

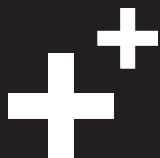
Python开发进阶

NSD PYTHON2

DAY02

内容

上午	09:00 ~ 09:30	作业讲解和回顾
	09:30 ~ 10:20	函数基础
	10:30 ~ 11:20	
	11:30 ~ 12:00	函数高级应用
下午	14:00 ~ 14:50	
	15:00 ~ 15:50	模块
	16:10 ~ 17:00	
	17:10 ~ 18:00	总结和答疑



函数基础

函数基础

创建函数

def语句

前向引用

调用函数

函数操作符

关键字参数

参数组

匿名函数

lambda

filter()函数

map()函数

创建函数

def 语句

- 函数用def语句创建，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



前向引用

- 函数不允许在函数未声明之前对其进行引用或者调用

```
def foo():
    print('in foo')
    bar()
```

```
foo()      #报错，因为bar没有定义
```

```
def foo():
    print('in foo')
    bar()
```

```
def bar():
    print('in bar')
```

```
foo()      #正常执行，虽然bar的定义在foo定义后面
```



调用函数

函数操作符

- 使用一对圆括号()调用函数，如果没有圆括号，只是对函数的引用
- 任何输入的参数都必须放置在括号中

```
>>> def foo():  
...     print('Hello world!')  
...  
>>> foo()  
Hello world!  
>>> foo  
<function foo at 0x7f18ce311b18>
```



关键字参数

- 关键字参数的概念仅仅针对函数的调用
- 这种理念是让调用者通过函数调用中的参数名字来区分参数
- 这样规范允许参数缺失或者不按顺序

```
>>> def get_info(name, age):
...     print("%s's age is %s" % (name, age))
...
>>> get_info(23, 'bob')
23's age is bob
>>> get_info(age = 23, name = 'bob')
bob's age is 23
```



参数组

- python允许程序员执行一个没有显式定义参数的函数
- 相应的方法是通过一个把元组（非关键字参数）或字典（关键字参数）作为参数组传递给函数

```
func(*tuple_grp_nonkw_args, **dict_grp_kw_args)
```



案例1：简单的加减法数学游戏

1. 随机生成两个100以内的数字
2. 随机选择加法或是减法
3. 总是使用大的数字减去小的数字
4. 如果用户答错三次，程序给出正确答案



匿名函数

lambda

- python允许用lambda关键字创造匿名函数
- 匿名是因为不需要以标准的def方式来声明
- 一个完整的lambda “语句” 代表了一个表达式，这个表达式的定义体必须和声明放在同一行

lambda [arg1[, arg2, ... argN]]: expression

```
>>> a = lambda x, y: x + y
```

```
>>> print(a(3, 4))
```

```
7
```



filter()函数

- filter(func, seq) : 调用一个布尔函数func来迭代遍历每个序列中的元素；返回一个使func返回值为true的元素的序列
- 如果布尔函数比较简单，直接使用lambda匿名函数就显得非常方便了

```
>>> data = filter(lambda x: x % 2, [num for num in range(10)])
>>> print(list(data))      #过滤出10以内的奇数
[1, 3, 5, 7, 9]
```



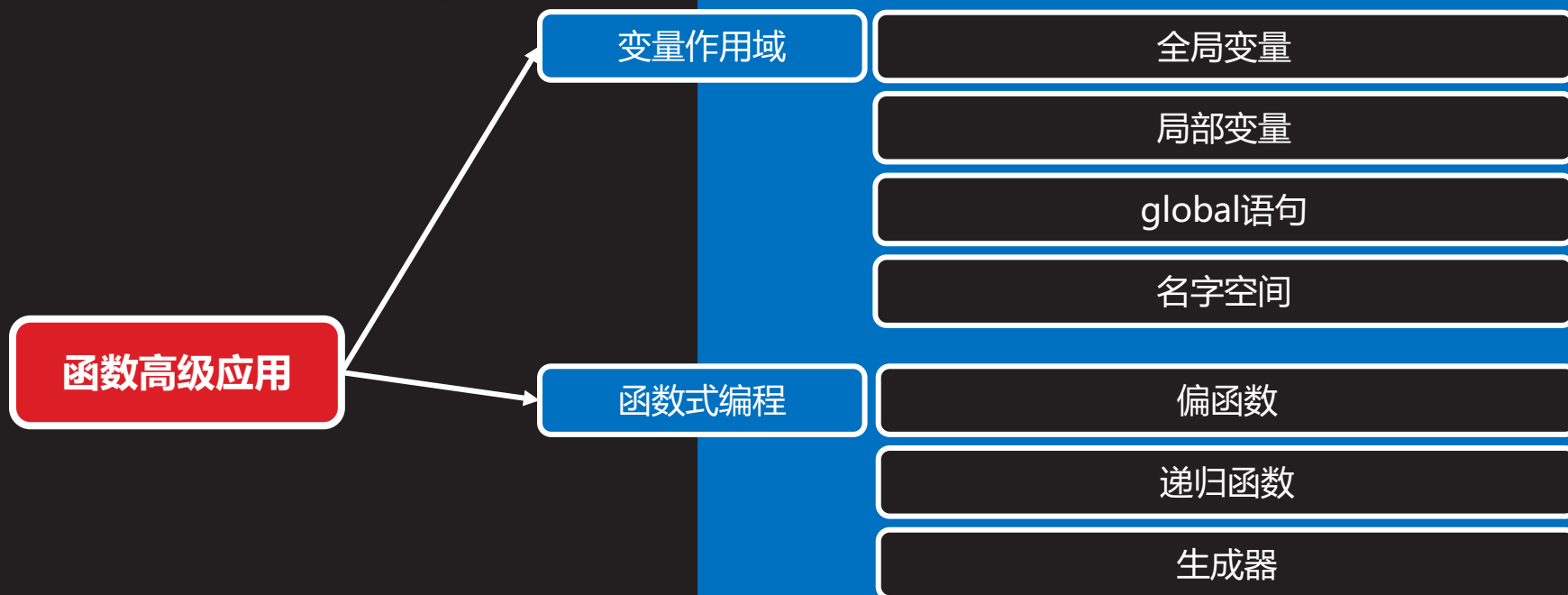
map()函数

- `map(func, seq)` : 调用一个函数`func`来迭代遍历每个序列中的元素 ; 返回一个经过`func`处理过的元素序列
- 如果布尔函数比较简单 , 直接使用`lambda`匿名函数就显得非常方便了

```
>>> data = map(lambda x: x*2+1, [num for num in range(10)])  
>>> list(data)  
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```



函数高级应用



变量作用域

全局变量

- 标识符的作用域是定义为其声明在程序里的可应用范围，也就是变量的可见性
- 在一个模块中最高级别的变量有全局作用域
- 全局变量的一个特征是除非被删除掉，否则它们的存活到脚本运行结束，且对于所有的函数，它们的值都是可以被访问的



局部变量

- 局部变量只时暂时地存在，仅仅只依赖于定义它们的函数现阶段是否处于活动
- 当一个函数调用出现时，其局部变量就进入声明它们的作用域。在那一刻，一个新的局部变量名为那个对象创建了
- 一旦函数完成，框架被释放，变量将会离开作用域



局部变量（续1）

- 如果局部与全局有相同名称的变量，那么函数运行时，局部变量的名称将会把全局变量名称遮盖住

```
>>> x = 4
>>> def foo():
...     x = 10
...     print('in foo, x =', x)
...
>>> foo()
in foo, x = 10
>>> print('in main, x =', x)
in main, x = 4
```



global语句

- 因为全局变量的名字能被局部变量给遮盖掉，所以为了明确地引用一个已命名的全局变量，必须使用global语句

```
>>> x = 4
>>> def foo():
...     global x
...     x = 10
...     print('in foo, x =', x)
...
>>> foo()
in foo, x = 10
>>> print('in main, x =', x)
in main, x = 10
```



函数式编程

偏函数

- 偏函数的概念是将函数式编程的概念和默认参数以及可变参数结合在一起
- 一个带有多个参数的函数，如果其中某些参数基本上固定的，那么就可以通过偏函数为这些参数赋默认值

```
>>> from operator import add
>>> from functools import partial
>>> add10 = partial(add, 10)
>>> print(add10(25))
35
```



案例2：简单GUI程序

1. 窗口程序提供三个按钮
2. 其中两个按钮的前景色均为白色，背景色为蓝色
3. 第三个按钮前景色为红色，背景色为红色
4. 按下第三个按钮后，程序退出



递归函数

- 如果函数包含了对其自身的调用，该函数就是递归的
- 在操作系统中，查看某一目录内所有文件、修改权限等都是递归的应用

```
>>> def func(num):
...     if num == 1:
...         return 1
...     else:
...         return num * func(num - 1)
...
>>> print(func(5))
120
>>> print(func(10))
3628800
```



案例3：快速排序

1. 随机生成10个数字
2. 利用递归，实现快速排序



生成器

- 从句法上讲，生成器是一个带yield语句的函数
- 一个函数或者子程序只返回一次，但一个生成器能暂停执行并返回一个中间的结果
- yield 语句返回一个值给调用者并暂停执行
- 当生成器的next()方法被调用的时候，它会准确地从离开地方继续



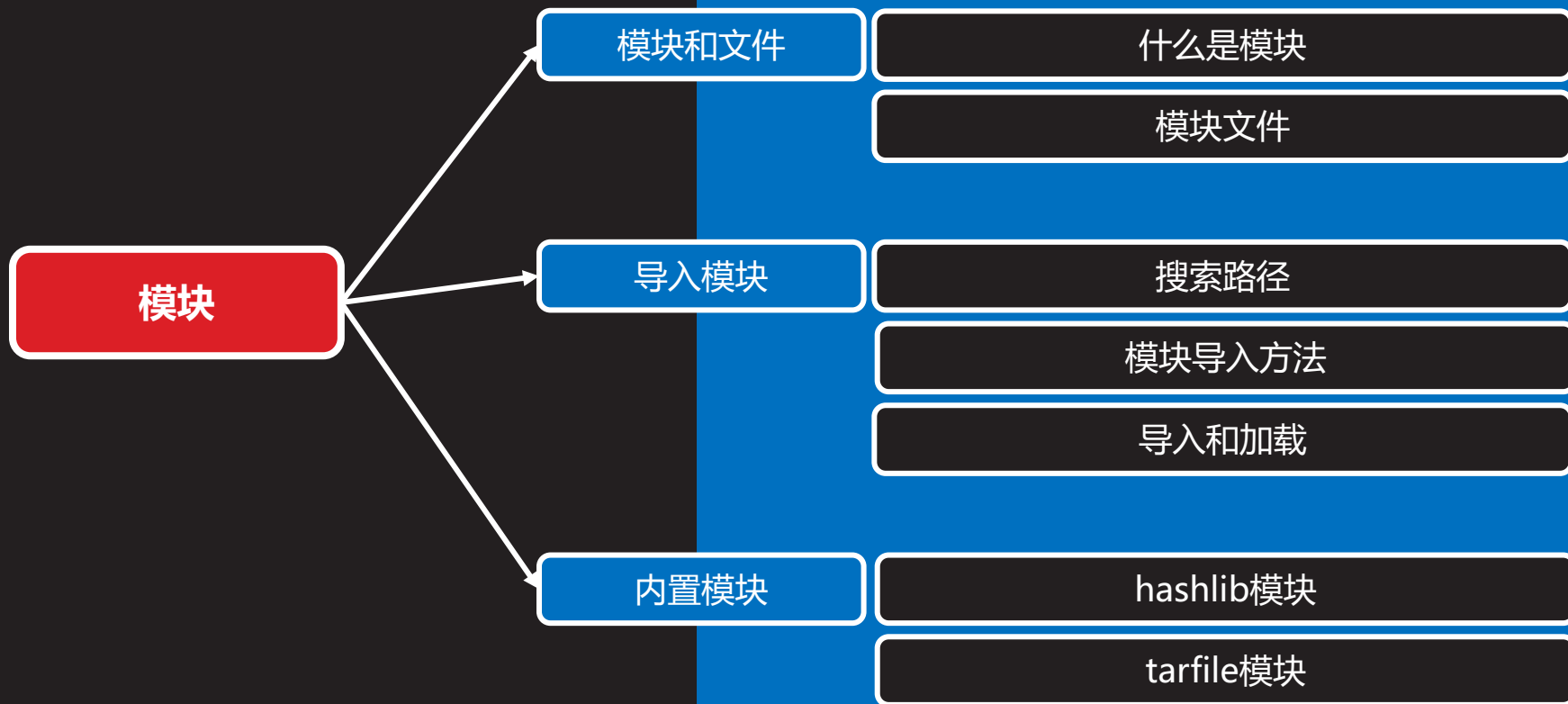
生成器（续1）

- 与迭代器相似，生成器以另外的方式来运作
- 当到达一个真正的返回或者函数结束没有更多的值返回，StopIteration异常就会被抛出

```
>>> def simp_gen():
...     yield '1'
...     yield '2 -> punch'
>>> mygen = simp_gen()
>>> mygen.next()
'1'
>>> mygen.next()
'2 -> punch'
>>> mygen.next()
... ..
StopIteration
```



模块



模块和文件

什么是模块

- 模块支持从逻辑上组织python代码
- 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
- 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
- 这些代码段是共享的, 所以python允许“调入”一个模块, 允许使用其他模块的属性来利用之前的工作成果, 实现代码重用



模块文件

- 说模块是按照逻辑来组织python代码的方法，文件是物理层上组织模块的方法
- 一个文件被看作是一个独立模块，一个模块也可以被看作是一个文件
- 模块的文件名就是模块的名字加上扩展名.py



导入模块



搜索路径

- 模块的导入需要一个叫做“路径搜索”的过程
- python在文件系统“预定义区域”中查找要调用的模块
- 搜索路径在sys.path中定义

```
>>> import sys
>>> print(sys.path)
['', '/usr/local/lib/python36.zip', '/usr/local/lib/python3.6',
'/usr/local/lib/python3.6/lib-dynload', '/usr/local/lib/python3.6/site-
packages']
```



模块导入方法

- 使用import导入模块
- 可以在一行导入多个模块，但是可读性会下降
- 可以只导入模块的某些属性
- 导入模块时，可以为模块取别名

```
>>> import time, os, sys  
>>> from random import choice  
>>> import pickle as p
```



导入和加载

- 当导入模块时，模块的顶层代码会被执行
- 一个模块不管被导入（import）多少次，只会被加载（load）一次

```
[root@py01 ~]# cat foo.py
```

```
hi = 'hello'
```

```
print(hi)
```

```
[root@py01 ~]# python3
```

```
>>> import foo
```

```
Hello
```

#第一次导入，执行print语句

```
>>> import foo
```

#再次导入，print语句不再执行



内置模块



hashlib模块

- hashlib用来替换md5和sha模块，并使他们的API一致，专门提供hash算法
- 包括md5、sha1、sha224、sha256、sha384、sha512，使用非常简单、方便

```
>>> import hashlib  
>>> m = hashlib.md5()  
>>> m.update('hello world!')  
>>> m.hexdigest()  
'fc3ff98e8c6a0d3087d515c0473f8677'
```



tarfile模块

- tarfile模块允许创建、访问tar文件
- 同时支持gzip、bzip2格式

```
[root@py01 home]# ls /home/demo/  
install.log mima  
[root@py01 home]# python  
>>> import tarfile  
>>> tar = tarfile.open('/home/demo.tar.gz', 'w:gz')  
>>> tar.add('demo')  
>>> tar.close()
```



案例4：备份程序

1. 需要支持完全和增量备份
2. 周一执行完全备份
3. 其他时间执行增量备份
4. 备份文件需要打包为tar文件并使用gzip格式压缩



总结和答疑
