

Pycharm使用

在这里，我们打开PyCharm，它是一种Python IDE(集成开发环境)。

在这个软件中，我们来学习Python的基础知识。

首先，我们先在软件中，打开Python控制台或者命令程序。


Python控制台打开之后，会直接进入交互式环境，显示提示符“>>>”。

而命令程序则会显示DOS的命令提示符，我们需要输入“Python”或者“Python3”进入交互式环境。

此处可参考[《Python \(3.6.1\) 简易安装教程》](#)。

在交互式环境中，我们可以尝试输入“1+1”，并按下回车键。

这个时候，我们能够看到，在我们输入内容的下一行，出现了刚才算式的运算结果。



```
Terminal
+ Microsoft Windows [版本 6.1.7601]
X 版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

D:\MyProject>Python3
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 18:41:36) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 1+1
2
>>>
```

这就是交互式环境，它自动会根据用户输入的内容反馈结果。

那么，接下来我们再尝试输入一串文字，比如：小楼好帅！

这个时候大家能够看到，程序给出了错误提示“SyntaxError:invalid character in identifier”。

这个提示的意思是“语法错误：标识符包含无效字符”

标识符可以理解为名称。

也就是说，当我们直接输入一串字符，程序会认为输入的是一个名称。

当这些字符不符合名称的命名规范，就给我们提示了错误。

我们暂且不管这个名称如何符合要求。

我们先来解决如何输入一串字符时，能够正确的显示出来。

这里有三种格式：

‘小楼好帅！’
“小楼好帅！”
“ ‘ 小楼好帅！ ’ ”

这里需要注意：

- 1、以上的单引号、双引号和三引号都是英文半角符号；**
- 2、三引号是三个单引号；**
- 3、各种引号不可混用；**
- 4、三引号用于输入多行字符。**

当我们改用上方的三种格式输入字符，并回车之后，就能够正常显示结果了。

以上是在交互式环境中，我们能够即时看到结果。

如果在PyCharm的工作区中编写代码，是不能够实时得到运行结果的。

我们在编写完代码之后，需要手动运行才能够看到结果。

手动运行的快捷键默认是<Ctrl>+<Shift>+<F10>。

但是，如果我们输入以上内容，然后手动运行代码，会发现并没有显示想要的结果。

如果想看到想要的结果，我们需要使用一个方法。

这个方法是这样的：

```
print(1+1)
print( '小楼好帅! ' )
```

注意：英文单词严格区分大小写。

然后，通过手动运行，我们就能够在界面下方看到正确的结果。

代码中第一行注释“# -*- coding:UTF-8 -*-”是用于告诉Python解释器，当前的代码是使用“UTF-8”的编码，使用这种编码能够支持中文字符。

实际上，Python3默认的源文件编码就是“UTF-8”，所以上面这一句注释写不写都无所谓，不过据说写上显得逼格高（意思就是你能从Python2一直玩到Python3了）。

PyCharm2017系统的默认编码类型是“GBK”，这种编码类型也支持中文字符。

并且，在PyCharm2017中，我们也可以在文件（File）-系统默认设置（Default Settings）-编辑器（Editor）-文件编码（File Encodings）中，设置全局编码类型、项目编码类型以及属性文件的编码类型。新创建的项目将会自动采用默认设置中的编码类型。

而对于当前已经创建的项目，可以在文件（File）-设置（Settings）中进行和默认设置同样的设置，改变编码类型。

另外，**“#”为单行注释的标记，在编写代码时，我们也可以在代码后方或代码行之间撰写注释**，在后文中大家能够见到。

特别说明：编写代码时，如果有些代码不需要运行可以把它们注释，操作为选中需要注释的代码，通过快捷键Ctrl+/进行注释。

最后，再做一些说明补充。

print('输出内容')：这是一个让我们在运行代码时，呈现所需显示内容的方法。

这个方法的内部代码是什么，我们无需关心，我们只需要知道这个方法如何使用。

那么，大家一定听说过“函数”这个词语，其实，在编程中方法的另外一个常用称呼就是“函数”。

也就是说，print('输出内容')就是一个函数。

函数是帮助我们在编程的时候进行一些程序运算的。

所以，在使用函数时，函数的结构我们能够看到，是下面这个样子。67667

这里大家能够看到，括号中的内容是参数，也就是参与函数运算的数值。

例如，如果有一个加法的函数，我们肯定需要输入两个加数作为参数来参与运算。

就像：plus(3,5)

通过这个假设的函数大家能够看到，一个函数的参数不一定仅有一个，也可能是多个。一个函数包含多少个参数，取决于该函数的自身运算需求。

字符串的操作

1、连接字符串

我们在PyCharm的编辑区中分别创建三个变量，并存入相应的值。

```
1  userName = '小楼'
2  attribute = '身高'
3  value = '182CM'
```

接下来，我们使用print()函数进行显示输出。

```
1  print(userName + attribute + value)
```

运行代码之后，显示的结果为：

除了加号这种连接方法之外，如果只是两段字符，大家还可以尝试下面这种方法。

```
1  str1 = '5月'
2  str2 = '21日'
3  str1 += str2
4  print(str1)
```

运行代码之后，显示的结果为：

上方代码中【str1 += str2】等同于【str1 = str1 + str2】。

其实，不仅仅字符串可以进行这样的运算，数字也可以。

例如：

```
1  num = 5
2  num += 1
3  print(num)
```

运行代码之后，显示的结果为：

上方代码中【num += 1】等同于【num = num + 1】。

2、获取字符串长度（字节数量）。

获取字符串长度的方法是：len()

例如，我们想知道“小楼是一个很帅的帅哥！”这句话一共多少个字节。

我们可以把这句话作为len函数的参数进行计算，然后保存到变量“l”中。

示例代码：

```
1 l = len('小楼是一个很帅的帅哥！')
2 print (l)
3
```

运行代码之后，显示的结果为：

注意，当我们对中文进行编码，采用“UTF-8”编码类型时，一个汉字的字节数量是3。

示例代码：

```
1 l = len('小楼是一个很帅的帅哥！'.encode('UTF-8'))
2 print (l)
```

运行代码之后，显示的结果为：

另外，如果对中文进行编码，采用“GBK”编码类型时，一个汉字的字节数量是2。

3、截取字符串

如果，我们需要对一个字符串进行截取，我们可以对字符串进行切片。

切片操作是将字符串赋值到一个变量，然后通过输入截取的起始位置、终止位置以及方向与间隔，对字符串进行不同方式的截取。

例如：

```
1 str[3:9:2]
```

这段代码表示对“str”这个变量中的字符串进行切片操作，“3”表示切片的起始位置为第4个字符（字符串中字符的索引位置从0开始），“9”表示切片的终止位置（终止位置不会被截取），“2”表示从左向右每隔1位进行截取（负数表示从右至左截取）。

接下来，为大家展示更详尽的切片操作。

我们为变量“str”赋值一个字符串“123456789”，然后切片操作的代码如下：

```
1 str = '123456789'
```

```

2
3 print (str[:]) # 截取全部字符
4 print (str[2]) # 截取第3位字符
5 print (str[:3]) # 截取第1位到第4位之前的字符
6 print (str[2:5]) # 截取第3位到第6位之前的字符
7 print (str[5:]) # 截取第6个字符到末尾的字符
8 print (str[5:3]) # 从第6个字符开始每隔2个字符截取到末尾的字符
9 print (str[-1]) # 截取倒数第一个字符
10 print (str[:-3]) # 截取第1位到倒数第3位之前的字符
11 print (str[-3:-1]) # 截取倒数第3位到倒数第1位之前的字符
12 print (str[-3:]) # 截取倒数第3位到末尾的字符
13 print (str[::-1]) # 倒序截取全部字符
14 print (str[::-2]) # 倒序从倒数第1位开始每间隔1个字符截取字符
15 print (str[5::-1]) # 倒序截取第6位到开始的字符
16 print (str[5:2:-1]) # 倒序截取第6位到第3位之后的字符
17 print (str[:-6:-1]) # 倒序截取末尾到倒数第6位之后的字符
18 print (str[-1:-6:-1]) # 倒序截取末尾到倒数第6位之后的字符
19 print (str[-2:-8:-2]) # 倒序从倒数第2位开始间隔1位截取到倒数第8位之后的字符

```

提示：上方代码中“#”为单行注释标记符号，该符号右侧为注释内容，不会被程序执行。

运行代码之后，显示的结果为：

通过代码中的中文注释内容与运行结果进行对比，大家就能够理解切片操作的规则。

4、重复字符串

在Python中，重复字符串的操作很简单。

大家可以尝试运行下方这段代码：

```

1 print ('小楼好帅！\n' * 9)

```

提示：上方代码中的“\n”是换行符，大家可以删除换行符查看代码的运行效果！

运行代码之后，显示的结果为：

大家能够看到，“小楼好帅！\n”这段字符被重复了9次。

也就是说，当我们对一个字符串进行“*”运算的时候，这个字符串能够被重复；重复的次数取决于“*”后方的数值。

5、字符串大小写转换

大小写转换的方法有以下几种：

- upper()：字符串全部字符转换为大写

- lower(): 字符串全部字符转换为小写
- swapcase(): 字符串全部字符大小写互换
- capitalize(): 字符串首个单词首字母大写
- title(): 字符串中全部单词首字母大写

这些方法均不用写入参数，直接调用。

这里我们简单的了解一个概念-对象。

对象就是一个具体的事物，比如一个人、一张桌子或者一台电脑都是一个对象。

不用着急，这个概念会在之后越来越明确。

Python中一个字符串或者一个变量也是对象。（其实Python中万物皆对象）

如果想对一个字符串进行一些操作，我们就需要通过对象去调用相应的方法。

就好像，小楼测量身高，小楼就是一个对象，测量是方法，而测出的数值数值是方法的返回值。

接下来，回到刚才我们看到的这些方法。

我们可以将字符串保存在变量中，通过变量调用这些函数。（当然也可以通过字符串直接调用，因为字符串自身就是对象，变量保存了字符串之后，变量等同于这个字符串对象）

通过对象调用函数或者方法，需要通过操作符 “.” 来完成。

例如：将字符串对象‘Abc’ 中的大写字母全部转换成小写字母，就需要字符串对象调用转换字符为小写的方法，也就是：‘Abc’.lower()

提示：把 “.” 读成 “的” 试试看，是不是更容易理解？

接下来，我们看一下如何使用前面提到的那些方法。

示例代码：

```
1 s = 'i need PyCharm.'  
2  
3 print (s.upper()) # 全部字符转换为大写  
4 print (s.lower()) # 全部字符转换为小写  
5 print (s.swapcase()) # 全部字符大小写互换  
6 print (s.capitalize()) # 字符串首个单词首字母大写  
7 print (s.title()) # 字符串中全部单词首字母大写
```

运行代码之后，显示的结果为：

6、替换部分字符

替换部分字符的方法是 “replace(old,new,count)” 。

如上所述，这个函数有三个参数：

- old: 表示需要被替换的字符或字符串；
- new: 表示替换后的新字符或字符串；
- count: 表示替换的次数，此参数可省略；如果省略表示替换所有需要被替换的字符或字符串。

示例代码：

```
1 s = '小楼好帅，我好喜欢他！'
2
3 print (s.replace('喜欢','崇拜'))
4 print (s.replace('好','很',1))
5 print (s.replace('好','很',2))
```

运行代码之后，显示的结果为：

大家能够看到，受第3个参数的影响，第2个print语句的执行结果中，只有前面的“好”被替换为“很”。

7、原始字符串

假如在代码中，我们使用的字符串是一个路径，有可能会出现下面这种情形。

```
1 path = 'D:\new_project'
2 print (path)
```

这段代码运行之后，结果如下：

这就尴尬了！

因为，在路径的字符串中，包含了“\n”。

在本篇开始，我们就看到过“\n”出现，它是一个换行符。

那么，如何化解这份突如其来的尴尬？

大家尝试再加入一个“\”。

```
1 path = 'D:\\new_project'
2 print (path)
```

上面这段代码的运行结果就正常了。

由此可见，“\\”是转义字符，它能够将第2个“\”从转义字符转回普通字符，从而“\n”就不再起到换行符的作用。

这样操作虽然简单，但是遇到下方这个路径，看起来就会有些麻烦！

```
1 path = 'D:\new_project\test\nt\files\data'
```

如果想正常使用这个路径，我们需要加上多个“\\”。

```
1 path = 'D:\\new_project\\test\\nt\\files\\data'
```

实际上，这样的路径我们可以将它转换为原始字符串，转换方法是在字符串之前写一个“r”。

```
1 path = r'D:\new_project\test\nt\files\data'
```

这样，我们就无须添加多个“\”，也能够正常使用这个路径。

不过，有的时候，我们使用的路径需要是“\”结尾。

例如：D:\new_project\test\nt\files\data\

这时要注意，原始字符串不能以“\”结尾，否则会抛出异常。

另外，也不能通过转义字符转换最后的“\”，程序会在路径末尾原样输出两个“\”。

示例代码：（错误示例）

```
1 path = r'D:\new_project\test\nt\files\data\' # 提示异常
```

```
1 path = r'D:\new_project\test\nt\files\data\\' # 末尾保持原样
```

当我们遇到这种需求时，我们需要对末尾的“\”单独进行转义处理，并和前面的字符串连接。

示例代码：（正确示例）

```
1 path = r'D:\new_project\test\nt\files\data' + '\\'  
2 print (path)
```

上面的代码，大家可以看到是两部分字符串并排在一起，这种书写方法是符合规则的，程序会自动把多个并列的字符串连接到一起。

关于这种操作，我们再来看一个例子。

示例代码：

```
1 name = '小楼'  
2 path = name + '是' + '一个' + '很帅很帅的' + '大帅哥' + '！'  
3 print (path) # 显示输出结果为：小楼是一个很帅很帅的大帅哥！
```

本节教程是字符串操作的一部分，在下一节教程中将继续讲解关于字符串的相关操作。

8、去除字符串两侧指定内容

字符串去除两侧指定内容的方法有三种：

- `strip(chars)`：去除字符串两侧的指定内容，并且，可以同时去除多个相同的指定内容；参数`chars`为指定的一个或多个字符，不填入该参数则去除字符串两侧所有空格。
- `lstrip(chars)`：去除字符串左侧的指定内容，并且，可以同时去除多个相同的指定内容；参数`chars`为指定的一个或多个字符，不填入该参数则去除字符串左侧所有空格。
- `rstrip(chars)`：去除字符串右侧的指定内容，并且，可以同时去除多个相同的指定内容；参数`chars`为指定的一个或多个字符，不填入该参数则去除字符串右侧所有空格。

大家可以尝试运行下方这段代码：

```
1  str1 = '  人人为我 我为人人  '
2  str2 = '人人为我 我为人人'
3
4  print (str1)
5  print (str1.strip())
6  print (str1.lstrip())
7  print (str1.rstrip())
8  print (str2.strip('人'))
9  print (str2.lstrip('人'))
10 print (str2.rstrip('人'))
11 print (str2.strip('人人为'))
```

运行代码之后，显示的结果为：

9、字符串查询

字符串查询的方法有两种：

第一种：`index(sub,start,end)`和`rindex(sub,start,end)`

`index`方法是从左至右查询，`rindex`方法是从右至左查询；

参数`sub`是指被查询的字符或字符串，参数`start`是查询的起始位置，参数`end`是查询的终止位置（终止位置不在查询范围内）；

参数`start`和参数`end`可以同时省略，这时为查询字符串中全部字符；也可以只省略参数`end`，表示查询范围为起始位置至末尾。

大家可以尝试运行下方这段代码：

```
1  s = '人人为我，我为人人。'
2
3  print (s.index('人')) # 从左向右查询全部字符
4  print (s.index('人',2)) # 从左侧第3个字符开始向右查询至末尾
```

```
5 print (s.rindex('人')) # 从右向左查询全部字符
6 print (s.rindex('人',0,8)) # 从右侧第9个字符之前向左查询至首位
```

运行代码之后，显示的结果为：

第二种：find(sub,start,end)和rfind(sub,start,end)

这两个函数和index函数以及rindex函数作用相同，并且用法相同。

但是，当没有查询结果时，index函数与rindex函数会提示错误：ValueError: substring not find（值错误：未发现被查询的字符串）。

而另外的两个函数，find函数和rfind函数，在没有查询结果时，返回值为-1。

大家可以尝试运行下方这段代码：

```
1 s = '人人为我，我为人人。'
2
3 print (s.find('地'))
4 print (s.index('地'))
```

运行代码之后，显示的结果为：

10、字符串中字符的计数

字符串的计数方法是count(x,start,end)，这个函数可以统计字符串中被查询的字符或字符串出现的次数。

参数x是指被查询的字符或字符串，参数start是查询的起始位置，参数end是查询的终止位置（终止位置不在查询范围内）；

参数start和参数end可以同时省略，这时为查询字符串中全部字符；也可以只省略参数end，表示查询范围为起始位置至末尾。

大家可以尝试运行下方这段代码：

```
1 s = '人人为我，我为人人。'
2
3 print (s.count('人')) # 计算字符串中被查询字符或字符串出现的次数
4 print (s.count('人人')) # 计算字符串中被查询字符或字符串出现的次数
5 print (s.count('人',0,8)) # 计算从字符串首位到第9个字符（不含第9个字符）之间被查询字符或字符串出现的次数
```

运行代码之后，显示的结果为：

获取手动输入内容，与字符串的格式化

首先，如果我们想获取手动输入的内容，我们需要使用input()方法。

input(prompt)：这个函数能够获取手动输入的内容，并且把内容转换为字符串类型；参数prompt为获取输入时需要显示的提示内容。

大家可以尝试运行下方这段代码：

```
1 id = input('请输入您的身份证号：')
2 print (id)
```

运行代码之后，显示的结果为：

通过这个尝试，大家能够看到，当我们运行程序时，程序会显示提示，输入的内容会变成绿色斜体的文字显示在后方。

这个时候，程序的执行暂停。

当我们输入内容“110115199001122511”，并按下回车键之后，程序才会继续运行print语句，将变量id中保存的内容显示输出。

那么，如果我们想在按下回车之后，程序显示输出“您的出生日期为：19900112”这样的内容呢？

大家可以继续尝试运行下方这段代码：

```
1 id = input('请输入您的身份证号：')
2 print ('您的出生日期为：%s' % id[6:14])
```

运行代码之后，显示的结果为：

通过上面的代码，就得到了我们想要的结果。

大家观察一下，刚才运行的代码，能够发现在print函数中我们输入的字符串中嵌入了一个“%s”，并且，“在字符串之后写入了一个“%”，最后才是我们对变量id进行切片操作获取的出生日期内容。我们可以这么理解，“%s”是一个格式符，其中“s”表示字符串，也就是说“%s”表示这个格式符所在的位置内容必须用字符串替代；而接下来的一个“%”表示进行格式化的操作，也就是要把这个符号后方的内容替换掉前面的“%s”，来形成我们想要的字符串内容。

接下来，我们再来看，如何在一个字符串中添加多个格式符进行格式化。

比如，刚才的显示输出内容我们换为另外一种呈现方式：“您的出生日期为：1990年01月12日”。

大家可以尝试运行下方这段代码：

```
1 id = input('请输入您的身份证号：')
```

```
2 print('您的出生日期为: %s年%s月%s日' % (id[6:10], id[10:12], id[12:14]))
```

运行代码之后，显示的结果为：

通过上方的代码，大家能够看到，前面有三个格式符，分别与后方括号中的三个对变量id切片操作的结果相对应。

也就是说，当一段字符串进行格式化时，如果有包含多个格式符，就需要在“%”操作符之后添加一个括号，括号中写入多个字符串内容并以逗号分隔。

再次强调，以上代码中的符号必须为英文半角符号，切勿使用中文符号或英文全角符号。(字符串中包含的冒号除外)

另外，在Python中，对字符串的格式化我们还可以使用format()方法。

format(args,kwarg): 对字符串进行格式化的函数；参数args表示可以输入多个参数

(argument)，参数间以逗号分隔；参数kwarg表示可以输入多个关键词参数，关键字函数的写法例如：age=' 18'，age为关键字，' 18' 为这个关键字对应的值。

接下来，大家可以尝试运行下方的代码：

```
1 id = input('请输入您的身份证号: ')
2
3 print('您的出生日期为: {}年{}月{}日'.format(id[6:10], id[10:12], id[12:14]))
4 print('您的出生日期为: {1}年{0}月{2}日'.format(id[10:12], id[6:10], id[12:14]))
5 print('您的出生日期为: {year}年{month}月{day}日'.format(month=id[10:12], year=id[6:10],
    day=id[12:14]))
```

运行代码之后，显示的结果为：

大家能够看到，上面的三条print语句对应的显示输出结果是一样的。

我们分别来理解一下上述3条print语句：

第1条：在字符串中我们嵌入了3对“{}”，并且在format函数的参数中写入了3个参数，程序按照参数从左至右的顺序将字符串进行了格式化。

第2条：在字符串中我们仍然嵌入了3对“{}”，但是每一对“{}”中都有一个数字，这些数字是从0开始递增的序号，“{0}”表示在该位置要显示从左至右第1个参数的内容，“{1}”表示在该位置要显示从左至右第2个参数的内容，以此类推。所以，在输入参数的时候，参数的顺序要与前面的序号相对应。

第3条：在字符串中我们也是嵌入了3对“{}”，这一次每一对“{}”中都有一个关键字，这些关键字与参数中的关键字相对应。例如，“{month}”表示在该位置要显示关键字参数中“month”后方的

值，以此类推。

数据结构

分别是元组、列表、集合、字典。

元组

在上一篇教程中，我们接触过元组。

```
1 id = input('请输入您的身份证号: ')
2 print('您的出生日期为: %s年%s月%s日' % (id[6:10], id[10:12], id[12:14]))
```

在上面这段代码中 “%” 操作符的后方，就是一个元组。

也就是说元组是由一对括号和以逗号分隔的值组成。

下面是对元组的创建操作，大家可以看注释进行理解。

```
1 tup1 = () # 创建空的元组
2 tup2 = (1,) # 当元组只有一个元素时，要在元素后方添加逗号
3 tup3 = (1,2,3) # 当元组有多个元素时，元素之间以逗号分隔
4 tup4 = (1,1.0,'abc','abc'.upper()) # 元组中的元素可以为Python中所有的对象
5
6 print (tup1,tup2,tup3,tup4)
```

上方代码的运行结果为：

接下来，再给大家介绍一些对元组的操作，看上去和字符串的操作比较相像。

- 获取元素个数：len()
- 连接两个元组：元组1 + 元组2
- 复制多个元组：元组 * 复制数量
- 获取单个元素：元组[索引位置]
- 获取多个元素：元组[起始位置:终止位置:间隔数量]
- 获取元组最大元素：使用max(iterable,key)函数；参数iterable为可迭代对象，例如列表、元组以及我们之后将学习的字典、集合都是可迭代对象；参数key为函数（function），该参数默认为空；注意元素必须为同一类型，否则会抛出异常。
- 获取元组最小元素：使用min(iterable,key)函数，参数同max函数。

- 判断元组是否包含指定元素：元素 in 元组
- 查找元组中指定元素的位置：使用index(object,start, stop)函数，参数object为对象，因为元组的元素可以是所有Python的对象的一种；参数start为查询起始位置；参数stop为查询终止位置。在元组的元素中查询到与参数相同的元素时返回该元素的位置，否则抛出异常。
- 获取元组中元素出现的次数：使用count(object)函数，参数object为对象。

```
1 tup1 = (1,2,3,4,5,6,7,8,9) # 创建元组
2 tup2 = ('a','b','c')
3 tup3 = ('你','我','你','他','我','我')
4
5 print (len(tup1)) # 获取元组的元素数量
6 print (tup1+tup2) # 连接多个元组为一个新元组
7 print (tup2*3) # 重复元组元素为一个新元组
8 print (tup1[0]) # 获取元组指定位置的元素，索引位置从左至右由0开始
9 print (tup1[-1]) # 获取元组指定位置的元素，索引位置从右至左由-1开始
10 print (tup1[2:8:2]) # 获取元组中指定片段的元素，并可以设置间隔获取
11 print (max(tup1)) # 元组中元素均为同一类型时，获取元组中最大的元素
12 print (min(tup1)) # 元组中元素均为同一类型时，获取元组中最小的元素
13 print ('a' in tup1) # 判断元组中是否包含某个元素，如果包含，返回值为True，否则为False
14 print (tup1.index(3)) # 查询元组中是否包含某个元素，如果包含，返回值为索引位置，否则抛出异常
15 print (tup3.count('我')) # 获取元组中某个元素的出现次数
```

运行代码之后，显示的结果为：

最后，大家要注意，元组不可更改，只能查询。

列表 (list)

列表和元组很相像。

下面是对列表的创建操作，大家可以看到，形式上列表和元组的区别只在于列表两侧是一对方括号。

```
1 lst1 = []
2 lst2 = [1]
3 lst3 = [1,2,3,4,5,6,7,8,9]
4
```

```
5 print (lst1,lst2,lst3)
```

运行代码之后，显示的结果为：

接下来，我来给大家介绍对列表的操作。

因为列表的元素是可变的，所以操作上比元组更丰富。

首先，大家会发现刚才我们对元组的操作，对列表同样有效。

大家可以尝试运行下方这段代码：

```
1 lst1 = [1,2,3,4,5,6,7,8,9] # 创建列表
2 lst2 = ['a','b','c']
3 lst3 = ['你','我','你','他','我','我']
4
5 print (len(lst1)) # 获取列表的元素数量
6 print (lst1+lst2) # 连接多个列表为一个新列表
7 print (lst2*3) # 重复列表元素为一个新列表
8 print (lst1[0]) # 获取列表指定位置的元素，索引位置从左至右由0开始
9 print (lst1[-1]) # 获取列表指定位置的元素，索引位置从右至左由-1开始
10 print (lst1[2:8:2]) # 获取列表中指定片段的元素，并可以设置间隔获取
11 print (max(lst1)) # 数组中元素均为数字类型时，获取列表中数值最大的元素
12 print (min(lst1)) # 数组中元素均为数字类型时，获取列表中数值最小的元素
13 print ('a' in lst1) # 判断列表中是否包含某个元素，如果包含，返回值为True，否则为False
14 print (lst1.index(3)) # 查询元组中是否包含某个元素，如果包含，返回值为索引位置，否则抛出异常
15 print (lst3.count('我')) # 获取元组中某个元素的出现次数
```

运行代码之后，显示的结果为：

除了和元组相同的操作之外，列表还有其他操作，例如添加、插入、取出、删除以及修改列表所包含的元素。

1、添加元素

添加单个元素：使用append(object)函数可以为列表添加单个元素，参数object为对象；也就是说所有Python的对象都可以添加到列表中。

添加多个元素（合并列表）：使用extend(iterable)函数可以为列表添加多个元素，参数iterable为可迭代对象。

以下代码演示了向列表中添加元素的操作：

```
1 lst1 = [1,2,3,4,5,6,7,8,9] # 创建列表
2 lst2 = ['a','b','c'] # 创建列表
3 lst1.append(10) # 添加单个元素到列表
4 lst2.extend(['d','e']) # 添加多个元素到列表
5 print (lst1,lst2)
```

运行代码之后，显示的结果为：

2、更改元素

更改单个元素：列表[索引位置] = 新元素

示例代码：

```
1 lst = [1,2,3,4,5,6,7,8,9] # 创建列表
2 lst[3]= 'x' # 更改指定位置的元素为新元素
3 print (lst)
4
5 # 以上代码运行结果：[1, 2, 3, 'x', 5, 6, 7, 8, 9]
```

更改多个元素：列表[起始位置,终止位置] = 新元素

```
1 lst = [1,2,3,4,5,6,7,8,9] # 创建列表
2 lst[3:7]= 'x','y' # 更改指定位置区间的元素为新元素，数量无需对应
3 print (lst)
4
5 # 以上代码运行结果：[1, 2, 3, 'x', 'y', 8, 9]
```

3、插入元素

插入单个元素：使用insert(index,object)函数，参数index为索引位置，表示在该位置之前插入新的元素；参数object为对象。

```
1 lst = [1,2,3,4,5,6,7,8,9] # 创建列表
2 lst.insert(0,'列表') # 起始位置插入新元素，0表示第一个索引位置
3 print (lst)
4
5 # 以上代码运行结果：['列表', 1, 2, 3, 4, 5, 6, 7, 8, 9]
```


插入多个元素：列表[索引位置:索引位置] = 新元素；注意，两个索引位置保持一致。

```
1 lst = [1,2,3,4,5,6,7,8,9] # 创建列表
2 lst[3:3]= '*', '*' # 指定位置前方插入新元素
3 print (lst)
4
5 # 以上代码运行结果: [1, 2, 3, '*', '*', 4, 5, 6, 7, 8, 9]
```

4、取出元素

使用pop(index)函数，参数index为被取出元素的索引位置。

示例代码：

```
1 lst = [1,2,3,4,5,6,7,8,9]
2 print (lst.pop(5))
3 print (lst)
```

运行代码之后，显示的结果为：

5、删除元素

删除指定内容元素：使用remove(object)函数可以删除列表中首次出现的与参数相同的元素，如果列表中不存在与参数相同的元素则抛出异常。

```
1 lst = ['你','我','他','我','你','他'] # 创建列表
2 lst.remove('我') # 删除从左至右第一个与指定内容匹配的元素
3 print (lst)
4
5 # 以上代码运行结果: ['你', '他', '我', '你', '他']
```

删除单个指定位置元素：del 列表[索引位置]

```
1 lst = ['小','楼','是','个','帅','哥'] # 创建列表
2 del lst[3] # 删除指定位置的元素
3 print (lst)
```

```
4
5 # 以上代码运行结果: ['小', '楼', '是', '帅', '哥']
```

删除多个指定位置元素: del 列表[起始位置:终止位置]

```
1 lst = ['小', '楼', '是', '个', '帅', '哥'] # 创建列表
2 del lst[2:4] # 删除从起始位置至终止位置的多个元素
3 print (lst)
4
5 # 以上代码运行结果: ['小', '楼', '帅', '哥']
```

删除末尾元素: 使用pop()函数, 参数为空即可。

```
1 lst = ['小', '楼', '真', '是', '帅', '哥'] # 创建列表
2 lst.pop() # 删除末尾的元素
3 print (lst)
4
5 # 以上代码运行结果: ['小', '楼', '真', '是', '帅']
```

清空所有元素:

使用clear()函数。

示例代码:

```
1 lst = [1,2,3,4,5,6] # 创建列表
2 lst.clear() # 清空全部元素
3 print (lst)
4
5 # 以上代码运行结果: []
```

或者, 使用del命令: del 列表[:]

```
1 lst = [1,2,3,4,5,6] # 创建列表
2 del lst[:] # 清空全部元素
3 print (lst)
4
```

```
5 # 以上代码运行结果: []
```

6、列表排序

反向排序：使用reverse()函数。

```
1 lst = [3,2,4,5,6,1] # 创建列表
2 lst.reverse() # 反向排序列表元素
3 print (lst)
4
5 # 以上代码运行结果: [1, 6, 5, 4, 2, 3]
```

升降排序：使用sort(cmp,key, reverse)函数，参数cmp为函数，参数key为函数，reverse为布尔值（True和False）。

参数为空时默认为升序排列。

示例代码：

```
1 lst = [6,1,2,3,4,5] # 创建列表
2 lst.sort() # 升序排列列表元素
3 print (lst)
4
5 # 以上代码运行结果: [1, 2, 3, 4, 5, 6]
```

通过设置参数reverse=True，转换为降序排列。

```
1 lst = [6,1,2,3,4,5] # 创建列表
2 lst.sort(reverse=True) # 降序排列列表元素
3 print (lst)
4
5 # 以上代码运行结果: [6, 5, 4, 3, 2, 1]
```

升降序排列也可以使用函数sorted(iterable,cmp, key, reverse)，参数iterable为可迭代对象；参数cmp为函数，参数key为函数，reverse为布尔值。

sorted()函数不会改变原列表。

```
1 lst = [3,2,4,5,6,1] # 创建列表
2 print (sorted(lst)) # 输出显示升序列表
3 print (sorted(lst,reverse=True)) # 输出显示降序列表
4 print (lst) # 输出显示原列表
5
6 # 以上代码运行结果:
7 # [1, 2, 3, 4, 5, 6]
8 # [6, 5, 4, 3, 2, 1]
9 # [3, 2, 4, 5, 6, 1]
```

7、元组转换为列表

使用list(iterable)函数，参数iterable为可迭代对象。

```
1 tup = (3,2,4,5,6,1) # 创建列表
2 lst = list(tup)
3 print (lst)
4
5
6 # 以上代码运行结果: [3, 2, 4, 5, 6, 1]
```

集合 (set)

1、创建集合

集合的创建不同于前两种数据结构。

集合通过set(iterable)方法创建，参数iterable为可迭代对象。

示例代码：

```
1 s1 = set('好好学习天天想上') # 将字符串分解为单个字符，作为集合的元素创建集合
2 s2 = set(('好好', '学习', '天天', '想上')) # 将元组分解为单个元素，作为集合元素创建集合
3 s3 = set(['好好', '学习', '天天', '想上']) # 将列表分解为单个元素，作为集合元素创建集合
4
5
6 print (s1) # 显示输出结果为: {'好', '习', '上', '天', '想', '学'}
7 print (s2) # 显示输出结果为: {'好好', '想上', '学习', '天天'}
```

```
8 print (s3) # 显示输出结果为: {'好好', '想上', '学习', '天天'}
```

通过上方示例，大家能够看出：

- 集合可以通过可迭代对象（字符串、元组、列表等）进行创建；
- 集合中的元素不可重复；
- 集合中的元素无序排列。

2、添加元素

集合添加元素的方法有两种。

添加单个元素：使用add(element)函数，参数element为集合允许添加的元素（例如数字、字符串、元组等）。

添加多个元素：使用update(iterable)函数，参数iterable为可迭代对象。

示例代码：

```
1 # 创建集合
2 s1 = set('123')
3 s2 = set('123')
4 s3 = set('abc')
5
6 # 添加单个元素
7 s1.add('4')
8
9 # 添加多个元素
10 s2.update(['4','5','6']) # 添加列表到集合，列表元素会被分解为单个元素后添加到集合
11 s3.update('de') # 添加字符串到集合，字符串会被分解为单个元素后添加到集合
12
13 # 显示输出
14 print (s1) # 显示输出结果为: {'4', '3', '1', '2'}
15 print (s2) # 显示输出结果为: {'4', '2', '6', '5', '3', '1'}
16 print (s3) # 显示输出结果为: {'c', 'b', 'd', 'e', 'a'}
17
18 #注意：因为集合元素是无序的，大家在测试以上代码时，显示输出的结果与上方结果，可能在顺序上不一致。
```

3、删除元素

集合删除元素的方法有两种。

第一种：使用remove(element)方法删除指定元素，参数element为需要删除的元素。

第二种：使用discard(element)方法删除指定元素，参数element为需要删除的元素。

示例代码：

```
1 # 创建集合
2 s1 = set(['Python', 'Java', 'C', 'C++', 'C#'])
3 s2 = set(['Python', 'Java', 'C', 'C++', 'C#'])
4
5 # 删除元素
6 s1.remove('C++')
7 s2.discard('C++')
8
9 # 显示输出
10 print(s1) # 显示输出结果为: {'C', 'Python', 'Java', 'C#'}
11 print(s2) # 显示输出结果为: {'C', 'Python', 'Java', 'C#'}
```

从上方示例中，大家能够看到remove()和discard()的作用是一样的。

不过，这两个方法是有区别的。

当集合中不存在这两个方法参数中填入的元素时，remove()方法会抛出异常，而discard()方法则没有任何影响。

4、取出元素

集合支持pop()方法取出元素。

示例代码：

```
1 # 创建集合
2 s = set(['Python', 'Java', 'C', 'C++', 'C#'])
3
4 # 显示输出
5 print(s) # 显示输出结果为: {'Python', 'C#', 'C++', 'Java', 'C'}
6 print(s.pop()) # 取出集合元素，显示输出结果为: Python
7 print(s) # 显示输出结果为: {'C#', 'C++', 'Java', 'C'}
```

5、清空集合

集合支持clear()方法进行清空。

示例代码：

```
1 # 创建集合
2 s = set(['Python', 'Java', 'C', 'C++', 'C#'])
```

```
3
4 # 清空集合
5 s.clear()
6
7 # 显示输出
8 print (s) # 显示输出结果为: set()
```

6、交集/并集/补集/差集

首先我们来看张图，理解交集、并集、补集、差集的概念。

假设有集合A{1,2,3}和B{3,4,5}。

交集：A和B中相同部分的内容，{3}。

并集：A和B去重后的全部内容，{1,2,3,4,5}。

补集：A去除与B相交部分后的内容，{1,2}。

差集：A和B不相交部分的全部内容，{1,2,4,5}。

示例代码：

```
1 # 创建集合
2 s1 = set('Python')
3 s2 = set('PyCharm')
4
5 # 交集操作：获取两个集合中共有元素。
6 print (s1 & s2) # 显示输出结果为: {'y', 'P', 'h'}
7 print (s1.intersection(s2)) # 显示输出结果为: {'y', 'P', 'h'}
8
9 # 并集操作：获取两个集合去除重复元素后的全部元素。
10 print (s1 | s2) # 显示输出结果为: {'y', 'a', 'C', 'o', 'P', 'n', 't', 'm', 'r', 'h'}
11 print (s1.union(s2)) # 显示输出结果为: {'y', 'a', 'C', 'o', 'P', 'n', 't', 'm', 'r', 'h'}
12
13 # 补集操作：获取当前集合去除与另一集合交集元素后的全部元素。
14 print (s1 - s2) # 显示输出结果为: {'o', 't', 'n'}
15 print (s1.difference(s2)) # 显示输出结果为: {'o', 't', 'n'}
16 print (s2 - s1) # 显示输出结果为: {'m', 'a', 'r', 'C'}
17 print (s2.difference(s1)) # 显示输出结果为: {'m', 'a', 'r', 'C'}
18
19 # 差集操作：获取两个集合去除交集元素后的全部元素。
20 print (s1 ^ s2) # 显示输出结果为: {'o', 't', 'm', 'a', 'r', 'n', 'C'}
```

```
21 print (s1.symmetric_difference(s2)) # 显示输出结果为: {'o', 't', 'm', 'a', 'r', 'n',  
    'c'}
```

以上操作中，对集合本身内容并无影响，大家可以在执行以上代码后，继续显示输出s1和s2的内容，能够看到没有任何变化。

接下来，我们再来看几个方法，这些方法会改变集合内容。

第一种：difference_update(set) 函数，能够将当前集合和指定集合进行补集运算，并将当前集合内容更新为运算结果。

示例代码：

```
1 s1=set('1234')  
2 s2=set('456')  
3  
4 s1.difference(s2) # 该操作对s1内容无影响  
5 print (s1) # s1无变化，显示输出结果为: {'3', '4', '2', '1'}  
6 s1.difference_update(s2) # 更新集合s1的内容为s1-s2后的结果  
7 print (s1) # s1内容被更新，显示输出结果为: {'3', '2', '1'}
```

第二种：intersection_update(set) 函数，能够将当前集合和指定集合进行交集运算，并将当前集合内容更新为运算结果。

示例代码：

```
1 s1=set('1234')  
2 s2=set('456')  
3  
4 s1.intersection_update(s2) # 更新集合s1的内容为s1 & s2后的结果  
5 print (s1) # s1内容被更新，显示输出结果为: {'4'}
```

第三种：symmetric_difference_update(set) 函数，能够将当前集合和指定集合进行差集运算，并将当前集合内容更新为运算结果。

示例代码：

```
1 s1 = set('1234')  
2 s2 = set('456')  
3  
4 s1.symmetric_difference_update(s2) # 更新集合s1的内容为s1 ^ s2后的结果
```



```
5 print(s1) # s1内容被更新，显示输出结果为：{'6', '3', '2', '5', '1'}
```

7、成员关系

Python中提供了一些方法，让我们能够判断一个集合中是否包含某一元素；

也可以判断一个集合是否另一个集合的子集或超集。

还可以判断一个集合与另一个集合是否没有交集。

在之前我们接触过“in”这操作符，可以用来判断操作符前方的值是否被后方的序列包含（成员关系）。

另外，我们还可以使用“not in”，判断操作符前方的值是否未被后方的序列包含（非成员关系）。在集合中，我们同样可以使用这两个操作符。

另外，我们还可以通过以下方法，判断一个集合是否另外一个集合的子集或超集以及没有交集。

isdisjoint(set)：可以判断集合是否与指定集合不存在交集，参数set为集合；如果成立返回结果为True，否则为False。

issubset(set)：可以判断集合是否指定集合的子集，参数set为集合；如果成立返回结果为True，否则为False。

issuperset(set)：可以判断集合是否指定集合的超集，参数set为集合；如果成立返回结果为True，否则为False。

示例代码：

```
1 s1 = set('好好学习')
2 s2 = set('天天想上')
3 s3 = set('好好学习天天想上')
4
5 print('好' in s1) # 显示输出结果为：True
6 print('好' not in s2) # 显示输出结果为：True
7 print(s1.isdisjoint(s2)) # 显示输出结果为：True
8 print(s1.issubset(s3)) # 显示输出结果为：True
9 print(s3.issuperset(s1)) # 显示输出结果为：True
```

8、复制集合

使用copy()方法能够对集合进行复制。

大家通过下方代码即可理解复制的用途。

示例代码：

```

1 a = set('小楼一夜听春语') # 创建集合存入变量a
2 b = a # 创建变量b引用变量a的集合
3 c = a.copy() # 创建变量c复制变量a的值
4 print(a) # 显示输出结果为: {'春', '夜', '楼', '听', '语', '小', '一'}
5 print(b) # 显示输出结果为: {'春', '夜', '楼', '听', '语', '小', '一'}
6 print(c) # 显示输出结果为: {'春', '夜', '楼', '听', '语', '小', '一'}
7
8 a.remove('一') # 删除变量a中集合的一个元素
9 print(a) # 变量a发生改变, 显示输出结果为: {'春', '夜', '楼', '听', '语', '小'}
10 print(b) # 变量b因为引用变量a, 同样发生改变, 显示输出结果为: {'春', '夜', '楼', '听', '语', '小'}
11 print(c) # 变量c没有改变, 显示输出结果为: {'春', '夜', '楼', '听', '语', '小', '一'}

```

如果还不能够理解，我们可以看下面这张图。

代码中，`b = a`实际上是将b指向了a的内容，所以当a的内容发生变化时，b同步发生了变化。

而`c = a.copy()`则是将a的内容真正进行了复制，不再受a的变化影响。

9、其他

集合也支持`len()`方法进行元素数量的获取，也支持`max()`方法和`min`方法获取集合中的最大元素与最小元素，在此不再赘述。

字典

字典同样是一个序列，不过字典的元素是由key（键，可理解为关键字或名称）与values（值）组成。

就好像我们查字典，一个拼音对应着与之关联的一个或多个汉字，拼音就key，而对应的汉字就是values。

字典两侧和集合一样是大括号，其中每一个元素都是“key:values”的形式，并且每个元素间以逗号分隔。

例如：{ 'yue' : ['月' ; '约' ; '乐'], 'ri' : '日' ; 'le' : '了' ; 'liao' : '了' }

字典中的值没有特定顺序，但必须用一个特定的键存储。

字典的键必须是不可变的数据类型，可以是数字、字符串或者元组。

说明：这种能够通过名称引用值的数据类型称做映射（Mapping），字典是Python中唯一内建的映射类型。映射的概念好像比较难懂，以我个人的理解，映射就是名称集合与值集合的对应关系。名称集合中每个名称都是唯一的（即Key不可重复），并有唯一的值（Key与Value相对应）；值集合中，值可以是唯一的也可以是重复的，但每个值也只能有唯一的名称。

使用字典需要注意以下几点：

- 字典中不能够出现相同的键，但可以出现相同的值。
- 字典中的键不能够更改，但值可以更改。
- 字典的值可以为Python中的任何对象。

接7下来，我们来看一下关于字典的相关操作。

1、创建字典

创建字典可以直接按格式创建，也可以使用dict()方法进行创建。

dict(**kwarg): 参数**kwarg为可变关键字参数。

dict(mapping, **kwarg): 参数mapping为映射函数。

dict(iterable, **kwarg): 参数iterable为可迭代对象。

示例代码：

```
1 d = {} # 创建空字典
```

```
1 d = dict() # 创建空字典
```

```
1 d = {'yue':['月','约','乐'],'ri':'日','le':'了','liao':'了'} # 创建字典
```

```
1 d = dict(小楼='好帅', 小美='好美') # 通过可变参数创建字典
```

```
1 d = dict([('小楼','好帅'),('小美','好美'])) # 通过可迭代对象（列表）创建字典
```

另外，还有一种字典的创建方式，通过fromkeys(seq,value)方法进行创建，参数seq为包含key的序列，参数value为key的默认值。

示例代码：

```
1 k = ['小楼','小黑'] # key的列表8
2 d1 = dict.fromkeys(k) # 从key的列表创建字典
3 d2 = dict.fromkeys(k,'哦哦') # 从key的列表创建字典，并赋予默认值
4 print (d1) # 显示输出结果为: {'小楼': None, '小黑': None}
5 print (d2) # 显示输出结果为: {'小楼': '哦哦', '小黑': '哦哦'}
```

2、查询元素

首先，查询字典中的元素。

我们可以使用items()方法，通过items()方法可以获取到字典中所有元素的迭代器。

提示：关于迭代器，大家可以暂时不用关心，在后面的教程中有专门的讲解。

示例代码：

```
1 d = {'小楼': '好帅', '小美': '好美'} # 创建字典
2 print (d.items()) # 显示输出结果为: dict_items([('小楼', '好帅'), ('小美', '好美')])
```

然后，查询字典中元素的键。

我们可以使用keys()方法，通过keys()方法可以获取到字典中所有元素键的迭代器。

示例代码：

```
1 d = {'小楼': '好帅', '小美': '好美'} # 创建字典
2 print (d.keys()) # 显示输出结果为: dict_keys(['小楼', '小美'])
```

最后，查询字典中元素的值。

查询元素值有多种方法：

第一种，通过键可以获取相对应的值：字典[键]

示例代码：

```
1 d = {'小楼': '好帅', '小美': '好美'} # 创建字典
2
3 print (d['小楼']) # 显示输出结果为: 好帅
4 print (d['小美']) # 显示输出结果为: 好美
```

第二种，通过get(k,default)方法查询，参数k为查询的键，参数default为未查询到结果时的默认值。

示例代码：

```
1 d = {'小楼': '好帅', '小美': '好美'} # 创建字典
2 print (d.get('小楼', '男神')) # 显示输出结果为: 好帅
3 print (d.get('小白', '女神')) # 显示输出结果为: 女神
```

第三种、通过values()方法可以获取到字典中所有元素值的迭代器。

示例代码：

```
1 d = {'小楼': '好帅', '小美': '好美'} # 创建字典
2 print (d.values()) # 显示输出结果为: dict_values(['好帅', '好美'])
```

3、添加元素

添加单个元素：

首先，可以通过“字典[键]=值”的方式进行添加，如果字典中不存在相同的键则自动添加，否则修改已存在的键所对应的值。

示例代码：

```
1 d = {'小楼': '好帅', '小美': '好美'}
2 d['小白'] = '好棒' # 添新元素到字典
3 d['小黑'] = '好黑', '好白' # 添加值为元组的新元素到字典
4 print (d) # 显示输出结果为: {'小楼': '好帅', '小美': '好美', '小白': '好棒', '小黑': ('好黑', '好白')}
```

另外，还可以通过setdefault(k,default)方法进行添加，参数k为指定的键，参数default为默认值。当字典中存在指定的键时，能够返回该键所对应的值；如果不存在指定的键时，则会返回参数default中设置的值，同时，在字典中会添加新元素，新元素的键即为参数k，值即为参数default。

示例代码：

```
1 d = {'小楼': '好帅', '小美': '好美'} # 创建字典
2 print (d.setdefault('小楼', '好棒')) # 字典中存在相应的键，则返回该键对应的值，显示输出结果为：好帅
3 print (d.setdefault('小白', '好爽')) # 字典中不存在相应的键，则返回default参数的值，显示输出结果为：好爽
4 print (d) # 当字典中不存在相应的键时，添加新元素，显示输出结果为: {'小楼': '好帅', '小美': '好美', '小白': '好爽'}
```

添加多个元素：通过update(m,kwargs)方法进行添加，参数m（mapping）为映射函数，kwargs为可变参数。

示例代码：

```
1 d = {'小楼': '好帅', '小美': '好美'} # 创建字典
2 d.update(小白='好白', 小黑='好黑') # 通过可变参数添加多个元素
3 print (d) # 显示输出结果为: {'小楼': '好帅', '小美': '好美', '小白': '好白', '小黑': '好黑'}
```

```
1 d = {'小楼': '好帅', '小美': '好美'} # 创建字典
2 d.update((( '小白', '好白'), ('小黑', '好黑')))) # 通过元组添加多个元素
3 print (d) # 显示输出结果为: {'小楼': '好帅', '小美': '好美', '小白': '好白', '小黑': '好黑'}
```

4、修改元素值

修改某个键对应的元素：字典[键]=新值

示例代码：

```
1 d = {'小楼': '好帅', '小美': '好美'} # 创建字典
2 d['小楼'] = '好棒'
3 print (d) # 显示输出结果为: {'小楼': '好棒', '小美': '好美'}
```

5、删除元素

使用del指令可以通过键删除某个元素：del 字典[键]

示例代码：

```
1 d = {'小楼': '好帅', '小美': '好美'} # 创建字典
2 del d['小美'] # 删除元素
3 print (d) # 显示输出结果为: {'小楼': '好帅'}
```

6、取出元素与元素值

我们可以使用popitem()方法在字典中取出元素。

示例代码：

```
1 d = {'小楼': '好帅', '小美': '好美'} # 创建字典
2 print (d.popitem()) # 显示输出结果为: ('小美', '好美')
3 print (d) # 显示输出结果为: {'小楼': '好帅'}
```

并且，我们可以使用pop(k,default)方法在字典中取出指定元素的值，参数k为指定元素的键，参数default为未取到结果时的默认值。

```
1 d = {'小楼': '好帅', '小美': '好美'} # 创建字典
2 print (d.pop('小楼')) # 显示输出结果为: 好帅
3 print (d.pop('小白', '好爽')) # 显示输出结果为: 好爽
```

```
4 print (d) # 显示输出结果为: {'小美': '好美'}
```

7、设置默认值

```
1 d = {'小楼': '好帅', '小美': '好美'} # 创建字典
2 print (d.setdefault('小楼', '好棒')) # 显示输出结果为: 好帅
3 print (d.setdefault('小白', '好爽')) # 显示输出结果为: 好爽
4 print (d) # 显示输出结果为: {'小楼': '好帅', '小美': '好美', '小白': '好爽'}
```

8、其他

字典也支持使用以下方法:

clear(): 清空字典

copy(): 复制字典:

len(): 获取字典元素数量

max(): 获取字典中最大的键

min(): 获取字典中最小的键

同时, 字典也支持通过in和not in进行成员关系的判断。

这一篇教程我们来接触循环。

循环有两种方法, 一种是for...in..., 另外一种是while....

首先, 我们来看for...in...循环。

语句内容为[for 变量 in 可迭代对象:], 注意不要遗漏结尾的冒号。

for循环能够对可迭代对象进行迭代(可以简单的理解为依次读取), 所以, 可迭代对象的元素数量, 决定循环的次数。

每一次迭代, 都会从可迭代对象中读取元素写入in前方的变量, 但是, 这个变量并非一定要使用。

另外, 每一次迭代, 都会执行for语句下方向右缩进的语句块。

特别说明1: 在Python的编程规范中, 用4个空格来缩进代码, 或者用tab缩进代码, 但不要空格和tab混用。

特别说明2: PyCharm提供了代码格式化功能, 通过快捷键Ctrl+Alt+L, 可以快速让代码变得规范整齐。

例如, 我们从一个列表中取出所有的元素, 并依次显示输出。

示例代码:

```
1 for s in ['小楼', '是', '一个', '帅哥']:
2     print(s) # 向右缩进的语句
```

运行代码之后, 显示的结果为:

小楼

是

一个

帅哥

大家能够看到，列表中每一个元素都被读取出来并且显示输出。

不过，我们能不能让显示输出的结果在同一行显示呢？

这里，我们再来看一下print这个方法。

print(*objects, sep, end, file, flush)方法，参数objects表示多个输出的对象；参数sep表示多个输出对象直接的分隔符，默认为1个空格；参数end表示输出对象结束符，默认为换行符；参数file表示数据流输出到的文件；参数flush表示数据流输出至文件时是否缓冲，默认为False（不缓冲）。

因为print方法中的参数end默认值为换行符，所以上面代码的运行结果会分多行显示。

如果需要在同一行显示，我们只需要将参数end设置为空即可。

示例代码：

```
1 for s in ['小楼', '是', '一个', '帅哥']:
2     print(s, end='') # 显示输出结果为：小楼是一个帅哥
```

循环不仅能够对可迭代对象进行读取和输出，还可以用于重复某一过程。

例如，我们把“小楼好帅！”这句话重复显示输出5次。

示例代码：

```
1 for i in range(5):
2     print('小楼好帅!')
```

大家能够看出，在上方代码中变量i并没有什么作用，而range(5)是关键。

我们来了解一下range这个方法。

range(start,stop,step)函数能够获取一个连续增加的不可变的整数序列，参数start为序列的起始数值；参数stop为序列的终止数值；参数step为步长，默认值为1。

那么，range(5)就能够获取一个0~4的整数序列，这个序列包含了5个元素。

所以，for循环在进行迭代时，能够迭代5次，也就将下方向右缩进的语句块执行了5次。

range方法还能够帮助我们取得偶数序列和奇数序列等。

示例代码：

```
1 print(list(range(0,10,2))) # 获取0~9中的偶数，显示输出结果为：[0, 2, 4, 6, 8]
2 print(list(range(1,10,2))) # 获取0~9中的奇数，显示输出结果为：[1, 3, 5, 7, 9]
```

接下来，我们来看另外一种while...循环。

语句内容为[while 条件:]，注意不要遗漏结尾的冒号。

通过语句内容，可以理解，只要满足循环的条件，就能够继续循环，否则，退出循环。

每一次循环，都会执行while语句下方向右缩进的语句块。

既然也是循环，那么刚才的示例效果也可以通过while来实现。

示例代码：循环显示输出列表内容


```

1 l = ['小楼', '是', '一个', '帅哥'] # 创建列表
2 n = 0 # 创建变量，用于计数
3 while n < len(l): # 循环，条件为计数数量小于列表元素数量
4     print (l[n], end='') # 符合条件时，将计数数量作为列表索引，获取元素显示输出
5     n += 1 # 计数数量自增1
6 # 显示输出结果为：小楼是一个帅哥

```

示例代码：重复显示输出“小楼好帅！”5次

```

1 n = 0 # 创建变量，用于计数
2 while n < 5: # 循环，条件为计数数量小于重复次数
3     print ('小楼好帅! ') # 符合条件时，显示输出字符串内容
4     n += 1 # 计数数量自增1
5 # 显示输出结果为：(略)

```

示例代码：获取0~9之间的奇数与偶数

```

1 l = [] # 创建空列表
2 n = 0 # 创建变量，用于获取偶数，获取奇数时变量初始值为1
3 while n < 10: # 循环，条件为计数数量小于重复次数
4     l.append(n) # 符合条件时，将偶数添加到列表
5     n += 2 # 计数数量自增2
6 print (l) # 显示输出结果为：[0, 2, 4, 6, 8]

```

以上是两种循环的使用方法和示例。

在下一篇教程中，我们结合条件判断，了解循环中continue和break的使用。

第一，只需要满足条件时，执行某个过程。

代码格式如下：

if 条件:

执行的语句块

“if”表示“如果”，如果条件成立时，将会执行下方向右缩进的语句块。

示例代码：

```

1 s = input('身体是不是不舒服？请回答：')

```

```
2 if s == '是': # 符合条件的情形
3     print ('快去医院看病!')
```

运行结果（符合条件时）：

运行结果（不符合条件时）：

通过上面这个示例，大家能够看到条件判断中的条件是一个表达式。

条件表达式通常包括以下关系运算符：

==：表示等于，用于判断运算符两侧的内容是否相同。

!=：表示不等于，用于判断运算符两侧的内容是否不相同。

>：表示大于，用于判断运算符左侧内容是否大于右侧内容。

>=：表示大于等于，用于判断运算符左侧内容是否大于等于右侧内容。

<：表示小于，用于判断运算符左侧内容是否小于右侧内容。

<=：表示小于等于，用于判断运算符左侧内容是否小于等于右侧内容。

is：表示是，用于判断运算符左侧内容和右侧内容是否同一对象。

is not：表示不是，用于判断运算符左侧内容和右侧内容是否非同一对象。

in：表示被包含，用于判断运算符左侧内容是否被右侧内容所包含。

not in：表示不被包含，用于判断运算符左侧内容是否不被右侧内容所包含。

除了以上的关系型运算符，还有以下这些逻辑运算符：

and：表示并且。

or：表示或者。

not：表示不是。

这些逻辑运算符能够将多个条件表达式连接到一起，形成更复杂的条件表达式。

第二、当满足条件时，执行某个过程，否则，执行另一个过程。

代码格式如下：

if 条件:

执行的语句块

else:

执行的语句块

“else”表示“否则”，如果条件成立时，将会执行if下方向右缩进的语句块，否则，执行else下方向右缩进的语句块。

示例代码：

```
1 s = input('是否喜欢苹果公司的产品？请回答：')
2 if s == '是': # 符合条件的情形
3     print ('请购买iPhone手机!')
4 else: # 不符合上述条件的情形
5     print ('请购买Android手机!')
```

运行结果（符合条件时）：

运行结果（不符合条件时）：

第三、当满足某个条件时，执行某个过程；否则，满足另外某个条件时，执行某个过程；最后，所有条件均不成立时，执行某个过程。

代码格式如下：

if 条件:

执行的语句块

elif:

执行的语句块

else:

执行的语句块

“elif”表示“否则，如果”。

如果符合某一条件，将会执行if下方向右缩进的语句块；

否则，如果符合某一条件，将会执行elif下方向右缩进的语句块；

否则，执行else下方向右缩进的语句块。

示例代码：

```
1 s = input('有什么吃的？请回答：')
2 if '面条' in s: # 符合当前条件的情形
3     print('我要吃面条！')
4 elif '馒头' in s: # 符合当前条件的情形
5     print('我要吃馒头！')
6 else: # 不符合上述所有条件的情形
7     print('随便来点吧！')
```

运行结果（满足if的条件时）：

运行结果（满足elif的条件时）：

运行结果（以上条件均未满足时）：

注意：在编写代码时，elif可以在if之后和else之前多次出现，进行多种条件的判断。

接下来，我们结合上一篇循环的教程看一下，如何有条件的进行循环操作。

示例：计算一个列表中所有奇数和偶数之和。

示例代码：

```
1 l = [1, 2, 3, 4, 5, 6, 7, 8, 9]
2 count1 = 0 # 创建变量，用于奇数求和
3 count2 = 0 # 创建变量，用于偶数求和
4 for i in l: # 循环遍历列表中全部元素
5     if i % 2 == 0: # 判断变量i中存储的为偶数
```

```

6         count2 += i # 进行偶数求和运算
7         continue # 结束当前循环过程，进入下一次循环
8         count1 += i # 进行奇数求和运算
9     print (count1) # 显示输出结果为：25
10    print (count2) # 显示输出结果为：20

```

在上方代码中，大家能够看到条件表达式： $i \% 2 == 0$

这里的“%”为取余运算符，即能够获取i除以2时的余数。

例如：11除以2商5余1，那么 $11 \% 2$ 等于1。

另外，在上方代码中，还能够看到一个英文单词“continue”。

continue的中文含义是“继续”，写在循环的代码中，表示结束当前循环，继续下一次循环。

所以，在上方代码中，当i为偶数时，会进行偶数求和运算，然后跳过当前的循环，不会进行奇数求和运算；而为奇数时，才会进行奇数求和运算。

示例：找到小数点出现的位置。

示例代码：

```

1  f = input('请输入一个小数：')
2  count = 1 # 创建变量，用于计数
3  for i in f: # 循环遍历输入的内容
4      if i == '.': # 判断变量i中存储的字符是否为小数点
5          break # 发现小数点时，跳出循环
6      else: # 不符合上述判断条件时
7          count += 1 # 进行计数
8  print ('小数点出现在第%s位。' % count)

```

运行结果：

在上方代码中，又出现了一个英文单词“break”。

break的中文含义是“突破”，写在循环的代码中，表示跳出并停止循环。

所以，在上方代码中，当i为小数点时，将停止循环，输出结果；而在跳出循环之前，则会进行计数运算。

这篇教程分两个部分：运算符和随机数。

一、运算符

在上一篇教程中，我们已经接触过关系运算符和逻辑运算符，接下来，我们来了解算术运算符和赋值运算符。

1、算术运算符

+: 加号，能够实现加法运算，还能够实现字符串、列表以及元组的连接。

```

1  print(1 + 1) # 显示输出结果为：2
2  print('小楼' + '好帅') # 显示输出结果为：小楼好帅

```

```
3 print([1, 2, 3] + [4, 5, 6]) # 显示输出结果为: [1, 2, 3, 4, 5, 6]
4 print((1, 2, 3) + (4, 5, 6)) # 显示输出结果为: (1, 2, 3, 4, 5, 6)
```

-: 减号, 能够实现减法运算。

```
1 print(1 - 1) # 显示输出结果为: 0
```

*: 乘号, 能够实现乘法运算, 还能够实现字符串、列表以及元组的重复。

```
1 print(1 * 3) # 显示输出结果为: 3
2 print('小楼' + '好帅' * 3) # 显示输出结果为: 小楼好帅好帅好帅
3 print([1] * 3) # 显示输出结果为: [1, 1, 1]
4 print((1,) * 3) # 显示输出结果为: (1, 1, 1)
```

/: 除号, 能够实现除法运算。

```
1 print(1 / 3) # 显示输出结果为: 0.3333333333333333
```

?: 百分号, 能够实现模运算 (取余运算), 还能够实现字符串格式化。

```
1 print(13 % 5) # 显示输出结果为: 3
2 print('小楼%s' % ('好帅' * 3)) # 显示输出结果为: 小楼好帅好帅好帅
```

** : 幂运算符, 能够实现某个数值的幂运算。

```
1 print(2 ** 3) # 计算2的3次方, 显示输出结果为: 8
```

//: 整除运算符, 能够实现除法运算, 但结果是向下取整数部分。

```
1 print(7 // 3) # 显示输出结果为: 2
2 print(-7 // 3) # 显示输出结果为: -3
3 print(7.0 // 3) # 显示输出结果为: 2.0
4 print(7 // 3.0) # 显示输出结果为: 2.0
```

```
5 print(7.00 // 3.00) # 显示输出结果为: 2.0
```

2、赋值运算符

=: 将右侧的值或运算结果赋值给左侧的变量

+=: $a += 3$ 等同于 $a = a + 3$

-=: $a -= 3$ 等同于 $a = a - 3$

*=: $a *= 3$ 等同于 $a = a * 3$

/=: $a /= 3$ 等同于 $a = a / 3$

%=: $a %= 3$ 等同于 $a = a \% 3$

**=: $a **= 3$ 等同于 $a = a ** 3$

//=: $a //= 3$ 等同于 $a = a // 3$

二、随机数

在编程中，随机数的应用十分的广泛。

接下来，我们来了解一下如何获取随机数。

如果想获取随机数，我们需要使用随机数的函数。

但是，随机数函数不能够直接使用，因为它不是内建函数，它存在于其它模块中。

那么，什么又是模块呢？

我们创建的Python文件（.py）就是模块（Module）。

Python解释器会自动搜索当前目录、所有已安装的内置模块目录和第三方模块目录，我们要做的只需要在代码中通过命令导入想要使用的模块。

不过为了，让大家了解的更深入一些，这里，大家输入一段代码，获取到Python的安装路径。

```
1 import sys # 导入自建模块
2 print(sys.path) # 显示输出路径集
```

运行结果如下：

大家能够看到，上图中第一行就是Python可执行程序的路径，我们复制前面的一段，粘贴到地址栏中，然后加上“/lib”后，回车打开这个路径。

打开的文件夹就是Python的库文件夹，里面存储的就是可以使用的各类模块，在这个文件夹中我们能够找到random.py这个提供随机数函数的模块。

接下来，我们就在代码中导入使用这个随机数模块。

在刚才的一段代码中，大家第一次见到“import sys”这段代码。

关键字import是导入命令，sys是系统自建模块。

我们就是通过导入sys这个模块，然后，使用了sys中的path属性，获取到了一个路径集。

如果要使用随机数模块，我们也需要通过这种方法导入。

```
1 import random
2 print(random.random()) # 调用模块中的函数random()，显示输出结果为一个18位0~1之间的随机小数。
```

```
3 print(random.randrange(10)) # 调用模块中的函数randfange(), 显示输出结果为一个0~9之间的随机整数。
4 print(random.randrange(3,21)) # 调用模块中的函数randfange(), 显示输出结果为一个3~20之间的随机数。
5 print(random.randrange(0,11,2)) # 调用模块中的函数randfange(), 显示输出结果为一个0~10之间的随机偶数。
```

大家能够看到，在上方代码中使用了两个函数。

函数random(), 能够随机获取一个18位0~1之间的小数。

函数randrange(start,stop,step), 能够随机获取一个指定区间的正整数; 参数start表示随机区间的起始数; 参数stop表示随机区间的终止数, 终止数不会被获取; 参数step表示步长 (数量), 步长之间的部分不会被获取 (例如0~9的随机数, 步长为2, 则随机数只能获取到0、2、4、6、8其中之一)。

其实, 如果我们只使用其中的randrange函数的话, 我们不需要导入random模块, 而是从random模块导入randrange这个函数。

```
1 from random import randrange # 从 random模块 导入 randrange函数
2 print(randrange(10)) # 调用模块中的函数randfange(), 显示输出结果为一个0~9之间的随机整数。
3 print(randrange(3,21)) # 调用模块中的函数randfange(), 显示输出结果为一个3~20之间的随机数。
4 print(randrange(0,11,2)) # 调用模块中的函数randfange(), 显示输出结果为一个0~10之间的随机偶数。
```

上方的代码, 大家能够看到, 我们通过关键字from从模块random单独导入了randrange这个函数。

然后, 在代码中我们就可以直接使用这个函数, 而不需要再使用random去调用。

在之前的教程中, 我们已经接触过了一些关键字 (Keyword), 例如: del、continue、and、from等。

这篇教程, 我把Python中所有的关键字全部整理出来, 并且说明用途, 让大家对关键字有一个全面的了解, 也方便日后查阅。

在Python安装目录下的lib文件夹中, 可以看到Keyword.py这个模块, 打开之后就能够看到以下这33个关键字。

内建常量 (Built-in Constants)

- True (真): 布尔类型的真值。
- False (假): 布尔类型的假值; 在Python中None、0、空字符串和空序列以及空字典 ("、" "、[]、()、{}) 均为假值。
- None (无): 唯一的一种空值类型, 经常用来表示缺少一个值; 例如函数中的一些参数默认值为None。

函数定义 (Function definitions)

- def: 定义, 定义用户自定义的函数对象。
- lambda (希腊字母 "λ"): 用于创建匿名函数, lambda表达式能够产生一个函数对象。

类的定义 (Class definitions)

- class: 类, 定义一个类的对象。

布尔运算 (Boolean Operations)

- and (并且): x and y, 如果x为假, 返回x, 否则返回y; 这是一个短路 (short-circuit) 逻辑运算符, 所以只有第一个参数是真的时, 它才对第二个参数求值。

- or (或者) : `x or y`, 如果x为假, 返回y, 否则返回x; 这是一个短路 (short-circuit) 逻辑运算符, 所以它只对第一个参数为false时的第二个参数进行求值。
- not (不是) : `not`比非布尔运算符优先级低, 所以`not a == b`等同于`not (a == b)`, 并且`a == not b`这种是错误语法。

操作语句 (Statement)

- `assert` (断言) : 该语句是将调试断言插入程序的一种方便方法。
- `pass` (通过) : 该语句被执行时, 什么也不会发生。它的作用是一个占位符, 当某个声明是必需的构成, 但没有代码需要执行时, 可以使用它。
- `del` (删除) : 该语句可以删除列表、名以及属性引用。
- `return` (返回) : 该语句用于某个函数返回结果, 只出现在函数的定义中, 不出现在类的定义中。
- `yield` (生产) : 该语句用于生成器。
- `try` (尝试) : 该语句为一组语句指定异常处理程序, 清除代码。
- `except` (排除) : 该语句用于指定一个或多个异常处理程序。
- `finally` (最后) : 该语句用于指定一个 “cleanup” (清理) 处理程序, 尝试执行`except`以外的语句。
- `raise` (唤起) : 该语句可以显示地引发异常, 一旦执行了`raise`语句, `raise`后面的语句将不能执行。
- `break` (跳出) : 该语句出现在`for`或`while`循环中, 用于跳出结束循环。
- `continue` (继续) : 该语句出现在`for`或`while`循环中, 用于结束当前循环过程, 进入下一次循环过程。
- `global` (全局) : 用于当前整个代码块, 列举出的标识符作用域将变为全局。
- `nonlocal` (外部) : 非局部且非全局, 列出的标识符作用域将变为最近的封闭范围, 但不是全局。作用域范围概念: 当前作用域 (局部变量) >>> 外层作用域 (外部变量) >>> 当前模块整体作用域 (全局变量) -> python内置变量。
- `if` (如果) : 该语句用于起始执行的条件。
- `elif` (否则, 如果) : 该语句用于未满足上一条件时执行的条件。
- `else` (否则) : 该语句用于未满足所有条件时, 执行相应语句。
- `from` (从...) : 该语句用于查找指定的模块。
- `import` (导入) : 该语句用于查找指定的模块, 并在必要时对模块进行加载和初始化;
- `with` (和...一致) : 该语句用于包装执行一个由上下文管理器定义方法块。
- `for` (对于) : 该语句用于遍历一个序列的元素 (如字符串, 元组和列表) 或其他可迭代的对象。
- `while` (在...期间) : 该语句用于条件为真时的重复执行。
- `as` (如同) : 该操作符用于将对象绑定到标识符。
- `in` (在...里面) : 该操作符用于判断成员关系。
- `is` (是) : 该操作符用于判断操作符两侧是否同一对象。

这一篇教程开始, 我们再来了解一下Python中的内置函数 (Built-in Functions) 。

本篇教程为首字母A-F部分。

内置函数我们也接触过了一些, 并掌握了它们的使用, 例如: `print()`、`dict()`、`min()`、`len()`等。

下面我把Python3中所有的内置函数 (共68个), 全部整理出来, 供大家理解参考。

因为内容量较大, 本篇教程只做简单描述, 具体说明可以参考官方文档《Python 3.6.1 documentation》, 此文档可在Python安装目录中找到, 如果是默认安装, 文档路径为 “C:\Users\Administrator (或你的用户名)\AppData\Local\Programs\Python\Python36\Doc” 。

另附文档下载地址: <http://pan.baidu.com/s/1i4TYV2X> 密码: z3bu。

特别提示：如果参数带有[]，表示可以省略该参数；如果参数带有“=”，表示带有默认值；如果参数带有“*”，表示该参数填入多个。

- `abs(x)`: `abs`<绝对值函数>; 参数`x`为整数或浮点数（小数），返回值为`x`的绝对值。
- `all(iterable)`: `all`<全部>; 参数`iterable`为可迭代对象，如果对象为空值或所有元素为`True`时，返回值为`True`。
- `any(iterable)`: `any`<任何>; 参数`iterable`为可迭代对象，如果对象为空值，返回值为`False`，任何元素为`True`时，返回值为`True`。
- `ascii(object)`: `ASCII`<American Standard Code for Information Interchange,美国信息互换标准代码>; 参数`object`为对象，与`repr()`函数相同，返回值为字符串方式表示的可打印对象。当遇到非ASCII码时，就会输出`\x`，`\u`或`\U`等字符来表示。
- `bin(x)`: `bin`<二进制>; 参数`x`为整数，返回值为二进制字符串，结果是一个有效的Python表达式。如果参数`x`不是一个Python的`int`对象，它定义了一个`__index__()`方法返回一个整数。
- `bool([x])`: `bool`<布尔>; 返回值为布尔值，`True`或`False`。参数`x`通过使用标准的真值检测程序，如果`x`为假值或省略，则返回`False`；否则返回`True`。`bool`类是`int`子类，它不能进一步划分子类，并且仅具有`False`和`True`实例。
- `class bytearray([source[, encoding[, errors]])`: `byte array`<字节数组>; 返回值为新的字节数组，一个整数值区间为 $0 \leq x < 256$ 的可变序列。参数`source`可以为整数、字符串、可迭代对象，参数`encoding`为字符串（编码类型），参数`errors`为字符串。
- `class bytes([source[, encoding[, errors]])`: `bytes`<字节>; 返回值为新的字节对象，一个整数值区间为 $0 \leq x < 256$ 的不可变序列。其它与`bytearray`函数相同。
- `callable(object)`: `callable`<可调用>; 如果`object`参数是可以调用的对象，返回值为`True`；否则返回值为`False`。对象可以调用并不表示调用该对象时一定会成功，但不可调用的对象去调用时一定不会成功。注意，类可以调用（调用类将返回一个新实例）；实例的调用取决于重载是否包含`__call__()`方法。
- `chr(i)`: `chr`<Char/Code,字符/编码>返回参数`i`表示的字符，参数`i`为Unicode编码序号。例如，`CHR(97)`返回字符“A”，而`CHR(8364)`返回字符串的“€”。`ord()`函数功能与之相反。参数的有效范围是从0到1114111（16进制：0x10FFFF），如果超出此范围则会抛出`ValueError`（值错误）异常。
- `classmethod(function)`: `class, method`<类，方法>; 为函数返回一个类的方法。类的方法第一个参数需要指明类，就像类中函数，第一个参数是指明类实例。
- `compile(source, filename, mode, flags=0, dont_inherit=False, optimize=-1)`: `compile`<编译>; 将源编译成编码或AST（抽象语法树）对象。编码对象可以通过`exec()`函数或`eval()`函数执行。参数`source`可以是普通的字符串，字节字符串，或AST对象；参数`filename`是字符串的文件对象；参数`mode`是用来指定源码的类型；参数`flags`和`dont_inherit`是用来控制编译源码时的标志；参数`optimize`是用来指定编译器进行优化的等级。
- `class complex([real[, imag]])`: `complex`<复数>; 返回值为复数，可以通过`real + imag*1j`的方式创建，也可以通过字符串或数字转换而成。如果第1个参数是字符串，将被解释为复数，并且不得写入第2个参数；第2个参数不能是字符串；每个参数可以是任何数字类型（包括复数）。如果省略`imag`参数，则该参数默认值为0，这个函数就相当于`int()`函数或`float()`函数。如果省略所有参数，则返回`0j`。
- `delattr(object, name)`: `delattr`<delete/attribute, 删除/属性>; 此函数相对于`setattr()`函数，用于删除对象的属性。参数`object`为对象；参数`name`为字符串，而且必须是对象的属性。例如：`delattr(x, 'foobar')`等同于`del x.foobar`。
- `class dict(**kwargs)`/`class dict(mapping, **kwargs)`/`class dict(mapping, **kwargs)`: `dict`<字典>; 此函数用于创建一个新字典。具体参考：<http://www.opython.com/225.html>
- `dir([object])`: `dir`<目录>; 参数`object`为对象；如果省略参数，返回当前作用域范围内的属性列表；如果输入参数，则试图返回该对象的有效属性列表。

- `divmod(a, b)` : `div`<division, 除法> , `mod`<modulo, 模>; 参数`a`和`b`为数字 (非复数) , 返回值为商和余数组成的元组; 如果两个参数均为整数, 则采用整数除法, 结果等同于 $(a//b, a \% b)$ 。如果任一参数是浮点小数, 相当于 $(\text{math.floor}(a/b), a\%b)$ 。
- `enumerate(iterable, start=0)` : `enumerate`<枚举>; 返回值为枚举对象; 参数`iterable`为可迭代对象, 例如列表、数组、字典等对象; 参数`start`是枚举的起始值, 默认为0。
- `eval(expression, globals=None, locals=None)` : `eval`<evaluate, 求值>; 参数`expression`为表达式字符串; 函数可以动态地执行参数`expression`的表达式字符串。参数`globals`是全局命名空间, 可以指定执行表达式时的全局作用域的范围; 参数`locals`是局部作用域命名空间, 是用来指定执行表达式时访问的局部命名空间。
- `exec(object[, globals[, locals]])` : `exec`<execute, 执行>; 该函数支持Python代码的动态执行; 参数`object`为对象, 必须是一个字符串或代码对象。如果是字符串, 则将字符串解析为一组Python语句, 然后执行这些语句 (除非出现语法错误); 如果是代码对象, 它则只是执行。参数`globals`为全局命名空间, 用来指定执行语句时可以访问的全局命名空间; 参数`locals`为局部命名空间, 用来指定执行语句时可以访问的局部作用域的命名空间。注意, 此函数没有返回值, 即便字符串或代码对象中包含返回值语句, 例如`return`或`yield`语句。
- `filter(function, iterable)` : `filter`<过滤>; 参数`function`为函数, 参数`iterable`为可迭代对象; 函数遍历参数`iterable`所有元素, 并将每个元素通过参数`function`进行判断, 判断为`true`的元素保留, 否则跳过, 最终返回一个由保留的元素组成的可迭代对象。参数`iterable`可以是序列, 可迭代对象, 或者是支持迭代的容器。如果省略参数`function`, 所有元素将不被保留。
- `class float([x])` : `float`<浮动>; 参数`x`为整数或字符串; 函数将参数转换为浮点数。
- `format(value[, format_spec])` : `format`<格式>; 参数`value`为值; 此函数将参数`value`通过`format_spec`的格式来格式化, 并根据参数`value`的类型进行`format_spec`解释。
- `class frozenset([iterable])` : `frozenset`<冻结,集合>; 参数`iterable`为可迭代对象, 例如列表、字典、元组等; 返回值是一个新的冻结集合对象, 可从迭代对象中获取任意元素。`frozenset`是一个内置的类。冻结集合不可添加或删除任何集合里的元素。

本篇教程为首字母G-O部分。

- `getattr(object, name[, default])` : `getattr`<get, attribute/获取, 属性>; 返回值为指定对象的属性值。参数`object`为对象; 参数`name`为属性名称, 必须是字符串; 参数`default`为默认返回值。如果参数`name`是参数`object`的属性名称, 返回结果是该属性的值。例如, `getattr(x, 'foobar')` 等同于`x.foobar`。如果指定的属性不存在, 返回值参数`default`的值, 否则抛出属性异常`AttributeError`。
- `globals()` : 返回值为当前全局符号表的字典, 并且始终是当前模块的字典 (在函数或方法中定义的模块, 而不是调用的模块)。通过这个字典, 能够查询可访问的模块、函数以及变量。
- `hasattr(object, name)` : `hasattr`<has, attribute/包含, 属性>; 参数`object`为对象; 参数`name`为属性名称的字符串; 如果参数`name`是对象`object`的属性名称, 则返回值为`True`; 否则, 返回值为`False`。
- `hash(object)` : `hash`<hash, 哈希>; 参数`object`为对象; 返回值为对象`object`的哈希值。返回的哈希值是一个整数; 可用于快速查询字典的键值。如果对象`object`是数字类型, 相同的数字具有相同的哈希值, 例如1和1.0的哈希值相同。
- `help([object])` : `help`<帮助>; 参数`object`为对象; 此函数能够调用内置的帮助系统 (用于交互模式)。如果省略参数, 则会在解释器控制台启动交互式帮助系统。如果参数`object`是一个字符串, 则会查找与之相同的模块, 函数, 类, 方法, 关键字的名称, 或者是文档标题, 并在控制台显示输出帮助页。如果参数`object`是任何其他类型的对象, 则生成该对象的帮助页面。
- `hex(x)` : `hex`<十六进制>; 参数`x`为整数; 此函数能够将一个整数转换为一个前缀为“0x”的小写十六进制字符串。
- `id(object)` : `id`<identify, 标识符>返回值为对象`object`的标识符。这个标识符是一个整数, 在对象`object`同一生

命周期中保持唯一且不会改变。在不重叠的生命周期中，两个对象可以具有相同的标识符。

- `input([prompt])` : `input`<输入>; 参数`prompt`为提示字符串; 如果写入参数`prompt`, 则会被标准输出且没有换行, 然后, 函数会读取输入的内容, 将其转换为字符串 (并去除结尾换行符), 返回结果。
- `class int(x=0)/class int(x, base=10)` : `int`<integer, 整数>; 参数`x`为数字或字符串, 函数将参数`x`转换为整数, 并返回结果。如果省略参数, 则返回值为0。对于浮点数, 会下取整。参数`base`为进制, 如果写入该参数, 则`x`必须为字符串, 例如`int('1c', base=16)`, 返回值为28。
- `isinstance(object, classinfo)` : `isinstance`<is, instance/是, 实例>; 参数`object`为对象; 参数`classinfo`为类型; 此函数能够判断参数`object`对象实例是否是参数`classinfo`的实例, 如果是, 返回值为`True`; 否则, 返回值为`False`。
- `issubclass(cls, classinfo)` : `issubclass`<is, subclass/是, 子类>; 参数`cls`为类; 参数`classinfo`为类型; 此函数能够判断参数`cls`是否是参数`classinfo`的子类, 如果是, 返回值为`True`; 否则, 返回值为`False`。
- `iter(object[, sentinel])` : `iter`<iterator, 迭代器>; 参数`object`为对象; 参数`sentinel`为哨兵; 第1个参数如何解释, 取决于第2个参数; 第2个参数省略时, 对象必须是一个支持迭代 (定义了`__iter__()`函数) 或者序列 (定义了`__getitem__()`函数) 的容器, 否则, 会抛出类型错误`TypeError`异常。第2个参数写入时, 参数`object`是一个可调用的对象 (定义了`__next__()`函数), 当枚举到的值等于哨兵时, 就会抛出停止迭代异常`StopIteration`。
- `len(s)` : `len`<length, 长度>; 返回值为参数`s`的长度 (项的数量)。参数`s`可以是一个序列 (如`string`、`bytes`、`tuple`、`list`或`range`) 或集合 (例如`dict`、`set`或`frozen set`) 。
- `class list([iterable])` : `list`<列表>; 参数`iterable`为可迭代对象, 此函数返回值是一个列表。
- `locals()` : 此函数更新并返回当前系统可用的局部符号表, 返回结果是一个字典。
- `map(function, iterable, ...)` : `map`<>; 参数`function`为函数, 参数`iterable`为可迭代对象; 此函数是把参数对象`function`作为函数, 参数`iterable`对象的每一项作为参数, 通过计算输出子迭代对象。如果参数`function`允许输入多参数, 则可以输入多个`iterable`参数。当有多个`iterable`参数, 运行计算次数以项数量最少的迭代对象为准。
- `max(iterable, *[key, default])/max(arg1, arg2, *args[, key])` : `max`<最大值>; 参数`iterable`为可迭代对象, 参数`key`为函数, 参数`default`为默认值, 参数`arg` (argument) 为数值; 此函数能够返回一个迭代对象中最大的项或者多个参数中的最大值; 当输入参数`key`时, 会依据`key`的函数进行计算, 对所有计算结果取最大一项; 参数`default`为函数不能返回结果时, 返回的默认值。
- `memoryview(obj)` : `memoryview`<memory, view/内存, 查看>参数`obj` (object) 为对象; 此函数能够返回参数`obj`的内存查看对象。
- `min(iterable, *[key, default])/min(arg1, arg2, *args[, key])` : `min`<最小值>; 此函数能够返回一个迭代对象中最小的项或者多个参数中的最小值; 参数部分与`max()`函数相同。
- `next(iterator[, default])` : `next`<下一个>; 参数`iterable`为可迭代对象; 参数`default`为默认值; 此函数的返回值为参数`iterable`中下一个元素的值。如果输入参数`default`, 当下一个元素不存在时, 返回`default`参数的值, 否则抛出停止迭代异常`StopIteration`。
- `class object` : `object`<对象>; 此函数返回一个新的无特征的对象。`object`类是Python中所有类的基类, 它包含Python中所有实例共有的方法; 如果定义一个类时没有指定继承的类, 则默认继承`object`类。这个函数不接受任何参数。
- `oct(x)` : `oct`<octal, 八进制>参数`x`为整数; 此函数能够将一个整数转换为一个八进制字符串, 结果是一个有效的Python表达式。如果参数`x`不是一个Python的`int` (integer, 整数) 对象, 它必须定义一个返回整数的`index()`方法。
- `open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)` : `open`<打开>; 此函数能够打开一个文件并返回文件对象。如果文件不能打开, 则抛出操作系统异常`OSError` (Operating System Error) 。

参数file是一个类似路径的对象，可以是字符串或数组表示的文件名称，文件名称是要打开文件的路径(绝对路径或者相对路径)。

参数mode是指明打开文件的模式。

‘r’：只读模式打开文件，不可写入，此为参数默认值。

‘w’：写入模式打开文件，并且清空打开的文件。

‘x’：独占模式打开文件，如果文件已经打开，则会打开失败。

‘a’：写入模式打开文件，如果文件存在，则在文件末尾追加内容。

‘b’：二进制模式，例如读取声音、图像文件。

‘t’：文本模式，此为参数默认值。

‘+’：打开文件进行更新，可以读取或写入。

参数buffering为可选参数，用于缓冲区的策略选择。

参数encoding为文件编码，仅适用于文本文件。

参数errors为编解码错误时进行的处理，但不能在二进制模式下使用。

参数newline为文本模式指定下一行的结束符。可以是None，"，\n，\r，\r\n等。

参数closefd用于传入的文件句柄，当退出文件时，不关闭文件句柄。

参数opener用于实现自己定义打开文件方式。

- ord(c): ord<order, 次序>; 参数c为表示Unicode字符的字符串，返回值为表示该字符的Unicode代码点的整数。例如，ord('a')返回值为整数97，ord('€')(欧元符号)返回值为整数8364。此函数与chr()函数功能相反。

本篇教程为首字母P-Z部分。

- pow(x, y[, z]): pow<power, 幂>参数x、y、z均为数值；参数z省略时，函数返回值为x的y次方；参数z输入时，返回值为x的y次方模z后的余数，即pow(x,y) %z；pow(x, y)等同于x**y。
- print(*objects, sep=' ', end=' \n' , file=sys.stdout, flush=False): print<打印>; 参数object为对象；参数sep是分隔符，用于对多个输出的参数进行分隔，默认为一个空格；参数end是输出结束时的字符，默认是换行符“\n”；参数file是指定流输出到的文件，默认是系统标准输出sys.stdout；参数flush是立即把内容输出到流文件，不作缓冲。此函数能够将对象输出文本流文件中，以参数sep进行分隔，以参数end为结束符。
- class property(fget=None, fset=None, fdel=None, doc=None): property<属性>; 此函数用于设置类成员的属性；参数fget是获取属性值的函数；参数fset是用于设置属性值的函数；参数fdel是删除一个属性值的函数；参数doc是创建属性的文档字符串。
- range(stop)/range(start, stop[, step]): range<范围>; range()实际上不是一个函数，而是一个不可变的序列；参数stop是序列的终止数字，仅有此参数时，序列从0开始；参数start是序列的起始数字；参数step是步长，即前后两个数字的差值；这三个参数均为整数。
- repr(object): repr<表示>; 参数object为对象；此函数能够返回参数对象object的说明字符串。
- reversed(seq): reversed<反转>; 参数seq为序列；此函数返回结果为参数序列seq反向的可迭代对象。
- round(number[, ndigits]): round<四舍五入>; 参数number为数字；参数ndigits<n,digits/n个,数字>为保留的小数位数；此函数用于对浮点小数进行指定保留位数的四舍五入计算。
- class set([iterable]): set<集合>; 参数iterable为可迭代对象；此函数能够从可迭代对象生成集合；集合是Python的一个内置类。
- setattr(object, name, value): <set, attribute/设置, 属性>; 此函数对应函数getattr(); 参数object为对象；参数name为属性名称，必须是字符串；参数value为属性的值；此函数能够设置或者增加参数对象object的属性名称（参数name），并设置相应的值（参数value）。
- class slice(stop)/class slice(start, stop[, step]): slice<切片>; 参数stop为切片范围的终止位置；参数start为

切片范围的起始位置；参数step为步长；此函数返回一个切片范围的对象，作为切片操作中的参数。

- sorted(iterable[, key][, reverse]): sorted<排序>; 参数iterable为可迭代对象；参数key为键的比较函数；参数reverse为布尔值，用于反向排序设置；此函数能够将参数对象iterable进行排序，返回一个新的已排序的列表。
- staticmethod(function): staticmethod<static, method/静态, 方法>; 参数function为函数；此函数能够返回一个静态函数的对象，主要用作静态函数的修饰符。静态函数可以直接通过类的命名空间调用，而无需将类进行实例化；并且静态函数也可以通过类的实例进行调用。
- class str(object=" ") / **class** str(object=b" , encoding=' utf-8', errors=' strict'): str<string, 字符串>; 参数encoding为编码类型；errors为错误处理方式；此函数用于将参数对象object转换为字符串对象。
- sum (iterable[, start]): sum<总计>; 参数iterable为可迭代对象；参数start为求和的初始值；此函数能够对参数对象iterable进行求和；参数start填入时，会将求和结果再加上参数start的数值。
- super([type[, object-or-type]]): super<超级>; 参数type为类型；参数object-or-type为对象或类型；此函数返回一个代理类对象，用于访问父类或同级类。
- tuple([iterable]): tuple<元组>; 参数iterable为可迭代对象；此函数能够从参数对象iterable生成一个元组对象。
- class type(object) / **class** type(name, bases, dict): type<类型>; 参数object为对象；参数name为类的名称；参数bases为基类的元组；参数dict为类中定义的命名空间；此函数只填入一个参数时，返回结果为参数对象object的类型；当填入3个参数时，返回一个新的类型的对象。
- vars([object]): vars<variables, 变量>; 参数object为对象；此函数返回一个包含参数对象object属性与属性值的字典对象，参数object可以是模块、类、实例或者其它对象。如果省略参数，相当于函数locals()。
- zip(*iterables): zip<拉链>; 参数iterables为多个可迭代对象；此函数能够从每个可迭代对象中逐一获取元素创建一个新的可迭代对象。如果在参数iterables前添加 "*" ，则会进行逆向处理，进行分离。
- __import__(name, globals=None, locals=None, fromlist=(), level=0): import<导入>; 此函数由导入模块的语句调用，以实现动态地加载模块。参数name为模块名称；参数globals为全局，参数locals为局部，指定这两个参数是用于判定如何在包的上下文中解释名称；参数fromlist为对象或子模块的名称，这些名称应该能够从参数name指定的模块中导入；参数level为级别，指定使用绝对或相对导入，默认值为0，即绝对导入。

自定义函数

在我们编程的过程中，往往要写一些进行某种运算的代码，通过这些代码获取我们想要的结果。

有时候，这些运算的代码，我们需要使用到多次，为了避免大量重复编写这些代码，我们可以把这些运算过程的代码定义为函数。

在第一篇教程中，我们见过下面这张图片，这张图片就是我们调用函数时要写入的内容。

函数名称 左括号 参数 右括号

print('小楼好帅')

那么，定义函数也必须包含这些内容。

另外，还要包含定义函数的关键字以及函数中运算过程的代码。

例如，我们定义一个获取身份证号中出生日期的函数。

示例代码：


```

1 def get_birthday(id): # 定义函数名称并设定参数
2     birthday = id[6:14] # 函数运算过程
3     print(birthday) # 函数运算过程
4
5 get_birthday('110123199001012121') # 调用函数，显示输出结果为：19900101
6 get_birthday('120122199508081321') # 调用函数，显示输出结果为：19950808

```

通过上方的代码，大家能够看到，定义函数时的一些关键内容：

- 1、要使用def这个关键字进行定义。
- 2、函数的名称建议使用小写单词组成，单词间以下划线分隔（下划线命名法），这样比较方便理解函数名称的含义。
- 3、参数是一个自定义的变量名称，通常也是使用小写的单词，用于提示输入的参数内容。
- 4、定义参数名称与参数以“:”结尾。
- 5、在定义函数名称与参数的下方，向右缩进编写运算代码的语句块。
- 6、通过函数名称并写入相应的参数即可调用函数，以实现相应的运算。

不过，上方我们定义的函数，并没有返回结果，而是直接在函数运算过程中进行了显示输出。

如果，我们需要获取到的出生日期，再进行使用，我们需要修改一下这个函数。

示例代码：

```

1 def get_birthday(id): # 定义函数名称并设定参数
2     birthday = id[6:14] # 函数运算过程
3     return birthday # 返回运算结果
4
5
6 b = get_birthday('110123199001012121') # 调用函数
7 print('您的出生日期是%s。' % b) # 显示输出结果为：您的出生日期是19900101

```

对比一下我们刚才的两段定义函数的代码，大家能够看到，第二段代码中我们通过return关键字，返回了运算结果（如果不加入return语句，则函数默认执行return None，即函数返回结果为None）。

这样，我们就可以在调用函数时，通过一个变量（例如上方代码中的b变量）接收到这个结果，进行使用。

注意：return语句会跳出结束函数，并返回结果，该语句之后的语句将不再被执行。

上面我们定义的函数，只返回了一个结果。

如果，我们想把身份证号码中出生日期的年、月、日分别获取到呢？

接下来，我们再修改一下这个函数。

示例代码：

```

1 def get_birthday(id): # 定义函数名称并设定参数
2     year = id[6:10] # 函数运算过程
3     month = id[10:12] # 函数运算过程
4     day = id[12:14] # 函数运算过程
5     return year, month, day # 返回多个运算结果
6
7 y, m, d = get_birthday('110123199001012121') # 调用函数，将3个返回值存入3个变量
8 print('您的出生日期是%s年%s月%s日。' % (y, m, d)) # 显示输出结果为：您的出生日期是1990年01
    月01日。

```

通过上面的代码，大家可以看到，我们可以通过return关键字，返回多个运算结果。

实际上，函数返回的仍然是1个运算结果。

我们再来看下方这段代码。

示例代码：

```

1 def get_birthday(id): # 定义函数名称并设定参数
2     year = id[6:10] # 函数运算过程
3     month = id[10:12]
4     day = id[12:14]
5     return year, month, day
6
7 birthday = get_birthday('110123199001012121') # 调用函数，返回结果存入变量
8 print(birthday) # 显示输出结果为：('1990', '01', '01')
9 print('您的出生日期是%s年%s月%s日。' % birthday) # 显示输出结果为：您的出生日期是1990年01月
    01日。

```

通过上方代码，我们能够看到，我们可以通过1个变量接收函数的返回结果，这时我们得到的是1个包含了3个元素元组。

在我们定义函数时，有些时候希望某些参数可以省略。

例如，我们通过身份证号码，获取出生日期或者年龄。

我们可以通过设定一个参数来控制函数如何返回结果。

示例代码：

```

1 def get_birthday(id, get_age=False): # 定义函数名称并设定参数,get_age为关键字参数,默认值为
    False
2     if get_age: # 对参数进行判断
3         return 2017 - int(id[6:10]) # 返回计算公式的计算结果
4     else:

```

```

5         year = id[6:10] # 函数运算过程
6         month = id[10:12] # 函数运算过程
7         day = id[12:14] # 函数运算过程
8         return year, month, day # 返回多个变量值
9
10 result = get_birthday('110123199001012121', True) # 调用函数，输入2个参数
11 print(result) # 显示输出结果为: 27
12 result = get_birthday('110123199001012121') # 调用函数，省略第2个参数
13 print(result) # 显示输出结果为: ('1990', '01', '01')

```

在上方这段代码中，我们新增了一个**关键字参数**，并为其设置默认值。

这样，我们在调用参数时，如果不填写第2个参数，一样可以执行函数。

此时，函数会读取第2个参数的默认值，参与运算过程。

如果，我们只定义参数而不写入默认值，即便函数运算过程中没有使用这个参数，程序也会抛出异常。

异常示例代码：

```

1 def get_birthday(id, get_age): # 定义函数名称与参数
2     birthday = id[6:14] # 函数运算过程
3     return birthday # 返回运算结果
4
5 result = get_birthday('110123199001012121') # 调用函数，此时会抛出异常

```

异常信息：

异常提示中TypeError是类型错误，提示内容为：get_birthday()函数丢失了1个必须位置的参数'get_age'。

所以，我们在定义函数时，设定的参数如果没有设置默认值，则调用函数时必须填入该参数。

那么，设置默认值的参数我们叫它**关键字参数**，没有默认值的参数是**位置参数**。

关键字参数对位置没有要求，不管是设定参数还是调用函数时，可以任意顺序写入。

而位置参数必须有着严格的位置和顺序，这样才能在调用函数时一一对应。

正是因为这个原因，当参数中同时包含位置参数和关键字参数时，位置参数要放在所有关键字参数的前面。

大家可以尝试将位置参数放在关键字参数的后方，运行程序会抛出异常提示。

SyntaxError: non-default argument follows default argument

语法错误：无默认值参数跟随默认值参数（即位置参数在关键字参数后方）

另外，我们定义的函数，往往对输入的参数有类型的要求，当输入了错误的类型，会有异常发生。

例如，我们定义一个减法运算的函数，参数为两个整数。

示例代码：

```

1 def subtraction(num1, num2): # 定义函数并设定参数

```



```

2     result = num1 - num2 # 函数运算过程
3     return result # 返回运算结果
4
5
6 print(subtraction(3, 2)) # 调用函数,并显示输出结果: 1。

```

上方代码中,我们调用函数时输入了两个整数,运行正常,得到了计算结果。
接下来,我们输入错误的参数,比如两个字符。

```

1 def subtraction(num1, num2): # 定义函数并设定参数
2     result = num1 - num2 # 函数运算过程
3     return result # 返回运算结果
4
5
6 print(subtraction('3', '2')) # 调用函数,程序抛出异常。

```

上方的代码运行之后,程序抛出异常。

TypeError: unsupported operand type(s) for -: 'str' and 'str' 。

提示内容为类型错误,不支持的操作数类型: 'str' 和 'str' 。

也就是说,字符串类型的参数不能进行减法运算。

这是系统给出的提示,我们也可以自定义这个异常的提示。

我们知道减法可以支持整数与小数,所以我们在函数中可以判断传入的参数值是否这两种类型。

如果不是,则给出异常提示。

示例代码:

```

1 def subtraction(num1, num2): # 定义函数并设定参数
2     if not isinstance(num1, (int, float)) \
3         or not isinstance(num2, (int, float)): # 判断参数是否指定类型,代码过长时可以使用"\n"换行
4         raise TypeError('参数类型错误,参数必须为整数或者小数。') # 设定自定义异常内容
5     result = num1 - num2 # 函数运算过程
6     return result # 返回运算结果
7
8
9 print(subtraction('3', 2)) # 调用函数,程序抛出异常。

```

上方代码中,我们使用内置函数isinstance()对输入的参数值进行类型的比较,当任何一个参数不是int(整数)或者float(小数)类型时,都将抛出异常。

关键字raise用于引发异常，TypeError()为异常类型，括号中可以输入自定义的异常提示。

最后，在我们编写代码时，我们还可以定义空函数。

空函数一般是在还没有确定函数内部代码如何编写，但是又需要不影响程序运行时使用。我们只需要在函数内容写入pass关键字即可。

示例代码：

```
1 def subtraction(num1, num2): # 定义函数并设定参数
2     pass # 使用pass占位，保证程序运行正常
3
4
5 print(subtraction(1,2)) # 调用函数，返回结果为：None。
```

在上一篇教程中，我们了解了函数的定义以及参数的设定。

这一篇教程，我们继续了解一些更加灵活的参数设定。

先来看一段示例代码。

例如，我们定义一个根据输入的姓名参数，返回一个姓名列表的方法。

示例代码：

```
1 def creat_name_list(name_list, name1, name2, name3): # 定义函数并设定参数
2     name_list.append(name1) # 为列表添加新元素
3     name_list.append(name2)
4     name_list.append(name3)
5
6 lst = [] # 创建空列表
7 creat_name_list(lst, '小楼老师', '苍井老师', '吉泽老师') # 调用函数并传入参数
8 print(lst) # 显示输出结果为：['小楼老师', '苍井老师', '吉泽老师']
```

特别说明：在上方的代码中，大家应该能够注意到，虽然我们没有对lst这个列表进行直接操作，但是它发生了改变。也就是说我们在函数外部创建的lst和函数内部的参数name_list是同一个列表。这样的操作仅限于可变的数据结构，而数字、字符串以及元组是不能够被改变的，所以无法进行这样的修改。

示例代码：

```
1 def change_number(num): # 定义函数并设定参数
2     num = 1 # 修改参数变量的值
3
4 n = 0 # 创建变量
```

```
5 change_number(n) # 调用函数并传入参数
6 print(n) # 显示输出结果为:0
```

以上是关于对修改函数外部变量的举例，让我们能够知道当外部变量是不可变的数据结构，作为参数传入函数时，在函数内部的修改操作无法对外部变量产生影响；而当外部变量是可变的数据结构，作为参数传入函数时，在函数内部的修改能够让外部变量发生改变。

接下来，我们回过头来看第一段代码。

当我们希望输入多个姓名，创建一个姓名列表时，往往姓名的数量是不固定的。

而我们定义参数时，位置参数或者关键字参数的数量却是固定的。

很显然，当我们想写入4个姓名时，第一段代码没有办法满足我们的需求。

我们只有通过将参数增加为4个才能实现。

但是，如果忽然我们改变了主意，要写入5个姓名呢？

所以，改变函数的参数数量并不是一个好方法。

这里，我们来使用**收集参数**来解决这个问题。

示例代码：

```
1 def creat_name_list(name_list, *names): # 定义函数并设定参数,*names为收集参数
2     print(type(names)) # 显示输出参数names的数据类型，结果为: <class 'tuple'>
3     if names: # 判断收集参数names不为空值
4         for name in names: # 循环遍历names
5             name_list.append(name) # 为列表添加新元素
6
7 lst = [] # 创建空列表
8 creat_name_list(lst, '小楼老师', '苍井老师', '吉泽老师', '樱井老师') # 调用函数并传入参数
9 print(lst) # 显示输出结果为: ['小楼老师', '苍井老师', '吉泽老师', '樱井老师']
```

在上方的代码中，大家能够看到，参数中除了位置参数name_list，只设定了一个参数“*names”。

这个names在第2句代码中，我们获取到它是一个元组类型。

也就是说，这个收集参数是一个元组类型的变量，它把我们传入的多个姓名（无论几个），全部存放在了一个元组中。

然后，我们对names进行循环遍历，就能够取出这些姓名，添加到列表中。

到这里，大家一定都已知道收集参数的设定只需要给变量名称前面加上一个拼接（Splicing）操作符“*”就可以了。特别说明：上方代码中，是在函数外部创建了列表，并作为参数传入到函数中。我们也可以在函数内部去创建这个列表，然后通过return语句返回。

示例代码：

```
1 def creat_name_list(*names): # 定义函数并设定参数,*names为收集参数
2     print(type(names)) # 显示输出参数names的数据类型，结果为: <class 'tuple'>
3     name_list = list(names) # 将元组names转换为列表并存入变量name_list
```

```

4     return name_list # 返回列表
5
6
7 lst = creat_name_list('小楼老师', '苍井老师', '吉泽老师', '樱井老师') # 调用函数并传入参数，通过变量接收返回值
8 print(lst) # 显示输出结果为: ['小楼老师', '苍井老师', '吉泽老师', '樱井老师']

```

另外，也可以通过收集参数的方式，灵活的传入多个关键字参数。

设定参数时，在变量名称前加上拼接操作符 “*” 即可。

例如，我们通过输入用户的姓名、年龄、身高、体重信息，创建一个用户信息的列表。

示例代码：

```

1 def creat_user(**user_info): # 定义函数并设定参数,*user_info为收集参数
2     print(type(user_info)) # 显示输出参数user_info的数据类型，结果为: <class 'dict'>
3     return user_info # 返回参数值
4
5
6 user=creat_user( name='小楼老师',age=18, height='182',weight= '90KG') # 调用函数并传入参数
7 print(user) # 显示输出结果为: {'name': '小楼老师', 'age': 18, 'height': '182', 'weight': '90KG'}

```

在上方代码中，我们设定了一个收集参数获取所有传入的关键字参数。

在代码第二句中，我们能够获取到变量user_info的类型为字典。

那么，这个字典我们可以直接作为返回值，通过return语句返回。

在之前的案例中，大家能够看到拼接操作符 “*” 和 “**” 在设定参数时的作用。

实际上，这两个操作符，我们还能够反过来使用。

示例代码：

```

1 def get_volume(length, width, height): # 定义函数并设定参数
2     return length * width * height # 返回计算结果
3
4 params = (3, 4, 5) # 创建参数元组
5 print(get_volume(*params)) # 显示输出结果为: 60

```

在上方的示例代码中，我们定义了一个计算体积的函数，并且设定了固定数量的参数。

然后，我们有一个变量params，是一个元素数量与函数参数数量相等的数组。

当我们调用这个函数时，就无需把数组中的元素取出，而是直接在变量名称前面加上星号 “*”，做为参数传入函数中。

这样的做法和我们分开传入参数的结果是一样的。

因为，星号 “*” 可以解析可迭代序列。

当然，“**” 也可以类似使用。

例如，我们通过一组用户信息，返回一个用户描述的字符串。

示例代码：

```
1 def user_info(name,age,height,sex='男'): # 定义函数并设定参数
2     info = '姓名-%s 性别-%s 年龄-%s 身高-%s'% (name,sex,age,height) # 格式化字符串
3     return info # 返回结果
4
5 params = {'name': '小楼', 'age': '18', 'height': '180CM'} # 创建参数字典
6 print(user_info(**params)) # 调用函数，并传入参数，显示输出结果为：姓名-小楼 性别-男 年龄-18 身高-180CM
```

另外，还要补充说明一点。

单星号 “*” 可以解析可迭代序列的作用不仅限在参数处理中。

举个小例子，让两个元组混合为集合。

示例代码：

```
1 s1 = {(1, 2, 3), (3, 4, 5)}
2 s2 = {*(1, 2, 3), *(3, 4, 5)}
3 print(s1) # 显示输出结果为：{(3, 4, 5), (1, 2, 3)}
4 print(s2) # 显示输出结果为：{1, 2, 3, 4, 5}
```

这一篇教程，我们先来看一段代码。

示例代码：

```
1 x = 0 # 全局变量
2
3 def outside(): # 定义函数
4     x = 1 # 局部变量，内嵌函数的外部变量
5
6     def inside(): # 定义内嵌函数
7         x = 2 # 局部变量
8         return x
9
```

```
10     return x, inside # 将变量值和函数返回
11
12 o, i = outside() # 通过两个变量接收outside函数的返回值x和inside
13 print(x) # 显示输出结果为: 0
14 print(o) # 显示输出结果为: 1
15 print(i()) # 显示输出结果为: 2
```

刚才的这段代码，可能不太容易理解。

因为里面包含了几部分我们没有接触过的内容。

第一部分，我们先来了解一下函数嵌套。

在Python中可以在函数的内部再定义函数。

大家能够看到，在上方代码中函数outside的内部，又定义了一个函数inside。

这种结构就是函数嵌套。

另外，在上方代码中，大家能够看到函数的返回值不仅可以返回多个，而且可以返回内嵌函数（这是闭包，后面会讲到）。

返回内嵌函数时，如果函数名称后方没有加上“()”，调用外层函数时不会立即执行返回的函数，需要在调用外层函数后，添加“()”来执行。例如，上方代码最后一句中的“i()”，就是执行变量中保存的函数。

而且，我们也可以下面这种方法去执行返回的函数。

示例代码：（接上一段代码）

```
1 outside()[1]() # 调用外层函数时，获取的返回值列表中第2项是函数，加上()则会被执行。
```

这是外层函数有多个返回值时的方法，通过对外层函数返回值列表进行索引，找到函数执行。

如果，外层函数只有一个返回值，我们可以通过函数名称后方直接加上“()”去执行返回的函数。

另外，在外层函数返回内嵌函数时，在函数名称后方加上了“()”，会在调用外层函数时自动执行。

大家可以通过运行以下代码进行对比。

示例代码：（返回函数不带括号）

```
1 def outside():
2     print('执行外层函数!')
3
4     def inside():
5         print('执行内嵌函数!')
6
7     return inside
8
9 outside()
```

示例代码：（返回函数带有括号）

```
1 def outside():
2     print('执行外层函数! ')
3
4     def inside():
5         print('执行内嵌函数! ')
6
7     return inside()
8
9 outside()
```

第二部分，我们来了解变量的作用域。

在上方代码中，我们能够看到有3个变量的名称都是x。

这3个x并非同一个变量，只是名称一样，它们的特点可以这么理解：

- 最外层的变量x是全局变量
- 函数outside中和inside定义的变量x，都是局部变量。
- 函数outside中的变量x，相对于内嵌的函数inside，它是外部变量。

这3个变量有着不同的作用域。

```
1 # -*- coding: utf-8 -*-
2 x = 0 # 全局变量
3
4
5 def outside():
6     x = 1 # 局部变量，相对inside函数为外部变量
7
8     def inside():
9         x = 2 # 局部变量
10        return x # 返回局部变量x的值 (2)
11
12    return x, inside # 返回局部变量x的值 (1)和inside函数
13
14
15 o, i = outside() # 调用函数，获取返回结果x的值 (1)写入变量o和inside函数，分别写入变量o和i。
16
17 print(x) # 显示输出全局变量x的值，结果为：0
18 print(o) # 显示输出变量o中写入的x的值，结果为：1
19 print(i()) # 显示输出变量i中写入的inside函数返回的x的值，结果为：2
20
```

如上图所示，3个x变量的作用域如下：

- 最外层的变量x，它的作用域是整个模块中。
- 函数outside中的变量x，作用域是函数outside内部以及内嵌函数inside。
- 函数inside中的变量x，作用域是函数inside内部。

也就是说，全局变量作用域是当前模块内，而局部变量作用域是函数以及内嵌函数中。

但是，这里大家可能会有疑问，既然全局变量作用域是全局的，那么在函数中又创建同名变量不是冲突了吗？

这个不用担心，当在函数中创建与全局变量同名的局部变量时，在函数内部会自动屏蔽同名的全局变量。

所以，在本文开头的代码中，显示输出的结果并不相同。

下图就是3个变量x的值输出显示时的过程。

```

1  # -*- coding: utf-8 -*-
2  x = 0 # 全局变量
3
4
5  def outside():
6      x = 1 # 局部变量，相对inside函数为外部变量
7
8      def inside():
9          x = 2 # 局部变量
10         return # 返回局部变量x的值(2)
11
12     return x, inside # 返回局部变量x的值(1)和inside函数
13
14 o, i = outside() # 调用函数，获取返回结果x的值(1)写入变量o和inside函数，分别写入变量o和i。
15
16 print(x) # 显示输出全局变量x的值，结果为：0
17 print(o) # 显示输出变量o中写入的x的值，结果为：1
18 print(i()) # 显示输出变量i中写入的inside函数返回的x的值，结果为：2
19
20

```

到这里，我们有必要来了解一下作用域的概念。

作用域也叫命名空间，命名空间是一个我们看不到的字典，字典的键记录变量的名称，字典的值记录着变量的值。每个模块都会创建一个全局命名空间，用于记录模块的变量，包括全局变量和局部变量，以及其它导入模块中的变量。

每个函数调用时都会创建一个局部命名空间，用于记录函数的变量，包括函数的参数和函数内部创建的变量。

另外，还有内置命名空间，任何模块都能够访问，记录了内置函数和异常。

第三部分，我们来了解如何在函数中读取与修改全局变量。

首先，我们先来看读取全局变量。

一般情况下，我们在函数中可以直接读取全局变量。

这里我们看一段根据半径计算圆形周长的代码。

示例代码：

```

1  radius = 21 # 创建半径的变量并赋值
2
3  def get_perimeter(): # 定义获取周长的函数
4      perimeter = round(2 * 3.14 * radius, 2) # 创建周长的变量保存计算结果
5      return perimeter # 返回结果
6
7  print(get_perimeter()) # 显示输出结果为：131.88

```

在上方代码中，我们定义了全局变量radius，在函数中我们可以直接调用这个变量参与计算。

但是，如果我们在函数中创建的局部变量如果与全局变量同名，这时候怎么调用全局变量呢？

我们可以借用内置函数globals来获取到全局变量。

示例代码：

```

1  def global_case(): # 定义函数
2      x = 5 # 创建同名局部变量
3      result = x * globals()['x'] # 通过globals函数获取全局变量x与局部变量x相乘
4      return result # 返回结果

```


5

```
6 print(global_case()) # 显示输出结果为: 15
```

接下来，我们来看如何在函数中修改全局变量（或者叫重新绑定全局变量为其他值）。

当我们创建了某个全局变量，在函数中无法直接修改这个变量。

就像之前的代码中，即便在函数中给x赋值（例如x=1），也只是创建了一个新的局部变量，而不是对全局变量x进行修改。

那么，如何在函数中修改全局变量呢？

我们可以使用global关键字，声明要重新绑定的全局变量。

这样，Python解释器就能够知道，我们是要对已有的全局变量进行修改，而不是创建一个新的同名局部变量。

示例代码：

```
1 x = 10 # 创建全局变量
2
3 def change_global(): # 定义函数
4     global x # 声明要修改的全局变量
5     x = 5 # 修改变量值
6
7 change_global() # 调用函数
8 print(x) # 显示输出全局变量x的值，结果为: 5
```

在上方代码中，大家能够看到，我们定义了全局变量x和修改这个全局变量的函数change_global。

在修改全局变量的函数change_global中我们并没有设定返回值，它只起到修改的作用。

所以，当我们执行了这个函数，再显示输出变量x的值，就是经过修改后的值。

补充说明：如果想在内嵌函数中修改外层函数中的局部变量，可以使用关键字nonlocal进行声明。使用方法和关键字global类似。

示例代码：

```
1 def nonlocal_case():
2     x = 10 # 创建局部变量
3
4     def change_nonlocal(): # 定义函数
5         nonlocal x # 声明要修改的外部变量
6         x = 5 # 修改变量值
7
8     change_nonlocal() # 调用修改函数执行修改
9     return x # 返回变量x的值
10
```

```
11 print(nonlocal_case()) # 显示输出全局变量x的值，结果为：5
```

第四部分，我们来了解一下闭包（closure）的概念。

在网上查了闭包的解释，百度百科中是这样说的：闭包由要执行的代码块（内嵌函数）和为自由变量（外部变量）提供绑定的计算环境（作用域/命名空间）组成。

注意，上面这段话中，括号里面的内容是我按照自己的理解添加的。

以我个人的观点，闭包就是由**返回的内嵌函数**和这个内嵌函数中用到的**外部变量（1）**以及**其他函数（2）**打包而成的一个整体。

（1）外部变量，包括：外层函数的参数变量以及外层函数中定义的局部变量

（2）其它函数，包括：外层函数中定义的其他内嵌函数

例如，在本文第一段代码中就有闭包的出现。

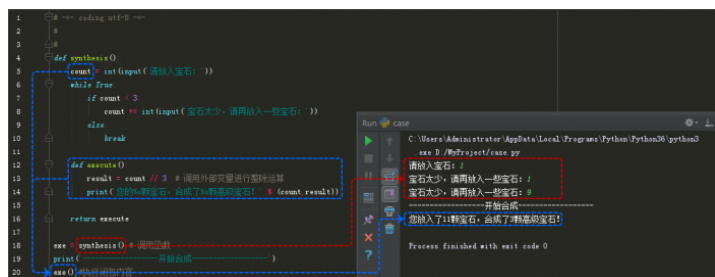
为了更清楚的解释闭包的概念，我们来看一段代码。

用户输入合成的宝石数量，然后执行合成计算。

示例代码：

```
1 def synthesis():
2     count = int(input('请放入宝石: '))
3     while True:
4         if count < 3:
5             count += int(input('宝石太少，请再放入一些宝石: '))
6         else:
7             break
8
9     def execute():
10         result = count // 3 # 调用外部变量进行整除运算
11         print('您放入了%s颗宝石，合成了%s颗高级宝石!' % (count,result))
12
13     return execute
14
15 exe = synthesis() # 调用函数
16 print('-----开始合成-----')
17 exe() #执行闭包内容
```

这段代码中，闭包包含的内容，如下图所示。



如图所示，外层函数被调用，执行return语句时，会将内嵌函数返回，形成闭包存入变量exe。

exe中的闭包内容包括：外层函数中变量count和内嵌函数excute。

exe()语句会执行闭包中的内容，我们可以认为执行了以下语句。

```
1 count = * # "*"表示外层函数执行后，变量count的最终值
2 def execute():
3     result = count // 3 # 调用外部变量进行整除运算
4     print('您的%s颗宝石，合成了%s颗高级宝石!' % (count, result))
5     return execute
6 exe = synthesize() # 调用函数
7 print(exe()) # 开始合成
```

这一篇教程的学习目标是了解什么是递归（Recursion）。

简单来说，递归就是函数自己调用自己。（听起来...好淫荡...）

但是，自己调用自己不会变成无限循环调用么？

例如下面这个代码：

```
1 def recursion():
2     return recursion()
3
4 recursion()
```

代码运行后会抛出异常，RecursionError: maximum recursion depth exceeded

意思是，递归错误：超过最大递归深度

也就是说，因为函数不停的循环调用自身超过了一定次数导致的异常。

这种叫无穷递归（Infinite Recursion），一般来说并没有什么用。

我们需要有用的递归。

比较经典的应用有阶乘运算、幂运算和二分查询（看到这些词语别晕，实际上很简单）。

我们先来看阶乘运算。

阶乘：一个正整数的阶乘是所有小于及等于该数的正整数的积

举例来说，5的阶乘就是5*4*3*2*1，4的阶乘就是4*3*2*1。

其实，阶乘的运算如果不使用递归我们也可以实现。

示例代码：

```
1 def factorial(n):
2     result = n
3     for i in range(1, n):
4         result *= i
5     return result
6
7 print(factorial(5)) # 显示输出结果为: 120
```

然后，我们来看使用递归方式的实现。

示例代码：

```
1 def factorial(n):
2     if n == 1:
3         return 1
4     else:
5         return n * factorial(n - 1) # 函数中调用函数自身
6
7 print(factorial(5)) # 显示输出结果为: 120
```

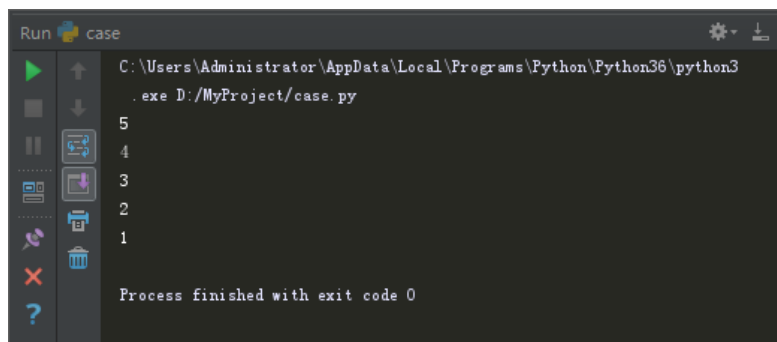
递归方式实现的阶乘，好像有些不太容易理解。

我们把计算过程中，参数变量的值通过print语句显示出来看一下。

修改后的代码：

```
1 def factorial(n):
2     print(n) # 显示输出参数值
3     if n == 1:
4         return 1
5     else:
6         return n * factorial(n - 1)
7
8 factorial(5)
```

当我们运行代码，得到如下结果：



```
Run case
C:\Users\Administrator\AppData\Local\Programs\Python\Python36\python3
.exe D:/MyProject/case.py
5
4
3
2
1
Process finished with exit code 0
```

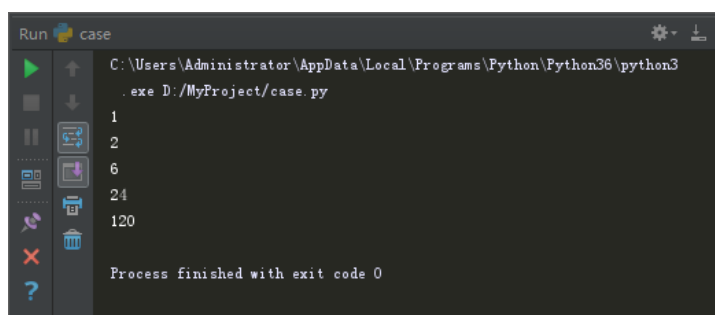
这意味着print语句被执行了5次，同时也意味着函数被调用了5次。

先记下这个特点，我们继续通过print语句把每次通过return语句计算的结果也显示出来。

修改后代码如下：

```
1 def factorial(n):
2     if n == 1:
3         print('1') # 显示输出当前的阶乘结果
4         return 1
5     else:
6         current = n * factorial(n - 1)
7         print(current) # 显示输出当前的阶乘结果
8         return current
9
10 factorial(5)
```

当我们运行代码，得到如下结果：

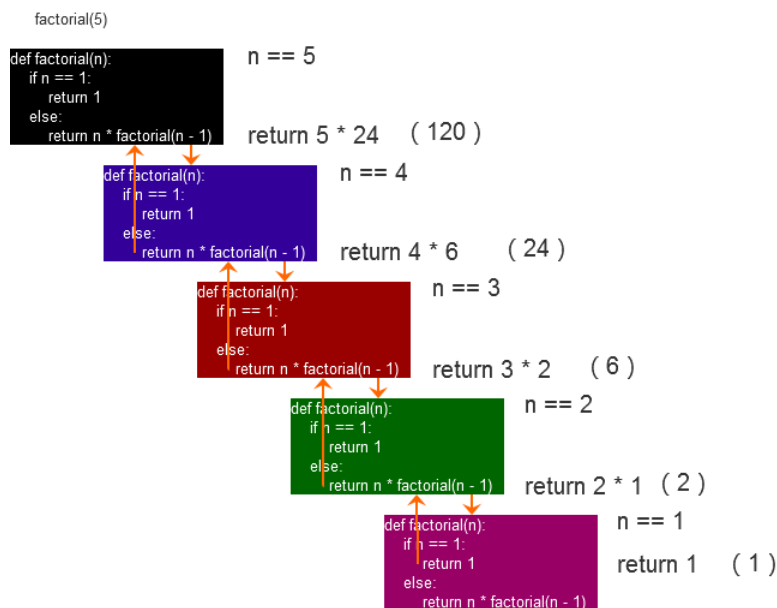


```
Run case
C:\Users\Administrator\AppData\Local\Programs\Python\Python36\python3
.exe D:/MyProject/case.py
1
2
6
24
120
Process finished with exit code 0
```

这个结果意味着，每次调用函数都会进行一次计算，并且将计算结果通过return语句返回。

但是，递归的具体执行过程仍然很模糊。

我们来看一个示意图。



上方5个色块，是代码调用了5此函数。

每次调用函数都会创建新的命名空间，大家可以理解为程序执行了5个同名的函数。

既然执行了5个函数，就有参数传入和返回结果的过程。

我们先来看调用函数时，参数传入的过程。

- 函数首次调用时，参数n的值为5；
- 首次调用函数的return语句中，进行了第二次调用函数，并设置参数为n-1；所以，在第二次调用的函数中，参数n的值变成了4；
- 以此类推，直至终止调用函数自身为止。

接下来，我们再来看返回函数执行结果的过程。

- 程序调用5次函数的同时，进行了参数的传入，第5次调用时，参数n的值是1；此时，参数数值满足n == 1的条件，不再继续调用函数自身，通过return语句返回值，也就是1；
- 当1这个值被返回，程序回到了倒数第2次函数调用的return语句，此时语句中对函数的最后一次调用变成了具体的值（1），和变量n相乘之后，作为返回值，再次返回给倒数第3次函数调用的return语句中；
- 以此类推，直至返回到首次调用的函数为止。

所以，在我们刚才的运行结果截图中，我们能够到，参数的显示结果是5、4、3、2、1的顺序（依次调用函数的顺序）；而返回值的显示结果是1、2、6、24、120的顺序（从最后一次调用函数向前返回计算结果的顺序）。

所以，递归可以这么理解，它就是函数循环调用函数自身；递是指在循环调用的过程中，将参数递进下一次函数的调用；归是指当满足终止循环调用函数的条件时，按照和调用时相反的顺序，将函数的执行结果依次回归。

接下来，我们再来看一下幂的计算。

例如，2的4次方实际上是2*2*2*2的乘积。

示例代码：

```
1 def power(x, y): # 定义函数计算参数x的y次方  
2     if y == 1: # 满足条件时终止函数循环调用  
3         return x # 参数x的1次方返回参数x  
4     else:  
5         return x * power(x, y - 1) # 调用函数自身形成递归  
6
```

```
7 print(power(2, 4)) # 显示输出结果为: 16
```

上方代码中，当我们调用函数power时，函数开始循环调用函数自身，并且依次将参数y (3,2,1) 传入每次调用的函数中，当参数y为1的时候，结束函数自身的循环调用，并且依次返回值 (2,4,8,16) 。

最后，我们再来看一下二分查询的案例。

例如，有一个0~100的数字列表，从中查询到一个指定的数字。

示例代码：

```
1 def search(seq, number, lower, upper): # 定义函数,参数seq为要查询的序列,参数number为要查
    询的数字
2     # 参数lower为查找区间的下限值,参数upper为查找区间的上限值
3     if lower == upper: # 查找区间仅剩一个数字时，即找到查询结果，停止循环调用
4         assert number != seq[upper], '这里有问题!' # 如果不相等会抛出异常AssertionError
            (断言错误),可以尝试把"=="改为"!="
5         print(str(lower)) # 显示输出结果为: 66
6     else: # 查找区间有多个数字时，继续查找
7         middle = (lower + upper) // 2 # 通过整除计算，获取查找区间数字的中间值
8         if number > seq[middle]: # 判断查找的数字是否大于中间值
9             return search(seq, number, middle + 1, upper) # 如果大于中间值，中间值作为查
                找区间下限值，继续查找
10        else:
11            return search(seq, number, lower, middle) # 如果小于中间值，中间值作为查找区
                间上限值，继续查找
12
13
14 search(range(0, 100), 66, 0, 100) # 显示输出结果为: 66
```

大家根据代码中的注释对递归过程进行理解，在此不再赘述。

不过在上方代码中，我添加了一个assert关键字，此关键字为断言，可以帮助我们运行程序检查语句中的错误。

使用格式为：assert 表达式，‘自定义的异常说明字符串’

自定义的异常说明字符串可以省略，未省略时，字符串和表达式之间必须用逗号(,)分隔。

如果断言语句中的表达式出现错误，则会抛出AssertionError异常。

如果自定义了异常说明字符串，则会显示 “AssertionError: 自定义的异常说明字符串” 。

列表推导式 (List Comprehension) 和lambda表达式

列表推导式

列表推导式的官方定义：一种采用简洁的方式来处理序列中的全部或部分元素，并返回结果列表。

定义中的序列实际上是指可迭代对象。

我们先来看一个例子，创建一个整数1~6平方的列表。

示例代码：

```
1 lst = [x * x for x in range(1, 7)] # 列表推导式：循环获取范围1~6的整数，计算乘积后添加到新列表。
2 print(lst) # 显示输出结果为：[1, 4, 9, 16, 25, 36]
```

上方代码中，创建列表时使用了列表推导式。

后方的for循环能够从1~6的范围取出整数，前方的计算公式将取出的整数自身相乘。

每一次循环都会取出数字和计算乘积，添加到一个新的列表中。

除了迭代取出元素和设定计算方法形成列表，列表推导式还能够设置条件，满足条件的元素才能够经过计算添加到列表。

例如，我们只需要1~6中偶数的平方。

示例代码：

```
1 lst = [x * x for x in range(1, 7) if x % 2 == 0] # 列表推导式：循环获取范围1~6的整数，符合条件时，计算乘积后添加到新列表。
2 print(lst) # 显示输出结果为：[4, 16, 36]
```

上方代码中，我们加入了条件，当取出的元素取余2为0时，进行计算。

由此我们能够看出列表推导式的组成为：[元素(或计算方法) for循环(允许多个) if语句]

再来看个例子，从整数列表中取出小于3的元素，并从平方列表中取出对应的元素，组成算式列表。

期待显示输出结果为：['1²=1' , '2²=4' , '3²=9']

示例代码：（错误示例）

```
1 number = [1, 2, 3, 4, 5, 6] # 整数列表
2 square = [1, 4, 9, 16, 25, 36] # 平方列表
3 lst = ['{0}2={1}'.format(str(x), str(y)) for x in number for y in square if x <= 3]
4 print(lst)
```

上方代码运行结果为：

运行的结果和我们期待的不一样。

实际上，在列表推导式中有多个循环时，会出现嵌套循环的效果，而不是同步循环的效果。

也就是说，当前面的循环取出第1个元素，后方的循环会进行一轮迭代；当前面的循环取出第2个元素，后方的循环又会进行一轮迭代；以此类推，直到前方的循环完成一轮迭代为止。（可以观察显示结果的列表元素和排列顺序）

那么，如何能够得到正确的结果呢？

我们需要增加条件，当前方循环取出元素的平方等于后方循环取出元素的时候，再添加到列表。

示例代码：（正确）


```

1 number = [1, 2, 3, 4, 5, 6] # 整数列表
2 square = [1, 4, 9, 16, 25, 36] # 平方列表
3 lst = ['{0}^2={1}'.format(str(x), str(y)) for x in number for y in square if x <= 3 and
x * x == y]
4 print(lst) # 显示输出结果为: ['1^2=1', '2^2=4', '3^2=9']

```

以上代码等同于下方代码。

示例代码：

```

1 number = [1, 2, 3, 4, 5, 6]
2 square = [1, 4, 9, 16, 25, 36]
3 lst = [] # 此部分用列表推导式替代
4 for x in number: # 此部分用列表推导式替代
5     for y in square: # 此部分用列表推导式替代
6         if x <= 3 and x * x == y: # 此部分用列表推导式替代
7             lst.append('{0}^2={1}'.format(str(x), str(y))) # 此部分用列表推导式替代
8 print(lst)

```

通过上方代码，大家可以看出，创建同样的列表，使用列表推导式更加简洁。

lambda表达式

lambda表达式的官方定义：一个匿名内联函数，由一个表达式组成，在函数被调用时求值。

创建lambda函数的语法：lambda [参数]: 表达式

lambda表达式可用于函数的参数。

例如，我们从一个整数列表中筛选所有的偶数，可以使用filter函数。

这个函数的参数是function和iterable，也就是1个函数和1个可迭代对象。

示例代码：

```

1 def iseven(n): # 定义验证数字是否偶数的函数
2     if n % 2 == 0: # 判断参数是否为偶数
3         return True # 符合条件返回真值
4     else:
5         return False # 否则返回假值
6
7 lst = list(filter(iseven, number)) # 通过函数iseven对每个number的元素进行验证，验证为真的保留，并将最终结果转换为list。

```

```
8 print(lst) # 显示输出结果为: [2, 4, 6]
```

从上面的示例中，我们能够看到函数（iseven）可以作为另外一个函数（filter）的参数，并通过这个函数对其它参数进行处理。

filter函数会把number的每一个元素作为函数iseven的参数传入，进行计算，并将返回结果。

不过这样的代码，很明显看上去有些复杂。

我们可以使用lambda表达式这种匿名函数作为参数，起到同样的作用。

```
1 number = [1, 2, 3, 4, 5, 6] # 整数列表
2
3 lst = list(filter(lambda x: x % 2 == 0, number)) # 通过lambda表达式对每个number的元素进行
   验证，并将所有验证结果转换为list。
4 print(lst) # 显示输出结果为: [2, 4, 6]
```

上方代码中，每一个number的元素都会作为lambda表达式的参数（冒号前面的x）进行验证，如果符合条件（冒号后方的表达式），则会保留元素。

很显然，在这种情况下通过lambda表达式可以让代码非常简化。

正则表达式 (Regular Expression)

正则表达式的特点：

- 具有很强的灵活性、逻辑性和功能性；
- 可以快速通过非常简单的方式达到字符串的复杂控制；
- 对于初学者比较晦涩难懂。

“正则表达式”这个名称听上去很深奥，实际上它还有另外一个名称叫“规则表达式”。

它是对字符串操作的一种逻辑公式，用事先定义好的一些特定字符、及这些特定字符的组合，组成一个“规则字符串”，用来表达对字符串的一种过滤逻辑。

接下来，我们来了解一下上面提到的特定字符和特定字符组合。

在这里，我对它们做了分类。

一、匹配单个字符

- \d：匹配单个数字
- \D：匹配单个非数字
- \w：匹配单个字母或数字或下划线
- \W：匹配单个字母或数字或下划线以外的字符
- \s：匹配单个不可见字符，例如：\w\s-\s\d匹配a - 3。（匹配的是空格）
- \S：匹配单个可见字符
- .：匹配任意一个字符，例如\w.d匹配a~3。
- \.：匹配字符 “.”

二、匹配多个字符

- *：匹配任意数量字符

三、匹配范围

- [xyz]: 匹配xyz中任意一个字符
- [^xyz]: 匹配非xyz的字符
- [x|y]: 匹配其中任意一个字符
- (xxx|yyy): 匹配其中任意一个字符串
- [a-z]: 匹配a-z的范围
- [^a-z]: 匹配a-z以外的范围

四、匹配数量

- ?: 0-1个前方子表达式。例如\w?\d匹配a3和3。
- +: 数量大于0个前方子表达式。例如: /s+ 表示至少一个空格。
- {n}: 前方子表达式数量为n次。
- {n,}: 前方子表达式数量至少为n次。
- {n,m}: 前方子表达式数量至少为n次, 最多为m次。

五、匹配首尾字符

- ^: 匹配字符串起始单个字符, 后方紧随首个字符或表达式。
- \$: 匹配字符串末尾单个字符, 前方紧随末尾字符或表达式。
- \b: 匹配单词边界, 即字符串末尾字符串。例如: er\b匹配player。
- \B: 匹配非单词边界, 即字符串末尾之前的字符串。例如: er\B匹配error。

六、特殊匹配

- \: 转义字符
- \f: 匹配一个换页符
- \n: 匹配一个换行符
- \r: 匹配一个回车符
- \t: 匹配一个制表符
- \v: 匹配一个垂直制表符

七、零宽度断言

零宽度断言是一种零宽度的匹配, 它匹配到的内容不会保存到匹配结果中去, 最终匹配结果只是一个位置。

它的作用是给指定位置添加一个限定条件, 用来规定此位置之前或者之后的字符必须满足限定条件才能使正则中的子表达式匹配成功。

- (?!表达式): 向后匹配一个字符, 如果不是表达式对应的字符, 则匹配成功。
- (?=表达式): 向后匹配一个字符, 如果是表达式对应的字符, 则匹配成功。
- (?<=表达式): 向前匹配一个字符, 如果是表达式对应的字符, 则匹配成功。
- (?<!表达式): 向前匹配一个字符, 如果不是表达式对应的字符, 则匹配成功。

另外, 补充一点, 如果想获取到零宽度断言匹配成功的字符, 需要在断言后方填写表示单个字符的表达式, 例如: (?!,).表示字符不是 “,” 则获取。

在上方的分类内容中, 有一些内容具有等价关系。

等价是等同于的意思, 表示同样的功能, 用不同符号来书写。

等价字符:

- ?等价于匹配长度{0,1}
- *等价于匹配长度{0,}
- +等价于匹配长度{1,}

- \d等价于[0-9]
- \D等价于[^0-9]
- \w等价于[A-Za-z_0-9]
- \W等价于[^A-Za-z_0-9]

既然是等价关系，在编写正则表达式的时候就可以互相替代，使用等价字符能够让表达式变得更简洁。

接下来，我们归纳一下常用的一些运算符与表达式，记住这些内容我们就能够

常用运算符与表达式：

- ^：字符串开始
- \$：字符串结尾
- ()：域段/组（group），能够将匹配表达式的字符临时保存，并通过函数group(args)获取。
- []：包含,默认是一个字符长度
- [^]：不包含,默认是一个字符长度
- {n,m}：匹配长度
- .：任何单个字符
- |：或
- \：转义
- [A-Z]：26个大写字母
- [a-z]：26个小写字母
- [0-9]：0至9数字
- [A-Za-z0-9]：26个大写字母、26个小写字母和0至9数字
- ,：分割，例如：[A,H,T,W] 包含字母A或H或T或W；[a,h,t,w] 包含字母a或h或t或w；[0,3,6,8] 包含数字0或3或6或8。

温馨提示：以上内容只是对正则表达式的入门了解，如果想更加深入还需要找一些更专业的资料进行学习。

通过对上面内容的了解，我们就可以在代码中使用正则表达式了。

一般来说，通过给定一个正则表达式和一个字符串，我们可以达到以下目的：

- 判断给定的字符串是否符合正则表达式的过滤逻辑（即匹配）；
- 通过正则表达式，从给定的字符串中获取特定部分。

我们先来看一个通过正则表达式验证手机号码的例子。

示例代码：

```
1 import re # 引入正则表达式模块
2
3 def check_phone(phone): # 定义检查手机号码的函数
4     regular = '^1[3,4,5,7,8]\d{9}$' # 定义正则表达式
5     if re.match(regular, phone): # 调用正则匹配函数进行匹配
6         print('手机号码格式正确！') # 匹配提示
7     else:
8         print('手机号码格式错误！') # 不匹配提示
9
10 check_phone('15111111111') # 调用函数，显示输出结果为：手机号码格式正确！
```

```
11 check_phone('25111111111') # 调用函数，显示输出结果为：手机号码格式错误！
12 check_phone('12111111111') # 调用函数，显示输出结果为：手机号码格式错误！
13 check_phone('15111111111') # 调用函数，显示输出结果为：手机号码格式错误！
14 check_phone('151111111111') # 调用函数，显示输出结果为：手机号码格式错误！
15 check_phone('15a11111111') # 调用函数，显示输出结果为：手机号码格式错误！
```

手机号码的验证规则为：

- 首位字符为1；
- 第2位字符是“3、4、5、7、8”之一。
- 其余9位是数字。
- 号码长度为11位。

那么，在组织正则表达式的时候，我们可以这样来完成。

- 首位字符为1：表达式需要写入“^1”；
- 第2位字符是“3、4、5、7、8”之一：表达式需要写入“[3,4,5,7,8]”；
- 其余9位是数字：表达式需要写入“\d{9}”；
- 号码长度为11位：表达式需要写入“\$”（如果不写入“\$”超出11位依然能够匹配，而不是精确匹配）。
- 把上面每一部分连到一起就是完整的正则表达式。

另外，在代码中看到，如果使用正则表达式需要导入相应的模块“re”；

在对正则表达式和字符串进行匹配的时候，需要使用re模块中的match(pattern, string[, flags=0])函数。

如果字符串从起始位置（注意不是任意位置）有0个或多个字符与正则表达式匹配，则返回相应的匹配对象。如果字符串与正则表达式不匹配，则返回None值；注意None与零长度匹配不同，例如表达式“^\$”匹配“”，即零长度匹配。

参数pattern是正则表达式；参数string是字符串；参数flags是标志（忽略大小写或全局匹配等）。

关于参数flags：

- re.I/re.IGNORECASE：忽略大小写。
- re.L/re.LOCALE：让\w、\W、\b、\B、\s和\S依赖当前的区域语言设定。
- re.M/re.MULTILINE：影响“^”和“\$”的行为，指定后，“^”会增加匹配每行的开始（即换行符后的位置）；“\$”会增加匹配每行的结束（即换行符前的位置）。
- re.S/re.DOTALL：影响“.”的行为，指定后，可以匹配换行符。
- re.U/re.UNICODE：让\w、\W、\b、\B、\d、\D、\s和\S依赖Unicode库。
- re.X/re.VERBOSE：指定后，忽略所有空白字符（方括号内以及被反斜杠转义的除外）；而且，在每行中“#”号后的所有字符也将被忽略（既能够在正则表达式内部写注释）。

提示：如果了解函数match的返回结果，可以上方代码的check_phone()函数中添加下列语句：

```
1 print(re.match(regular, phone))
```

会显示类似“<_sre.SRE_Match object; span=(0, 11), match='15111111111'>”的结果。

其中，span是匹配的区间，match是匹配的全部字符或字符片段。

我们再来看一个通过正则表达式获取身份证号码中出生日期的例子。

示例代码：

```

1 import re # 引入正则表达式模块
2
3 def get_birthday(card_id): # 定义检查身份证号码的函数
4     regular = '^d{6}(d{2})(d{2}(d{4}))d{3}[d|X]$\'' # 定义正则表达式并指定域段
5     result=re.match(regular, card_id) #将匹配结果保存到变量
6     if result: #如果匹配结果不为None
7         return result.group(1,2,3) # 获取匹配结果中的指定域段并返回结果
8
9 print(get_birthday('11011219990109633X')) # 调用函数，显示输出结果为: ('1999', '01',
    '09')

```

注意：上方代码中的正则表达式并非真正身份证验证的正则表达式，此案例中仅用于取出出生日期。

大家能够看到，在正则表达式中，我们将一段表达式放入()中，如果有与这段表达式相匹配的字符串，就会被保存。然后，我们通过group(args)函数能够获取到保存的这些内容，参数args是保存内容的顺序位置，参数为0，可获取匹配的完整字符串。

最后，我们再来看两个示例关于预编译和贪婪模式。

示例代码：（预编译）

```

1 import re # 引入正则表达式模块
2
3 regular = r'd{3}' # 创建正则表达式
4 c = re.compile(regular) # 预编译正则表达式，
5 print(c.match('abc123',3)) # 显示输出结果为: <_sre.SRE_Match object; span=(3, 6),
    match='123'>

```

函数compile(pattern, flags)可以对正则表达式进行预编译并保存到变量中，通过变量调用match(string[, pos[, endpos]])函数即可进行匹配。

大家注意，预编译处理之后调用match函数时的参数是不同的，参数pos为匹配的起始位置，参数endpos为匹配的终止位置。

也就是说，当进行预编译处理后，和指定字符串的匹配可以不是从起始位置开始，而是可以指定匹配的区间。

示例代码：（贪婪模式）

```

1 import re # 引入正则表达式模块
2
3 print(re.match(r'^(\d+)(0*)$', '102300').groups()) # 结果为('102300', '') # 贪婪匹配
4 print(re.match(r'^(\d+?)(0*)$', '102300').groups()) # 结果为('1023', '00') # 非贪婪匹配

```

上方代码中，两个正则表达式有如下特点：

- 第一个正则表达式中`\d+`为贪婪匹配，会导致`0*`无法匹配`00`。
- 第二个正则表达式中`\d+?`为非贪婪匹配，`0*`不受影响可以正常匹配`00`。

贪婪模式会在整个表达式匹配成功的前提下，尽可能多的匹配，而非贪婪模式会在整个表达式匹配成功的前提下，尽可能少的匹配。

关于贪婪模式的量词（特定字符和字符组合）包括：“`{m,n}`”、“`{m,}`”、“`?`”、“`*`”和“`+`”。

这些贪婪模式的量词叫匹配优先量词，在这些匹配优先量词的后方后加上“`?`”，即变成非贪婪模式，叫做忽略优先量词。

补充

变量的赋值与交换

一、链式赋值（Chained Assignment）

打开Python控制台，在提示符后方我们进行下面的操作。

```
1 >>> a = b = c = 0
2 >>> a # 查看变量a的值
3 0
4 >>> b # 查看变量b的值
5 0
6 >>> c # 查看变量c的值
7 0
8
```

通过上方的示例大家能够看出，当多个变量具有相同的值，我们可以通过这种多个变量之间写等号的方式进行赋值。

二、序列解包（Sequence Unpacking）

如果我们给多个变量写入不同的值呢？

我们可以采用下面这种方式。

```
1 >>> a,b,c = 1,2,3 # 多变量赋值
2 >>> a # 查看变量a的值
3 1
4 >>> b # 查看变量b的值
5 2
6 >>> c # 查看变量c的值
7 3
```

大家能够看到，我们可以同时对多个变量进行赋值操作，只需要用逗号把它们隔开，并且等号右侧值的数量和变量数量相同即可。

这是什么原理呢？

大家再来继续尝试。

```
1 >>> 1,2,3
2 (1, 2, 3)
```

大家能够看到，当我们输入多个值并用逗号分隔，实际上是把这些值变成了一个序列（元组）。

Python会自动把这个序列分解，给到对应的每一个变量。

既然能够对序列进行解包，那么列表也一样。

我们再来做尝试。

```
1 >>> a,b,c = [1,2,3]
2 >>> a
3 1
4 >>> b
5 2
6 >>> c
7 3
```

实际上，字符串、列表、元组、集合、字典等都可以进行类似的操作。

那么，如果右侧值的数量与左侧不相等呢？

我们可以变量名称前面写入 “*” ，这样该变量的值会变成一个列表。

我们来做两个尝试。

```
1 >>> *a,b,c = 1,2
2 >>> a
3 []
4 >>> b
5 1
6 >>> c
7 2
```

```
1 >>> *a,b,c = 1,2,3,4
```



```
2 >>> a
3 [1,2]
4 >>> b
5 3
6 >>> c
7 4
```

当最左侧变量带有 “*” ，程序会先对右侧变量赋值，然后将剩余的值（或空值）写入带有 “*” 的变量。

“*” 可以写在任意一个变量的前方，但是只能有一个。

由此还能够看出， “*” 写在最右侧变量前方时，与写在最左侧变量前方时正好相反。

而 “*” 写在中间某个变量前方时，会先对应着为两侧变量赋值，然后将剩余的值（或空值）写入带有 “*” 的变量。

三、变量交换

我们先来做尝试。

```
1 >>> a,b = 1,2
2 >>> a,b = b,a
3 >>> a
4 2
5 >>> b
6 1
```

大家能够看到，我们可以通过上面这种方式对变量的值进行互换。

这里只是通过2个变量互换举例，实际上可以支持多个变量的互换。

四、增量赋值

增量赋值我们在之前的关于运算符教程中已经接触过。

```
1 >>> a = 1
2 >>> a += 1
3 >>> a
4 2
```

进制与数据类型

一、进制

所谓进制就是人们规定的一种进位方法。

2进制就是满2进1位；

8进制就是满8进1位；

10进制就是满10进1位；

16进制就是满16进1位。

不过，在Python中这些进制有不同的表达方式。

```
1 >>> 0x11 # 非10进制以0开头，字母x代表16进制
2 17
3 >>> 0o11 # 非10进制以0开头，字母o代表8进制
4 9
5 >>> 0b11 # 非10进制以0开头，字母b代表2进制
6 3
```

关于进制的内容，先了解这些就可以了。

二、数据类型

Python中有六种标准数据类型，分别为Number（数字）、String（字符串）、List（列表）、Tuple（元组）、Set（集合）和Dict（字典）。

这里我们主要补充一下Number（数字）和String（字符串）两种类型。

首先，数字类型包含：Int（整数）、Float（浮点数）、Bool（布尔）和Complex（复数）。

整型（Int）：即整数，可以通过int(x[,base=10])函数将非整数或整数组成的字符串以及字节转换成整数。

```
1 print(int()) # 显示输出结果为0
2 print(int(1.23)) # 显示输出结果为1
3 print(int('5')) # 显示输出结果为5
4 print(int('0xF',16)) # 显示输出结果为15
```

浮点型（Float）：即小数，float([x])函数将非整数或整数组成的字符串以及字节转换成整数。

```
1 print(float()) # 显示输出结果为0.0
2 print(float(1)) # 显示输出结果为1.0
3 print(float('5')) # 显示输出结果为5.0
4 print(float(0xf)) # 显示输出结果为15.0
```

布尔类型（Bool）：通过bool(x)函数，能够得到参数x的布尔值。

```
1 print(bool()) # 显示输出结果为False
2 print(bool(False)) # 显示输出结果为False
3 print(bool('')) # 显示输出结果为False
```

```
4 print(bool(0)) # 显示输出结果为False
5 print(bool(None)) # 显示输出结果为False
6 print(bool([])) # 显示输出结果为False
7 print(bool(())) # 显示输出结果为False
8 print(bool({})) # 显示输出结果为False
```

在Python中除了空值、`''`、`False`、`0`、`()`、`[]`、`{}`、`None`为`False`，其它均为`True`。

复数 (Complex)：能够通过`complex(x)`函数创建一个复数。

```
1 print(complex(1)) # 显示输出结果为: (1+0j)
2 print(complex('2+1j')) # 显示输出结果为: (2+1j)
3 print(complex(2, 5)) # 显示输出结果为: (2+5j)
```

接下来，是字符串 (String) 类型。

字符串 (String)：通过`str(object=)`函数能够将对象转换为字符串。

错误示例：

```
1 a = str(1)
2 b = a + 1
```

当我们把数字转换为字符串，进行加法运算时，会抛出异常。

`TypeError: unsupported operand type(s) for -: 'str' and 'int'`

类型错误：不被支持的操作数类型：字符串和整型

字符串相关函数

一、迭代对象合并为字符串

`join(iterable)`函数能够将可迭代对象的元素或多个字符串合并为一个字符串，参数`iterable`为可迭代对象。如果参数不是字符串会抛出类型错误的异常，包括字节对象。此函数通过元素间的分隔符调用。

```
1 lst1 = '1', '2', '3'
2 print('+'.join(lst1)) # 显示输出结果为: 1+2+3
3 lst2 = ['1', '2', '3']
4 print(''.join(lst2)) # 显示输出结果为: 123
```

二、字符串内单词首字母大写

这里需要使用string模块中的capwords(s[,sep])函数，参数s为需要转换的字符串，参数sep为分隔符，省略参数sep时，每个单词均转换为首字母大写单词；未省略参数sep时，除了第一个单词首字母转换为大写，分隔符后方的第一个字母均转换为大写。

示例代码：

```
1 import string
2
3 old_str = 'my name is xiaolou.'
4 new_str1 = string.capwords(old_str)
5 new_str2 = string.capwords(old_str, 's ')
6 print(new_str1) # 显示输出结果为: My Name Is Xiaolou.
7 print(new_str2) # 显示输出结果为: My name is Xiaolou.
```

三、多字符转换

这里需要使用字符串对象调用translate(table)函数，参数table为转换表；

很显然使用这个函数我们需要先创建转换表，转换表包含256个 ASCII字符。

我们并不需要手工去创建转换表，而是可以使用关键字str调用maketrans(x[,y[,z]])函数创建转换表。

只有1个参数时，必须输入一个字典，字典的键值可以是ASCII字符或对应的数字序号，转换时会用值对应的字符替换键对应的字符。

```
1 table=string.maketrans({'111':'*'})
2 print('photo'.translate(table)) # 显示输出结果为: ph*t*
```

输入2个参数时，前面的参数是被替换的字符，后面的参数是替换后的字符。

```
1 table=string.maketrans('o','*')
2 print('photo'.translate(table)) # 显示输出结果为: ph*t*
```

输入3个参数时，最后一个参数必须是字符串，包含在这个字符串中的每一个字符都被替换为None（空值）。

注意：函数会先完成第3个参数的替换，再进行前两个参数的替换。

```
1 table=string.maketrans('o','*', 'po')
2 print('photo'.translate(table)) # 显示输出结果为: ht
```

第四部分、字符串转义字符

\: 在字符串结尾时为续行符

\\: 斜杠
\' : 单引号
\" : 双引号
\a: 响铃
\b: 退格符
\n: 换行符
\t: 横向制表符
\v: 纵向制表符
\r: 回车符
\f: 换页符
\ooo: 八进制字符
\xhh: 十六进制字符
\000: 终止符
\other: 其它的字符以普通格式输出
\N{name}: Unicode数据库中的字符名
\uhhhh: 值为16位的十六进制字符
\Uhhhhhhh: 值为32位的十六进制字符

第五部分、字符串格式化

之前我们接触过简单的字符串格式化方法，通过转换说明符%标记转换内容的位置，然后通过操作符%，将后方的内容转换到标记的位置。

实际上，转换说明符的组成包括：%[转换标志: +,-," ,0][最小字段宽度][.精度值]转换类型

%: 转换说明符的开始的标记

转换标志（可选）：-表示左对齐；+表示转换后的内容前方加上正负号；' '（空格）表示正数前方保留一个空格；0表示如果转换位置位数不够时，用0填充。

最小字段宽度（可选）：转换后字段的最小宽度，如果不足自动用空格补齐；如果最小字段宽度值为“*”，则从元组中读取。

.精度值：如果转换内容为数字，精度值表示保留的小数位数；如果转换内容为字符串，精度值表示最大宽度。

转换类型：

- i/d: 带符号的十进制整数
- o: 不带符号的八进制
- u: 不带符号的十进制
- x/X: 不带符号的十六进制（x为小写，X为大写）
- e/E: 科学计数法的浮点数（e为小写，E为大写）
- f/F: 十进制浮点数
- g/G: 指数大于等于-4但小于精度值时，等同于f/F；否则，等同于e/E。
- C: 单字符（整数或单个字符）
- r: 字符串（非字符串时自动使用repr函数转换）
- s: 字符串（非字符串时自动使用str函数转换）
- %: 取消%转义（输出一个“%”字符）

接下来，我们看一些示例代码，加深理解。

示例代码：（最小字段宽度）

```

1 print('%2s' % 123456, '/', sep='') # 最小字段宽度（超过长度）
2 print('%8s' % 123456, '/', sep='') # 最小字段宽度（长度不足）
3 print('%*s' % (8, 123456), '/', sep='') # 最小字段宽度（*）
4
5 # 显示输出结果为：123456/
6 # 显示输出结果为：   123456/
7 # 显示输出结果为：   123456/

```

示例代码：（.精度值）

```

1 print('%.2f' % 3.14159) # 精度值（浮点数）
2 print('%2s' % '3.14159') # 精度值（字符串）
3
4 # 显示输出结果为：3.14
5 # 显示输出结果为：3.

```

示例代码：（转换标志）

```

1 print('%8.2f' % 3.14159, '/', sep='') # 未左对齐
2 print('%-8.2f' % 3.14159, '/', sep='') # 左对齐
3
4 # 显示输出结果为：      3.14/
5 # 显示输出结果为：3.14   /
6
7 print('%+d是一个正数' % 666) # 加号
8 print('%+d是一个负数' % -666) # 加号
9 # 显示输出结果为：+666是一个正数
10 # 显示输出结果为：-666是一个负数
11
12
13 print('% d是一个正数' % 666) # 空格
14 print('% d是一个负数' % -666) # 空格
15
16 # 显示输出结果为： 666是一个正数
17 # 显示输出结果为：-666是一个正数
18

```

```
19 print('%08d是一个8位数' % 666) # 填充0
20 # 显示输出结果为: 00000666是一个8位数
```

第六部分、执行字符串语句

如果想在Python中动态的执行一些代码，我们可以使用内置函数exec()和eval()。

示例代码：

```
1 exec('print(\'语句被执行了！\')')
```

上方代码会执行print语句。

但是，这样执行语句会有不安全因素。

示例代码：（异常示例）

```
1 exec('len=1')
2 len('abc')
```

上方代码，因为执行了语句len=1，导致内置函数len()失效，从而程序抛出异常。

TypeError: 'int' object is not callable

类型错误：整数对象不能调用

显然，这样执行字符串中的语句非常不安全。

我们可以把执行字符串代码的语句放在字典中，让字符串代码具有独立的作用域（命名空间）。

示例代码：

```
1 scope = {}
2 exec('len=1', scope)
3 print(scope.keys()) # 显示输出结果为: dict_keys(['__builtins__', 'len'])
4 print(scope['len']) # 显示输出结果为: 1
5 print(len('abc')) # 显示输出结果为: 3
```

当然，也可以执行其他语句，比如一个计算字符串长度的语句。

示例代码：

```
1 scope = {}
2 exec('code=len(\'小楼喜欢苍老师!\')', scope)
3 print(scope.keys()) # 显示输出结果为: dict_keys(['__builtins__', 'code'])
4 print(scope['code']) # 显示输出结果为: 8
```

exec()函数可以执行语句，而eval()函数可以计算表达式。

示例代码：

```
1 scope = {'x':3,'y':6}
2 print(eval('x*y', scope))# 显示输出结果为：18
```

第七部分、函数说明

如果想为函数添加说明文档，除了使用“#”进行注释，还可以在def语句后面直接写入说明字符串。

```
1 def get_area(width, height):
2     '这是一个计算面积的函数。'
3     return width * height
```

当一个函数添加了文档，我们可以通过调用函数的__doc__属性查看文档。

```
1 print(get_area.__doc__) # 显示输出结果为：这是一个计算面积的函数。
```

而且，我们还可以使用内置函数help()进行查看。

```
1 help(get_area)
```

上方这句代码会显示输出以下结果：

Help on function get_area in module __main__:

get_area(width, height)

这是一个计算面积的函数。

Python中的类 (Class)

一、类相关的概念

我们先来接触一些概念。

1、类的概念。

类是**根据事物本身的性质或特点（简称特性）而分成的门类。**（来自百度百科）

例如，我们通常说人类、鸟类、家具类、电器类、文具类.....

2、特征的概念。

特征是一个**客体或一组客体特性的抽象结果。**（来自百度百科）

通俗一些来说，特征是一个或多个事物所具有的相同或相像特性的总和。

3、抽象的概念

抽象简单来说就是抽取相像的部分，所以对特性的抽象，就是把所有相像或相同的特性进行抽取。

4、特性的概念

参考类的概念，特性（Attribute）就是事物本身的性质或特点。

例如，人类有名字，鸟类会飞，家具可以用来存储，电器需要电源，文具用来学习等等，都是事物的性质或特点。注意，特性不仅仅包含属性（例如名称、颜色等内在的状态）还包含了方法（例如飞行，说话等外在的行为），理解这个很重要。

5、对象的概念

对象（Object）就是客体，客观存在的具体的事物。

所以说“万物皆对象”。（这句话在Python中尤其正确）

那么，类的实例（Instance）显然就是对象。

例如，人类的一个实例就是一个具体的人，这个具体的人就是对象。

通过类创建实例对象的操作，我们称之为实例化。

实例对象包含了类的所有特性。

通过上面这些概念的理解，我们回过头来重新理解类的概念。

类是某一种对象的总称，它包含从某一种对象中抽取出来的相同或相像的全部特性。

相信现在上面这句话理解起来不会再困难，而对理解接下来的内容也很有帮助。

二、定义类

定义类使用class关键字，类的名称一般会使用首字母大写的单词。

格式：class 类名称(超类名称):

超类是指当前的类所继承的类，如果没有继承超类括号部分可以省略。（继承的概念在后面的内容中有详细的介绍）

示例代码：

```
1 class Human: # 定义类（人类）
2     def set_name(self, user_name): # 类的函数
3         self.name = user_name # 修改类的变量
4
5     def get_name(self): # 类的函数
6         return self.name # 返回类的变量
7
8     def say_hello(self): # 类的函数
9         print('嗨！大家好，我是%s！' % self.name) # 使用类的特性
```

上方代码就是类的创建，注意里面的变量和函数都是类的特性。例如：name是姓名，say_hello是打招呼，这些都是人类通常情况下共有的特性。

另外，值得注意的是，类的定义实际上就是执行代码块，而不仅仅是在类的里面定义方法。

示例代码：

```
1 class Class: # 创建类
2     print('你好，我是类中的语句，我能直接被执行！')
```

运行一下代码，我们能够看到类中的print语句直接被执行了。

而如果仅仅是一个函数的话，即便里面包含同样的语句，运行代码时，print语句是不会被执行的。

提示：回想一下我们以前写过的自定义函数，都需要编写语句调用才能够执行。

三、封装 (Encapsulation)

将抽象得到的全部特性结合为一个整体（也可以看做将数据和操作捆绑到一起），形成类即为封装。

封装后，所有特性都是类的成员。

除此之外，封装还是达到接口与实现分离的过程。

封装，即隐藏对象的特性和实现细节（实现），仅对外提供可调用的特性（接口），将访问控制在只能够对被允许的对特性进行读与修改的级别。

从这个方面来看，封装就像组装的电视机，使用它的人无需知道电视机由哪些元器件和运行程序（特性）组成，也不需要知道它的具体工作过程（方法的实现细节），只需要知道能够进行的开、关以及换台（允许调用的方法）等操作就可以了。

那么，封装有什么样的好处呢？

我们通过一些示例来了解。

示例代码：（未封装）

```
1 global_name = '' # 创建全局变量并赋值
2
3 class Human(object): # 创建类（人类）
4     def set_name(self, name): # 定义方法修改全局变量的值
5         global global_name # 声明引用全局变量
6         global_name = name # 全局变量重新绑定值
7
8     def get_name(self): # 定义方法获取全局变量值
9         return global_name
10
11     def say_hello(self): # 类的方法
12         print('嗨！大家好，我是%s!' % global_name) # 使用类的特性
```

上面的代码中，我们把特性name（类里面的变量）改为为全局变量。

接下来，我们通过Human类（人类）实例化出一个实例对象person（人）。

然后，为这个实例对象（人）设置姓名。

并且，调用实例对象中（人）的say_hello()方法，让这个实例对象（人）和我们打个招呼！

示例代码：（类的实例化和实例方法的调用）

```
1 person1 = Human() # 类的实例化
2 person1.set_name('小楼')
3 person1.say_hello() # 调用实例对象的方法，显示输出结果为：嗨！大家好，我是小楼！
```

看上去没有问题。

接下来，我们通过类实例化出第2个实例对象（人），然后试试给新的实例对象（人）也取个名字，并且让新的实例对象（人）也打个招呼。

示例代码：（错误示例）

```
1 person1 = Human() # 类的实例化
2 person1.set_name('小楼')
3
4 person2 = Human() # 类的实例化
5 person2.set_name('樱井')
6
7 person1.say_hello() # 显示输出结果为：嗨！大家好，我是樱井！
8 person2.say_hello() # 显示输出结果为：嗨！大家好，我是樱井！
```

现在大家能够看到问题了。

虽然是两个实例对象，并且调用每个实例对象自己的方法取名字，但是他们的set_name()方法都修改同一个全局变量，这样就会互相干扰。

而将全局变量变为封装到类里面的特性，就不会有这样的问題。

封装后的代码，就是上方我们创建的第一个类的代码。

我们同样对这个类进行两次实例化，并且为两个实例对象设置姓名，再调用每个实例对象打招呼的方法。

示例代码：（封装后）

```
1 person1 = Human() # 类的实例化
2 person1.set_name('小楼')
3 person2 = Human() # 类的实例化
4 person2.set_name('樱井')
5 person1.say_hello() # 显示输出结果为：嗨！大家好，我是小楼！
6 person2.say_hello() # 显示输出结果为：嗨！大家好，我是樱井！
```

这一次就没有任何问题了。

最后，我们再了解一块内容，就是函数与方法的区别。

可能看到这里有些人会有点儿懵：函数和方法不是同一个东西吗？

在之前的教程中，为了便于理解函数，我们没有严格区分函数与方法的概念。

实际上，函数和方法是有区别的。

这里借用官方文档的示例与说明。

示例代码：

```
1 class MyClass:
2     """A simple example class (一个简单的示例类) """
3
4     i = 12345
5
6     def f(self):
7         return 'hello world'
8
9 x = MyClass()
```

说明原文：

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, `x.f` is a valid method reference, since `MyClass.f` is a function, but `x.i` is not, since `MyClass.i` is not. But `x.f` is not the same thing as `MyClass.f` — it is a method object, not a function object.

说明大意：

实例对象有效的方法名取决于类。根据定义，类的所有特性中，函数对象都是在定义其实例中相应的方法。所以，在我们的例子中，`x.f`将是一个有效的方法调用，因为`MyClass.f`是一个函数。但是`x.i`不是有效的方法调用，因为`MyClass.i`不是一个函数。但是，`x.f`和`MyClass.f`不同，`x.f`是一个方法对象，而不是一个函数对象。

我们通过`print`函数来查看一下，是不是这样？

示例代码：

```
1 print(x.f) # 显示输出结果为: <bound method MyClass.f of <__main__.MyClass object at
0x000000000265D630>>
2 print(MyClass.f) # 显示输出结果为: <function MyClass.f at 0x000000000265FA60>
```

大家能够看到`x.f`是`MyClass`实例对象的一个“bound method”（绑定方法），由`MyClass.f`函数定义。而`MyClass.f`是一个“function”（函数）。

那么，我们可以这样理解：**绑定到实例对象特性上的函数就是方法。**

另外，还有一个区别在于类的函数与实例方法的第一个参数是“self”。

参数“self”绑定到了方法所属的实例，通过参数“self”我们能够访问实例对象的特性。

也正是因为如此，通过实例调用方法时，这个参数不需要提供。

而通过类调用函数的时候，参数“self”需要提供，如果访问类的特性填入类名称，如果访问实例特性填入实例名称。

例如，我们借用封装好的`Human`类，继续写入一些代码。

示例代码：（访问类的特性）

```
1 Human.set_name(Human, '明步') # 设置类的特性
2 Human.say_hello(Human)      # 显示输出结果为：嗨！大家好，我是明步！
```

如果调用时，我们不输入Human这个参数，则会抛出异常。

例如：set_name()未输入参数Human，则会抛出异常。

TypeError: set_name() missing 1 required positional argument: 'user_name'

类型错误：set_name() 函数丢失了1个必须的位置参数：'user_name'

也就是说，程序把输入的参数字符串当成了传入的self参数，然后异常提示还需要传入第2个user_name参数。

四、私有化 (Private)

虽然，封装是为了避免让使用者看到多余的信息或是对类的特性直接进行控制，但是，在Python中这好像没有什么用。

我们看下面这段代码：

示例代码：

```
1 person = Human() # 类的实例化
2 person.name = '小楼' # 修改类的特性
3 print(person.get_name()) # 显示输出结果为：小楼
```

很显然，我们还是能够对对象的特性直接进行修改。

下面有一种解决方案，就是给特性名称前面加上双下划线“__”（包含方法名称），但是也只是“自欺欺人”。

示例代码：

```
1 class Human: # 创建类（人类）
2     def set_name(self, user_name): # 类的方法
3         self.__name = user_name # 修改类的特性
4
5     def get_name(self): # 类的方法
6         return self.__name # 返回类的特性
7
8     def say_hello(self): # 类的方法
9         print('嗨！大家好，我的是%s！' % self.__name) # 使用类的特性
10
11
12 person = Human() # 类的实例化
13 person.__name = '小楼' # 抛出异常
```

运行代码，这时会抛出异常。

AttributeError: 'Human' object has no attribute '_Human__name'

属性异常：' Human' 对象不包含'_Human__name' 特性。

真的不包含吗？然并卵！

实际上，这个异常提示暴露了特性信息，名称前方带有双下划线的特性（包含方法）可以通过“单下划线+类名称+双下划线+特性名称”进行控制，就像异常中的“_Human__name”。

示例代码：

```
1 person = Human() # 类的实例化
2 person._Human__name = '小楼' # 修改对象的特性
3 print(person.get_name()) # 显示输出结果为：小楼
```

运行代码，我们能够看到类的特性依然被修改了。

所以，不想他人直接访问对象的特性（包含方法）是不可能的，只是带有双下划线的名称带有很明显的警示作用，就像提示前面的红色叹号一样。

五、类的命名空间

模块有命名空间；变量有命名空间；函数有命名空间。

当然，类也会有命名空间。

所有在类中定义的代码都会在独立的命名空间中执行，也就是类的命名空间（class namespace）。

这个命名空间，可以由类的所有成员（类的实例）进行访问。

接下来，我们再通过示例看一下成员（类的实例）对类的访问。

示例代码：

```
1 class Register: # 创建注册类
2     reg_count=0 # 定义计数变量
3     def register(self): # 定义计数方法
4         Register.reg_count+=1 # 累加注册数写入计数变量
5         print('新增1名注册用户，当前注册数量为%d人' % Register.reg_count )
6
7 reg1=Register() # 类的实例化
8 reg1.register() # 调用计数方法，显示输出结果为：新增1名注册用户，当前注册数量为1人。
9
10 reg2=Register() # 类的实例化
11 reg2.register() # 调用计数方法，显示输出结果为：新增1名注册用户，当前注册数量为2人。
```

在上方代码中，我们在类的里面定义了一个计数变量，这个变量供全体类的成员（类的实例）访问。

每一个实例都可以调用自身的方法对类的计数变量进行访问。

注意，访问类的变量是通过“类名.变量名”进行访问，而不是“self.变量名”进行访问。

类的实例包含类的所有内容，所以，“self.变量”名访问的是实例对象自身的变量，而不是类的变量。

就如同全国人口统计系统（类）会记录人员数量（变量），所以每一个统计员（实例）也会记录人员数量（变量）。如果每个统计员（实例）进行人口统计时，只是自己（self）做了人员数量（变量）的记录（访问），而没有记录（访问）到全国人口统计系统（类）的人员数量（变量），那么，全国人口统计（类）的人员数量（变量）则不会发生改变。

我们在上方代码的基础上，再加入一些内容。

示例代码：

```
1 print(reg1.reg_count) # 示输出结果为: 2
2 print(reg2.reg_count) # 示输出结果为: 2
```

大家能够看到，实例可以直接访问类的变量。

那么，如果在实例中尝试重新绑定类的特性呢？也就是，通过实例对类的变量进行修改。

示例代码：

```
1 reg1.reg_count = 0 # 修改变量值
2 print(reg1.reg_count) # 示输出结果为: 0
3 print(reg2.reg_count) # 示输出结果为: 2
```

上方代码中，我们试图通过实例对类的变量进行修改。

但是，很明显修改后并没有让类的变量发生变化，而是实例自身的变量产生了变化。

这很像之前我们学过的全局变量和同名的局部变量，局部变量会屏蔽全局变量。

为什么会这样？

理解的关键，在于对“绑定”这个词的理解。

绑定实际上是建立名称与值的关系。

我们需要把变量的名称和值分开来说，则很容易理解什么叫绑定。

就拿中国人（类）来说，中国人有一个头发颜色的特性（变量），默认是黑色（变量值）。

就像每一个出生的中国人（实例）头发颜色的特性（变量）都是黑色（变量值）一样，默认情况下，每一个实例变量的值绑定的是类变量的值。

那么，像刚才我们通过一个实例试图对类变量进行修改，实际上是将实例的变量绑定到了一个新值，其他实例变量的值并不会受到影响，绑定的仍然是类变量的值。

这就像某一个中国人（实例）将自己头发颜色（变量）染（重绑定）成了黄色（变量值），其它中国人（实例）的头发颜色（变量）仍然是黑色（类变量值），并不会一起改变。

绑定示意图：

六、继承 (Inheritance)

还是以人类举例，还可以细分为白种人类、黄种人类、黑种人类和棕色人类。

人类是所有人的集合，这些细分的类是人类的子集。

所以，当一个对象所属的类是另一个对象所属的类的子集时，它就是后者的子类（subclass），而后者是前者的超类（superclass），或叫做父类、基类。

在面向对象的程序设计中，子类的关系是隐藏的。

简单来说，类的定义取决于它的所有支持的方法。

类的对象都会包含这些方法。

那么，子类也会包含超类中的这些方法（但是看不到）。

定义子类要做的只是定义更多特性的过程或者重写（override）超类中已有的特性。

我们来看一段代码。

示例代码：

```
1 class Shielding: # 创建屏蔽类
2     def __init__(self):
3         self.words = [] # 屏蔽内容列表
4         self.symbol = '' # 屏蔽后显示的符号
5
6     def change(self, sentence): # 定义修改屏蔽内容的方法
7         string = sentence
8         for word in self.words: # 遍历屏蔽词列表
9             if word in sentence: # 判断句子中是否包含屏蔽词
10                 string = string.replace(word, self.symbol * len(word)) # 将句子中的屏蔽
    内容替换为符号
11         return string # 返回修改结果
12
13
14 class ShieldingWords(Shielding): # 创建屏蔽词类
15     def __init__(self): # 重写超类的init方法
16         self.words = ['银行', '账号', '密码'] # 设置屏蔽词列表
17         self.symbol = '*'
18
19
20 class ShieldingSymbols(Shielding): # 创建屏蔽字符类
21     def __init__(self): # 重写超类的init方法
22         self.words = '@' # 设置屏蔽字符列表
23         self.symbol = '#'
24     def message(self):
25         print('禁止在邮箱地址中使用"%s"，已使用"%s"代替！' % (self.words, self.symbol))
26
27 s = Shielding()
28 print(s.change('你银行的账号和密码是什么？')) # 显示输出结果为：你银行的账号和密码是什么？
29
```



```

30 w = ShieldingWords()
31 print(w.change('你银行的账号和密码是什么? ')) # 显示输出结果为: 你**的**和**是什么?
32
33 c = ShieldingSymbols()
34 print(c.change('我的邮箱是4907442@qq.com')) # 显示输出结果为: 我的邮箱是4907442#qq.com
35 c.message() # # 显示输出结果为: 禁止在邮箱地址中使用"@", 已使用"#"代替!

```

代码有些长，不过通过注释应该非常容易读懂。

这里我们创建了1个超类（Shielding）和2个子类（ShieldingWords和ShieldingSymbols）。

超类和子类怎么区分呢？

很简单，在子类名称后方的括号中所写的名称就是超类的名称。

接下来，再来观察。

在超类Shielding中，我们定义了2个方法。

这里值得一提的是init()这个方法，它是无需调用自动执行的方法，所以在下方代码中我们看不到这个方法的调用。

另外一个方法是替换原有词语为符号的方法，这个方法对类的两个特性words和symbol进行了操作。

大家能够看到，在超类Shielding中的这两个方法，实际上是无法完成词语屏蔽的，通过实例化的对象s，调用change()方法，传入的字符串没有任何变化的被输出了。

而两个子类都对init()方法进行了重写，修改了特性的值。

然后，虽然两个子类都没有定义change()这个方法，但是能够正常调用，并输出了修改后的内容。

并且，在子类ShieldingSymbols中我们额外增加了一个message()方法。

通过这段示例代码，我们做个总结：

- 子类包含父类的所有特性；
- 子类中能够对超类的特性进行重写；
- 子类能够添加父类中不存在的特性。

七、查验继承关系

假如我们使用了别人写的代码，怎么知道代码中的某一个类是不是另外一个类的子类或超类呢？

我们可以使用issubclass(class, classinfo)函数。

这个函数能够检查参数cls是否符合参数classinfo类型，或者说前者是否后者的子类。

例如检查上面代码中类的关系。

示例代码：

```

1 print(issubclass(ShieldingWords,Shielding)) # 显示输出结果为: True
2 print(issubclass(Shielding,ShieldingWords)) # 显示输出结果为: False

```

上面的代码是检查两个类的关系。

那么，如果只有一个类，如何知道谁是它的超类？

我们可以使用类的__bases__特性来查看。

示例代码：

```
1 print(ShieldingWords.__bases__) # 显示输出结果为: (<class '__main__.Shielding'>,)
```

还有，如果想知道某个对象是哪个类的实例，可以调用__class__特性。

示例代码：

```
1 s = ShieldingWords()
2 print(s.__class__) # 显示输出结果为: <class '__main__.ShieldingWords'>
```

除此之外，我们还可以使用isinstance()函数来检查一个对象是不是某一个类的实例。

但是，不推荐使用这种方法，因为不符合多态的原则（多态在后边会学到）。

示例代码：

```
1 s = ShieldingWords()
2 print(isinstance(s,ShieldingWords))# 显示输出结果为: True
3 print(isinstance(s,Shielding))# 显示输出结果为: True
4 print(isinstance(s,ShieldingSymbols))# 显示输出结果为: False
```

大家能够看到对象s是子类ShieldingWords的实例，同时也是超类Shielding的实例，但不是子类ShieldingSymbols的实例。

八、多继承 (Multiple Inheritance)

注意：不要和多重继承搞混，多重继承是指C继承B，B继承A这样的继承形式。

示例代码：（多重继承）

```
1 class A:
2     pass
3 class B(A):
4     pass
5 class C(B):
6     pass
```

多继承是指类能够继承自多个超类。

所以，在上一篇教程我们在使用__bases__特性时，能够看到bases是一个复数。

多继承的使用很简单，只需要在类名称后面的括号中写入超类的名称并用逗号分隔即可。

示例代码：（多继承）

```
1 class A:
2     pass
3 class B:
4     pass
5 class C(A,B):
6     pass
```

不过要注意，当多个超类都具有相同的特性时，只会继承第一个（最左侧）超类中的特性。

在现实生活中，有很多这种情况，例如小学生既是学生又是儿童。

就像下面这段代码。

示例代码：

```
1 class Children: # 创建儿童类
2     age = 10
3
4     def activitie(self): # 定义活动方法
5         print('我周末去儿童游乐园玩! ')
6
7 class Student: # 创建学生类
8     grade = 3
9
10    def activitie(self): # 定义活动方法
11        print('我每天放学在家写作业! ')
12
13 class Pupil(Children, Student): # 创建小学生类
14     def __init__(self):
15         print('我今年%d岁，已经上%d年级啦!' % (self.age, self.grade))
16
17 p = Pupil() # 显示输出结果为：我今年10岁，已经上3年级啦！
18 p.activitie() # 显示输出结果为：我周末去儿童游乐园玩！
```

九、检查对象的特性

如果想知道一个对象是否具有某个特性，可以使用hasattr(o,name)函数，当包含指定特性的名称时，返回值为True；否则，为False。

借用上面的学生类，我们进行特性的检查。

```
1 c = Children()
```

```

2 print(hasattr(c, 'age')) # 检查特性age, 显示输出结果为: True
3 print(hasattr(c, 'grade')) # 检查特性grade, 显示输出结果为: False
4 print(hasattr(c, 'activitie')) # 检查方法activitie, 显示输出结果为: True
5 print(hasattr(c, 'study')) # 检查方法study, 显示输出结果为: False

```

另外, 我们还可以检查一个对象的特性是否能被调用。

我们可以使用`getattr(o,name,default)`获取对象的特性, 然后通过`callable(object)`检查。

```

1 print(callable(getattr(c, 'activitie', None))) # 显示输出结果为: True
2 print(callable(getattr(c, 'study', None))) # 显示输出结果为: False

```

上方代码中, 我们为`getattr()`函数指定了默认值`None`, 这样当找不到特性时, 返回值为`False`。

如果不设置默认值`None`, 则会抛出异常。

`AttributeError: 'Children' object has no attribute 'study'`

特性错误: 'Children' 对象不包含 'study' 特性。

十、多态 (Polymorphic)

多态的字面意思是多种形式。

在Python中是指多个不同类的对象, 都具有一些共同的特性, 这些对象中的任何一个对象, 都可以调用这些共同的特性。但是, 因为是不同类的对象, 所以, 在调用同一个特性时, 会表现出不同的行为。

这里提到的对象允许外部访问的共同特性, 也就是在前面我们提到过接口。

当我们处理多态对象时, 只需要关心它所提供的接口。

例如, `count()`方法就是多态特性。

示例代码:

```

1 obj = '小楼棒, 小楼帅, 小楼好厉害!' # 变量引用字符串对象
2 print(obj.count('小楼')) # 显示输出结果为: 3
3 obj = ('小楼', '樱井', '小楼', '明步') # 变量引用元组对象
4 print(obj.count('小楼')) # 显示输出结果为: 2

```

在上方代码中, “obj” 就是多态对象, 它可以是不同的对象。

我们不管 “obj” 是什么对象 (字符串或元组或其它), 只需要关心它的接口中有没有`count()`这个方法。

如果有`count()`这个方法, 就能够对对象进行处理。

但是, 虽然调用的方法都是同一个名称, 因为处理的是不同的对象, 实际上具体的处理行为是不一样的。

接下来, 我引用一个其它编程语言中的示例, 来帮助大家了解多态。

动物园的饲养员 (Feeder), 能够喂养狮子 (Lion), 老虎 (Tiger) 和狗熊 (Bear)。

饲养员要对这些动物进行喂食的操作。

假设我们不做多态处理。

示例代码：（非多态）

```
1 # 定义动物
2 class Lion: # 定义狮子类
3     def lion_eat(self): # 定义进食函数
4         print('狮子在吃东西! ')
5
6 class Tiger: # 定义老虎类
7     def tiger_eat(self): # 定义进食函数
8         print('老虎在吃东西! ')
9
10 class Bear: # 定义狗熊类
11     def bear_eat(self): # 定义进食函数
12         print('狗熊在吃东西! ')
13 # 定义饲养员
14 class Feeder:
15     def feed_lion(self, lion): # 定义喂养狮子的函数
16         lion.lion_eat()
17
18     def feed_tiger(self, tiger): # 定义喂养老虎的函数
19         tiger.tiger_eat()
20
21     def feed_bear(self, bear): # 定义喂养猴子的函数
22         bear.bear_eat()
23
24 # 喂养过程
25 feeder = Feeder()
26 lion = Lion()
27 feeder.feed_lion(lion) # 显示输出结果为：狮子在吃东西！
28 tiger = Tiger()
29 feeder.feed_tiger(tiger) # 显示输出结果为：老虎在吃东西！
30 bear = Bear()
31 feeder.feed_bear(bear) # 显示输出结果为：狗熊在吃东西！
```

上方的代码虽然运行正常，但是如果给饲养员增加工作量，让他再多喂一只猴子。

我们就需要继续增加一些代码。

示例代码：（增加的代码）

```

1  #定义动物
2  class Monkey:
3      def monkey_eat(self):
4          print('猴子在吃东西! ')
5
6  #定义饲养员
7  class Feeder:
8      def feed_monkey(self, monkey):
9          monke.monkey_eat()
10
11 # 喂养过程
12 monkey = Monkey()
13 feeder.feed_monkey(monkey)

```

很显然，每增加一个喂养一个动物，我们都要增加这么多代码。

那么，通过多态处理呢？

示例代码：（多态）

```

1  class Lion: # 定义狮子类
2      def eat(self): # 定义进食函数
3          print('狮子在吃东西! ')
4
5  class Tiger: # 定义老虎类
6      def eat(self): # 定义进食函数
7          print('老虎在吃东西! ')
8
9  class Bear: # 定义狗熊类
10     def eat(self): # 定义进食函数
11         print('狗熊在吃东西! ')
12
13 class Feeder: # 定义饲养员类
14     def feed_animal(self, animal): # 定义喂食方法
15         animal.eat()
16
17 # 喂养过程
18 feeder = Feeder()
19 animal = Lion()

```

```
20 feeder.feed_animal(animal) # 显示输出结果为：我是狮子，在吃东西！
21 animal = Tiger()
22 feeder.feed_animal(animal) # 显示输出结果为：我是狮子，在吃东西！
23 animal = Bear()
24 feeder.feed_animal(animal) # 显示输出结果为：我是狗熊，在吃东西！
```

在上方代码中，每一个动物的进食方法都采用了相同的名称，喂食方法只保留了一个。

并且，喂养过程中调用的喂食方法都是相同的。（因为喂食方法就一个）

这样，多个类中都有一个相同的特性（eat函数），实例对象也就具有了这个相同的特性（eat方法），所以，不管调用的时候是哪一种实例对象（变量animal），都可以调用这个特性（eat方法），但是因为调用这个特性的实例对象（动物）不同，所以产生的行为（eat方法的处理过程）是不同的。

这就是多态。

那么，对比之前未做多态处理的代码，有什么好处呢？

当同样需要增加喂养猴子时，通过多态处理的代码，只需要增加新的动物种类，以及喂养过程中对动物的实例化和调用喂养方法。

示例代码：（增加部分）

```
1 #定义动物
2 class Monkey:
3     def eat(self):
4         print('猴子在吃东西！')
5
6 # 喂养过程
7 animal = Monkey()
8 feeder.feed_animal(animal)
```

从这个示例我们也能够看出，喂食方法无需再重复定义，多喂养一种动物只需要定义一个新的类。

这也就说明了，多态能够提高代码的复用性，让代码更加精简易读，并且降低了代码的耦合度（紧密配合与相互影响的程度），提升了代码的扩展性。

另外，值得注意的是，Python中多态的概念不同于其它编程语言，在大多编程语言中，如果需要进行多态处理，会要求具有相同特性的类（例如上方代码中的狮子类、老虎类等）必须继承自同一个超类（需要定义Animal类），这样在调用特性时是通过超类寻找相应的子类。

而Python中无需这么做，就像上方的代码，我们并没有定义一个动物的超类，让动物的子类去继承这个超类。

所以，严格来说Python中没有多态这个概念，因为Python自身就是多态的语言。

面向对象程序设计 (Object Oriented Programming)

在之前几篇教程中，我们了解了类的定义和使用方法。

在类的使用中，包含了多态、封装和继承这三个显著的特点。

这三个特点，其实就是面向对象程序设计的三大特性。

它们让程序设计变得灵活，增加了程序的安全性（封装），具有良好的重用性（继承），易于扩展和维护（多态）。如果大家对面向过程的程序设计做一些了解，会对这三个特性有更深的体会。

面向过程的程序设计，是通过一个main()函数定义程序要完成的任务，由一系列的子函数组成。

这些子函数，还会细化成更多的子函数，以满足程序的各种需求。

重复这样的过程，就是面向过程的程序设计。

所以，面向过程的程序设计把整个程序视为一系列的命令集合，即一组函数的顺序执行。函数细分为子函数，是为了降低程序的复杂度。

这就好像我们书写论文，main()函数就是提纲，子函数就是论题，子函数细分的子函数就是论点，数据是论据，而函数的计算过程就论证的过程。

论文的这种结构，逐层划分，逐一解决，就是为了降低整篇论文的复杂程度。

如果不进行划分，直接从头至尾书写论文，很显然难以完成。

这样的程序设计易于理解，但是，很明显具有很高的关联性。

在程序设计的过程中，一旦发生需求的改变，很难应对。

而面向对象的程序设计，不是以过程为导向，而是以对象为导向。

程序设计中，会划分出不同的类，通过类来创建出对象。

整个程序是一系列对象的集合，每个对象都可以接收其他对象发过来的消息，并处理这些消息，程序的执行过程就是一系列消息在各个对象之间传递和处理。

面向对象程序设计的优势还体现在它的五个基本原则。

单一职责原则：SRP(Single Responsibility Principle)

是指一个类的功能要单一，具备的特性与类要具有很高的相关性，降低了代码的复杂程度。

开放封闭原则：OCP(Open - Close Principle)

功能模块在扩展性方面应该是开放的而在修改性方面应该是封闭的。

例如饲养动物的示例中，允许增加喂养其它动物的功能，也就是说对于扩展是开放的。

而增加喂养其他动物的功能时，不用对已有的喂养函数进行修改，这是说明对于修改是封闭的。

里氏替换原则：LSP (the Liskov Substitution Principle)

子类应当可以替换超类并出现在超类能够出现的任何地方。

就像俗语说的“龙生龙，凤生凤，老鼠儿子会打洞”，需要打洞了，不管是老鼠爸爸还是老鼠儿子都能胜任。

但是，网上有一个关于鸟类的例子。

在生物学中，企鹅属于鸟类；不过，企鹅不会飞。

这特么就尴尬了。如果我们为鸟类定义飞的方法，企鹅这个子类无法实现这个方法。

很明显这违反了里氏替换原则。

那么是如何解决的呢？

解决方案是：鸟类不包含会飞的方法，而是定义一个会飞的鸟的子类和一个企鹅类，这样飞的方法就变成了子类的扩展，企鹅类也就不需要对飞的方法进行处理，不再违反里氏替换原则。

依赖倒置原则：DIP(the Dependency Inversion Principle)

具体依赖抽象，上层依赖下层。这样的依赖是倒置的。

例如饲养动物的示例中：

- 各个动物类具有抽象的特性，饲养员类是具体的实现。
- 进食是饲养的前提条件，没有进食无需饲养，所以动物类是下层，饲养类是上层。
- 在饲养动物这一功能的具体实现时，依赖的是各个动物进食的特性，没有进食特性，则无法实现饲养。（具体依赖抽象）

- 并且，只要具备进食特性的动物，都可以被饲养，而不是，依赖新增一种饲养方法。（底层不能依赖上层）

接口分离原则：ISP(the Interface Segregation Principle)

接口分离原则指在设计时，多个专用接口优于一个单一的通用接口。每个接口可以为单一的客户程序调用；这样能够避免，当某个客户程序需求变化时，对其他客户程序带来影响。

这就好像家里的电源，如果总电源上，通过一个插座（通用接口）连接所有的电器，当一个电器发生故障需要维修，就需要断掉总电源，导致其他电器受到影响。

那么，如果总电源上分出多个插座，每个插座（专用接口）连接一个电器，维修的时候，就只需要断掉专用的插座电源，其他电器依然能够正常工作。

另外，在程序设计中，大家还要理解耦合和内聚的概念。

这里，我偷懒引用网络上的一段内容，大家自行理解。

耦合简单地说，软件工程中对象之间的耦合度就是对象之间的依赖性。指导使用和维护对象的主要问题是对象之间的多重依赖性。对象之间的耦合越高，维护成本越高。因此对象的设计应使类和构件之间的耦合最小。耦合可以分为以下几种，它们之间的耦合度由高到低排列如下：

内容耦合。当一个模块直接修改或操作另一个模块的数据时，或一个模块不通过正常入口而转入另一个模块时，这样的耦合被称为内容耦合。内容耦合是最高程度的耦合，应该避免使用之。

- 公共耦合。两个或两个以上的模块共同引用一个全局数据项，这种耦合被称为公共耦合。在具有大量公共耦合的结构中，确定究竟是哪个模块给全局变量赋了一个特定的值是十分困难的。
- 外部耦合。一组模块都访问同一全局简单变量而不是同一全局数据结构，而且不是通过参数表传递该全局变量的信息，则称之为外部耦合。
- 控制耦合。一个模块通过接口向另一个模块传递一个控制信号，接受信号的模块根据信号值而进行适当的动作，这种耦合被称为控制耦合。
- 标记耦合。若一个模块A通过接口向两个模块B和C传递一个公共参数，那么称模块B和C之间存在一个标记耦合。
- 数据耦合。模块之间通过参数来传递数据，那么被称为数据耦合。数据耦合是最低的一种耦合形式，系统中一般都存在这种类型的耦合，因为为了完成一些有意义的功能，往往需要将某些模块的输出数据作为另一些模块的输入数据。
- 非直接耦合。两个模块之间没有直接关系，它们之间的联系完全是通过主模块的控制和调用来实现的。
- 耦合是影响软件复杂程度和设计质量的一个重要因素，在设计上我们应采用以下原则：如果模块间必须存在耦合，就尽量使用数据耦合，少用控制耦合，限制公共耦合的范围，尽量避免使用内容耦合。

内聚与耦合

内聚：标志一个模块内各个元素彼此结合的紧密程度，它是信息隐蔽和局部化概念的自然扩展。内聚是从功能角度来度量模块内的联系，一个好的内聚模块应当恰好做一件事。它描述的是模块内的功能联系。

总结程序讲究的是低耦合，高内聚。

就是同一个模块内的各个元素之间要高度紧密，但是各个模块之间的相互依存度却要不那么紧密。

内聚和耦合是密切相关的，同其他模块存在高耦合的模块意味着低内聚，而高内聚的模块意味着该模块同其他模块之间是低耦合。在进行软件设计时，应力争做到高内聚，低耦合。

异常的处理

编写的代码出现异常的情况十分常见，例如下面这段代码。

示例代码：（零除异常）

```
1 def get_error():
2     print(1 / 0)
3
4 get_error()
```

异常信息:

```
1 Traceback (most recent call last): # 回溯 (最近一次命令)
2   File "D:/MyProject/case.py", line 5, in <module> # 文件"所在路径", 位于第2行, 模块内
3     get_error() # 出错的函数名称
4   File "D:/MyProject/case.py", line 3, in get_error # 文件"所在路径", 位于第3行,
   get_error方法中
5     print(1 / 0) # 出错的语句
6 ZeroDivisionError: division by zero # 0除错误: 除数为0
```

异常信息中, 我做了中文标注, 应该很容易看懂。

当发生异常时, 程序捕捉到这个异常显示出来, 并且按从外到内的顺序给出了不同层级出错的位置。

这是程序给我们自动抛出的异常信息。

在Python中内置的异常有很多, 见下表: (简单看看即可)

```
1 BaseException # 所有异常的超类
2 +-- SystemExit # 解释器退出错误
3 +-- KeyboardInterrupt # 键盘中断执行错误
4 +-- GeneratorExit # 生成器错误
5 +-- Exception # 所有标准异常的超类
6     +-- StopIteration # 停止迭代错误
7     +-- StopAsyncIteration # 停止异步迭代错误
8     +-- ArithmeticError # 计算错误
9         | +-- FloatingPointError # 浮点计算错误
10        | +-- OverflowError # 数值溢出错误
11        | +-- ZeroDivisionError # 零除错误
12    +-- AssertionError # 断言失败错误
13    +-- AttributeError # 特性错误
14    +-- BufferError # 缓冲错误
15    +-- EOFError # EOF标记错误
16    +-- ImportError # 导入错误
17        +-- ModuleNotFoundError # 模块不存在错误
18    +-- LookupError # 查询错误
```

```
19 |     +-- IndexError # 索引错误
20 |     +-- KeyError # 键错误
21 +-- MemoryError # 内存错误
22 +-- NameError # 标识符错误
23 |     +-- UnboundLocalError # 未绑定局部变量错误
24 +-- OSError # 操作系统错误
25 |     +-- BlockingIOError # 阻塞输入输出错误
26 |     +-- ChildProcessError # 子进程错误
27 |     +-- ConnectionError # 连接错误
28 |         +-- BrokenPipeError # 管道中断错误
29 |         +-- ConnectionAbortedError # 连接失败错误
30 |         +-- ConnectionRefusedError # 连接拒绝错误
31 |         +-- ConnectionResetError # 连接重置错误
32 |     +-- FileNotFoundError # 文件已存在的错误
33 |     +-- FileNotFoundError # 文件未发现的错误
34 |     +-- InterruptedError # 中断错误
35 |     +-- IsADirectoryError # 目标为目录的错误
36 |     +-- NotADirectoryError # 目录不存在的错误
37 |     +-- PermissionError # 许可错误
38 |     +-- ProcessLookupError # 进程查询错误
39 |     +-- TimeoutError # 超时错误
40 +-- ReferenceError # 引用错误
41 +-- RuntimeError # 运行时错误
42 |     +-- NotImplementedError # 未执行错误
43 |     +-- RecursionError # 递归错误
44 +-- SyntaxError # 语法错误
45 |     +-- IndentationError # 缩进错误
46 |         +-- TabError # Tab错误
47 +-- SystemError # 系统错误
48 +-- TypeError # 类型错误
49 +-- ValueError # 值错误
50 |     +-- UnicodeError # Unicode相关错误
51 |         +-- UnicodeDecodeError # Unicode解码错误
52 |         +-- UnicodeEncodeError # Unicode编码错误
53 |         +-- UnicodeTranslateError # Unicode转换错误
54 +-- Warning # 警告的超类
55 |     +-- DeprecationWarning # 关于弃用功能的警告
56 |     +-- PendingDeprecationWarning # 关于功能将被弃用的警告
57 |     +-- RuntimeWarning # 关于可疑运行时行为的警告
58 |     +-- SyntaxWarning # 关于可疑语法的警告
```

```
59      +-- UserWarning # 关于由用户代码生成的警告
60      +-- FutureWarning # 关于构造将在语义上有改变的警告
61      +-- ImportError # 关于模块导入中可能出现错误的警告
62      +-- UnicodeWarning # 关于Unicode的警告
63      +-- BytesWarning # 关于字节和字节数组的警告
64      +-- ResourceWarning # 关于资源使用的警告
```

当异常出现时，系统会给出相应的异常信息。

不过，这些系统给出的异常信息有些艰涩难懂，不够友好。

那么，能不能显示自定义的异常信息呢？

引发异常：raise 语句

我们可以使用raise 语句，显式的引发异常。

示例代码：

```
1 num1 = input('请输入被除数：')
2 num2 = input('请输入除数：')
3 if int(num2) == 0:
4     raise ZeroDivisionError('除数不能为零！')
```

通过异常类可以调用raise语句，此时我们可以为异常类指定显示的异常信息。

当我们运行代码，输入1和0回车之后，就会显示出异常。

```
1 Traceback (most recent call last):
2   File "D:/MyProject/case.py", line 5, in <module>
3     raise ZeroDivisionError('除数不能为零！')
4 ZeroDivisionError: 除数不能为零！
```

这样虽然能够引发异常，也能给出提示，但是需要我们自己来判断输入的内容。

并且，异常信息还不是很友好，至少我想让异常类型也是中文，我也不想看到除了最后一句提示之外的信息。

捕捉异常：try/except 语句

接下来，我们采用科学的方法捕捉可能发生的异常，并在捕捉到异常时进行处理。

示例代码：

```
1 while True:
2     try:
3         num1 = int(input('请输入被除数：'))
4         num2 = int(input('请输入除数：'))
```

```

5         print('计算结果为: ', num1 / num2)
6     except ZeroDivisionError:
7         print('零除错误: 除数不能为零! ')

```

当我们运行代码，输入1和0回车之后，这时显示了我们在print语句中输入的提示内容。

try和except语句之间，我们要写入可能会发生异常的语句块，一旦发生指定的异常（except语句中的异常类所对应的异常）就会被except语句捕捉到，然后，在except语句下方可以进行处理。

当然，我们输入的内容也可能不是数值，这个时候怎么将值异常也捕捉到呢？我们可以再加入一个except语句进行捕捉。

示例代码：

```

1  except ZeroDivisionError:
2      print('零除错误: 除数不能为零! ')
3  except ValueError:
4      print('数值错误: 必须输入数字! ')

```

但是，如果我们想不管是0除错误，还是输入值的错误都给一个共同的提示，怎么处理呢？

我们可以通过except语句捕捉多个异常，然后进行相同的处理。

示例代码：

```

1  except (ZeroDivisionError, ValueError):
2      print('错误提示: 请输入正确的数值! ')

```

还有，如果不想做什么处理，还是显示系统异常信息又怎么处理呢？

还是使用raise语句，不带任何参数即可。

示例代码：

```

1  except (ZeroDivisionError, ValueError):
2      raise

```

另外，如果我们不想让程序终止，也要显示异常信息呢？

我们可以把捕捉到的异常通过as关键字存入变量，然后显示输出这个变量。

示例代码：

```

1  except (ZeroDivisionError, ValueError) as record:
2      print(record)

```

有的时候，我们不知道都会发生什么异常，可以在except语句后方什么都不添加。

示例代码：

```
1 except:
2     print('发生了一些错误，但不知道是什么！')
```

不过这样显然不可取，我们还是希望知道发生了什么？

那就用标准异常的超类帮我们获取所有的异常。

示例代码：

```
1 except Exception as e:
2     print(e)
```

这样就能捕获到所有的异常，并且把相应的异常信息显示出来。

除了这些操作，还有这样的操作！

一旦输入错误，我们提示用户，并且重新获取输入，指导输入正确，给出计算结果，

示例代码：

```
1 while True:
2     try:
3         num1 = int(input('请输入被除数: '))
4         num2 = int(input('请输入除数: '))
5         print('计算结果为: ', num1 / num2)
6
7     except Exception as e:
8         print(e)
9     else:
10        break
```

在上方代码中，我们加入了while循环，并且加入了else子句，这样只要是异常出现就会不停执行try中的语句块，直到输入正确，不再发现异常。这时，会执行else中的语句块，跳出循环。

异常收尾：finally 语句

最后，还有一个内容。

比如，不管程序有没有正确执行，我们都想提示用户当前程序进行了一次计算。

示例代码：

```

1  count = 0
2  while True:
3      try:
4          num1 = int(input('请输入被除数: '))
5          num2 = int(input('请输入除数: '))
6          print('计算结果为: ', num1 / num2)
7
8      except Exception as e:
9          print(e)
10     else:
11         break
12     finally:
13         count += 1
14         print('第%d次计算完毕! ' % count)

```

我们在程序的开始定义了一个全局变量count，不管计算是否有异常出现，我们都在finally中进行计数的递增，并显示输出提示。

构造方法和属性

构造方法在之前的教程中我们已经接触过。

它就是__init__()这个方法。

特别说明：在Python中，这种两侧带有下划线的方法称为魔法方法或特殊方法，它们都有一些特殊的用途。

对象的创建就是通过构造方法来完成的，它的主要功能是完成对象的初始化。

当实例化一个类的对象时，会自动调用构造方法。

构造方法和其他方法一样也可以重写。

不过需要注意，重载构造方法，有的时候会出现问题。

例如，我们在超类中为构造方法定义了某个特性，而在子类中也为构造方法定义了某个特性，这个时候构造方法被重写。

当我们通过子类实例化的对象，调用超类中的特性时，会找不到这个特性，从而引发异常。

示例代码：（错误示例）

```

1  class SuperClass:
2      def __init__(self): # 构造方法
3          self.name = '小楼'
4
5  class SubClass(SuperClass):
6      def __init__(self): # 重写构造方法
7          self.learn = 'Python'

```

```

8
9     def say(self):
10         print('我是%s, 我在学习%s。' % (self.name, self.learn))
11
12 SubClass().say() # 抛出异常

```

运行上方代码，会抛出异常。

AttributeError: 'SubClass' object has no attribute 'name'

特性错误：'SubClass' 类的对象不包含特性 'name'

那么，如何解决这个问题？

我们可以使用super函数，通过super函数访问超类的特性。

示例代码：

```

1 class SubClass(SuperClass):
2     def __init__(self): # 重写构造方法
3         super().__init__()
4         self.learn = 'Python'
5
6     def say(self):
7         print('我是%s, 我在学习%s。' % (self.name, self.learn))
8
9 SubClass().say() # 显示输出结果为：我是小楼，我在学习Python。

```

并且，不管有多少个超类，都只需要一个super()函数，就能够访问所有超类的特性。

但是，需要为每一个超类的构造方法使用super()函数才可以。

```

1 class SuperClass1:
2     def __init__(self): # 构造方法
3         super().__init__() # 调用super函数
4         self.name = '小楼'
5
6 class SuperClass2:
7     def __init__(self):
8         super().__init__() # 调用super函数
9         self.sex = '帅哥'
10
11 class SubClass(SuperClass1, SuperClass2):
12     def __init__(self): # 重写构造方法

```



```

13         super().__init__() # 调用super函数
14         self.learn = 'Python'
15
16     def say(self):
17         print('我是%s%s, 我在学习%s。' % (self.sex, self.name, self.learn))
18
19
20 SubClass().say() # 显示输出结果为: 我是帅哥小楼, 我在学习Python。

```

接下来，我们了解什么是属性。

在之前的教程中，我们接触过访问器，就是类中定义的get_name()和set_name()这样的函数，用于控制特性的设置和读取。

这样定义的函数，我们是希望能够在设置和读取特性时，能够做一些其他的事情。

示例代码：

```

1 class Shape: # 定义形状类
2     def get_height(self): # 定义获取特性的函数
3         return self.height # 返回特性值
4
5     def set_height(self, height): # 定义修改特性的函数
6         if isinstance(height, (int, float)): # 验证参数类型
7             self.height = height # 设置特性值
8         else:
9             raise TypeError # 引发类型异常

```

上方代码中，我们在修改特性时，加入了对参数类型验证的语句，如果输入的参数是整数或小数，能够修改特性的值，否则，就会引发异常。

特别说明：代码中除非必须尽量回避isinstance()函数，它不是多态的。

不过，并不是所有的特性都会通过私有的访问器进行访问的控制。

接下来，我们再来看一段代码。

对于立方体来说，有长、宽、高的特性，如果我们提供了体积的元组，这些长宽高的特性能够动态获取。

示例代码：

```

1 class Cube: # 定义立方体类
2     def __init__(self):
3         self.length = 0
4         self.width = 0
5         self.height = 0

```

```

6
7     def set_cubage(self, cubage): # 定义修改特性的函数
8         if len([i for i in cubage if isinstance(i, (int, float))]) == 3: # 验证参数类型，验证通过的数值形成列表。
9             self.length, self.width, self.height = cubage # 设置特性值
10        else:
11            raise TypeError # 引发类型异常
12
13    def get_cubage(self): # 定义获取特性的函数
14        return self.length, self.width, self.height # 返回特性值
15
16    cube = Cube()
17    cube.set_cubage((10, 10, 10))
18    print(cube.get_cubage()) # 显示输出结果为: (10, 10, 10)
19    cube.length = 20
20    print(cube.get_cubage()) # 显示输出结果为: (20, 10, 10)

```

在上方这段代码中，体积不是特性。

如果我们想把体积也变成特性，那么，需要将体积特性放在访问器。

示例代码：（错误示例）

```

1     def set_cubage(self, cubage):
2         if len([i for i in cubage if isinstance(i, (int, float))]) == 3:
3             self.cubage = cubage # 新增特性
4             self.length, self.width, self.height = cubage
5         else:
6             raise TypeError
7
8     def get_cubage(self):
9         return self.cubage # 修改返回值为新增特性
10
11    cube = Cube()
12    cube.set_cubage((10, 10, 10))
13    print(cube.get_cubage()) # 显示输出结果为: (10, 10, 10)
14    cube.length = 20
15    print(cube.get_cubage()) # 显示输出结果没有改变，仍然为: (10, 10, 10)

```

经过修改后的代码，虽然将新特性放入了访问器，但是，当修改长度特性时，结果没有变化。

这是因为修改length特性，并没有调用set_cubage()方法，所以get_cubage()方法的返回值不变。

那怎么解决这个问题呢？

如果，将每个特性都放入访问器，在设置特性的方法中重新绑定cubage的值，是能够正常实现的。

但是，这样的话，整个代码会变得很乱。

而且，导致使用了这个类的客户程序（使用代码的代码）也都需要修改。

例如，设置长度就不能再使用cube.length这样的特性进行设置，而需要使用cube.set_length()这样的方法。

其实，只加入一句代码来解决这个问题。

示例代码：

```
1 cubage = property(get_cubage,set_cubage)
```

上方代码中，cubage是新定义的特性，后方的property函数将两个访问器（注意顺序）绑定到这个特性，这样就可以正常使用新特性了。

我们测试一下。

示例代码：

```
1 cube = Cube()
2 cube.set_cubage((10, 10, 10))
3 print(cube.get_cubage()) # 显示输出结果为: (10, 10, 10)
4 cube.length = 20
5 print(cube.get_cubage()) # 显示输出结果为: (20, 10, 10)
6 cube.cubage=(30,40,50)
7 print(cube.get_cubage()) # 显示输出结果为: (30, 40, 50)
```

不管通过set_cubage()方法，还是直接设置特性，都能够正常返回结果。

property()实际上不是函数，它是一个类，所有的工作由这个类的特殊方法来完成。

property(fget=None,fset=None,fdel=None,fdoc=None)具有四个参数,这四个参数分别是特性的获取、设置、删除特性的方法以及说明文档字符串。

如果没有参数，属性不可读写；只写入一个参数，则属性为只读；写入两个参数，属性可以读写。

通过property()函数定义的特性（变量），称为属性。

而普通的特性（变量），称为字段。

所以，属性就是包含了访问器的字段（变量）。

那么，到此为止我们了解到类的实例特性包含：字段、属性与方法。

最后，我们再来接触一些魔法方法。

__getattr__(self,name): 获取特性值时自动调用的方法，参数name填入特性名称。

__getattr__(self, item): 通过调用来获取特性值的方法，参数填入特性名称。

__setattr__(self,key,value): 设置特性值时自动调用的方法，参数key填入特性名称，参数value填入特性的值。

__delattr__(self,name): 删除特性的方法，参数name填入特性名称。

__dict__: 包含一个字典，存储实例的所有特性，包括新增的特性。

这些魔法方法，我们可以重写来满足我们的一些需求。

示例代码：（__getattr__，__setattr__，delattr__，__dict__）

```
1 class Class:
2     def __init__(self, sex, age):
3         self.sex = sex
4         self.age = age
5
6     def __getattr__(self, item): # 重写getattr方法
7         if self.sex == '女' and item == 'age': # 符合指定条件的情形
8             return '保密' # 返回指定的值
9         else: # 其他情形
10            return self.__dict__[item] # 从字典获取返回值
11
12    def __setattr__(self, key, value): # 重写setattr方法
13        if key == 'age' and value <= 0: # 符合指定条件的情形
14            raise ValueError('年龄数值错误!') # 抛出异常
15        else:
16            self.__dict__[key] = value # 修改字典中的值
17
18    def __delattr__(self, item): # 重写delattr方法
19        del self.__dict__[item] # 删除字典中指定的键值
20
21 cls = Class('男', 18)
22 print("我是%s孩,我的年龄%s。" % (cls.__getattr__('sex'), cls.__getattr__('age')))
23 # 显示输出结果为：我是男孩,我的年龄18。
24
25 cls.__setattr__('sex', '女')
26 cls.__setattr__('age', 16)
27 print("我是%s孩,我的年龄%s。" % (cls.__getattr__('sex'), cls.__getattr__('age')))
28 # 显示输出结果为：我是女孩,我的年龄保密。
29
30 cls.__delattr__('sex')
31 print(cls.__dict__) # 显示输出结果为：{'age': 16}
```

示例代码：（__getattribute__）

```
1 class Class:
2     def __init__(self, sex, age):
```

```

3         self.sex = sex
4         self.age = age
5
6     def __getattr__(self, item): # 重写getattr方法
7         attr = super().__getattr__
8         if attr('sex') == '女' and item == 'age': # 符合指定条件的情形
9             return '保密' # 返回指定的值
10        else: # 其他情形
11            return attr(item) # 返回原始的值
12
13
14    cls = Class('男', 18)
15    print("我是%s孩,我的年龄%s。" % (cls.sex, cls.age)) # 显示输出结果为: 我是男孩,我的年龄18。
16    cls.sex = '女'
17    cls.age = 16
18    print("我是%s孩,我的年龄%s。" % (cls.sex, cls.age)) # 显示输出结果为: 我是女孩,我的年龄保密。

```

上方代码中，要格外注意：在重写方法时，读取特性不能通过“self.特性”的方式去获取，否则会引发异常。因为，通过“self.特性”的方式获取时，会自动调用__getattr__()方法自身，这就会导致不停的循环调用。所以，一定要通过超类的方法进行获取，这里使用了super函数。其实，还有另外一种方法，就是通过object这个超类去调用。

因为，在Python 3（注意版本）中定义的类，都会继承object类（即便没有显示继承）。

示例代码：（方法部分）

```

1 def __getattr__(self, item): # 重写getattr方法
2     attr = object.__getattr__
3     if attr(self, 'sex') == '女' and item == 'age': # 符合指定条件的情形
4         return '保密' # 返回指定的值
5     else: # 其他情形
6         return attr(self, item) # 返回原始的值

```

注意：使用object类去调用方法的时候，需要提供两个参数，第一个是self，第二个是特性名称。

迭代器和生成器

一、迭代器

“迭代”这个词，如果没有接触过（例如IT界之外的人或IT小白），可能理解起来会有一些困难。

其实，迭代就是指重复的去做一些事情（例如：一晚上迭代七次，就是把某件事情做七次）。

不过，有些时候，迭代需要有限制，例如刚才的例子，迭代太多了容易精尽人亡。

还有就是，有需要了再迭代，不需要的时候迭代，往往会影响性能。

那么，迭代器又是什么？

迭代器就是一段代码，这段代码能够根据需求，重复完成某些事情。

这段代码由两部分组成，一部分是`__iter__()`方法，一部分是`__next__()`方法。

你可能会理解成，“官人我还要”是`__next__`方法来完成，而官人要干的事是由`__iter__()`方法完成。

那你就想错了！

正好相反，要做的事是通过`__next__()`方法来实现。

我们来看一段关于偶数迭代器的代码。（又是偶数.....）

示例代码：

```
1 class Even:
2     def __init__(self):
3         self.even = 0 # 定义变量保存数值
4
5     def __next__(self): # 重写next方法
6         self.even += 2 # 计算每次迭代的偶数
7         return self.even # 返回计算结果
8
9     def __iter__(self): # 重写iter方法
10        return self # 返回实例对象为可迭代对象
11
12
13 even = Even()
14 # print(list(even)) # 想体验精尽人亡就取消这句代码的注释，然后运行.....
15 for i in range(5):
16     print(next(even)) # 显示输出结果为：2 4 6 8 10
```

上方这段代码中，`__next__()`方法中的代码定义了如何获取偶数。

而`__iter__()`方法只是返回实例对象。

其实，实现了`__next__()`方法的对象就是迭代器。

大家可以注释掉上方代码中的`__iter__()`方法，运行依然正常。

那么，`__iter__()`方法的作用是什么呢？

大家先别取消`__iter__()`方法的注释，把上面代码中的for循环改成下面这段代码。

示例代码：

```
1 for i in even:
2     if i <= 10:
```

```

3         print(i) # 显示输出结果为: 2 4 6 8 10
4     else:
5         break

```

运行代码，会发现这次for循环抛出了异常。

TypeError: 'Even' object is not iterable

类型错误: 'Even' 对象不能够迭代

然后，大家取消__iter__()方法的注释，再次运行代码，就能够正常运行了。

也就是说，实现了__iter__()方法的对象才是可迭代对象。

接下来，我们观察一下刚才的代码。

这个迭代器能够生成无限数量的偶数。

所以，如果不加限制，直接通过print语句显示输出迭代器的后果.....（如果运行后没死机，请告诉我电脑哪买的.....）

通过print()函数显示输出迭代器对象，就会让迭代器不停.....不停.....不停.....生成偶数，一直到系统资源耗光.....

实际上，迭代器并不是直接生成所有的计算结果。

如果不用print()函数调用，它不会有任何偶数产生。

我们通过内置函数next()，可以每次获取迭代器计算出来的一个偶数。

每一次取值的时候，迭代器对象才会调用自身的__next__()方法计算新的数值并返回。

直到迭代器没有值被返回时，会抛出StopIteration（停止迭代）异常。

所以，当我们需要一个很多值的列表（list），这个列表又能够用迭代器产生，就不要用列表（list）。

列表包含的值太多，会占用大量内存，而迭代器则不会。

接下来，我们看一下如何通过迭代器获取一个有限数量列表。

还是用获取偶数的例子。

示例代码：

```

1 class Even:
2     def __init__(self, count): # 定义获取次数的参数
3         self.even = 0
4         self.count = count # 定义变量保存获取次数
5
6     def __next__(self):
7         if self.even < self.count * 2: # 判断最后一次生成的偶数小于最大的偶数
8             self.even += 2
9             return self.even
10        else:
11            raise StopIteration # 到达指定数量时停止迭代
12
13    def __iter__(self):
14        return self
15

```

```
16 even = Even(10) # 实例化并传入参数
17 print(list(even)) # 显示输出结果为: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

通过上方代码大家能够看到，通过内置函数list()可以把有限数量的迭代器对象转换为列表。
另外，内置函数iter()能够从可迭代对象中获取生成器，并通过内置函数next()逐一读取。
示例代码：

```
1 tup=(1,2,3,4,5)
2 ite=iter(tup)
3 for i in range(5):
4     print(next(ite)) # 显示输出结果为: 1 2 3 4 5
```

注意：for语句不要写成“for i in ite”，因为这样也在调用迭代器取值，每个迭代器的值只能获取一次，取完为止。
示例代码：（错误）

```
1 for i in ite:
2     print(next(ite)) # 显示输出结果为: 2 4和StopIteration异常
```

上面代码之所以显示输出2和4，是因为1、3、5被for语句中的i取走了。
而StopIteration异常，是因为没有限制取值次数，取不到值所产生的。

二、生成器

生成器是通过函数实现的迭代器。

这里我们会接触到一个新的关键词yield（生产）。

看到英文后面的中文，是不是感到亲切了很多？

所以，不要被任何英文单词所困扰。

yield这个关键词能够指定生成某个函数计算结果。

所以，包含yield语句的函数都是生成器。

和迭代器一样，这些结果不是直接生成好的，而是能够在使用时逐一生成。

例如，能够从一个列表中获取所有能够被3整除的数字的生成器。

示例代码：

```
1 def generate(lst): # 定义生成器函数
2     for i in lst: # 循环遍历参数列表
3         if i % 3 == 0: # 判断是否符合条件
4             yield i # 生成符合条件的数值
5
6 lst = [97, 19, 29, 72, 16, 93, 47, 92, 26, 75, 62, 89, 58, 10, 65, 63, 13, 52, 51, 60]
```



```

7 g = generate(lst)
8 while True:
9     try:
10         print(next(g)) # 通过内置函数next逐一生成数值
11     except: # 捕获到异常时跳出循环
12         break

```

上方代码中的while循环也可以用for语句代替

```

1 for num in g: print(num)

```

这里主要展示可以使用内置函数next()逐一获取生成器的计算结果。

另外，像这种通过生成器生成列表的操作，我们还可以使用生成器推导式（也叫生成器表达式）。

这个生成器推导式和我们学过的列表推导式非常相像，不同的是生成器推导式是写在两个圆括号“()”中。

刚才的例子，我们可以写成生成器推导式：

```

1 exp=(x for x in lst if x %3==0 ) # 通过生成器推导式定义生成器
2 print(list(exp)) # 使用内置函数list()将生成器转换为列表，显示输出结果为：[72, 93, 75, 63, 51, 60]

```

接下来，我们来看一个能够将嵌套列表中的所有元素取出生成非嵌套列表的生成器。

例如：通过列表[[1,2],3,4,[5,6],[7,8,9]]生成列表[1,2,3,4,5,6,7,8,9]

目标列表是双层嵌套的列表，所以我们可以通过嵌套双层for循环来完成。

示例代码：

```

1 def generate(lst): # 定义生成器函数
2     for sublist in lst: # 循环遍历参数列表
3         try:
4             for num in sublist: # 循环遍历参数列表的子列表
5                 yield num # 生成符合条件的数值
6         except: # 如果sublist为单个数字会发生异常
7             yield sublist # 发生异常时将单个数字生成
8
9
10 lst = [[1, 2], 3, 4, [5, 6], [7, 8, 9]]
11 print(list(generate(lst))) # 使用内置函数list()将生成器转换为列表，显示输出结果为：[1, 2, 3, 4, 5, 6, 7, 8, 9]

```

看上去完美解决了问题，但是，如果不知道列表的嵌套层级呢？

例如，通过列表[1,2,[3,[4,5]],[6,[7,[8,9]]]]生成列表[1,2,3,4,5,6,7,8,9]

很显然，上面的生成器就没有办法处理了。

我们来定义一个完美的生成器，不管列表嵌套多少层都能够给出正确的结果。

这就需要使用递归来实现了。

示例代码：

```
1 def generate(lst): # 定义生成器函数
2     for sublist in lst: # 循环遍历参数列表
3         try:
4             for element in generate(sublist): # 循环遍历递归的生成器元素
5                 yield element # 将递归的生成器元素生成（3,4,6,7,8,9）
6         except: # 如果sublist为单个数字会发生异常
7             yield sublist # （1,2,5）发生异常时将数字生成，递归中（3,4,6,7,8,9）发生异常时
            将数字生成。
8
9 lst = [1, [2, [3, 4]], 5, [6, [7, [8, 9]]]]
10 print(list(generate(lst))) # 使用内置函数list()将生成器转换为列表，显示输出结果为：[1, 2,
    3, 4, 5, 6, 7, 8, 9]
```

在上方的代码中，列表中的单个数字元素，都会直接发生异常，通过yield sublist语句直接生成，而列表中嵌套的列表数量决定了递归的次数，每一次的递归都会产生一个生成器，这些生成器中的元素内容全部通过yield element语句生成。

这样的生成器，不管列表有多少层嵌套都能够进行转换。

但是，这个生成器只能够对元素全部都是数字的列表进行处理。

如果包含字符串元素，就无法得到正确结果。

例如，对列表[[1, 2], 3, 'abc', 4, [[5, 6, [7, 8]]]]的处理结果是[1, 2, 3, 'a', 'b', 'c', 4, 5, 6, 7, 8]。

而我们期望的结果是[1, 2, 3, 'abc', 4, 5, 6, 7, 8]

所以，代码需要进行更改。

示例代码：（带有注释的语句为新增语句）

```
1 def generate(lst):
2     for sublist in lst:
3         try:
4             try: # 验证元素是否为字符串
5                 sublist + ''
6             except: # 如果发生异常（不是字符串）
7                 pass # 继续执行下方的for循环
8             else: # 否则抛出异常，被外层except捕获。
```

```

9         raise Exception
10    for element in generate(sublst):
11        yield element
12
13    except:
14        yield sublst

```

到这里，我们来做个总结。

生成器是由生成器函数和返回的迭代器组成。

```

1 print(generate) # 未调用函数时，显示输出结果为: <function generate at 0x000000000294E730>
2 print(generate([])) # 调用函数时，显示输出结果为: <generator object generate at
0x0000000002576888>

```

最后，我们再来看3个关于生成器的方法：send()、throw()和close()。

1、send()

send()方法不能直接使用，他只有在生成器挂起状态（至少生成一次）下才能使用。

这里通过前面的案例，演示一下send()方法的使用。

大家可以通过代码注释进行理解。

示例代码：

```

1 def generate(lst):
2     for i in lst:
3         if i % 3 == 0:
4             s = yield i # 通过变量s获取外部send()函数传入的字符串
5             print(s % i) # 显示输出传入的字符串，并将上一次的生成结果插入到字符串中转换说明
                           符的位置。
6
7 lst = [97, 19, 29, 72, 16, 93, 47, 92, 26, 75, 62, 89, 58, 10, 65, 63, 13, 52, 51, 60]
8 g = generate(lst)
9 next(g) # 生成第1个数值，此时生成器为挂起状态。
10 while True:
11     try:
12         g.send('数字%s能够被3整除...') # 将格式化字符串发送到生成器内部
13     except:
14         break

```

注意：send()方法获取到的是前一次生成的结果。

2、throw()

这个方法能够引发生成器异常，还是借用之前的案例。

在生成器中，我们加入try/except语句，捕捉并抛出异常。

示例代码：

```
1 def generate(lst):
2     try: # 捕获异常
3         for i in lst:
4             if i % 3 == 0:
5                 yield i
6     except: # 抛出异常
7         raise Exception('程序已终止')
8
9 lst = [97, 19, 29, 72, 16, 93, 47, 92, 26, 75, 62, 89, 58, 10, 65, 63, 13, 52, 51, 60]
10 g = generate(lst)
11 print(next(g)) # 显示输出结果为: 72
12 g.throw(Exception) # 引发异常
```

3、close()

close()方法用于关闭生成器，关闭之后无法再继续生成。

示例代码：

```
1 def generate(lst):
2     for i in lst:
3         if i % 3 == 0:
4             yield i
5
6 lst = [97, 19, 29, 72, 16, 93, 47, 92, 26, 75, 62, 89, 58, 10, 65, 63, 13, 52, 51, 60]
7 g = generate(lst)
8 print(next(g)) # 显示输出结果为: 72
9 g.close()
10 print(next(g)) # 抛出异常: StopIteration
```

另外，我们还可以通过生成器表达式创建生成器。

大家应该还记得列表推导式可以帮助我们创建一个列表。

生成器表达式和列表表达式非常相像，它的语法是：(表达式 for 变量 in 迭代对象 [条件表达式])。

举个简单的例子。

示例代码：

```
1 n = (i * 2 for i in [1, 2, 3, 4])
2 print(type(n))
3 print(next(n))
4 print(next(n))
5 print(next(n))
6 print(next(n))
7 print(next(n))
```

上方的代码运行结果如下：

```
1 <class 'generator'>
2 2
3 4
4 6
5 8
6 Traceback (most recent call last):
7   File "G:/Python/abc.py", line 7, in <module>
8     print(next(n))
9 StopIteration
```

生成器中使用递归

如果在之前的教程中对递归没有很好的理解，在这里我们可以继续强化。

首先，我们先来看一个场景。

有若干组门，每组有两个门，一扇门后面可以通行，另一扇门后面堵塞，需要找出正确的路线。

如果想找出正确的路线，只能打开任意一扇门，如果通行则记下当前这扇门，然后继续往前，如果堵塞则退出后从另外一扇门通过，并且记下另外一扇门，以此类推，直到走过所有的门。

大家先看下图的左侧（示例代码结合右侧进行理解），理解找出正确路线的过程。

这里，以4组门并且每次先进入左侧门为例。

递归就类似这样的场景，每次都同样的方法去找出正确的门，不同的只是是每次的门不是同一组。

接下来，我们就来看如何通过代码找到路线。

我们把每一组门进行编号，左侧为0，右侧为1。

示例代码：

```

1 def go(doors, count=0, route=[]): # 参数doors为多组门的列表, 参数count为每组门的序号, 参数
  route为路线列表。
2     try: # 捕捉列表索引超出时的异常 (走过最后一组门之后, 找不到新的门。)
3         if doors[count][0] == '通行': # 打开左侧的门, 查看是否通行。
4             return [0] + go(doors, count + 1)
5             # 如果通行, 将左侧的门编号暂存 (和之后所有正确的门编号一起返回), 并进入下一组门。
6         else: # 否则
7             return [1] + go(doors, count + 1)
8             # 进入右侧的门, 将右侧的门编号暂存 (和之后所有正确的门编号一起返回), 并进入下一组
  门。
9     except: # 异常处理 (找不到新的门)
10         return route # 返回正确路线。
11
12 doors = [('堵塞', '通行'), ('通行', '堵塞'), ('堵塞', '通行'), ('堵塞', '通行')]
13 print(go(doors))

```

这个打开多组门寻找正确路线的案例, 比较简单。

接下来, 我们来看一个比较经典的题目: 八皇后

这个题目是说有八个皇后 (种马风格...), 皇后与皇后之间不能互搞 (禁止女同...)。

也就是每个皇后与其他皇后不能再同一条水平线、垂直线、以及对角线上。

我们需要通过代码, 计算出所有的摆放方法, 也就是把每一种摆放方法以列表或元组的形式生成。

如下图: “0, 4, 7, 5, 2, 6, 1, 3”, 就是一种正确的摆放方法。

我们编写代码基本都是为了解决某些问题, 而解决问题的先决条件是理清思路, 这样才能找出解决问题的方法。

接下来, 我们就来分析一下八皇后的解题思路。

首先, 我们模拟一下寻找所有摆放方案的过程。

- 1、棋盘有8行8列, 按由上至下的顺序摆放的话, 每一行只能有一个皇后。
- 2、当我们在第1行任意位置 (列) 摆下皇后, 第2行摆放皇后的时候, 就不能与第1行的皇后在同一列或同一对角线上。而第3行摆放的时候, 不但要注意不能与第1行的皇后有冲突, 也要注意不能与第2行的皇后有冲突。
- 3、每一行的每一列都要尝试摆放, 直到找出所有摆放方案。
- 4、每一行摆放时, 如果有可摆放的位置, 摆放皇后之后进行下一行的摆放, 否则, 放弃当前行的摆放。
- 3、如果摆放的是最后一行, 仍然能够找到合理的摆放位置, 将结果生成; 否则, 继续下一行摆放。

接下来, 我们通过代码, 实现上面的过程。

首先, 解题的关键在于如何确定当前行皇后的摆放位置与之前所有已摆放皇后的位置不冲突。如果与之前皇后位置重合的话, 列编号相同, 也就是列的间隔为0; 如果与之前皇后位置处于对角线的话, 列的间隔与行的间隔相同。这里要注意, 摆放位置在左侧对角线时, 计算的间隔结果为负数, 而右侧对角线计算间隔的结果是正数, 所以, 计算间隔时还需要计算绝对值。

然后，我们可以通过元组或列表保存每一次正确的摆放方案，元组或列表中元素的编号（0-7）为行编号，而元素本身为列编号（每一行摆放的位置）。如果通过列表保存，可以先创建一个长度为8的列表，修改列表中的每一个元素；如果通过元组保存，可以创建一个空元组和每一行正确的摆放位置相连接。

这里，我们通过元组保存结果，来完成这个题目。

示例代码：

```
1 def check(place=(), column=0):
2     # 定义检查位置的函数，参数place为已正确摆放皇后位置的元组，参数column为最新一行将要摆放皇后的位置（列编号）。
3     current = len(place) # 获取当前摆放行的编号
4     for row in range(current): # 遍历当前行之前的每一行
5         if abs(column - place[row]) in (0, current - row):
6             # 检查当前行摆放皇后的列与任何一行摆放皇后的列，是否重合或者处于对角线。place[row]为每一行皇后所处的位置（列编号）。
7             return False # 如果检查到皇后的位置有冲突，则返回假值。
8     return True # 默认返回真值
9
10
11 def queen(place=()):
12     for column in range(8): # 轮流将皇后摆放到行的每一列
13         if check(place, column): # 判断当前行皇后的位置是否与其它已摆放皇后的位置处于同一列或者对角线的位置
14             if len(place) == 7: # 如果是最后一行
15                 yield (column,) # 生成最后一行列的元组
16                 # yield place + (column,) # 生成最终结果（方案2：替代上一句）
17             else: # 否则
18                 for result in queen(place + (column,)): # 获取递归生成的结果
19                     yield (column,) + result # 生成当前行的列与递归生成的结果组成的新元组
20                     # yield result # 生成递归生成的最终结果（方案2：替代上一句）
21
22
```

在上方代码中check()函数负责检查当前皇后摆放位置是否与之前皇后摆放位置有冲突。

另外，在上方代码中，yield语句有两种方案，大家可以替换尝试。

最后，我们还可以将生成的摆放结果，以图形的形式显示输出。

在上方代码的基础上，继续添加代码。

示例代码：

```
1 from random import choice # 从随机模块中导入随机选择的方法
```

```

2
3 gen = queen()
4 ran = choice(list(gen)) # 随机获取一个生成结果存入变量
5 print(ran)
6 for i in ran:
7     print('□ ' * i + '■ ' + '□ ' * (7 - i)) # 注意特殊符号后方都带有一个空格

```

显示输出结果类似下方内容：

(7, 2, 0, 5, 1, 4, 6, 3)

```

□ □ □ □ □ □ ■
□ □ ■ □ □ □ □
■ □ □ □ □ □ □
□ □ □ □ ■ □ □
□ ■ □ □ □ □ □
□ □ □ □ ■ □ □
□ □ □ □ □ ■ □
□ □ □ ■ □ □ □

```

最后，再为大家呈上通过列表完成的代码。

这段代码中，思路和上方代码一样，不过没有把检查冲突的代码独立抽出为函数。

示例代码：

```

1 def queen(place=[None] * 8, current=0):
2     # 定义皇后摆放方法，参数place为摆放位置列表，参数current为当前要摆放皇后的行（简称：当前行）。
3     for column in range(8): # 尝试将皇后摆放到行的每一列
4         place[current], mark = column, True # 在当前行的每一列轮流摆放皇后，并且设置标记为真值。
5         for row in range(current): # 遍历当前行之前的每一行
6             if abs(column - place[row]) in (0, current - row):
7                 # 检查当前行摆放皇后的列与任何一行摆放皇后的列，是否重合或者处于对角线。
8                 mark = False # 设置标记为假值，即当前行摆放皇后的位置和其他行摆放皇后的位置有冲突。
9                 break # 跳出对每一行皇后摆放位置的检查
10        if mark: # 如果检查皇后摆放的位置没有冲突
11            if current == 7: # 如果当前行是第8行
12                yield place # 生成最终结果
13                # yield tuple(place) # 如果使用list()函数获取全部生成结果需要转换为数组，否则都是[7,7,7,7,7,7,7,7]。
14            else: # 否则
15                for result in queen(place, current + 1): # 获取递归生成的最终结果

```


装饰器 (Decorators) 的使用

先忘掉“装饰器”这三个字。（你刚才不说不就完了吗...）

我们先来看一些代码，这些代码是分别获取当前系统时间的时、分、秒。

示例代码：

```
1 import time
2
3 def get_hours():
4     return time.localtime()[3]
5
6 def get_minutes():
7     return time.localtime()[4]
8
9 def get_seconds():
10    return time.localtime()[5]
11
12 print(get_hours(), get_minutes(), get_seconds(), sep=':') # 显示输出结果类似: 12:49:2
```

上方代码中，用到了time模块，通过localtime()函数，我们能够获取到本地时间的元组。

在本地时间的元组中包含的元素依次为：年,月,日,时,分,秒,星期几,本年第几天，是否夏令时。

所以，大家能够看到，在上方代码中，通过元组中元素的编号3、4、5获取到了时、分、秒的数值。

不过，当时、分、秒的数值小于10的时候，都是1位数字（0:0:0），如果想得到两位数字（00:00:00）怎么办呢？

当然，我们可以修改每一个写好的函数；但是，这样有点笨。

而且，不一定哪一天，你又想要1位数字，还要全部改回来。

最好的办法应该是再写一个函数，用这个函数处理时、分、秒这三个函数的返回值。

接下来，我们加入一段代码。

示例代码：

```
1 def new_time(fun): # 定义处理时间的函数，对传入函数的返回值进行再处理。
2     return ('0' + str(fun()))[-2:] # 返回处理结果
3
4 print(new_time(get_hours), new_time(get_minutes), new_time(get_seconds), sep=':') # 显示输出结果类似: 12:49:02
```

在上方代码中，我们把函数作为参数传入新写的函数，然后对传入函数的返回值进行处理，将处理后的值返回。这样做效果上没有问题。

但是，大家注意print语句，参数全部和刚才不一样了。

那么大家想一下，如果是之前写好的函数，也在被很多客户代码（使用这个函数的代码）使用，通过这样处理的话，大量的客户代码都需要修改，这样还不如修改写好的这几个函数。

那么，能不能既不修改写好的函数，也不修改客户代码，来解决这个问题呢？

大家是否还记得之前学过的闭包？

简单来说，闭包是函数里面嵌套函数，外层函数返回值为内层函数。

我们把刚才新增的代码修改一下。

示例代码：

```
1 def format_time(fun): # 函数传入
2     def new_time(): # 定义处理函数返回值的函数，注意，参数由外层函数获取，这个函数不需要参数。
3         return ('0' + str(fun()))[-2:] # 返回处理结果
4     return new_time # 闭包，返回处理函数的函数。
```

上面这段代码中，外层函数负责返回内嵌函数，内嵌函数负责对外层函数的参数进行处理，返回新的值。

这实际上就是传入一个函数，处理函数的返回值，再传出一个函数。

这和我们想要的解决方案已经很相像了。

以秒的函数为例，我们希望把这个获取1位秒数返回值的函数，变成获取双位秒数返回值的函数。

那么，这段代码怎么用呢？

示例代码：（错误示例）

```
1 print(format_time(get_hours)(), format_time(get_minutes)(), format_time(get_seconds)(),
      sep=':')
```

这特么有点尴尬，跟刚才没什么区别，还是要修改所有客户代码。

打个比方，我们和女朋友约会。

我们肯定希望女朋友化完妆再约会，而不是约会时候还要化妆。

刚才的代码，就很像这个场景。

format_time()函数就是化妆，参数fun是自己的女朋友。

我们希望print语句中只有format_time()函数中的参数，而不希望看到format_time()函数的出现。

实际上，format_time()函数就是一个装饰器。

我们怎么能够让它不出现，还能够得到它的装饰效果？

我们使用装饰符 “@” 。

示例代码：

```
1 import time
```

```

2
3 def format_time(fun): # 装饰器
4     def new_time():
5         return ('0' + str(fun()))[-2:]
6     return new_time
7
8 @format_time # 为函数指定装饰器
9 def get_hours():
10     return time.localtime()[3]
11
12 @format_time # 为函数指定装饰器
13 def get_minutes():
14     return time.localtime()[4]
15
16 @format_time # 为函数指定装饰器
17 def get_seconds():
18     return time.localtime()[5]
19
20 print(get_hours(), get_minutes(), get_seconds(), sep=':')

```

上方代码中，时、分、秒的函数之前都加了一句“@format_time”。

这就是声明在调用这个函数的时候，要使用哪个装饰器进行处理，并得到处理后的结果。

是不是很简单，就像约会之前@自己的女朋友（被装饰的函数）化妆（装饰器）后再来。

另外，装饰器也能够而外接收参数。

我们再来看个例子，对计算合计的函数添加货币符号。

```

1 def add_symbol(symbol): # 获取装饰器参数
2     def dec_function(fun): # 被装饰函数传入
3         def new_total(price, count): # 获取被装饰函数的参数
4             return symbol + str(fun(price, count)) # 对被装饰函数进行处理，并返回结果。
5         return new_total # 返回装饰后的函数
6     return dec_function # 返回装饰后的函数
7
8 @add_symbol('¥')
9 def total(price, count):
10     return price * count
11
12 print(total(2.5, 3)) # 显示输出结果为：¥7.5

```

在上方代码中，大家能够看到，装饰器的嵌套函数变成了三层。

最外层的函数是用于接收货币符号的参数，中间层函数用于接收被装饰的函数，最内层函数用于装饰处理。

这里要注意，最内层函数的参数与被装饰函数的参数一致。

以上是关于自定义的装饰器。

在Python中，也有内置的装饰器，例如，我们之前接触的属性property。

在之前的教程《[Python3萌新入门笔记 \(31\)](#)》中，我们通过代码 “cubage = property(get_cubage,set_cubage)” 定义了体积这个特性。

其实，我们也可以通过装饰器来完成这个操作。

示例代码：

```
1 @property # 将方法装饰为属性
2 def cubage(self):
3     return self.length, self.width, self.height
4
5 @cubage.setter # 将方法装饰为cubage属性的setter方法
6 def cubage(self, tup):
7     if len([i for i in tup if isinstance(i, (int, float))]) == 3: #isinstance(,) 验证
        参数类型
8         self.length, self.width, self.height = tup
9     else:
10         raise TypeError
```

除了@property 这个装饰器，还有两个内置的装饰器@staticmethod和@classmethod。

先来看一段代码。

```
1 class MyClass:
2     def sm(): # 静态方法没有self参数
3         print('静态方法...')
4
5     def cm(cls): # 类成员方法带有cls参数
6         print('类成员方法...')
7
8 MyClass.sm() # 类直接调用静态方法，显示输出结果：静态方法...
9 MyClass().sm() # 类的实例调用静态方法，抛出异常，无法调用。
10 MyClass.cm() # 类直接调用类成员方法，抛出异常，无法调用。
11 MyClass().cm() # 类的实例调用类成员方法，显示输出结果：类成员方法...
```

为上方代码中的两个方法指定装饰器。

```
1 class MyClass:
2     @staticmethod
3     def sm(): # 静态方法没有self参数
4         print('静态方法...')
5
6     @classmethod
7     def cm(cls): # 类成员方法带有cls参数
8         print('类成员方法...')
```

通过指定装饰器，静态方法和类成员方法就都能够被类和实例访问了。

包和模块

一、包

包比较简单，可以理解成为包含模块的文件夹。

不过，并不是只把模块放入文件夹，就形成了包。

在包的目录中，需要添加一个名“__init__.py”的文件，这个文件可以创建一个空的文本文档，修改成这个文件名就可以了。

例如，我们在PyCharm中创建了一个项目，这个项目的文件都在同一个文件夹中（D:\MyProject）。

首先，点击文件（File）-新建项目（New Project...）创建新的项目。

然后，选择路径，并填写要创建的文件夹名称，点击确定（OK）保存，继续点击创建（create）按钮进行创建。。

最后，选择打开方式，这里选择的是当前窗口打开。继续点击确定（create）按钮打开项目。

这个时候，在我们创建项目的本地磁盘中就会生成项目的文件夹。

接下来，我们在项目文件夹中创建一个新的文件夹，例如名称为“package”。

然后，在这个文件夹中创建“__init__.py”文件，再放入一个写好代码的py文件（也可以通过文本文档创建），例如“sayhello.py”。

示例代码：（sayhello.py）

```
1 def hello():
2     print('你好!帅帅的小楼!')
3 hello()
```

完成上面的步骤，回到PyCharm中，我们就可以使用这个包。

我们在项目名称上点击鼠标右键，选择新建（New）Python文件（Python File）。

注意：包的创建也可以在PyCharm中完成（见上图）。

为这个新建的文件取一个名称，例如：case。点击确定（OK）完成创建。

然后，在这个新建的case模块中，我们就可以导入包和包的模块了。

用法有下面这几种：

```
1 from package import sayhello # 从包中导入模块
2 sayhello.hello() # 通过模块调用方法
3
4 from package.sayhello import hello # 从包的模块中导入方法
5 hello() # 直接调用方法
6
7 import package # 导入包
8 package.sayhello.hello() # 通过包调用模块再调用方法
9
10 import package.sayhello as say # 导入包的模块对象到变量
11 say.hello() # 通过模块对象调用方法
```

以上就是包的创建和使用。

当一个项目有大量的模块，这个时候往往需要通过包将关联性比较高或者同类的模块归纳到一起，方便模块的查找调用。

二、模块

1、模块的运行

在前面我们已经了解，模块通过import语句或者from/import语句进行导入。

那么，导入模块会发生什么？

示例代码：

```
1 from package import sayhello # 显示输出结果为：你好!帅帅的小楼！
```

大家能够发现，sayhello模块中的hello()语句被执行了。

所以，模块就是程序，它会在导入时被运行（定义模块的命名空间）。

不过，模块只是在第一次导入时会被执行，这避免了模块互相调用时无限运行的问题。

```
1 from package import sayhello # 显示输出结果为：你好!帅帅的小楼！
2 from package import sayhello # 没有显示输出任何内容
```

2、有条件运行模块

继续上面的例子，我们能够看到hello()语句被执行，但是这个语句是我们用来测试模块中编写的函数的，在其它模块调用sayhello模块的时候，我们并不想让这个测试语句被执行。

怎么避免这些测试代码也被运行呢？

首先，我们先来看看一个模块的名称是什么？

通过下面的语句，我们就能看到模块的名称。

sayhello模块中运行的语句：

```
1 print(__name__) # 显示输出结果为: __main__
```

case模块中运行的语句：

```
1 print(sayhello.__name__) # 显示输出结果为: package.sayhello
```

很显然，同样是获取sayhello模块的名称，在调用sayhello模块的模块中显示的是“package.sayhello”（包和模块的名称），而在sayhello模块中显示的是“__main__”。

知道了这个特点，我们就可以对测试代码添加运行的限制。

在sayhello模块中，写入以下代码：

```
1 def hello():
2     print('你好!帅帅的小楼!')
3
4 if __name__ == '__main__': # 判断测试语句是否在模块自身运行
5     hello() # 测试语句，显示输出结果为: 你好!帅帅的小楼!
```

通过条件判断，我们就完成了对测试语句运行的限制。

```
1 from package import sayhello # 没有显示输出任何内容
```

3、模块的优点

模块的优点在于，它可以是抽象的，能够被复用的。

这就像一个高富帅（功能代码），如果结婚了（和其他代码写在一起），他的功能（钱、车、小丁丁...）都只能被老婆使用（写在一起的代码）；但是如果他是单身（独立为模块），那么它的功能（钱、车、小丁丁...）可以被任何一个想和他啪啪啪（导入模块）的妹子们（所有使用模块的代码）使用。

所以，我们应该把编写的程序中可以抽象的部分独立为模块，这样会很大程度上提升工作效率，也能够让程序变得更加清晰、简洁。

4、模块的位置

当我们在PyCharm中创建了模块，这些模块会保存到项目的文件夹中，我们可以直接通过import进行导入使用。

那么，如果是一些外来的模块，怎么进行使用呢？

有三种使用办法：

1、把模块添加到当前项目的文件夹中（使用PyCharm这样的开发环境可以这么做）。

例如，我们把一个模块放在我们前面创建的MyProject文件夹中（D:\MyProject），就能够在case模块中导入。

2、把模块添加到Python解释器查找可用模块时，默认被查询的文件夹中。

Python查找可用模块时，会查找哪些文件夹呢？

我们可以查询一下这些文件夹的路径。

示例代码：

```
1 import sys, pprint # 导入内置的sys和pprint模块
2 pprint.pprint(sys.path) # 显示输出系统路径
```

注意：上方代码中pprint()函数，可以让路径列表中的内容自动换行显示输出。如果使用print()函数都会在一行显示。

代码运行结果如下：

上方运行结果中的文件夹路径，都是Python解释器会查询的文件夹路径，外来的模块放在这些路径中就能够直接导入使用，但是大家能够看到有一个路径结尾的文件夹名称是“site-packages”，建议把外来模块放入到这个文件夹中。

3、在程序中指定需要使用模块的路径。

我们可以在代码中，手动添加外来模块的文件夹路径，到Python解释器默认查询的文件夹路径列表中，从而保证能够正常导入模块。

假设：我们在“D:\Other”的文件夹中有一个“mod.py”模块。里面加入一条语句：print(r' 我是“D:\Other”中的模块！')

示例代码：（Windows系统）

```
1 import sys
2 sys.path.append('D:\\Other') # 将外来模块所在文件夹路径添加到默认查询的文件夹路径列表
3 import mod # 导入模块，显示输出结果为：我是“D:\Other”中的模块！
```

注意：UNIX/Mac OS X系统中要写入文件的完整路径。

以上就是使用外来模块的方法。

另外，当运行了上方代码之后，大家打开外来模块所在的文件夹，会看到一个名为“__pycache__”的文件夹，这是一个缓存文件夹，里面有后缀为“.pyc”的文件，从文件名我们可以看出这个文件和“mod.py”文件有关联。

没错！这个文件是“mod.py”编译后的文件。

当我们重复导入（第一次之后）模块“mod.py”的时候，实际上导入的是这个“.pyc”的文件。

这个文件可以删除，对程序没有任何影响，当导入模块时，还会自动生成。
并且，当我们修改模块，这个文件也会即时刷新。

Python的内置模块

内置模块中，往往包含了很多内容。

在接触具体的模块之前，我们先来看一下，怎么知道一个模块中都包含什么？有什么内容可以供我们使用？这些内容又如何使用？

- 使用内置函数dir()查看模块的组成

内置函数dir()可以帮助我们查看一个对象的组成，那么，在Python中万物皆对象，当然也可以使用这个函数查看模块的组成。

示例代码：

```
1 print(dir(pprint))
2 # 显示输出结果为: ['PrettyPrinter', '_StringIO', '__all__', '__builtins__',
  '__cached__', '__doc__',.....]
```

显示输出的结果很长，所以做了省略。

这个结果就是模块中包含的所有特性（函数、类、变量等）全部列出。

但是，并不是这些内容，我们都能够导入使用。

大家应该还记得，特性名称之前带下划线是表示警告，禁止使用者对这些特性进行操作。

那么，怎么知道哪些特性是允许使用者使用的呢？

- 使用模块特性__all__查看可用的模块内容

如果模块的特性中有一个__all__的特性（不是所有的模块都有，例如math模块就没有这个特性。），我们可以通过这个特性查看模块中所有可用的特性。

通过__all__特性获取的内容是所有可用特性名称的列表。

示例代码：

```
1 print(pprint.__all__)
2 # 显示输出结果为: ['pprint', 'pformat', 'isreadable', 'isrecursive', 'saferepr',
  'PrettyPrinter']
```

这次显示输出的结果就少了许多，这个列表中的特性，我们都可以调用。

不过，紧接着又面临一个问题，怎么使用？

- 使用内置函数help()查看模块帮助信息

还是以pprint模块举例。

如果想查看模块的帮助信息，我们可以通过语句help(pprint) 进行获取，并通过print()函数显示输出获取的结果。

这个结果里面会包含：名称（功能）、描述、类、函数、数据、文件的信息。

不过，这信息太多了，我们可以单独获取模块某一个特性的信息。

以上一段示例代码显示输出结果的第一个列表元素“pprint”为例。

如果要查看某个模块的特性，我们可以在参数中输入“模块.特性”。

提示：把“.”读成“的”试试看，是不是更容易理解？

所以，我们想获取pprint模块中pprint特性的帮助信息，就在参数中输入“pprint.pprint”。

示例代码：

```
1 print(help(pprint.pprint)) # help(pprint)能够获取模块所有特性的帮助信息
```

运行代码之后，会得到下方这些信息。

Help on function pprint in module pprint: # pprint模块中pprint函数的帮助信息

pprint(object, stream=None, indent=1, width=80, depth=None, *, compact=False) # 函数的参数信息

Pretty-print a Python object to a stream [default is sys.stdout]. # 漂亮打印输出一个Python对象到一个流[默认为系统标准输出]

是不是需要一些英文功底？

这是肯定的。

不过，这件事情急不来，学习需要一个过程。

就好像一开始都进不了洞，慢慢的不但能一杆进洞，还能领悟很多花样姿势。（多打台球有益健康）

- 使用__doc__特性查看模块说明文档

除了使用内置函数help()能够查看模块的帮助信息，通过模块的__doc__（前提是有这个特性）也能够查看。

示例代码：

```
1 print(pprint.pprint.__doc__)
```

代码运行结果：

Pretty-print a Python object to a stream [default is sys.stdout].

使用__doc__特性能够查看到模块或模块特性相应的文档，通过内置函数help()获取帮助信息时，文档部分的内容就来自这里。

- 使用__file__特性查看模块所在的位置

除了了解模块的组成与用途，有时候我们还想看看模块中的源代码，从而深入学习或参考。

那么，我们可以通过模块的__file__特性来查看模块所在的目录。

示例代码：

```
1 print(pprint.__file__)
2 # 显示输出结果为：
   C:\Users\Administrator\AppData\Local\Programs\Python\Python36\lib\pprint.py
```

同样，如果模块有__file__特性，我们可以这样查找，如果没有的话会提示错误。

例如，sys模块就不包含__file__特性，因为这个模块是包含在Python解释器中，而不是某个模块文件中。

AttributeError: module 'sys' has no attribute '__file__'

特性错误：模块 “sys” 不包含 “__file__” 特性。

如果使用的是PyCharm这个集成开发环境，我们可以查看这个模块。

在代码编辑区中先导入模块，然后，在模块名称上点击鼠标右键，选择去往（Go To）选项中的实现（Implementation）选项，就能够打开这个模块文件。

那么，既然能够打开这个模块，它不就应该有这个py文件吗？

大家可以看PyCharm软件的顶部，这里显示了当前模块所在的路径。

并且，在这个路径上，我们可以在模块前方的文件夹名称上点击鼠标右键，选择复制路径（Copy Path）的选项。然后，在本地资源管理器窗口中粘贴这个路径，打开文件夹，就能够看到当前的模块文件。

不过，虽然我们能够找到这个模块，但是它的里面看不到关键的源代码，这些源代码是融入到Python解释器中的。

具体的内置模块

因为，内置模块数量庞大，而且还会有更新、添加，所以，我们只了解一些比较常用、重要的模块。

一、sys模块

1、argv：用于程序外部向程序内部传递参数，例如在命令行中打开py文件时填入的参数。

示例代码：

```
1 import sys
2
3 if '--' in sys.argv[1]:
4     if sys.argv[1]=='--a':
5         print('接收到参数外部参数a...')
6     else:
7         print('模块"%s"不支持参数"%s"...' % (sys.argv[0],sys.argv[1]))
```

PyCharm中点击下方的Terminal打开命令行窗口。（通过Windows运行对话框中输入“cmd” 打开）

提示：通过“cmd”方式打开的窗口，需要通过“cd 项目文件夹路径”进入项目文件夹，这样才能找到自建的模块。

D:\MyProject>case.py -a

<module 'sys' (built-in)>

接收到参数外部参数a...

D:\MyProject>case.py -b

<module 'sys' (built-in)>

模块“ D:\MyProject\case.py” 不支持参数“ -b” ...

通过测试，我们能够看到，argv变量中包含两个元素，分别是模块名称（包含路径）和传入的参数。

我们可以在程序中，根据获取的参数，进行不同的处理。

2、exit(status)：用于中途退出程序，并可传入整数参数。

一般来说，程序执行到末尾时，会自动退出。

如果需要中途退出，可以使用`exit()`函数，并且这个函数可以传入整数参数（0表示正常退出，其它表示异常退出。），被调用的程序捕获。

示例代码：

```
1 import sys
2
3 def exitfunc(status): # 定义处理异常的函数，并接收传入的异常信息。
4     if not status: # 判断异常信息，进行不同的处理。
5         print('程序正常关闭! ')
6     else:
7         print('程序发生异常! ')
8     sys.exit(0) # 正常结束程序
9
10 if __name__ == '__main__': # 测试代码
11     print('----- 程序开始运行... -----')
12     try:
13         sys.exit(1) # 设定程序异常状态退出
14     except SystemExit as e: # 捕获到系统退出异常，并将异常信息存入e变量。
15         exitfunc(e) # 调用处理异常函数，并传入异常信息。
16     finally:
17         print('----- 程序运行结束... -----') # 无论是否发生异常都要执行的语句块
18     print("这里是程序末尾...") # 程序中途退出，此语句不会被执行。
```

以上代码运行结果为：

----- 程序开始运行... -----

程序发生异常!

----- 程序运行结束... -----

3、`path`：用于获取指定模块搜索路径的字符串列表。

在上一篇教程中，已经有具体的使用，此处不再演示。

4、`modules`：此变量是一个全局字典，键中保存了程序导入的所有模块的名称，值中保存了模块对应的存储路径。

```
1 import sys, pprint
2 pprint.pprint(sys.modules)
```

以上代码运行结果为：

```
{ '__main__' : <module ' __main__ ' from 'D:/MyProject/case.py' >,
  '_codecs' : <module ' _codecs ' (built-in)>,
```

'_collections' : <module '_collections' (built-in)>,
.....(省略部分内容)}

5、stdin\stdout\stderr: 这三个变量分别是标准输入/输出/错误对应的流对象。（暂时无需过多了解）

二、os模块

注意：以下内容os模块中的部分常用变量和函数，并未全部列出，并且部分函数仅列出主要参数。

1、os.sep: 操作系统路径中的分隔符。

2、os.pathsep: 操作系统中分隔路径的分隔符。

3、os.linesep: 操作系统中行的分隔符。

- \r: return, 表示回车, 移动到行的最左边。
- \n: newline, 表示向下移动一行, 并不移动左右。
- Linux系统中 "\n" 表示回车+换行。
- Mac系统中 "\r" 表示回车+换行。
- Windows中 "\r\n" 表示回车+换行。

4、os.name: 操作系统的类型, Windows系统为 "nt", Linux/Unix (包含Mac系统) 系统是 "posix" 。

5、os.getcwd(): 获取当前的工作路径。

6、os.system(command): 执行操作系统命令, 例如: os.system('dir'), 可以执行 "dir" 命令, 查看项目文件夹的目录构成。也可打开指定路径的外部程序, 打开外部程序时, 如果路径名称包含空格, 则需要将路径名称放置在双引号中。例如: os.system(r' C:\ " Program Files (x86)" \Tencent\QQ\Bin\QQ.exe')

7、os.remove(path): 删除指定路径的文件。

8、os.stat(path): 获取文件状态 (属性) 。

9、os.chmod(path): 修改文件的权限和时间戳。

10、os.mkdir(path): 创建新的目录。

11、os.rmdir(path): 删除指定目录。

12、os.removedirs(path): 删除多个目录。

13、os.listdir(path): 列出指定路径下的目录和文件。

14、os.curdir: 当前目录。

15、os.pardir: 上一级目录。

16、os.chdir(path): 改变工作目录到指定路径。

17、os._exit(int): 终止当前进程。

18、os.path.split(path): 获取一个路径的目录路径和文件名称。

19、os.path.isfile(path): 检查指定的路径是否为文件。

20、os.path.isdir(args,kwargs): 检查指定的路径是否为目录。

21、os.path.exists(path): 检查指定的路径是否存在。

22、os.path.getsize(filename): 获取指定文件的大小, 如果参数是目录返回0。

23、os.path.abspath(filename): 获取指定文件的绝对路径。

24、os.path.isabs(): 检查是否为绝对路径。

25、os.path.normpath(path): 规范路径字符串。

26、os.path.splitext(): 获取路径中文件的名称和扩展名。

27、os.path.join(path,paths): 连接目录与文件名或目录。

28、os.path.basename(path): 获取路径中的文件名称。

29、os.path.dirname(path): 获取路径中的目录部分。

30、os.startfile(filepath,operation): 执行指定路径的外部程序; 参数operation未指定或指定为“open”时, 相当于双击资源管理器中的文件, 或DOS中为文件名添加的一个参数。

三、fileinput模块

在接触新的模块之前, 我们先来尝试通过Python内置的函数open进行文件的读写。

1、文件的读取

提示: open()函数的具体说明参考《[Python3萌新入门笔记 \(16\)](#)》。

示例代码:

```
1 path = r'C:\Users\Administrator\Desktop\song\lyric.txt' # 设置读取文件的路径
2 file = open(path, 'r') # 打开文件存入变量, 'r' (read) 为只读模式。
3 while True: # 循环
4     line = file.readline() # 读取一行内容存入变量
5     if line: # 如果变量不是空值 (末尾行为空值)
6         print(line, end='') # 显示输出读取结果, 设置不换行输出 (文本的每一行都带有换行符)。
7     else: # 否则
8         break # 结束循环
9 file.close() # 关闭文件
```

在上方代码中, 我们通过本地路径读取了一个文本文件, 并将内容逐行显示输出。

需要注意的是, 读取结束后要通过close()方法, 将文件对象关闭。

不过, 关闭文件对象的操作, 很容易被忘记, 并且, 通过while循环读取文件的每一行感觉很麻烦。

接下来, 我们通过for循环来读取文件内容, 并且结合关键字with打开文件。

使用关键字with进行文件的打开操作, 文件对象会在语句块执行结束后自动关闭。

示例代码:

```
1 path = r'C:\Users\Administrator\Desktop\song\lyric.txt' # 设置读取文件的路径
2 with open(path, 'r') as file: # 结合with关键字打开文件并存入变量
3     for line in file: # 如果变量不是空值 (末尾行为空值)
4         print(line.replace('\n', '')) # 显示输出读取结果, 替换掉每行内容末尾的换行符。
```

2、文件的创建\清空\写入

打开文件时, 指定模式为'w' (write), 如果文件不存在, 则会创建; 如果文件存在, 则会打开后清空。

示例代码:

```
1 path = r'C:\Users\Administrator\Desktop\song\test.txt'
2 with open(path, 'w') as file: # 创建文件或清空文件内容
3     file.write('小楼真的好帅好帅的!') # 写入内容
```

注意：写入内容时，如果需要换行需要显式的加入换行符。

3、文件的追加

打开文件时，指定模式为' a' （append），就能够在文件末尾追加内容；如果文件不存在，则会创建。

示例代码：

```
1 path = r'C:\Users\Administrator\Desktop\song\lyric.txt'
2 with open(path, 'a') as file: # 文件追加内容
3     file.write('这是我很喜欢的一首歌！') # 文件内容末尾写入一行内容
```

4、覆盖

打开文件时，指定模式为' r+'，能够在开始位置写入新内容，覆盖同等字符数量的原内容。

注意：' +' 表示可对文件进行写入或读取，但不可单独出现。

示例代码：（覆盖）

```
1 path = r'C:\Users\Administrator\Desktop\song\test.txt' # 假设当前文件内容为“abcde”。
2 with open(path, 'r+') as file: # 设置打开模式为读写模式。
3     file.write('aaa') # 程序执行后，文件内容变更为“aaade”。
```

这种模式下，也能够实现追加。

示例代码：（追加）

```
1 path = r'C:\Users\Administrator\Desktop\song\test.txt' # 假设当前文件内容为“abcde”。
2 with open(path, 'r+') as file: # 设置打开模式为读写模式。
3     file.read() # 读取至末尾
4     file.write('aaa') # 此时从读取后的位置写入内容，程序执行一次后，文件内容变更为
    “abcdeaaa”。
```

上方代码中的read()方法，参数为整数，能够读取指定数量的字符，省略参数时，读取全部文件内容，并将读写指针停留在文件内容的末尾。

write()方法从读写指针所在的位置开始写入新的内容。

5、可读追加

打开文件时，指定模式为' a+'，能够在追加内容之后读取文件内容。

示例代码：

```
1 path = r'C:\Users\Administrator\Desktop\song\test.txt' # 假设当前文件内容为“abcde”。
```

```

2 with open(path, 'a+') as file: # 设置打开模式为读写模式。
3     file.write('aaa') # 在文件末尾写入内容，程序执行一次后，文件内容变更为“abcdeaaa”。
4     file.flush() # 刷新缓存，将缓存内容立即写入文件。
5     file.seek(0) # 移动文件读取指针到指定位置，0为文件开始位置。
6     print(file.read()) # 显示输出结果为：abcdeaaa

```

注意：在文件操作过程中（文件未关闭），写入的内容并未真正的写入到文件，而是保存在缓存中；所以，需要使用flush()方法刷新缓存，将文件更新，才能够读取到文件内容。

在上方代码中，除了刷新缓存的函数，还有seek()方法，用于将读写指针移动到指定的位置。这里将读写指针移动回文件开始位置（因为写入内容后停留在末尾的位置），从头进行读取。

6、可读写入

打开文件时，指定模式为‘w+’，能够在写入文件内容之后读取文件内容。

示例代码：

```

1 path = r'C:\Users\Administrator\Desktop\song\test.txt' # 假设当前文件内容为“abcde”。
2 with open(path, 'w+') as file: # 打开并清空文件
3     file.write('aaa') # 写入内容，程序执行后，文件内容变更为“aaa”。
4     file.flush()
5     file.seek(0)
6     print(file.read()) # 显示输出结果为：aaa

```

接下来，我们一起来看一下fileinput模块包含的内容。

示例代码：

```

1 import fileinput
2 print(fileinput.__all__)

```

通过模块的__all__特性，我们得到以下结果：

['input' , 'close' , 'nextfile' , 'filename' , 'lineno' , 'filelineno' , 'fileno' , 'isfirstline' , 'isstdin' , 'FileInput' , 'hook_compressed' , 'hook_encoded']

- input(): 获取包含多个文件的可迭代对象
- filename(): 获取当前读取文件的名称
- fileno(): 获取文件编号
- lineno(): 获取当前已经读取总行数
- filelineno(): 获取当前读取的行的行号
- isfirstline(): 检查是否当前所读取文件的第一行
- isstdin(): 检查是否从标准输入中读取的行
- nextfile(): 读取下一个文件

- close(): 关闭读取的文件对象
- hook_compressed: 用于读写.gz .bz2 文件
- hook_encoded(encoding): 用于处理文件编码

读取文件往往不是简单的读取与写入，fileinput模块提供的功能，能够让我们更灵活的进行文件读取。

下面这个例子，是对某一文件夹中的全部文件进行读取，并显示输出读取结果。

通过这个例子，大家可以对fileinput模块中的常用方法进行了了解。

示例代码：

```
1 import os, fileinput # 导入需要使用的模块
2
3 path = r'C:\Users\Administrator\Desktop\song' + '\\ ' # 设置查询的目标文件夹路径
4 names = os.listdir(path) # 获取到目标文件夹下所有的文件名称
5 paths = [] # 创建路径列表
6 for name in names: # 循环遍历所有的文件名称
7     paths.append(path + name) # 为路径列表添加文件路径
8
9 files = fileinput.input(paths) # 获取目标文件夹中所有文件
10 for line in files: # 遍历所有文件
11     all_no = fileinput.lineno() # 获取当前已被读取的总行数
12     line_no = fileinput.filelineno() # 获取当前文件被读取行的行号
13     if line_no == 10: # 如果当前文件读取到第10行
14         fileinput.nextfile() # 读取下一个文件
15     if fileinput.isfirstline(): # 检查是否第一行
16         print(fileinput.filename()) # 显示输出当前读取的文件名称
17     if all_no >= 15: # 如果已被读取的总行数达到15行
18         fileinput.close() # 关闭当前读取的文件
19     print(all_no, line_no, line.replace('\n', '')) # 显示输出已读取的总行数、当前文件被读
    取行的行号和当前读取的行内容
```

代码运行结果：

示例相关文件下载：[【点击下载】](#)

在fileinput模块中，input()函数是最重要的函数，它有很多参数。

fileinput.input(files=None, inplace=False, backup="", bufsize=0, mode=' r' , openhook=None)

- files: 文件或文件列表
- inplace: 指定是否将标准输出结果写入文件，默认为False（不写入）。
- backup: 指定备份文件的扩展名，默认为空值（不备份）。
- bufsize: 指定缓冲区大小，默认为0（不缓冲）。
- mode: 指定读写模式，默认为' r' （只读）。

- openhook: 控制打开的全部文件, 例如编码类型, 默认为None。

四、glob模块

在上一篇教程中, 我们通过os模块获取文件夹中的所有文件名, 然后组织成多个文件的路径列表, 通过fileinput模块进行遍历。

这样的操作比较麻烦。

实际上, 我们可以使用glob模块中的函数帮我们简便的实现这个需求。

glob模块支持通配符 “*”、 “?”、和 “[]”, “*” 表示0个或多个字符, “?” 表示一个字符, “[]” 表示匹配指定范围内的字符, 例如 “[0-9]” 匹配数字0-9。

另外, glob模块还支持通配符 “**”, 能够获取一个目录下所有的目录以及目录中的文件。

glob模块的内容很少, 包括3个函数。

- glob(pathname, recursive=False): 获取所有匹配路径的列表。参数pathname为需要匹配的路径字符串; 参数recursive为递归, 调用递归时 (值为True), 需配合通配符 “**” 使用, 默认为False (不调用递归)。
- iglob(pathname, recursive=False): 获取所有匹配路径的生成器, 参数同上。
- escape(pathname): 忽略所有通配符。

接下来, 我们通过glob模块, 简化之前的示例代码。

示例代码:

```
1 import fileinput,glob
2
3 path = r'C:\Users\Administrator\Desktop\song\*.txt' # 设置查询的目标文件夹路径的所有txt文件
4 files = fileinput.input(glob.glob(path)) # 获取目标文件夹中所有文件
5 for line in files:
6     (以下代码省略)
```

五、heapq模块

heap (堆) 是一种数据结构。

普通的队列是顺序排列。

例如: 列表[3,2,5,7,1,9,8,6,0,4]进行由小到大的排列就是[0,1,2,3,4,5,6,7,8,9]。

如果我们把这个列表转为带有堆属性的列表 (以下简称: 堆列表), 结果是: [0,1,5,2,3,9,8,6,7,4]

这个堆列表看上去是无序的, 实际上是按优先级进行排列。

所以, 堆的排列是有规则的排列。

那么, 堆的排列规则是怎样的呢?

首先, 我们要了解堆的结构, **堆是完全二叉树的结构。**

列表[3,2,5,7,1,9,8,6,0,4]转换为堆列表时, 初始结构如下图:

这个结构从上至下, 从左至右的获取元素, 就是列表中的顺序。

图中的结构, 就像倒过来的树 (我看更像树根), 每一个节点最多有两个分支, 所以是完全二叉树的结构。

如果不把这个结构看成树, 它也像将多个元素堆在了一起, 所以它叫堆。

概念搞清楚, 就不会那么迷茫了。

接下来，我们看它是按什么规则重新排序的。

我们堆东西的时候，假设每个物品都有承重上限，为了不把物品压破，我们需要把承重能力小的摆放在上方。

假设列表中的元素数值就是每个元素的承重上限，在已有结构上怎么调整才不会有物品被压破呢？

我们可以按照从下往上，从左往右的顺序进行调整。

每个上层物品和它下层的两个物品比较承重上限，取承重上限最小的放在上层。

也就是说，如果承重最小的物品在下层的话，就和上层的物品位置互换。

那么，在刚才堆的基础之上，我们进行调整，过程如下：

注意，由下至上完成一次调整之后，可能还会存在不符合要求的排列（图中第4步），还要从有问题的位置由下至上继续调整，直到所有排列都符合要求为止。

通过这个调整过程，我们就能够看出堆的排列规则。

这种排列规则可以表示为： $\text{heap}[\text{index}] \leq \text{heap}[2*\text{index}+1]$ and $\text{heap}[\text{index}] \leq \text{heap}[2*\text{index}+2]$

也就是每一个父节点的数值小于等于两个子节点的数值。

因为，堆的这种优先规则，在获取一个列表的最小值时（可以增加元素的同时找到最小元素），要比列表的`min()`函数更加快捷。

Python中没有独立的堆类型，只是在内置模块`heapq`中提供了一些操作堆的函数。

`heapify(list)`：将列表转换堆列表。

`heappush(heap,item)`：执行将元素入堆的操作，`heap`为已有堆列表，`item`为新的入堆元素。

`heappop(heap)`：取出堆中最小元素。

`heapreplace(heap,item)`：替换堆列表中最小的元素为新元素。

`heappushpop(heap,item)`：新元素入堆，然后取出堆列表中最小的元素。

`nlargest(n, iterable, key=None)`：获取最大的`n`个元素。

`nsmallest(n, iterable, key=None)`：获取最小的`n`个元素。

`merge(*iterables)`：将多个列表合并，并进行堆调整，返回的是合并后的列表的迭代器。

接下来，通过示例代码演示上方函数的使用方法。

示例代码：

```
1 import heapq # 导入需要使用的模块
2
3 heap = [3, 2, 5, 7, 1, 9, 8, 6, 0, 4]
4 heapq.heapify(heap) # 列表转换为堆列表
5 print(heap) # 显示输出结果为: [0, 1, 5, 2, 3, 9, 8, 6, 7, 4]
6 heapq.heappush(heap, 0.5) # 新元素入堆
7 print(heap) # 显示输出结果为: [0, 0.5, 5, 2, 1, 9, 8, 6, 7, 4, 3]
8 print(heapq.heappop(heap)) # 取出堆列表中最小的元素，显示输出结果为: 0
9 print(heap) # 显示输出结果为: [0.5, 1, 5, 2, 3, 9, 8, 6, 7, 4]
10 heapq.heapreplace(heap, 10) # 替换堆列表中最小的元素为新元素
11 print(heap) # 显示输出结果为: [1, 2, 5, 6, 3, 9, 8, 10, 7, 4]
12 print(heapq.heappushpop(heap, 11)) # 新元素入堆，然后取出堆列表中最小的元素，显示输出结果
    为: 1
```

```
13 print(heap) # 显示输出结果为: [2, 3, 5, 6, 4, 9, 8, 10, 7, 11]
14 print(heapq.nlargest(5, heap)) # 显示输出结果为: [11, 10, 9, 8, 7]
15 print(heapq.nsmallest(5, heap)) # 显示输出结果为: [2, 3, 4, 5, 6]
16 print(list(heapq.merge([1, 4, 7], [0, 2, 5]))) # 显示输出结果为: [0, 1, 2, 4, 5, 7]
```

六、collections模块

这个模块中，我们只了解一块内容，deque类型的使用。

deque (double ended queue) 是双端队列。

按官方文档的说法，这是一个类似列表的容器，能够在两端快速插入（好爽）和弹出元素。

具体的使用方法，通过示例代码就能够直观的了解。

示例代码：

```
1 from collections import deque
2
3 deq = deque((2, 3, 4, 5, 6)) # 序列转为双端
4 print(deq) # 显示输出结果为: deque([2, 3, 4, 5, 6])
5 deq.appendleft('开始') # 队列左侧添加单个元素
6 print(deq) # 显示输出结果为: deque(['开始', 2, 3, 4, 5, 6])
7 deq.append('结尾') # 队列右侧添加单个元素
8 print(deq) # 显示输出结果为: deque(['开始', 2, 3, 4, 5, 6, '结尾'])
9 print(deq.pop()) # 弹出右侧最后一个元素，显示输出结果为: 结尾
10 print(deq) # 显示输出结果为: deque(['开始', 2, 3, 4, 5, 6])
11 print(deq.popleft()) # 弹出左侧第一个元素，显示输出结果为: 开始
12 print(deq) # 显示输出结果为: deque([2, 3, 4, 5, 6])
13 deq.rotate(2) # 所有元素循环（末尾元素移动至最左侧）向右移动2个位置
14 print(deq) # 显示输出结果为: deque([5, 6, 2, 3, 4])
15 deq.rotate(-1) # 所有元素循环（首个元素移动至最右侧）向左移动1个位置
16 print(deq) # 显示输出结果为: deque([6, 2, 3, 4, 5])
17 deq.extend([7, 8]) # 逐一读取列表元素并增加到队列右侧
18 print(deq) # 显示输出结果为: deque([6, 2, 3, 4, 5, 7, 8])
19 deq.extendleft([1, 0]) # 逐一读取列表元素并增加到队列左侧
20 print(deq) # 显示输出结果为: deque([0, 1, 6, 2, 3, 4, 5, 7, 8])
```

除了上述的这些方法，还有一些其它比较通用的方法，例如：copy、count、remove.....等等，在此不做演示。

七、time模块

在编程开发中，往往需要获取系统日期时间，或者对日期时间对象进行一些操作。

这里，我们来了解一下time模块为我们提供的功能。

示例代码：

```
1 import time
2 print(time.time())
```

我们先导入time模块，然后调用time()函数，看一下获取到的内容。

获取到的内容类似：1508381851.557155

这是一个自1970年1月1日起到当前系统时间的秒数（小数点后方为毫秒、微秒和纳秒）。

那么，如何获取一个时间的字符串呢？

示例代码：

```
1 print(time.asctime()) # 显示输出结果类似于：Thu Oct 19 11:05:26 2017
```

对于国内用户来说，这个函数通常没什么卵用，我们更希望获取到具体的年、月、日、时、分、秒的数值，然后加上中文和冒号。

就像：2017年10月19日 12:01:25

是不是感觉有些熟悉？

没错，在教程《[Python3萌新入门笔记 \(34\)](#)》中我们使用过获取时间元组的函数。

示例代码：

```
1 print(time.localtime())
```

运行代码，我们能够看到一个时间的元组。

类似于：time.struct_time(tm_year=2017, tm_mon=10, tm_mday=19, tm_hour=11, tm_min=10, tm_sec=9, tm_wday=3, tm_yday=292, tm_isdst=0)

在本地时间的元组中包含的元素依次为：年,月,日,时,分,秒,星期几,本年第几天，是否夏令时（1为真，0为假，-1为未知）。

这个元组中的内容，我们就能够通过元组的索引取到了。

示例代码：

```
1 print(time.localtime()[0]) # 显示输出结果为：2017
```

既然毫秒能够变为时间的元组，那么，时间的元组也能转换为毫秒。

示例代码：

```
1 print(time.mktime((2017, 10, 19, 12, 0, 0, 0, 0, 0))) # 显示输出结果为：1508385600.0
```

mktime方法需要提供9个整数参数，这些参数和localtime()方法获取到的时间元组相一致。

接下来，有一个非常重要的方法，它用于让Python解释器等待一段时间。

示例代码：

```
1 for i in range(5):
2     print(time.time())
3     time.sleep(1)
```

运行代码，显示结果为：

1508383373.3071938

1508383374.3072512

1508383375.3073082

1508383376.3073654

1508383377.3074226

sleep()函数能够让解释器等待指定的秒数，所以上方循环获取系统时间的代码，输出的结果都间隔1秒钟（有一些微秒级的误差）。

另外，还有一个strptime()函数，能够将日期时间字符串转换为时间元组。

这个函数有两个参数，参数string为时间字符串，参数format是格式化字符串。

示例代码：

```
1 print(time.strptime('2017-10-19 14:01:23', '%Y-%m-%d %H:%M:%S'))
```

运行代码，显示输出结果为：

time.struct_time(tm_year=2017, tm_mon=10, tm_mday=19, tm_hour=14, tm_min=1, tm_sec=23, tm_wday=3, tm_yday=292, tm_isdst=-1)

这里我们需要掌握的是格式化字符串中需要写的转换说明符。

- %y：两位数的年份（00~99）
- %Y：四位数的年份（0000~9999）
- %m：月份（1~12）
- %d：天数（1~31）
- %H 24小时制小时数（0~23）
- %I 12小时制小时数（1~12）
- %M 分钟数（0~59）
- %S 秒（0~61） # 没看错，因为有闰秒和双闰秒的存在。
- %a 本地简化英文星期名称
- %A 本地完整英文星期名称
- %b 本地简化英文月份名称
- %B 本地完整英文月份名称
- %c 标准日期时间

- %j 年积日 (1-366)
- %p 本地AM或PM的等价显示
- %U 年中第几周 (0-53, 以星期日为第一天)
- %w 星期一 (0) 至星期日 (6) 的数值 (0-6)
- %W 年中第几周 (0-53, 以星期一为第一天)
- %x 标准日期
- %X 标准时间
- %Z 当前时区英文名称
- %% %号

八、datetime模块

datetime模块也是一个用于操作日期时间的模块。

这个模块中也为我们提供了一些易用的功能。

例如，模块里面的datetime类，能够创建日期时间对象，也能够获取系统当前日期时间。

示例代码：

```
1 import datetime
2
3 date = datetime.datetime(2017, 10, 19) # 创建日期时间对象
4 print(date) # 显示输出结果为: 2017-10-19 00:00:00
5 now = datetime.datetime.now() # 获取系统当前日期时间
6 print(now) # 显示输出结果类似于: 2017-10-19 14:17:11.483406
7 print(now.year, now.month, now.day, now.hour, now.minute, now.second, now.weekday())
8 # 获取年、月、日、时、分、秒和周几, 显示输出结果类似于: 2017 10 19 14 17 11 3
```

在datetime类中还有today()方法，获取的是系统当前的日期时间对象。

示例代码：

```
1 date = datetime.datetime.today()
2 print(date) # 显示输出结果类似于: 2017-10-19 16:12:53.564471
```

而且，我们还可以通过datetime类中的timetuple()方法，获取日期时间对象的元组。

```
1 print(datetime.datetime.timetuple(date))
```

显示输出结果类似于：time.struct_time(tm_year=2017, tm_mon=10, tm_mday=19, tm_hour=16, tm_min=17, tm_sec=6, tm_wday=3, tm_yday=292, tm_isdst=-1)

另外，datetime模块还有一个date类。

示例代码：

```
1 print(datetime.date.today()) # 显示输出结果类似于：2017-10-19
```

而且，在datetime模块中还有timedelta，能够对日期时间对象进行计算。

```
1 date = datetime.datetime(2017, 10, 19) # 创建日期时间对象
2 print(date) # 显示输出结果为：2017-10-19 00:00:00
3 print(date.date()) # 获取日期时间对象中的日期，显示输出结果为：2017-10-19
4 print(date + datetime.timedelta(days=-1)) # 日期减少1，显示输出结果为：2017-10-18 00:00:00
5 print(date + datetime.timedelta(days=1)) # 日期增加1，显示输出结果为：2017-10-20 00:00:00
6 print(date + datetime.timedelta(hours=-1)) # 小时减少1，显示输出结果为：2017-10-18 23:00:00
7 print(date + datetime.timedelta(minutes=-1)) # 分钟减少1，显示输出结果为：2017-10-18 23:59:00
8 print(date + datetime.timedelta(seconds=-1)) # 秒数减少1，显示输出结果为：2017-10-18 23:59:59
```

datetime模块的功能并不仅仅这么多，其它功能大家可以查询官方文档了解。

九、math模块

math模块包含了一些数学计算的函数，例如平方根函数、正余弦函数、正余切函数、角度与弧度转换函数等等。不过，一般数学函数的用途比较少，这里只做一些简单的举例。

pi：保存了圆周率的变量

ceil(x)：向上取整函数，返回参数x向上相邻的整数，如果参数为整数，返回原值。

floor(x)：向下取整函数，返回参数x的整数部分，如果参数为整数，返回原值。

fsum(iterable)：求和函数，返回值为可迭代参数中的所有数值的和。

pow(x,y)：幂计算函数，返回参数x的y次方。

sqrt(x)：平方根计算函数，返回参数x的平方根。

示例代码：

```
1 import math
2
3 print(math.pi) # 圆周率，显示输出结果为：3.141592653589793
4 print(math.ceil(1.1)) # 向上取整，显示输出结果为：2
5 print(math.floor(1.9)) # 向下取整，显示输出结果为：1
6 print(math.fsum([1.2, 2.24, 5.58])) # 求和，显示输出结果为：9.02
```



```
7 print(math.pow(2,3)) # 幂计算，显示输出结果为：8.0
8 print(math.sqrt(4)) # 平方根将计算，显示输出结果为：2.0
```

十、random模块

random模块是关于随机数的模块。

随机数的应用比较常见，所以对这个模块的了解比较重要。

同样，我先把模块中的常用功能——列出。

random(): 返回值为0~1之间的随机小数。（长度为小数点后18位）

uniform(a, b): 返回值为大于等于参数a且小于参数b的随机小数。

randint(a, b): 返回值为大于等于参数a且小于等于参数b的随机整数。

randrange(start, stop[, step]): 返回值为一定范围内的随机整数。参数start为起始数值，参数stop为终止数值（返回值不包含终止数值），参数step为步长。只输入一个参数（stop）时，返回值为大于等于0且小于参数数值的随机整数。

choice(seq): 返回值为参数序列中的某一元素。

choices(population, weights=None, *, cum_weights=None, k=1): 返回值为参数k大小的元素列表，列表中元素可重复，列表元素来自于参数population。参数weights为相对权重列表，列表中的元素个数与参数population相同，参数weights中元素数值大小决定参数population中相同位置元素的权重（可以理解为随机概率），权重越高随机出现概率越大。参数cum_weights是累积权重列表，累积权重通过相对权重计算，例如相对权重[1,5,1]等同于累积权重[1,6,7]，即[(0+1),(1+5),(1+5+1)]。使用累积权重有更高的执行效率。

random.sample(population, k): 返回值为参数k大小的元素列表，列表中元素不重复，列表元素来自于参数population。

shuffle(x): 返回值为重新随机排序的序列，参数x为序列。如果对不可变序列重新随机排序，可以使用sample(x, k=len(x))。

getrandbits(k): 返回值为整数，返回值的二进制数值长度为k。例如，k为2时，返回值为0,1,2,3之一，即(00,01,10,11)之一。

示例代码：

```
1 import random
2
3 print(random.random()) # 显示输出结果为0~1之间的小数。（小数点后18位数字）
4
5 print(random.uniform(0, 10)) # 显示输出结果为大于等于0且小于10的随机小数。
6 print(random.randint(0, 10)) # 显示输出结果为大于等于0且小于等于10的随机整数。
7 print(random.randrange(0, 10, 2)) # 显示输出结果为0、2、4、6、8中的某个数字。
8
9 print(random.choice(['好帅', '好酷', '好厉害', '棒棒哒'])) # 显示输出结果为列表中某一元素。
10 print(random.choices([1, 2, 3], cum_weights=[1, 6, 7], k=2)) # 显示输出结果为基于首个参数列表的长度为2的随机列表。
11 print(random.choices([1, 2, 3], [1, 5, 1], k=2)) # 显示输出结果为基于首个参数列表的长度为2的随机列表。
```

```

12 print(random.sample([1, 2, 3, 4, 5], 2)) # 显示输出结果为基于首个参数列表的长度为2的随机列表。
13
14 lst = [1, 2, 3, 4, 5]
15 random.shuffle(lst) # 列表元素随机改变位置。
16 print(lst) # 显示输出结果为被随机重排位置的列表。
17
18 print(random.getrandbits(2)) # 显示输出结果为0~3的随机整数，参数为随机整数对应的二进制数字的最大位数。

```

接下来，我们通过随机数来完成获取4位包含字母与数字的随机验证码（字符可重复）。

示例代码：

```

1 code = ''.join(random.choices('abcdefghijklmnopqrstuvwxyz0123456789', k=4))
2 # 通过join()方法，将可迭代对象中的字符连接为字符串,这个方法通过字符串分隔符（' '为不添加分隔符）调用。
3 print(code) # 显示输出结果类似于：n22r

```

最后，说明一下，random模块所获取的随机数是伪随机数。

伪随机数不是假随机数的意思，伪随机数确实能够获取到每次不同的随机数值，能够满足我们一般获取随机数的需求。

但是，因为伪随机数是通过不断变化的随机数种子（例如：系统时间），通过特定的算法得到的随机数，因此，伪随机数具有一定的规律，它可以被预测。

如果想获取安全性更高的接近真正随机性的随机数，可以使用os模块中的urandom()函数或者当前模块中的SystemRandom类。

十一、pickle模块

pickle模块用于对象持久化，也就是将内存中的对象保存到可永久保存的存储设备中（例如：磁盘）。

pickle模块提供简单的持久化功能，包含两个主要函数（此处仅列出必填参数）：

dump(obj, file):将对象obj序列化并存储到文件对象file中。

load(file)：按dump的顺序，依次读取文件对象（file）中每次dump的内容。

示例代码：

```

1 import pickle
2
3 path = r'C:\Users\Administrator\Desktop\date.dat'
4 with open(path, 'wb+') as file: # 打开文件进行序列化操作
5     user = ['小楼', '男', '18', '帅']
6     learn = ['Python', 'Axure']
7     pickle.dump(user, file) # 数据倾出到文件

```

```

8     pickle.dump(learn, file) # 数据倾出到文件
9     user.append('很厉害')
10    pickle.dump(user, file) # 数据倾出到文件（不会覆盖上次倾出）
11
12    with open(path, 'rb+') as file: # 打开文件进行反序列化操作
13        print(pickle.load(file)) # 显示输出结果为: ['小楼', '男', '18', '帅']
14        print(pickle.load(file)) # 显示输出结果为: ['Python', 'Axure']
15        print(pickle.load(file)) # 显示输出结果为: ['小楼', '男', '18', '帅', '很厉害']
16        print(pickle.load(file)) # EOFError: Ran out of input (文件末端异常: 输入已耗尽)

```

十二、shelve模块

使用pickle模块虽然能够进行数据的持久化，但是大家也能够看到，dump操作可以进行多次，load操作却只能按照dump的顺序依次读取一次。

如果我们想要实现比较灵活的dump和load操作，可以使用shelve模块。

shelve模块对pickle模块进行了封装，可以持久化所有pickle所支持的数据类型。

通过shelve模块创建的文件存储的是一个字典。

这个模块只提供了一个open()函数，用于打开持久化的字典，进行数据的持久化和读取操作。

open(filename, flag='c', protocol=None, writeback=False): 参数filename为文件名或文件路径；参数flag可选值为'c'（create：不存在则创建）和'n'（new：新建空文件）；参数protocol为整数值，数值1、2可支持早期Python版本对文件的读取；参数writeback默认为False，即不将缓存写回打开的文件，当需要写回时，需要指定为True。

open()函数会创建一个DbfilenameShelf对象，对这个对象的操作和字典一样。

但是，要注意，键值必须是字符串。

示例代码：

```

1  import shelve
2
3  path = r'C:\Users\Administrator\Desktop\date'
4  with shelve.open(path, 'c', writeback=True) as dat: # 打开已有文件或创建新文件
5      dat['user'] = ['小楼', '男', '18', '帅'] # 数据添加字典
6      dat['learn'] = ['Python', 'Axure']
7      print(dat['user']) # 显示输出结果为: ['小楼', '男', '18', '帅']
8      print(dat['learn']) # 显示输出结果为: ['Python', 'Axure']
9      dat['user'].append('很厉害')
10     # 参数writeback为False时，append所做的修改不会被写回。print(dat['user'])的结果仍然为:
    ['小楼', '男', '18', '帅']
11     print(dat['user']) # 显示输出结果为: ['小楼', '男', '18', '帅', '很厉害']

```

十三、re模块

在《Python3萌新入门笔记 (23) 》中，我们接触过正则表达式的使用，并且使用了re模块中的match()函数和compile()函数。

示例代码：

```
1 import re
2
3 print(re.match(r'Python.', 'Python\n', re.DOTALL)) # 匹配结果为: <_sre.SRE_Match
object; span=(0, 7), match='Python\n'>
4 print(re.match(r'Python.', 'Python\r')) # 匹配结果为: <_sre.SRE_Match object; span=(0,
7), match='Python\r'>
5 print(re.match(r'Python.', 'Python! ')) # 匹配结果为: <_sre.SRE_Match object; span=(0,
7), match='Python! '>
6 print(re.match(r'Python.', 'Python\n')) # 匹配结果为: None
7
8 pat=re.compile(r'^[A-Z][0-9a-z]{5,}') # 预编译表达式：由字母与数字组成，首位为大写字母，长度
6位以上的字符串。
9 print(pat.match('Ab12345'))
10 print(pat.match('Ab123'))
11 print(pat.match('ab1234'))
12 print(pat.match('A@1234'))
```

除了上述两个函数，re模块中还有其它一些函数供我们调用（这里只标出主要参数）。

search(pattern, string)：从字符串参数string中查找与正则表达式pattern相匹配的子字符串，如果查找到匹配的子字符串，返回match对象，否则，返回None。

通过search()函数获取的match对象，可以通过group()方法获取其中的子字符串。

示例代码：

```
1 result = re.search(r'小楼.{2}', '帅帅的小楼哥哥！迷人的小楼欧巴！')
2 print(result) # 显示输出结果为: <_sre.SRE_Match object; span=(3, 8), match='小楼哥哥'>
3 print(result.group()) # 显示输出结果为: 小楼哥哥
4
5 print(re.search('楼叔.+', '帅帅的小楼哥哥！迷人的小楼欧巴！')) # 显示输出结果为: None
```

不过，search只能获取首个匹配的子字符串。

如果想获取到所有匹配的子字符串，我们可以使用findall()函数。

findall(pattern, string)：从字符串参数string中查找所有与正则表达式pattern相匹配的子字符串，如果查找到匹配的子字符串，返回包含所有子字符串的列表，否则，返回空列表。

示例代码：

```
1 result = re.findall(r'小楼.{2}', '帅帅的小楼哥哥！迷人的小楼欧巴！')
2 print(result) # 显示输出结果为：['小楼哥哥', '小楼欧巴']
3
4 print(re.findall(r'楼叔.+', '帅帅的小楼哥哥！迷人的小楼欧巴！')) # 显示输出结果为：[]
```

findall()函数能够把所有匹配的子字符串全部保存到列表。

当匹配到的子字符串数量比较庞大，不想一次性得到所有匹配结果时，我们可以使用finditer()函数。

finditer(pattern, string)：返回在字符串string中所有非重叠匹配所生成的match对象的迭代器。

示例代码：

```
1 result=re.finditer(r'小楼.{2}', '帅帅的小楼哥哥！迷人的小楼欧巴！')
2 for i in result:
3     print(i.group())
```

除了查询功能，re模块还提供了split()函数，用于对字符串进行分割。

split(pattern, string, maxsplit=0)：参数pattern用于匹配分隔符，根据匹配的分隔符对参数字符串string进行分割，返回分割后的子字符串列表。参数maxsplit为最大分割次数。如果没有匹配的分隔符，返回结果为包含参数字符串string的列表。

```
1 result=re.split(r'\.+', '小楼好帅...小楼好棒.小楼好厉害')
2 print(result)# 显示输出结果为：['小楼好帅', '小楼好棒', '小楼好厉害']
3 result=re.split(r'\.+', '小楼好帅...小楼好棒.小楼好厉害',1)
4 print(result)# 显示输出结果为：['小楼好帅', '小楼好棒..小楼好厉害']
5
6 print(re.split(r',+', '小楼好帅...小楼好棒..小楼好厉害')) # 显示输出结果为：['小楼好帅...小楼好棒.小楼好厉害']
```

接下来，是fullmatch()函数。

fullmatch(pattern, string)：如果整个字符串与正则表达式匹配，则返回相应的match对象。如果字符串与正则表达式不匹配，则返回None；注意这与零长度匹配不同。

```
1 print(re.fullmatch(r'小楼.+', '小楼帅哥')) # 显示输出结果为：<_sre.SRE_Match object; span=(0, 6), match='小楼帅哥'>
2 print(re.fullmatch(r'小楼.{2}', '小楼帅哥')) # 显示输出结果为：<_sre.SRE_Match object; span=(0, 6), match='小楼帅哥'>
```

```
3 print(re.fullmatch(r'小楼.{1}', '小楼帅哥')) # 部分匹配，显示输出结果为：None
```

re模块中还有对match对象进行替换的函数sub()。

sub(pattern, repl, string, count=0): 由参数repl取代字符串string中匹配的子字符串，并将整个字符串返回。如果找不到匹配的子字符串，返回没有改变的字符串string。参数repl可以是字符串，也可以是一个函数；如果是一个字符串，函数会处理字符串中的所有反斜杠。例如，“\n”转换为一个换行符，“\r”转换为回车符等等。如果不是有效的转义，则会被保留。反向引用，如“\1”，可以将匹配后的match对象中第1组字符串获取与repl组成新的字符串。

例如，有如下字符串：

```
1 names = r'李学庆,薛之谦,李茂,于波,李连杰,李光洁'
```

1、如果将所有3个字的李姓姓名中前两个姓名替换为“\”，示例代码如下：

```
1 print(re.sub(r'李.(?!),.', '\\\\', names, 2)) # 显示输出结果为：\\,薛之谦,李茂,于波,\\,李光洁
```

在上方代码中，大家注意以下几点：

- “\\\\”最终显示为“\\”，这是因为4条“\”会被Python解释器和正则表达式各转义一次。
- 正则表达式中的“李.”和后方的“.”决定了匹配的长度为3位。
- 正则表达式中的“(?!).”为零宽度断言，表示从已匹配部分“李.”开始，向后方的第1个字符继续进行匹配验证，如果字符不是“,”，则通过“.”获取该字符，当前匹配成功完成；否则，获取到的子字符串不足3位，当前匹配失败。

2、如果将所有3个字的李姓姓名中前两个姓名的姓氏替换为“*”，示例代码如下：

```
1 print(re.sub(r'李(.)?(?!),(.)', r'*\1\2', names, 2)) # 显示输出结果为：*学庆,薛之谦,李茂,于波,*连杰,李光洁
```

在上方代码中，当匹配成功时，两个“(.)”所获取的内容会存储到match对象的group中；这两组内容会被替换字符串中的“\1\2”获取到，形成反向引用。

3、获取所有2~3个字的李姓姓名，并加上“[]”。

```
1 print(re.sub(r'李.?(?!),.', lambda x: '[' + x.group() + ']', names))  
2 # 显示输出结果为：[李学庆],薛之谦,[李茂],于波,[李连杰],[李光洁]
```

在上方代码中，大家注意以下几点：

- “李.?” 和最后的 “.” 决定了匹配的长度为2~3位。
- 正则表达式会先完成 “李.?” 的最大匹配（子字符串的前两位），然后，通过 “(?!,)” 对最后一位字符进行验证，如果不是 “.”，通过 “.” 获取该字符，匹配为3位的姓名字符串；否则匹配为2位的姓名字符串。
- lambda表达式会获取每一个匹配成功的match对象，通过group()方法获取到子字符串后，返回带有 “[]” 的字符串。

subn()函数与sub()函数用法一致，只不过返回的是包含新字符串和替换次数的元组。

示例代码：

```
1 print(re.subn(r'李.?(?!),).', lambda x: '[' + x.group() + ']', names))
2 # 显示输出结果为：(['[李学庆]',薛之谦,['李茂'],于波,['李连杰'],[李光洁]'], 4)
```

如果你匹配的一个字符串，包含了一些可能被Python解释器解释为正则表达式运算符的字符，可以使用escape()函数进行全部转义。

escape(pattern)：此函数能够将参数字符串pattern中，除了ASCII字符、数字和下划线 “_” 以外所有的字符进行转义，并返回新的字符串。

示例代码：

```
1 print(re.escape(r'C:\Users\Administrator\Desktop\date.dat'))
2 # 显示输出结果为：C:\\Users\\Administrator\\Desktop\\date\\.dat
```

除了上述函数外，还有re.purge()函数，用于清空缓存中的正则表达式。

以上是关于re模块的相关函数，与这些函数一起使用的还包含了一些方法。例如，已经使用过的group()方法和groups()方法。

group([group1, ...])：根据参数个数，返回结果为match对象中某一组的内容或多组内容的元组，参数group为0或者省略参数时，可获取被匹配的字符串，参数group大于0或为某一键值时，返回结果为对应的子字符串。

groups(default=None)：返回结果为match对象中所有子字符串的元组，参数default可以指定某一组不存在时的默认值。

start([group])：返回结果为match对象中某一组子字符串在被匹配字符串中的起始位置。

end([group])：返回结果为match对象中某一组子字符串在被匹配字符串中终止位置的下一个位置。

span([group])：返回结果为match对象中某一组子字符串在被匹配字符串中起始位置与终止位置的下一个位置组成的元组。

groupdict(default=None)：返回结果为match对象中所有子字符串与其键值的字典，参数default可以指定某一组不存在时的默认值。

示例代码：

```
1 path = r'C:\Users\Administrator\Desktop\\' # 包含三级目录的路径
2 pat = re.compile(r'..\\(\\w+)\\(\\w+)\\(\\w+)\\(\\w+)?') # 匹配三级或四级目录的表达式
```

```

3
4 result = pat.match(path)
5 print(result) # 显示输出结果为: <_sre.SRE_Match object; span=(0, 31),
match='C:\\Users\\Administrator\\Desktop\\'>
6 print(result.groups()) # 显示输出结果为: ('Users', 'Administrator', 'Desktop', None)
7 print(result.groups(default='date')) # 显示输出结果为: ('Users', 'Administrator',
'Desktop', 'date')
8 print(result.group(0)) # 显示输出结果为: C:\\Users\\Administrator\\Desktop\\
9 print(result.group(1,2)) # 显示输出结果为: ('Users', 'Administrator')
10 print(result.start(1)) # 显示输出结果为: 3
11 print(result.end(1)) # 显示输出结果为: 8
12 print(result.span(1)) # 显示输出结果为: (3,8)
13
14 dict_pat = re.compile(r'..\\(?:P<一级目录>\w+)\(?:P<二级目录>\w+)\(?:P<三级目录>\w+)\|')
# 匹配三级目录并加入键值
15 result_dict = dict_pat.match(path)
16 print(result_dict.groupdict()) # 显示输出结果为: {'一级目录': 'Users', '二级目录':
'Administrator', '三级目录': 'Desktop'}
17 print(result_dict.group('一级目录')) # 显示输出结果为: Users

```

在上方代码中，正则表达式中写入的 “?P<...>” 可以为获取的子字符串指定键值。

如何在PyCharm中安装和卸载第三方库

一、安装

1、在文件（File）菜单中选择默认设置（Default Setting）。

2、在打开的界面中，左侧列表中点击项目解释器选项（Project Interpreter）；界面右侧的项目解释器设置（Project Interpreter）中选择要设置的Python解释器（如果不想每次都选择，可以点击界面右下角的应用（Apply）按钮。）；然后，点击 “+” 按钮。另外，如果想升级某个安装过的库，可以点击带有升级提示的库后，点击右侧的升级按钮进行升级。

3、在打开的可安装包（Available Packages）界面中，输入第三方库的关键字，下方列表会自动筛选出相近的库名称；点击要安装的库名称，右侧会出现相应的描述以及版本选择（一般无需选择）；如果需要把库安装到Python安装目录中，可以勾选 “Install to user’s site packages directory” 选项；最后，点击安装包（Install Package）。

安装包（Install Package）按钮右侧的管理源（Manage Repositories）按钮，用于管理pip源的链接。

默认pip源：<https://pypi.python.org/simple>

如果默认pip源下载速度较慢或者无法连接，可以使用国内的pip镜像源地址。

清华大学：<https://pypi.tuna.tsinghua.edu.cn/simple>

阿里云: <http://mirrors.aliyun.com/pypi/simple>

豆瓣: <http://pypi.douban.com/simple>

当安装开始, 软件会连接到pip源下载选择的库, 下载完毕后自动进行安装, 此时在PyCharm主界面下方能够看到安装状态。

除了通过以上步骤安装第三方库, 而我們也可以在PyCharm的命令行界面中, 通过pip命令进行安装。

以刚才的pymysql库为例, 默认安装命令如下:

```
X:\>pip install pymysql
```

安装指定版本, 需要使用 “==” 操作符指定版本号:

```
X:\>pip install pymysql==0.7.11
```

键入命令并回车后, 程序开始连接默认pip源, 下载指定的库, 并自动进行安装。

安装成功后, 显示 “Successfully install XXXXX” 字样。

如果, 想使用国内镜像源下载安装第三方库, 需要在命令中添加参数 (-i) 。

例如: `pip install -i http://pypi.douban.com/simple pymysql`

二、卸载

卸载非常简单。

在PyCharm的默认设置 (Default Settings) 界面中, 选择完Python解释器后, 点中需要卸载的库, 然后点击 “-” 按钮, 即可完成卸载。

也可以通过pip命令进行卸载:

```
X:\>pip uninstall pymysql
```

Python的图形界面 (GUI) 编程

Python的GUI库有很多, 这里我只介绍wxPython这个库 (很有名气的一个库) 。

当然, 也只是入门级的介绍。

wxPython这个库包含了很丰富的内容, 如果想深入了解, 可以下载官方的文档和Demo文件进行了解。

下载地址: <https://extras.wxpython.org/wxPython4/extras/>

提示: 进入下载之后, 选择版本号, 在打开的新一级目录中, 下载包含demo和docs的压缩文件 (.tar.gz) 。

wxPython可以直接通过pip命令进行安装, 或者在PyCharm中进行安装。

安装操作可以参考《[PyCharm中安装与卸载Python第三方库](#)》。

完成安装之后, 我们就可以通过import wx来使用这个库。

创建一个简单的图形界面应用程序, 有几个必须的步骤。

1. 创建应用
2. 添加框架
3. 添加面板
4. 添加控件
5. 添加函数
6. 绑定控件函数
7. 显示框架

8. 运行主程序

接下来，我们就基于这几个步骤来创建一个简单的图像界面应用程序。

一、显示一个应用程序的图形界面

这里我们先完成步骤1、2、7、8，就能够完成一个空白程序界面的创建。

示例代码：

```
1 import wx
2
3 app = wx.App() # 创建应用
4 win = wx.Frame(None, title='小楼帅哥哥的文本编辑器', size=(400, 320)) # 创建框架
5 # 此处省略其它步骤
6 win.Show() # 显示框架
7 app.MainLoop() # 运行主程序
```

在上方代码中，我们对App和Frame两个类进行了实例化。

这里我们来看一下官方文档中对这两个类的介绍。

wx.App：每个wx应用程序都必须有一个wx.App实例。为了确保GUI平台和wxWidgets已经完全初始化，所有创建UI对象的过程都应该放在wx.App实例化之后。

wx.Frame：框架是一个窗口，其大小和位置可以(通常)由用户更改。它通常有粗边框和标题栏，并且可以选择包含菜单栏、工具栏和状态栏。框架可以包含任何不是框架或对话框的窗口。

在对Frame类进行实例化时，可以设定一些参数。

`Frame(parent, id=ID_ANY, title=" ", pos=DefaultPosition, size=DefaultSize, style=DEFAULT_FRAME_STYLE, name=FrameNameStr)`

- **parent**：父窗口。一般情况下是None。如果不是None，当前窗口的父窗口被最小化并恢复时，当前窗口将被最小化。
- **id**：窗口的标识符。它的值可以为-1，表示默认值。
- **title**：窗口的标题。它的值是一个字符串。
- **pos**：窗口的位置。参数值DefaultPosition表示默认位置，可以输入包含x轴与y轴坐标数值的元组。
- **size**：窗口的尺寸。参数值DefaultSize表示默认尺寸，可以输入包含宽度与高度数值的元组。
- **style**：窗口的样式。具体设置参考官方文档wx.Frame类描述中的Window Styles部分【[点此查看](#)】。例如，创建一个不可以调整尺寸的窗口，该参数的设置为：使用默认样式并且 (&) 不包含 (~) 更改边框尺寸和 (!) 最大化按钮

```
1 style = wx.DEFAULT_FRAME_STYLE & ~(wx.RESIZE_BORDER | wx.MAXIMIZE_BOX )
```

- **name**：窗口的名称。值为字符串；该参数用于将名称与项目关联起来，允许应用程序用户为单个窗口设置主题资源值。

除了两个类的实例化，我们还分别调用了这两各类的实例方法。

注意，wxPython库中的方法命名都是首字母大写的格式，这和python的习惯不一致。

- Show(show=True): 用于显示隐藏窗口。不填入参数或参数为True时，显示窗口；参数为False时，隐藏窗口。
- MainLoop(): 循环执行GUI主程序

运行代码之后，我们就能够看到一个程序的界面。

接下来，我们向窗口中添加一些控件。

虽然，我们可以在窗口中直接添加控件，但是，这样并不便于管理。

当一个窗口中，划分了不同的功能（一个功能可能是多个控件组成），有些功能需要能够单独的显示或隐藏时，实现起来会非常复杂。

解决这样的问题，我们可以使用wx.Panel类。

Panel是放置控件的窗口。

在实例化一个Panel时，我们同样可以设定一些参数。

Panel(parent, id=ID_ANY, pos=DefaultPosition, size=DefaultSize, style=TAB_TRAVERSAL, name=PanelNameStr)

- parent: 父窗口。
- id: 面板的标识符。
- pos: 窗口的位置。参数值DefaultPosition表示默认位置，可以输入包含x轴与y轴坐标数值的元组。
- size: 窗口的尺寸。参数值DefaultSize表示默认尺寸，可以输入包含宽度与高度数值的元组。
- style: 面板的样式。具体设置参考官方文档wx.Panel类描述中的Window Styles部分。
- name: 面板的名称。参数值为字符串。

默认情况下，面板的尺寸和父窗口一致，颜色和对话框相同；如果想更改可以用面板对象调用相关的Set方法。

例如，添加一个面板并设置颜色为灰色。

示例代码：

```
1 pan = wx.Panel(win) # 添加面板
2 pan.SetBackgroundColour('#666666') # 设置面板颜色
```

运行代码，我们就能够看到当前的效果。

有了面板之后，我们可以在面板中添加其他的控件。

例如，我们做一个可以选择文件/打开文件/编辑文本/保存文件的文本编辑器。

这里我们需要添加几个控件：

- 文件选择控件: FilePickerCtrl
- 按钮: Button
- 文本控件: TextCtrl

这些控件的参数大同小异，一般包含以下参数：

- parent: 父窗口。
- id: 控件的标识符。
- label/value: 控件上默认的文本或值。

- pos: 控件在父窗口中的位置。
- size: 控件的尺寸。
- style: 控件的样式。
- validator: 控件的验证器。
- name: 控件的名称。

示例代码:

```
1 file_btn = wx.FilePickerCtrl(pan, pos=(5, 5))
2 file_btn.GetPickerCtrl().SetLabel('选择')
3 open_btn = wx.Button(pan, label='打开', pos=(215, 5), size=(80, 30))
4 save_btn = wx.Button(pan, label='保存', pos=(300, 5), size=(80, 30))
5 cont_ipt = wx.TextCtrl(pan, pos=(5, 40), size=(375, 240), style=wx.TE_MULTILINE |
    wx.HSCROLL)
```

这里特别说一下FilePickerCtrl控件和TextCtrl控件。

FilePickerCtrl控件默认的文字为Browse (浏览), 如果想改成中文, 不能直接调用SetLabel()方法。

原因就是在这个控件由两个控件 (TextCtrl和PickerCtrl) 组成, 如果设置选择控件 (PickerCtrl) 的文本, 必须先获取选择控件。

TextCtrl控件默认是单行文本输入框, 如果想能够多行输入并且带有滚动条, 需要设置style参数。

- wx.TE_MULTILINE: 文本允许多行 (自动允许垂直滚动)
- wx.HSCROLL: 允许水平滚动

通过以上代码, 我们就实现了我们想要的图形界面。

示例代码: (当前全部代码)

```
1 import wx
2
3 app = wx.App() # 创建应用
4 win = wx.Frame(None, title='小楼帅哥哥的文本编辑器', size=(400, 320)) # 创建框架
5
6 pan = wx.Panel(win) # 添加面板
7 pan.SetBackgroundColour('#666666') # 设置面板颜色
8
9 file_btn = wx.FilePickerCtrl(pan, pos=(5, 5)) # 添加文件选择控件
10 file_btn.GetPickerCtrl().SetLabel('选择') # 设置选择控件文本
11 open_btn = wx.Button(pan, label='打开', pos=(215, 5), size=(80, 30)) # 添加按钮控件
12 save_btn = wx.Button(pan, label='保存', pos=(300, 5), size=(80, 30)) # 添加按钮控件
13 cont_ipt = wx.TextCtrl(pan, pos=(5, 40), size=(375, 240), style=wx.TE_MULTILINE |
    wx.HSCROLL) # 添加文本控件
14
```

```
15 win.Show() # 显示框架
16 app.MainLoop() # 运行主程序
```

不过，完成这个界面的过程很复杂，每个控件我们都需要指定位置；并且，还存在一个问题，就是改变窗口尺寸的时候，控件不能够跟随着改变尺寸。

如果想布局更加容易，而且能达到控件尺寸自动适应窗口尺寸的效果，我们可以使用sizer（尺寸器）。

wx.BoxSizer类，能够帮助我们创建尺寸器对象。

大家可以把尺寸器理解为一个控件容器，尺寸器中的控件可以水平摆放或者垂直摆放，并且尺寸器可以嵌套。

我们刚刚写好的界面，如果使用尺寸器，需要先划分。

如上图所示，hbox中包含3个控件：文件选择/打开按钮/保存按钮。

而vbox中，包含hbox和文本控件。

示例代码：

```
1 hbox = wx.BoxSizer() # 尺寸器实例化（默认水平）
2 hbox.Add(file_btn, proportion=1, flag=wx.EXPAND) # 添加控件到尺寸器
3 hbox.Add(open_btn, proportion=0, flag=wx.LEFT, border=5) # 添加控件到尺寸器
4 hbox.Add(save_btn, proportion=0, flag=wx.LEFT, border=5) # 添加控件到尺寸器
5
6 vbox = wx.BoxSizer(wx.VERTICAL) # 尺寸器实例化（垂直）
7 vbox.Add(hbox, proportion=0, flag=wx.EXPAND | wx.ALL, border=5) # 添加尺寸器到尺寸器
8 vbox.Add(cont_ip, proportion=1, flag=wx.EXPAND | wx.LEFT | wx.BOTTOM | wx.RIGHT,
9 border=5) # 添加控件到尺寸器
10
11 pan.SetSizer(vbox) # 为窗口设置尺寸器
```

在上方代码中，需要为大家介绍一下Add()方法和SetSizer()方法。（这里只介绍部分参数）

Add (sizer, proportion=0, flag=0, border=0)：此方法能够将子尺寸器（或控件）添加到尺寸器。

- sizer：尺寸器或控件。
- proportion：参数值为0或者1；值为1时，尺寸器或控件尺寸将随窗口尺寸发生变化；值为0时，为固定尺寸。
- flag：影响尺寸器行为的标志；参数为整数或值为整数的常量。
- border：边距；值为整数。

这里比较难懂的是flag这个参数，它的值可以为多个，值之间需要使用管道符“|”分隔。

- wx.EXPAND：指定尺寸器或控件是否被扩展以填充分配给它的空间。
- wx.LEFT/wx.RIGHT/wx.BOTTOM/wx.ALL：指定边界宽度将应用到的尺寸器的哪一边。

更多参数信息请参考官方文档中[sizers_overview](#)的Sizer Flag部分【[点此查看](#)】。

SetSizer(sizer)：将设置好布局的尺寸器添加到窗口。

更多参数信息请参考官方文档中wx.Window的方法部分【[点此查看](#)】。

在我们之前编写的图形界面程序中包含以下功能：

- 打开选取的文件到文本控件
- 保存文本控件的内容到文件

那么，对于这两个功能，我们要定义相应的功能函数。

示例代码：

```
1 def open_file(event):
2     file_path = file_btn.GetPath() # 从文件选择器控件获取文件路径
3     with open(file_path) as file: # 打开文件
4         cont_ipt.SetValue(file.read()) # 读取文件内容设置到文本控件的值
5
6 def save_file(event):
7     file_path = file_btn.GetPath() # 从文件选择器控件获取文件路径
8     with open(file_path, 'w') as file: # 写入模式打开文件
9         file.write(cont_ipt.GetValue()) # 获取文本控件的值写入文件
```

在上方代码中，大家能够看到，我们通过Get方法能够获取到控件中的路径以及值。

实际上控件的Get方法还有很多，在这里没有办法一一介绍。

这里只介绍比较常用的一些方法：

- GetPath(): 获取路径。
- GetValue(): 获取控件的值。
- GetLabel(): 获取控件的标签文本。
- GetName(): 获取控件的名称。
- GetId(): 获取控件的ID。
- GetSize(): 获取控件的尺寸。
- GetPosition(): 获取控件的位置。

更多的Get方法可以查看官方文档中的详细介绍。【[点此查看](#)】

完成了功能函数的编写，接下来我们需要将功能函数和控件进行绑定，才能够在用户和控件产生交互时，调用功能函数。

示例代码：

```
1 open_btn.Bind(wx.EVT_BUTTON, open_file) # 绑定打开文件按钮
2 save_btn.Bind(wx.EVT_BUTTON, save_file) # 绑定保存文件按钮
```

到这里，这个图形界面程序就完成了。

我们不但可以在PyCharm中运行代码查看效果，也可以到本地磁盘的文件夹中，双击py文件运行这个程序。

不过，在运行时会弹出一个py.exe的窗口，这是因为py文件会通过py.exe运行。

如果不想看到这个窗口，可以把文件的后缀改为“.pyw”，这样的话程序就会通过pyw.exe运行，不会再显示多余的窗口。

在PyCharm中修改文件名，可以在文件上点击鼠标右键，选择重构（Refactor）选项中的重命名（Rename）选项进行修改。

也可以选中文件后，通过快捷键<Shift+F6>进行修改。

最后，我们再了解一些关于Python文件类型的知识。

Python文件有以下几种类型：

- py：源代码文件。由 py.exe 运行，也可以通过命令行终端运行。
- pyw：图形界面程序源代码文件。由pyw.exe运行，和py运行的区别在于不会显示命令行窗口。不过，还是建议大家编程过程中，先将源代码命名为py文件，当程序出现错误时，能够在命令行窗口看到相关信息。
- pyc：py文件经过编译后产生的文件，无法直接看到源代码。因为已经经过编译，运行速度比py文件更快。
- pyo：py文件优化编译后产生的文件，无法直接看到源代码。可以在命令行窗口，通过 “python -O 源代码文件” 将源代码文件编译为pyo 文件。
- pyd：这类文件不是用 python 编写成的，一般是其他语言编写的 python 扩展模块。

提示：本篇教程相关源代码较长，不方便在此展示，大家可以下载这个程序的py文件。【[点击下载](#)】

本节知识点：

- 1、控件绑定函数
- 2、控件对象的Get方法
- 3、Python文件类型简介

本节英文单词与中文释义：

- 1、refactor：重构
- 2、rename：重命名

练习：

编写一个简单的计算器。

提示：

- 1、布局需要尺寸器嵌套，参考下图。

第一种嵌套结构：

红色：面板的垂直布局尺寸器，嵌套蓝色尺寸器。

蓝色：除了文本控件，另外两个是水平布局尺寸器，第2个尺寸器中嵌套紫色尺寸器。

紫色：左侧为垂直布局尺寸器，嵌套绿色尺寸器，右侧为按钮控件。

绿色：3个水平布局尺寸器，每个尺寸器添加相应的按钮控件。

第二种嵌套结构：

红色：面板的垂直布局尺寸器，嵌套蓝色尺寸器。

蓝色：上方为文本控件，下方为水平布局尺寸器，嵌套5个紫色尺寸器。

紫色：垂直布局尺寸器，每个尺寸器添加相应的按钮控件。

参考文件使用的是第二种嵌套结构。

- 2、控件绑定功能函数时，不能够直接传递参数，可以通过lambda表达式调用功能函数并传入参数。

参考文件：【[点击下载](#)】

说明：参考文件仅供参考，并非最优方案。

数据库相关的知识

这里我们所接触的是关系型数据库。

当前主流的关系型数据库有Oracle、DB2、PostgreSQL、Microsoft SQL Server、Microsoft Access、MySQL。

那么，什么是关系型数据库？

关系型数据库，是指采用了关系模型来组织数据的数据库。关系模型就是指二维表格模型，因而一个关系型数据库就是由二维表及其之间的联系组成的一个数据组织。

这句话不好理解，看起来有些晕。

实际上，你可以把关系型数据库想象成一个Excel工作簿，工作簿的名称就相当于数据库的名称，工作簿中包含的工作表就是二维数据表。同一个工作簿中的工作表之间存在联系，从而形成了工作簿的数据组织。

除了要知道这些内容，我们还要了解一些相关的概念：

- 关系：可以理解为一张二维数据表，关系名就是数据表的名称。例如班级数据库中的成绩表。
- 元组：可以理解为二维数据表中的一行，它是数据库中的一条记录。例如成绩表中某一名学生的成绩记录。
- 属性：可以理解为二维数据表中的一列，它是数据库中的一个字段。例如成绩表中的学号。
- 域：是指属性的取值范围，也就是数据库中某一列的取值限制。例如某一成绩要在0~100之间。
- 关键字：是指一组可以唯一标识元组的属性。它是数据库中的主键，主键可以是一个或多个。例如学号是唯一的。
- 关系模式：是指对关系的描述，也就是数据库中的表结构。格式为：关系名（属性1，属性2，...）。例如：成绩表（学号，姓名，语文，数学）

了解了关系型数据库的一些基本概念，接下来我们来看如何使用关系型数据库。

作为一篇入门教程，我们从一个相对快捷、简单的数据库SQLite来入手。

Python官方文档中对SQLite的介绍如下：

SQLite是一个C语言库，它提供了一个轻量级的基于磁盘的数据库，不需要单独的服务器进程，并且允许使用SQL查询语言的非标准变体来访问数据库。一些应用程序可以使用SQLite进行内部数据存储。还可以使用SQLite对应用程序进行原型化，然后将代码移植到更大的数据库，如PostgreSQL或Oracle。

这个数据库不用单独安装，因为Python3中已经包含这个数据库引擎，并且在Python标准库中包含了对这个数据库进行操作的PySqlite模块（标准库中模块名为sqlite3），所以使用起来非常方便。

对于这个数据库的基本操作，在本篇教程中主要涉及以下内容：

- 创建/连接数据库
- 打开游标
- 创建表结构
- 操作数据（增、删、改、查）
- 提交事务
- 关闭游标链接

一、创建/连接数据库

不管是使用一个已有的数据库（.db文件），还是创建一个新的数据库，我们都需要和数据库进行连接。

然后通过连接对象进行后续操作。

这里我们可以使用connect()函数。

这个函数能够创建数据库连接对象，并且当数据库不存在时，创建新的数据库。

示例代码：（创建数据库student）

```
1 import sqlite3
2
3 conn = sqlite3.connect('student.db')
```

二、创建游标

当完成一个新数据库的创建，接下来就是在数据库中添加新的数据表以及表中的数据。

但是，创建表以及对表中数据的操作，我们都需要执行相应的SQL（Structured Query Language：结构化查询语言）语句。

而执行SQL语句，我们需要先打开游标，通过游标执行SQL语句。

示例代码：

```
1 cur = conn.cursor()
```

三、表结构

打开游标之后，我们就可以通过游标的execute()方法执行SQL（Structured Query Language：结构化查询语言）语句了。

这里我们创建成绩表“score”。

如果已存在同名数据表，会无法完成新数据表的创建，所以在创建之前，我们需要先删除已有的数据表，再创建新的数据表。

删除数据表的SQL语句为：DROP TABLE IF EXISTS 表名

创建数据表的SQL语句为：

CREATE TABLE 表名(

字段名 数据类型 PRIMARY KEY[主键，一个或多个字段],

字段名 数据类型 约束[可选],

.....

字段名 数据类型

)

创建数据表时，我们需要定义表名，然后定义表中（括号中）的字段名、数据类型以及相关的约束。

关于数据类型，SQLite具有以下五种数据类型：

- NULL：空值。
- INTEGER：带符号的整数，根据值的大小存储在 1、2、3、4、6 或 8 字节中。
- REAL：浮点数，存储为8-byte IEEE浮点数。
- TEXT：字符串文本。
- BLOB：二进制对象。

另外还要注意，被指定为主键的列，列值不可有重复。

除了数据类型，我们还可以给字段值添加约束，例如不能为空值、数值必须在某一区间等。

约束的类型包含以下几种：

- PRIMARY KEY：主键约束，唯一标识数据库表中的每条记录。
- NOT NULL：非空约束，确保某列不能有 NULL 值。
- DEFAULT：默认约束，当某列没有指定值时，为该列提供默认值。
- UNIQUE：唯一约束，确保某列中所有的值互不相同。
- CHECK：检查约束，确保某列中所有的值满足指定的条件。

那么，参照着上方的SQL语句知识，我们将删除表以及创建表结构的语句添加到代码中。

示例代码：（删除/创建表score）

```
1 cur.execute('DROP TABLE IF EXISTS score')
2 cur.execute('
3 CREATE TABLE score(
4     StuId    INTEGER    PRIMARY KEY NOT NULL,
5     StuName TEXT        NOT NULL,
6     ChineseScore REAL    DEFAULT 0    CHECK(ChineseScore>=0 AND ChineseScore<=100),
7     MathScore REAL      DEFAULT 0    CHECK(MathScore>=0 AND MathScore<=100)
8 )
9 ''')
```

上方代码中，表结构中共4个字段：

- StuId：学号，整数，主键，非空字段
- StuName：姓名，文本，非空字段
- ChineseScore：语文成绩，浮点数，默认值为0，值必须大于等于0并且小于等于100
- MathScore：数学成绩，浮点数，默认值为0，值必须大于等于0并且小于等于100

四、添加数据

通过之前的步骤，完成了表的创建，但这时还只是一个空表。

我们需要使用这个表存储数据，就要把数据添加到表中。

添加数据的SQL语句为：INSERT INTO 表名 [(列1, 列2,...)] VALUES (值1, 值2,...)

如果添加一行数据时，不指定仅为某些列添加数据，可以省略指定列的部分。

例如，为成绩表score添加一行数据：' INSERT INTO score VALUES(20712420205," 小楼",100,100)'

注意：SQL语句中的值如果是字符串需要加上双引号。

接下来，我们将一个文本文档“score.txt”中的数据逐行读取，并添加到这个数据表中。【[点此下载](#)】

文档的内容如下：

'20712420001' ; 吴陈' ,92,93

'20712420002' ; 刘捷' ,81,97

'20712420003' ; 万刚' ,82,65

（省略部分内容）

如上所示，文档中每一行数据都是逗号分隔，学号和姓名带有单引号，语文成绩和数学成绩均为数字。

根据表中相应字段的数据类型，很明显学号带有引号，不符合INTEGER类型。

所以，我们需要在将每一行数据插入到数据表之前，先进行数据的修改，去除字段值两侧的单引号。

示例代码：

```

1 query = 'INSERT INTO score VALUES(?,?,?,?)' # 定义SQL语句
2 with open(r'C:\Users\Administrator\Desktop\score.txt') as file: # 打开文件
3     for line in file: # 读取文件每一行
4         values = [i.strip("'") for i in line.split(',')] # 以逗号为分隔符，将每一行内容变为值，去除单引号后存入列表。
5         conn.execute(query, values) # 将列表中的值代入SQL语句，并执行SQL语句。

```

上方代码中，定义SQL语句时使用了字段标记“?”，类似于格式化字符串时字符串中的“%”。

当我们使用execute()方法时，可以通过第2个可选参数parameters，将需要添加的值以列表的形式（代码中的values）传入。

五、提交事务

但是，当我们运行写好的代码，会发现数据表仍然是空表。

这是因为，我们只做了数据的修改，但是没有提交。

所以，我们还需要加上一句提交事务的代码。

示例代码：

```

1 conn.commit()

```

六、关闭游标与连接

当我们不再使用游标游标和连接，我们要确保它们正常的关闭。

示例代码：

```

1 cur.close()
2 conn.close()

```

不过，如果程序发生异常，怎么确保它们被关闭了呢？

可以通过try:...except:...finally:...的处理，将关闭语句写在finally的语句块中。

示例代码：（完整代码）

```

1 import sqlite3
2
3 conn = sqlite3.connect('student.db')
4 try:
5     cur = conn.cursor()
6     cur.execute('DROP TABLE IF EXISTS score')
7     cur.execute('')

```

```

8      CREATE TABLE score(
9          StuId    INTEGER    PRIMARY KEY NOT NULL,
10         StuName TEXT      NOT NULL,
11         ChineseScore REAL    DEFAULT 0    CHECK(ChineseScore>=0 AND
ChineseScore<=100),
12         MathScore REAL    DEFAULT 0    CHECK(ChineseScore>=0 AND ChineseScore<=100)
13     )
14     '''
15     query = 'INSERT INTO score VALUES(?,?,?,?)' # 定义SQL语句
16     with open(r'C:\Users\Administrator\Desktop\score.txt') as file: # 打开文件
17         for line in file: # 读取文件每一行
18             values = [i.strip("'") for i in line.split(',')] # 以逗号为分隔符,将每一行
内容变为值,去除单引号后存入列表。
19             conn.execute(query, values) # 将列表中的值代入SQL语句,并执行SQL语句。
20     conn.commit()
21 except:
22     raise
23 finally:
24     cur.close()
25     conn.close()

```

SQLite数据库的一些内容。

如果想用好数据库，一些常用的语句是必须掌握的。

SQL语句以关键字开始，常用的关键字包括：

- CREATE (创建)
- DROP (丢弃)
- ALTER (改变)
- INSERT (插入)
- SELECT (查询)
- UPDATE (更新)
- DELETE (删除)

通过关键字的释义，大家就基本上能看出关键字的用途。

接下来，我们结合之前创建的数据库student中的score表来了解这些关键字的使用。

一、CREATE

这个关键字，比较基本的用法的是通过这个关键字创建表。

```

1  CREATE TABLE 表名(
2      列名1 数据类型 [约束],
3      列名2 数据类型 [约束],
4      列名3 数据类型 [约束],

```

```
5      .....  
6  )
```

二、DROP

这个关键字，比较基本的用法是通过这个关键字删除表。

```
1 DROP TABLE 表名;
```

三、ALTER

这个关键字，比较基本的用法是通过这个关键字更改表的名称和结构。

```
1 ALTER TABLE 表名 RENAME TO 新表名  
2 ALTER TABLE 表名 ADD COLUMN 新列的定义
```

例如，我们将表score重命名为achievement，并添加英语成绩列EnglishScore。

示例代码：

```
1 import sqlite3  
2  
3 conn = sqlite3.connect('student.db')  
4 cur = conn.cursor()  
5 query1 = 'ALTER TABLE score RENAME TO achievement'  
6 query2 = 'ALTER TABLE achievement ADD COLUMN EnglishScore REAL DEFAULT 0 ' \  
7          'CHECK(EnglishScore >= 0 AND EnglishScore <= 100)'  
8 cur.execute(query1)  
9 cur.execute(query2)  
10 conn.commit()
```

四、INSERT

这个关键字，用于向数据表添加新的数据。

```
1 INSERT INTO 表名( 列名1, 列名2, 列名3,...) VALUES ( 列值1, 列值2, 列值3,...) # 添加数据  
   时，只设定部分列值。  
2 INSERT INTO 表名 VALUES ( 列值1, 列值2, 列值3,...) # 添加数据时，设定全部列值。
```

五、SELECT

这个关键字，用于查询数据表中的内容。

- 1 `SELECT` 列名1, 列名2, 列名3,... `FROM` 表名 # 查询数据表中所有数据行，每一行数据包含指定列的列值。
- 2 `SELECT *` `FROM` 表名 # 查询数据表中所有数据行，每一行数据包含所有列的列值。

在之前的示例代码中，我们为数据表achievement（重命名后的表名）添加了数据，但是一直不知道我们添加的数据以及所做的更改是否有效。

这里，我们通过SELECT关键字查询一下数据表achievement的所有内容，并把它们显示输出。

示例代码：（继上一段示例代码之后）

```
1 query3 = 'SELECT * FROM achievement'
2 cur.execute(query3)
3 result = cur.fetchall() # 获取所有查询结果的对象
4 for row in result:
5     print(row)
```

在上方代码中，我们定义了查询表achievement中所有数据的语句，并通过execute()方法执行了这个语句。但是，只执行语句还是不能够看到查询结果。

我们还需要通过fetchall()方法获取所有查询结果，然后循环遍历这些查询结果，将它们显示输出。

除了fetchall()方法，还有一个fetchone()方法，能够每次获取查询结果中的一条数据。

例如，我们想显示输出查询结果中的前10条数据。

示例代码：

```
1 for row in range(10):
2     print(cur.fetchone())
```

六、UPDATE

- 1 `UPDATE` 表名 `SET` 列名1 = 值1, 列名2 = 列值2, 列名3 = 列值3,...[WHERE 条件]

这个关键字，用于更新数据表记录中指定列的列值。

不过，在上方的语句中大家能够看到WHERE这个关键字，关键字后面跟随条件语句。

因为我们更新记录时，一般不会更新所有的记录，而是更新一条或多条符合条件的记录。

所以，需要给UPDATE语句加上WHERE子句。

例如，我们更新成绩表中姓名为“胡小丽”的语文成绩为“100”。

示例代码：

```
1 query4 = 'UPDATE achievement SET ChineseScore = 100 WHERE StuName = "胡小丽"'
2 query5 = 'SELECT * FROM achievement WHERE StuName = "胡小丽"'
3 cur.execute(query4)
4 conn.commit()
5 cur.execute(query5)
6 print(cur.fetchall()) #显示输出结果为: [(20712420004, '胡小丽', 100.0, 84.0)]
```

通过上方代码，大家能够看到在SELECT语句中也添加了WHERE子句，进行有条件查询。

七、DELETE

这个关键字，用于删除数据表中的数据行。

```
1 DELETE FROM 表名 [WHERE 条件]
```

例如，删除数据表中姓名为“胡小丽”的记录。

示例代码：

```
1 query6 = 'DELETE FROM achievement WHERE StuName = "胡小丽"'
2 cur.execute(query6)
3 conn.commit()
```

以上就是SQLite数据库的一些基本常用语句。

不过，在对数据库进行操作时，这些基本语句还远远不够，我们往往需要为基本语句添加子句，来满足更多的操作需求。例如大家所看到的WHERE子句。

关于子句，也有一些常用的关键字：

- WHERE（哪里）
- AND（并且）
- OR（或者）
- BETWEEN（在...之间）
- COUNT（计数）
- SUM（求和）
- DISTINCT（去重）
- IN（在...里面）
- NOT IN（不在...里面）
- LIKE（相似）

- ORDER BY (排序)
- GROUP BY (组合)

下面通过示例代码为大家演示这些关键字的使用，请大家阅读示例代码中的注释进行理解。

注意：示例代码仅通过查询（SELECT）操作演示，其它操作自行研究。

示例代码：

```
1  # AND: 查询语文成绩并且数学成绩均满90分的学生记录
2  query7 = 'SELECT * FROM achievement WHERE ChineseScore >= 90 AND MathScore >= 90'
3  # OR: 查询语文成绩或者数学成绩满90分的学生记录
4  query8 = 'SELECT * FROM achievement WHERE ChineseScore >= 90 OR MathScore >= 90'
5  # BETWEEN: 查询总分数（语文+数学）在180~190之间的记录
6  query9 = 'SELECT * FROM achievement WHERE ChineseScore + MathScore BETWEEN 180 AND 190'
7  # COUNT: 查询语文成绩满90分的学生数量
8  query10 = 'SELECT COUNT(ChineseScore) FROM achievement WHERE ChineseScore >= 90'
9  # SUM: 查询语文成绩满90分的语文成绩总和
10 query11 = 'SELECT SUM(ChineseScore) FROM achievement WHERE ChineseScore >= 90'
11 # DISTINCT: 查询所有不同的语文分数
12 query12 = 'SELECT DISTINCT ChineseScore FROM achievement'
13 # IN: 查询名字为“邓恒”、“郑宇”、“邓强”的名字与总分记录
14 query13 = 'SELECT StuName, ChineseScore + MathScore FROM achievement WHERE StuName IN
15 ("邓恒","郑宇","邓强")'
16 # NOT IN: 查询除“邓恒”、“郑宇”、“邓强”以外的名字与总分记录
17 query14 = 'SELECT StuName, ChineseScore + MathScore FROM achievement WHERE StuName NOT
18 IN ("邓恒","郑宇","邓强")'
19 # LIKE: 查询所有“刘”姓学生的记录（通配符“%”代表零个、一个或多个数字或字符，下划线“_”代表一个数
20 字或字符。）
21 query15 = 'SELECT * FROM achievement WHERE StuName LIKE "刘%"'
22 # ORDER BY: 查询并按总分降序DESC（升序为ASC）排列所有记录
23 query16 = 'SELECT * FROM achievement ORDER BY ChineseScore + MathScore DESC'
24 # GROUP BY: 查询总分数满180分各组分数的记录数量和总分数
25 query17 = 'SELECT COUNT(*),ChineseScore + MathScore FROM achievement ' \
26           'WHERE ChineseScore + MathScore >= 180 Group By ChineseScore + MathScore'
```

Python的多进程

首先，需要知道进程的概念。

它的概念有些枯燥。

狭义的定义：进程是正在运行的程序的实例。

广义的定义：进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动，是操作系统动态执行的基本单元。

是不是看着头晕了？

那就忘掉它。

你需要知道的是，进程只会在程序运行时出现。

程序的概念是指令和数据的有序集合，但是其本身没有任何运行的含义，是一个静态的概念。

而进程是程序的一次执行过程，它是一个动态的概念。

以windows系统中为例，你下载了一个软件程序，在你没有运行它之前，它只是一个静态的程序文件。

当我们打开这个软件程序，这时在系统任务管理器的进程列表中，我们能够看到这个软件程序的进程。

当我们关闭这个软件程序，这时在系统任务管理器的进程列表中，和这个软件程序对应的进程会消失。

再次打开软件程序，进程又会再次出现。

当我们在系统任务管理器中结束这个软件程序的进程，这个软件程序也会被结束运行。

这个时候，你是否能够理解进程的狭义定义了呢？

死记硬背概念是没有用的，理解才是最重要的。

我们来看一段代码！

示例代码：

```
1 import time
2
3 def task01(name):
4     print(name, 1)
5     time.sleep(0.001)
6     print(name, 2)
7     time.sleep(0.001)
8     print(name, 3)
9
10 def task02(name):
11     print(name, 1)
12     time.sleep(0.001)
13     print(name, 2)
14     time.sleep(0.001)
15     print(name, 3)
16
17 if __name__ == '__main__':
18     task01('函数1: ')
19     task02('函数2: ')
```

在上方代码中，我定义了两个函数。

然后，在运行代码时调用了这两个函数。

程序运行的结果为：

函数1： 1

函数1： 2

函数1： 3

函数2： 1

函数2： 2

函数2： 3

不管运行多少次，都是这样的结果。

通过这个结果我们能够看出，两个函数是按顺序，先执行完上方的task01，再执行下方的task02。

而当我们执行这段代码时，实际上我们已经开启了一个进程。

正是这个进程完成了这段代码的执行。

我们继刚才的代码之后，再加入一些代码看一下。

示例代码：

```
1 import multiprocessing
2 print(multiprocessing.current_process().name) # 显示输出结果为: MainProcess
```

multiprocessing模块为多进程模块，可以通过这个模块中的current_process()函数，访问当前进程实例的name特性，这个特性是当前进程实例的名称。

通过运行上方代码，我们看到了一个进程名称：MainProcess（主进程）

任何程序在运行时，都至少有一个进程，也就是说都会有MainProcess这个主进程的存在。

既然提到主进程，那么，显然还有不是主进程的进程，也就是子进程。

多进程模块，能够让我们在主进程的基础之上，添加多个子进程。

例如刚才的代码中，运行程序时只能按照次序运行。

如果，我们想在程序运行时，两个函数一起执行是否可以呢？

用多进程，每个进程执行不同的函数就可以实现这样的效果。

我们在之前的代码上增加一些新内容。

示例代码：

```
1 import time,multiprocessing # 导入需要使用的模块
2
3 def task01(name):
4     print(multiprocessing.current_process().name) # 显示输出结果为: Process-1
5     print(name, 1)
6     time.sleep(0.001)
7     print(name, 2)
8     time.sleep(0.001)
9     print(name, 3)
10
```

```

11 def task02(name):
12     print(multiprocessing.current_process().name) # 显示输出结果为: Process-2
13     print(name, 1)
14     time.sleep(0.001)
15     print(name, 2)
16     time.sleep(0.001)
17     print(name, 3)
18
19 if __name__ == '__main__':
20     print(multiprocessing.current_process().name) # 显示输出结果为: MainProcess
21     process1 = multiprocessing.Process(target=task01, args=('函数01: ',)) # 创建子进程运
    行函数task01
22     process2 = multiprocessing.Process(target=task02, args=('函数02: ',)) # 创建子进程运
    行函数task02
23     process1.start() # 启动子进程
24     process2.start() # 启动子进程

```

在之前的代码中，新增了一些代码（带有注释的部分）。

通过这些代码，创建了子进程，并在主进程运行时，启动这些子进程。

当我们运行程序，显示输出结果为：

MainProcess

Process-1

函数01: 1

函数01: 2

函数01: 3

Process-2

函数02: 1

函数02: 2

函数02: 3

这个运行结果看上去，仍然是先执行了函数task01（进程process1），再执行了函数task02（进程process2）。

先别管这个结果，我们把每个time.sleep()的语句中的参数都从“0.001”改成“0.01”，再运行程序看一下。

提示：在PyCharm中可以通过导航菜单-编辑（Edit）-查找（Find）-替换（Replace）选项进行批量替换，快捷键为<Ctrl+R>。

运行程序之后，显示输出结果类似：

MainProcess

Process-1

函数01: 1

函数01: 2

Process-2

函数02: 1

函数01: 3

函数02: 2

函数02: 3

注意，每次运行的显示输出结果不一定相同，次序会发生变化。

通过上方的程序运行结果，大家能够看出函数task01和函数task02的运行过程混合在了一起。

也就是说，两个函数（进程）同时在运行（并行）。

不过，因为每次都是进程process1先启动，所以运行结果中也是函数task01先执行。

在我们运行程序时，是由cpu在顺序执行代码。

那么，也就意味着多个进程中的程序代码需要多个cpu去执行。

所以，只有多核cpu才能真正的并行执行多个进程。

但是，cpu的核心数量即便是多个，也是有限的。例如4核和8核。

当进程数量超过cpu核心数量的时候，会怎么样呢？

操作系统会让cpu执行某个进程一段时间后，就去执行另外一个进程一段是时间，如此轮流执行进程，而不是完全执行某个进程的代码后再执行下一个进程中的代码。

所以，即使单核cpu，操作系统也能让它处理多进程，这样的操作系统就是通常所说的多任务操作系统。

对于操作系统来说，一个任务就是一个进程，并非执行一个完整的程序才是一个进程。

就像前面的代码中，我们可以在同一个程序的代码中，启动多个进程去执行多个任务。

所以，一个程序可以启动多个进程，同时让多种功能同时执行。

很多软件程序也是这么做的，例如有道词典取词功能和词典功能都是独立的进程在起作用。

那么，假如取词之后要通过词典翻译，就需要两个进程之间的通信。

进程间通信，我们可以使用Queue类来实现。

通过Queue实例对象的put()和get()方法，能够送出和获取进程的数据。

例如，我们虚拟一个取词和词典进程间的通信。

示例代码：

```
1 import multiprocessing
2 import time
3
4 def ocr(que):
5     for value in ['one', 'two', 'three']:
6         print('完成取词...')
7         que.put(value) # 将数据送出到进程共享队列
8         time.sleep(1)
9
10 def dict(que):
11     dict = {'one': '一', 'two': '二', 'three': '三'}
12     while True:
13         value = que.get() # 从进程共享队列获取到数据
14         print(value, ': ', dict[value], sep='')
15
```

```

16 if __name__ == '__main__':
17     que = multiprocessing.Queue() # 创建进程共享队列
18     process1 = multiprocessing.Process(target=ocr, args=(que,))
19     process2 = multiprocessing.Process(target=dict, args=(que,))
20     process1.start()
21     process2.start()

```

运行代码，我们可以看到如下结果：

完成取词...

one: 一

完成取词...

two: 二

完成取词...

three: 三

看似没有问题，但是，大家注意程序没有结束。

因为，dict函数里面是死循环。

所以，我们需要结束这个进程。

继续添加一句代码：

```

1 process2.terminate() # 终止进程

```

运行程序，问题随之而来。显示输出结果如下：

完成取词...

完成取词...

完成取词...

这是因为进程process2还没来得及对process1进程送出的数据进行处理，就被终止了。

那么，能不能等process1的所有数据都送出，再结束进程process2呢？

在终止process2进程的代码之前，我们再加入一句代码。

```

1 process1.join() # 等待进程process1结束

```

join()方法能够阻塞主进程，等待调用该方法的子进程执行结束,再释放主进程的阻塞。

所以，当加入这一句代码，就能够等待process1进程执行结束后，再执行终止process2进程的代码。

最后，在Python中开启多进程，我们还可以使用进程池（Pool）。

例如，开启多个进程，分别运行不同的函数。

示例代码：

```

1 import multiprocessing, time

```

```

2
3 def task01(name):
4     print(name)
5     time.sleep(0.01)
6     print(name)
7
8 def task02(name):
9     print(name)
10    time.sleep(0.01)
11    print(name)
12
13 def task03(name):
14     print(name)
15     time.sleep(0.01)
16     print(name)
17
18 if __name__ == '__main__':
19     task_lst = [task01, task02, task03] # 创建需运行函数的列表。
20     processes = multiprocessing.Pool(3) # 创建进程池对象，设定并行最大进程数为3个。
21     for i in range(len(task_lst)):
22         processes.apply_async(task_lst[i], args=('进程' + str(i + 1),))
23         # 维持执行的进程总数为3个，当一个进程执行完毕后会添加新的进程进去。
24     processes.close() # 关闭进程池对象，禁止再加入新的进程。
25     processes.join() # 等待所有子进程结束。

```

注意：进程池必须先关闭，再等待所有子进程结束。否则，会发生错误。因为，关闭了进程池，不再有新的进程加入，才能知道需要等待结束的子进程共有哪些。

多线程

我们通过进程的学习，能够知道一个进程是一个任务。

那么，线程又是什么？

线程的作用执行进程的任务。

实际上，每一个进程的启动，都至少包含了一个线程。

这就好像一家食品公司（进程），至少得有一条生产线（线程），食品公司（进程）要生产食品（任务），但是实际上进行生产（执行任务）的是这家公司的生产线（线程）。

我们可以通过代码，验证一下这个概念。

示例代码：

```
1 import multiprocessing, threading
2
3 print(multiprocessing.current_process().name) # 显示输出结果为: MainProcess
4 print(threading.current_thread().name) # 显示输出结果为: MainThread
```

通过运行上方代码，我们能够看到主进程和主线程的名称。

也就意味着，程序执行的时候，启动了主进程，主进程启动了主线程。

接下来是多线程。

还是以食品公司举例。

速冻食品公司不但能生产饺子，还能够生产馄饨。

假如这时来了多个订单（任务），有些订单是饺子，有些订单是馄饨。

虽然一条生产线能够生产这两种食品，但是必须先生产完其中一种，再生产另外一种。

那么，当有多个订单要同时进行的时候，怎么解决呢？

一种方法，我们可以把公司（主进程）变成集团，在集团下开设多个分公司（子进程），每个公司建立一条生产线（主线程）。

另一种方法，我们可以把食品公司（主进程）的生产线升级生产组（主线程），在生产组内建立多条生产线（子线程）。

还有一种方法，我们可以把公司（主进程）变成集团，在集团下开设多个分公司（子进程），每个公司设立生产组（主线程），每个生产组建立多条生产线（子线程）。

简单来说，在程序中多任务的实现有3种方式：

- 多进程单线程
- 单进程多线程
- 多进程多线程

接下来，我们通过代码看一下如何通过多线程执行多任务。

示例代码：

```
1 import time, threading # 导入需要使用的模块
2
3 def task01(name):
4     print(threading.current_thread().name) # 显示输出结果为: Thread-1
5     print(name, 1)
6     time.sleep(0.001)
7     print(name, 2)
8     time.sleep(0.001)
9     print(name, 3)
10
11 def task02(name):
12     print(threading.current_thread().name) # 显示输出结果为: Thread-1
```

```

13     print(name, 1)
14     time.sleep(0.001)
15     print(name, 2)
16     time.sleep(0.001)
17     print(name, 3)
18
19 if __name__ == '__main__':
20     print(threading.current_thread().name) # 显示输出结果为: MainProcess
21     thread1 = threading.Thread(target=task01, args=('函数01: ',)) # 创建子线程运行函数
    task01
22     thread2 = threading.Thread(target=task02, args=('函数02: ',)) # 创建子线程运行函数
    task02
23     thread1.start() # 启动子进程
24     thread2.start() # 启动子进程

```

运行上方代码，显示输出结果类似：

MainThread

Thread-1

函数01: 1

Thread-2

函数01: 2

函数02: 1

函数01: 3

函数02: 2

函数02: 3

这里，大家能够看出，这段代码和之前我们通过多进程实现多任务的代码非常相像，只是把进程模块换为了线程模块，创建进程变成了创建线程。

不过，多进程和多线程是有区别的。

多进程中对于同一个变量，各有一份副本在进程中，所以对同一个变量的操作互不影响。

多线程中对于同一个变量，是各个线程共享的，所以对同一个变量，每个线程都能够进行操作。

我们通过代码来尝试一下。

示例代码：（多进程）

```

1 import multiprocessing
2
3 count = 0
4 def task01():
5     print(multiprocessing.current_process().name) # 显示输出结果为: Process-1
6     global count
7     count += 1

```



```

8     print(count) # 显示输出结果为: 1
9
10    def task02():
11        print(multiprocessing.current_process().name) # 显示输出结果为: Process-2
12        global count
13        count += 1
14        print(count) # 显示输出结果为: 1
15
16    if __name__ == '__main__':
17        process1 = multiprocessing.Process(target=task01)
18        process2 = multiprocessing.Process(target=task02)
19        process1.start()
20        process2.start()

```

示例代码：（多线程）

```

1    import threading
2
3    count = 0
4    def task01():
5        print(threading.current_thread().name) # 显示输出结果为: Thread-1
6        global count
7        count+=1
8        print(count) # 显示输出结果为: 1
9
10   def task02():
11       print(threading.current_thread().name) # 显示输出结果为: Thread-2
12       global count
13       count+=1
14       print(count) # 显示输出结果为: 2
15
16   if __name__ == '__main__':
17       thread1 = threading.Thread(target=task01)
18       thread2 = threading.Thread(target=task02)
19       thread1.start()
20       thread2.start()

```

通过段代码显示输出结果的对比，就能够看出不同。

而且，因为这个原因，使用多线程对同一个变量进行操作时，还容易出现意料之外的错误。

例如下面这段代码。

示例代码：

```
1 import threading
2
3 count = 0
4 def task01(proc):
5     global count
6     for i in range(500000):
7         count += 1
8         count -= 1
9
10 def task02(proc):
11     global count
12     for i in range(500000):
13         count += 1
14         count -= 1
15
16 if __name__ == '__main__':
17     thread1=threading.Thread(target=task01,args=('线程1',))
18     thread2=threading.Thread(target=task02,args=('线程2',))
19     thread1.start()
20     thread2.start()
21     thread1.join()
22     thread2.join()
23
24 print(count)
```

按代码里面的内容推断，代码运行结束时，全局变量count的值应该为0。

但是，实际上你会发现，运行之后结果不是固定的。

这是因为两个线程同时操作一个变量所导致的问题。

其中一种情况就是，thread1在执行了加法语句之后，尚未执行减法语句时，thread2的加法语句被执行了。

那后面不还是会执行两个减法语句吗？

看似是这个道理，但是实际上“count += 1”这个语句，在给变量count赋值的时候，并不是直接把计算结果写入变量count，而是先把“count+1”的计算结果写入到一个临时的局部变量，再把这个临时局部变量的值写入全局变量count。

过程是这样的（count初始值为0）：

1. thread1: temp1 = count + 1 # 此时temp1为1，count为0。

2. thread2: temp2 = count + 1 # 此时temp2为1, count为0
3. thread1: count = temp1 # 此时count为1。
4. thread2: count = temp2 # 此时count为1。
5. thread1: temp1 = count - 1 # 此时temp1为0, count为1。
6. thread1: count = temp1 # 此时count为0。
7. thread2: temp2 = count - 1 # 此时temp2为-1, count=0。
8. thread2: count = temp2 # 此时count为-1。

这样的错误, 在代码运行足够的次数后就会发生。

那么, 如何避免这种错误呢?

我们通过Lock类来解决。

Lock类是实现原始锁对象的类。一旦一个线程获得了一个锁, 就会执行锁之后的语句块, 直到锁被释放; 锁的释放可以在任何线程中执行。

示例代码: (以task01函数为例)

```
1 lock = threading.Lock() # 创建锁
2 def task01(proc):
3     global count
4     for i in range(500000):
5         lock.acquire() # 加锁
6         count += 1
7         count -= 1
8         lock.release() # 解锁
```

在上方代码中, 创建一个锁的对象, 并且为函数task01和task02都添加acquire()获得锁和release()释放锁的语句, 就可以避免之前的错误出现。

加锁固然能够让某一段代码在执行完毕前不被其它线程所干扰, 但是这样也导致其它线程不能并发执行, 从而降低了代码的效率。

就拿上方代码来说, 大家应该可以感受到加锁后代码的运行速度比加锁前慢了许多。

如果感受不明显, 可以把循环次数再增加10倍试一试。

最后, 大家还要知道, 在Python解释器执行代码时, 也有一个锁叫Global Interpreter Lock (简称GIL: 全局解释器锁), 这个锁导致任何Python线程执行之前, 必须先开启GIL锁, 每执行一定数量的代码后, 再自动关闭GIL锁, 让其它线程能够执行。这个全局解释器锁的机制导致Python中的线程不是真正的并发, 而是轮流执行, 所以, 再多的线程也是一个CPU核心在执行, 无法使用多个CPU核心。这也就意味着多核CPU执行多线程的Python代码时, 无法发挥多核的性能。如果想有效利用CPU的多个核心, 可以在Python的代码中使用多进程。多进程虽然也有多个线程, 但是各个进程的GIL锁是独立的, 不会互相影响。

协程 (Coroutine)

前面我们学习了进程和线程, 它们的调度由系统决定。

我们也知道Python中的多线程是抢占式调度，在线程切换执行的过程中，每个线程执行的代码数量和时间都是由系统决定，编程人员无法控制。

如果是CPU密集型的程序，涉及大量计算，要求计算速度，多线程切换的耗时（需要保存当前线程状态，下次执行时再取回状态）带来的开销，反而不如单线程执行速度更快。

而I/O密集型程序选择多线程会比较高效，因为I/O密集型程序（例如Web应用）往往需要等待I/O操作完成，这时线程切换的耗时不会带来什么影响（某一线程等待，就跳转到另一线程执行），反而任务越多越显得高效。

但是，多线程也有很明显的缺点，就是多个线程执行的程序之间需要协作的话，会非常麻烦。大家可以回想一下多线程对同一变量的操作。

一个线程实际上就是执行一个子程序，也就是执行一个函数。

如果不好理解，回想一下创建多线程的语句，Thread的参数target就是一个函数名称。

那么，既然知道线程之间子程序的协作有很明显的弊端，怎么解决它呢？

这里就可以用到我们这篇教程的学习目标-协程。

如果想理解协程，建议记住以下几点：

- 协程是在单线程中子程序之间的协同工作，没有切换线程的开销，效率高于多线程。
- 协程中不存在同时写入变量的冲突，控制共享资源无需加锁。
- 协程不是函数调用。（这点很重要，下面会详细解释。）
- 协程需要使用生成器来实现。（忘了的话，请回顾《[Python3萌新入门笔记（32）](#)》）
- 协程可以通过异步I/O实现多任务并发。

首先，我们先来区分协程和函数调用。

先来看一个例子。

首先，函数run()通过循环得到数字；

然后，函数fun()负责将产生的数字加1；

最后，再由函数run()显示输出计算后的结果。

示例代码：（函数调用）

```
1 import time
2
3 def fun(i):
4     print('接到任务...')
5     time.sleep(1)
6     i += 1
7     print('完成任务...')
8     return i
9
10 def run():
11     for i in range(3):
12         num = fun(i) # 调用函数
13         print('当前数字: ', num)
14
15 if __name__ == '__main__':
```

示例代码：（协程）

```

1  import time
2
3  def fun():
4      i = 0
5      while True:
6          value = yield i
7          if value is not None:
8              print('接到任务...')
9              time.sleep(1)
10             i = value + 1
11             print('完成任务...')
12
13 def run(cor):
14     next(cor)
15     for i in range(3):
16         num = cor.send(i) # 执行其它子程序
17         print('当前数字: ', num)
18     cor.close()
19
20 if __name__ == '__main__':
21     cor = fun()
22     run(cor)

```

大家对比一下两段代码，就能够看出区别。

调用函数：是在函数run()的执行过程中调用了函数fun()，当函数fun()运行结束后，又回到函数run()继续执行。所以，两个函数并没有同时运行，只是调用没有协作。

协程：是先挂起（启动并等待）生成器fun()，然后执行run()函数，在函数run()的执行过程中，通过send()方法执行一次生成器的计算代码，再继续执行。在这个过程中，函数fun()和函数run()同时都在运行，形成了协作关系。通过理解协程的工作过程，大家能够看到协程中什么时候执行哪一个子程序的代码以及执行哪些代码都是可以由编程人员控制的。

所以，协程的**是一种用户态的轻量级线程**。

提示：这个概念很硬，可以延伸了解用户态、内核态、系统空间和用户空间这些名词的含义。

有了生成器才有了协程，因为生成器能够挂起，所以就有了多个子程序同时运行的可能。

而且，因为生成器的send()方法，能够将数据发送到生成器，并且让生成器的执行指定的代码，才实现了子程序之间的协作。

后来，随着Python版本的更新，又出现了yield from表达式（Python3.3）。

yield from表达式的作用，官方文档是这么描述的：

允许生成器将其部分操作委托给另一个生成器。也就是允许一段包含yield的代码被分解出来放到另一个生成器中。此外，子生成器允许返回一个值，该值可用于委托生成器。

从这段描述可以看出，通过yield from表达式，我们可以获取一个生成器的部分操作和它的返回值，并在此基础上进行重构。

来看一段获取一定数量奇数的示例代码。

示例代码：

```
1  def odds(max): # 定义子生成器
2      odd = 1 # 起始奇数
3      count = 0 # 获取次数
4      while odd <= max: # 循环获取到最大奇数值
5          value = yield odd # 生成奇数
6          odd += 2 # 计算下一个奇数
7          count += 1 # 生成次数递增
8          if value is None or True: # 获取外部传入的值
9              print('第', count, '生成成功...')
10     return count # 返回生成次数
11
12 def copy_odds(max): # 定义委托生成器
13     print('-' * 8, '开始', '-' * 8)
14     count = yield from odds(max) # 获取生成器部分操作以及返回值
15     print('-' * 8, '完成', '-' * 8)
16     print('共生成了', count, '个奇数。')
17
18 if __name__ == '__main__':
19     gen = copy_odds(9) # 创建生成器对象
20     print(gen.send(None)) # 挂起生成器
21     while True:
22         try:
23             print(gen.send(True)) # 运行一次生成器并将值传入
24         except: # 迭代结束跳出循环
25             break
```

通过上方代码，我们可以更直观的看出yield from表达式的作用：

- 可以获取子生成器的部分操作创建委托生成器；
- 可以获取子生成器的返回值；
- 可以将send信息传递给子生成器，并且处理好了各种异常情况。

接下来，我们来了解协程结合异步I/O实现多任务并发。

asyncio是一个基于事件循环实现异步I/O的模块。

提示：事件循环可以简单的理解为等待程序分配事件并进行相应的处理，它能同时处理多个事件。

如何使用asyncio模块，我们先来看一段代码。

示例代码：

```
1 import asyncio
2
3 @asyncio.coroutine # 装饰函数为协程
4 def fun(i): # 定义协程函数
5     while True:
6         print('接到任务...')
7         yield from asyncio.sleep(1) # 挂起当前协程，获得内层协程的Future对象，并执行内层协程到完成。
8         i += 1
9         print('完成任务...')
10        return i
11
12 @asyncio.coroutine # 装饰函数为协程
13 def run(): # 定义协程函数
14     for i in range(3):
15         num = yield from fun(i) # 挂起当前协程，获得内层协程的Future对象，并执行内层协程到完成。
16         print('当前数字: ', num)
17
18 if __name__ == '__main__':
19     loop = asyncio.get_event_loop() # 创建循环事件
20     tasks = [] # 创建任务列表
21     for i in range(2):
22         tasks.append(run()) # 添加任务到任务列表
23     loop.run_until_complete(asyncio.wait(tasks)) # 循环运行直到完成（等待任务列表中的协程和Future对象执行完成）
24     loop.close() # 关闭循环事件
```

上方代码和我们第一段协程示例代码的功能相同。

这里有一些关键内容需要理解（方法与函数仅列出部分主要参数）。

- @asyncio.coroutine装饰器：根据官方文档的描述，基于生成器的协程函数应该使用@asyncio.coroutine进行装饰，虽然并不需要严格执行。所以，你会发现，即便没有装饰程序依然正常。
- asyncio.sleep(delay)：创建一个协程并在指定时间完成，参数delay为秒数。
- asyncio.get_event_loop()：获取当前上下文的事件循环对象。

- `asyncio.wait(futures)`: 等待指定Future对象序列中的Future对象和协程对象执行完成。协程将被包含到任务(Task) 中, 返回两种Future对象(完成的和即将执行的)。
- Task: 安排一个协程的执行: 把它包含到一个Future对象中。Task是Future的一个子类。一个任务对象(Task) 是负责在事件循环执行一个协程对象。
- `run_until_complete`: 运行所有的Future对象直到完成。

接下来, 我们来看一下这段代码的执行过程。

- 1、创建了事件循环对象;
- 2、创建了任务列表;
- 3、运行任务列表中的任务, 直到所有任务结束。
- 4、关闭事件循环。

整个过程中的第3步, 其实非常复杂, 我们分解来看。

注意: 任务列表中是2个任务。

首先, 事件循环会同时执行任务1 (`tasks[0]`) 和任务2 (`tasks[1]`) ;

然后, 以执行任务1为例。

- 先执行任务1的协程`run()`, 当执行到`yield from`表达式, 开始执行协程`fun()`, 此时会返回一个Future对象 (假设叫F1) 传递给事件循环, 事件循环开始一直监视F1, 并暂停协程`run()`;
- 执行协程`fun()`的过程中又遇到`yield from`表达式, 开始执行协程`asyncio.sleep(1)`, 这时也会返回一个 Future对象 (假设叫F2) 并将其传递给事件循环, 事件循环开始一直监控这个F2 (同时也在监视F1), 同时暂停协程`fun()`的执行。
- 1秒钟之后, 协程`asyncio.sleep()`执行结束, 事件循环会选择F2以及被暂停协程`fun()`, 将F2的结果返回给协程`fun()`, 然后协程`fun()`继续执行到结束。
- 协程`fun()`执行结束后, 事件循环选择F1以及被暂停协程`run()`, 将F1的结果返回给协程`run()`, 然后将协程`run()`继续执行。
- 因为协程`run()`会循环执行3次`yield from`表达式, 所以上面的过程也会重复执行3次。

通过`yield from`表达式, 我们可以将协程的控制权交给事件循环, 然后挂起当前协程, 进入内层协程; 并且, 由事件循环决定何时返回当前协程接着向后执行代码。

以上就是通过协程和异步I/O实现多任务并发。

不过, Python 3.5版本中, 出现了新的语法`async`和`await`。

通过`async`语法定义的函数即为协程, 无需通过`@asyncio.coroutine`装饰器进行装饰。

通过`await`语法等待内层协程的执行完成。

所以, 之前协程函数的代码可以使用新语法来编写。

示例代码:

```

1  async def fun(i):  # 定义协程函数
2      while True:
3          print('接到任务...')
4          await asyncio.sleep(1)  # 等待内层协程执行完成
5          i += 1
6          print('完成任务...')
7          return i
8

```



```
9  async def run(): # 定义协程函数
10      for i in range(3):
11          num = await fun(i) # 等待内层协程执行完成
12          print('当前数字: ', num)
```

如果不需要兼容旧版本的Python，官方文档中建议使用新的语法。

Python网络编程中涉及的一些概念

我们知道如果两个人互相邮寄物品，都需要知道对方的地址和姓名。

寄送物品的地址能够保证物品运输到的具体位置，例如某个学校的某个班级。

寄送物品的姓名能够保证把物品交给某个人。

网络编程中也一样，两台计算机（或虚拟机）之间进行通信传递数据的话，就需要知道对方的通信地址和端口号。

通信地址能够保证将数据送达到网络中的某一台设备。

端口号能够保证将数据送达到设备中的某个程序。

这里所说的通信地址是指计算机设备在网络中的IP地址或IP地址绑定的域名。

IP地址是一个32位的二进制整数，按8位划分后，每一段都转换为十进制数。例如192.168.1.33，二进制整数是11000000101010000000000100100001。

而端口号是端口的名称，端口是计算机设备的系统中某一程序绑定的端口或系统为程序随机分配的端口。

一般来说，一个服务端能够和多个客户端进行通信。

例如，本站点的HTTP服务器和访问本站点的用户所使用的计算机上的浏览器。

那么，服务端需要指定一个固定IP地址或与这个IP地址绑定的域名，例如：本站域名www.opython.com。

还需要一个固定的端口供客户端请求访问，例如：HTTP服务端口默认为80。

而客户端在访问服务端的时候，计算机设备中的系统会分配一个可用的随机端口供浏览器进行通信，服务端会在客户端发起通信请求后，获取到客户端的IP地址和端口号，并通过这个IP地址将数据发送给客户端的计算机，再根据端口号传送到浏览器。

除了需要知道IP和端口号，我们还要了解一下TCP/IP协议和UDP协议。

TCP/IP全称是Transmission Control Protocol/Internet Protocol，中文译名为传输控制协议/因特网互联协议。

这两个协议有什么用处呢？

简单来说，IP协议负责传输数据，TCP协议负责控制传输的数据，所以IP协议是TCP协议的底层协议。

还是用邮寄物品举例。

我们想寄送多件物品给对方，但是这些物品收货方要求按顺序寄到，并且保证全部送达。

首先，如果想进行邮寄，快递公司会要求必须有寄送地址和收件人姓名，否则就不能帮你运送物品，这实际上就是确定了运送物品的协议。只有遵循这样的协议要求，才能够进行物品的邮寄。

IP协议就是类似这样的一个传输协议，它负责选择网络线路建立连接，并且把数据分割为数据包进行传输，但是，它不保证物品能够按照顺序全部送达。

就像快递公司一样，虽然接收了物品，但是有可能丢掉快件，并且派送也是无序的。

为了解决这个问题，我们可以和收货方协定，发货方把寄送的物品全部进行顺序编号，按顺序发出，收货方收到物品时，将收货回执给快递公司送回发货方。当发送一件物品之后，如果48小时内收到收货方的回执，就继续发出下一件，否则，认为快件丢失，马上补发。通过这样的协定，就能够保证所有物品能够有序完整的寄送给对方。

TCP协议就是类似这样的一个对传输数据进行控制的协议。为了保证不发生丢失数据包的现象，就给每个数据包一个序号，当成功接收数据包，接收端会发回一个相应的确认；如果发送端在合理的往返时延内未收到确认，那么就假设

对应的数据包已丢失，将会进行重发。

基于TCP/IP的特点，比较适合在数据安全性要求较高的网络编程中使用。例如，文件的传输。

然后，我们再了解UDP。

UDP协议全称是User Datagram Protocol，中文译名为用户数据报协议。

它是一种不可靠的数据报传输协议。

它可以简单的理解为仅仅将要发送的数据传送至网络，并接收网络传回的反馈，而不与接收端建立连接。

也举个例子来说明。

例如，某公司为员工在网上为每人定了一份订餐，然后一起赶火车，因为时间紧急，要求快速送达。

快餐店是做好快餐打包装好交给送餐小哥，送餐小哥送餐路上路况不好，不小心丢了几份快餐，但好在按时送到。订餐的公司发现快餐少了，这时让快餐店重送也来不及，只好有多少收多少，没有收到的部分，可以在附近的商店临时买些面包解决。

在这个例子中，快餐店就是UDP协议中的发送端，送餐小哥是IP协议，订餐公司是UDP协议的接收端。

快餐店只需要把足量的快餐给送餐小哥，得到送餐小哥的确认，就不再负责后面的事情，至于订餐公司是否收到或者能不能全部收到不是快餐店负责的。

送餐小哥负责送达，但不能保证不发生意外，迟到了或者丢了几份快餐都有可能。

很明显，UDP存在不安全的风险，但是它也有速度快的特点，就好像快餐店不需要为每一份快餐编号，还要确认送到一份再发下一份。

所以，基于UDP协议的特点，比较适合对数据完整性要求较低（或是可以保障可靠性），但对传输即时性要求较高的网络编程中使用。例如，网络视频聊天的数据传输，在网络较差时丢失一些画面是可以接受的。

最后，我们要了解的一部分内容是套接字（Socket）的概念。

Socket的中文翻译是“插座”，通常也称作“套接字”，用于描述IP地址和端口。

应用程序能够通过套接字向网络发出请求或者应答网络请求。

套接字包括两个：服务端套接字和客户端套接字

当我们创建一个服务端套接字之后，它就开始在网络地址中（服务器的IP地址+端口号）进行监听，随时处理来自客户端的连接。

因为来自客户端的连接可能是多个，所以处理服务端的套接字比处理客户端的套接字复杂。

处理客户端的套接字，只是建立连接，处理事务和关闭连接。

以上就是关于Python网络编程中会涉及的一些概念性的知识，需要大家进行了解。

当然，只了解这些是远远不够的，建议大家从网上学习更多相关的专业知识内容。

网络编程相关的一些模块

一、socket模块

通过socket模块，我们能够创建服务器和客户端。

这里我们需要使用socket模块中的socket()函数，这个函数能够使用给定的地址族、套接字类型和协议号创建一个新的套接字。

- 地址族：默认为socket.AF_INET，指定使用IPv4 网络协议；如果填入socket.AF_INET6，则能够使用IPv6 网络协议。
- 套接字类型：默认为socket.SOCK_STREAM，指定使用TCP协议的套接字流类型；如果填入socket.SOCK_DGRAM，则是使用UDP的套接字数据报类型。
- 协议号：默认为0。

创建一个普通的套接字，可以省略所有参数。

当创建了一个套接字，就需要使用bind()方法为其绑定主机地址和端口号，以便能够进行连接。

并且，我们还需要通过listen()方法，指定套接字最大等待连接数。也就是客户端与服务器连接时，允许有多少个客户端同时等待连接。当等待连接的数量超出最大限制数量时，后发起连接的客户端会被拒绝连接。

服务器与客户端的连接通过accept()方法完成，这个方法能够返回一个SSL（Secure Sockets Layer 安全套接层）通道和客户端的IP地址。

通过SSL通道，我们可以实现服务器和客户端的通信。

示例代码：（服务器）

```
1 import socket
2
3 skt = socket.socket() # 创建套接字对象
4 host = '127.0.0.1' # 指定主机IP地址
5 port = 6666 # 指定连接端口号
6 skt.bind((host, port)) # 为套接字对象绑定主机名称与端口号
7 skt.listen(3) # 设置套接字最大等待连接数
8 while True:
9     print('等待连接...')
10    ssl, addr = skt.accept() # 获取服务器端的SSL通道和远程客户端的地址
11    print('连接到: ', addr)
12    cname = ssl.recv(1024).decode() # 获取客户端发来的信息
13    ssl.send('欢迎来自{0}的{1}'.format(addr, cname).encode()) # 通过SSL通道发送信息
14    print('关闭连接...')
15    ssl.close() # 关闭SSL通道
```

如上方代码所示，一个简单的服务器，我们只需要以下几个步骤：

- 创建套接字对象：socket()
- 绑定主机与端口：bind()
- 设置等待连接数：listen()
- 接收客户端连接：accept()

在此基础上，就能够进行更多的操作。例如，示例代码中通过send()方法向客户端发送信息、通过recv()方法接收服务器发送的信息以及通过close()方法关闭与客户端的连接。

在使用send()方法和recv()方法时，注意进行编解码的操作。

send()方法参数为发送的数据。

recv()方法的参数是一次读取的字节数量，如果没有特别的要求或无法确定，不妨就写1024。

但是，只有服务器的话，我们无法验证这个服务器能否正常的工作。

我们再来创建一个客户端。

相对于服务器，客户端的创建非常简单，只需要创建一个套接字，然后通过connect()方法向服务器请求连接。

当连接到服务器之后，即可与服务器进行通信。

示例代码：（客户端）

```

1 import socket
2
3 skt = socket.socket() # 创建套接字对象
4 host = '127.0.0.1' # 指定要连接到的主机IP地址
5 port = 6666 # 指定连接端口号
6 skt.connect((host, port)) # 发起连接
7 print('接收信息: ', skt.recv(1024).decode()) # 接收来自服务器端的信息

```

以上两段代码，创建了简单的服务器和客户端。

我们可以运行服务器，然后运行多个客户端进行测试。

在PyCharm中，可以在客户端的编辑区多次按下快捷键<Ctrl>+<Shift>+<F10>启动多个客户端。

当服务器启动之后，会显示“等待连接...”。

当第一个客户端启动之后，会显示类似“连接到: ('127.0.0.1' , 57409)”的信息。

当完成数据的发送之后，会显示“关闭连接...”。

而能够与服务器正常连接的客户端，会显示类似“接收信息: 欢迎你, ('127.0.0.1' , 57337)”的信息（注意进行解码）。

不能够与服务器进行连接的客户端，会抛出连接拒绝异常“ConnectionRefusedError: [WinError 10061] 由于目标计算机积极拒绝，无法连接。”。

二、socketserver模块

除了使用socket模块能够创建服务器，通过socketserver模块也能够创建服务器。

socketserver模块的TCPServer类和UDPServer类分别针对TCP套接字流和UDP套接字数据报。

仍然以TCP服务器的创建举例。

在这个socketserver模块中有这样对请求进行处理的类。

如果使用socketserver模块创建一个服务器，我们需要创建一个对请求进行处理的类，继承StreamRequestHandler类，并重写请求处理方法handle()。

这样，当服务器接收到一个客户端连接发来的请求时，就实例化一个对请求进行处理的类，并调用实例中的请求处理方法进行处理。

另外，当服务器与客户端进行通信时，我们可以使用以下方法：

- self.request.recv(): 接收客户端连接请求中发来的信息
- self.rfile.readline(): 读取客户端发来的信息
- self.request.send(): 向客户端连接请求发送信息（可能多次发送，直至完成）
- self.request.sendall(): 向客户端连接请求发送信息（一次全部发送）
- self.wfile.write(): 向客户端写入并发送信息

rfile和wfile是StreamRequestHandler类的两个属性，通过这两个属性能够进行写入与读取，所以通过这两个类文件对象（file-like Object）就能够实现服务器和客户端的通信。

提示：Python中的类文件对象（file-like object）是指实现了read()方法或write()的对象。根据创建的方式，一个文件对象可以是一个真正的磁盘文件，也可以是对存储或通信设备的访问（例如标准输入/输出、内存缓冲区、套接字、管道等）。所以，文件对象也可称为类文件对象或流。

示例代码：（服务端）

```

1 from socketserver import TCPServer, StreamRequestHandler
2
3 class Handler(StreamRequestHandler): # 定义类继承对数据流请求处理的类
4     def handle(self): # 定义处理方法
5         addr = self.client_address # 获取客户端地址
6         print('处理来自%s的连接...' % (addr,))
7         cname = self.request.recv(1024).decode() # 获取客户端发来的信息
8         # cname=self.rfile.readline().decode() # 获取客户端发来的信息
9         self.request.sendall('欢迎来自{0}的{1}'.format(addr, cname).encode()) # 向客户端
        发送信息
10        # self.request.send('欢迎来自{0}的{1}'.format(addr, cname).encode()) # 向客户端
        发送信息
11        # self.wfile.write('欢迎来自{0}的{1}'.format(addr, cname).encode()) # 向客户端发
        送信息
12
13 server = TCPServer(('', 6666), Handler) # 实例化TCP服务器对象
14 server.serve_forever() # 持久运行服务器

```

不过，前两个服务端示例代码我们通过测试能够知道，多个客户端与服务器的连接是按顺序进行的。

在一个客户端的连接未执行结束之前，无法进行下一个客户端的连接。

这样的服务器端编程称为阻塞或同步网络编程。

那么，如何实现非阻塞的网络编程，或者说异步网络编程呢？

对于通过socketserver模块创建的服务器，我们可以使用多线程实现多连接的处理。

通过创建一个新的服务器类，继承ThreadingMixIn类和 TCPServer类，就能够完成多连接并发的处理。

示例代码：

```

1 from socketserver import TCPServer, StreamRequestHandler, ThreadingMixIn # 导入
    ThreadingMixIn类
2 import threading
3
4 class Server(ThreadingMixIn, TCPServer): #创建服务器类(注意继承顺序)
5     pass # 无需添加代码
6
7 class Handler(StreamRequestHandler):
8     def handle(self):
9         addr = self.client_address
10        print('线程%s处理来自%s的连接...' % (threading.current_thread().name, addr))
11        cname = self.request.recv(1024).decode()
12        self.request.sendall('欢迎来自{0}的{1}'.format(addr, cname).encode())
13

```

```
14
15 server = Server(('', 6666), Handler) # 实例化服务器对象
16 server.serve_forever()
```

三、select模块

通过socket模块创建的服务器，我们可以使用select模块来实现多连接并发的处理。

我们来看一个例子，这个例子实现下述功能：

- 客户端向服务器发送需要翻译的汉字，服务器端将相应的翻译结果（英文单词）发送回客户端。
- 如果没有相应的翻译结果，服务器端发回信息：未找到对应的英文单词！
- 如果客户端未发送的信息为空值或“exit”，能够关闭服务器。

首先，我们先编写客户端的代码。

示例代码：（客户端）

```
1 import socket
2
3 s = socket.socket() # 创建套接字
4 host = socket.gethostname()
5 port = 6666
6 s.connect((host, port)) # 向服务器端请求连接
7 while True:
8     char = input('输入中文: ') # 获取用户输入
9     s.send(char.encode()) # 编码后发送到服务器端
10    if char in ['exit', '']: # 如果发送的内容为"exit"或空值
11        break # 跳出循环，结束程序
12    print('英文单词: ', s.recv(1024).decode()) # 显示输出来自服务器端的信息
```

接下来，编写服务器端的代码。

先定义一个翻译功能的函数。

示例代码：（服务器端）

```
1 import socket, select
2
3 def translate(char): # 定义翻译功能的函数
4     my_dict = {'你': 'you', '我': 'me', '他': 'he'} # 定义可翻译的内容
5     if char not in my_dict: # 如果字典中不包含客户端发来的汉字
6         return '未找到对应的英文单词!' # 返回提示信息
7     else: # 否则
8         return my_dict[char] # 返回翻译结果
```


再创建套接字，绑定主机地址与端口，并设置最大等待连接数。

示例代码：

```
1 s = socket.socket() # 创建套接字
2 host = socket.gethostname() # 获取主机名
3 port = 6666 # 指定端口号
4 s.bind((host, port)) # 套接字绑定主机名与端口号
5 s.listen(5) # 指定最大等待连接数
```

然后，通过select模块实现异步I/O。

这里，我们使用select.select(rlist, wlist, xlist[, timeout])函数。

这个函数的三个必填参数均为序列，序列中为“等待的对象”，可选参数为数字：

- rlist: 一个列表，包含等待直到能够读取的对象；可以为空列表。
- wlist: 一个列表，包含等待直到能够写入的对象；可以为空列表。
- xlist: 一个列表，包含等待出现一个“异常状态”的对象；可以为空列表。
- timeout: 连接超时的时长，值为秒的浮点数；省略此参数时，函数将阻塞，直到至少一个“等待的对象”准备好；如果参数为0，则从不阻塞。

函数的返回值是一个元组，包含3个已经准备好的对象的列表，当没有准备好的对象或连接超时，则返回3个空列表。

示例代码：

```
1 inputs = [s] # 创建等待能够读取对象的列表，默认包含等待连接的套接字。
2 while True:
3     print('等待连接...')
4     rs, ws, xs = select.select(inputs, [], []) # 获取已经准备好的对象列表
5     for r in rs: # 遍历已经准备好的可读取对象列表
6         if r is s: # 如果列表元素是套接字对象
7             ssl, addr = r.accept() # 等待客户端连接
8             print('连接到: ', addr)
9             print(type(ssl))
10            inputs.append(ssl) # 连接成功后，将SSL通道对象添加到等待可读对象列表
11        else: # 否则(对象是SSL通道)
12            try:
13                char = r.recv(1024).decode() # 通过SSL通道接收来自客户端的信息
14                if char in ['', 'exit']: # 判断信息内容是否空值或“exit”
15                    print(addr, '主动关闭连接! ')
16                    inputs.remove(r) # 从等待可读取对象的列表中移除SSL通道对象
17            else: # 否则
```

```

18         r.send(translate(char).encode()) # 通过SSL通道向客户端发送翻译结果
19         print('完成任务...')
20     except: # 捕获连接出现的异常
21         print(addr, '异常断开连接!')
22         inputs.remove(r) # 从等待可读取对象的列表中移除SSL通道对象

```

当我们启动上方的服务器和多个客户端，就能够看到任何一个客户端都能够即时与服务器进行连接，而无需等待前一个发起连接的客户端连接结束。

如何爬取网页内容

这里我们可以使用urllib这个模块。

基于前面的学习，对于模块中使用的一些方法、函数不再做具体说明，大家可以参考示例代码以及相应的注释进行理解。

另外，本篇内容需要读者具备一些HTML基础，可以自行查看一些相关资料，推荐学习[W3school](http://www.w3school.com.cn/)上的资料。

接下来，我们先来看一个案例。

例如，我发现了一个古诗文网，里面有一个唐诗三百首的页面。

<http://so.gushiwen.org/gushi/tangshi.aspx>

这个页面中包含了唐诗三百的目录，每一首诗都有一个单独的页面。

当我们在浏览器中查看这个页面的源代码，就能够看到目录所对应的源代码内容。

如果想爬取每一个古诗的页面，我们需要先获取到<a>标签中“href”属性的内容，然后连接“<http://so.gushiwen.org>”组成页面的绝对路径。

这里，我们可以创建一个生成器，能够生成所有古诗页面路径。

这个生成器我们需要包含两个步骤：获取目录页面的全部内容和获取<a>标签中的相对路径内容。

示例代码：（获取页面路径）

```

1  from urllib import request
2  import re
3
4  def get_urls():
5      pages = request.urlopen('
        http://so.gushiwen.org/gushi/tangshi.aspx
        pages = request.urlopen('
6      for line in pages: # 遍历页面内容的每一行HTML代码
7          rslt = re.search(r'<span><a href="(.)+" target="_blank">', line.decode())
8          # 通过正则表达式获取每首古诗页面的相对路径，保存到组。
9          if rslt: # 页面的HTML代码与正则表达式匹配
10             url = '
                http://so.gushiwen.org

```



```
url = '  
11     yield url # 生成绝对路径
```

完成了每首古诗页面绝对路径的生成，接下来就可以进行古诗内容的抓取。

我们定义一个抓取古诗页面内容并写入文件的方法。

示例代码：（抓取写入文件）

```
1 def write_txt(urls, file): # 定义抓取内容并写入文件的方法，参数为绝对路径的生成器对象和打开  
   的文件对象。  
2     count = 0 # 创建计数变量  
3     for url in urls: # 遍历所有的绝对路径  
4         count += 1  
5         file.write(str(count) + '\n') # 向文件中写入抓取的序号  
6         html = request.urlopen(url) # 请求打开古诗页面的链接，获取页面内容。  
7         for line in html: # 遍历页面内容的每一行HTML代码  
8             title = re.search('<h1.+>(.)</h1>', line.decode()) # 通过正则表达式获取古诗  
   名称  
9             source = re.search(r'.+source.+>(.)</a>.+<a href=".">(.)</a>',  
   line.decode())  
10            # 通过正则表达式获取年代与作者  
11            contson = re.search(r'([</>\w]+<br />[</>\w]+)', line.decode()) # 通过  
   正则表达式获取古诗正文  
12            if title: # 如果获取到标题  
13                file.write(title.group(1) + '\n') # 向文件中写入古诗名称  
14            if source: # 如果获取到古诗年代与作者  
15                file.write(source.group(1) + ': ' + source.group(2) + '\n') # 向文件中  
   写入年代与作者  
16            if contson: # 如果获取到古诗正文  
17                file.write(contson.group(1).replace('<br />', '\n').strip('\n') +  
   '\n\n')  
18                # 向文件中写入古诗正文  
19                break # 写入正文后跳出当前页面的读取
```

有了以上代码，我们就能够对目标页面进行抓取了。

示例代码：

```
1 if __name__ == '__main__':  
2     with open('gushi.txt', 'w') as file: # 写入模式打开文件创建文件对象
```

```
3         urls = get_urls() # 创建生成器对象
4         write_txt(urls, file) # 调用抓取页面并写入文件的方法，将生成器对象和文件对象作为参数
    传入。
```

测试结果应该没有什么问题。

不过，上面抓取写入文件的代码中，正则表达式的可读性很差。

如果抓取一些结构更加复杂的网页，使用正则表达式也会变得非常困难。

那有没有更好的方法呢？

我们知道，在HTML中标签一般都是成对出现的，例如：

- 页面标签：开始标签<html>和结束标签</html>
- 主体标签：开始标签<body>和结束标签</body>
- 层的标签：开始标签<div>和结束标签</div>
- 段落标签：开始标签<p>和结束标签</p>
- 强调标签：开始标签和结束标签

当然，标签还有许多，也有一些自结束标签不是成对出现，例如图片标签。

我们一般抓取网页内容，实际上就是这些标签中的内容。

在Python中，提供了解析HTML的类HTMLParser，这个类存在于html.parser模块中。

HTMLParser类提供了一些方法，我们通过对这些方法进行重写，就能够满足抓取页面内容的需求。

这里，我们会用到其中的3个方法：

- handle_starttag：处理开始标签的方法
- handle_data：处理标签中数据的方法
- handle_endtag：处理结束标签的方法

handle_starttag方法会自动传入2个参数tag和attrs，我们能够通过这两个参数获取到当前所读取的标签名称与标签所包含的字段。

那么，就可以通过判断字段中是否包含指定的内容，来决定是否取出标签中的数据。

handle_data方法会自动传入1个data参数，这个参数中的内容就是当前所读取的标签中包含的内容，我们可以根据标签是否符合要求，来决定是否处理这个数据。

handle_endtag方法会传入1个tag参数，也就是结束标签的名称；我们可以在读取到指定结束标签时，进行结束数据读取的处理。

上面的解释不是很清晰，大家可以对照下方的示例代码进行理解，就能够非常清楚了。

在之前代码中，我们创建一个类继承HTMLParser类，并重写上述的3个方法。

示例代码：（创建解析器类）

```
1 from html.parser import HTMLParser
2
3 class MyHtmlParser(HTMLParser): # 创建一个类继承HTMLParser类
4     is_title = False # 定义添加古诗名称的开关变量
5     is_source = False # 定义添加古诗年代与作者的开关变量
6     is_contson = False # 定义添加古诗正文的开关变量
7     is_add = True # 定义是否添加当前页面数据的开关变量
```

```

8
9     def handle_starttag(self, tag, attrs): # 重写方法
10         attrs = dict(attrs) # 将参数转换为字典类型
11         if tag == 'h1': # 判断标签名称
12             self.is_title = True # 符合条件时开启添加古诗名称
13         if 'source' in attrs.values(): # 判断字典值中是否包含指定的值
14             self.is_source = True # 符合条件时开启添加古诗年代与作者
15         if 'contson' in attrs.values(): # 判断字典值中是否包含指定的值
16             self.is_contson = True # 符合条件时开启添加古诗正文
17         if 'tool' in attrs.values(): # 判断字典值中是否包含指定的值
18             self.is_add = False # 符合条件时关闭添加当前页面的数据
19         if tag == 'html': # 判断字典值中是否包含指定的值
20             self.is_add = True # 符合条件时开启添加当前页面的数据
21             self.content = [''] * 3 # 初始化保存某一页面内容的列表
22
23     def handle_endtag(self, tag): # 重写方法
24         if tag == 'h1': # 判断标签名称
25             self.is_title = False # 符合条件时关闭添加古诗名称
26         if tag == 'p': # 判断标签名称
27             self.is_source = False # 符合条件时关闭添加古诗年代与作者
28         if tag == 'div': # 判断标签名称
29             self.is_contson = False # 符合条件时关闭添加古诗正文
30
31     def handle_data(self, data): # 重写方法
32         data = data.replace('\n', '') # 替换掉数据中所有的换行符
33         if self.is_add: # 判断是否能够添加页面数据
34             if self.is_title: # 如果添加古诗名称为开启状态
35                 self.content[0] = data # 添加古诗名称到页面内容列表
36             if self.is_source: # 如果添加古诗年代与作者为开启状态
37                 self.content[1] = self.content[1] + data # 添加古诗年代与作者到页面内容列
38             if self.is_contson: # 如果添加古诗正文为开启状态
39                 self.content[2] = self.content[2] + data + '\n' # 添加古诗正文到页面内容

```

表

列表

在上方代码中handle_starttag()方法负责判断当前所读取的标签是否抓取目标，然后打开相应的开关。

并且，初始化保存抓取内容的列表。

同时，因为页面中还有与其它推荐的古诗，我们在抓取完第一首古诗之后，就不能再抓取当前页面的内容；所以，这个抓取页面内容的开关也放在handle_starttag()方法中进行开启与关闭。

- 当读取到<html>标签（页面内容的开始）时开启内容抓取；

- 当读取的标签属性值包含“tool”时，关闭内容抓取。（大家可以观察页面源代码，在第一首古诗的下方会出现这个“tool”值。）

然后，handle_endtag()方法则是根据读取到的结束标签，关闭相应的开关。

最后，handle_data()方法根据各个开关的状态，对数据进行处理。如果读取页面数据的开关为打开状态，并且某项数据对应的开关也是打开状态，则将该项数据写入页面内容列表的指定位置。

完成了上面类的创建之后，接下来，我们需要修改一个抓取与写入文件的方法。

示例代码：

```
1 def write_txt(urls, file, pars): # 定义抓取并写入文件的方法，增加解析器对象的参数
2     count = 0 # 创建计数变量
3     for url in urls: # 遍历所有的绝对路径
4         html = request.urlopen(url).read().decode().replace('<strong>',
5             '').replace('</strong>', '')
6         # 请求打开古诗页面的链接，获取页面内容。并且将页面中的<strong>标签替换为空值，因为它会
7         # 影响格式。
8         pars.feed(html) # 通过解析器对象解析页面代码
9         count += 1
10        file.write(str(count) + '\n') # 向文件中写入抓取的序号
11        file.write(pars.content[0] + '\n') # 向文件中写入古诗名称
12        file.write(pars.content[1] + '\n') # 向文件中写入年代与作者
13        file.write(pars.content[2] + '\n') # 向文件中写入古诗正文
```

完成了方法的修改，接下来我们就能够对目标页面进行抓取了。

示例代码：

```
1 if __name__ == '__main__':
2     with open('gushi.txt', 'w') as file: # 写入模式打开文件创建文件对象
3         urls = get_urls() # 创建生成器对象
4         pars = MyHtmlParser() # 创建解析器对象
5         write_txt(urls, file, pars) # 调用抓取页面并写入文件的方法，将生成器对象、文件对象
6         # 以及解析器对象作为参数传入。
7         pars.close() # 关闭解析器对象
```

本篇教程只是对页面内容抓取的入门内容，更深入的内容还需要查阅更多专业资料。

Python编程中有关配置、日志和测试的内容

有关这些内容，对于大多数人来说并不常用。

所以，我觉得只要有一些了解就可以了。

真正提高到一定水平的时候，再去深入学习也为时不晚。

在入门阶段，没有必要因为这些暂时无用的内容对学习造成负担。

一、配置

读取配置文件，我们这里使用configparser模块。

首先，我们来看一下配置文件的格式。

创建配置文件，可以通过一个文本文档来创建，或者在PyCharm的文件（File）菜单中新建（New）一个文件（File）。

配置文件的扩展名可以自定义，一般可以用“.ini”作为后缀，表示该文件是初始化文件（Initialization File）。

配置文件中类别的格式为：[类别名称]

配置项与其对应的值可以有两种格式：

- 配置项的名称：值
- 配置项的名称 = 值

配置文件内容示例：

[Size]

height : 768

width : 1024

[Color]

background = 白色

border = 黑色

上面的配置文件包含了两个类别“Size”和“Color”，每个类别中都有两个配置项，采用两种不同的格式设置了对应的值。

接下来，我们就可以通过configparser模块来读取配置文件的信息。

示例代码：

```
1 import configparser # 导入模块
2
3 file = 'config.ini' # 指定文件路径
4 config = configparser.ConfigParser() # 实例化配置解析的类
5 config.read(file,encoding='utf-8') # 读取配置文件内容
6 print('程序界面宽度: ', config.get('Size', 'width')) # 获取某一项配置的值
7 print('程序界面高度: ', config.get('Size', 'height')) # 获取某一项配置的值
8 print('程序界面背景颜色: ', config.get('Color', 'background')) # 获取某一项配置的值
9 print('程序界面边框颜色: ', config.get('Color', 'border')) # 获取某一项配置的值
```

一般来说，配置文件是给程序的使用者使用的，比如通过配置文件更改图形界面程序的皮肤，或者通过配置文件隐藏显示某些功能等等。

二、测试

有一种开发方法叫测试驱动开发（Test-Driven Development）。

测试驱动开发的基本思想就是在开发功能代码之前，先编写测试代码，然后只编写使测试通过的功能代码，从而以测试来驱动整个开发过程的进行。

关于这种开发方法的具体细节，不是一两句话就能说清楚的。

大家可以参考一下IBM在线开发者社区中的这篇文章《[浅谈测试驱动开发（TDD）](#)》。

测试一点代码，编写一点代码，可能不太好理解。

实际上，在测试代码中是要考虑到功能代码完成什么样的功能，以及种种可能出现的问题。

这样就规范了功能代码的实现范围，并且避免功能代码可预见问题的出现。

1、文档测试

文档测试需要使用doctest模块。

它可以通过代码文档中的交互式环境操作内容进行测试。

示例代码：

```
1 import doctest
2 '''
3 '>>>' 开头的行为测试用例，测试用例的下一行为测试用例的结果。
4 '''
5 def perimeter(r):
6     '''
7     计算圆形的周长。
8     :param r: 圆形半径的数值
9     :return: 圆形的周长
10    >>> perimeter(3)
11    18.85
12    '''
13    return round(2 * 3.14159 * r, 2)
14
15 if __name__ == '__main__':
16     doctest.testmod() # 运行测试
```

在上方代码中，我们能够看到perimeter()函数的文档中有如下两句：

```
>>> perimeter(3)
```

```
18.85
```

这两句内容就是我们在交互式环境中，调用perimeter()函数并传入参数的操作以及正确的结果。

doctest模块能够获取到这部分内容，执行测试。

当我们运行代码，没有发生错误时，只会有一条提示测试开始运行的信息：

```
Testing started at 12:47 ...
```

而发生错误时（例如把圆周率数值改为3.14），在PyCharm中我们能够看到测试失败的数量和一些具体的测试信息。

2、单元测试

对于大型的测试，我们可以使用unittest模块。

通过继承unittest模块定义测试类，并在类中定义不同的方法对功能代码进行测试。

假设我们有一个计算面积的area模块。

模块中包含两个函数，用于计算圆形面积和矩形面积。

示例代码：

```
1 def circular(r): # 定义计算圆形面积的函数
2     return round(3.14 * r * 2, 2) # 注意：这个计算方法是错误的
3
4 def rectangle(w, h): # 定义计算矩形面积的函数
5     return w * h
```

然后，我们再创建一个用于测试的模块。

在测试模块中，我们就可以使用unittest模块编写测试组。

```
1 import unittest, area
2
3 class TestArea(unittest.TestCase): # 定义测试类，继承unittest模块的TestCase类
4     def test_circular(self): # 定义测试圆形面积计算函数的方法
5         rst = area.circular(2) # 调用被测试的函数，传入参数，获取计算结果。
6         self.assertEqual(rst, 3.14 * 4, '圆形的面积计算错误！')
7         # 比较计算结果和预置的结果是否相等，如果不相等则测试失败，给出自定义提示。
8
9     def test_rectangle(self): # 定义测试矩形面积计算函数的方法
10        rst = area.rectangle(3, 4) # 调用被测试的函数，传入参数，获取计算结果。
11        self.assertTrue(rst == 12, '矩形的面积计算错误！')
12        # 通过表达式判断计算结果与预定的结果是否相等，如果表达式返回值为False，则测试失败，给出自定义提示。
13
14 if __name__ == '__main__':
15     unittest.main() # 运行测试
```

上方代码中，我写出了两种对被测试函数计算结果和预置结果进行比较的方法：

- `assertEqual()`方法能够比较第1个参数和第2个参数是否相等，如果不相等则会给出测试失败的结果，并显示第3个参数中自定义的提示。
- `assertTrue()`方法的第1个参数是表达式，如果表达式的结果是False，则会给出测试失败的结果，并显示第2个参数中自定义的提示。

而且，上方代码中每一个测试方法都是以“test”开头，这是必须的。

因为，运行测试的语句“`unittest.main()`”只会自动运行测试类中所有以“test”开头的方法。

不过，上方的代码运行后虽然没有给出测试失败的提示，但是，其实计算圆形面积函数的计算方法是错误的，只不过刚好我们给的参数（2）无法发现错误。

当我们把测试的参数改为3，就会提示测试失败，显示提示信息：

(...省略了部分提示...)

AssertionError: 18.84 != 28.26 : 圆形的面积计算错误!

(...省略了部分提示...)

所以，当我们编写测试程序的时候，一定要考虑全面，尽量模拟出所有可能导致程序出现问题的情形。

另外，还要注意，这里的示例只是演示了模块的基本使用方法，在实际编程中测试不会这么简单。

三、日志

日志一般是用来记录程序运行相关的数据，也能在程序出现问题时帮我们排查错误。

接下来，我们使用logging模块来完成记录程序运行日志的功能。

日志文件通过logging模块中的basicConfig()函数创建，函数的第一个参数是日志记录的级别，第二个参数是日志文件的名称。

示例代码：

```
1 import logging, time
2
3 log_time = time.strftime('%Y-%m-%d', time.localtime(time.time())) # 获取当前日期
4 logging.basicConfig(level=logging.INFO, filename=log_time + '.log') # 创建日志文件
5 logging.info('%s 程序开始...' % time.ctime()) # 写入日志记录
6 try:
7     for i in range(-5, 5):
8         logging.info('正在计算 12 /%d ...' % i) # 写入日志记录
9         print(12 / i)
10 except Exception as e:
11     logging.info('发生错误: ' + str(e)) # 写入日志记录
12     raise
13 logging.info('%s 程序结束...' % time.ctime()) # 写入日志记录
```

在上方代码中，程序的功能是除法计算，但会有零除的错误产生。

通过日志，我们就能够得到程序每一次运算过程的记录以及出错时的异常信息。

运行代码之后，会生成当天的日志文件，并记录内容。

日志内容如下：

INFO:root:Thu Dec 7 13:30:09 2017 程序开始...

INFO:root:正在计算 12 /-5 ...

INFO:root:正在计算 12 /-4 ...

INFO:root:正在计算 12 /-3 ...

INFO:root:正在计算 12 /-2 ...

INFO:root:正在计算 12 /-1 ...

INFO:root:正在计算 12 /0 ...

INFO:root:发生错误：division by zero

Python代码的打包发布

首先，打包的用途分为以下几种：

- 生成可以通过命令行安装的安装包
- 生成带有Windows中带有安装向导界面的安装包
- 生成Windows可执行程序（exe）

一、生成可以通过命令行安装的安装包

distutils模块提供了在Python环境中构建和安装额外模块的支持。新的模块可以是纯粹的Python模块，也可以是C语言编写的扩展模块，也可以是Python这两种模块的集合。

使用这个模块创建可安装的自定义模块，我们需要先创建一个setup.py的脚本。

示例代码：

```
1 from distutils.core import setup
2
3 setup(name='小楼的计算器', # 程序名称
4       version='1.0', # 程序版本号
5       description='一个萌萌的计算器。', # 程序描述
6       author='小楼一夜听春雨', # 程序作者
7       py_modules=['calculator'], # 包含的模块列表
8       packages=['other_module'], # 包含的包列表
9       )
```

当我们完成了setup脚本的设置，就可以通过命令行终端（CMD）进行自定义包的创建。

这里有一些命令，非常简单。

1、build（构建）

将现有的自定义模块以及关联的文件构建为可安装文件包，需要使用build命令。

如上图所示，当我们在命令行终端输入命令：python setup.py build

这个时候，程序就自动的创建了一个build文件夹和该文件夹下的lib子文件夹，并且按照我们写好的setup.py脚本，将模块和关联的其它文件复制到了lib文件夹中，完成了安装包的构建。

所以，实际上build命令就是根据setup.py脚本，将安装包相关的内容全部复制出来，形成了一个新的单独的包。

2、install（安装）

安装命令install就是在build包的基础之上进行安装。

实际上，在本机直接安装自定义模块（或第三方模块/包），我们无需执行build命令，在执行install命令的同时，程序会自动的先执行build命令。

如上图所示，当我们执行了命令：`python setup.py install`

程序自动的执行了build操作，然后将build包中的内容复制到了Python安装目录下的Python36\Lib\site-packages文件夹中，完成了自定义模块（第三方模块/包）的安装。

当我们打开Python安装目录下的Python36\Lib\site-packages文件夹，就能够看到build包中的所有文件内容。

3、sdist（分发）

如果想把自己开发的自定义模块发给他人并能够通过install命令进行安装，我们可以通过sdist命令进行程序发布。

如上图所示，当我们执行了命令：`python setup.py sdist`

程序就开始进行安装包的创建。

这个过程中会出现一些警告信息，例如没有在setup.py中设置作者的邮箱。

这些警告信息不会影响程序的发布。

当完成程序的发布，在项目文件夹下，会出现一个新的dist包，在这个包的里面，我们能够看到新生成的安装包文件（.tar.gz），这个安装包文件根据setup.py脚本中的文件名称和版本号为命名，例如：小楼的计算器-1.0.tar.gz。

这个生成的安装包文件是一个压缩包文件，解压缩之后，在命令行终端中进入解压缩后的目录，通过“`python setup.py install`”就能够完成这个模块的安装。

实际上，我们完全可以自己去创建这样的压缩包文件，只需要将setup.py脚本和需要发布的文件整理到一起，压缩之后发给别人，也能够进行安装。

不过，如果想做成一个带有安装向导界面的安装包，就不能自己完成了。

二、生成带有Windows中带有安装向导界面的安装包

生成带有Windows中带有安装向导界面的安装包也非常简单，只需要执行“`bdist_wininst`”命令。

当我们执行命令：`python setup.py bdist_wininst`

这个时候，程序也会自动的创建build包和dist包。

不过在dist包中会出现一个Windows可执行程序（exe）的文件，例如：小楼的计算器-1.0.win-amd64.exe

这个文件就可以发送给他人，通过双击这个文件会显示安装向导界面，多次点击“下一步”按钮就可以完成模块的安装操作。

三、生成Windows可执行程序（exe）

1、生成可执行程序

上面的“`bdist_wininst`”命令虽然能够生成Windows可执行程序，但是那只是安装模块的程序。

如果我们开发了一个图形界面程序（例如计算器），如何能够发布成能够直接双击运行的Windows可执行程序呢？

这里需要使用第三方库PyInstaller。

这个第三方库可以到官网下载：<http://www.pyinstaller.org/downloads.html>

也可以通过pip命令直接在线安装：`pip install pyinstaller`。

当然，也可以在PyCharm的默认设置（Default Settings）中，通过搜索模块名称进行安装。

注意，安装完毕之后，在Windows的环境变量添加这个库的安装路径，路径末尾以英文半角的分号“`;`”结束。

以Windows7默认安装路径为例，操作步骤为：计算机-属性-高级系统设置-环境变量-XXX的用户变量-PATH，在PATH变量原有内容的前面添加“`%APPDATA%\Python\Python36\Scripts\`”。

如果没有配置环境变量，就必须在命令行终端进入这个库的安装目录，才能够执行pyinstaller的命令。

当完成上面的配置，我们就可以尝试生成可执行程序的操作了。

使用PyInstaller不需要setup.py这个脚本文件，只需要执行pyinstaller的命令。

例如，我们想把上面示例中的calculator模块生成可执行程序。

在PyCharm中打开命令行终端，并在命令行终端执行命令：`pyinstaller calculator.py`

注意：如果是通过系统的开始菜单键入“cmd”打开的命令行窗口，则需要先通过DOS命令“`cd 项目文件夹路径`”进入到项目文件夹。

当命令执行完毕，命令提示符之前的一行能够看看“completed successfully.”字样，就说明程序生成成功了。这个时候自动出现的dist包中，会包含一个名称为“calculator”的文件夹，并且里面包含了“calculator.exe”文件。

双击运行“calculator.exe”这个可执行程序，就能够打开这个计算器程序。

但是，这里会有两个问题。

- 运行可执行程序时，会显示一个多余的黑色控制台窗口；
- 生成的文件，除了“calculator.exe”之外还有很多。

这两个问题，我们可以在执行“pyinstaller”命令时添加参数来解决。

- -w：不显示控制台窗口
- -F：只生成单个文件

删除之前生成的内容，我们再次尝试执行新的命令：`pyinstaller -w -F calculator.py`

这一次，就只生成了一个运行时没有控制台窗口的“calculator.exe”文件。

2、为程序添加图标

生成的可执行程序可以自定义程序的图标。

例如，在项目文件夹中添加一个名称为“App.ico”的图标文件。

然后，我们通过pyinstaller命令添加“-i”的参数，就能够让生成的可执行程序显示这个图标。

具体命令：`pyinstaller -w -F -i App.ico calculator.py`

3、为程序添加依赖文件

我们开发一款程序，往往不是一个模块文件，还可能包含其他相关的文件。

那么，这些相关的文件，如何一起打包？

当我们运行了“pyinstaller”命令，生成了一次可执行程序之后，在项目文件夹中会自动出现一个说明文件（.spec）。

这个说明文件，我们可以进行修改。

将说明文件中的语句“`datas=[]`,” 更改为“`datas=[('文件路径' , 生成时的文件夹名称)]`,”

例如，我们想在生成可执行程序的时候，将项目文件夹中一个名为“res”文件夹中的所有文件一起打包，并且把这些文件包含到生成的可执行程序文件夹的“resource”文件夹中。

那么，我们就可以将“datas”这一句代码写为：`datas=[('D:\\MyProject\\other\\res*.*' , resource')]`,

修改为说明文件之后，我们就可以通过“pyinstaller”命令执行这个说明文件：`pyinstaller calculator.spec`

重新生成可执行程序文件成功之后，我们就能够在可执行程序所在的文件夹中看到名为“resource”文件夹，里面包含我们想要一起打包的文件。

关于PyInstaller的深入使用，请参考官方文档：<https://pyinstaller.readthedocs.io/en/stable/index.html>