Table of Contents

关于本书

在熟悉Python语法知识基础上，学习使用Python-Django后端框架的使用以前端框架Bootstrap、Semantic-UI的使用。

Author Email	**x_jwei@qq.com**
Create Time	**2018年1月25日 23:57:13**
版本	**v2.0**

Python部分

Python介绍

[TOC]

1.Python应用场景

- **Web**应用开发

  Python经常被用于Web开发。比如，通过mod_wsgi模块，Apache可以运行用Python编写的Web程序。Python定义了WSGI标准应用接口来协调Http服务器与基于Python的Web程序之间的通信。一些Web框架，如Django , Tornado，Falsk，TurboGears, web2py, Zope等，可以让程序员轻松地开发和管理复杂的Web程序。

- 操作系统管理、服务器运维的自动化脚本

  在很多操作系统里，Python是标准的系统组件。 。一般说来，Python编写的系统管理脚本在可读性、性能、代码重用度、扩展性几方面都优于普通的shell脚本。

- 科学计算

  NumPy,SciPy,Matplotlib可以让Python程序员编写科学计算程序。

- 桌面软件

  PyQt、PySide、wxPython、PyGTK是Python快速开发桌面应用程序的利器。

- 服务器软件（网络软件）

  Python对于各种网络协议的支持很完善，因此经常被用于编写服务器软件、网络爬虫。第三方库Twisted支持异步网络编程和多数标准的网络协议(包含客户端和服务器)，并且提供了多种工具，被广泛用于编写高性能的服务器软件。

- 游戏

  很多游戏使用C++编写图形显示等高性能模块，而使用Python或者Lua编写游戏的逻辑、服务器。相较于Python，Lua的功能更简单、体积更小；而Python则支持更多的特性和数据类型。

- 构思实现，产品早期原型和迭代

  YouTube、Google、Yahoo!、NASA都在内部大量地使用Python。

2.Python的优缺点

- 优点

1. 简单————Python是一种代表简单主义思想的语言。阅读一个良好的Python程序就感觉像是在读英语一样，尽管这个英语的要求非常严格！Python的这种伪代码本质是它最大的优点之一。它使你能够专注于解决问题而不是去搞明白语言本身。

2. 易学————就如同你即将看到的一样，Python极其容易上手。前面已经提到了，Python有极其简单的语法。

3. 免费、开源————Python是FLOSS（自由/开放源码软件）之一。简单地说，你可以自由地发布这个软件的拷贝、阅读它的源代码、对它做改动、把它的一部分用于新的自由软件中。FLOSS是基于一个团体分享知识的概念。这是为什么Python如此优秀的原因之一——它是由一群希望看到一个更加优秀的Python的人创造并经常改进着的。

4. 高层语言————当你用Python语言编写程序的时候，你无需考虑诸如如何管理你的程序使用的内存一类的底层细节。

5. 可移植性————由于它的开源本质，Python已经被移植在许多平台上（经过改动使它能够工作在不同平台上）。如果你小心地避免使用依赖于系统的特性，那么你的所有Python程序无需修改就可以在下述任何平台上面运行。这些平台包括Linux、Windows、FreeBSD、Macintosh、Solaris、OS/2、Amiga、AROS、AS/400、BeOS、OS/390、z/OS、Palm OS、QNX、VMS、Psion、Acom RISC OS、VxWorks、PlayStation、Sharp Zaurus、Windows CE甚至还有PocketPC、Symbian以及Google基于linux开发的Android平台！

6. 解释性————这一点需要一些解释。一个用编译性语言比如C或C++写的程序可以从源文件（即C或C++语言）转换到一个你的计算机使用的语言（二进制代码，即0和1）。这个过程通过编译器和不同的标记、选项完成。当你运行你的程序的时候，连接/转载器软件把你的程序从硬盘复制到内存中并且运行。而Python语言写的程序不需要编译成二进制代码。你可以直接从源代码运行程序。在计算机内部，Python解释器把源代码转换成称为字节码的中间形式，然后再把它翻译成计算机使用的机器语言并运行。事实上，由于你不再需要担心如何编译程序，如何确保连接转载正确的库等等，所有这一切使得使用Python更加简单。由于你只需要把你的Python程序拷贝到另外一台计算机上，它就可以工作了，这也使得你的Python程序更加易于移植。

7. 面向对象————Python既支持面向过程的编程也支持面向对象的编程。在“面向过程”的语言中，程序是由过程或仅仅是可重用代码的函数构建起来的。在“面向对象”的语言中，程序是由数据和功能组合而成的对象构建起来的。与其他主要的语言如C++和Java相比，Python以一种非常强大又简单的方式实现面向对象编程。

8. 可扩展性————如果你需要你的一段关键代码运行得更快或者希望某些算法不公开，你可以把你的部分程序用C或C++编写，然后在你的Python程序中使用它们。
9. 丰富的库————Python标准库确实很庞大。它可以帮助你处理各种工作，包括正则表达式、文档生成、单元测试、线程、数据库、网页浏览器、CGI、FTP、电子邮件、XML、XML-RPC、HTML、WAV文件、密码系统、GUI（图形用户界面）、Tk和其他与系统有关的操作。记住，只要安装了Python，所有这些功能都是可用的。这被称作Python的“功能齐全”理念。除了标准库以外，还有许多其他高质量的库，如wxPython、Twisted和Python图像库等等。
10. 规范的代码————Python采用强制缩进的方式使得代码具有极佳的可读性。

- 缺点
  1. 运行速度，有速度要求的话，用C++改写关键部分吧。
  2. 国内市场较小（国内以python来做主要开发的，目前只有一些web2.0公司）。但时间推移，目前很多国内软件公司，尤其是游戏公司，也开始规模使用他。
  3. 中文资料匮乏（好的python中文资料屈指可数）。托社区的福，有几本优秀的教材已经被翻译了，但入门级教材多，高级内容还是只能看英语版。
  4. 构架选择太多（没有像C#这样的官方.net构架，也没有像ruby由于历史较短，构架开发的相对集中。Ruby on Rails 构架开发中小型web程序天下无敌）。不过这也从另一个侧面说明，python比较优秀，吸引的人才多，项目也多

3.Python之禅

> 优美胜于丑陋（Python 以编写优美的代码为目标）
>
> 明了胜于晦涩（优美的代码应当是明了的，命名规范，风格相似）
>
> 简洁胜于复杂（优美的代码应当是简洁的，不要有复杂的内部实现）
>
> 复杂胜于凌乱（如果复杂不可避免，那代码间也不能有难懂的关系，要保持接口简洁）
>
> 扁平胜于嵌套（优美的代码应当是扁平的，不能有太多的嵌套）
>
> 间隔胜于紧凑（优美的代码有适当的间隔，不要奢望一行代码解决问题）
>
> 可读性很重要（优美的代码是可读的）
>
> 即便假借特例的实用性之名，也不可违背这些规则（这些规则至高无上）

> 不要包容所有错误，除非你确定需要这样做（精准地捕获异常，不写 except:pass 风格的代码）
>
> 当存在多种可能，不要尝试去猜测
>
> 而是尽量找一种，最好是唯一一种明显的解决方案（如果不确定，就用穷举法）
>
> 虽然这并不容易，因为你不是 Python 之父（这里的 Dutch 是指 Guido ）
>
> 做也许好过不做，但不假思索就动手还不如不做（动手之前要细思量）
>
> 如果你无法向人描述你的方案，那肯定不是一个好方案；反之亦然（方案测评标准）
>
> 命名空间是一种绝妙的理念，我们应当多加利用（倡导与号召）

4.Python帮助的使用

- [ ] python的官网： www.python.org
- [ ] 命令提示符：通过help查找模块中的注释
- [x] 离线CHM帮助：推荐。￣◡￣。
- [ ] 离线html帮助
- [ ] IDE工具：PyCharm、eclipse

5.关于版本

- 版本的查看：`python -V`
- 版本的进化

最初**2**系列	趋势系列	过渡**2**和**3**
2.x.x	3.x.x	2.7.x

- 关于版本的选择

  除非你仍然要维护老版本代码，否则现在学习Python的底线是2.6。不能再低于这个版本了，python2.7.x系列2020年放弃维护。

- 关于版本差异：《python学习手册》

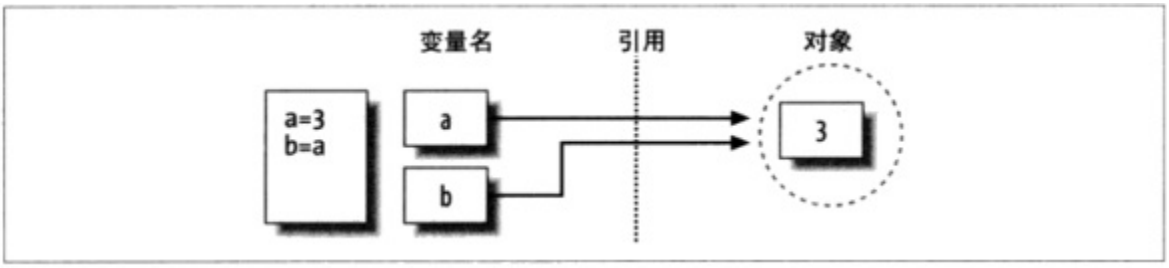
Python初级

1.Python程序的运行机制

- 逐行解释

  和C/C++/Java等语言需要提前静态编译不同。python的代码是逐行解释的。缺少静态编译的检查意味着在运行时可能会出现错误，导致代码稳定性差一些。PyChecker是一个python代码的静态分析工具，它可以帮助查找python代码的bug, 会对代码的复杂度和格式提出警告。

- 引用



- 查看变量在内存的id： `id(var)`
- 查看变量的引用次数（PGC机制）：

```
import sys
a = 2
b = a
sys.getrefcount(a)
```

- 可变对象和不可变对象
  - 不可变（immutable）：int、字符串(string)、float、（数值型number）、元组（tuple)
  - 可变（mutable）：字典型(dictionary)、集合(set)、列表型(list)、无*init*的class中的对象
- 深拷贝，浅拷贝
  - 在Python中=即引用，所以在必要时需要深拷贝， 对比以下情况的不同

```
#不的数据类型结果不同：
#如果是可变类型：字典、列表传引用；基本数据类型传值
def test(num):
```

```
    num += num
    # num = num + num
    print(num)

test(a)
print(a)
```

- 使用深拷贝

```
# 万能
from copy import deepcopy
a = [100]
b = deepcopy(a)
print(id(a),id(b))

# list等
l = [100,200]
ll = l[::]
```

2.Python的变量

- 类型

- 类型转换函数

函数	说明
int(x [,base])	将x转换为一个整数
long(x [,base])	将x转换为一个长整数
float(x)	将x转换到一个浮点数
complex(real [,imag])	创建一个复数
str(x)	将对象 x 转换为字符串
repr(x)	将对象 x 转换为表达式字符串
eval(str)	用来计算在字符串中的有效Python表达式,并返回一个对象
tuple(s)	将序列 s 转换为一个元组
list(s)	将序列 s 转换为一个列表
chr(x)	将一个整数转换为一个字符
unichr(x)	将一个整数转换为Unicode字符
ord(x)	将一个字符转换为它的整数值
hex(x)	将一个整数转换为一个十六进制字符串
oct(x)	将一个整数转换为一个八进制字符串

- 判断一个变量的类型
  - 方法一

```
>>> a = 4
>>> print type(a)
<type 'int'>
>>> type(a) == int
True
```

  - 方法二

```
>>> a = 7
>>> isinstance(a,int)
True
```

3.Python的关键字

- 什么是关键字

> python一些具有特殊功能的标示符，这就是所谓的关键字
>
> 关键字，是python已经使用的了，所以不允许开发者自己定义和关键字相同的名字的标示符

- 查看关键字:

```
and      as      assert     break     class      continue    def      del
elif     else    except     exec      finally    for         from     global
if       in      import     is        lambda     not         or       pass
print    raise   return     try       while      with        yield
```

- 通过代码查看

```
import keyword
print(keyword.list)
```

4.Python生成随机数

random模块

- 随机整数
  - random.randint(a,b)：返回随机整数x,a<=x<=b
  - random.randrange(start,stop,[,step])：返回一个范围在(start,stop,step)之间的随机整数，不包括结束值。
- 随机实数
  - random.random( ):返回0到1之间的浮点数
  - random.uniform(a,b):返回指定范围内的浮点数。
- random.shuffle(*x*[, *random*])
- random.sample(seq[, k])

Python中级

1、面向过程

(1)特殊函数

函数	格式	用法
map	map(func, seq1[, seq2...])	
zip	zip(*iterater)	
reduce	reduce(func, seq[, init])	
filter	filter(function or None, sequence)	

(2)lambda的特殊用法

```
def test(a,b,func)
    return func(a,b)

func_new = input("请输入一个表达式") #resume func_new: lambda x,y:x+y
func_new = eval(func_new) #python 3
result = test(3,4,func_new)
print(result)
```

2、面向对象

(1) 类

- 变量标识符

标识符	含义
_x	(1)解释器中使用：重复上一句命令(2)临时变量，不关心取值：`for _ in range(10):print("_")` (3)约定俗成， 应该被视为API中非公开的部分(4) “from <模块/包名> import *”中不会导入，除非以模块导入“import <模块/包名>”,访问：<模块/包名>.<属性名>
__x	相当于C++的private，只能在本类中使用。（实际上可以用，只不过发生了name mangling而已，可以用_Class__x 的方式访问）
__x__	系统（build-in）内置变量，如：__init__(), __name__

- __init__()方法和__new__()方法

    - __new__()只负责创建对象，然后return对象
    - __init__() 利用 __new__()创建后的对象，对其初始化
- 类属性、实例属性

```
class Tool(object):
    # 类属性
    num = 0

    # 方法
    def __init__(self, new_name):
        # 实例属性
        self.name = new_name
        # 对类属性+=1
        Tool.num += 1

if __name__ == "__main__":
    tool1 = Tool("铁锹")
    tool2 = Tool("工兵铲")
    tool3 = Tool("水桶")
    print(Tool.num)
```

- 类方法、实例方法、静态方法

```
class Game(object):

    #类属性
    num = 0

    #实例方法
    def __init__(self):
        #实例属性
        self.name = "laowang"

    #类方法
    @classmethod
    def add_num(cls):
        cls.num = 100

    #静态方法
    @staticmethod
    def print_menu():
        print("----------------------")
        print(" 穿越火线V11.1 ")
        print(" 1. 开始游戏")
        print(" 2. 结束游戏")
        print("----------------------")

game = Game()
# 调用类方法的方式一：可以通过类的名字调用类方法
Game.add_num()#
# 调用类方法的方式二：还可以通过这个类创建出来的对象 去调用这个类方法
```

```
game.add_num()
print(Game.num)

#Game.print_menu()#通过类 去调用静态方法
game.print_menu()#通过实例对象 去调用静态方法
```

- 子类调用父类的重写方法

```
class Animal:
    def eat(self):
        print("-----吃----")
    def drink(self):
        print("-----喝----")
    def sleep(self):
        print("-----睡觉----")
    def run(self):
        print("-----跑----")

class Dog(Animal):
    def bark(self):
        print("----汪汪叫---")

class Xiaotq(Dog):
    def fly(self):
      print("----飞----")

    def bark(self):
      print("----狂叫-----")

      #第1种调用被重写的父类的方法
      #Dog.bark(self)

      #第2种
      super().bark()

  xiaotq = Xiaotq()
  xiaotq.fly()
  xiaotq.bark()
  xiaotq.eat()
```

- python实现类似C++伪多态

```
class Dog(object):
def print_self(self):
print("Dog...")
class Xiaotq(Dog):
def print_self(self):
print("Xiaotq...")
def introduce(temp):
temp.print_self()
```

```
if __name__ == "__main__":
dog1 = Dog()
dog2 = Xiaotq()
introduce(dog1)
introduce(dog2)
```

(2)异常

- 普通异常

```
  #coding=utf-8

  try:
      num = input("xxx:")
      int(num)
      #11/0
      #open("xxx.txt")
      #print(num)
      print("-----1----")
  # python2这样写：except NameError,FileNotFoundError：
  except (NameError,FileNotFoundError):
      print("如果捕获到异常后做的 处理....")
  # python2这样写：except
  except Exception as ret:
      print("如果用了Exception,那么意味着只要上面的except没有捕获到异常,这个except一定会捕获到"
)
      print(ret)
  else:
      print("没有异常才会执行的功能")
  finally:
      print("------finally-----")

  print("-----2----")
```

- 异常类

```
class ShortInputException(Exception):
    # 用户自定义异常
    def __init__(self, length, atleast):
        self.length = length
        self.atleast = atleast

def main():
    try:
        s = input("请输入")
        if len(s) < 3:
            # 引发自己的异常
```

```
            raise ShortInputException(len(s), 3)
    # result指向了创建的ShortInputExcception对象
    except ShortInputExcception as result:
        print("ShortInputException:输入的长度是 %d,长度至少应该是 %d",%(result.length,resul
t.atleast))
    else:
        print("没有发生异常")
```

(3)设计模式

- 简单工厂模式：通过剥离CarStore类，解耦特定的方法

```
class CarStore(object):
    def __init__(self):
        self.factory = Factory()

    def order(self, car_type):
        return self.factory.select_car_by_type(car_type)

class Factory(object):
    def select_car_by_type(self, car_type):
        if car_type=="索纳塔":
            return Suonata()
        elif car_type=="名图":
            return Mingtu()
        elif car_type=="ix35":
            return Ix35()

class Car(object):
    def move(self):
        print("车在移动....")
    def music(self):
        print("正在播放音乐....")
    def stop(self):
        print("车在停止....")

class Suonata(Car):
    pass

class Mingtu(Car):
    pass

class Ix35(Car):
    pass

car_store = CarStore()
car = car_store.order("索纳塔")
car.move()
car.music()
car.stop()
```

- 工厂方法模式:父类只设计接口，子类通过继承负责实现

```
from abc import ABCMeta, abstractmethod
class Store(object):
    __metaclass__ = ABCMeta

    @abstractmethod
    def select_car(self):
        pass

    def order(self, car_type):
        return self.select_car(car_type)

class BMWCarStore(Store):
    def select_car(self, car_type):
        return BMWFactory().select_car_by_type(car_type)

class BMWFactory(object):
    def select_car_by_type(self, car_type):
        pass

bmw_store = BMWCarStore()
bmw = bmw_store.order("720li")
```

- 单例模式

```
class Dog(object):
    # 类的私有属性，用来保存初次创建的对象实例
    __instance = None
    __init_flag = False
    def __new__(cls, *more):
        if cls.__instance is None:
            cls.__instance = object.__new__(cls)
            return cls.__instance
        else:
            # return 上一次初次创建对象的引用
            return cls.__instance

    def __init__(self, name):
        if Dog.__init_flag is False:
            self.name = name
            Dog.__init_flag = True

 if "main" == name:
    a = Dog("旺财")
    b = Dog("哮天犬")
    print(id(a))
    print(a.name)
    print(id(b), b.name)
```

3、包和模块

- __all__ 在模块中的使用

```
# sendmsg.py
# test2不可被其他模块使用
__all__ = ['test1','num','Test']

def test1():
    print('-------test1---------')

def test2():
    print('-------test2---------')
    num = 100

class Test(object)
    pass
```

- __init__.py在包中的使用

```
__all__ = ['sendmsg']

from . import sendmsg
```

- python模块的发布
  - 在要发布包（一个目录）的同级目录建立setup.py文件
  - 编辑setup.py文件,py_modules指明包中的模块

```
from distutils.core import setup

setup(name="willxin",
      version="1.0",
      description="willxin's module",
      author="willxin",
      py_modules=['TestMsg.sendmsg', 'TestMsg.recvmsg'])
```

  - 构建bulid目录：$ python setup.py build
  - 创建tar包：$ python setup.py sdist
  - 他人安装包：python setup.py install

4、命名空间

就像Python之禅中的最后一句所说，Python对命名空间（NameSpace）频繁使用

命名空间	本质	备注
包	文件夹	包含许多.py文件，可用__init__.py定制
模块	.py文件	包含各种属性和方法，可用__all__定制
类	Class	定义了各种属性和方法
函数	Method	包含各种属性和内嵌方法
变量	对应的实例化对象	包含各种属性和方法

5、生成器

（1）创建的方式

- b = (x*2 for x in range(10))
- yield

```
def createNum():
    a, b =  0, 1
    for i in range(5):
        yield b
        a,b = b, a+b

result = createNum()
next(result)
```

Python高级

1、模块重载入

用于模块被外部修改后需要重新载入的情况

```
>>>  from imp import reload
>>>  reload(md5_code) #假设是md5_code.py模块
```

2、循环导入

假设A模块需要调用B模块的b()，B模块的b()中又调用了A模块中的a()，此时出现循环导入的现象。

解决方法：

（1）设计的时候就独立a()、b()，让C模块调用a()、b()。

（2）放在函数题目导入

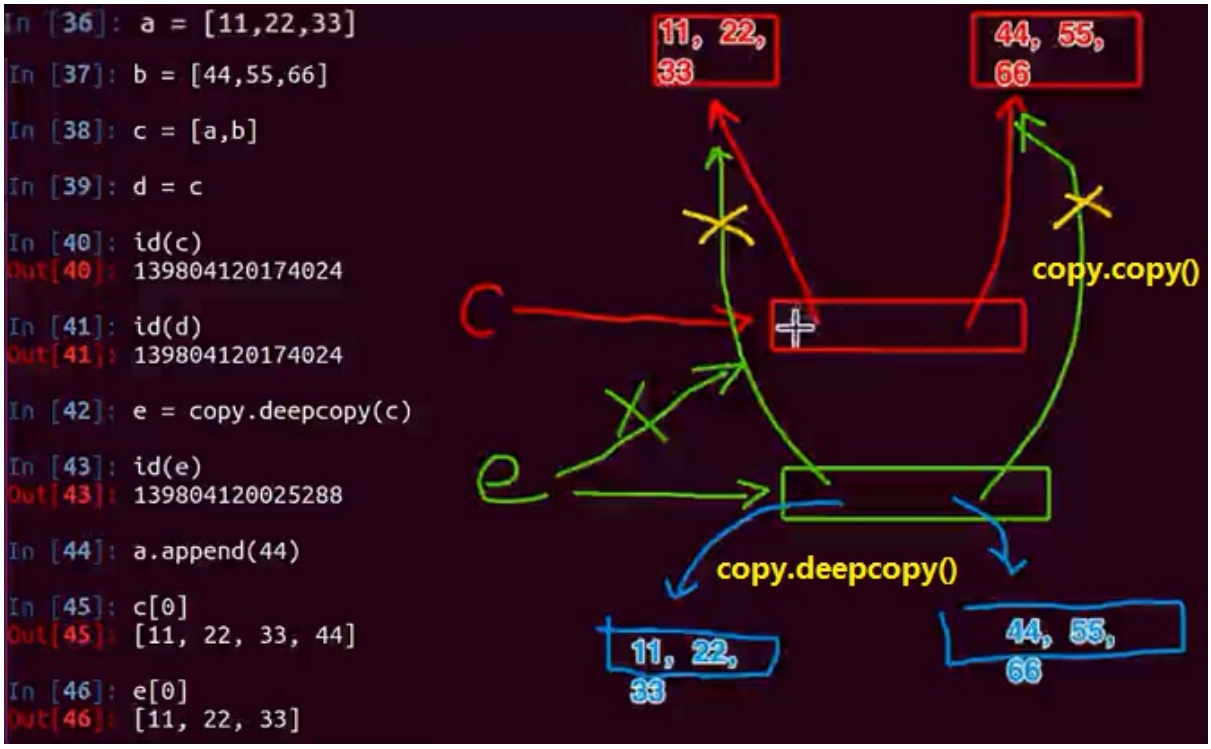
3、is和==

（1）is用来比较是否指向了同一个对象，即id()之后相等。

（2）==比较的是值相等

4、深拷贝和浅拷贝

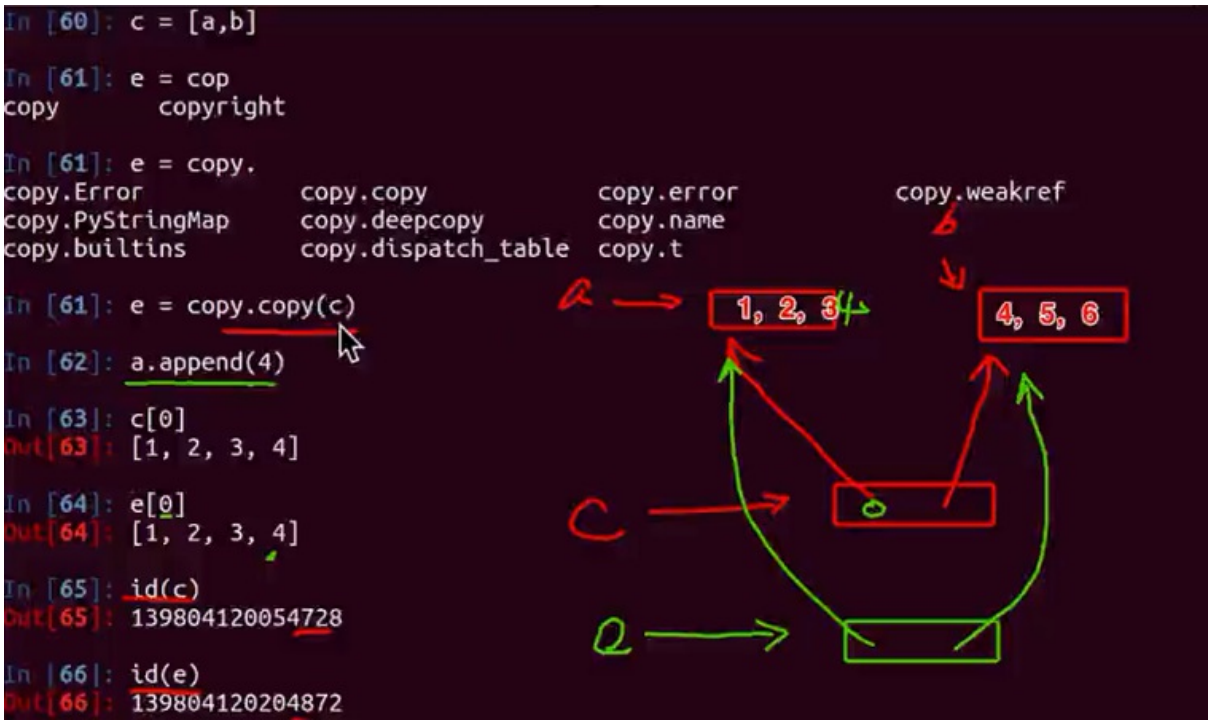
（1）列表的深浅拷贝



（2）**copy.deepcopy()** 、**copy.copy()**的区别

copy.deepcopy()：能把所有的引用都识别，进行深拷贝

copy.copy()：只能对可变类型（tuple除外）进行第一层深拷贝



5、属性**property**

（1）私有属性添加getter和setter方法

```
class Money(object):
    def __init__(self):
        self.__money = 0

    def getMoney(self):
        return self.__money

    def setMoney(self, value):
        if isinstance(value, int):
            self.__money = value

t = Money()
t.setMoney(200)
print(t.getMoney())
```

（2）使用property升级getter和setter方法：

```
class Money(object):
    def __init__(self):
        self.__money = 0

    def getMoney(self):
        return self.__money

    def setMoney(self, value):
        if isinstance(value, int):
            self.__money = value
    money = property(getMoney, setMoney)

t = Money()
t.money = 200
print(t.money)
```

（3）使用property的装饰器方法

```
class Money(object):
    def __init__(self):
        self.__money = 0

    # 即getter方法
    @property
    def money(self):
        return self.__money

    #即setter方法
    @money.setter
    def money(self, value):
        if isinstance(value, int):
```

```
            self.__money = value

t = Money()
t.money = 200
print(t.money)
```

6、迭代

- 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Iterable类型
- 集合数据类型如set、list、tuple、dict等都是Iterable，但不是Iterator。可以通过iter()函数生成为Iterator。

（1）可以迭代（Iterable）：

一类是list，tuple，dict，set，“abcd”

一类是generate function或yield

（2）判断是否可以迭代

```
>>> from collections import Iterator
>>> isinstance([11,22,33], Iterator)
False
>>> isinstance([i for i in range(10)], Iterator)
True
```

（3）iter()函数

```
>>> from collections import Iterator
>>> isinstance(iter([11,22,33]), Iterator)
True

# example
>>> b = [11,22,33]
>>> c = iter(b)
>>> next(c)
11
```