Functions in Python

8 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

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() 可以将。

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。现成的函数不能满足“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 vs 编程函数

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)。

一元函数通常表示为 $f(x)$，其中 x 是输入变量，$f(x)$ 是输出变量。比如一元一次函数，$f(x) = 2x + 1$，一元二次函数 $f(x) = x^2 + 2x$，正弦函数 $f(x) = \sin(x)$，指数函数 $f(x) = \exp(x)$。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。

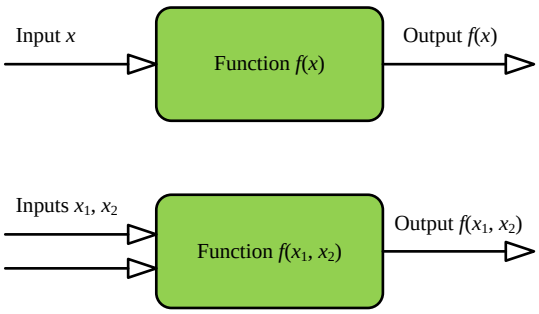


图 1. 一元函数、二元函数的映射

数学上的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，三元函数 $f(x_1, x_2, x_3)$ 有 3 个输入，多元函数 $f(x_1, x_2, \ldots, x_D)$ 有 D 个输入。

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。

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。

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。

数学上的函数还是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。

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!"。

```
def say_hello():
    # 自定义函数：打印问候
    # 输入：无
    # 输出：无
    print("Hello!")

# 调用自定义函数
say_hello()
```

Function block

无输入、无输出自定义函数

图 2. 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 3 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。

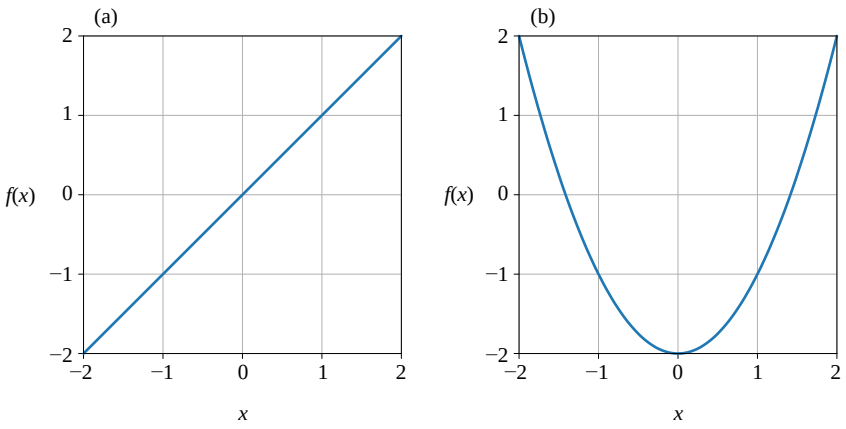


图 3. 绘制线图并美化

无输入、无输出自定义函数

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

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

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

# 绘制直线
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

# 绘制抛物线
fig, ax = plt.subplots(figsize = (4,4))
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
```

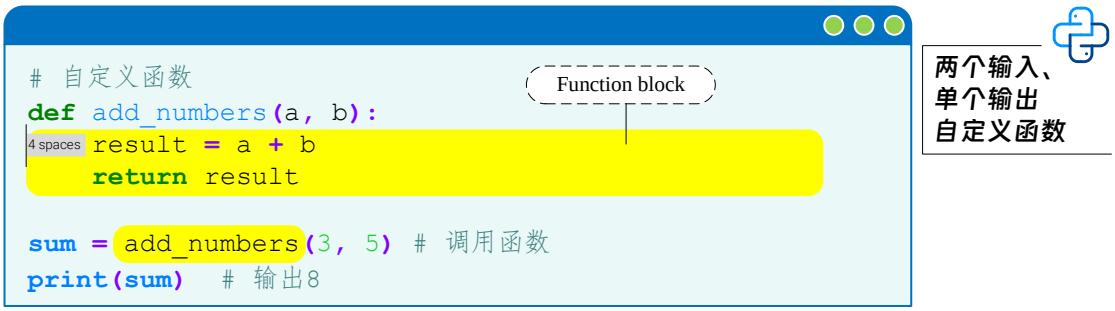
Function block

4 spaces

图 4. 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。

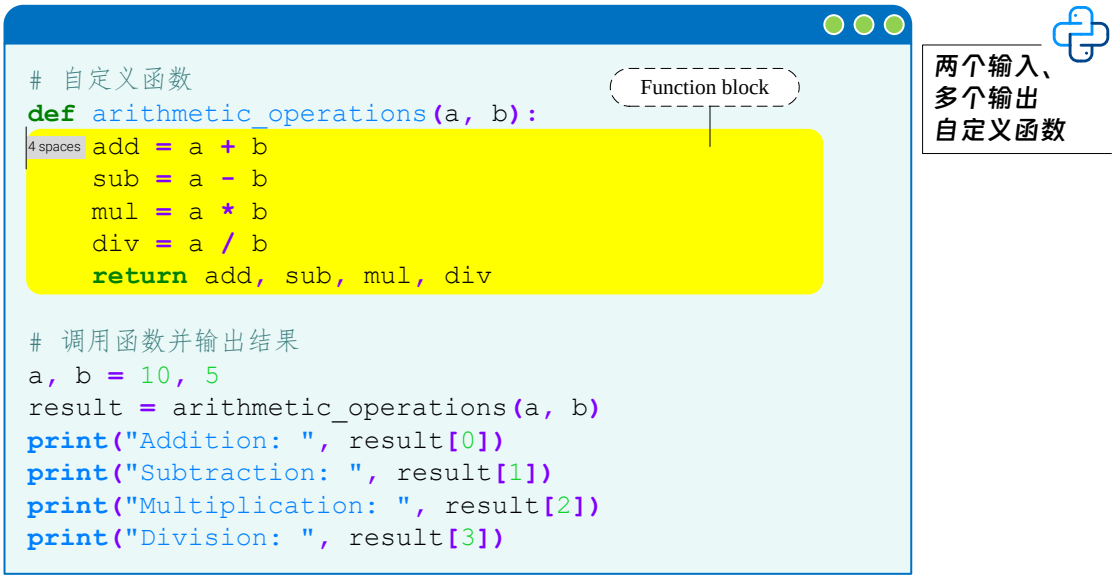



```python
# 自定义函数
def add_numbers(a, b):
    result = a + b
    return result

sum = add_numbers(3, 5) # 调用函数
print(sum)  # 输出8
```

图 5. 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。




```python
# 自定义函数
def arithmetic_operations(a, b):
    add = a + b
    sub = a - b
    mul = a * b
    div = a / b
    return add, sub, mul, div

# 调用函数并输出结果
a, b = 10, 5
result = arithmetic_operations(a, b)
print("Addition: ", result[0])
print("Subtraction: ", result[1])
print("Multiplication: ", result[2])
print("Division: ", result[3])
```

图 6. 两个输入、多个输出函数

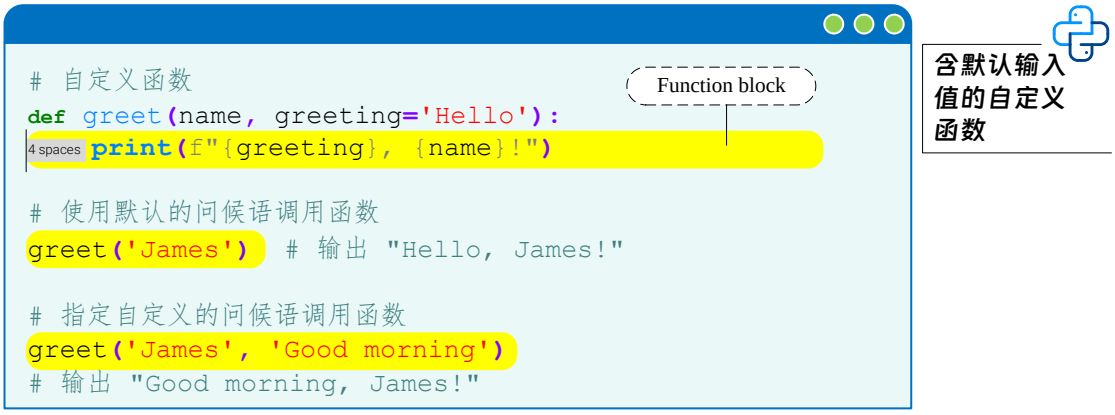
部分输入有默认值

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，先定义的参数必须先被传入，后定义的参数后被传入。如果违反了这个顺序，Python 解释器就会抛出 SyntaxError 异常。




```
# 自定义函数
def greet(name, greeting='Hello'):
    print(f"{greeting}, {name}!")

# 使用默认的问候语调用函数
greet('James')  # 输出 "Hello, James!"

# 指定自定义的问候语调用函数
greet('James', 'Good morning')
# 输出 "Good morning, James!"
```

图 7. 函数输入有默认值

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“多次调取”，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。

矩阵乘法

```
# 自定义函数
def matrix_multiplication(A,B):
    # 定义全 0 矩阵 C 用来存放结果
    C = [[0] * len(B[0]) for i in range(len(A))]

    # 遍历 A 的行
    for i in range(len(A)): # len(A) 给出 A 的行数

        # 遍历 B 的列
        for j in range(len(B[0])):
        # len(B[0]) 给出 B 的列数

            # 这一层相当于消去 p 所在的维度，即压缩
            for k in range(len(B)):
                C[i][j] += A[i][k] * B[k][j]
                # 完成对应元素相乘，再求和

    return C

# 定义矩阵 A 和 B
A = [[1, 2, 10, 20],
     [3, 4, 30, 40]]
B = [[4, 2],
     [3, 1],
     [40, 20],
     [30, 10]]

C = matrix_multiplication(A,B) # 调用自定义函数

print('A @ B = ')
# 输出结果
for row in C:
    print(row)

# 定义矩阵 X 和 Y
X = [[1], [2], [3]]
Y = [[1, 2, 3]]

print('X @ Y = ')
Z = matrix_multiplication(X,Y) # 调用自定义函数
# 输出结果
for row in Z:
    print(row)
```

Function block

4 spaces

图 8. 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。

矩阵乘法，检测矩阵形状

```python
# 自定义函数
def matrix_multiplication(A,B):

    # 检查两个矩阵形状是否匹配
    if len(A[0]) != len(B):
        raise ValueError("Error: check matrix sizes")

    else:
        # 定义全 0 矩阵 C 用来存放结果
        C = [[0] * len(B[0]) for i in range(len(A))]

        # 遍历 A 的行
        for i in range(len(A)): # len(A) 给出 A 的行数

            # 遍历 B 的列
            for j in range(len(B[0])):
            # len(B[0]) 给出 B 的列数

                # 这一层相当于消去 p 所在的维度，即压缩
                for k in range(len(B)):
                    C[i][j] += A[i][k] * B[k][j]
                    # 完成对应元素相乘，再求和

        return C

# 定义矩阵 A 和 B
A = [[1, 2, 10, 20],
     [3, 4, 30, 40]]

B = [[4, 2],
     [3, 1]]

# 调用自定义函数
C = matrix_multiplication(A,B) # 报错
```

图 9. 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

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__ 属性来查看。

帮助文档

```
# 计算向量内积
def inner_prod(a,b):

    '''
    自定义函数计算两个向量内积
    输入:
    a: 向量，类型为数据列表
    b: 向量，类型为数据列表
    输出:
    c: 标量
    参考:
    https://mathworld.wolfram.com/InnerProduct.html
    '''

    # 检查两个向量元素数量是否相同
    if len(a) != len(b):
        raise ValueError("Error: check a/b lengths")

    # 初始化内积为0
    dot_product = 0
    # 使用for循环计算内积
    for i in range(len(a)):
        dot_product += a[i] * b[i]

    return dot_product

# 查询自定义函数文档，两种办法
help(inner_prod)
print(inner_prod.__doc__)

# 定义向量a和b
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
b = a[::-1]

# 调用函数
c = inner_prod(a,b)

# 打印内积
print("向量内积为: ", c)
```

图 10. 帮助文档

8.3 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。

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

# 将列表中的所有元素加 1
list_plus_1 = list(map(lambda x: x+1, my_list))
print(list_plus_1)
# [2, 3, 4, 5, 6]

# 将列表中的所有元素分别求平方
list_squared = list(map(lambda x: x**2, my_list))
print(list_squared)
# [1, 4, 9, 16, 25]
```

lambda函数

图 11. 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。

8.4 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。

```python
import math

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

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

# 将其存为文件circle.py
```

构造模块

图 12. 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。

请大家完成下面 1 道题目。

Q1. 本章的唯一的题目就是请大家在 JupyterLab 中练习本章正文给出的示例代码。

* 不提供答案。