第 3 章

函数与分支

第 2 章介绍了 Python 中的基本运算和基本数据类型，示例程序仅执行了少量计算，这些计算使用常规计算器即可以轻松完成。现在考虑以下高斯函数：

$$f(x)=\frac{1}{\sigma\sqrt{2\pi}}\mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3-1}$$

这是一个数学中比较复杂的函数。即使使用科学计算器，也不能很轻松地完成函数运算。这一章将重点介绍如下内容：

(1) 如何将数学方程编写成 Python 的程序。

(2) 如何使用条件判断控制程序执行的流程。

3.1 使用函数

此处需要进行求平方根和指数运算，它们均不属于 Python 的基本运算。Python 之所以流行，一个主要的原因是它拥有丰富的库(标准库和第三方库)。Python 的库包含了一些定义和语句。库不属于核心语言，在使用前需要导入。

高级运算以函数的形式被封装在 Python 的 Math 库里。以下为式(3-1)的代码实现：

```
In[1]: from math import sqrt, pi, exp
       σ = 1
       μ = 2
       x = 3.5
       f = 1/sqrt(2 * σ * pi) * exp( - 0.5 * ((x - μ) ** 2/σ))
       print(f)
0.12951759566589174
```

这里，sqrt()和 exp()是两个函数，pi 是一个常量，它们均由 Math 库导入。print()是 Python 的内建函数，因此不需导入便可以直接使用。

库导入的方法有如下 3 种：

(1) 仅导入库名，但是不将库中的任何符号导入当前程序的符号表。

(2) 将某库中指定的符号导入当前程序的符号表。

(3) 将某库中所有符号导入当前符号表。

这 3 种方法各有利弊。

第 1 种方法最安全，因为它实际上并未导入任何符号，因此不会造成命名空间中出现名字冲突，但是每次使用某个符号前，必须指明它来自哪个库，代码如下：

```
In [2]: import math
        math.log(2.3)
Out[2]: 0.8329091229351039

In [3]: math.sin(1.2)
Out[3]: 0.9320390859672263
```

第 2 种方法，在确定不会引入冲突的前提下，仅导入需要使用的符号，使用时不需要再指明库名，代码如下：

```
In [4]: from math import log
         log(2.3)
Out[4]: 0.8329091229351039
```

第 3 种方法，程序写起来会很简单，但增加了名字冲突的可能性，在复杂的程序中不建议使用。例如 Math 库含有一个数学常数 math.e，代码如下：

```
In [5]: math.e
Out[5]: 2.718281828459045
```

而在某些第三方库中也可能存在具有不同意义的常量 e，例如在 scipy.constants 中，代码如下：

```
In [6]: import scipy.constants
         scipy.constants.e
Out[6]: 1.6021766208e-19
```

如果我们将它们一起导入，则会出现名字空间污染的情况，代码如下：

```
In [7]: from math import *
        from scipy.constants import *
        print(e)
1.602176634e-19
```

此处，我们看到无法使用 Math 库中的 e，因为在第二次导入之后，这个名字现在关联的是代表电子电量的对象。

为解决这个问题，Python 引入了别名机制，代码如下：

```
In [8]: from math import e
        from scipy.constants import e as charge_e
        print(e, charge_e)
2.718281828459045 1.6021766208e-19
```

这里，我们将电子电量重命名为 charge_e。这一导入方式不仅可以避免符号名的冲突，

还可以简化某些较长的符号名，代码如下：

```
In [9]: import numpy as np
        import matplotlib.pyplot as plt
```

这样在后续代码中，就可以节省输入的字符数了。

对于某个库的使用如有任何疑问，可以在该库的发布网站找到最权威的解答。对于Python标准库，帮助文件可以在Python官网查询，也可以使用pydoc。如果只是想查看某个库内开放的外部符号，则可以使用Python内建函数dir()，示例代码如下：

```
In [10]: import math
          dir(math)
Out[10]:
['__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 …… 
'tau',
'trunc']
```

3.2　Python Math 模块

对于Python中简单的数学计算，可以使用内置的数学运算符，例如加法“+”、减法“-”、除法“/”和乘法“*”，但更高级的运算，如指数、对数、三角或幂函数，则没有内置。这是否意味着需要从头开始实现这些函数呢？幸运的是，这是没有必要的。Python提供了Math模块，该模块将提供高级数学函数。

接下来，将讲述如下内容：

(1) Python Math模块是什么。

(2) 如何利用Math模块功能解决现实生活中的问题。

(3) Math模块的常数是什么，包括pi、tau和欧拉数。

(4) 内置函数和Math库函数有什么区别。

(5) Math和cmath之间的区别是什么。

Python Math模块被用来处理数学运算。它随标准Python一起打包发布。Math模块的大部分函数是对C平台的数学函数的简单包装。因为它的底层函数是用CPython编写的，所以数学模块是高效的，并且符合C标准。

Python Math模块使应用程序可以执行常见和有用的数学计算。下面是Math模块的一些实际用途：

(1) 使用阶乘计算组合数和排列数。

(2) 用三角函数计算杆的高度。

(3) 用指数函数计算放射性衰变。

(4) 用双曲函数计算悬索桥的曲线。

(5) 解二次方程。

(6) 利用三角函数模拟周期函数,例如声波和光波等。

由于 Math 模块随 Python 发行版一起发布,所以不必单独安装它。使用它只需使用下面的代码直接导入:

```
In [11]: import math
```

导入之后,我们就可以使用 Math 模块了。

3.2.1 常数

Python Math 模块提供了各种预定义的常量。访问这些常量有几个优点:首先,不必手动将它们硬编码到程序中,这节省了大量时间;另外,它们为整个代码提供了一致性。该模块包括如下几个重要的数学常数和重要值:

(1) π。

(2) τ。

(3) 欧拉数。

(4) ∞。

(5) 非数值(NaN)。

1. Pi

圆周率(π)是圆的周长(c)与直径(d)之比:$\pi=c/d$,这个比率对任何圆都是相同的。

π 是一个无理数,这意味着它不能表示为一个简单的分数,因此,π 有无限个小数位数,但它可以近似为 22/7 或 3.141。圆周率是世界上公认的非常重要的数学常数。它有自己的庆祝日,称为圆周率日,即 3 月 14 日。

以下代码使用 Math 模块中的 Pi 常量显示 π 的近似值:

```
In [12]: math.pi
Out[12]: 3.141592653589793
```

在 Python 中 Pi 的值被指定为小数点后 15 位。所提供的位数取决于底层 C 编译器。Python 默认情况下输出前 15 位数字。Pi 总是返回一个浮点值。

我们可以用 $2\pi r$ 计算一个圆的周长,其中 r 是圆的半径,代码如下:

```
In [13]: r = 3
         circumference = 2 * math.pi * r
         print("圆的周长 = 2 * %.4f * %d = %.4f" %(math.pi, r, circumference))
圆的周长 = 2 * 3.1416 * 3 = 18.8496
```

以下代码用于计算圆的面积:

```
In [14]: r = 5
         area = math.pi * r * r
         print("圆的面积 = %.4f * %d^2 = %.4f" %(math.pi, r, area))
圆的面积 = 3.1416 * 5^2 = 78.5398
```

如上所示，使用 Python 进行数学计算，如果遇到 π，则最好的做法是使用 Math 模块给出的 Pi 值，而不是对该值进行硬编码。

2. Tau

Tau(τ)是圆的周长与半径的比值。这个常数等于 2π，它也是一个无理数，大约是 6.28。

许多数学表达式含有 2π，而使用 τ 可以帮助简化算式。例如，可以代入 τ，使用更简单的算式 τr，而不是用 $2\pi r$ 来计算圆的周长。

然而，使用 τ 作为常数仍在争论中。如果需要，则可以使用如下代码：

```
In [15]: math.tau
Out[15]: 6.283185307179586

In [16]: r = 3
         circumference = math.tau * r
         print("圆的周长 = %.4f * %d = %.4f" %(math.tau, r, circumference))
圆的周长 = 6.2832 * 3 = 18.8496
```

3. 欧拉数

欧拉数(e)是一个常数，它是自然对数的基础，自然对数是一种常用来计算增长率或衰减率的数学函数。和 π、τ 一样，欧拉数是一个无理数，小数点后有无限个数位。e 的值常近似为 2.718。

欧拉数是一个重要的常数，因为它有许多实际用途，如计算人口随时间的增长或确定放射性衰变率等。可以从 Math 模块中访问欧拉数，代码如下：

```
In [17]: math.e
Out[17]: 2.718281828459045
```

4. 无穷大

无穷大(∞)不能用数来定义。相反，它是一个数学概念，代表着无穷无尽的事物。无穷大可以向正负两个方向移动。

如果需要将给定值与绝对最大值或最小值进行比较，则可以在算法中使用无穷大。Python 中表示正无穷大和负无穷大的值的代码如下：

```
In [18]: print("正无穷大是", math.inf)
         print("负无穷大是", -math.inf)
正无穷大是 inf
负无穷大是 -inf
```

无穷大不是一个数值。相反，它被定义为 math.inf。Python 在 3.5 版本中引入了这个常量，它相当于 float("inf")，代码如下：

```
In [19]: float("inf") == math.inf
Out[19]: True
```

float("inf")和 math.inf 都表示无穷大的概念，math.inf 大于任何数值，示例代码如下：

```
In [20]: x = 10e308
         math.inf > x
Out[20]: True
```

在上面的代码中，math.inf 大于 10^{308}（双精度浮点数的最大值）。同样，－math.inf 小于任何值。

没有一个数可以大于无穷大或小于负无穷大。任何针对 math.inf 的数学运算都不能改变它的值，示例代码如下：

```
In [21]: math.inf + 1e308, math.inf / 1e308
Out[21]: (inf, inf)
```

可见，加法和除法都不会改变 math.inf 的值。

5. 非数值

非数值（NaN）并不是一个真正的数学概念。它起源于计算机科学领域，是指对非数值的引用。Python 在 3.5 版中引入了 NaN 常量。NaN 值可能是由于无效输入引起的，也可能表示作为数的变量已被文本字符或符号破坏。在 Python 数值运算中，如果引入 NaN，则可能导致程序的结果值无效，因此，在必要的情况下，必须检测一个变量的值是否为 NaN。

NaN 不是一个数。math.nan 的值是 nan，与 float("nan")的值相同。

3.2.2 算术函数

数论是纯数学的一个分支，纯数学是对自然数的研究。数论通常处理正整数或整数。

Python Math 模块提供了在数论和表示理论（一个相关领域）中有用的函数。这些函数允许计算一系列重要的值，包括以下内容：

（1）数的阶乘。

（2）两个数的最大公约数。

（3）可迭代对象的和。

1. factorial()

函数 factorial()用于排列或组合运算，可以通过将所选数和 1 之间的所有整数相乘来确定一个数的阶乘。阶乘的数学描述为 $n!$。

表 3-1 给出了 4、6 和 7 的阶乘值。

表 3-1　4、6 和 7 的阶乘

符　号	描　述	表 达 式	结　果
4!	4 的阶乘	4×3×2×1	24
6!	6 的阶乘	6×5×4×3×2×1	720
7!	7 的阶乘	7×6×5×4×3×2×1	5040

从表 3-1 中可以看到，4！也就是 4 的阶乘，将 4 到 1 的所有整数相乘得到 24。同样，6！和 7！分别得到 720 和 5040。

在 Python 中，可以使用以下方法实现阶乘函数：

（1）for 循环。

（2）递归函数。

（3）math.factorial()。

以下代码使用 for 循环实现阶乘，这是一个相对简单的方法：

```
In [22]: def Factorial(num):
             """
             使用 for 循环实现阶乘

             参数

             ----
             num: int
                 求 num 的阶乘

             返回值

             -----
             factorial: int
                 阶乘值
             """
             if num < 0:
                 return 0
             if num == 0:
                 return 1

             factorial = 1
             for i in range(1, num + 1):
                 factorial = factorial * i
             return factorial

         Factorial(7)
Out[22]: 5040
```

更方便的方法是直接使用 Math 库的函数 math.factorial()，代码如下：

```
In [23]: math.factorial(7)
Out[23]: 5040
```

函数 math.factorial()仅接收非负整数，如果传入负数或小数作为参数，则将得到 ValueError 异常。示例代码如下：

```
In [24]: math.factorial(4.3)
Traceback (most recent call last):
```

```
  File "< ipython - input - 24 - ad0d56075f62 >", line 1, in < module >
    math.factorial(4.3)

ValueError: factorial()only accepts integral values
```

错误提示中显示，该函数仅能接收整数。

```
In [25]: math.factorial( - 5)
Traceback (most recent call last):

  File "< ipython - input - 25 - a46d876612ec >", line 1, in < module >
    math.factorial( - 5)

ValueError: factorial()not defined for negative values
```

错误提示中显示，该函数不支持负数作为入参。

下面我们比较以下 Math 库函数 math.factorial()与我们自定义的函数 Factorial()运算效率的差别。

```
In [26]: % timeit Factorial(100)
9.58 μs ± 7.42 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

In [27]: % timeit math.factorial(100)
1.56 μs ± 8.02 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

从执行时间可以看出，Math 库函数 math.factorial()比纯 Python 代码的实现更快。这是因为它的底层由 C 实现。尽管由于 CPU 的不同，可能会得到不同的计时结果，但 math.factorial()总是最快的。

函数 math.factorial()不仅更快，而且更稳定。当实现自己的函数时，必须显式地为各种异常情况编写代码，例如处理负数或小数输入。实现中的一个错误可能导致程序运行出现 Bug，但是在使用 math.factorial()时，不必担心灾难情况的出现，因为函数会处理所有灾难情况，因此，最好的实践是尽可能地使用 math.factorial()。

2. ceil()

函数 math.ceil()将返回大于或等于给定值的最小整数。无论正数或负数，该函数都将返回下一个大于给定值的整数。例如，输入 5.43 将返回值 6，而输入－12.43 将返回值－12。

math.ceil()可以将正实数或负实数作为输入值，并且始终返回整数。当向 math.ceil()输入一个整数时，它将返回相同的数。示例代码如下：

```
In [28]: math.ceil(5.43), math.ceil( - 12.43)
Out[28]: (6, - 12)

In [29]: math.ceil(6), math.ceil( - 11)
Out[29]: (6, - 11)
```

如果输入的值不是数，则函数将返回 TypeError，代码如下：

```
In [30]: math.ceil("x")
Traceback (most recent call last):

  File "<ipython-input-30-6b47497c589c>", line 1, in <module>
    math.ceil("x")

TypeError: must be real number, not str
```

3. floor()

函数 math.floor()的行为与 math.ceil()相反，它将返回小于或等于给定值的最接近的整数。例如，输入 8.72 将返回 8，输入－12.34 将返回－13。

math.floor()可以接收正数或负数作为输入，并返回一个整数。如果输入一个整数，则函数将返回相同的值。示例代码如下：

```
In [31]: math.floor(8.72), math.floor(-12.34)
Out[31]: (8, -13)

In [32]: math.floor(6), math.floor(-11)
Out[32]: (6, -11)
```

同样，如果输入的值不是数，则函数 math.floor()将返回 TypeError，代码如下：

```
In [33]: math.floor("x")
Traceback (most recent call last):

  File "<ipython-input-33-c49ad4f39c09>", line 1, in <module>
    math.floor("x")

TypeError: must be real number, not str
```

4. trunc()

当需要只保留某个小数的整数部分时，可以使用 Math 模块的函数 math.trunc()。

去掉小数是一种舍入方法。使用函数 math.trunc()，负数总是向上取整（类似于函数 math.ceil()），正数总是向下取整（类似于函数 math.floor()）。

下面的代码演示函数 math.trunc()如何使正数或负数截尾：

```
In [34]: math.trunc(12.52), math.trunc(-43.24)
Out[34]: (12, -43)
```

5. isclose()

在某些情况下，特别是在数据科学领域中，可能需要确定两个数是否彼此接近，但要做到这一点，首先需要回答一个重要的问题：多接近才算接近？

例如，取一组数：2.32、2.33 和 2.331。当我们仅比较小数点后两位时，会觉得 2.32 和

2.33非常接近，但实际上，2.33和2.331更接近。“接近”是一个相对的概念。如果没有某种阈值，则无法确定接近程度。

幸运的是，Math模块提供了一个名为isclose()的函数，允许设置阈值或容忍值。如果两个数值的差值在设定的接近度容忍范围内，则返回值为True，否则返回值为False。

下面来看一看如何使用默认公差来比较两个数值。

(1) 相对容差(rel_{tol})，是评估实际值与预期值之间的差异相对于预期值的量值。这是容忍的百分比。默认值为1e−09或0.000000001。

(2) 绝对容差(abs_{tol})，被认为是“接近”的最大差值，而不管输入值的大小。默认值是0.0。

当满足以下条件时，函数math.isclose()的返回值为True：

$$abs(a-b) \leqslant max(rel_{tol} \times max(abs(a),abs(b)),abs_{tol}) \tag{3-2}$$

函数math.isclose()使用上面的表达式来确定两个数的接近度。

在下面的例子中，数字6和7被认为不接近：

```
In [35]: math.isclose(6, 7)
Out[35]: False
```

数字6和7被认为不接近，因为默认的相对容差设置为小数点后9位，但是如果在相同的容差下输入6.999999999和7，则它们被认为是接近的，代码如下：

```
In [36]: math.isclose(6.999999999, 7)
Out[36]: True
```

可以看到6.999999999在7的小数点后9位之内，因此，基于默认的相对容差，6.999999999和7被认为是接近的。

可以根据需要任意调整相对容差。如果将rel_tol设置为0.2，则认为6和7很接近，代码如下：

```
In [37]: math.isclose(6, 7, rel_tol = 0.2)
Out[37]: True
```

可以观察到6和7现在已经被认为接近了。这是因为它们彼此之间的距离不超过20%。

与rel_tol一样，也可以根据需要调整abs_tol的值。要被视为接近，输入值之间的差异必须小于或等于绝对容差值。以下代码设置abs_tol的值：

```
In [38]: math.isclose(6, 7, abs_tol = 1.0)
Out[38]: True

In [39]: math.isclose(6, 7, abs_tol = 0.2)
Out[39]: False
```

我们可以使用函数math.isclose()确定非常小的数之间的接近程度。下面的代码使用

nan 和 inf 来定义几个关于接近程度的特殊情况：

```
In [40]: math.isclose(math.nan, 1e308)
Out[40]: False

In [41]: math.isclose(math.nan, math.nan)
Out[41]: False

In [42]: math.isclose(math.inf, 1e308)
Out[42]: False

In [43]: math.isclose(math.inf, math.inf)
Out[43]: True
```

从上面的例子可以看出，nan 不接近任何值，甚至不接近它本身。另一方面，inf 不接近任何数值，甚至不接近非常大的数值，但它很接近自身。

6. pow()

函数 math.pow()用来求一个数的幂。Python 另有一个内置函数 pow()，它与 math.pow()不同。关于它们的不同，稍后会做出说明。math.pow()接收 2 个参数，代码如下：

```
In [44]: math.pow(2, 5)
Out[44]: 32.0

In [45]: math.pow(5, 2.4)
Out[45]: 47.59134846789696
```

函数 math.pow()的第 1 个参数是底数，第 2 个参数是指数。可以提供整数或小数作为输入，函数总是返回一个浮点值。在 math.pow()中定义了一些特殊情况。

当以 1 为底取任意次幂时，其结果均为 1.0。示例代码如下：

```
In [46]: math.pow(1.0, 3)
Out[46]: 1.0
```

任何底数的 0 次幂的结果总是 1.0。即使底是 nan，其结果也是 1.0。示例代码如下：

```
In [47]: math.pow(4, 0.0)
Out[47]: 1.0

In [48]: math.pow(-4, 0.0)
Out[48]: 1.0

In [49]: math.pow(0, 0.0)
Out[49]: 1.0

In [50]: math.pow(math.nan, 0.0)
Out[50]: 1.0
```

0.0 的任何正数次幂都是 0.0。示例代码如下：

```
In [51]: math.pow(0.0, 2)
Out[51]: 0.0

In [52]: math.pow(0.0, 2.3)
Out[52]: 0.0
```

但是取 0.0 的负数次幂将得到 ValueError。

除了 math.pow()之外，Python 中还有 2 种计算幂值的内置方法：

(1) x ** y。

(2) pow()。

第 1 种方法很简单。前文中已经用过了。第 2 种方法是一个通用的内置函数。内置的 pow()不必任何导入便可以使用，该函数有 3 个参数：

(1) base 为底数。

(2) power 为幂指数。

(3) modulus 为模数。

前 2 个参数是强制的，第 3 个参数是可选的。入参可以是整型或浮点型，函数将根据输入参数的类型返回适当的结果。示例代码如下：

```
In [53]: math.pow(3, 2)
Out[53]: 9

In [54]: math.pow(2, 3.3)
Out[54]: 9.849155306759329
```

密码术中经常使用此参数。带有可选模数参数的内置函数 pow()等价于$(x ** y)\%z$。Python 语法如下：

```
In [55]: math.pow(32, 6, 5), (32 ** 6) % 5 == math.pow(32, 6, 5)
Out[55]: (4, True)
```

函数 pow()计算底数 32 的 6 次幂模 5，在这种情况下，结果是 4。

尽管这 3 种计算幂的方法可以得到相同的结果，但它们之间有一些实现上的差异，因此会影响执行的效率。下面是对它们执行效率的比较：

```
In [56]: %timeit 10 ** 308
1.04 μs ± 2.49 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

In [57]: %timeit pow(10, 308)
1.09 μs ± 11.3 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

In [58]: %timeit math.pow(10, 308)
231 ns ± 0.765 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

在不同的 CPU 下可能会得到不同的数值结果，但无论在哪一种平台上，最终的结果都表明 Python 内建函数 pow()的运行效率是最低的。函数 math.pow()的底层实现依赖于 C 语言，这使得它具有比较高的运行效率，然而 math.pow()不能处理复数，而 Python 内建函数 pow()和 x ** y 使用输入对象自己的操作符 ** ，因此 pow()和 ** 都可以处理复数。

7. exp()

当以欧拉数作为指数函数的底数时，该指数函数就是自然指数函数。函数 math.exp()与数学中的自然指数函数相对应。输入可以是正数也可以是负数，函数 math.exp()总是返回一个浮点值。如果输入的不是数值，则该方法将返回 TypeError，代码如下：

```
In [59]: math.exp(21)
Out[59]: 1318815734.4832146

In [60]: math.exp( - 1.2)
Out[60]: 0.30119421191220214

In [61]: math.exp("x")
Traceback (most recent call last):

  File "< ipython - input - 61 - d942eff83a60 >", line 1, in < module >
    math.exp("x")

TypeError: must be real number, not str
```

也可以用 e ** x 表达式或通过使用 pow(math.e,x)计算自然指数。这 3 种方法的执行时间对比如下：

```
In [62]: % timeit math.e ** 308
230 ns ± 13.5 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

In [63]: % timeit pow(math.e, 308)
271 ns ± 18.9 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

In [64]: % timeit math.exp(308)
171 ns ± 3.31 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)

In [65]: % timeit math.pow(math.e, 308)
265 ns ± 1.32 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

可以看到 math.exp()比其他方法快，而 pow(e,x)是最慢的。这与预期相同，因为 Math 模块的底层是由 C 语言实现的。

值得注意的是，e ** x 和 pow(e,x)返回相同的值，但 math.exp()返回的值略有不同。这是由于实现的差异造成的。Python 文档指出，math.exp()比其他 2 种方法更精确，代码如下：

```
In [66]: math.e ** 21, pow(math.e, 21), math.exp(21)
Out[66]: (1318815734.4832132, 1318815734.4832132, 1318815734.4832146)
```

当一个不稳定的原子发射电离辐射而失去能量时，就会发生放射性衰变。放射性衰变的速率是用半衰期来测量的，半衰期是母核衰变一半所花费的时间。可以用以下公式计算衰减过程：

$$N(t)=N(0)e^{\frac{-693t}{T}} \tag{3-3}$$

现在用上面的公式计算某放射性元素在一定时间后的剩余量。给定公式的变量如下：

(1) $N(0)$是物质的初始量。

(2) $N(t)$是经过一段时间 t 还没有衰减的量。

(3) T 是衰变量的半衰期。

(4) e 是欧拉数。

科学研究已经确定了所有放射性元素的半衰期。可以将相应的值代入方程式来计算任何放射性物质的剩余量。

放射性同位素锶-90 的半衰期为 38.1 年。样本中含有 100 毫克锶-90。以下代码计算 100 年后剩余的锶-90：

```
In [67]: half_life = 38.1
         initial = 100
         time = 100
         remaining = initial * math.exp(-0.693 * time / half_life)
         print("剩余的锶-90 有: %.4f 毫克" %remaining)
剩余的锶-90 有: 16.2204 毫克
```

8. log()

对数函数可以看作指数函数的逆函数。一个数的自然对数是以自然常数(或欧拉数)e 为底数的对数。Math 模块的函数 math.log()有 2 个参数。第 1 个参数是强制性的，第 2 个参数是底数，它是可选的，默认为 e。通过第 1 个参数，可以得到输入值的自然对数(以 e 为底)，代码如下：

```
In [68]: math.log(4)
Out[68]: 1.3862943611198906

In [69]: math.log(math.e)
Out[69]: 1.0

In [70]: math.log(-3)
Traceback (most recent call last):

  File "<ipython-input-70-986324ee0bdd>", line 1, in <module>
    math.log(-3)

ValueError: math domain error
```

```
In [71]: math.log('x')
Traceback (most recent call last):

  File "<ipython-input-71-0c782b7d43c0>", line 1, in <module>
    math.log('x')

TypeError: must be real number, not str
```

函数 math.log()接收正数为输入参数，不接收负数和字符串，因为负数和 0 的对数是没有定义的。

通过输入不同的值，可以求不同底数的对数，代码如下：

```
In [72]: math.log(4, 2)
Out[72]: 2.0

In [73]: math.log(math.pi, 5)
Out[73]: 0.711260668712669
```

以上代码分别求以 2 为底 4 的对数和以 5 为底 π 的对数。

Python Math 模块还提供了两个单独的函数，分别用以计算以 2 和 10 为底的对数，代码如下：

```
In [74]: math.log10(math.pi), math.log(math.pi, 10)
Out[74]: (0.49714987269413385, 0.4971498726941338)

In [75]: math.log2(math.pi), math.log(math.pi, 2)
Out[75]: (1.6514961294723187, 1.651496129472319)
```

以上结果略有不同，Python 文档指出 log10()和 log2()拥有更高的准确度，尽管它们都可以通过向 log()传入第 2 个参数替换，因此在有较高精确度要求的情况下，尽量使用这 2 个函数。

前述的例子使用 math.exp()计算放射性元素在一段时间后的剩余质量。我们也可以通过测量一个间隔的质量变化来找到未知放射性元素的半衰期，此时需要使用 math.log()。

下式是用来计算放射性元素的半衰期的公式：

$$T = \frac{-693t}{\ln \frac{N(t)}{N(0)}} \tag{3-4}$$

给定公式的变量说明如下：

(1) T 是半衰期。

(2) $N(0)$是物质的初始量。

(3) $N(t)$是经过一段时间 t 还没有衰减的量。

(4) ln 是自然对数。

假设有一个未知的放射性元素样本。100 年前它被发现时，样本量是 100 毫克。经过

100 年的衰变后，只剩下 16.22 毫克。以下代码使用式(3-4)计算出这个未知元素的半衰期：

```
In [76]: initial = 100
         remaining = 16.22
         time = 100
         half_life = (-0.693 * time) / math.log(remaining / initial)
         print("未知物的半衰期是 %f" %(half_life 年))
未知物的半衰期是 38.099424 年
```

可以看到这个未知元素的半衰期大约是 38.1 年。根据这些信息，可以确定这个未知元素是锶-90。

3.2.3 Math 库中其他的重要数学函数

Python Math 模块有许多用于数学计算的有用函数，接下来将简要地介绍 Math 模块中其他一些重要函数。

1. gcd()

两个正数的最大公约数(GCD)是能将两个数整除的最大正整数。

例如，15 和 25 的 GCD 是 5，5 是能够同时整除 15 和 25 的最大整数。15 和 30 的 GCD 是 15，因为 15 和 30 都可以被 15 整除，没有余数。

计算 GCD 的算法很多，但是我们无须自己实现。Python Math 模块提供了一个名为 math.gcd()的函数，可以计算两个数的 GCD。它接收正数或负数作为输入，并返回适当的 GCD，但是，小数不能作为输入。

2. sum()

如果想要在不使用循环的情况下找到可迭代对象值的和，则 math.fsum()可能是最简单的方法。可以使用数组、元组或列表等迭代对象作为输入，求函数返回值的和。一个名为 sum()的内置函数也允许计算可迭代对象的和，但 math.fsum()比 sum()更精确。

3. sqrt()

一个数的算术平方根是一个值，当它与自己相乘时，就得到这个数。可以使用 math.sqrt()找到任何正实数(整数或小数)的算术平方根。返回值始终是一个浮点数。如果尝试输入一个负数，则函数将抛出一个 ValueError 异常。

4. radians()

在现实生活和数学中，经常会遇到需要测量角度才能进行计算的情况。角度可以用度或弧度来测量。有时必须将度转换成弧度，反之亦然。如果想把度转换成弧度，则可以使用 math.radians()，它返回输入的弧度值。同样地，如果想将弧度转换为度，则可以使用 math.degrees()。

5. 三角函数

三角学是对三角形的研究。它处理三角形的角和边之间的关系。三角学主要对直角三角形感兴趣，但它也可以应用于其他类型的三角形。Python Math 模块提供了非常有用的函数，可以执行三角计算。相关函数如表 3-2 所示。

表 3-2　Math 库的三角函数

函　　数	描　　述
math. sin()	计算正弦值
math. cos()	计算余弦值
math. tan()	计算正切值
math. asin()	计算反正弦
math. acos()	计算反余弦
math. atan()	计算反正切
math. hypot()	计算直角三角形的斜边长

以上三角函数的入参和反三角函数返回值都是弧度，参看以下代码：

```
In [77]: math.cos(math.pi)
Out[77]: -1.0

In [78]: math.acos(-1)
Out[78]: 3.141592653589793
```

math. cos()的输入是弧度，如果应用中是度，则需要首先使用 math. radians()将其转化为弧度。其他的三角函数具有相同的特征。

反三角函数的输入是度，如果应用中是弧度，则需要首先使用 math. degrees()将其转化为度。其他的反三角函数具有相同的特征。

函数 math. hypot()根据直角三角形的两个直角边计算斜边，代码如下：

```
In [79]: parendicular = 3
         base = 4
         math.hypot(parendicular, base)
Out[79]: 5.0
```

3.3　定义函数

如果我们需要求高斯函数的多个值，则需要重复输入复杂的表达式，显然这不是一个高效的做法。编码中有一个 DRY 原则，即 Do not Repeat Yourself 原则。无论是库函数还是内建函数，最主要的目的是使具有明确功能的代码可以被复用。在 3.1 节的几段代码中，我们无须编写进行方根运算和指数运算的代码。在每次需要输出时，也无须特意编写用于输出的代码。我们所需的功能已经作为函数存在了。同样，我们可以自定义一个函数 gaussian()，在需要求值的地方调用它。

通过 3.2 节的例子，我们可以看到程序中的函数和数学中的函数有很多相似的地方。简单来讲就是将输入映射成输出，每一组输入均有唯一的输出与之对应。不过程序中的函数不仅仅可以做数学运算。

函数使我们可以把复杂的任务分解成多个较小的任务，这对于解决复杂的问题是必不可少的。另外，将一个程序分割成多个较小的函数对于测试和验证一个程序是否正常工作

也很方便。我们可以编写小段代码来测试单独的函数，在将这些函数放入一个完整的程序之前确保它们能够正常工作。如果这些测试都正确地完成了，就可以确信主程序能够按照预期的方式工作。

现在，我们将数学上的高斯方程写成 Python 函数，代码如下：

```
In [80]: from math import sqrt, pi, exp
         def gaussian(x):
             σ = 1
             μ = 2
             g = 1/sqrt(2 * σ * pi) * exp( - 0.5 * ((x - μ) ** 2/σ))
             return g
```

与 3.2 节相比，这里代码有三处改动。def gaussian(x)：被称为函数头，用于定义函数的接口。Python 中使用关键字 def 定义函数，def 之后是函数的名称，紧接着是括号包裹的函数入参。入参的个数可以是 0 个也可以是多个。如果有多个入参，则各入参之间需要使用逗号分隔。

函数名的命名约定和变量名类似（小写字符、不能有空格、下画线连接不同字段）。

函数头之后为函数体，这部分需要相对于函数头进行缩进。Python 中的缩进（Indentation）决定了代码的作用域范围，相同缩进行的代码处于同一作用域范围。

注意：不要混合使用制表符和空格来缩进。

混合缩进将会引发 IndentationError 异常。强烈建议每个缩进层次使用单个制表符（Tab）或 4 个空格。更加重要的是，一旦选择一种风格，以后最好只使用这一种风格。

函数体的前三行定义了 3 个变量，由于它们均在函数体中被定义，作用域仅仅在函数体内，无法在函数体外被访问，因此被称为函数的局部变量。最后一行的关键字 return 指明函数的返回值（此处是局部变量 f 的值）。相比 3.2 节使用 print()将计算结果输出到屏幕，return 使得函数调用者可以得到计算的结果。

如果我们的程序仅仅定义了一个函数，则运行结果是什么也没有的。定义的函数如果不被调用，实质上等同于什么都没做。这一点和数学中的函数很类似，当你写下函数 $f(x)$ 时，它仅仅表示一个映射关系，但此处并无具体的映射。只有输入具体值之后，才有输出产生。Python 的主程序通常指源代码中不在特定函数内的每一行代码。运行程序时，只执行主程序中的语句。只有在主程序中包含对函数的调用时，函数定义中的代码才会运行。在前面的章节中，我们已经调用了预先定义的函数，例如 print 等。现在我们也以完全相同的方式调用自己写的一个函数，代码如下：

```
In [81]: x = 0
         g1 = gaussian(x)
         g2 = gaussian(4)
         print(g1, g2)
0.05399096651318806 0.05399096651318806
```

对于 gaussian()的调用将返回一个浮点型对象，在程序运行中 gaussian(x)将被一个浮

点型对象替换。不同于 C/C++、Java 等编程语言，Python 不要求指明函数输入参数和返回值的类型。对于简单的函数，我们可以通过查看代码来确定它们的类型，但是对于复杂的函数，则需要增添相应的注释进一步说明。这些注释文字被称为函数文档或者 docstring。推荐注释格式如下：

```
"""计算正态分布概率密度。

在本函数中，标准差 σ 固定为 1，数学期望 μ 固定为 2。

参数
----
x: float
      随机变量 x 的值。

返回值
-----
g: float
      随机变量的概率密度。
"""
```

这些说明性文字被加在函数头之后，这样便可以使用 help 函数显示出来，代码如下：

```
In [82]: help(gaussian)
计算正态分布概率密度。

在本函数中，标准差 σ 固定为 1，数学期望 μ 固定为 2。

参数
----
x: float
        随机变量 x 的值。

返回值
-----
g: float
        随机变量的概率密度。
```

函数注释需要包含以下几部分：

(1) 头部为简要的说明，通常使用祈使句。如“计算……”“返回……”“做……”。

(2) 接下来是对于函数较为详细的说明。这部分是可选的，对于简单的函数，可以省略。

(3) 第三部分是对于入参的详细说明。

(4) 第四部分为返回值的详细说明。

注释的格式有很多种，本书中将采用 Numpydoc 中的格式规范，因为 Spyder IDE 针对这种规范在显示上有特别的处理。可以在 Spyder IDE 的编辑窗或者 Console 输入以上代码，将光标置于函数名上，按 Ctrl+I 快捷键。此时 Help 窗口的输出如图 3-1 所示。

图 3-1　函数注释

也可以在 Help 窗口的 Object 框中输入函数名查询。此时需要根据函数是在编辑窗还是 Console 里定义的，在 Source 框里做不同的选择。

3.4　括号匹配

注意：对于像本例中如此复杂的公式，括号的使用很容易出错。这样的错误通常会导致一个指向下一行的错误消息。这种错误信息会令新手感到非常困惑，因此，如果你获得的错误消息直接指向复杂数学公式下面一行，则表示错误通常是在公式本身。

很多编辑器具有语法分析的功能，能够实时检测括号的配对情况。在 Spyder 的编辑器中，将光标放置在右括号之后，该右括号及与之配对的左括号会自动高亮显示。如果当前行中存在括号失配，则会出现错误警示。如图 3-2 所示，第 29 行匹配的两个括号被高亮显示。由于第 29 行的括号目前处于失配状态，所以行首有错误警示标识。

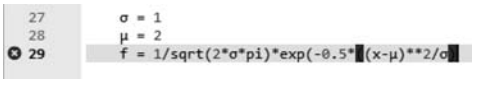

图 3-2　括号匹配高亮显示及失配告警

如果这个错误在编辑时被忽略了，则 Spyder 编辑器会认为出现超长代码换行的情况，光标不会自动出现在下一行的对齐处。参看如图 3-3 所示的光标位置，而新增的第 30 行的行首也被自动加上了错误警示标识。当遇到这种错误警示时，需仔细查看前一行的代码。

如果编辑第 30 行时忽略了缩进的异常，则在行首的错误警示标识会一直存在，虽然第 30 行不存在任何问题，如图 3-4 所示。在用 Spyder 打开已经编写好的程序时，如果发现这种错误，则需查看前一行是否存在括号失配的情况。

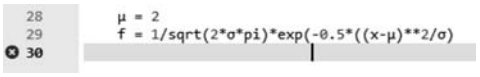

图 3-3　代码换行

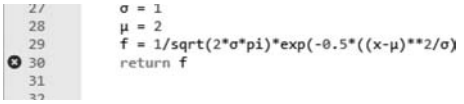

图 3-4　上一行代码括号失配

3.5　入参和局部变量

前文的例子中，期望值 μ 和方差 σ^2 在函数中被赋予了固定值。在数学函数中，这两个值也是可以变化的。也就是说，概率密度是由 μ、σ、x 这 3 个量决定的。Python 的函数可以有多个入参，我们可以将代码做如下修改：

```
In [83]: def gaussian(x, μ, σ):
             """计算正态分布概率密度。

             参数
             ----
             x: float
                随机变量 x 的值。
             σ:float
                标准差
             μ: float
                数学期望
             返回值
             -----
             g: float
                    随机变量的概率密度。
             """

             f = 1/sqrt(2 * σ * pi) * exp( - 0.5 * ((x - μ) ** 2/σ))
             return f

In [84]: gaussian(14, 0, 1)
Out[84]: 1.0966065593889713e - 43

In [85]: gaussian(14, 0, 0.1)
Out[85]: 0.0

In [86]: gaussian(14, 1, 0.5)
Out[86]: 2.2680760989571468e - 74
```

函数定义中用于描述预期参数的标识符被称为形式参数，而调用函数时由调用者发送的对象是实际参数。Python 中的参数传递遵循赋值语句的语义。当函数被调用时，函数调用者将实际参数赋值给该函数的每个形式参数。

以调用 gaussian(14，0，1)为例，在函数执行之前，形式参数分别被赋值如下：

(1) $x=14$。

(2) $\mu=0$。

(3) $\sigma=1$。

函数在使用 return 返回时,同样会进行一个类似的赋值过程。在 gaussian()内,我们创建了一个对象并称为 f。函数返回时,这个对象会和调用者作用域中的一个变量名关联。

3.5.1 参数默认值

Python 支持函数的多态(Polymorphic),这样的函数支持一种以上的调用方式。最值得注意的是,函数可以声明一个或多个参数的默认值,从而允许调用者在调用时传入具有变化数量的实际参数。这样的函数通常具有以下定义:

```
def foo(a, b = 15, c = 27):
```

该函数有 3 个参数,最后 2 个提供了默认值。调用者可以传入 3 个实际参数,代码如下:

```
foo(4,12,8)
```

此种情况下,函数执行时不使用默认值。如果调用者只发送 1 个参数,则代码如下:

```
foo(4)
```

函数执行时,$a=4$、$b=15$、$c=27$。如果调用者传入 2 个参数,假设它们是前 2 个,则第 3 个是默认值,代码如下:

```
foo(8, 20)
```

函数执行时,$a=8$、$b=20$、$c=27$。

默认参数必须处于函数入参列表的尾部,不能出现在列表中部。如下定义是非法的:

```
bar(a, b = 15, c)
```

该定义中入参 b 有一个默认值,其后所有参数也必须有默认参数值。

现在将之前的函数再次修改,代码如下:

```
In [87]: def gaussian(x, μ = 0, σ = 1):
             """计算正态分布概率密度。

             参数
             ----
             x: float
                    随机变量 x 的值。
             σ:float
                标准差, 默认值为 1。
```

```
    μ: float
        数学期望，默认值 0。

    返回值
    -----
    g: float
       随机变量的概率密度。
    """

    g = 1/sqrt(2 * σ * pi) * exp( - 0.5 * ((x - μ) ** 2/σ))
    return g
```

3.5.2　关键字参数

现在对于带有默认参数的函数分别进行三次调用：使用默认参数、修改第 2 个参数值、修改 2 个参数值，代码如下：

```
In [88]: gaussian(0.1)
Out[88]: 0.3969525474770118

In [89]: gaussian(0.1, - 2)
Out[89]: 0.04398359598042719

In [90]: gaussian(0.1, 1, 2.4)
Out[90]: 0.15493962244573706
```

但是这里有一个问题，如何只修改第 3 个参数的值呢？传统的参数传入机制是依次匹配调用者传入的实际参数，在这种情况下，实参需要和形参在位置上一一对应，因此，如果我们只想修改第 3 个参数的默认值，则须传入第 2 个参数的默认值，代码如下：

```
In [91]: gaussian(0.1, 0, 2.4)
Out[91]: 0.1660817194169522
```

Python 支持另一种机制——关键字参数(Keyword Argument)。关键字参数传入的方式是显式指定形式参数。示例代码如下：

```
In [92]: gaussian(0.1, σ = 2.4)
Out[92]: 0.1660817194169522
```

此时，仅第 3 个参数的默认值被改变了。因为调用时并未给第 2 个参数赋值。

3.5.3　局部变量和全局变量

在编程中，必须理解局部变量和全局变量之间的区别。一旦将两者混淆，便会为程序引入不可预知的问题。如前文所述，传递给函数的参数及我们在函数内部定义的变量都是局部变量。这些变量仅在函数内可见，其作用域也仅在函数内，但是全局变量的作用域覆盖整

个程序，也可以在函数内部被访问，就像代码中的其他任何地方一样，这样便可能会引入混乱。

我们看以下代码：

```
In [93]: %reset -f
         def gaussian(x):
             σ = 1
             μ = 0
             g = 1/sqrt(2*σ*pi)*exp(-0.5*((x-μ)**2/σ))
         return g
         gaussian(0.1)
         g
Traceback (most recent call last):

  File "<ipython-input-93-30230dfc8314>", line 1, in <module>
    g

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

首先，我们需要使用 IPython 的魔术命令清除当前命名空间中的所有全局变量，因为有可能名为 g 的对象已经存在于当前的全局命名空间。

后续代码在访问 g 时出现错误，原因是变量 g 在函数 gaussian()之外不存在。变量 g 是函数内部定义的局部变量，其作用域仅局限在函数内部，因此无法在函数体外被访问。

现在，我们将 σ 和 μ 的定义移到函数定义外。此时函数仍然能够工作，函数执行时将使用 σ 和 μ 的值，最终能够得到预期的结果，代码如下：

```
In [94]: σ = 1
         μ = 0

         def gaussian(x):
             g = 1/sqrt(2*σ*pi)*exp(-0.5*((x-μ)**2/σ))
             return g
```

但是如果在函数体内也定义了变量 σ 或 μ，则会是什么结果呢？函数被调用时，会使用了哪个 σ 的值呢？

Python 解释器处理同名变量的原则是：局部变量名总是优先于全局变量名，因此，在上面的代码中，Python 解释器在查找公式中出现的变量 σ、μ 和 x 的值时，首先搜索本地命名空间，即在函数 gaussian()内定义的本地变量。一旦找到匹配的本地变量，如代码中的 σ 和 x，则它们的值被使用。如果在本地命名空间中找不到某些变量，则 Python 解释器将移至全局命名空间查找与给定名称匹配的全局变量。如果在全局变量中找到了具有正确名称的变量，则使用相应的值。如果未找到具有正确名称的全局变量，并且没有其他可搜索的位置，则程序便以错误消息结尾。这种对变量的顺序搜索是很自然且合乎逻辑的，但也是造成混乱和编程错误的潜在原因。如果试图在函数内部更改全局变量，则可能还会引起其他混乱。示例代码如下：

```
In [95]: from math import sqrt, pi, exp

         σ = 1                    #全局变量
         μ = 0

         def gaussian(x):
             σ = 5                #新的局部变量
             g = 1/sqrt(2 * σ * pi) * exp( - 0.5 * ((x - μ) ** 2/σ))
             return g

         print(gaussian(0.01),σ)
0.17841062750008155 1
```

虽然函数内部存在语句σ=5,但是在调用函数后全局变量σ的值保持不变。由于出现在函数内部,所以 Python 解释器会将其视为重新定义一个局部变量,而不是试图改变一个全局变量。局部变量的作用域范围仅在函数内部,函数返回后,局部变量将不再存在,而全局变量仍然存在并且具有其原始值。如果要在函数内部更改全局变量的值,则必须使用关键字 global 明确声明。现将代码改动如下:

```
In [96]: from math import sqrt, pi, exp

         global σ = 1 #全局变量
         μ = 0

         def gaussian(x):
             σ = 5 #全局变量σ的值将被修改
             f = 1/sqrt(2 * σ * pi) * exp( - 0.5 * ((x - μ) ** 2/σ))
             return f

         print(gaussian(0.01),σ)
0.17841062750008155 5
```

在这种情况下,全局变量的值发生了变化。关键字 global 告诉 Python 解释器确实希望更改全局变量的值,而不是定义新的局部变量。通常情况下,应尽量减少在函数内部使用全局变量,而应将函数内部使用的所有变量定义为局部变量或以参数的形式传递给函数。如果我们希望更改全局变量的值,则应使用函数返回值来更新它的值,而不要使用关键字 global。很难想到一个示例,其中使用 global 是最佳解决方案。现在,将以上代码更新如下:

```
In [97]: from math import sqrt, pi, exp

         σ = 1
         μ = 0

         def gaussian(x):
             g = 1/sqrt(2 * σ * pi) * exp( - 0.5 * ((x - μ) ** 2/σ))
             return g, 5
```

```
        g, σ = gaussian(0.001)
        print(g,σ)
0.3989420809303424 5
```

需要注意，在这里，从函数返回的是一个元组（Python tuple 类型）。此元组有两个成员，它们之间用逗号隔开，就像在参数列表中一样。函数被调用后，其返回值被分配给全局变量 g 和 σ。本例可能并不具有实际的应用意义，但在许多情况下，需要通过函数调用并更改全局变量。在这种情况下，应始终通过赋值方式执行更改。

注意：将全局变量作为参数传入，从函数中返回变量，然后将返回值赋给全局变量。

遵循这一规范远胜于在函数内部使用全局关键字 global，因为它确保了每个函数都是独立的实体，并通过定义明确参数列表、返回值和其他代码接口。

3.6 函数返回值

函数使用 return 语句结束执行。如同前面代码中所示，通常情况下 return 语句是函数体内的最后一条语句。return 语句也可能在函数体中多次出现，此时必然存在一些条件逻辑。在我们的例子中，return 语句带有显式的参数。如果在没有显式参数的情况下执行 return 语句，则将自动返回 None。

注意：None 是 Python 一种 NoneType 的对象。

Python 函数可能同时返回多个值。当有多个返回值时，return 语句中的各个参数之间需要用逗号分隔，接收这些返回值的变量也需要使用逗号分隔。此时函数返回的对象类型是 tuple。第 4 章会详细介绍 tuple 这一对象类型。

3.7 Lambda 表达式

Python 函数的参数可以是任何 Python 对象，甚至是另一个函数。此功能对于许多科学应用程序非常有用。在数学中，很多函数的定义需要使用其他的数学函数。例如求积分 $\int_a^b f(x)\mathrm{d}x$、求导数 $f'(x)$。在这些应用程序中，我们需要将一个函数作为参数传入新定义的函数。

下面的例子是求一个函数在某一点的二阶导数：

$$f''(x) \approx \frac{f(x-h)-2f(x)+f(x+h)}{h^2} \tag{3-5}$$

相应的 Python 代码如下：

```
In [98]: def diff2(f, x, h = 1E - 6):
             '''求给定函数的二阶导数
```

```
        入参
        ----
        f:function
            待求微分的函数
        x:float
            求微分值的点
        h:float
            距离微分点的增量,默认值为 1e-6

        返回值
        ------
        d2: float
            函数的二阶导数
        '''
            r = (f(x-h) - 2*f(x) + f(x+h))/float(h*h)
            return r
```

函数 diff2 的第一个入参 f 是一个函数。在函数体中,它像普通函数一样被调用。此时需要注意,在调用时必须传入一个函数对象,否则程序运行会出错。因为 diff2 执行到 $f(x-h)$时,需要调用一个函数。

在使用 diff2 之前,有一个函数必须事先定义好,这样才可以作为 diff2 的一个参数。示例代码如下:

```
In [99]: def f(x):
                return x**2 - 1
         print(diff2(f, 1.5))
1.999733711954832
```

Lambda 函数提供了一种便捷的方式来定义函数。Lambda 是一个小型的匿名函数,与常规的 Python 函数相比,它受更严格但更简洁的语法约束。一个最简单的 Lambda 函数的代码如下:

```
In [100]: lambda x: x
Out[100]: <function __main__.<lambda>(x)>
```

在上面的示例中,表达式由以下内容组成:

(1) 关键字:lambda。

(2) 绑定变量:x。

(3) 函数体:x。

Lambda 函数的语法规范如下:

```
somefunc = lambda a1, a2, some_expression
```

因为 Lambda 函数是一个表达式,所以可以命名它。它等同于以下代码:

```
def somefunc(a1, a2, ...):
    return some_expression
```

Lambda 函数中的参数无须用括号包围，多个入参之间使用逗号分隔。

在我们这个例子中，使用 Lambda 函数可以使代码更加紧凑，以下代码仅使用一行代码即完成了之前的函数定义：

```
In [101]: f = lambda x: x ** 2 - 1
```

以上的例子，可能还难于表明 Lambda 函数的全部优势，毕竟它只是将两行代码简化为一行。现在参看下面的代码：

```
In [102]: print(diff2(lambda x: x ** 2 - 1, 1.5))
1.999733711954832
```

在需要将一个简单的数学表达式作为函数入参传递时，以这种方式使用 Lambda 函数是非常方便的。它不仅节省了一些输入，还可以提高代码的可读性，此时，我们不需要再去看某一个函数的定义。

也可以将函数及其参数括在括号中，代码如下：

```
In [103]: (lambda x: x ** 2 - 1)(1.5)
Out[103]: 1.25
```

约简是一种 Lambda 演算策略，用于计算表达式的值。在当前示例中，它用参数 1.5 替换绑定变量 x。

就像使用 def 定义的普通函数对象一样，Python Lambda 表达式支持所有不同的参数传递方式，包括：

(1) 位置参数。

(2) 关键字参数。

(3) 变量参数列表(通常称为 varargs)。

(4) 关键字参数变量列表。

(5) 仅关键字参数。

以下示例演示了向 Lambda 表达式传递参数的不同方式：

```
In [104]: (lambda x, y, z: x + y + z)(1, 2, 3)       #位置参数
Out[104]: 6

In [105]: (lambda x, y, z=3: x + y + z)(1, 2)        #默认参数
Out[105]: 6

In [106]: (lambda x, y, z=3: x + y + z)(1, y=2)      #关键字参数
Out[106]: 6

In [107]: (lambda *args: sum(args))(1,2,3)           #变量参数列表
```

```
Out[107]: 6

In [108]: (lambda **kwargs: sum(kwargs.values()))(one = 1, two = 2, three = 3)
Out[108]: 6

In [109]: (lambda x, *, y = 0, z = 0: x + y + z)(1, y = 2, z = 3)
Out[109]: 6
```

3.8　条件分支

计算机程序的流程常常需要分支。也就是说，如果符合条件，就做一件事。如果不符合此条件，就做另一件事。一个简单的例子是单位阶跃函数：

$$H(n)=\begin{cases}0, & n<0\\ 1, & n\geqslant 0\end{cases} \tag{3-6}$$

单位阶跃函数的 Python 实现检查输入参数 n 的值，并根据 n 的值选择执行不同的语句。以下是公式(3-6)的代码实现：

```
In [110]: def h1(n):
              """单位阶跃函数。

              参数
              ----
              n: float
                  -∞～ ∞

              返回值
              -----
              h: float
                1,n >= 0
                0,otherwise
              """
              if n >= 0:
                  value = 1
              else:
                  value = 0
              return value
```

条件语句基于一个或多个布尔表达式。Python 解释器首先会对布尔表达式进行评估，然后根据评估的结果选定并执行相应的代码块。在 Python 中，条件语句的形式如下：

```
if first_condition:
    first_block
elif second_condition:
    second_block
elif third_condition:
    third_block
else:
    fourth_block
```

每个条件都是一个布尔表达式，每个代码块都包含一个或多个条件执行语句。如果第一个条件成立，则将执行第一个代码块。如果第一个条件不成立，则以类似的方式继续对第二个条件进行评估。最后，从所有待选分支中选中并执行一个代码块。可能有任意多个 elif 子句(包括 0)，而最后的 else 子句也是可选的。

与函数定义类似，每个条件判断语句以冒号“:”结尾，其后的条件代码块需保持相同的缩进。

对于阶跃函数的另一种定义如下：

$$H(n)=\begin{cases}1, & n>0\\ \frac{1}{2}, & n=0\\ 0, & n<0\end{cases} \tag{3-7}$$

Python 函数实现的代码如下：

```
In [111]: def h2(n):
              """阶跃函数 2

              参数
              ----
              n: float
                 - ∞ ~ ∞

              返回值
              -----
              h: float
                 1,n > 0
                 0.5, n = 0
                 0,otherwise
              """
              if n > 0:
                 value = 1
              elif n == 0:
                 value = 0.5
              else:
                 value = 0
              return value
```

我们还可以如下编写代码：

```
In [112]: def h3(n):
                """阶跃函数 3
                h2()的另一种实现

                参数
                ----
                n: float
                   - ∞ ~ ∞
```

```
                返回值
                -----
                h: float
                   1,n > 0
                   0.5, n = 0
                   0,otherwise
                """
                if n >= 0:
                   if n == 0:
                         value = 0.5
                   else:
                         value = 1
                else:
                   value = 0
                return value
```

此时，在 $n \geqslant 0$ 的条件下，我们又嵌入了一个条件分支。

非布尔类型可以被评估为具有直观含义的布尔值。例如，如果 response 是用户输入的字符串，并且希望以非空字符串来限制行为，则可以编写如下代码：

```
if response:
    do_calculate()
```

以上代码等同于如下代码：

```
if response != '':
    do_calculate()
```

对于简单的条件判断，可以将其嵌入特定的语句中。其格式如下：

```
variable = (value1 if condition else value2)
```

采用这种方式，可以将 h1()改写为如下代码：

```
In [113]: def h3(n):
                return (1 if x >= 0 else 0)
```

3.9　程序验证

在 3.3 节中，我们提到了将程序中功能明确的代码编写成独立的函数，这样便于程序正确性的验证。本节将介绍如何编写测试代码以验证某一函数的功能是否能够按预期想法工作，这一编程方法对开发出功能正确的程序非常有效。

尽管需要花费额外的时间来编写测试用例，但由于可及早发现错误，因此通常可以为后期调试节省更多时间。该过程通常被称为单元测试，因为每个测试都会只验证程序的一小

部分是否按预期工作。许多编程者甚至会更进一步，他们会在编写实际功能之前编写测试用例。这种方法通常被称为测试驱动开发(Test-Driven Development，TDD)，它是越来越受欢迎的软件开发方法。

3.9.1 编写测试函数

我们将用于测试功能的代码编写为函数，它是一种专门用于测试的特殊类型的函数，但代码在实际运行时不会用到它们。编写好的测试函数是一个具有挑战性的工作，因为这个函数需要以一种可靠的方式测试已有代码的功能。测试函数的整体思想非常简单。对于通常需要一个或多个参数的给定函数，我们选择一些参数并手工计算函数的运行结果。在测试函数内部，我们只需使用正确的参数调用函数，然后将函数返回的结果与预期的(手工计算得到)结果进行比较。下面的示例说明了如何编写测试函数来测试函数 double(x)是否可以正常工作，代码如下：

```
In [114]: def double(x):
              """将输入加倍

              一个测试用例。

              参数
              ----
              x: float
                  被加倍的数。

              返回值
              -----
              2x: float
                  输入的两倍。
              """
              return 2 * x

          def test_double():
              """函数 double()的测试函数。

              参数
              ----
                   无

              返回值
              -----
                   无
              """
              x = 4                    # 函数入参
              expected = 8             # 期待函数输出
              computed = double(x)
              success = computed == expected
              msg = f'computed {computed}, expected {expected}'
              assert success, msg
```

这段代码中，函数 test_double()没有使用 return 返回，测试函数通常不应返回任何内容，因为无论返回对或错都无太大意义。测试函数的唯一目的是发现被测函数中潜在的错误，一旦发现被测试函数的返回值与我们的期望值不同，则需要立即将错误显示出来，因此代码中使用了 Python 关键字 assert(断言)。断言用于判断一个条件表达式是否为 True，当条件表达式为 False 时触发异常。断言语句的基本格式如下：

```
assert <test>, <message>
```

其中 test 是一个条件表达式，message 是一段字符串。其在逻辑上等同于以下代码：

```
if not test:
    raise AssertionError(message)
```

在示例的测试代码中，我们将期望值与返回的计算结果进行比较。这个布尔表达式的返回值将为 True 或 False，然后将此值赋给变量 success，变量 success 在断言中被用作条件表达式。断言语句中的字符串 message 是对错误原因进行说明的文字，在断言被触发时将被输出。如果断言语句中没有这一项，则在断言被触发时，仅有通用的一条消息 assertion error 被输出。为了使测试结果更加明确，一定要向断言语句中添加明确的错误信息。

可以在一个测试函数中添加多个断言。这对于使用不同的参数测试同一个函数会很有用。例如，如果为 3.8 节的单位阶跃函数编写一个测试函数，则自然会测试定义该函数的所有单独间隔。以下代码说明了如何完成此操作：

```
In [115]: def test_h1():
              """
              函数 h1(n)的测试函数

              参数
              ----
                     无

              返回值
              -----
                     无

              """
              x1, exp1 = 2, 1
              x2, exp2 = 0, 1
              x3, exp3 = -1, 0

              assert h1(x1) == exp1, '输入大于 0 时结果错误 %d' %(h1(x1))
              assert h1(x2) == exp2, '输入等于 0 时结果错误 %d' %(h1(x2))
              assert h1(x3) == exp3, '输入小于 0 时结果错误 %d' %(h1(x3))

          test_h1()
```

函数 test_h1()执行成功，未引发异常，因而函数 h1()的实现通过了测试。

编写测试函数时，应遵守如下规则：

(1) 测试函数必须至少具有一个形如 assert success，<msg>的断言语句。其中success是布尔变量或表达式，如果测试通过则为True，否则为False。如果需要，则可以包含多个断言。

(2) 测试函数不应使用任何输入参数。应将测试过程中使用的所有参数定义为测试函数内部的局部变量。

(3) 函数的名称应始终为test_，后面跟要测试的函数的名称。遵循此约定很有用，因为它使任何阅读代码的人都可以明显看出该函数是一个测试函数。这一命名方式的另一好处是：这是某些测试工具的函数命名约定，因为它们会自动调用这种函数对代码进行自动测试。

用于科学计算的Python函数执行的是某种数学函数的功能，它返回数值或数值列表/元组。针对它们的测试函数在结构上通常都比较固定。遵循以上这些规则，并记住测试函数只是将被测函数的返回值与预期结果进行比较，这样编写测试函数就不是一件复杂的事情了。

遵循上面定义的测试函数命名约定的一个优点是，有一些工具可用于自动运行文件或文件夹中的所有测试函数，并报告是否有任何Bug潜入代码。在多人共同参与的大型开发项目中，使用这样的自动化测试工具是必不可少的。即使是个人开发和维护的项目，使用这样的工具也是有益处的。

3.9.2 使用 pytest

这里向大家推荐pytest，其官方网站为https://docs.pytest.org/en/stable/index.html。

如果我们向它传递一个文件名，pytest将在这个文件中查找以test_开头的函数，正如上面的命名约定所指定的那样。所有这些函数都将被标识为测试函数并由pytest调用，无论这些函数是否在实际应用中被调用。执行之后，pytest将打印一个简短的摘要，说明它找到了多少个测试函数，以及针对通过和失败的统计值。现在我们将3.9.1节的程序稍做修改，将第三处检验的预期改成一个错误的值，运行pytest的结果如下：

```
(base) PS C:\Users\samuel\Documents\用 Python 探索数学书稿\code\ch3>
     pytest .\s2.py
========================== test session starts ==========================
platform win32 -- Python 3.8.3, pytest-6.1.1, py-1.9.0, pluggy-0.13.1
rootdir: C:\Users\samuel\Documents\用 Python 探索数学书稿\code\ch3
collected 2 items

s2.py .F                                                          [100%]

================================ FAILURES ================================
_____________________________________ test_h1 _____________________________________

    def test_h1():
        x1, exp1 = 2, 1
```

```
        x2, exp2 = 0, 1
        x3, exp3 = -1, -1

        assert h1(x1) == exp1, '输入大于 0 时结果错误 %d' % (h1(x1))
        assert h1(x2) == exp2, '输入等于 0 时结果错误 %d' % (h1(x2))
>       assert h1(x3) == exp3, '输入小于 0 时结果错误 %d' % (h1(x3))
E       AssertionError: 输入小于 0 时结果错误 0
E       assert 0 == -1
E        + where 0 = h1(-1)

s2.py:33: AssertionError
============================ warnings summary ============================
..\..\..\..\..\..\..\..\programdata\anaconda3\lib\site-
packages\pyreadline\py3k_compat.py:8
  c:\programdata\anaconda3\lib\site-packages\pyreadline\py3k_compat.py:8:
DeprecationWarning: Using or importing the ABCs from 'collections' instead of from 'collections.
abc' is deprecated since Python 3.3, and in 3.9 it will stop working
    return isinstance(x, collections.Callable)

-- Docs: https://docs.pytest.org/en/stable/warnings.html
====================== short test summary info ======================
FAILED s2.py::test_h1 - AssertionError: 输入小于 0 时结果错误 0
================ 1 failed, 1 passed, 1 warning in 0.19s ================
(base) PS C:\Users\samuel\Documents\用 Python 探索数学书稿\code\ch3>
```

对于规模较大的软件项目，通常将目录名作为 pytest 的运行参数。在这种情况下，该工具将在给定目录及其所有子目录中搜索文件名以 test 开头或结尾（例如，test_math.py，math_test.py 等）的 Python 文件，并在所有这些文件中搜索。所有符合测试函数命名约定的函数（以 test_作为函数名前缀）都被其视为测试函数，pytest 一旦检测到这种函数将按上述方式执行它。大型软件项目通常具有数千个测试函数，将它们收集到单独的文件中并使用自动工具（如 pytest）会非常高效。当然，对于本书中编写的小程序，将测试函数与要测试的函数写在同一文件中则比较方便。

重点是，我们直接运行测试函数时，如果测试通过，则测试函数会静默运行。因为只有在存在断言错误的情况下，测试函数才会有输出。这可能会造成混淆，有时甚至会怀疑是否调用了该测试函数。首次编写测试函数时，在该函数内部包含一个打印语句可能会很有用，只需验证该函数是否已被实际调用。一旦我们知道函数可以正常工作并且习惯了测试函数的工作方式，就应该删除该语句。

3.10　本章小结

本章详细介绍了如下内容：

（1）如何从别的模块导入并使用库函数。

（2）如何自定义函数。

（3）如何为函数编写格式良好的注释。

(4) 局部变量和全局变量的差别。

(5) 如何编写条件判读语句。

在第 4 章中,我们将学习如何使用循环语句使程序能够更加高效地运行。

3.11 练习

练习 1:

编写一个用来计算长方体体积的函数,假设长方体的长、宽和高分别是 a、b 和 c,并使用以下值验证函数是否正确。

(1) $a=1, b=1, c=1$。

(2) $a=1, b=2, c=3.5$。

(3) $a=0, b=1, c=1$。

(4) $a=2, b=-1, c=1$。

练习 2:

假设三角形的三条边的长度分别是 a、b 和 c。编写一个用来计算三角形面积的函数程序并验证任意三角形的面积可由公式(3-8)求得。

$$A=\sqrt{s(s-a)(s-b)(s-c)}, \quad s=\frac{a+b+c}{2} \tag{3-8}$$

使用以下数据验证:

(1) $a=1, b=1, c=1$。

(2) $a=3, b=4, c=5$。

(3) $a=7, b=8, c=9$。

(4) $a=2, b=-1, c=1$。

练习 3:

编写一个函数程序计算物体从高度 H(单位:m)处下落所需要的时间 t,并用以下数据验证。

(1) $H=1$ m($t\approx 0.452$ s)。

(2) $H=10$ m($t\approx 1.428$ s)。

(3) $H=0$ m($t=0$ s)。

(4) $H=-1$ m(思考合理的解)。

练习 4:

编写程序计算若干年后某人银行账户内的资产总额,函数如下:

$$f(n)=A\left(1+\frac{p}{100}\right)^n \tag{3-9}$$

p 为银行一年定期的存款利率,A 为某人存入的初始资金,n 为存款年限。

练习 5:

开普勒第三定律指出,绕以某天体为焦点的椭圆轨道运行的所有行星或卫星,其各自椭圆轨道半长轴(r)的立方与周期(T)的平方之比是一个常量。

$$\left(\frac{T_A}{T_B}\right)^2=\left(\frac{r_A}{r_B}\right)^3 \tag{3-10}$$

伽利略用木星直径作为度量单位测量了木星卫星的轨道大小。他发现：木星卫星一是离木星最近的卫星，它离木星 4.2 个单位长度，其公转周期为 1.8 天。木星卫星四的公转周期是 16.7 天，编写一个函数程序计算木星卫星四距离木星的距离（使用与木星卫星一相同的距离单位）。

练习 6：

给定二次方程如下：

$$ax^2 + bx + c = 0 \quad (a \neq 0)$$

其两个根为

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{3-11}$$

编写一个程序求任意二次方程的实数解。

注意：不是所有二次方程都有实数解。
