@命令行模式与python交互模式

python 进入交互模式

python hello.py是命令行模式

```
print('The quick brown fox', 'jumps over', 'the lazy dog')
```

`print()` 会依次打印每个字符串，遇到逗号“,”会输出一个空格 。

字符串

Python允许用 `'''...'''` 的格式表示多行内容

格式化

```
'Hi, %s, you have $%d.' % ('Michael', 1000000)
```

和C的风格类似

数据结构

列表 list 有序集合 随时添加和删除其中的元素

其元素类型可以不相同

其元素可以为列表，作用相当于二维数组

```
classmates = ['Michael', 'Bob', 'Tracy']
classmates.append('Adam')
classmates.insert(1, 'Jack')
classmates.pop(i)
```

元组 tuple 一旦初始化就不能修改

tuple的每个元素，指向永远不变。

```
classmates = ('Michael', 'Bob', 'Tracy')
t = (1,) 只有一个元素的元组的定义方法
```

字典 key-value对

```
d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}
d.pop('Bob')
```

set 一组key的集合，但不存储value. 没有重复值。要创建一个set，必须提供一个list作为输入集合。

```
s = set([1, 2, 3])
s.remove(4)
```

条件判断

if elif else

循环

for x in……:

while:

函数

Python的函数返回多值其实就是返回一个tuple

参数

位置参数

```
def enroll(name, gender):
```

默认参数

```
def enroll(name, gender, age=6, city='Beijing'):
```

可变参数

```
def calc(*numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum

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

tuple

可变参数就是传入的参数个数是可变的，可以是1个、2个到任意个，还可以是0个。

关键字参数

```
def person(name, age, **kw):
```

关键字参数允许你传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个dict 。

命名关键字参数

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

如果要限制关键字参数的名字，就可以用命名关键字参数

切片

```
L[:10:2]
```

字符串 `'xxx'` 也可以看成是一种list，每个元素就是一个字符。因此，字符串也可以用切片操作，只是操作结果仍是字符串

迭代

默认情况下，dict迭代的是key。如果要迭代value，可以用 `for value in d.values()` ，如果要同时迭代key和value，可以用 `for k, v in d.items()`

Python内置的 `enumerate` 函数可以把一个list变成索引-元素对，这样就可以在 `for` 循环中同时迭代索引和元素本身

列表生成式

```
[x * x for x in range(1, 11)]
```

列表生成式也可以使用两个变量来生成list

```
[k + '=' + v for k, v in d.items()]
```

for循环后面还可以加上if判断

```
[x * x for x in range(1, 11) if x % 2 == 0]
```

```
[m + n for m in 'ABC' for n in 'XYZ']
```

[]生成列表，而{}生成字典。

在Python中，这种一边循环一边计算的机制，称为生成器：generator

生成器：列表生成式把[]换成()即可。可以用next(g)来迭代，也可以用for循环

如果一个函数中包括yield关键字，那么就是一个generator

```
```

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        yield b
        a, b = b, a + b
        n = n + 1
    return 'done'
```

凡是可作用于for循环的对象都是Iterable类型

凡是可作用于next()函数的对象都是Iterator类型，惰性计算序列

高阶函数

一个函数接收另一个函数作为参数，就称之为高阶函数。

map

`map()` 传入的第一个参数是 `f` ，即函数对象本身。由于结果 `r` 是一个 `Iterator` ， `Iterator` 是惰性序列，因此通过 `list()` 函数让它把整个序列都计算出来并返回一个list。

reduce

```
reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)
```

filter

`filter()` 也接收一个函数和一个序列。和 `map()` 不同的是， `filter()` 把传入的函数依次作用于每个元素，然后根据返回值是 `True` 还是 `False` 决定保留还是丢弃该元素。

sorted

```
sorted([36, 5, -12, 9, -21], key=abs)
```

匿名函数

```
list(map(lambda x: x * x, [1, 2, 3, 4, 5, 6, 7, 8, 9]))
```

装饰器

本质上，decorator就是一个返回函数的高阶函数。所以，我们要定义一个能打印日志的decorator，可以定义如下：

```
```

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

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

把 `@log` 放到 `now()` 函数的定义处，相当于执行了语句：

```
now = log(now)
```

如果decorator本身需要传入参数，那就需要编写一个返回decorator的高阶函数，写出来会更复杂

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

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

```
# now = decorator(now)
# now = wrapper()
```

偏函数

`functools.partial` 就是帮助我们创建一个偏函数的，不需要我们自己定义 `int2()` ，可以直接使用下面的代码创建一个新的函数 `int2` ：

```
>>> import functools
>>> int2 = functools.partial(int, base=2)
>>> int2('1000000')
64
>>> int2('1010101')
85
```

所以，简单总结 `functools.partial` 的作用就是，把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单。

注意到上面的新的 `int2` 函数，仅仅是把 `base` 参数重新设定默认值为 `2` ，但也可以在函数调用时传入其他值

模块

一个.py文件就是一个模块，多个模块层级组合起来可以成为一个package

类

由于类可以起到模板的作用，因此，可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。通过定义一个特殊的 `__init__` 方法，在创建实例的时候，就把 `name` ， `score` 等属性绑上去：

```
class Student(object):

    def __init__(self, name, score):
        self.name = name
        self.score = score
```

注意到 `__init__` 方法的第一个参数永远是 `self` ，表示创建的实例本身，因此，在 `__init__` 方法内部，就可以把各种属性绑定到 `self` ，因为 `self` 就指向创建的实例本身。

和普通的函数相比，在类中定义的函数只有一点不同，就是第一个参数永远是实例变量 `self` ，并且，调用时，不用传递该参数。除此之外，类的方法和普通函数没有什么区别，所以，你仍然可以用默认参数、可变参数、关键字参数和命名关键字参数。

如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线 `__` ，在Python中，实例的变量名如果以 `__` 开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问。

有些时候，你会看到以一个下划线开头的实例变量名，比如 `_name` ，这样的实例变量外部是可以访问的，但是，按照约定俗成的规定，当你看到这样的变量时，意思就是，“虽然我可以被访问，但是，请把我视为私有变量，不要随意访问”。

继承和多态

if **name** = '**main**':

测试的时候调用

__ slots __ 限制该类允许定义的属性

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

@property 装饰器负责把一个方法变成属性调用

```
class Student(object):

    @property
```

```
@property
# 实质上是getter
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
```

定制类

__ str __:控制print的行为

__ iter __ __ next __ ： 控制 for ..... in的行为

__ getitem __ ： 实现类似list的按照下标取元素

__ getattr __ :当调用不存在的属性或函数a时，python会尝试调用__ getattr __ (self,'a')来尝试获得属性。

元类

通过type()函数创建的类和直接写class是完全一样的

要创建一个class对象， `type()` 函数依次传入3个参数：

1. class的名称；
2. 继承的父类集合，注意Python支持多重继承，如果只有一个父类，别忘了tuple的单元素写法；
3. class的方法名称与函数绑定，这里我们把函数 `fn` 绑定到方法名 `hello` 上。

try:

except:

except:

finally:

调试：

print()大法好

assert condition,expression

logging 允许记录信息的级别有debug，info，warning，error等

pdb调试器，单步方式运行. pdb.set_trace()设置断点

单元测试

为了编写单元测试，我们需要引入Python自带的 `unittest` 模块，编写 `mydict_test.py` 如下：

```
import unittest

from mydict import Dict

class TestDict(unittest.TestCase):

    def test_init(self):
        d = Dict(a=1, b='test')
        self.assertEqual(d.a, 1)
        self.assertEqual(d.b, 'test')
        self.assertTrue(isinstance(d, dict))

    def test_key(self):
        d = Dict()
        d['key'] = 'value'
        self.assertEqual(d.key, 'value')

    def test_attr(self):
        d = Dict()
        d.key = 'value'
        self.assertTrue('key' in d)
        self.assertEqual(d['key'], 'value')
```

```
    def test_keyerror(self):
        d = Dict()
        with self.assertRaises(KeyError):
            value = d['empty']

    def test_attrerror(self):
        d = Dict()
        with self.assertRaises(AttributeError):
            value = d.empty
```

编写单元测试时，我们需要编写一个测试类，从 `unittest.TestCase` 继承。

以 `test` 开头的方法就是测试方法，不以 `test` 开头的方法不被认为是测试方法，测试的时候不会被执行。

对每一类测试都需要编写一个 `test_xxx()` 方法。由于 `unittest.TestCase` 提供了很多内置的条件判断，我们只需要调用这些方法就可以断言输出是否是我们所期望的。最常用的断言就是 `assertEqual()`：

文件读写

读文件

要以读文件的模式打开一个文件对象，使用Python内置的 `open()` 函数，传入文件名和标示符：

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

调用 `read()` 方法可以一次读取文件的全部内容，Python把内容读到内存，用一个 `str` 对象表示，为了保证无论是否出错都能正确地关闭文件，我们可以使用 `try ... finally` 来实现：

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

Python引入了 `with` 语句来自动帮我们调用 `close()` 方法：

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

调用 `read()` 会一次性读取文件的全部内容，如果文件有10G，内存就爆了，所以，要保险起见，可以反复调用 `read(size)` 方法，每次最多读取size个字节的内容。另外，调用 `readline()` 可以每次读取一行内容，调用 `readlines()` 一次读取所有内容并按行返回 `list` 。因此，要根据需要决定怎么调用。

前面讲的默认都是读取文本文件，并且是UTF-8编码的文本文件。要读取二进制文件，比如图片、视频等等，用 `'rb'` 模式打开文件即可 。

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

要读取非UTF-8编码的文本文件，需要给 `open()` 函数传入 `encoding` 参数，例如，读取GBK编码的文件 。

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

写文件和读文件是一样的，唯一区别是调用 `open()` 函数时，传入标识符 `'w'` 或者 `'wb'` 表示写文本文件或写二进制文件

```
f = open('/Users/michael/test.txt', 'w')
>>> f.write('Hello, world!')
>>> f.close()
```

with 语句可以保证close，close就意味着buffer里面的内容真正写入了磁盘。

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

操作文件和目录

把两个路径合成一个时，不要直接拼字符串，而要通过 `os.path.join()` 函数，这样可以正确处理不同操作系统的路径分隔符。

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

同样的道理，要拆分路径时，也不要直接去拆字符串，而要通过 `os.path.split()` 函数，这样可以把一个路径拆分为两部分，后一部分总是最后级别的目录或文件名：

```
os.path.split('/Users/michael/testdir/file.txt')
```

`os.path.splitext()` 可以直接让你得到文件扩展名

```
>>> os.path.splitext('/path/to/file.txt')
('/path/to/file', '.txt')
```

Python内置的 `json` 模块提供了非常完善的Python对象到JSON格式的转换。我们先看看如何把Python对象变成一个JSON

```
>>> import json
>>> d = dict(name='Bob', age=20, score=88)
>>> json.dumps(d)
'{"age": 20, "score": 88, "name": "Bob"}'
```

`json` 模块的 `dumps()` 和 `loads()` 函数是定义得非常好的接口的典范。当我们使用时，只需要传入一个必须的参数。但是，当默认的序列化或反序列机制不满足我们的要求时，我们又可以传入更多的参数来定制序列化或反序列化的规则

```
json.dumps(s, default=student2dict)
```

多线程

我们只需要使用 `threading` 这个高级模块，启动一个线程就是把一个函数传入并创建 `Thread` 实例，然后调用 `start()` 开始执行

```
import time, threading
```

```
# 新线程执行的代码:
def loop():
    print('thread %s is running...' % threading.current_thread().name)
    n = 0
    while n < 5:
        n = n + 1
        print('thread %s >>> %s' % (threading.current_thread().name, n))
        time.sleep(1)
    print('thread %s ended.' % threading.current_thread().name)

print('thread %s is running...' % threading.current_thread().name)
t = threading.Thread(target=loop, name='LoopThread')
t.start()
t.join()
print('thread %s ended.' % threading.current_thread().name)
```

lock问题可以用lock.acquire()和lock.release()解决。

```
lock.acquire()
        try:
            # 放心地改吧:
            change_it(n)
        finally:
            # 改完了一定要释放锁:
            lock.release()
```

ThreadLocal

在多线程环境下，每个线程都有自己的数据。一个线程使用自己的局部变量比使用全局变量好，因为局部变量只有线程自己能看见，不会影响其他线程，而全局变量的修改必须加锁。

但是局部变量也有问题，就是在函数调用的时候，传递起来很麻烦。

```
import threading

# 创建全局ThreadLocal对象:
local_school = threading.local()

def process_student():
    # 获取当前线程关联的student:
    std = local_school.student
    print('Hello, %s (in %s)' % (std, threading.current_thread().name))

def process_thread(name):
    # 绑定ThreadLocal的student:
    local_school.student = name
    process_student()

t1 = threading.Thread(target= process_thread, args=('Alice',),
name='Thread-A')
t2 = threading.Thread(target= process_thread, args=('Bob',),
name='Thread-B')
t1.start()
t2.start()
t1.join()
t2.join()
```

正则表达式

因此我们强烈建议使用Python的 r 前缀，就不用考虑转义的问题了。

```
s = r'ABC\-001' # Python的字符串
# 对应的正则表达式字符串不变：
# 'ABC\-001'
```

match() 方法判断是否匹配，如果匹配成功，返回一个 Match 对象，否则返回 None 。常见的判断方法就是：

```
test = '用户输入的字符串'
if re.match(r'正则表达式', test):
    print('ok')
else:
    print('failed')
```

split() 方法切分

```
>>> re.split(r'[\s\,\;]+', 'a,b;; c  d')
['a', 'b', 'c', 'd']
```

Group() 分组提取内容——()。

```
>>> m = re.match(r'^(\d{3})-(\d{3,8})$', '010-12345')
>>> m
<_sre.SRE_Match object; span=(0, 9), match='010-12345'>
>>> m.group(0)
'010-12345'
>>> m.group(1)
'010'
>>> m.group(2)
'12345'
```

注意到 `group(0)` 永远是原始字符串，`group(1)`、`group(2)`……表示第1、2、……个子串。

datetime

datetime是Python处理日期和时间的标准库。

```
from datetime import datetime
datetime.now()
```

collections

namedtuple

`namedtuple` 是一个函数，它用来创建一个自定义的 `tuple` 对象，并且规定了 `tuple` 元素的个数，并可以用属性而不是索引来引用 `tuple` 的某个元素。

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(1, 2)
>>> p.x
1
>>> p.y
2
```

deque

使用 `list` 存储数据时，按索引访问元素很快，但是插入和删除元素就很慢了，因为 `list` 是线性存储，数据量大的时候，插入和删除效率很低。

deque是为了高效实现插入和删除操作的双向列表，适合用于队列和栈

```
>>> from collections import deque
>>> q = deque(['a', 'b', 'c'])
>>> q.append('x')
>>> q.appendleft('y')
>>> q
deque(['y', 'a', 'b', 'c', 'x'])
```

hashlib & hmac

提供哈希算法支持

itertools

Python的内建模块 `itertools` 提供了非常有用的用于操作迭代对象的函数。

`cycle()` 会把传入的一个序列无限重复下去：

```
>>> import itertools
>>> cs = itertools.cycle('ABC') # 注意字符串也是序列的一种
>>> for c in cs:
...     print(c)
...
'A'
'B'
'C'
'A'
'B'
'C'
...
```

`repeat()` 负责把一个元素无限重复下去，不过如果提供第二个参数就可以限定重复次数 。

我们会通过 `takewhile()` 等函数根据条件判断来截取出一个有限的序列

```
>> natuals = itertools.count(1)
>>> ns = itertools.takewhile(lambda x: x <= 10, natuals)
>>> list(ns)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

`chain()` 可以把一组迭代对象串联起来，形成一个更大的迭代器

contextlib

有很多装饰器，便于编写更简洁的代码。

任何对象，只要正确实现了上下文管理，就可以用于 `with` 语句 。

实现上下文管理是通过 `__enter__` 和 `__exit__` 这两个方法实现的。

@contextmanager

编写 `__enter__` 和 `__exit__` 仍然很繁琐，因此Python的标准库 `contextlib` 提供了更简单的写法，上面的代码可以改写如下

```
from contextlib import contextmanager

class Query(object):

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

    def query(self):
        print('Query info about %s...' % self.name)

@contextmanager
def create_query(name):
    print('Begin')
    q = Query(name)
    yield q

    print('End')
```

```
    print('End')
```

很多时候，我们希望在某段代码执行前后自动执行特定代码，也可以用 `@contextmanager` 实现。

```
@contextmanager
def tag(name):
    print("<%s>" % name)
    yield
    print("</%s>" % name)

with tag("h1"):
    print("hello")
    print("world")
```

1. `with` 语句首先执行 `yield` 之前的语句，因此打印出 `<h1>`；
2. `yield` 调用会执行 `with` 语句内部的所有语句，因此打印出 `hello` 和 `world`；
3. 最后执行 `yield` 之后的语句，打印出 `</h1>`。

如果一个对象没有实现上下文，我们就不能把它用于 `with` 语句。这个时候，可以用 `closing()` 来把该对象变为上下文对象。例如，用 `with` 语句使用 `urlopen()`：

```
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('https://www.python.org')) as page:
    for line in page:
        print(line)
```

常见第三方库

pillow

图像操作

virtualenv

```
Mac:myproject michael$ virtualenv --no-site-packages venv
Mac:myproject michael$ source venv/bin/activate
(venv)Mac:myproject michael$ deactivate
```