



# Python\_面向对象



# Python进阶课程

0

函数

03

1

文件操作、综合应用

37

2

面向对象1

58

3

面向对象2

96

4



# 1、函数



## 1.1 函数介绍

- 函数介绍
- <1>什么是函数
- 请看代码:

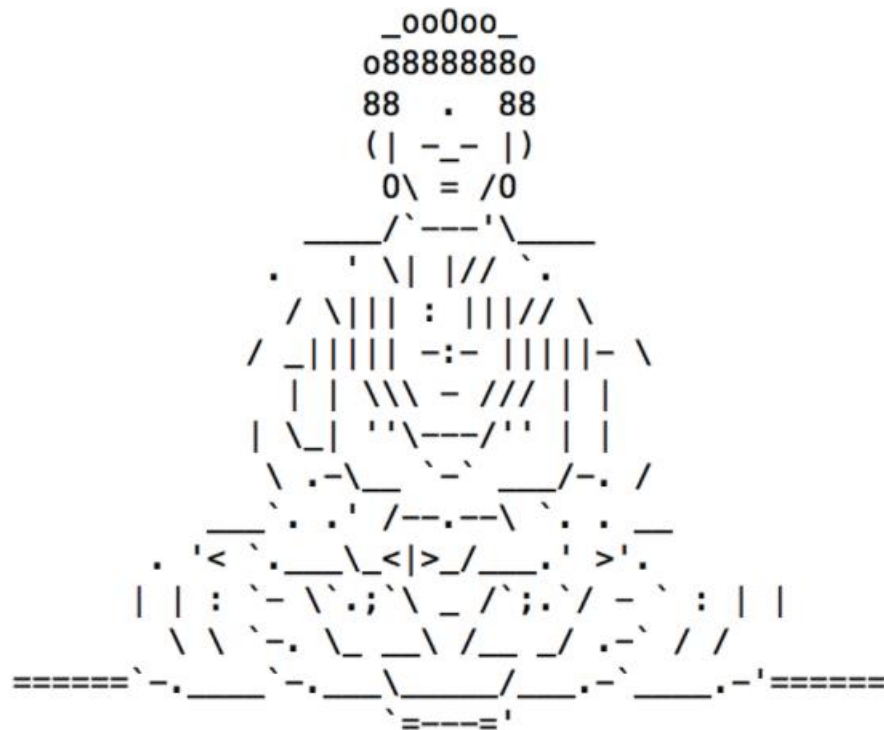
```

print "          _ooOoo_  "
print "          o8888888o  "
print "          88 . 88  "
print "          (| -_- |)  "
print "          O\\ = /O  "
print "          ___/`---'\___  "
print "          .   .  \\\|| |//` .  "
print "          / \\\||| : |||// \\\  "
print "          / _||| | -:- |||| - \\\  "
print "          | | \\\||| - /// | |  "
print "          | \\\_|| ''\\|---/'' | |  "
print "          \\\ .-\\|_ `-' ___/-. /  "
print "          ___`-. .' /-.-.-\\|`. . _  "
print "          ."" '<`.___\\|_<|>_/_.' >'"" .  "
print "          | | : `-' \\\`.;`\\|_ /`.;`/ -` : | |  "
print "          \\\ || `-' . \\\_ __\\| /__ _/ .-` / /  "
print "          =====`-.____`-.____\\|____/____.-`____.-'=====  "
print "          `=---='  "
print "  "
print "          .....  "
print "          佛祖镇楼                      BUG辟易  "
print "          佛曰：  "
print "          写字楼里写字间，写字间里程序员；  "
print "          程序人员写程序，又拿程序换酒钱。  "
print "          酒醒只在网上坐，酒醉还来网下眠；  "
print "          酒醉酒醒日复日，网上网下年复年。  "
print "          但愿老死电脑间，不愿鞠躬老板前；  "
print "          奔驰宝马贵者趣，公交自行程序员。  "
print "          别人笑我忒疯癫，我笑自己命太贱；  "
print "          不见满街漂亮妹，哪个归得程序员？"

```

# 1.1 函数介绍

- 运行后的现象:



.....

佛祖镇楼	BUG辟易
------	-------

佛曰：

写字楼里写字间，写字间里程序员；  
 程序人员写程序，又拿程序换酒钱。  
 酒醒只在网上坐，酒醉还来网下眠；  
 酒醉酒醒日复日，网上网下年复年。  
 但愿老死电脑间，不愿鞠躬老板前；  
 奔驰宝马贵者趣，公交自行程序员。  
 别人笑我忒疯癫，我笑自己命太贱；  
 不见满街漂亮妹，哪个归得程序员？

# 1.1 函数介绍

想一想：

如果一个程序在不同的地方需要输出“佛祖镇楼”，程序应该怎样设计？

```
if 条件1:  
    输出‘佛祖镇楼’  
  
...(省略)...  
  
if 条件2:  
    输出‘佛祖镇楼’  
  
...(省略)...
```

如果需要输出多次，是否意味着要编写这块代码多次呢？

- 小总结：
- 如果在开发程序时，需要某块代码多次，但是为了提高编写的效率以及代码的重用，所以把具有独立功能的代码块组织为一个小模块，这就是函数

## 1.2 函数定义、调用

- <1>定义函数
- 定义函数的格式如下：

```
def 函数名():  
    代码
```

- demo:

```
# 定义一个函数，能够完成打印信息的功能  
def printInfo():  
    print '-----'  
    print '        人生苦短，我用Python'  
    print '-----'
```

## 1.2 函数定义、调用

- <2>调用函数
- 定义了函数之后，就相当于有了一个具有某些功能的代码，想要让这些代码能够执行，需要调用它
- 调用函数很简单的，通过 `函数名()` 即可完成调用
- demo:

```
# 定义完函数后，函数是不会自动执行的，需要调用它才可以  
printInfo()
```



## 1.2 函数定义、调用

- <3>练一练
- 要求：定义一个函数，能够输出自己的姓名和年龄，并且调用这个函数让它执行
- 使用def定义函数
- 编写完函数之后，通过 函数名() 进行调用
- =====
- 姓名： xxxx
- 年龄： xxxx
- 性别： xxxx
- 手机号码： xxxx
- 微信： xxxx
- QQ： xxxx
- =====

## 1.3 函数参数(一)

思考一个问题，如下：

现在需要定义一个函数，这个函数能够完成2个数的加法运算，并且把结果打印出来，该怎样设计？下面的代码可以吗？有什么缺陷吗？

```
def add2num():  
    a = 11  
    b = 22  
    c = a+b  
    print c
```

- 为了让一个函数更通用，即想让它计算哪两个数的和，就让它计算哪两个数的和，在定义函数的时候可以让函数接收数据，就解决了这个问题，这就是 函数的参数

## 1.3 函数参数(一)

- <1> 定义带有参数的函数

示例如下：

```
def add2num(a, b):  
    c = a+b  
    print c
```

- <2> 调用带有参数的函数

以调用上面的add2num(a, b)函数为例:

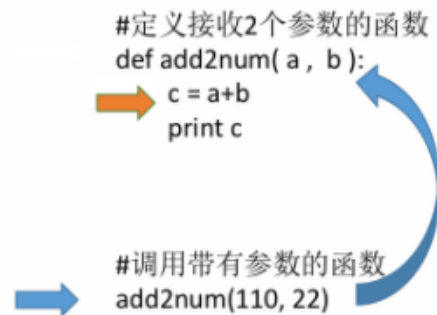
```
def add2num(a, b):  
    c = a+b  
    print c  
  
add2num(11, 22) #调用带有参数的函数时，需要在小括号中，传递数据
```

## 1.3 函数参数(一)

- 调用带有参数函数的运行过程:

```
#定义接收2个参数的函数
def add2num( a , b ):
    c = a+b
    print c

#调用带有参数的函数
add2num(110, 22)
```



此时  
a=110  
b=22  
  
c=132

终端

## 1.3 函数参数(一)

- <3> 练一练
- 要求：定义一个函数，完成前2个数完成加法运算，然后对第3个数，进行减法；然后调用这个函数
- 使用def定义函数，要注意有3个参数
- 调用的时候，这个函数定义时有几个参数，那么就需要传递几个参数
- 要求：定义一个函数，完成年份的输入，得出闰年还是平年
- 要求：定义一个函数，完成行、列的输入，打印矩形

## 1.4 函数返回值(一)

- <1>“返回值” 介绍

现实生活中的场景:

我给儿子10块钱，让他给我买包烟。这个例子中，10块钱是我给儿子的，就相当于调用函数时传递到参数，让儿子买烟这个事情最终的目标是，让他把烟给你带回来然后给你对么，，，此时烟就是返回值

开发中的场景：

定义了一个函数，完成了获取室内温度，想一想是不是应该把这个结果给调用者，只有调用者拥有了这个返回值，才能够根据当前的温度做适当的调整

综上所述：

- 所谓“返回值”，就是程序中函数完成一件事情后，最后给调用者的结果

## 1.4 函数返回值(一)

- <2>带有返回值的函数
- 想要在函数中把结果返回给调用者，需要在函数中使用`return`
- 如下示例:

```
def add2num(a, b):  
    c = a+b  
    return c
```

- 或者

```
def add2num(a, b):  
    return a+b
```

## 1.4 函数返回值(一)

- <3>保存函数的返回值
- 在本小节刚开始的时候，说过的“买烟”的例子中，最后儿子给你烟时，你一定从儿子手中接过来 对么，程序也是如此，如果一个函数返回了一个数据，那么想要用这个数据，那么就需要保存
- 保存函数的返回值示例如下：

```
#定义函数
def add2num(a, b):
    return a+b

#调用函数，顺便保存函数的返回值
result = add2num(100,98)

#因为result已经保存了add2num的返回值，所以接下来就可以使用了
print result
```



## 1.5 4种函数的类型

- 函数根据有没有参数，有没有返回值，可以相互组合，一共有4种
- 无参数，无返回值
- 无参数，又反悔
- 有参数，无返回值
- 有参数，有返回值

## 1.5 4种函数的类型

- <1>无参数，无返回值的函数
- 此类函数，不能接收参数，也没有返回值，一般情况下，打印提示灯类似的功能，使用这类的函数

```
def printMenu():  
    print('-----')  
    print('      xx涮涮锅 点菜系统')  
    print('')  
    print('  1.  羊肉涮涮锅')  
    print('  2.  牛肉涮涮锅')  
    print('  3.  猪肉涮涮锅')  
    print('-----')
```

结果:

```
-----  
      xx涮涮锅 点菜系统  
  
  1.  羊肉涮涮锅  
  2.  牛肉涮涮锅  
  3.  猪肉涮涮锅  
-----
```

## 1.5 4种函数的类型

- <2>无参数，有返回值的函数
- 此类函数，不能接收参数，但是可以返回某个数据，一般情况下，像采集数据，用此类函数

```
# 获取温度
def getTemperature():

    #这里是获取温度的一些处理过程

    #为了简单起见，先模拟返回一个数据
    return 24

temperature = getTemperature()
print('当前的温度为:%d'%temperature)
```

结果:

当前的温度为: 24

## 1.5 4种函数的类型

- **<3>有参数，无返回值的函数**
- 此类函数，能接收参数，但不可以返回数据，
- 一般情况下，对某些变量设置数据而不需结果时，
- 用此类函数
- **<4>有参数，有返回值的函数**
- 此类函数，不仅能接收参数，还可以返回某个数据，
- 一般情况下，像数据处理并需要结果的应用，
- 用此类函数
- **<5>小总结**
- 函数根据有没有参数，有没有返回值可以相互组合
- 定义函数时，是根据实际的功能需求来设计的，所以不同开发人员编写的函数类型各不相同

```
# 计算1~num的累积和
def calculateNum(num):

    result = 0
    i = 1
    while i<=num:

        result = result + i

        i+=1

    return result

result = calculateNum(100)
print('1~100的累积和为:%d'%result)
```

结果:

1~100的累积和为: 5050

## 1.5函数的嵌套调用

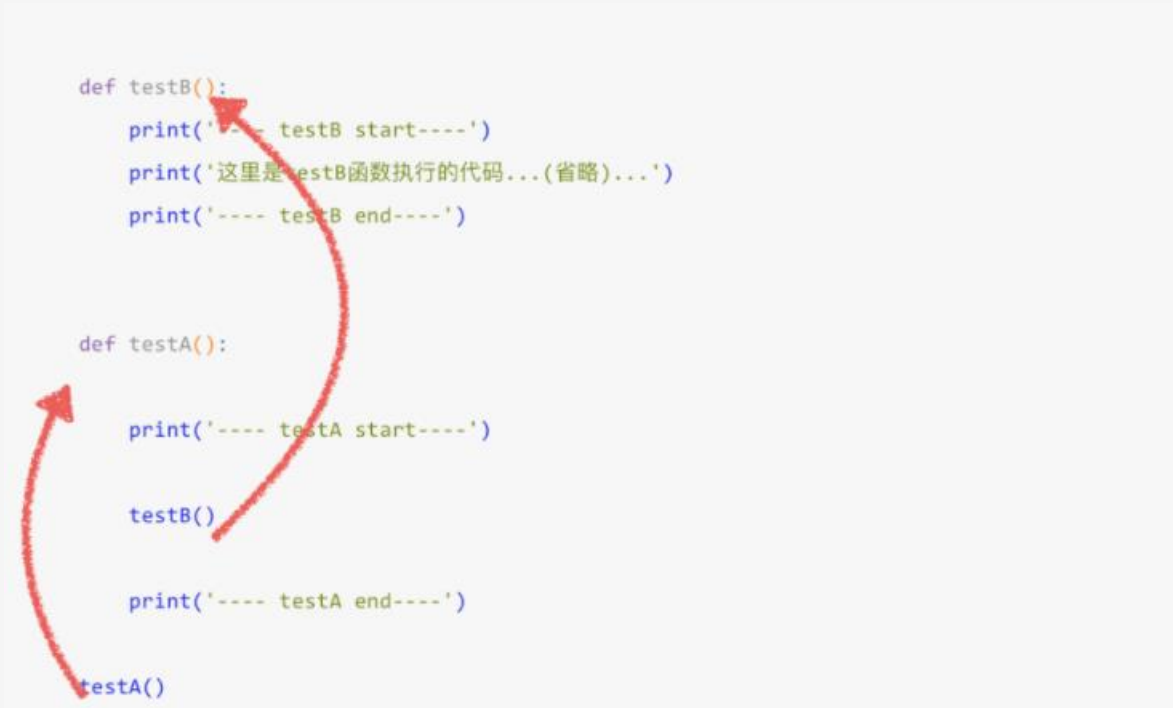
```
def testB():  
    print('---- testB start----')  
    print('这里是testB函数执行的代码...(省略)...')  
    print('---- testB end----')  
  
def testA():  
  
    print('---- testA start----')  
  
    testB()  
  
    print('---- testA end----')  
  
testA()
```

结果：

```
---- testA start----  
---- testB start----  
这里是testB函数执行的代码...(省略)...  
---- testB end----  
---- testA end----
```

## 1.5 函数的嵌套调用

- 小总结:
- 一个函数里面又调用了另外一个函数，这就是所谓的函数嵌套调用



- 如果函数A中，调用了另外一个函数B，那么先把函数B中的任务都执行完毕之后才会回到上次 函数A执行的位置
-

# 函数的嵌套应用

- 思考&实现
  - 写一个函数求三个数的和
  - 写一个函数求三个数的平均值
- 
- 思考&实现
  - 写一个函数求列表或者元组的平均值

```
# 求3个数的和
def sum3Number(a,b,c):
    return a+b+c # return 的后面可以是数值，也可是一个表达式

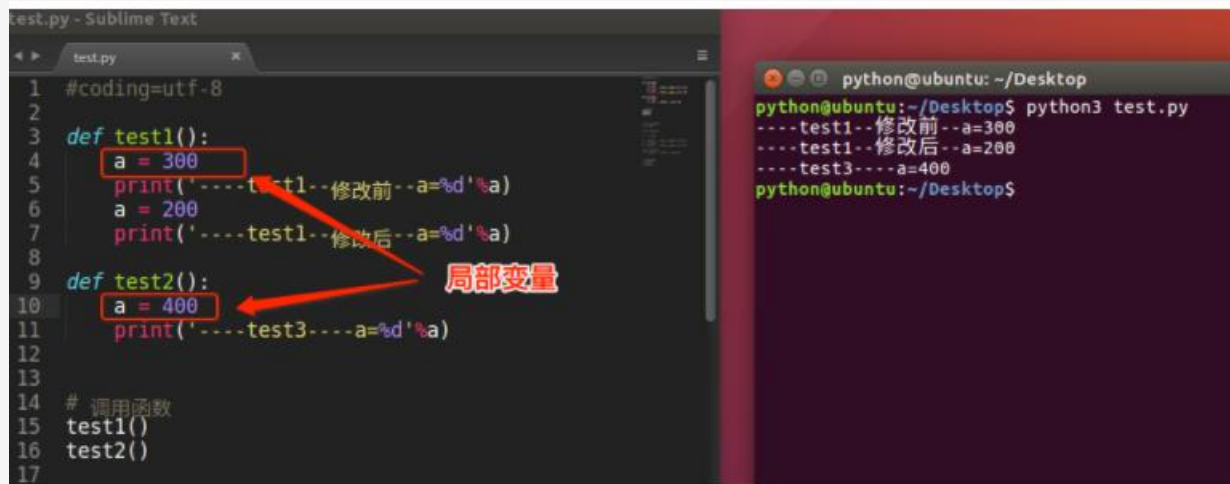
# 完成对3个数求平均值
def average3Number(a,b,c):

    # 因为sum3Number函数已经完成了3个数的求和，所以只需调用即可
    # 即把接收到的3个数，当做实参传递即可
    sumResult = sum3Number(a,b,c)
    aveResult = sumResult/3.0
    return aveResult

# 调用函数，完成对3个数求平均值
result = average3Number(11,2,55)
print("average is %d"%result)
```

## 1.6 局部变量

- <1>什么是局部变量
- 如下图所示:



```
test.py - Sublime Text
1 #coding=utf-8
2
3 def test1():
4     a = 300
5     print('----test1--修改前--a=%d'%a)
6     a = 200
7     print('----test1--修改后--a=%d'%a)
8
9 def test2():
10    a = 400
11    print('----test3----a=%d'%a)
12
13
14 # 调用函数
15 test1()
16 test2()
17
```

```
python@ubuntu: ~/Desktop
python@ubuntu:~/Desktop$ python3 test.py
----test1--修改前--a=300
----test1--修改后--a=200
----test3----a=400
python@ubuntu:~/Desktop$
```

- <2>小总结
- 局部变量，就是在函数内部定义的变量
- 不同的函数，可以定义相同的名字的局部变量，但是各用个的不会产生影响
- 局部变量的作用，为了临时保存数据需要在函数中定义变量来进行存储，这就是它的作用
-



## 1.7 全局变量

- 全局变量
- <1>什么是全局变量
- 如果一个变量，既能在一个函数中使用，也能在其他的函数中使用，这样的变量就是全局变量
- demo如下:
- 

```
# 定义全局变量
a = 100

def test1():
    print(a)

def test2():
    print(a)

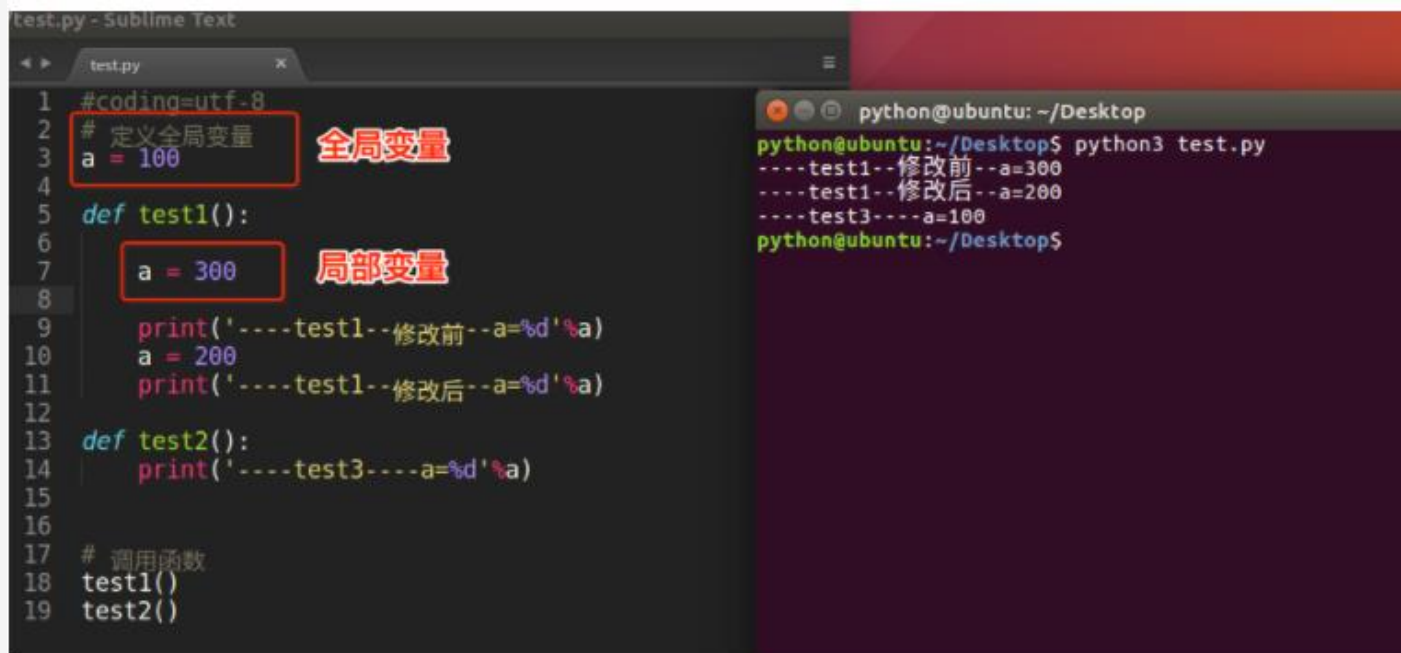
# 调用函数
test1()
test2()
```

运行结果:

```
python@ubuntu:~/Desktop$ python3 test.py
100
100
```

## 1.7 全局变量

- <2>全局变量和局部变量名字相同问题
- 看如下代码:



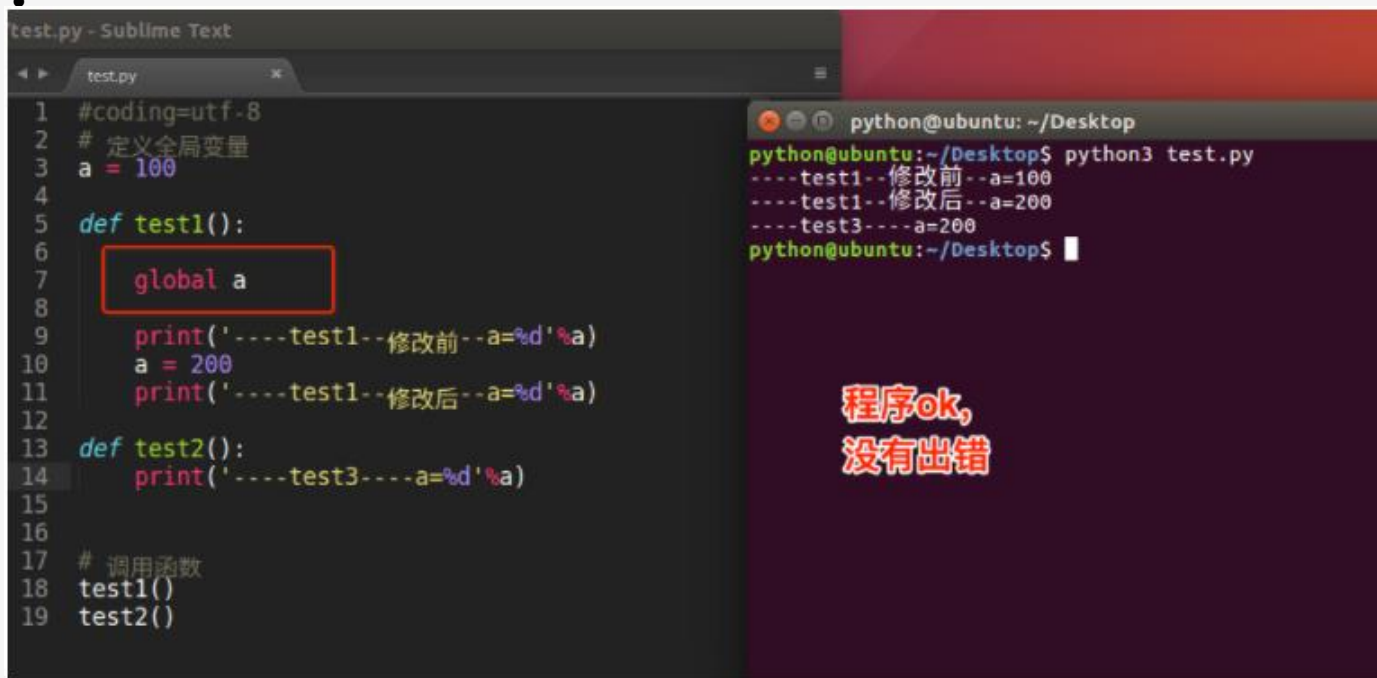
The image shows a code editor window titled 'test.py - Sublime Text' and a terminal window. The code in the editor defines a global variable 'a' and two functions, 'test1()' and 'test2()'. 'test1()' modifies 'a' and prints its value at three points. 'test2()' prints the value of 'a'. The terminal shows the output of running 'python3 test.py', which matches the print statements in the code.

```
test.py - Sublime Text
1 #coding=utf-8
2 # 定义全局变量
3 a = 100
4
5 def test1():
6     a = 300
7
8     print('----test1--修改前--a=%d'%a)
9     a = 200
10    print('----test1--修改后--a=%d'%a)
11
12 def test2():
13    print('----test3----a=%d'%a)
14
15
16
17 # 调用函数
18 test1()
19 test2()
```

```
python@ubuntu: ~/Desktop
python@ubuntu:~/Desktop$ python3 test.py
----test1--修改前--a=300
----test1--修改后--a=200
----test3----a=100
python@ubuntu:~/Desktop$
```

## 1.7 全局变量

- <3>修改全局变量
- 既然全局变量，就是能够在所有的函数中进行使用，那么可否进行修改呢？
- 代码如下：



The image shows a code editor window titled 'test.py - Sublime Text' and a terminal window. The code in the editor defines a global variable 'a' and two functions, 'test1()' and 'test2()'. 'test1()' prints the value of 'a' before and after changing it to 200. 'test2()' prints the value of 'a' without changing it. The terminal shows the output of running 'python3 test.py', which matches the expected behavior: 'test1' changes 'a' to 200, and 'test2' prints 200.

```
test.py - Sublime Text
1 #coding=utf-8
2 # 定义全局变量
3 a = 100
4
5 def test1():
6     global a
7
8     print('----test1--修改前--a=%d'%a)
9     a = 200
10    print('----test1--修改后--a=%d'%a)
11
12 def test2():
13    print('----test3----a=%d'%a)
14
15
16
17 # 调用函数
18 test1()
19 test2()
```

```
python@ubuntu: ~/Desktop
python@ubuntu:~/Desktop$ python3 test.py
----test1--修改前--a=100
----test1--修改后--a=200
----test3----a=200
python@ubuntu:~/Desktop$
```

程序ok,  
没有出错

## 1.7 全局变量

- **<4>总结1:**
  - 在函数外边定义的变量叫做全局变量
  - 全局变量能够在所有的函数中进行访问
  - 如果在函数中修改全局变量，那么就需要使用`global`进行声明，否则出错
  - 如果全局变量的名字和局部变量的名字相同，那么使用的是局部变量的，小技巧强龙不压地头蛇

## 1.7 全局变量

- <5>可变类型的全局变量
- <6>总结2:
- 在函数中不使用`global`声明全局变量时不能修改全局变量的本质是不能修改全局变量的指向，即不能将全局变量指向新的数据。
- 对于不可变类型的全局变量来说，因其指向的数据不能修改，所以不使用`global`时无法修改全局变量。
- 对于可变类型的全局变量来说，因其指向的数据可以修改，所以不使用`global`时也可修改全局变量。

```
>>> a = 1
>>> def f():
...     a += 1
...     print a
...
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
UnboundLocalError: local variable 'a' referenced before assignment
>>>
>>>
>>> li = [1,]
>>> def f2():
...     li.append(1)
...     print li
...
>>> f2()
[1, 1]
>>> li
[1, 1]
```

## 1.8 函数返回值(二)

- 函数返回值(二)
- 在python中我们可不可以返回多个值？

```
>>> def divid(a, b):  
...     shang = a//b  
...     yushu = a%b  
...     return shang, yushu  
...  
>>> sh, yu = divid(5, 2)  
>>> sh  
5  
>>> yu  
1
```

本质是利用了元组

## 1.8 函数参数(二)

- **1. 缺省参数**
- 调用函数时，缺省参数的值如果没有传入，则被认为是默认值。下例会打印默认的age，如果age没有被传入：

```
def printinfo( name, age = 35 ):
    # 打印任何传入的字符串
    print "Name: ", name
    print "Age ", age

# 调用printinfo函数
printinfo(name="miki" )
printinfo( age=9,name="miki" )
```

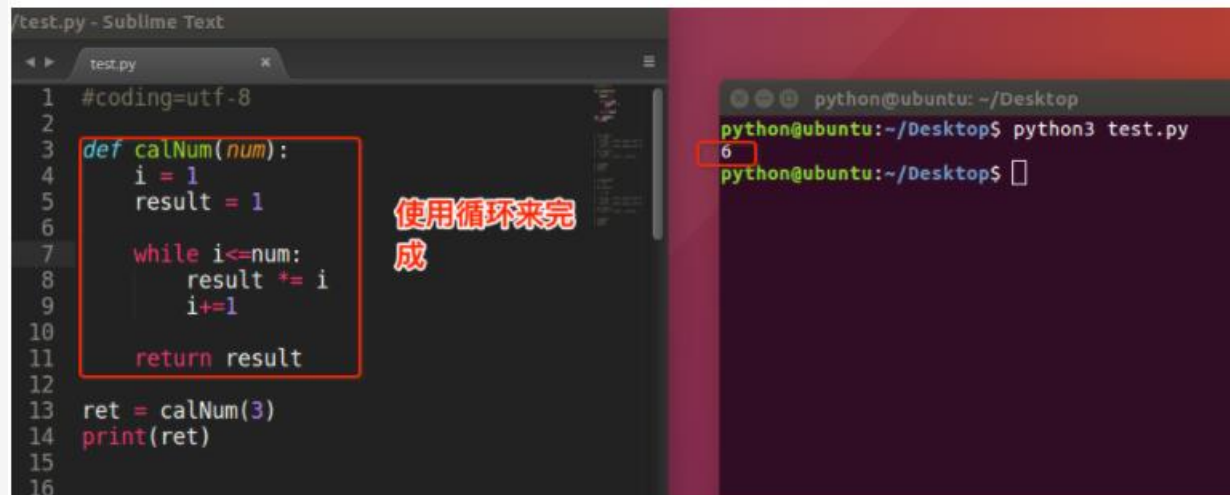
- 以上实例输出结果：

```
Name: miki
Age  35
Name: miki
Age  9
```

- 注意：带有默认值的参数一定要位于参数列表的最后面。

## 1.9 递归函数

- <1>什么是递归函数
- 通过前面的学习知道一个函数可以调用其他函数。
- 如果一个函数在内部不调用其它的函数，而是自己本身的话，这个函数就是递归函数。
- <2>递归函数的作用
- 举个例子，我们来计算阶乘  $n! = 1 * 2 * 3 * \dots * n$
- 解决办法1:



The image shows a code editor window titled 'test.py - Sublime Text' and a terminal window. The code in the editor is a Python function 'calNum(num)' that calculates the factorial of a number using a while loop. The terminal shows the command 'python3 test.py' being executed, and the output '6' is displayed, which is the factorial of 3.

```
1 #coding=utf-8
2
3 def calNum(num):
4     i = 1
5     result = 1
6
7     while i<=num:
8         result *= i
9         i+=1
10
11     return result
12
13 ret = calNum(3)
14 print(ret)
15
16
```

使用循环来完成

```
python@ubuntu: ~/Desktop
python@ubuntu:~/Desktop$ python3 test.py
6
python@ubuntu:~/Desktop$
```

看阶乘的规律

```
1! = 1
2! = 2 × 1 = 2 × 1!
3! = 3 × 2 × 1 = 3 × 2!
4! = 4 × 3 × 2 × 1 = 4 × 3!
...
n! = n × (n-1)!
```



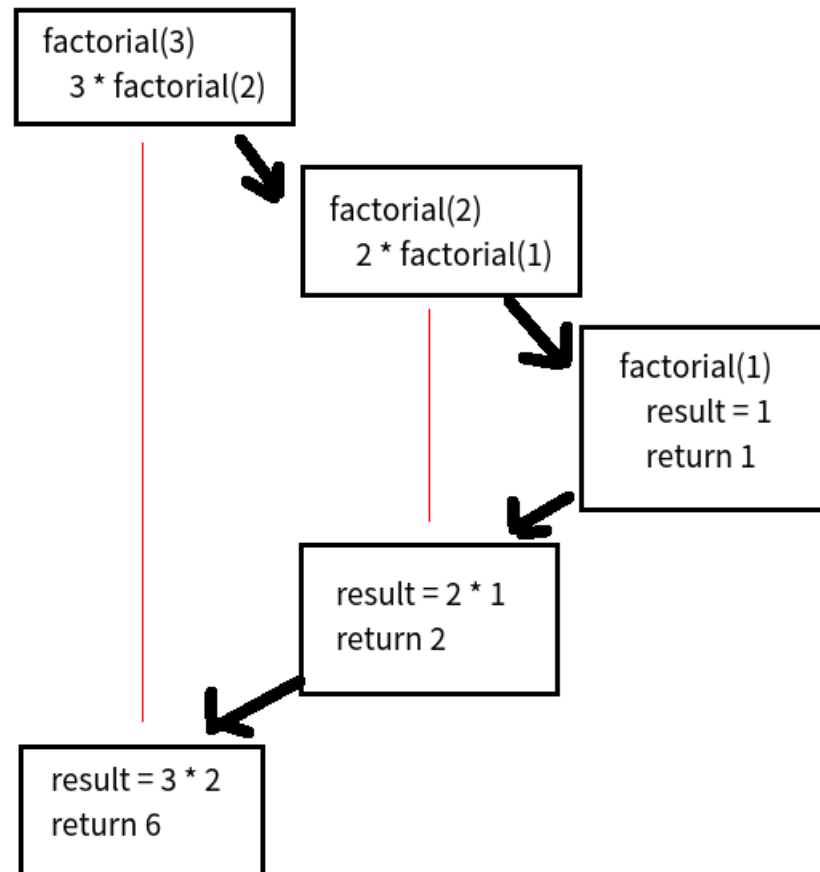
## 1.9 递归函数

```
test.py - Sublime Text
test.py
1 #coding=utf-8
2
3 def calNum(num):
4     if num>=1:
5         result = num * calNum(num-1)
6     else:
7         result = 1
8     return result
9
10 # def calNum(num):
11 #     i = 1
12 #     result = 1
13 #     while i<=num:
14 #         result *= i
15 #         i+=1
16 #     return result
17
18 ret = calNum(3)
19 print(ret)
```

使用递归来完成

```
python@ubuntu: ~/Desktop
python@ubuntu:~/Desktop$ python3 test.py
6
python@ubuntu:~/Desktop$
```

factorial(3)调用过程:



## 1.9 匿名函数

- 用lambda关键词能创建小型匿名函数。这种函数得名于省略了用def声明函数的标准步骤。
- lambda函数的语法只包含一个语句，如下：

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

- 如下实例：

```
sum = lambda arg1, arg2: arg1 + arg2

#调用sum函数
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

- 以上实例输出结果：

```
Value of total : 30
Value of total : 40
```

- Lambda函数能接收任何数量的参数但只能返回一个表达式的值
- 匿名函数不能直接调用print，因为lambda需要一个表达式

## 1.9 匿名函数

- 应用场合
- 函数作为参数传递
- 1、自己定义函数

```
>>> def fun(a, b, opt):  
...     print "a =", a  
...     print "b =", b  
...     print "result =", opt(a, b)  
...  
>>> fun(1, 2, lambda x,y:x+y)  
  
a = 1  
b = 2  
result = 3
```

## 1.9 匿名函数

- 2、作为内置函数的参数
- 想一想，下面的数据如何指定按age或name排序？

```
stus = [  
    {"name": "zhangsan", "age": 18},  
    {"name": "lisi", "age": 19},  
    {"name": "wangwu", "age": 17}  
]
```

- 按name排序：

```
>>> stus.sort(key = lambda x: x['name'])  
>>> stus  
[{'age': 19, 'name': 'lisi'}, {'age': 17, 'name': 'wangwu'}, {'age': 18, 'name': 'zhangsan'}]
```

- 按age排序：

```
>>> stus.sort(key = lambda x: x['age'])  
>>> stus  
[{'age': 17, 'name': 'wangwu'}, {'age': 18, 'name': 'zhangsan'}, {'age': 19, 'name': 'lisi'}]
```



## 2、文件操作、综合应用



## 2.1 文件操作介绍

- <1>什么是文件
- 示例如下：



## 2.1 文件操作介绍

- <2>文件的作用
- 大家应该听说过一句话：“好记性不如烂笔头”。
- 不仅人的大脑会遗忘事情，计算机也会如此，比如一个程序在运行过程中用了九牛二虎之力终于计算出了结果，试想一下如果不把这些数据存放起来，相比重启电脑之后，“哭都没地方哭了”
- 可见，在把数据存储起来有做么大的价值
- 使用文件的目的

就是把一些存储存放起来，可以让程序下一次执行的时候直接使用，而不必重新制作一份，省时省力

## 2.2 文件的打开与关闭

想一想：

如果想用word编写一份简历，应该有哪些流程呢？

1. 打开word软件，新建一个word文件
2. 写入个人简历信息
3. 保存文件
4. 关闭word软件

同样，在操作文件的整体过程与使用word编写一份简历的过程是很相似的

1. 打开文件，或者新建立一个文件
2. 读/写数据
3. 关闭文件

- <1>打开文件
- 在python，使用open函数，可以打开一个已经存在的文件，或者创建一个新文件
- open(文件名，访问模式)
- 示例如下：

```
f = open('test.txt', 'w')
```



## 2.2 文件的打开与关闭

访问模式	说明
r	以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。
w	打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
a	打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
rb	以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。
wb	以二进制格式打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
ab	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
r+	打开一个文件用于读写。文件指针将会放在文件的开头。
w+	打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
a+	打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会追加模式。如果该文件不存在，创建新文件用于读写。
rb+	以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。
wb+	以二进制格式打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
ab+	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文件不存在，创建新文件用于读写。

## 2.2 文件的打开与关闭

- <2>关闭文件
- `close()`
- 示例如下:

```
# 新建一个文件，文件名为:test.txt
f = open('test.txt', 'w')

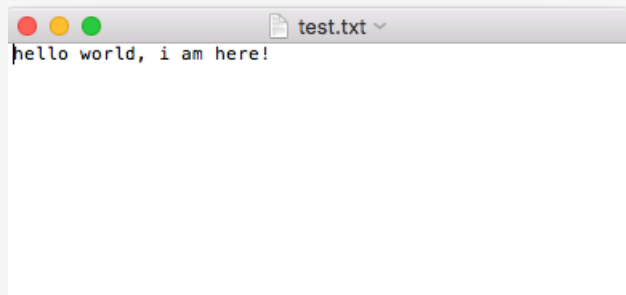
# 关闭这个文件
f.close()
```

## 2.3 文件的读写

- <1>写数据(write)
- 使用write()可以完成向文件写入数据
- demo:

```
f = open('test.txt', 'w')  
f.write('hello world, i am here!')  
f.close()
```

- 运行现象:



- 注意:
- 如果文件不存在那么创建, 如果存在那么就先清空, 然后写入数据

## 2.3 文件的读写

- <2>读数据(read)
- 使用read(num)可以从文件中读取数据，num表示要从文件中读取的数据的长度（单位是字节），如果没有传入num，那么就表示读取文件中所有的数据
- demo:

```
f = open('test.txt', 'r')
```

```
content = f.read(5)
```

```
print(content)
```

```
print("-"*30)
```

```
content = f.read()
```

```
print(content)
```

```
f.close()
```

运行现象：

```
dongGe@dongGe-Mac Desktop$ python filetest.py  
hello  
-----  
world, i am here!
```

- 注意：
- 如果open是打开一个文件，那么可以不用写打开的模式，即只写 open('test.txt')
- 如果使用读了多次，那么后面读取的数据是从上次读完后的位置开始的

## 2.3 文件的读写

- <3>读数据（**readlines**）
  - 就像read没有参数时一样，**readlines**可以按照行的方式把整个文件中的内容进行一次性读取，并且返回的是一个列表，其中每一行的数据为一个元素
- <4>读数据（**readline**）

# 应用：制作文件的备份

- 任务描述
- 输入文件的名字，然后程序自动完成对文件进行备份
- 要求：
- 原件： xxxx.xxxx
- 附件： xxxx[附件].xxxx

## 2.4 文件的定位读写

- <1>获取当前读写的位置
- 在读写文件的过程中，如果想知道当前的位置，可以使用`tell()`来获取

```
# 打开一个已经存在的文件
f = open("test.txt", "r")
str = f.read(3)
print "读取的数据是 :", str

# 查找当前位置
position = f.tell()
print "当前文件位置 :", position

str = f.read(3)
print "读取的数据是 :", str

# 查找当前位置
position = f.tell()
print "当前文件位置 :", position

f.close()
```

## 2.4 文件的定位读写

- <2>定位到某个位置
- 如果在读写文件的过程中，需要从另外一个位置进行操作的话，可以使用`seek()`
- `seek(offset, from)`有2个参数
- `offset`:偏移量
- `from`:方向
  - 0:表示文件开头
  - 1:表示当前位置
  - 2:表示文件末尾
- 1和2: `open`函数只能以二进制模式打开文件，才能正常运行，否则就会报出上面的错误
- 如果没有以二进制**b**的方式打开，则`offset`无法使用负值（即向左侧移动）



## 2.4 文件的定位读写

- demo:把位置设置为：从文件开头，偏移5个字节

```
# 打开一个已经存在的文件
f = open("test.txt", "r")
str = f.read(30)
print "读取的数据是 :", str

# 查找当前位置
position = f.tell()
print "当前文件位置 :", position

# 重新设置位置
f.seek(5,0)

# 查找当前位置
position = f.tell()
print "当前文件位置 :", position

f.close()
```

## 2.4 文件的定位读写

- demo:把位置设置为：离文件末尾，3字节处
- 

```
# 打开一个已经存在的文件
f = open("test.txt", "r")

# 查找当前位置
position = f.tell()
print "当前文件位置 :", position

# 重新设置位置
f.seek(-3,2)

# 读取到的数据为：文件最后3个字节数据
str = f.read()
print "读取的数据是 :", str

f.close()
```

## 2.5 文件的重命名、删除-os

- 有些时候，需要对文件进行重命名、删除等一些操作，python的os模块中都有这么功能
- <1>文件重命名
- os模块中的rename()可以完成对文件的重命名操作
- rename(需要修改的文件名, 新的文件名)

```
import os

os.rename("毕业论文.txt", "毕业论文-最终版.txt")
```

## 2.5 文件的重命名、删除-os

- <2>删除文件
- os模块中的remove()可以完成对文件的删除操作
- remove(待删除的文件名)

```
import os
```

```
os.remove("毕业论文.txt")
```

## 2.6 文件夹的相关操作-os

- 实际开发中，有时需要用程序的方式对文件夹进行一定的操作，比如创建、删除等
- 就像对文件操作需要os模块一样，如果要操作文件夹，同样需要os模块
- <1>创建文件夹

```
import os  
  
os.mkdir("张三")
```

- <2>获取当前目录

```
import os  
  
os.getcwd()
```

## 2.6 文件夹的相关操作-os

- 实际开发中，有时需要用程序的方式对文件夹进行一定的操作，比如创建、删除等
- 就像对文件操作需要os模块一样，如果要操作文件夹，同样需要os模块

- **<3>改变默认目录**

```
import os  
  
os.chdir("../")
```

- **<4>获取目录列表**

```
import os  
  
os.listdir("../")
```

- **<**

## 2.6 文件夹的相关操作-os

- 实际开发中，有时需要用程序的方式对文件夹进行一定的操作，比如创建、删除等
- 就像对文件操作需要os模块一样，如果要操作文件夹，同样需要os模块
- <5>删除文件夹

```
import os  
  
os.rmdir("张三")
```

# 应用：批量修改文件名

- <1>运行过程演示

- 运行程序之前

```
[dongGe@localhost renameDir$ ls -l
total 0
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-1.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-2.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-3.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-4.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-5.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-6.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-7.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-8.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-9.txt
```

- 运行程序之后

```
[dongGe@localhost renameDir$ ls -l
total 0
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-1.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-2.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-3.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-4.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-5.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-6.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-7.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-8.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-9.txt
```



# 应用：批量修改文件名

- <1>运行过程演示

- 运行程序之前

```
[dongGe@localhost renameDir$ ls -l
total 0
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-1.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-2.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-3.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-4.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-5.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-6.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-7.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-8.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-9.txt
```

- 运行程序之后

```
[dongGe@localhost renameDir$ ls -l
total 0
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-1.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-2.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-3.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-4.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-5.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-6.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-7.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-8.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-9.txt
```



### 3、面向对象1



## 3.1 面向对象编程介绍

- 面向对象编程介绍
- 想一想
- 请用程序描述如下事情:
- A同学报道登记信息
- B同学报道登记信息
- C同学报道登记信息
- A同学做自我介绍
- B同学做自我介绍
- C同学做自我介绍

```
stu_a = {  
    "name": "A",  
    "age": 21,  
    "gender": 1,  
    "hometown": "河北"  
}  
  
stu_b = {  
    "name": "B",  
    "age": 22,  
    "gender": 0,  
    "hometown": "山东"  
}  
  
stu_c = {  
    "name": "C",  
    "age": 20,  
    "gender": 1,  
    "hometown": "安徽"  
}  
  
def stu_intro(stu):  
    """自我介绍"""  
    for key, value in stu.items():  
        print("key=%s, value=%d"%(key, value))  
  
stu_intro(stu_a)  
stu_intro(stu_b)  
stu_intro(stu_c)
```

## 3.1 面向对象编程介绍

- 考虑现实生活中，我们的思维方式是放在学生这个个人上，是学生做了自我介绍。而不是像我们刚刚写出的代码，先有了介绍的行为，再去看介绍了谁。
- 用我们的现实思维方式该怎么用程序表达呢？

```
stu_a = Student(个人信息)
stu_b = Student(个人信息)
stu_c = Student(个人信息)
stu_a.intro()
stu_a.intro()
stu_a.intro()
```

- 面向过程：根据业务逻辑从上到下写代码
- 面向对象：将数据与函数绑定到一起，进行封装，这样能够更快速的开发程序，减少了重复代码的重写过程

## 3.1 面向对象编程介绍

- 面向过程编程最易被初学者接受，其往往用一长段代码来实现指定功能，开发过程的思路是将数据与函数按照执行的逻辑顺序组织在一起，数据与函数分开考虑。

```
def 发送邮件(内容)
    #发送邮件提醒
    连接邮箱服务器
    发送邮件
    关闭连接

while True:

    if cpu利用率 > 90%:
        发送邮件('CPU报警')

    if 硬盘使用空间 > 90%:
        发送邮件('硬盘报警')

    if 内存占用 > 80%:
        发送邮件('内存报警')
```

## 3.1 面向对象编程介绍

- 今天我们来学习一种新的编程方式：面向对象编程（Object Oriented Programming, OOP, 面向对象程序设计）

- 1) 解决菜鸟买电脑的故事

第一种方式:

- 1)在网上查找资料
  - 2)根据自己预算和需求定电脑的型号 MacBook 15 顶配 1W8
  - 3)去市场找到苹果店各种店无法甄别真假 随便找了一家
  - 4)找到业务员,业务员推荐了另外一款 配置更高价格便宜,也是苹果系统的 1W
  - 5)砍价30分钟 付款9999
  - 6)成交
- 回去之后发现各种问题

第二种方式:

- 1)找一个靠谱的电脑高手
- 2)给钱交易

## 3.1 面向对象编程介绍

- 今天我们来学习一种新的编程方式：面向对象编程（Object Oriented Programming, OOP，面向对象程序设计）
  - 面向对象和面向过程都是解决问题的一种思路而已
    - 买电脑的第一种方式:
      - 强调的是步骤、过程、每一步都是自己亲自去实现的
      - 这种解决问题的思路我们就叫做面向过程
    - 买电脑的第二种方式:
      - 强调的是电脑高手, 电脑高手是处理这件事的主角,对我们而言,我们并不必亲自实现整个步骤只需要调用电脑高手就可以解决问题
      - 这种解决问题的思路就是面向对象
    - 用面向对象的思维解决问题的重点
      - 当遇到一个需求的时候不用自己去实现，如果自己一步步实现那就是面向过程
      - 应该找一个专门做这个事的人来做
      - 面向对象是基于面向过程的

## 3.2类和对象

- 面向对象编程的2个非常重要的概念：类和对象
- 对象是面向对象编程的核心，在使用对象的过程中，为了将具有共同特征和行为的一组对象抽象定义，提出了另外一个新的概念——类
- 类就相当于制造飞机时的图纸，用它来进行创建的飞机就相当于对象



## 3.2类和对象

- 1. 类

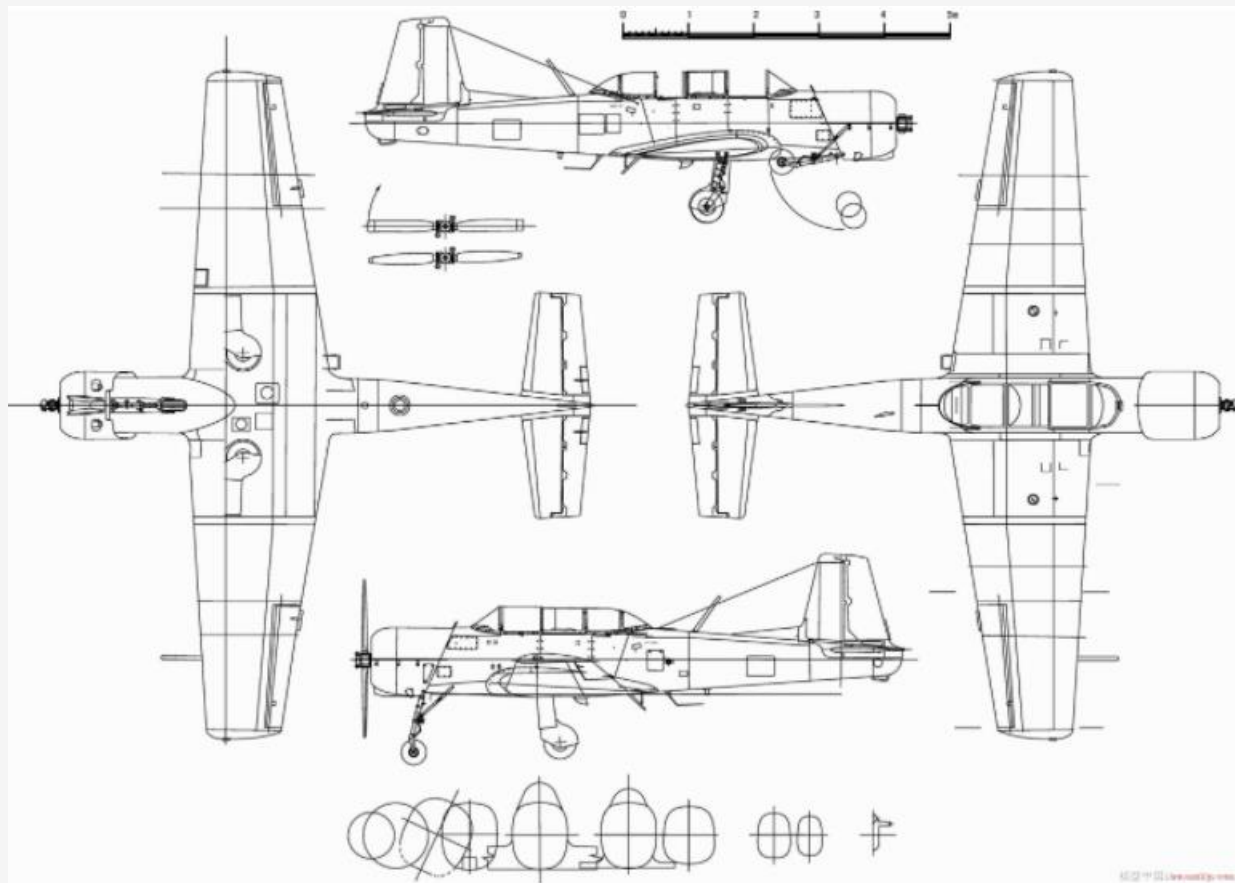
人以类聚 物以群分。

具有相似内部状态和运动规律的实体的集合(或统称为抽象)。

具有相同属性和行为事物的统称

## 3.2类和对象

- 类是抽象的,在使用的时候通常会找到这个类的一个具体的存在,使用这个具体的存在。一个类可以找到多个对象



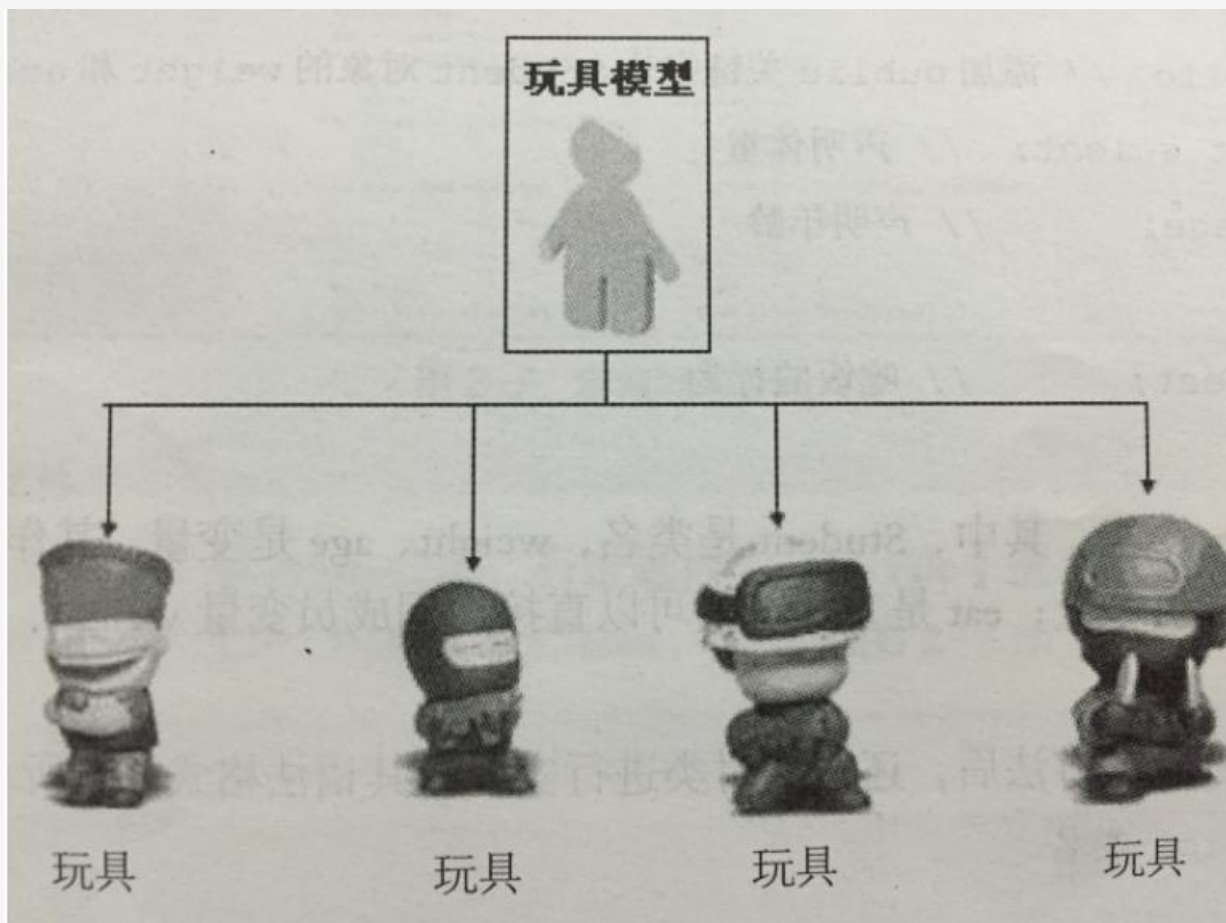
## 3.2类和对象

- 2. 对象:某一个具体事物的存在,在现实世界中可以是看得见摸得着的。



## 3.2类和对象

- 3. 类和对象之间的关系：类就是创建对象的模板



## 3.2类和对象

- 4. 练习：区分类和对象

奔驰汽车 类

奔驰smart 类

张三的那辆奔驰smart 对象

狗 类

大黄狗 类

李四家那只大黄狗 对象

水果 类

苹果 类

红苹果 类 红富士苹果 类

我嘴里吃了一半的苹果 对象

## 3.2类和对象

- 5. 类的构成
  - 类(Class) 由3个部分构成
    - 类的名称:类名
    - 类的属性:一组数据
    - 类的方法:允许对进行操作的方法 (行为)
  - <1> 举例:
    - 1) 人类设计,只关心3样东西:
      - 事物名称(类名):人(Person)
      - 属性:身高(height)、年龄(age)
      - 方法(行为/功能):跑(run)、打架(fight)

## 3.2类和对象

- 2) 狗类的设计
- 类名:狗(Dog)
- 属性:品种、毛色、性别、名字、腿儿的数量
- 方法(行为/功能):叫、跑、咬人、吃、摇尾巴

狗类	类名
名字 性别 毛色	属性, 特征
奔跑 叫	方法, 行为

## 3.2类和对象

- 6. 类的抽象
- 如何把日常生活中的事物抽象成程序中的类?
- 拥有相同(或者类似)属性和行为的对象都可以抽象出一个类
- 方法:一般名词都是类(名词提炼法)

### <1> 坦克发射3颗炮弹轰掉了2架飞机

- 坦克--》可以抽象成 类
- 炮弹--》可以抽象成类
- 飞机--》可以抽象成类

### <2> 小明在公车上牵着一只热狗

- 小明--》 人类
- 公车--》 交通工具类
- 热狗--》 食物类
- 狗--》 狗类



## 3.2类和对象



说明:

- 向日葵
  - 类名: xrk
  - 属性:
  - 行为: 放阳光
- 豌豆
  - 类名: wd
  - 属性: 颜色、发型、血量
  - 行为: 发炮, 摇头
- 坚果:
  - 类名: jg
  - 属性: 血量、类型
  - 行为: 阻挡;
- 僵尸:
  - 类名: js
  - 属性: 颜色、血量、类型、速度
  - 行为: 走、跑、跳、吃、死

## 3.2类和对象

- 定义类
- 定义一个类，格式如下：

```
class 类名:  
    方法列表
```

- demo: 定义一个Car类

```
# 定义类  
class Car:  
    # 方法  
    def getCarInfo(self):  
        print('车轮子个数:%d, 颜色%s'%(self.wheelNum, self.color))  
  
    def move(self):  
        print("车正在移动...")
```

- 说明:
- 定义类时有2种：新式类和经典类，上面的Car为经典类，如果是Car(object)则为新式类
- 类名 的命名规则按照"大驼峰"

## 3.2类和对象

- 创建对象
- 通过上一节课程，定义了一个Car类；就好比有车一个张图纸，那么接下来就应该把图纸交给生成工人们去生成了
- python中，可以根据已经定义的类去创建出一个个对象
- 创建对象的格式为:

```
对象名 = 类名()
```

## 3.2类和对象

- 创建对象demo:

```
# 定义类
class Car:
    # 移动
    def move(self):
        print('车在奔跑...')

    # 鸣笛
    def toot(self):
        print("车在鸣笛...嘟嘟..")

# 创建一个对象，并用变量BMW来保存它的引用
BMW = Car()
BMW.color = '黑色'
BMW.wheelNum = 4 #轮子数量
BMW.move()
BMW.toot()
print(BMW.color)
print(BMW.wheelNum)
```

## 3.2类和对象

```
1 class Car:
2
3     # 移动
4     def move(self):
5         print('车在奔跑...')
6
7     # 鸣笛
8     def toot(self):
9         print("车在鸣笛...嘟嘟..")
10
11
12 # 创建一个对象，并用变量BMW来保存它的引用
13 BMW = Car()
14 BMW.color = '黑色'
15 BMW.oil = 30 #油量为30升
16 BMW.move()
17 BMW.toot()
18 print(BMW.color)
19 print(BMW.oil)
20
```

告诉python  
我们在这里定  
义了一个类

定义的方法

给对象添加 属性

调用对象的方法

## 3.2类和对象

- 总结:
- `BMW = Car()`, 这样就产生了一个Car的实例对象, 此时也可以通过实例对象BMW来访问属性或者方法
- 第一次使用`BMW.color = '黑色'`表示给BMW这个对象添加属性, 如果后面再次出现`BMW.color = xxx`表示对属性进行修改
- BMW是一个对象, 它拥有属性 (数据) 和方法 (函数)
- 当创建一个对象时, 就是用一个模子, 来制造一个实物



## 3.3 \_\_init\_\_()

- \_\_init\_\_()方法

想一想:

在上一小节的demo中，我们已经给BMW这个对象添加了2个属性，wheelNum（车的轮胎数量）以及color（车的颜色），试想如果再次创建一个对象的话，肯定也需要进行添加属性，显然这样做很费事，那么有没有办法能够在创建对象的时候，就顺便把车这个对象的属性给设置呢？

答:

`__init__()` 方法

- <1>使用方式

```
def 类名:
    #初始化函数，用来完成一些默认的设置
    def __init__():
        pass
```

## 3.3 \_\_init\_\_()

- <2>\_\_init\_\_()方法的调用

```
# 定义汽车类
class Car:

    def __init__(self):
        self.wheelNum = 4
        self.color = '蓝色'

    def move(self):
        print('车在跑，目标:夏威夷')

# 创建对象
BMW = Car()

print('车的颜色为:%s'%BMW.color)
print('车轮胎数量为:%d'%BMW.wheelNum)
```

### 总结1

当创建Car对象后，在没有调用 \_\_init\_\_() 方法的前提下，BMW就默认拥有了2个属性wheelNum和color，原因是 \_\_init\_\_() 方法是在创建对象后，就立刻被默认调用了



## 3.3 \_\_init\_\_()

- 想一想

既然在创建完对象后 `__init__()` 方法已经被默认的执行了，那么能否让对象在调用 `__init__()` 方法的时候传递一些参数呢？如果可以，那怎样传递呢？

```
# 定义汽车类
class Car:

    def __init__(self, newWheelNum, newColor):
        self.wheelNum = newWheelNum
        self.color = newColor

    def move(self):
        print('车在跑，目标:夏威夷')

# 创建对象
BMW = Car(4, 'green')

print('车的颜色为:%s'%BMW.color)
print('车轮子数量为:%d'%BMW.wheelNum)
```

## 3.3 `__init__()`

- 总结2
  - 1、`__init__()`方法，在创建一个对象时默认被调用，不需要手动调用
  - 2、`__init__(self)`中，默认有1个参数名字为`self`，如果在创建对象时传递了2个实参，那么`__init__(self)`中出了`self`作为第一个形参外还需要2个形参，例如`__init__(self, x, y)`
  - 3、`__init__(self)`中的`self`参数，不需要开发者传递，python解释器会自动把当前的对象引用传递进去

## 3.4 “魔法”方法 `__str__()`

- 1. 打印`id()`

如果把BMW使用`print`进行输出的话，会看到如下的信息

```
[python@ubuntu:~/Desktop/test$ python3 oop.py
车在奔跑...
车在鸣笛...嘟嘟..
黑色
30
-----分割线-----
<__main__.Car object at 0x7ff4da7a7940>
python@ubuntu:~/Desktop/test$
```

即看到的是创建出来的BMW对象在内存中的地址

## 3.4 “魔法”方法 `__str__()`

- 2. 定义`__str__()`方法

```
class Car:

    def __init__(self, newWheelNum, newColor):
        self.wheelNum = newWheelNum
        self.color = newColor

    def __str__(self):
        msg = "嘿。。。我的颜色是" + self.color + "我有" + int(self.wheelNum) + "个轮胎..."
        return msg

    def move(self):
        print('车在跑，目标:夏威夷')

BMW = Car(4, "白色")
print(BMW)
```

```
python@ubuntu:~/Desktop/test$ python3 oop.py
嘿。。。我的颜色是白色我有4个轮胎...
python@ubuntu:~/Desktop/test$
```

## 3.4 “魔法”方法 `__str__()`

- 总结
  - 1、在python中方法名如果是`__xxxx__()`的，那么就有特殊的功能，因此叫做“魔法”方法
  - 2、当使用`print`输出对象的时候，只要自己定义了`__str__(self)`方法，那么就会打印从在这个方法中`return`的数据

## 3.5 self

- 1. 理解self
- 看如下示例:

```
# 定义一个类
class Animal:

    # 方法
    def __init__(self, name):
        self.name = name

    def printName(self):
        print('名字为:%s'%self.name)

# 定义一个函数
def myPrint(animal):
    animal.printName()

dog1 = Animal('西西')
myPrint(dog1)

dog2 = Animal('北北')
myPrint(dog2)
```

运行结果:

```
python@ubuntu:~/Desktop/test$ python3 oop2.py
名字为:西西
名字为:北北
python@ubuntu:~/Desktop/test$
```

## 3.5 self

- 总结
  - 1、所谓的self，可以理解为自己
  - 2、可以把self当做C++中类里面的this指针一样理解，就是对象自身的意思
  - 3、某个对象调用其方法时，python解释器会把这个对象作为第一个参数传递给self，所以开发者只需要传递后面的参数即可

# 应用:烤地瓜

- 应用:烤地瓜
- 为了更好的理解面向对象编程，下面以“烤地瓜”为案例，进行分析
- 1. 分析“烤地瓜”的属性和方法
- 示例属性如下:
  - `cookedLevel` : 这是数字；0~3表示还是生的，超过3表示半生不熟，超过5表示已经烤好了，超过8表示已经烤成木炭了！我们的地瓜开始时生的
  - `cookedString` : 这是字符串；描述地瓜的生熟程度
  - `condiments` : 这是地瓜的配料列表，比如番茄酱、芥末酱等
- 示例方法如下:
  - `cook()` : 把地瓜烤一段时间
  - `addCondiments()` : 给地瓜添加配料
  - `__init__()` : 设置默认的属性
  - `__str__()` : 让print的结果看起来更好一些



# 应用:烤地瓜

- 2. 定义类, 并且定义\_\_init\_\_()方法

```
#定义`地瓜`类
class SweetPotato:
    '这是烤地瓜的类'

    #定义初始化方法
    def __init__(self):
        self.cookedLevel = 0
        self.cookedString = "生的"
        self.condiments = []
```

# 应用:烤地瓜

- 3. 添加"烤地瓜"方法

```
#烤地瓜方法
def cook(self, time):
    self.cookedLevel += time
    if self.cookedLevel > 8:
        self.cookedString = "烤成灰了"
    elif self.cookedLevel > 5:
        self.cookedString = "烤好了"
    elif self.cookedLevel > 3:
        self.cookedString = "半生不熟"
    else:
        self.cookedString = "生的"
```

# 应用:烤地瓜

- 4. 基本的功能已经有了一部分，赶紧测试一下
- 把上面2块代码合并为一个程序后，在代码的下面添加以下代码进行测试

```
mySweetPotato = SweetPotato()
print(mySweetPotato.cookedLevel)
print(mySweetPotato.cookedString)
print(mySweetPotato.condiments)
```

在上面的代码最后面添加如下代码：

```
print("-----接下来要进行烤地瓜了-----")
mySweetPotato.cook(4) #烤4分钟
print(mySweetPotato.cookedLevel)
print(mySweetPotato.cookedString)
```

- 5. 测试cook方法是否好用

```
[python@ubuntu:~/Desktop/test$ python3 oop3.py
0
生的
[]
-----接下来要进行烤地瓜了-----
4
半生不熟
python@ubuntu:~/Desktop/test$
```

# 应用:烤地瓜

- 6. 定义addCondiments()方法和\_\_str\_\_()方法

- 

```
def __str__(self):  
    msg = self.cookedString + " 地瓜"  
    if len(self.condiments) > 0:  
        msg = msg + "("  
        for temp in self.condiments:  
            msg = msg + temp + ", "  
        msg = msg.strip(", ")  
        msg = msg + ")"  
    return msg  
  
def addCondiments(self, condiments):  
    self.condiments.append(condiments)
```

## 3.6 隐藏属性（私有属性）

- 保护对象的属性
- 如果有一个对象，当需要对其进行修改属性时，有2种方法
- 对象名.属性名 = 数据 ---->直接修改
- 对象名.方法名() ---->间接修改
- 为了更好的保存属性安全，即不能随意修改，一般的处理方式为
- 将属性定义为私有属性
- 添加一个可以调用的方法，供调用
- 

```
class People(object):  
  
    def __init__(self, name):  
        self.__name = name  
  
    def getName(self):  
        return self.__name  
  
    def setName(self, newName):  
        if len(newName) >= 5:  
            self.__name = newName  
        else:  
            print("error:名字长度需要大于或者等于5")  
  
xiaoming = People("dongGe")  
print(xiaoming.__name)
```

## 3.6 隐藏属性（私有属性）

- 总结
- Python中没有像C++中public和private这些关键字来区别公有属性和私有属性
- 它是以属性命名方式来区分，如果在属性名前面加了2个下划线'\_\_'，则表明该属性是私有属性，否则为公有属性（方法也是一样，方法名前面加了2个下划线的话表示该方法是私有的，否则为公有的）。

## 3.7 `__del__()`方法

- 创建对象后，python解释器默认调用`__init__()`方法；
- 当删除一个对象时，python解释器也会默认调用一个方法，这个方法为`__del__()`方法



## 4、面向对象2





## 4.1 继承

- 继承介绍以及单继承
- 1. 继承的概念

在现实生活中，继承一般指的是子女继承父辈的财产，如下图

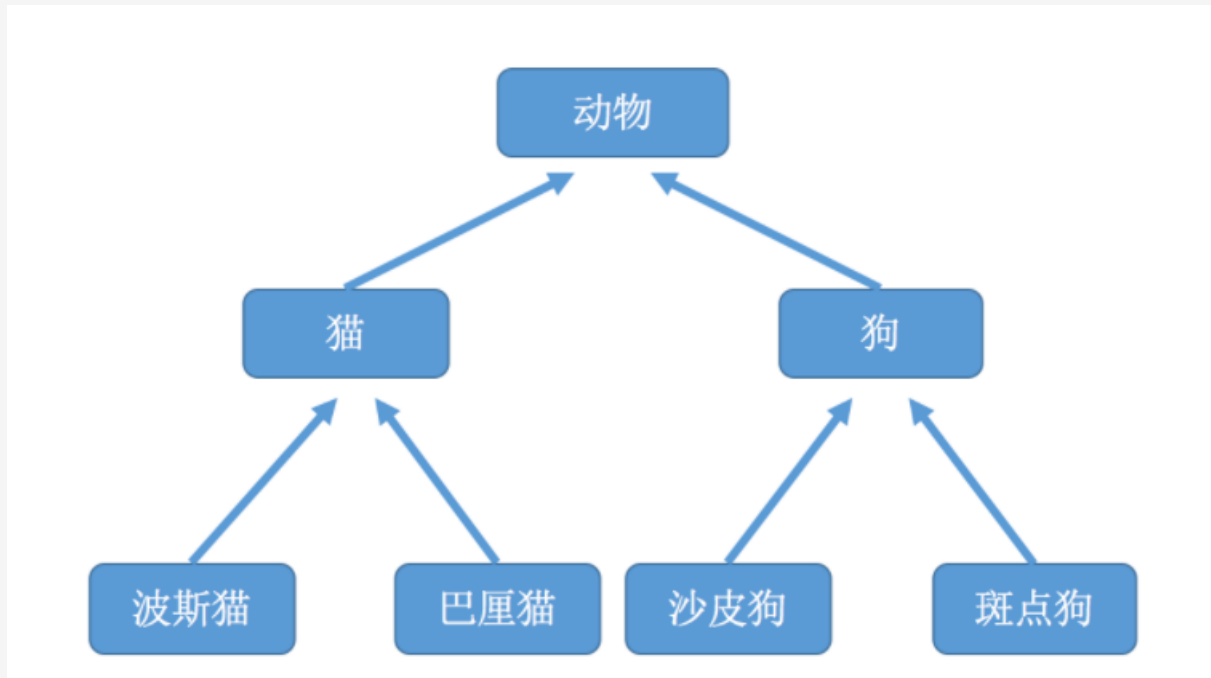


搞不好,结果如下..



## 4.1 继承

- 在程序中，继承描述的是事物之间的所属关系，例如猫和狗都属于动物，程序中便可以描述为猫和狗继承自动物；同理，波斯猫和巴厘猫都继承自猫，而沙皮狗和斑点狗都继承自狗，如下如所示：



# 4.1 继承

- 2. 继承示例

```
# 定义一个父类，如下：
class Cat(object):

    def __init__(self, name, color="白色"):
        self.name = name
        self.color = color

    def run(self):
        print("%s--在跑"%self.name)

# 定义一个子类，继承Cat类如下：
class Bosi(Cat):

    def setNewName(self, newName):
        self.name = newName

    def eat(self):
        print("%s--在吃"%self.name)
```

## 4.1 继承

- 说明：
- 虽然子类没有定义\_\_init\_\_方法，但是父类有，所以在子类继承父类的时候这个方法就被继承了，所以只要创建Bosi的对象，就默认执行了那个继承过来的\_\_init\_\_方法
- 总结
- 子类在继承的时候，在定义类时，小括号()中为父类的名字
- 父类的属性、方法，会被继承给子类

## 4.1 继承

- **3. 注意点**
  - A、私有的属性，不能通过对象直接访问，但是可以通过方法访问
  - B、私有的方法，不能通过对象直接访问
  - C、私有的属性、方法，不会被子类继承，也不能被访问
  - D、一般情况下，私有的属性、方法都是不对外公布的，往往用来做内部的事情，起到安全的作用

## 4.2 多继承

- 从图中能够看出，所谓多继承，即子类有多个父类，并且具有它们的特征

### 1. 多继承



## 4.2 多继承

```
# 定义一个父类
class A:
    def printA(self):
        print('----A----')

# 定义一个父类
class B:
    def printB(self):
        print('----B----')

# 定义一个子类，继承自A、B
class C(A,B):
    def printC(self):
        print('----C----')

obj_C = C()
obj_C.printA()
obj_C.printB()
```

运行结果:

```
----A----
----B----
```

说明

- python中是可以多继承的
- 父类中的方法、属性，子类会继承

## 4.2 多继承

- 注意点
- 想一想:如果在上面的多继承例子中，如果父类A和父类B中，有一个同名的方法，那么通过子类去调用的时候，调用哪个？

```
#coding=utf-8
class base(object):
    def test(self):
        print('----base test----')
class A(base):
    def test(self):
        print('----A test----')

# 定义一个父类
class B(base):
    def test(self):
        print('----B test----')

# 定义一个子类，继承自A、B
class C(A,B):
    pass

obj_C = C()
obj_C.test()

print(C.__mro__) #可以查看C类的对象搜索方法时的先后顺序
```



## 4.3 重写

- 重写父类方法与调用父类方法
- **1. 重写父类方法**
- 所谓重写，就是子类中，有一个和父类相同名字的方法，在子类中的方法会覆盖掉父类中同名的方法

```
#coding=utf-8
class Cat(object):
    def sayHello(self):
        print("halou-----1")

class Bosi(Cat):

    def sayHello(self):
        print("halou-----2")

bosi = Bosi()

bosi.sayHello()
```



A terminal window titled '桌面 - bash - 63x12' with the path '~ / Desktop - bash'. It shows the command 'python oop.py' being executed, which outputs 'halou-----2'.

```
dongGe@dongGe-Mac Desktop$ python oop.py
halou-----2
dongGe@dongGe-Mac Desktop$
```

## 4.3 重写

- 2. 调用父类的方法

```
#coding=utf-8
class Cat(object):
    def __init__(self,name):
        self.name = name
        self.color = 'yellow'

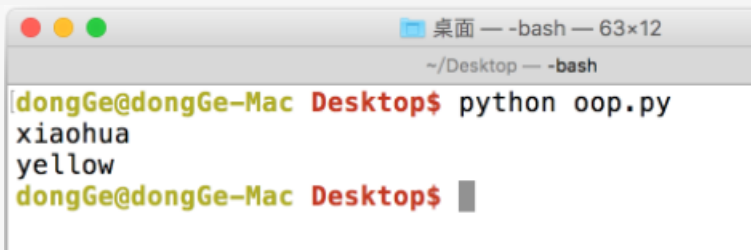
class Bosi(Cat):

    def __init__(self,name):
        # 调用父类的__init__方法1(python2)
        #Cat.__init__(self,name)
        # 调用父类的__init__方法2
        #super(Bosi,self).__init__(name)
        # 调用父类的__init__方法3
        super().__init__(name)

    def getName(self):
        return self.name

bosi = Bosi('xiaohua')

print(bosi.name)
print(bosi.color)
```



A terminal window titled '桌面 — -bash — 63x12' with a subtitle '~/.Desktop — -bash'. The prompt is 'dongGe@dongGe-Mac Desktop\$'. The command 'python oop.py' has been executed, resulting in the output 'xiaohua' and 'yellow' on separate lines. The prompt is now 'dongGe@dongGe-Mac Desktop\$'.

## 4.4 类属性、实例属性

- 在了解了类基本的东西之后，下面看一下python中这几个概念的区别
- 先来谈一下类属性和实例属性
- 在前面的例子中我们接触到的就是实例属性（对象属性），顾名思义，类属性就是类对象所拥有的属性，它被所有类对象的实例对象所共有，在内存中只存在一个副本，这个和C++中类的静态成员变量有点类似。对于公有的类属性，在类外可以通过类对象和实例对象访问

- 类属性

```
class People(object):  
    name = 'Tom' #公有的类属性  
    __age = 12   #私有的类属性  
  
p = People()  
  
print(p.name)      #正确  
print(People.name) #正确  
print(p.__age)     #错误，不能在类外通过实例对象访问私有的类属性  
print(People.__age) #错误，不能在类外通过类对象访问私有的类属性
```

## 4.4 类属性、实例属性

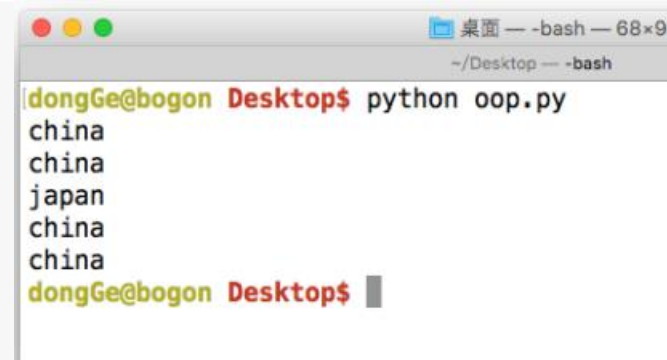
- 实例属性(对象属性)

```
class People(object):  
    address = '山东' #类属性  
    def __init__(self):  
        self.name = 'xiaowang' #实例属性  
        self.age = 20 #实例属性  
  
p = People()  
p.age = 12 #实例属性  
print(p.address) #正确  
print(p.name)    #正确  
print(p.age)     #正确  
  
print(People.address) #正确  
print(People.name)    #错误  
print(People.age)     #错误
```

## 4.4 类属性、实例属性

- 通过实例(对象)去修改类属性

```
class People(object):  
    country = 'china' #类属性  
  
print(People.country)  
p = People()  
print(p.country)  
p.country = 'japan'  
print(p.country)      #实例属性会屏蔽掉同名的类属性  
print(People.country)  
del p.country         #删除实例属性  
print(p.country)
```



```
桌面 - bash - 68x9  
~/Desktop - bash  
dongGe@bogon Desktop$ python oop.py  
china  
china  
japan  
china  
china  
dongGe@bogon Desktop$
```

## 4.4 类属性、实例属性

- 总结
- 如果需要在类外修改类属性，必须通过类对象去引用然后进行修改。如果通过实例对象去引用，会产生一个同名的实例属性，这种方式修改的是实例属性，不会影响到类属性，并且之后如果通过实例对象去引用该名称的属性，实例属性会强制屏蔽掉类属性，即引用的是实例属性，除非删除了该实例属性。

## 4.5 工厂模式

- 工厂方法模式的定义
- 定义了一个创建对象的接口(可以理解为函数)，但由子类决定要实例化的类是哪一个，工厂方法模式让类的实例化推迟到子类，抽象的CarStore提供了一个创建对象的方法createCar，也叫作工厂方法。
- 子类真正实现这个createCar方法创建出具体产品。创建者类不需要直到实际创建的产品是哪一个，选择了使用了哪个子类，自然也就决定了实际创建的产品是什么。

## 4.6 \_\_new\_\_方法

- \_\_new\_\_和\_\_init\_\_的作用

```
class A(object):  
    def __init__(self):  
        print("这是 init 方法")  
  
    def __new__(cls):  
        print("这是 new 方法")  
        return object.__new__(cls)
```

A()

- 总结

- \_\_new\_\_ 至少要有有一个参数cls，代表要实例化的类，此参数在实例化时由Python解释器自动提供
- \_\_new\_\_ 必须要有返回值，返回实例化出来的实例，这点在自己实现 \_\_new\_\_ 时要特别注意，可以return父类 \_\_new\_\_ 出来的实例，或者是object的 \_\_new\_\_ 出来的实例
- \_\_init\_\_ 有一个参数self，就是这个 \_\_new\_\_ 返回的实例，\_\_init\_\_ 在 \_\_new\_\_ 的基础上可以完成一些其它初始化的动作，\_\_init\_\_ 不需要返回值
- 我们可以将类比作制造商，\_\_new\_\_ 方法就是前期的原材料购买环节，\_\_init\_\_ 方法就是在有原材料的基础上，加工，初始化商品环节



## 4.6 \_\_new\_\_方法

注意点

```
In [8]: class A(object):
...:     def __init__(self):
...:         print(self)
...:         print("这是 init 方法")
...:
...:     def __new__(cls):
...:         print(id(cls))
...:         print("这是 new 方法")
...:         ret = object.__new__(cls)
...:         print(ret)
...:         return ret
...:
```

```
[In [9]: print(id(A))
140592776843416
```

```
[In [10]: a = A()
140592776843416
这是 new 方法
<__main__.A object at 0x105b96ac8>
<__main__.A object at 0x105b96ac8>
这是 init 方法
```

```
In [11]: █
```

## 4.7 单例模式

- 单例模式
- 1. 单例是什么
- 举个常见的单例模式例子，我们日常使用的电脑上都有一个回收站，在整个操作系统中，回收站只能有一个实例，整个系统都使用这个唯一的实例，而且回收站自行提供自己的实例。因此回收站是单例模式的应用。
- 确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，单例模式是一种对象创建型模式。

## 4.7 单例模式

- 2. 创建单例-保证只有1个对象

```
# 实例化一个单例

class Singleton(object):
    __instance = None

    def __new__(cls, age, name):
        #如果类数字能够__instance没有或者没有赋值
        #那么就创建一个对象，并且赋值为这个对象的引用，保证下次调用这个方法时
        #能够知道之前已经创建过对象了，这样就保证了只有1个对象
        if not cls.__instance:
            cls.__instance = object.__new__(cls)
        return cls.__instance

a = Singleton(18, "dongGe")
b = Singleton(8, "dongGe")

print(id(a))
print(id(b))

a.age = 19 #给a指向的对象添加一个属性
print(b.age)#获取b指向的对象的age属性
```