





















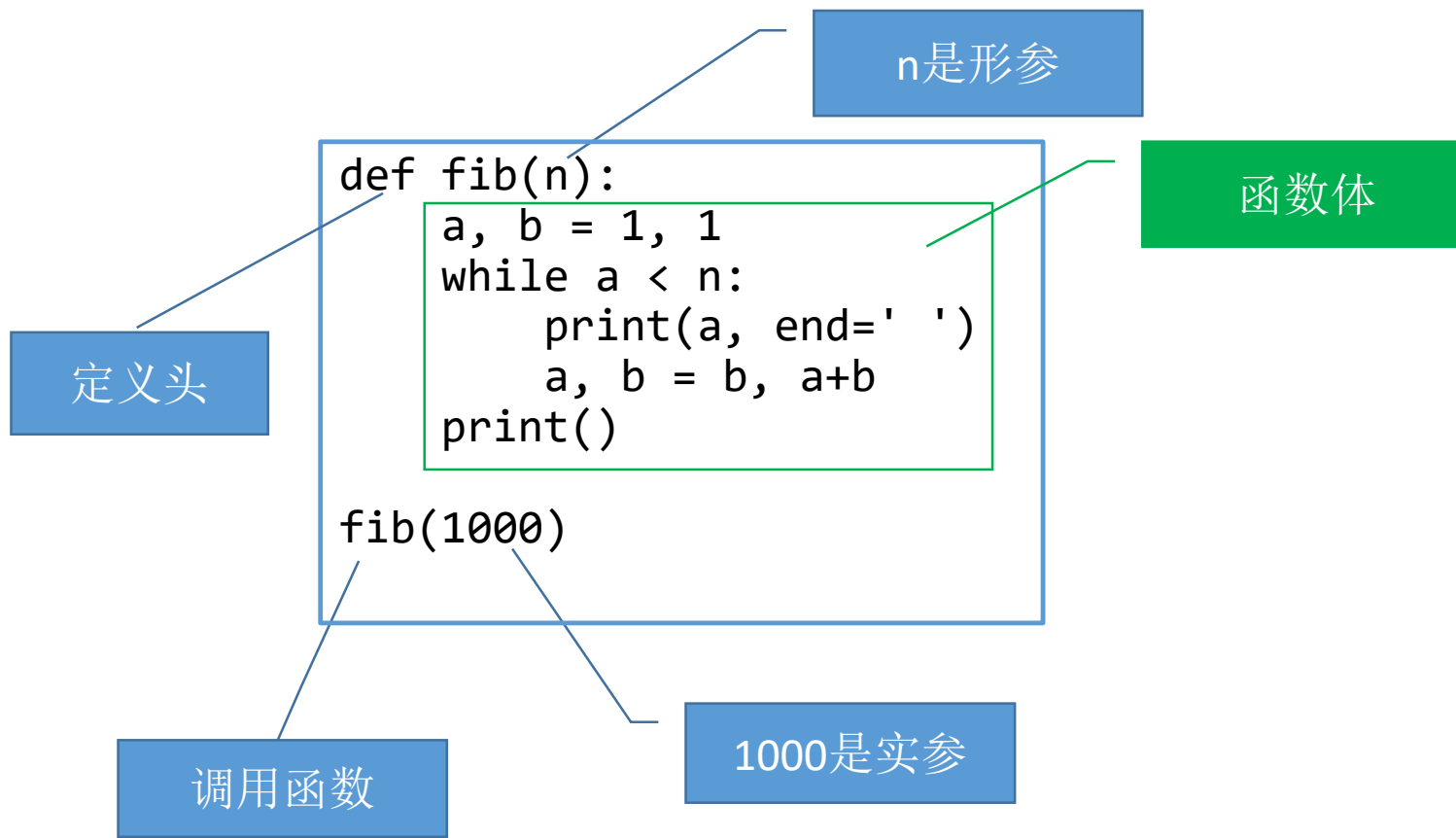
- **TIOBE排行榜**：根据互联网上有经验的程序员、课程和第三方厂商的数量统计出语言排名，反映编程语言的热门程度，并不能说明一门编程语言好不好，或者一门语言所编写的代码数量多少。
- 根据2022年10月榜单，Python（17.1%）、C（15.2%）、Java（12.8%）和C++（9.9%）持续保持Top 4语言，和其他语言的差距还在不断扩大。Top 4语言的整体份额超越了55%
- C、C++、Java的竞争对手：C#（4.4%）、VB（4.0%）、Rust（0.7%）
- Python的竞争对手：R（1.2%）、Ruby（0.9%）
- 相比通用型编程语言，剩下大多是特定型语言

	Oct 2022	Oct 2021	Change	Programming Language		Ratings	Change
1		1			Python	17.08%	+5.81%
2		2			C	15.21%	+4.05%
3		3			Java	12.84%	+2.38%
4		4			C++	9.92%	+2.42%
5		5			C#	4.42%	-0.84%
6		6			Visual Basic	3.95%	-1.29%
7		7			JavaScript	2.74%	+0.55%
8		10	▲		Assembly language	2.39%	+0.33%
9		9			PHP	2.04%	-0.06%
10		8	▼		SQL	1.78%	-0.39%
11		12	▲		Go	1.27%	-0.01%
12		14	▲		R	1.22%	+0.03%
13		29	▲▲		Objective-C	1.21%	+0.76%
14		13	▼		MATLAB	1.18%	-0.02%
15		17	▲		Swift	1.05%	-0.06%
16		16			Ruby	0.88%	-0.24%
17		11	▼▼		Classic Visual Basic	0.87%	-0.96%
18		20	▲		Delphi/Object Pascal	0.85%	-0.09%
19		18	▼		Fortran	0.79%	-0.29%
20		26	▲▲		Rust	0.70%	+0.17%

# 第5章 函数

## 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。



# 第5章 函数 $Y=F(x)$

- 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
  - **任务分解**，将复杂任务分解成多个相对独立的部分，对每个部分进行抽象化，得到针对某一问题通用的解决方案，然后用函数包裹这些代码。
  - **代码复用**，利用函数将代码打包后，可以在多处使用。在使用过程中不再关注针对具体问题的解决方案。一次编写后，可以在多处使用，减少了代码冗余，也使得代码维护更加容易。
  - **代码一致性**，只需要修改该函数代码则所有调用均受到影响。

# 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1, 参数2, ... 参数n]):  
    '''注解'''  
    函数体
```

## ❖ 注意事项

- ✓ 函数参数（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**

## 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。
- 与内置函数一样，定义并创建完函数后，可以通过函数名调用执行。

```
>>> 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()
```

注解

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

函数调用

# 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。
  - 有点类似**break**等，但它不仅是跳出循环，而是结束整段代码
  - 如果结束循环后就是到了结束整段代码的时候，可以用**return**替代**break**
- 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**。

```
while 条件表达式1:
    语句块1
    if 条件表达式2:
        语句X
        break
    if 条件表达式3:
        语句Y
        continue
    if 条件表达式4:
        return
    语句块2
    语句块3
```

Break: **跳出**while(结束循环), 执行语句块3，不执行语句块2

Continue: **跳回**while，**此轮**不执行语句块2-3

Return: **跳出**while(结束函数)，不执行语句块2-3



## 5.1.1 函数定义与调用基本语法

- 在很多情况下，函数需要将计算结果返回到调用处。在这类函数的函数体中，通常包含一条 `return` 语句：

```
def函数名([参数1, 参数2, ... 参数n]):  
    '''注解'''  
    函数体  
    return 数值
```

- `return` 语句不一定出现在函数体的最后，而是可以在任何位置。只要执行到 `return` 语句，函数就结束，将 `value` 返回到调用处。
- 在 `Python` 中，还允许在函数中返回多个值。只需将返回值以逗号隔开，放在 `return` 关键字后面即可

## 5.1.2 函数嵌套定义、可调用对象

### (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(str(item)+op+str(value))
        #eval(计算并返回字符串的值，比如eval("3+5"),返回8,P39
        #书上用的repr()将变量转化为字符串，类似str()函数，但差异在于是否保留引号
    return map(nested, iterable)     #使用在函数内部定义的函数
# map(func, iterables) 返回包含若干函数值的对象，函数func的参数分别来自于iterables指定的每个迭代对象，P34

print(list(myMap(range(5), '+', 5)))      #调用外部函数，不需要关心其内部实现
print(list(myMap(range(5), '-', 5)))
```

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

# 真相大白: `item1`实际是`iterable`列表中的元素，通过`map`一个个喂给`nested`

## 5.1.2 函数嵌套定义、可调用对象

### (2) 可调用对象

函数属于Python可调用对象之一，像list()、tuple()、dict()、set()

\* 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

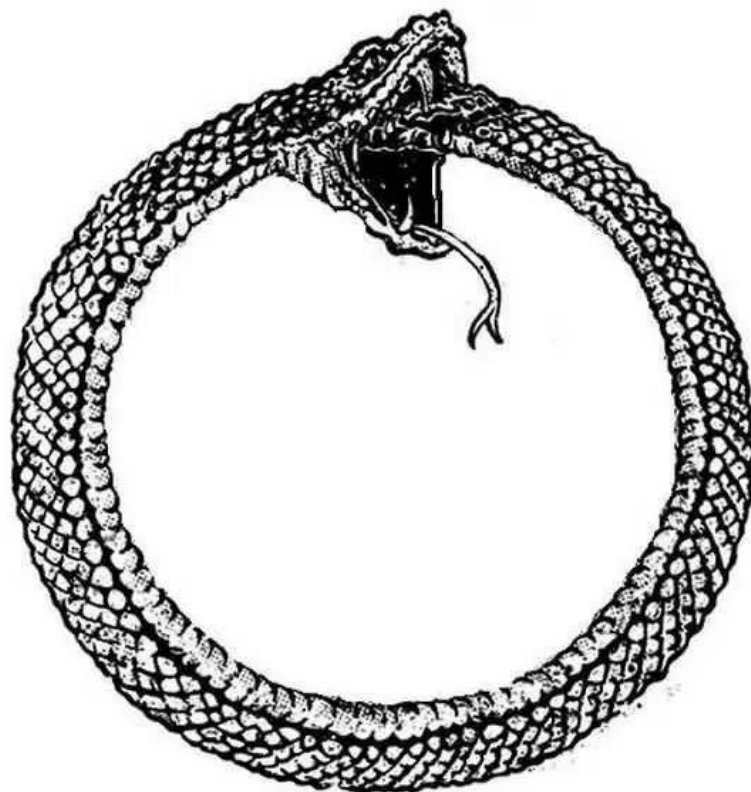
- 2. 可调用对象：通过函数的嵌套定义和逐步调用来实现：

```
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  
# x是linear定义的对象所需要的参数，第6章  
#返回可被调用的函数

```
taxes=linear(2,3)  
print(taxes(5))
```

函数可以调用自己么，就像贪吃蛇那样？

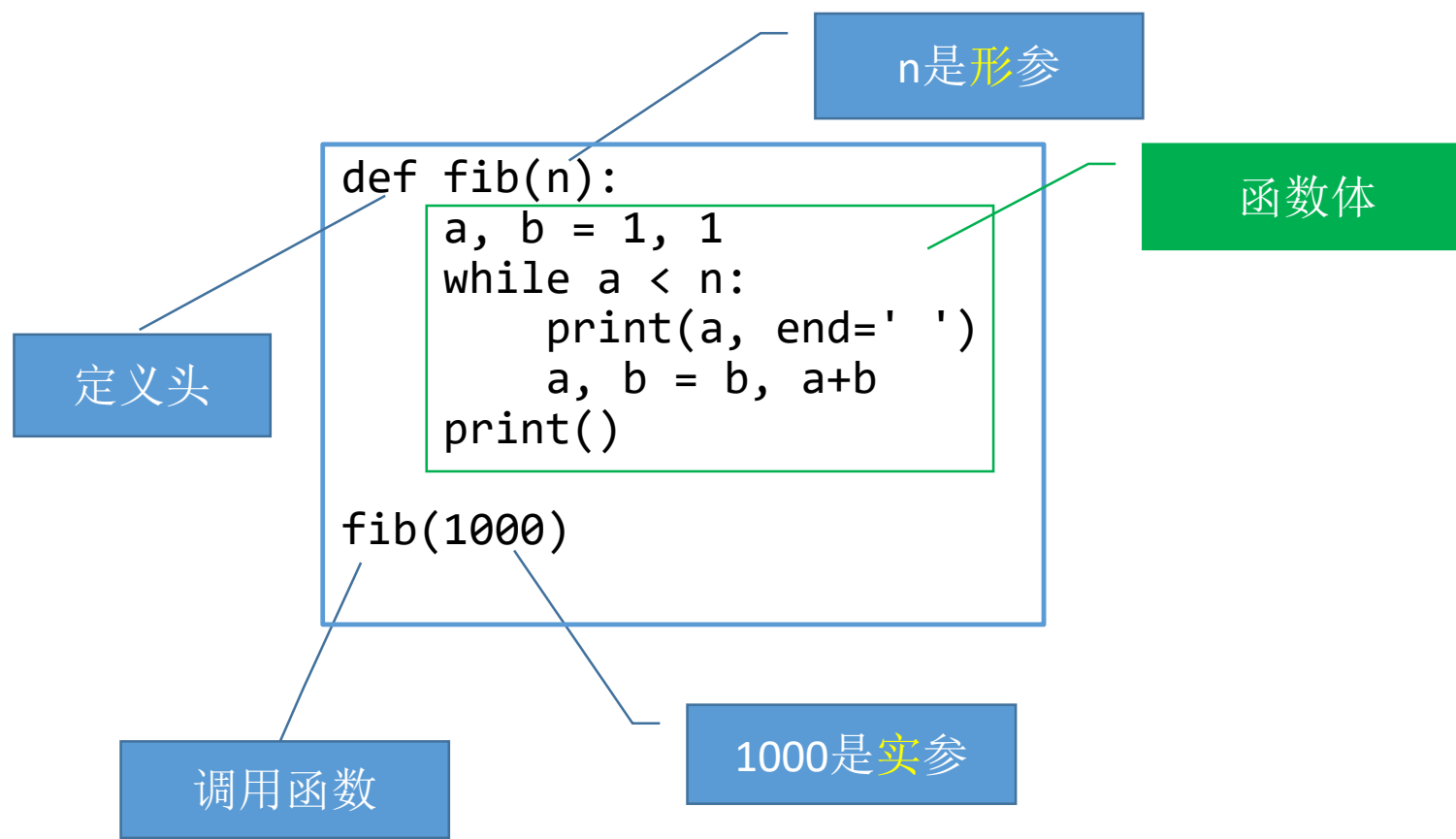


## 5.2 函数参数

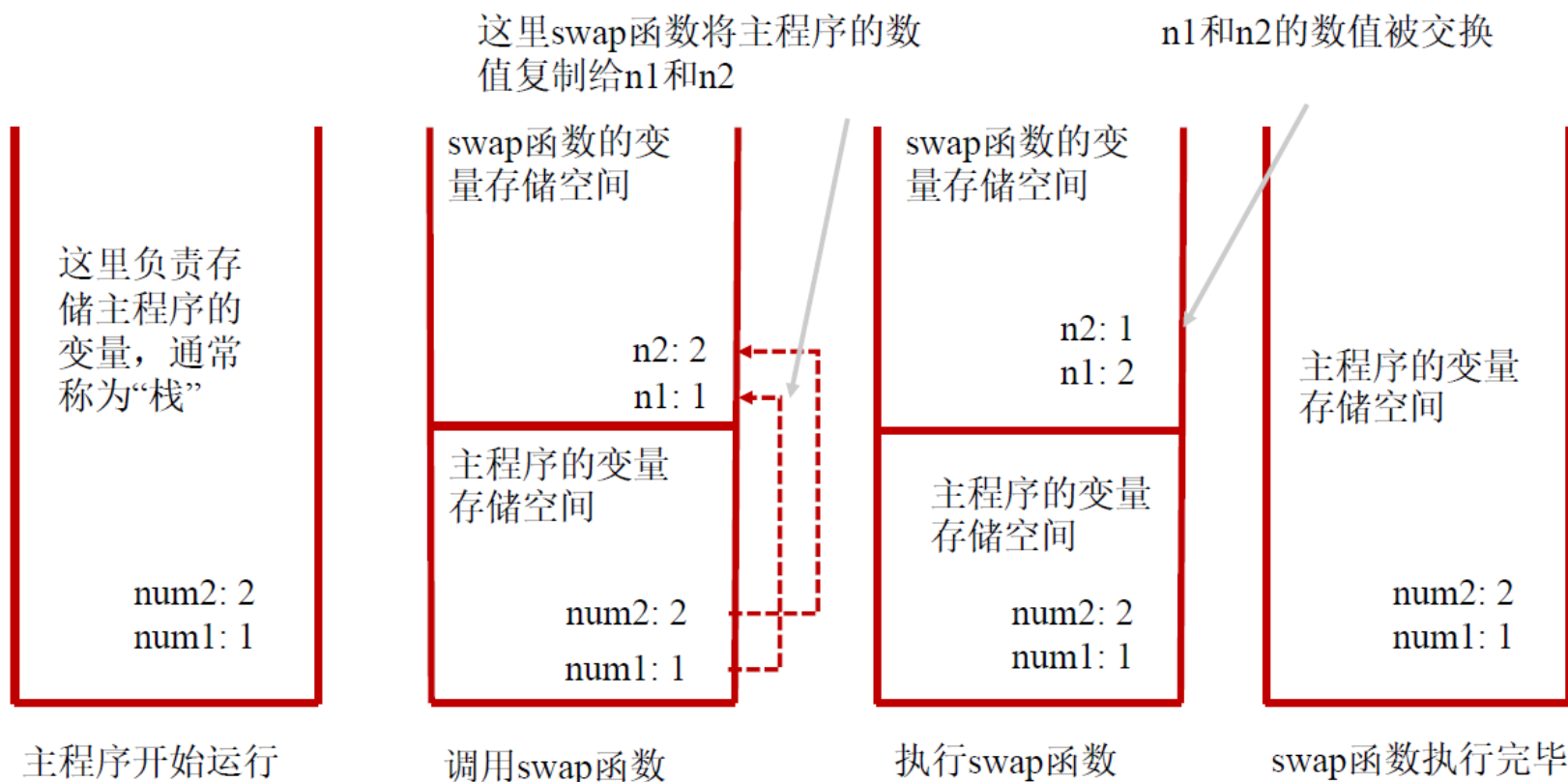
- 函数定义时圆括弧内是使用逗号分隔开的**形参**列表（parameters）
  - 函数可以有多个参数，也可以没有参数，但定义和调用时一对圆括弧必须要有，表示这是一个函数并且不接收参数。
- 调用函数时向其传递**实参**（arguments）
  - 根据不同的参数类型，将实参的**引用**传递给形参。
- 参数的传递过程，实际上是一个赋值的过程。在调用函数时，调用者的实际参数自动赋值给函数的形式参数变量。
- 定义函数时**不需要声明参数类型**，解释器会根据实参的类型自动推断形参类型，在一定程度上类似于函数重载和泛型函数的功能。

# 5.1.1 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。



# 理解参数传递过程（重要！）



```
def swap(n1,n2):  
    t=n1  
    n1=n2  
    n2=t  
    print(n1,n2)
```

```
num1, num2=1,2  
swap(1,2)  
print(num1,num2)
```



## 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：

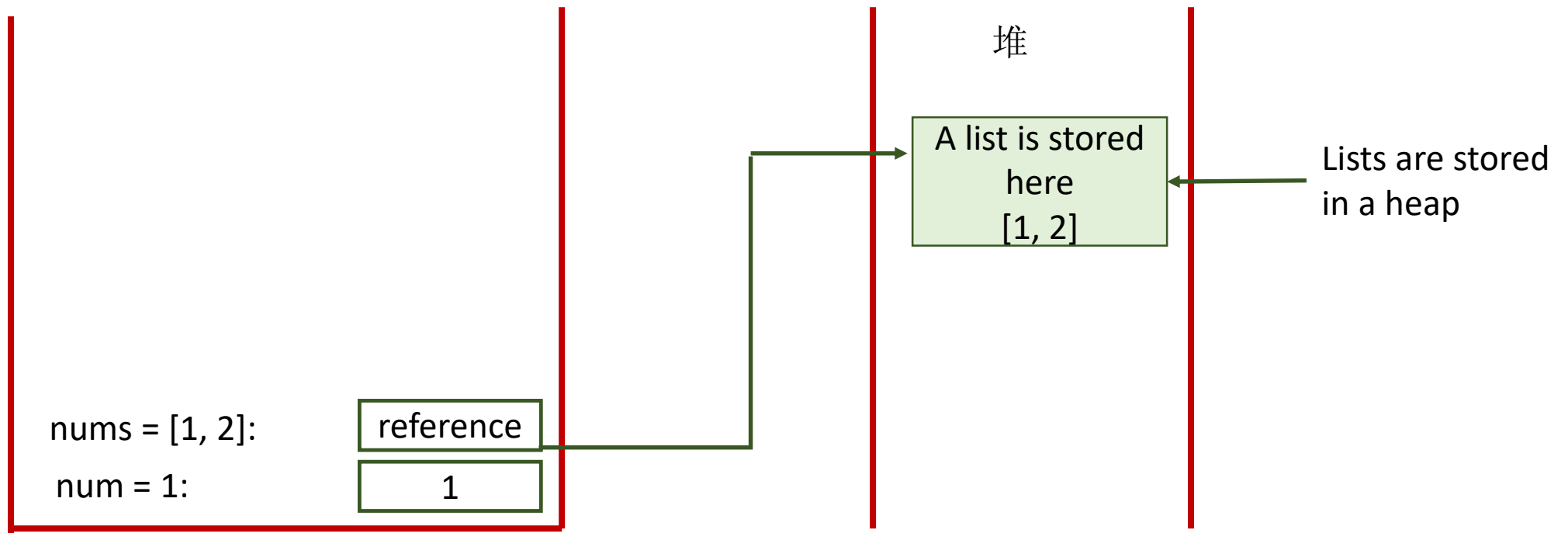
```
>>> def addOne(a):  
    print(id(a), ': ', a)  
    a += 1  
    print(id(a), ': ', a)
```

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

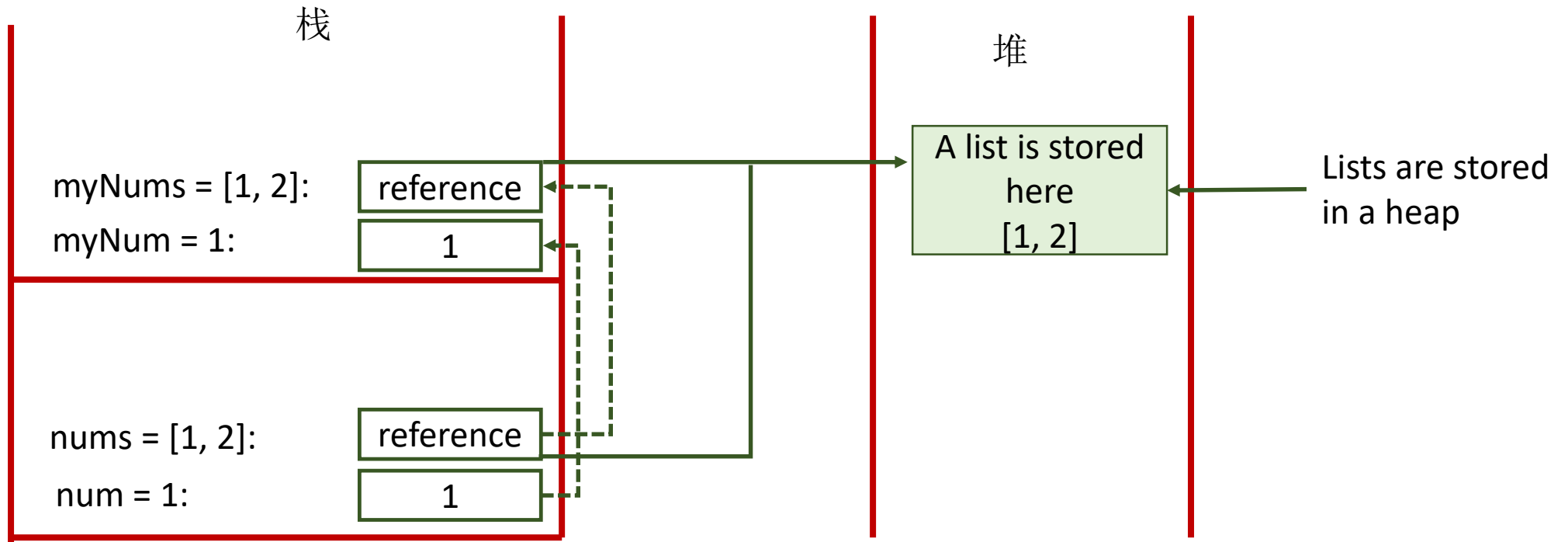
注意：此时a的地址与v的地址相同

现在a的地址和v的地址不一样了

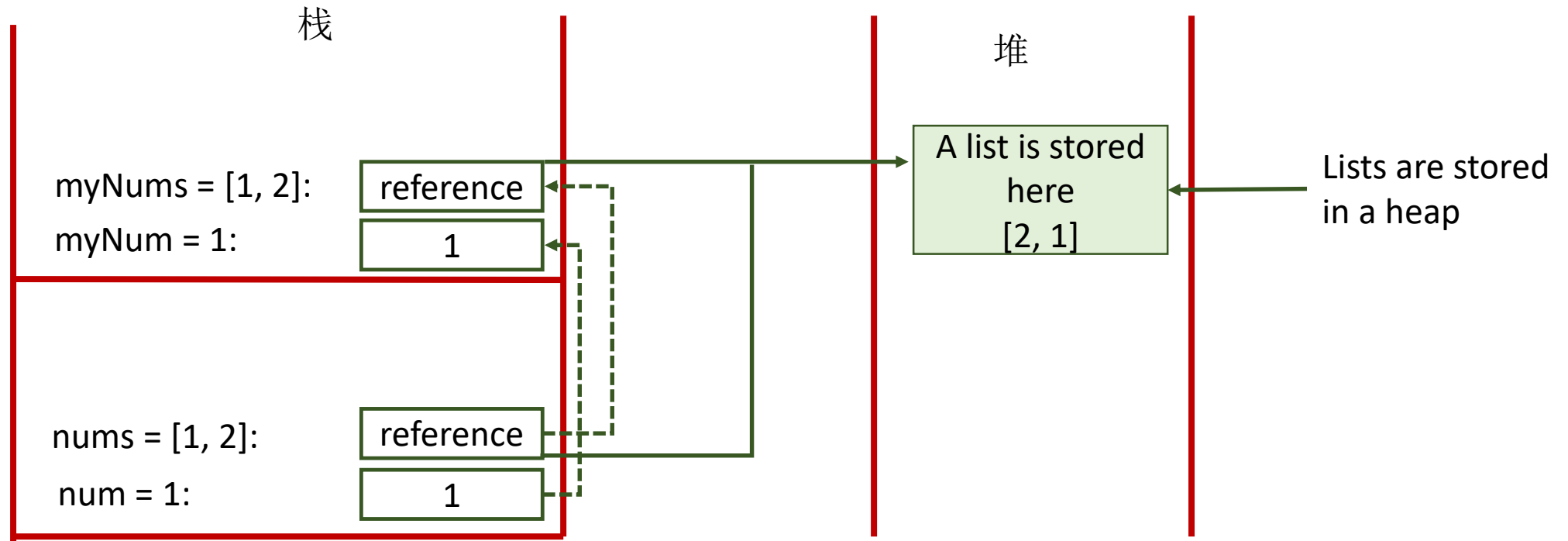
# 如果传递的是List...



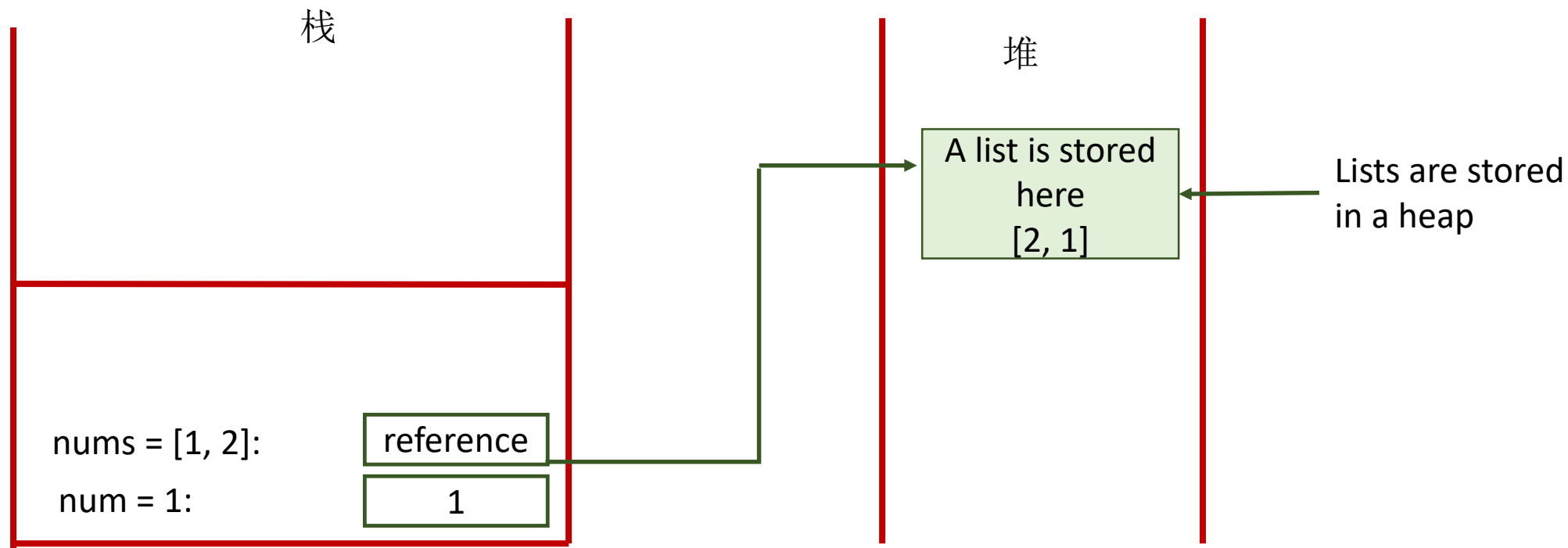
# 如果传递的是List...



# 在Heap里存储的list元素互换



# 调用结束后, 上层栈从内存中移除



```
def swap(n1,n2):  
    t=n1  
    n1=n2  
    n2=t  
    print(n1,n2)
```

```
num1, num2=1,2  
swap(1,2)  
print(num1,num2)
```

```
def swap(list1):  
    t=list1[0]  
    list1[0]=list1[1]  
    list1[1]=t  
    print(list1)
```

```
list=[1,2]  
swap(list)  
print(list)
```

## 5.2 函数参数

```
def modify(v):                # 使用下标修改列表元素值
    v[0] = v[0]+1
a = [2]
modify(a)
print(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)
print(a)
[2, 3]
```

## 5.2 函数参数

- 因此，如果传递给函数的实参是可变序列（list, dict, set），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 modify1(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'}
print(a)
```

```
modify1(a)
print(a)
{'age': 37, 'name': 'Dong', 'sex': 'Male'}
{'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
```

# 玩个游戏吧



# 编写程序，检查用户输入的一个字符串是否符合密码规则

- 包括以下规则
  - -至少6个字符
  - -至少一个大写字符（调用isupper函数）
  - -至少包含一个数字（调用isdigit函数）
- 当且仅当同时成立，则祝贺用户密码成立
- 如果不成立，则要求用户重新输入
  
- 请学号被4整除的同学编写主程序main()，假设存在三个函数：hasSix()、hasUpper()、hasDigit()，这三个函数在符合条件时返回1，否则返回0.
- 请学号被4除余1的同学，编写hasSix()
- 请学号被4除余1的同学，编写hasUpper()
- 请学号被4除余1的同学，编写hasDigit()

- `def hasSix(passwd):`
- `"""`
- 判断密码长度是否不小于6
- `:param passwd:` 密码, `str`类型
- `:return:` 长度不小于6返回`True`, 否则返回`False`
- `"""`
- 
- `def hasUpper(passwd):`
- `"""`
- 判断密码中是否包含大写字母
- `:param passwd:` 密码, `str`类型
- 内部函数`isupper()`
- `:return:` 包含大写字母返回`True`, 否则返回`False`
- `"""`

```
def hasDigit(passwd):
    """
    判断密码中是否包含数字
    :param passwd: 密码, str类型
    :return: 包含数字返回True, 否则返回False
    内部函数isdigit()
    """
```

```
def main():
    """
    主函数, 运行此函数提示用户输入密码, 如果密码格式正
    确则返回, 否则提示用户密码错误且让用户重新输入
    :return: None
    """

    main()
```

# 举例

```
numbers = []                                #使用列表存放临时数据
while True:
    x = input('请输入一个成绩: ')
    numbers.append(float(x))
while True:
    flag = input('继续输入吗? (yes/no) ')
    if flag.lower() not in ('yes', 'no'): #限定用户输入内容必须为yes或no
        print('只能输入yes或no')
    else:
        break
    if flag.lower()=='no':
        break

print(sum(numbers)/len(numbers))
```

## 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
```

- 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='ZC', times =1 ):  
    print((message+' ') * times)  
print(say.__defaults__)  
( 'ZC',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]))  
print(demo('aaa', ['a', 'b']))  
print(demo('a'))  
print(demo('b'))  
[1, 2, 3, 4, '5']  
['a', 'b', 'aaa']  
['a']  
['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)
```

#参数n的值仅取决于i的当前值

```
f()
```

```
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)
```

#重新定义函数

#Python允许在同一程序里重新定义函数

```
f()
```

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

#定义函数

```
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))          #指定关键参数时，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

- Python 中规定每个变量都有它起作用的代码范围称为变量的作用域，即变量只有在作用域范围内才是可用的。
- 不同作用域内变量名可以相同，互不影响。
- 在全部程序中都可以调用的变量，称为全局变量
- 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，并要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()
```

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

#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

## 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

- Python 中提供了一项非常有用的功能：利用 **lambda表达式**来替代 `def`，创建一个临时简单函数。
- 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。
- 定义匿名函数的语法如下所示：
  - **lambda 参数名: 表达式**

## 5.4 lambda表达式

```
f = lambda x, y, z: x+y+z  
print(f(1,2,3))
```

#可以给lambda表达式起名字  
#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

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

```
print(demo(5))
```

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

```
print(list(map(lambda x: demo(x), a_list))) #在lambda表达式中调用函数  
[1, 4, 9, 16, 25]
```