table = {'Google': 1, 'Runoob': 2, 'Taobao': 3}
>>> print('Runoob: {0[Runoob]:d}; Google: {0[Google]:d}; Taobao: {0[Taobao]:d}'.format(table))
Runoob: 2; Google: 1; Taobao: 3
```

也可以通过在 table 变量前使用 ** 来实现相同的功能：

```
>>> table = {'Google': 1, 'Runoob': 2, 'Taobao': 3}
>>> print('Runoob: {Runoob:d}; Google: {Google:d}; Taobao: {Taobao:d}'.format(**table))
Runoob: 2; Google: 1; Taobao: 3
```

旧式字符串格式化

% 操作符也可以实现字符串格式化。 它将左边的参数作为类似 **sprintf()** 式的格式化字符串, 而将右边的代入, 然后返回格式化后的字符串. 例如:

```
>>> import math
>>> print('常量 PI 的值近似为： %5.3f。' % math.pi)
常量 PI 的值近似为： 3.142。
```

因为 str.format() 是比较新的函数， 大多数的 Python 代码仍然使用 % 操作符。但是因为这种旧式的格式化最终会从该语言中移除, 应该更多的使用 str.format().

读取键盘输入

Python 提供了 input() 内置函数从标准输入读入一行文本，默认的标准输入是键盘。

实例

```
#!/usr/bin/python3

str = input("请输入：");
print ("你输入的内容是: ", str)
```

这会产生如下的对应着输入的结果:
请输入：菜鸟教程
你输入的内容是: 菜鸟教程

读和写文件

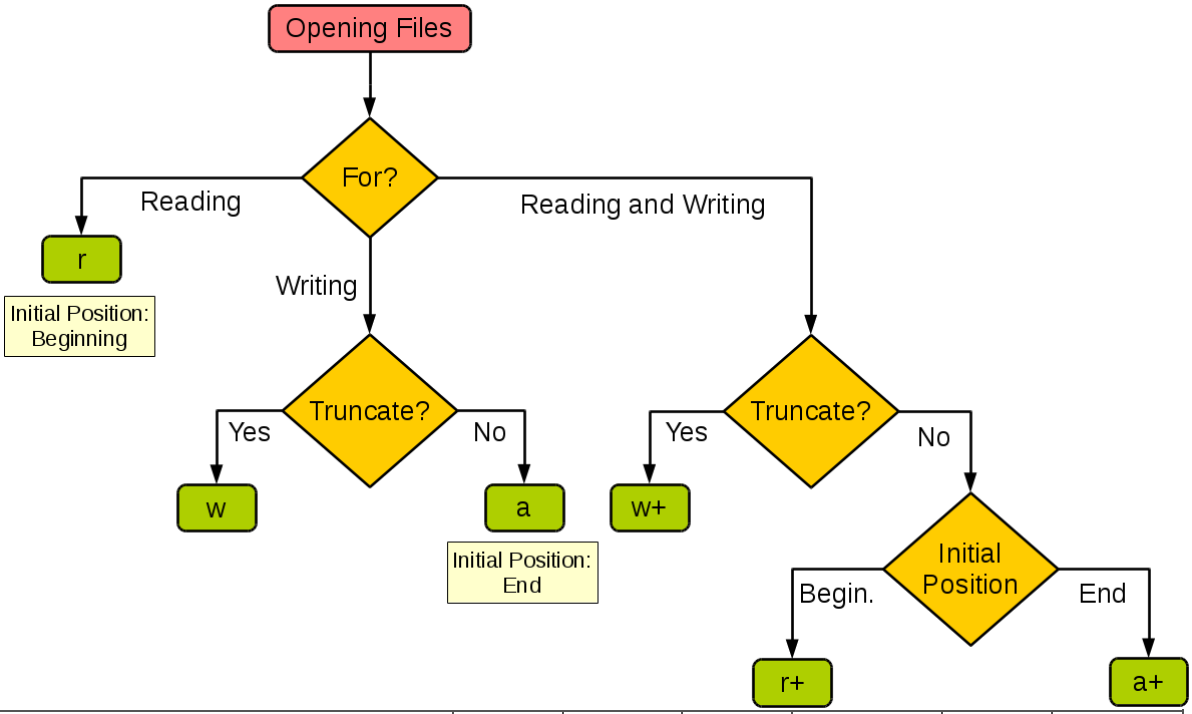
open() 将会返回一个 file 对象，基本语法格式如下:
open(filename, mode)

- filename：包含了你要访问的文件名称的字符串值。
- 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)。

不同模式打开文件的完全列表:

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。
r+	打开一个文件用于读写。文件指针将会放在文件的开头。
rb+	以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。
w	打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。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打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。
w+	打开一个文件用于读写。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。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打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。
a	打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
ab	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
a+	打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会是追加模式。如果该文件不存在，创建新文件用于读写。
ab+	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文件不存在，创建新文件用于读写。

下图很好的总结了这几种模式:



模式	r	r+	w	w+	a	a+
读	+	+		+		+
写		+	+	+	+	+
创建			+	+	+	+
覆盖			+	+		
指针在开始	+	+	+	+		
指针在结尾					+	+

以下实例将字符串写入到文件 foo.txt 中:

实例

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "w")

f.write( "Python 是一个非常好的语言。\n 是的，的确非常好!!\n" )

# 关闭打开的文件
f.close()
```

- 第一个参数为要打开的文件名。
- 第二个参数描述文件如何使用的字符。 mode 可以是 'r' 如果文件只读, 'w' 只用于写 (如果存在同名文件则将被删除), 和 'a' 用于追加文件内容; 所写的任何数据都会被自动增加到末尾. 'r+' 同时用于读写。 mode 参数是可选的; 'r' 将是默认值。

此时打开文件 foo.txt,显示如下:

```
$ cat /tmp/foo.txt
Python 是一个非常好的语言。
```

是的，的确非常好!!

文件对象的方法

本节中剩下的例子假设已经创建了一个称为 f 的文件对象。

f.read()

为了读取一个文件的内容，调用 f.read(size), 这将读取一定数目的数据, 然后作为字符串或字节对象返回。

size 是一个可选的数字类型的参数。 当 size 被忽略了或者为负, 那么该文件的所有内容都将被读取并且返回。

以下实例假定文件 foo.txt 已存在（上面实例中已创建）:

实例

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "r")

str = f.read()
print(str)

# 关闭打开的文件
f.close()
```

执行以上程序，输出结果为:

```
Python 是一个非常好的语言。
是的，的确非常好!!
```

f.readline()

f.readline() 会从文件中读取单独的一行。换行符为 '\n'。f.readline() 如果返回一个空字符串, 说明已经已经读取到最后一行。

实例

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "r")

str = f.readline()
print(str)

# 关闭打开的文件
f.close()
```

执行以上程序，输出结果为:

```
Python 是一个非常好的语言。
```

f.readlines()

f.readlines() 将返回该文件中包含的所有行。

如果设置可选参数 sizehint, 则读取指定长度的字节, 并且将这些字节按行分割。

实例

```
#!/usr/bin/python3
```

```
# 打开一个文件
f = open("/tmp/foo.txt", "r")

str = f.readlines()
print(str)

# 关闭打开的文件
f.close()
```

执行以上程序，输出结果为：

```
['Python 是一个非常好的语言。\n', '是的，的确非常好!!\n']
```

另一种方式是迭代一个文件对象然后读取每行:

实例

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "r")

for line in f:
    print(line, end='')

# 关闭打开的文件
f.close()
```

执行以上程序，输出结果为：

```
Python 是一个非常好的语言。
是的，的确非常好!!
```

这个方法很简单, 但是并没有提供一个很好的控制。 因为两者的处理机制不同, 最好不要混用。

f.write()

f.write(string) 将 string 写入到文件中, 然后返回写入的字符数。

实例

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "w")

num = f.write( "Python 是一个非常好的语言。\n 是的，的确非常好!!\n" )
print(num)
# 关闭打开的文件
f.close()
```

执行以上程序，输出结果为：

```
29
```

如果要写入一些不是字符串的东西, 那么将需要先进行转换:

实例

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo1.txt", "w")

value = ('www.runoob.com', 14)
s = str(value)
f.write(s)

# 关闭打开的文件
f.close()
```

执行以上程序，打开 foo1.txt 文件：

```
$ cat /tmp/foo1.txt
('www.runoob.com', 14)
```

f.tell()

f.tell() 用于返回文件当前的读/写位置（即文件指针的位置）。文件指针表示从文件开头开始的字节数偏移量。

f.tell() 返回一个整数，表示文件指针的当前位置。

f.seek()

如果要改变文件指针当前的位置, 可以使用 f.seek(offset, from_what) 函数。

f.seek(offset, whence) 用于移动文件指针到指定位置。

offset 表示相对于 whence 参数的偏移量，from_what 的值, 如果是 0 表示开头, 如果是 1 表示当前位置, 2 表示文件的结尾，例如：

- seek(x,0)： 从起始位置即文件首行首字符开始移动 x 个字符
- seek(x,1)： 表示从当前位置往后移动 x 个字符
- seek(-x,2)：表示从文件的结尾往前移动 x 个字符

from_what 值为默认为 0，即文件开头。

下面给出一个完整的例子：

```
>>> f = open('/tmp/foo.txt', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)     # 移动到文件的第六个字节
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # 移动到文件的倒数第三字节
13
>>> f.read(1)
b'd'
```

f.close()

在文本文件中 (那些打开文件的模式下没有 b 的), 只会相对于文件起始位置进行定位。

当你处理完一个文件后，调用 f.close() 来关闭文件并释放系统的资源，如果尝试再调用该文件，则会抛出异常。

```
>>> f.close()
>>> f.read()
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

当处理一个文件对象时, 使用 with 关键字是非常好的方式。在结束后, 它会帮你正确的关闭文件。 而且写起来也比 try - finally 语句块要简短:

```
>>> with open('/tmp/foo.txt', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

文件对象还有其他方法, 如 isatty() 和 trucate(), 但这些通常比较少用。

---

pickle 模块

python 的 pickle 模块实现了基本的数据序列和反序列化。
通过 pickle 模块的序列化操作我们能够将程序中运行的对象信息保存到文件中去，永久存储。
通过 pickle 模块的反序列化操作，我们能够从文件中创建上一次程序保存的对象。

基本接口：

```
pickle.dump(obj, file, [,protocol])
```

有了 pickle 这个对象, 就能对 file 以读取的形式打开:

```
x = pickle.load(file)
```

注解：从 file 中读取一个字符串，并将它重构为原来的 python 对象。
file: 类文件对象，有 read()和 readline()接口。

实例 1

```
#!/usr/bin/python3
import pickle

# 使用 pickle 模块将数据对象保存到文件
data1 = {'a': [1, 2.0, 3, 4+6j],
         'b': ('string', u'Unicode string'),
         'c': None}

selfref_list = [1, 2, 3]
selfref_list.append(selfref_list)

output = open('data.pkl', 'wb')

# Pickle dictionary using protocol 0.
pickle.dump(data1, output)

# Pickle the list using the highest protocol available.
pickle.dump(selfref_list, output, -1)

output.close()
```

实例 2

```
#!/usr/bin/python3
import pprint, pickle

#使用 pickle 模块从文件中重构 python 对象
pkl_file = open('data.pkl', 'rb')

data1 = pickle.load(pkl_file)
pprint.pprint(data1)

data2 = pickle.load(pkl_file)
pprint.pprint(data2)

pkl_file.close()
```

Python3 File(文件) 方法

open() 方法

Python **open()** 方法用于打开一个文件，并返回文件对象。

在对文件进行处理过程都需要使用到这个函数，如果该文件无法被打开，会抛出 **OSError**。

注意：使用 **open()** 方法一定要保证关闭文件对象，即调用 **close()** 方法。

open() 函数常用形式是接收两个参数：文件名(file)和模式(mode)。

open(file, mode='r')

完整的语法格式为：

open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)

参数说明:

- file: 必需，文件路径（相对或者绝对路径）。
- mode: 可选，文件打开模式
- buffering: 设置缓冲
- encoding: 一般使用 utf8
- errors: 报错级别
- newline: 区分换行符
- closefd: 传入的 file 参数类型
- opener: 设置自定义开启器，开启器的返回值必须是一个打开的文件描述符。

mode 参数有：

模式	描述
t	文本模式 (默认)。
x	写模式，新建一个文件，如果该文件已存在则会报错。
b	二进制模式。
+	打开一个文件进行更新(可读可写)。
U	通用换行模式（**Python 3 不支持**）。
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打开文件，并从开头开始编辑，即原有内容会被删除。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打开文件，并从开头开始编辑，即原有内容会被删除。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打开文件，并从开头开始编辑，即原有

	内容会被删除。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打开文件，并从开头开始编辑，即原有内容会被删除。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。

默认为文本模式，如果要以二进制模式打开，加上 **b** 。

file 对象

file 对象使用 open 函数来创建，下表列出了 file 对象常用的函数：

序号	方法及描述
1	file.close() 关闭文件。关闭后文件不能再进行读写操作。
2	file.flush() 刷新文件内部缓冲，直接把内部缓冲区的数据立刻写入文件, 而不是被动的等待输出缓冲区写入。
3	file.fileno() 返回一个整型的文件描述符(file descriptor FD 整型), 可以用在如 os 模块的 read 方法等一些底层操作上。
4	file.isatty() 如果文件连接到一个终端设备返回 True，否则返回 False。
5	file.next() **Python 3 中的 File 对象不支持 next() 方法。** 返回文件下一行。
6	file.read([size]) 从文件读取指定的字节数，如果未给定或为负则读取所有。
7	file.readline([size]) 读取整行，包括 "\n" 字符。
8	file.readlines([sizeint]) 读取所有行并返回列表，若给定 sizeint>0，返回总和大约为 sizeint 字节的行，实际读取值可能比 sizeint 较大, 因为需要填充缓冲区。
9	file.seek(offset[, whence])

	移动文件读取指针到指定位置
10	file.tell() 返回文件当前位置。
11	file.truncate([size]) 从文件的首行首字符开始截断，截断文件为 size 个字符，无 size 表示从当前位置截断；截断之后后面的所有字符被删除，其中 windows 系统下的换行代表 2 个字符大小。
12	file.write(str) 将字符串写入文件，返回的是写入的字符长度。
13	file.writelines(sequence) 向文件写入一个序列字符串列表，如果需要换行则要自己加入每行的换行符。

Python3 OS 文件/目录方法

os 模块提供了非常丰富的方法用来处理文件和目录。常用的方法如下表所示:

序号	方法及描述
1	os.access(path, mode) 检验权限模式
2	os.chdir(path) 改变当前工作目录
3	os.chflags(path, flags) 设置路径的标记为数字标记。
4	os.chmod(path, mode) 更改权限
5	os.chown(path, uid, gid) 更改文件所有者
6	os.chroot(path) 改变当前进程的根目录
7	os.close(fd) 关闭文件描述符 fd
8	os.closerange(fd_low, fd_high) 关闭所有文件描述符，从 fd_low (包含) 到 fd_high (不包含), 错误会忽略
9	os.dup(fd) 复制文件描述符 fd
10	os.dup2(fd, fd2) 将一个文件描述符 fd 复制到另一个 fd2
11	os.fchdir(fd) 通过文件描述符改变当前工作目录
12	os.fchmod(fd, mode) 改变一个文件的访问权限，该文件由参数 fd 指定，参数 mode 是 Unix 下的文件访问权限。
13	os.fchown(fd, uid, gid) 修改一个文件的所有权，这个函数修改一个文件的用户 ID 和用户组 ID，该文件由文件描述符 fd 指定。
14	os.fdatasync(fd) 强制将文件写入磁盘，该文件由文件描述符 fd 指定，但是不强制更新文件的状态信息。
15	os.fdopen(fd[, mode[, bufsize]]) 通过文件描述符 fd 创建一个文件对象，并返回这个文件对象
16	os.fpathconf(fd, name) 返回一个打开的文件的系统配置信息。name 为检索的系统配置的值，它也许是一个定义系统值的字符串，这些名字在很多标准中指定（POSIX.1, Unix 95, Unix 98, 和其它）。

17	os.fstat(fd) 返回文件描述符 fd 的状态，像 stat()。
18	os.fstatvfs(fd) 返回包含文件描述符 fd 的文件的文件系统的信息，Python 3.3 相等于 statvfs()。
19	os.fsync(fd) 强制将文件描述符为 fd 的文件写入硬盘。
20	os.ftruncate(fd, length) 裁剪文件描述符 fd 对应的文件, 所以它最大不能超过文件大小。
21	os.getcwd() 返回当前工作目录
22	os.getcwdb() 返回一个当前工作目录的 Unicode 对象
23	os.isatty(fd) 如果文件描述符 fd 是打开的，同时与 tty(-like)设备相连，则返回 true, 否则 False。
24	os.lchflags(path, flags) 设置路径的标记为数字标记，类似 chflags()，但是没有软链接
25	os.lchmod(path, mode) 修改连接文件权限
26	os.lchown(path, uid, gid) 更改文件所有者，类似 chown，但是不追踪链接。
27	os.link(src, dst) 创建硬链接，名为参数 dst，指向参数 src
28	os.listdir(path) 返回 path 指定的文件夹包含的文件或文件夹的名字的列表。
29	os.lseek(fd, pos, how) 设置文件描述符 fd 当前位置为 pos, how 方式修改: SEEK_SET 或者 0 设置从文件开始的计算的 pos; SEEK_CUR 或者 1 则从当前位置计算; os.SEEK_END 或者 2 则从文件尾部开始. 在 unix，Windows 中有效
30	os.lstat(path) 像 stat(),但是没有软链接
31	os.major(device) 从原始的设备号中提取设备 major 号码 (使用 stat 中的 st_dev 或者 st_rdev field)。
32	os.makedev(major, minor) 以 major 和 minor 设备号组成一个原始设备号
33	os.makedirs(path[, mode]) 递归文件夹创建函数。像 mkdir()，但创建的所有 intermediate-level 文件夹需要包含子文件夹。
34	os.minor(device) 从原始的设备号中提取设备 minor 号码 (使用 stat 中的 st_dev 或者 st_rdev field)。

35	os.mkdir(path[, mode]) 以数字 mode 的 mode 创建一个名为 path 的文件夹.默认的 mode 是 0777 (八进制)。
36	os.mkfifo(path[, mode]) 创建命名管道，mode 为数字，默认为 0666 (八进制)
37	os.mknod(filename[, mode=0600, device]) 创建一个名为 filename 文件系统节点（文件，设备特别文件或者命名 pipe）。
38	os.open(file, flags[, mode]) 打开一个文件，并且设置需要的打开选项，mode 参数是可选的
39	os.openpty() 打开一个新的伪终端对。返回 pty 和 tty 的文件描述符。
40	os.pathconf(path, name) 返回相关文件的系统配置信息。
41	os.pipe() 创建一个管道. 返回一对文件描述符(r, w) 分别为读和写
42	os.popen(command[, mode[, bufsize]]) 从一个 command 打开一个管道
43	os.read(fd, n) 从文件描述符 fd 中读取最多 n 个字节，返回包含读取字节的字符串，文件描述符 fd 对应文件已达到结尾, 返回一个空字符串。
44	os.readlink(path) 返回软链接所指向的文件
45	os.remove(path) 删除路径为 path 的文件。如果 path 是一个文件夹，将抛出 OSError; 查看下面的 rmdir()删除一个 directory。
46	os.removedirs(path) 递归删除目录。
47	os.rename(src, dst) 重命名文件或目录，从 src 到 dst
48	os.renames(old, new) 递归地对目录进行更名，也可以对文件进行更名。
49	os.rmdir(path) 删除 path 指定的空目录，如果目录非空，则抛出一个 OSError 异常。
50	os.stat(path) 获取 path 指定的路径的信息，功能等同于 C API 中的 stat()系统调用。
51	os.stat_float_times([newvalue]) 决定 stat_result 是否以 float 对象显示时间戳
52	os.statvfs(path) 获取指定路径的文件系统统计信息

53	os.symlink(src, dst) 创建一个软链接
54	os.tcgetpgrp(fd) 返回与终端 fd（一个由 os.open()返回的打开的文件描述符）关联的进程组
55	os.tcsetpgrp(fd, pg) 设置与终端 fd（一个由 os.open()返回的打开的文件描述符）关联的进程组为 pg。
56	os.tempnam([dir[, prefix]]) **Python3 中已删除。**返回唯一的路径名用于创建临时文件。
57	os.tmpfile() **Python3 中已删除。**返回一个打开的模式为(w+b)的文件对象 .这文件对象没有文件夹入口，没有文件描述符，将会自动删除。
58	os.tmpnam() **Python3 中已删除。**为创建一个临时文件返回一个唯一的路径
59	os.ttyname(fd) 返回一个字符串，它表示与文件描述符 fd 关联的终端设备。如果 fd 没有与终端设备关联，则引发一个异常。
60	os.unlink(path) 删除文件路径
61	os.utime(path, times) 返回指定的 path 文件的访问和修改的时间。
62	os.walk(top[, topdown=True[, onerror=None[, followlinks=False]]]) 输出在文件夹中的文件名通过在树中游走，向上或者向下。
63	os.write(fd, str) 写入字符串到文件描述符 fd 中. 返回实际写入的字符串长度
64	os.path 模块 获取文件的属性信息。
65	os.pardir() 获取当前目录的父目录，以字符串形式显示目录名。
66	os.replace() 重命名文件或目录。
67	os.startfile() 用于在 Windows 上打开一个文件或文件夹。

Python3 错误和异常

作为 Python 初学者，在刚学习 Python 编程时，经常会看到一些报错信息，在前面我们没有提及，这章节我们会专门介绍。

Python 有两种错误很容易辨认：语法错误和异常。

Python assert（断言）用于判断一个表达式，在表达式条件为 false 的时候触发异常。

语法错误

Python 的语法错误或者称之为解析错，是初学者经常碰到的，如下实例

```
>>> while True print('Hello world')
  File "<stdin>", line 1, in ?
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

这个例子中，函数 print() 被检查到有错误，是它前面缺少了一个冒号：。

语法分析器指出了出错的一行，并且在最先找到的错误的位置标记了一个小小的箭头。

异常

即便 Python 程序的语法是正确的，在运行它的时候，也有可能发生错误。运行期检测到的错误被称为异常。

大多数的异常都不会被程序处理，都以错误信息的形式展现在这里:

实例

```
>>> 10 * (1/0)          # 0 不能作为除数，触发异常
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: division by zero
>>> 4 + spam*3          # spam 未定义，触发异常
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2             # int 不能与 str 相加，触发异常
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

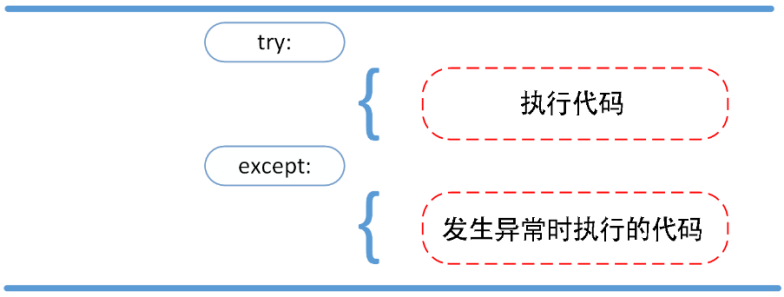
异常以不同的类型出现，这些类型都作为信息的一部分打印出来：例子中的类型有 ZeroDivisionError，NameError 和 TypeError。

错误信息的前面部分显示了异常发生的上下文，并以调用栈的形式显示具体信息。

异常处理

try/except

异常捕捉可以使用 **try/except** 语句。



以下例子中，让用户输入一个合法的整数，但是允许用户中断这个程序（使用 Control-C 或者操作系统提供的方法）。用户中断的信息会引发一个 KeyboardInterrupt 异常。

```
while True:
    try:
        x = int(input("请输入一个数字: "))
        break
    except ValueError:
        print("您输入的不是数字，请再次尝试输入！")
```

try 语句按照如下方式工作;

- 首先，执行 try 子句（在关键字 try 和关键字 except 之间的语句）。
- 如果没有异常发生，忽略 except 子句，try 子句执行后结束。
- 如果在执行 try 子句的过程中发生了异常，那么 try 子句余下的部分将被忽略。如果异常的类型和 except 之后的名称相符，那么对应的 except 子句将被执行。
- 如果一个异常没有与任何的 except 匹配，那么这个异常将会传递给上层的 try 中。

一个 try 语句可能包含多个 except 子句，分别来处理不同的特定的异常。最多只有一个分支会被执行。
处理程序将只针对对应的 try 子句中的异常进行处理，而不是其他的 try 的处理程序中的异常。

一个 except 子句可以同时处理多个异常，这些异常将被放在一个括号里成为一个元组，例如:

```
except (RuntimeError, TypeError, NameError):
    pass
```

最后一个 except 子句可以忽略异常的名称，它将被当作通配符使用。你可以使用这种方法打印一个错误信息，然后再次把异常抛出。

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

try/except...else

try/except 语句还有一个可选的 else 子句，如果使用这个子句，那么必须放在所有的 except 子句之后。

else 子句将在 try 子句没有发生任何异常的时候执行。

以下实例在 try 语句中判断文件是否可以打开，如果打开文件时正常的没有发生异常则执行 else 部分的语句，读取文件内容：

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

使用 else 子句比把所有的语句都放在 try 子句里面要好，这样可以避免一些意想不到，而 except 又无法捕获的异常。

异常处理并不仅仅处理那些直接发生在 try 子句中的异常，而且还能处理子句中调用的函数（甚至间接调用的函数）里抛出的异常。例如：

```
>>> def this_fails():
        x = 1/0

>>> try:
        this_fails()
    except ZeroDivisionError as err:
        print('Handling run-time error:', err)
```

```
Handling run-time error: int division or modulo by zero
```

try-finally 语句

try-finally 语句无论是否发生异常都将执行最后的代码。

以下实例中 finally 语句无论异常是否发生都会执行:

实例

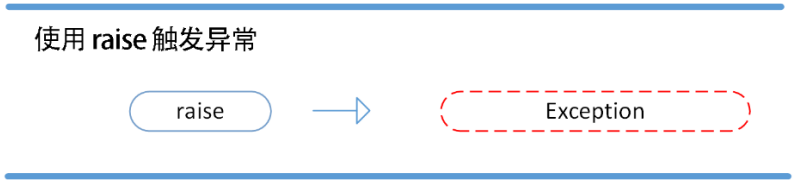
```
try:
    runoob()
except AssertionError as error:
    print(error)
else:
    try:
        with open('file.log') as file:
            read_data = file.read()
    except FileNotFoundError as fnf_error:
        print(fnf_error)
finally:
    print('这句话，无论异常是否发生都会执行。')
```

抛出异常

Python 使用 raise 语句抛出一个指定的异常。

raise 语法格式如下:

raise [Exception [, args [, traceback]]]



以下实例如果 x 大于 5 就触发异常:

```
x = 10
if x > 5:
    raise Exception('x 不能大于 5。x 的值为: {}'.format(x))
```

执行以上代码会触发异常:

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    raise Exception('x 不能大于 5。x 的值为: {}'.format(x))
Exception: x 不能大于 5。x 的值为: 10
```

raise 唯一的一个参数指定了要被抛出的异常。它必须是一个异常的实例或者是异常的类（也就是 Exception 的子类）。
如果你只想知道这是否抛出了一个异常，并不想去处理它，那么一个简单的 raise 语句就可以再次把它抛出。

```
>>> try:
        raise NameError('HiThere') # 模拟一个异常。
    except NameError:
        print('An exception flew by!')
        raise

An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

用户自定义异常

你可以通过创建一个新的异常类来拥有自己的异常。异常类继承自 Exception 类，可以直接继承，或者间接继承，例如：

```
>>> class MyError(Exception):
        def __init__(self, value):
            self.value = value
        def __str__(self):
            return repr(self.value)

>>> try:
        raise MyError(2*2)
    except MyError as e:
        print('My exception occurred, value:', e.value)

My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

在这个例子中，类 Exception 默认的 __init__() 被覆盖。

当创建一个模块有可能抛出多种不同的异常时，一种通常的做法是为这个包建立一个基础异常类，然后基于这个基础类为不同的错误情况创建不同的子类：

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """
```

```
    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is not allowed
    """


    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

大多数的异常的名字都以"Error"结尾，就跟标准的异常命名一样。

定义清理行为

try 语句还有另外一个可选的子句，它定义了无论在任何情况下都会执行的清理行为。 例如:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

以上例子不管 try 子句里面有没有发生异常，finally 子句都会执行。

如果一个异常在 try 子句里（或者在 except 和 else 子句里）被抛出，而又没有任何的 except 把它截住，那么这个异常会在 finally 子句执行后被抛出。

下面是一个更加复杂的例子（在同一个 try 语句里包含 except 和 finally 子句）：

```
>>> def divide(x, y):
        try:
            result = x / y
        except ZeroDivisionError:
            print("division by zero!")
        else:
            print("result is", result)
        finally:
            print("executing finally clause")
```

```
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

预定义的清理行为

一些对象定义了标准的清理行为，无论系统是否成功的使用了它，一旦不需要它了，那么这个标准的清理行为就会执行。

下面这个例子展示了尝试打开一个文件，然后把内容打印到屏幕上:

```
for line in open("myfile.txt"):
    print(line, end="")
```

以上这段代码的问题是，当执行完毕后，文件会保持打开状态，并没有被关闭。

关键词 with 语句就可以保证诸如文件之类的对象在使用完之后一定会正确的执行他的清理方法:

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

以上这段代码执行完毕后，就算在处理过程中出问题了，文件 f 总是会关闭。

更多 with 关键字内容参考：Python with 关键字

相关内容

Python assert（断言）
Python with 关键字

Python3 面向对象

Python 从设计之初就已经是一门面向对象的语言，正因为如此，在 Python 中创建一个类和对象是很容易的。本章节我们将详细介绍 Python 的面向对象编程。

如果你以前没有接触过面向对象的编程语言，那你可能需要先了解一些面向对象语言的一些基本特征，在头脑里头形成一个基本的面向对象的概念，这样有助于你更容易的学习 Python 的面向对象编程。

接下来我们先来简单的了解下面向对象的一些基本特征。

面向对象技术简介

- **类(Class):** 用来描述具有相同的属性和方法的对象的集合。它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。
- **方法：**类中定义的函数。
- **类变量：**类变量在整个实例化的对象中是公用的。类变量定义在类中且在函数体之外。类变量通常不作为实例变量使用。
- **数据成员：**类变量或者实例变量用于处理类及其实例对象的相关的数据。
- **方法重写：**如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫方法的覆盖（override），也称为方法的重写。
- **局部变量：**定义在方法中的变量，只作用于当前实例的类。
- **实例变量：**在类的声明中，属性是用变量来表示的，这种变量就称为实例变量，实例变量就是一个用 self 修饰的变量。
- **继承：**即一个派生类（derived class）继承基类（base class）的字段和方法。继承也允许把一个派生类的对象作为一个基类对象对待。例如，有这样一个设计：一个 Dog 类型的对象派生自 Animal 类，这是模拟"是一个（is-a）"关系（例图，Dog 是一个 Animal）。
- **实例化：**创建一个类的实例，类的具体对象。
- **对象：**通过类定义的数据结构实例。对象包括两个数据成员（类变量和实例变量）和方法。

和其它编程语言相比，Python 在尽可能不增加新的语法和语义的情况下加入了类机制。

Python 中的类提供了面向对象编程的所有基本功能：类的继承机制允许多个基类，派生类可以覆盖基类中的任何方法，方法中可以调用基类中的同名方法。

对象可以包含任意数量和类型的数据。

类定义

语法格式如下：

class ClassName: <statement-1> . . . <statement-N>

类实例化后，可以使用其属性，实际上，创建一个类之后，可以通过类名访问其属性。

类对象

类对象支持两种操作：属性引用和实例化。

属性引用使用和 Python 中所有的属性引用一样的标准语法：**obj.name**。

类对象创建后，类命名空间中所有的命名都是有效属性名。所以如果类定义是这样:

实例(Python 3.0+)

```
#!/usr/bin/python3

class MyClass:
    """一个简单的类实例"""
    i = 12345
    def f(self):
        return 'hello world'

# 实例化类
x = MyClass()
```

```
# 访问类的属性和方法
print("MyClass 类的属性 i 为：", x.i)
print("MyClass 类的方法 f 输出为：", x.f())
```

以上创建了一个新的类实例并将该对象赋给局部变量 x，x 为空的对象。
执行以上程序输出结果为：
MyClass 类的属性 i 为： 12345
MyClass 类的方法 f 输出为： hello world

类有一个名为 __init__() 的特殊方法（**构造方法**），该方法在类实例化时会自动调用，像下面这样：

```
def __init__(self):
    self.data = []
```

类定义了 __init__() 方法，类的实例化操作会自动调用 __init__() 方法。如下实例化类 MyClass，对应的 __init__() 方法就会被调用：

```
x = MyClass()
```

当然， __init__() 方法可以有参数，参数通过 __init__() 传递到类的实例化操作上。例如：

实例(Python 3.0+)

```
#!/usr/bin/python3

class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart
x = Complex(3.0, -4.5)
print(x.r, x.i)   # 输出结果：3.0 -4.5
```

self 代表类的实例，而非类

类的方法与普通的函数只有一个特别的区别——它们必须有一个额外的**第一个参数名称**，按照惯例它的名称是 self。

```
class Test:
    def prt(self):
        print(self)
        print(self.__class__)

t = Test()
t.prt()
```

以上实例执行结果为：

```
<__main__.Test instance at 0x100771878>
__main__.Test
```

从执行结果可以很明显的看出，self 代表的是类的实例，代表当前对象的地址，而 self.class 则指向类。
self 不是 python 关键字，我们把他换成 runoob 也是可以正常执行的：

```
class Test:
    def prt(runoob):
        print(runoob)
```

```
        print(runoob.__class__)

t = Test()
t.prt()
```

以上实例执行结果为：

```
<__main__.Test instance at 0x100771878>
__main__.Test
```

在 Python 中，self 是一个惯用的名称，用于表示类的实例（对象）自身。它是一个指向实例的引用，使得类的方法能够访问和操作实例的属性。

当你定义一个类，并在类中定义方法时，第一个参数通常被命名为 self，尽管你可以使用其他名称，但强烈建议使用 self，以保持代码的一致性和可读性。

实例

```
class MyClass:
    def __init__(self, value):
        self.value = value

    def display_value(self):
        print(self.value)

# 创建一个类的实例
obj = MyClass(42)

# 调用实例的方法
obj.display_value() # 输出 42
```

在上面的例子中，self 是一个指向类实例的引用，它在 **__init__** 构造函数中用于初始化实例的属性，也在 **display_value** 方法中用于访问实例的属性。通过使用 self，你可以在类的方法中访问和操作实例的属性，从而实现类的行为。

类的方法

在类的内部，使用 **def** 关键字来定义一个方法，与一般函数定义不同，类方法必须包含参数 self，且为第一个参数，self 代表的是类的实例。

实例(Python 3.0+)

```
#!/usr/bin/python3

#类定义
class people:
    #定义基本属性
    name = ''
    age = 0
    #定义私有属性,私有属性在类外部无法直接进行访问
    __weight = 0
    #定义构造方法
    def __init__(self,n,a,w):
        self.name = n
        self.age = a
```

```
        self.__weight = w
    def speak(self):
        print("%s 说: 我 %d 岁。" %(self.name,self.age))

# 实例化类
p = people('runoob',10,30)
p.speak()
```

执行以上程序输出结果为:
runoob 说: 我 10 岁。

继承

Python 同样支持类的继承，如果一种语言不支持继承，类就没有什么意义。派生类的定义如下所示:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

子类（派生类 DerivedClassName）会继承父类（基类 BaseClassName）的属性和方法。

BaseClassName（实例中的基类名）必须与派生类定义在一个作用域内。除了类，还可以用表达式，基类定义在另一个模块中时这一点非常有用:

```
class DerivedClassName(modname.BaseClassName):
```

实例(Python 3.0+)

```
#!/usr/bin/python3

#类定义
class people:
    #定义基本属性
    name = ''
    age = 0
    #定义私有属性,私有属性在类外部无法直接进行访问
    __weight = 0
    #定义构造方法
    def __init__(self,n,a,w):
        self.name = n
        self.age = a
        self.__weight = w
    def speak(self):
        print("%s 说: 我 %d 岁。" %(self.name,self.age))

#单继承示例
class student(people):
    grade = ''
    def __init__(self,n,a,w,g):
        #调用父类的构函
```

```
        people.__init__(self,n,a,w)
        self.grade = g
    #覆写父类的方法
    def speak(self):
        print("%s 说: 我 %d 岁了，我在读 %d 年级"%(self.name,self.age,self.grade))



s = student('ken',10,60,3)
s.speak()
```

执行以上程序输出结果为:
ken 说: 我 10 岁了，我在读 3 年级

多继承

Python 同样有限的支持多继承形式。多继承的类定义形如下例:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

需要注意圆括号中父类的顺序，若是父类中有相同的方法名，而在子类使用时未指定，python 从左至右搜索 即方法在子类中未找到时，从左到右查找父类中是否包含方法。

实例(Python 3.0+)

```
#!/usr/bin/python3

#类定义
class people:
    #定义基本属性
    name = ''
    age = 0
    #定义私有属性,私有属性在类外部无法直接进行访问
    __weight = 0
    #定义构造方法
    def __init__(self,n,a,w):
        self.name = n
        self.age = a
        self.__weight = w
    def speak(self):
        print("%s 说: 我 %d 岁。" %(self.name,self.age))

#单继承示例
class student(people):
    grade = ''
    def __init__(self,n,a,w,g):
        #调用父类的构函
```

```
        people.__init__(self,n,a,w)
        self.grade = g
    #覆写父类的方法
    def speak(self):
        print("%s 说: 我 %d 岁了，我在读 %d 年级"%(self.name,self.age,self.grade))

#另一个类，多继承之前的准备
class speaker():
    topic = ''
    name = ''
    def __init__(self,n,t):
        self.name = n
        self.topic = t
    def speak(self):
        print("我叫 %s，我是一个演说家，我演讲的主题是 %s"%(self.name,self.topic))

#多继承
class sample(speaker,student):
    a =''
    def __init__(self,n,a,w,g,t):
        student.__init__(self,n,a,w,g)
        speaker.__init__(self,n,t)

test = sample("Tim",25,80,4,"Python")
test.speak()  #方法名同，默认调用的是在括号中参数位置排前父类的方法
```

执行以上程序输出结果为:
我叫 Tim，我是一个演说家，我演讲的主题是 Python

方法重写

如果你的父类方法的功能不能满足你的需求，你可以在子类重写你父类的方法，实例如下:

实例(Python 3.0+)

```
#!/usr/bin/python3

class Parent:        # 定义父类
  def myMethod(self):
     print ('调用父类方法')

class Child(Parent): # 定义子类
  def myMethod(self):
     print ('调用子类方法')

c = Child()          # 子类实例
c.myMethod()         # 子类调用重写方法
super(Child,c).myMethod() #用子类对象调用父类已被覆盖的方法
```

super() 函数是用于调用父类(超类)的一个方法。

执行以上程序输出结果为:

调用子类方法

调用父类方法

更多文档：
Python 子类继承父类构造函数说明

类属性与方法
类的私有属性
__private_attrs：两个下划线开头，声明该属性为私有，不能在类的外部被使用或直接访问。在类内部的方法中使用时 **self.__private_attrs**。

类的方法
在类的内部，使用 def 关键字来定义一个方法，与一般函数定义不同，类方法必须包含参数 **self**，且为第一个参数，**self** 代表的是类的实例。
self 的名字并不是规定死的，也可以使用 **this**，但是最好还是按照约定使用 **self**。

类的私有方法
__private_method：两个下划线开头，声明该方法为私有方法，只能在类的内部调用 ，不能在类的外部调用。**self.__private_methods**。

实例
类的私有属性实例如下：
实例(Python 3.0+)

```
#!/usr/bin/python3

class JustCounter:
    __secretCount = 0  # 私有变量
    publicCount = 0    # 公开变量

    def count(self):
        self.__secretCount += 1
        self.publicCount += 1
        print (self.__secretCount)

counter = JustCounter()
counter.count()
counter.count()
print (counter.publicCount)
print (counter.__secretCount)  # 报错，实例不能访问私有变量
```

执行以上程序输出结果为：

```
1
2
2
Traceback (most recent call last):
  File "test.py", line 16, in <module>
    print (counter.__secretCount)  # 报错，实例不能访问私有变量
AttributeError: 'JustCounter' object has no attribute '__secretCount'
```

类的私有方法实例如下：

实例(Python 3.0+)

```
#!/usr/bin/python3

class Site:
    def __init__(self, name, url):
        self.name = name       # public
        self.__url = url   # private

    def who(self):
        print('name  : ', self.name)
        print('url : ', self.__url)

    def __foo(self):          # 私有方法
        print('这是私有方法')

    def foo(self):            # 公共方法
        print('这是公共方法')
        self.__foo()

x = Site('菜鸟教程', 'www.runoob.com')
x.who()        # 正常输出
x.foo()        # 正常输出
x.__foo()      # 报错
```

以上实例执行结果:

```
root@iZ23mtq8bs1Z:~/test# python3 test.py
name  :  菜鸟教程
url :  www.runoob.com
这是公共方法
这是私有方法
Traceback (most recent call last):
  File "test.py", line 22, in <module>
    x.__foo()
AttributeError: 'Site' object has no attribute '__foo'
```

外部不能调用私有方法

类的专有方法：

- **__init__** : 构造函数，在生成对象时调用
- **__del__** : 析构函数，释放对象时使用
- **__repr__** : 打印，转换
- **__setitem__** : 按照索引赋值
- **__getitem__**: 按照索引获取值
- **__len__**: 获得长度
- **__cmp__**: 比较运算
- **__call__**: 函数调用
- **__add__**: 加运算
- **__sub__**: 减运算
- **__mul__**: 乘运算
- **__truediv__**: 除运算
- **__mod__**: 求余运算
- **__pow__**: 乘方

运算符重载

Python 同样支持运算符重载，我们可以对类的专有方法进行重载，实例如下：

实例(Python 3.0+)

```
#!/usr/bin/python3

class Vector:
   def __init__(self, a, b):
      self.a = a
      self.b = b

   def __str__(self):
      return 'Vector (%d, %d)' % (self.a, self.b)

   def __add__(self,other):
      return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print (v1 + v2)
```

以上代码执行结果如下所示:

```
Vector(7,8)
```

Python3 命名空间和作用域

命名空间

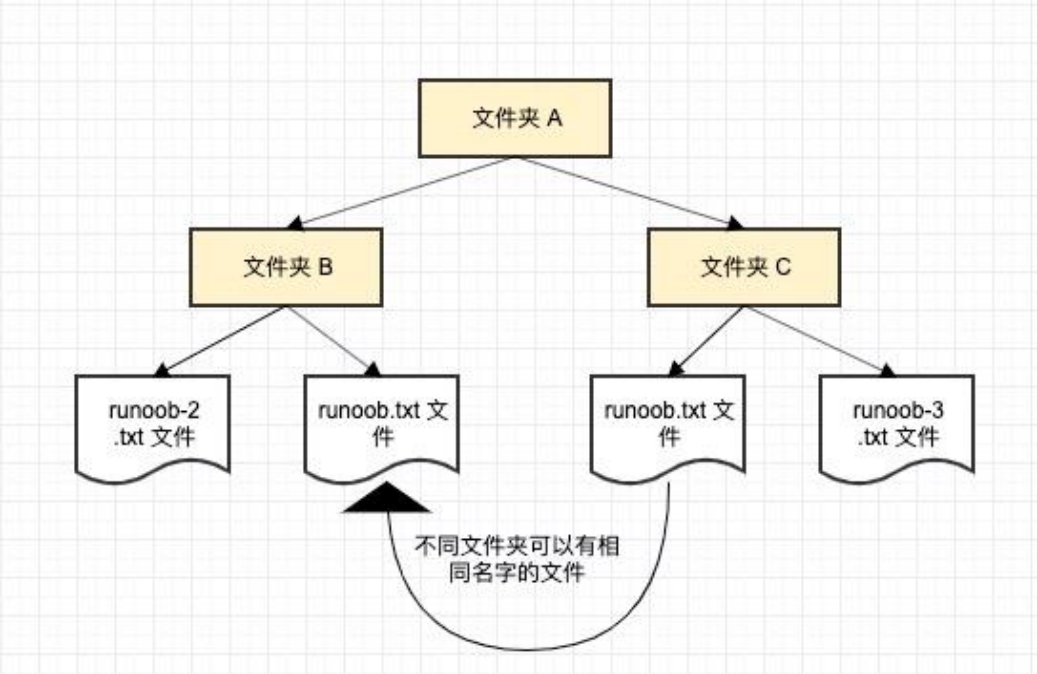
先看看官方文档的一段话:

A namespace is a mapping from names to objects.Most namespaces are currently implemented as Python dictionaries。

命名空间(Namespace)是从名称到对象的映射，大部分的命名空间都是通过 Python 字典来实现的。

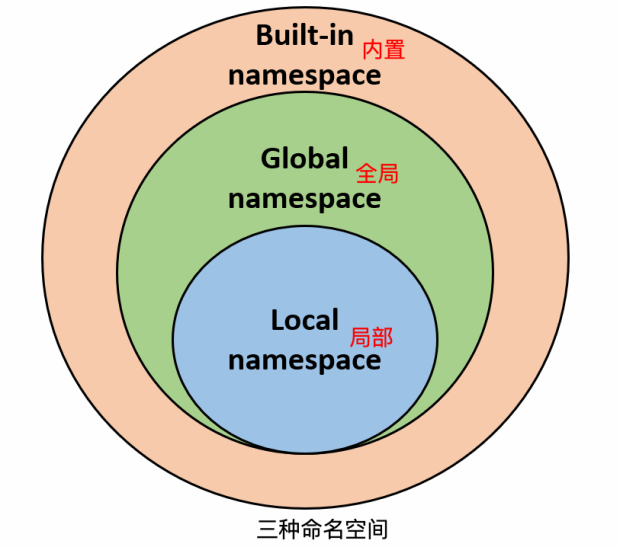
命名空间提供了在项目中避免名字冲突的一种方法。各个命名空间是独立的，没有任何关系的，所以一个命名空间中不能有重名，但不同的命名空间是可以重名而没有任何影响。

我们举一个计算机系统中的例子，一个文件夹(目录)中可以包含多个文件夹，每个文件夹中不能有相同的文件名，但不同文件夹中的文件可以重名。



一般有三种命名空间:

- **内置名称（built-in names）**， Python 语言内置的名称，比如函数名 abs、char 和异常名称 BaseException、Exception 等等。
- **全局名称（global names）**，模块中定义的名称，记录了模块的变量，包括函数、类、其它导入的模块、模块级的变量和常量。
- **局部名称（local names）**，函数中定义的名称，记录了函数的变量，包括函数的参数和局部定义的变量。(类中定义的也是)



三种命名空间

命名空间查找顺序:
假设我们要使用变量 runoob，则 Python 的查找顺序为：**局部的命名空间 -> 全局命名空间 -> 内置命名空间**。

如果找不到变量 runoob，它将放弃查找并引发一个 NameError 异常:
NameError: name 'runoob' is not defined。

命名空间的生命周期:
命名空间的生命周期取决于对象的作用域，如果对象执行完成，则该命名空间的生命周期就结束。
因此，我们无法从外部命名空间访问内部命名空间的对象。

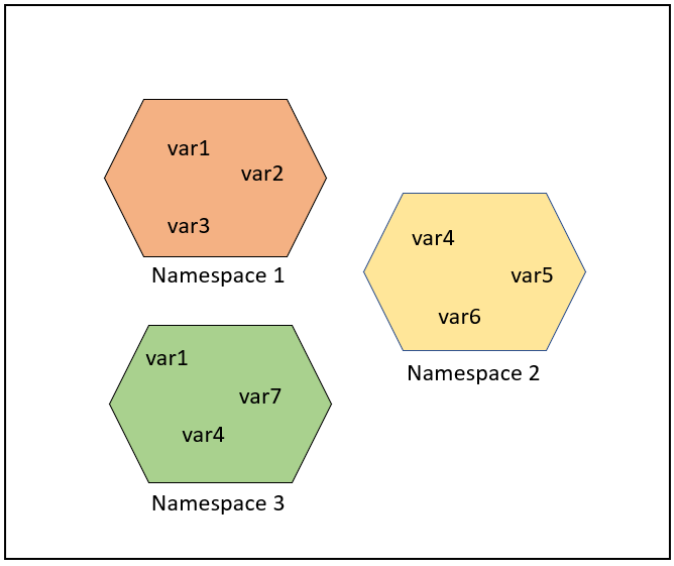
实例

```
# var1 是全局名称
var1 = 5
def some_func():

    # var2 是局部名称
    var2 = 6
    def some_inner_func():

        # var3 是内嵌的局部名称
        var3 = 7
```

如下图所示，相同的对象名称可以存在于多个命名空间中。



作用域

A scope is a textual region of a Python program where a namespace is directly accessible. "Directly accessible" here means that an unqualified reference to a name attempts to find the name in the namespace.
作用域就是一个 Python 程序可以直接访问命名空间的正文区域。
在一个 python 程序中，直接访问一个变量，会从内到外依次访问所有的作用域直到找到，否则会报未定义的错误。
Python 中，程序的变量并不是在哪个位置都可以访问的，访问权限决定于这个变量是在哪里赋值的。

变量的作用域决定了在哪一部分程序可以访问哪个特定的变量名称。Python 的作用域一共有 4 种，分别是:
有四种作用域:

- **L（Local）**：最内层，包含局部变量，比如一个函数/方法内部。
- **E（Enclosing）**：包含了非局部(non-local)也非全局(non-global)的变量。比如两个嵌套函数，一个函数（或类） A 里面又包含了一个函数 B ，那么对于 B 中的名称来说 A 中的作用域就为 nonlocal。
- **G（Global）**：当前脚本的最外层，比如当前模块的全局变量。
- **B（Built-in）**： 包含了内建的变量/关键字等，最后被搜索。

规则顺序： **L –> E –> G –> B**。
在局部找不到，便会去局部外的局部找（例如闭包），再找不到就会去全局找，再者去内置中找。


```
g_count = 0  # 全局作用域
def outer():
    o_count = 1  # 闭包函数外的函数中
    def inner():
        i_count = 2  # 局部作用域
```

内置作用域是通过一个名为 builtin 的标准模块来实现的，但是这个变量名自身并没有放入内置作用域内，所以必须导入这个文件才能够使用它。在 Python3.0 中，可以使用以下的代码来查看到底预定义了哪些变量:

```
>>> import builtins
>>> dir(builtins)
```

Python 中只有模块（module），类（class）以及函数（def、lambda）才会引入新的作用域，其它的代码块（如 if/elif/else/、try/except、for/while 等）是不会引入新的作用域的，也就是说这些语句内定义的变量，外部也可以访问，如下代码:

```
>>> if True:
...  msg = 'I am from Runoob'
...
>>> msg
'I am from Runoob'
>>>
```

实例中 msg 变量定义在 if 语句块中，但外部还是可以访问的。
如果将 msg 定义在函数中，则它就是局部变量，外部不能访问:

```
>>> def test():
...     msg_inner = 'I am from Runoob'
...
>>> msg_inner
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'msg_inner' is not defined
>>>
```

从报错的信息上看，说明了 msg_inner 未定义，无法使用，因为它是局部变量，只有在函数内可以使用。

全局变量和局部变量

定义在函数内部的变量拥有一个局部作用域，定义在函数外的拥有全局作用域。

局部变量只能在其被声明的函数内部访问，而全局变量可以在整个程序范围内访问。调用函数时，所有在函数内声明的变量名称都将被加入到作用域中。如下实例：

实例(Python 3.0+)

```
#!/usr/bin/python3

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

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

以上实例输出结果：

```
函数内是局部变量 :  30
函数外是全局变量 :  0
```

global 和 nonlocal 关键字

当内部作用域想修改外部作用域的变量时，就要用到 global 和 nonlocal 关键字了。

以下实例修改全局变量 num：

实例(Python 3.0+)

```
#!/usr/bin/python3

num = 1
def fun1():
    global num  # 需要使用 global 关键字声明
    print(num)
    num = 123
    print(num)
fun1()
print(num)
```

以上实例输出结果：

```
1
123
123
```

如果要修改嵌套作用域（enclosing 作用域，外层非全局作用域）中的变量则需要 nonlocal 关键字了，如下实例：

实例(Python 3.0+)

```
#!/usr/bin/python3
```

```
def outer():
    num = 10
    def inner():
        nonlocal num   # nonlocal 关键字声明
        num = 100
        print(num)
    inner()
    print(num)
outer()
```

以上实例输出结果：

```
100
100
```

另外有一种特殊情况，假设下面这段代码被运行：

实例(Python 3.0+)

```
#!/usr/bin/python3

a = 10
def test():
    a = a + 1
    print(a)
test()
```

以上程序执行，报错信息如下：

```
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    test()
  File "test.py", line 5, in test
    a = a + 1
UnboundLocalError: local variable 'a' referenced before assignment
```

错误信息为局部作用域引用错误，因为 test 函数中的 a 使用的是局部，未定义，无法修改。

修改 a 为全局变量：

实例

```
#!/usr/bin/python3

a = 10
def test():
    global a
    a = a + 1
    print(a)
test()
```

执行输出结果为：

```
11
```

也可以通过函数参数传递：

实例(Python 3.0+)

```
#!/usr/bin/python3

a = 10
def test(a):
    a = a + 1
    print(a)
test(a)
```

执行输出结果为:

```
11
```

Python3 标准库概览

Python 标准库非常庞大，所提供的组件涉及范围十分广泛，使用标准库我们可以让您轻松地完成各种任务。

以下是一些 Python3 标准库中的模块：

- os 模块：os 模块提供了许多与操作系统交互的函数，例如创建、移动和删除文件和目录，以及访问环境变量等。
- sys 模块：sys 模块提供了与 Python 解释器和系统相关的功能，例如解释器的版本和路径，以及与 stdin、stdout 和 stderr 相关的信息。
- time 模块：time 模块提供了处理时间的函数，例如获取当前时间、格式化日期和时间、计时等。
- datetime 模块：datetime 模块提供了更高级的日期和时间处理函数，例如处理时区、计算时间差、计算日期差等。
- random 模块：random 模块提供了生成随机数的函数，例如生成随机整数、浮点数、序列等。
- math 模块：math 模块提供了数学函数，例如三角函数、对数函数、指数函数、常数等。
- re 模块：re 模块提供了正则表达式处理函数，可以用于文本搜索、替换、分割等。
- json 模块：json 模块提供了 JSON 编码和解码函数，可以将 Python 对象转换为 JSON 格式，并从 JSON 格式中解析出 Python 对象。
- urllib 模块：urllib 模块提供了访问网页和处理 URL 的功能，包括下载文件、发送 POST 请求、处理 cookies 等。

操作系统接口

os 模块提供了不少与操作系统相关联的函数，例如文件和目录的操作。

实例

```
import os

# 获取当前工作目录
current_dir = os.getcwd()
print("当前工作目录:", current_dir)

# 列出目录下的文件
files = os.listdir(current_dir)
print("目录下的文件:", files)
```

建议使用 **import os** 风格而非 **from os import ***，这样可以保证随操作系统不同而有所变化的 **os.open()** 不会覆盖内置函数 open()。

在使用 os 这样的大型模块时内置的 dir() 和 help() 函数非常有用:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

针对日常的文件和目录管理任务，:mod:shutil 模块提供了一个易于使用的高级接口:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```

文件通配符

glob 模块提供了一个函数用于从目录通配符搜索中生成文件列表:

实例

```
>>> import glob
```

```
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

命令行参数

通用工具脚本经常调用命令行参数。这些命令行参数以链表形式存储于 sys 模块的 argv 变量。例如在命令行中执行 "python demo.py one two three" 后可以得到以下输出结果:

实例

```
>>> import sys
>>> print(sys.argv)
['demo.py', 'one', 'two', 'three']
```

错误输出重定向和程序终止

sys 还有 stdin，stdout 和 stderr 属性，即使在 stdout 被重定向时，后者也可以用于显示警告和错误信息。

实例

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

大多脚本的定向终止都使用 **sys.exit()**。

字符串正则匹配

re 模块为高级字符串处理提供了正则表达式工具。对于复杂的匹配和处理，正则表达式提供了简洁、优化的解决方案:

实例

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

如果只需要简单的功能，应该首先考虑字符串方法，因为它们非常简单，易于阅读和调试:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

数学

math 模块为浮点运算提供了对底层 C 函数库的访问:

实例

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

random 提供了生成随机数的工具。

实例

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)   # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float
```

```
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

访问 互联网

有几个模块用于访问互联网以及处理网络通信协议。其中最简单的两个是用于处理从 urls 接收的数据的 urllib.request 以及用于发送电子邮件的 smtplib:

实例

```
>>> from urllib.request import urlopen
>>> for line in urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     line = line.decode('utf-8')  # Decoding the binary data to text.
...     if 'EST' in line or 'EDT' in line:  # look for Eastern Time
...         print(line)

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

注意第二个例子需要本地有一个在运行的邮件服务器。

日期和时间

datetime 模块为日期和时间处理同时提供了简单和复杂的方法。

支持日期和时间算法的同时，实现的重点放在更有效的处理和格式化输出。

实例

```
import datetime

#获取当前日期和时间
current_datetime = datetime.datetime.now()
print(current_datetime)

# 获取当前日期
current_date = datetime.date.today()
print(current_date)

# 格式化日期
formatted_datetime = current_datetime.strftime("%Y-%m-%d %H:%M:%S")
print(formatted_datetime)  # 输出：2023-07-17 15:30:45
```

输出结果为：

```
2023-07-17 18:37:56.036914
2023-07-17
2023-07-17 18:37:56
```

该模块还支持时区处理:

```
>>> # 导入了 datetime 模块中的 date 类
>>> from datetime import date
>>> now = date.today()  # 当前日期
>>> now
datetime.date(2023, 7, 17)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'07-17-23. 17 Jul 2023 is a Monday on the 17 day of July.'

>>> # 创建了一个表示生日的日期对象
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday  # 计算两个日期之间的时间差
>>> age.days          # 变量 age 的 days 属性，表示时间差的天数
21535
```

数据压缩

以下模块直接支持通用的数据打包和压缩格式：zlib，gzip，bz2，zipfile，以及 tarfile。

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

性能度量

有些用户对了解解决同一问题的不同方法之间的性能差异很感兴趣。Python 提供了一个度量工具，为这些问题提供了直接答案。

例如，使用元组封装和拆封来交换元素看起来要比使用传统的方法要诱人的多,timeit 证明了现代的方法更快一些。

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

相对于 timeit 的细粒度，:mod:profile 和 pstats 模块提供了针对更大代码块的时间度量工具。

测试模块

开发高质量软件的方法之一是为每一个函数开发测试代码，并且在开发过程中经常进行测试

doctest 模块提供了一个工具，扫描模块并根据程序中内嵌的文档字符串执行测试。

测试构造如同简单的将它的输出结果剪切并粘贴到文档字符串中。

通过用户提供的例子，它强化了文档，允许 doctest 模块确认代码的结果是否与文档一致:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod()   # 自动验证嵌入测试
```

unittest 模块不像 doctest 模块那么容易使用，不过它可以在一个独立的文件里提供一个更全面的测试集:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        self.assertRaises(ZeroDivisionError, average, [])
        self.assertRaises(TypeError, average, 20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

以上我们看到的只是 Python3 标准库中的一部分模块，还有很多其他模块可以在官方文档中查看完整的标准库文档：https://docs.python.org/zh-cn/3/library/index.html