# 第5章　函数

# 第5章 函数

- 函数是一种重要的代码复用技术。将可能需要反复执行的代码封装为函数，然后在需要执行该功能的地方进行调用，不仅可以实现代码重复使用，更重要的是可以保证代码的一致性，修改函数定义后所有调用均受到影响。
- 设计函数时，应注意提高模块的内聚性，同时降低模块之间的隐式耦合。
- 在编写函数时，应严格按照接口规范和定义编写代码，不应增加额外的功能，尽量减少副作用。
- 大型项目开发时，应把所有常用的功能函数封装到一个模块中，并放置于项目顶层文件夹或专门的文件夹中。

# 5.1.1 函数定义与调用基本语法

❖函数定义语法：

```
def 函数名([参数列表]):
    '''注释'''
    函数体
```

❖注意事项

✓ 函数形参不需要声明类型（也可以声明类型，但实际不起作用），也不需要指定函数返回值类型

✓ 即使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

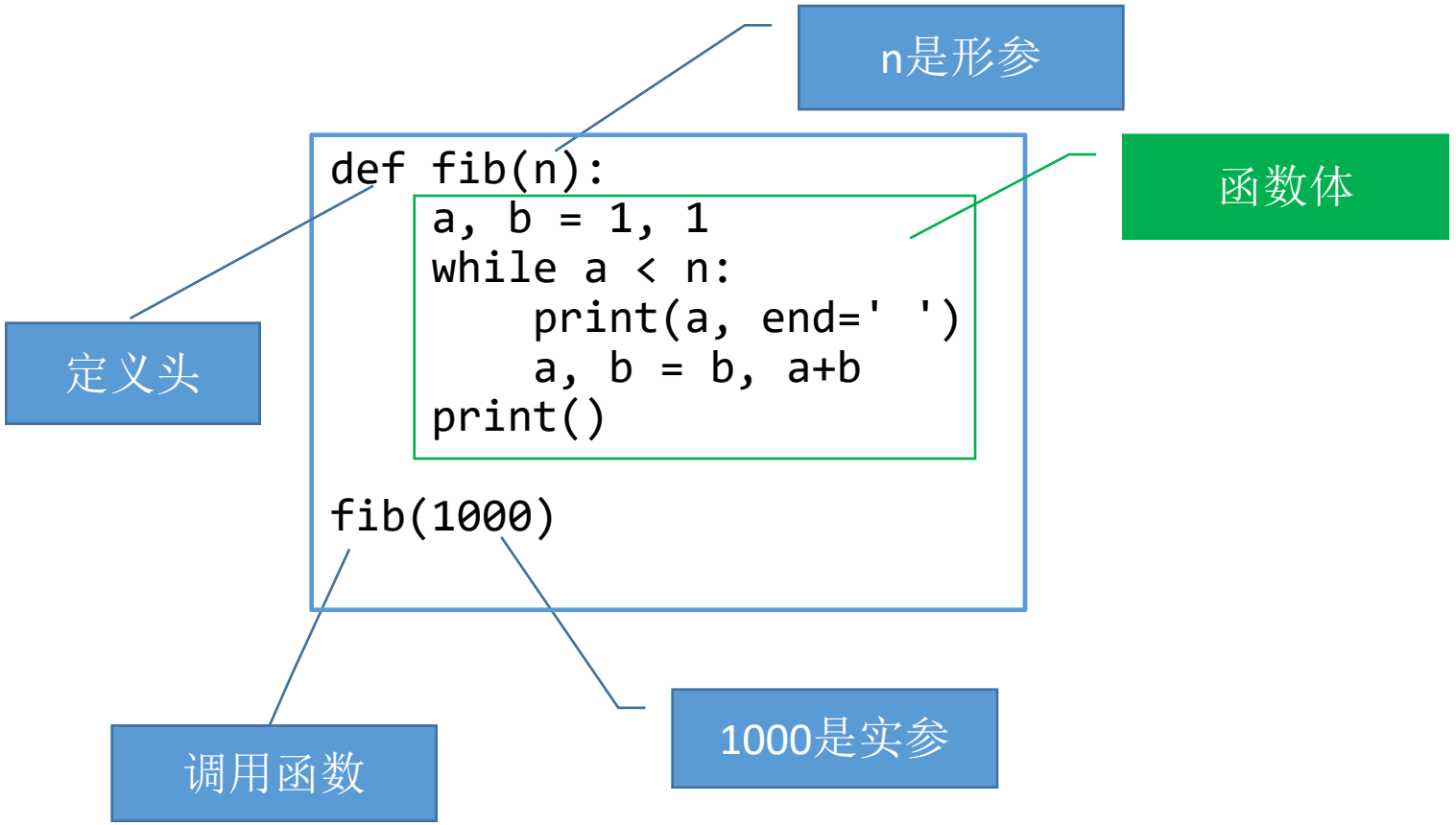
✓ Python允许嵌套定义函数

# 5.1.1 函数定义与调用基本语法

- 在Python中，定义函数时也不需要声明函数的返回值类型，而是使用return语句结束函数执行的同时返回任意类型的值，函数返回值类型与return语句返回表达式的类型一致。
- 不论return语句出现在函数的什么位置，一旦得到执行将直接结束函数的执行。
- 如果函数没有return语句、有return语句但是没有执行到或者执行了不返回任何值的return语句，解释器都会认为该函数以return None结束，即返回空值。

# 5.1.1 函数定义与调用基本语法

- **应用：**编写生成斐波那契数列的函数并调用。



# 5.1.1 函数定义与调用基本语法

- 在定义函数时，开头部分的注释（文档字符串）并不是必需的，但如果为函数的定义加上注释的话，可以为用户提供友好的提示。

```
>>> def fib(n):
        '''accept an integer n.
           return the numbers less than n in Fibonacci sequence.'''
        a, b = 1, 1
        while a < n:
            print(a, end=' ')
            a, b = b, a+b
        print()


>>> fib(
        (n)
        accept an integer n.
        return the numbers less than n in Fibonacci sequence.
```

# 5.1.2 函数嵌套定义、可调用对象与修饰器

（1）函数嵌套定义

```
>>> def myMap(iterable, op, value):          # 自定义函数
    if op not in '+-*/':
        return 'Error operator'
    def nested(item):                         # 嵌套定义函数
        return eval(repr(item)+op+repr(value))

    return map(nested, iterable)              # 使用在函数内部定义的函数

>>> list(myMap(range(5), '+', 5))             # 调用外部函数，不需要关心其内部实现
[5, 6, 7, 8, 9]
>>> list(myMap(range(5), '-', 5))
[-5, -4, -3, -2, -1]
```

# 5.1.2 函数嵌套定义、可调用对象与修饰器

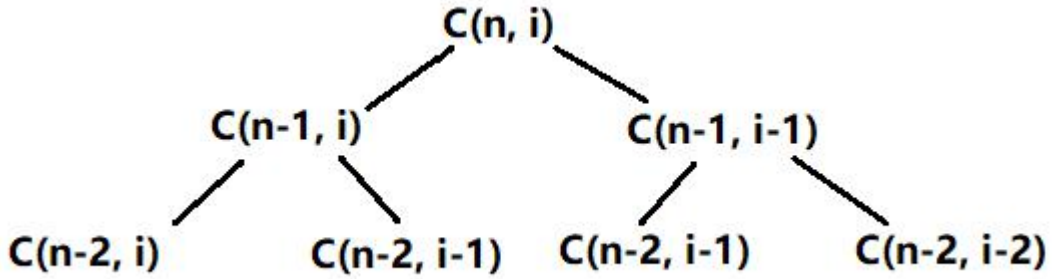
- **应用：**用函数嵌套定义和递归实现帕斯卡公式C(n,i) = C(n-1, i) + C(n-1, i-1)，进行组合数C(n,i)的快速求解。[code\5.1.2_1.py](code\5.1.2_1.py)

```
def f2(n, i):
    cache2 = dict()

    def f(n, i):
        if n==i or i==0:
            return 1
        elif (n,i) not in cache2:
            cache2[(n,i)] = f(n-1, i) + f(n-1, i-1)
        return cache2[(n,i)]

    return f(n,i)

print(f2(60,9))
```



# 5.1.2 函数嵌套定义、可调用对象与修饰器

- 使用标准库提供的缓冲机制进行改写和优化。code\5.1.2_2.py

```
from functools import lru_cache

@lru_cache(maxsize=64)
def cni(n, i):
    if n==i or i==0:
        return 1
    return cni(n-1, i) + cni(n-1, i-1)

print(cni(60, 9))
```

# 5.1.2 函数嵌套定义、可调用对象与修饰器

（2）可调用对象

- 函数和lambda表达式是比较常用的Python可调用对象。
- 由于构造方法的存在，类也是可调用的。像int()、float()、str()、list()、tuple()、dict()、set()这样的用法实际上都是调用了类的构造方法。
- 另外，任何包含__call__()方法的类的对象也是可调用的。

# 5.1.2 函数嵌套定义、可调用对象与修饰器

- 嵌套函数定义中，外层函数可以返回内层函数的名称，用来创建一个可调用对象。

```
def linear(a, b):
    def result(x):                  # 在Python中，函数是可以嵌套定义的
        return a * x + b
    return result                   # 返回可被调用的函数
```

# 5.1.2 函数嵌套定义、可调用对象与修饰器

- 如果一个类的定义中实现了特殊方法__call__()，那么这个类的所有对象都是可调用对象。

```
class linear:
    def __init__(self, a, b):
        self.a, self.b = a, b

    def __call__(self, x):          # 这里是关键
        return self.a * x + self.b
```

# 5.1.2 函数嵌套定义、可调用对象与修饰器

- 使用上面的嵌套函数和类这两种方式中任何一个，都可以通过以下的方式来创建一个可调用对象：

```
taxes = linear(0.3, 2)
```

- 然后通过下面的方式来调用该对象：

```
taxes(5)
```

# 5.1.2 函数嵌套定义、可调用对象与修饰器

（3）修饰器

- 修饰器（decorator）是函数嵌套定义的另一个重要应用。修饰器本质上也是一个函数，只不过这个函数接收其他函数作为参数并对其进行一定的修改之后使用新函数替换原来的函数。
- Python面向对象程序设计中的静态方法、类方法、抽象方法、属性等都是通过修饰器实现的。

code\5.1.2_3.py

# 5.1.2 函数嵌套定义、可调用对象与修饰器

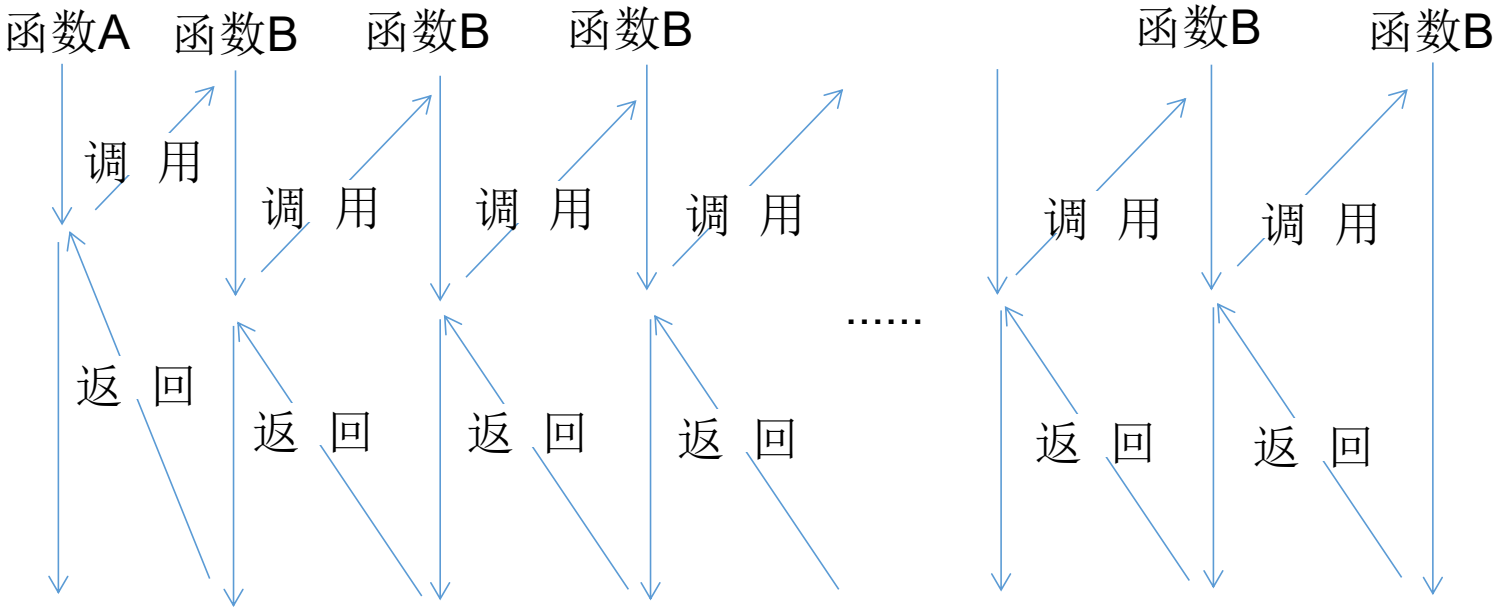
```
def before(func):                       # 定义修饰器
    def wrapper(*args, **kwargs):       # 在内层函数中对被修饰的函数进行一定修改
        print('Before function called.')
        return func(*args, **kwargs)
    return wrapper                      # 返回内层函数

def after(func):                        # 定义修饰器
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        print('After function called.')
        return result
    return wrapper

@before
@after
def test():                             # 同时使用两个修饰器改造函数，距离近的先起作用
    print(3)
# 调用被修饰的函数
test()
```

# 5.1.3 函数递归调用

- 函数的递归调用是函数调用的一种特殊情况，函数调用自己，自己再调用自己，自己再调用自己，...，当某个条件得到满足的时候就不再调用了，然后再一层一层地返回直到该函数第一次调用的位置。



# 5.1.3 函数递归调用

- 定义递归函数时应注意：
  - 保持问题性质不变；
  - 问题规模越来越小；
  - 问题规模小到一定程度时可以直接解决，不需要再次递归；
  - 递归深度不能太深，否则会因为线程栈空间不足而导致内存崩溃。

# 5.1.3 函数递归调用

- **应用：**使用递归法对整数进行因数分解。code\5.1.3.py

```
from random import randint

def factors(num):
    # 每次都从2开始查找因数
    for i in range(2, int(num**0.5)+1):
        # 找到一个因数
        if num%i == 0:
            facs.append(i)
            # 对商继续分解，重复这个过程
            factors(num//i)
            # 注意，这个break非常重要
            break
    else:
        # 不可分解了，自身也是个因数
        facs.append(num)
```

# 5.1.3 函数递归调用

```
facs = []
n = randint(2, 10**8)
factors(n)
result = '*'.join(map(str, facs))
if n == eval(result):
    print(n, '= '+result)
```

# 5.2 函数参数

- 函数定义时圆括号内是使用逗号分隔开的形参列表（parameters），函数可以有多个参数，也可以没有参数，但定义和调用时一对圆括号必须要有。
- 调用函数时向其传递实参（arguments），根据不同的参数类型，将实参的引用传递给形参。
- 定义函数时不需要声明参数类型，解释器会根据实参的类型自动推断形参类型，在一定程度上类似于函数重载和泛型函数的功能。

# 5.2 函数参数

- 在函数内部直接修改形参的引用不会影响实参，例如：

```
>>> def addOne(a):
    print(id(a), ':', a)
    a += 1
    print(id(a), ':', a)

>>> v = 3
>>> id(v)
1599055008
>>> addOne(v)
1599055008 : 3
1599055040 : 4
>>> v
3
>>> id(v)
1599055008
```

注意：此时a的地址与v的地址相同

现在a的地址和v的地址不一样了

# 5.2 函数参数

- 可以通过下标或可变对象自身提供的原地修改方法在函数内部修改实参的值。

```
>>> def modify(v):                  # 使用下标修改列表元素值
    v[0] = v[0] + 1
>>> a = [2]
>>> modify(a)
>>> a
[3]
>>> def modify(v, item):            # 使用列表的方法为列表增加元素
    v.append(item)
>>> a = [2]
>>> modify(a, 3)
>>> a
[2, 3]
```

# 5.2 函数参数

- 也就是说，如果传递给函数的实参是可变对象，并且在函数内部使用下标或可变对象自身的方法增加、删除元素或修改元素时，实参也得到相应的修改。

```
>>> def modify(d):                    # 修改字典元素值或为字典增加元素
    d['age'] = 38
>>> a = {'name':'Dong', 'age':37, 'sex':'Male'}
>>> a
{'name': 'Dong', 'age': 37, 'sex': 'Male'}
>>> modify(a)
>>> a
{'name': 'Dong', 'age': 38, 'sex': 'Male'}
```

# 5.2.1 位置参数

- 位置参数（positional arguments）是比较常用的形式，调用函数时实参和形参的顺序必须严格一致，并且实参和形参的数量必须相同。

```
>>> def demo(a, b, c):
    print(a, b, c)

>>> demo(3, 4, 5)                  # 按位置传递参数
                                   # 第一个实参传递给第一个形参
                                   # 第二个实参传递给第二个形参，以此类推
3 4 5
>>> demo(3, 5, 4)
3 5 4
>>> demo(1, 2, 3, 4)               # 实参与形参数量必须相同
TypeError: demo() takes 3 positional arguments but 4 were given
```

# 5.2.2 默认值参数

- 带有默认值参数的函数定义语法如下：

```
def 函数名(……，形参名=默认值):
    函数体
```

- 在定义带有默认值参数的函数时，任何一个默认值参数的后面都不能再出现没有默认值的普通位置参数，否则会提示语法错。
- 在调用函数时是否为默认值参数传递实参是可选的，如果没有给带默认值的形参传递实参就使用函数定义时设置的默认值，如果为带有默认值的形参传递了实参就使用传递的实参。例如内置函数print()的参数sep、end，内置函数sorted()和列表方法sort()的参数key、reverse等。

# 5.2.2 默认值参数

- 可以使用“函数名.__defaults__”随时查看函数所有默认值参数的当前值，其返回值为一个元组，其中的元素依次表示每个默认值参数的当前值。

```
>>> def say(message, times=1):
    print((message+' ') * times)

>>> say.__defaults__
(1,)
```

# 5.2.2 默认值参数

- 参数默认值的引用是在函数定义时确定的，对于列表、字典这样可变类型的参数默认值，函数中基于引用的操作可能会导致逻辑错误。例如：

```
>>> def demo(newitem, old_list=[]):
    old_list.append(newitem)
    return old_list

>>> print(demo('5', [1, 2, 3, 4]))
[1, 2, 3, 4, '5']
>>> print(demo('aaa', ['a', 'b']))
['a', 'b', 'aaa']
>>> print(demo('a'))
['a']
>>> print(demo('b'))                    # 注意这里的输出结果
['a', 'b']
```

# 5.2.2 默认值参数

- 一般来说，要避免使用列表、字典、集合或其他可变对象作为函数参数默认值，对于上面的函数，更建议使用下面的写法。

```
def demo(newitem, old_list=None):
    if old_list is None:
        old_list = []
    old_list.append(newitem)
    return old_list
```

# 5.2.2 默认值参数

- 函数参数默认值的引用是在函数定义时确定的。

```
>>> i = 3
>>> def f(n=i):                    # 参数n的值仅取决于i的当前值
    print(n)
>>> f()
3
>>> i = 5                          # 函数定义后修改i的值不影响参数n的默认值
>>> f()
3
>>> i = 7
>>> f()
3
>>> def f(n=i):                    # 重新定义函数
    print(n)
>>> f()
7
```

# 5.2.3 关键参数

- 通过关键参数可以按名字传递参数，明确指定哪个实参传递给哪个形参，实参顺序可以和形参顺序不一致，避免了用户需要牢记参数位置和顺序的麻烦，使得参数传递更加灵活方便。

```
>>> def demo(a, b, c=5):
    print(a, b, c)
>>> demo(3, 7)
3 7 5
>>> demo(a=7, b=3, c=6)
7 3 6
>>> demo(c=8, a=9, b=0)
9 0 8
```

# 5.2.4 可变长度参数

- 可变长度参数主要有两种形式：定义函数时在形参名称前面加1个或2个星号。
  - *parameter用来接收多个位置参数并将其放在一个元组中，元组长度取决于实际接收的位置参数个数；
  - **parameter用来接收多个关键参数并存放到一个字典中，字典中元素个数取决于实际接收的关键参数个数。

# 5.2.4 可变长度参数

- *parameter的用法

```
>>> def demo(*p):
    print(p)

>>> demo(1,2,3)
(1, 2, 3)
>>> demo(1,2)
(1, 2)
>>> demo(1,2,3,4,5,6,7)
(1, 2, 3, 4, 5, 6, 7)
```

# 5.2.4 可变长度参数

- **parameter的用法

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

>>> demo(x=1, y=2, z=3)
('x', 1)
('y', 2)
('z', 3)
```

# 5.2.4 可变长度参数

- 几种不同类型的参数可以混合使用，但不建议这样做。

```
>>> def func_4(a, b, c=4, *aa, **bb):
    print(a,b,c)
    print(aa)
    print(bb)

>>> func_4(1, 2, 3, 4, 5, 6, 7, 8, 9, xx='1', yy='2', zz=3)
1 2 3
(4, 5, 6, 7, 8, 9)
{'xx': '1', 'yy': '2', 'zz': 3}
>>> func_4(1, 2, 3, 4, 5, 6, 7, xx='1', yy='2', zz=3)
1 2 3
(4, 5, 6, 7)
{'xx': '1', 'yy': '2', 'zz': 3}
```

# 5.2.5 传递参数时的序列解包

- 调用函数时，可以通过在实参序列前加一个星号将其中的元素解包为普通位置参数。

```
>>> def demo(a, b, c):
    print(a+b+c)


>>> seq = [1, 2, 3]
>>> demo(*seq)
6
>>> tup = (1, 2, 3)
>>> demo(*tup)
6
```

```
>>> dic = {1:'a', 2:'b', 3:'c'}
>>> demo(*dic)
6
>>> Set = {1, 2, 3}
>>> demo(*Set)
6
>>> demo(*dic.values())
abc
```

# 5.2.5 实参序列解包

- 如果实参是字典，可以在前面加两个星号进行解包为关键参数。

```
>>> def demo(a, b, c):
    print(a+b+c)

>>> dic = {'a':1, 'b':2, 'c':3}
>>> demo(**dic)
6
>>> demo(a=1, b=2, c=3)          # 与上一行代码等价
6
```

# 5.2.5 实参序列解包

- 注意：对实参序列使用一个星号*进行解包后其中的元素作为普通位置参数对待，会在关键参数和使用两个星号**进行解包的参数之前进行处理。

```
>>> def demo(a, b, c):                    # 定义函数
    print(a, b, c)

>>> demo(*(1, 2, 3))                      # 调用函数，序列解包
1 2 3
>>> demo(1, *(2, 3))                      # 位置参数和序列解包同时使用
1 2 3
>>> demo(1, *(2,), 3)
1 2 3
```

# 5.2.5 实参序列解包

```
>>> demo(a=1, *(2, 3))          # 序列解包相当于位置参数，优先处理
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    demo(a=1, *(2, 3))
TypeError: demo() got multiple values for argument 'a'
>>> demo(b=1, *(2, 3))
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    demo(b=1, *(2, 3))
TypeError: demo() got multiple values for argument 'b'
>>> demo(*(3,), **{'a':1, 'b':2})
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    demo(*(3,), **{'a':1, 'b':2})
TypeError: demo() got multiple values for argument 'a'
```

# 5.2.5 实参序列解包

```
>>> demo(**{'a':1, 'b':2}, *(3,)) # 序列解包不能在关键参数解包之后
SyntaxError: iterable argument unpacking follows keyword argument
unpacking
>>> demo(c=1, *(2, 3))
2 3 1
>>> demo(*(3,), **{'c':1, 'b':2})
3 2 1
```

# 5.3 变量作用域

- 变量起作用的代码范围称为变量的作用域，不同作用域内变量名可以相同，互不影响。
- 在函数内部定义的普通变量只在函数内部起作用，称为局部变量。当函数执行结束后，局部变量自动删除，不再可以使用。
- 局部变量的引用比全局变量速度略快，应优先考虑使用。

# 5.3 变量作用域

- 全局变量可以通过关键字global来定义。这分为两种情况：

✓ 一个变量已在函数外定义，如果在函数内需要修改这个变量的值，可以在函数内使用global声明为要使用的全局变量。

✓ 如果一个变量在函数外没有定义，在函数内部也可以直接将一个变量定义为全局变量，该函数执行后，将增加一个新的全局变量。不建议这种用法。

# 5.3 变量作用域

- 也可以这么理解：

✓ 在函数内只引用某个变量的值而没有为其赋新值，如果这样的操作可以执行，那么该变量为（隐式的）全局变量；

✓ 如果在函数内任意位置有为变量赋值的操作，该变量即被认为是（隐式的）局部变量，除非在函数内显式地用关键字global进行声明。

# 5.3 变量作用域

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

>>> x = 5
>>> demo()
3  4
>>> x
3
>>> y
NameError: name 'y' is not defined
```

# 5.3 变量作用域

```
>>> del x
>>> x
NameError: name 'x' is not defined
>>> demo()
3  4
>>> x
3
>>> y
NameError: name 'y' is not defined
```

# 5.3 变量作用域

- 注意：在某个作用域内任意位置只要有为变量赋值的操作，该变量在这个作用域内就是局部变量，除非使用global进行了声明。

```
>>> x = 10
>>> def f():
    print(x)
    x = x + 1
    print(x)

>>> f()
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    f()
  File "<pyshell#9>", line 2, in f
    print(x)
UnboundLocalError: local variable 'x' referenced before assignment
```

# 5.3 变量作用域

- 如果局部变量与全局变量具有相同的名字，那么该局部变量会在自己的作用域内隐藏同名的全局变量。

```
>>> def demo():
    x = 3              # 创建了局部变量，并自动隐藏了同名的全局变量
>>> x = 5
>>> x
5
>>> demo()
>>> x                  # 函数执行不影响外面全局变量的值
5
```

# 5.4 lambda表达式

- lambda表达式可以用来声明匿名函数，也就是没有函数名字的临时使用的小函数，尤其适合需要一个函数作为另一个函数参数的场合。也可以定义具名函数。
- lambda表达式只可以包含一个表达式，该表达式的计算结果可以看作是函数的返回值，不允许包含复合语句，但在表达式中可以调用其他函数。

# 5.4 lambda表达式

```
>>> f = lambda x, y, z: x+y+z          # 可以给lambda表达式起名字
>>> f(1,2,3)                           # 像函数一样调用
6
>>> g = lambda x, y=2, z=3: x+y+z      # 参数默认值
>>> g(1)
6
>>> g(2, z=4, y=5)                     # 关键参数
11
```

# 5.4 lambda表达式

```
>>> L = [lambda x: x**2, lambda x: x**3, lambda x: x**4]
>>> print(L[0](2),L[1](2),L[2](2))              # 通过下标访问lambda表达式
4 8 16
>>> D = {'f1': lambda:2+3, 'f2': lambda:2*3, 'f3': lambda:2**3}
>>> print(D['f1'](), D['f2'](), D['f3']())      # 通过“键”访问lambda表达式
5 6 8
>>> L = [1, 2, 3, 4, 5]
>>> print(list(map(lambda x: x+10, L)))         # 模拟向量运算
[11, 12, 13, 14, 15]
```

# 5.4 lambda表达式

```
>>> data = list(range(20))                          # 创建列表
>>> data
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> import random
>>> random.shuffle(data)                            # 随机打乱顺序
>>> data
[4, 3, 11, 13, 12, 15, 9, 2, 10, 6, 19, 18, 14, 8, 0, 7, 5, 17, 1, 16]
>>> data.sort(key=lambda x: len(str(x)))            # 按转换成字符串以后的长度排序
>>> data
[4, 3, 9, 2, 6, 8, 0, 7, 5, 1, 11, 13, 12, 15, 10, 19, 18, 14, 17, 16]
>>> data.sort(key=lambda x: len(str(x)), reverse=True)
                                                    # 降序排序
>>> data
[11, 13, 12, 15, 10, 19, 18, 14, 17, 16, 4, 3, 9, 2, 6, 8, 0, 7, 5, 1]
```

# 5.4 lambda表达式

- 在循环结构中定义的lambda表达式或函数在循环结构之外使用时，要注意变量作用域带来的问题。

- 有问题的代码：

```
>>> r = []
>>> for x in range(10):
    r.append(lambda: x**2)

>>> r[1]()
81
>>> r[5]()
81
>>> x = 3
>>> r[5]()
9
```

- 改进代码：

```
>>> r = []
>>> for x in range(10):
    r.append(lambda n=x: n**2)

>>> r[1]()
1
>>> r[5]()
25
>>> x = 3
>>> r[5]()
25
```

# 5.5 生成器函数设计与使用

- 包含yield语句的函数称作生成器函数，调用生成器函数返回生成器对象（属于迭代器对象），而不是具体的值。
- yield语句与return语句的作用相似，都是用来从函数中返回值。return语句一旦执行会立刻结束函数的运行，但每次执行到yield语句生成并返回一个值之后会暂停或挂起后面代码的执行，下次通过生成器对象的__next__()方法、内置函数next()、for循环遍历生成器对象元素或其他方式显式“索要”数据时恢复执行。
- 生成器具有惰性求值的特点，占用内存空间小，适合大数据处理。

# 5.5 生成器函数设计与使用

```
>>> def f():
    a, b = 1, 1                 # 序列解包，同时为多个变量赋值
    while True:
        yield a                 # 暂停执行，需要时再产生一个新元素
        a, b = b, a+b           # 序列解包，继续生成新元素
>>> a = f()                     # 创建生成器对象
>>> for i in range(10):         # 斐波那契数列中前10个元素
    print(a.__next__(), end=' ')

1 1 2 3 5 8 13 21 34 55
```

# 5.5 生成器函数设计与使用

```
>>> for i in f():                    # 斐波那契数列中第一个大于100的元素
    if i > 100:
        print(i, end=' ')
        break

144
>>> a = f()                          # 创建生成器对象
>>> next(a)                          # 使用内置函数next()获取生成器对象中的元素
1
>>> next(a)                          # 每次索取新元素时，由yield语句生成
1
>>> a.__next__()                     # 也可以调用生成器对象的__next__()方法
2
>>> a.__next__()
3
```

# 5.5 生成器函数设计与使用

```
>>> def f():
    yield from 'abcdefg'          # 使用yield表达式创建生成器

>>> x = f()
>>> next(x)
'a'
>>> next(x)
'b'
>>> for item in x:               # 输出x中的剩余元素
    print(item, end=' ')

c d e f g
```

# 5.5 生成器函数设计与使用

```
>>> def gen():
    yield 1
    yield 2
    yield 3

>>> x, y, z = gen()            # 生成器对象支持序列解包
```

# 5.6 精彩案例赏析

- **例5-1** 编写函数，接收任意多个实数，返回一个元组，其中第一个元素为所有参数的平均值，其他元素为所有参数中大于平均值的实数。code\例5-1.py

```
def demo(*para):
    avg = sum(para) / len(para)              # 平均值
    g = [i for i in para if i>avg]           # 列表推导式
    return (avg,) + tuple(g)

print(demo(1, 2, 3, 4, 5))                   # 输出：(3.0, 4, 5)
```