Python Note

Basic

1.两种运行方式，cmd中直接输入python进入命令行模式；在进入.PY文件目录下 在CMD中用python XXX.py运行即可。

2.python的靠缩进来区分模块，业内约定是四个空格，在某句代　码后加上：即认为下方缩进部分为内部代码模块。

例:

```
a = 100
if a >= 0:
    print(a)
else:
    print(-a)
```

3.#号用来注释语句，为单行注释

4.用单引号和双引号括起来的都是文本，但是当单引号也是文本本身的时候要用双引号括起来。

例："I'm OK"

　在遇到多个符号的时候我们要使用转义符 \来。

例：‘I\'m OK’ 其输出和上方是一样的。

　当我们遇到\n（换行），\t(制表符)的时候，多个\会很迷惑人，所以我们使用r''来表达，会默认在r''中的字符串不再转意

例:print(r'\\\t\\\')所输出的就是\\\t\\\

　print('\\\t\\\')所输出的就是\　　\

　CMD中一个句子内部有多个\n换行符的时候阅读很困难，所以可以使用'''...'''来实现多行，sublime下在打出''' '''后可以直接回车换行。

5.语言中的True 和 False 区分大小写。

6.空值使用none来表示。

7.PI为圆周率常量，除法//和MOD相同，求整。除法%,取余。

8.中文在UTF-8编译后会占据3个字节，英文1个

b''表示为该字符串为bytes类型。

字符串后用.encode()来转换 例：　'陈'.encode('ascii')　输出为b'\xe9\x99\x88'

字符串在bytes格式下可以使用.decode()来转换成相应字符串　例：　b'\xe9\x99\x88'.decode('utf-8')　输出为 '陈'

同样可以使用 len('')来确定字符数 例如len(b'abc') = 3,len('中文'.encode('utf-8')) = 6

9.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

在python 的开头写上这个头部，第一行表示在Lunix环境下为PYTHON可执行文件，第二行表示使用utf-8字符集。文本编辑器要使用utf-8 without BOM

10. python的语言格式话方式与C语言相同。 使用%实现。使用%? 作为占位符来表示 %d 整数 %f浮点数 %s字符串

例: 'hello ,%s you have $%d' % ('chan',10000) 输出 'hello chan, you have $10000'

使用%2d 或者 %02d 这两者都规定了是有最少两位空间数，第一个 '%2d' % 2 输出 ' 2'，第二个'%02d' % 2 输出 '02'

11.list 和 tuple ,

(1)list类似于数组，正序下标从0开始，倒序从-1开始。len()函数可以判定list长度。使用.append()可以将内容直接插入list尾部，使用.insert(nubmer,content)可以直接内容插入固定位置。删除尾部元素的时候可以使用.pop()方法。和数组一样，可以使用下标的方式直接将list内容达到替换。list内部是可以包含不同类型的内容，同时list内的元素可以是list。

例:classmate = ['jack','alan'] ,len(classmate) =2 ,classmate.append('niko') --> ['jack','alan','niko'].

(2)tuple和LIST的使用方式是类似和相同的，主要是它是静态的，是不可变的。同时，tuple在只有一个元素的时候，申明方式比较特殊，要尤其记得。我们可以通过让tuple来加入一个list的方式，来达到实现对tuple内容内的'修改'。实质上是修改了LIST的内容，而tuple本身是没有修改的。

例：空tuple t = t(), 含有一个元素的tuple t=(1,) 要含有一个逗号来消除 (1)是数学运算还是tuple的歧义。

12.if...else...和if...elif...else的判断，注意：要在IF后加上:号来标记，用缩进来表示内部代码。 同时，input是一个STR格式，需要用INT()函数来做一个类型转换达到判定。

例:

```
i = input('i = ')
a = int(i)
if 0< a <3:
        print ('Ok!')
```

13.循环，格式如下:

使用if的循环:

```
sum = 0
for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    sum = sum + x
print(sum)
```

使用while的循环:

```
sum = 0
n = 99
while n > 0:
    sum = sum + n
    n = n - 2
print(sum)
```

(1)rang(5)，生成0-4 5个数字，使用list[range(5)]，生成[0，1，2，3，4]

14.break语句

添加IF和BREAK之前，是有1打印到100，加入后，在N=11时，break生效，跳出循环，打印出END。

```
n = 1
while n <= 100:
    if n > 10: # 当n = 11时，条件满足，执行break语句
        break # break语句会结束当前循环
    print(n)
    n = n + 1
print('END')
```

15.continue语句

添加continue之前，是有1打印到100，加入后，在偶数情况下讲不被输出，继续下一个循环，所以输出内容是1-100的奇数。

```
n = 0
while n < 10:
    n = n + 1
    if n % 2 == 0: # 如果n是偶数，执行continue语句
        continue # continue语句会直接继续下一轮循环，后续的print()语句不会执行
    print(n)
```

16.dict字典：类似于其他语言中的map。

和list比较，dict有以下几个特点：

1. 查找和插入的速度极快，不会随着key的增加而变慢；
2. 需要占用大量的内存，内存浪费多。

与list相反：

1. 查找和插入的时间随着元素的增加而增加；
2. 占用空间小，浪费内存很少。

```
>>>d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}
>>>d['Michael']
>>> 95
```

17.set 与dict类似，只包含KEY字段，同时，是无序不重复排列的，且可以做数学上的交，并集计算的。

```
>>> s1 = set([1, 2, 3])
>>> s2 = set([2, 3, 4])
>>> s1 & s2
{2, 3}
>>> s1 | s2
{1, 2, 3, 4}
```

18.str是一个不可变变量的。

Function

19.定义函数： 可以在函数判断后加入pass来实现空函数，让程序往下走。

```
def my_abs(x):
    if x >= 0:
        return x
    else:
        return -x
```

参数检测

```
def my_abs(x):
    if not isinstance(x, (int, float)):
        raise TypeError('bad operand type')
    if x >= 0:
        return x
    else:
        return -x
```

导入已有函数包

```
import math
```

python的多返回值是一个truple

20.函数参数

(1)位置参数：例如f(x) 其中的x就是位置参数，使用函数的时候一定要输入参数才可以。

(2)默认参数：例如f(x,n =2),在缺省不对n赋值的时候，默认=2 。这时候的f(1)即为f(x=1,n =2)，默认参数必须指向不变对象。

(3)可变参数：使用list或者truple来传入参数计算，参数的个数是不固定的。

```
def calc(numbers): #def calc(*numbers)：使用这种就可以来将list和tuple
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

使用的时候要先自己拼起来一个list或者tuple。

而当我们已有一个list或者tuple的时候，使用这种方式进行传递很麻烦。

```
>>> nums = [1, 2, 3]
>>> calc(nums[0], nums[1], nums[2])
```

用*number直接将list或者tuple直接传入当做参数使用。 来简化。

```
>>> nums = [1, 2, 3]
>>> calc(*nums)
```

(4)关键字参数：可变参数将变量转换成list或tuple，关键字参数在函数内测组成一个dict

```
def person(name, age, **kw):  #**kw 为关键字参数
    print('name:', name, 'age:', age, 'other:', kw)

>>> person('Michael', 30)
>>> name: Michael age: 30 other: {}

>>> person('Bob', 35, city='Beijing')
>>> name: Bob age: 35 other: {'city': 'Beijing'}
```

(5)命名关键字参数，需要使用*来分隔开

```
def person(name, age, *, city, job):
    print(name, age, city, job)
```

当前面有一个可变参数的时候就不需要了

```
def person(name, age, *args, city, job):
    print(name, age, args, city, job)
```

命名关键字参数必须传入参数名

(6)参数是有组合使用的情况的，所以使用的时候参照顺序**【必选参数、默认参数、可变参数、命名关键字参数和关键字参数】**

21.递归函数，使用递归函数要注意栈堆溢出

```
def fact(n):
    if n==1:
        return 1
    return n * fact(n - 1)
```

解决溢出的办法是使用尾递归，在有针对尾递归有优化的时候栈的大小是不变化的。

```
def fact(n):
    return fact_iter(n, 1)

def fact_iter(num, product):
    if num == 1:
        return product
    return fact_iter(num - 1, num * product)
```

和上面的函数对比的话，这个递归只产生函数本身。

Advanced

22.切片：有一个list ,L[] ，使用L[:5]取前五个元素，L[-5:]取后五个元素，L[10:20]取下标10-20的元素。L[::5]每5个取一个元素，L[:10:2]前10个元素，每2个取一个。

23.迭代：使用 for X in OBJ 只要OBJ是可以迭代的对象，那么即可迭代。使用函数 Iterable来检测是否可以迭代。

```
>>> from collections import Iterable
>>> isinstance('abc', Iterable) # str是否可迭代
True
```

24.列表生成式：直接在list中来嵌套循环等操作来是实现

```
>>> [m + n for m in 'ABC' for n in 'XYZ']
['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']
```

25.生成器 generator 将列表生成式的[]换成()就是一个生成器

```
列生成式↓
>>> L = [x * x for x in range(10)]
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
生成器↓
>>> g = (x * x for x in range(10))
>>> g
<generator object <genexpr> at 0x1022ef630>
```

列表生成式是可以逐行打印出来的，但是生成器不全部，一个个显示只能使用next()来获得，同样可以使用FOR 来迭代循环使用。

```
>>> next(g)
0
```

普通的方式输出斐波拉契数列用

```
# -*- coding: utf-8 -*-
def fib(max):
        n,a,b = 0,0,1
        while n<max:
                print(b) # print(b)改成yield b;这个函数就是一个generator
                a,b=b,a+b
                n = n+1
        return 'done'

fib(5)
```

但是用for循环调用generator时，发现拿不到generator的return语句的返回值。如果想要拿到返回值，必须捕获StopIteration错误，返回值包含在StopIteration的value中。后续再讲。

```
 while True:
```

```
...     try:
...         x = next(g)
...         print('g:', x)
...     except StopIteration as e:
...         print('Generator return value:', e.value)
...         break
```

26.迭代器

使用isinstance 来检查对象是不是 Iterable，

可以被for迭代调用的对象，成为可迭代对象：Iterable

```
>>> from collections import Iterable
>>> isinstance([], Iterable)
True
```

可以被next()函数调用并不断返回下一个值的对象称为迭代器：Iterator 同样可以使用insatance()函数来检测。

将list、dict、str 等不是迭代器的东西可以使用iter()函数转换。

```
>>> isinstance(iter([]), Iterator)
True
>>> isinstance(iter('abc'), Iterator)
True
```

Function Porgamming

27.高阶函数 函数名也是一种变量,将函数名指定为变量后，默认函数就失去原来的功能。

```
>>> abs = 10
>>> abs(-10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

但是可以将函数赋值给变量来获得功能。

```
>>> f = abs
>>> f(-10)
10
```

这样我们通过这种方式可以获得高级函数编程

```
def add(x, y, f):
    return f(x) + f(y)

>>> add(-5, 6, abs)
11
```

28.map/reduce

map()是函数接受两个参数，一个是函数，一个是Iterable,map将传入的函数依次做用于序列的每一个元素，并把结果作

为一个Iterator返回。

ex:两个的结果是一样的，但是使用

```
def f(x):
    return x * x
#for循环
L = []
for n in [1, 2, 3, 4, 5, 6, 7, 8, 9]:
    L.append(f(n))
print(L)
>>>[1, 4, 9, 16, 25, 36, 49, 64, 81]
#使用MAP
>>> r = map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> list(r)
>>>[1, 4, 9, 16, 25, 36, 49, 64, 81]
#简单的就可以发现一行来代替
list(map(f,[1, 2, 3, 4, 5, 6, 7, 8, 9]))
```

reduce把一个函数作用在一个序列[x1, x2, x3, ...]上，这个函数必须接收两个参数，reduce把结果继续和序列的下一个元素做累积计算。

ex:求和

```
>>> from functools import reduce
>>> def fn(x, y):
...     return x * 10 + y
...
>>> reduce(fn, [1, 3, 5, 7, 9])
13579
```

29.filter

filter()是函数接受两个参数，一个是函数，一个是序列。与map相似，同样是将函数一个个作用于序列内的对象，但是filter是返回Ture和False来决定元素是保留还是去除。

```
def is_odd(n):
    return n % 2 == 1

list(filter(is_odd, [1, 2, 4, 5, 6, 9, 10, 15]))
```

30.sorted

sorted类似于Map，将函数一个个作用于序列中的对象，然后将对象按照一定的顺序进行排列。

```
# -*- coding: utf-8 -*-
L = [('Bob', 75), ('Adam', 92), ('Bart', 66), ('Lisa', 88)]

def by_name(t):
        return t[1]

L2 = sorted(L, key=by_name,reverse=True)#recerse 是反序排列的意思。
print(L2)
```

31.返回函数：

通过对变量赋值为函数的时候，来延迟对函数的计算，当使用的时候再真正的计算来获得好的性能。

```
def lazy_sum(*args):
    def sum():
        ax = 0
        for n in args:
            ax = ax + n
        return ax
    return sum
```

这时候我们所返回的不是所以的和，而是带有要计算元素的求和函数。

```
>>> f = lazy_sum(1, 3, 5, 7, 9)
>>> f
<function lazy_sum.<locals>.sum at 0x101c6ed90>
```

在调用的时候，并没有计算，而是在使用过f的时候才真正的返回了和。

这里要描述的是当我们连续两次调用的时候，即使传入的是相同的参数，返回的都是一个新的函数。

```
>>> f1 = lazy_sum(1, 3, 5, 7, 9)
>>> f2 = lazy_sum(1, 3, 5, 7, 9)
>>> f1==f2
False
```

32.闭包

不要在任何返回函数中使用循环变量，不然在重复调用中，返回函数都会绑定循环变量当前的值，所返回的数据将不正确。

```
def count():
    fs = []
    for i in range(1, 4):
        def f():
             return i*i
        fs.append(f)
    return fs

f1, f2, f3 = count()
>>> f1()
9
>>> f2()
9
>>> f3()
9
```

```
def count():
    def f(j):
        def g():
            return j*j
        return g
    fs = []
    for i in range(1, 4):
        fs.append(f(i)) # f(i)立刻被执行，因此i的当前值被传入f()
    return fs
>>> f1, f2, f3 = count()
>>> f1()
```

```
1
>>> f2()
4
>>> f3()
9
```

33.匿名函数lambda

限制：只有一个表达式，没有return。

```
>>> f = lambda x: x * x
>>> f
<function <lambda> at 0x101c6ef28>
>>> f(5)
25
```

34.装饰器 Decorator

在不改变函数定义的情况下，在代码运行期间增加功能的方式称之为装饰器，实际上就是返回函数的高阶函数。

```
def log(text):
    def decorator(func):
        def wrapper(*args, **kw):
            print('%s %s():' % (text, func.__name__))
            return func(*args, **kw)
        return wrapper
    return decorator
```

```
@log
def now():
    print('2015-3-25')
```

```
>>> now()
execute now():
2015-3-25
```

```
now = log('execute')(now)
```

实际的执行方式可以视为上面那句，先执行log(execute),然后执行decorator(new),最终返回warapper函数。在这里的时候decorator已经变成了wrapper了。我们需要把now的__name__属性复制到wrapper中。但是不能使用wrapper.__name__= now.__name__这种。使用内置的functools.wraps。使用一下格式来书写。

```
import functools

def log(text):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kw):
            print('%s %s():' % (text, func.__name__))
            return func(*args, **kw)
        return wrapper
    return decorator
```

35.偏函数

使用functools.partial来固定一些函数的参数达到快速生成函数的目的。

```
def int2(x, base=2):
    return int(x, base)
```

```
int2 = functools.partial(int, base=2)
```

Module

36.模块

1.主要讲解怎么将PY分包组织，每个包都必须有__init.py__.py，才能被认可为包，其中__init__.py可以是空文件也可以是有代码的。

2.使用模块 import **

3.作用域：变量名是公开的就可以直接引用（EX: abc）,特殊变量是可以直接访问，但是有特殊用途。（EX:__author__）,私有变量是不能被直接访问的，但是是可以访问的，python没有完全不可访问的韩硕或者变量。但是在实际使用中，我们会对此有些控制。所以用函数嵌套的方式来实现。

```
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)
```

37.第三方模块

1.使用pip包管理工具来实现。

```
pip install Pillow #Pillow是一个图片处理模块。
```

2.模块搜索路径

import 包的时候可能会找不到报错，我们要调整Python包的路径。

1)修改sys.path

2)和环境变量一样设置pyhtonpath

Object Oriented Programming

38.类和实例

```
class Student(object):
```

```
    def __init__(self, name, score):  #构造Student类
        self.name = name
        self.score = score

    def print_score(self):            #构造类中的一个方法
        print('%s: %s' % (self.name, self.score))
```

39.访问限制

class内部有属性和方法，外部代码可以直接调用。是可以自由修改的。

```
>>> bart = Student('Bart Simpson', 98)
>>> bart.score
98
>>> bart.score = 59
>>> bart.score
59
```

在变量前面添加两个下划线__就可以把变量变成私有变量，不能够外部访问。

```
class Student(object):

    def __init__(self, name, score):
        self.__name = name
        self.__score = score

    def print_score(self):
        print('%s: %s' % (self.__name, self.__score))
```

```
>>> bart = Student('Bart Simpson', 98)
>>> bart.__name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute '__name'
```

我们要获得这些变量的值或者对这些值做设定的时候只能通过使用增加方法的方式实现了：因为方法是可以自己编写定制的，所以在使用的时候还可以通过方法来检验函数输入值的正确与否。

```
class Student(object):
    ...
    def get_name(self):   #定义获得数据name
        return self.__name

    def get_score(self):  #定义获得数据score的方法
        return self.__score

    def set_score(self, score): #定义设定score的方法
        self.__score = score

    def set_score(self, score): #扩充定定义score的方法，包含函数参数检验
        if 0 <= score <= 100:
            self.__score = score
        else:
            raise ValueError('bad score')
```

是不是一定不能外部访问这些私有变量呢？不是，是Python编译器将它编译为了__Student__name，所以我们通过__Student__name来直接访问，但是不推荐这么做。

40.继承和多态

定义一个class的时候，可以从某个已经有的class来集成，新的class就成为子集（Subclass）,被继承的class被成为基类、父类、超类(Base class ,Super class)。

我们使用一个类来继承的时候是可以自动集成获得父类的所有方法，如果自己在子类中定义同名方法的时候，会直接调用子类的方法。这种子类用定义的方式是多态的一种。

```
class Animal(object):
    def run(self):
        print('Animal is running...')
```

```
class Dog(Animal):
    pass


>>>dog = Dog()
>>>dog.run()

Animal is running...
```

```
class Dog(Animal):

    def run(self):
        print('Dog is running...')

    def eat(self):
        print('Eating meat...')

>>>dog = Dog()
>>>dog.run()

dog is running...
```

类是和系统自带的数据类型没有区别的。同时，子类不是但数据自己子类的类型，同时也是父类的类型。

```
a = list() # a是list类型
b = Animal() # b是Animal类型
c = Dog() # c是Dog类型

>>> isinstance(a, list)
True
>>> isinstance(b, Animal)
True
>>> isinstance(c, Dog)
True
>>> isinstance(c, Animal)
True
```

多态的另一种体现是当我们定一个基于父类的函数的时候，我们在传入子类参数的时候也是能够运行的。我们在新增新类的时候不需要去修改参数，获得便捷的扩充。

```
def run_twice(animal):
    animal.run()
```

```
    animal.run()

>>> run_twice(Animal())
Animal is running...
Animal is running...

>>> run_twice(Dog())
Dog is running...
Dog is running...
```

由运行时该对象的确切类型决定，这就是多态真正的威力：

这就是著名的“开闭”原则：1.对扩展开放：允许新增子类；2.对修改封闭：不需要修改依赖类型的等函数。

41.静态预言和动态语言

对于静态语言（例如Java）来说，如果需要传入Animal类型，则传入的对象必须是Animal类型或者它的子类，否则，将无法调用run()方法。

对于Python这样的动态语言来说，则不一定需要传入Animal类型。我们只需要保证传入的对象有一个run()方法就可以了：

```
class Timer(object):
    def run(self):
        print('Start...')
```

这就是动态语言的“鸭子类型”，它并不要求严格的继承体系，一个对象只要“看起来像鸭子，走起路来像鸭子”，那它就可以被看做是鸭子。

42.获取对象信息

我们怎么知道一个对象什么类型，有什么方法呢?

1)使用type(),简单的对象类型我们可以直接写，例如：INT等，判断是否函数等，我们使用type()模块的定义常量来判断。(详阅文档)

```
>>> type(123)
<class 'int'>

>>> type(abs)
<class 'builtin_function_or_method'>
```

2)使用isinstance(),对于calss的继承关系来说。使用type()不是很方便。使用上面的Animal类的例子，我们的继承关系是：

```
object -> Animal -> Dog -> Husky
```

```
>>> a = Animal()
>>> d = Dog()
>>> h = Husky()

>>> isinstance(h, Husky)
True
>>> isinstance(h, Dog)
True
>>> isinstance(d, Husky)
False
```

isinstance()也可以来判定基本类型

```
>>> isinstance('a', str)
True
>>> isinstance([1, 2, 3], (list, tuple))
True
```

3)使用dir(),使用后会返回一个包含字符串的list,包含所要求对象的多有属性和方法

```
>>> dir('ABC')
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__g
```

43.实例属性和类属性

在给实例绑定属性的方式是通过实例变量，或者SELF变量实现的。

```
class Student(object):
    def __init__(self, name):
        self.name = name

s = Student('Bob')
s.score = 90
```

但是同时候，我们可以直接在类中定义一个属性。叫做类属性。

```
class Student(object):
    name = 'Student'
```

但是，当我们定义一个类属性后，再从类中定义一个实例时候。实例仍然是可以访问到的。只是，在属性名字相同的时候，实例属性会覆盖类属性。

```
实例属性和类属性

阅读: 134860
由于Python是动态语言，根据类创建的实例可以任意绑定属性。

给实例绑定属性的方法是通过实例变量，或者通过self变量:

class Student(object):
    def __init__(self, name):
        self.name = name

s = Student('Bob')
s.score = 90
但是，如果Student类本身需要绑定一个属性呢？可以直接在class中定义属性，这种属性是类属性，归Student类所有:

class Student(object):
    name = 'Student'
当我们定义了一个类属性后，这个属性虽然归类所有，但类的所有实例都可以访问到。来测试一下:

>>> 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
```

Advanced OOP

44.__slots__

我们创建一个类的实例后，我们可以给该实例绑定任何属性和方法。

但是，我们在给实例绑定方法后，并不能在其他新的同类实例中使用。

```
class Student(object):
    pass

>>> s = Student()
>>> s.name = 'Michael' # 动态给实例绑定一个属性
>>> print(s.name)
Michael

>>> def set_age(self, age): # 定义一个函数作为实例方法
...     self.age = age
...
>>> from types import MethodType
>>> s.set_age = MethodType(set_age, s) # 给实例绑定一个方法
>>> s.set_age(25) # 调用实例方法
>>> s.age # 测试结果
25
```

```
>>> s2 = Student() # 创建新的实例
>>> s2.set_age(25) # 尝试调用方法
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'set_age'
```

所以我们通过给类绑定新的方法来实现大家通用的方式。通常我们是把这些方法直接写在class中的，但是动态语言让我们实现了动态绑定。

```
>>> def set_score(self, score):
...     self.score = score
...
>>> Student.set_score = set_score
```

```
>>> s = Student()
>>> s.set_score(100)
>>> s.score
100
```

如此一来我们要限制对类的方法的动态绑定，所以使用__slots__来做限制。我们只能绑定在__slots__中限定的方法。

```
class Student(object):
    __slots__ = ('name', 'age') # 用tuple定义允许绑定的属性名称
```

```
>>> s = Student() # 创建新的实例
>>> s.name = 'Michael' # 绑定属性'name'
>>> s.age = 25 # 绑定属性'age'
>>> s.score = 99 # 绑定属性'score'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'score'
```

在使用是时候要注意，__slots__只对当前类有效果。对其的子类是无效的。

```
>>> class GraduateStudent(Student):
...     pass
...
>>> g = GraduateStudent()
>>> g.score = 9999
```

但是，在写好了__slots__的时候，我们可以使用添加方法的方式来添加属性。

```
>>> class Student(object):
...     __slots__ = ('a')

>>>def set_b(selt,b):
...     self.b = b

>>>s = Student()
>>>s.a = 1
>>>s.b = 2
AttributeError: 'Student' object has no attribute 'b'#说明没有b属性，也不能添加

>>>from types import MethodType
>>>Student.set_b = MethodType(set_b,Student)

>>>s.set_b(2)
>>>s.b
2
```

还有一种情况，当我们使用多个实例来更新类属性的时候，值会是最后一个

```
>>> class stu(object): #声明一个类
...  pass
...
>>> def set_age(self,age):#给类绑定一个方法的方式来设置一个age属性
...  self.age =age
...
>>> a = stu() #实例a
>>> b = stu() #实例b

>>> from types import MethodType  #将方法set_age绑定在class上
>>> stu.set_age = MethodType(set_age,stu)
>>> stu.age
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'stu' has no attribute 'age'  #只有方法没有调用一次是没有age属性的
```

```
>>> a.set_age(20)  #用实例a调用方法，修改了age属性
>>> a.age
20
>>> stu.age  #这时候的类属性是20
20

>>> b.set_age(25) #用实例b调用方法，修改了age属性
>>> b.age
25
>>> a.age
25
>>> stu.age
25            #这个时候，我们从上面可以看出set_age()是调用方法修改的是class的age属性。而不是a,b两个实例的属性

>>> a.age=10  #我们这样的赋值才是赋予了实例a 自己的同名age属性
>>> b.age=15
>>> a.age
10
>>> b.age
15
>>> stu.age
25           #所以在此我们输出的结果不再一样，class的属性没有被修改。证明了实例属性和类属性的区别。
```

45. @property

我们再绑定属性后，虽然对属性操作起来很简单方便，但是这样会把属性暴露，也不能够对值进行检测，所以我们通过方法来实现。

```
s = Student()
s.score = 9999 #这里就是我们将score属性暴露在外，且没有什么控制
```

我们来定义set_score()和get_score()方法来做读取和值检测，这样就不能随意的对score来制作。

```
class Student(object):

    def get_score(self):
         return self._score

    def set_score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~ 100!')
        self._score = value
```

```
>>> s = Student()
>>> s.set_score(60) # ok!
>>> s.get_score()
60
>>> s.set_score(9999)
Traceback (most recent call last):
  ...
ValueError: score must between 0 ~ 100!
```

但是这种方式，我们使用的时候不能狗直接使用要不停的调用，不是很方便。所有用@property来实现。

```
class Student(object):

    @property
    def score(self):
        return self._score

    @score.setter
    def score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~ 100!')
        self._score = value
```

```
>>> s = Student()
>>> s.score = 60 # OK，实际转化为s.set_score(60)
>>> s.score # OK，实际转化为s.get_score()
60
>>> s.score = 9999
Traceback (most recent call last):
  ...
ValueError: score must between 0 ~ 100!
```

当我们只定义getter()不定义setter()时，就是一个只读属性。下面的例子就是 age就是一个制度变量。

```
class Student(object):

    @property
    def birth(self):
        return self._birth

    @birth.setter
    def birth(self, value):
        self._birth = value

    @property
    def age(self):
        return 2015 - self._birth
```

46.多重继承

在我们使用各种类的继承的时候，我们用动物为例，在界门纲目科属种等很多层级后，我们用多重继承来快速的生成要的类型

```
class Animal(object):    #1 高级
    pass

# 大类:
class Mammal(Animal):#哺乳类   2中级
    pass

class Bird(Animal):#鸟类   2中级
    pass

class Dog(Mammal):#声明是哺乳类的狗  3 低级
    pass
```

```
class Runnable(object) ： # 是可以跑的动物类  2中级
    def run(self):
        print('Running...')

class Dog(Mammal, Runnablev):  # 这样我们就完善了dog是一个可以跑的哺乳动物。
    pass
```

MixIn是一个命名方式，目的是给类增加多个功能。这样设计的时候我们类就考虑主要的线性集成，用MixIn组合多个功能。Python自己有很多库使用MixIn，例如TCPServer和UDPServer这两类网络服务。我们要为多个用户服务的时候，就需要一个多线程来提供。这样就可以获得一个多线程的TCP服务。

```
class MyTCPServer(TCPServer, ForkingMixIn):
    pass
```

47.定制类

__str__ 在使用一个类的时候，我们会遇到返回的是一个地址，如果可以返回我们想要的结果就好了。

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...
>>> print(Student('Michael'))
<__main__.Student object at 0x109afb190>
```

我们通过定制函数__str__来实现。

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...     def __str__(self):
...         return 'Student object (name: %s)' % self.name
...
>>> print(Student('Michael'))
Student object (name: Michael)
```

但是我们直接调用变量的时候还是变量的内存名。是因为直接调用用的不是__str__而是__repr__，两者的区别是__str__是给用户使用的，__repr__是给开发者使用的。我们同样可以定义。但如果相同的话，我们可以直接覆盖函数就可以了。

```
class Student(object):
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return 'Student object (name=%s)' % self.name
    __repr__ = __str__
```

__iter__当我们创建一个类，想要他可以被for...in循环的时候，我们要实现一个__iter__方法，然后for循环会不断的调用迭代对象的__next__()方法，直到遇到StopIteration错误的时候退出循环。

```
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 > 100: # 退出循环的条件
            raise StopIteration()
        return self.a # 返回下一个值
```

```
class Fib(object):
    def __getitem__(self, n):
        a, b = 1, 1
        for x in range(n):
            a, b = b, a + b
        return a
```

__getitem__我们在实现了for循环后，但是不能使用下标来取数，这时候我们用__getitem__()来实现.

```
class Fib(object):
    def __getitem__(self, n):
        a, b = 1, 1
        for x in range(n):
            a, b = b, a + b
        return a
```

同样，list还有一个切片功能，这时候我们去使用的时候构造的Fib类的时候却是报错的。因为__getitem__传入的是一个int而切片是一个slice所以我们要对__getitem__增加判断升级。

```
class Fib(object):
    def __getitem__(self, n):
        if isinstance(n, int): # n是索引
            a, b = 1, 1
            for x in range(n):
                a, b = b, a + b
            return a
        if isinstance(n, slice): # n是切片
            start = n.start
            stop = n.stop
            if start is None:
                start = 0
            a, b = 1, 1
            L = []
            for x in range(stop):
                if x >= start:
                    L.append(a)
                a, b = b, a + b
            return L
```

此时，我们还没有完成切片中的step功能，所以要晚上一个__getitem__需要很多的工作。如果我们将对象看成一个dict,__getitem__函数的参数可能是一个可以作为KEY的object，例如str。与此对应的还有设定和赋予值的__setitem__和删除值的__delitem__。

__getattr__

我们在调用类内没有的属性的时候，一般会直接报错，为了避免这个错误，我们出了可以加上一个属性，也可以使用__getattr__()来动态返回一个属性或者是一个函数。

```
class Student(object):

    def __init__(self):
        self.name = 'Michael'
```

```
    def __getattr__(self, attr): #动态返回变量
        if attr=='score':
            return 99

    def __getattr__(self, attr): #动态返回函数
        if attr=='age':
            return lambda: 25
```

```
>>> s = Student()
>>> s.name
'Michael'
>>> s.score
99
>>> s.age()
25
```

在实际的使用过程中，只有没有的属性，我们才去__getattr__中去寻找，但是做处理，我们在随意调用其他的属性是，返回的都是NONE，所有我们要按照约定，将__getattr__中找不到的返回原来的错误信息。

```
>>>s.abc
None
```

```
class Student(object):

    def __getattr__(self, attr):
        if attr=='age':
            return lambda: 25
        raise AttributeError('\'Student\' object has no attribute \'%s\'' % attr)
```

如此，我们可以实现把所有的属性和方法调用全部动态化。这有什么用呢？我们可以用这个方法来解决调用各类API接口。写SDK的时候面临只要有改动，就要去修改一次。使用__getattr__我们可以做一个链式的动态调用。

```
class Chain(object):
    """docstring for Chain"""
    def __init__(self, path=''):
            self._path = path

    def __getattr__(self,path):
            return Chain('%s/%s' % (self._path,path))

    def __str__(self):
            return self._path

    __repr__ = __str__

print(Chain().status.html.homepage.list)
```

还有一些是包含参数的动态调用，例如在路径中要添加用户名。像GitHub的我们要把:user变量替换实际用户名

```
GET /users/:user/repos
```

__call__

实例是可以拥有自己的属性和方法的，我们使用的方式是instance.method()来调用的，我们可以用__call__来实现直接在实例本身上调用。

```
class Student(object):
    def __init__(self, name):
        self.name = name

    def __call__(self):
        print('My name is %s.' % self.name)

>>> s = Student('Michael')
>>> s() # self参数不要传入
My name is Michael.
```

__call__()还可以定义参数的，对实例直接调用就好像是对函数直接调用，这样我们就模糊了对象和函数的界限，所以我们使用callable()来判定这个对象是否可以被调用。

```
>>> callable(Student())
True
>>> callable([1, 2, 3])
False
```

48.枚举类

我们需要定义常量的时候，一个一个写很简单，但是声明的时候仍然是一个变量。Python中有和好的方法。定义个一枚举类型，每个常量都是class的唯一实例。在这里我们用Enum来实现

```
from enum import Enum

Month = Enum('Month', ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'))
```

这样我们就获得了Month类型的枚举，可以直接调用Month.Jan来调用一个常量或者枚举他的所有成员

```
for name, member in Month.__members__.items():
    print(name, '=>', member, ',', member.value)
```

结果是如下,member.value是自动赋予的常量，默认从1开始。

```
Jan => Month.Jan , 1
Feb => Month.Feb , 2
Mar => Month.Mar , 3
Apr => Month.Apr , 4
May => Month.May , 5
Jun => Month.Jun , 6
Jul => Month.Jul , 7
Aug => Month.Aug , 8
Sep => Month.Sep , 9
Oct => Month.Oct , 10
Nov => Month.Nov , 11
Dec => Month.Dec , 12
```

如果我们要自己更精确的控制，我们要从Enum上派生出自定义类

```
from enum import Enum, unique

@unique
class Weekday(Enum):
    Sun = 0 # Sun的value被设定为0
```

```
    Mon = 1
    Tue = 2
    Wed = 3
    Thu = 4
    Fri = 5
    Sat = 6

print(Weekday.Tue)
print(Weekday.Sat.value)
print(Weekday(1))

for name,user in Weekday.__members__.items():
        print(name,'->',user)
```

```
Weekday.Tue
6
Weekday.mon
sun -> Weekday.sun
mon -> Weekday.mon
Tue -> Weekday.Tue
Wed -> Weekday.Wed
Thu -> Weekday.Thu
Fri -> Weekday.Fri
Sat -> Weekday.Sat
```

49.元类

type()

type()是一个类型或者变量的类型，Hello是一个class，它的类型就是type，而h是一个实例，它的类型是Hello。

```
class Hello(object):
    def hello(self, name='world'):
        print('Hello, %s.' % name)

>>> from hello import Hello
>>> h = Hello()
>>> h.hello()
Hello, world.
>>> print(type(Hello))
<class 'type'>
>>> print(type(h))
<class 'hello.Hello'>
```

class的定义是可以动态创建的，而实现动态创建的方法就是所以type()函数。type()即可以返回一个对象的类型，也可以创建出新的类型。用下面的方式可以直接动态生成Hello类型。

```
>>> def fn(self, name='world'): # 先定义函数
...     print('Hello, %s.' % name)
...
>>> Hello = type('Hello', (object,), dict(hello=fn)) # 创建Hello class
```

创建一个新的类型，我们一要传入三个参数。

1)class的名称 对应 'Hello'

2)集成的父类集合，因为Python支持多重继承，如果只有一个父类，集成组的是元组，要是用tuple的单元素写法。对应 '(object,)'

3)class的方法名和函数绑定。这里我们把函数fn绑定在Hello上。对应 'dict(hello=fn)'

metaclass

metaclass，直译为元类,我们可以使用metaclass来创建class,然后在实例化使用。

正常情况我们都是定义class ,创建实例。而元类是定义metaclass,然后创建类，然后创建实例。我们在定义metaclass的时候，都用metaclass作为尾坠来表明。

```
class ListMetaclass(type):
    def __new__(cls, name, bases, attrs):
        attrs['add'] = lambda self, value: self.append(value)
        return type.__new__(cls, name, bases, attrs)
```

```
class MyList(list, metaclass=ListMetaclass):
    pass
```

我们在metaclass传入关键字的时候，解释器通过ListMetaclass.__new__()创建了list。其传入参数的要求是:

1)创建类的对象

2) 类的名字

3)类集成的父类集合

4)类的方法集合

Test and Catch

50.错误处理

我们有时候会定义一些特殊的返回值来表示错误，但是在实际过程中，我们会和其他的返回值结果混淆。所以在JAVA中我们有一套try...except...finally...的方式。python同样也有。

try。try是在我们认为该段落会出错的时候，我们用try执行，若出错我们就进入except块，执行完毕后如果有finally块，就执行。至此一个流程完毕。

```
try:
    print('try...')
    r = 10 / 0
    print('result:', r)
except ZeroDivisionError as e:
    print('except:', e)
finally:
    print('finally...')
print('END')
```

这个代码会在10/0产生一个错误，所以后续语句没有被执行，expect捕捉到了ZeroDivisionError错误后被执行了，然后执行了finally。最后程序按照接下来的流程走动。如果我们把0换成2

```
try...
result: 5
finally...
END
```

错误是没有发生的，所以ecpect没有被执行，试试finally如果有就一定会执行。当错误有很多种的时候我们就可以多个

ecpect处理不同的错误。如果没有错误发生，我们可以在except后面加入一个else，没有错误的时候会自动执行，有else的时候finally也会被执行。

```
try:
    print('try...')
    r = 10 / int('2')
    print('result:', r)
except ValueError as e:
    print('ValueError:', e)
except ZeroDivisionError as e:
    print('ZeroDivisionError:', e)
else:
    print('no error!')
finally:
    print('finally...')
print('END')
```

Python的所有错误都是从类BaseException类中派生出来的。我们使用的时候，except在捕捉父类后，其子类也被捕捉。在except中写含有父子关系的错误类，子类是永远捕捉不到的。例如:UnicodeError就是ValueError的子类就不会被捕捉。

```
try:
    foo()
except ValueError as e:
    print('ValueError')
except UnicodeError as e:
    print('UnicodeError')
```

https://docs.python.org/3/library/exceptions.html#exception-hierarchy是Pyhton的错误类集成关系文档。

Python在使try...except...的时候，是可以多层调用的，如下面的例子：我们在main()中定义了try...except..嵌套调用了bar()和foo()，但是如果在foo()中出错了，只要main()捕捉到了我们就可以处理。

```
def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    try:
        bar('0')
    except Exception as e:
        print('Error:', e)
    finally:
        print('finally...')
```

调用堆栈

如果错误没有被捕捉，它就是会一直往上抛，最后被Python解释器捕捉，打印一个错误信息，然后程序退出。

```
$ python3 err.py
Traceback (most recent call last):
  File "err.py", line 11, in <module> #main出错，在11行，原因在9行
    main()
  File "err.py", line 9, in main #bar('0')出错，在9行，原因在6行
    bar('0')
  File "err.py", line 6, in bar #foo(s)出错，在6行，原因在3行
    return foo(s) * 2
  File "err.py", line 3, in foo  #foo(s)中return 10 / int(s)出错，原因打印
```

```
    return 10 / int(s)
ZeroDivisionError: division by zero
```

记录错误

不捕捉错误，解释器虽然会答应出来，但是程序也被结束了。我们希望可以打印错误，同时也希望继续执行。我们使用内置的logging模块来实现。经过配置，我们还可以把错误写进日志，来解读。

```
# err_logging.py

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) #在这里我们可以打印出错误，同时程序也被执行下去。

main()
print('END')
```

抛出错误

Python自己是有很多的错误类型来抛出，我们自己写的函数也可以抛出。首先我们要定义有一个错误类，选择好继承关系，然后用raise语句抛出一个错误实例。

```
# err_raise.py
class FooError(ValueError):
    pass

def foo(s):
    n = int(s)
    if n==0:
        raise FooError('invalid value: %s' % s)
    return 10 / n

foo('0')
```

还有另外的一种错误处理方式。

```
# err_reraise.py

def foo(s):
    n = int(s)
    if n==0:
        raise ValueError('invalid value: %s' % s)
    return 10 / n

def bar():
    try:
        foo('0')
    except ValueError as e:
        print('ValueError!')
        raise
```

```
bar()
```

我们可以看到，出错输出后，我们不但打印了一个ValueError后，还用raise把语句抛出，看似是措多此一举。实际上，这里我们打印只是为了追踪错误，实际过程中我们可能不知道怎么对这个错误进行处理，从而抛出到上级来处理。所以raise语句如果不带参数，就是把当前错误原样抛出。

51.调试

断言 assert

我们在某个测定的地方觉得有问题的时候，用断言写出来。

```
def foo(s):
    n = int(s)
    assert n != 0, 'n is zero!'
    return 10 / n

def main():
    foo('0')
```

assert表示的意思是：当 n!= 0 为真时继续，不为真时输出'n is zero!'。如果断言失败，会抛出AssertionError错误。

当一段程序中有很多和assert的时候，很麻烦，我们在运行的时候输入

```
python3 -0 test.py
```

就可以关闭assert。此处的assert就相当于pass。

logging

和assert相比，logging不会抛出错误，而是输出到文件中。

```
import logging
logging.basicConfig(level=logging.INFO)#添加后改变logging的配置。

s = '0'
n = int(s)
logging.info('n = %d' % n)
print(10 / n)
```

打开文件发现只有一句ZeroDivisionError，没有其他信息。应为logging是可以配置信息级别的。如上在import添加后，可以得到详细的输出。

```
$ python3 err.py
INFO:root:n = 0
Traceback (most recent call last):
  File "err.py", line 8, in <module>
    print(10 / n)
ZeroDivisionError: division by zero
```

logging的等级有：

1)debug 2)info 3)warning 4)error

当我们指定后面的等级时候，前面的就不再生效。可以直接动态的控制信息的输出等级。不用删除。

调试器 pdb

让Python来单步运行的方式，查看运行状态。

```
python -m pdb test.py
```

使用这个语句开启，后面使用命令来执行。

```
$ python3 -m pdb err.py
> /Users/michael/Github/learn-python3/samples/debug/err.py(2)<module>()
-> s = '0'
```

使用命令 l 来查看代码；

```
(Pdb) l
  1     # err.py
  2  -> s = '0'
  3     n = int(s)
  4     print(10 / n)
```

使用命令 n 来单步执行代码；

```
(Pdb) n
> /Users/michael/Github/learn-python3/samples/debug/err.py(3)<module>()
-> n = int(s)
```

使用命令 p 变量名称 来查询现在步骤的各个变量。

```
(Pdb) p s
'0'
```

使用命令 q 结束调试。

```
(Pdb) q
```

pdb.set_trace()

使用pdb逐步调试在代码有很多行的时候很难执行下去。这个时候我们import pdb后，在可能出错的地方放一个pdb.set_trace()，就可以设置一个断点。

```
# err.py
import pdb

s = '0'
n = int(s)
pdb.set_trace() # 运行到这里会自动暂停
print(10 / n)
```

遇到pdb/set_trace()，暂停后会进入调试模式，命令 p 变量名 查看变量。 命令 c 可以继续执行。

```
> /Users/michael/Github/learn-python3/samples/debug/err.py(7)<module>()
-> print(10 / n)
(Pdb) p n
0
(Pdb) c
Traceback (most recent call last):
  File "err.py", line 7, in <module>
    print(10 / n)
```

```
ZeroDivisionError: division by zero
```

这样效率会比pdb高一些，但是没有高到哪里去。

总结：工作中可以借用IDE来实现调试。

52.文件读写

一个正常的文件读流程如下：

```
>>> f = open('/Users/michael/test.txt', 'r') #前面为文件路径，后面为打开方式
>>> f.read()  #调用read()读取文件内容
'Hello, world!'
>>> f.close() # 关闭文件读取，释放系统资源。
```

文件读写的时候有可能产生IOError，这样close()不会被执行，那么系统资源不能被释放。为了保证每次都能关闭，我们可以使用try...except..来实现。

```
try:
    f = open('/path/to/file', 'r')
    print(f.read())
finally:
    if f:
        f.close()
```

但是如此使用实在繁琐，所以我们使用with语句来实现。不但更简介，而且不用调用close()方法。

```
with open('/path/to/file', 'r') as f:
    print(f.read())
```

读的模式：

1)read()是直接读取所有内容的，但是当文件很大的时候，内存就不够了。所以使用read(size)每次读取size个字节内容。

2)readline()可以每次读取一行内容，调用readlines()是一次读取所有行，并返回一个List.

总结：read()最方便，read(size)最保险，配置文件最好使用readlines()一次读取完成。

```
for line in f.readlines():
    print(line.strip()) # 把末尾的'\n'删掉
```

file-like Object

像open()函数这样有个read()方法的，都称为file-like Object。除了file外，还有字节流，网络流，自定义流等。StringIO是在内存中创建的file-like Object，常用作临时缓冲。

二进制文件

读取图片、视屏、二进制文本等二进制文件的时候：其他相同

```
>>> f = open('/Users/michael/test.jpg', 'rb')
```

字符编码

读取非UTF-8文本集的时候要加注读取的的字符集：

```
>>> f = open('/Users/michael/gbk.txt', 'r', encoding='gbk')
```

字符集如果出错要一个个处理很麻烦，我们可以普遍设置为忽略。

```
>>> f = open('/Users/michael/gbk.txt', 'r', encoding='gbk', errors='ignore')
```

写文件和读取文件都是使用open()函数的，只是在标志位使用的是w来标记。二进制使用wb。

```
>>> f = open('/Users/michael/test.txt', 'w') #标志位和读文件不同的是是w
>>> f.write('Hello, world!')
>>> f.close()
```

我们是可以反复使用write()方法来是多次写入，但是要保证写文件能够正常写入，所以要正确执行close()函数。系统可能在多次写入不是实时存储的，在最后执行close()的时候才正式提交。使用with的方式来做。

```
with open('/Users/michael/test.txt', 'w') as f:
    f.write('Hello, world!')
```

和read()相同，我们传入encoding参数的时候，可以把写入文件的字符格式转换成特定格式。

```
 f = open('/Users/michael/gbk.txt', 'w', encoding='gbk')
```

多使用with来控制IO是最好的。

53. StringIO 和 BytesIO

StringIO

写StringIO

```
>>> from io import StringIO
>>> f = StringIO()
>>> f.write('hello')  #多次写入
5
>>> f.write(' ')
1
>>> f.write('world!')
6
>>> print(f.getvalue()) #getvalue()用于获得写入的后的str
hello world!
```

读StringIO，和读取文件一样。

```
>>> from io import StringIO
>>> f = StringIO('Hello!\nHi!\nGoodbye!') #这里预先始化，可以直接读取内存中的。
>>> while True:  #使用with语法来读取。
...     s = f.readline()   #行读取模式。
...     if s == '':
...         break
...     print(s.strip()) #strip()去除\n达到换行
...
Hello!
Hi!
Goodbye!
```

BytesIO

写

```
>>> from io import BytesIO
>>> f = BytesIO()
>>> f.write('中文'.encode('utf-8'))
6
>>> print(f.getvalue())
b'\xe4\xb8\xad\xe6\x96\x87'
```

这里写入的是UTF-8编码的二进制字符。

读

```
>>> from io import BytesIO
>>> f = BytesIO(b'\xe4\xb8\xad\xe6\x96\x87')
>>> f.read()
b'\xe4\xb8\xad\xe6\x96\x87'
```

54.系统和文件目录

系统

```
>>> import os
>>> os.name # 操作系统类型 posix 就是linux或者unix、mac nt是windows
'nt'

>>> os.uname()  #win系统上没有此函数
posix.uname_result(sysname='Darwin', nodename='MichaelMacPro.local', release='14.3.0', version='Darwin Kernel Versic

AttributeError: module 'os' has no attribute 'uname'#win输入os.uname的结果

>>> os.environ #查询环境变量。

>>> os.environ.get('PATH') #查询环境变量Path
```

文件目录

操作文件和目录的方法一部分在os一部分在os.path中的。

在目录中的一些常用操作:

```
#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')
#拆分一个路径
>>> os.path.split('/Users/michael/testdir/file.txt')
('/Users/michael/testdir', 'file.txt')
#用拆分得到一个尾缀名。
>>> os.path.splitext('/path/to/file.txt')
('/path/to/file', '.txt')
#重命名文件
```

```
>>> os.rename('test.txt', 'test.py')
#删掉文件:
>>> os.remove('test.py')
```

我们在文件路径中不要自己去拼接，我们要使用os.path.join()这样在不同系统能返回正确的文件目录格式。拆分的时候就用os.path.splitext()

os模块中本身是没有文件复制的，但在shutil模块中补充filecopy()。

过滤文件

列出当前目录下的所有目录：

```
>>> [x for x in os.listdir('.') if os.path.isdir(x)]
['.lein', '.local', '.m2', '.npm', '.ssh', '.Trash', '.vim', 'Applications', 'Desktop', ...]
```

列出当前目录下的所有py文件：

```
>>> [x for x in os.listdir('.') if os.path.isfile(x) and os.path.splitext(x)[1]=='.py']
['apis.py', 'config.py', 'models.py', 'pymonitor.py', 'test_db.py', 'urls.py', 'wsgiapp.py']-
```