尚观培训 --- 24天精通Python运维自动化

`python`

该教材针对运维自动化开发定制

第一周课程

Python简介

1.1 Python是什么及Python的特点

1.1.1 Python是一门解释型、可交互和面向对象的优雅而健壮的编程语言，只有你想不到，没有Python做不到。

> - Python 是一种解释型语言：程序不需要编译，程序在运行时才翻译成机器语言，对象类型和内存占用都是在运行时确定的。
> - Python 是交互式语言： 你可以在Python shell中直接交互执行你的代码
> - Python 是面向对象语言：Python支持面向对象编程技术
> - 容易学习： 关键字少、结构简单、语法清晰。对于初学者可以在短时间内轻松上手
> - 代码可读性强： Python语法简洁，没有像其他语言的命令式符号。例如：美元符号（$）、分号（;）、方向箭头(->)等。python代码是相当容易理解的。
> - 可移植性：因为Python是C写的，C的可移植性使得Python可以运行在许多平台。
> - 可扩展：可以使用C、C++、Java、C#编写Python的扩展模块

1.2 安装Python

1.2.1 基于Unix的系统已经安装了Python,建议童鞋们使用Ubuntu、Mac OS系统作为自己的

开发系统
检查系统Python是否安装，直接通过命令行运行python, 查看它是否在搜索路径中而且运行正常：

```
[bigv@MacV ~ ]$ python
Python 2.7.13 (default, Apr  4 2017, 08:47:57)
[GCC 4.2.1 Compatible Apple LLVM 8.1.0 (clang-802.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

如上所示">>>"是解释器提示符，您已经启动了命令行上的交互式解释器，并等待你输入python命令。

1.3 运行Python

1.3.1 交互式解释器运行

```
[bigv@MacV ~ ]$ python
Python 2.7.13 (default, Apr  4 2017, 08:47:57)
[GCC 4.2.1 Compatible Apple LLVM 8.1.0 (clang-802.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print "hello BigV"
hello BigV
>>>
```

1.3.3 命令行运行Python。咱们先来看一段Python代码 hello.py

```
[bigv@MacV ~ ]$ cat hello.py
#!/usr/bin/env python

print "hello BigV"

[bigv@MacV ~ ]$ python hello.py
hello BigV
```

Python基础

2.1 Python语法

2.1.1 注释： 单行注释以#字符开始;多行注释''' 注释内容 '''

```
# 这是单行注释，以#开头后面的内容会被python解释器忽略
'''
    三对单引号，这是多行注释
    三对单引号，这是多行注释
'''
"""
    三对双引号，这是多行注释
    三对双引号，这是多行注释
"""
```

代码中存在中文字符注释要在python文件中加上 #coding:utf-8 或 #-- *coding:utf-8* --
2.1.2 反斜杠() 继续上一行, 另一种跨行的语法是使用闭合操作符，例如：小括号、中括号、花括号。

```
# 反斜杠跨行
if name == "BigV" and \
    age == "18":
    print "Hello BigV"
# 闭合操作符跨行
num, size, width, height = (1, 2,
3, 4)
```

2.1.3 冒号：将代码块的头和体分开

```
def hello(who):
    print "hello %s"%who
```

2.1.4 代码缩进： 推荐使用4个空格的宽度，避免使用tab制表符，因为不同的文本编译器中制表符的空白宽度不一， 必须严格控制缩进宽度，不同缩进执行时解释器会报错。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
# 文件名：test.py

 if True:
```

```
    print "True"
    print "False"
else:
    print "True"
    # 没有严格缩进，在执行时会报错
  print "False"

[bigv@MacV ~ ]$ python test.py
  File "test.py", line 6
    print "F"
            ^
IndentationError: unindent does not match any outer indentation level
```

2.2 变量赋值

2.2.1 变量的赋值使用（ = ）操作符, 变量在赋值时自动声明， ***变量赋值是将该对象的引用赋值给变量。***

```
name = "BigV"
age = 25
host, port = "127.0.0.1", 3306    # 多元赋值
```

2.3 标识符

2.3.1 python中的命名规范 即合法的标识符

> - 标识符不能以数字开头
> - 第一个字符必须是以字母或下划线开头
> - 不能包含除下划线以外的符号
> - 区分大小写
> - 不能使用保留关键字
> - _abc 类中的私有变量
> - __abc__有特殊含义的变量，普通变量避免使用这种命名风格

2.4 Python中的关键字

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

2.5 Python数据类型及内建函数

2.5.1 python数据类型分为数字类型及非数字类型
非数字类型：
字符串(string)
列表(list)
元祖(tuple相当于只读列表)
字典(dict)
数字类型包括：
整型（Integer）
长整型(Long Integer)
浮点型（float）
复数（complex）

布尔型(True | False)
其他内建类型
Null/NoneType 它的值是None,布尔值总是False

> 重要提示：
> 每个对象都具有布尔True或False值，空对象、值为零的任何数字或者Null对象 None的布尔值都是False. 例如:空列表、空元祖、空字典、空字符串，所有值为0的数

2.5.2 内建函数

函数	功能
cmp(x, y)	比较x,y，如果x小于y返回负整数；如果x大于y返回正整数，如果相等返回0
str(x)	返回对象的字符串表示,追求可读性
repr(x)	返回对象的字符串表示,追求明确性
type(x)	得到一个对象的类型
isinstance(x, type)	参数是元祖，判断x的类型返回布尔值
int(x[,base])	将x转换为一个整数
long(x)	将x转换为一个长整型数
float(x)	将x转换为浮点数
tuple(x)	将x转换为元祖
list(x)	将x转换为列表
dict(x)	将x转换为字典，可传入**kwargs或iterable

```
In [43]: dict(a="1") #关键字
Out[43]: {'a': '1'}

In [44]: dict([(1, 'a'), (2, 'b'), (3, 'c')])    #可迭代对象
Out[44]: {1: 'a', 2: 'b', 3: 'c'
```

2.6 运算符 (x=5,y=25)

算数运算符

运算符	描述	示例
+	两个对象相加	x+y = 30
-	两个对象相减	x-y = -20
*	两个对象相乘	x*y = 125

运算符	描述	示例
/	两个对象相除	y/x = 5
%	取模-返回余数	y%x = 2
**	返回a的b次幂	x**y = 4
//	取整除,返回商的整数部分	x+y = 2

比较运算符

运算符	描述	示例
==	比较两个对象是否相等	x == y 返回布尔类型False
!=	比较两个对象是否不相等	x != y返回true
<>	比较两个对象是否不相	x <> y返回ture,与!=操作符类似
>	大于返回x是否大于y	x > y返回False
<	小于返回x是否小于y	x < y返回true
>=	大于等于返回x是否大于等于y	a>=b返回False
<=	小于等于返回x是否小于等于y	a<=b返回true

赋值运算符

运算符	描述	示例
=	赋值运算符	name='BigV'将字符串BigV赋值给name变量
+=	加法赋值运算符	y += x等效为y = y+x
-=	减法赋值运算符	y -= x等效为y = y - x
*=	乘法赋值运算符	y *= x等效为y = y * x
/=	除法赋值运算符	y /= x等效为y = y / x
%=	取模赋值运算符	y %= x等效为y = y % x
**=	幂赋值运算符	y **= x等效为y = y ** x

运算符	描述	示例
//=	取整除赋值运算符	y //= x等效为y = y // x

逻辑运算符

运算符	描述	示例
and	布尔与 布尔"与" - 如果 x 为 False，x and y 返回 False，否则它返回 y 的计算值	x and y 返回 5
or	布尔"或" - 如果 x 是非 0，它返回 x 的值，否则它返回 y 的计算值	x or y 返回 25
not	布尔"非" - 如果 x 为 True，返回 False 。如果 x 为 False，它返回 True	not a返回 False

位运算符

运算符	描述	示例
&	位与运算符，参与运算的两个二进制位相应的位都为1则结果为1，否则为0	x & y的值为1
|位或运算符，参与运算的两个二进制位位有一个为1，结果就为1	x|y的值为29	
^	位异或运算符，参与运算的两个数的二进制位不相同时，结果为1	x^y的值为28
~	按位取反运算符	~x的值为-6
<<	左移运算符,运算数的各二进位全部左移若干位	x << 2的值为20
>>	右移运算符	x>>2的值为1

成员运算符

运算符	描述	示例
in	如果在指定的序列中找到值返回 True，否则返回 False	x 在 y 序列中，如果 x 在 y 序列中返回 True
not in	如果在指定的序列中没有找到值返回 True，否则返回 False	x 不在 y 序列中，如果 x 不在 y 序列中返回 True

身份运算符

运算符	描述	示例
is	is 是判断两个标识符是不是引用自一个对象	x is y, 类似 id(x) == id(y)，如果引用的是同一个对象则返回 True，否则返回 False
is not	is not 是判断两个标识符是不是引用自不同对象	x is not y ， 类似 id(a) != id(b)。如果引用的不是同一个对象则返回结果 True，否则返回 False

第二周课程

序列

序列的定义：由一些成员组成，并且可以通过下标偏移量访问到一个或者几个成员

3.1 字符串

3.1.1 字符串的创建，通过单引号或双引号

```
>>> name = "BigV"
>>> age = '25'
```

3.1.2 字符串的访问

```
>>> name            #通过变量名直接访问
'BigV'
>>> print name      #直接打印变量名
BigV
>>> name[0]         #通过索引访问
'B'
>>> name[0:3]       #通过切片访问
'Big'
>>> name[:4]        #通过切片访问
'BigV'
```

3.1.3 改变字符串

```
>>> name = "BigV"
>>> name = "BigV" + " Yang"
>>> name
'BigV Yang'
```

3.1.4 删除字符串

```
>>> del name    #Python中没必要显示删除，程序结束会自动释放
```

3.1.5 字符串内建函数

方法	描述
string.count(str)	返回字符串出现str的次数
string.join(seq)	以string作为分隔符，将seq中的元素组成一个新的字符串
sting.endswith(obj, beg=0, end=len(string))	检查字符串是否以obj结束
string.startwith(obj, beg=0, end=len(string))	检查字符串是否以obj开头
string.split(str, num)	以字符串str作为分割符，如果指定num，则只分割num个字符串
string.lstrip()	去除字符串左边空格

方法	描述
string.rstrip()	去除字符串右边空格
string.strip()	去除字符串前后空格
string.upper()	转换字符串中小写字母为大写字母
string.lower()	转换字符串中大写字母为小写字母
string.isdigit()	如果string只包含数字返回True,否则返回False

3.1.6 三引号： 用单引号或双引号定义字符串，如果你要创建的字符串中包含换行符、制表符以及其他特殊字符时就不那么方便了，三引号允许一个字符串跨多行并可以包含换行符、制表符等特殊字符。

3.1.7 字符串格式化
字符串格式化的2种方式：

```
In [23]: s = "My name is %s, age is %i, My book cost %0.2f"%("BigV", 25, 68.2)
In [24]: s
Out[24]: 'My name is BigV, age is 25, My book cost 68.20'

n [25]: s = "My name is {}, age is {}, My book cost {}".format("BigV", 25, 68.2)

In [26]: s
Out[26]: 'My name is BigV, age is 25, My book cost 68.2'

In [27]: s = "My name is {name}, age is {age}, My book cost {cost}".format(name="BigV", age=25, cost=68.2)

In [28]: s
```

3.2 列表

3.2.1 列表的创建 列表由方括号[]定义

```
>>> aList = [1, "BigV", 3.2, ["nginx", "apache"]]
>>> print aList
```

```
[1, 'BigV', 3.2, ['nginx', 'apache']]
```

3.2.2列表的访问， 与操作字符串类似使用切片操作符[]和索引值或值范围一起使用

```
>>> aList[1]
'BigV'
>>> aList[1:3]
['BigV', 3.2]
>>> aList[3][1]
'apache'
```

3.2.3 更新列表

```
>>> aList[1] = 18
>>> aList[1]
18
>>> aList.append("os")
>>> aList
[1, 18, 3.2, ['nginx', 'apache'], 'os']

>>> bList = ["dns", "web"]
>>> aList + bList
[['nginx', 'apache'], 3.2, 1, 'dns', 'web']
```

3.2.4 列表元素删除

```
>>> aList.remove("os")
>>> aList
[1, 18, 3.2, ['nginx', 'apache']]
>>> aList.pop(1)
18
```

3.2.5 列表内建函数

方法	描述
list.append(obj)	向列表中追加一个对象obj
list.count(obj)	返回对象obj在列表中出现的次数

方法	描述
list.extend(seq)	向列表中添加序列seq
list.insert(index,obj)	在索引Index位置插入obj
list.pop(index)	删除指定index索引位置的对象，默认是最后一个
list.remove(obj)	删除对象obj
list.reverse()	翻转列表
list.sort(func=None, key=None, reverse=False)	对列表排序

3.2.6 内建函数

|reversed(list)|翻转列表|

|sorted(list)|对列表排序|

|len()|返回列表长度|

|max()|返回列表最大值|

|min()|返回列表最小值|

|sum()|返回列表元素的和|

3.2.7 列表解析-动态创建列表

```
>>> aList = [1,2,3,4,5,6,7,8]
>>> [i for i in aList if i%2 == 0]
[2, 4, 6, 8]
```

> 重要提示：
> list.sort()和list.reverse()是list内置的排序方法，原列表会内容会改变，不返回新对象。
> sorted()和reversed()是Python内置的排序方法，原列表会保留，返回一个新对象

3.3 元组

3.3.1 元组的创建 元组使用()定义，是一种不可变类型

```
>>> aTuple = (1, "BigV", 3.2, ["nginx", "apache"])
>>> aTuple
(1, 'BigV', 3.2, ['nginx', 'apache'])
```

3.3.2 访问元组的值

```
>>> aTuple[1]
'BigV'
>>> aTuple[1:3]
('BigV', 3.2)
>>> aTuple[3][0]
'nginx'
```

3.3.3 更新元组

```
>>> aTuple = aTuple + ("python", "go")
aTuple
(1, 'BigV', 3.2, ['nginx', 'apache'], 'ddd', 'ccc', 'python', 'go')
```

> 重要提示：
> 1. 字符串、列表、元组都属于序列
> 2. 列表是有序的，可变的
> 3. 元组是有序的，不可变
> 4. 它们的共同操作符及内建函数：

描述	方法
成员操作符	in/not in
连接操作符	+
重复操作符	*
切片操作	[]/[:]
算数操作符	< > ==
内置函数	len()、max()、min()、sorted()、reversed()、tuple()、list()

5.元组对象本身不可变，但是元组包含的可变对象是可以变得：

```
>>> aList
[['nginx', 'apache'], 3.2, 1, 'dns', 'web', 'hello', 'world']
>>> aList[0][1] = "python"
>>> aList
[['nginx', 'python'], 3.2, 1, 'dns', 'web', 'hello', 'world']
```

6.深拷贝与浅拷贝
. 变量赋值是把值的地址引用赋值给变量
. 不可变对象（str/tuple），修改时会分配新的地址空间
. 可变对象(list),修改时不需要分配新的空间，修改的原地址空间内容

```
###浅拷贝
>>> aList = ["python",["nginx", "apache"]]
>>> aList
['python', ['nginx', 'apache']]
>>> bList = aList[:]
>>> [id(i) for i in aList]
[4481419952, 4482708040]
>>> [id(i) for i in bList]
[4481419952, 4482708040]
>>> aList[0] = "Django"
>>> aList
['Django', ['nginx', 'apache']]
>>> bList
['python', ['nginx', 'apache']]
>>> aList[1][0] = "C#"
>>> aList
['Django', ['C#', 'apache']]
>>> bList
['python', ['C#', 'apache']]
>>>
###深拷贝
>>> from copy import deepcopy
>>> aList = ["python",["nginx", "apache"]]
>>> bList = deepcopy(aList)
>>> bList
['python', ['nginx', 'apache']]
>>> [id(i) for i in aList]
[4481419952, 4482808432]
```

```
>>> [id(i) for i in bList]
[4481419952, 4482809728]
>>> aList[0] = "Django"
>>> aList
['Django', ['nginx', 'apache']]
>>> bList
['python', ['nginx', 'apache']]
>>> aList[1][0] = "C#"
>>> aList
['Django', ['C#', 'apache']]
>>> bList
['python', ['nginx', 'apache']]
```

字典

4.1 创建字典 使用{}定义字典，字典是以key-value键值对存在的

```
>>> dDict = {}
>>> dDict1 = {"name":"BigV", "age":"25"}
```

4.2 访问字典的值

```
#获取单个值
>>> dDict1 = {"name":"BigV", "age":"25"}
>>> dDict1["name"]
'BigV'
#遍历所有一
>>> for key in dDict1.keys():
...     print "Key=%s, value=%s"%(key,dDict1[key])
...
Key=age, value=25
Key=name, value=BigV
#遍历所有二
>>> for k, v in dDict1.items():
...     print "Key=%s, value=%s"%(k,v)
...
Key=age, value=25
Key=name, value=BigV
```

4.3 修改字典的值

```
>>> dDict1['age'] = 18
>>> dDict1
{'age': 18, 'name': 'BigV'}

>>> dDict2 = {"book1":"python", "book2":"Django"}
>>> dDict1.update(dDict2)
>>> dDict1
{'book2': 'Django', 'book1': 'python', 'age': 18, 'name': 'BigV'}
```

4.4 删除字典元素

```
>>> dDict1.pop("name")
```

4.5 字典的内建方法

方法	描述
dict.keys()	返回字典的所有keys的列表
dict.values()	返回字典的所有values列表
dict.get(key,default=None)	获取字典中key的值，如果不存在返回default设置的值
dict.items()	返回一个键值对的元组列表
dict.update(dict2)	将dict2添加到dict中

> 重要提示：
> 1.字典是以键值对的形式存在
> 2.字典的key是唯一的
> 3.字典存储是无序的

条件和循环

4.1 if语句

if语句由三部分组成：关键字；用于条件判断真假的表达式；执行的代码块。

语句语法：

```
    if expression:
        body
```

4.1.1 多重条件表达式

```
flag = False
a = 0
if not flag and a == 0:
    print "条件成立执行代码块"
```

4.1.2 else 语句

```
if expression:
    #为真执行这里的代码块
else：
    #为假执行这里的代码块
```

4.1.3 else-if语句 如果检查多个条件表达式是否为真，采用else-if语句

```
if expression:
    #如果条件为真执行这里的代码块
elif expression:
    #如果条件为真执行这里的代码块
elif expression:
    #如果条件为真执行这里的代码块
else:
    #如果条件为假执行这里的代码块
```

4.2 while语句

while循环会一直执行，直到条件不为真
语法：

```
while expression:
    #执行代码块，直到expression为假
num = 10
```

```
while num>0:
    print "执行次数为：{} 次".format(num)
    num -= 1
```

4.3 for语句

for循环会访问一个可迭代对象，知道所有元素处理完结束循环
语法：

```
for var in iterable:
    #执行代码块，知道所有元素处理完后退出循环

#通过值迭代
aList = ["python", "django", "flask", "mysql"]
for book in aList:
    print book
#通过索引迭代
for index in range(len(aList)):
    print "Book is ",aList[index]
#通过索引和值迭代
for index, value in enumerate(aList):
    print "%d %s"%(index+1, value)
```

> 重要提示：
> xrange()和range()的区别
> range()生成的是一个序列，全部加载到内存在
> xrange()生成的而是一个生成器，在每次调用的时候返回一个值

4.4 break和continue语句

break语句用于结束当前循环跳转到下条语句
continue语句结束当前循环，忽略下面的语句，开始下一次迭代

练习题：

> 1.输出 1-100 内的所有奇数和偶数
> 2.猜数字：随机选择一个数字作为答案。用户输入一个数字，程序会提示大了或是小了，

> 直到用户猜对
> 3.FizzBuzz:：遍历并打印0到100，如果数字能被3整除，显示Fizz；如果数字能被5整除，显示Buzz；如果能同时被3和5整除，就显示FizzBuzz。结果应该类似：
> 0,1,2，Fizz，4，Buzz，6……14，FizzBuzz，16……
> 4.生成16位用户激活码 random string
> 5.用户登录注册：1.注册用户 2 登录输入用户名密码 3:认证成功系那是欢迎信息 4:输错三次后锁定 5.登录成功后可修改密码
> 6. 判断学生成绩，90-100为A，89-80为B，79-70为C,69-60为D，60分以下为E

文件I/O操作

5.1 文件内建函数

5.1.1 open()函数用于操作I/O操作的通用接口，打开文件成功后会返回一个文件对象，否则触发IOError异常

> 语法：
> fobj = open(filename, access_mode="r", buffering=-1)
> 参数：
> filename：要打开的文件名称，相对路径或绝对路径
> access_mode: 代表文件的打开模式。r,w,a代表读取，写入，追加；a+,w+,r+代表以读写模式访问文件
> buffering: 代表访问文件的缓冲方式，0表示不缓冲，1表示缓冲一行，大于1代表缓冲给定的大小

5.1.2 一般采用with语句，它会创建一个上下文环境，当离开with语句块后，文件将自动关闭,如果不适用with语句，请确保手动关闭文件。

```
with open(filename, access_mode="r+") as f:
    f.readlines()
```

讨论： 如果文件已经存在,但是又不想覆盖原文件怎么办

文件对象方法	描述
file.next()	返回文件下一行
file.tell()	返回文件的当前位置，即文件指针当前位置。
file.flush()	刷新文件内部缓冲区
file.read(size)	从文件读取size字节，当size=-1读取所有字节
file.readline()	从文件读取并返回一行
file.readlines()	返回文件所有行并返回一个列表
file.write	与read相反把文本数据写入到文件中
file.writelines	接受一个字符串列表，并写入到文件
file.seek(off, where)	文件移动指针，where（0-文件起始位置，1-当前位置，2-文件末尾），off代表偏移量

重要提示：

> - read()和readlines()从文件读取时并不会删除行结束符，类似write()和writelines()也不会自动加入结束符,需要程序员自己处理。

```
>>> f = open("host.txt", 'r')
>>> dir(f)
['__class__', '__delattr__', '__doc__', '__enter__', '__exit__',
'__format__', '__getattribute__', '__hash__', '__init__', '__iter__', '
__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__
sizeof__', '__str__', '__subclasshook__', 'close', 'closed', 'encoding'
, 'errors', 'fileno', 'flush', 'isatty', 'mode', 'name', 'newlines', 'ne
xt', 'read', 'readinto', 'readline', 'readlines', 'seek', 'softspace', '
tell', 'truncate', 'write', 'writelines', 'xreadlines']
>>> f = open("host.txt","w+")
>>> f.tell()
0
>>> f.write("1.1.1.1")
>>> f.tell()
7
>>> f.write("2.2.2.2")
>>> f.readlines()
```

```
[]
>>> f.tell()
14
>>> f.seek(1,0)
>>> f.readlines()
['.1.1.12.2.2.2']
>>> f.writelines(["a","b"])
>>> f.readlines()
[]
>>> f.seek(1,0)
>>> f.readlines()
['.1.1.12.2.2.2ab']
>>> f.writelines(["a\n","b\n"])
>>> f.seek(1,0)
>>> f.readlines()
['.1.1.12.2.2.2aba\n', 'b\n']
```

练习题：

> 1. 敏感词文本文件 filtered_words.txt，当用户输入敏感词语，则用 星号 * 替换，例如当用户输入「这哥们玩的真垃圾」，则变成「这哥们玩的真**」
> 2. 有个目录，里面是你自己写过的程序，统计一下你写过多少行代码。包括空行和注释，但是要分别列出来。 (或获取目录下每个文件大小)

第三周课程

函数

6.1 什么是函数

函数是对程序逻辑进行结构化，将重复的代码组织一起，提高程序的模块性和代码的重复利用率。

6.1.1 函数的定义与声明

```
def func_name():
```

```
    'this is a func'
    #do something
    return expression
```

1. 函数以def关键字开头，跟上函数名及函数操作符()
2. 函数可以接受参数，必须放在()括号中
3. 函数体第一行为函数添加说明
4. 函数定义以冒号结尾
5. 函数的结束，以return结束返回给调用者返回值，没有return语句默认返回None

函数的调用

6.1.2.1 无参函数调用

```
>>> def myFunc():
...     'this is a func'
...     print "hello BigV"
...     return
...
>>> type(myFunc)
<type 'function'>
>>> myFunc()
hello BigV
>>> print myFunc()
hello BigV
None
>>> m = myFunc()
hello BigV
>>> print m
None
```

> 调用方式：[变量 =]函数名()

有参函数调用

6.2.1.1 位置参数

```
def get_conn(host, port):
    return "http://{}:{}".format(host,port)
get_conn("127.0.0.1", 80)
```

6.2.1.2 关键参数

```
def get_conn(host, port):
    return "http://{}:{}".format(host,port)
get_conn( port=80,  host="127.0.0.1")
```

> 重要提示：
> 位置参数需要注意参数传递的顺序
> 关键字参数不需要注意参数的顺序，

6.2.1.3 默认参数

可以为参数定义默认值，如果该参数有值传入，则覆盖默认值

```
def get_conn(host, port=8080):
    return "http://{}:{}".format(host,port)
get_conn("127.0.0.1",port=80)
```

6.3.1 可变长参数

6.3.1.1 星号（*）

指定元组作为非关键字参数, 可变长元组必须在位置参数和默认参数之后

```
In [3]: def tupleArgs(arg1, arg2=10, *argTuple):
   ...:     print arg1
   ...:     print arg2
   ...:     print argTuple
   ...:

In [4]: tupleArgs(1,2,3,4,5,6)
1
2
(3, 4, 5, 6)
```

6.3.1.2 双星号(**) 关键字变量参数

```
In [12]: def tupleArgs(arg1, arg2=10, *argTuple, **argDict):
    ...:     print arg1
    ...:     print arg2
    ...:     print argTuple
    ...:     print argDict
    ...:
    ...:

In [13]: tupleArgs(1, 2, 3,4, name="bigv",age="25")
1
2
(3, 4)
{'age': '25', 'name': 'bigv'}
```

6.4 匿名函数与lambda

6.4.1 匿名函数不需要使用def语句声明，通过lambda关键字创建匿名函数返回一个可调用的函数对象。
6.4.2 lambda语法
lambda [参数]: 表达式

```
In [6]: a = lambda x, y=2: x+y
In [7]: a(1,3)
Out[7]: 4
```

6.5 闭包

6.5.1 闭包的定义：如果一个内部函数里，对外部作用域的变量进行引用，那么内部函数就被认为是闭包，定义在外部函数内的但由内部函数引用或者使用的变量成为自由变量，即使外部函数生命周期结束了，单闭包所引用的自由变量仍会存在

```
In [65]: def counter(num=0):
    ...:     count = [num]
    ...:     def incr():
    ...:         count[0] += 1
    ...:         return count[0]
    ...:     return inc
In [66]: c = counter(5)
In [67]: c()
```

```
Out[67]: 6
In [68]: c()
Out[68]: 7
```

6.6 生成器

6.6.1 什么是生成器：生成器就是一个带yield语句的函数，生成器能暂停执行并返回结果，当生成器next()方法调用时，会从暂停的地方继续执行

```
In [21]: def gen():
    ...:     for i in range(10):
    ...:         yield i
    ...:

In [22]: a = gen()

In [23]: a.next()
Out[23]: 0

In [24]: a.next()
Out[24]: 1

In [25]: a.next()
Out[25]: 2
```

模块

7.1 什么是模块

模块是按照逻辑来组织python代码的方法，一个文件是一个模块，一个模块也可以被看做是一个文件。

模块名称空间

每个模块有自己的名称空间，也就是每个模块都有自己的属性与方法，模块之间的调用是通过Import导入操作，不同的模块不会出现名称交叉现象，例如a和b模块都有whoami()，通过模块名.whoami()指定各自的名称空间防止名称冲突。

7.2 搜索路径

```
In [16]: import a
ImportError: No module named b
In [17]: import sys

In [18]: sys.path
Out[18]:
['',
 '/usr/local/bin',
 '/usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python27.zip',
 '/usr/local/Cellar/python/2.7.13/Frameworks/Python.framework/Versions/2.7/lib/python2.7',
 ]
 In [19]:
sys.path.append('/Users/bigv/yangdw/dev/daemon/python/training')

In [20]: import a

In [21]: a.whoami()
SuperMan
```

7.3 模块导入

7.3.1 import 语句
import mymodule1, mymodule2 可以在一行内导入多个模块

7.3.2 from-import 语句
可以在指定的模块中导入指定的模块属性
from mymodule import whoami
from mymodule import *

7.3.3 import as语句
如果一个模块的名称或模块中的属性名称过长可以通过as指定别名

7.3.4 包

常用内建函数及模块介绍

os

sys

commands

subprocess

> 管理子进程，通常用于系统调用
> 1. subprocess.Popen(args, bufsize=0, executable=None, stdin=None, stdout=None, stderr=None, preexec_fn=None, close_fds=False, shell=False, cwd=None, env=None, universal_newlines=False, startupinfo=None, creationflags=0)
> 2. Popen()函数默认不等待子进程完成，必须调用wait()方法或communicate()
> 3. 父进程管理子进程：child.poll() 检查子进程状态 child.kill()终止子进程 child.send_signal()向子进程发送信号
> 4. time & datetime
> time.time() 返回自1970.1.1 00:00:00以秒计算的偏移量，时间戳
> time.localtime() 返回以数组形式的time.struct_time类型
> strftime(format[,t])可以将struct_time,datetime.datetime类型自由转换成字符型

```
1.
2.  #timestamp 时间戳
3.  In [26]: import time
4.  In [27]: time.time()
5.  Out[27]: 1515220505.013005
6.  #时间tuple
7.  In [28]: time.localtime()
8.  Out[28]: time.struct_time(tm_year=2018, tm_mon=1, tm_mday=6, tm_hour=14
    , tm_min=36, tm_sec=3, tm_wday=5, tm_yday=6, tm_isdst=0)
9.  #datetime转string
10. In [22]: from datetime import datetime
11. In [23]: datetime.now().strftime("%Y-%m-%d %H:%M:%S")
12. Out[23]: '2018-01-06 14:32:32'
13. #string转datetime
14. In [25]: datetime.strptime("2014-12-31 18:20:10", "%Y-%m-%d %H:%M:%S")
15. Out[25]: datetime.datetime(2014, 12, 31, 18, 20, 10)
```

json

random
choise
paramiko模块

> paramiko实现了SSH协议，能够方便地与远程计算机交互

```
In [1]: import paramiko

In [2]: client = paramiko.SSHClient()

In [3]: client.set_missing_host_key_policy(paramiko.AutoAddPolicy())  #paramiko默认只允许连接know_hosts文件中的主机，这里关闭改功能

In [4]: client.connect(hostname="1.1.1.1", username="root", password="password")    #创建连接

In [6]: stdin, stdout, stderr = client.exec_command("ls /home")  #远程执行sell命令

In [7]: stdout.readlines()  #读取输出
Out[7]: [u'a.txt\n', u'host.txt\n', u'uftp\n', u'yangdawei\n']

#传输文件
In [27]: sftp = paramiko.SFTPClient.from_transport(s.get_transport())

In [28]: sftp = s.open_sftp()
In [31]: sftp.put("host.txt", "/home/host.txt")  #put上传，get下载
Out[31]: <SFTPAttributes: [ size=20 uid=0 gid=0 mode=0100644 atime=1514056059 mtime=1514056059 ]>
```

面向对象编程

8.1 类和实例

在OOP编程中最重要的2个概念就是类(Calss)和（Instance）实例，类是一类对象的定义，实例则是根据类创建出来的一个个具体的对象。

```
In [36]: class People(object):                    #定义一个类
```

```
    ...:     def __init__(self, name, age):      #定义构造器
    ...:         self.name = name                设置实例属性name
    ...:         self.age = age
    ...:     def talk(self):                     #定义实例方法
    ...:         print "My name is %s, I'm %s years old!"%(self.name,
self.age)
    ...:

In [37]: mike = People("mike", "18")             #创建一个实例mike

In [38]: mike
Out[38]: <__main__.People at 0x10f0b0c10>

In [39]: People
Out[39]: __main__.People

In [40]: mike.name                               #访问实例属性name
Out[40]: 'mike'

In [41]: mike.talk()                             #访问实例方法talk
My name is mike, I'm 18 years old!
```

8.2 类属性

8.2.1 什么是类属性：

> *类属性就是属于一个对象的数据或者函数。
> * 通过.属性来访问。
> * 类属性（静态属性）是与类绑定一起的，与实例无关。

```
In [13]: class People(object):
    ...:     name = "BigV"
    ...:     age = 25
    ...:

In [14]: People.name
Out[14]: 'BigV'

In [15]: People.age
Out[15]: 25
```

使用dir()查看类属性或使用__dict__访问类的字典属性

8.2.2 类的特殊属性

属性	描述
__name__	类的名字
__doc__	类的文档字符串
__bases__	类的所有父类构成的元组
__dict__	类的字典属性
__module__	类所在的模块
__class__	实例对应的类

8.3 实例属性

> - 属于某个实例的属性
> - 使用句点标识符访问
> - 一般在__init__构造器中声明并初始化

```
In [49]: class Factory(object):
    ...:     def __init__(self, address, phone, name="car"):
    ...:         self.address = address
    ...:         self.phone = phone
    ...:         self.name = name
    ...:     def product(self, day):
    ...:         return "Factory product {} is cost {} day".format(this.
name, day)
In [54]: fact = Factory("bj","110","gun")  #实例化一个实例

In [55]: fact.name
Out[55]: 'gun'
In [56]: fact.__dict__
Out[56]: {'address': 'bj', 'name': 'car', 'phone': '110'}
```

8.4 类属性vs实例属性

8.4.1 区别：

- 类属性可以通过类对象访问
- 如果实例没有同名属性也可以通过实例访问类属性
- 通过类对象来修改
- 如果通过实例对象修改类对象，实际是创建了一个实例属性覆盖了类属性
- 类属性的修改会影响到所有实例
- 方法属于类的属性而不是实例属性
- 实例属性与类属性同名，则会覆盖类属性
- 方法只有类的实例才能被调用，也就是方法必须和实例绑定

8.4.2 方法中的self代表什么
self其实就是实例，必须在方法中传入self，代表实例与方法的绑定

8.4 静态方法

- 通过@staticmethod装饰器标识一个静态方法
- 静态方法的参数不是非必须的

```
In [32]: class Father(object):
    ...:     def __init__(self, name, age):
    ...:         self.name = name
    ...:         self.age = age
    ...:     @staticmethod
    ...:     def staticMethod():
    ...:         print "I'm static method"
```

8.5 类方法

- 通过@classmethod装饰器标识一个类方法
- 第一个参数传入类，一般用cls

```
In [41]: class Father(object):
    ...:     def __init__(self, name, age):
    ...:         self.name = name
    ...:         self.age = age
    ...:     @classmethod
    ...:     def classMethod(cls):
    ...:         print "I'm class Method"
```

8.6 静态方法vs类方法

> - 静态方法参数没有要求；类方法默认传递类
> - 静态方法无法访问类属性，实例属性； 类方法可以访问类属性，无法访问实例属性
> - 类和实例都可以调用

8.7 继承

8.7.1 继承描述的是基类也就是父类的属性传递给子类，继承是面向对象编程中实现代码重用的机制。

```
In [67]: class Parent(object):     #定义基类（父类）
    ...:     pass
    ...:

In [68]: class Child(Parent):      #定义子类
    ...:     pass
    ...:
In [69]: c = Child()               #实例化一个子类
In [70]: Child.__bases__           #子类的父类
Out[70]: (__main__.P,)
```

继承分为单继承和多继承
单继承：一个类只有一个直接父类
多继承：一个类有多个直接父类

8.8 封装

8.8.1 封装就是把类的属性和方法私有化(隐藏)，外部无法直接调用，调用者不必关心具体实现细节，而只要通过提供对外的接口，程序通过对外接口来访问或修改内部属性。
8.8.2 实现方式
双下划线（__）由双下划线开头的属性或方法不能被直接访问或修改
单下划线(_) 由单下划线开头的属性或方法不能使用from import 导入

第四周课程

正则表达式

网络编程

第五周课程

并发编程

11.1 多线程

Python中实现线程有2种方式：函数或类实例的方式
11.1.1 python的threading模块
python中可以使用thread和threading模块创建线程对象，thread模块提供了基本的线程和锁定支持，而threading模块则提供了功能更全面的线程管理， 所以避免使用thread模块，请把它遗忘把。
11.1.1.1 threading.Thread模块构造参数

> - class threading.Thread(group=None, target=None, name=None, args=(), kwargs={})¶
> - target 可回调的目标方法
> - args 可回调目标函数的参数元组()

- kwargs 可回调目标函数的字典关键字参数
- name 线程的名字，默认Thread-N
- 派生的子类必须实现threading.Thread.**init**(self)构造方法

11.1.1.2 threading.Thread模块方法

- start() 启动线程
- run() 派生子类需要重写父类的run()，当线程启动自动激活线程
- join() 阻塞主线程，直到线程正常结束
- setName() 设置线程名字
- getName() 获取线程名称
- daemon 标识线程为守护线程，必须在start()调用之前被设置，否则会抛出RuntimeError,该值默认继承自主线程，主线程daemon默认为False
- isDaemon() 线程是否为守护线程
- setDaemon() 设置线程为守护线程

```
# 函数方式的多线程实现
def counter(name, delay=10):
    ¦   time.sleep(delay)

if __name__ == "__main__":
    start = time.time()
    threads = []
    for i in range(10):
    ¦   t = threading.Thread(target=counter, args=("Mythread",),kwargs={"delay":1002}, name="mythread-%s"%i)
    ¦   t.start()
    ¦   threads.append(t)

    for i in threads:
    ¦   i.join()
    print "end cost %s s"%(time.time() - start)
```

```
#以类方式的多线程实现
class MyThread(threading.Thread):
    def __init__(self, name):
    ¦   threading.Thread.__init__(self)
```

```
        self.name = name
    def run(self):
        time.sleep(5)
        print "Thread-Name is :%s"%self.name

if __name__ == "__main__":
    for i in range(5):
        t = MyThread("Thread-%s"%i)
        t.start()
        print t.ident
```

11.1.2 线程锁

一般在某些特定的代码块不希望多个线程同时执行，比如修改数据库、更新文件等，通过锁保证同该时刻只有一个线程可以访问

> - threading.Lock() 互斥锁
> - acquire() 加锁
> - release() 释放锁

```
import threading
import time
num = 0

def counter(name):
    global num
    l.acquire()
    num += 1
    l.release()
    print "%s n %s:"%(name, num)
    time.sleep(1)


if __name__ == "__main__":
    l = threading.Lock()
    threads=[]
    for i in range(1000):
        t = threading.Thread(target=counter, args=("thread-%s"%i,))
        threads.append(t)
    for t in threads:
        t.start()
    for j in threads:
```

```
        |   j.join()
    print num
```

第六周课程

Mysql常用命令

6.1创建数据库
create database test;
grant all on test.* to user;
6.2. 查看表的字符集
SHOW CREATE TABLE tbl_name;
6.3. 查看字段字符集
SHOW FULL COLUMNS FROM tbl_name;
6.4. 修改字段字符集
ALTER TABLE srv CHANGE site site VARCHAR(100) CHARACTER SET utf8 COLLATE utf8_general_ci;
6.5. 修改表的字符集
ALTER TABLE logtest DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci;
6.6. 修改表和字段的字符集
ALTER TABLE logtest CONVERT TO CHARACTER SET utf8 COLLATE utf8_general_ci;
6.7. 选择要使用的数据库
use test;
6.8. 删除数据库
drop database test;
6.9. 创建表
create table users (login varchar(8), uid int, prid int);
6.10.删除表
drop table users;
6.11. 插入
insert into users values('bob', 110,1);
6.12.更新行

update users set prid=4 where uid=110;
6.13.删除行
delete from users where prid=4;
6.14. 清空表
delete from users;
6.15. 创建数据库并指定字符集
CREATE DATABASE flaskweb DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci;

Python操作Mysql

6.15

第七周课程

Flask Web

7.1 virtual env安装使用

7.1.1 安装

```
pip install virtualenv
```

7.1.2 使用

7.1.2.1 创建virtual环境

```
virtualenv project_name #会在当前目录下创建project_name新环境
```

7.1.2.2 使用虚拟环境

```
cd project_name && source ./bin/actiavte #激活虚拟环境
deactiavte # 退出虚拟环境
```

7.2.1 虚拟环境管理

7.2.1.1 安装vitrualenvwrapper扩展包

作用

> - 将所有虚拟环境整合在一个目录下,统一管理虚拟环境
> - 管理（新增，删除，复制）虚拟环境
> - 切换虚拟环境

```
pip install virtualenvwrapper
```

配置, 添加到~/.bashrc

```
export WORKON_HOME=$HOME/yangdw/dev/training/virtualenv    指定虚拟环境管理目录
source /usr/local/bin/virtualenvwrapper.sh
```

7.2.1.2 使用方法

> - 创建基本环境：mkvirtualenv [环境名]
> - 删除环境：rmvirtualenv [环境名]
> - 激活环境：workon [环境名]
> - 退出环境：deactivate
> - 列出所有环境：workon 或者 lsvirtualenv -b

安装flask

```
mkvirtualenv flaskweb
workon flaskweb
pip install flask
```

7.2 初识Flask

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

from flask import Flask
app = Flask(__name__)        #__name__参数用于决定项目的根目录

@app.route('/')              #路由
def index():                 #视图函数
    return "<h1>Hello World</h1>"

@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, %s</h1>'%name

if __name__ == '__main__':
    app.run(debug=True)      #启动web服务器
```

7.3 Flask请求调度

flask收到客户端发来的请求时，要找到处理该请求的视图函数，flask程序会根据路由中的url找到对应的映射视图函数。

```
In [1]: from helloworld import app

In [2]: app.url_map
Out[2]:
Map([<Rule '/' (HEAD, OPTIONS, GET) -> index>,
 <Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
 <Rule '/user/<name>' (HEAD, OPTIONS, GET) -> user>])
```

7.4 Flask响应

响应就是视图函数处理后的结果作为html页面返回给客户端

7.4.1 make_response模块

make_response可接收三个参数并返回一个response对象，可以在响应response对象上继续

调用各种方法设置响应。

```
from flask import Flask
from flask import make_response
app = Flask(__name__)

@app.route('/')
def index():
    response = make_response('<h1>Hello World</h1>')
    response.set_cookie('name', 'BigV')
    return response
```

7.4.2 redirect模块

redirect是重定向的特殊响应类型，这种响应没有页面内容，只告诉浏览器请求一个Location提供的新地址，状态码为302

```
from flask import Flask
from flask import redirect
app = Flask(__name__)

@app.route('/')
def index():
    return redirect("http://www.baidu.com")
```

7.5 程序上下文和请求上下文

变量名	上下文	说明
current_app	程序上下文	当前激活程序的程序实例
g	程序上下文	用于临时存储对象的全局变量，每次请求都会重置
request	请求上下文	请求对象，封装了客户端发出http请求的内容
session	请求上下文	用户会话，用于存储请求之间需要记住的值字典

7.6 请求钩子

请求钩子就是在请求开始处理之前或之后执行的函数，使用装饰器实现的4种钩子：

> - befrost_first_request 在处理第一个请求之前运行
> - befrost_request 在每次请求之前运行
> - after_request 在每次请求之后运行
> - teardown_request 即使有未处理的异常抛出，也在每次请求之后运行

请求钩子函数与视图函数之间共享的数据使用上下文全局变量g。例如：befrost_request来处理用户登录验证，从db读取到用户名后并保存早g.user中，在视图函数使用g.user获取用户。

7.7 Flask 模板

Flask模板是一个包含响应文本及占位符变量标识动态部分的文件，使用上下文中的变量替换占位符变量，这个过程称为渲染。 Flask使用Jinjia2模板引擎。

```
#首先需要在项目目录中创建templates目录及html模板文件
from flask import Flask
from flask import make_response, g, request, render_template
app = Flask(__name__)

@app.route('/')
def index():
    return render_template("index.html")

@app.route('/user/<name>')
```

```
def user(name):
    return render_template("user.html", name=name)

if __name__ == '__main__':
    app.run(debug=True)
```

7.7.1 模板变量

```
模板变量使用{{ name }}结构标识一个变量name
Jinjia2能标识所有类型的变量，列表，字典，对象。
```

7.7.2 过滤器

过滤器可以修改变量，{{name|lower}} 用哪个竖线分割{{变量|过滤器}}

过滤器名称	说明
safe	渲染时不转义
capitalize	把首字母转换成大写，其他小写
lower	转小写
upper	转大写
title	每个单词的首字母转成大写
trim	去除首尾空格
striptags	把值中的HTML标签都删掉

7.7.3 控制结构

```
# 条件控制
{% if user %}
    Hello, {{user}}
{% else %}
    dosomthing
{% endif %}
# 循环 动态生成表格
```

```
{% for i in user_list%}
    <li>{{i}}</li>
{% endfor %}
# 模板继承
{% extends 'base.html'%}
base.html
<html>
    <head>
        {% block head %}
        <title>{% block title %}{% endblock %} Flask App </title>
        {% endblock %}
    </head>
    <body>
        {% block body %}
        {% endblock %}
    </body>
</html>
```

7.8 Flask ORM

ORM(Object-Relational Mapper)对象关系映射，是对数据库的抽象

7.8.1 Flask数据管理框架

Flask-SQLAlchemy关系型数据库框架，支持多种db后台，即提供高层的ORM,也支持原生SQL。

```
pip install flask-sqlalchmy
```

数据库引擎	URL
MySQL	mysql://username:password@hostname/database
Postgres	postgresql://username:password@hostname/database
sqlite	sqlite:////absolute/path/to/database

7.8.2 配置数据库&定义模型及初始化

```
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
# 配置数据库
app.config['SQLALCHEMY_DATABASE_URI'] = \
    'mysql://root@localhost/flaskweb'
app.config['SQLALCHEMY_COMMIT_ON_TEARDOWN'] = True
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = True

db = SQLAlchemy(app)
bootstrap = Bootstrap(app)
moment = Moment(app)
# 定义模型
class Role(db.Model):
    __tablename__ = "roles"
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)
    users = db.relationship('User', backref='role')

    def __repr__(self):
    	return '<Role %r>' % self.name

class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username  = db.Column(db.String(64), unique=True, index=True)
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))

    def __repr__(self):
    	return '<User %r>'% self.username

# 一对多
表的外键由db.ForeignKey指定，传入的参数是表的字段。db.relations它声明的属性不作为表字段，第一个参数是关联类的名字，backref是一个反向身份的代理,相当于在User类中添加了role的属性。relations一般在（一）的一方添加。
```

7.8.3 数据库操作

8.8.3.1 创建表

```
from hello import db
db.create_all()
```

8.8.3.2 插入行

```
from hello import Role, User
admin_role = Role(name='admin')
user_role = Role(name='user')
user_join = User(username='join', role='admin_role')
user_bob = User(username='bob', role='user_role')
db.session.add_all([admin_role, user_role, user_join, user_bob])  #添加会话
db.session.commit()    #提交会话
```

8.8.3.3 查询

```
Role.query.all() 查询所有角色记录
User.query.filter_by(role=user_role).all()  查询角色为user的所有用户
Role.query.filter_by(name='user').all()     查询角色为user的
```

7.8.4 常用的列类型及对应的python类型

类型名	Python类型	说明
Integer	int	整数
Float	float	浮点数
String	str	字符串
Text	str	较长的字符串
Unicode	unicode	unicode字符串
Boolen	bool	布尔值
Date	datetime.date	日期
Time	datetime.time	时间
DateTime	datetime.datetime	日期和时间
Interval	datetime.timedelta	时间间隔
PickleType	任何Python对象	使用pickle序列化

7.8.4 常用列选项

选项名	说明
primary_key	主键
unique	唯一
index	添加索引，提升查询效率
nullable	列不允许为空