第5章 函数及模块

函数是组织好的，可重复使用的，用来实现单一，或相关联功能的代码段。函数能提高应用的模块性和代码的重复利用率。可重用性是软件工程的重要理念，软件工程的目标之一就是要像生产硬件那样生产软件，实现零件化、组件化开发。Python 提供了许多内置函数，如 print()等。也可以自己创建函数，这叫作用户自定义函数。若干相关的函数集合写在一个 Python 文件中构成模块，若干模块文件保存在文件夹中构成包。本章介绍函数、模块和包的语法与用法。

5.1 函　　数

5.1.1 函数定义

利用以下语法形式定义一个函数：

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

函数代码块以 def 关键词开头，def 是单词 define 的缩写，这个单词的中文翻译是定义的意思，后接函数标识符名称和圆括号()。圆括号内可以用于定义参数，称为函数的形参。任何传入参数和自变量，称为实参，必须放在圆括号内。

函数的第一行语句可以选择性地使用文档字符串，用于存放函数说明。函数内容以冒号起始，并且缩进。

return [表达式] 结束函数，选择性地返回一个值给调用方。不带表达式的 return 相当于返回 None。在 Python 中，定义函数时不需要声明函数的返回值类型，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，一旦执行将直接结束函数的执行。

图 5.1 是计算斐波那契数列中小于参数 n 的值的函数。在定义该函数时，开头部分的注释不是必需的，但如果为函数的定义加上注释，可以为用户在调用该函数时提供友好的提示。一定要在函数调用之前定义函数，否则抛出异常。在面向对象程序设计的类中成员方法的定义和调用，不受该限制。

斐波那契数列(Fibonacci sequence)，又称黄金分割数列，因数学家莱昂纳多·斐波那契(Leonardoda Fibonacci)以兔子繁殖为例而引入，故又称“兔子数列”，其序列值如式(5.1)

递推。图 5.1 在函数体巧妙地利用循环代替递归函数实现斐波那契数列计算。

$$F(n)=\begin{cases}1, & n=1\\ 1, & n=2\\ F(n-1)+F(n-2), & n\geqslant 3\end{cases} \tag{5.1}$$

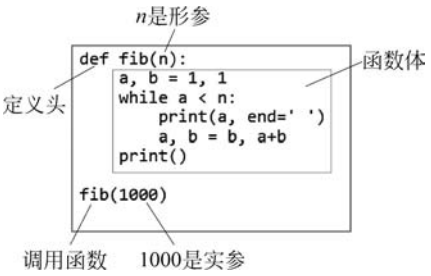

图 5.1　函数计算斐波那契数列中小于参数 n 的所有值

5.1.2　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，……当某个条件得到满足时就不再调用了，然后逐层返回，到该函数第一次调用的位置。程序 5.1 就是依式(5.1)的递归方式实现斐波那契数列，第 6 行是在 $n>1$ 时，不断地调用自身。

```
1  # 程序 5.1  依公式(5.1)递归实现斐波那契数列
2  def fib_recur(n):
3      assert n >= 0, "n > 0"
4      if n <= 1:
5          return n
6      return fib_recur(n - 1) + fib_recur(n - 2)
7  for i in range(1, 20):
8      print(fib_recur(i), end = ' ')
```

输出结果：

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

程序 5.2 使用函数递归调用实现序列求和。其中，第 6 行调用了自身，参数是 $n-1$，每次调用参数都递减，直到参数变成 1，第 6 行就变成了 $n+n-1+n-2+\cdots+1$，同样实现了序列求和。

```
1  # 程序 5.2  递归函数对序列求和
2  def my_sum(n):
3      if n == 1:
4          return n
5      else:
6          return n + my_sum(n - 1)
7  print(my_sum(10))
```

输出结果：

```
55
```

每次调用函数必须记住离开时的位置才能保证函数运行结束以后回到正确的位置，这个过程称为保存现场，这需要一定的栈空间。调用一个函数时会为该函数分配一个栈存放普通参数和函数内部局部变量的值，这个栈会在函数调用结束自动释放。在函数递归调用的过程中，一个函数执行尚未结束又调用自己，原来的栈还没有释放又分配了新的栈，这会占用大量的内存空间。所以递归调用不宜太深，否则可能导致栈空间不足而使程序崩溃。

5.1.3 函数参数

函数定义时圆括号内是使用逗号分隔开的形参列表(parameter)，函数可以有多个参数，也可以没有参数，但定义和调用时一对圆括号必须有，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(argument)，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。

1. 参数传递

参数的传递与其他高级语言类似，分为值传递和序列类型传递。当把一个基本数据类型的实参传递给函数的形参时，函数内部发生的对形参的改变不会影响实参。当实参是序列类型的数据结构时，函数内部对序列元素的改变会影响实参。这种传递方式类似 C 语言的指针参数传递或者 C++语言的引用传递，在 Java 语言中则是对象参数传递。

对于基本数据类型的变量在参数传递时，函数内部直接修改形参的值不会影响实参。程序 5.3 演示的是基本数据类型参数的传递方式。

```
1   #程序 5.3  基本数据类型参数传递
2   def addOne(a):
3       print('a 的初始地址:',id(a), 'a 的值为:', a)
4       a += 1
5       print('a 的新地址:',id(a), 'a 的新值为:', a)
6   v = 3
7   print('v 的初始地址:',id(v))          #v 与 a 的地址相同
8   addOne(v)
9   print('v 的值不变:',v)                #形参的值发生改变,实参不变
10  print('v 的地址不变:',id(v))
```

输出结果：

```
v 的初始地址: 140712111940048
a 的初始地址: 140712111940048 a 的值为: 3
a 的新地址: 140712111940080  a 的新值为: 4
v 的值不变: 3
v 的地址不变: 140712111940048
```

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，实参也得到了相应的修改。程序 5.4 演示的是函数序列结构参数的传递方式。

```
1  #程序 5.4  函数序列结构参数传递
2  def modify(d):                #修改字典元素值或为字典增加元素
3      d['age'] = 38
4  a = {'name':'Yang', 'age':41}
5  print('初始值:',a)
6  modify(a)
7  print('修改后:',a)
```

输出结果：

```
初始值: {'name': 'Yang', 'age': 41}
修改后: {'name': 'Yang', 'age': 38}
```

2. 位置参数

位置参数(positional argument)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。程序 5.5 中，第 5 行调用 position()函数时传递了两个实参，但是该函数有 3 个形参，因此在执行该语句时抛出异常。异常的提示是类型错误，position()函数缺少一个必需的位置参数 c。无论用哪种语言编写程序，都要耐心地阅读编译器给出的提示信息。这些信息为程序员快速精准地找到程序的错误位置和错误的原因提供了很多帮助，这也是程序员应具备的基本素质。

```
1  #程序 5.5  位置参数
2  def position(a, b, c):
3      print(a, b, c)
4  position(3,4,5)          #按位置传递参数
5  position(7,8)            #抛出异常
```

输出结果：

```
3 4 5
TypeError:position() missing 1 required positional argument: 'c'
```

3. 默认值参数

在定义函数时可以为形参设置默认值，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，函数会直接使用函数定义时设置的默认值，也可以通过显式赋值替换默认值。也就是说，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侧都不能再出现没有默认值的普通参数，否则会提示语法错误。一般来说，避免使用列表、字典、集合或者其他可变序列作为函数参数默认值。

带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
```

使用“函数名.__defaults__”可以随时查看函数所有默认参数的当前值，返回一个元组，其中的元素依次表示每个默认值参数的当前值。程序 5.6 演示的是默认值参数的函数。其中，第 2 行函数定义的第二个参数是默认值参数；第 4 行调用的时候给该默认值参数传递

了实参，因此函数执行就按照实参进行；第 5 行查看函数默认参数 time 的当前值，返回的是元组，因为实际使用过程中默认参数的个数可能不止 1 个。

```
1  #程序 5.6  默认值参数
2  def say(message, times = 1):
3      print((message + ' ') * times)
4  say('hello',3)
5  print(say.__defaults__)
```

输出结果：

```
hello hello hello
(1,)
```

4. 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。程序 5.7 演示的是关键参数函数，由于指定了关键字，所以在第 4 行调用时不需要按照函数定义时形参的顺序去传递。

```
1  #程序 5.7  关键参数
2  def key_para(a, b, c = 5):
3      print(a, b, c)
4  key_para(c = 8, a = 9, b = 0)
```

输出结果：

```
9 0 8
```

5. 可变长度参数

可变长度参数在定义函数时主要有两种形式：* parameter 和 ** parameter。前者用来接收任意多个实参并将其放在一个元组中；后者接收类似于关键参数一样显式赋值形式的多个实参并将其放在字典中。

程序 5.8 中，无论调用该函数时传递了多少个实参，一律放在元组中。第 3 行打印的 p 值就是一个元组。

```
1  #程序 5.8  可变长度参数
2  def opti_para( * p):
3      print(p)
4  opti_para(3,7,8)
```

输出结果：

```
(3, 7, 8)
```

程序 5.9 是第二种形式的可变长度参数，在调用该函数时自动将接收的参数转换为字典。

```
1  #程序 5.9  第二种形式的可变长度参数
2  def opti_para( ** p):
3      print(p)
4  opti_para(x = 3,y = 7,z = 8)
```

输出结果：

```
{'x': 3, 'y': 7, 'z': 8}
```

6. 传递参数是序列解包

序列解包是指实参是序列结构，使用“ * ”和“ ** ”两种形式传递时，Python 解释器会自动进行解包，把序列中的值分别传递给多个单变量形参。程序 5.10 对列表和字典进行了解包。

```
1  #程序 5.10  参数是序列解包
2  def jiebao(a,b,c):
3      print(a + b + c)
4  seq = [3,6,9]
5  jiebao( * seq)          #对列表进行解包
6  dic = {1:'a',2:'b',3:'c'}
7  jiebao( * dic)          #对字典进行解包
```

输出结果：

```
18
6
```

如果实参是字典，还可以使用第二种形式的序列解包，会把字典转换成类似于关键参数的形式进行参数传递。对于这种形式的序列解包，要求实参字典中的所有键都必须是函数的形参名称，参见程序 5.11。

```
1  #程序 5.11  字典参数的序列解包
2  def jiebao(a,b,c):
3      print(a + b + c)
4  dic = {'a':3,'b':6,'c':9}
5  jiebao( ** dic)              #对字典进行另一种解包
```

输出结果：

```
18
```

5.1.4 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。在函数外部和在函数内部定义的变量，其作用域不同。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。函数外部定义的变量为全局变量。不管是局部变量还是全局变量，其作用域都适合从定义的位置开始，在此之前无法访问。

在函数内部定义的局部变量只在该函数内部可见，当函数执行结束，局部变量自动删

除,不可以再使用。在函数内部使用 global 定义的全局变量,当函数结束以后,仍然存在并且可以访问。局部变量的引用比全局变量速度快,应优先考虑使用。程序 5.12 演示了变量作用域。

```
#程序 5.12  变量作用域
def zuoyongyu():
    global x                #声明或创建全局变量,必须在使用 x 之前执行
    x,y = 5,7
    print(x,y)
x = 10;                     #函数外部定义了全局变量
zuoyongyu()                 #函数调用修改了全局变量 x 的值
print(x)
```

输出结果:

```
5 7
5
```

如果局部变量与全局变量有相同的名字,那么该局部变量会在自己的作用域内暂时隐藏同名的全局变量。程序 5.13 第 3 行函数内部的 x 变量就屏蔽了全局变量 x。

```
#程序 5.13  同名局部变量隐藏全部变量
def zuoyongyu():
    x = 5               #创建局部变量,并自动隐藏了同名全局变量
    print(x)
x = 10                  #创建全局变量
zuoyongyu()             #调用结束全局变量不受影响
print(x)
```

输出结果:

```
5
10
```

5.1.5 lambda 表达式

lambda 表达式可以用来声明匿名函数,也就是没有函数名称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。

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被看作函数的返回值,不允许包含复合语句,但在表达式中可以调用其他函数,参见程序 5.14。

```
#程序 5.14  lambda 表达式
f = lambda x,y,z:x + y + z          #可以给 lambda 表达式起一个名字 f
print(f(1,3,4))                     #lambda 表达式当作函数使用
g = lambda x,y = 2,z = 4:x + y + z  #支持默认值参数
print(g(10))
print(g(10,z = 20,y = 30))          #调用时使用关键参数
```

输出结果：

```
8
16
60
```

lambda 表达式可以很方便地定义一些小函数，但是如果仅仅需要一个简单的运算，那么尽量使用标准库 operator 中提供的函数，避免自定义 lambda 表达式，operator 中的函数执行效率更高一些。程序 5.15 中第 4 行 operator.inv()函数不是像之前的 linalg.inv()求逆矩阵，容易让人联想到此处的 inv()是求倒数，这里实际是取相反数。

```
1  #程序 5.15  operator 中的函数
2  import operator
3  a=[1,2,3,4]
4  print(list(map(operator.inv,a)))      #利用 operator 取相反数
5  print(list(map(lambda x:-x,a)))       #利用 lambda 表达式取相反数
```

输出结果：

```
[-2, -3, -4, -5]
[-1, -2, -3, -4]
```

5.1.6 生成器函数

包含 yield 语句的函数可以用来创建生成器对象，这样的函数也称为生成器函数。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，参见程序 5.16。生成器函数具有惰性求值的特点，适合大数据处理。

```
1  #程序 5.16  生成器函数(1)
2  def f():
3      yield from 'abcdefg'   # 使用 yield 表达式创建生成器
4  x = f()                    #创建生成器对象
5  print(next(x))
6  print(next(x))
7  for item in x:             # 输出 x 中的剩余元素
8      print(item, end='')
```

输出结果：

```
A
b
c d e f g
```

程序 5.17 第 5 行可以执行一次返回 a 之后就停下了，只有在第 9 行不断调用__next__()方法时才继续执行；第 7 行也与平时函数调用不一样，a 不是函数 f()的返回值，而是该函数

创建的生成器对象，所以 a 才会有一些特殊的对象方法，如__next__()。

```
1  #程序 5.17  生成器函数(2)
2  def f():
3      a, b = 1, 1                    #序列解包,同时为多个元素赋值
4      while True:
5          yield a                    #暂停执行,需要时再产生一个新元素
6          a, b = b, a+b              #序列解包,继续生成新元素
7  a = f()                            #创建生成器对象
8  for i in range(10):                #斐波那契数列中前 10 个元素
9      print(a.__next__(), end = ' ')
```

输出结果：

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

5.1.7 关于__main__

Python 不同于 C/C++，程序执行并不需要主程序，如 main()，而是依文件自上而下地执行。Python 中的__main__代表的是程序内置名称，并不是普通编程语言的主函数。

很多 Python 程序中都有__name__，它属于 Python 中的内置类属性，存在于一个 Python 程序中，代表对应程序名称。程序 5.18 顺序执行，但是函数在没有调用时不执行，所以第 4 行没有执行打印字符串'子函数'。

```
1  #程序 5.18  __main__函数
2  print('第一条语句')
3  def add(a,b):
4      print('子函数')
5  if __name__ == '__main__':
6      print('主程序')
```

输出结果：

```
第一条语句
主程序
```

从输出结果看程序是依次执行，子函数并没有执行。当程序作为执行脚本时，内置名称就是__main__，所以 if 语句块的内部被执行。但是，当此程序作为模块被其他文件导入时，内置名称就变成模块名字，if 语句块就不会被执行。

5.2 模块、包、库

5.2.1 模块

Python 语言提供了强大的模块支持，不仅标准库中包含了大量的模块（称为标准模块），还有大量的第三方模块，开发者自己也可以开发自定义模块。通过这些强大的模块可以极大地提高开发者的开发效率。

模块，英文为 Module，可以用一句话总结：模块就是 Python 程序，任何 Python 程序都可以作为模块，包括在前面章节中写的所有 Python 程序，都可以作为模块。

可以把模块比作一盒积木，通过它可以拼出多种主题的玩具，这与前面介绍的函数不同，一个函数仅相当于一块积木，而一个模块(.py 文件)中可以包含多个函数，也就是很多积木。

将 Python 代码写到一个文件中，随着程序功能复杂度的增加，程序篇幅会不断变大。为了便于维护，通常会将其分为多个文件(模块)。这样，不仅可以提高代码的可维护性，还可以提高代码的可重用性。代码的可重用性体现在，当编写好一个模块后，只要编程过程中需要用到该模块中的某个功能(由变量、函数、类实现)，无须做重复性的编写工作，直接在程序中导入该模块即可使用该功能。可以将模块理解为是对代码更高级的封装，即把能够实现某一特定功能的代码编写在同一个 .py 文件中，并将其作为一个独立的模块，这样既可以方便其他程序或脚本导入并使用，同时还能有效避免函数名和变量名发生冲突。

程序 5.19 是在某一目录下创建一个名为 hello.py 的文件，该文件中包含了 1 个函数，这时 hello 就是一个模块。

```
1   #程序 5.19   hello.py 文件
2   def say ():
3       print("Hello,World!")
```

在同一目录下，再创建一个 say.py 文件，其包含的代码如程序 5.20 所示，第 2 行导入 hello 模块；第 3 行就能调用该模块的函数 say()。

```
1   #程序 5.20   say.py 文件
2   import hello
3   hello.say()
```

运行 say.py 文件，输出结果：

```
Hello,World!
```

say.py 文件中使用了原本在 hello.py 文件中才有的 say()函数。相对于 say.py 文件，hello.py 就是一个自定义的模块，需要将 hello.py 模块导入 say.py 文件中，然后就可以直接在 say.py 文件中使用模块中的资源。

当调用模块中的 say()函数时，使用的语法格式为“模块名.函数”。这是因为，相对于 say.py 文件，hello.py 文件中的代码自成一个命名空间，因此在调用其他模块中的函数时，需要明确指明函数的出处，否则 Python 解释器将会报错。

使用 Python 进行编程时，有些功能没必要自己实现，可以借助 Python 语言现有的标准库或者其他人提供的第三方库。例如，余弦函数 cos()、绝对值函数 fabs() 等，它们位于 Python 标准库的 math 模块中，只需要将此模块导入当前程序，就可以直接使用。

在前面章节中，已经讲解过使用 import 导入模块的语法。实际上，import 还有更多详细的用法，主要介绍以下两种。

第一种导入形式，import 模块名 1 [as 别名 1]，模块名 2 [as 别名 2]，…，使用这种语法格式的 import 语句，会导入指定模块中的所有成员(包括变量、函数、类等)。不仅如此，当需要使用模块中的成员时，需用该模块名(或别名)作为前缀。

第二种导入形式，from 模块名 import 成员名 1 [as 别名 1]，成员名 2 [as 别名 2]，…，使用这种语法格式的 import 语句，只会导入模块中指定的成员，而不是全部成员。同时，当程序中使用该成员时，无须附加任何前缀，直接使用成员名(或别名)即可。

注意，用[]括起来的部分是可选项，可以使用，也可以省略。

其中，第二种 import 语句也可以导入指定模块中的所有成员，即使用 form 模块名 import *，但此方式不推荐使用。

程序 5.21 使用导入整个模块的最简单语法来导入指定模块，输出的结果都是当前文件的路径和文件名。

```
1  #程序 5.21  模块导入方式
2  import sys                    # 导入 sys 整个模块
3  print(sys.argv[0])            # 使用 sys 模块名作为前缀来访问模块中的成员
4  from sys import argv          # 导入 sys 模块的 argv 成员
5  print(argv[0])                # 使用导入成员的语法，直接使用成员名访问
```

5.2.2 包

实际开发中，一个大型的项目往往需要使用成百上千的 Python 模块。如果将这些模块堆放在一起，势必不好管理。使用模块可以有效避免变量名或函数名重名引发的冲突，但如果模块名重复怎么办呢？因此，Python 语言提出了包(Package)的概念。

包即文件夹，只不过在该文件夹下必须存在一个名为__init__.py 的文件。

每个包的目录下都必须建立一个 __init__.py 的模块，可以是一个空模块，也可以写一些初始化代码，其作用就是告诉 Python 要将该目录当成包来处理。

注意，__init__.py 不同于其他模块文件，此模块的模块名不是__init__，而是它所在的包名。例如，在 settings 包中的 __init__.py 文件，其模块名就是 settings。

包是一个包含多个模块的文件夹，它的本质依然是模块，因此包中也可以包含包。例如，在安装了 NumPy 包之后可以在 Lib\site－packages 安装目录下找到名为 numpy 的文件夹，它就是安装的 NumPy 包，它所包含的内容如图 5.2 所示。在 NumPy 包(模块)中，有必须包含的__init__.py 文件，还有 matlib.py 等模块源文件以及 random 等子包。

手动创建一个包，只需进行以下两步操作。

(1) 新建一个文件夹，文件夹的名称就是新建包的包名。

(2) 在该文件夹中，创建一个__init__.py 文件(前后各有两个下画线“_”)，该文件中可以不编写任何代码。当然，也可以编写一些 Python 初始化代码。当有其他程序文件导入包时，会自动执行该文件中的代码。

示例：创建一个包，该包的名称为 my_package。

第一步，创建一个文件夹，命名为 my_package；

第二步，在该文件夹中添加一个__init__.py 文件，在该文件中编写程序 5.22，也可以什么都不写，只是一个空文件。

```
1  #程序 5.22  __init__文件
2  #也可以不写内容，直接是一个空文件
3  print('测试包信息')
```

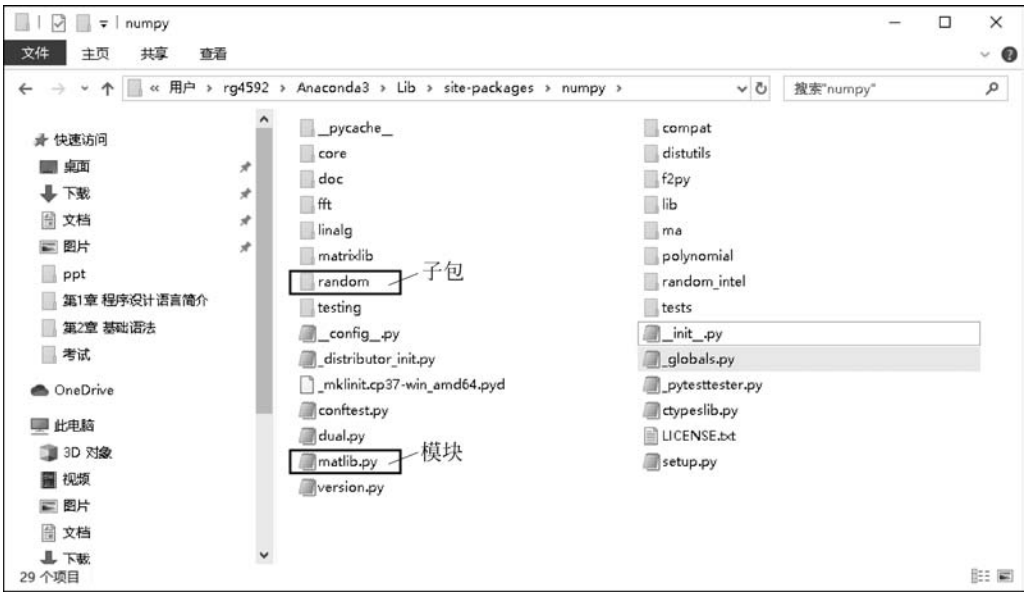

图 5.2　NumPy 包中的子包和模块

__init__.py 文件中，包含了两部分信息，分别是此包的注释信息和一条 print 输出语句。由此，这样就成功创建好了一个 Python 包。

在与 my_package 同级的文件夹中创建测试程序 5.23，第 2 行在导入包的时候，首先执行__init__文件中的内容，就像实例化一个对象时，会首先执行对象的构造函数，不需要显式调用。

```
1   #程序 5.23　导入自定义包
2   import my_package
```

输出结果：

```
测试包信息
```

在 PyCharm 集成环境中新建一个包更加简单，可以通过依次执行菜单 File→new→Python Package 选项实现，PyCharm 会自动添加一个空的__init__文件。

5.2.3　库

与模块和包相比，库是一个更大的概念。例如，在 Python 标准库中的每个库都有好多个包，而每个包中都有若干模块。Python 标准库非常庞大，所提供的组件涉及范围十分广泛，包含多个内置模块(以 C 语言编写)，Python 程序员必须依靠它们来实现系统级功能，如文件 I/O。此外，还有大量以 Python 编写的模块，以及第三方库(也称扩展库)，提供了日常编程中许多问题的标准解决方案，以下是一些常用库。

1. Scipy

Scipy 是一个用于数学、科学、工程领域的常用软件包，可以处理插值、积分、优化、图像处理、常微分方程数值解的求解、信号处理等问题。它用于有效计算 NumPy 矩阵，使

NumPy 和 Scipy 协同工作，高效解决问题。

2. Pillow

PIL(Python Imaging Library)已经是 Python 平台事实上的图像处理标准库了。PIL 功能非常强大，但 API 却非常简单易用。

由于 PIL 仅支持到 Python 2.7，加上年久失修，于是一群志愿者在 PIL 的基础上创建了兼容的版本——Pillow。它支持新版本 Python 3.x，又加入了许多新特性，因此，我们可以直接安装使用 Pillow。

3. OpenCV

OpenCV 是一个基于 BSD 许可(开源)发行的跨平台计算机视觉库，可以运行在 Linux、Windows、Android 和 Mac OS 操作系统上。它轻量级而且高效，由一系列 C 函数和少量 C++类构成，同时提供了 Python、Ruby、MATLAB 等语言的接口，实现了图像处理和计算机视觉方面的很多通用算法。OpenCV 用 C++语言编写，它的主要接口也是 C++语言，但是依然保留了大量的 C 语言接口。

4. Matplotlib

Matplotlib 是一个 Python 的 2D 绘图库，在前面章节涉及绘图的案例中已经接触过，它以各种硬拷贝格式和跨平台的交互式环境生成出版质量级别的图形。Matplotlib 是 Python 2D 绘图领域使用最广泛的库。它能让使用者很轻松地将数据图形化，并且提供多样化的输出格式。

5. NumPy

NumPy 是高性能科学计算和数据分析的基础包，是 Python 的一种开源的数值计算扩展。NumPy(Numeric Python)提供了许多高级的数值编程工具，如矩阵数据类型、矢量处理，以及精密的运算库，专为进行严格的数字处理而产生。

6. pandas

APython Data Analysis Library 或 pandas 是基于 NumPy 的一种工具，该工具是为了解决数据分析任务而创建的。pandas 纳入了大量的库和一些标准的数据模型，提供了高效地操作大型数据集所需的工具。pandas 提供了大量能快速便捷地处理数据的函数和方法。

7. Flask

Flask 是一个基于 Python 开发并且依赖 jinja 2 模板和 Werkzeug WSGI 服务的一个微型框架，对于 Werkzeug 本质是 Socket 服务端，其用于接收 http 请求并对请求进行预处理，然后触发 Flask 框架，开发人员基于 Flask 框架提供的功能对请求进行相应的处理，并返回给用户，如果要返回用户复杂的内容，需要借助 jinja 2 模板来实现对模板的处理，即将模板和数据进行渲染，将渲染后的字符串返回用户浏览器。

8. Keras

高阶神经网络开发库可运行在 TensorFlow 或 Theano 上，是基于 Python 的深度学习库。Keras 是一个高层神经网络 API，由纯 Python 编写而成，并基于 TensorFlow、Theano 以及 CNTK 后端。Keras 支持快速实验，能够把你的 idea 迅速转换为结果，支持循环神经网络(CNN)和递归神经网络(RNN)，或二者的结合，无缝切换 CPU 和 GPU。

9. Sklearn

Sklearn 是 Python 的重要机器学习库，其中封装了大量的机器学习算法，如分类、回归、降维以及聚类；还包含了监督学习、非监督学习和数据变换三大模块。Sklearn 拥有完善的文档，具有上手容易的优势；并且它内置了大量的数据集，节省了获取和整理数据集的时间。因而，Sklearn 成为了广泛应用的重要的机器学习库。本教材中的很多案例都会用到这个库。

需要指出的是，Python 库和包的区别不像包和模块的区别那么明显，有时也会直接把包称为库。

5.3 异常处理

5.3.1 异常

异常是指程序运行时引发的错误，引发错误的原因有很多，如除零、下标越界、文件不存在、网络异常、类型错误、名字错误、字典键错误、磁盘空间不足等。如果这些错误得不到正确的处理将会导致程序终止运行。合理地使用异常处理结构可以使得程序更加健壮，具有更强的容错性，不会因为用户不小心的错误输入或其他运行时的原因而造成程序终止；也可以使用异常处理结构为用户提供更加友好的提示。

严格来说，语法错误和逻辑错误不属于异常，但有些语法错误会导致异常，例如由于大小写拼写错误而试图访问不存在的对象，或者试图访问不存在的文件等。当 Python 检测到一个错误时，解释器就会指出当前程序已经无法再继续执行下去，这时候就出现了异常。

为了避免因为程序运行可能出现的异常而退出，可以使用捕获异常的方式获取这个异常，再通过其他的逻辑代码让程序继续运行，这种根据异常做出的逻辑处理称为异常处理。

1. Python 语法错误

语法错误，也就是解析代码时出现的错误。当代码不符合 Python 语法规则时，Python 解释器在解析时就会报出 SyntaxError 语法错误，同时还会明确指出最早探测到的错误的语句。例如，语句 print "Hello，World!"在 Python 3x 解释器上运行，会报如下错误：

SyntaxError：Missing parentheses in call to 'print'

语法错误大多是由于开发者疏忽导致的，属于真正意义上的错误，是解释器无法容忍的。因此，只有将程序中的所有语法错误全部纠正，程序才能执行。

2. Python 运行时错误

运行时错误，即程序在语法上是正确的，但在运行时发生了错误。例如，语句 a＝1/0，该语句试图用 1 除以 0，并赋值给 a。因为 0 作除数是没有意义的，所以运行后会产生如下错误：ZeroDivisionError：division by zero。

以上运行输出结果中，会同时给出程序中出错的位置和出错的类型。在 Python 中，把这种运行时产生错误的情况叫作异常(Exception)。这种异常情况很多，常见的异常类型及含义如表 5.1 所示。

表 5.1 常见的异常类型及含义

异常类型	含　义
AssertionError	当 assert 关键字后的条件为假时，程序运行会停止并抛出 AssertionError 异常
AttributeError	当试图访问的对象属性不存在时抛出的异常
IndexError	索引超出序列范围会引发此异常
KeyError	字典中查找一个不存在的关键字时引发此异常
NameError	尝试访问一个未声明的变量时，引发此异常
TypeError	不同类型数据之间的无效操作
ZeroDivisionError	除法运算中除数为 0 引发此异常

表中的异常类型不需要记住，只需简单了解即可。当一个程序发生异常时，代表该程序在执行时出现了非正常的情况，无法再执行下去。默认情况下，程序是要终止的。如果要避免程序退出，可以使用捕获异常的方式获取这个异常的名称，再通过其他的逻辑代码让程序继续运行，大大提高了程序的健壮性和人机交互的友好性。

5.3.2 异常处理

在 Python 中，有 3 种常见的异常处理结构，具体如下所述。

1. try...except...

其中，try 子句中的代码块包含可能会引发异常的语句，而 except 子句则用来捕捉相应的异常。

如果 try 子句中的代码引发异常并被 except 子句捕捉，就执行 except 子句的代码块；如果 try 中的代码块没有出现异常就继续往下执行异常处理结构后面的代码；如果出现异常但没有被 except 捕获，继续往外层抛出；如果所有层都没有捕获并处理该异常，程序崩溃并将该异常呈现给最终用户。

该结构语法如下：

```
try:
    可能产生异常的代码块
except [ (Error1, Error2, ... ) [as e] ]:
    处理异常的代码块 1
except [ (Error3, Error4, ... ) [as e] ]:
    处理异常的代码块 2
except [Exception]:
    处理其他异常
```

该格式中，[] 括起来的部分可以使用，也可以省略，其各部分的含义如下。

- （Error1，Error2，...）、（Error3，Error4，...）：其中，Error1、Error2、Error3 和 Error4 都是具体的异常类型。显然，一个 except 块可以同时处理多种异常。
- [as e]：作为可选参数，表示给异常类型起一个别名 e，这样做的好处是方便在 except 块中调用异常类型。
- [Exception]：作为可选参数，可以代指程序可能发生的所有异常情况，其通常用在最后一个 except 块。

2. try...except...else...

如果 try 中的代码抛出了异常，并且被 except 语句捕捉，则执行相应的异常处理代码，这种情况下就不会执行 else 中的代码；如果 try 中的代码没有引发异常，则执行 else 块的代码。

该结构的语法如下：

```
try:
    #可能会引发异常的代码
except Exception [ as e]:
    #用来处理异常的代码
else:
    #如果 try 子句中的代码没有引发异常，就继续执行这里的代码
```

3. try...except...finally...

在这种结构中，无论 try 中的代码是否发生异常，也不管抛出的异常有没有被 except 语句捕获，finally 子句中的代码总是会得到执行。

该结构语法为：

```
try:
    #可能会引发异常的代码
except Exception [ as e]:
    #处理异常的代码
finally:
    #无论 try 子句中的代码是否引发异常，都会执行这里的代码
```

从异常处理的语法格式可以看出，try 块有且仅有一个，但 except 代码块可以有多个，且每个 except 块都可以同时处理多种异常。当程序发生不同的意外情况时，会对应特定的异常类型，Python 解释器会根据该异常类型选择对应的 except 块来处理该异常，参见程序 5.24。

```
1   #程序 5.24  异常处理
2   try:
3       a = int(input("输入被除数:"))
4       b = int(input("输入除数:"))
5       c = a / b
6       print("您输入的两个数相除的结果是:", c )
7   except (ValueError, ArithmeticError):
8       print("程序发生了数字格式异常、算术异常之一")
9   except :
10      print("未知异常")
11  print("程序继续运行")
```

输出结果：

```
输入被除数: 4
输入除数: 0
程序发生了数字格式异常、算术异常之一
程序继续运行
```

程序 5.24 中，第 7 行使用了 ValueError 和 ArithmeticError 来指定所捕获的异常类型，这表明该 except 块可以同时捕获这两种类型的异常。除此之外的异常类型被最后一个 except 块成功捕获，一旦出现异常程序不会中断执行，而是由 except 中的代码块来处理，因此程序才可以继续执行，有了“程序继续运行”的输出结果。

程序 5.25 演示的是带 else 和 finally 块的异常处理，如果没有异常则执行 else 代码块，不管有没有异常都会执行 finally 代码块。

```
1   #程序 5.25  finally异常处理
2   try:
3       a = int(input("请输入 a 的值:"))
4       print(20/a)
5   except:
6       print("发生异常!")
7   else:
8       print("执行 else 块中的代码")
9   finally :
10      print("执行 finally 块中的代码")
```

输出结果：

```
请输入 a 的值:2
10.0
执行 else 块中的代码
执行 finally 块中的代码
```

5.4 PyCharm 单步跟踪

有时，需要查看 Python 函数内部变量是如何变化的，也就是函数内部的执行过程，目的是为了确定函数是按照程序员预定的目标执行，这就要用到 PyCharm 开发环境提供的单步跟踪和调试功能。

图 5.3 中的程序是为了使用函数 switch()实现两个变量的交换，但运行之后在图中下方的输出窗口并没有按照预定的目标输出想要的结果。这时，可通过单步调试查看一下函数内部的执行情况。

在 PyCharm 中单步跟踪程序，首先要为程序设置断点，所谓断点就是程序中的某一行，此时程序的上一行已经运行结束，即将执行该行，以便查看当前变量临时的结果和状态，所以如果需要查看某一行的执行结果，就在该行左侧（如图 5.3 中箭头所指的位置）单击，PyCharm 会将该行设置为断点。

接下来，依次单击菜单 Run→Debug 选项，或者按 Shift＋F9 快捷键进入调试状态。此时，程序会自动运行到断点所在行就停止运行，在图 5.4 右下方的 Variables 窗口可以查看当前程序的变量临时结果。

图 5.4 箭头所指工具栏上的按钮 Step into 就是单步进入按钮。如果当前行是调用的某一个函数，单击该按钮后会进入函数的内部继续单步执行。该按钮左右两侧还有 Step over、Step out 和 Run to cursor 三个常用的按钮。

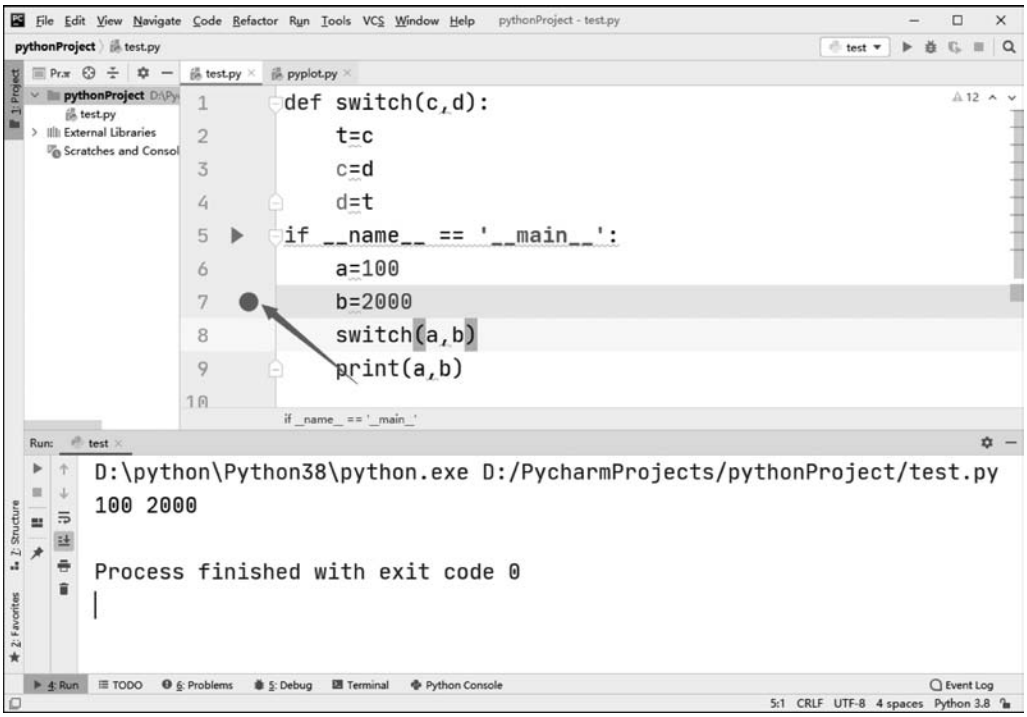

图 5.3　在 PyCharm 环境中为程序设置断点

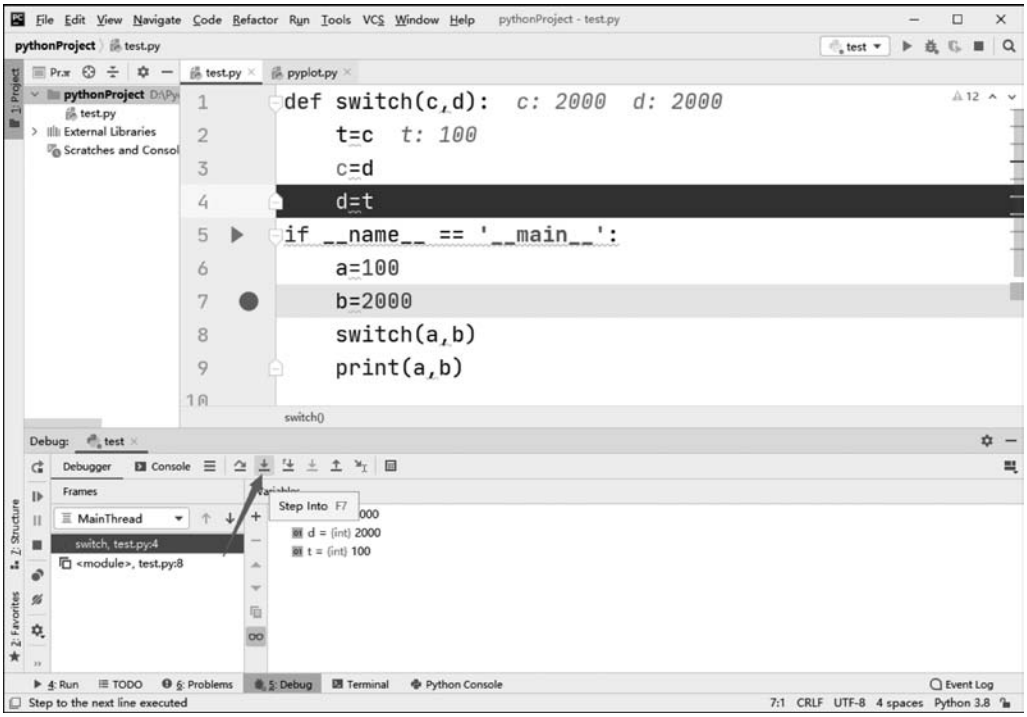

图 5.4　在 PyCharm 环境中单步跟踪调试

- Step over 按钮的功能是越过当前行。如果当前行是一个函数调用，单击该按钮则不会进入函数内部单步执行，而是一次性运行完成函数之后再转到下一行。
- Step out 按钮的功能和 Step into 是相反的。如果进入函数内部单步执行后，不想继续单步执行，则单击该按钮会一次性运行函数接下来的程序返回断点处。
- Run to cursor 按钮的功能是运行到光标所在的行。这几个按钮中使用频率最高的是 Step into 和 Step out 两个按钮，要善于利用这两个按钮辅助观察函数内部的运行状态。

图 5.4 中通过单步跟踪发现在函数内部的两形参确实已经交换了值，但是继续单步执行，回到断点处时，那些形参的临时变量都消失不见了。也就是说，形参的结果并没有影响实参。

单步跟踪程序可以窥探到函数的内部，尤其当函数不是自定义，而是来自第三方函数库，这时可以通过单步执行查看函数内部的执行过程理解函数功能；另一方面，这些函数库的代码都是经过优化而且高效的代码，学习这些代码的编写方法，可以提高编程水平。

视频 10

5.5 机器学习中的矩阵分析

5.5.1 正规方程计算线性模型参数

回顾 4.4 节，随机梯度下降算法恰好利用的循环结构迭代求解模型参数，一次循环处理一个样本。这里利用 Python 语言包 numpy 提供的矩阵运算函数来求解同样的问题，即线性回归模型的参数计算。相对于梯度下降算法，这种方法称为正规方程方法。在计算之前，先将样本组织成设计矩阵 $\boldsymbol{X}$。$\boldsymbol{X}$ 的每一行是一个样本向量，维度为 n 列，总共 m 个样本，$\boldsymbol{X}$ 就是 $m \times n$ 的矩阵。

这样公式(1.3)就重写为公式(5.2)。

$$J(\boldsymbol{\theta}) = \frac{1}{2}(\boldsymbol{X}\boldsymbol{\theta} - \vec{y})^{\mathrm{T}}(\boldsymbol{X}\boldsymbol{\theta} - \vec{y}) \tag{5.2}$$

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \frac{1}{2}(\boldsymbol{X}\boldsymbol{\theta} - \vec{y})^{\mathrm{T}}(\boldsymbol{X}\boldsymbol{\theta} - \vec{y}) \\ &= \frac{1}{2}\nabla_{\boldsymbol{\theta}}(\boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{\theta}^{\mathrm{T}}X^{\mathrm{T}}\vec{y} - \vec{y}^{\mathrm{T}}\boldsymbol{X}\boldsymbol{\theta} + \vec{y}^{\mathrm{T}}\vec{y})\end{aligned} \tag{5.3}$$

公式(5.3)是损失函数的梯度推导，式中$\boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{X}^{\mathrm{T}}\vec{y}$ 和 $\vec{y}^{\mathrm{T}}\boldsymbol{X}\boldsymbol{\theta}$ 相等，$\vec{y}^{\mathrm{T}}\vec{y}$ 与$\boldsymbol{\theta}$ 无关，令导数为零，$\boldsymbol{\theta}^{\mathrm{T}}$和$\boldsymbol{\theta}$ 只是行向量和列向量的区别，对$\boldsymbol{\theta}^{\mathrm{T}}$求导数，因此有：

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{2}\nabla_{\boldsymbol{\theta}}(\theta^{\mathrm{T}}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\boldsymbol{\theta} - 2\boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{X}^{\mathrm{T}}\vec{y}) = \boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{X}^{\mathrm{T}}\vec{y} = 0 \tag{5.4}$$

$$\boldsymbol{\theta} = (\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}\vec{y} \tag{5.5}$$

进一步可推导出公式(5.5)，由矩阵分析可以指导，$(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}$ 其实就是 $\boldsymbol{X}$ 的广义逆，也就是方程 $\boldsymbol{X}\boldsymbol{\theta} = \vec{y}$ 的最小二乘解，利用该公式可以计算模型参数。程序 5.26 演示的是依公式(5.5)计算模型参数。第 6 行依然是生成训练样本 $y = 2 + 3x + \varepsilon$；第 9 行～第 11 行是正规方程求解参数；第 13 行绘制原始样本的散点图和预测值的曲线，如图 5.5 所示。计算输出的参数值与 4.4 节中的梯度下降算法得到的值基本相似。

```
#程序 5.26  正规方程求模型参数
import numpy as np
import matplotlib.pyplot as plt
m = 101
x = np.linspace(1,10,m)                          #生成 x 的值
y = 3 * x + 2 + np.random.randn(m)               #生成与 x 线性关系的 y
X = np.vstack((np.ones(m),x)).transpose()        #整理数据为设计矩阵的形式
theta = [0,0]                                    #参数初始化
xtx_ni = np.linalg.inv(X.transpose()@X)          #计算(X^T X)^-1
xtx_ni_xt = xtx_ni@X.transpose()                 #计算广义逆矩阵 (X^T X)^-1 X^T
theta = np.matmul(xtx_ni_xt,y)                   #函数计算矩阵相乘
print(theta)
plt.plot(x,y,'ro',x,theta[1] * x + theta[0],'g')
plt.show()
```

输出结果：

```
[2.07819626 2.96396209]
```

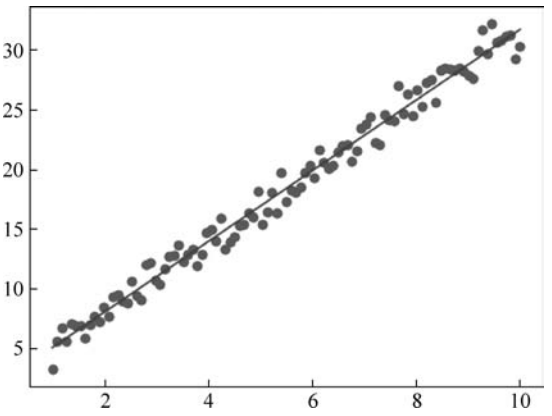

图 5.5　正规方程计算线性模型参数

5.5.2　矩阵奇异值分解

视频 11

奇异值分解在数据降维和图像压缩中有较多的应用，这里简单总结一下它的原理，并且演示一个图片压缩的例子。

1. 特征值分解

如果矩阵 $\boldsymbol{A}$ 是个 $m \times m$ 的实对称矩阵，即 $\boldsymbol{A}^{\mathrm{T}}=\boldsymbol{A}$，那么它就可以分解成如下形式：

$$\boldsymbol{A}=\boldsymbol{Q}\boldsymbol{\Sigma}\boldsymbol{Q}^{\mathrm{T}} \tag{5.6}$$

其中，$\boldsymbol{Q}$ 为标准正交矩阵，每一列为特征向量；$\boldsymbol{\Sigma}$ 为对角矩阵，对角线上的元素为特征向量对应的特征值。特征值的大小反映了矩阵在该特征值对应的特征向量的方向上能量分布，或者把特征值看作是该方向上的能量分布权重。这种特征值分解，对矩阵有着较高的要求，它要求被分解的矩阵 $\boldsymbol{A}$ 为实对称矩阵。现实中所遇到的问题一般不是实对称矩阵，这就要做一般性的推广，即矩阵奇异值分解。

2. 奇异值分解

矩阵 $\boldsymbol{A}$ 的维度为 $m \times n$，有如下分解形式：

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\mathrm{T}} \tag{5.7}$$

其中，$\boldsymbol{U}$ 和 $\boldsymbol{V}$ 是单位正交矩阵，即 $\boldsymbol{U}^{\mathrm{T}}\boldsymbol{U}=\boldsymbol{I}$、$\boldsymbol{V}^{\mathrm{T}}\boldsymbol{V}=\boldsymbol{I}$。$\boldsymbol{U}$ 称为左奇异矩阵，$\boldsymbol{V}$ 称为右奇异矩阵，$\boldsymbol{\Sigma}$ 仅在对角线上有值，这些值称为奇异值，其他元素均为 0。这些奇异值扮演着特征值分解中的特征值的角色，因此奇异值也是矩阵在奇异向量上的能量分布，如果取比较大的部分奇异值和它们对应的部分奇异向量来恢复矩阵，就能起到数据压缩的作用。

程序 5.27 演示了图像矩阵分解再恢复的过程，先在 PyCharm 中新建一个 Python 文件，命名为 huifu.py，作为一个模块文件。

```
#程序 5.27  huifu.py
import numpy as np
#参数从左到右依次为奇异值矩阵、左特征矩阵、右特征矩阵、奇异值的个数
def restore(sigma, u, v, K):
    m = len(u)                          #取 u 矩阵的行数
    n = len(v[0])                       #取 v 矩阵的列数
    a = np.zeros((m, n))                #准备一个跟恢复矩阵一样大小的零矩阵
    for k in range(K):
        uk = u[:, k].reshape(m, 1)      #每次循环取 u 矩阵的第 k 列
        vk = v[k].reshape(1, n)         #每次循环取 v 矩阵的第 k 行
        #一个列向量与一个行向量内积结果为一个矩阵，一共产生 k 个矩阵，再求和
        a += sigma[k] * np.dot(uk, vk)
    a[a < 0] = 0
    a[a > 255] = 255
    #将矩阵的元素四舍五入到整数，再转成图像数据类型
    return np.rint(a).astype('uint8')
```

程序 5.27 是一个包含一个恢复矩阵的函数 restore 的模块，在主文件中可以直接导入之后再调用，程序 5.28 为主文件，程序虽然长，但是分块理解也很好掌握。在主文件中，首先加载 sklearn 包中 datasets 子包的图像，该图像大小为 427×640×3 像素，最后的 3 对应 3 个颜色通道，即 R、G、B 三基色。然后，分别对每个颜色矩阵做奇异值分解，再计算前 50 个恢复矩阵的图像保存在当前目录下的 pic 目录中。

这里存在 3 个坐标系，即数学坐标系、矩阵行列坐标、NumPy 中 array 数组的坐标，它们的对应关系如图 5.6 所示。图中一幅彩色图像有 3 个颜色矩阵，矩阵的行对应数学坐标的 y 轴，但是在 array 数组中 axis=0 轴；矩阵的列对应数学坐标的 x 轴，array 数组的 axis=1 轴；数学坐标的 z 轴是三基色图像矩阵的叠加，array 数组中对应 axis=2 轴。在实际 Python 程序设计中，图像的索引都是按照矩阵的行、列、基色 3 个索引来指定图像中的位置。例如，索引[0,1,2]对应的图像矩阵第 1 行，第 2 列，蓝色，即蓝色矩阵的第 1 行第 2 列这个位置。

第 11 行字符串中的点代表的是当前目录，这和 DOS 系统的目录一致，若是两个点则代表上一级目录；第 25 行～第 27 行，分别调用模块计算恢复图像的每个颜色矩阵，矩阵的大小为 427×640 像素，当然可以直接显示每个颜色分量的图像，但是要还原彩色图像就必须将 3 个颜色图像矩阵堆叠起来；第 28 行是对 3 个颜色矩阵进行在第三轴向上堆叠，这样形成的矩阵为 427×640×3 像素，axis=2 指定堆叠的方向，axis=0 代表的是行方向，axis=1 代表的是列方向。

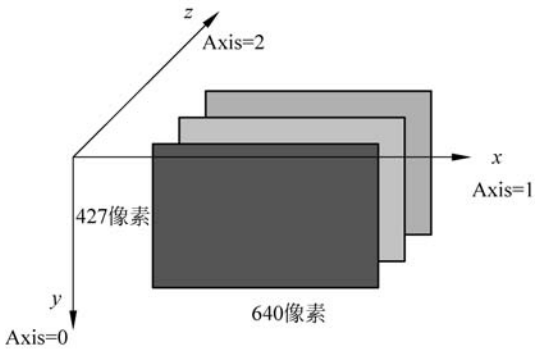

图 5.6　图像坐标系

```
#程序 5.28　主文件
import numpy as np
import os
from sklearn import datasets
import matplotlib.pyplot as plt
import matplotlib as mpl
from PIL import Image
import huifu
if __name__ == '__main__':
    A = datasets.load_sample_image('china.jpg')   #加载图像
    output_path = r'.\Pic'                        #保存恢复图像的目录
    if not os.path.exists(output_path):           #如果不存在就新建一个目录
        os.mkdir(output_path)
    a = np.array(A)                               #将图像矩阵转换为 array 结构
    print(a.shape)
    K = 50                                        #保留前 50 个奇异值和对应的奇异向量
    #3 个颜色 r,g,b 矩阵分别做奇异值分解
    u_r, sigma_r, v_r = np.linalg.svd(a[:, :, 0])#红色矩阵做奇异值分解
    u_g, sigma_g, v_g = np.linalg.svd(a[:, :, 1])#绿色矩阵做奇异值分解
    u_b, sigma_b, v_b = np.linalg.svd(a[:, :, 2])#蓝色矩阵做奇异值分解
    mpl.rcParams['font.sans-serif'] = [u'simHei']
    mpl.rcParams['axes.unicode_minus'] = False
    for k in range(1, K+1):                       #矩阵恢复
        print(k)
        R = huifu.restore(sigma_r, u_r, v_r, k)
        G = huifu.restore(sigma_g, u_g, v_g, k)
        B = huifu.restore(sigma_b, u_b, v_b, k)
        I = np.stack((R, G, B), axis=2)           #堆叠 3 个颜色矩阵,恢复彩色图像
        Image.fromarray(I).save('%s\\svd_%d.png' % (output_path, k))
        if k <= 12:                               #显示前 12 个恢复图像
            plt.subplot(3, 4, k)
            plt.imshow(I)
            plt.axis('off')
            plt.title(u'奇异值个数:%d' % k)
    plt.suptitle(u'SVD 与图像分解', fontsize=14)
    plt.tight_layout(0.3, rect=(0, 0, 1, 0.92))
    plt.subplots_adjust(top=0.9)
    plt.show()
```

主文件的恢复矩阵的计算，调用了之前保存的模块文件中的函数 restore()。恢复程序执行了 12 次，对于这样一段程序多次重复执行，写成一个函数是理所当然的。每次输入的奇异值矩阵、左奇异矩阵、右奇异矩阵相同，但奇异值的个数 k 不同，k 值越大，恢复的效果越好，需要保存的值也相应增加，也就是图像压缩比就降低。这是一对矛盾，想要无损恢复矩阵，那就保存原始文件，压缩比为 1；想要提高压缩比，减少保存数据，那就对图像质量有所降低。

图 5.7 是原始图像，图 5.8 是利用不同奇异值和对应的奇异向量对图 5.7 恢复的图像。从恢复的结果看，k 值越大，恢复得效果越好，但是压缩比降低。原始图像的数据量是 427×640×3＝819840(B)，压缩之后的数据量以奇异值为 1 的情况为例说明，假设 1 个奇异值占 1 字节，一个颜色矩阵只需要保存左奇异值矩阵 1 列 427 个数据，加上右奇异值矩阵一行 640 个数据，这样就是(427＋640＋1)×3＝3204(B)。压缩比利用原始图像的数据量除以压缩之后的数据量来计算。压缩之后的数据量及压缩比如表 5.2 所示。

图 5.7　图像压缩的原始图像

表 5.2　前 12 幅恢复图像的压缩比

K 值	1	2	3	4	5	6
数据量	3204	6408	9612	12816	16020	19224
压缩比	255.88	127.94	85.29	63.97	51.18	42.65
K 值	7	8	9	10	11	12
数据量	22428	25632	28836	32040	35244	38448
压缩比	36.55	31.99	28.43	25.59	23.26	21.32

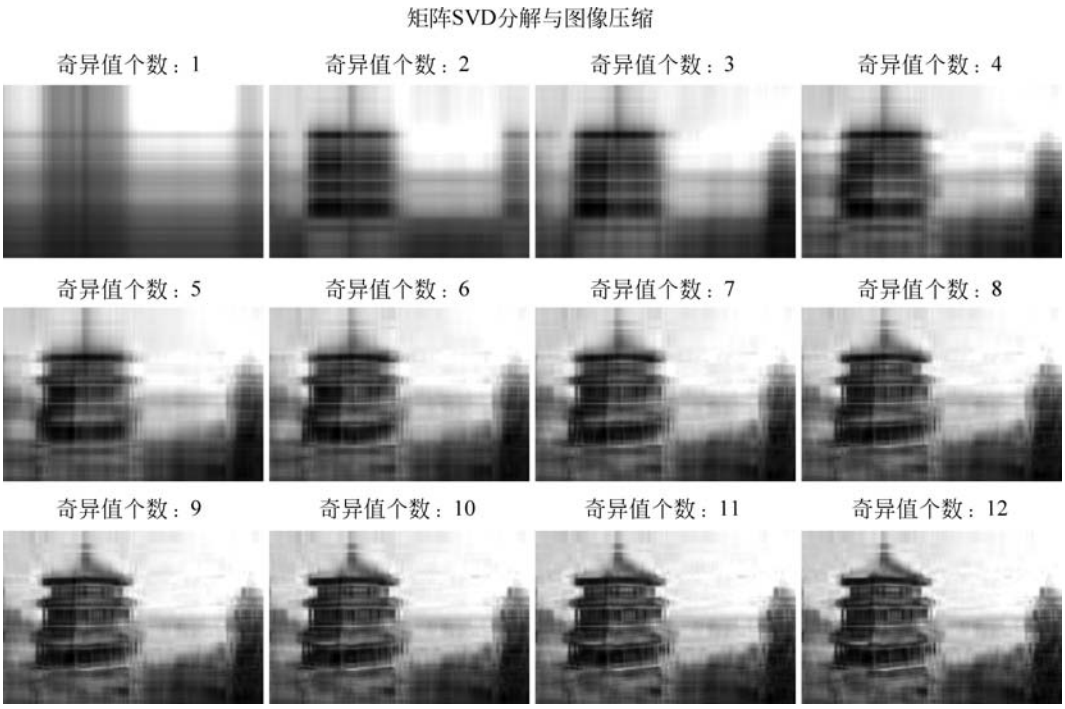

图 5.8　分别利用前 12 个奇异值和奇异向量恢复的图像

矩阵奇异值分解的案例代码的篇幅明显变长，代码量增加并不完全意味着复杂度增加。代码由一些功能点组成，每个功能点并不难理解，在这些功能点的基础上组合拼凑构成能够实现相对复杂功能的长篇代码。

以上代码演示了如下语法点。

（1）库的导入与库函数的调用。

（2）自定义函数和模块的创建。

（3）内置名称__main__的问题。

（4）图像的显示和保存问题。

（5）多种方法计算矩阵乘法。

这些语法点在代码中都有体现，读者注意细心体会，在程序设计时遇到类似的应用可以加以模仿，变成自己解决实际问题的手段和方法。

5.6　实　　验

1. 实验目的

（1）掌握函数的定义和使用。

（2）掌握 lambda 表达式的使用。

（3）掌握模块、包的创建。

（4）掌握函数不同参数的使用。

2. 实验内容

(1) 编写函数,接收任意多个实数,返回一个元组,其中第 1 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。参考程序 5.29。

```
#程序 5.29  函数计算平均值
def pingjun(*para):
    avg=sum(para)/len(para)
    g=[i for i in para if i>avg]
    return (avg,)+tuple(g)
print(pingjun(5,6,7,82,3,1,20))
```

(2) 编写函数,接收字符串参数,返回一个元组,其中第 1 个元素为大写字母个数,第 2 个元素为小写字母个数。参考程序 5.30。

```
#程序 5.30  函数计数大小写字母个数
def charac(s):
    result = [0, 0]
    for ch in s:
        if ch.islower():
            result[1] += 1
        elif ch.isupper():
            result[0] += 1
    return tuple(result)
print(charac('Python is very easy, I\'m interested in programming'))
```

(3) 编写函数,接收包含 n 个整数的列表 lst 和一个整数 $k(0\leqslant k<n)$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。参考程序 5.31。

```
#程序 5.31  函数处理列表
def nixu(lst, k):
    x=lst[k-1::-1]
    y=lst[:,k-1::-1]
    return list(reversed(x+y))
```

(4) 从第(1)项～第(3)项有多种不同的方法可以实现,虽然给出了参考程序,请读者自行尝试使用不同的方法实现同样的功能。

(5) 参考 5.3 节,将自己的人脸图像进行图像压缩,并查看恢复的人脸图像效果。

(6) 根据每一步的结果写出实验报告。

本 章 小 结

本章主要介绍了函数的定义及使用,不同种类的函数参数的用法,以及由函数组合构成的模块和包的使用;还介绍了函数的简化版本 lambda 表达式,以及包含 yield 语句的生成器函数的用法,给出了函数在机器学习中矩阵分析上的应用。

习　题

一、选择题

1. 可以使用(　　)关键字创建 Python 自定义函数。

A. function　　B. func　　C. produce　　D. def

2. 下面程序运行的结果为(　　)。

```
a = 10
def setNumber():
    a = 100
setNumber()
print(a)
```

A. 10　　B. 100　　C. 10100　　D. 10010

3. 关于函数参数传递,形参和实参描述错误的是(　　)。

A. Python 按照值传递参数,调用函数是将常量或变量的值传递给函数的参数

2. 实参与形参分别存储在各自的内存空间中,是两个独立不相关的变量

3. 在函数内部改变形参的值时,实参的值一般不会改变

4. 实参与形参的名字必须相同

4. 下面程序运行的结果为(　　)

```
def swap(list):
    temp = list[0]
    list[0] = list[1]
    list[1] = temp
list = [1,2]
swap(list)
print(list)
```

A. [1,2]　　B. [2,1]　　C. [1,1]　　D. [2,2]

二、填空题

1. 函数可以包含多个参数,参数之间使用_____分隔。

2. 使用_____语句可以返回函数值并退出函数。

3. 返回 x 的 y 次幂的函数是_____。

4. 返回 x 的绝对值的函数是_____。

5. 将字符串的字母转换成小写字母的函数是_____。

6. 替换字符串中子串的函数为_____。

7. _____函数用于显示指定参数的帮助信息。

三、程序和简答题

1. 杨辉三角是二项式系数在三角形中的一种几何排列,编写函数,接收一个整数 t 作为参数,打印杨辉三角前 t 行。

2. 编写函数,接收两个正整数为参数,返回一个元组,其中第 1 个元素为最大公约数,第 2 个元素为最小公倍数。

3. 编写函数模拟报数游戏，有 n 个人围成一个圈，顺序编号，第 1 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，问最后留下的人是原来的第几号。

4. 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 1 个元素为序列最大值，第 2 个元素为序列最小值。

5. 编写函数，实现冒泡排序算法。所谓冒泡排序就是重复地遍历要排序的序列，依次比较两个相邻的元素，如果顺序相反就把他们交换过来。重复直到所有元素都顺序正确。

6. 利用 lambda 表达式和 filter()函数求 100 以内的素数。

7. 编写函数，接收一个字符串，分别统计大写字母、小写字母、数字以及其他字符的个数，并以元组的形式返回。