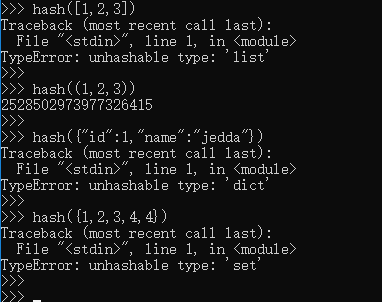
# 1 Python

## 标准数据类型

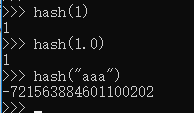
图1



（注意：在list，tuple，dict，set中只有tuple是可hash()的，即只有tuple是不可变的）



（Numbers和String都是可hash的，即Numbers和String也是不可变的）



### 1.1.1 list 列表（序列）

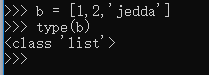
列表：列表是有序的元素的集合。

用a[0]的格式来访问列表中的元素，索引从0开始。

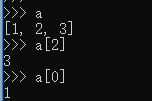
（用[1,2,3]的格式创建一个list）



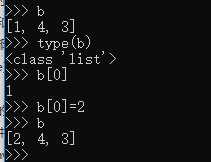
（list中可以包含类型不同的元素）



（用a[0]的格式来访问列表中的元素，索引从0开始）



（list的元素是可以被修改的）



#### 切片

:进行切片，切片的结果包含切片的第一个元素，不包含第二个元素。

（str[start:end]进行切片，截取子串，要头不要尾）



#### None

None表示什么都没有。

图1



#### in（成员资格）

in判断一个元素是否在列表中。

（in判断一个元素是否在列表）

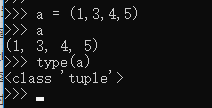


### 1.1.2 tuple元组

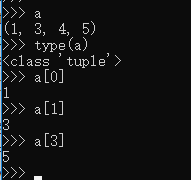
（注意：列表是可以修改的，元组是不能修改的）

（注意：用(1,2,3)是创建tuple，用[1,2,3]是创建list）

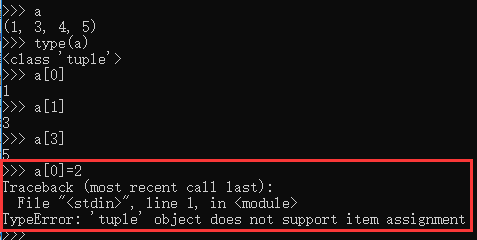
（用(1,2,3)的格式创建一个元组）



（用a[0]的格式访问元组的元素，索引最小值是0）



（tuple的元素是不能修改的）



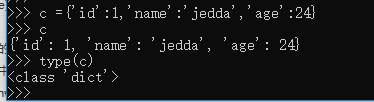
（tuple是可hash()的，所以tuple是不可变的）



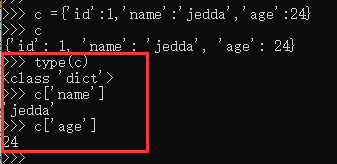
### 1.1.3 dict字典

字典：字段类似于Java的Map，是key/value的结构。

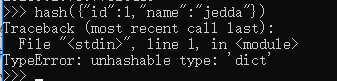
（用{‘key1’:value1,’key2’:value2}的格式创建dict）



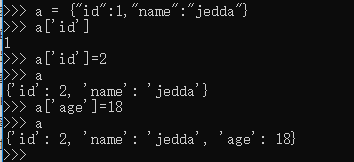
（用c[‘key1’]的格式访问dict的元素）



（dict是不可hash()的，所以dict是可变的）



（事实证明dict的确是可变的）



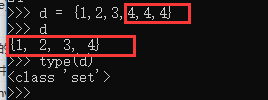
### 1.1.4 set集合

集合中不包含重复的元素。

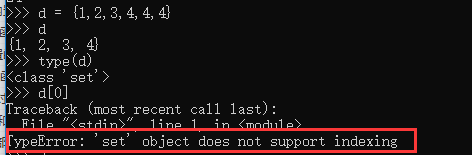
（注意：set是不重复的）

（注意：set是无序的，所以不能用索引访问）

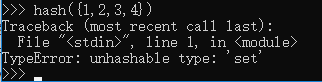
（用{1,2,3,4,4}的格式创建一个set，set是不重复的）



（set是无序的，所以不能用索引访问）



（set是不可hash()的，所以set是可变的）



### 1.1.5 str

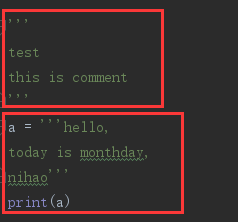
（注意：在python中，’hello’和”hello”是没有区别的，但是有些情况需要’’和””组合使用。当需要编写多行的字符串时，使用三对单引号或三对双引号）

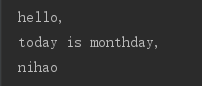
（注意：三对单引号或三对双引号即可以表示多行字符串，也可以表示多行注释；如果三对单引号或三对双引号的内容被使用到了，那它就是字符串；如果这个内容独立在一个地方，没有被使用到，那它就是注释）

（’’和””需要组合使用的情况）



（注意：三对单引号或三对双引号即可以表示多行字符串，也可以表示多行注释；如果三对单引号或三对双引号的内容被使用到了，那它就是字符串）



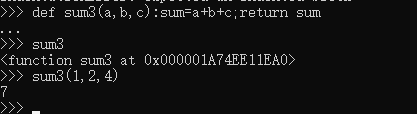


## 函数

（def创建函数的格式）



（def创建一个函数，return一个值）



### 匿名函数

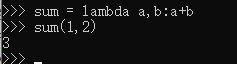
python中的匿名函数用lambda来创建。

（lambda创建匿名函数的格式）



（用lambda arg1,arg2:{arg1+agr2}的格式创建匿名函数）

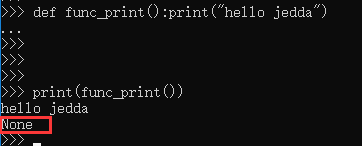
（注意：匿名函数会自动return最后一句语句）



### 无返回值的函数

函数如果无返回值，则返回值是None。

（print打印函数的返回值，如果该函数无返回值，则打印出None，说明无返回值的函数的返回值是None）



### 返回值是一个函数名的函数

当一个函数的返回值是一个函数名时，可以用一个变量接收函数名，然后用()调用。

图1

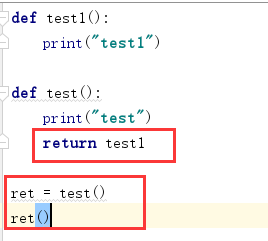
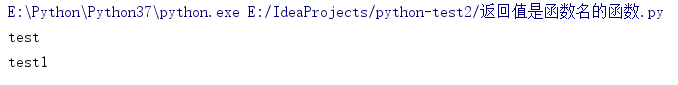


图2



### 匿名函数

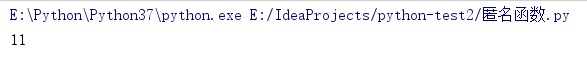
使用lambda 形参:函数体 的格式来定义匿名函数，匿名函数没有函数名。

（注意：lambda函数的函数体自动return）

图1（注意：a是变量，不是函数名）



图2



### map内置函数

map，对列表中的每一个元素，经过func计算后，计算出对应的另一个元素。

（注意：一对一）

（注意：map的结果是一个迭代器，该迭代器只能迭代一次，第二次迭代时迭代器会清空）

图1

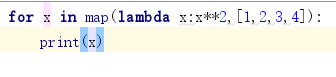
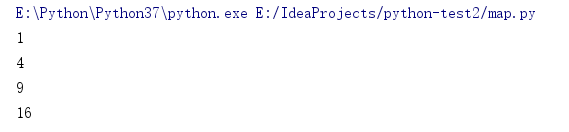


图2



### filter内置函数

和map类似，对列表中的每一个元素，经过func过滤后（只要那些func返回true的元素），保留剩余的元素。

（注意：filter的结果也是一个迭代器）

图1

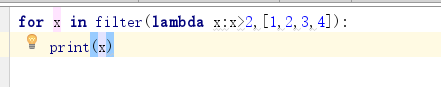
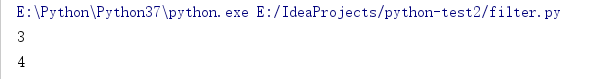


图2



### reduce内置函数

reduce，对列表中的两两元素执行func计算，两两的结果再和下一个组成两两（其实就是合并过程），最终得出一个值。

（注意：reduce的结果是一个值，不是一个迭代器）

图1

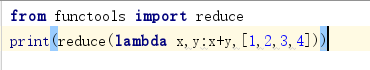
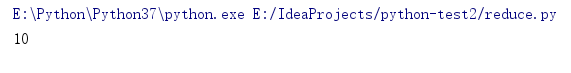


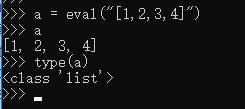
图2



### eval内置函数

eval将字符串解析出来。

（eval(“[1,2,3,4]”)将字符串解析成list）



（eval(“1\*2+3”)计算字符串表达式的值）

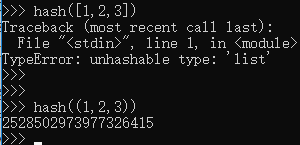


### hash内置函数

可hash的则是不可变数据类型，不可hash的则是可变数据类型。

（注意：只要变量的值不变，hash值就不变）

（可hash的则是不可变数据类型，不可hash的则是可变数据类型，验证知道list是可变的，所以不可hash()；tuple是不可变的，所以可以hash()）



### zip内置函数

zip是压缩/拉链，参数是多个可迭代对象，将多个可迭代对象中对应位置的元素组成元组，最后返回元组组成的zip对象。

可以用list将zip对象转换成列表。

图1

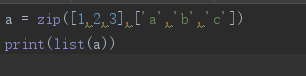
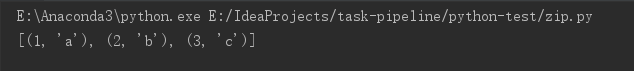


图2



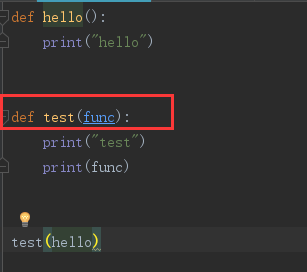
### 高阶函数

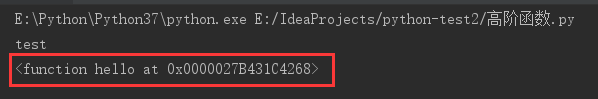
只要满足以下其中一个条件就是高阶函数：

1. 函数的参数是一个函数名
2. 函数的返回值是一个函数名

（函数的参数是一个函数名）

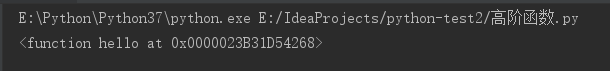
（注意：打印func打印出来的是func这个函数的内存地址）





（函数的返回值是一个函数名）





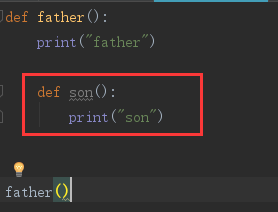
### 函数嵌套

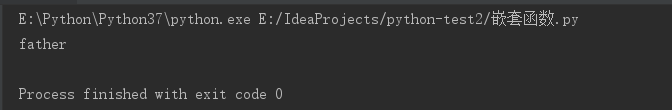
函数嵌套就是在函数内部定义另一个函数。

（注意：是定义，不是调用）

（注意：如果函数A内部定义了函数B，则B只能在A内部被调用，而不能在A外部被调用，因为B其实就是A的一个局部变量）

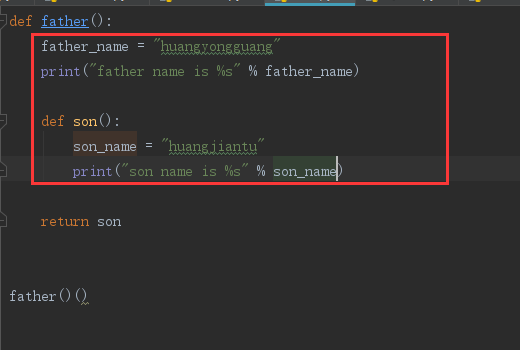
（函数嵌套就是在函数内部定义另一个函数）

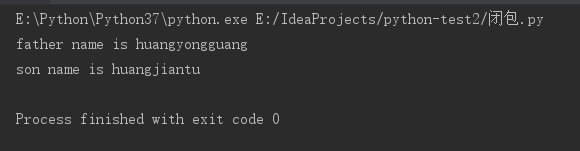




### 闭包

当函数father内部定义了函数son，且返回值是内部函数son的名称时，内部函数son本身（def son()），及其能使用的局部变量（father\_name,son\_name）组成的包，称为闭包。

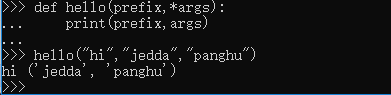




### 函数参数是\*args，\*\*kwargs

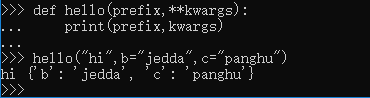
\*args将函数该参数位置的所有参数封装到args这个tuple中。

（将第二个位置的参数：”jedda”,”panghu”都封装到args这个tuple中）

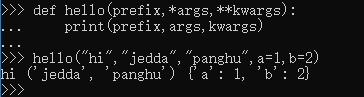


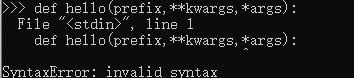
\*\*kwargs将函数该参数位置的带参数名关键字的所有参数封装到kwargs这个dict中。

（将第二个位置所有带参数名的参数，都封装到kwargs这个dict中）



（注意：当arg，\*args，\*\*kwargs组合使用时，必须按照arg，\*args，\*\*kwargs顺序定义函数，否则会报错）





## 可变和不可变对象

在python中，strings，tuples和numbers是不可变的对象，list和dict是可变的对象。

图1



### 可变对象和不可变对象的参数传递

不管是可变对象和不可变对象，传递的都是对象的地址。但当尝试去修改一个不可变的对象时，因为该对象不可变，所以只能创建一个新的对象，然后让引用指向这个新对象。

如果该对象可变，则引入传入的参数是对象的地址，所以直接可以对该对象本身进行修改，且修改成功。

（如果该对象（String）不可变，所以只能创建一个新的对象，然后让引用指向这个新对象）



（如果该对象（list）可变，则引入传入的参数是对象的地址，所以直接可以对该对象本身进行修改，且修改成功）



## 全局变量和局部变量

函数内是局部变量，函数外是全局变量。

（注意：全局变量是没有任何缩进的，局部变量是有缩进的）

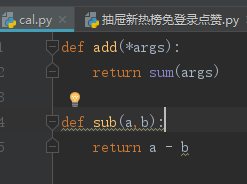
图1



## 模块（Module）

在python中，一个模块就是一个Python文件（\*.py文件）。

图1



### 引入模块

模块定义好后，可以用import引入模块。

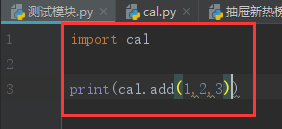
import的执行过程是：

1. 如果import cal，会把cal.py这个\*.py文件先执行一遍。
2. 然后引入对应的变量或函数。

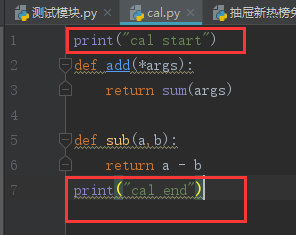
（import引入多个模块）

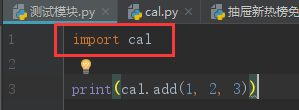


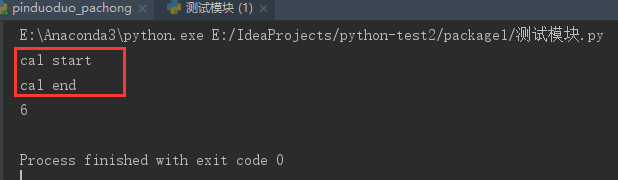
（import引入模块后，可以使用该模块中定义的函数）



（如果import cal，会把cal.py这个\*.py文件先执行一遍）







### 引入模块的一部分函数

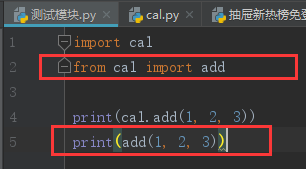
模块定义好后，可以用from...import...引入模块的一部分函数。通过from...import...引入函数后，就可以不必使用模块名.函数名方式来调用函数，而是可以直接使用函数名方式来调用函数。

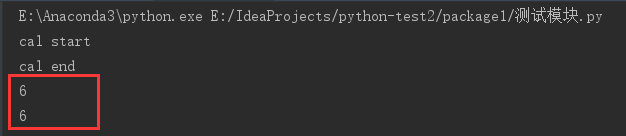
（注意：用from...import...的方式，就算只import该模块的一部分函数，也会执行整个from指定的\*.py文件的全部代码）

（从模块math中导入sqrt函数）



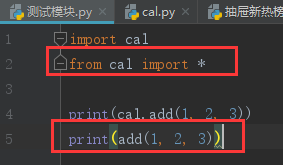
（从cal模块中导入add函数，直接使用函数名方式来调用函数）





（使用from ... import \* 可以导入该模块的所有函数，且函数可以直接使用函数名方式来调用）

（注意：这种方式不推荐，因为多个模块可能有多个重名函数；而且很容易导致自己定义的函数和导入的\*的某个函数重名了）



### 使用模块中的函数

使用import导入模块后，使用module.function的方式调用模块的函数。

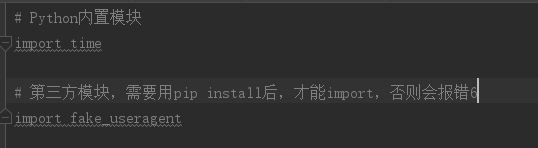
（使用import导入模块后，使用module.function的方式调用模块的函数）



### 内置模块和第三方模块

内置模块是Python自带的，直接可以import；第三方模块是需要pip install后，安装到Python目录，然后才可以import。

（内置模块是Python自带的，直接可以import；第三方模块是需要pip install后，安装到Python目录，然后才可以import）



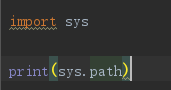
### import的模块查找顺序

会先从执行的\*.py文件所在目录（同级目录的会找）找，然后再往上一级一级找到工程目录，再找Python的一些环境变量的目录。

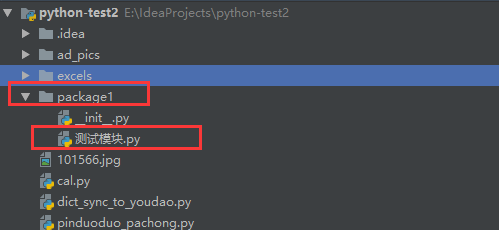
可以用sys.path查看，sys.path的结果的第0个元素优先级最高。

（注意：不会从当前\*.py文件所在目录的子目录去找）

（注意：不管有多少个\*.py文件，这些\*.py文件中有多少个import，但只有1个执行文件，那么import都会从该执行的\*.py文件所在目录开始找）







### Python的内置模块

#### time模块

## #! /usr/bin/env python

如果不加#! /usr/bin/env python，则必须指定python解释器来执行\*.py脚本，比如c:\python hello.py。

但UNIX支持在py脚本的第一行用#! /usr/bin/env python指定python的绝对路径后，不管python库在什么地方，都不需要再指定python解释器，直接hello.py运行py脚本。

当同一台就机器安装了多个python版本时，使用python3/python2代替python。

## 1.7多态

python中的多态，让我们不需要知道对象所属什么类，因为不管对象是什么类的实例，它只要该类覆盖了该函数，就可以利用多态来调用。

图1

（不需要x到底是字符串还是列表，因为多态的存在，只要字符串和列表都覆盖了count函数即可）





图2

（因为多态的关系，所以 + 既可以用于整数，也可以用于字符串，因为整数和字符串类型都支持加法）



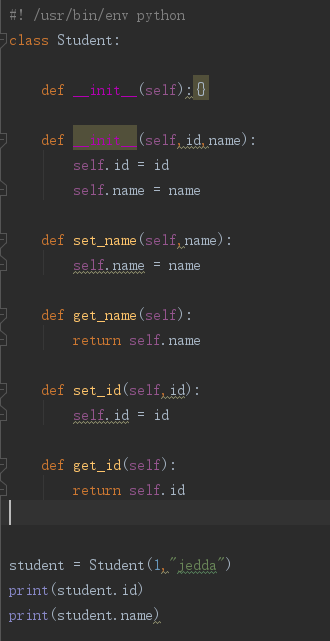
## 1.8类

### 1.8.1 自定义类

self类似于Java的this，表示对象实例本身。

（注意：class中的方法的第一个参数默认是self）

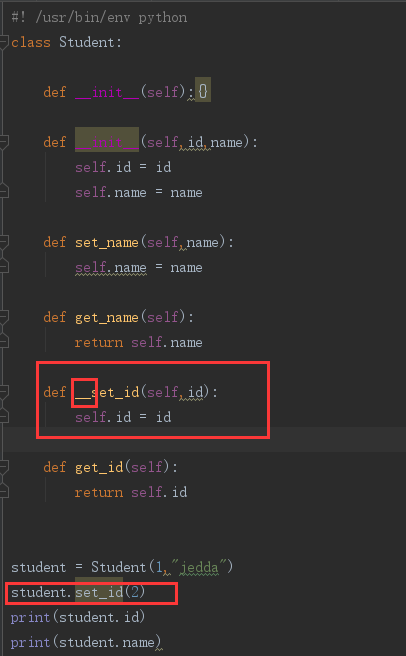
（创建自定义的class）

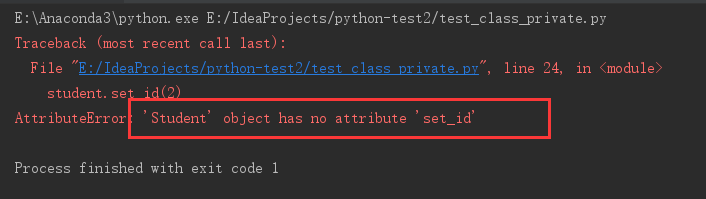


### 1.8.2 私有属性和私有方法

使用2个下划线\_\_加在属性/方法前面，表示这个属性/方法是私有的。

（\_\_将属性/方法变成私有，class以外的代码无法访问该属性/方法）

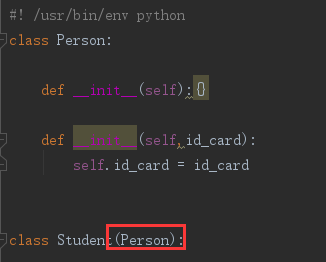




### 1.8.3 指定超类

在类名后面用()指定超类。

（在类名后面用()指定超类）



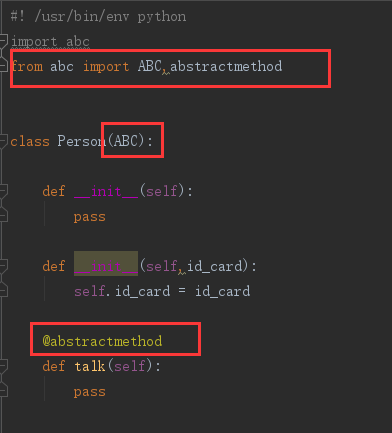
### 1.8.4 抽象基类

利用abc模块来创建抽象类。

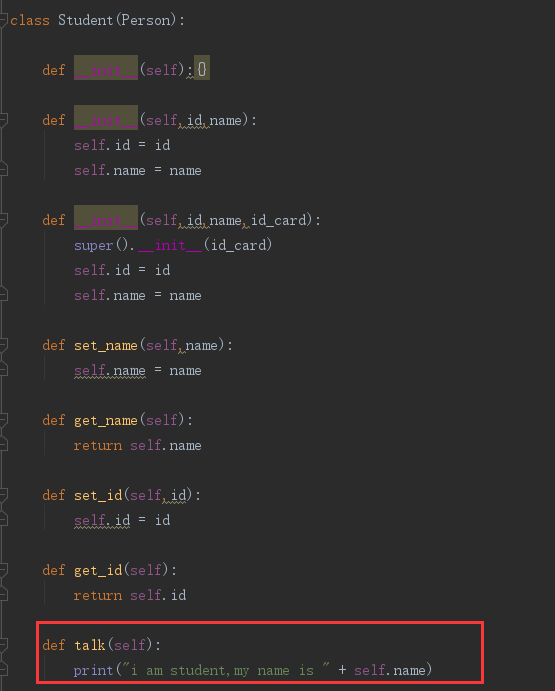
（注意：抽象类不能被实例化。）

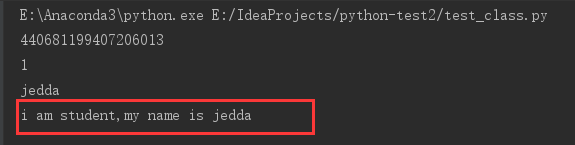
（注意：如果一个类继承了抽象类，就必须实现所有抽象方法，否则该类也是抽象类）

（使用abc模块来创建一个抽象类）



（Student继承了抽象类Person，重写了task()方法）

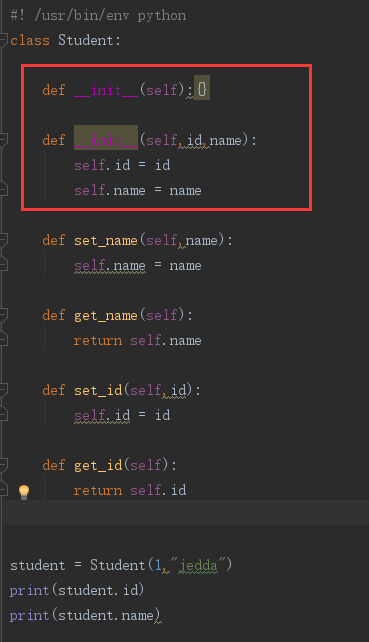




### 1.8.5 构造函数

构造函数命名为\_\_int\_\_，构造函数不同于普通函数在于：对象创建后会自动调用构造函数。

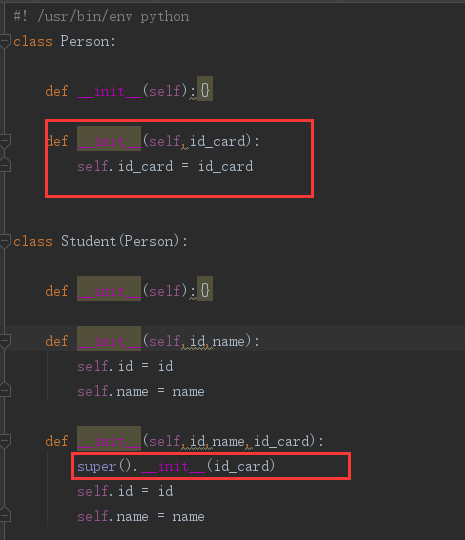
（\_\_init\_\_创建构造函数，以及重载构造函数）

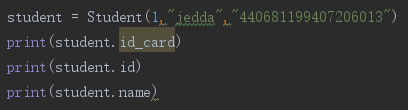


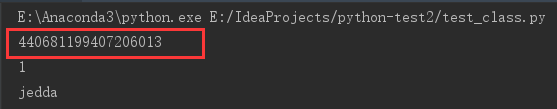
重写构造函数时，如果要继承超类的属性和方法，必须在构造函数中调用超类的构造函数。

（如果要继承超类的属性和方法，必须在构造函数中调用超类的构造函数）

（注意：如果在构造函数中不调用超类的构造函数，尝试去调用超类的属性和方法时，会报错）





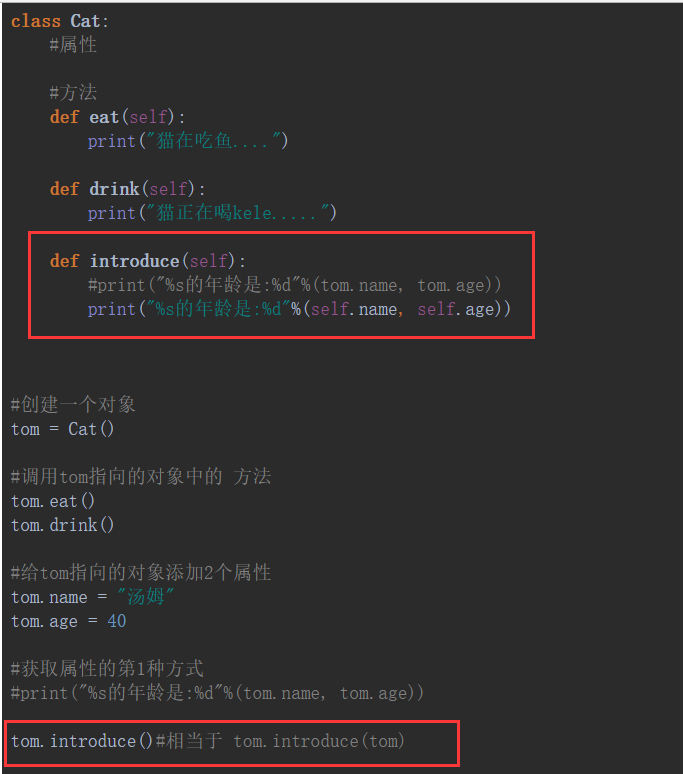


### 1.8.6 self

self类似于Java的this，表示对象实例本身。

在调用对象的方法时，python解释器会自动将对象实例作为方法的第一个参数传入，所以调用者只需要传入其他参数即可。

图1



### 1.8.7 property

用property函数创建了一个特性，然后将名称size关联到这个特性上，这个名称size使用时就像该类的属性一样，但size的计算依赖于property函数中的get和set方法。

图1

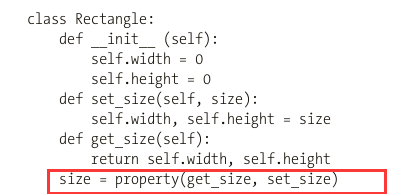
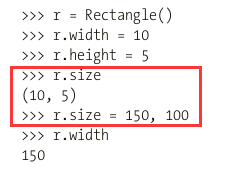


图2



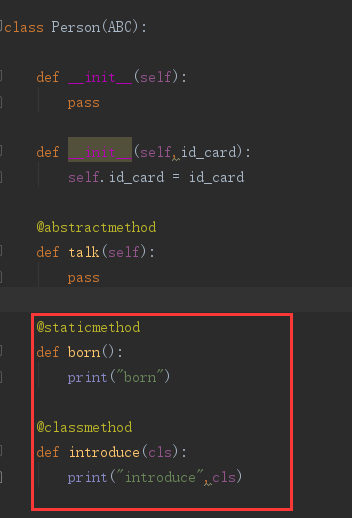
### 1.8.8 静态方法和类方法

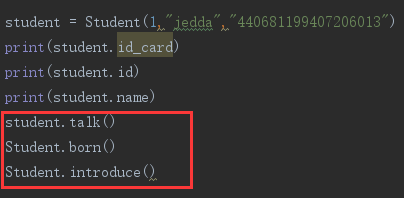
静态方法不需要self作为参数（加上self为参数也不会报错），一般放入和实例无关的逻辑。

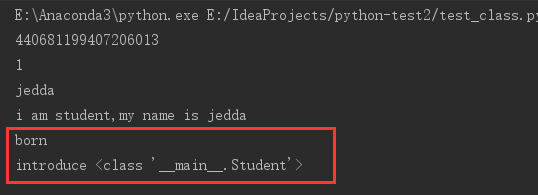
类方法的第一个参数必须是self。

静态方法和类方法都可以通过类和实例直接调用，但类方法有参数cls表示该类（注意：cls表示的是该类，而不是该类的实例）。

（创建静态方法和类方法）







### 1.8.9\_\_getsttr\_\_和\_\_setattr\_\_

\_\_getsttr\_\_和\_\_setattr\_\_用于属性被访问或者赋值时被调用。

图1

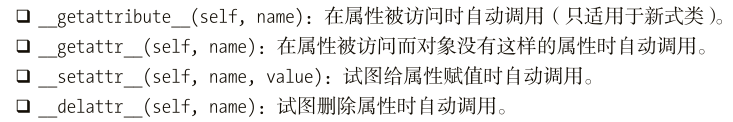
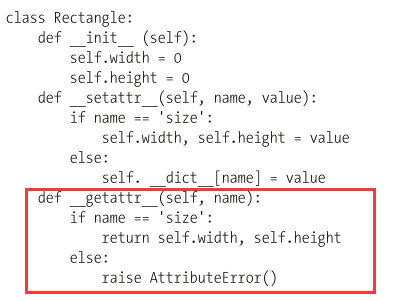
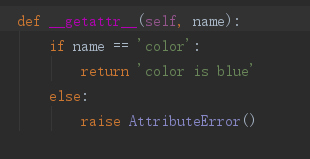
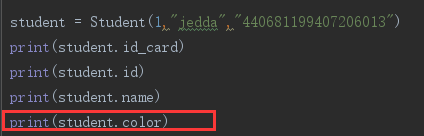


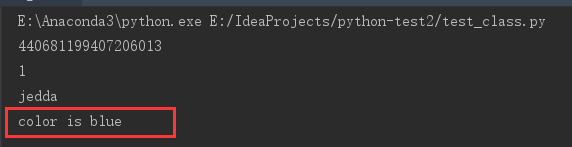
图2



（当用.获取某属性的值，且在类的继承树上没有找到该属性，\_\_getattr\_\_()会被调用）



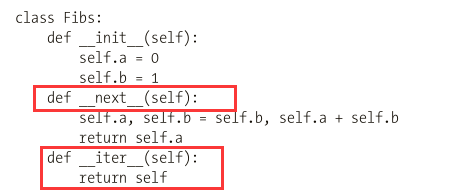




### 1.8.10 \_\_iter\_\_

\_\_iter\_\_魔法方法返回一个迭代器，\_\_next\_\_获取迭代器中的下一个元素。

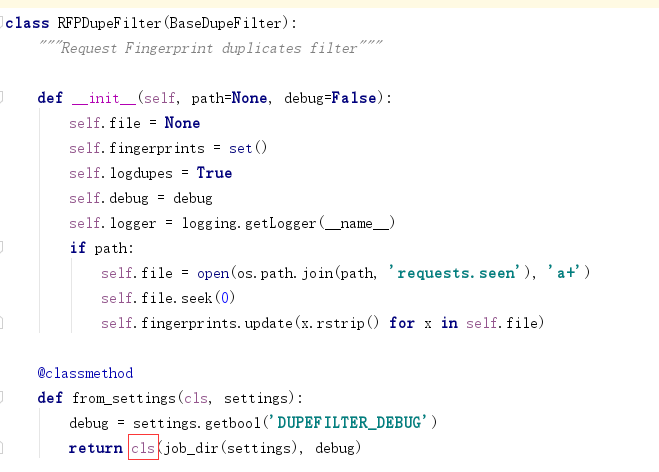
图1



### 1.8.10 cls

cls表示本类的类名。cls(xxx)其实就是调用构造函数。

（cls表示本类的类名。cls(xxx)其实就是调用构造函数）



## 1.9 异常

在python中，几乎所有的异常都是Exception类的之类。

### 1.9.1 自定义异常

自定义异常时，只需要指定超类是Exception类。

图1



### 1.9.2 捕捉异常进行处理

使用try/except捕捉异常。

图1



## 1.10 文件

### 1.10.1 open

open函数是io模块的函数，而io模块是自动导入的。open函数返回一个类似于文件的对象。

图1

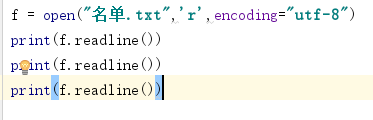
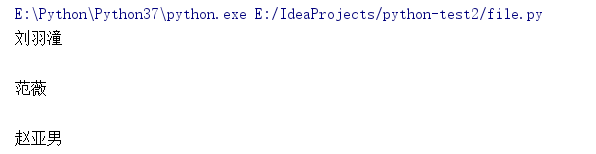


图2



### 1.10.2 read()

read()一次读取全部，执行完后光标会到达文件的末端，再用readline()会读不出数据。 （注意：read()返回的是文件中所有的字符串。）

（注意：read(3)在t模式下读的是3个字符，不是3个字节）

图1

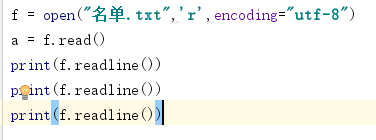
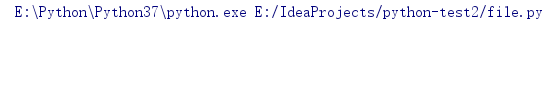


图2



### 1.10.3 文件模式

r：读模式

w：写模式

（注意：w模式当文件不存在时会创建文件）

（注意：w模式会清空文件中已有的数据，然后再写入）

a：附加模式

（注意：a模式就是追加的w，a模式不会清空文件中已有的数据，而是追加在文件的末尾）

r+：可读可写模式

（注意：r+默认的光标位置是文件的开头，所以r+模式下会从光标位置写入）

b：以字节模式读/写。

（注意：不能指定编码，因为每一种编码都是以N数量的字节来读取。）

t：以文本模式读/写。

（注意：如果什么模式都不指定，默认是rt，以文本模式读 ）

### 1.10.4 with

with可以同时打开多个文件。

图1



### 1.10.5 编码和解码

编码：字符串->二进制字节码

解码：二进制字节码->字符串

码是二进制码。

### 1.10.6 打开文件的编码和文件的实际编码

注意：打开文件按的编码和文件的实际编码是两回事，encoding函数返回的是文件打开的编码。

图1（文件打开编码是gbk）

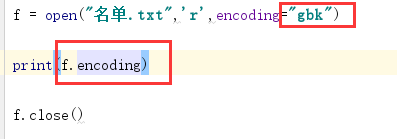


图2

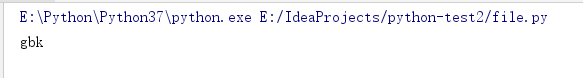
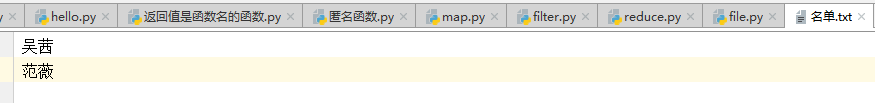


图3（文件实际编码是utf-8）



### 1.10.7 flush()

文件open打开后数据会加载到内存，之后读/写都在内存中，flush是强制将内存中的数据写入到磁盘中。

### 1.10.8 tell()

返回光标的位置，以字节为单位。tell()返回的是当前光标前面的字节数。

（注意：utf-8编码一个中文3个字节，windows中换行是\r\n，共2字节）

图1

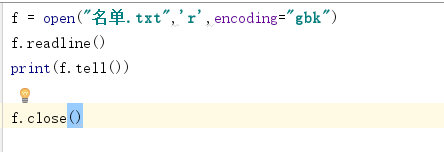


图2



图3



（注意：seek有3种默认，0（默认）：以文件开头（光标0）为相对位置移动；1：以当前位置位置为相对位置移动；2：以文件末尾为位置移动）

图3

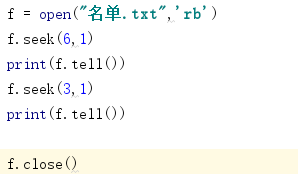
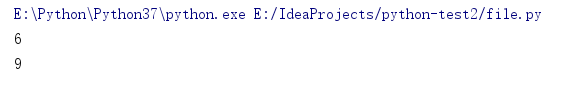


图4



### 1.10.9 seek

seek将光标移动到指定字节的位置。

（注意：只要不是read(3)方法，其余方法在处理光标的时候，都是以字节为单位）

图1

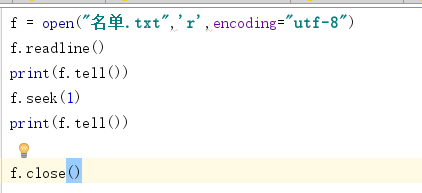
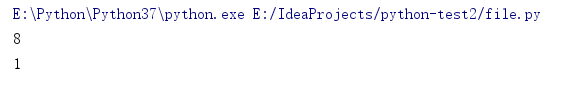


图2



### 1.10.10 truncate

truncate是截取，truncate(10)只保留0~10字节的数据。

图1

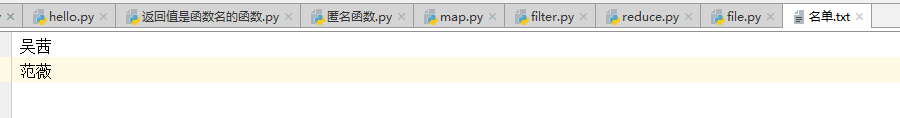


图2

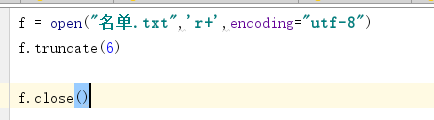
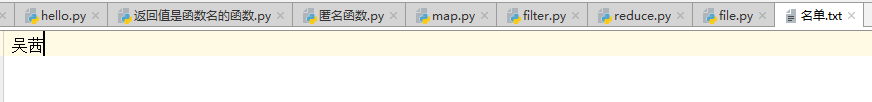


图3



## 1.11 迭代器

### 1.11.1 迭代器协议

迭代器协议：对象必须提供一个\_\_next()\_\_方法，这个next方法要么返回迭代器中的下一个元素（爸爸生儿子），要么引起一个StopIteration异常来终止迭代。

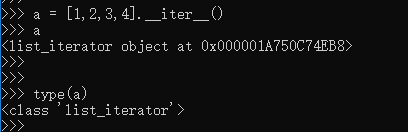
（注意：迭代器只能往下走，不能玩前退，因为爸爸能生儿子，儿子不能生爸爸）

### 1.11.2 可迭代对象

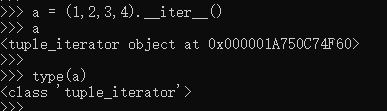
可迭代对象就是遵循了迭代器协议的对象。

（注意：字符串，列表，元组，集合，文件对象因为都没有\_\_next\_\_()方法，所以实际上都不是可迭代对象，只不过因为for循环调用了它们内部的\_\_iter\_\_()魔法方法，返回了一个可迭代对象）

（list内部的\_\_iter\_\_()魔法方法返回了一个可迭代对象）



（tuple内部的\_\_iter\_\_()魔法方法返回了一个可迭代对象）



## 1.12 生成器

生成器可以理解成是一种数据类型，但它实现了迭代器协议，所以同时也是可迭代对象。因为它实现了可迭代协议，所以可以直接调用\_\_\_next\_\_()方法，不需要调用\_\_iter\_\_方法来获取可迭代对象。

### 1.12.1 生成器的创建

1.由函数通过yield替换return来返回一个生成器。

（注意：yield可以yield多次，每次调用生成器的\_\_next()\_\_函数取下一个yield的值，每次\_\_next()\_\_执行上一次yield到下一次yield之间的代码）

（注意：可以使用next(生成器)来代替生成器.\_\_next\_\_()，也可以用生成器.send()代替生成器.\_\_next\_\_()）

图1

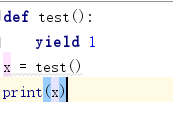


图2

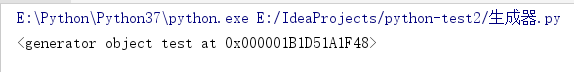


图3（yield return多次）

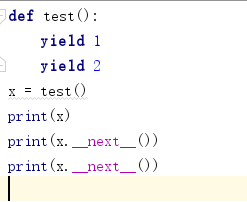
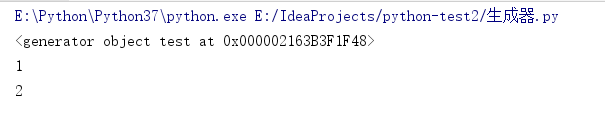


图4（每次\_\_next()\_\_取下一个return的值）



1. 使用表达式返回一个生成器：

（注意：用表达式创建生成器，要用()包含表达式的内容）

图1

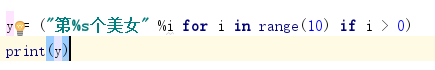
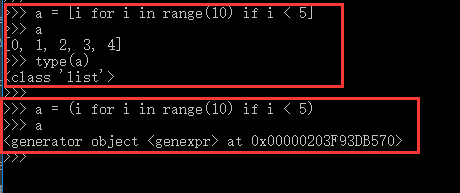


图2

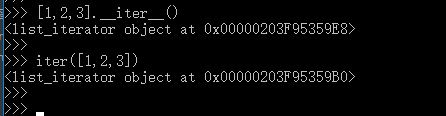


（用表达式创建生成器，要用()包含表达式的内容）



1. 调用集合的\_\_iter\_\_()方法，或者iter(集合)获得该集合的生成器。

（调用集合的\_\_iter\_\_()方法，或者iter(集合)获得该集合的生成器）



### 1.12.2 生成器的优点

1.生成器不会一次把这个可迭代对象的所有元素都加载在内存中，只会在每次获取下个元素时把该元素载入内存，所以生成器更节省内存。

（注意：调用list.\_\_iter()\_\_获取可迭代对象这个工程不消耗内存，但list定义时list=[1,2,3,4...1000000]时确实占用比较大的内存）

1. 生成器函数可以保留函数的上一次状态。

### 1.12.3 send

send(x)使生成器继续执行，获取下一个元素，相当于调用\_\_next()\_，同时将x赋值给上一个yield的返回值。

（注意：第一次运行生成器获取第一个元素时，只能用\_\_next()\_\_或send(None)）

图1

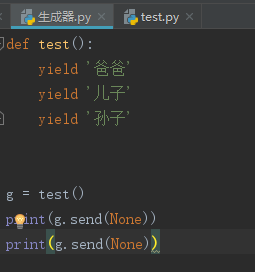


图2

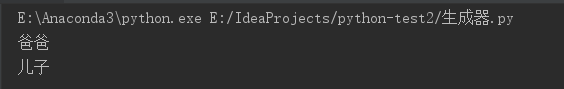


图3（将3赋给上一个yield的返回值first）

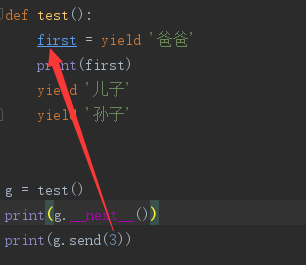
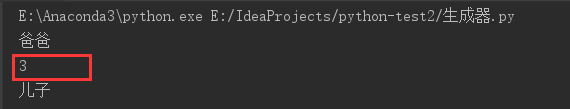


图4

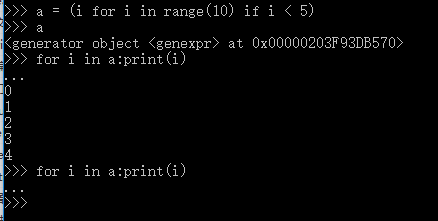


### 1.12.3 生成器只能迭代一次

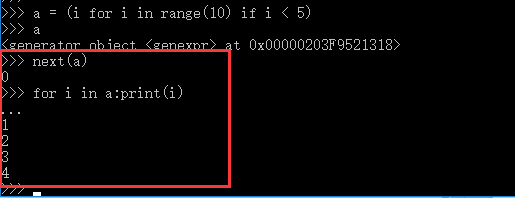
因为生成器实现了迭代器协议，所以生成器的每个值只能被迭代一次。

（注意：生成器如果分开了2次去迭代，第一次未迭代的值，在第二次还能被迭代）

（因为生成器实现了迭代器协议，所以生成器只能被迭代一次）



（生成器如果分开了2次去迭代，第一次未迭代的值，在第二次还能被迭代）



## 1.13 三元表达式

三元表达式是：用空格分开后，有3部分的表达式。

### 1.13.1 三元表达式实现if语句

如果if返回true，返回if前面的值；如果返回false，返回else后面的值.

图1

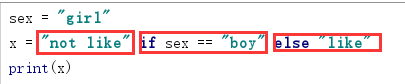
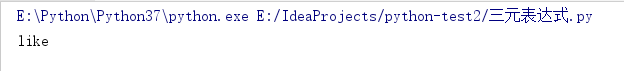


图2



### 1.13.1 使用三元表达式构造列表list

图1

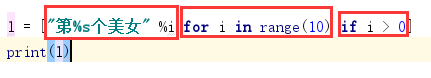
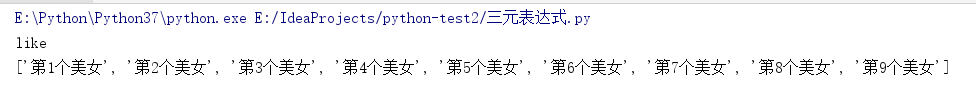


图2



## 1.14 装饰器

装饰器本质就是一个附加函数，为其他函数添加附加的功能。

装饰器的2个原则：

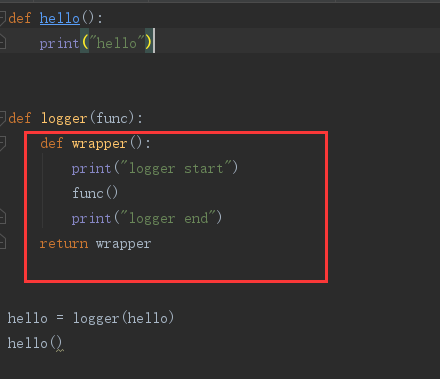
1. 不修改被修饰函数的代码
2. 不修改被修饰函数的被调用方式

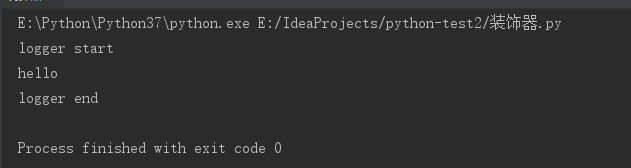
优点：可以在不修改被修饰函数的代码的情况下，为该函数附加功能。

装饰器=高阶函数+嵌套函数+闭包

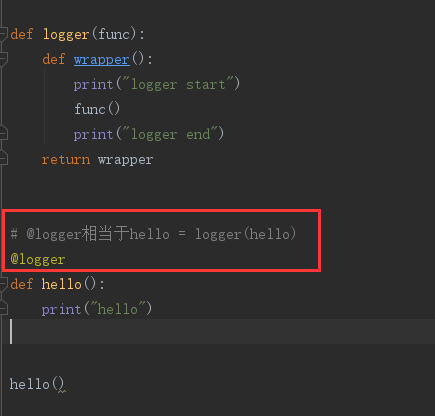
（注意：装饰器的最终结果就是：执行hello()，实际上是执行了wrapper()）

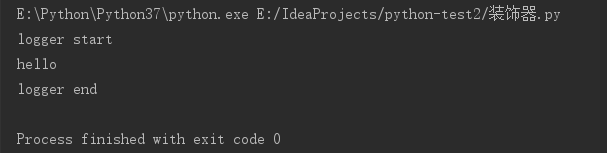
（装饰器=高阶函数+嵌套函数+闭包）





（上面的写法还要手动为待修饰方法执行hello=logger(hello)，可以使用Python的语法糖@logger来代替hello=logger(hello)）



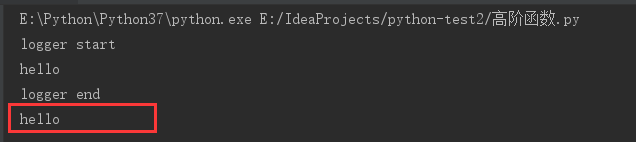


### 1.14.1 高阶函数实现装饰器

仅仅依靠高阶函数，无法满足装饰器的功能，因为实际上logger(hello)执行了一次hello()，然后返回了hello的地址，再执行了hello()，实际上执行了2次hello()，并在第一个hello()中加入了logger的逻辑。

（实际上执行了2次hello()，并在第一个hello()中加入了logger的逻辑）

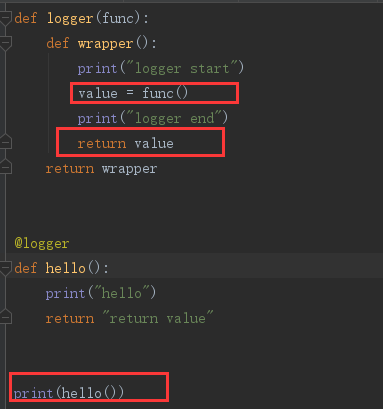


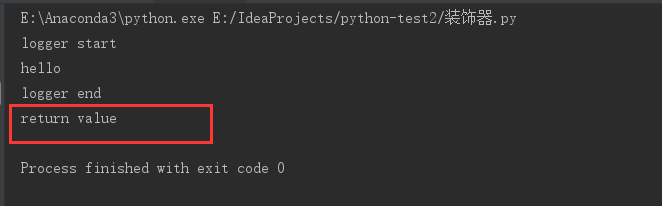


### 1.14.2 带返回值的装饰器

（注意：装饰器的最终结果就是：执行hello()，实际上是执行了wrapper()，因为logger(func)执行的过程中只是定义（def）了一个wrapper函数，然后返回了，所以logger(func)的执行后返回值是wrapper）

（带返回值的装饰器）

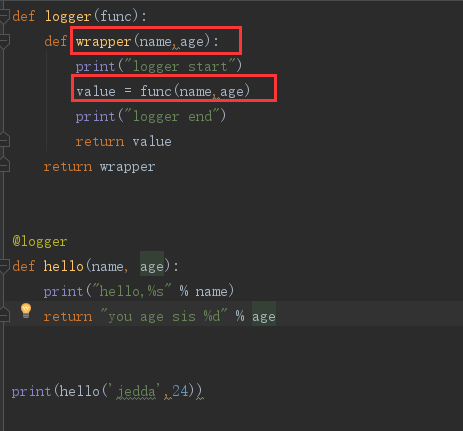


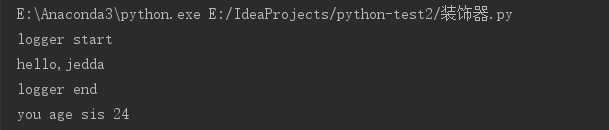


### 1.14.3 装饰器修饰带参数的函数

因为装饰器的最终结果就是执行hello()，实际上是执行了wrapper()，因为logger(func)执行的过程中只是定义（def）了一个wrapper函数，然后返回了，所以logger(func)的执行后返回值是wrapper。所以，如果要修饰带参数的函数，只要在wrapper()中加上参数即可。

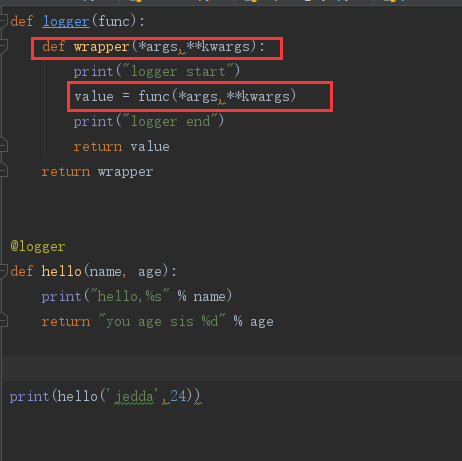
（在wrapper()中加上参数，修饰带参数的函数）

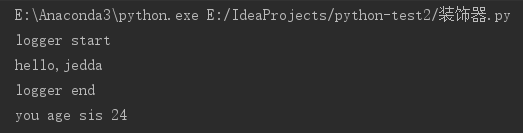




（使用\*args以及\*\*kwargs可以适配待修饰函数的任意个参数）

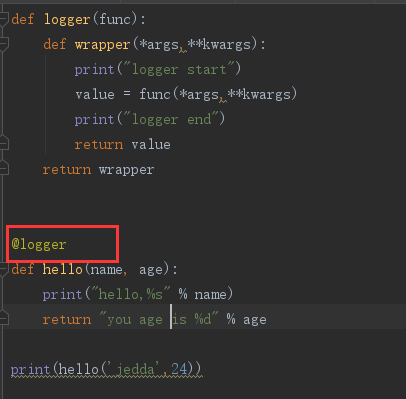
（注意：这里在wrapper内部的func传入的参数是\*args,\*\*kwargs，而不能是args,kwargs，因为args是一个tuple，kwargs是一个dict；而\*args就是将tuple还原成多个参数，\*\*kwargs是将dict还原成多个参数）

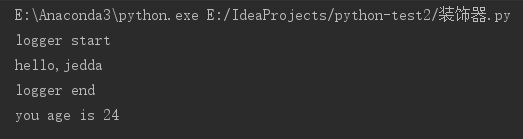




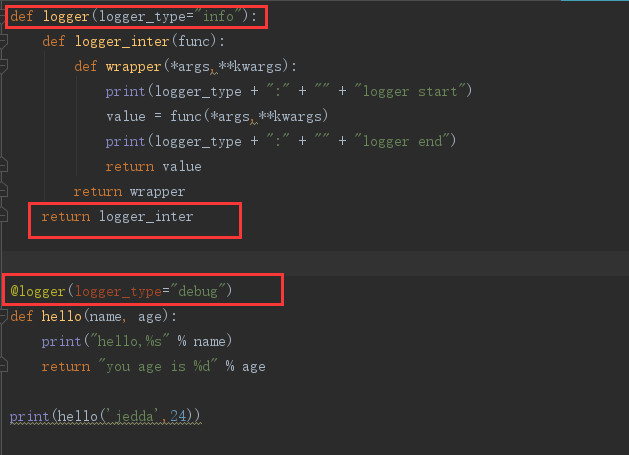
### 1.14.3 装饰器本身加上参数

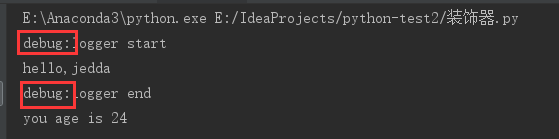
（原装饰器，@logger没有参数）





（在外面再套一层，因为logger(logger\_type=”debug”)的执行结果就是logger\_inter）



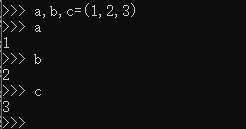


## 1.15 解压序列

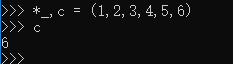
解压序列就是将某个序列解压成多个值，然后一一给左边的变量赋值。

解压序列可以给多个变量同时赋值，但必须要求左边变量的个数和右边值的个数一一对应。

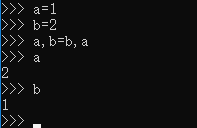
（加压序列可以给多个变量同时赋值）



（可以用\*\_来代替任意个数的变量）



（解压序列交换2个变量的值）



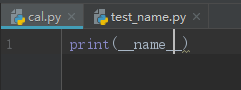
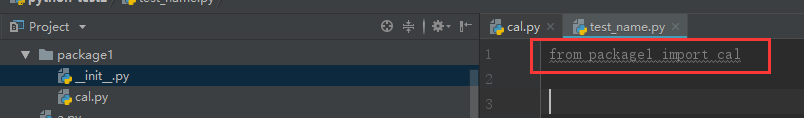
## 1.16 Python内置变量

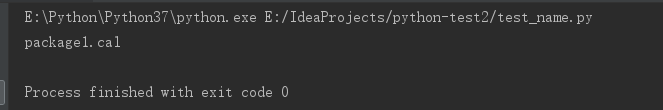
### 1.16.1 \_\_name\_\_

每个.py文件都有一个\_\_name\_\_内置属性，当一个x.py的文件被直接运行时，\_\_name\_\_的值就等于\_\_main\_\_；但如果是被其他.py文件import进来，那么被import的.py文件的\_\_name\_\_就等于该py文件的路径：package.module。

（注意：这种规范是因为你编写的.py文件不仅仅是你自己运行，还可能被别人import进来运行，但有些代码是你希望你自己直接运行才会去执行的，所以使用\_\_name\_\_ == \_\_mian\_\_来达到这种限制效果）

（被import的.py文件的\_\_name\_\_就等于该py文件的路径：package.module）



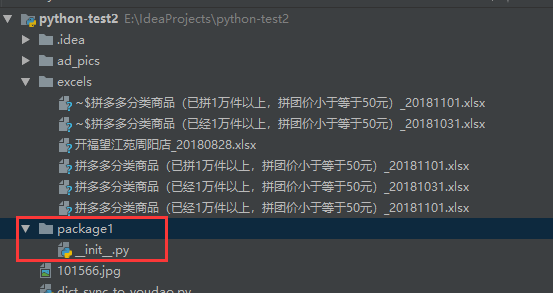


## 1.17 Python Package（包）

包是为了防止模块名重名；而模块是为了分工，方便功能的维护。

包也是一个文件夹，但包和其他文件夹的区别是：包的目录下会多一个\_\_init\_\_.py文件。

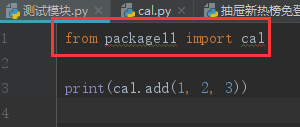
（注意：\_\_init\_\_.py文件一开始内容是空的）

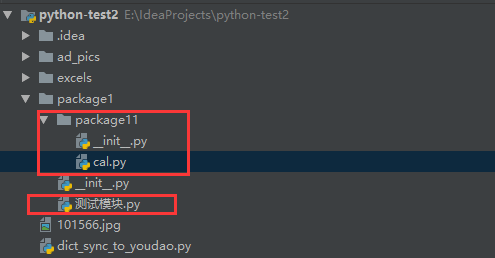


### 1.17.1 from package import module

从某个包引入该模块。

（注意：通过from package import module后，必须通过module.function来调用函数，不能直接用函数名的方式调用函数，因为import后面跟的是模块名）





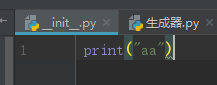
### 1.17.2 import package

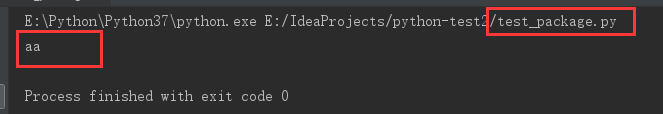
import package实际上相当于执行该package里面的\_\_init\_\_.py文件。

（import package实际上相当于执行该package里面的\_\_init\_\_.py文件）





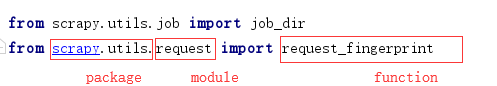




### 1.17.3 package的通常的命名规范

package通常使用的命名规范是：

1. import后面是函数名或类名，指明引入的函数，从而不引入某module的所有函数，防止重名
2. from的最后一个.后面的是module，module前面都是package。



## 1.18 线程

### 1.18.1 协程（微线程）

一个函数在执行到一半可以去执行另一个函数，而另一个函数执行到一半也可以执行别的函数。这种函数执行到一半，中断，然后去执行另一个函数，就是协程。

（注意：协程都是同一个线程在来回执行不同的函数）

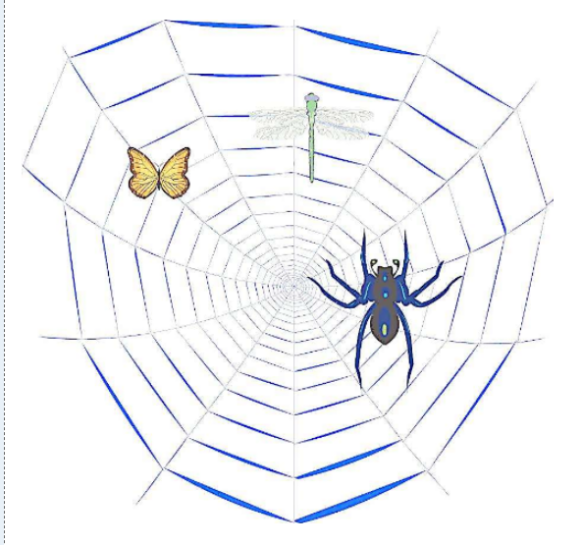
# 2 Python 爬虫

原文链接：<https://www.cnblogs.com/sss4/p/7809821.html>

如果把互联网比喻成一张蜘蛛网，爬虫就是一只小蜘蛛。

爬虫就是通过程序来模拟浏览器，去请求站点，将站点返回的HTML代码/JSON数据/二进制数据（图片、视频）爬到本地，然后提取自己需要的数据来供自己使用。

图1



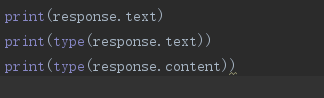
## 2.1 requests模块

### 2.1.1 response.text和response.content区别

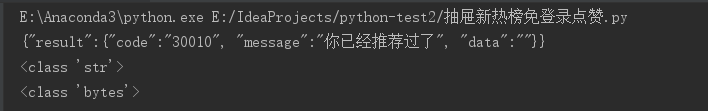
response.text是文本，response.content是二进制字节。

（注意：当接收的是图片或视频时，则使用response.content；当接收的是文本时，可以用response.text，也可以将response.content解码）

（response.text是文本，response.content是二进制字节）



（response.text是文本，response.content是二进制字节）



## 2.2 BeautifulSoup

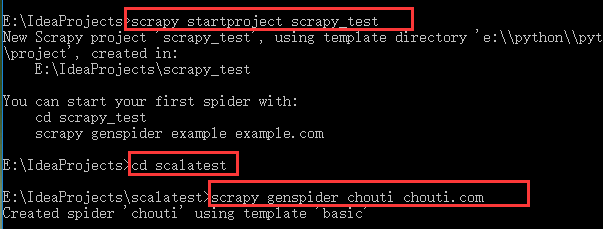
### 2.2.1 Tag

## 2.3 Scrapy框架

### 2.3.1 Scrapy使用

1.生成Spider，并指定该Spider的初始URL

（创建Spider（蜘蛛），并指定该蜘蛛的初始URL）



2.解析响应结果

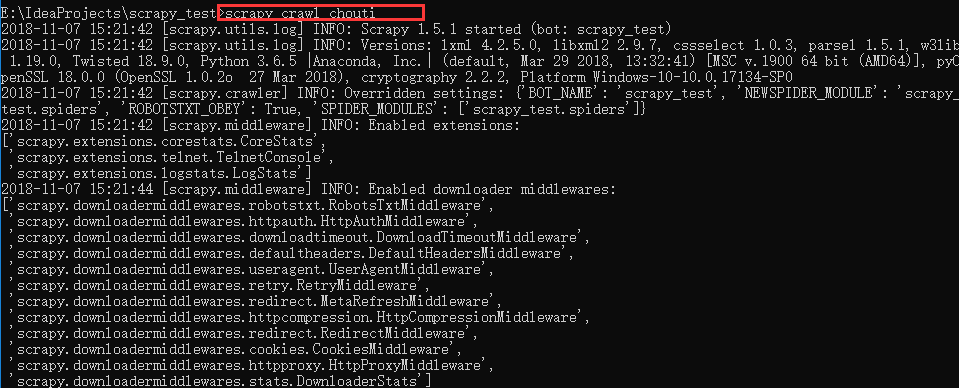
（在parse函数中解析响应结果）



3.将响应结果给item做格式化，pipeline做持久化

4.运行Spider

（使用scrapy的crawl命令运行Spider）



### 2.3.2 xpath

//表示从整个文档找子孙。

（//表示从整个文档找子孙）



/表示从整个文档找儿子。

.//或xxx//表示从当前标签找子孙。

（.//表示从当前标签找子孙）



（xxx//表示从当前标签找子孙）



./或xxx/表示从当前标签找儿子。

div[@attr=value]过滤标签，表示只寻找div标签中attr属性=value的标签。

（div[@attr=value]过滤标签，表示只寻找div标签中attr属性=value的标签）



/text()表示提取原来的带标签的Selector的文本，并形成一个新的不带标签的Selector。

（/text()表示提取原来的带标签的Selector的文本，并形成一个新的不带标签的Selector）



a/@attr获取a标签的属性attr的值。

（注意：a/@attr和a[@attr]是不一样的，a/@attr表示获取a标签的属性attr的值；a[@attr]表示获取带有attr属性的a标签）

extract()：遍历SelectorList，将每一个Selector对象转换成标签字符串，返回一个array。

（extract\_first()遍历SelectorList，取第一个Selector对象，并转换测标签字符串，返回一个string）



### 2.3.3 递归获取所有页码

为要访问的url创建一个Request对象，并为该Request对象指定回调函数。

Scrapy的scheduler会自动调度该Request对象中的url，然后通过downloader去下载url，然后调用回调函数。

（注意：创建的Request对象必须用yield返回，yield返回的Request对象会自动发送给scheduler）

（注意：可以通过DEPTH\_LIMIT限制深度，表示在当前基础上再深入访问多少层）

（注意：如果传给Request对象的回调函数不是self.parse，是一个普通的不递归的函数，则不会递归无限执行）

### 2.3.4 Pipeline

#### 2.3.4.1 Pipeline实现持久化

为每一条记录创建一个Item对象，然后通过yeild将该Item对象传递给Pipeline，由Pipeline将收到的Item对象持久化。

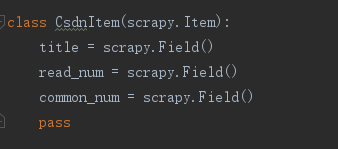
（注意：Pipeline使用前需要配置ITEM\_PIPELINES来注册）

（注意：如果注册了多个Pipeline，则多个Pipeline都会执行，执行的先后顺序由ITEM\_PIPELINES中配置的权重决定，值越小，权重越高，越先执行）

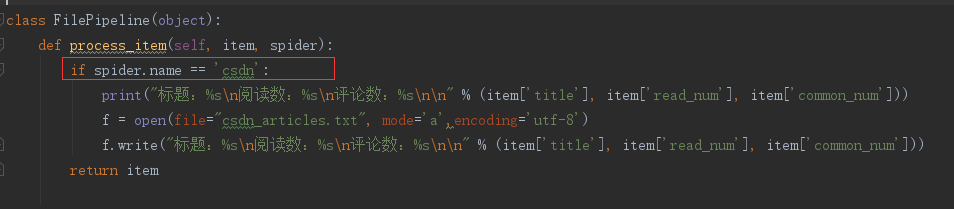
（注意：如果注册了多个Pipeline，则process\_item()函数必须return item才会交给下一个Pipeline类）

1. 在items.py中定义Item类
2. 在pipelines.py中定义Pipeline类，并在settings.py中并注册Pipeline类
3. spider类中创建一个Item类，并通过yield返回该Item类

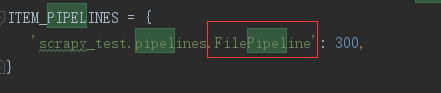
（在items.py中定义Item类）



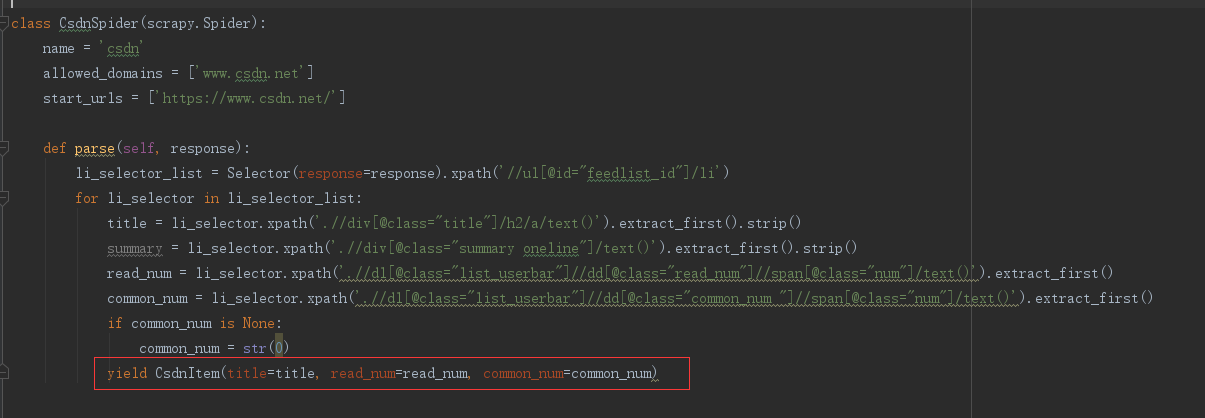
（在pipelines.py中定义Pipeline类）



（在settings.py中注册该Pipeline类）



（spider类中创建一个Item类，并通过yield返回该Item类）

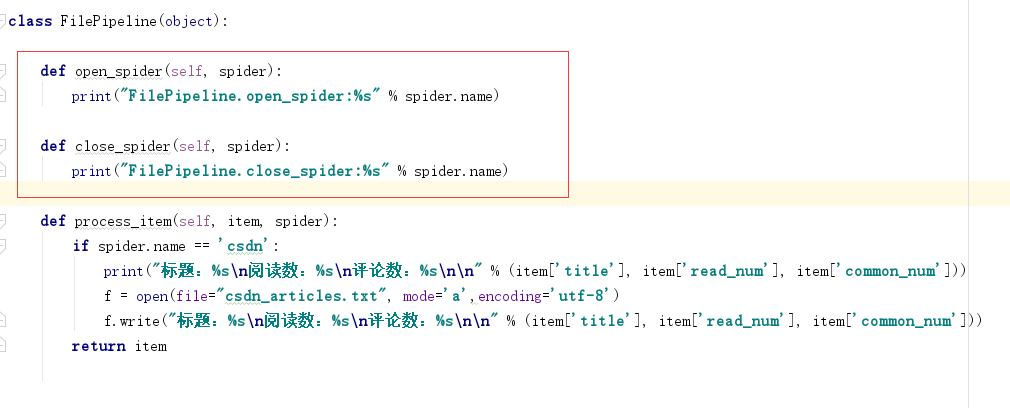


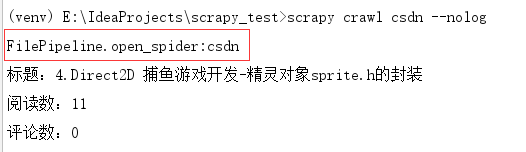
#### 2.3.4.2 Pipeline.open\_spider()和close\_spider()

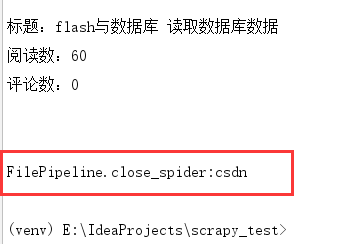
Pipeline类除了process\_item()函数来处理item，还有2个方法open\_spider()和close\_spider()方法，当spider启动和关闭时分别调用。

（注意：像文件打开和文件关闭就可以分别写在open\_spider()和close\_spider()方法中，以免每次process\_item()处理一个item都打开和关闭文件一次）

（Pipeline类还有open\_spider()和close\_spider()方法，当spider启动和关闭时分别调用）





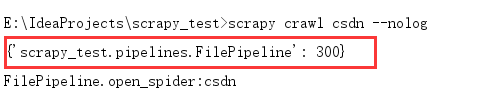
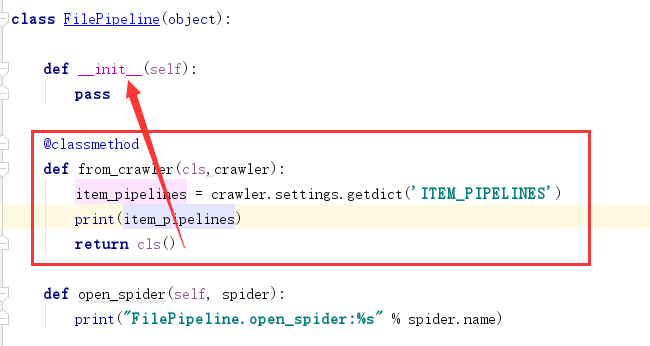


#### 2.3.4.2 Pipeline.from\_crawler()

Pipeline类有一个from\_crawler()函数，from\_crawler()函数中有一个crawler参数，crawler.settings可以拿到settings.py中的配置内容。

（注意：settings.py的key必须大写，才能通过crawler.settings读取到）

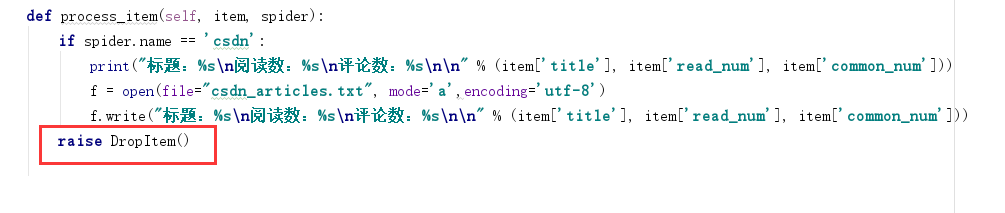
（Pipeline类的from\_crawler()函数中有一个crawler参数，crawler.settings可以拿到settings.py中的配置内容）



#### 2.3.4.3 raise DropItem()

当本Pipeline的process\_item()函数处理完Item后，不想将该Item交给下一个Pipeline类，可以不return item，或者raise DropItem()。

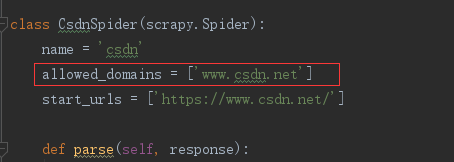
（不return item，或者raise DropItem()，从而不将该Item交给下一个Pipeline类）



### 2.3.5 allowed\_domains

第一次访问url，和后续递归访问url时，都会先去allowed\_domains中检测本url是否在allowed\_domains所允许的域名内，如果不匹配，则不会访问。

（第一次访问url，和后续递归访问url时，都会先去allowed\_domains中检测本url是否在allowed\_domains所允许的域名内，如果不匹配，则不会访问）



### 2.3.6 url去重

如果要使用Scrapy自带的url去重，默认情况下，会调用scrapy.dupefilters的request\_seen函数来去重。可以自定义一个filter类，重写request\_seen函数来代替。

（注意：使用自定义filter类前，需要在settings.py中通过DUPEFITER\_CLASS配置该类）

（注意：当访问量很大时，这个记录重复url的集合如果放内存会导致OOM，可以通过这个自定义filter类将集合保存在Redis中）

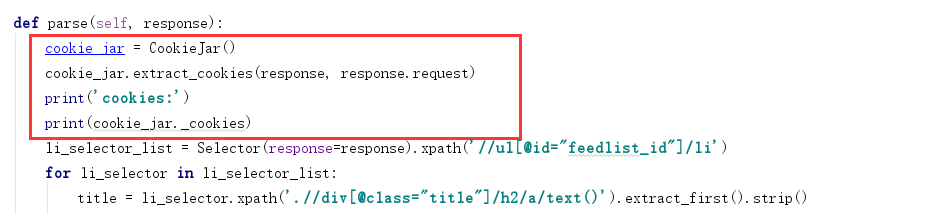
### 2.3.7 Cookie

#### 2.3.7.1 获取响应的Cookie

创建一个CookieJar类，然后调用CookieJar的extract\_cookies(response, request)函数将response和该response的request的提取到cookie\_jar对象中。

然后通过cookie\_jar对象的\_cookies获取这个对象中的cookies。

（调用CookieJar的extract\_cookies(response, request)函数将response和该response的request的提取到cookie\_jar对象）



#### 2.3.7.2 请求中加入Cookie

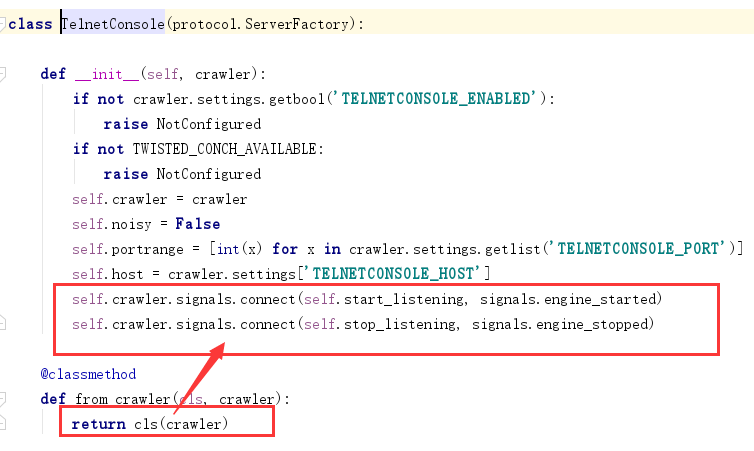
在Request对象创建时指定cookies属性。

### 2.3.7 Scrapy框架拓展（在钩子位置添加自定义函数）

可以参考settings.py中的EXTENSIONS的TelnetConsole类，from\_crawler()函数会被执行，然后通过crawler.signals.connect()函数在钩子位置添加自定义函数。

（from\_crawler()函数会被执行，然后通过crawler.signals.connect()函数在钩子位置添加自定义函数）

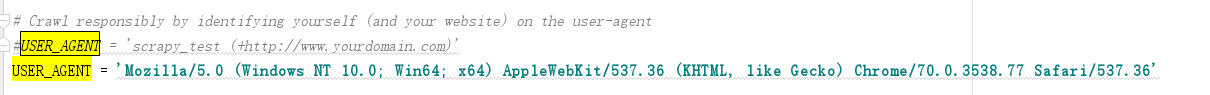




### 2.3.8 配置文件

#### 2.3.8.1 USER\_AGENT

USER\_AGENT是spider发送请求时会带上的User-Agent，可以在这里指定。



#### 2.3.8.2 ROBOTSTXT\_OBEY

ROBOTSTXT\_OBEY表示本爬虫程序是否循环别人服务器上的爬虫规则。如果为False，就算别人的规则禁止我们爬虫，我们也忽略它的规则，直接爬取。



#### 2.3.8.3 CONCURRENT\_REQUESTS

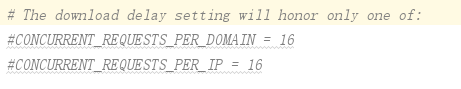
CONCURRENT\_REQUESTS：表示针对所有请求，最大的并发请求数，默认是16。



CONCURRENT\_REQUESTS\_PRE\_DOMAIN：表示针对每个域名，最大的并发请求数。

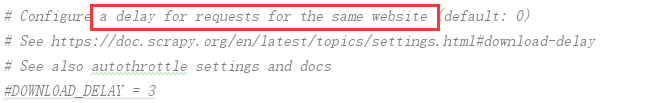
CONCURRENT\_REQUESTS\_PRE\_IP：表示针对每个IP，最大的并发请求数。

（注意：一个域名可能有多个IP，因为一般像www.baidu.com，访问量太大，会有很多台服务器去接收连接）



#### 2.3.8.4 DOWNLOAD\_DELAY

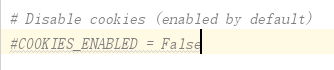
DOWNLOAD\_DELAY表示访问相同url时，2个请求之前的延迟。



#### 2.3.8.5 COOKIES\_ENABLE

COOKIES\_ENABLE表示response时，是否拿取Cookies，默认是True。

COOKIES\_DEBUG表示DEBUG模式（不加--nolog时就是DEBUG模式）时是否打印Cookies。



#### 2.3.8.6 TELNETCONSOLE\_ENABLE

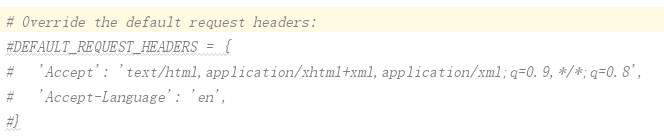
TELNETCONSOLE\_ENABLE表示是否可以在爬虫中途，通过telnet（默认是spider运行机器的6023端口）去查看爬虫状态。



#### 2.3.8.7 DEFAULT\_REQUEST\_HEADERS

DEFAULT\_REQUEST\_HEADERS表示请求时默认会带上的请求头。

（注意：如果想自定义请求头，可以在Request对象中指定）



#### 2.3.8.8 AUTOTHROTTLE\_XXX（自动限速）

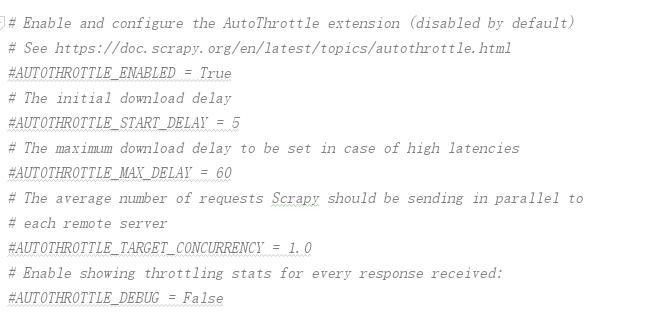
因为DOWNLOAD\_DELAY是稳定的延迟几秒，实际上用户操作不可能这么产生这么稳定的请求间隔，从而有可能会被服务器识别为爬虫。

而AUTOTHROTTLE\_XXX（自动限速）系列的参数就是智能地调节这个时间。

DOWNLOAD\_DELAY：会被认为是最小延迟

AUTOTHROTTLE\_MAX\_DELAY：最大延迟

AUTOTHROTTLE\_START\_DELAY：初始延迟



#### 2.3.8.9 DEPTH\_LIMIT

DEPTH\_LIMIT限制深度，表示在当前基础上再深入访问多少层

#### 2.3.8.10 DEPTH\_PRIORITY

DEPTH\_LIMIT：0表示深度优先，1表示广度优先，默认是0。

#### 2.3.8.11 HTTPCACHE\_XXX（HTTP缓存）

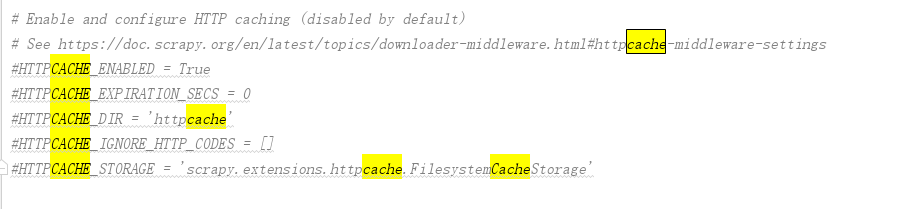
HTTPCACHE\_ENABLED：表示是否开启HTTP缓存。

HTTPCACHE\_POLICY：表示缓存策略，默认是scrapy.extensions.httpcache.DummyPolicy，所有请求都缓存。

HTTPCACHE\_DIR：缓存的数据的保存目录。

HTTPCACHE\_IGNORE\_HTTP\_CORES：表示哪些状态码进行忽略，不缓存。

HTTPCACHE\_STORAGE：缓存插件，该插件和具体缓存的介质有关，默认是缓存在文件系统中。

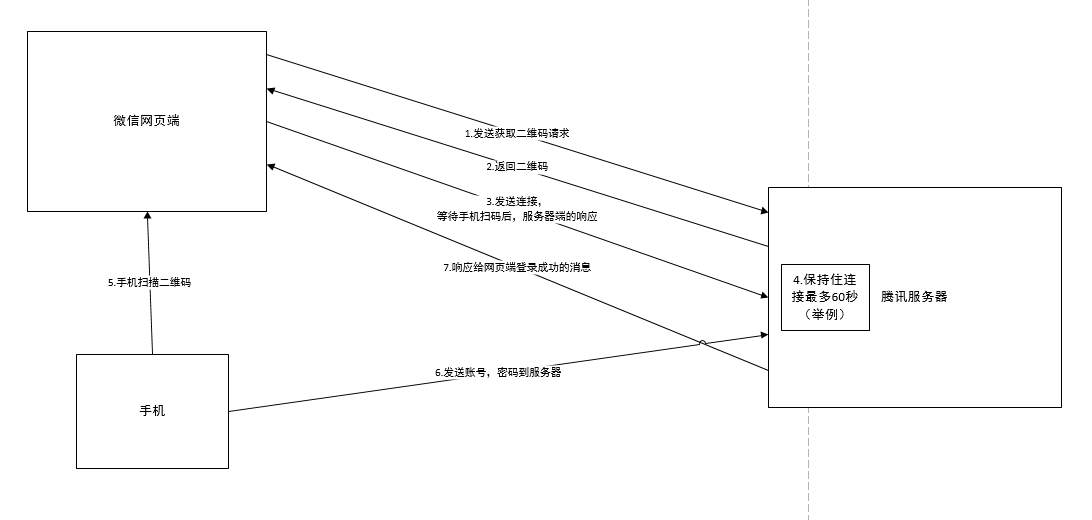


### 2.3.8 代理

## 2.4 自动登录

### 2.4.1 自动登录微信

（微信登录流程）



#### 2.4.1.1 轮询，长轮询，websocket的区别

假设A和B互相发送消息。

轮询：A每隔1秒（可设置）去服务器查询消息，查询消息后，无论有没有，都断开A和服务器的连接（A最多1秒的延迟能收到B发送的消息）

长轮序：A去服务器查询消息，服务器不返回，最多持有连接60秒（可设置）后返回，然后断开A和服务器的连接，断开后A马上又建立与服务器的连接。（A在60秒内能马上收到B发送的消息）

websocket：服务器端主动给A发消息，不需要A每次主动创建HTTP连接，节省了HTTP连接。

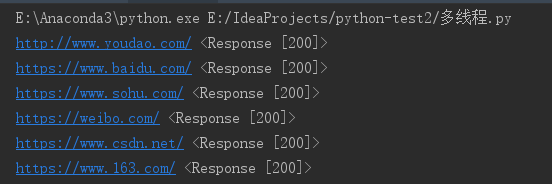
## 2.5 多线程实现并发请求

### 2.5.1 多线程并发同步

并发是指多个线程同时发送请求；同步是指每一个请求都是同步的，线程必须等待服务器的返回。

（多线程并发同步，可以看到响应结果和url列表顺序不一致，因为是同时发送的请求，但谁先返回就谁先打印结果）



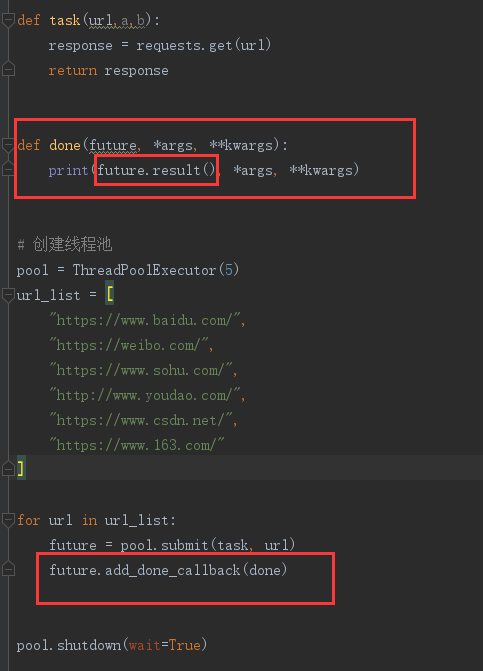


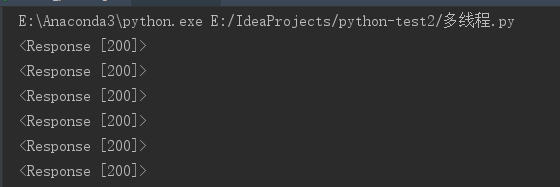
### 2.5.2 多线程并发异步

并发是指多个线程同时发送请求；异步是指每一个请求都是异步的，线程不需要等待服务器的返回，而是服务器响应成功后，调用回调函数。

（注意：回调函数可以指定多个）

（多线程并发异步，服务器响应成功后，调用回调函数done，done的第一个参数是一个Future对象，通过Future.result()可以拿到Response对象）





### 2.5.2 协程 + 异步

# 3 机器学习（machine learning）

机器学习就是开发一系列的算法，从而让机器自己通过大量历史数据去学习，然后建模。最后利用建立好的模型和新的输入来进行预测的学科。

机器学习的应用有：

机器人下棋，

语音识别，

自动驾驶汽车，

语言翻译（比如：即时的英文->中文翻译），

计算机视觉（比如：人脸识别），

推荐系统（比如：抖音的爱好推荐，淘宝的猜你喜欢商品），

无人机（比如：美团的无人送外卖，无一机拯救人类），

识别垃圾邮件等。

## 概念学习

概念学习就是：机器去学习某个概念（概念是：什么是鸟，什么是车，什么是享受运动），学习这个概念的方法是：从输入输出训练样例集中，推断出该布尔函数（c）。

（“享受运动”这个概念的输入输出训练样例）



### 实例

实例（x）就是输入输出训练样例集（X）中的一个样例（x）。

y=c（x），c就是待学习的概念或函数；x是一个样例，X是所有样例；y是函数的值，Y是所有函数的值。

（实例就是训练样例集中的一个样例）



### 训练集（traning set）和测试集（testing set）

训练集（traning set）就是用来产生函数的样例集。

测试集（testing set）就是用来测试（已经产生的函数）的样例集。

（注意：训练集和测试集是不重复的，因为用产生函数的样例去测试该函数毫无意义）

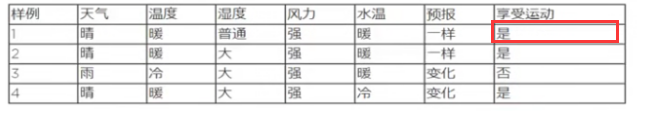
### 特征向量（freatures/freature vectors）

对于每一个实例（样例），都用一个向量来表示所有的属性的集合，这个向量就是特征向量。



### 标记（label）

对于每一个实例（样例），它的函数的值就是标记（label）。



### 正例（positive example）和反例（negative example）

正例就是c(x)==y的样例。

（正例）



反例就是c(x)!=y的样例。

（反例）



### 分类（classification）和回归（regression）

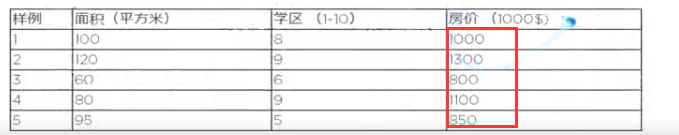
分类（classification）：标记为类别型数据（category）。

回归（regression）：标记为连续型数值（continuous numeric value）。

（分类（classification）：标记为类别型数据（category））



（回归（regression）：标记为连续型数值（continuous numeric value））



### 有监督学习，无监督学习，半监督学习

有监督学习（supervised learning）：训练集中的样例全都有类别标记（class label）。

无监督学习（unsupervised learning）：训练集中的样例全都无类别标记（class label）。

半监督学习（semi-supervised learning）：训练集中的样例部分有类别标记（class label），部分无类别标记。

比如：

特征值：肿瘤尺寸，颜色

标记：良性/恶性

有监督学习：训练集中的样例的标记都是良性或恶性。

无监督学习：训练集中的样例的标记都不知道，但是训练集是存在的。

半监督学习：训练集中的样例的标记一部分是良性或恶心，一部分标记不知道。

## 机器学习算法

### 从哪些角度评估算法好坏

1. 准确率：表示预测结果的正确率
2. 速度：算法的复杂度，决定算法的运行速度
3. 强壮性：当训练集中有些特别偏离正常值的数据时，算法的预测结果是否还能表现良好
4. 可规模性：小数据量正常时，当数据量指数型增长时，算法是否还能表现hian正常
5. 可解释性：算法是否能结果具体业务场景进行解释

### 3.2.2 决策树（Decision Tree）

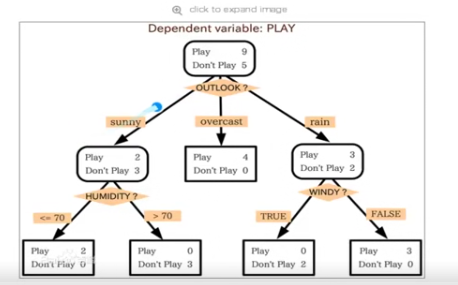
决策树是一个类似于流程图的树结构。

每个非树叶节点表示一个属性上的测试（比如：OUTLOOK属性是多少？，HUMIDITY>=70?）；

每个分支表示一个属性的值的输出（比如：OUTLOOK=sunny或overcast或rain）；

每个树叶节点表示满足以上所有属性测试的样例集，这个样例集中的样例的标记值都是相同或满足以上相同属性条件的。

（注意：决策树就像是嵌套多层的if else对样例集进行归类，将原始数据集划分成多个数据子集）



#### 3.2.2.1 熵（entropy）

信息熵表示信息内容的不确定性，内容（样例）不确定性越多，信息熵越高。熵的单位是比特（bit）。

（注意：信息熵与内容的数据量大小无关）

（注意：n表示分类的数目，xi表示某一个分类，p(xi)表示某一个分类出现的概率=该分类的样例数÷样例总数）

（注意：xi表示一个分类，而不是一个样例）

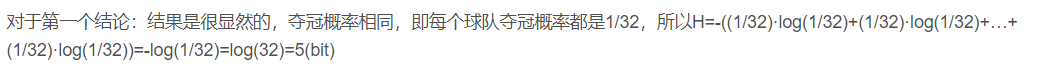
信息熵的计算公式：





在世界杯比赛中，如果有32支球队，假设每只球队的夺冠几率都相同，则H(X)=5(bit)

（在世界杯比赛中，如果有32支球队，假设每只球队的夺冠几率都相同，则H(X)=5(bit)）



#### 3.2.2.2 决策树归纳算法（ID3算法）

信息获取量（信息增益）的算法：



Info(D)：表示没有属性A进行分类时（即按当前的标记进行分类）的熵。

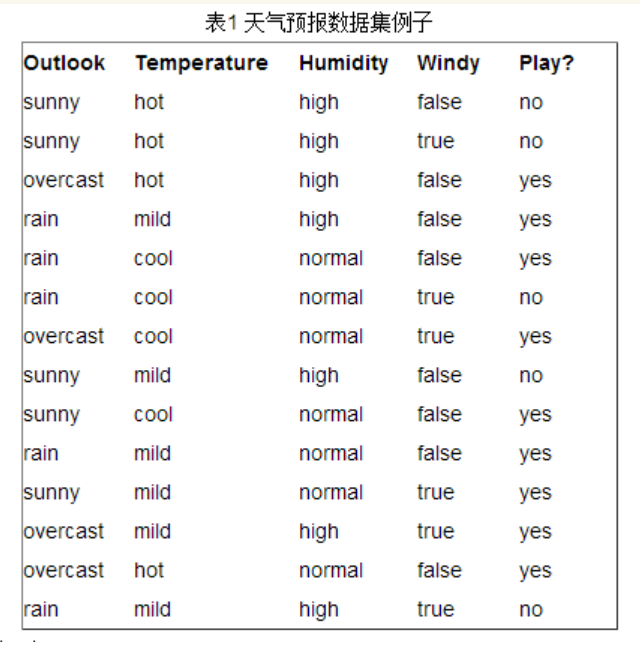
InfoA(D)：表示有属性A进行分类时的（各个分支熵\*分支出现的概率）的和。

Gain(A)：表示属性A的信息增益。Gain(A) = Info(D) - InfoA(D)。

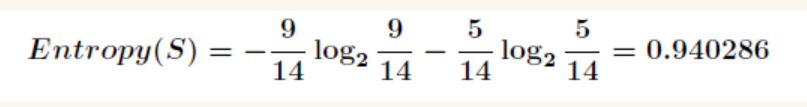
ID3算法思想：属性A的信息增益越大，该属性区分样本的能力越强，越具有代表性，应该优先选择信息增益最大的属性来划分。

（注意：D表示当前样例集）

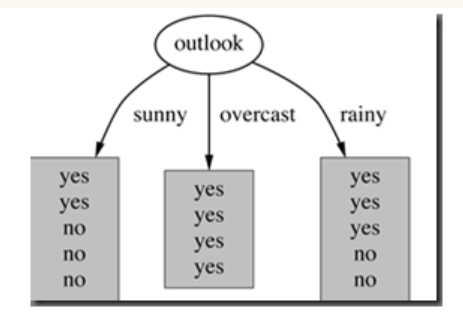
举例：

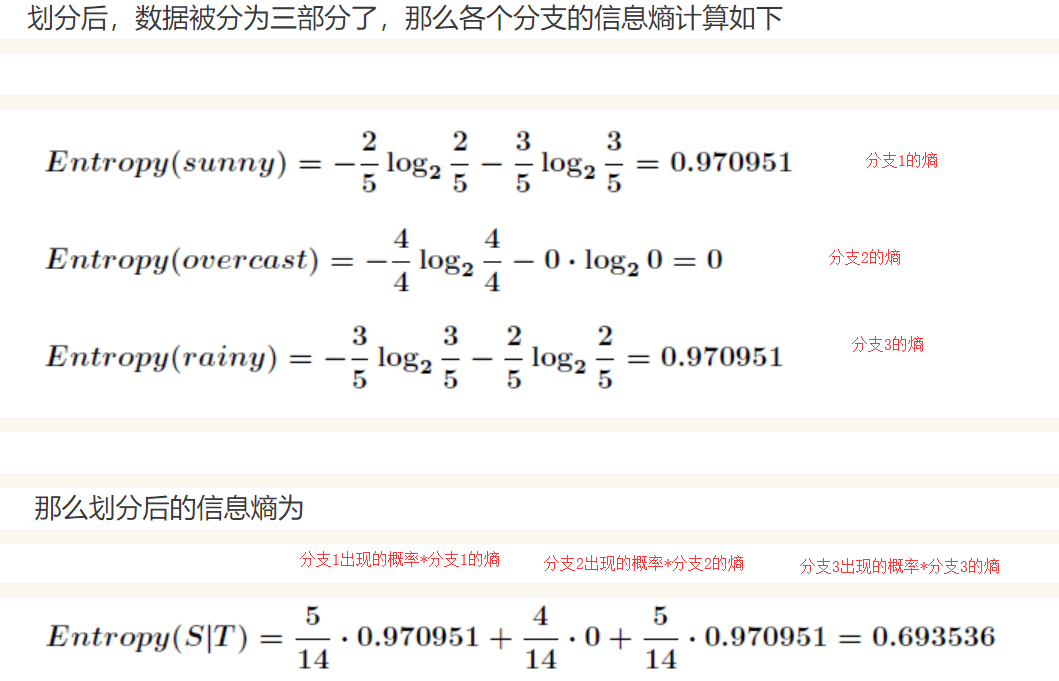


Info(D)：表示没有属性A进行分类时（即按当前的标记进行分类）的熵。就等于：

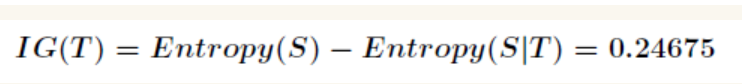


InfoA（D）：表示有属性A进行分类时的（各个分支熵\*分支出现的概率）的和，就等于：





Gain(A)：表示属性A的信息增益。Gain(A) = Info(D) - InfoA(D)，就等于：



##### 3.2.2.2.1 ID3算法的Python应用

#### 3.2.2.3 连续值必须离散化

对于标记的值是连续的，比如：年龄，不能1,2,3,4....100每一个数字为1个分支，这样分支就无穷无尽了。必须对连续的标记值进行离散化（分类化），最后的分支是：<18,18~30,30~100。

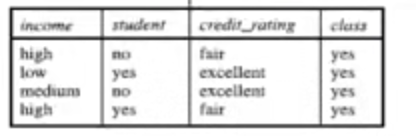
#### 3.2.2.4 已分类过的属性不再考虑

如果某个属性已经进行分类过，则后续的分类不能再考虑该属性。

#### 3.2.2.5 决策树何时停止

1.当某个分支的结果的标记都是同类或同值时，不需要再继续分。

（比如：某个分支的结果的标记都是yes）



2.当前样例集合已经找不到新的属性进行分类（即所有属性已经用完）。

3.所有样例集合都已经被处理完，没有任何未被处理的样例集合时停止。

#### 3.2.2.6 决策树的优点和缺点

优点：

1. 直观，便于理解
2. 对于小规模数据有效，因为小规模数据的分支少，从而最终出来的决策树不深，却很有效。

缺点：

1. 处理连续变量不好，因为离散化的范围对结果影响很大
2. 可规模性一般，因为小规模数据出来的决策树不一定适用于大规模的数据。
3. 当类别越多时，出错越快。

# 4 深度学习（deep learning）