

## 8

## Functions in Python

## Python 函数

内置函数、自定义函数、Lambda 函数 ...



很多人在二十五岁便垂垂老矣，直到七十五岁才入土为安。

*Many people die at twenty five and aren't buried until they are seventy five.*

—— 本杰明·富兰克林 (Benjamin Franklin) | 美国政治家 | 1706 ~ 1790



- ◀ `numpy.linalg.det` 计算一个方阵的行列式
- ◀ `numpy.linalg.inv` 计算一个方阵的逆矩阵
- ◀ `numpy.linalg.eig` 计算一个方阵的特征值和特征向量
- ◀ `numpy.linalg.svd` 计算一个矩阵的奇异值分解
- ◀ `numpy.random.rand` 生成 0~1 之间均匀分布的随机数
- ◀ `numpy.random.randn` 生成符合标准正态分布的随机数
- ◀ `numpy.random.randint` 生成指定范围内的整数随机数
- ◀ `def ... (return ...)` Python 中用于定义函数的关键字，其中 `def` 用于定义函数名称和参数列表，`return` 用于指定函数返回的结果，可以没有函数返回
- ◀ `matplotlib.pyplot.grid()` 在当前图表中添加网格线
- ◀ `matplotlib.pyplot.plot()` 绘制折线图
- ◀ `matplotlib.pyplot.subplots()` 创建一个包含多个子图的图表，返回一个包含图表对象和子图对象的元组
- ◀ `matplotlib.pyplot.title()` 设置当前图表的标题
- ◀ `matplotlib.pyplot.xlabel()` 设置当前图表 x 轴的标签
- ◀ `matplotlib.pyplot.xlim()` 设置当前图表 x 轴显示范围
- ◀ `matplotlib.pyplot.xticks()` 设置当前图表 x 轴刻度位置
- ◀ `matplotlib.pyplot.ylabel()` 设置当前图表 y 轴的标签
- ◀ `matplotlib.pyplot.ylim()` 设置当前图表 y 轴显示范围
- ◀ `matplotlib.pyplot.yticks()` 设置当前图表 y 轴刻度位置
- ◀ `numpy.linspace()` 用于在指定的范围内创建等间隔的一维数组，可以指定数组的长度
- ◀ `numpy.sin()` 用于计算给定弧度数组中每个元素的正弦值
- ◀ `lambda` 创建匿名函数（没有函数名）的关键字，通常用于简单的函数定义或作为函数的参数传递
- ◀ `map()` 内置函数，用于对一个可迭代对象中的每个元素应用指定的函数，并返回一个新的可迭代对象



## 8.1 什么是 Python 函数?

这本书学到这里，相信大家对函数这个概念已经不陌生。简单来说，在 Python 中，函数是一段可重复使用的代码块，用于执行特定任务或完成特定操作。函数可以接受输入参数，并且可以返回具体值、或者不返回任何值作为结果。

比如，大家已经非常熟悉的 `print()`，这个函数的输入参数是要打印的字符串，在完成打印之后，这个函数并没有任何的输出值。

再举个几例子，很多函数都返回具体值，比如 `len()` 返回 `list` 元素个数，`range()` 生成一个可以用在 `for` 循环的整数序列，`list()` 可以创建列表或将其他对转化为列表。

再者，很多数值操作、科学计算的函数都打包在 NumPy、SciPy 这样的库中，比如大家已经见过的 `numpy.array()` 等等。

通过使用函数，可以将代码分解成小块，每个块都完成一个特定的任务。这使得代码更易于理解、测试和维护。同时，函数也可以在不同的上下文中重复使用，提高代码的重用性和可维护性。



### 代数角度，什么是函数？

从代数角度来看，函数是一种数学概念，描述了输入和输出之间的关系。它将一个集合中的每个元素映射到另一个集合中的唯一元素。函数用公式、图表或描述性语言定义，具有定义域和值域的概念。函数在数学中被用于解决问题、建模现实世界，并具有单值性、唯一性等特性。代数中的函数描述了数学方程、曲线和变换，并帮助我们理解数学关系及其应用。

### 几种函数类型

在 Python 中，有以下几种函数类型：

- ▶ 内置函数：Python 解释器提供的函数，例如 `print()`、`len()`、`range()` 等。
- ▶ 自定义函数：由用户定义的函数。
- ▶ Lambda 函数：也称为匿名函数，是一种简单的函数形式，可以通过 `lambda` 关键字定义。
- ▶ 生成器函数：是一种特殊的函数，用于生成一个迭代器，可以使用 `yield` 关键字定义。本章不展开介绍生成器函数。
- ▶ 方法：是与对象相关联的函数，可以使用 `"."` 符号调用。例如字符串类型的方法，可以使用字符串变量名.方法名()的形式调用。大家会在 Pandas 中经常看到这种用法。

### 为什么需要自定义函数？

既然 NumPy、SciPy、SymPy 等等库中提供大量可重复利用的函数，为什么还要兴师动众“自定义函数”？

这个答案其实很简单。现成的函数面向一般需求，不能满足大家的各种“私人订制”需求。

此外，自定义函数在 Python 中的作用是提高代码复用性、模块化和组织性，抽象和封装复杂问题，使代码结构和逻辑更清晰，增加可扩展性和灵活性。通过封装可重复使用的代码块为函数，避免重复编写相同的代码，并将大型任务分解为小型函数，使程序更易理解和维护。自定义函数提高代码的可读性、可维护性，并支持程序扩展和修改，使代码更结构化和可管理。

## 包、模块、函数

在 Python 中，一个包（package）是一组相关模块（module）的集合，一个模块是包含 Python 定义和语句的文件。而一个函数则是在模块或者在包中定义的可重用代码块，用于执行特定任务或计算特定值。

通常情况下，一个模块通常是一个 .py 文件，包含了多个函数和类等定义。一个包则是一个包含了多个模块的目录，通常还包括一个特殊的 \_\_init\_\_.py 文件，用于初始化该包。

在使用时，需要使用 import 关键字导入模块或者包，从而可以使用其中定义的函数和类等。而函数则是模块或包中定义的一段可重用的代码块，用于完成特定的功能。

因此，包中可以包含多个模块，模块中可以包含多个函数，而函数是模块和包中的可重用代码块。

以 NumPy 为例，NumPy 是 Python 中用于科学计算的一个库，其包含了很多有用的数值计算函数和数据结构。下面是 NumPy 库中常见的模块和函数的介绍：

numpy.linalg 这个模块提供了一些线性代数相关的函数，包括矩阵分解、行列式计算、特征值和特征向量计算等。常见的函数有：

- ▶ numpy.linalg.det: 计算一个方阵的行列式。
- ▶ numpy.linalg.inv: 计算一个方阵的逆矩阵。
- ▶ numpy.linalg.eig: 计算一个方阵的特征值和特征向量。
- ▶ numpy.linalg.svd: 计算一个矩阵的奇异值分解。

numpy.random 这个模块提供了随机数生成的函数，包括生成服从不同分布的随机数。常见的函数有：

- ▶ numpy.random.rand: 生成 0~1 之间均匀分布的随机数。
- ▶ numpy.random.randn: 生成符合标准正态分布的随机数。
- ▶ numpy.random.randint: 生成指定范围内的整数随机数。

## 数学函数

在代数中，函数是一种数学关系，它将一个或多个输入值映射 (mapping) 到唯一的输出值。函数可以用一个规则或方程式来表示，其中输入值称为自变量，输出值称为因变量。

从代数角度来看，函数是一种数学对象，用于描述两个集合之间的关系。一个函数将一个集合中的每个元素（称为输入）映射到另一个集合中的唯一元素（称为输出）。

数学上，函数的定义包括以下要素：

- ▶ 定义域 (domain): 定义域是输入变量可能的取值范围。它是函数的输入集合。
- ▶ 值域 (range): 值域是函数的输出可能的取值范围。它是函数的输出集合。
- ▶ 规则 (rule): 规则定义了输入和输出之间的映射关系。它描述了如何根据给定的输入计算输出。

如图 1 所示，函数也可以有不止一个输入，比如二元函数  $f(x_1, x_2)$  便有 2 个输入。

函数可以用各种方式定义，包括通过公式、算法、图表或描述性语言。它可以是连续的、离散的或混合的，具体取决于输入和输出的集合的性质。

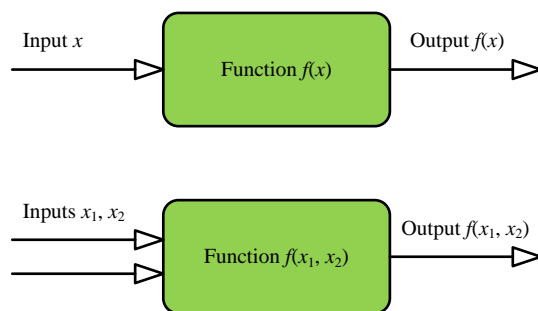


图 1. 一元函数、二元函数的映射

函数描述了不同变量之间的依赖关系，并且可以用来表示数学问题的模型。函数可以通过数学符号、图表或文字描述来表示，它们在代数中广泛应用于方程求解、图形绘制和数值计算等领域。

一句话概括来说，函数就是映射，输入值映射到唯一的输出值。如图 2 所示，我们设计了两个函数：左侧函数 `Shape()` 输入为彩色几何图形，函数输出为图形形状；右侧函数 `Color()` 输入还是彩色几何形状，函数输出为图形颜色。

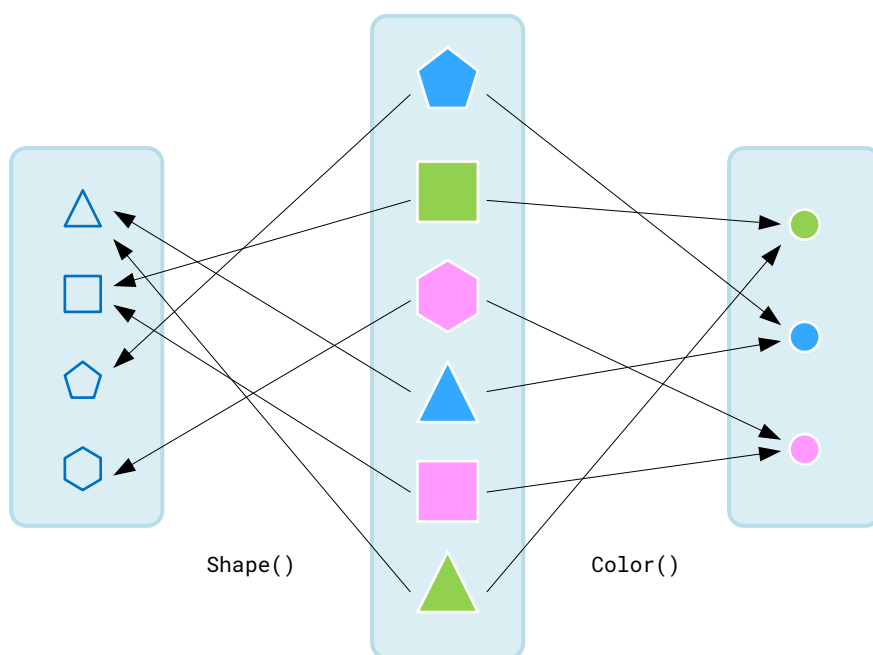


图 2. 识别颜色、形状的函数

### 单射、满射

单射、非单射、满射和非满射是函数映射中的性质，描述了输入值和输出值之间的关系。

单射 (injective) 是指函数中不同的输入值对应着不同的输出值，即每个输出值只有一个对应的输入值。

非单射 (non-injective) 是指函数中存在多个不同的输入值对应着相同的输出值，即至少有一个输出值有多个对应的输入值。

满射 (surjective) 是指函数的所有可能的输出值都能够被映射到，即每个输出值都有至少一个对应的输入值。

非满射 (non-surjective) 是指函数中存在至少一个输出值无法被映射到，即存在某些输出值没有对应的输入值。

图 3 所示为单射、非单射、满射、非满射构成的“四象限”。单射、非单射更关注输入值，而满射、非满射则关注输出值。同时满足单射与满射叫双射 (bijective)，也叫一一映射。

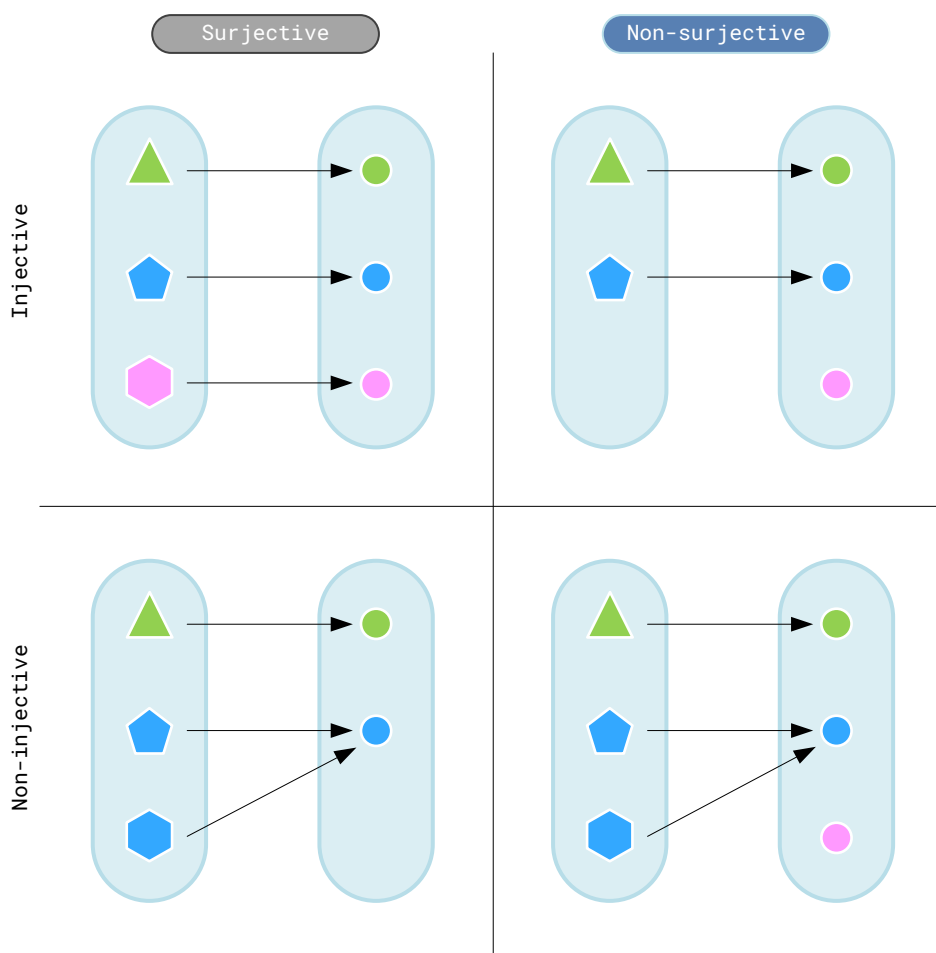


图 3. 单射、非单射、满射、非满射构成的四象限

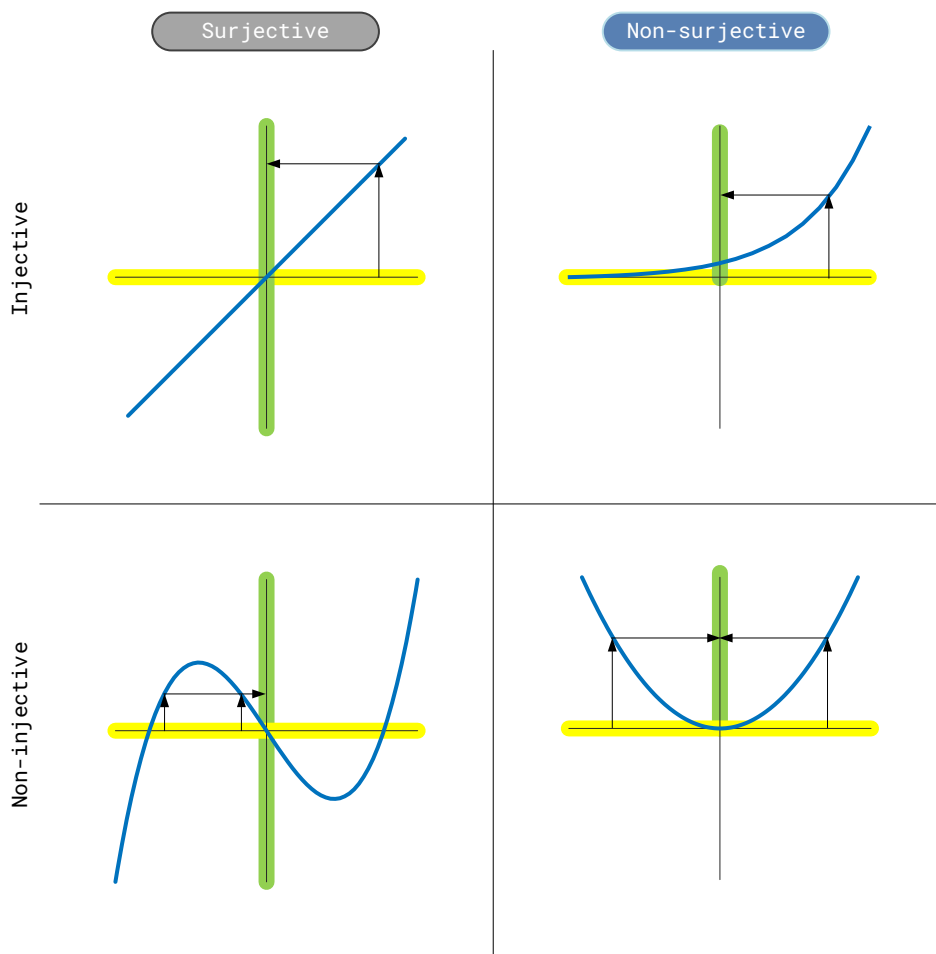


图 4. 单射、非单射、满射、非满射构成的四象限，具体实例

## 一元、二元、三元、多元

在数学中，函数的元 (arity) 指的是函数接受的参数个数。

常见的函数元数包括：

一元函数 (unary function) 接受一个参数。例如， $f_1(x) = x$  是一个一元函数，它接受一个参数  $x$ 。

二元函数 (binary function) 接受两个参数。例如， $f_2(x_1, x_2) = x_1 + x_2$  是一个二元函数，它接受两个参数  $x_1$  和  $x_2$ 。

三元函数 (ternary function) 接受三个参数。例如， $f_3(x_1, x_2, x_3) = x_1 + x_2 + x_3$  是一个三元函数，它接受三个参数  $x_1$ 、 $x_2$  和  $x_3$ 。

多元函数 ( $n$ -ary function) 接受  $n$  个参数。多元函数的参数个数可以是任意多个，例如  $f_n(x_1, x_2, \dots, x_n) = x_1 + x_2 + \dots + x_n$  是一个多元函数，它接受任意  $n$  个参数  $x_1$ 、 $x_2$ 、 $\dots$ 、 $x_n$ 。

## 数学函数 vs 编程函数

代数角度的函数概念与计算机编程中的函数概念有些相似，但也有一些不同之处。在代数中，函数是描述输入和输出之间关系的抽象概念，而在编程中，函数是可执行的代码块，用于执行特定的任务。然而，两者之间的基本思想都是处理输入并生成输出。

本 PDF 文件为作者草稿，发布目的为方便读者在移动终端学习，终稿内容以清华大学出版社纸质出版物为准。

版权归清华大学出版社所有，请勿商用，引用请注明出处。

代码及 PDF 文件下载：<https://github.com/Visualize-ML>

本书配套微课视频均发布在 B 站——生姜 DrGinger：<https://space.bilibili.com/513194466>

欢迎大家批评指教，本书专属邮箱：[jiang.visualize.ml@gmail.com](mailto:jiang.visualize.ml@gmail.com)

数学上的函数和编程上的函数在概念和应用上存在一些异同之处。

无论是数学上的函数，还是编程上的函数，它们都涉及输入和输出。数学函数接受输入值并产生相应的输出值，而编程函数接受参数但是未必返回结果。

数学上的函数和编程上的函数都有一个定义域和一个规则，描述了如何将输入转换为输出。无论是通过公式、算法还是逻辑操作，函数都定义了输入和输出之间的关系。

无论是数学上的函数，还是编程上的函数的概念都具有可重用性。无论是在数学中还是在编程中，函数可以在多个场景中被多次调用和使用，避免了重复编写相同的代码。

数学上的函数和编程上的函数显然也有很大区别。数学函数通常用符号、公式或描述性语言来表示，如  $f(x) = x^2$ 。而编程函数则以编程语言的语法和结构来定义和表示，如 `def square(x): return x**2`。编程函数可以包含额外的程序控制结构，如条件语句、循环等，以实现更复杂的逻辑和操作。

总体而言，数学上的函数更关注描述数学关系，而编程上的函数更侧重于实现特定的计算或操作。虽然两者有相似的概念，但具体的表示方式、范围和应用场景可能会有所不同。

## 8.2 自定义函数

### 无输入、无返回

在 Python 中，我们可以自定义函数来完成一些特定的任务。函数通常接受输入参数并返回输出结果。但有时我们需要定义一个函数，它既没有输入参数，也不返回任何结果。这种函数被称为没有输入、没有返回值的函数。

定义这种函数的方法和定义其他函数类似，只是在定义函数时省略了输入参数和 `return` 语句。比如下例，这个函数名为 `say_hello`，它不接受任何输入参数，执行函数体中的代码时会输出字符串 `"Hello!"`。

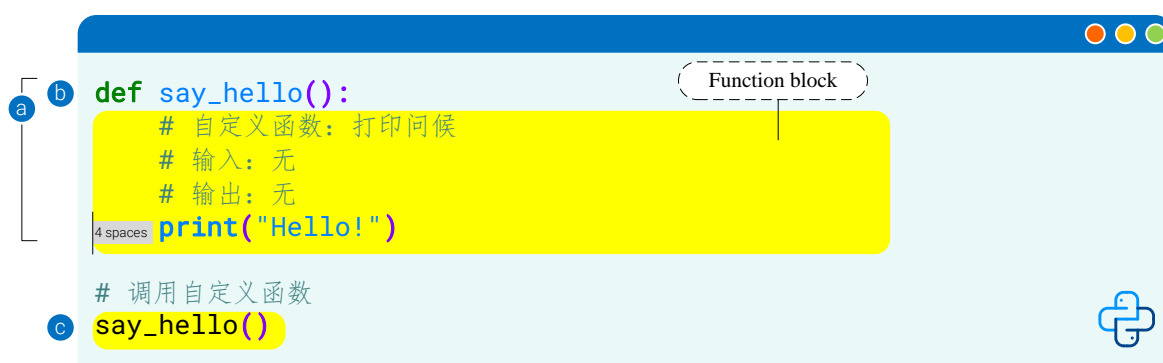


图 5. 无输入、无输出函数

下面，我们再看一个复杂的例子。这个例子，我们也定义了一个无输入、无输出函数用来美化线图。图 6 所示为利用 Matplotlib 绘制的一元一次函数、一元二次函数线图美化之后的结果。

本书第 10 章将专门介绍如何绘制线图，此外鸢尾花书《可视之美》将专门介绍 Python 可视化专题。



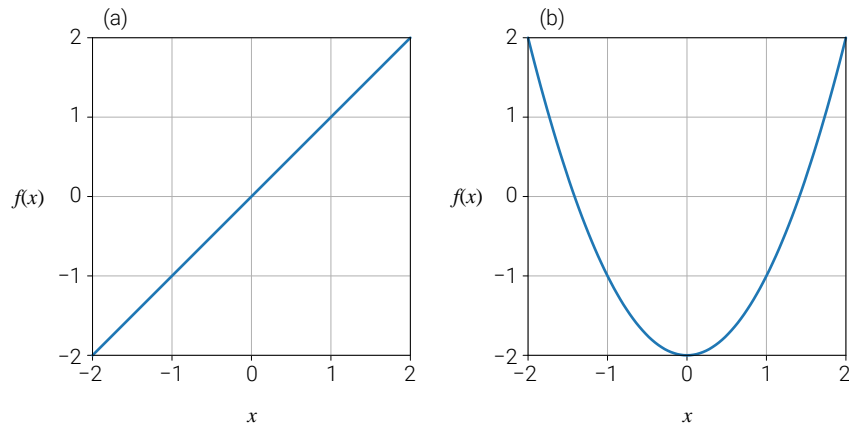


图 6. 绘制线图并美化

```

# 导入包
a import matplotlib.pyplot as plt
# 导入 Matplotlib 库中的 pyplot 模块，并将其命名为 plt
b import numpy as np
# 导入 NumPy 库，并将其命名为 np

# 自定义函数
c def beautify_line_chart():
    # 添加标签
    d plt.xlabel("x")
    e plt.ylabel("f(x)")
    # 设置坐标轴范围
    f plt.xlim(-2, 2)
    g plt.ylim(-2, 2)
    # 设置横纵轴刻度
    h plt.xticks([-2, -1, 0, 1, 2])
    i plt.yticks([-2, -1, 0, 1, 2])
    # 添加网格线
    j plt.grid(True)
    # 横纵轴统一标尺
    k plt.gca().set_aspect('equal', adjustable='box')
    # 显示图形
    l plt.show()

# 绘制直线
m x_array = np.linspace(-2, 2, 101)
# 使用NumPy的linspace函数创建一个包含101个元素的数组
# 这些元素均匀地分布在区间[-2, 2]上，左闭右闭

# 绘制抛物线
n fig, ax = plt.subplots(figsize = (4, 4))
# plt.subplots()返回值解包为两个变量: fig 和 ax
# fig图形窗口对象，可以用于设置图形窗口的属性
# ax 是坐标轴对象，用于绘制具体的图形和设置坐标轴的属性
# figsize=(4, 4) 表示图形窗口的宽度为4英寸，高度为4英寸

y_array = x_array # 一次函数 y = x
plt.plot(x_array, y_array)
beautify_line_chart() # 调用自定义函数绘制美化的线图

y_array = x_array**2 - 2 # 二次函数
plt.plot(x_array, y_array)
beautify_line_chart() # 调用自定义函数绘制美化的线图

```

本 PDF 文件为作者草稿，发布目的为方便读者在移动终端学习，终稿内容以清华大学出版社纸质出版物为准。

版权归清华大学出版社所有，请勿商用，引用请注明出处。

代码及 PDF 文件下载: <https://github.com/Visualize-ML>

本书配套微课视频均发布在 B 站——生姜 DrGinger: <https://space.bilibili.com/513194466>

欢迎大家批评指教，本书专属邮箱: [jiang.visualize.ml@gmail.com](mailto:jiang.visualize.ml@gmail.com)



图 7. 无输入、无输出函数，装饰线图

### 多个输入、单一返回

一个函数可以有多个输入参数，一个或多个返回值。下面是一个示例函数，它有两个输入参数 *a* 和 *b*，返回它们的和。

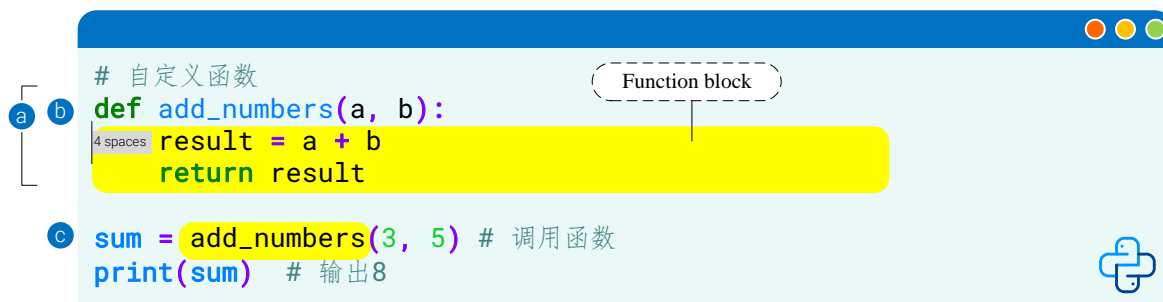


图 8. 两个输入、一个输出函数

下面这个例子中，我们定义了一个名为 `arithmetic_operations()` 的函数，它有两个参数 *a* 和 *b*。在函数体内，我们进行了四个基本的算术运算，并将其结果存储在四个变量中。最后，我们使用 `return` 语句返回这四个变量。当我们调用这个函数时，我们将 *a* 和 *b* 的值作为参数传递给函数，函数将返回四个值。我们将这四个返回值存储在一个元组 `result` 中，并使用索引访问和打印这四个值。

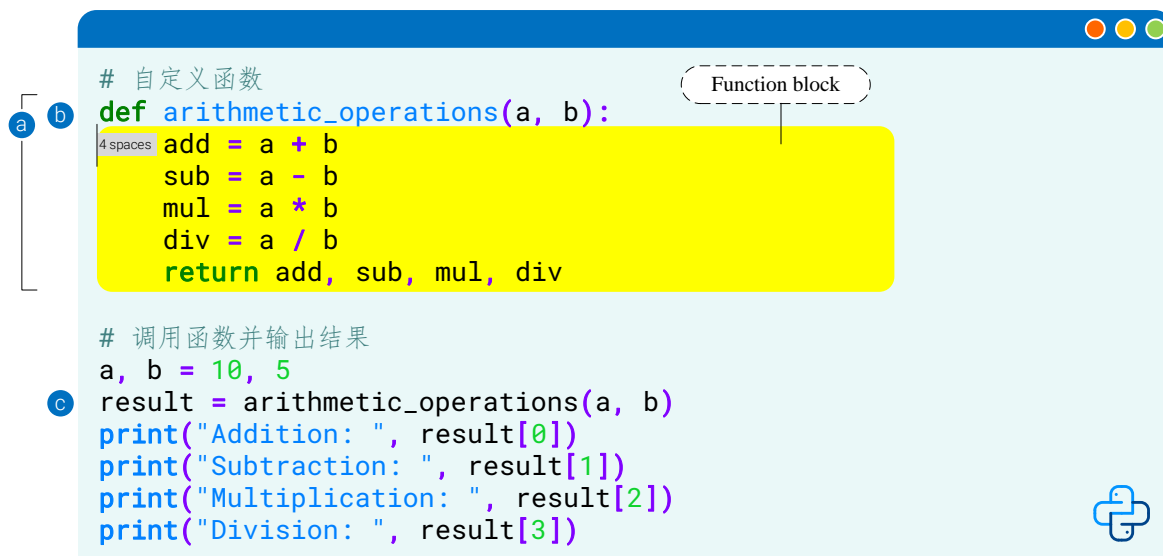


图 9. 两个输入、多个输出函数

### 部分输入有默认值

在 Python 中，我们可以为自定义函数中的某些参数设置默认值，这样在调用函数时，如果不指定这些参数的值，就会使用默认值。这种设置默认值的参数称为默认参数。

下面是一个例子，展示如何在自定义函数中设置默认参数。greet()函数有两个参数：name 和 greeting。name 是必需的参数，没有默认值。而 greeting 是可选的，默认值为'Hello'。

当我们调用 greet()函数时，如果只传入了 name 参数，那么 greeting 就会使用默认值'Hello'。如果需要自定义问候语，可以在调用时传入自定义的值，如上面的第二个调用例子所示。

需要注意的是，默认参数必须放在非默认参数的后面。在函数定义中，先定义参数必须先被传入，后定义参数后被传入。如果违反了这个顺序，Python 解释器就会抛出 SyntaxError 异常。



图 10. 函数输入有默认值

## 全局变量 vs 局部变量

在 Python 中，自定义函数中可以包含全局变量 (global variable) 和局部变量 (local variable)。全局变量是在整个脚本或模块范围内可见的变量，而局部变量只在函数内部可见。

如图 11 所示，全局变量，比如 global\_x，通常在模块的顶部定义，并可以在整个模块中使用。在 my\_function 函数内部定义的 local\_x 是一个局部变量，只能在函数内部访问。局部变量的作用范围仅限于函数内部，一旦函数执行完毕，局部变量就会被销毁。如果尝试在函数外部访问局部变量，将会引发 NameError 错误。

需要注意的是，如果在函数内部使用与全局变量同名的变量，Python 会将其视为一个新的局部变量，而不会修改全局变量的值。如图 11 代码，在函数内部定义了和 global\_x 同名的变量并赋值，这个变量还是局部变量，并不改变外部全局变量的数值。

如图 12 所示，如果要在函数内部修改全局变量的值，需要使用 global 关键字来声明该变量是全局变量。

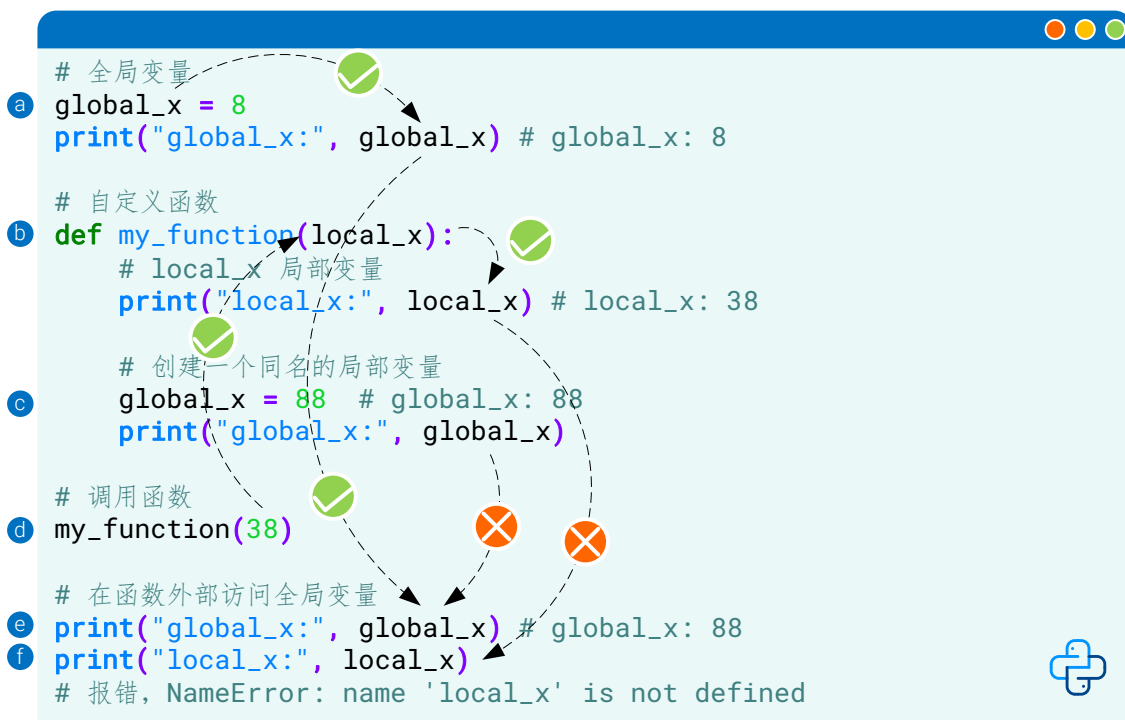


图 11. 全局变量和局部变量的传递路径

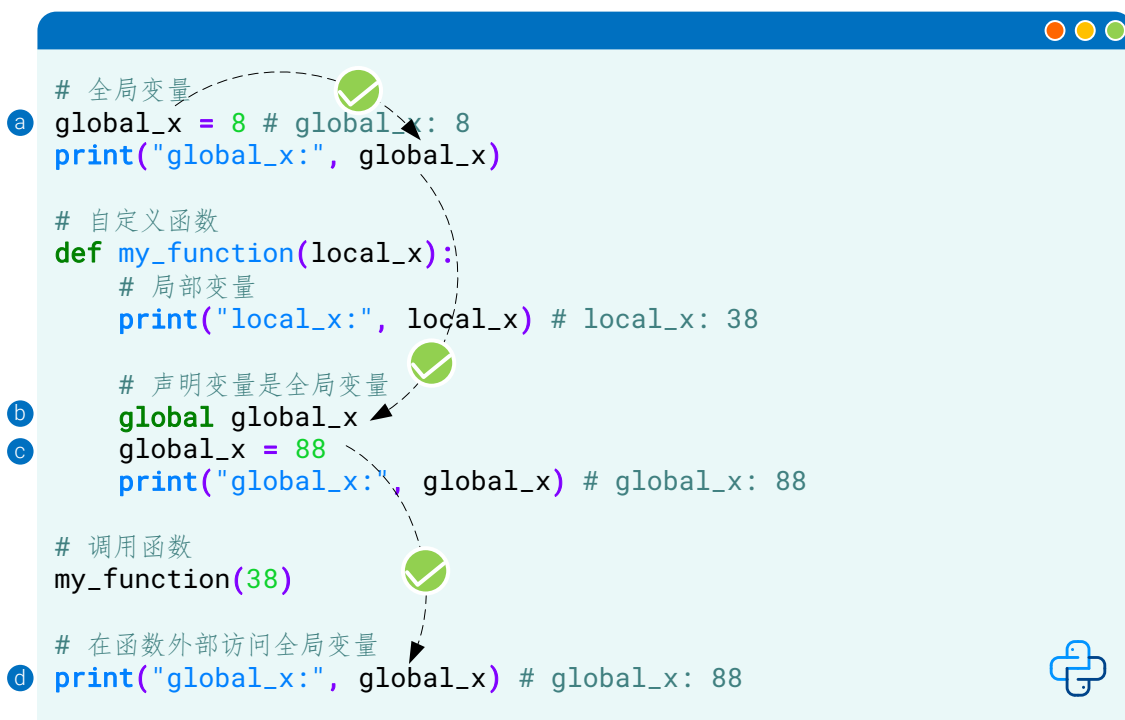


图 12. 全局变量和局部变量的传递路径，利用 global 在函数内部声明全局变量

### 将矩阵乘法打包成一个函数

上一章中，我们自定义了计算矩阵乘法代码。为了方便“多次调用”，下面我们将这段代码写成一个自定义函数。改良版的自定义函数，根据输入函数的形状，自行判断矩阵乘法结果矩阵的形状。

本 PDF 文件为作者草稿，发布目的为方便读者在移动终端学习，终稿内容以清华大学出版社纸质出版物为准。

版权归清华大学出版社所有，请勿商用，引用请注明出处。

代码及 PDF 文件下载：<https://github.com/Visualize-ML>

本书配套微课视频均发布在 B 站——生姜 DrGinger：<https://space.bilibili.com/513194466>

欢迎大家批评指教，本书专属邮箱：[jiang.visualize.ml@gmail.com](mailto:jiang.visualize.ml@gmail.com)

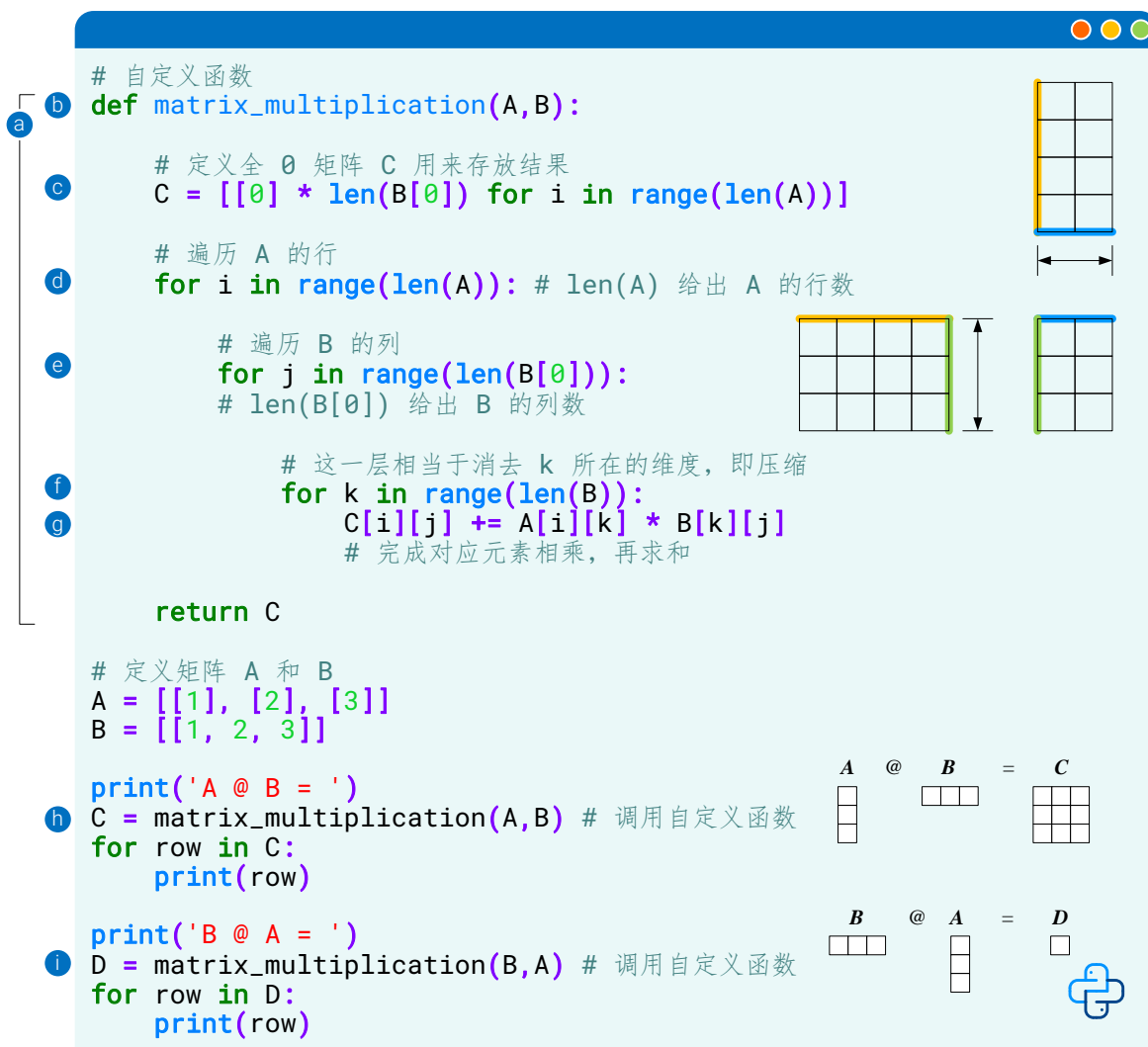


图 13. 将矩阵乘法打包成一个函数

大家可能会问怎么在自定义函数内添加一个判断语句来检查两个矩阵的尺寸是否匹配。如果不匹配，就抛出一个异常并提示错误信息。

以下是修改后的代码示例。

在函数中，我们使用 `len(A[0])` 和 `len(B)` 来检查第一个矩阵的列数是否等于第二个矩阵的行数。如果不相等，我们就使用 `raise` 语句抛出一个 `ValueError` 异常，并输出错误信息。这样，在调用函数时，如果输入的两个矩阵无法相乘，就会得到一个错误提示。

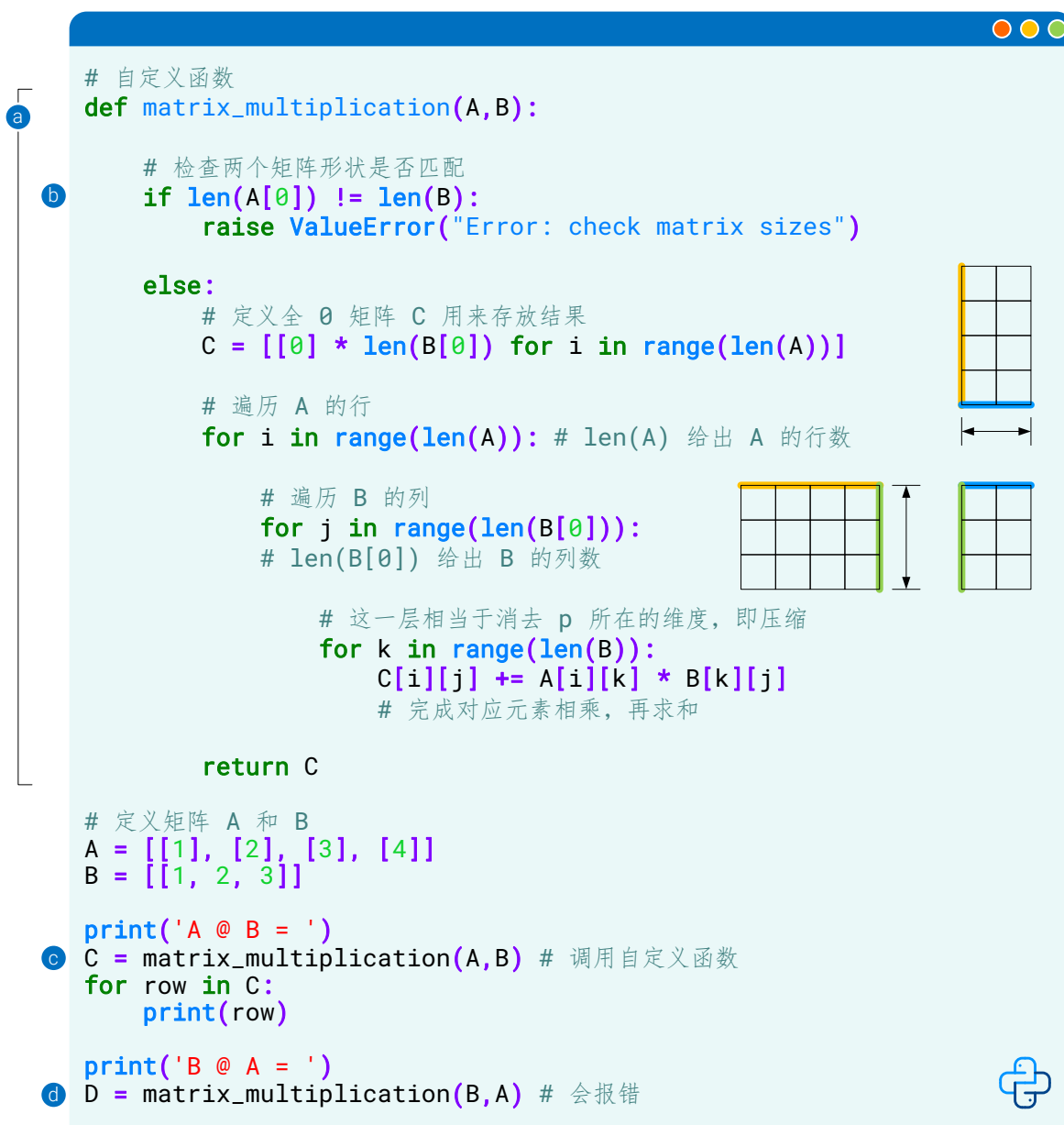


图 14. 将矩阵乘法打包成一个函数，增加矩阵形状不匹配的报错信息

除了用 `raise`，我们还可以用 `assert`。在 Python 中，`assert` 语句用于在代码中插入断言 (assertion)，用于检查程序的某个条件是否为真。如果条件为假 (False)，则会引发一个 `AssertionError` 异常，从而表示程序出现了一个错误。`assert` 通常用于在开发和调试过程中验证程序的假设和约束，以便更早期发现和诊断问题。

如图 15 所示，代码中 `assert b != 0` 用于确保除数不为零。如果除数为零，`assert` 语句将引发异常，从而防止程序继续执行不安全的操作。一般情况，`assert` 通常用于开发和调试阶段，以帮助发现和解决问题。

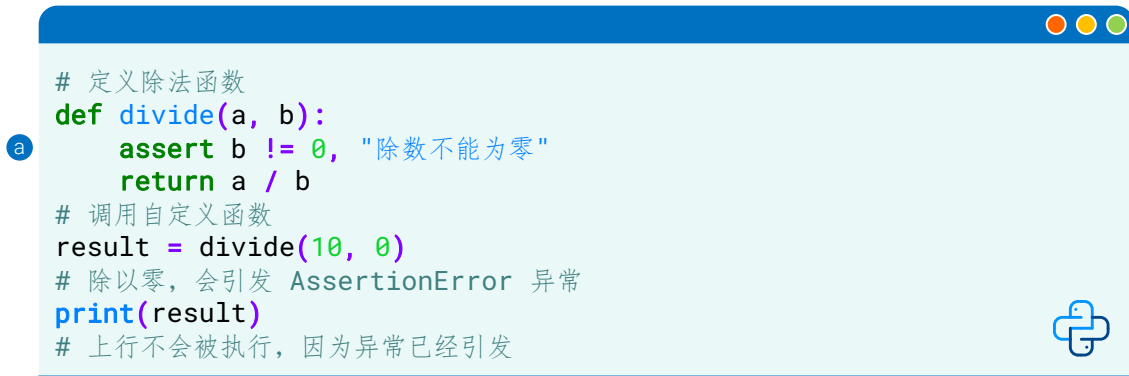


图 15. 用 assert 检查条件是否为真

## 帮助文档

在 Python 中，可以使用 docstring 来编写函数的帮助文档，即在函数定义的第一行或第二行写入字符串来描述函数的作用、参数、返回值等信息。通常使用三个单引号 (') 或三个双引号 (") 来表示 docstring，如下所示。如果要查询这个文档，可以使用 Python 内置的 help() 函数或者 \_\_doc\_\_ 属性来看。

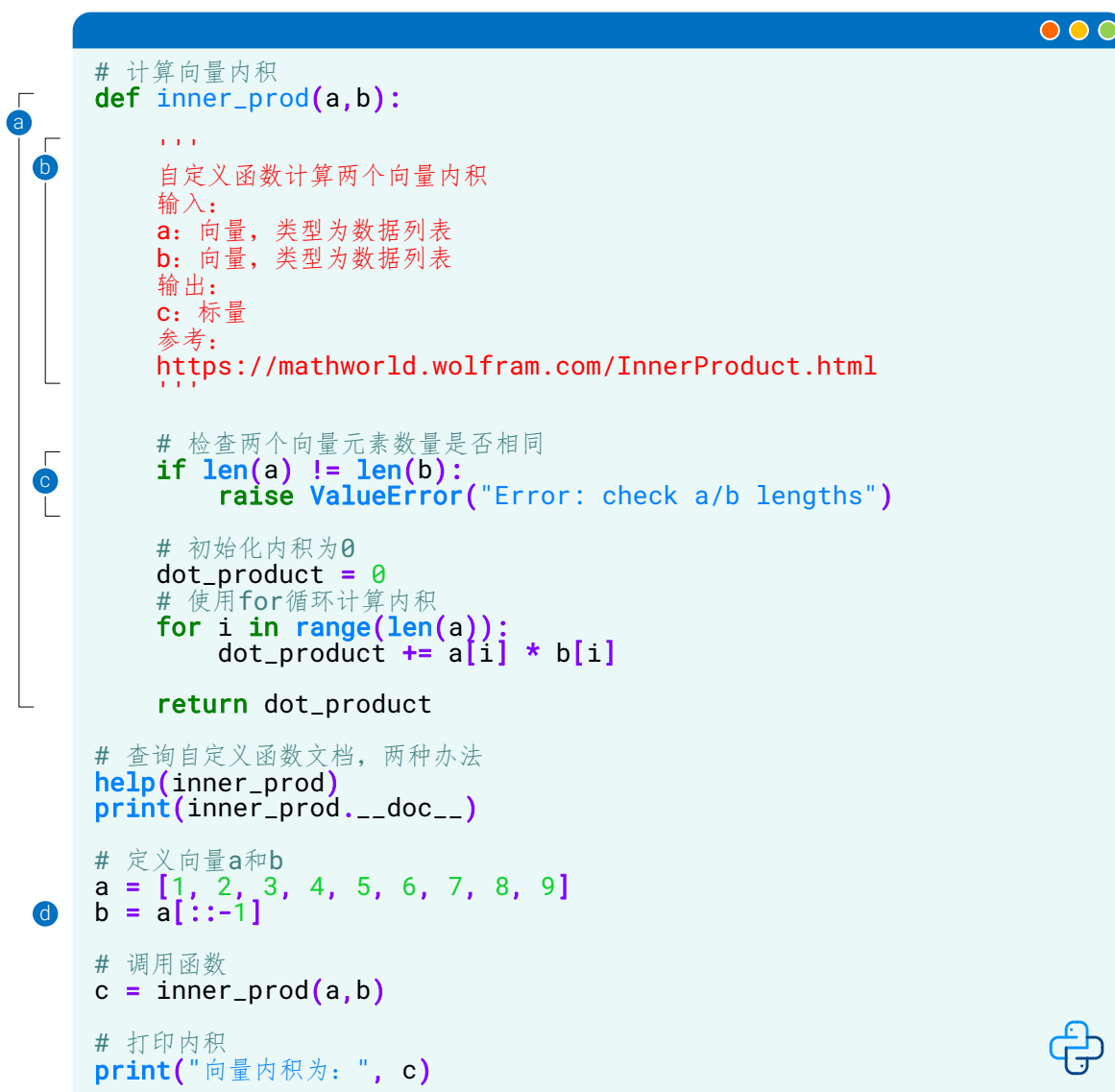


图 16. 自定义函数中的帮助文档

## 8.3 更多自定义线性代数函数

### 产生全 0 矩阵：一层 for 循环

下面举例如何用一层 for 循环产生全 0 矩阵。本书后文会介绍如何利用 `numpy.zeros()` 和 `numpy.zeros_like()` 生成全 0 矩阵。

下一章专门介绍如何自定义函数。



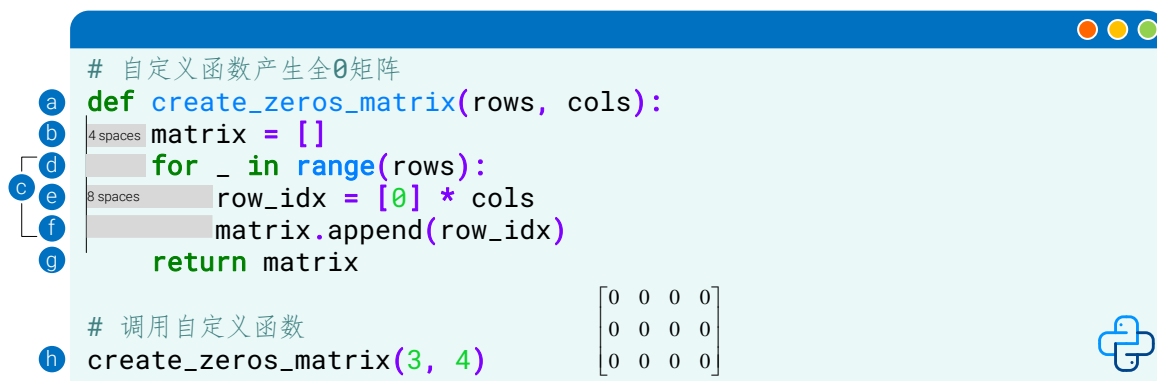


图 17. 产生全 0 矩阵

### 产生单位矩阵矩阵：一层 for 循环

下面举例如何用一层 for 循环产生单位矩阵。本书后文会介绍如何利用 `numpy.identity()` 产生单位矩阵。

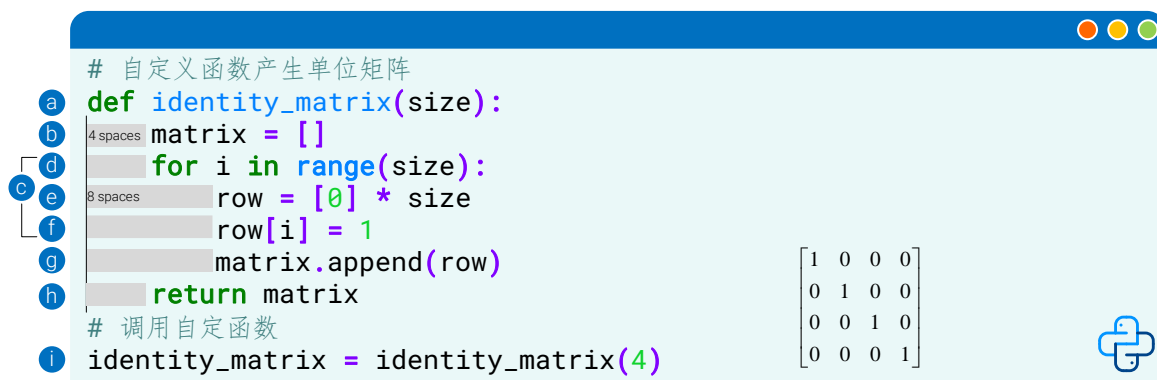


图 18. 产生单位矩阵

### 产生对角方阵：一层 for 循环

下面举例如何用一层 for 循环产生对角方阵。本书后文会介绍如何利用 `numpy.diag()` 产生对角方阵。

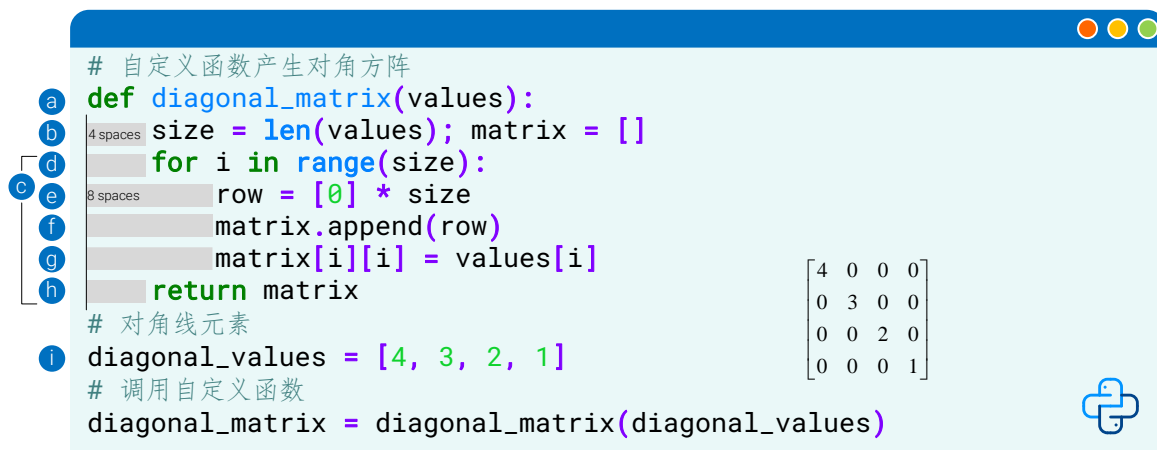


图 19. 产生对角方阵

本 PDF 文件为作者草稿，发布目的为方便读者在移动终端学习，终稿内容以清华大学出版社纸质出版物为准。

版权归清华大学出版社所有，请勿商用，引用请注明出处。

代码及 PDF 文件下载：<https://github.com/Visualize-ML>

本书配套微视频均发布在 B 站——生姜 DrGinger：<https://space.bilibili.com/513194466>

欢迎大家批评指教，本书专属邮箱：[jiang.visualize.ml@gmail.com](mailto:jiang.visualize.ml@gmail.com)

### 提取对角线元素：一层 for 循环

下面举例如何用一层 for 循环提取矩阵 (未必是方阵) 对角线元素。大家会发现 `numpy.diag()` 也可以用来提取矩阵对角线元素。

```
def extract_main_diagonal(matrix):
    rows = len(matrix); cols = len(matrix[0])
    size = min(rows, cols)
    diagonal = [matrix[i][i] for i in range(size)]
    return diagonal

matrix = [[1, 2, 3],
          [4, 5, 6]]

main_diagonal = extract_main_diagonal(matrix)
main_diagonal
```

图 20. 提取对角线元素

### 计算方阵迹

方阵的迹是指矩阵中主对角线上元素的总和。通常用  $\text{tr}(A)$  表示，其中  $A$  是方阵。

```
def trace(matrix):
    rows = len(matrix)
    cols = len(matrix[0])
    if rows != cols:
        raise ValueError("Matrix is not square")
    diagonal_sum = sum(matrix[i][i] for i in range(rows))
    return diagonal_sum

# 示例用法
A = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]

trace_A = trace(A)
print("矩阵的迹为:", trace_A)
```

图 21. 提取对角线元素

### 判断矩阵是否对称：两层 for 循环

下面举例如何用两层 for 循环判断矩阵是否对称。本书后文会介绍如何利用 `numpy.diag()` 产生对角方阵。

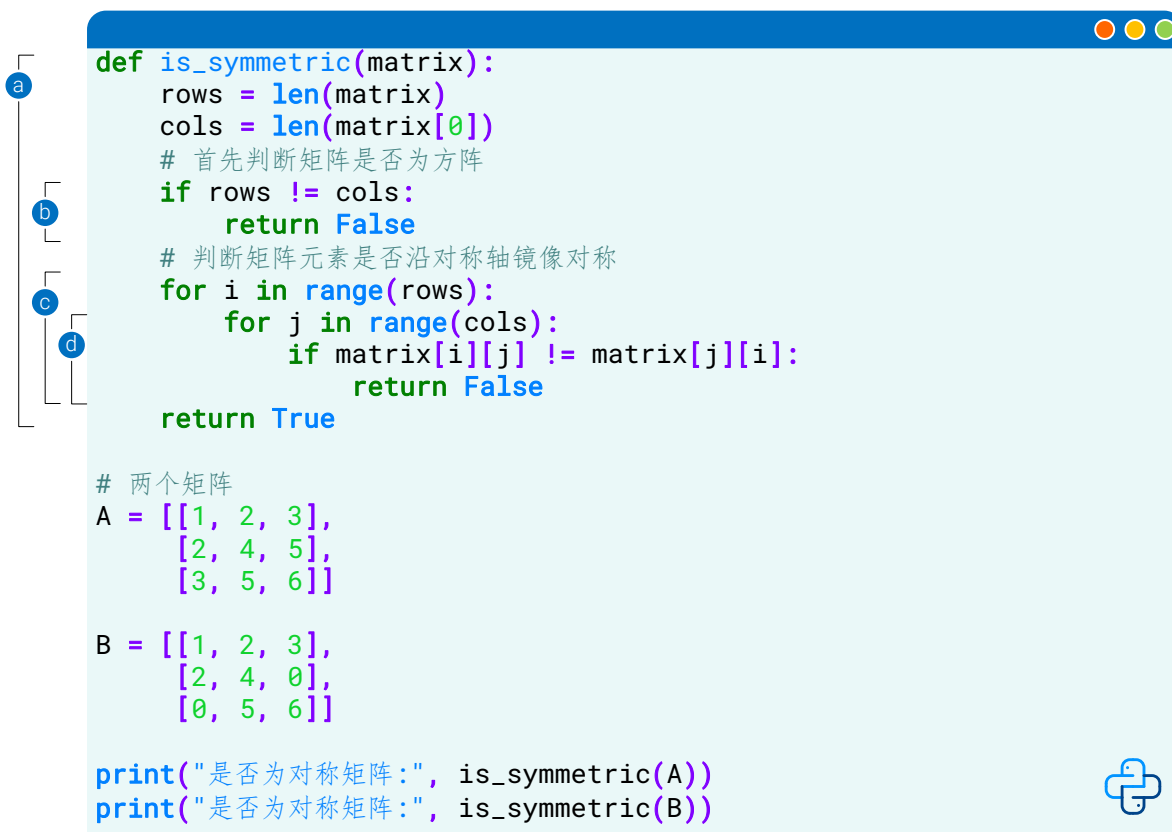


图 22. 判断矩阵是否为对称矩阵

矩阵转置来判断对称矩阵的方法。如果矩阵等于其转置，那么它是对称的，否则不是。



图 23. 利用矩阵转置判断矩阵是否对称

## 矩阵行列式

$2 \times 2$  矩阵  $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$  的行列式值为  $\det(A) = ad - bc$ 。

```
def determinant_2x2(matrix):
    a if len(matrix) != 2 or len(matrix[0]) != 2:
        raise ValueError("Matrix must be 2x2")
    a = matrix[0][0]
    b = matrix[0][1]
    c = matrix[1][0]
    d = matrix[1][1]
    det = a*d - b*c
    return det

# 示例用法
A = [[3, 2],
     [1, 4]]
det = determinant_2x2(A)
print("矩阵行列式:", det)
```

图 24.  $2 \times 2$  矩阵的行列式值

## 矩阵逆

$2 \times 2$  矩阵  $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$  的逆为  $\frac{1}{ad-bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$ 。

```
def inverse_2x2(matrix):
    if len(matrix) != 2 or len(matrix[0]) != 2:
        raise ValueError("Matrix must be 2x2")

    a = matrix[0][0]
    b = matrix[0][1]
    c = matrix[1][0]
    d = matrix[1][1]

    det = a * d - b * c
    if det == 0:
        raise ValueError("Matrix is not invertible")

    inv_det = 1 / det
    inv_matrix = [[d * inv_det, -b * inv_det],
                  [-c * inv_det, a * inv_det]]

    return inv_matrix

A = [[2, 3],
     [4, 5]]
inv_matrix = inverse_2x2(A)
```

图 25.  $2 \times 2$  矩阵的行列式值

## 8.4 递归函数：自己反复调用自己

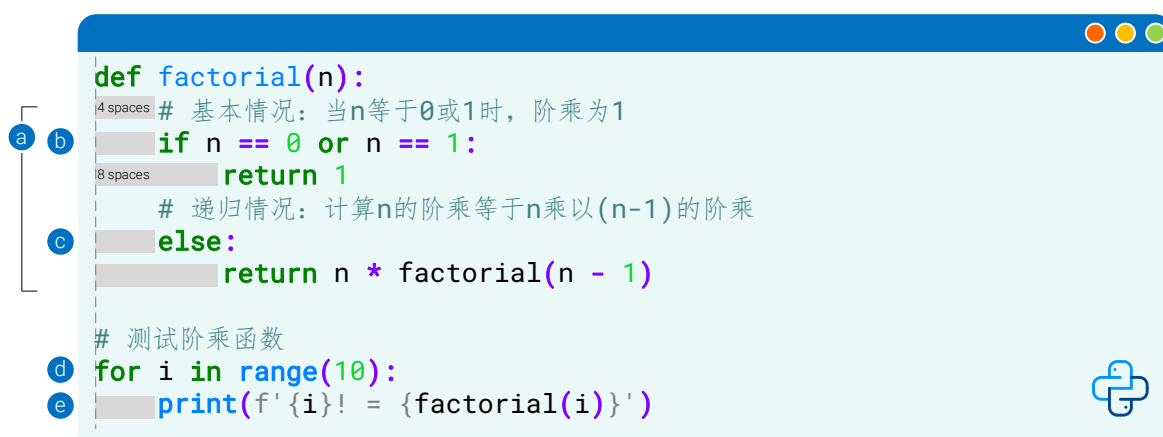
递归函数 (recursive function) 是一种在函数内部调用自身的编程技术。这种方法通常用于解决可以被分解成相似子问题的问题，每个子问题都可以用相同的方法来解决。

递归函数包括两个部分：基本情况和递归情况。基本情况是一个或多个条件，它们确定何时停止递归，而递归情况则是函数调用自身以处理更小的子问题。

下面通过两个例子讲解递归函数。

在图 26 这个例子中，我们定义 `factorial()` 函数生成阶乘 (`factorial`)。这个函数在基本情况返回 1，这是递归停止的条件。在递归情况下，它调用自身并将问题分解为更小的子问题，直到达到基本情况为止。

以 `factorial(5)` 为例，首先调用 `factorial(4)`，然后 `factorial(4)` 调用 `factorial(3)`，依此类推，直到 `factorial(1)` 返回 1。最终得到 `factorial(5)` 的值为 120。



```
def factorial(n):
    # 基本情况：当n等于0或1时，阶乘为1
    if n == 0 or n == 1:
        return 1
    # 递归情况：计算n的阶乘等于n乘以(n-1)的阶乘
    else:
        return n * factorial(n - 1)

# 测试阶乘函数
for i in range(10):
    print(f'{i}! = {factorial(i)}')
```

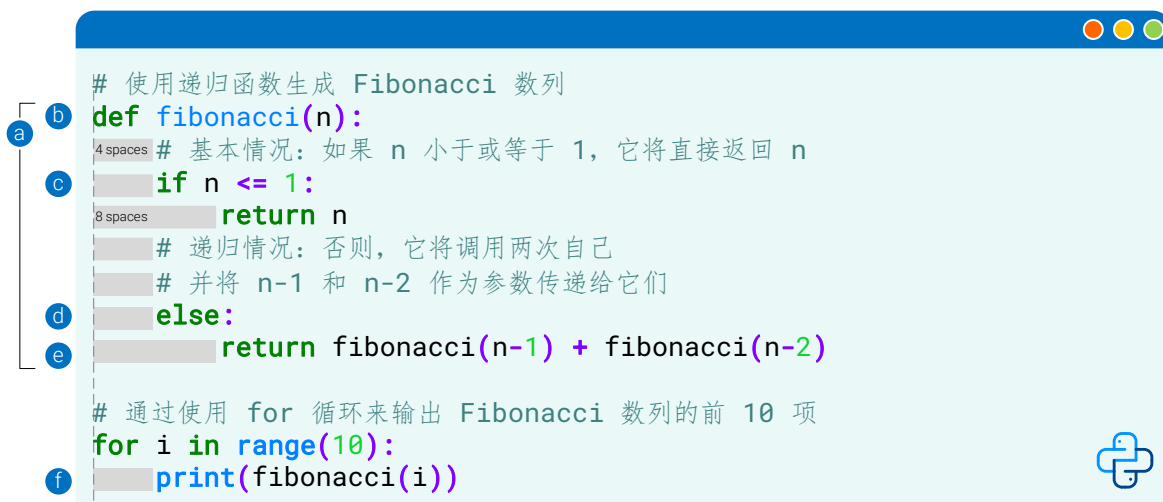
图 26. 使用递归方法生成阶乘

在图 27 这个例子中，我们定义 `fibonacci()` 函数生成斐波那契数列 (Fibonacci sequence) 接受一个整数 `n`，它返回 Fibonacci 数列的第 `n` 项。最终，当 `n` 达到 0 或 1 时（基本情况），递归将停止，返回相应的值。否则，它将调用两次自己，并将 `n-1` 和 `n-2` 作为参数传递给它。

图 27 代码通过使用 `for` 循环来输出 Fibonacci 数列的前 10 项，可以看到这个函数在工作时是如何递归调用自己的。



《可视之美》介绍如何可视化斐波那契数列。《数学要素》将专门介绍斐波那契数列。《矩阵力量》讲解如何用线性代数工具求解斐波那契数列通项公式。



```

# 使用递归函数生成 Fibonacci 数列
def fibonacci(n):
    # 基本情况：如果 n 小于或等于 1，它将直接返回 n
    if n <= 1:
        return n
    # 递归情况：否则，它将调用两次自己
    # 并将 n-1 和 n-2 作为参数传递给它们
    else:
        return fibonacci(n-1) + fibonacci(n-2)

# 通过使用 for 循环来输出 Fibonacci 数列的前 10 项
for i in range(10):
    print(fibonacci(i))

```

图 27. 使用递归方法生成斐波那契数列



### 什么是斐波那契数列？

斐波那契数列是一组数字，其中每个数字都是前两个数字的和。斐波那契数列的前几个数字是 0、1、1、2、3、5、8、13、21、34 等等。斐波那契数列是计算机科学中常用的例子，用于介绍递归和动态规划等概念。在植物学中，叶子、花瓣和果实的排列顺序可以遵循斐波那契数列。许多音乐家和作曲家使用斐波那契数列的规律来创建旋律和和弦。

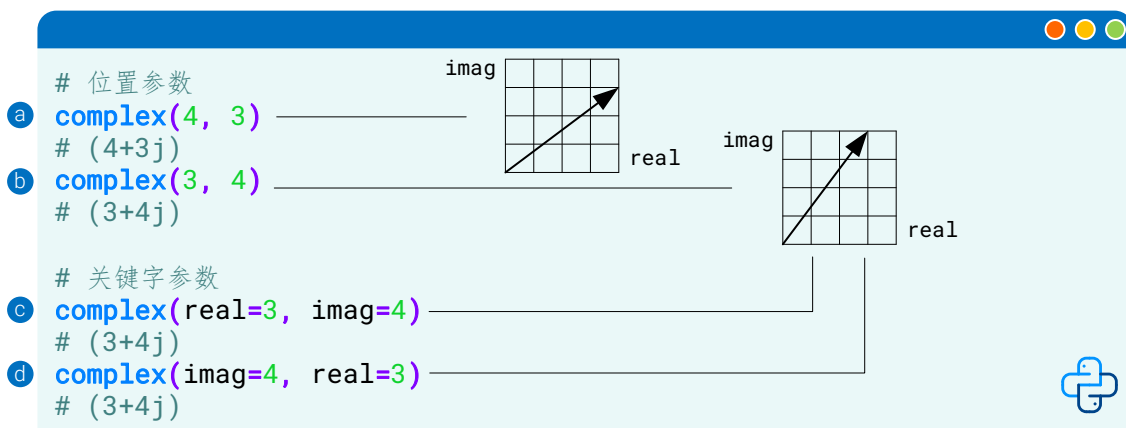
## 8.5 位置参数、关键字参数

Python 在定义函数时，有三类不同的函数输入参数。

- ▶ 位置参数 (positional arguments) 按照函数定义中参数的位置顺序传递参数值。
- ▶ 关键字参数 (keyword arguments) 通过参数的名称来传递参数值，而不依赖于参数的位置。
- ▶ 按位置或关键字传递 (positional or keyword arguments) 是指在函数调用时，可以选择按照参数在函数定义中的位置来传递参数值，也可以使用关键字来传递参数值。

图 28 以 Python 中的 complex 函数介绍位置参数、关键字参数的差别。

- ▶ **a** 和 **b** 利用位置参数创建复数。比较 **a** 和 **b**，可以发现当参数位置不同时，结果不同。
- ▶ **c** 和 **d** 利用关键字参数创建复数。可以发现，当指定参数名称后，参数的位置不会影响结果。

图 28. 函数 `complex()` 的位置参数、关键字参数

在自定义函数时，可以使用 `/` 和 `*` 来声明参数传递方式。在 Python 的函数定义中，正斜杠 `/` 之前的参数是位置参数；在正斜杠 `/` 和星号 `*` 之间位置或关键字传递都可以；在星号 `*` 之后必须按关键字传递。

以 `def fcn_name(a, b, /, c, d, *, e, f)` 为例，参数 `a` 和 `b` 是位置参数，必须按位置传递。

参数 `c` 和 `d` 既可以按位置传递，也可以按关键字传递。

而参数 `e` 和 `f` 是关键字参数，必须按关键字传递。

图 29 自定义了三个函数，自定义并计算抛物线  $y = ax^2 + bx + c$  某一点的函数值。

- a** 的自定义函数中四个参数都是位置参数。
- b** 的自定义函数中四个参数都是关键字参数。
- c** 自定义函数中，`a` 和 `b` 为位置参数，`c` 为位置/关键字参数，`x` 为关键字参数。



```

# 位置参数
a def quadratic_f(a, b, c, x, /):
    f = a * x **2 + b * x + c
    return f

quadratic_f(1, 2, 3, 4)
quadratic_f(3, 2, 1, 4)

# 关键字参数
b def quadratic_f_2(*, a, b, c, x):
    f = a * x **2 + b * x + c
    return f

quadratic_f_2(a = 1, b = 2, c = 3, x = 4)
quadratic_f_2(c = 3, x = 4, a = 1, b = 2)

# 关键字/位置参数
c def quadratic_f_3(a, b, /, c, *, x):
    f = a * x **2 + b * x + c
    return f

quadratic_f_3(1, 2, 3, x = 4)
quadratic_f_3(1, 2, c = 3, x = 4)

```

图 29. 自定义抛物线函数，位置参数、关键字参数

回到 `complex()` 定义复数，我们还可以用采用如下方法。

在 Python 中，一个星号 `*` 在图 30 中被用来进行“拆包” (unpacking) 操作。它的作用是将一个可迭代对象，比如列表、元组等，中的元素分别传递给函数的位置参数。

在图 30 示例中，`complex_list` 是一个包含两个元素的列表 `[3, 4]`。使用 `*complex_list` 来拆包这个列表，然后将拆包后的元素分别传递给 `complex` 函数的两个位置参数，创建一个复数对象，实际上相当于执行了 `complex(3, 4)`。

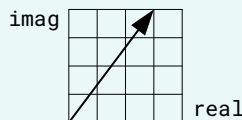
在 Python 中，两个星号 `**` 用于进行关键字参数拆包，将一个字典中的键值对作为关键字参数传递给函数。`complex_dict` 是一个包含两个键值对的字典 `{'real': 3, 'imag': 4}`，使用 `**complex_dict` 来将字典中的键值对拆包，并将它们作为关键字参数传递给 `complex` 函数。

```

# 使用一个星号
a complex_list = [3, 4]
b complex(*complex_list)

# 使用两个星号
c complex_dict = {'real': 3, 'imag': 4}
d complex(**complex_dict)

```

图 30. 采用 `*` 和 `**` 拆包

## 8.6 使用\*args 和\*\*kwargs

在 Python 中，自定义函数时，\*args 和 \*\*kwargs 是用于处理不定数量的参数的特殊语法。\*args 和 \*\*kwargs 让我们可以编写更加灵活的函数，可以接受不同数量的位置参数和关键字参数。其中，args 是 arguments 的简写，而 kw 是 keyword 的简写。

```
# 利用*args
a def multiply_all(*args):
    result = 1
    b print(args)
    c for num_idx in args:
        d result *= num_idx
    return result

# 计算4个值的乘积
print(multiply_all(1, 2, 3, 4))
# 计算6个值的乘积
print(multiply_all(1, 2, 3, 4, 5, 6))
```

图 31. 使用\*args

```
# 利用**kwargs
a def multiply_all_2(**kwargs):
    result = 1
    b print(type(kwargs))
    # 循环dict()
    c for key, value in kwargs.items():
        d print("The value of {} is {}".format(key, value))
        result *= value

    return result

# 计算3个key-value pairs中值的乘积
print(multiply_all_2(A = 1, B = 2, C = 3))
# 计算4个key-value pairs中值的乘积
print(multiply_all_2(A = 1, B = 2, C = 3, D = 4))
```

图 32. 使用\*\*kwargs

```

import statistics
# 混合 *args, **kwargs
a def calc_stats(operation, *args, **kwargs):
    result = 0
    # 计算标准差
    if operation == "stdev":
        # 总体标准差
        b if "TYPE" in kwargs and kwargs["TYPE"] == 'population':
            c result = statistics.pstdev(args)
        # 样本标准差
        d elif "TYPE" in kwargs and kwargs["TYPE"] == 'sample':
            e result = statistics.stdev(args)
        f else:
            raise ValueError('TYPE, either population or sample')
    # 计算方差
    elif operation == "var":
        # 总体方差
        g if "TYPE" in kwargs and kwargs["TYPE"] == 'population':
            result = statistics.pvariance(args)
        # 样本方差
        h elif "TYPE" in kwargs and kwargs["TYPE"] == 'sample':
            result = statistics.variance(args)
        else:
            raise ValueError('TYPE, either population or sample')
    else:
        print("Unsupported operation")
        return None
    # 保留小数位
    i if "ROUND" in kwargs:
        result = round(result, kwargs["ROUND"])
    return result

# 计算总体标准差
calc_stats("stdev", 1, 2, 3, 4, 5, 6,
           TYPE = 'population', ROUND = 3)
# 计算样本标准差
calc_stats("stdev", 1, 2, 3, 4, 5, 6, TYPE = 'sample')
# 计算总体方差
calc_stats("var", 1, 2, 3, 4, 5, 6,
           TYPE = 'population', ROUND = 4)
# 计算样本方差
calc_stats("var", 1, 2, 3, 4, 5, 6, TYPE = 'sample')

```

图 33. 混合使用\*args 和\*\*kwargs

表 1. Python 中星号 \* 的不同用法

用法	举例
乘法	a, b = 2, 3 a * b
乘幂	a, b = 2, 3

本 PDF 文件为作者草稿，发布目的为方便读者在移动终端学习，终稿内容以清华大学出版社纸质出版物为准。

版权归清华大学出版社所有，请勿商用，引用请注明出处。

代码及 PDF 文件下载：<https://github.com/Visualize-ML>

本书配套微视频均发布在 B 站——生姜 DrGinger: <https://space.bilibili.com/513194466>

欢迎大家批评指教，本书专属邮箱：[jiang.visualize.ml@gmail.com](mailto:jiang.visualize.ml@gmail.com)

	<code>a ** b</code>
复制列表	<code>a = [1, 2, 3]</code> <code>a * 3</code>
合并列表	<code>a_list = [1, 2]</code> <code>b_list = range(5)</code> <code>[*a_list, *b_list]</code>
拆包	<code># 拆包</code> <code>a_list = [1, 2, 3, 4, 5]</code> <code>first, *b_list = a_list</code>
分割位置参数和关键字参数	<code>def fcn_name(a, b, /, c, d, *, e, f)</code> <code>pass</code>
位置参数	<code>def fcn_name(*args):</code> <code>pass</code>
关键字参数	<code>def fcn_name(**kwargs):</code> <code>pass</code>

## 8.7 匿名函数

在 Python 中，匿名函数也被称为 lambda 函数，是一种快速定义单行函数的方式。使用 lambda 函数可以避免为简单的操作编写大量的代码，而且可以作为其他函数的参数来使用。

匿名函数的语法格式为：lambda arguments: expression。其中，arguments 是参数列表，expression 是一个表达式。当匿名函数被调用时，它将返回 expression 的结果。

下面是一些使用匿名函数的例子。



图 34. lambda 函数

在这个例子中，我们定义了一个匿名函数 `lambda x: x + 1`，该函数接受一个参数 `x` 并返回 `x` 加 1。然后我们使用 `map()` 将这个函数应用于列表 `my_list` 中的每个元素，并将结果存储在 `list_plus_1` 列表中。类似地，我们还计算了 `my_list` 中的每个元素的平方。

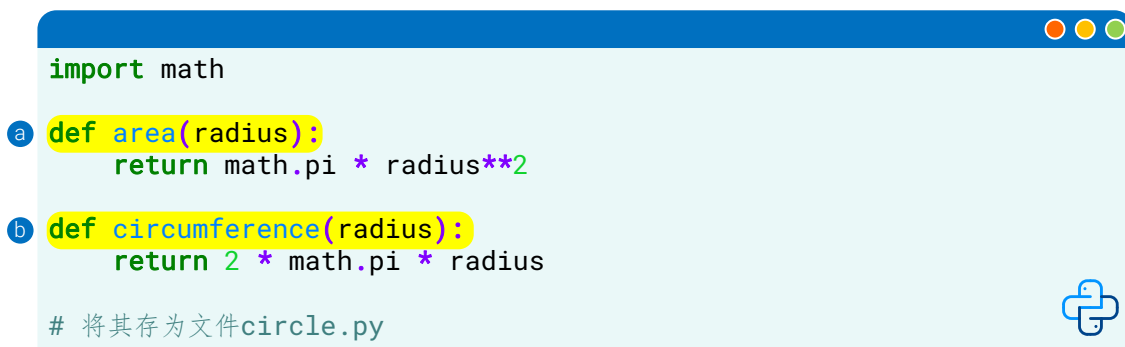
在 Python 中，`map()` 是一种内置的高阶函数，它接受一个函数和一个可迭代对象作为输入，将函数应用于可迭代对象的每个元素并返回一个可迭代对象，其中每个元素都是应用于原始可迭代对象的函数的结果。

## 8.8 构造模块、库

简单来说，若干函数可以打包成一个模块，几个模块可以打包成一个库。本节简单聊一聊如何创建模块、创建库，对于大部分读者来说这一节可以跳过不读。

### 自定义模块

在 Python 中，我们可以将几个相关的函数放在一个文件中，这个文件就成为一个模块。下面是一个例子。假设我们有两个函数，一个是计算圆的面积，一个是计算圆的周长，我们可以将这两个函数放在一个文件中，例如我们可以创建一个名为 `"circle.py"` 的文件，并将以下代码添加到该文件中。我们首先导入了 `math` 模块，然后定义了两个函数 `area()` 和 `circumference()`，分别用于计算圆的面积和周长。



```
import math

a def area(radius):
    return math.pi * radius**2

b def circumference(radius):
    return 2 * math.pi * radius

# 将其存为文件circle.py
```

图 35. 构造模块 `circle.py`

在本章配套的代码中，我们调用了 `circle.py`。使用 `import` 语句导入了 `circle` 模块，并命名为 `cc`，然后通过 `cc.area()`、`cc.circumference()` 调用函数。

### 自定义库

在 Python 中，可以使用 `setuptools` 库中的 `setup()` 函数将多个模块打包成一个库。本章配套代码中给出的例子对应的具体步骤如下：

创建一个文件夹，用于存放库的代码文件，例如命名为 `mylibrary`。

在 `mylibrary` 文件夹中创建一个名为 `setup.py` 的文件，引入 `setuptools` 库，并使用 `setup()` 函数来描述库的信息，包括名称、版本、作者、依赖、模块文件等信息。

在 mylibrary 文件夹中创建一个名为\_\_init\_\_.py 的空文件(内容空白)，用于声明这个文件夹是一个 Python 包。

在 mylibrary 文件夹中创建多个模块文件，这些模块文件包含需要打包的函数或类。比如，mylibrary 中含有 linear\_alg.py 和 circle.py 两个模块。linear\_alg.py 有矩阵乘法、向量内积两个函数。circle.py 有计算圆面积、周长两个函数。

本章配套的代码中给出如何调用自定义库。

## 8.9 模仿学习别人的代码

模仿学习别人的优质代码是提高编程技能的重要途径之一。

大家可以在各种常见 Python 第三方库，比如 Matplotlib、NumPy、Pandas 等等，找到示例代码。此外，在 GitHub 中大家也可以通过关键词找到自己感兴趣的代码库作为模仿对象。

此外，对于初学者，最好的模仿学习对象莫过于在安装 Anaconda 时已经安装在本地的 Python 代码。本节以 Python 中 Statistics 库的几个函数为例和大家探讨如何通过模仿学习别人的优质代码范例。

以 Statistics 库为例，以下途径可以帮助我们找到源代码。

第一种方法，在 JupyterLab 中，利用图 36 语句可以查看 statistics 中的 linear\_regression 函数的源代码。在 JupyterLab 中双问号 ?? 通常用于获取函数或模块的帮助文档、源代码。

而 help(statistics.linear\_regression) 只会打开相关函数的帮助文档。

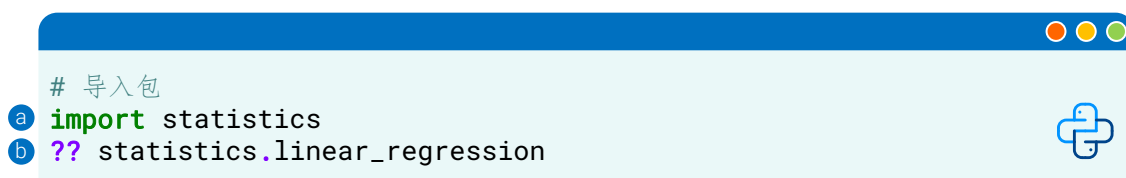


图 36. 在 JupyterLab 中打开 statistics 库中 linear\_regression 函数的源代码

第二种方法是直接进入 Python 官方的 GitHub 查看源代码文件，比如如下链接是 Statistics 库的源代码。大家可能注意到这个源代码对应 Python 3.12 版本。大家可以根据自己需求查看不同版本的 Python 库原函数。

<https://github.com/python/cpython/blob/3.12/Lib/statistics.py>

此外，Statistics 库的官方文档也有指向源代码仓库的超链接。

<https://docs.python.org/3/library/statistics.html>

第三种方法是找到本地安装地址，比如：

c:\users\user\_name\anaconda3\lib\statistics.py

然后使用 JupyterLab 打开，或者直接用 Notepad++ 等文本编辑软件打开查看。

第四种方法需要用到 Spyder。Spyder 也是一种 IDE，在安装 Anaconda 时，Spyder 也同时被安装。在 Spyder 中利用 `ctrl + O` 快捷键也可以打开 `statistics.linear_regression`。本书第 34 章将专门介绍 Spyder。

不管用什么方法，我们可以找到如图 37 代码（删除帮助文档）。大家可以在本章配套代码中找到 `statistics.linear_regression` 的帮助文档。

这段源代码虽然看着很短，但是有很多有意思的知识点，很值得聊一聊。

本章前文介绍过 <sup>a</sup> 中的正斜杠 (forward slash) /。简单来说，正斜杠 / 用来指定函数输入中位置参数的结束位置。在正斜杠之前的所有参数，必须安装特定位置顺序传递；但是，不能通过关键字参数方式传递，也就是不能用 `statistics.linear_regression(x = x_data, y = y_data)`。正斜杠的使用可以限制参数的传递方式，有助于确保参数按照正确的位置顺序传递给函数，提高了函数的可读性和可维护性。

<sup>b</sup> 计算变量 `x` 中元素的数量，并将结果赋值给变量 `n`。

<sup>c</sup> 用 `if` 判断如果变量 `y` 中的元素数量不等于变量 `x` 中的元素数量，则执行条件语句中的代码块。

当 `x` 和 `y` 两个列表中元素数量不同时，<sup>d</sup> 中 `raise` 语句用于显式地引发异常。`StatisticsError` 是在 `Statistics` 库中定义的异常类型。

<sup>e</sup> 用 `if` 判断元素数量是否小于 2。如果小于 2，则引发 <sup>f</sup> 中异常。

<sup>g</sup> 计算了一组数据 `x` 的平均值 `xbar`，对应  $\frac{\sum_{i=1}^n x_i}{n}$ ，通过将数据中的所有元素相加并除以元素的数量来实现。其中，`fsum()` 来自 `math` 库，用来对浮点数精确求和。变量 `xbar` 对应样本均值  $\bar{x}$ ，`bar` 就是 `x` 上面的横线。

$\sum_{i=1}^n ( )$  是求和符号，读作 Sigma，代表从序号 `i` 从 1 到 `n` 的求和。

<sup>h</sup> 和上一句相同，计算一组数据 `y` 的平均值 `ybar`，对应  $\frac{\sum_{i=1}^n y_i}{n}$ 。

<sup>i</sup> 计算  $\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$ 。

其中， $x_i$  是 `x` 列表中序号为 `i` 的元素， $y_i$  是 `y` 列表中序号为 `i` 的元素。 $\bar{x}$  和  $\bar{y}$  对应 `x` 和 `y` 列表的样本均值。

本书前文介绍过 `zip()`，这一句中 `zip(x, y)` 将两个序列 `x` 和 `y` 中的对应元素顺序配对，创建一个迭代器，用于同时迭代这两个序列。这样，`(xi, yi)` 表示 `x` 和 `y` 中对应位置的数据点。

`(xi - xbar)` 和 `(yi - ybar)` 分别表示每个数据点与其对应的平均值的偏差。这两个值分别表示了数据点相对于平均值的位置。

`(xi - xbar) * (yi - ybar)` 计算了每对数据点的偏差的乘积，即每个数据点相对于各自的平均值的位置乘积。

所以，`sxy = fsum((xi - xbar) * (yi - ybar) for xi, yi in zip(x, y))` 这行代码的作用是计算了 `x` 和 `y` 之间的样本协方差 (的 `n - 1` 倍)，用于衡量这两组数据之间的线性关系。

<sup>j</sup> 计算  $\sum_{i=1}^n (x_i - \bar{x})^2$ 。这行代码的作用是计算了一组数据 `x` 的样本方差 (的 `n - 1` 倍)，用于衡量数据点相对于均值的分散程度。方差越大表示数据点越分散，方差越小表示数据点越集中在均值附近。

本 PDF 文件为作者草稿，发布目的为方便读者在移动终端学习，终稿内容以清华大学出版社纸质出版物为准。

版权归清华大学出版社所有，请勿商用，引用请注明出处。

代码及 PDF 文件下载：<https://github.com/Visualize-ML>

本书配套微课视频均发布在 B 站——生姜 DrGinger：<https://space.bilibili.com/513194466>

欢迎大家批评指教，本书专属邮箱：[jiang.visualize.ml@gmail.com](mailto:jiang.visualize.ml@gmail.com)



**k** 的 try 是一个异常处理 (try ... except ...) 的开始部分。代码块中的操作会被尝试执行，如果发生异常，则会跳转到 except 语句块中的代码，用于处理异常情况。

**l** 计算一元 OLS 线性回归模型 ( $y = b_1x + b_0$ ) 斜率  $b_1$ ，使用了之前计算得到之前得到的 sxy 和 sxx。

**m** 是一句注释，告诉大家还可以用协方差和方差比例值计算斜率  $b_1$ 。

**n** 是一个异常处理的一部分，它捕获可能发生的 ZeroDivisionError 异常，这是因为在计算斜率时，如果 sxx 为零，就会发生除以零的错误。大家想一想什么情况下 sxx 为零？

**o** 在捕获到 ZeroDivisionError 异常时，会引发一个自定义的 StatisticsError 异常，并提供了错误消息 'x is constant'。

**p** 计算一元 OLS 线性回归模型 ( $y = b_1x + b_0$ ) 截距  $b_0$ 。

**q** 返回斜率、截距。这一句用到了 LinearRegression，它是用 Python 中 collections 模块中 namedtuple 函数创建的具有命名字段的轻量级的类似元组的数据结构。

代码中使用 collections.namedtuple，可以提高代码的可读性，因为字段名称可以充当注释，帮助我们理解代码的含义。

图 37 的这段代码虽然简单，但是却很好地展示了“从公式到代码”。表 2 总结了这段代码中涉及的数学公式和对应代码。把数学公式、算法逻辑变成代码，然后再想办法提高运算效率，这是大家需要掌握的重要编程技能。

表 2. 从公式到代码

公式	代码
$\bar{x} = \frac{\sum_{i=1}^n x_i}{n}$	<code>xbar = fsum(x) / n</code> # 计算 x 序列样本均值
$\bar{y} = \frac{\sum_{i=1}^n y_i}{n}$	<code>ybar = fsum(y) / n</code> # 计算 y 序列样本均值
$s_{x,y} = \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$	<code>sxy = fsum((xi - xbar) * (yi - ybar) for xi, yi in zip(x, y))</code> # 计算协方差 (的 n-1 倍)
$s_x^2 = \sum_{i=1}^n (x_i - \bar{x})^2$	<code>sxx = fsum((xi - xbar) ** 2.0 for xi in x)</code> # 计算方差 (的 n-1 倍)
$b_1 = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2} = \frac{s_{x,y}}{s_x^2}$	<code>slope = sxy / sxx</code> # 计算一元线性回归函数 $y = b_1x + b_0$ 的斜率 $b_1$
$b_0 = \bar{y} - b_1\bar{x}$	<code>intercept = ybar - slope * xbar</code> # 计算一元线性回归函数 $y = b_1x + b_0$ 的截距 $b_0$

```

a def linear_regression(x, y, /):
b     n = len(x)
c     if len(y) != n:
d         raise StatisticsError('linear regression requires that
                                both inputs have same number of
                                data points')
e
f     if n < 2:
g         raise StatisticsError('linear regression requires at least
                                two data points')
h
g     xbar = fsum(x) / n
h     ybar = fsum(y) / n
i     sxy = fsum((xi - xbar) * (yi - ybar) for xi, yi in zip(x, y))
j     sxx = fsum((xi - xbar) ** 2.0 for xi in x)
k     try:
l         slope = sxy / sxx
m         # equivalent to: covariance(x, y) / variance(x)
n     except ZeroDivisionError:
o         raise StatisticsError('x is constant')
p     intercept = ybar - slope * xbar
q     return LinearRegression(slope=slope, intercept=intercept)

```

图 37. Statistics 库中 linear\_regression 函数的源代码



请大家完成下面题目。

Q1. 把本章第 3 节介绍的有关线性代数函数打包成一个模块，并存成一个.py 文件；然后，从 Jupyter Notebook 中分别调用这些函数。

Q2. 找到 statistics.variance 的源代码，并逐句分析注释。

Q3. 找到 statistics.covariance 的源代码，并逐句分析注释。

Q4. 找到 statistics.correlation 的源代码，并逐句分析注释。

Q5. 参考第 6 章线性回归代码，利用 statistics 库中 mean、variance、correlation 函数计算斜率、截距。

\* 不提供答案。

