

# Python运维开发

NSD PYTHON

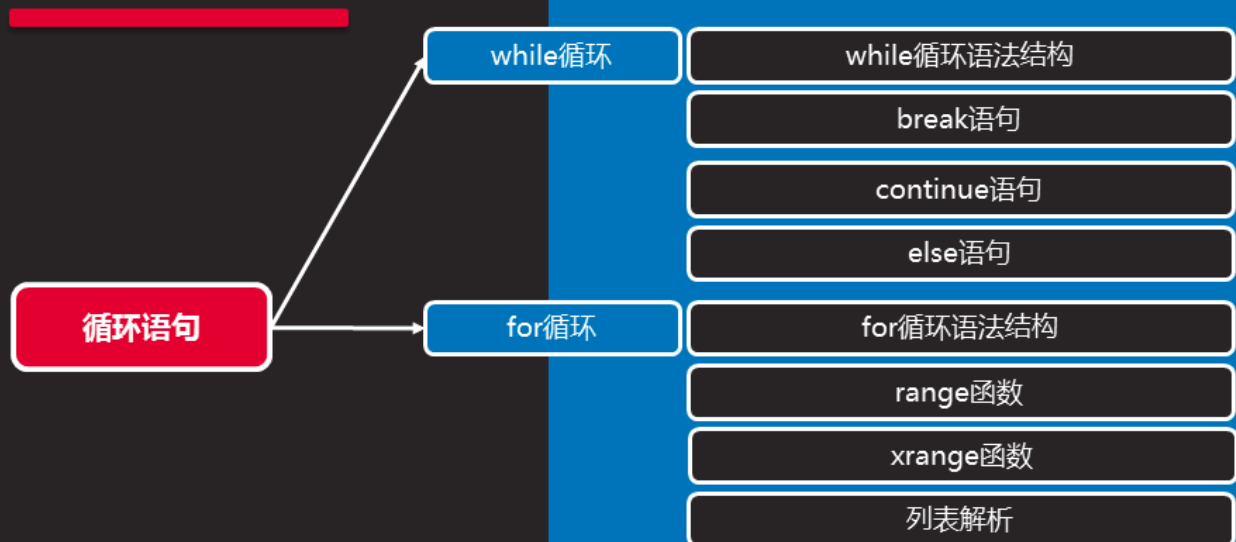
DAY02

# 内容

|    |               |         |
|----|---------------|---------|
| 上午 | 09:00 ~ 09:30 | 作业讲解和回顾 |
|    | 09:30 ~ 10:20 | 循环语句    |
|    | 10:30 ~ 11:20 |         |
|    | 11:30 ~ 12:20 |         |
| 下午 | 14:00 ~ 14:50 | 文件对象    |
|    | 15:00 ~ 15:50 | 函数      |
|    | 16:00 ~ 16:50 | 模块      |
|    | 17:00 ~ 17:30 | 总结和答疑   |



## 循环语句



# while循环

## while循环语法结构

- 当需要语句不断的重复执行时，可以使用while循环
- 语句while\_suite会被连续不断的循环执行，直到表达式的值变成0或False

```
#!/usr/bin/env python  
sum100 = 0  
counter = 1
```

```
while counter <= 100:  
    sum100 += counter  
    counter += 1
```

```
print "result is %d" % sum100
```

## break语句

知识讲解

- break语句可以结束当前循环然后跳转到下条语句
- 写程序的时候，应尽量避免重复的代码，在这种情况下可以使用while-break结构

```
name = raw_input('username: ')
while name != 'tom':
    name = raw_input('username: ')
#可以替换为
while True:
    name = raw_input('username: ')
    if name == 'tom':
        break
```



## continue语句

知识讲解

- 当遇到continue语句时，程序会终止当前循环，并忽略剩余的语句，然后回到循环的顶端
- 如果仍然满足循环条件，循环体内语句继续执行，否则退出循环

```
#!/usr/bin/env python
sum100 = 0
counter = 0
while counter <= 100:
    counter += 1
    if counter % 2:
        continue
    sum100 += counter
print "result is %d" % sum100
```



## else语句

知识讲解

- python中的while语句也支持else子句
- else子句只在循环完成后执行
- break语句也会跳过else块

```
#!/usr/bin/env python
```

```
sum10 = 0
```

```
i = 1
```

```
while i <= 10:
```

```
    sum10 += i
```

```
    i += 1
```

```
else:
```

```
    print sum10
```



## for循环



## for循环语法结构

知识讲解

- python中的for接受可迭代对象（例如序列或迭代器）作为其参数，每次迭代其中一个元素。  
`for iter_var in iterable:`  
    `suite_to_repeat`
- 与while循环一样，支持break、continue、else语句
- 一般情况下，循环次数未知采用while循环，循环次数已知，采用for循环



## range函数

知识讲解

- for循环常与range函数一起使用
- range函数提供循环条件
- range函数的完整语法为：  
    `range(start, end, step = 1)`

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(1, 5)
[1, 2, 3, 4]
>>> range(1, 5, 2)
[1, 3]
```



## xrange函数

知识讲解

- xrange()类似range()，不过当你有一个很大的范围列表时，xrange()可能更为适合，因为它不会在内存里创建列表的完整拷贝
- 它只被用在for循环中，在for循环外使用它没有意义
- 它的性能远高出range()，因为它不生成整个列表

```
>>> for x in xrange(3):  
...   print x  
0  
1  
2
```



## 案例1：斐波那契数列

课堂练习

1. 斐波那契数列就是某一个数，总是前两个数之和，比如0，1，1，2，3，5，8
2. 使用for循环和range函数编写一个程序，计算有10个数字的斐波那契数列
3. 改进程序，要求用户输入一个数字，可以生成用户需要长度的斐波那契数列



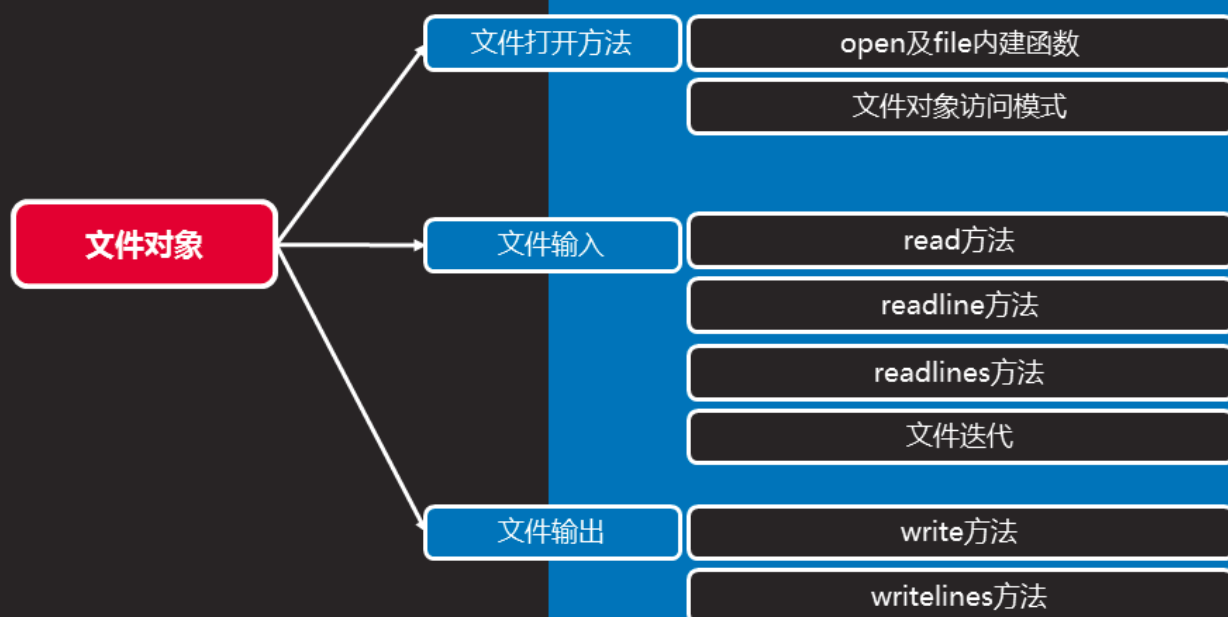
# 列表解析

知识讲解

- 它是一个非常有用、简单、而且灵活的工具，可以用来动态地创建列表
- 语法：  
`[expr for iter_var in iterable]`
- 这个语句的核心是for循环，它迭代iterable对象的所有条目
- expr应用于序列的每个成员，最后的结果值是该表达式产生的列表



## 文件对象





# 文件打开方法

## open及file内建函数

知识讲解

- 作为打开文件之门的“钥匙”，内建函数open()以及file()提供了初始化输入/输出（I/O）操作的通用接口
- 成功打开文件后时候会返回一个文件对象，否则引发一个错误
- open()方法和file()方法可以完全相互替换
- 基本语法：

```
file_object = open(file_name, access_mode='r', buffering=-1)
```



# 文件对象访问模式

知识讲解

| 文件模式 | 操 作                    |
|------|------------------------|
| r    | 以读方式打开（文件不存在则报错）       |
| w    | 以写方式打开（文件存在则清空，不存在则创建） |
| a    | 以追加模式打开（必要时创建新文件）      |
| r+   | 以读写模式打开（参见r）           |
| w+   | 以读写模式打开（参见w）           |
| a+   | 以读写模式打开（参见a）           |



# 文件输入

## read方法

知识讲解

- read()方法用来直接读取字节到字符串中，最多读取给定数目个字节
- 如果没有给定size参数（默认值为-1）或者size值为负，文件将被读取直至末尾

```
>>> data = fobj.read()  
>>> print data
```



## readline方法

知识讲解

- 读取打开文件的一行（读取下个行结束符之前的所有字节）
- 然后整行，包括行结束符，作为字符串返回
- 它也有一个可选的size参数，默认为-1，代表读至行结束符
- 如果提供了该参数，那么在超过size个字节后会返回不完整的行

```
>>> data = fobj.readline()  
>>> print data
```



## readlines方法

- readlines()方法读取所有（剩余的）行然后把它们作为一个字符串列表返回

```
>>> data = fobj.readlines()
>>> print data
```

知识讲解



## 文件迭代

- 如果需要逐行处理文件，可以结合for循环迭代文件
- 迭代文件的方法与处理其他序列类型的数据类似

```
>>> fobj = open('star.py')
>>> for eachLine in fobj:
...     print eachLine,
```

知识讲解



# 文件输出

## write方法

- write()内建方法功能与read()和readline()相反。它把含有文本数据或二进制数据块的字符串写入到文件中去
- 写入文件时，不会自动添加行结束标志，需要程序员手工输入

```
>>> fobj.write('Hello World!\n')
```

## writelines方法

知识讲解

- 和readlines()一样，writelines()方法是针对列表的操作
- 它接受一个字符串列表作为参数，将它们写入文件
- 行结束符并不会被自动加入，所以如果需要的话，必须在调用writelines()前给每行结尾加上行结束符

```
>>> fobj.writelines(['Hello World!\n', 'python programing\n'])
```



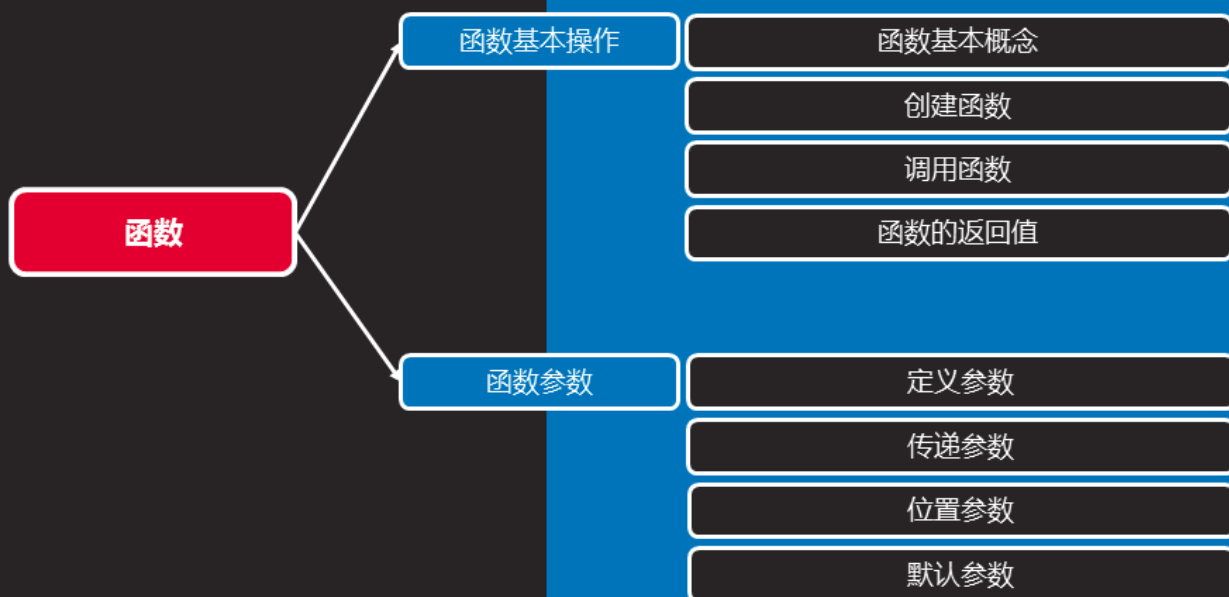
## 案例2：模拟cp操作

课堂练习

1. 创建~/bin/cp.py文件
2. 将/bin/ls “拷贝” 到/root/目录下
3. 要求读取/bin/ls后，每次读取4096字节，依次写入到新文件
4. 不要修改原始文件



# 函数



## 函数基本操作

## 函数基本概念

知识讲解

- 函数是对程序逻辑进行结构化或过程化的一种编程方法
- 将整块代码巧妙地隔离成易于管理的小块
- 把重复代码放到函数中而不是进行大量的拷贝，这样既能节省空间，也有助于保持一致性
- 通常函数都是用于实现某一种功能



## 创建函数

知识讲解

- 函数是用def语句来创建的，语法如下：  

```
def function_name(arguments):  
    "function_documentation_string"  
    function_body_suite
```
- 标题行由def关键字，函数的名字，以及参数的集合（如果有的话）组成
- def子句的剩余部分包括了一个虽然可选但是强烈推荐的文档字符串，和必需的函数体





## 调用函数

知识讲解

- 同大多数语言相同，python用一对圆括号调用函数
- 如果没有加圆括号，只是对函数的引用

```
>>> def foo():  
...     print 'hello'  
...  
>>> foo()  
hello  
>>> foo  
<function foo at 0x7ff2328967d0>
```



## 函数的返回值

知识讲解

- 多数情况下，函数并不直接输出数据，而是向调用者返回值
- 函数的返回值使用return关键字
- 没有return的话，函数默认返回None

```
>>> def foo():  
...     res = 3 + 4  
>>> i = foo()  
>>> print i  
None
```



# 函数参数

## 定义参数

- 形式参数
  - 函数定义时，紧跟在函数名后（圆括号内）的参数被称为形式参数，简称形参。由于它不是实际存在变量，所以又称虚拟变量
- 实际参数
  - 在主调函数中调用一个函数时，函数名后面括弧中的参数（可以是一个表达式）称为“实际参数”，简称实参

## 传递参数

知识讲解

- 调用函数时，实参的个数需要与形参个数一致
- 实参将依次传递给形参

```
>>> def foo(x, y):
...     print 'x=%d, y=%d' % (x, y)
>>> foo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() takes exactly 2 arguments (0 given)
>>> foo(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() takes exactly 2 arguments (1 given)
>>> foo(3, 4)
x=3, y=4
```



## 位置参数

知识讲解

- 与shell脚本类似，程序名以及参数都以位置参数的方式传递给python程序
- 使用sys模块的argv列表接收

```
[root@py01 bin]# vim args.py
#!/usr/bin/env python
import sys
print sys.argv
[root@py01 bin]# ./args.py hello world
['./args.py', 'hello', 'world']
```



# 默认参数

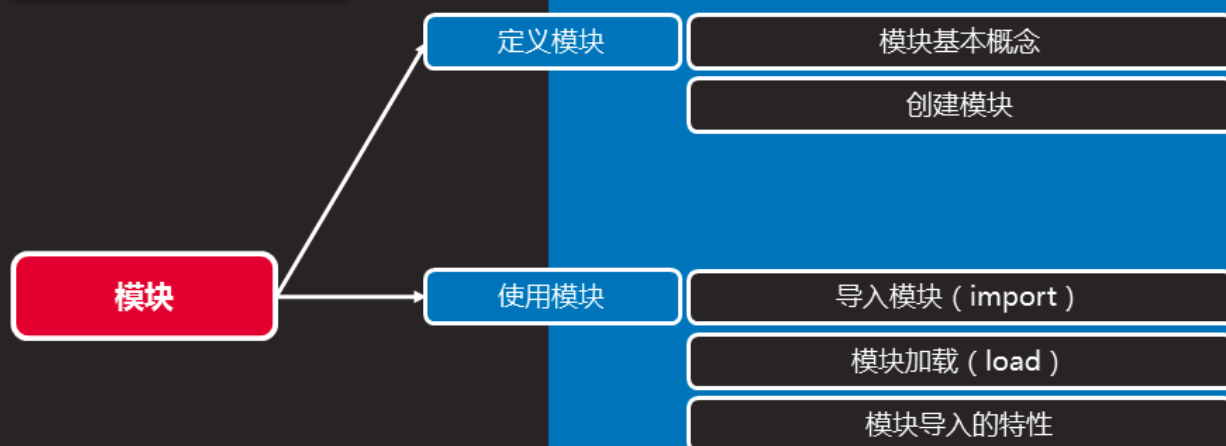
知识讲解

- 默认参数就是声明了默认值的参数
- 因为给参数赋予了默认值，所以在函数调用时，不向该参数传入值也是允许的

```
>>> def pstar(num = 30):
...     print '*' * num
...
>>> pstar()
*****
>>> pstar(40)
*****
```



## 模块



# 定义模块

## 模块基本概念

- 模块是从逻辑上组织python代码的形式
- 当代码量变得相当大的时候，最好把代码分成一些有组织的代码段，前提是保证它们的彼此交互
- 这些代码片段相互间有一定的联系，可能是一个包含数据成员和方法的类，也可能是一组相关但彼此独立的操作函数

## 创建模块

知识讲解

- 模块物理层面上组织模块的方法是文件，每一个以.py作为结尾的python文件都是一个模块
- 模块名称切记不要与系统中已存在的模块重名
- 模块文件名字去掉后面的扩展名（.py）即为模块名



## 使用模块

---

## 导入模块 ( import )

知识讲解

- 使用import导入模块
- 模块被导入后，程序会自动生成pyc的字节码文件以提升性能
- 模块属性通过“模块名.属性”的方法调用
- 如果仅需要模块中的某些属性，也可以单独导入

```
>>> import sys
>>> import os, string
>>> string.digits
'0123456789'
>>> from random import randint
>>> randint(1, 10)
3
```



## 模块加载 ( load )

知识讲解

- 一个模块只被加载一次，无论它被导入多少次
- 只加载一次可以阻止多重导入时代码被多次执行
- 如果两个文件相互导入，防止了无限的相互加载
- 模块加载时，顶层代码会自动执行，所以只将函数放入模块的顶层是良好的编程习惯



## 模块导入的特性

知识讲解

- 模块具有一个\_\_name\_\_特殊属性
- 当模块文件直接执行时，\_\_name\_\_的值为'\_\_main\_\_'
- 当模块被另一个文件导入时，\_\_name\_\_的值就是该模块的名字

```
[root@py01 bin]# vim foo.py
#!/usr/bin/env python
print __name__
[root@py01 bin]# ./foo.py
__main__
[root@py01 bin]# python
>>> import foo
foo
```



## 案例3：生成8位随机密码

课堂练习

1. 编写一个能生成8位随机密码的程序
2. 使用random的choice函数随机取出字符
3. 用于密码的字符通过string模块获得
4. 改进程序，用户可以自己决定生成多少位的密码





## 总结和答疑

---



# IOError

---

## 问题现象

- 试图读写文件时出现以下错误

```
>>> f.write('hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: File not open for writing
```

知识讲解



## 故障分析及排除

- 原因分析
  - 文件以r方式打开，不能执行写入操作
  - 文件以w方式打开，不能执行读取操作
- 解决办法
  - 打开文件时，指定正确的打开模式

知识讲解



# 调用函数失败

## 问题现象

- 调用函数时，没有得到正确的结果

```
>>> def pstar():  
...     print '*' * 20  
...  
>>> pstar  
<function pstar at 0x100697410>
```

# 故障分析及排除

知识讲解

- 原因分析
  - 函数的调用需要使用圆括号
  - 没有使用圆括号，是对函数的引用，而不是调用
- 解决办法
  - 执行函数时，使用圆括号调用

