第6课　函数 与 函数式编程

函数概述

- 函数的基本概念
  - 函数用于在程序中分离不同的任务
  - 函数允许程序的控制在调用代码和函数代码之间切换
- 函数的功能

（1）实现结构化程序设计。通过把程序分割为不同的功能模块，可以实现自顶向下的结构化设计

（2）减少程序的复杂度。简化程序的结构，提高程序的可阅读性

（3）实现代码的复用。一次定义多次调用，实现代码的可重用性

（4）提高代码的质量。实现分割后子任务的代码相对简单，易于开发、调试、修改和维护

（5）协作开发。大型项目分割成不同的子任务后，团队多人可以分工合作，同时进行协作开发

（6）实现特殊功能。递归函数可以实现许多复杂的算法

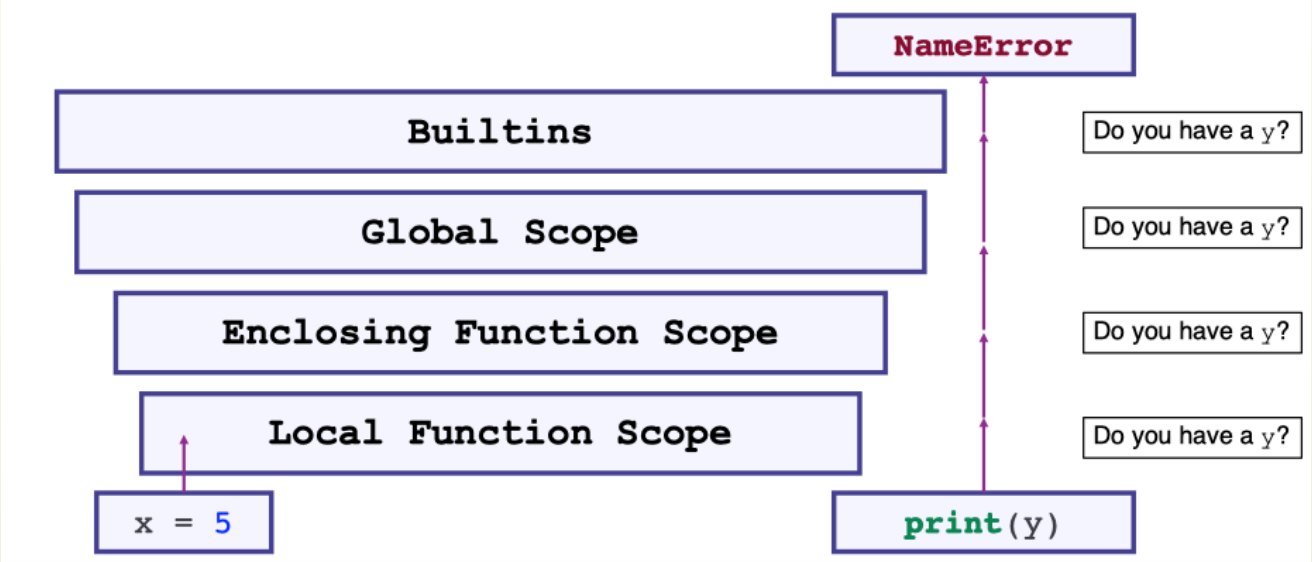
函数

```
def fn_name(param1, param2):
    ...
    return some_values
```

- 函数接收的参数不需要声明其类型，也不需要指定函数返回值类型
- 即使该函数不需要接收任何参数，也必须保留一对空的圆括号
- 括号后面的冒号必不可少
- 函数体相对于def关键字必须保持一定的空格缩进
- 可以返回多个值；如果没有return 语句， 函数返回 None

函数

- **全局变量**：在函数外面定义的变量称为全局变量。全局变量的作用域在整个代码段（文件、模块），在整个程序代码中都能被访问到。
- **局部变量**：在函数内部定义的参数和变量称为局部变量，超出了这个函数的作用域局部变量是无效的，它的作用域仅在函数内部。
- 变量的引用：**变量解析规则 Variable Resolution**.



变量解析规则

- 作用域（Scope）：LEGB Rule

L. Local. (在函数中以任何方式声明的名称，未声明为全局名称)

E. Enclosing function locals. (封闭式的作用域规则适应于在函数里定义函数时，也就是说，**在函数体内定义了一个新的函数**。这个函数体内的函数是外函数的局部命名空间中的一部分，意味着只有在外函数执行期间才能够运行）

G. Global (module).（在模块文件的顶层定义的名称）

B. Built-in (Python). (Python自带的: 如open, range, SyntaxError)

变量解析规则

- 例子

```
x = 2
def foo(y):
    z = 5
    print(locals())
    print(globals()['x'])
    print(x, y, z)

foo(3)

# {'y': 3, 'z': 5}
# 2
# 2 3 5
```

```
x = 2
def foo(y):
    x = 41
    z = 5
    print(locals())
    print(globals()['x'])
    print(x, y, z)
foo(3)

# {'y': 3, 'x': 41, 'z': 5}
# 2
# 41 3 5
```

变量解析规则

- 全局变量

```
def test_global():
        global x
        x = 3
        y = 4
        print(x,y)


x=5
test_global()
print(x)
print(y)
```

```
3 4
3
Traceback (most recent call last):
  File "test.py", line 10, in
<module>
    print(y)
NameError: name 'y' is not defined
```

全局变量？

```
m = 100
n = 200
def f():
    print(m+5)
    n += 10
    print(n)

#测试代码
f()
```

全局变量？

在函数体中，可以引用全局变量，但如果函数内部的变量名是第一次出现且在赋值语句之前（变量赋值），则解释为定义局部变量

```
m = 100
n = 200
def f():
    print(m+5)  #引用全局变量m
    n += 10 #错误，n在赋值语句前面，解释为局部变量（不存在）
#测试代码
f()
```

全局变量？

```
pi = 3.141592653589793 #全局变量
e = 2.718281828459045 #全局变量
def my_func():
    global pi
   pi = 3.14
    print('global pi =', pi)
    e = 2.718
    print('local e =', e)

#测试代码
print('module pi =', pi)
print('module e =', e)
my_func()
print('module pi =', pi)
print('module e =', e)
```

全局变量？

```
module pi = 3.141592653589793
module e = 2.718281828459045
global pi = 3.14
local e = 2.718
module pi = 3.14
module e = 2.718281828459045
```

```
pi = 3.141592653589793 #全局变量
e = 2.718281828459045 #全局变量
def my_func():
    global pi #全局变量，与前面的全局变量pi指向相同的对象
    pi = 3.14 #改变了全局变量的值
    print('global pi =', pi) #输出全局变量的值
    e = 2.718 #局部变量，与前面的全局变量e指向不同的对象
    print('local e =', e)  #输出局部变量的值
#测试代码
print('module pi =', pi)   #输出全局变量的值
print('module e =', e)   #输出全局变量的值
my_func()          #调用函数
print('module pi =', pi)  #输出全局变量的值，该值在函数中已被更改
print('module e =', e)   #输出全局变量的值
```

非局部语句 nonlocal

- 在函数体中，可以定义嵌套函数，在嵌套函数中，如果要为定义在上级函数体的局部变量赋值，可以使用nonlocal语句，表明变量不是所在块的局部变量，而是在上级函数体中定义的局部变量。nonlocal语句可以指定多个非局部变量。例如nonlocal x, y, z

```
def outer_func():
    tax_rate = 0.17
    print('outer func tax rate =', tax_rate)

    def innner_func():
        nonlocal tax_rate
        tax_rate = 0.05
        print('inner func tax rate =', tax_rate)

    innner_func()                    #调用函数
    print('outer func tax rate =', tax_rate)

#测试代码
outer_func()
```

非局部语句 nonlocal

- 在函数体中，可以定义嵌套函数，在嵌套函数中，如果要为定义在上级函数体的局部变量赋值，可以使用nonlocal语句，表明变量不是所在块的局部变量，而是在上级函数体中定义的局部变量。nonlocal语句可以指定多个非局部变量。例如nonlocal x, y, z

```
def outer_func():
    tax_rate = 0.17          #上级函数体中的局部变量
    print('outer func tax rate =', tax_rate) #输出上级函数体中局部变量的值

    def innner_func():
        nonlocal tax_rate    #不是所在块的局部变量，而是在上级函数体中定义的局部变量
        tax_rate = 0.05  #上级函数体中的局部变量重新赋值
        print('inner func tax rate =', tax_rate) #输出上级函数体中局部变量的值

    innner_func()                    #调用函数
    print('outer func tax rate =', tax_rate)   #输出上级函数体中局部变量的值（已更改）

#测试代码
outer_func()
```

```
outer func tax rate = 0.17
inner func tax rate = 0.05
outer func tax rate = 0.05
```

函数的参数

- 假设我们要写一个函数，让一个用户连接一台远程的机器（服务器）
- 输入：用户名，密码，服务器IP地址，端口

```
def connect(uname, pword, server, port):
    print("Connecting to", server, ":", port, "...”)
    # Connecting code here ...
```

- `connect('admin', 'ilovecats', '192.168.81.11', 9160)`
- `connect('jdoe', '12345678', '120.241.186.165', 6370)`

函数的参数

```
def connect(uname, pword, server, port):
    print("Connecting to", server, ":", port, "...”)
    # Connecting code here ...
```

- connect('Alice', 'ilovedogs', '120.241.186.165', 9160)
- connect('Bob', 'asdfghjkl', '120.241.186.165', 9160)
- connect('Caren', '12345678', '120.241.186.165', 9160)
- connect('David', '20200101', '120.241.186.165', 9160)

存在什么问题？

函数的参数

```
def connect(uname, pword, server = '120.241.186.165', port = 9160):
    # connecting code
```

- 函数的默认参数
  - 我们可以给函数里面的每个参数提供一个默认参数
  - 调用函数的时候，带有默认参数的参数可以不用初始化
  - 带有默认参数的**必须写到最后**

```
>>> def f(a=1,b,c=5):
...     return a
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default
argument
```

函数的参数

```
def connect(uname, pword, server = '120.241.186.165', port = 9160):
    # connecting code
```

- 以下函数调用都是合法的（注意输入参数的个数不一定是4个！）.

  - `connect('admin', 'ilovecats')`
  - `connect('admin', 'ilovecats', '183.232.231.174')`
  - `connect(Alice', '12345678', '183.232.231.174', 6379)`

重写（override）默认值

- 总结：调用带有默认值参数的函数时，可以不对默认值参数进行赋值，也可以赋值，具有较大的灵活性。

有趣的现象

```
''' Module adder.py '''

def add_item(item, item_list = []):
    item_list.append(item) # 添加新元素到列表中
    print(item_list)
```

```
$ python
>>> from adder import
*
>>> add_item(3, [])
[3]
>>> add_item(4)
[4]
>>> add_item(5)
[4, 5]
```

有趣的现象

```
''' Module adder.py '''

def add_item(item, item_list = []):
    item_list.append(item)
    print(item_list)
```

```
$ python
>>> from adder import *
>>> add_item(3, [])
[3]
>>> add_item(4)
[4]
>>> add_item(5)
[4, 5]
```

- 默认参数的赋值只会在函数定义时被解释一次，而不是每次调用函数时都重新定义一次。
- 当使用可变序列作为参数默认值时: 谨慎操作!

有趣的现象

- 如何修复bug?

```
''' Module adder.py '''

def add_item(item, item_list = None):
    if item_list == None:
        item_list = []
    item_list.append(item)
    print(item_list)
```

```
$ python
>>> from adder import
*
>>> add_item(3, [])
[3]
>>> add_item(4)
[4]
>>> add_item(5)
[5]
```

位置参数和关键字参数

```
def connect(uname, pword, server = '120.241.186.165', port = 9160):
    # connecting code
```

- **位置参数**(positional argument)：通过在参数列表中的相对位置确定传递给哪个参数

```
connect('admin', 'ilovecats', '120.241.186.165', 6379)
```

- **关键字参数**(keyword argument)：通过 name=value 的形式，根据name确定传递给哪个参数

```
connect(uname='admin', pword='ilovecats',
server='120.241.186.165', port=6379)
```

- 关键字参数必须跟随在位置参数后面! 关键字参数的相对位置不重要！
  - python函数在解析参数时是按照顺序来的，位置参数是必须先满足，才能考虑其他可变参数

位置参数和关键字参数

```
def connect(uname, pword, server = 'localhost', port = 9160):
    # connecting code
```

- 哪个函数的调用是合法的？

```
1. connect('admin', 'ilovecats', '120.241.186.165')

2. connect(uname='admin', pword='ilovecats',
'120.241.186.165')

3. connect('admin', 'ilovecats', port=6379,
server='120.241.186.165')
```

位置参数和关键字参数

```
def connect(uname, pword, server = 'localhost', port = 9160):
    # connecting code
```

- 哪个函数的调用是合法的？

```
1. connect('admin', 'ilovecats', '120.241.186.165')
                                                -- VALID

2. connect(uname='admin', pword='ilovecats',
'120.241.186.165')                              -- INVALID

3. connect('admin', 'ilovecats', port=6379,
server='120.241.186.165')                       -- VALID
```

可变长度参数

- *param形式的参数包含可变数量的参数。
- **param形式的参数包含可变数量的关键字参数。

```
def connect(uname, *args, **kwargs):
        print(uname)
        print(type(args))
        print(type(kwargs))
        for arg in args:
                print(arg)
        for key in kwargs.keys():
                print(key, ":", kwargs[key])

connect('admin', 'ilovecats', server='localhost', port=9160)
```

输出什么？

可变长度参数

```
def connect(uname, *args, **kwargs):
        print(uname)
        print(type(args))
        print(type(kwargs))
        for arg in args:
                print(arg)
        for key in kwargs.keys():
                print(key, ":", kwargs[key])

connect('admin', 'ilovecats', server='localhost', port=9160)
```

```
admin
<class 'tuple'>
<class 'dict'>
ilovecats
server : localhost
port : 9160
```

可变长度参数

总结：

- 在声明函数时，通过带星的参数，如*param1，允许向函数传递可变数量的实参。调用函数时，从*后所有的参数被收集为一个元组
- 在声明函数时，也可以通过带双星的参数，如**param2，允许向函数传递可变数量的实参。调用函数时，从**后所有的参数被收集为一个字典

可变长度参数

- *param 用法

```
def one_star(*p):
        print(p)

one_star(1)
one_star(1,2,3)
one_star([1,2,3])
```

```
(1,)
(1, 2, 3)
([1, 2, 3],)
```

- **param 用法

```
def two_stars(**p):
        for item in p.items():
                print(item)

two_stars(x=1,y=2,z=3)
two_stars(1,2,3)
```

```
('x', 1)
('y', 2)
('z', 3)
```

```
TypeError: two_stars() takes
0 positional arguments but 3
were given
```

可变长度参数

- 传递参数时，可以通过加星号*进行序列解包

```
def sum(a, b, c):
        print(a+b+c)

lis = [1, 2, 3]
sum(*lis) #6

tup = (1, 2, 3)
sum(*tup) #6

Set = {1, 2, 3}
sum(*Set) #6
```

```
dic = {1:'a', 2:'b', 3:'c'}
sum(*dic.keys()) #6
sum(*dic.values()) #abc
```

可变长度参数

```
def my_sum3(a, b, *c):          #各数字累加和
    total = a + b
    for n in c:
        total = total + n
    return total
print(my_sum3(1, 2))            #计算1+2
print(my_sum3(1, 2, 3, 4, 5))     #计算1+2+3+4+5
print(my_sum3(1, 2, 3, 4, 5, 6, 7))  #计算1+2+3+4+5+6+7
```

程序运行结果如下：

```
3
15
28
```

参数类型检查

- 定义函数时，不用限定其参数和返回值的类型
  - 这种灵活性可以实现多态性，即允许函数适用于不同类型的对象，例如，my_average(a,b)函数，即可以返回两个int对象的平均值，也可以返回两个float对象的平均值
- 当使用不支持的类型参数调用函数时，则会产生错误。例如，my_average(a,b)函数传递的参数为str对象时，Python在运行时将抛出错误TypeError
  - 用户调用函数时必须理解并保证传入正确类型的参数值

函数式编程
Functional Programming

函数式编程
Functional Programming

不同的编程范式

- **面向过程编程**
  - -程序是一系列指令，告诉电脑做什么。示例：C、Pascal、Unix shell。
- **面向对象编程**
  - -计算机程序由单个能够起到子程序作用的单元或对象组合而成。示例：Java，C++。
- **函数式编程**
  - - 程序分解成一函数组，每个函数接受输入并产生输出，而没有内部状态。示例：Haskell。

Python支持多种范型

函数式编程
Functional Programming

```
# 面向过程 - "程序流"
def get_odds(arr):
        ret_list = []
        for elem in arr:
                if elem % 2 == 1:
                        ret_list.append(elem)
        return ret_list

# 函数式 - "程序由一组函数组成"，wow～
def get_odds(arr):
        return list(filter(lambda elem: elem % 2 == 0, arr)
```

为什么选择函数式编程

- 简化调试，以便更容易找到故障点。
- 更短、更清晰的代码。
- 模块化——函数通常是可重用的，支持逐模块测试和调试。

Lambdas

Inline, Anonymous Functions

- lambda是一种简便的、在同一行中定义函数的方法。
- lambda实际上生成一个函数对象

➡ 匿名的动态函数，可以**作为参数传递给其他函数**。

`lambda params: expression`

```
>>> #检查一对元组中的第一项是否大于第二项
>>> lambda tup: tup[0] > tup[1]
```

定制特别的功能

```
>>> #在成对的列表中寻找第二个元素的最大值。
>>> pairs = [(3, 2), (-1, 5), (4, 4)]
>>> max(pairs, key=lambda tup: tup[1])
(-1, 5)
```

Lambdas
Inline, Anonymous Functions

练习

```
>>> str = ["I","love","Shenzhen","University"]
>>> max(str, key=lambda aa: len(aa))
??
```

```
>>> str = ["I","love","Shenzhen","University"]
>>> sorted(str, key=lambda aa: aa[0])
??
```

Lambdas
Inline, Anonymous Functions

练习

```
>>> str = ["I","love","Shenzhen","University"]
>>> max(str, key=lambda aa: len(aa))
'University'
```

```
>>> str = ["I","love","Shenzhen","University"]
>>> sorted(str, key=lambda aa: aa[0])
['I', 'Shenzhen', 'University', 'love']
```

Map

- map (function，sequence)
  - ✓ 将函数function按顺序应用于sequence的每个项，结果作为列表返回。
  - ✓ 如果函数function支持, 可以提供多个参数。

```
def square(x):
        return x**2

list(map(square, range(0,11)))
#[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

注意喔~
不是square()
也不是square(x)

```
def expo(x, y):
    return x**y

list(map(expo, range(0,5), range(0,5)))
#[1，1，4，27，256]
```

Map

map的好处是什么？

```
#求每个字符串的长度，并返回一个列表

#最最直接的方法
def length_of_all_elements(arr):
        ret_arr = []

        for elem in arr:
                ret_arr.append(len(elem))

        return ret_arr

>>> length_of_all_elements(["Parth", "Unicorn", "Michael"])
[5, 7, 7]
```

Map

列表推导式 (list comprehensions)

```
[f(x) for x in iterable]
```

```
#求每个字符串的长度，并返回一个列表

#列表推导式
def length_of_all_elements(arr):
        return [len(elem) for elem in arr]
```

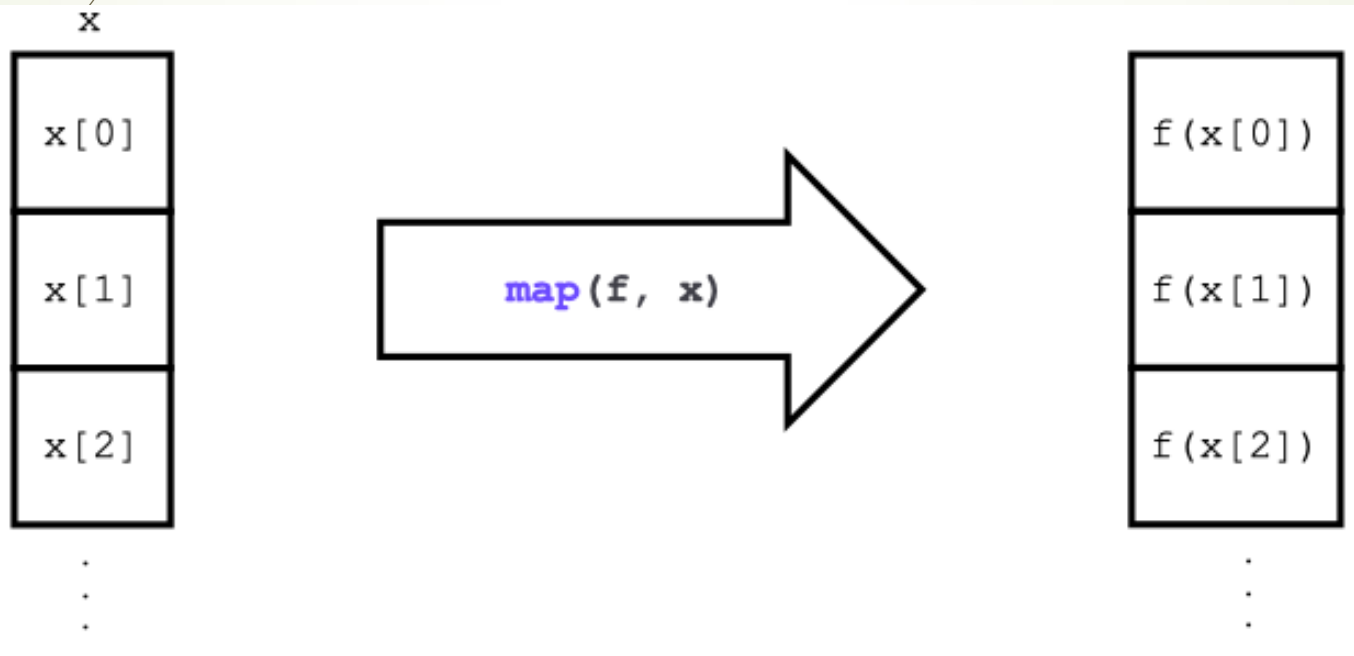
需要提及列表中的元素

```
>>>length_of_all_elements(["Parth","Unicorn", "Michael"])
[5, 7, 7]
```

Map

其实，是否可以不用提及列表中的每个元素？

因为每个元素做的事情（函数）都一样呀？



Map

map (function，sequence)

- 将function作用于sequence里的每一个元素，返回经过函数 function之后的结果。
- 没有提及sequence中的元素。

```
#求每个字符串的长度，并返回一个列表
def length_of_all_elements(arr): #list comprehension
        return [len(elem) for elem in arr]

def length_of_all_elements(arr):#map
        return list(map(len, arr))

>>>length_of_all_elements(["Parth",  "Unicorn",  "Michael"])
[5, 7, 7]
```

Map

练习：

```
def foo(arr):
        return list(map(max, arr))

>>>tuple_list = [(1,3,9),(8,0,6,9),(6,1,2,3)]
>>>foo(tuple_list)

???
```

Map

练习：

```
def foo(arr):
        return list(map(max, arr))

>>>tuple_list = [(1,3,9),(8,0,6,9),(6,1,2,3)]
>>>foo(tuple_list)

[9,9,6]
```

Filter

提取、筛选序列中满足一定条件的元素

```
#寻找以m开头的字符串

#最最直接的方法
def starts_with_m(arr):
        ret_arr = []
        for elem in arr:
                if elem[0].lower() == "m":
                        ret_arr.append(elem)
        return ret_arr

>>> starts_with_m(["Michael", "Parth"])
["Michael"]
```

Filter

推导式 **[x for x in iterable if condition]**

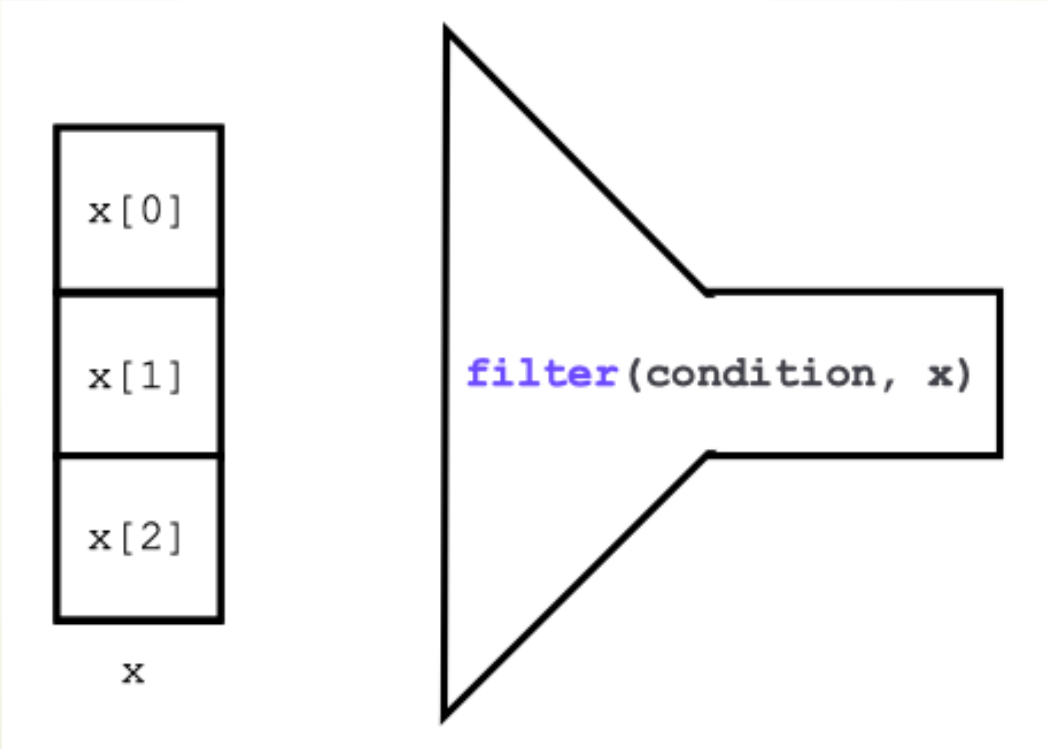
```
#寻找以m开头的字符串
def starts_with_m(arr):
        return [x for x in arr if x[0].lower() == "m"]

>>>starts_with_m(["Michael", "Parth"])
["Michael"]
```

Filter

filter(*function*, *sequence*)：

从sequence所有item中筛选function(item)为True的item。



Filter

filter(*function*, *sequence*)：

从sequence所有item中筛选function(item)为True的item。

```
#寻找以m开头的字符串

def starts_with_M(arr): #filter
        return  list(filter(lambda  word:  word[0].lower()
== "m", arr))

>>> starts_with_M(["Michael", "Parth"])
["Michael"]
```

Filter

再来一个例子

```
def even(x):
    if x % 2 == 0:
        return True
    else:
        return False

list(filter(even, range(0,30)))
```

输出:
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]

内存与运行速度的比较

内存

- 列表推导式缓冲所有计算结果
- map/filter仅在被调用时才计算元素（更节省内存）

运行速度

- 列表推导式没有函数调用开销
- 向map或filter传递lambda：调用lambda会带来额外的开销
- map或filter偶尔会快一些，但通常函数的调用开销使得它们更慢

Filter

练习

```
def rule(x):
    if ((x % 5 == 0) or(x % 6 ==0)) and
       (not (x % 5 == 0 and x % 6 ==0)):
        return True
    else:
        return False

list(filter(rule, range(0,50)))
```

输出:
? ?

Filter

练习

```
def rule(x):
    if ((x % 5 == 0) or(x % 6 ==0)) and
       (not (x % 5 == 0 and x % 6 ==0)):
        return True
    else:
        return False

list(filter(rule, range(0,50)))
```

输出:
[5, 6, 10, 12, 15, 18, 20, 24, 25, 35, 36, 40, 42, 45, 48]

Reduce

- reduce(*function*, *sequence*)
  - ✓ 用传给 reduce 中的函数 function（有两个参数）先对*sequence* 中的第 1、2 个元素进行操作，得到的结果再与第三个数据用 function 函数运算，最后得到一个结果。

```
# reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)

from functools import reduce
        #python 3;  python 2 不需要import

def factorial(x, y):
    return x*y

print(reduce(factorial, range(1,5)))
```

输出：
24 #不是120？

Map Revisit

```
#求每个字符串的长度，并返回一个列表
def length_of_all_elements(arr):
        return list(map(len, arr))

>>>length_of_all_elements(["Parth",   "Unicorn",   "Michael"])
[5, 7, 7]


def length_of_all_elements(arr):
        return map(len, arr)                    #不转换成list会怎么样？？

>>>length_of_all_elements(["Parth","Unicorn", "Michael"])
???
```

迭代器(iterators)

```
#求每个字符串的长度，并返回一个列表
def length_of_all_elements(arr):#map
        return map(len, arr)                    #不转换成list会怎么样？？

>>>length_of_all_elements(["Parth","Unicorn", "Michael"])
<map object at 0x107fd3d30>
```

- map和filter是**迭代器(iterators)**：表示有限或者无限的数据流。
- 使用next() 函数遍历迭代器中的所有元素。
  - 终止时引发StopIteration错误。
- 使用iter() 在数据结构上构建迭代器。
  - 例如，iter([1，2，3]) 在列表上构建一个迭代器

迭代器(iterators)

```
>>> names = ["Parth", "Michael", "Unicorn"]
>>> length_filter = filter(lambda word:
len(word) >= 7, names)

>>> next(length_filter)
"Michael"
>>> next(length_filter)
"Unicorn"
>>> next(length_filter)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
StopIteration
>>>
```

迭代器(iterators)

```
# 这段代码...
for data in data_source:
        do_something_to(data)


# 等价于...
data_iter = iter(data_source)
while True:
        try:
                data = next(data_iter)
        except StopIteration:
                break
        else:
                do_something_to(data)
```

使用iter() 在数据结构上构建迭代器。

生成器(generators)

- 一个普通的函数经常返回一个值：无论它是列表、整数还是其他对象。
- 一个函数**能否产生一系列值呢？**
- **生成器**: 返回能产生数据流的迭代器。
- 生成器函数需要使用 **yield** 语句。

```
# 用生成器产生 Fibonacci 序列!
def fib():
        a, b = 0, 1
        while True:
                a, b = b, a+b
                yield a
```

生成器(generators)

```
# 用生成器产生 Fibonacci 序列!
def fib():
        a, b = 0, 1
        while True:
                a, b = b, a+b
                yield a                 #这个生成器创建一个无限的序列

>>> g = fib()
>>> type(g)
<class 'generator'>
>>> next(g)
1
>>> next(g)
1                       # next(g) : 2, 3, 5, 8, 13, 21, 34...
>>> max(g)              # 会输出什么？？
```

生成器(generators)

- 我们可以使用生成器**以有限的方式表示无限的数据流**。
- 有时候无需处理整个Fibonacci序列——它是无穷的——生成器允许我们**根据需要**对序列的元素执行计算。
- 避免函数调用，减少内存缓冲

```
def fib():
        a, b = 0, 1
        while True:
                a, b = b, a+b
                yield a
#生成器创建一个无限的序列
```

```
def fibs_under(n):
        for num in fib():
                if num > n:
                        break
                print(num)
#按照需求创建有限的序列
```

函数作为参数

map (function，sequence）
filter (function，sequence）
reduce (function，sequence）

```
#我们也可以编写自己的函数，将函数作为参数
def do_twice(function, *args):
    function(*args)
    function(*args)

>>> do_twice(print, "Shenzhen is a good place")
Shenzhen is a good place
Shenzhen is a good place
```

函数作为参数

```
def shout(text):
  return text.upper()

def whisper(text):
  return text.lower()

def greet(func):
  # storing the function in a variable
  greeting = func("Hi, I am created by a function passed as an argument.")
  print(greeting)

greet(shout)
greet(whisper)
```

```
HI, I AM CREATED BY A FUNCTION PASSED AS AN ARGUMENT.
hi, I am created by a function passed as an argument.
```

函数作为返回值

```
def make_divisibility_test(n):
        def is_divisible_by(m):
                return m % n == 0
        return is_divisible_by

>>> div_test = make_divisibility_test(5)

>>> div_test(256)
False
>>> div_test(10)
True
>>>
```

Decorators: Best of Both Worlds

函数作为参数

函数作为返回值

- decorators 装饰器
  - ✓ 接受一个函数作为参数
  - ✓ 修改该函数
  - ✓ 然后返回修改后的版本

装饰器 Decorators

```
def debug(function):
        def modified_function(*args):
                print("Arguments:", args)
                return function(*args)
        return modified_function


def foo(a, b, c):
        print ((a + b) * c)

>>> foo = debug(foo)          #decorator函数来显式地装饰foo
>>> foo(2, 3, 1)
Arguments: (2, 3, 1)
5

>>> foo(2, 1, 3)
Arguments: (2, 1, 3)
9
```

装饰器 Decorators

```
def debug(function):
        def modified_function(*args):
                print("Arguments:", args)
                return function(*args)
        return modified_function


@debug                          #在函数声明时应用一个decorator
def foo(a, b, c):
        print ((a + b) * c)

>>> foo(2, 3, 1)
Arguments: (2, 3, 1)
5

>>> foo(2, 1, 3)
Arguments: (2, 1, 3)
9
```

装饰器 Decorators

```
def p_decorat(func):
    def func_wrapper(name):
        return "<p>" + func(name) + "</p>"
    return func_wrapper
```

```
def say_hello(name):
    return "Hello, " + str(name) + "!"

my_say_hello = p_decorat(say_hello)

print(my_say_hello("John") )
```

```
@p_decorat
def say_hello(name):
    return "Hello, " + str(name) + "!"

print(say_hello("John") )
```