第六章 函数和模块

6.1 创建函数

6.2 函数的参数

6.3 作用域

6.4 递归

6.5 函数式编程

函数

- 可以编写一小段代码来进行计算工作，如计算斐波那契数列

```
fibs = [0, 1]
for i in range(8):
    fibs.append(fibs[-2] + fibs[-1])
print(fibs)
```

- 当你需要在多处使用同一段代码或者更为复杂的代码时，无需每次将这些代码复制，程序员会通过抽象的方式来解决，即创建函数

创建函数

- 函数是可以调用的，用于执行某种行为并且返回值，可以使用 hasattr(x, '__call__') 来判断函数是否可调用

```
>>> import math
>>> x = 1
>>> hasattr(x, '__call__')
False
>>> hasattr(math.sqrt, '__call__')
True
```

创建函数

- 定义函数使用def语句即可

```
>>> def hello(name):
        return('Hello, ' +  name +  '!')
```

- ***执行上面这段程序***后可以得到一个名为hello的函数，它接受一个名为name的参数，返回一个问候语；可以像使用其它内建函数那样调用该函数

```
>>> greetings = hello('Mr. Gumby')
>>> print(greetings)
Hello, Mr. Gumby!
```

调用函数

- 将前面用于计算斐波那契数列的代码放入到一个函数fibs中

```
def fibs(num):
    result = [0, 1]
    for i in range(num - 2):
        result.append(result[-2] + result[-1])
    return result
```

- **执行上面这段程序**后可以得到一个名为fibs的函数，编译器也就知道如何计算斐波那契数列，以后再需要同样功能，只需调用fibs函数并提供参数num的值就可以

```
>>> fibs(15)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

函数的文档字符串

- 如果需要给函数添加文档，可以加入以‘#’开头的注释，此外也可以直接写入解释性的字符串
- 如果在函数开头写下字符串，这些字符串会作为函数的一部分进行存储，称之为**文档字符串**

```
def square(x):
    'Calculates the square of the number x.'
    return x * x
```

函数的文档字符串

- 当如前所示在函数中添加了文档字符串后，可以对函数的文档字符串进行访问（__doc__是函数属性，名字中是双下划线）

```
>>> square.__doc__
'Calculates the square of the number x.‘
```

- 在交互式解释器中可以使用内建函数help得到关于特定函数，包括其中文档字符串的信息

```
>>> help(square)
Help on function square in module __main__:

square(x)
    Calculates the square of the number x.
```

函数的返回值

- Python函数并非总是返回值，也就是说有的函数虽然有return语句但是其中没有包含具体的值，有的函数甚至没有return语句

```
def test_return():
    s = input('Please enter your name :')
    if s != '':
        print('Hello,', s, '!')
    else:
        print('Hello, stranger!')
    return
```

函数的返回值

```
>>> x = test_return()
Please enter your name :Gumby
Hello, Gumby !
>>> y = test_return()
Please enter your name :
Hello, stranger!
>>> (x, y)
(None, None)
```

- 前面的代码中return语句并未明确返回任何值，但是通过以打印的方式查看函数的返回值发现函数还是“返回”了None值（将return语句去掉后此结论仍然成立）

第六章 函数和模块

6.1 创建函数

6.2 函数的参数

6.3 作用域

6.4 递归

6.5 函数式编程

函数的参数

- 定义函数之后，其操作的值来自于调用函数提供的参数
- 函数定义的def语句中圆括号包含的变量叫做**函数的形参**，而调用函数时提供的值叫做**函数的实参**

函数的参数

- 参数只是变量而已，在函数内对参数赋予新值并不会改变外部任何变量的值

```
>>> def test_change(n):
        n = 'Mr. Gumby'
        print('Internal name :', n)


>>> name = 'Mrs. Smith'
>>> test_change(name)
Internal name : Mr. Gumby
>>> name
'Mrs. Smith'
```

函数参数的改变

- 不可变的对象类型，包括数字、字符串和元组等，作为函数的参数时依然不能被改变
- 可变对象类型（包括列表等）作为参数就可以被改变

```
>>> def test_change(n):
        n[0] = 'Mr. Gumby'


>>> names = ['Mrs.Smith', 'Mrs. Wills']
>>> test_change(names)
>>> names
['Mr. Gumby', 'Mrs. Wills']
```

函数参数的改变

- 当两个变量同时引用一个列表时，它们同时使用该列表

```
>>> names = ['Mrs. Smith', 'Mrs. Wills']
>>> n = names
>>> n[0] = 'Mr. Gumby'
>>> names
['Mr. Gumby', 'Mrs. Wills']
```

- 这时如果需要保留原列表的内容不变，可以复制列表的副本，通过分片操作比较直接且简单。此时，若对副本修改则不会影响原来的列表

```
>>> n = names[:]
>>> n == names
True
>>> n is names
False
```

函数示例

```
def init(data):
    'Initialize data.'
    data['first'] = {}
    data['middle'] = {}
    data['last'] = {}

def lookup(data, label, name):
    'Look up a name in data.'
    return(data[label].get(name))  # 如果字典中没有对应项,get将返回None
```

```
def store(data, full_name):
    'Store a full_name of someone in data.'
    names = full_name.split()
    if len(names) == 2: names.insert(1, ' ')  # 在名和姓中间插入一个空格
    labels = 'first', 'middle', 'last'
    for label, name in zip(labels, names): # zip函数进行并行迭代,返回一个元组的列表
        people = lookup(data, label, name)
        if people:
            people.append(full_name)
        else:  # 若字典中没有被查找项
            data[label][name] = [full_name]
```

- 在运行过前面定义的函数后可以调用它们

```
>>> MyNames = {}
>>> init(MyNames)
>>> store(MyNames, ‘Magnus Lie Hetland')
>>> lookup(MyNames, 'middle', 'Lie')
[Magnus Lie Hetland ']
```

不可变参数与可变参数

- 在某些语言（比如C++或Pascal）中，重新绑定参数并且使这些改变影响函数外的变量是非常平常的。在Python中却是不可能的！函数只能修改参数对象本身。

- 如果真的需要改变参数，则可以使用一点小技巧，即将值放置在列表中

```
>>> def inc(x): x[0] += 1

>>> foo = [10]
>>> inc(foo)
>>> foo
[11]
```

关键字参数和默认值

- 目前我们使用的参数都是**位置参数**，因为他们的位置比名字本身更为重要

- 考虑以下两个函数：

```
>>> def hello_1(greeting, name):
        print('%s, %s!' % (greeting, name))
>>> def hello_2(name， greeting):
        print(‘%s, %s!’ % (name, greeting))
```

- 以上两个函数的功能一致，只是参数顺序相反

```
>>> hello_1('Hello', 'World')
Hello, World!
>>> hello_2('Hello', 'World')
Hello, World!
```

关键字参数和默认值

- 有时参数的顺序是很难记住的，为了让事情简单些，可以使用**关键字参数**

```
>>> hello_1(greeting = 'Hello', name = 'World')
Hello, World!
```

- 这样顺序就完全没有影响了，只是参数名和值一定要对应

```
>>> hello_1(name = 'World', greeting = 'Hello')
Hello, World!
```

- 使用关键字参数可以让每个参数的含义变得更加清晰

关键字参数和默认值

- 关键字参数的重要作用还体现在可以给参数提供默认值，当参数具有默认值时，调用的时候就不用再提供参数了

```
>>> hello_3()
Hello, World!

>>> hello_3(name = 'Mr. Gumby')
Hello, Mr. Gumby!
```

关键字参数和位置参数联合使用

- 关键字参数和位置参数是可以联合使用的，此时**应该把位置参数放在前面，关键字参数放在后面**，否则解释器无法判断哪些是位置参数

```
>>> def hello_4(name, greeting = 'Hello', punctuation = '!'):
        print('%s, %s%s' % (greeting, name, punctuation))

>>> hello_4('Mars')
Hello, Mars!
>>> hello_4('Mars', 'Howdy')
Howdy, Mars!
>>> hello_4('Mars', greeting = 'Top of the morning to ya')
Top of the morning to ya, Mars!
>>> hello_4(greeting = 'Hi', 'Mars')
SyntaxError: positional argument follows keyword argument
```

收集参数

- 有时让用户提供任意数量的参数是有用的

```
>>> def print_param(*params):
        print(params)


>>> print_param('Testing')
('Testing',)
>>> print_param(1, 2, 3)
(1, 2, 3)
```

- 参数前的星号（*）表示将所有值放在同一元组中

收集参数

- 星号（*）的意思是“收集其余的位置参数”

```
>>> def print_param_2(title, *params):
        print(title)
        print(params)


>>> print_param_2('Params:', 1, 2, 3)
Params:
(1, 2, 3)
>>> print_param_2('Nothing:')
Nothing:
()
>>>
```

收集参数

- 能够处理关键字参数的收集操作如下

```
>>> def print_param_3(**params):
        print(params)


>>> print_param_3(x = 1, y = 2, z = 3)
{'z': 3, 'x': 1, 'y': 2}
```

- 此时返回的是字典而不是元组

收集多种参数

```
>>> def print_param_4(x, y, z = 3, *pospar, **keypar):
        print(x, y, z)
        print(pospar)
        print(keypar)


>>> print_param_4(1, 2, 3, 4, 5, foo = 1, bar = 2)
1 2 3
(4, 5)
{'foo': 1, 'bar': 2}
```

```
>>> def with_stars(**kwds):
        print(kwds['name'], 'is', kwds['age'], 'years old.')

>>> def without_stars(kwds):
        print(kwds['name'], 'is', kwds['age'], 'years old.')

>>> args = {'name':'Mr. Gumby', 'age':42}
>>> with_stars(**args)
Mr. Gumby is 42 years old.
>>> without_stars(args)
Mr. Gumby is 42 years old.
```

- 由上例可以看出，星号（*）只在定义函数（允许使用不定数目的参数）或者调用（“分割”字典或元组）时才有用

第六章 函数和模块

6.1 创建函数

6.2 函数的参数

6.3 作用域

6.4 递归

6.5 函数式编程

作用域

```
>>> x = 1
>>> scope = vars()
>>> scope['x']
1
>>> scope['x'] += 1
>>> x
2
```

- 变量和对应的值存放于一个“不可见”的字典中，也叫**命名空间**或者**作用域**，每个函数调用都会创建一个新的作用域
- 使用内建函数vars()可以返回这个字典

```
>>> vars()

{‘__spec__’: None, ‘__package__’: None, ‘__name__’: ‘__main__’, ‘x’: 2,……}
```

局部变量

- 函数内的变量被称为**局部变量**，对局部变量的赋值只在内部作用域中起作用
- 参数的工作原理类似于局部变量，所以用全局变量的名字作为参数名并没有问题

```
>>> def output(x): x = 4; print(x)

>>> x = 1
>>> output(x)
4
>>> x
1
```

全局变量的屏蔽

- 一般来说可以在函数内部直接访问全局变量

```
>>> def combine(param): print(param + ext)

>>> ext = 'berry'
>>> combine('blue')
blueberry
```

- 但是如果局部变量或者参数的名字和全局变量名相同，后者将被屏蔽，此时可以通过globals()函数先返回全局变量的字典，再从中取出全局变量的值

```
>>> def combine(param): print(param + globals()['param'])

>>> param = 'berry'
>>> combine('blue')
blueberry
```

重绑定全局变量

- 如果在函数内部将值赋给某个变量，它将自动成为局部变量，但如果希望其成为全局变量呢？

```
>>> x = 1
>>> def change_global():
        global x
        x = x + 1


>>> change_global()
>>> x
2
```

嵌套作用域

- Python的函数可以嵌套，可用于通过一个函数“创建”另外一个函数

```
>>> def multiplier(factor):
        def multiplyByFactor(number):
                return number * factor
        return multiplyByFactor  # 函数multiplyByFactor被返回，不是被调用

>>> double = multiplier(2)  # 调用外层multiplier函数后，来自外部作用域的
                            #  变量factor将被内层函数访问
>>> double(6)            # 相当于multiplier(2)(6)
12
>>> multiplier(3)(4)
12
```

第六章 函数和模块

6.1 创建函数

6.2 函数的参数

6.3 作用域

6.4 递归

6.5 函数式编程

递归函数

- **递归**，即函数直接或间接调用自身

- 每次函数被调用时，针对这个调用的新命名空间会被创建，意味着当函数调用“自身”时，实际上运行的是两个不同的函数

- 考虑求n的阶乘的函数，如果用循环实现

```
>>> def factorial(n):
        result = n
        for i in range(1, n):
                result *= i
        return result
```

递归函数

- 如果用递归方法实现求n的阶乘

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

- 考虑计算一个数n的整数幂，也可使用递归函数实现

```
def power(x, n):
    if n == 0:
        return 1
    else:
        return x * power(x, n - 1)
```

二分法查找

```
def binarysearch(sequence, number, lower, upper):
    'Searches number in a sorted sequence.'
    mid = (lower + upper) // 2

    if lower > upper:
        return 'No such a number in the sequence.‘

    if sequence[mid] == number:
        return 'I found it and its index is ' + str(mid) + '.'
    elif sequence[mid] > number:
        return binarysearch(sequence, number, lower, mid - 1)
    else:
        return binarysearch(sequence, number, mid + 1, upper)
>>> seq = [34, 67, 8, 123, 4, 100, 95]
>>> seq.sort()   # 需要先将序列中的元素排序
>>> binarysearch(seq, 34, 0, len(seq))
2
```

第六章 函数和模块

6.1 创建函数

6.2 函数的参数

6.3 作用域

6.4 递归

6.5 函数式编程

函数式编程

- **函数式编程**将问题分解成一系列的函数来解决，模块化是成功编程的关键，而函数式编程可以极大地改进模块化
  - 在函数编程中，编程人员有一个天然框架用来开发更小的、更简单的和更一般化的模块， 然后将它们组合在一起（百度百科）
- Python和C++一样支持**多重编程范式**（multi-paradigm language），即编写的程序和库既可以是过程化、面向对象的，也可以是函数式的
- 函数的使用方法和其它对象一样，可以分配给变量、作为参数传递以及从其它函数返回；Python为支持函数式编程提供了一些有用的函数，例如map、filter和reduce函数（在functools模块中）

函数式编程

- map函数可以将序列中的元素全部传递给函数

```
>>> list(map(str, range(10)))
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
#以上map()相当于执行[str(i) for i in range(10)]
```

- filter函数可以基于一个返回布尔值的函数对元素进行过滤

```
>>> def func(x):
        return x.isalnum()    # 判断字符串x是否仅包含字母和数字

>>> seq = ['foo', 'x41', '?!', '***']
>>> filter(func, seq)
<filter object at 0x02D25990>
>>> list(filter(func, seq))
['foo', 'x41']
```

lambda表达式

- 当需要的函数不存在时，除了可以使用def定义一个新的函数外，还可以考虑使用lambda语句
- lambda语句接收若干参数和一个使用这些参数的表达式，然后创建一个返回该表达式的值的匿名函数

```
>>> seq = ['foo', 'x41', '?!', '***']
>>> filter(lambda x:x.isalnum() , seq)
<filter object at 0x02D5C6F0>
>>> list(filter(lambda x:x.isalnum() , seq))
['foo', 'x41']
```

函数式编程

- reduce函数将序列中的前两个元素与给定的函数联合使用，然后将返回值和第三个元素继续联合使用，直到整个序列处理完毕

```
>>> from functools import *
>>> numbers = [1, 2, 3, 4,5]
>>> reduce(lambda x, y: x + y, numbers)
15
```

lambda表达式的使用

- 定义函数时，使用def语句还是lambda语句是一个编程风格的问题；但是在某些情况下，并不推荐使用lambda语句，其理由如下
  - lambda语句在可以定义的函数种类方面具有局限性，例如其结果必须是一个可计算的表达式，诸如if…elif…else这样的条件语句就无法表达
  - 过多使用lambda语句还会使得表达式变得非常复杂难懂

```
>>> items = ['abc', 'def', 'ghi']
>>> total = functools.reduce(lambda a, b:(0, a[1] + b[1]), items)[1]
>>> total
'beh'
```

lambda表达式的使用

```
>>> items = ['abc', 'def', 'ghi']
>>> total = functools.reduce(lambda a, b:(0, a[1] + b[1]), items)[1]
>>> total
'beh‘
```

上面例子可以改写为

```
>>> def combine(a, b):
        return(0, a[1] + b[1])

>>> total = functools.reduce(combine, items)[1]
```

或者使用for循环，

```
result = ''
for a, b, c in items:                    # 序列解包
    result += b          # 将每个字符串拆分成3个长度为1的字符串

total = (0, result)[1]
```

小结

- **抽象**：是隐藏多余细节的艺术，定义处理细节的函数可以让程序更加抽象
- **函数定义**：可以使用def语句，他们是由语句组成的块，可以通过形参获取值，也可返回一个或多个值作为运算的结果
- **参数**：向函数提供信息，包括位置参数和关键字参数，可以为形参设置默认值
- **作用域**：用于存放变量和值的映射，也叫命名空间；Python中有全局作用域和局部作用域

小结

- **递归**：函数直接或间接调用自身，一切用递归实现的功能都可以用循环来实现，但是递归函数可读性更强
- **函数式编程**：Python有一些支持函数式编程的机制，包括lambda表达式和map、filter及reduce函数

本章的新函数

map(func, seq[, seq…]) 对序列中每个元素应用函数

filter(func, seq) 返回函数值为真的元素的列表

reduce(func, seq[, initial]) 对序列中的元素反复调用函数，等同于

func(func(func(seq[0], seq[1]), seq[2],…)

题目1：请编写一段代码，其中至少包含以下两个函数，

# 将摄氏度转换成华氏度
def celsiusToFahrenhei(celsius):
# 将华氏度转换成摄氏度
def fahrenheitToCelsius(fahrenheit):
其中涉及到的转换公式如下：
celsius = (5/9) * (fahrenheit-32)
fahrenheit = (9/5) * celsius + 32

然后写一段代码调用以上两个函数并显示如下信息：

Celsius	Fahrenheit	Fahrenheit	Celsius
40.0	104.0	120.0	48.89
39.0	102.2	110.0	43.33
31.0	87.8	30.0	-1.11

题目2：孪生素数（素数也叫质数）是一对相差2的素数，例如3和5，5和7，11和13都是孪生素数。请写一段代码找到所有1000以内的孪生素数，结果显示如下：

（3，5）

（5，7）

……

要求：定义一个函数isPrime(num)，该函数判断给定数num是否是素数（或质数）