7 Python中的函数与函数式编程（下）

> AI领域中的Python开发 --- by 丁宁

@(SIGAI课程录制)

- 上节课：函数定义，函数的参数，命名空间与作用域解析
- 本节课：函数式编程概述，一等函数，高阶函数，匿名函数

函数式编程概述

- 前提：函数在Python中是一等对象
- 工具：**built-in**高阶函数；**lambda**函数；**operator**模块；**functools**模块
- 模式：闭包与装饰器
- 替代：用**List Comprehension**可轻松替代**map**和**filter**（**reduce**替代起来比较困难）
- 原则：**No Side Effect**

> 何为**No Side Effect**？ 函数的所有功能就仅仅是返回一个新的值而已，没有其他行为，尤其是不得修改外部变量 因而，各个独立的部分的执行顺序可以随意打乱，带来执行顺序上的自由 执行顺序的自由使得一系列新的特性得以实现：无锁的并发；惰性求值；编译器级别的性能优化等

程序的状态与命令式编程

- 程序的状态首先包含了当前定义的全部变量
- 有了程序的状态，我们的程序才能不断往前推进
- 命令式编程，就是通过不断修改变量的值，来保存当前运行的状态，来步步推进

函数式编程

- 通过函数来保存程序的状态（通过函数创建新的参数和返回值来保存状态）
- 函数一层层的叠加起来，每个函数的参数或返回值代表了一个中间状态
- 命令式编程里一次变量值的修改，在函数式编程里变成了一个函数的转换
- 最自然的方式：递归

一等函数

一等对象

定义：

- 在运行时创建
- 能赋值给变量或数据结构中的元素
- 能作为参数传给函数
- 能作为函数的返回结果

> Python中，所有函数的都是一等对象，简称为一等函数

> 什么是运行时?

在运行时创建

```
sigai@81af80ff9515:~$ python
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> def say_hi(): print("Hello SigAI")
...
>>> say_hi()
Hello SigAI
```

能赋值给变量或数据结构中的元素

```
>>> test = say_hi
>>> test()
Hello SigAI
```

能作为参数传递给函数

```
>>> def repeat(f,num):
...     [f() for i in range(num)]
...
>>> repeat(say_hi,3)
Hello SigAI
Hello SigAI
Hello SigAI
```

能作为函数的返回结果

```
>>> def repeat_func(f,num):
...     def new_func():
...         [f() for i in range(num)]
...     return new_func
...
>>> repeated_func = repeat_func(say_hi,3)
>>> repeated_func()
Hello SigAI
Hello SigAI
Hello SigAI
```

高阶函数

- 定义：接受函数为参数，或把函数作为返回结果的函数

map高阶函数

```
>>> def square(x): return x * x
...
>>> xx = map(square, range(10))
>>> xx
<map object at 0x7f993713bef0>
>>> xx = list(xx)
>>> xx
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

> 可用`[x * x for x in range(10)]`替代

filter高阶函数

```
>>> x = [(), [], {}, None, '', False, 0, True, 1, 2, -3]
>>> x_result = filter(bool, x)
>>> x_result
<filter object at 0x7f993713beb8>
>>> x_result = list(x_result)
>>> x_result
[True, 1, 2, -3]
```

> 可用`[i for i in x if bool(i)]`替代

reduce高阶函数

```
>>> def multiply(a,b): return a * b
...
>>> from functools import reduce
>>> reduce(multiply, range(1,5))
24
```

sorted高阶函数

```
>>> sorted([x * (-1) ** x for x in range(10)])
[-9, -7, -5, -3, -1, 0, 2, 4, 6, 8]
>>> sorted([x * (-1) ** x for x in range(10)], reverse=True)
[8, 6, 4, 2, 0, -1, -3, -5, -7, -9]
>>> sorted([x * (-1) ** x for x in range(10)], key=abs)
[0, -1, 2, -3, 4, -5, 6, -7, 8, -9]
>>> sorted([x * (-1) ** x for x in range(10)], reverse=True, key=abs)
[-9, 8, -7, 6, -5, 4, -3, 2, -1, 0]
```

> **max min**同理

partial高阶函数

```
>>> from functools import partial
>>> abs_sorted = partial(sorted, key=abs)
>>> abs_sorted([x * (-1) ** x for x in range(10)])
[0, -1, 2, -3, 4, -5, 6, -7, 8, -9]
>>> abs_reverse_sorted = partial(sorted, key=abs, reverse=True)
>>> abs_reverse_sorted([x * (-1) ** x for x in range(10)])
[-9, 8, -7, 6, -5, 4, -3, 2, -1, 0]
```

匿名函数

- 定义：使用**lambda**表达式创建的函数，函数本身没有名字
- 特点：只能使用纯表达式，不能赋值，不能使用while和try等块语句
- 语法：`lambda [arg1 [,arg2 [,arg3……]]]: expression`

> 什么是表达式？ Expressions get a value; Statements do something

用`lambda`与用`def`的区别在哪

- 写法上：

1. **def**可以用代码块，一个代码块包含多个语句
2. **lambda**只能用单行表达式，而表达式仅仅是单个语句中的一种

- 结果上：

1. **def**语句一定会增加一个函数名称
2. **lambda**不会，这就降低了变量名污染的风险

> Compound Statements > Simple Statements > Expressions

能用一个表达式直接放到`return`里返回的函数都可以用`lambda`速写

```
>>> def multiply(a,b): return a * b
...
>>> multiply(3,4)
12
>>> multiply_by_lambda = lambda x,y: x * y
>>> multiply_by_lambda(3,4)
12
```

List + lambda 可以得到行为列表

```
>>> f_list = [lambda x: x + 1, lambda x: x ** 2, lambda x: x ** 3]
>>> [f_list[j](10) for j in range(3)]
[11, 100, 1000]
```

> 在AI领域里，这种写法常用于处理数据，比如按预定的一系列模式处理数据

入门小坑系列之---惰性计算

请推测以下代码的执行结果，并思考为什么?

```
>>> f_list = [lambda x: x**i for i in range(5)]
>>> [f_list[j](10) for j in range(5)]
```