第5章 函数和模块

本章学习目标

(1) 掌握函数的定义和调用方法；

(2) 掌握函数的多种参数形式和应用场景；

(3) 掌握 lambda 表达式和变量的作用域；

(4) 掌握函数的高级特性；

(5) 能够进行模块化编程，并对应用进行打包；

(6) 掌握密码学中的雪崩效应。

本章内容概要

编程大师 MartinFowler 先生曾经说过："代码有很多种坏味道，重复是最坏的一种！"要写出高质量的代码首先要解决重复代码的问题。函数是一段具有特定功能的、可重用的语句组，是一种功能抽象。面向过程编程中函数是代码复用的重要手段。

本章首先从函数的定义、函数的调用和函数的参数等方面介绍函数的编写和复用，接着介绍 Python 开发中有关函数的几个高级特性，进而介绍 Python 中代码复用重要方式，即模块化编程。

本章的安全专题主要介绍对称加密算法 AES、哈希算法的雪崩效应和脚本验证。

5.1 函数的定义和调用

函数的使用包括两部分：函数的定义和函数的调用。


观看视频

5.1.1 函数的定义方式

Python 定义一个函数使用 def 关键字，语法形式如下：

```
def 函数名(<参数列表>):
    语句块
    [return <返回值列表>]
```

说明：

(1) 函数代码块以 def 关键字开头，后接函数名和圆括号，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；

(2) 函数形参不需要声明其类型，也不需要指定函数返回值类型；

(3) 参数列表是调用该函数时需要传递给它的值，可以有零个、一个或多个，当传递多个参数时各参数用逗号分隔；

(4) 函数的主体语句块相对于 def 关键字要保持一致的缩进。

示例：定义一个求阶乘的函数。要求接收一个整型参数，返回值是该参数阶乘计算的结果。

在下面的 factorialExp1.py 示例程序中，第 2～6 行是函数的定义，函数名为 factorial，参数为 num，返回值为 result。第 6 行通过 return 语句将计算结果返回。第 7、8 行代码通过函数名调用函数，分别传入实参 4 和 10，并定义变量 x 和 y 接收返回值。

```
1  #factorialExp1.py
2  def factorial(num):
3      result = 1
4      for n in range(1, num + 1):
5          result *= n
6      return result
7  x = factorial(4)
8  y = factorial(10)
9  print(x)
10 print(y)
```

运行结果如下：

```
24
3628800
```

程序在运行过程中，遇到第 2 行的关键字 def 时，跳过该行以及后面缩进的语句块(第 3～6 行代码)，执行后面的语句；当遇到第 7 行时，调用函数 factorial，接收实参 4，程序这时才运行第 3～6 行代码。运行时，实参 4 将被传递给形参 num，继续运行程序，运行到第 6 行代码时，遇到 return 语句，这时，函数将 result 返回到调用的地方(即：将 result 的值赋给 x)，函数调用结束，继续执行第 8 行代码以及后面的程序。

本示例只是为了讲解函数的定义和使用，Python 内置模块 math 中的 factorial()函数已经实现了计算阶乘的功能，因此实际开发中直接使用这个现成的函数而不用做低级的重复性实现工作。

```
>>> import math
>>> math.factorial(4)
24
```

前面定义的函数 factorial(num)仅仅有一个形参。一个函数可以设置多个形参。在下面的 funExp.py 程序中，函数 printinfo(name,age)有两个参数，函数主体是第 4～6 行的代码，它们的缩进相同。该函数无返回值。

```
1  #funExp.py
2  #函数形参不需要声明其类型
3  def printinfo(name,age):
4      print("你的姓名是",name)
5      age = age+1
6      print("明年你的年龄是",age)
7
8  print(printinfo('Alice',19))          #调用函数时,第二个参数为整型
```

运行结果如下：

```
你的姓名是 Alice
明年你的年龄是 20
None
```

通过函数名调用函数功能，对函数的各个参数赋予实际值，实际值可以是数据，也可以是在调用函数前已经定义过的变量。函数被调用后，实参参与函数内部代码的运行，如果有结果则进行输出。函数执行结束后，根据 return 保留字决定是否返回结果，如果返回结果，则结果将被放置到函数被调用的位置，函数调用完毕，程序继续运行。

5.1.2 函数说明文档

通过 help()函数查看函数的帮助文档，这是开发者经常用到的操作，也是开发者需要掌握的基本技能。如何为自定义的函数编写说明文档呢？函数的说明文档是放在函数声明之后、函数体之前的一段字符串，例如，下面的 funHelp.py 示例程序中的第 3～7 行。一个函数在定义时如果写了说明文档，除了可以使用 help()查看帮助文档之外，如第 12 行代码所示，还可以通过函数的__doc__属性访问函数的说明文档，如第 14 行代码所示。这两种查看方式的输出有些差别，请读者注意观察运行结果。

```
1  #funHelp.py
2  def is_prime(num):
3      """
4      num 获取一个整数
5      is_prime(num)
6          判断 num 是不是一个素数
7      """
8      for factor in range(2, num):
9          if num % factor == 0:
10             return False
11     return True if num != 1 else False
12 help(is_prime)
13 #可以通过函数的__doc__属性访问函数的说明文档
14 print(is_prime.__doc__)
```

运行结果如下：

```
Help on function is_prime in module __main__:

is_prime(num)
    num 获取一个整数
    is_prime(num)
        判断 num 是不是一个素数

    num 获取一个整数
    is_prime(num)
        判断 num 是不是一个素数
```


观看视频

5.1.3 返回值

return［表达式］用以结束函数，函数将表达式的结果返回到调用的地方。函数可以没有 return，此时函数并不返回值；如果一个函数没有 return 语句，相当于返回 None。

例如，在 funExp.py 示例程序中，运行结果的最后一行为 None。而如果有返回值，则返回值可以是一个值，也可以是多个值。

1. 返回一个值

示例：编写函数，进行阿拉伯数值与中文数值之间的转换。要求用户输入阿拉伯数字金额，输出转换后的中文大写格式金额。

假设最高位考虑到亿，最低位考虑到分（如数字金额为 1023，转换为中文大写金额为：壹仟零佰贰拾叁元零角零分）。

编程思路：

（1）如果是 1023 元，首先应该得到叁元，然后依次是贰拾元、零佰和壹仟，之后拼接得到“壹仟零佰贰拾叁元零角零分”。思考，如何得到 3、2、0 和 1 这几个数字呢？首先想到的是，采用 Python 中的取模运算，1023%10，所得结果是 3；然后 102%10，得到 2；其他数字 0 和 1 是类似的过程。进一步思考，如何从 1023 得到 102 呢？采用是 Python 中的整除运算，即 1023//10，所得结果是 102，重复本步骤就可以得到其他对应的阿拉伯数字。

（2）如何从阿拉伯数字转换为中文数字呢？如果有一个中文序列“零壹贰叁肆伍陆柒捌玖”，那么可以看到，数字的大写中文序列对应的索引就是阿拉伯数字。例如，当阿拉伯数字是 3 时，观察中文序列发现，“叁”的索引值就是 3。

（3）如何加上中文单位？观察算法发现，针对 1023 元，第一个 3 的单位是元，2 的单位是拾，0 的单位是佰。那么，建立一个序列：“元拾佰仟万拾佰仟亿”，每得到一个中文数字，则从这个序列中取一个单位，并逐渐向右移动。即第一次取元，第二次取拾，第三次取佰，以此类推。

（4）如果数据有小数点，则单位变为“分角元拾佰仟万拾佰仟亿”。取阿拉伯数字前，首先将数字乘以 100，然后从编程思路（1）开始处理。在得到对应的中文数字时，应当对中文进行拼接（即字符串拼接）。请注意拼接顺序：先得到的是小单位，所以最新的数据在前，已得到的数据在后。

在下面的 cashChange.py 示例程序中，第 2～13 行代码定义 change(m_count)函数，参数 m_count 是用户输入的阿拉伯数字金额，第 13 行代码输出转换后的中文大写格式金额。

由于考虑带有小数点，第 15 行代码使用 eval()函数，目的是将输入的数据去除两边的引号，这时，cash_1 是一个浮点数。然后在第 16 行调用函数。在函数内转换完毕后，将转换后的字符串通过 return str_1 语句返回中文数据（本程序返回的是字符串，也可以返回其他类型的数据）。

```
1  #cashChange.py
2  def change(m_count):
3      c_count="零壹贰叁肆伍陆柒捌玖"
4      c_unit='分角元拾佰仟万拾百千亿'
5      m_count=m_count*100
6      str_1=''
7      for i in range(len(str(m_count))):
8          k=m_count%10
9          str_1=c_count[int(k)]+c_unit[i]+str_1
10         m_count=m_count//10
11         if m_count==0:
```

```
            break
    return str_1

cash_1=eval(input('请输入金额:'))
k=change(cash_1)
print("转换后的大写金额为:",k)
```

运行结果如下：

```
请输入金额:1023
转换后的大写金额为: 壹仟零佰贰拾叁元零角零分
```

2. 返回多个值

一个函数除了可以一次性返回一个数值之外，也可以同时返回多个数值。这种返回有两种方式：第一种方式是直接返回多个数值，用逗号分隔，这种直接返回多个数值时，Python 自动将它们封装为元组；第二种方式是将多个数值用字典封装并返回。

在下面的 returnExp1.py 示例程序中，函数 gcd_lcm()的返回值有两个，分别是最大公约数和最小公倍数。在第 8 行通过 return 定义两个返回值。在第 10 行调用函数 gcd_lcm()，并将返回值放到变量 z 中，变量 z 以元组形式封装两个返回值。可以使用元组索引来检索返回值，如第 13、14 行代码所示。这种方式的缺点是调用者必须知道哪个返回值对应最大公约数，哪个返回值对应最小公倍数。也可以像第 16 行代码那样在调用时使用序列解包获取多个返回值，用两个变量接收返回值，这种方式的缺点是需要知道返回值的个数。

```
1  #returnExp1.py
2  import math
3  def gcd_lcm(a,b):
4      #求最大公约数
5      x = math.gcd(a,b)
6      #求最小公倍数
7      y = a * b // x
8      return x,y          #同时返回多个数值,多个数值用逗号分隔,视为一个元组
9
10 z = gcd_lcm(12,20)    #用一个数据接收返回值,这个数据是元组类型的数据
11 print("返回值的类型是:",type(z))
12 print("返回值是:",z)
13 print("最大公约数是:",z[0])
14 print("最小公倍数是:",z[1])
15 ## 使用序列解包获取多个返回值
16 mygcd,mylcm = gcd_lcm(12,20) #用与返回数据个数相同的变量接收数据,将分别赋值
17 print("最大公约数是:",mygcd)
18 print("最小公倍数是:",mylcm)
```

运行结果如下：

```
返回值的类型是: <class 'tuple'>
返回值是: (4, 60)
最大公约数是: 4
最小公倍数是: 60
最大公约数是: 4
最小公倍数是: 60
```

一个相对友好的方法是返回一个字典对象。在下面的 returnExp2.py 示例程序中，第 8 行代码将返回值封装为字典，其中字典的键为"gcd"和"lcm"，分别对应值 x 和 y。调用函数时，不必担心返回值的顺序，也需要使用索引来获取具体的返回值，只需要用键"gcd"来检索 x，用键"lcm"检索 y 即可。

```
1  #returnExp2.py
2  import math
3  def gcd_lcm(a,b):
4      #求最大公约数
5      x = math.gcd(a,b)
6      #求最小公倍数
7      y = a * b // x
8      return {"gcd":x,"lcm":y }      #用字典封装返回值
9  z = gcd_lcm(12,20)                 #用一个数据接收返回值,这个数据是元组类型的数据
10 print("返回值的类型是:",type(z))
11 x = z["gcd"]
12 y = z["lcm"]
13
14 print("最大公约数是:",x)
15 print("最小公倍数是:",y)
```

运行结果如下：

```
返回值的类型是: <class 'dict'>
最大公约数是: 4
最小公倍数是: 60
```

示例：鸡兔同笼问题。假设一个笼中有鸡和兔，用户输入笼中动物的腿的数量(要求偶数)，求笼中最少的动物数量，对应输出鸡和兔分别是多少只。

分析：笼中总共有 n 条腿。因为需要求最少的动物数量。先假设笼中所有的动物都是兔，腿的数量必然是 4 的倍数(即：n/4 是一个整数或者 n%4 的结果是 0)，如果不是 4 的倍数，则剩余的腿必然是鸡的腿。

```
1  #chickenRabit.py
2  def chrab(n):
3      if n%2==0:# 腿的数量应当是偶数
4          rabit=n//4
5          chicken=int((n-rabit*4)/2)
6          return {"chicken":chicken,"rabit":rabit}
7      else:
8          print("数值错误,请输入偶数个腿数")
9
10 chickenAndrabit=int(input("请输入腿数:"))
11 z = chrab(chickenAndrabit)
12 if z!=None:
13     print("鸡的数量是:",z["chicken"],"\n兔子的数量是:",z["rabit"])
```

运行结果如下：

```
请输入腿数:18
鸡的数量是: 1
兔子的数量是: 4
```

再次运行结果如下：

```
请输入腿数:5
数值错误,请输入偶数个腿数
```

5.1.4 函数的嵌套

对于递增有序顺序表，查找过程可以采用二分查找，也称为折半查。查找过程是：首先将要查找的元素和有序表的中间元素比较，如果相等，则查找成功；如果大于中间元素，则在后半区间继续查找；如果小于中间元素，则在前半区间查找。不断重复上述过程，直到查找成功或查找失败为止。二分查找示意图如图 5-1 所示。

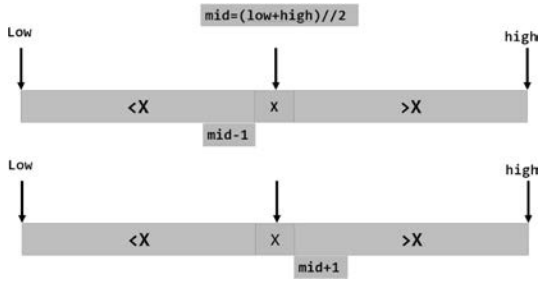

图 5-1 二分查找示意图

例如，对于有序列表[5,10,25,27,30,35,45,49,50,52,55,60,70]，查找元素 30 的过程如图 5-2 所示。

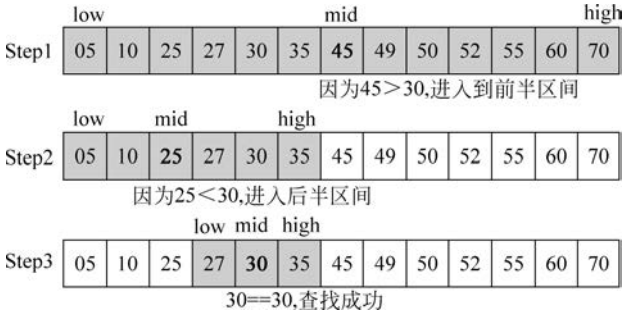

图 5-2 查找元素 30 的过程

(1) low=0、high=12、mid=6，要查找的元素 30 和 mid 位置的 45 比较，由于 30 小于 45，所以在前半区间继续查找；

(2) low=0、high=mid−1=5、mid=2，要查找的元素 30 和 mid 位置的 25 比较，由于 30 大于 25，所以在后半区间继续查找；

(3) low=mid+1=3、high=5、mid=4，要查找的元素 30 和 mid 位置的 30 比较，相等，查找成功，结束。

查找元素 48 的过程如图 5-3 所示，当 low>high 时说明查找失败。

通过以上过程，可以看到查找过程是一个递归的过程。Python 允许嵌套定义函数。递归程序是函数自己调用自己。例如，在下面的 binSearch.py 二分查找示例程序中，第 2 行代码定义函数为 binary_search()，第 10、12 行代码调用 binary_search()函数，此时传入的

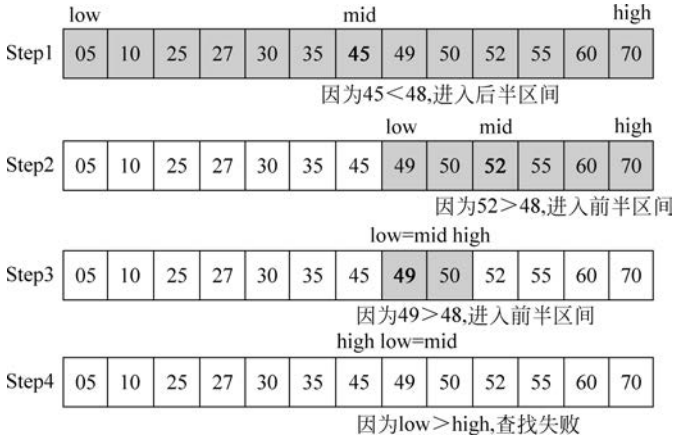

图 5-3　查找元素 48 的过程

参数不同。

```
#binSearch.py
def binary_search(target,data,low,high):
    if low > high:
        return False
    else:
        mid = (low+high)//2
        if data[mid] == target:
            return True
        elif target < data[mid]:
            return binary_search(target,data,low,mid-1)
        else:
            return binary_search(target,data,mid+1,high)

number = [1,3,5,7,9]

print("请输入要查找的数据:")
target = int(input())
print(binary_search(target,number,0,(len(number)-1)))
```

运行结果如下：

```
请输入要查找的数据:
3
True
```

再次运行结果如下：

```
请输入要查找的数据:
8
False
```

递归中最危险的事情是无限递归。Python 语言中默认的递归数为 1000，如果超过这个限制，就会产生运行时错误 RuntimeError：maximum recursion depth exceeded。但是 Python 语言中用户可以限定递归的总数，通过 sys 模块中的 setrecursionlimit()进行设置，通过 getrecursionlimit()获得当前的递归深度。

```
>>> import sys
>>> sys.getrecursionlimit()
1000
>>> sys.setrecursionlimit(2000)
>>> sys.getrecursionlimit()
2000
```


观看视频

5.1.5 函数执行的起点

Python 使用缩进对齐组织代码的执行，所有没有缩进的代码（非函数定义和类定义），都会在载入时自动执行，这些代码，可以认为是 Python 的 main 函数。

前面章节的代码经常使用 import 语句，将其他模块载入进来。而这些载入的模块中，每个文件（模块）都包含一些没有缩进的代码，这些文件在载入时自动执行。为了区分主执行文件和被调用文件，Python 引入了一个变量__name__，当文件是被调用时，__name__的值为模块名，当文件被执行时，__name__为'__main__'。这个特性，为测试驱动开发提供了极好的支持，我们可以在每个模块中写上测试代码，这些测试代码仅当模块被 Python 直接执行时才会运行，代码和测试完美地结合在一起。

Python 语言的每个脚本都可独立运行，C 语言中程序的起点是 main，在 Python 脚本程序中可以不写 main 函数，程序运行的起点在哪里呢？在 Python 程序中，有一特殊的顶层模块，它包含“主程序”。当这个模块被导入运行时，其__name__属性为__main__，而当这个模块被其他文件导入时，则其__name__属性为该模块的名字。

示例：测试脚本的__name__属性。在下面的 mainExp1.py 示例程序中，当在命令行运行该模块，或在 IDLE 中运行该模块时，它作为主程序即顶层模块运行。第 4 行代码进行条件判断，从运行结果可以看出是满足条件的，也就说明了脚本的__name__属性是__main__。同时在程序的第 7 行中输出结果也可以说明 mainExp1.py 脚本的__name__属性是__main__。

```
1  #mainExp1.py
2  def test():
3      print("hello")
4  if __name__ == '__main__':
5      print('main')
6      test()
7  print(__name__)
```

运行结果如下：

```
main
hello
__main__
```

下面建立两个文件，分别命名为 mainExp2.py 和 mainExp3.py，两个脚本文件保存在同一个目录中。

```
1  #mainExp2.py
2  def fun_1():
```

```
    print(f'{__name__}')
def fun_2():
    print('fun_2 函数来自 mainExp2')
if __name__ == "__main__":
    fun_2()
    print(f'程序名是{__name__}')
```

```
# mainExp3.py
from mainExp2 import fun_1
fun_1()
print(f'程序名字是{__name__}')
```

运行 mainExp3.py，运行结果如下：

```
mainExp2
程序名字是__main__
```

在本代码中导入了 mainExp2 并运行该模块中的 fun_1()函数，此时 fun_1()函数中的__name__输出是 mainExp2(也就是保存的脚本名)，但程序的__name__还是__main__。

如果希望定义的某些函数只能被其他模块调用，则可以在程序中添加一些代码，如下第14、15行代码。此时这个脚本作为顶层模块运行时，满足第14行的条件判断，给用户输出提示信息，告诉用户该模块要作为模块使用。关于如何自定义和使用模块，在5.6节进行介绍。

```
# mainExp4.py
def binary_search(target,data,low,high):
    if low > high:
            return False
        else:
            mid = (low+high)//2
            if target == data[mid]:
                return True
            elif target < data[mid]:
                return binary_search(target,data,low,mid-1)
            else:
                return binary_search(target,data,mid,high)

    if __name__ == '__main__':
        print("请作为模块使用")
```

运行结果如下：

```
请作为模块使用
```

5.2　函数的参数

Python 语言中函数定义、调用和返回值都比较简单，但使用时非常灵活，特别是参数方面尤为突出。定义函数的形参时，其定义方式多种多样，可以是位置参数、默认参数、可变参数、关键字参数或命名关键字参数。通过不同的参数使得函数定义出来的接口不但能处理

复杂的参数，还可以简化调用。下面具体介绍各种形式的参数。

5.2.1 位置参数


观看视频

位置参数是指在定义函数的形参时，各个参数的位置和参数的数量是确定的，由于每个参数都有一定的意义，因此在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。位置参数又称为必备参数。

示例：二分查找算法。在下面的 positionParaExp1.py 示例程序中，定义 binary_search() 函数时，target、data、low、high 等参数分别表示了目标数据、数据序列、低位、高位，而这些数据是在实际查找中用到的，因此在调用函数 binary_search()时，实参必须按照 target、data、low、high 位置进行赋值，不能颠倒，否则将无法正确运行或无法得到正确的结果。如第 15 行代码是正确的调用情况，而第 16 行代码是错误的调用情况，一般情况下会抛出异常。

```
1  #positionParaExp1.py
2  def binary_search(target,data,low,high):
3      if low > high:
4          return False
5      else:
6          mid = (low+high)//2
7          if target == data[mid]:
8              return True
9          elif target < data[mid]:
10             return binary_search(target,data,low,mid-1)
11         else:
12             return binary_search(target,data,mid+1,high)
13
14 number = [1,3,5,7,9]
15 res = binary_search(3,number,0,len(number)-1)
16 # res = binary_search('3',number,0,(len(number)-1))
17 print(res)
```

运行结果如下：（注释第 16 行代码，执行第 15 行代码）

```
True
```

再次运行结果如下：（注释第 15 行代码，执行第 16 行代码）

```
TypeError: '<' not supported between instances of 'str' and 'int'
```

示例：打印用户的姓名和年龄。

在下面的 positionParaExp2.py 示例程序中，如果第 8 行调用函数 printinfo(29,'Alice')时，实参姓名和年龄的顺序不正确，导致抛出 TypeError 异常。

```
1      #positionParaExp2.py
2      #打印用户姓名和年龄
3  def printinfo(name,age):
4          print("你的姓名是",name)
5          age = age+1
6          print("明年你的年龄是",age)
7      printinfo('Alice',29)          #正确调用
8      #printinfo(29,'Alice')         #错误调用
```

运行结果如下：(注释第 8 行代码，执行第 7 行代码)

```
你的姓名是 Alice
明年你的年龄是 30
```

运行结果如下：(注释第 7 行代码，执行第 8 行代码)

```
你的姓名是 29
TypeError: can only concatenate str (not "int") to str
```

5.2.2　默认值参数


观看视频

在设计一个函数时，如果函数的某个参数在大部分情况下是某个确定的值，可以将这个参数设置为默认值参数，从而简化函数的调用。

场景 1：定义一个上户口的函数，由于初生婴儿在上户口时，年龄是 0 岁，而其他人迁移户口，则年龄不定。但初生婴儿上户口的比例很大，此时可以将年龄设置为默认值参数，值为 0。这样在婴儿上户口时，只输入姓名，不输入年龄，而其他人上户口，需要输入年龄。

场景 2：大学生注册，大部分学生的年龄是 18 岁，此时可以将年龄设置默认值参数，值为 18 岁。

示例：学生年龄为 18 的默认值参数。在下面的 defaultParaExp1.py 示例程序中，函数 printinfo()中的 age 参数为默认值参数，默认值为 18。第 9 行代码调用函数时只给定了一个实参 name，参数 age 使用了默认值参数。在第 12 行代码调用函数时，不使用默认值参数，给定了两个实参。

```
 1  #defaultParaExp1.py
 2  #测试默认值参数
 3  def printinfo(name,age = 18):
 4      print("你的姓名是",name)
 5      age = age+1
 6      print("明年你的年龄是",age)
 7
 8  #仅给第一个形参传递参数,第二个形参使用默认值
 9  printinfo('白居易')
10
11  #两个形参都传递参数
12  printinfo('杜甫',19)
```

运行结果如下：

```
你的姓名是 白居易
明年你的年龄是 19
你的姓名是 杜甫
明年你的年龄是 20
```

可以发现，当定义函数时使用了默认值参数，在调用函数时，如果该参数有数据传入，则形参的值为传入的数据；如果在调用时没有给默认值参数传递数据，则该形参使用默认值作为其实参。

默认值参数必须出现在函数形参列表的最右端，任何一个默认值参数右边不能有非默认值参数。如果默认参数不是在参数最后，则 Python 解释器会报错，抛出语法错误异常。

例如，在下面的 defaultParaExp2.py 示例程序中，运行结果报告错误提示，默认值参数后面跟着非默认值参数。

```
#defaultParaExp2.py
#测试默认值参数位置
def printinfo(age = 18, name):
    print("你的姓名是",name)
    age = age+1
    print("明年你的年龄是",age)

#第二个参数使用默认值
printinfo('白居易')

#两个参数都不使用默认值
printinfo(19,'杜甫')
```

运行结果如下：

```
SyntaxError: non-default argument follows default argument
```

在一般情况下，默认值参数指向不变对象，如 str、int、None 等。当默认值参数是可变对象时，如 list、dict 等可变对象，会导致数据错误。例如，在下面的 defaultParaExp3.py 示例程序中，第 7、8 行代码调用 fun()函数时，不能得到期望的正确结果。这是因为 fun()函数的第二个参数 L 设置为默认值参数，但是列表 L 是可变对象，因此每次调用 fun()函数后，都会使列表 L 增加一个元素。即它只是在第一次调用时初始化，再次调用时，其值已经发生了变化。

```
#defaultParaExp3.py
#测试默认值参数为可变对象
def fun(a, L=[]):
    L.append(a)
    print(f'{id(L)}')
    return L

print(fun(1))
print(fun(2))
print(fun(3))
```

运行结果如下：

```
2125732910080
[1]
2125732910080
[1, 2]
2125732910080
[1, 2, 3]
```

程序的本意是每次调用 fun()函数，准备得到不同的列表，即[1]、[2]、[3]。但由于 L 指向了一个列表，而列表是可变的；同时，Python 解释器遇到 def 语句时，就会对函数的默认参数自动构造对象，而且只构造一次，所以多次调用 fun()函数却没有给默认参数传值时，在函数内部实际使用的对象都是同一个，因此在第二次、第三次调用 fun()函数时，列表

都是在已有对象的基础上增加数据。第5行代码输出列表L的内存信息，三次调用输出的结果是相同的，即是同一列表对象。

为了得到正确的结果，可以修改代码如defaultParaExp4.py所示，将默认值参数L设置为None。

```
#defaultParaExp4.py
#测试默认值参数为不变对象
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

运行结果如下：

```
[1]
[2]
[3]
```

通过f.__defaults__查看函数f默认值参数的当前值。

5.2.3　可变参数


观看视频

函数的参数如果是位置参数或默认值参数，参数个数是固定的，调用函数时将参数按合适的格式进行传递。在函数参数的数量不确定的情况下，可以使用Python语言提供的可变参数传递。设置可变参数时在形参前加上*，如*arg，arg以元组的方式接收不确定个数的参数。可变参数又称为不定长参数。

示例：编写一个函数，实现不同个数的数值相加(减、乘、除)。

在实际生活中，可能会遇到3+6+59+36…，但参与运算的数值不确定，在编写代码时并不知道具体有多少数值参与相加(减、乘、除)运算。用户在调用函数时参与运算的格式是确定的，这种情况可以用可变参数形式。

在下面的variableParaExp.py示例程序中，定义函数calc()有三个参数，第一个是操作数，第二个是操作符号，第三个是操作数。其功能是操作数a和操作数c进行b操作。

```
#variableParaExp.py
def calc(a,b, *arg):
    '''
    a:操作数
    b:操作符号(+-*/)
    arg:操作数
    功能:用a和arg进行b的操作
    '''
    if b=='+':
        for item in arg:
            a=a+item
    elif b=='-':
        for item in arg:
```

```
14                  a=a-item
15          return a
16  x=calc(20,'-',4,5,6)
17  print(x)
18  y=calc(20,'+',5,6)
19  print(y)
```

calc()函数定义时，参数有三个，其中，a、b 是位置参数，其值必须明确，而 arg 前面有个 *，表示其是一个元组，用来接收数据(个数不确定)。在第 16 行调用时，20 赋予 a，'-'传递给 b，而 4、5、6 组成一个元组传递给 arg，即：arg=(4,5,6)。在第 18 行调用时，20 赋予 a，'+'传递给 b，而 5、6 组成一个元组传递给 arg，即：arg=(5,6)。


观看视频

5.2.4 关键字参数

如果函数的参数个数和名称都不确定时，则设置为关键字参数。设置方式是在形参前加上 **，如 ** kwarg，kwarg 可以接收关键字参数并以字典形式接收数据。Python 的内置函数大量使用了关键字参数。

在下面的 keywordParaExp.py 示例程序中，定义 person()函数，有三个参数，其中 ** kwarg 为关键字参数，表示接收的参数个数和名称都不确定，并将参数存放到字典中。例如，第 7 行传入 city='NanNing'，即将{'city': 'NanNing'}作为 person()函数的第三个参数。第 8 行传入 gender='M'，job='Engineer'，即将{'gender': 'M','job': 'Engineer'}作为 person()函数的第三个参数。另外，关键字参数也可以先封装为字典，再作为实参传入函数，如第 11、12 行代码所示。

```
1  #keywordParaExp.py
2  def person(name,age, * * kwarg):
3      print('name:',name, 'age:',age, 'other:',kwarg)
4      for key in kwarg:
5          print(key,kwarg[key])
6  #传入任意的其他参数,满足注册的需求
7  person('Bob',35,city='NanNing')
8  person('Peter',45,gender='M',job='Engineer')
9
10 #封装为字典,再传入参数
11 extra = {'city':'NanNing','job':'Engineer'}
12 person('Jack',24, * * extra)
```

运行结果如下：

```
name: Bob age: 35 other: {'city': 'NanNing'}
city NanNing
name: Peter age: 45 other: {'gender': 'M', 'job': 'Engineer'}
gender M
job Engineer
name: Jack age: 24 other: {'city': 'NanNing', 'job': 'Engineer'}
city NanNing
job Engineer
```

5.2.5　命名关键字

关键字参数在使用时，函数的调用者可以传入任意不受限制的关键字参数，在使用过程中传入了哪些，需要在函数内部通过 kwarg 检查。但有时希望将参数的名称确定下来，这就是命名关键字参数。之所以在关键字参数前有一个“命名”，就是指参数的名称已经确定了下来。命名关键字参数在调用时，必须使用已经命名好的参数名称。

命名关键字参数需要在可变参数后面，如果没有可变参数，命名关键字参数需要一个特殊分隔符 *，* 后面的参数被视为命名关键字参数，如果存在可变参数，则不需要 *。

和位置参数不同，命名关键字的函数调用时，需要使用“变量名＝变量值”的方式传递参数，可以不考虑位置顺序。

在下面的 namedKeywordsParaExp.py 示例程序中，定义函数 person(name,age,*,city,job)，其中，形参 name、age 是位置参数，必须按照其位置传递参数；其后是 *，表明参数 city 和 job 为命名关键字。

```
1  #namedKeywordsParaExp.py
2  def person(name,age,*,city,job):
3      print('name:',name, 'age:',age, 'city:',city,'job:',job)
4  person('Jack',24,city='NanNing',job='Engineer')
5  person('Jack',24,job='Engineer',city='NanNing')
6  #person('Jack',24,city='NanNing')
7  #person('Jack',24,city='NanNing',gender='Female')
```

在调用函数 person()时，对于命名关键字参数使用 city＝'nanjing'、job＝'Engineer'方式调用。如第 4 行代码是正确的调用示例，即根据函数定义的命名关键字参数，传入了 city 和 job 实参。输入结果如下。

```
name: Jack age: 24 city: NanNing job: Engineer
```

第 5 行代码是正确的调用示例，即根据函数定义的命名关键字参数，传入了 city 和 job 实参，但顺序有所变动。

```
name: Jack age: 24 city: NanNing job: Engineer
```

第 6 行代码是错误的调用示例，因为只传入了 city 实参，未传入 job 实参。解释器报告错误如下。

```
TypeError: person() missing 1 required keyword-only argument: 'job'
```

第 7 行代码是错误的调用示例，因为在函数定义的命名关键字中没有定义 genger 关键字，解释器报告如下。

```
TypeError: person() got an unexpected keyword argument 'gender'
```

虽然不同类型的参数可以混合使用，但是一般不建议混合使用。

5.2.6　综合实例

已知某博物馆月访客量如表 5-1 所示。


观看视频

表 5-1 某博物馆月访客量

月　份	1	2	3	4	5	6	7	8	9	10	11	12
访问人数/人	2200	3088	4203	4506	3986	3342	5767	6234	3124	6345	1123	2234

要求：自定义函数计算该博物馆指定条件下的月平均访客量，给出以下 4 种要求的函数定义和调用。并调用函数计算 1～9 月的月平均访客量。

（1）使用位置参数，计算 start～end 月的月平均访客量。

（2）使用默认参数 end＝9。

（3）使用命名关键字参数 end。

（4）使用可变参数。

具体实现如下。

（1）位置参数。即在函数定义时，给出起始和终止的月份。

```
#positionPara.py
#使用函数计算 start～end 的月平均访客量,求 start～end 月的平均访客量.
#使用位置参数
def start_to_end(start,end):
    #博物馆的月访客量保存到列表中
    data=[2200,3088,4203,4506,3986,3342,5767,6234,3124,6345,\
          1123,2234]
    sum = 0
    for month in range(start-1,end):
        sum+=data[month]
    avg=sum/(end-start+1)
    print(avg)

start_to_end(1,9)
```

（2）默认值参数 end＝9。

```
#defaultPara.py
#使用默认参数 end=9
def start_to_end(start,end=9):
    #博物馆的月访客量保存到列表中
    data=[2200,3088,4203,4506,3986,3342,5767,6234,3124,6345,\
          1123,2234]
    sum = 0
    for month in range(start-1,end):
        sum+=data[month]
    avg=sum/(end-start+1)
    print(avg)

start_to_end(1)
```

（3）命名关键字参数 end。

```
#namedKeywordsPara.py
#使用命名关键字参数
def start_to_end(start, * ,end):
    #博物馆的月访客量保存到列表中
```

```
    data=[2200,3088,4203,4506,3986,3342,5767,6234,3124,6345,\
          1123,2234]
    sum = 0
    for month in range(start-1,end):
        sum+=data[month]
    avg=sum/(end-start+1)
    print(avg)

start_to_end(1,end=9)
```

(4) 可变参数。

```
#variablePara.py
#使用可变参数
def specifty(*args):
    #博物馆的月访客量保存到列表中
    data=[2200,3088,4203,4506,3986,3342,5767,6234,3124,6345,\
          1123,2234]

    sum = 0
    for item in args:
        sum+=data[item-1]
    avg=sum/len(args)
    print('{:.2f}'.format(avg))

specifty(9,8,7,6,5,4,3,2,1)
```

5.2.7　函数参数传递机制


观看视频

Python 中函数的参数传递方式，同变量一样，采用的是“值传递”方式。对于不可变对象，按照值传递，当参数的值在函数内发生变化时，函数外的值并不受影响。

例如，在下面的 passByValue1.py 示例程序中，在调用 swap()函数前，a、b 的值分别为 3 和 5，在 swap()函数内，a、b 的值进行了交换，分别为 5 和 3，在 swap()函数调用后，a、b 的值依然是 3 和 5。

```
#passByValue1.py
def swap(x,y):
    x,y = y,x
    print("swap 函数里面 a,b 的值分别是:",x,y)

a = 3
b = 5
print("swap 之前 a,b 的值分别是:",a,b)
swap(a,b)
print("swap 之后 a,b 的值分别是:",a,b)
```

运行结果如下：

```
swap 之前 a,b 的值分别是: 3 5
swap 函数里面 a,b 的值分别是: 5 3
swap 之后 a,b 的值分别是: 3 5
```

分析如下。在主程序中，第 6 行代码 a=3 的执行过程是这样的：先申请一段内存空间分配给一个整型对象来存储整型值 3，然后让变量 a 指向这个对象，实际上是指向这段内存空间。这里的变量 a 就是对象 3 的一个引用。第 7 行代码的执行过程类似，如图 5-4(a)所示；第 9 行代码调用 swap()函数，由于是值传递，因此，先获取实参 a 和 b 的 id()值，然后形参 x 和 y 分别指向对应地址的对象 3 和对象 5，如图 5-4(b)所示；在 swap()函数内部，在执行交换语句之前，变量 x 和变量 y 指向对象 3 和对象 5，执行交换语句后，变量 x 和 y 分别指向对象 5 和对象 3，如图 5-4(c)所示；第 9 行代码在执行 swap()函数返回后，变量 a 和 b 仍然指向对象 3 和对象 5，如图 5-4(d)所示。

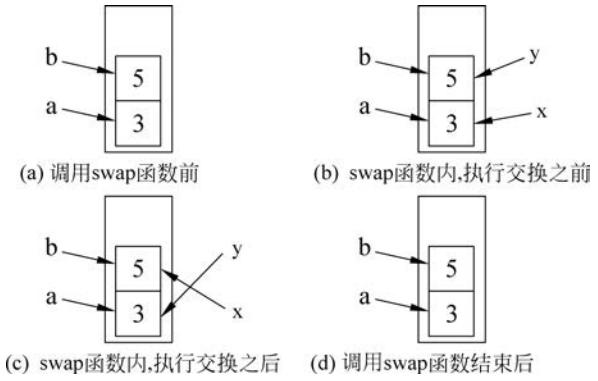

图 5-4　函数参数传递示意图

将 passByValue1.py 中添加辅助的输出语句，如 passByValue2.py 示例程序中的第 3、4 行代码、第 7、8 行代码、第 13、14 行代码、第 16、17 行代码，用来输出变量在调用之前和之后的内存位置，从运行结果可以清楚地看出以上的分析。

```
1  #passByValue2.py
2  def swap(x,y):
3      print(f'函数内交换前 x 的 id 是{id(x)}')
4      print(f'函数内交换前 y 的 id 是{id(y)}')
5
6      x,y = y,x
7      print(f'函数内交换后 x 的 id 是{id(x)}')
8      print(f'函数内交换后 y 的 id 是{id(y)}')
9      print("swap 函数里面 x,y 的值分别是:",x,y)
10
11 a = 3
12 b = 5
13 print(f'swap 之前 a 的 id 是{id(a)}')
14 print(f'swap 之前 b 的 id 是{id(b)}')
15 swap(a,b)
16 print(f'swap 之后 a 的 id 是{id(a)}')
17 print(f'swap 之后 b 的 id 是{id(b)}')
18 print("swap 之后 a,b 的值分别是:",a,b)
```

运行结果如下：

```
swap 之前 a 的 id 是 2371256609136
swap 之前 b 的 id 是 2371256609200
```

```
函数内交换前 x 的 id 是 2371256609136
函数内交换前 y 的 id 是 2371256609200
函数内交换后 x 的 id 是 2371256609200
函数内交换后 y 的 id 是 2371256609136
swap 函数里面 x,y 的值分别是: 5 3
swap 之后 a 的 id 是 2371256609136
swap 之后 b 的 id 是 2371256609200
swap 之后 a,b 的值分别是: 3 5
```

对于可变对象，参数传递依然是值传递。但由于在 Python 中，一切皆对象。对于可变数据类型(如列表、字典等)，这些对象在内存中建立时，实际上变量存储的是其地址，当传递参数时，是将地址传递过去。由于数据可变，在函数内改变可变对象，但其地址并没有改变，相当于在原地址上对数据进行了修改。因此，参数的值在函数内发生了变化后，在函数外也发生了变化。因此当需要在函数中修改某些数据时，可以把这些数封装为列表、字典等可变对象，并把这些可变对象作为函数参数传入函数，在函数中修改对象元素，这样函数结束后，这些数据就实现了修改。

例如，在下面的 passByRef.py 示例程序中，字典 mydict 作为 fun1()函数的参数传入函数，在 fun1()函数中修改键值，当 fun1()函数调用结束后，字典 mydict 的键值已经实现了修改。但是字典 mydict 作为 fun2()函数的参数传入函数，在 fun2()函数中修改字典，即让 d 重新指向对象{'e': 2,'f': 4}，当 fun2()函数调用结束后，字典 mydict 的键值没有发生变化。

```
1  #passByRef.py
2  def fun1(d):
3      d['a'],d['b'] = 2,4
4      print("fun1 函数里面 a 的键值,b 的键值分别是:",d['a'],d['b'])
5
6  def fun2(d):
7      d = {'e':2,'f':4}
8
9  mydict = {'a':3,'b':5}
10 fun1(mydict)
11 print("fun1 函数之后 a 的键值,b 的键值分别是:",mydict['a'],mydict['b'])
12
13 mydict = {'a':3,'b':5}
14 fun2(mydict)
15 print("fun2 函数之后 a 的键值,b 的键值分别是:",mydict['a'],mydict['b'])
```

运行结果如下：

```
fun1 函数里面 a 的键值,b 的键值分别是: 2 4
fun1 函数之后 a 的键值,b 的键值分别是: 2 4
fun2 函数之后 a 的键值,b 的键值分别是: 3 5
```

```
1  #
2  def swap(mydict):
3      print("swap 函数里面 mydict 的值",id(mydict))
4      mydict['a'],mydict['b'] = mydict['b'],mydict['a']
5      print("swap 函数里面 a,b 的值是",mydict['a'],mydict['b'])
6
```

```
mydict = {'a':3,'b':5}
print("swap 之前 mydict 的值",id(mydict))
swap(mydict)
print("swap 之后 a,b 的值",mydict['a'],mydict['b'])
print("swap 之后 mydict 的值",id(mydict))
```

运行结果如下：

```
swap 之前 mydict 的地址是: 2499177061504
swap 函数里面 mydict 的地址是: 2499177061504
swap 函数里面 a,b 的值是: 5 3
swap 之后 a,b 的值: 5 3
swap 之后 mydict 的值: 2499177061504
```


观看视频

5.3 lambda 表达式

有些情况下，在使用函数时不需要给函数定义一个名称，则该函数是“匿名函数”。Python 中使用 lambda 关键字创建匿名函数。语法格式如下。

```
lambda 参数列表:表达式
```

在 lambda 表达式中，参数列表与函数中的参数列表一样，但不需要用小括号括起来，冒号后面是 lambda 表达式，类似于函数体。其功能等价于下面的函数定义。

```
def <函数名>(参数列表):
    return 表达式
```

这两种方式的差异在于：使用 def 函数往往用来处理较大的任务，且需要命名函数；而使用 lambda 无须命名函数，能够自行返回结果，并且 lambda 在内部只能包含一行代码，因此代码更加简洁；lambda 表达式用完后立即释放空间，这对于不需要多次重复使用的函数提高了程序的性能。

例如，生成一个全部是偶数的列表，可以用下面的表达式：

```
List_1 = [x for x in range(1,20) if x%2 == 0]
```

某些情况下，给 lambda 函数一个标记名称，以便于在合适的时候调用并传递参数。例如，下面 lambdaExp1.py 示例程序中第 3 行定义 lambda 表达式，其中，mysum 是函数的标记名称，lambda 表达式中有 2 个参数，分别是 x 和 y，lambda 可以用 mysum(参数 1，参数 2)来调用，并将参数 1 传递给 x、参数 2 传递给 y，执行 x+y 的操作并将结果返回。第 3 行定义 lambda 后，第 4、5 行代码就可以调用。

```
# lambdaExp1.py
#求和 lambda 表达式
mysum = lambda x,y:x+y
print(mysum(1,2))
print(mysum(3,5))
```

上面示例中的 lambda 表达式的定义、调用和下面的函数的定义、调用是等价的。

```
1  def mysum(x,y):
2      return(x+y)
3  print(mysum(1,2))
4  print(mysum(3,5))
```

在 lambda 函数中还可以调用 Python 的内置函数。例如，在下面的 lambdaExp2.py 示例程序中的第 11 行代码，通过使用 lambda 表达式给定 sorted()函数排序的关键字，这样能够按照字典的键值排序，并输出键值对。第 5、7 行是按照字典的键进行排序输出的，而第 9 行是按照字典的值进行排序输出的。

示例：字典的多种排序方式。

```
1  #lambdaExp2.py
2  #将字典进行排序
3  portdict ={"http":80,"https":443,"ftp":21,"ssh":22}
4  #按照字典的键进行排序
5  print(sorted(portdict))
6  #按照字典的键进行排序
7  print(sorted(portdict.items()))
8  #按照字典的值进行排序
9  print(sorted(portdict.values()))
10 #按照字典的值进行排序
11 print(sorted(portdict.items(),key=lambda e:e[1]))
```

运行结果如下：

```
['ftp', 'http', 'https', 'ssh']
[('ftp', 21), ('http', 80), ('https', 443), ('ssh', 22)]
[21, 22, 80, 443]
[('ftp', 21), ('ssh', 22), ('http', 80), ('https', 443)]
```

(sorted()函数请参考 3.3.4 节)

分析：sorted()函数返回的是一个列表，而对于字典来说，如果排序的是 items()，则由于 items 返回的是元组(键值对)，因此返回列表的组成元素就是键值对形成的元组，其中键的索引为 0，值的索引为 1。而程序第 11 行代码的目的就是根据值排序键值对。

类似地，在下面的示例中，列表中的元素是元组，如果对列表按照颜色进行排序，则同样可以使用 lambda 表达式指定排序的关键字，如第 3 行代码所示。

```
1  #对列表中的元素进行排序,要求按照颜色排序
2  data = [('red', 2), ('blue', 1), ('red', 1), ('blue', 2)]
3  print(sorted(data, key=lambda x:x[0]))
```

运行结果如下：

```
[('blue', 1), ('blue', 2), ('red', 2), ('red', 1)]
```

从以上示例可以进一步看出，sorted()函数进行的排序是一种稳定排序。

示例：使用 lambda 表达式作为 takewhile()方法的条件判断。

```
1  #lambdaExp3.py
2  # takewhile 根据条件判断来截取出一个有限的序列
```

```
3  import itertools
4  a = itertools.count(5)
5  b = itertools.takewhile(lambda x: x <= 10, a)
6  for i in b:
7      print(i,end = ',')
```

运行结果如下：

```
5,6,7,8,9,10,
```

总结：对于逻辑简单的函数，使用 lambda 表达式代码更简洁；对于不需要重复调用的函数，使用 lambda 表达式后能够立即释放，因此会提高程序的空间效率。


观看视频

5.4 变量的作用域和命名空间

变量是有作用范围的，变量的作用范围称为作用域。根据作用域不同将变量分为局部变量和全局变量。局部变量是指定义在函数体内部的变量，作用域仅限于函数体内部。离开函数体就会无效。全局变量指在函数外定义的变量，它的作用域是整个程序，也就是所有的源文件。全局变量在程序执行全过程有效。

例如，在下面的 scopeExp1.py 示例程序中，变量 n 是在函数外定义的，是全局变量，在函数内可以使用。因此在函数 p1()内打印 n 的值，结果仍然是 10。

```
1  #scopeExp1.py
2  n = 10
3  def p1():
4      print(n)
5  p1()
```

需要注意，在函数 p1()内修改全局变量，对其进行操作，则出现错误，即不允许进行操作。例如，在下面的 scopeExp2.py 示例程序中，n 定义为全局变量，在函数内部 n 为局部变量，第 4 行修改局部变量引起越界错误。局部变量 n 赋值前被引用。其原因在于系统认为 n 是一个局部变量。但在本程序中，n 在函数中并没有被定义却开始使用。

```
1  #scopeExp2.py
2  n = 10                  #这里 n 是全局变量
3  def p1():
4      n = n+1             #这里 n 是局部变量
5      print(n)
6  p1()
```

运行结果如下：

```
UnboundLocalError: local variable 'n' referenced before assignment
```

错误含义为：局部变量 n 在没有被定义之前开始使用。

局部变量是在函数内定义的变量，仅仅在函数内起作用，仅在函数内部有效，当函数退出时变量将不再存在。例如，在下面的 scopeExp3.py 示例程序中，n 变量在函数 p2()内定义，是局部变量，其作用域是 p2()函数内部。可以在函数内正常使用，但是不能在函数外使

用。因此第 6 行对 n 的引用抛出 NameError 错误。

```
1  #scopeExp3.py
2  def p2():
3      n = 1                #局部变量
4      print(n)             #引用局部变量
5  p2()
6  print(n)
```

运行结果如下：

```
NameError: name 'n' is not defined
```

分析下面的程序输出结果。在下面的 scopeExp4.py 示例程序中，可以发现，尽管局部变量和全局变量名称相同，但仍然是两个不同的变量。

```
1  #scopeExp4.py
2  def p3(n):
3      y = n                #定义一个变量 y,并赋值为传递过来的 n.这里 y 是局部变量
4      if y<0:
5          y = -y
6      else:
7          y = y
8      print("函数内 y 的值是",y)
9
10 y = -3                   #这里 y 是全局变量
11 p3(y)
12 print("函数外 y 的值是",y)
```

运行结果如下：

```
函数内 y 的值是: 3
函数外 y 的值是: -3
```

但是变量在函数内部使用时，如果使用了保留字 global，则该变量是全局变量，语法形式如下：

```
global <全局变量>
```

分析下面的 scopeExp5.py 示例程序输出结果。定义 n 为全局变量，则在函数体内部的对 n 的引用都指向全局变量 n。同时定义 z 为全局变量，并且在第 5 行赋值为 9，在第 9 行代码中的变量 z 是全局变量，即输出为 9。

```
1  #scopeExp5.py
2  n = 2
3  def multiply(x, y):
4      global z             #z 是全局变量
5      z = x
6      return x*y*n         #使用全局变量 n,由于并没有尝试改变 n 的值,所以不会出错
7  s = multiply(9, 2)
8  print(s)
9  print(z)                 #这里 z 是全局变量
```

运行结果如下：

```
36
9
```

分析下面的 scopeExp6.py 示例程序输出结果。定义 n 为全局变量，并且在函数体内部定义 n 为 global，则在第 5 行代码中修改了全局变量的值为 3，其后对 n 的引用都是指向全局变量 n，即第 9 行代码输出结果是 3。

```
1  #scopeExp6.py
2  n = 2                  #n是全局变量
3  def multiply(x, y):
4      global n
5      n = 3              #由于使用了global语句,所以修改n的值后,函数内外的n全部改变
6      return x*y*n       #这里使用的全局变量n
7  s = multiply(9, 2)
8  print(s)
9  print(n)               #n是全局变量
```

运行结果如下：

```
54
3
```

总结：Python 程序中，每个变量都有其存在的命名空间。解释器确定一个命名空间的顺序是：首先是在包含该变量的函数调用命名空间；接着在全局命名空间；最后是在 builtins 模块的命名空间（builtins 模块是 Python 解释器启动时自动导入的模块）。Python 程序中，无论是变量，还是函数名，或是类名，都遵循以上的命名空间的查找顺序。

使用 globals()和 locals()访问全局变量和局部变量。这两个命令将开发环境中的变量和系统的变量全部显示了出来。

```
1  #scopeExp7.py
2  def print_var():
3      n = 30
4      m = "hello"
5      #print(globals())
6      print('.............................')
7      print(locals())
8  print_var()
```

运行结果如下：

```
.............................
{'n': 30, 'm': 'hello'}
```

为了方便查看，可以将 print(globals())和 print(locals())分开执行。由于全局变量较多，这里只输出局部变量作为示例。可以发现，locals()函数打印局部变量是按照字典格式输出。

5.5 函数高级特性


观看视频

5.5.1 生成器函数

第 4 章介绍了生成器的概念，生成器是一种惰性计算。函数生成器的关键字是 yield。

斐波那契数列是指一个数列中，第一项是1、第二项是1，从第三项开始，每一项都是前面两项的和。用函数来形成斐波那契数列，如fibNoYield.py示例程序所示。curr变量表示当前要计算出的元素项，pre表示它前面的那个元素项。

示例：斐波那契数列，不使用yield的情况。

```
1  #fibNoYield.py
2  def fib1(num):
3      n, pre, curr = 0, 0, 1
4      while n < num:
5          print(curr,end='\t')
6          pre, curr = curr, pre + curr          #序列解包，继续生成新元素
7          n = n + 1
8  fib1(6)
```

运行结果如下：

```
1  1  2  3  5  8
```

斐波那契数列的算法其实质是定义了一种计算规则，因此可以使用函数生成器的方式，如下面fibYield1.py示例程序中的第5行代码，定义yield curr。生成器并不是一次性执行完毕，而是进行持续的调用沟通。第一次迭代中，fib2()函数会执行，从开始到yield关键字的第5行，然后返回yield后的值作为第一次迭代的返回值。接着，每次执行fib2()函数都会继续执行在函数内部定义的循环的下一次，再返回那个值，直到没有可以返回的值结束。生成器中的元素可以通过for循环取出。在示例中使用了两种方式，如第10、11行代码所示，其效果是相同的。

示例：使用yield生成num大小的斐波那契数列。

```
1  #fibYield1.py
2  def fib2(num):
3      n, pre, curr = 0, 0, 1
4      while n < num:
5          yield curr
6          pre, curr = curr, pre + curr
7          n = n + 1
8  g = fib2(6)
9  for i in range(3):
10     print(g.__next__(),end='\t')
11     print(next(g),end='\t')
```

运行结果如下：

```
1  1  2  3  5  8
```

示例：使用yield生成无穷大的斐波那契数列。

```
1  #fibYield2.py
2  def fib2():
3      n, a, b = 0, 0, 1
4      while True:
5          yield b          #需要时再产生一个新元素
```

```
        a, b = b, a + b
        n = n + 1

g = fib2()
for i in range(10):
    print(g.__next__(),end='\t')
```

运行结果如下：

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

使用 yield 的执行流程：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，并发送数据，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”数据时才恢复执行。下面通过几个输出语句帮助理解执行流程。

同前面介绍的斐波那契数列例子类似，杨辉三角也存在着一定的规律，因此可以使用函数生成器定义，然后输出所需大小的杨辉三角。

示例：用函数生成器打印杨辉三角。

```
#yanghuiYield.py
def gen_row():
    row=[1]
    while True:
        yield row
        row=[x+y for x,y in zip([0]+row,row+[0])]
g= gen_row()
for i in range(6):
    print(g.__next__())
```

运行结果如下：

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
```


观看视频

5.5.2 高阶函数

高阶函数接收一个函数和一系列的数值作为参数，这个参数函数应用到数值中的每一项，最终返回一个结果集合或单一数值。高阶函数中包括两个计算，对每一项数值的转换任务和最终结果的累积计算。高阶函数实现的是将这两个逻辑进行了分离。本节介绍三个高阶函数，包括映射 map、化简 reduce 和过滤 filter。

1. map(func, * iterables) - -> map object

参数：接收两个参数，一个是函数 func，一个是 iterable 可迭代的对象。

作用：函数作用于序列中的每个元素。

返回值：一个 iterator 迭代器，map 对象。注意：map 返回的 iterator 是惰性序列。

示例：

```
1  #mapExp1.py
2  def fun(x):
3      return pow(x,2)
4  L = map(fun,[1,2,3,4,5])
5  print(type(L))
6  print(next(L))
7  print(next(L))
```

运行结果如下：

```
<class 'map'>
1
4
```

上面的 mapExp1.py 示例程序中，可以不使用 map，用 for 循环也可以实现上述功能。但是可以发现使用 map 更为简洁，因为 map 已经进行了抽象。

```
1  #mapExp2.py
2  def fun(x):
3      return pow(x,2)
4  L = []
5  for n in [1, 2, 3, 4, 5]:
6      L.append(fun(n))
7  print(L)
```

运行结果如下：

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

将 map 和 list 函数结合到一个表达式中进行简化，fun()函数功能简单同时还可以使用 lambda 表达式进一步简化。

```
1  #mapExp3.py
2  #第一种简化:
3  def fun(x):
4      return pow(x,2)
5  L1 = list(map(fun,[1,2,3,4,5]))
6  print(L1)
7
8  #进一步简化
9  L2 = list(map(lambda x:x**2,[1,2,3,4,5]))
10 print(L2)
```

运行结果如下：

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

2. reduce(func,iterable[,initializer])

参数：接收两个参数，一个是函数 func，一个是 iterable 可迭代对象。

作用：对序列中的第 1～2 个元素进行操作，得到的结果再与第 3 个元素用 func 函数

运算，以此类推，最后得到一个结果。

使用 reduce()函数需要导入 functools 模块。如果函数 func 的功能比较简单，还可以用 lambda 表达式代替。

示例：对列表元素的累加求和。

```
1  #reduceExp1.py
2  from functools import reduce
3  #((((1+2)+3)+4)+5)
4  reduce(lambda x, y: x+y, [1,2,3,4,5])
```

运行结果如下：

```
15
```

示例：对列表元素的加法、乘法和拼接运算。

```
1  #reduceExp2.py
2  from functools import reduce
3  from operator import mul,add,concat
4  print(reduce(mul, [1, 2, 3]))
5  print(reduce(add, [1, 2, 3, 4]))
6  print(reduce(concat, ['A', 'BB', 'C']))
```

运行结果如下：

```
6
10
ABBC
```

3. filter(func or None,iterable) --> filter object

参数：两个参数，函数参数 func，参数 iterable 可迭代对象。

作用：函数依次作用于每个元素，然后根据返回值是 True 还是 False 决定保留还是丢弃该元素；如果没有第一个函数参数，则返回 iterable 可迭代对象。

返回值：filter 对象。

filter 函数是惰性计算。即当“索取”元素时，才会计算输出。如果函数 func()功能简单，也可以使用 lambda 表达式代替。

示例：计算列表中的奇数。

```
1  #filterExp.py
2  f = filter(lambda x: x%2==1, [1, 2, 4, 5, 6, 9, 10, 15])
3  #print(next(f))
4  #print(next(f))
5  #print(next(f))
6  for i in range(3):
7      print(f.__next__())
```

运行结果如下：

```
1
5
9
```

5.5.3 偏函数

partial(func, * args, ** keywords)

参数：三个参数，函数参数 func、可变参数 args、关键字参数 keywords。

作用：func 函数接收参数 args 和参数 keywords 产生一个新的函数。


观看视频

内置函数 int(x,d)功能是将 d 进制的数字 x 转换为十进制数。通过 partial 函数可以定义不同进制的转换。

```
1  #partialExp1.py
2  from functools import partial
3  #二进制数
4  intNew2 = partial(int, base=2)
5  print(intNew2('100'))
6  #八进制数
7  intNew8 = partial(int, base=8)
8  print(intNew8('100'))
```

运行结果如下：

```
4
64
```

```
1  #partialExp2.py
2  from functools import partial
3
4  def log(message, subsystem):
5      print('%s: %s' % (subsystem, message))
6
7  server_log = partial(log, subsystem='server') #定义 partial 函数功能
8  server_log('Unable to open socket')
```

运行结果如下：

```
server: Unable to open socket
```

```
1  #partialExp3.py
2  from functools import partial
3  def sum( * args):
4      s = 0
5      for n in args:
6          s = s + n
7      return s
8  sum_add_5 = partial(sum,5)                    #定义 partial 函数功能
9  print(sum_add_5(1,2,3))
```

运行结果如下：

```
11
```

5.5.4 修饰器(装饰器)


观看视频

修饰器是一种以函数为参数，为该函数添加额外功能，并返回被修饰过的函数。因此，

可以使用修饰器(Decorator)为现有的代码添加功能。这种使用一部分程序在编译时改变剩余部分程序的技术，被称为元编程(Metaprogramming)。为了理解修饰器，必须明确一个概念，即 Python 中一切都是对象，包括函数对象。函数也可以作为另一个函数的返回值或参数。

一个基本的修饰器接收一个函数，然后为其添加新功能，最后返回这个函数。把@和修饰器的名称放在需要被修饰的函数 f 上方，这样对函数 f 起到了修饰作用。

下面的 decoratorExp1.py 示例程序中定义了一个简单的修饰器 make_pretty。可以看到，调用 ordinary 函数时，已经增加了新的功能。

```
#decoratorExp1.py
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner

#ordinary=make_pretty(ordinary)
@make_pretty
def ordinary():
    print("I am ordinary")

ordinary()
```

运行结果如下：

```
I got decorated
I am ordinary
```

具体过程分析：首先，装饰器 make_pretty()函数接收一个参数 func，其实就是接收一个方法名，make_pretty 内部又定义一个函数 inner()，在 inner()函数中调用打印语句，接着调用传入的参数 func，同时 make_pretty 的返回值为内部函数 inner()，它其实就是闭包函数。

在 ordinary 上增加@ make_pretty，当 Python 解释器执行到这条语句时，会去调用 make_pretty()函数，同时将被装饰的函数名作为参数传入(此时为 ordinary)，在执行 make_pretty()函数时直接把 inner()函数返回，同时把它赋值给 ordinary()，此时的 ordinary()指向 make_pretty.inner()函数地址。相当于 ordinary = make_pretty (ordinary)。接下来，在调用 ordinary()时，其实调用的是 make_pretty.inner()函数，那么此时就会先执行 print()函数，然后再调用原来的 ordinary()，该处的 ordinary 就是通过装饰传入的参数 ordinary，这样下来，就完成了对 ordinary()函数的装饰。

一个装饰器可以对多个函数进行装饰。例如，在下面的 decoratorExp2.py 示例程序中，定义装饰器 w1，分别对函数 f1()和 f2()进行了装饰。

```
#decoratorExp2.py
def w1(func):
    def inner():
        print('.....验证权限.....')
```

```
        func()
    return inner
@w1
def f1():
    print('f1 called')

@w1
def f2():
    print('f2 called')
f1()
f2()
```

运行结果如下：

```
.....验证权限.....
f1 called
.....验证权限.....
f2 called
```

如果被修饰函数有参数，那么闭包函数必须有参数，且个数一致。例如，在下面的decoratorExp3.py示例程序中，hello()函数参数和闭包函数inner()函数参数一致。

```
#decoratorExp3.py
def w_say(fun):
    def inner(name1):
        print('inner called')
        fun(name1)
    return inner

@w_say
def hello(name):
    print('hello ' + name)

hello('zhang')
```

运行结果如下：

```
inner called
hello zhang
```

为了适应对不同函数的装饰，装饰器函数中的闭包函数参数可以设置为可变参数和关键字参数。例如，在下面的decoratorExp4.py示例程序中，被修饰函数add1()和add2()定义了不同数量的参数。

```
#decoratorExp4.py
def w_add(func):
    #args包含可变参数元组,kwargs包含关键字参数的字典
    def inner(*args, **kwargs):
        print('add inner called')
        func(*args, **kwargs)
    return inner

```

```
 9  @w_add
10  def add1(a, b):
11      print('%d + %d = %d' % (a, b, a + b))
12
13  @w_add
14  def add2(a, b, c):
15      print('%d + %d + %d = %d' % (a, b, c, a + b + c))

16
17  add1(2, 4)
18  add2(2, 4, 6)
```

运行结果如下：

```
add inner called
2 + 4 = 6
add inner called
2 + 4 + 6 = 12
```

5.6 模块化编程


观看视频

5.6.1 内置模块

Python本身就内置了很多常用的模块，Python解释器安装完毕后，就可以使用这些模块。前面已经多次用到了内置模块，本节进行总结。

1. 导入整个模块

语法格式如下：

```
import 模块名1[as 别名1],模块2,...
```

使用方法：模块名.成员。

示例：导入time内置模块。

```
1  #importExp1.py
2  import time
3  #返回当前时间的时间戳 timestamp
4  #定义为从格林尼治时间 1970 年 01 月 01 日 00 时 00 分 00 秒起至现在的总秒数
5  print(time.time())
6  print(time.asctime())
7  print(time.ctime())
```

运行结果如下：

```
1621420346.564489
Wed May 19 18:32:26 2021
Wed May 19 18:32:26 2021
```

可以为模块定义别名，使用模块别名方便记忆和使用。

语法格式如下：

```
import 模块名 as 别名
```

示例：为模块定义别名。

```
1  #importExp2.py
2  import itertools as it
3  ns = it.repeat('q',5)
4  # ns = itertools.repteat('q',5)
5  for n in ns:
6      print(n,end = ' ')
```

运行结果如下：

```
q q q q q
```

当为模块定义了别名后，则不能再使用原来的模块名。例如，在上面的 importExp2.py 示例程序中，如果是第 4 行的调用方式，则输出如下结果：

```
AttributeError: module 'itertools' has no attribute 'repteat'
```

2. 导入模块中的某些成员

语法格式如下：

```
from 模块名 import 成员 1,成员 2,...
```

使用方法：直接调用模块的成员即可。

示例：导入模块的某些成员。

```
1  #importExp3.py
2  from hashlib import md5,sha512
3  md5('中国'.encode(encoding='UTF-8'))
4  sha512('中国'.encode(encoding='UTF-8')).hexdigest()
```

运行结果如下：

```
'6a169e7d5b7526651086d0d37d6e7686c7e75ff7039d063ad100aefab1057a4c1db1f1e5d088c9585db1d7531
a461ab3f4490cc63809c08cc074574b3fff759a'
```

3. 导入多个模块，多个模块之间用逗号隔开

示例：导入多个模块。

```
1  #importExp4.py
2  import sys,os
3  print(sys.version)
4  #平台上的路径分隔符
5  print(os.sep)
```

运行结果如下：

```
3.8.3 (default, Jul 2 2020, 17:30:36) [MSC v.1916 64 bit (AMD64)]
\
```

要查看模块内容，可以使用 dir()函数查看，也可以使用模块本身提供的_all_变量进行查看。

示例：使用列表推导式列出没有下画线的方法。

```
import math
#dir(math)
[e for e in dir(math) if not e.startswith('_')]
```

5.6.2 安装第三方模块

可以通过网址 https://pypi.org/查找第三方模块，并通过包管理工具 pip 安装第三方模块，语法格式如下：

```
pip 模块名称
```

当使用到很多个第三方模块时，需要注意版本的兼容性。Anaconda 是一个基于 Python 的数据处理和科学计算平台，它已经内置了许多非常有用的第三方库，非常简单、易用，并且在安装新的模块时会自动解决版本的兼容性和依赖性问题。


观看视频

5.6.3 自定义模块

Python 语言编程时，更多时候会引用自定义的模块。例如，下面代码，命名为 module1.py，并将其保存在 C:\Users\me 目录下面。

```
# module1.py
'''
这是一个测试模块，模块内容包括：
一个输出语句
my_name：一个字符串变量
sayhello：一个简单函数
'''
print('这是我的第一个模块')
my_name = '张瑞霞'
def sayhello():
    print('hello',my_name)
```

文件名 module1 就是模块名，当需要该模块时，直接导入，然后调用其相应的方法时，出现了错误提示信息。例如：

```
>>> import module1
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'module1'
```

以上错误的原因是 module1 模块没有在 Python 解释器的搜索路径中。Python 解释器通过搜索路径来定位模块。通过 sys.path 查看当前的模块搜索路径。

```
>>>>>> import sys
>>> sys.path
['', 'D:\\Programs\\Python\\Python38\\python38.zip', 'D:\\Programs\\Python\\Python38\\
DLLs', 'D:\\Programs\\Python\\Python38\\lib', 'D:\\Programs\\Python\\Python38', 'D:\\
Programs\\Python\\Python38\\lib\\site-packages']
```

将自定义的 module1 模块保存在搜索路径中，这样能够正确导入和使用模块。也可以将 module1 模块所在目录添加到 path 路径中。

```
>>> sys.path.append('C:/Users/me')
>>> sys.path
['', 'D:\\Programs\\Python\\Python38\\python38.zip', 'D:\\Programs\\Python\\Python38\\
DLLs', 'D:\\Programs\\Python\\Python38\\lib', 'D:\\Programs\\Python\\Python38', 'D:\\
Programs\\Python\\Python38\\lib\\site-packages', 'C:/Users/me'
>>> import module1
这是我的第一个模块
```

除了以上两种方式外，还可以在系统的环境变量中增加一项 PYTHONPATH，将模块路径添加到这个环境变量中，这样更方便使用。右击“我的计算机”→“属性”，则出现图 5-5 所示的界面。单击“高级系统设置”，出现图 5-6 所示的界面。单击“环境变量”，出现图 5-7 所示的界面。单击“编辑”按钮，出现图 5-8 所示的界面，添加变量名为 PYTHONPATH，变量值为 C:\Users\me。

图 5-5　计算机属性

图 5-6　高级系统属性

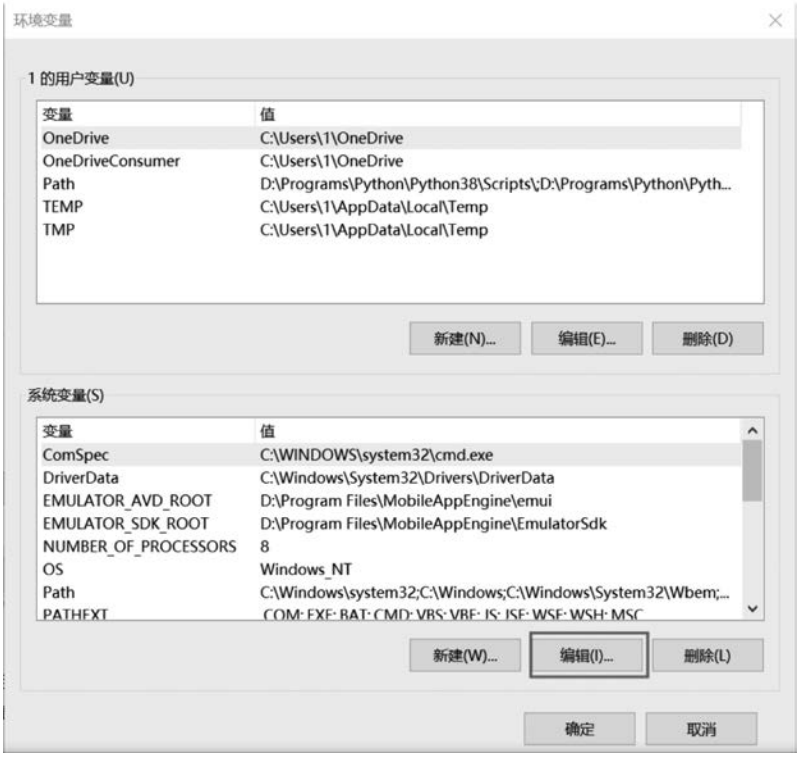

图 5-7　编辑环境变量

图 5-8　添加环境变量

5.6.4　模块导入顺序

一个程序需要导入多种模块时，需要按照下面的先后顺序导入：

（1）导入内置模块；

（2）导入第三方模块；

（3）导入自定义模块。

当 Python 解释器导入模块时，执行以下操作：

（1）查找模块对应的文件；

（2）运行模块中的代码，创建模块中定义的对象，包括各种对象，如字符串、整型、函数、模块或类等；

（3）创建对象的命名空间。

三种模块的导入顺序对应着它们的执行顺序，也意味着命名空间的优先级。导入自定

义模块时，其命名空间就是该模块的名称。没有导入 module1 模块之前，通过 dir()函数查看当前的命名空间的名称，module1 没有在命名空间列表中。

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']
```

当导入 module1 模块后，再次查看当前的命名空间的名称，module1 和 sayhello 在当前的命名空间列表中。

```
>>> import module1
这是我的第一个模块
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__',
'module1', 'sayhello']
```

如果只导入模块 module1 的 sayhello()方法，则当前的命名空间没有变量 my_name 命名空间。这是因为变量 my_name 命名空间只在 module1 模块中，而不是在 sayhello()方法中。

```
>>> from module1 import sayhello
这是我的第一个模块
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__',
'sayhello']
```

5.7　PyInstaller 打包


观看视频

当创建独立 Python 脚本(包含该应用的依赖包)后，可以将 Python 源文件打包，生成可直接运行的程序。打包后的程序不依赖于操作系统、解释器和相应的包，可以分发到不同操作系统平台上，如 Windows、Linux 或 macOS X 平台上运行。PyInstaller 模块是跨平台的第三方打包模块，它既可以在 Windows 平台上使用，也可以在 Linux 和 macOS X 平台上运行。在不同的平台上使用 PyInstaller 工具的方法是一样的，它们支持的选项也是一样的。使用该模块之前首先需要安装该模块。

```
C:\Users\me > pip install pyinstaller
```

安装完成后，使用十分简单。可以在安装目录下运行帮助命令，查看命令选项如下。

```
C:\Users\me > pyinstaller -h
C:\Users\me > pyinstaller --help
-F,--onefile          Create a one-file bundled executable.
--distpath DIR        Where to put the bundled app (default: .\dist)
(省略了其他命令)
```

例如，module1.py 文件存在 C:\Users\me 目录下，现在要将 module1.py 脚本进行打包，并将应用放置默认位置目录下。默认目录是 C:\Users\me\dist，则执行以下命令。

```
C:\Users\me > pyinstaller -F .\module1.py
```

执行后在 dist 目录中就出现了 module1.exe 文件，这时执行它即可。

如果要自定义存放位置，则添加--distpath 指出其存放位置，例如，要将应用放置 C:\test

目录下，则执行如下命令。

```
C:\Users\me > pyinstaller --distpath C:\test  -F .\module1.py
```

执行后 module1.exe 文件出现在 C:\test 目录下。相对于原有的 Python 脚本文件，打包后的 exe 文件相对比较大，这是因为在脚本文件中会导入需要的库，特别是当代码导入整个库时。因此，写代码时最好用到库的什么方法写就导入什么方法，而不是直接导入整个库，以减少占用的内存空间。


观看视频

5.8 安全专题

5.8.1 摘要算法的雪崩效应

在密码学中，雪崩效应(Avalanche Effect)指当输入发生最微小的改变，例如，反转一个二进制位(0 变为 1 或 1 变为 0)时，会导致输出的每个二进制位有 50%的概率发生反转。雪崩效应是分组密码和加密散列函数的一种理想属性。本节编写函数验证散列函数 SHA256 存在的雪崩效应。要求输入为两个不同字符串的十六进制摘要值，输出为反转的 bit 数。

定义函数 avalanche(digest1,digest2)，接收两个摘要值作为参数，输出反转的 bit 位数量。使用 hashlib 模块中 SHA256()计算不同字符串的摘要值，并作为 avalanche()函数的输入。由于输入的是十六进制的摘要值，而雪崩效应比较的是 bit 位的差异性，因此，需要将十六进制转换为二进制，定义函数 hex_to_bin 实现进制直接的转换。由于最终需要比较二进制位串中不同的位数，因此定义 cmpcount(str1,str2)函数实现此功能。在下面的 hash_avalanche.py 示例程序中，定义两个字符串第一部分相同，都为"First"，第二部分不同，分别是字符串"Second"和"Recond"，即比较的字符串分别为"FirstSecond"和"FirstRecond"，字母 S 和字母 R 的 ASCII 正好相差 1 个二进制 bit，但是两个字符串的 SHA256 摘要值相差 130 个 bit 位，超过了一半的反转位，从而验证了雪崩效应。

```
1  #hash_avalanche.py
2
3  from hashlib import sha256
4
5  def hex_to_bin(string):
6      return "{0:0128d}".format(int(bin(int("0x"+string,16))[2:]))
7
8  def cmpcount(str1,str2):
9      cmpcount_num = 0
10     for i,v in enumerate(str1):
12         if v!=str2[i]:
13             cmpcount_num=cmpcount_num+1
14     return cmpcount_num
15
16 def avalanche(text1,text2):
17     bindigest1 = hex_to_bin(test1)
18     bindigest2 = hex_to_bin(test2)
19     print(bindigest1)
```

```
    print(bindigest2)
    count = str(cmpcount(bindigest1, bindigest2))
    return count

if __name__ == "__main__":
    hash_object1 = sha256()
    hash_object1.update(b'FirstSecond')
    digest1 = hash_object1.hexdigest()
    hash_object2 = sha256()
    hash_object2.update(b'FirstRecond')
    digest2 = hash_object2.hexdigest()
    print("相差的 bit 位个数是:",avalanche(digest1,digest2))
```

运行结果如下：

```
11001010110101001100010101100010001111101111110000001010111001100111110110111110100000
10011101001110011111111111111000001100001001100101010010001110111110011011110101000111 0
10000100010001100010010000100100000011111000111011101111000100110000000011001110011 1
11000111001011100011110000111100110110111100010011110001010010110001111100100110001000
11001010010111001001001011100001001001110110011000110010010100011110100000011000010001
11110000010000001101101101101000001100000101010101011111011000100010010011100110010 10
相差的 bit 位个数是: 130
```

5.8.2　AES 算法的雪崩效应

AES 算法同样存在雪崩效应，本节使用密码学库 PyCryptodome 验证 AES 分组密码算法的雪崩效应。PyCryptodome 是 Python 的第三方库。网址为 https://pypi.org/project/pycryptodome/。该包实现的是对密码学原语的低层封装包，包名为 Crypto。使用前需要安装：pip install pycryptodome。

在下面的 AES_avalanche.py 示例程序中，两个明文字符串"zhang123" 和"zhang323" 只差一个二进制位，第一次运行结果中，128 bit 密文的输出中相差了 70 个二进制位，第二次运行结果中 128 bit 密文的输出中相差了 64 个二进制位。多次运行相差位数会不同，这是因为在代码中，加密的密钥是通过 get_random_bytes 接口产生的随机数，而不是固定的 key，即在第一次和第二次运行时，使用了不同的对称密钥。本示例程序相对于上一节的示例程序使用更为灵活，可以根据用户的输入，输出反转的 bit 位数量。但是这要求用户在输入时，需要明确知道两个输入字符串确实只存在一个 bit 位的差异。

```
#AES_avalanche.py

from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from Crypto.Random import get_random_bytes
from binascii import b2a_hex, a2b_hex

def hex_to_bin(string):
    return \ "{0:0128d}".format(int(bin(int("0x"+string,16))[2:]))

def encrypt(key, text):
    cryptor = AES.new(key, AES.MODE_ECB)
```

```
    text = bytes(text.encode('UTF-8'))
    #通过接口自动填充
    ciphertext = cryptor.encrypt(pad(text,16))
    entext = b2a_hex(ciphertext).decode("UTF-8")
    return entext

def cmpcount(str1,str2):
    cmpcount_num = 0
    for i,v in enumerate(str1):
        if v!=str2[i]:
            cmpcount_num=cmpcount_num+1
    return cmpcount_num

def avalanche(text1,text2):
    #密钥通过随机数接口产生
    key = get_random_bytes(16)
    enc1 = encrypt(key,text1)
    enc2 = encrypt(key,text2)
    binstr1 = hex_to_bin(enc1)
    binstr2 = hex_to_bin(enc2)
    print(binstr1)
    print(binstr2)
    count = str(cmpcount(binstr1, binstr2))
    print("相差的 bit 位个数是:",count)

if __name__ == "__main__":
    str1, str2 = input().split()
    avalanche(str1,str2)
```

第一次运行结果如下：

```
zhang123 zhang323
11001101100011000101111101001111101111010111011110100101000111001001111000000011011000
10000001010100100011110010100100010000101 1
01111100011010110110000100001101000110010110010100100011111000110000011110111110000100
11110001001111101110000111111010111001000 0
```

相差的 bit 位个数是：70

第二次运行结果如下：

```
zhang123 zhang323
10110011010001000110111111010011101000010001100011000101010111110110100001001110001111 0
11001101100010111110110111000010100101000 1
01001101110000000101111000011100111101111010110001001011000010001111011000001110001110111
11001000111110011111001111100000101100000 11
```

相差的 bit 位个数是：64

习题

1. 定义一个函数，判断输入的字符串是不是回文字符串，是回文字符串，输出 1，否则输出 0。例如，输入字符串"reviver"，输出 1；输入字符串"sender"，则输出 0。

2. 定义一个函数，判断输入的正整数是不是回文素数。

3. 已知某医院门诊的月访问量，如表 5-2 所示。

表 5-2　某医院门诊的月访问量

月　份	1	2	3	4	5	6	7	8	9	10	11	12
月访问量/人	560	689	452	567	345	231	267	523	432	325	562	359

请根据要求设计函数。

(1) 使用函数计算 start～end 月的月平均访问量，并计算 2～7 月的月平均访问量。

(2) 使用默认参数 end＝12 设计函数，并计算 9～12 月的月平均访问量。

(3) 定义函数，实现对不同季度的统计，要求使用关键字参数。

4. Josephus 问题，即约瑟夫问题，又称约瑟夫环：设有 n 人围坐在一个圆桌周围，现从第 s 人开始报数，数到第 k 的人出列，然后从出列的下一个人重新开始报数，数到第 k 的人又出列，……，如此反复直到所有的人全部出列为止。对于任意给定的 n、s 和 k，求出 n 人的出列序列。Josephus 问题举例：例如，n＝9、s＝1、k＝5，则出列人的顺序为 5、1、7、4、3、6、9、2、8。

5. 编写函数，计算并输出 15 位精度的 Π 值，要求采用下面两种方式：

(1) 莱布尼茨公式计算 Π 值。

(2) Bailey-Borwein-Plouffe 公式计算 Π 值。

6. 编写一个能实现双色球选号的程序。双色球选号由 7 个数字组成 y，其中有 6 个红球，其号码的取值范围为[1,33]，1 个蓝球的取值范围为[1,16]，要求 6 个红球从小到大排列，蓝球在最后输出。其输出格式为“09 12 16 20 30 33 | 03”。(注意，如双色球号码为 3，则必须输出 03)。例如，输入为 7 注，则输出格式为：

09 12 16 20 30 33 | 03
01 07 08 09 18 31 | 16
05 08 21 26 28 31 | 05
01 03 06 22 25 33 | 02
02 09 16 20 27 28 | 13
15 19 24 26 28 32 | 05
02 05 07 16 24 32 | 09

7. 编写函数，从定量的角度比较不同插入方法时间效率。append()和 insert()是列表的两种插入方法，二者的时间效率不同。从定性角度分析，append()方法是在列表后面追加元素，这样不需要移动原有元素，时间复杂度为 O(1)，时间效率高；而 insert()方法与插入的位置 i 有关(这里假设 i≥0)。如果 i 设置为 n，则在尾部插入，同 append()方法一样不需要移动元素；如果 i 小于 n，则需要移动 n－i－1 个元素，时间复杂度为 O(n)。

8. 编写杨辉三角函数，要求使用函数生成器，并说明使用函数生成器的作用。

9. 一个陷入迷宫的老鼠如何找到出口的问题，要求输出老鼠探索出的从入口到出口的路径。老鼠希望尽快地找到出口走出迷宫。如果它到达一个死胡同，将原路返回到上一个位置，尝试新的路径。在每个位置上老鼠可以向八个方向运动：从正东按照顺时针。无论离出口多远，它总是按照这样的顺序尝试，当到达一个死胡同之后，老鼠将进行“回溯”。例如，下面图 5-9 所示迷宫，入口是(1,1)，出口是(6,6)，则输出的路径为：

(6,6) (5,7) (4,6) (4,5) (3,4) (2,5) (2,4) (2,3) (1,2) (1,1)。

```
1 1 1 1 1 1 1 1 1
1 0 0 1 1 0 1 1 1
1 1 0 0 0 0 0 0 1
1 0 1 0 0 1 1 1 1
1 0 1 1 1 0 0 1 1
1 1 0 0 1 0 0 0 1
1 0 1 1 0 0 0 1 1
1 1 1 1 1 1 1 0 1
1 1 1 1 1 1 1 1 1
```

图 5-9 迷宫示意图

10. 编写一个函数，实现搜索单词的功能，要求能够在竖、横和斜线方向上实现搜索。

11. 给定一个字符串 s，计算这个字符串中有多少个回文子串。具有不同开始位置或结束位置的子串，即使是由相同的字符组成，也会被视作不同的子串。

示例 1：如果输入的字符串是 s="abc"，则输出三个回文子串，即"a"、"b"、"c"。

示例 2：如果输入的字符串是 s="aaa"，则输出 6 个回文子串，即"a"、"a"、"a"、"aa"、"aa"、"aaa"。

12. 将第 4 章的凯撒加、解密写成函数形式，函数包括三个参数，分别是加密模式 mode、密钥 key 和待加解密的信息 m。

13. 将第 4 章的仿射加、解密写成函数形式，函数包括四个参数，分别是加密模式 mode、密钥 key1 和 key2，以及待加解密的信息 m。

14. 编写函数验证 AES 算法的雪崩效应，要求固定加密的密钥信息，例如 key='keyskeyskeyskeys'，用户输入不同的明文，输出密文中翻转的比特位数。

15. Xtime 运算是 AES 算法的基本运算，请编写函数实现其功能，并将其写成 lambda 表达式。

16. 编写函数，实现欧几里得算法和扩展的欧几里得算法。

17. 孙子定理是中国古代求解一次同余方程组（见图 5-10）的方法，是数论中一个重要定理，又称中国余数定理。一元线性同余方程组问题最早可见于中国南北朝时期（公元 5 世纪）的数学著作《孙子算经》卷下第二十六题，叫作“物不知数”问题，原文如下：有物不知其数，三三数之剩二，五五数之剩三，七七数之剩二。问物几何？《孙子算经》中首次提到了同余方程组问题，以及以上具体问题的解法，因此在中文数学文献中也会将中国余数定理称为孙子定理。编写函数实现中国余数定理。

$$\begin{cases} a_1 (\mathrm{mod}\ m_1) \equiv x \\ a_2 (\mathrm{mod}\ m_2) \equiv x \\ \cdots \\ a_k (\mathrm{mod}\ m_k) \equiv x \end{cases}$$

图 5-10 同余方程组