

https://woaielf.github.io/  
—— made by ZY

## 函数二

### 递归函数

定义

★ 函数在内部调用「自身」

原因

栈就会加一层栈帧 每进入一个函数调用  
栈就会减一层栈帧 每当函数返回  
导致栈溢出 递归调用的次数过多

防止「栈溢出」

注意

解决

效果 = 循环 本质上  
调用自身本身 函数返回时  
不能包含表达式 return  
针对尾递归继续做优化 若没有  
都存在「栈溢出」的问题 任何递归函数

### 返回（回调）函数

★ 牢记

「闭包」的程序结构

一个新的函数 都会返回  
相同的参数 即使传入 每次调用 函数作为「结果值」返回  
互不影响 调用结果  
外部函数的「参数和局部变量」 引用 内部函数  
已保存「参数和变量」的内部函数 返回 调用外部函数

再创建一个函数  
用该函数的参数  
绑定循环变量当前的值

如果一定要引用

任何「循环变量」  
后续会发生变化的变量 or 返回函数不要引用

### 变量作用域

标识符的作用域

在程序里的可应用范围 其声明  
★ 除非声明 global 不能修改 函数内  
脚本运行结束 否则存活到 除非被删除  
只允许函数内访问 在函数内定义  
函数是否处于「活动状态」 依赖于 存在与否  
★ 局部作用域 → 全局域  
全局变量 可能隐藏 / 覆盖 局部变量 p.s. 变量搜索顺序

### 偏函数

概述

偏函数应用 (PFA)

Partial function application  
设置默认值 即 函数的某些参数 固定  
以供调用 一个新的函数 返回 作用

使代码紧凑易读！

需要简化 函数的参数太多  
原函数的部分参数 固定住 创建一个新函数 使用场景

应用

模块

functools

from functools import partial  
partial(func, \*args, \*\*keywords)

创建

# add1(x) == add(1, x) add1 = partial(add, 1)  
# int\_new == int(2, base) int\_new = partial(int, 2)

固定「位置参数」

int2 = partial(int, base=2)  
→ 18 int2('10010')

可更改，仍为关键字参数

int2('10010', base=10)

重设「关键字参数」

### 高阶函数 BIF

定义

一个函数接收「另一个函数」作为「参数」  
可与 lambda 完美结合

① apply()

已经允许函数用「可变参数」

没有存在的价值了

filter()

filter(func, seq)  
将函数对象 依次作用于每一个元素  
根据返回值为 True / False 决定保留 / 丢弃该元素  
e.g. filter(lambda n: n%2, allNums)  
★ 可用「列表解析」实现同样功能 [n for n in allNums if n % 2]

map()

map(func, seq1[, seq2, ...])  
将函数对象 依次作用于每一个元素  
每次作用的结果 储存于返回的 list 中  
可以有 n 个列表 对应函数的 n 个参数  
e.g. map(lambda x: x\*\*2, range(5)) 等价于 [x\*\*2 for x in range(5)]  
map((lambda x, y: x + y), [1, 2, 3], [6, 7, 9]) → [7, 9, 12]  
map(None, [1, 3, 5], [2, 4, 6]) → [(1, 2), (3, 4), (5, 6)]  
思想类似 zip()

vs

① from functools import reduce

reduce(func, seq[, init])

累进作用

二元函数

★ func 接收两个参数  
上一次 返回值 / 运算结果  
序列下一个元素

初始化器

若设定 第一次计算为 初始化器 & 第一个序列元素

e.g. reduce(lambda x, y: x+y, range(5)) 相当于 (((0+1)+2)+3)+4  
reduce(func, [1, 2, 3]) = func(func(1, 2), 3)

依次作用

返回「迭代器」

返回 1 个值

释义

一种「编程范式」 如何编写程序的方法论 e.g. 面向对象  
属于「结构化编程」 采用 子程序 面向过程  
程式码区块  
for / while 循环

思想

把运算过程 尽量写成 一系列嵌套的函数调用

特点

允许 把函数作为参数 传入另一个函数  
返回一个函数

Python

并非「函数式编程语言」  
但支持一些 函数式编程语言的构建 匿名函数  
BIF filter, map  
偏函数  
本质上 ① 通过「封装对象」实现「函数式编程」

关键字

lambda 不需要以 标准的方式 来声明

表达式

lambda [arg1[, arg2, ..., argN]]: expression  
定义体 + 声明  
同一行 不用写 return

使用

方法一  
① 赋值给一个变量 func = lambda x, y: x + y  
lambda 生成一个函数对象  
参数为 x, y  
返回值为 x+y  
函数对象赋给 func  
② 用变量来调用该函数 func(3, 4) func 的调用与正常函数无异  
方法二  
def build(x, y):  
return lambda: x \* x + y \* y