**一、模块化思想**

1. 标准化模块格式：

**标准模块：.py**

#!/usr/bin/env python3

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

' a test module '

\_\_author\_\_ = 'Michael Liao'

import sys

def test():

args = sys.argv

if len(args)==1:

print('Hello, world!')

elif len(args)==2:

print('Hello, %s!' % args[1])

else:

print('Too many arguments!')

if \_\_name\_\_=='\_\_main\_\_':

test()

当我们在命令行运行hello模块文件时，Python解释器把一个特殊变量\_\_name\_\_置为\_\_main\_\_，而如果在其他地方导入该hello模块时，if判断将失败，因此，这种if测试可以让一个模块通过命令行运行时执行一些额外的代码，最常见的就是**运行测试**。

2.封装

def \_private\_1(name):

return 'Hello, %s' % name

def \_private\_2(name):

return 'Hi, %s' % name

def greeting(name):

if len(name) > 3:

return \_private\_1(name)

else:

return \_private\_2(name)

思想：

1. 类似\_\_xxx\_\_这样的变量是特殊变量，
2. 类似\_xxx和\_\_xxx这样的函数或变量就是非公开的（private），不应该被直接引用，比如\_abc，\_\_abc等
3. 在模块里公开greeting()函数，而把内部逻辑用private函数隐藏起来了，这样，调用greeting()函数不用关心内部的private函数细节，**这也是一种非常有用的代码封装和抽象的方法，**即：
4. 外部不需要引用的函数全部定义成private，只有外部需要引用的函数才定义为public。

3.面向的对象开发

1. 通过内置的一系列函数，我们可以对任意一个Python对象进行剖析，拿到其内部的数据。要注意的是，只有在不知道对象信息的时候，我们才会去获取对象信息。如果可以**直接写：**sum = obj.x + obj.y

**不用写**：sum = getattr(obj, 'x') + getattr(obj, 'y')

**正确是使用代码如下：**

def readImage(fp):

if hasattr(fp, 'read'):

return readData(fp)

return None

从文件流fp中读取图像，首先判断该fp对象是否存在read方法，如果存在，则该对象是一个流，如果不存在，则无法读取。hasattr()就派上了用场。

注意：在Python这类动态语言中，根据鸭子类型，有read()方法，不代表该fp对象就是一个文件流，它也可能是网络流，也可能是内存中的一个字节流，但只要read()方法返回的是有效的图像数据，就不影响读取图像的功能。

1. 不要对实例属性和类属性使用相同的名字，否则将产生难以发现的错误。

>>> class Student(object):

... name = 'Student'

...

>>> s = Student() # 创建实例s

>>> print(s.name) # 打印name属性，因为实例并没有name属性，所以会继续查找class的name属性

Student

>>> print(Student.name) # 打印类的name属性

Student

>>> s.name = 'Michael' # 给实例绑定name属性

>>> print(s.name) # 由于实例属性优先级比类属性高，因此，它会屏蔽掉类的name属性

Michael

>>> print(Student.name) # 但是类属性并未消失，用Student.name仍然可以访问

Student

>>> del s.name # 如果删除实例的name属性

>>> print(s.name) # 再次调用s.name，由于实例的name属性没有找到，类的name属性就显示出来了

Student

1. **装饰器的使用**

（1）@property装饰器

负责把一个方法变成属性调用。

@property本身又创建了另一个装饰器@score.setter，负责把一个setter方法变成属性赋值，由此，得到可控的属性操作.

示例代码：

**class** Student(object):  
 **def** \_\_init\_\_(self,name):  
 self.name = name

*#使用装饰器* @property  
 **def** score(self):  
 **return** print(**"（装饰器返回值）%s's score is %d"** % (self.name,self.\_score))  
 @score.setter  
 **def** score(self,valve):  
 self.\_score = valve  
 print(**"（装饰器返回值）%s's score = %d"** % (self.name,**self.\_score**))

*#测试my\_classe*Vicky = src.my\_class.Student(**'Vicky'**)  
*#使用装饰器*Vicky.score = 100  
Vicky.score

**结果：**



此处必须是私有属性

（2）@unique装饰器可以帮助我们检查保证没有重复值。

（3）@classmethod装饰器：用于定义类方法，告诉解释器下面的方法为类方法，**只需要访问类属性**

**示例代码：**

class Tool:

**#定义类属性**

**count = 0**

**#定义类方法**

**@classmethod**

**def show\_tool\_count(cls):**

**print("当前的工具数量: %d" % cls.count)**

**#调用类方法**

**Tool.show\_tool\_count()**

1. 使用多继承

在设计类的继承关系时，主线都是单一继承下来的，例如，Ostrich继承自Bird。但是，如果需要“混入”额外的功能，通过**多重继承**实现，比如，让Ostrich除了继承自Bird外，再同时继承Runnable。这种设计称为MixIn。

MixIn目的：给一个类增加多个功能，在设计类时，优先考虑通过多重继承来组合多个MixIn的功能，而不是设计多层次的复杂的继承关系。（python自带的库也使用了Mixln）

**总结**：需要什么功能，就混入哪种父类。

1. 定制类

（1）\_\_iter\_\_():

如果一个类想被用于for ... in循环，类似list或tuple那样，就必须实现一个\_\_iter\_\_()方法，该方法返回一个迭代对象，然后，Python的for循环就会不断调用该迭代对象的\_\_next\_\_()方法拿到循环的下一个值，直到遇到StopIteration错误时退出循环

示例代码：

class Fib(object):

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

self.a, self.b = 0, 1 # 初始化两个计数器a，b

def \_\_iter\_\_(self):

return self # 实例本身就是迭代对象，故返回自己

def \_\_next\_\_(self):

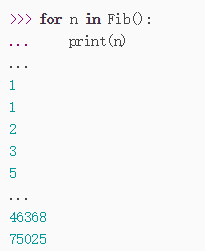
self.a, self.b = self.b, self.a + self.b # 计算下一个值

if self.a > 100000: # 退出循环的条件

raise StopIteration()

return self.a # 返回下一个值

结果：



（2）\_\_getitem\_\_()

功能：像list那样按照下标取出元素，需要实现\_\_getitem\_\_()方法

示例代码：

class Fib(object):

def \_\_getitem\_\_(self, n):

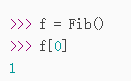
a, b = 1, 1

for x in range(n):

a, b = b, a + b

return a

结果：



（3）\_\_getattr\_\_()

**功能：**当调用不存在的属性时，比如score，Python解释器会试图调用\_\_getattr\_\_(self, 'score')来尝试获得属性，由此就会返回score的值。

**注意：**只有在没有找到属性的情况下，才调用\_\_getattr\_\_

（4）\_\_call\_\_()

功能：要定义一个\_\_call\_\_()方法，就可以直接对实例进行调用

示例代码：

class Student(object):

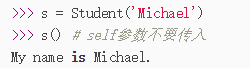
def \_\_init\_\_(self, name):

self.name = name

def \_\_call\_\_(self):

print('My name is %s.' % self.name)

结果：



（5）\_\_str\_\_()

功能：提供便捷的打印服务（打印的格式好看）

示例代码：

class Student(object):

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

self.name = name

def \_\_str\_\_(self):

return 'Student object (name: %s)' % self.name

结果：



1. 异常、错误捕获

Python内置的logging模块可以非常容易地记录错误信息：

**同样是出错，但程序打印完错误信息后会继续执行，并正常退出**：

（logging还可以把错误记录到日志文件里，方便事后排查。）

**示例代码：**

import logging

def foo(s):

return 10 / int(s)

def bar(s):

return foo(s) \* 2

def main():

try:

bar('0')

except Exception as e:

logging.exception(e)

main()

print('END')

**执行：**$ python3 err\_logging.py

ERROR:root:division by zero

Traceback (most recent call last):

File "err\_logging.py", line 13, in main

bar('0')

File "err\_logging.py", line 9, in bar

return foo(s) \* 2

File "err\_logging.py", line 6, in foo

return 10 / int(s)

ZeroDivisionError: division by zero

END

1. 断言assert

用print()来辅助查看的地方，以用断言（assert）来替代：

**assert n != 0, 'n is zero!'**：表达式n != 0应该是True，否则，后面的代码肯定会出错。

如果断言失败，assert语句本身就会抛出AssertionError：

**示例代码**：

def foo(s):

n = int(s)

assert n != 0, 'n is zero!'

return 10 / n

def main():

foo('0')

**执行：**

$ python err.py

Traceback (most recent call last):

...

AssertionError: n is zero!

启动Python解释器时可以用-O参数来关闭assert，也就是：assert相当于pass

命令：python -0 文件名

1. 读取文件
2. **def** load\_data(filename):  
    *"""  
    打开文件函数* **:param** *filename: 输入文件路径* **:return***: 返回array，对应的数据矩阵  
    """* data = []  
    file = open(filename)  
    **for** line **in** file.readlines():  
    lineArr = line.strip().split(**','**)  
    col\_num = len(lineArr)  
    temp = []  
    **for** i **in** range(col\_num):  
    temp.append(float(lineArr[i]))  
    data.append(temp)  
    **return** np.array(data)
3. 保存数据（txt）

以'w'模式写入文件时，如果文件已存在，会直接覆盖（相当于删掉后新写入一个文件）。如果我们希望追加到文件末尾怎么办？可以传入'a'以追加（append）模式写入。

**def** saveFile(fliename,data): *"""  
 保存数据函数* **:param** *fliename:* **:param** *data:  
 """*

*#用十进制形式保存数据* np.set\_printoptions(suppress=**True**)  
 file = open(fliename, **'w'**)  
 rowNum, columnNum = data.shape  
 **for** line **in** range(rowNum):  
 *#实现自动换行* file.write(str(data[line,0:2])+**'\n'**)  
 file.close()

1. os模块

操作文件

*# 查看当前目录的绝对路径:*

>>> os.path.abspath('.')

'/Users/michael'

*# 在某个目录下创建一个新目录，首先把新目录的完整路径表示出来:*

>>> os.path.join('/Users/michael', 'testdir')

'/Users/michael/testdir'

*# 然后创建一个目录:*

>>> os.mkdir('/Users/michael/testdir')

*# 删掉一个目录:*

>>> os.rmdir('/Users/michael/testdir')

*# 对文件重命名:*

>>> os.rename('test.txt', 'test.py')

*# 删掉文件:*

>>> os.remove('test.py')

1. **序列化：**

**序列化：**变量从内存中变成可存储或传输的过程，Python提供了pickle模块来实现序列化。

序列化之后，就可以把序列化后的内容写入磁盘，或者通过网络传输到别的机器上。

反过来，把变量内容从序列化的对象重新读到内存里称之为反序列化，即unpickling。

代码：

>>> **import** pickle

>>> d = dict(name='Bob', age=20, score=88)

#序列化

>>> pickle.dumps(d)

b'\x80\x03}q\x00(X\x03\x00\x00\x00ageq\x01K\x14X\x05\x00\x00\x00scoreq\x02KXX\x04\x00\x00\x00nameq\x03X\x03\x00\x00\x00Bobq\x04u.'

#保存到文件

>>> f = open('dump.txt', 'wb')

>>> pickle.dump(d, f)

>>> f.close()

#读取文件

>>> f = open('dump.txt', 'rb')

>>> d = pickle.load(f)

>>> f.close()

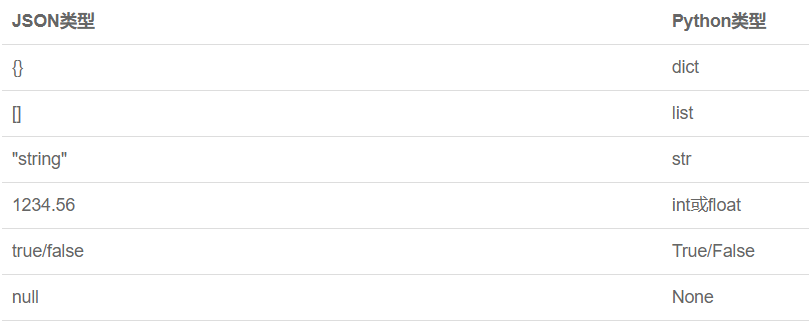
>>> d

{'age': 20, 'score': 88, 'name': 'Bob'}

1. **JSON：**

将要保存的数据序列化为JSON，**JSON表示的是一个字符串**，可以被所有语言读取，可直接再Web中读取，速度快。**JSON是标准的格式。**

JSON标准规定JSON编码是UTF-8。



代码：

>>> **import** json

#序列化为JSON

>>> d = dict(name='Bob', age=20, score=88)

>>> json.dumps(d)

'{"age": 20, "score": 88, "name": "Bob"}'

#反序列化

>>> json\_str = '{"age": 20, "score": 88, "name": "Bob"}'

>>> json.loads(json\_str)

{'age': 20, 'score': 88, 'name': 'Bob'}

1. JSON进阶，将类、对象序列化

**import** json

**def** **student2dict**(std):

**return** {

'name': std.name,

'age': std.age,

'score': std.score

}

**def** **dict2student**(d):

**return** Student(d['name'], d['age'], d['score'])

**class Student(object):**

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

self.name = name

self.age = age

self.score = score

s = Student('Bob', 20, 88)

#序列化

>>> **print**(json.dumps(s, **default**=student2dict))

{"age": 20, "name": "Bob", "score": 88}

#反序列化

>>> json\_str = '{"age": 20, "score": 88, "name": "Bob"}'

>>> print(json.loads(json\_str, object\_hook=dict2student))

<\_\_main\_\_.Student object at 0x10cd3c190>

1. **进程、线程：**

**进程：**对于操作系统而言，一个任务就等同于一个进程（比如打开一个word）

**线程**：在一个进程内部的子任务，一个进程至少包含一个线程。

**python实现多任务的方式：**

1）多进程模式；

2）多线程模式；

3）多进程+多线程模式。

=====================================================

**eg：（由于windows上没有fork()，所以无法运行该代码）**

import os

print('Process (%s) start...' % os.getpid())

# Only works on Unix/Linux/Mac:

pid = os.fork()

if pid == 0:

print('I am child process (%s) and my parent is %s.' % (os.getpid(), os.getppid()))

else:

print('I (%s) just created a child process (%s).' % (os.getpid(), pid))

**结果：**

Process (876) start...

I (876) just created a child process (877).

I am child process (877) and my parent is 876.

=============================================================

**join（）：**等待子进程结束后再继续往下运行，通常用于进程间的同步。

**multiprocessing：**模块封装了fork()调用，提供一个Process类来代表一个进程对象。

**eg:在windows中使用多线程：**

from **multiprocessing** import Process  
import os  
# 子进程要执行的代码  
def run\_proc(name):  
 print('Run child process %s (%s)...' % (name, os.getpid()))  
if \_\_name\_\_=='\_\_main\_\_':  
 print('Parent process %s.' % os.getpid())  
 p = Process(target=run\_proc, args=('test',))  
 print('Child process will start.')  
 p.start()  
 p.join()  
 print('Child process end.')

**结果：**

Parent process 28300.

Child process will start.

Run child process test (20708)...

Child process end.