

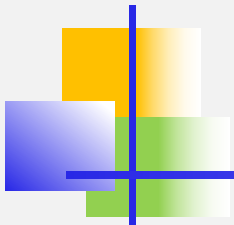


桂林理工大学
GUILIN UNIVERSITY OF TECHNOLOGY

第五章

函数

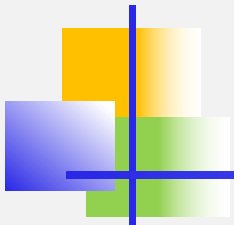




第五章 函数

- 将可能需要反复执行的代码封装为函数，并在需要该功能的地方进行调用，不仅可以实现**代码复用**，更重要的是可以**保证代码的一致性**，只需要修改该函数代码则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**。





第五章 函数

- 5.1 函数定义与调用基本语法
- 5.2 函数参数
- 5.3 变量作用域
- 5.4 lambda表达式
- 5.5 生成器函数设计要点
- 5.6 精彩案例赏析



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

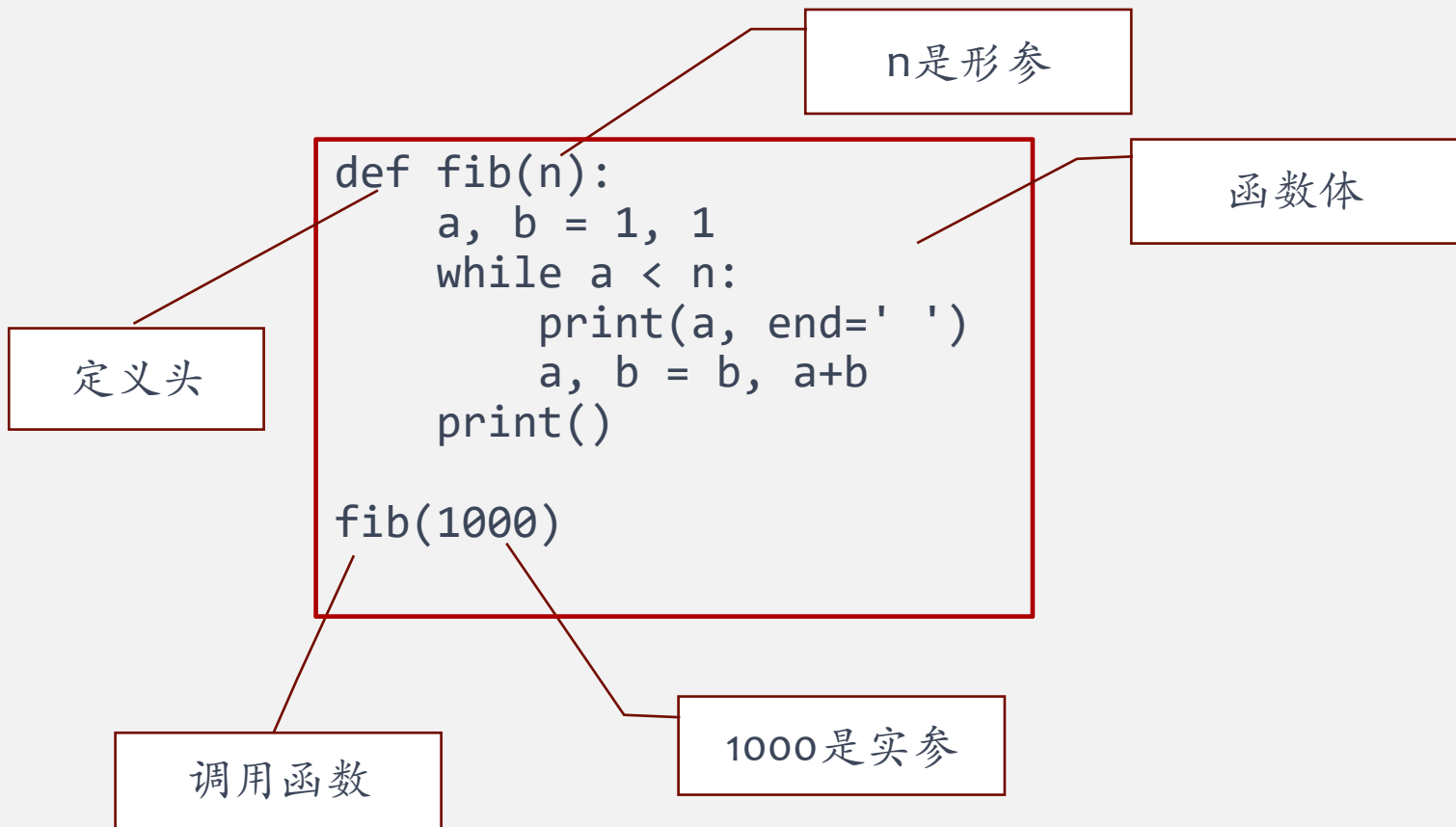
- ✓ 函数形参不需要声明类型，也不需要指定函数返回值类型
- ✓ 即使该函数不需要接收任何参数，也必须保留一对空的圆括号
- ✓ 括号后面的冒号必不可少
- ✓ 函数体相对于def关键字必须保持一定的空格缩进
- ✓ Python允许嵌套定义函数

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

- 问题解决：编写生成斐波那契数列的函数并调用。



0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368 ...

如果设 a_n 为该数列的第 n 项 ($n \in \mathbb{N}^*$)，那么这句话可以写成如下形式： $a_n = a_{n-1} + a_{n-2}$

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

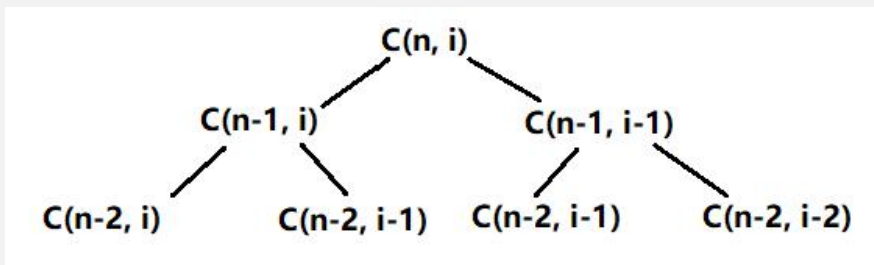
Python允许函数的嵌套定义，在函数内部可以再定义另外一个函数。

```
>>> 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

- **问题解决：**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。

```
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)
```



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。

```
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)
```



5.1.2 函数嵌套定义、可调用对象与修饰器

(2) 可调用对象

函数属于Python可调用对象之一，由于构造方法的存在，类也是可调用的。像`list()`、`tuple()`、`dict()`、`set()`这样的工厂函数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

- 下面的代码使用函数的嵌套定义实现了可调用对象的定义：

```
def linear(a, b):  
    def result(x):  
        return a * x + b  
    return result
```

#在Python中，函数是可以嵌套定义的
#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

- 下面的代码演示了可调用对象类的定义：

```
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定义一个可调用对象：

```
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改造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、属性等也都是通过修饰器实现的。

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):
        print('Before function called.')
        return func(*args, **kwargs)
    return wrapper
```

```
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():
```

```
    print(3)
```

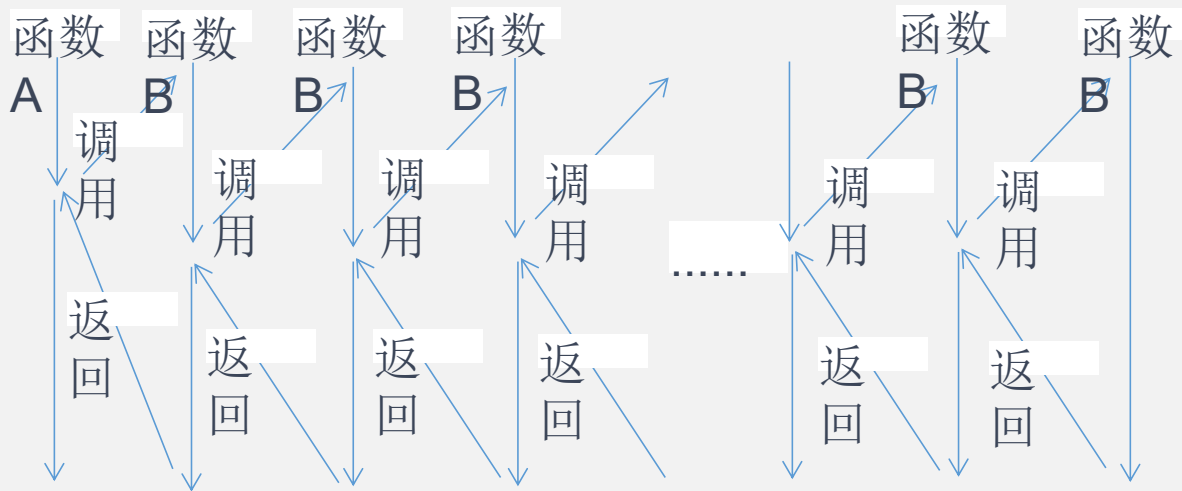
```
#调用被修饰的函数
```

```
test()
```

```
#同时使用两个修饰器改造函数
```


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

- **问题解决：**使用递归法对整数进行因数分解。

```
from random import randint
```

```
def factors(num, fac=[]):  
    #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:  
            fac.append(i)  
            #对商继续分解，重复这个过程  
            factors(num//i, fac)  
            #注意，这个break非常重要  
            break  
    else:  
        #不可分解了，自身也是个因数  
        fac.append(num)
```



5.1.3 函数递归调用

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



5.2函数参数

- 函数定义时圆括弧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弧必须要有，表示这是一个函数并且不接收参数。
- 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

- 对于绝大多数情况下，在函数内部直接修改形参的值不会影响实参，而是**创建一个新变量**。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

- 在有些情况下，可以通过特殊的方式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序列，并且在函数内部使用下标或可变序列自身的方法增加、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
{'age': 37, 'name': 'Dong', 'sex': 'Male'}
>>> modify(a)
>>> a
{'age': 38, 'name': 'Dong', '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

```
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

- 在调用带有默认值参数的函数时，可以不用为设置了默认值的形参进行传值，此时函数将会直接使用函数定义时设置的默认值，当然也可以通过显式赋值来替换其默认值。

在调用函数时是否为默认值参数传递实参是可选的。

- 需要注意的是，在定义带有默认值参数的函数时，任何一个默认值参数右边都**不能**再出现没有默认值的普通位置参数，否则会提示语法错误。



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
```



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

- 多次调用函数并且不为默认值参数传递值时，默认值参数只在定义时进行一次解释和初始化，对于列表、字典这样可变类型的默认值参数，这一点可能会导致很严重的逻辑错误。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'))  
['a', 'b']
```

#注意这里的输出结果



5.2.2 默认值参数

- 一般来说，要避免使用列表、字典、集合或其他可变序列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

- 函数的默认值参数是在函数定义时确定值的，所以只会被初始化一次。

```
>>> i = 3
>>> def f(n=i):
    print(n)
>>> f()
```

#参数n的值仅取决于i的当前值

```
3
>>> i = 5
>>> f()
```

#函数定义后修改i的值不影响参数n的默认值

```
3
>>> i = 7
>>> f()
```

```
3
>>> def f(n=i):
    print(n)
>>> f()
```

#重新定义函数

```
7
```

5.2.3 关键参数

- 关键参数主要指调用函数时的参数传递方式，与函数定义无关。通过关键参数可以按参数名字传递值，明确指定哪个值传递给哪个参数，**实参顺序可以和形参顺序不一致**，但不影响参数值的传递结果，避免了用户需要牢记参数位置和顺序的麻烦，使得函数的调用和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：在参数名前加1个*或2个**
 - *parameter用来接受多个位置参数并将其放在一个元组中
 - **parameter接受多个关键参数并存放到字典中



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)  
( 'y', 2)  
( 'x', 1)  
( '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，但是不建议这样做。

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

5.2.5 传递参数时的序列解包

- 传递参数时，可以通过在实参序列前加一个星号将其解包，然后传递给多个单变量形参。

```
>>> 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 传递参数时的序列解包

- 如果函数实参是字典，可以在前面加两个星号进行解包，等价于关键参数。

```
>>> 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)
```

6

```
>>> demo(*dic.values())
```

6

5.2.5 传递参数时的序列解包

- **注意：**调用函数时对实参序列使用一个星号*进行解包后的实参将会被当做普通位置参数对待，并且会在关键参数和使用两个星号**进行序列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，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 传递参数时的序列解包

```
>>> demo(a=1, *(2, 3))  
先处理
```

#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(c=1, *(2, 3))
```

```
2 3 1
```

5.2.5 传递参数时的序列解包

```
>>> 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(*(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'
```

```
>>> 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，如果在函数内需要为这个变量赋值，并将这个赋值结果反映到函数外，可以在函数内使用global将其声明为全局变量。
 - ✓ 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。

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新值的操作，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 = 3
```

```
>>> def f():
```

```
    print(x)
```

```
    x = 5
```

```
    print(x)
```

```
>>> f()
```

#本意是先输出全局变量x的值，但是不允许这样做

#有赋值操作，因此在整个作用域内x都是局部变量

```
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))
4 8 16
>>> D = {'f1':(lambda:2+3),
          'f2':(lambda:2*3),
          'f3':(lambda:2**3)}
>>> print(D['f1'](), D['f2'](), D['f3']())
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]
>>> L
[1, 2, 3, 4, 5]
```



5.4 lambda表达式

```
>>> def demo(n):  
    return n*n
```

```
>>> demo(5)
```

```
25
```

```
>>> a_list = [1,2,3,4,5]
```

```
>>> list(map(lambda x: demo(x), a_list)) #在lambda表达式中调用  
    函数
```

```
[1, 4, 9, 16, 25]
```



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)             #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: x)       #和不指定规则效果一样
>>> data
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

5.4 lambda表达式

```
>>> data.sort(key=lambda x: len(str(x)))      #按转换成字符串以后的长度排
序
>>> data
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> data.sort(key=lambda x: len(str(x)), reverse=True)
                                     #降序排序
>>> data
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

5.4 lambda表达式

```
>>> import random
>>> x = [[random.randint(1,10) for j in range(5)] for i in range(5)]
#使用列表推导式创建列表
#包含5个子列表的列表
#每个子列表中包含5个1到10之间的随机数

>>> for item in x:
    print(item)
```

```
[5, 6, 8, 7, 4]
[1, 5, 3, 9, 4]
[9, 6, 10, 7, 6]
[8, 2, 7, 1, 6]
[1, 7, 5, 3, 5]
```



5.4 lambda表达式

```
>>> y = sorted(x, key=lambda item: (item[1], item[4]))
```

#按子列表中第2个元素升序、第5个元素升序排序

```
>>> for item in y:  
    print(item)
```

```
[8, 2, 7, 1, 6]
```

```
[1, 5, 3, 9, 4]
```

```
[5, 6, 8, 7, 4]
```

```
[9, 6, 10, 7, 6]
```

```
[1, 7, 5, 3, 5]
```



5.5 生成器函数设计要点

- 包含yield语句的函数可以用来创建生成器对象，这样的函数也称生成器函数。
- yield语句与return语句的作用相似，都是用来从函数中返回值。与return语句不同的是，return语句一旦执行会立刻结束函数的运行，而每次执行到yield语句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，适合大数据处理。

5.5 生成器函数设计要点

```
>>> def f():  
    a, b = 1, 1                #序列解包，同时为多个元素赋值  
    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 生成器函数设计要点

```
>>> 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__()      #也可以调用生成器对象的__next__()方法
```

3

5.5 生成器函数设计要点

```
>>> 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:  
    print(item, end=' ')
```

#输出x中的剩余元素

```
c d e f g
```



5.5 生成器函数设计要点

```
>>> def gen():
```

```
    yield 1
```

```
    yield 2
```

```
    yield 3
```

```
>>> x, y, z = gen()
```

#生成器对象支持序列解包



5.5 生成器函数设计要点

问题解决：使用生成器模拟了标准库itertools中的count()函数。

```
>>> def count(start, step):  
    num = start  
    while True:                                #无穷循环  
        yield num                               #返回一个数，暂停执行，等待下一  
        num += step                             次索要数据  
  
>>> x = count(3, 5)  
>>> for i in range(10):  
    print(next(x), end=' ')  
3 8 13 18 23 28 33 38 43 48  
>>> for i in range(10):  
    print(next(x), end=' ')  
53 58 63 68 73 78 83 88 93 98
```



5.6 案例赏析

- **示例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。



5.6 案例赏析

- **示例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。

```
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)
```



5.6 精彩案例赏析

- **示例5-2** 编写函数，接收字符串参数，返回一个元组，其中第一个元素为大写字母个数，第二个元素为小写字母个数。

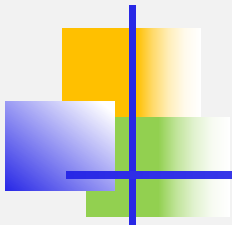
`islower()` 表示小写，`isupper()`表示大写



5.6 精彩案例赏析

- **示例5-2** 编写函数，接收字符串参数，返回一个元组，其中第一个元素为大写字母个数，第二个元素为小写字母个数。

```
def demo(s):  
    result = [0, 0]  
    for ch in s:  
        if ch.islower():  
            result[1] += 1  
        elif ch.isupper():  
            result[0] += 1  
    return tuple(result)
```



5.6 案例赏析

- **示例5-3** 编写函数，接收包含 n 个整数的列表`lst`和一个整数 k ($0 \leq k < n$) 作为参数，返回新列表。处理规则为：将列表`lst`中下标 k 之前的元素逆序，下标 k 之后的元素逆序，然后将整个列表`lst`中的所有元素逆序。



5.6 案例赏析

- **示例5-3** 编写函数，接收包含 n 个整数的列表`lst`和一个整数 k ($0 \leq k < n$) 作为参数，返回新列表。处理规则为：将列表`lst`中下标 k 之前的元素逆序，下标 k 之后的元素逆序，然后将整个列表`lst`中的所有元素逆序。

```
def demo(lst, k):  
    x = lst[k-1::-1]  
    y = lst[:k-1:-1]  
    return list(reversed(x+y))
```



5.6 精彩案例赏析

- 本例描述的实际上是将列表循环左移 k 位的算法实现，下面的代码使用了更加直接的方法，但对于长列表来说效率远不如上面的代码高，因为`pop(0)`操作在列表首部删除元素，这会引起大量元素的前移。



5.6 精彩案例赏析

- 本例描述的实际上是将列表循环左移 k 位的算法实现，下面的代码使用了更加直接的方法，但对于长列表来说效率远不如上面的代码高，因为`pop(0)`操作在列表首部删除元素，这会引起大量元素的前移。

```
def demo(lst, k):  
    temp = lst[:]  
    for i in range(k):  
        temp.append(temp.pop(0))  
    return temp
```



5.6 精彩案例赏析

- 搞清楚问题本质以后，对于本例中描述的问题，使用切片可以直接实现，可以达到最快的速度。

```
def demo(lst, k):  
    return lst[k:] + lst[:k]
```



5.6 精彩案例赏析

- **示例5-4** 编写函数，接收一个整数t为参数，打印杨辉三角前t行。

第0行		1	第1斜列										
第1行		1	1	第2斜列									
第2行		1	2	1	第3斜列								
第3行		1	3	3	1	第4斜列							
第4行		1	4	6	4	1	第5斜列						
第5行		1	5	10	10	5	1	第6斜列					
第6行		1	6	15	20	15	6	1	第7斜列				
第7行		1	7	21	35	35	21	7	1	第8斜列			
第8行		1	8	26	56	70	56	28	8	1	第9斜列		
第9行		1	9	36	84	126	126	84	36	9	1	第10斜列	
第10行	1	10	45	120	210	252	210	120	45	10	1	第11斜列	
第11行	1	11	55	165	330	462	462	330	165	55	11	1	...	第12斜列

11阶杨辉三角



5.6 精彩案例赏析

- **示例5-4** 编写函数，接收一个整数 t 为参数，打印杨辉三角前 t 行。

```
def yanghui(t):  
    print([1])  
    line = [1, 1]  
    print(line)  
    for i in range(2, t):  
        r = []  
        for j in range(0, len(line)-1):  
            r.append(line[j]+line[j+1])  
        line = [1]+r+[1]  
        print(line)
```



5.6 精彩案例赏析

- 示例5-5 编写函数，使用collections标准库的defaultdict实现上例的功能。

```
from collections import defaultdict
```

```
def yanghui(n):  
    #所有元素默认值0  
    triangle = defaultdict(int)  
    for row in range(n):  
        #每行第一个元素为1  
        triangle[row,0] = 1  
        print(triangle[row,0], end='\t')  
        #生成该行后续元素  
        for col in range(1, row+1):  
            #如果指定位置的元素不存在，默认为0  
            triangle[row,col] = triangle[row-1,col-1] + triangle[row-1,col]  
            print(triangle[row,col], end='\t')  
        print()
```

```
yanghui(14)
```



5.6 精彩案例赏析

- **示例5-6** 编写函数，接收一个正偶数为参数，输出两个素数，并且这两个素数之和等于原来的正偶数。如果存在多组符合条件的素数，则全部输出。



5.6 精彩案例赏析

- **示例5-6** 编写函数，接收一个正偶数为参数，输出两个素数，并且这两个素数之和等于原来的正偶数。如果存在多组符合条件的素数，则全部输出。

```
def demo(n):  
    def IsPrime(p):  
        if p == 2:  
            return True  
        if p%2 == 0:  
            return False  
        for i in range(3, int(p**0.5)+1, 2):  
            if p%i==0:  
                return False  
        return True  
  
    if isinstance(n, int) and n>0 and n%2==0:  
        for i in range(2, n//2+1):  
            if IsPrime(i) and IsPrime(n-i):  
                print(i, '+', n-i, '=', n)
```



5.6 精彩案例赏析

- Python标准库fractions中提供了gcd()函数用来计算最大公约数，在Python 3.5和更新版本中，标准库math也提供了计算最大公约数的函数gcd()。





5.6 精彩案例赏析

- Python标准库fractions中提供了gcd()函数用来计算最大公约数，在Python 3.5和更新版本中，标准库math也提供了计算最大公约数的函数gcd()。利用gcd()函数，上面的代码也可以写作：

```
def demo(m, n):  
    import math  
  
    r = math.gcd(m,n)  
  
    return (r, (m*n)//r)
```



5.6 精彩案例赏析

- **示例5-8** 编写函数，接收一个所有元素值都不相等的整数列表 x 和一个整数 n ，要求将值为 n 的元素作为支点，将列表中所有值小于 n 的元素全部放到 n 的前面，所有值大于 n 的元素放到 n 的后面。

```
def demo(x, n):  
    t1 = [i for i in x if i<n]  
    t2 = [i for i in x if i>n]  
    return t1 + [n] + t2
```



5.6 精彩案例赏析

- **示例5-9** 编写函数，计算字符串匹配的准确率。

以打字练习程序为例，假设`origin`为原始内容，`userInput`为用户输入的内容，下面的代码用来测试用户输入的准确率。

```
def Rate(origin, userInput):  
    if not (isinstance(origin, str) and isinstance(userInput, str)):  
        print('The two parameters must be strings.')        return  
    right = sum((1 for o, u in zip(origin, userInput) if o==u))  
    return round(right/len(origin), 2)
```





5.6 精彩案例赏析

- **示例5-10** 编写函数，使用非递归方法对整数进行因数分解。



5.6 精彩案例赏析

```
from random import randint
from math import sqrt

def factoring(n):
    '''对大数进行因数分解'''
    if not isinstance(n, int):
        print('You must give me an integer')
        return
    #开始分解, 把所有因数都添加到result列表中
    result = []
    for p in primes:
        while n!=1:
            if n%p == 0:
                n = n/p
                result.append(p)
            else:
                break
        else:
            result = '*'.join(map(str, result))
            return result
    #考虑参数本身就是素数的情况
    if not result:
        return n
```

5.6 精彩案例赏析

```
testData = [randint(10, 100000) for i in range(50)]  
#随机数中的最大数  
maxData = max(testData)  
#小于maxData的所有素数  
primes = [ p for p in range(2, maxData) if 0 not in  
           [p%d for d in range(2, int(sqrt(p))+1)]]  
  
for data in testData:  
    r = factoring(data)  
    print(data, '=', r)  
    #测试分解结果是否正确  
    print(data==eval(r))
```



5.6 精彩案例赏析

- **示例5-11** 编写函数模拟猜数游戏。系统随机产生一个数，玩家最多可以猜5次，系统会根据玩家的猜测进行提示，玩家则可以根据系统的提示对下一次的猜测进行适当调整。



5.6 精彩案例赏析

```
from random import randint

def guess(maxValue=100, maxTimes=5):
    #随机生成一个整数
    value = randint(1,maxValue)
    for i in range(maxTimes):
        prompt = 'Start to GUESS:' if i==0 else 'Guess again:'
        #使用异常处理结构，防止输入不是数字的情况
        try:
            x = int(input(prompt))
        except:
            print('Must input an integer between 1 and ', maxValue)
        else:
            #猜对了
            if x == value:
                print('Congratulations!')
                break
            elif x > value:
                print('Too big')
            else:
                print('Too little')
    else:
        #次数用完还没猜对，游戏结束，提示正确答案
        print('Game over. FAIL.')
        print('The value is ', value)
```

5.6 精彩案例赏析

- **示例5-12** 编写函数，计算形式如 $a+aa+aaa+aaaa+\dots+aaa\dots aaa$ 的表达式的值，其中 a 为小于10的自然数。

```
def demo(v, n):  
    assert type(n)==int and 0<v<10, 'v must be integer between 1 and 9'  
    result, t = 0, 0  
    for i in range(n):  
        t = t*10 + v  
        result += t  
    return result  
  
print(demo(3, 4))
```



5.6 精彩案例赏析

- **示例5-13** 编写函数模拟报数游戏。有 n 个人围成一圈，顺序编号，从第一个人开始从1到 k （假设 $k=3$ ）报数，报到 k 的人退出圈子，然后圈子缩小，从下一个人继续游戏，问最后留下的是原来的第几号。



5.6 精彩案例赏析

```
from itertools import cycle

def demo(lst, k):
    #切片，以免影响原来的数据
    t_lst = lst[:]
    #游戏一直进行到只剩下最后一个人
    while len(t_lst)>1:
        #创建cycle对象
        c = cycle(t_lst)
        #从1到k报数
        for i in range(k):
            t = next(c)
            #一个人出局，圈子缩小
            index = t_lst.index(t)
            t_lst = t_lst[index+1:] + t_lst[:index]
    #游戏结束
    return t_lst[0]

lst = list(range(1,11))
print(demo(lst, 3))
```

5.6 精彩案例赏析

- **示例5-14** 汉诺塔问题基于递归算法的实现。
- 据说古代有一个梵塔，塔内有三个底座A、B、C，A座上有64个盘子，盘子大小不等，大的在下，小的在上。有一个和尚想把这64个盘子从A座移到C座，但每次只能允许移动一个盘子，在移动盘子的过程中可以利用B座，但任何时刻3个座上的盘子都必须始终保持大盘在下、小盘在上的顺序。如果只有一个盘子，则不需要利用B座，直接将盘子从A移动到C即可。和尚想知道这项任务的详细移动步骤和顺序。这实际上是一个非常巨大的工程，是一个不可能完成的任务。根据数学知识我们可以知道，移动n个盘子需要 $2^n - 1$ 步，64个盘子需要18446744073709551615步。如果每步需要一秒钟的话，那么就需要584942417355.072年。

5.6 精彩案例赏析

```
def hanoi(num, src, dst, temp=None):
    #声明用来记录移动次数的变量为全局变量
    global times
    #确认参数类型和范围
    assert type(num) == int, 'num must be integer'
    assert num > 0, 'num must > 0'
    #只剩最后或只有一个盘子需要移动, 这也是函数递归调用的结束条件
    if num == 1:
        print('The {0} Times move:{1}==>{2}'.format(times, src, dst))
        times += 1
    else:
        #递归调用函数自身,
        #先把除最后一个盘子之外的所有盘子移动到临时柱子上
        hannuo(num-1, src, temp, dst)
        #把最后一个盘子直接移动到目标柱子上
        hannuo(1, src, dst)
        #把除最后一个盘子之外的其他盘子从临时柱子上移动到目标柱子上
        hannuo(num-1, temp, dst, src)
#用来记录移动次数的变量
times = 1
#A表示最初放置盘子的柱子, C是目标柱子, B是临时柱子
hanoi(3, 'A', 'C', 'B')
```

5.6 精彩案例赏析

- **示例5-15** 编写函数计算任意位数的黑洞数。黑洞数是指这样的整数：由这个数字每位上的数字组成的最大数减去每位数字组成的最小数仍然得到这个数自身。例如3位黑洞数是495，因为 $954-459=495$ ，4位数字是6174，因为 $7641-1467=6174$ 。

```
def main(n):  
    '''参数n表示数字的位数，例如n=3时返回495，n=4时返回6174'''  
    #待测试数范围的起点和结束值  
    start = 10**(n-1)  
    end = 10**n  
    #依次测试每个数  
    for i in range(start, end):  
        #由这几个数字组成的最大数和最小数  
        big = ''.join(sorted(str(i), reverse=True))  
        little = ''.join(reversed(big))  
        big, little = map(int, (big, little))  
        if big-little == i:  
            print(i)  
  
n = 4  
main(n)
```



5.6 精彩案例赏析

- **示例5-16** 24点游戏是指随机选取4张扑克牌（不包括大小王），然后通过四则运算来构造表达式，如果表达式的值恰好等于24就赢一次。下面的代码定义了一个函数用来测试随机给定的4个数是否符合24点游戏规则，如果符合就输出所有可能的表达式。





5.6 精彩案例赏析

```
from random import randint
```

```
from itertools import permutations
```

#4个数字和2个运算符可能组成的表达式形式

```
exps = ('((%s %s %s) %s %s) %s %s',
```

```
        '(%s %s %s) %s (%s %s %s)',
```

```
        '(%s %s (%s %s %s)) %s %s',
```

```
        '%s %s ((%s %s %s) %s %s)',
```

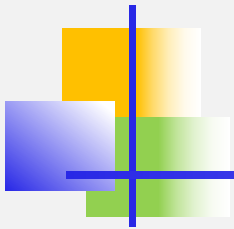
```
        '%s %s (%s %s (%s %s %s))')
```

```
ops = r'+-*/'
```



5.6 精彩案例赏析

```
def test24(v):  
    result = []  
    #Python允许函数的嵌套定义  
    #这个函数对字符串表达式求值并验证是否等于24  
    def check(exp):  
        try:  
            #有可能会出除0异常，所以放到异常处理结构中  
            return int(eval(exp)) == 24  
        except:  
            return False  
    #全排列，枚举4个数的所有可能顺序  
    for a in permutations(v):  
        #查找4个数的当前排列能实现24的表达式  
        t = [exp % (a[0], op1, a[1], op2, a[2], op3, a[3]) for op1 in  
ops for op2 in ops for op3 in ops for exp in exps if check(exp % (a[0],  
op1, a[1], op2, a[2], op3, a[3]))]  
        if t:  
            result.append(t)  
    return result
```



5.6 精彩案例赏析

```
for i in range(20):  
    print('='*20)  
    #生成随机数字进行测试  
    lst = [randint(1, 14) for j in range(4)]  
    r = test24(lst)  
    if r:  
        print(r)  
    else:  
        print('No answer for ', lst)
```



5.6 精彩案例赏析

- **示例5-17** 编写函数，查找序列元素的最大值和最小值。给定一个序列，返回一个元组，其中元组第一个元素为序列最大值，第二个元素为序列最小值。

```
def myMaxMin(iterable):  
    '''返回序列的最大值和最小值'''  
    tMax = tMin = iterable[0]  
    for item in iterable[1:]:  
        if item > tMax:  
            tMax = item  
        elif item < tMin:  
            tMin = item  
  
    return (tMax, tMin)
```

5.6 精彩案例赏析

- 示例5-18 编写函数，模拟内置函数all()、any()和zip()。

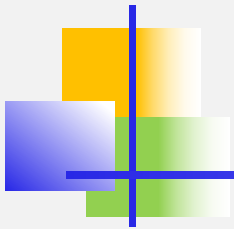
```
def myAll(iterable):  
    '''模拟内置函数all()'''  
    #只要有一个元素等价于False，返回False  
    for item in iterable:  
        if not item:  
            return False  
    #如果所有元素都等价于True，返回True  
    return True
```




5.6 精彩案例赏析

```
def myAny(iterable):  
    '''模拟内置函数any()'''  
    #只要有一个元素等价于True, 返回True  
    for item in iterable:  
        if item:  
            return True  
    #如果所有元素都等价于False, 返回False  
    return False
```





5.6 精彩案例赏析

```
def myZip(*iterables):  
    '''模拟内置函数zip()'''  
  
    #获取所有迭代对象的最小长度  
  
    min_length = min(map(len,iterables))  
  
    #依次返回所有迭代对象中对应位置上元素组成的元组  
  
    for i in range(min_length):  
        yield tuple((it[i] for it in iterables))
```



5.6 精彩案例赏析

- 示例5-19 编写函数，使用非递归算法实现冒泡排序算法。

```
from random import randint
```

```
def bubbleSort(lst, reverse=False):
    length = len(lst)
    for i in range(0, length):
        flag = False
        for j in range(0, length-i-1):
            #比较相邻两个元素大小，并根据需要进行交换，默认升序排序
            exp = 'lst[j] > lst[j+1]'
            #如果reverse=True则降序排序
            if reverse:
                exp = 'lst[j] < lst[j+1]'
            if eval(exp):
                lst[j], lst[j+1] = lst[j+1], lst[j]
                #flag=True表示本次扫描发生过元素交换
                flag = True
        #如果一次扫描结束后，没有发生过元素交换，说明已经按序排列
        if not flag:
            break
```

5.6 精彩案例赏析

● **示例5-20** 编写函数，使用递归算法实现冒泡排序算法。

```
from random import randint
```

```
def bubbleSort(lst, end=None, reverse=False):
```

```
    if end==None:
```

```
        length = len(lst)
```

```
    else:
```

```
        length = end
```

```
    if length<=1:
```

```
        return
```

```
    flag = False
```

否发生了元素的交换

```
    for j in range(length-1):
```

```
        exp = 'lst[j] > lst[j+1]'
```

需要进行交换，默认升序排序

```
        if reverse:
```

```
            exp = 'lst[j] < lst[j+1]'
```

```
        if eval(exp):
```

```
            lst[j], lst[j+1] = lst[j+1], lst[j]
```

```
            flag = True
```

```
    if flag==False:
```

已按序排列

```
        return
```

```
    else:
```

```
        bubbleSort(lst, length-1, reverse)#对剩余的元素进行排序
```

#flag用来标记本次扫描过程中是

#比较相邻两个元素大小，并根据

#如果reverse=True则降序排序

#如果没有发生元素交换，则表示

5.6 精彩案例赏析

- 示例5-21 编写函数，模拟选择法排序。

```
def selectSort(lst, reverse=False):  
    length = len(lst)  
    for i in range(0, length):  
        #假设剩余元素中第一个最小或最大  
        m = i  
        #扫描剩余元素  
        for j in range(i+1, length):  
            #如果有更小或更大的，就记录下它的位置  
            exp = 'lst[j] < lst[m]'  
            if reverse:  
                exp = 'lst[j] > lst[m]'  
            if eval(exp):  
                m = j  
        #如果发现更小或更大的，就交换值  
        if m != i:  
            lst[i], lst[m] = lst[m], lst[i]
```

5.6 精彩案例赏析

- 示例5-22 编写函数，模拟二分法查找。

```
def binarySearch(lst, value):  
    start = 0  
    end = len(lst)  
    while start < end:  
        #计算中间位置  
        middle = (start + end) // 2  
        #查找成功，返回元素对应的位置  
        if value == lst[middle]:  
            return middle  
        #在后面一半元素中继续查找  
        elif value > lst[middle]:  
            start = middle + 1  
        #在前面一半元素中继续查找  
        elif value < lst[middle]:  
            end = middle - 1  
    #查找不成功，返回False  
    return False
```

5.6 精彩案例赏析

- **示例5-23** 编写函数，模拟快速排序算法。

```
from random import randint
```

```
def quickSort(lst, reverse=False):  
    if len(lst) <= 1:  
        return lst  
    #默认使用最后一个元素作为枢点  
    pivot = lst.pop()  
    first, second = [], []  
    #默认使用升序排序  
    exp = 'x<=pivot'  
    #reverse=True表示降序排列  
    if reverse == True:  
        exp = 'x>=pivot'  
    for x in lst:  
        first.append(x) if eval(exp) else second.append(x)  
    #递归调用  
    return quickSort(first, reverse) + [pivot] +  
    quickSort(second, reverse)
```