

## 7

## Control Flow Statements in Python

## Python 控制结构

日后尽量避免 for 循环，争取用向量化绕行



幸存下来的不是最强壮的物种，也不是最聪明的物种，而是对变化最敏感的物种。

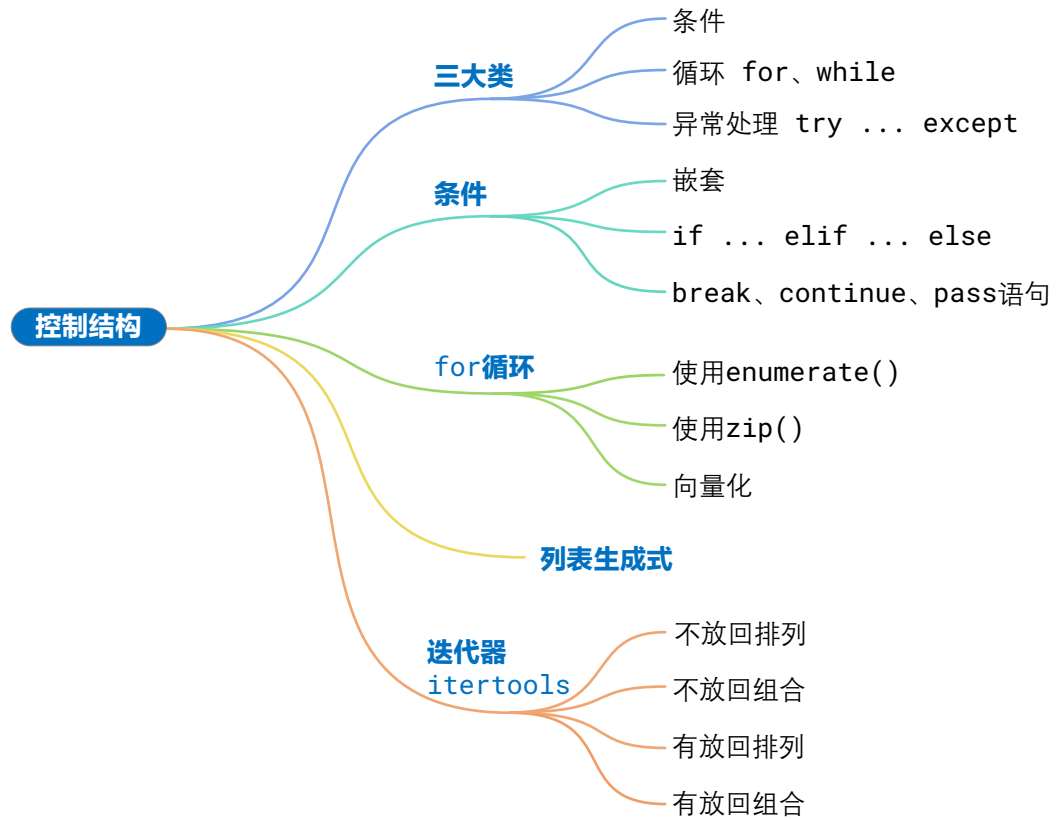
*It is not the strongest of the species that survives, nor the most intelligent, but the one most responsive to change.*

—— 查尔斯·达尔文 (Charles Darwin) | 进化论之父 | 1809 ~ 1882



- ◀ `enumerate()` 用于在迭代过程中同时获取元素的索引和对应的值
- ◀ `for ... in ...` Python 循环结构，用于迭代遍历一个可迭代对象中的元素，每次迭代时执行相应的代码块
- ◀ `if ... elif ... else` Python 条件语句，用于根据多个条件之间的关系执行不同的代码块，如果前面的条件不满足则逐个检查后续的条件
- ◀ `if ... else ...` Python 条件语句，用于在满足 `if` 条件时执行一个代码块，否则执行另一个 `else` 代码块
- ◀ `itertools.combinations()` 用于生成指定序列中元素的所有组合，并返回一个迭代器
- ◀ `itertools.combinations_with_replacement()` 用于生成指定序列中元素的所有带有重复元素的组合，并返回一个迭代器
- ◀ `itertools.permutations()` 用于生成指定序列中元素的所有排列，并返回一个迭代器
- ◀ `itertools.product()` 用于生成多个序列的笛卡尔积（所有可能的组合），并返回一个迭代器
- ◀ `try ... except ...` Python 中的异常处理结构，用于尝试执行一段可能会出现异常的代码，如果发生异常则会跳转到对应的异常处理块进行处理，而不会导致程序崩溃
- ◀ `while`
- ◀ `zip()` 用于将多个可迭代对象按对应位置的元素打包成元组的形式，并返回一个新的可迭代对象，常用于并行遍历多个序列





## 7.1 什么是控制结构？

在 Python 中，控制结构是一种用于控制程序流程的结构，包括条件语句、循环语句和异常处理语句。这些结构可以根据不同的条件决定程序运行的路径，并根据需要重复执行代码块或捕获和处理异常情况。

这一节我们用实例全景展示这几种常见的控制结构。

### 条件语句

条件语句在程序中用于根据不同的条件来控制执行不同的代码块。Python 中最常用的条件语句是 `if` 语句，`if` 语句后面跟一个布尔表达式，如果布尔表达式为真，就执行 `if` 语句块中的代码，否则执行 `else` 语句块中的代码。还有 `elif` 语句可以用来处理多种情况。

图 1 是一个简单例子，如果成绩大于等于 60 分，输出“及格”，否则输出“不及格”。图 1 中代码对应的流程图（flowchart）如图 2 所示。

注意，代码中用到了本书第 4 章讲的“四空格”缩进，还用到了上一章讲过的 `>=` 判断运算，忘记的话请回顾。此外，大家在 JupyterLab 练习图 1 给出代码时，注意字符串要用半角引号。

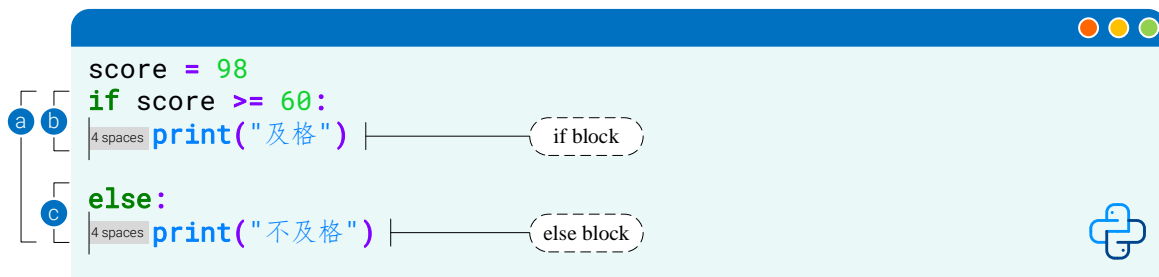


图 1. 用 `if` 判断是否成绩及格； Bk1\_Ch07\_01.ipynb

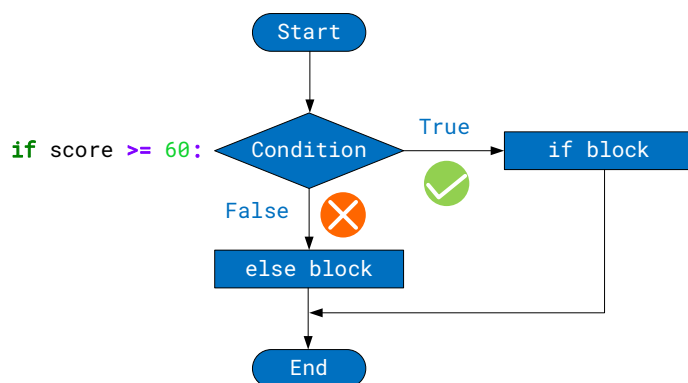


图 2. 用 `if` 判断是否成绩及格，流程图

### for 循环语句

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)

循环语句用于在程序中重复执行相同的代码块，直到某个条件满足为止。Python 中有两种循环语句：for 循环和 while 循环。

本书前文几次使用过 for 循环，相信大家已经不再陌生。简单来说，for 循环通常用于遍历序列，例如列表或字符串。在 for 循环中，代码块会在每个元素上执行一次，直到循环结束。

图 3 代码 <sup>a</sup> 中定义了一个字符串。 <sup>b</sup> 遍历字符串每一个元素，用 print() 打印。

<sup>c</sup> 中 for 循环则迭代列表中所有字符串。

<sup>d</sup> 在 for 循环中嵌套了 if 判断， <sup>e</sup> 判断某个字符串元素是否在 packages\_visual 中。

<sup>f</sup> 则是一个嵌套 for 循环，有两层。外层 for 循环遍历列表中每个字符串。 <sup>g</sup> 中内层 for 循环遍历当前字符串的每个字符。

请大家在 JupyterLab 中自行练习图 3 代码。

```

# 循环字符串内字符
a str_for_loop = 'Matplotlib'
b for str_idx in str_for_loop:
    print(str_idx)

# 循环list中元素
list_for_loop = ['Matplotlib', 'NumPy', 'Seaborn',
                 'Pandas', 'Plotly', 'SK-learn']
c for item_idx in list_for_loop:
    print(item_idx)

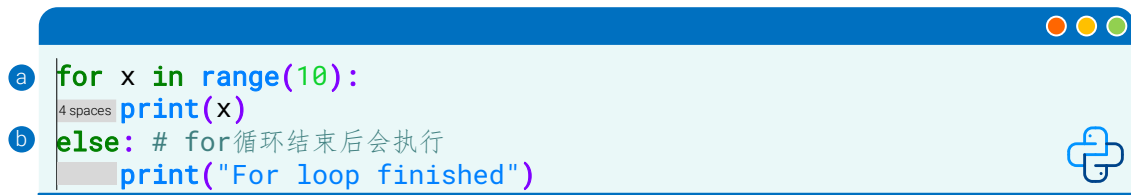
# 循环中嵌入 if 判断
packages_visual = ['Matplotlib', 'Seaborn', 'Plotly']
d for item_idx in list_for_loop:
    print('====')
    print(item_idx)
    e if item_idx in packages_visual:
        print('A visualization tool')

# 嵌套 for 循环
f for item_idx in list_for_loop:
    print('====')
    print(item_idx)
    g for item_idx in item_idx:
        print(item_idx)

```

图 3. 四个 for 循环例子; Bk1\_Ch07\_02.ipynb


此外，for 也可以和 else 结合构成 for ... else ... 语句。在 Python 中，for...else 语句用于在循环结束时执行一些特定的代码。请大家自己在 JupyterLab 中练习图 4。



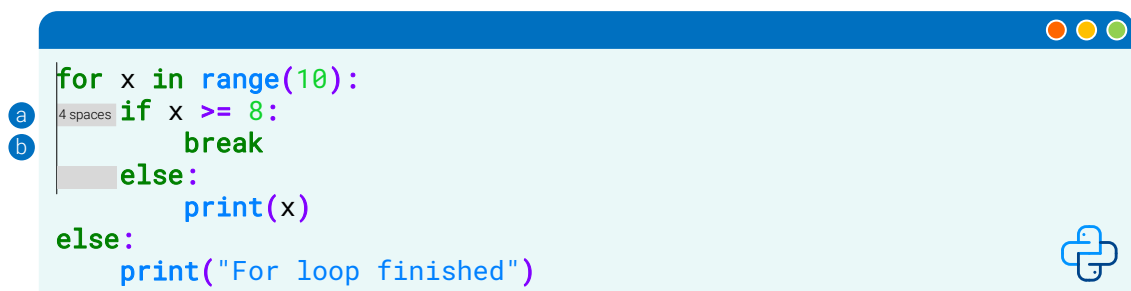
```

a for x in range(10):
  print(x)
b else: # for循环结束后会执行
  print("For loop finished")

```

图 4. for ... else ... 语句例子;  Bk1\_Ch07\_03.ipynb

但是，如图 5 代码所示，当循环被 `break` 语句打断后，`else` 语句中的附加操作不会被执行。



```

a for x in range(10):
  if x >= 8:
    break
  else:
    print(x)
b else:
  print("For loop finished")

```

图 5. for ... else ... 语句例子，`break` 打断 `for` 循环;  Bk1\_Ch07\_04.ipynb

## while 循环语句

`while` 循环会重复执行代码块，直到循环条件不再满足为止。循环条件在每次循环开始前都会被检查。


图 6 给出的例子为使用 `while` 循环输出 0 到 4。`while` 循环相对简单，本书不展开介绍 `while` 循环。



```

a i = 0
b while i < 5:
  print(i)
  i += 1

```

图 6. while 循环例子;  Bk1\_Ch07\_05.ipynb

## 异常处理语句

异常处理语句用于捕获和处理程序中出现的异常情况。Python 中的异常处理语句使用 `try` 和 `except` 关键字，`try` 语句块包含可能引发异常的代码，而 `except` 语句块用于处理异常情况。

图 7 是一个例子，使用 `try` 和 `except` 捕获除数为零的异常。本章不展开讲解 `try ... except`，本书后文用到时再深入探究。

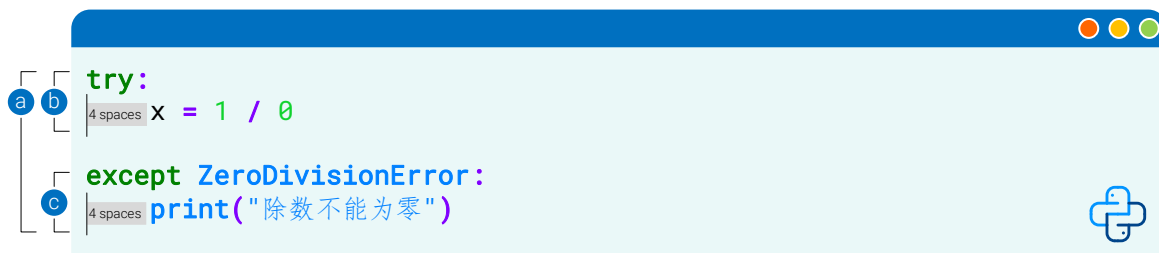


图 7. 用 `try ... except` 捕捉异常; Bk1\_Ch07\_06.ipynb

## 7.2 条件语句：相当于开关

打个比方，条件语句相当于开关。如图 8 (a) 所示，当只有一个 `if` 语句时，它的功能就像是一个单刀单掷开关 (Single Pole Single Throw, SPST)。如果条件满足，就执行分支中相应的代码。

如图 8 (b) 所示，`if-else` 语句相当于单刀双掷开关 (Single Pole Double Throw, SPDT)。当条件语句中分别由 `if` 和 `else` 两个分支，根据条件的真假，可以有两个选项来执行不同的操作。

如图 8 (c) 所示，`if-elif-else` 语句相当于单刀三掷开关 (Single Pole Triple Throw, SPTT)，有三个不同选择。

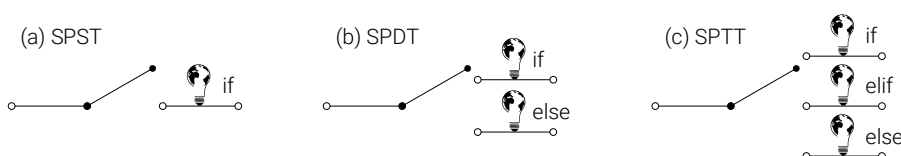


图 8. 不同开关

### 嵌套 `if` 判断

大家可能好奇，如果图 1 中赋值能否为用户输入？此外，如果用户输入错误是否有提示信息？我们当然可以在图 2 基础上用多层判断完成这些需求。图 9 给出的代码可以完成上述要求，对应的流程图如图 10 所示。

在图 9 这段代码中，首先 <sup>a</sup> 使用 `input()` 函数获取用户输入的数值，并将其存储在 `value` 变量中。<sup>b</sup> 是最外层 `if` 判断，使用 `isdigit()` 方法检查输入是否是一个数值。如果是数值，则执行 `if` 语句块内的代码。将数值转换为整数类型，并存储在 `number` 变量中。如果输入不是一个数值，将打印“输入不是一个数值”。

c 使用嵌套的 if 语句来检查 number 是否在 0~100 之间。如果在该范围内，则继续执行内部的 if 语句块。如果输入的数值不在 0 ~ 100 之间，将打印"数值不在 0 ~ 100 之间"。

d 在内部的 if 语句块中，判断 number 是否小于 60。如果小于 60，则打印"不及格"；否则打印"及格"。

? 请大家思考图 9 中代码第一层 if 对应的代码块是什么？

▲ 注意，上述代码假设用户输入的数值为整数。如果需要支持浮点数，请相应地调整代码。

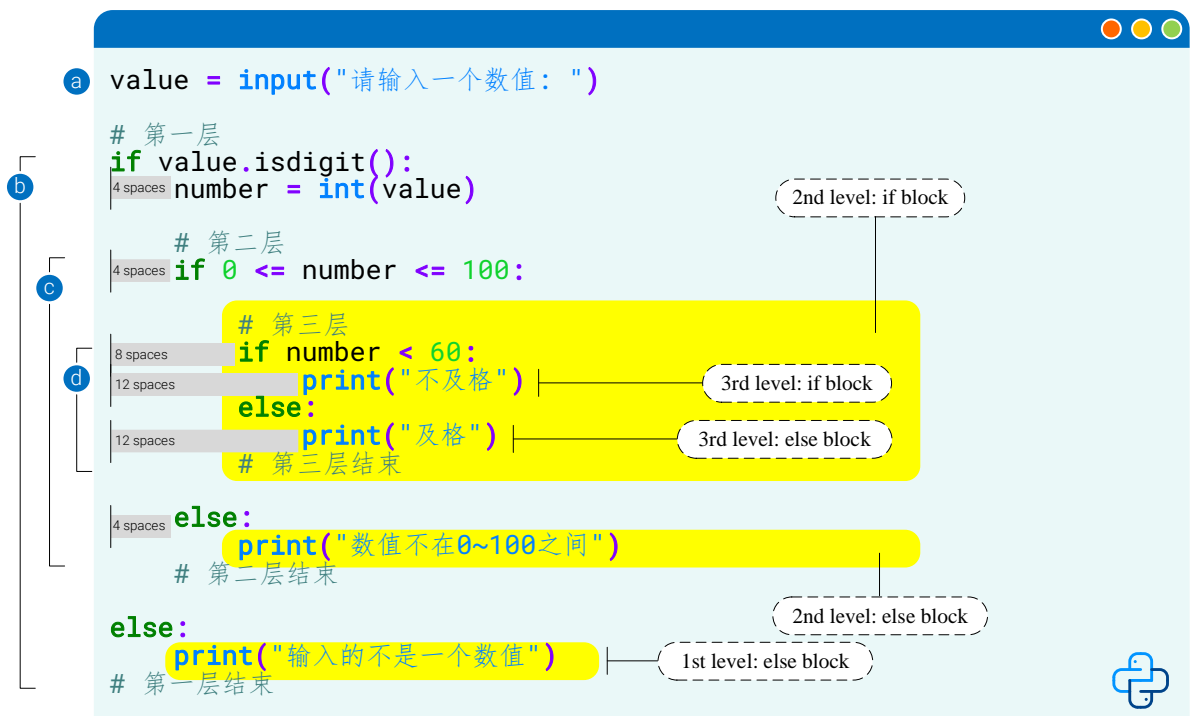
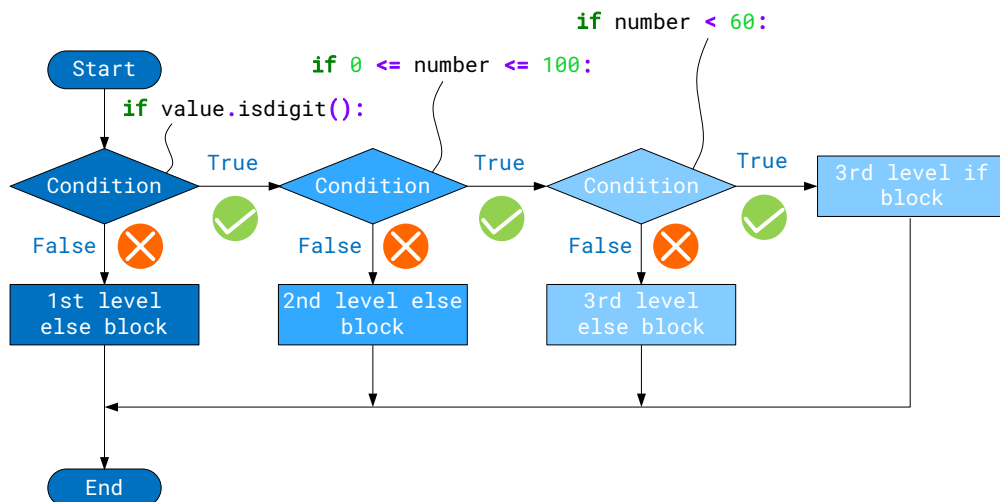


图 9. 用 if 判断是否成绩及格，三层判断； Bk1\_Ch07\_07.ipynb



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)

图 10. 用 if 判断是否成绩及格，三层判断，流程图

### if...elif...else 语句

if...elif...else 语句用于判断多个条件，如果第一个条件成立，则执行 if 语句中的代码块；如果第一个条件不成立，但第二个条件成立，则执行 elif 语句中的代码块；如果前面的条件都不成立，则执行 else 语句中的代码块。

⚠ 注意，elif 的语句数量没有上限。但是，如果代码中 elif 数量过多，需要考虑简化代码结构。

图 11 代码判断一个数是正数、负数还是 0，请大家自己分析这段代码，运行并逐行注释。

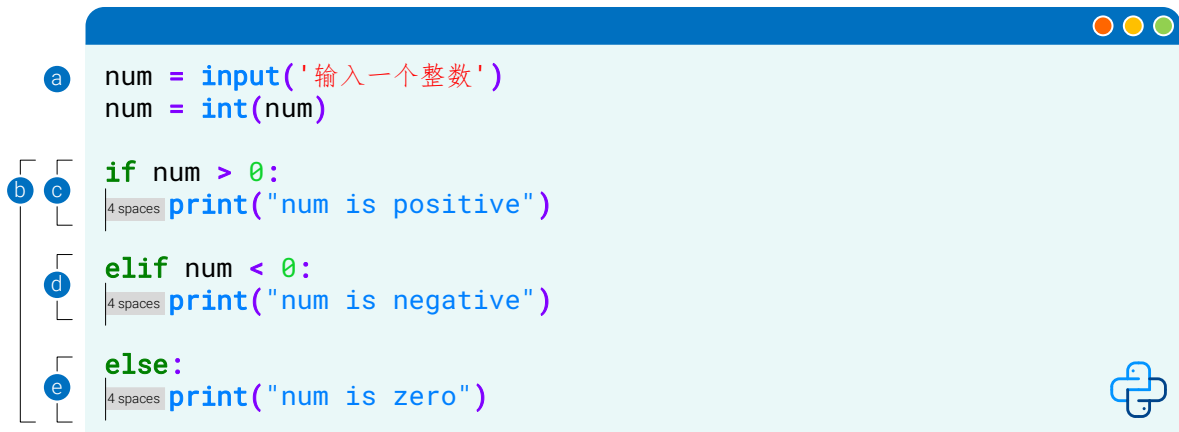


图 11. if...elif...else 语句; Bk1\_Ch07\_08.ipynb

### break、continue、pass 语句

在 Python 的 if 条件语句、for 循环语句中，可以使用 break、continue 和 pass 来控制循环的行为。

break 语句可以用来跳出当前循环。当循环执行到 break 语句时，程序将立即跳出循环体，继续执行循环外的语句。图 12 是一个使用 break 的例子，该循环会在 i 等于 3 时跳出。

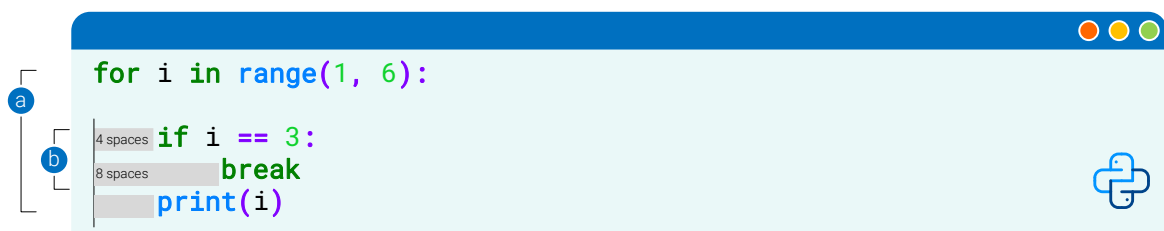


图 12. for 循环中使用 break; Bk1\_Ch07\_09.ipynb

continue 语句可以用来跳过当前循环中的某些语句。当循环执行到 continue 语句时，程序将立即跳过本次循环，继续执行下一次循环。图 13 是一个使用 continue 的例子，该循环会在 i 等于 3 时跳过本次循环。



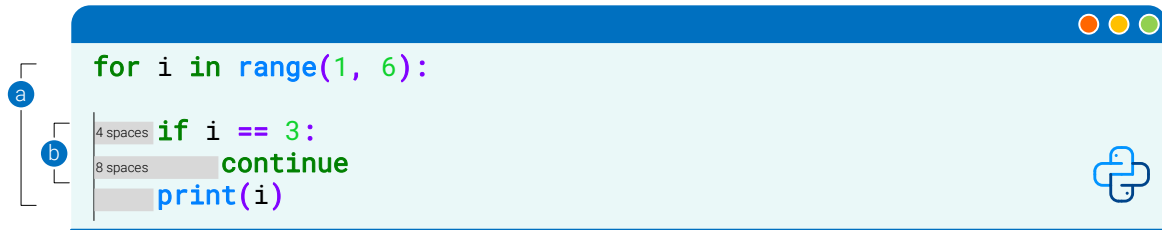


图 13. for 循环中使用 continue; Bk1\_Ch07\_09.ipynb

`pass` 语句什么也不做，它只是一个空语句占位符。在需要有语句的地方，但是暂时不想编写任何语句时，可以使用 `pass` 语句。图 14 是一个使用 `pass` 的例子，该循环中的所有元素都会被输出。

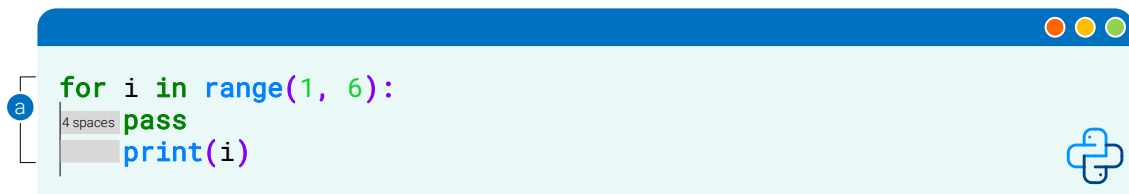


图 14. for 循环中使用 pass; Bk1\_Ch07\_09.ipynb

## 7.3 for 循环语句

本节介绍 `for` 循环一些常见用法，并复刻一些常见线性代数运算法则。希望这些练习帮助大家更好掌握 `for` 循环，以及线性代数运算法则。

### 计算向量内积

下例展示如何利用 `for` 循环计算向量内积。我们用两个 `list` 代表向量，两个 `list` 中元素都是整数，而且 `list` 中元素数量一致。

图 15 代码 <sup>a</sup> 用 `for` 循环遍历 `list` 索引。其中，`len(a)` 计算向量 `a` 元素数量，`range(len(a))` 创建一个包含从 0 到 `len(a)-1` 的整数可迭代对象。大家用 `type()` 可以发现，`range()` 函数产生的对象类型就是 `range`。

想要看到 `range` 中的具体整数，可以用 `list(range(len(a)))`。

<sup>b</sup> 计算 `a` 和 `b` 对应元素的乘积，然后逐项求和，结果就是向量内积。

当然，在实际应用中，我们会利用 `NumPy` 库计算向量内积，不会使用图 15 这种方式。但是，图 15 代码可以帮助我们理解 `for` 循环，以及向量内积的运算规则。

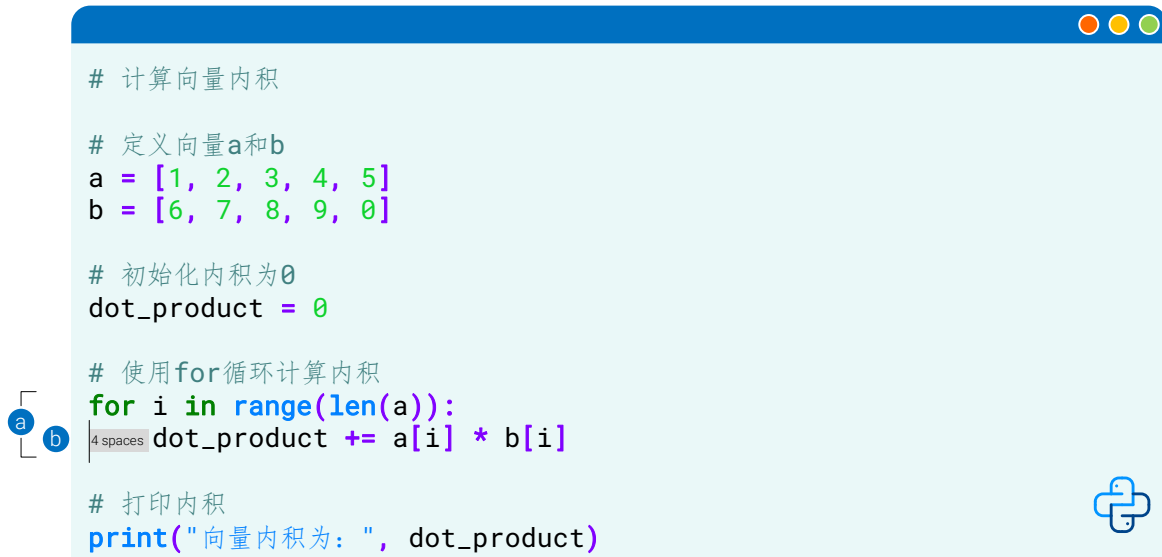


图 15. 计算向量内积; Bk1\_Ch07\_10.ipynb



### 什么是向量内积?

向量内积 (inner product), 也称为点积 (dot product)、标量积 (scalar product), 是在线性代数中常见的一种运算, 它是两个向量之间的一种数学运算。

给定两个  $n$  维向量  $\mathbf{a} = [a_1, a_2, \dots, a_n]$  和  $\mathbf{b} = [b_1, b_2, \dots, b_n]$ , 它们的内积定义为  $\mathbf{a} \cdot \mathbf{b} = a_1b_1 + a_2b_2 + \dots + a_nb_n$ 。这个公式的意义是将两个向量的对应分量相乘, 然后将乘积相加, 从而得到它们的内积。

例如, 如果有两个二维向量分别为  $\mathbf{a} = [1, 2]$  和  $\mathbf{b} = [3, 4]$ , 则它们的内积为:  $\mathbf{a} \cdot \mathbf{b} = 1 \times 3 + 2 \times 4 = 11$ 。向量内积的结果是一个标量, 也就是一个值, 而不是向量。它可以用来计算向量之间的夹角, 衡量它们的相似性, 以及用于向量空间的正交分解等。

在实际应用中, 向量内积被广泛用于机器学习、计算机视觉、信号处理、物理学等领域。在机器学习中, 向量内积常用于计算特征之间的相似度, 从而进行分类、聚类任务。在计算机视觉中, 向量内积可以用于计算两个图像之间的相似度。

*fx*

### range(start [, stop, step])

range() 是 Python 内置的函数, 用于生成一个整数序列, 常用于 for 循环中的计数器。参数为:

- start 是序列起始值;
- stop 是序列结束值 (不包含);
- step 是序列中相邻两个数之间的步长 (默认为 1)。

range() 函数生成的是一个可迭代对象, 而不是一个列表。这样做的好处是, 可以节省内存空间, 尤其在需要生成很长的序列时。

下面是一些使用 range() 函数的示例:

a) 生成从 0 到 4 的整数序列

```
for i in range(4 + 1):
    print(i)
```

b) 生成从 10 到 20 的整数序列

```
for i in range(10, 20 + 1):
    print(i)
```

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)

c) 生成从 1 到 10 的奇数序列

```
for i in range(1, 10 + 1, 2):
    print(i)
```

d) 生成从 10 到 1 的倒序整数序列

```
for i in range(10, 1 - 1, -1):
    print(i)
```

d) 将 range() 生成的可迭代对象变成 list:

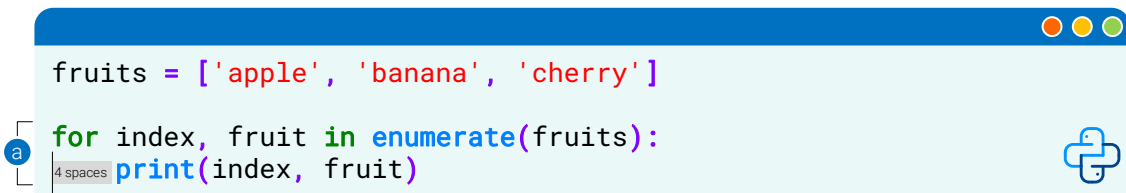
```
list(range(10, 1 - 1, -1))
```

请大家在 JupyterLab 中自行运行如上几段代码。

### 使用 enumerate()

在 Python 中，`enumerate()` 是一个用于在迭代时跟踪索引的内置函数。`enumerate()` 函数可以将一个可迭代对象转换为一个由索引和元素组成的枚举对象。

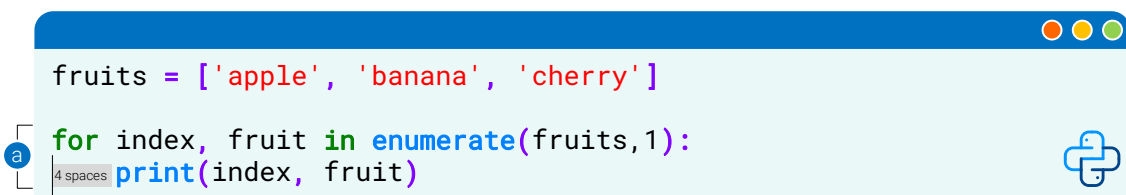
图 16 是一个简单的例子，展示了如何在 `for` 循环中使用 `enumerate()` 函数。在这个例子中，`fruits` 列表中的每个元素都会被遍历一遍，每次遍历都会获得该元素的值和其在列表中的索引。这些值分别被赋给 `index` 和 `fruit` 变量，并打印输出。



```
fruits = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(fruits):
    print(index, fruit)
```

图 16. 使用 `enumerate()`，从 0 开始编号； Bk1\_Ch07\_11.ipynb

需要注意的是，`enumerate` 函数的默认起始编号为 0（索引），但是也可以通过传递第二个参数来指定起始编号。例如，如果想要从 1 开始编号，可以使用图 17 代码。



```
fruits = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(fruits, 1):
    print(index, fruit)
```

图 17. 使用 `enumerate()`，从 1 开始编号； Bk1\_Ch07\_11.ipynb

### 使用 zip()

在 Python 中，`zip()` 函数可以将多个可迭代对象的元素组合成元组，然后返回这些元组组成的迭代器。在 `for` 循环中使用 `zip()` 函数可以方便地同时遍历多个可迭代对象。

特别地，当这些可迭代对象的长度不同时，`zip()` 函数会以最短长度的可迭代对象为准进行迭代。

如果想要打印出每个学生的姓名和对应的成绩，可以使用 `zip()` 函数和 `for` 循环，代码如图 18 所示。

在这个例子中，`zip()` 函数将 `names` 和 `scores` 两个列表按照位置进行组合，然后返回一个迭代器，其中的每个元素都是一个元组，元组的第一个元素为 `names` 列表中对应位置的元素，第二个元素为 `scores` 列表中对应位置的元素。在 `for` 循环中使用了两个变量 `name` 和 `score`，分别用来接收每个元组中的两个元素，然后打印出来即可。

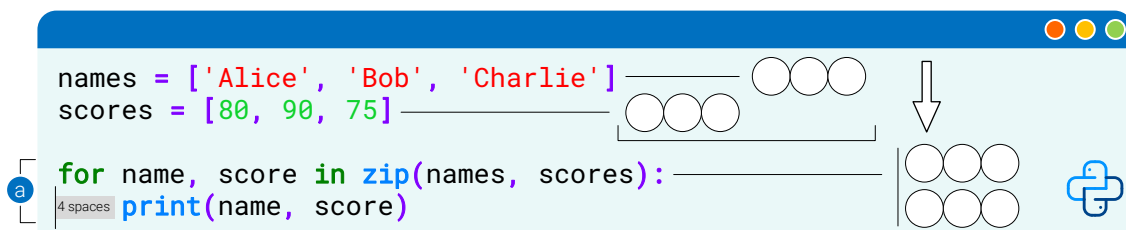


图 18. 使用 `zip()` 同步遍历多个对象； Bk1\_Ch07\_12.ipynb

刚刚提过，如果可迭代对象的长度不相等，`zip()` 函数会以长度最短的可迭代对象为准进行迭代。如果想要以长度最长的可迭代对象为准进行迭代，可以用 `itertools.zip_longest(*iterables, fillvalue=None)`。缺失元素默认以 `None` 补齐，或者以用户指定值补齐。图 19 比较这两种方法。

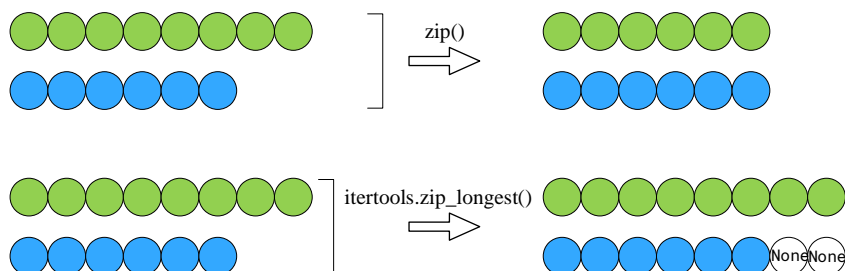


图 19. 比较 `zip()` 和 `itertools.zip_longest()`

### 计算向量内积：使用 `zip()`

在计算向量内积时，我们也可以使用 Python 的内置函数 `zip()` 和 `for` 循环，对两个向量中的对应元素逐一相乘并相加，实现向量内积运算。

在图 20 示例中，通过 `zip()` 函数将两个 `list`，`a` 和 `b`，中对应位置的元素组合成了元组，然后使用 `for` 循环逐个遍历并相乘求和，最终得到了向量内积的结果。请大家对比图 15。

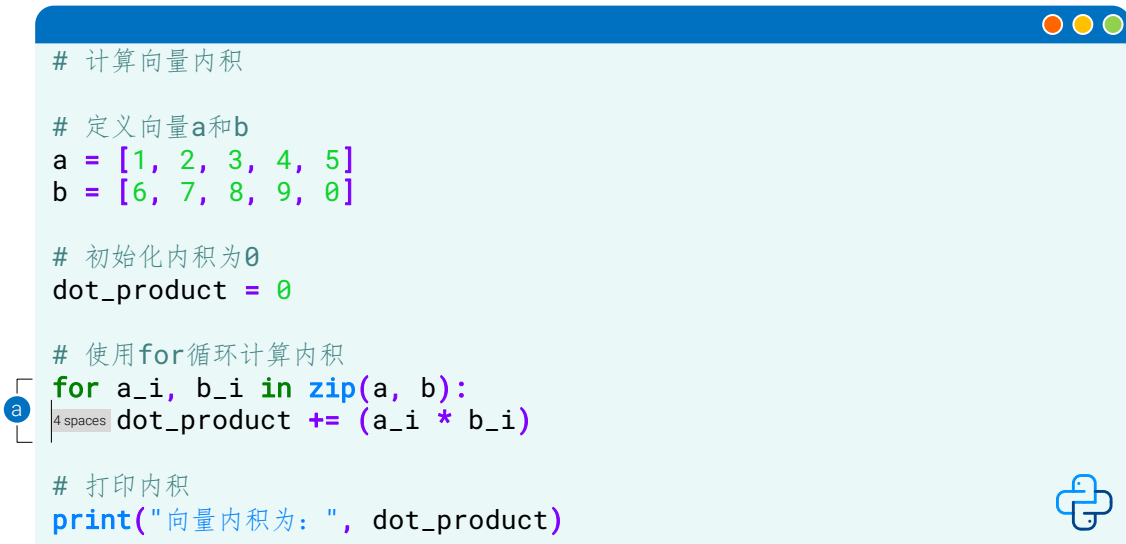


图 20. 使用 zip() 计算向量内积; Bk1\_Ch07\_13.ipynb

## 生成二维坐标

图 21 介绍如何分离一组二维平面直角坐标系横纵坐标。观察这幅图，我们可以发现横纵坐标好比织布的经线纬线。两者串在一起构成了整个平面的坐标。

图 22 代码展示如何用两层 for 循环实现图 21。

- a 用 def 定义了一个名为 custom\_meshgrid 函数。函数的输入为 x 和 y。本书下一章将专门介绍如何自定义函数。
- b 分别计算 x 和 y 两个列表的长度。
- c 定义了两个空列表，X 和 Y，用来储存横纵轴坐标。
- d 外层 for 循环用来遍历列表 y 的索引 i。
- e 内层 for 循环用来遍历列表 x 的索引 j。
- f 在每次内层循环中，将列表 x 中索引为 j 的元素添加到 X\_row 中，同时将列表 y 中索引为 i 的元素添加到 Y\_row 中。
- g 生成二维数组。在外层 for 循环的每次迭代结束时，将 X\_row 添加到二维数组 X 中，将 Y\_row 添加到二维数组 Y 中。这样，最终得到的 X 和 Y 就是由 x 和 y 列表生成的二维数组。
- h 定义了列表 x，代表一组横坐标。
- i 定义了列表 y，代表一组纵坐标。
- j 调用自定义函数，生成二维网格坐标。

用过 NumPy 库的同学可能已经发现，这段代码实际上在复刻 `numpy.meshgrid()`。当然，`numpy.meshgrid()` 要比我们自定义函数强大得多，它可以创建多维坐标数组。在本书后续很多可视化方案中都会用到 `numpy.meshgrid()` 函数，请大家务必理解它的原理。

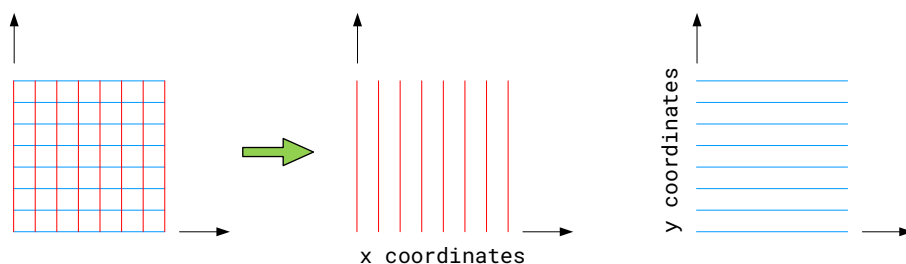


图 21. 分离横轴、纵轴坐标;

```

# 自定义函数
a def custom_meshgrid(x, y):
b     num_x = len(x); num_y = len(y)
c     X = []; Y = []
    # 外层for循环
d     for i in range(num_y):
        X_row = []; Y_row = []
        # 内层for循环
e         for j in range(num_x):
f             X_row.append(x[j])
            Y_row.append(y[i])
        # 生成二维数组
g         X.append(X_row); Y.append(Y_row)

    return X, Y

# 示例用法
h x = [0, 1, 2, 3, 4, 5] # 横坐标列表
i y = [0, 1, 2, 3]      # 纵坐标列表
# 调用自定义函数
j X, Y = custom_meshgrid(x, y)
print("X坐标: "); print(X)
print("Y坐标: "); print(Y)

```

图 22. 分离纵横网格坐标; Bk1\_Ch07\_14.ipynb

### 矩阵乘法：三层 for 循环

下面介绍如何使用嵌套 for 循环完成矩阵乘法。

图 23 所示为矩阵乘法规则示意图。

矩阵  $A$  的第一行元素和矩阵  $B$  第一列对应元素分别相乘，再相加，结果为矩阵  $C$  的第一行、第一列元素  $c_{1,1}$ 。

矩阵  $A$  的第一行元素和矩阵  $B$  第二列对应元素分别相乘，再相加，得到  $c_{1,2}$ 。

同理，依次获得矩阵  $C$  剩余元素。

为了完成矩阵乘法运算，我们设计了图 23 这种三个 for 循环嵌套的方法。

第一层 for 循环遍历矩阵  $A$  的行，第二层 for 循环遍历矩阵  $B$  的列，第三层 for 循环完成“逐项乘积 + 求和”运算。

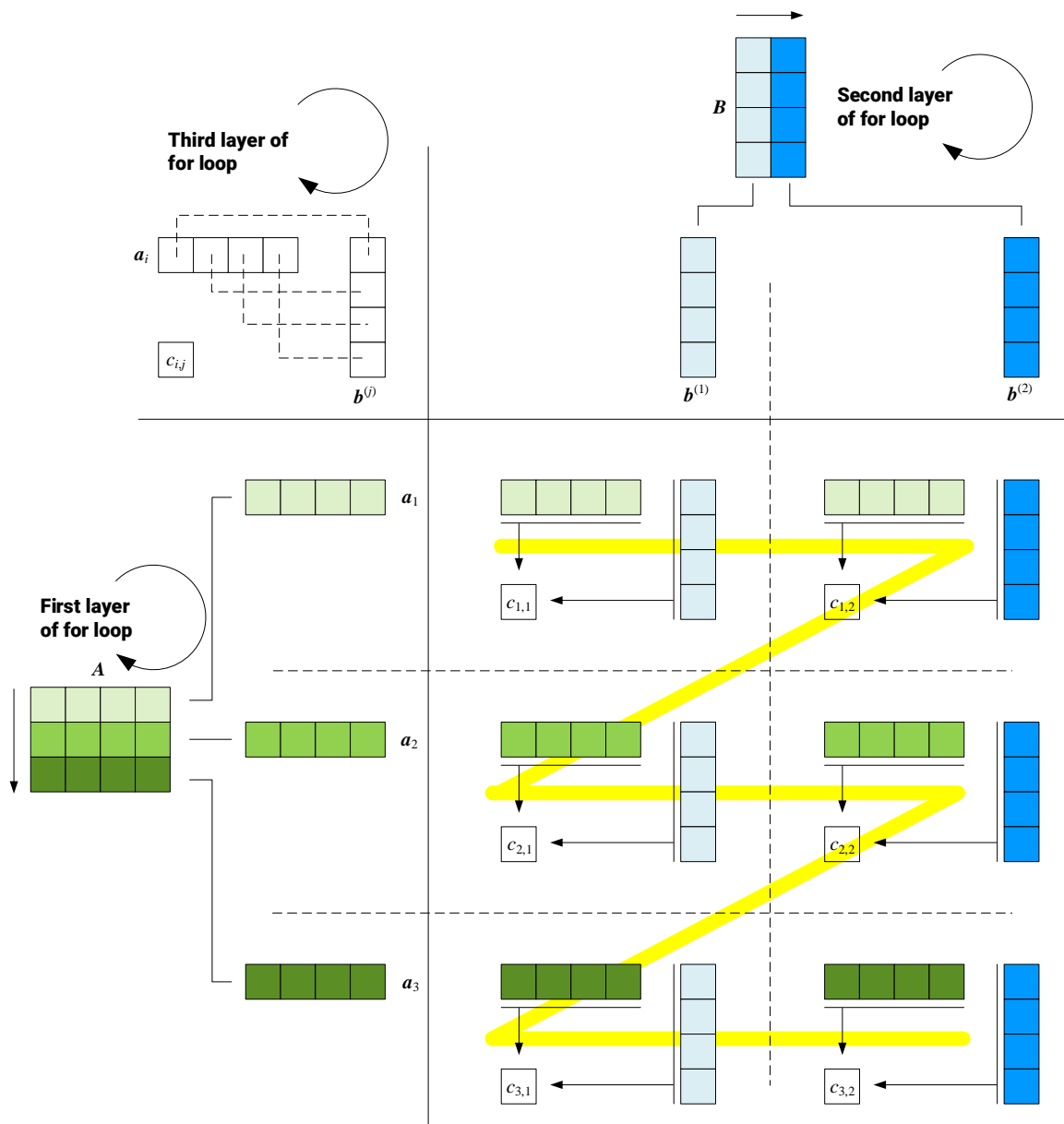
本 PDF 文件为作者草稿，发布目的为方便读者在移动终端学习，终稿内容以清华大学出版社纸质出版物为准。

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)

图 23. 矩阵乘法规则，利用三层 for 循环实现；最外层遍历矩阵  $A$  的行，第二层遍历矩阵  $B$  的列

### 什么是矩阵乘法？

矩阵乘法 (matrix multiplication) 是一种线性代数运算，用于将两个矩阵相乘。对于两个矩阵  $A$  和  $B$ ，它们的乘积  $AB$  的元素是通过将  $A$  的每一行与  $B$  的每一列进行内积运算得到的。

具体而言，假设  $A$  是一个  $m \times n$  的矩阵， $B$  是一个  $n \times p$  的矩阵，则它们的乘积  $C = AB$  是一个  $m \times p$  的矩阵，其中第  $i$  行第  $j$  列的元素  $c_{i,j}$  为  $A$  的第  $i$  行与  $B$  的第  $j$  列的内积。如果  $A$  的第  $i$  行元素为  $a_{i,1}, a_{i,2}, \dots, a_{i,n}$ ， $B$  的第  $j$  列元素为  $b_{1,j}, b_{2,j}, \dots, b_{n,j}$ ，则  $C = AB$  的第  $i$  行第  $j$  列的元素为  $a_{i,1}b_{1,j} + a_{i,2}b_{2,j} + \dots + a_{i,n}b_{n,j}$ 。

矩阵乘法在许多领域都有广泛的应用，例如线性代数、信号处理、图形学和机器学习等。在机器学习中，矩阵乘法通常用于计算神经网络的前向传播过程，其中输入矩阵与权重矩阵相乘，得到隐藏层的输出矩阵。

图 24 代码实现图 23 所示矩阵乘法法则，请大家自行分析这段代码，运行并逐行注释。

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)

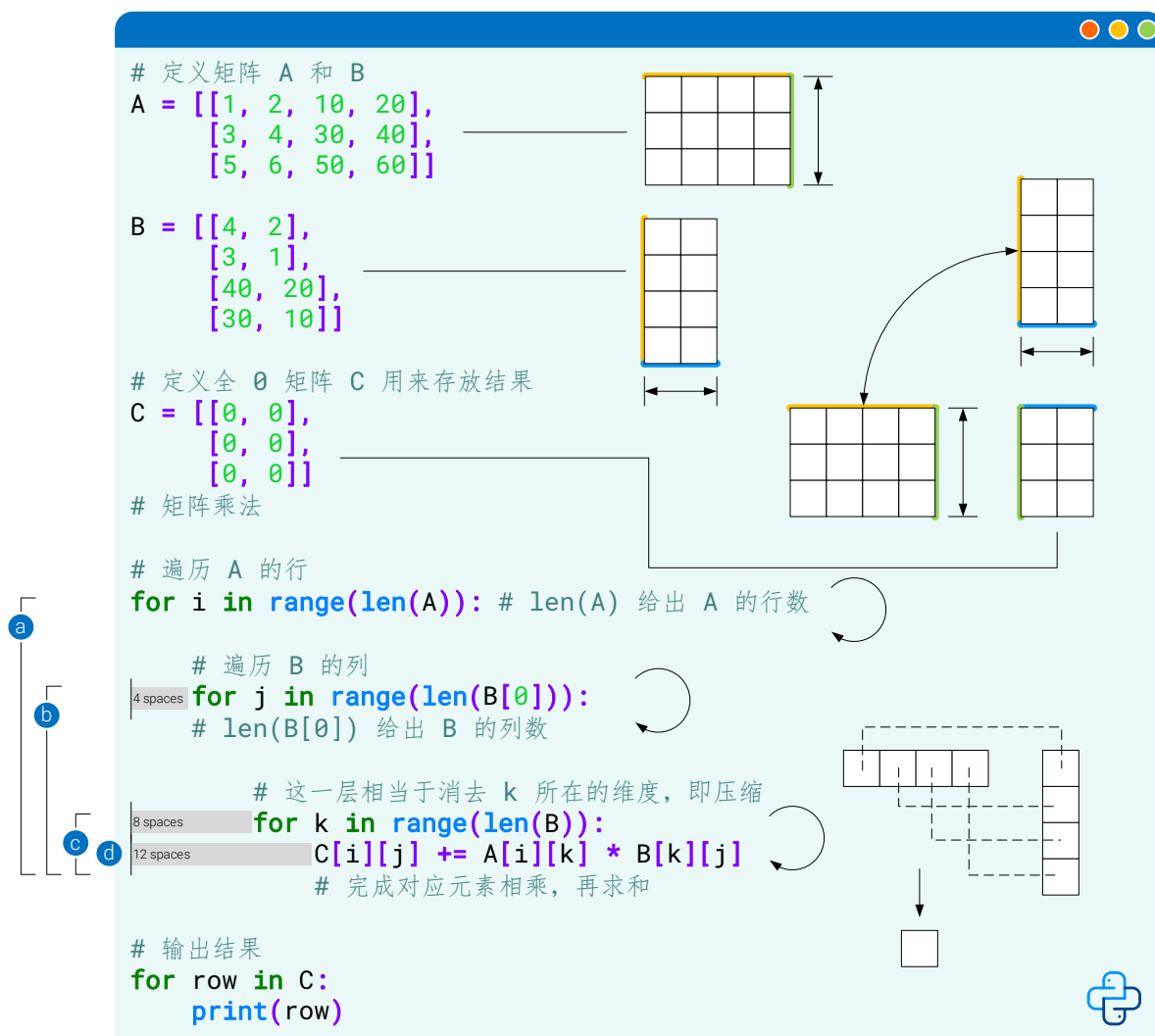


图 24. 使用嵌套 for 循环计算矩阵乘法; Bk1\_Ch07\_15.ipynb

相比图 23，图 25 所示矩阵乘法法则交换的第一二层 for 循环顺序，请大家根据图 25 修改图 24 代码。



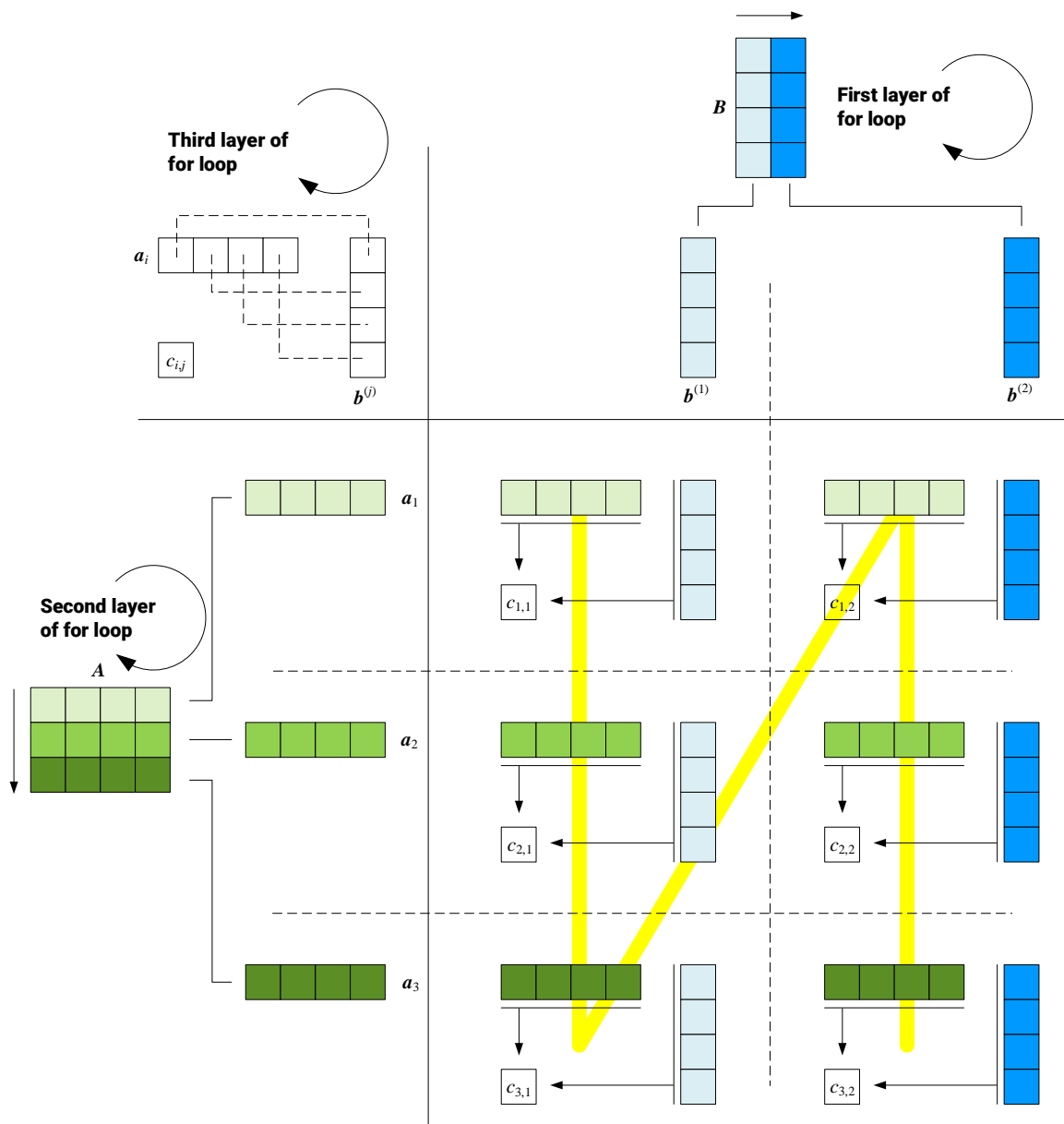


图 25. 矩阵乘法规则，利用三层 for 循环实现；最外层遍历矩阵  $B$  的列，第二层遍历矩阵  $A$  的行

## 向量化

向量化运算是使用 NumPy 等库的一种高效运算处理方式，可以避免使用 for 循环。图 26、图 27 所示为利用 NumPy 完成向量内积、矩阵乘法运算。

图 26 中 **a** 将 numpy 导入，简作 np。

**b** 用 `numpy.ndarray()`，简作 `np.array()`，构造一维数组。类似地，**c** 构造了第二个一维数组。

**d** 用 `numpy.dot()`，简作 `np.dot()`，计算  $a$  和  $b$  的内积。

```

a import numpy as np

# 定义向量a和b; 准确来说是一维数组
b a = np.array([1, 2, 3, 4, 5])
c b = np.array([6, 7, 8, 9, 0])

# 计算向量内积
d dot_product = np.dot(a,b)

# 打印内积
print("向量内积为: ", dot_product)

```

图 26. 使用 `numpy.dot()` 计算向量内积; Bk1\_Ch07\_16.ipynb

图 27 中 **a** 用 `numpy.array()`, 简作 `np.array()`, 定义二维数组 A。这个数组有 2 行、4 列。  
**b** 用同样的方法定义二维数组 B, 形状为 4 行、2 列。  
**c** 用 `@` 运算符计算 A 和 B 乘积, 结果为二维数组, 形状为 2 行、2 列。  
**d** 用 `@` 运算符计算 B 和 A 乘积, 结果也为二维数组, 形状为 4 行、4 列。  
 显然, `A*B` 不同于 `B*A`。这一点本书第 25 章还要提及。

```

import numpy as np

# 定义矩阵 A 和 B
a A = np.array([[1, 2, 10, 20],
                [3, 4, 30, 40]])
b B = np.array([[1, 3],
                [2, 4],
                [10, 30],
                [20, 40]])

c C = A @ B; print(C)
d D = B @ A; print(D)

```

图 27. 使用 NumPy 计算矩阵乘法; Bk1\_Ch07\_17.ipynb

请大家在 JupyterLab 中练习图 26 和图 27 这两段代码, 并逐行注释。

再次强调, 有了 NumPy 库, 不意味着前文自己写代码计算向量内积、矩阵乘法是无用功! 在前文的代码练习中, 一方面我们掌握如何使用 `for` 循环, 此外理解了向量内积、矩阵乘法两种数学工具的运算规则。本章及下一章还会介绍更多线性代数运算法则, 并和大家探讨如何写代码实现这些运算。

## 7.4 列表生成式

本书前文介绍过如何用列表生成式创建列表，本节深入介绍列表生成式。

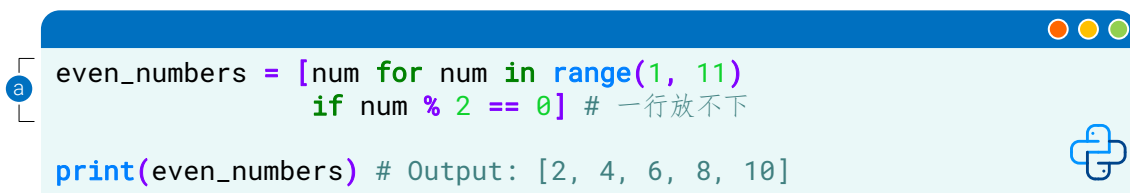
在 Python 中，列表生成式 (list comprehension) 是一种简洁的语法形式，用于快速生成新的列表。

它的语法形式为 `[expression for item in iterable if condition]`，其中 `expression` 表示要生成的元素，`item` 表示迭代的变量，`iterable` 表示迭代的对象，`if condition` 表示可选的过滤条件。

举个例子，假设我们想要生成一个包含 1 到 10 之间所有偶数的列表，我们可以使用图 28 代码中列表生成式完成运算。

`for num in range(1, 11)` 这部分定义了一个 `for` 循环，遍历范围在 1 到 10（不包括 11）之间的整数。`num` 是循环中的变量，它依次取遍这个范围内的每个数字。

在循环中，使用条件语句 `if num % 2 == 0` 来筛选偶数。`num % 2 == 0` 表示数字 `num` 除以 2 的余数为 0，即 `num` 是偶数。只有满足这个条件的数字才会被包含在生成的列表中。

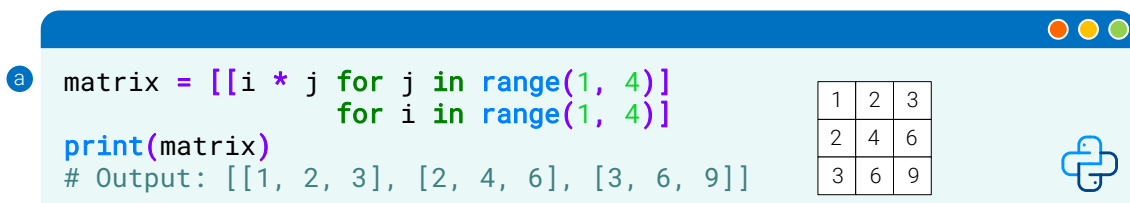


```
even_numbers = [num for num in range(1, 11)
                 if num % 2 == 0] # 一行放不下
print(even_numbers) # Output: [2, 4, 6, 8, 10]
```

图 28. 使用列表生成式，获得 1 ~ 10 之间所有偶数列表； Bk1\_Ch07\_18.ipynb

使用列表生成式还可以嵌套，比如图 29 代码示例。

在代码中，我们使用嵌套的列表生成式创建了一个  $3 \times 3$  的矩阵。具体来说，我们使用外部的列表生成式迭代 1 到 3 的数字，对每个数字使用内部的列表生成式迭代 1 到 3 的数字，计算它们的乘积并将结果存储到一个新的二维列表中。请大家用上述代码生成图 24 中全 0 矩阵 `C`。



```
matrix = [[i * j for j in range(1, 4)]
           for i in range(1, 4)]
print(matrix)
# Output: [[1, 2, 3], [2, 4, 6], [3, 6, 9]]
```

1	2	3
2	4	6
3	6	9

图 29. 嵌套列表生成式； Bk1\_Ch07\_18.ipynb

使用列表生成式可以大大简化代码，提高代码的可读性和可维护性。

### 复刻 `numpy.linspace()`

本书后续在绘制二维线图时，经常会使用 `numpy.linspace()` 生成颗粒度高的等差数列。下面，我们就利用列表生成式来复刻这个函数的基本功能。

图 30 中 **a** 自定义函数，叫做 `linspace`，函数有 4 个输入，`start`，`stop`，`num`，`endpoint`。其中，`start` 代表等差数列的起始值，`stop` 代表等差数列的结束值。特别注意，`num` 和 `endpoint` 有各自的默认值。也就是说，在调用自定义函数时，如果 `num` 和 `endpoint` 这两个参数缺省时，就会使用默认值。本书下一章将深入介绍这一点。

**b** 判断 `num` 是否小于 2。如果条件为真，即 `num` 小于 2，**c** 会被执行。其中，`raise` 用于引发一个异常，这里是引发 `ValueError` 异常，异常的消息是 "Number of samples must be at least 2"。这意味着如果 `num` 小于 2，程序将引发一个值错误，并且程序的执行将停止，错误消息将被打印出来。

**d** 用条件语句检查 `endpoint` 是否为 `True`。如果 `endpoint` 为 `True`，则执行 **e**，即等差数列包含 `stop` 值，所以步长为 `step = (stop - start) / (num - 1)`。

**f** 用列表生成式生成等差数列并返回。

**g** 为 `if ... else` 中的 `else` 语句。如果 `endpoint` 不为 `True`，**h** 中计算步长时，用 `(stop - start) / num`。

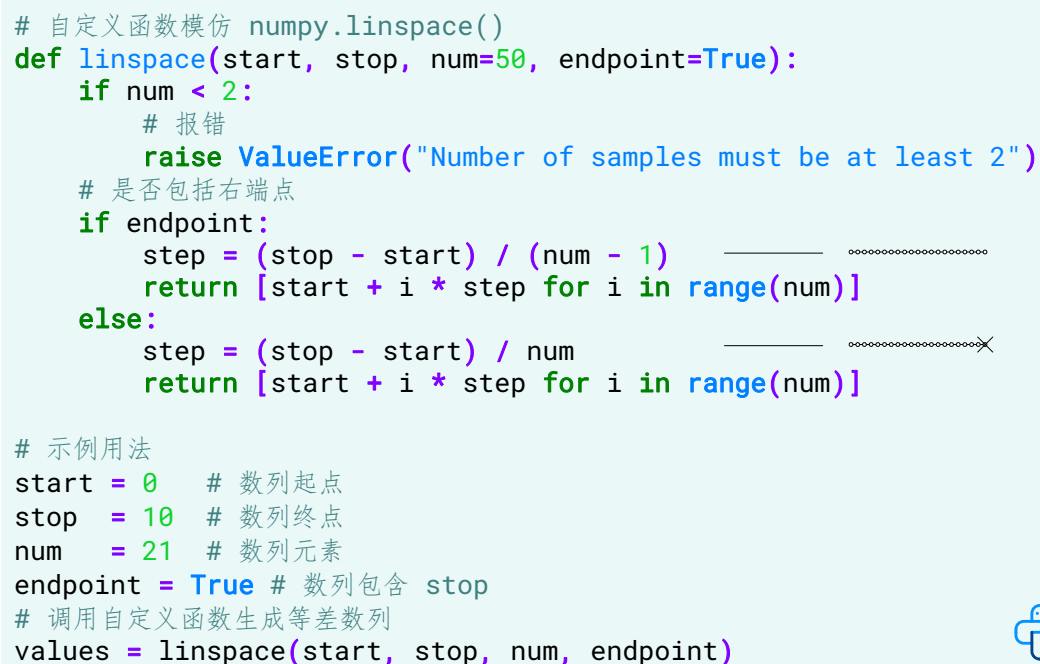
**i** 也是用列表生成式构造等差数列并返回。

请大家按如下要求修改图 30 代码。

a) 加一段判断 `start` 和 `stop` 大小。如果 `start` 和 `stop` 相等，则报错。

b) 判断 `num` 为不小于 2 的正整数。

c) 将 **f** 和 **i** 两个列表生成式合并成一句。



```
# 自定义函数模仿 numpy.linspace()
a def linspace(start, stop, num=50, endpoint=True):
b     if num < 2:
c         # 报错
         raise ValueError("Number of samples must be at least 2")
         # 是否包括右端点
d     if endpoint:
e         step = (stop - start) / (num - 1)
f         return [start + i * step for i in range(num)]
g     else:
h         step = (stop - start) / num
i         return [start + i * step for i in range(num)]

# 示例用法
start = 0 # 数列起点
stop = 10 # 数列终点
num = 21 # 数列元素
endpoint = True # 数列包含 stop
# 调用自定义函数生成等差数列
values = linspace(start, stop, num, endpoint)
```

图 30. 复刻 `numpy.linspace()` 基本功能; Bk1\_Ch07\_19.ipynb

### 矩阵转置：一层列表生成式

如下代码展示如何用一层列表生成式转置矩阵（the transpose of a matrix）。

本书前文介绍过矩阵转置运算法则。简单来说，矩阵转置是指将矩阵的行和列互换的操作。如果一个矩阵  $A$ ，其形状为  $m \times n$ ，即  $m$  行  $n$  列，那么矩阵  $A$  的转置，记作  $A^T$ ，的形状为  $n \times m$ ，即  $n$  行  $m$  列。在转置后的矩阵中，原矩阵的行变成了新矩阵的列，原矩阵的列变成了新矩阵的行。

从运算角度来看，对于原矩阵中的位置为  $[i][j]$  元素，将其放置到转置后位置  $[j][i]$ 。据此规则，请大家自行分析图 31 代码。

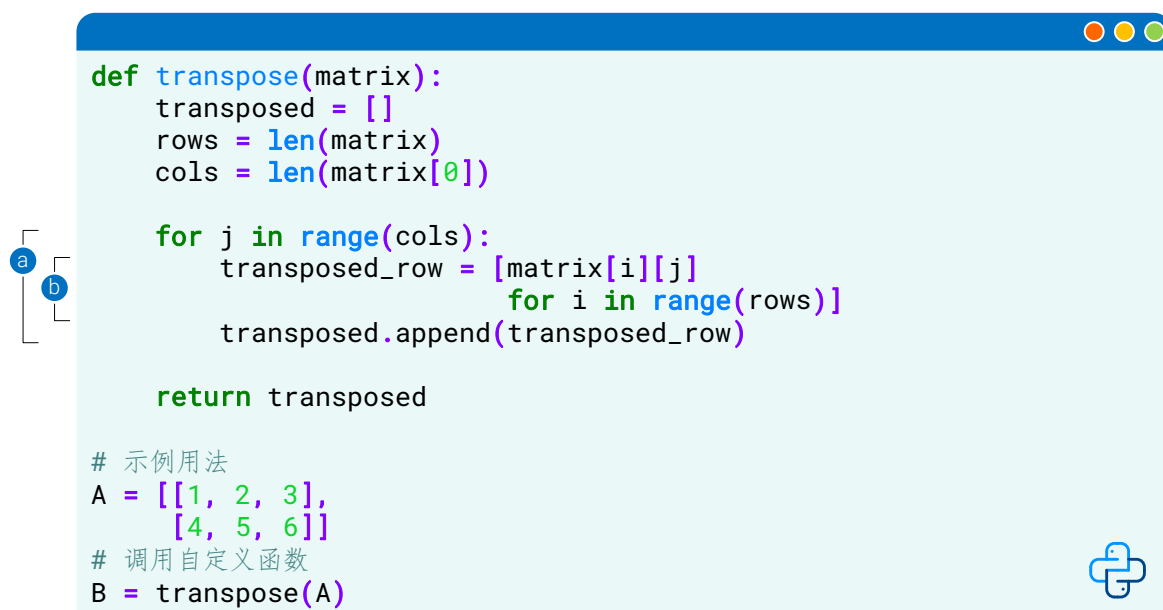


图 31. 利用一层列表生成式完成矩阵转置; Bk1\_Ch07\_20.ipynb

### 矩阵转置：两层列表生成式

图 32 代码展示如何用两层列表生成式转置矩阵，也请大家自行分析并逐行注释。

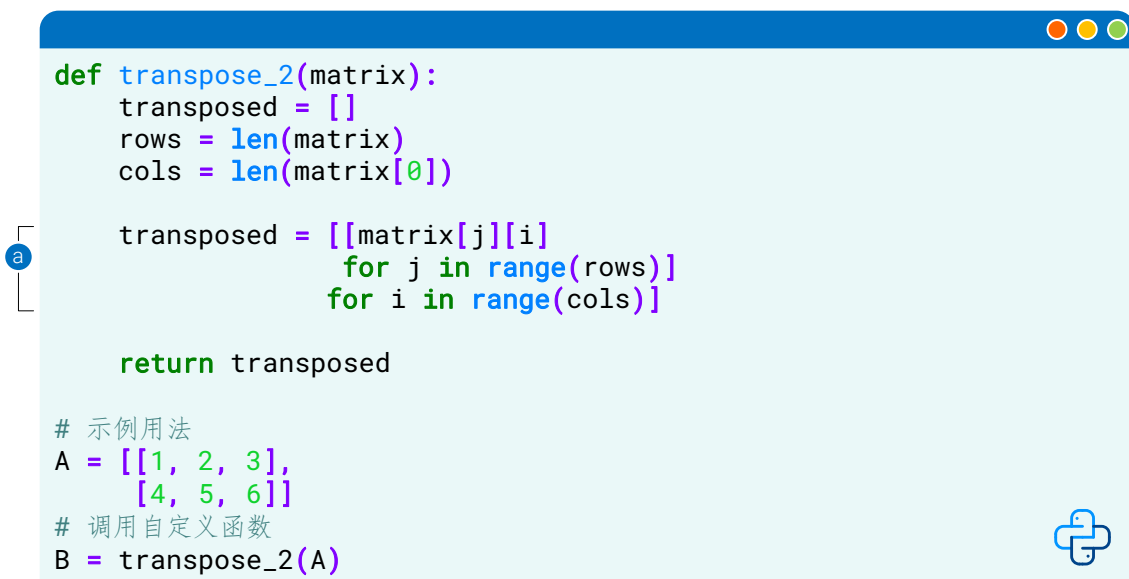


图 32. 利用两层列表生成式完成矩阵转置; Bk1\_Ch07\_21.ipynb

### 计算矩阵逐项积：两层列表生成式

矩阵逐项积是指两个相同矩阵中相应位置上的元素进行逐一相乘，得到一个新的矩阵。

图 33 代码 **a** 首先判断两个二维数组的形状是否相同。

**b** 用两层列表生成式计算矩阵逐项积。

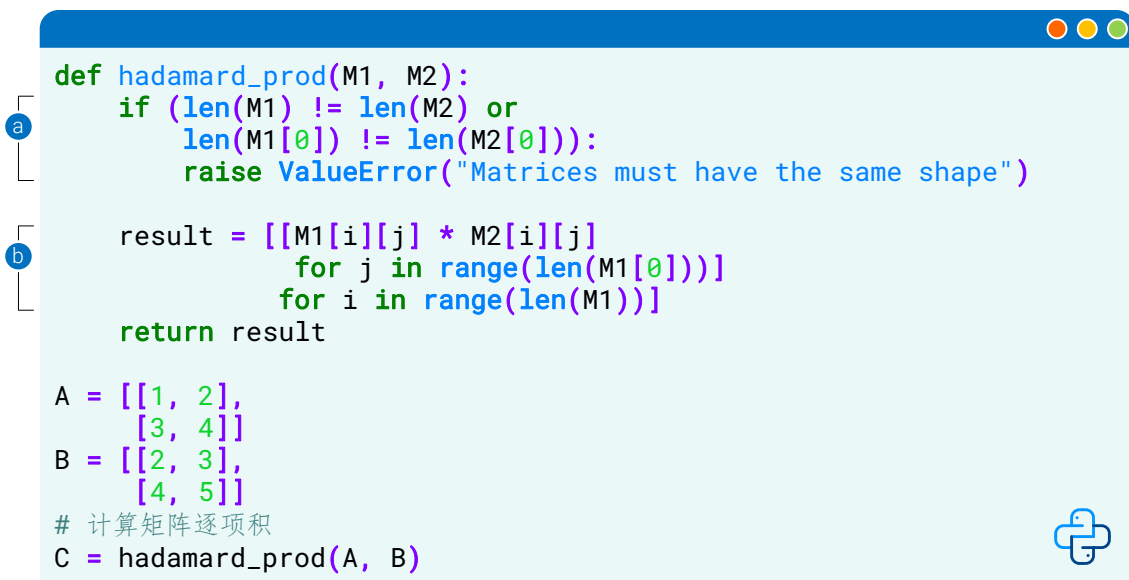


图 33. 利用两层列表生成式计算矩阵逐项积; Bk1\_Ch07\_22.ipynb

### 笛卡儿积

数学上，如果集合  $A$  中有  $a$  个元素，集合  $B$  中有  $b$  个元素，那么  $A$  和  $B$  的笛卡儿积 (Cartesian product) 就有  $a \times b$  个元素。图 34 所示为笛卡儿积原理。

举个简单的例子，假设有两个集合： $A = \{1, 2\}$  和  $B = \{'a', 'b'\}$ 。它们的笛卡尔积为  $\{(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')\}$ 。

图 23 中给出的矩阵乘法原理也可以看成是笛卡儿积的一种应用。

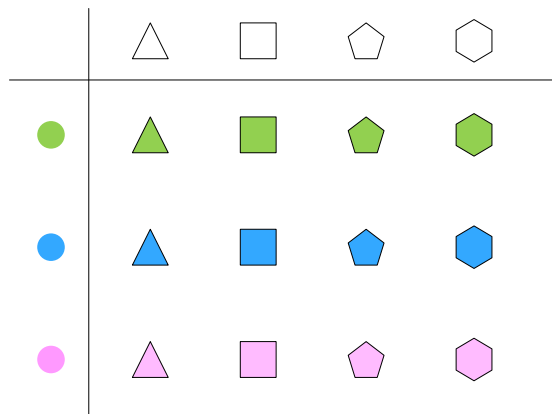


图 34. 笛卡儿积

图 35 代码采用一层列表生成式计算笛卡儿积，结果为列表，列表的每一个元素为元组。

```
a column1 = [1, 2, 3, 4]
b column2 = ['a', 'b', 'c']
c cartesian_product = [(x, y) for x in column1 for y in column2]
print(cartesian_product)
```



图 35. 笛卡儿积，结果为列表，采用单层列表生成式；  Bk1\_Ch07\_23.ipynb

图 36 采用两层列表生成式计算笛卡儿积，结果为二维列表（如图 37 所示）。

```
column1 = [1, 2, 3, 4]
column2 = ['a', 'b', 'c']
a cartesian_product = [(x, y) for x in column1] for y in column2]
for prod_idx in cartesian_product:
    print(prod_idx)
```

图 36. 笛卡儿积，结果为嵌套列表，采用两层列表生成式；  Bk1\_Ch07\_23.ipynb

	1	2	3	4
'a'	(1, 'a')	(2, 'a')	(3, 'a')	(4, 'a')
'b'	(1, 'b')	(2, 'b')	(3, 'b')	(4, 'b')
'c'	(1, 'c')	(2, 'c')	(3, 'c')	(4, 'c')

↓

```
[[ (1, 'a'), (2, 'a'), (3, 'a'), (4, 'a') ],
 [ (1, 'b'), (2, 'b'), (3, 'b'), (4, 'b') ],
 [ (1, 'c'), (2, 'c'), (3, 'c'), (4, 'c') ]]
```

图 37. 嵌套列表

图 38 代码所示为利用 `itertools.product()` 函数完成笛卡儿积计算。这个函数的结果是一个可迭代对象。利用 `list()`，我们将其转化为列表，列表的每一个元素为元组。本章下一节还会介绍 `itertools` 其他用法。

```
a from itertools import product
   column1 = [1, 2, 3, 4]
   column2 = ['a', 'b', 'c']

b cartesian_product = list(product(column1, column2))
  print(cartesian_product)
```

图 38. 笛卡儿积，列表，采用 `itertools.product` 生成，采用列表生成式； Bk1\_Ch07\_23.ipynb

请大家在 JupyterLab 中练习这三段代码，并逐行注释。

## 7.5 迭代器 `itertools`

`itertools` 是 Python 标准库中的一个模块，提供了用于创建和操作迭代器的函数。迭代器是一种用于遍历数据集合的对象，它能够逐个返回数据元素，而无需提前将整个数据集加载到内存中。

`itertools` 模块包含了一系列用于高效处理迭代器的工具函数，这些函数可以帮助我们在处理数据集时节省内存和提高效率。它提供了诸如组合、排列、重复元素等功能，以及其他有关迭代器操作的函数。本节介绍 `itertools` 模块中有关排列组合常用函数。

### 不放回排列

`itertools.permutations` 是 Python 标准库中的一个函数，用于返回指定长度的所有可能排列方式。下面举例如何使用 `itertools.permutations` 函数。



假设有一个字符串 `string = 'abc'`，我们想要获取它的所有字符排列方式，可以按照以下步骤操作。其中，`''.join(perm_idx)` 将当前排列中的元素连接成一个字符串，然后再用 `print()` 打印出来。

```

a import itertools
b string = 'abc'
c perms_all = itertools.permutations(string)
  # 返回一个可迭代对象perms，其中包含了string的所有排列方式

  # 全排列
d for perm_idx in perms_all:
e     print(''.join(perm_idx))

```

图 39. 3 个字符全排列; Bk1\_Ch07\_24.ipynb

这就好比，一个袋子里有三个球，它们分别印有 a、b、c，先后将所有球取出排成一排共有 6 种排列，具体如图 40 所示。

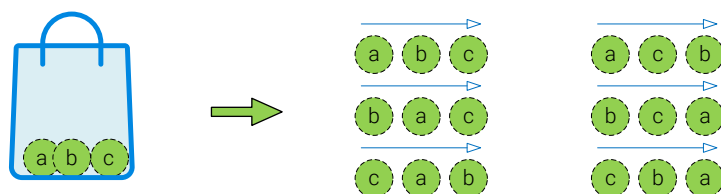


图 40. 3 个元素无放回抽取 3 个，结果有 6 个排列

`itertools.permutations` 函数还有一个可选参数 `r`，用于指定返回的排列长度。如果不指定 `r`，则默认返回与输入序列长度相同的排列。例如，我们可以通过以下方式获取 `string` 的所有长度为 2 的排列。

```

import itertools
string = 'abc'

# 3 个不放回取 2 个的排列
a perms_2 = itertools.permutations(string, 2)
  # 返回一个包含所有长度为 2 的排列的可迭代对象perms

for perm_idx in perms_2:
    print(''.join(perm_idx))

```

图 41. 3 个字符无放回取 2 个排列; Bk1\_Ch07\_25.ipynb

还是以前文小球为例，如图 42 所示，3 个元素无放回抽取 2 个，结果有 6 个排列。大家可能已经发现这个结果和一致。这也不难理解，袋子里一共有 3 个球，无放回拿出两个之后，第三个球是什么字母已经确定，没有任何悬念。

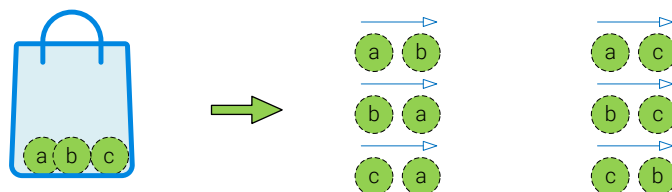


图 42. 3 个元素无放回抽取 2 个，结果有 6 个排列

## 不放回组合

`itertools.combinations` 是 Python 中的一个模块，它提供了一种用于生成组合的函数。

使用 `itertools.combinations` 函数，需要导入 `itertools` 模块，然后调用 `combinations` 函数，传入两个参数：一个可迭代对象和一个整数，表示要选择的元素个数。该函数会返回一个迭代器，通过迭代器你可以获得所有可能的组合。

```
import itertools
string = 'abc'

# 3个取2个的组合
a combs_2 = itertools.combinations(string, 2)
# 返回一个包含所有长度为2的组合的可迭代对象combs_2

for combo_idx in combs_2:
    print(''.join(combo_idx))
```

图 43. 3 个字符无放回取 2 个组合; Bk1\_Ch07\_26.ipynb

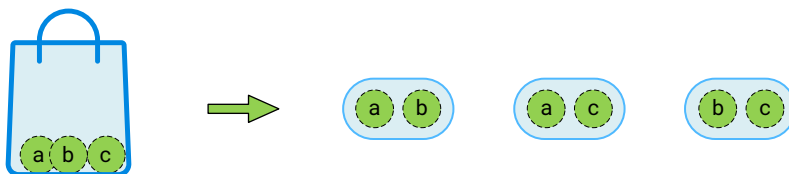


图 44. 3 个元素无放回抽取 2 个，结果有 3 个组合



### 什么是排列？什么是组合？

排列是指从一组元素中按照一定顺序选择若干个元素形成的不同序列，每个元素只能选取一次。

组合是指从一组元素中无序地选择若干个元素形成的不同集合，每个元素只能选取一次。

## 有放回排列

前文介绍的排列、组合都是无放回抽样，下面聊聊有放回抽样。还是以小球为例，如图 45 所示，有放回抽样就是从口袋中摸出一个球之后，记录字母，然后将小球再放回口袋。下一次抽取时，这个球还有被抽到的机会。



### 什么是有放回？什么是无放回？

有放回抽取是指在进行抽样时，每次抽取后将选中的元素放回原始集合中，使得下一次抽取时仍然有可能选中同一个元素。无放回抽取是指在进行抽样时，每次抽取后将选中的元素从原始集合中移除，使得下一次抽取时不会再选中相同的元素。简而言之，有放回抽取可以多次选中相同元素，而无放回抽取每次选中后都会从集合中移除，确保不会重复选中同一元素。

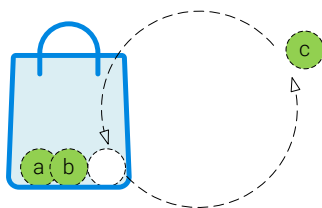


图 45. 有放回抽样

`itertools` 模块中的 `itertools.product` 函数可以用于生成有放回排列。它接受一个可迭代对象和一个重复参数，用于指定每个元素可以重复出现的次数。

```
import itertools

string = 'abc'
# 定义元素列表
a elements = list(string)
# 指定重复次数
repeat = 2

# 生成有放回排列
b permutations = itertools.product(elements, repeat=repeat)

# 遍历并打印所有排列
for permutation_idx in permutations:
    print(''.join(permutation_idx))
```

图 46. 3 个字符有放回取 2 个排列; Bk1\_Ch07\_27.ipynb

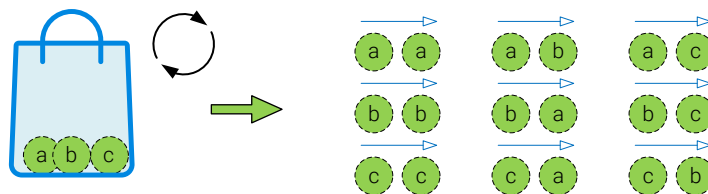


图 47. 3 个元素有放回抽取 2 个，结果有 9 个排列

## 有放回组合

`itertools` 模块中的 `itertools.combinations_with_replacement` 函数可以用于生成有放回组合。该函数接受一个可迭代对象和一个整数参数，用于指定从可迭代对象中选择元素的个数。

```
import itertools

string = 'abc'
# 定义元素列表
elements = list(string)

# 指定组合长度
length = 2

# 生成有放回组合
a combos = itertools.combinations_with_replacement(elements, length)

# 遍历并打印所有组合
for combination_idx in combos:
    print(''.join(combination_idx))
```

图 48. 3 个字符有放回取 2 个的组合; Bk1\_Ch07\_28.ipynb

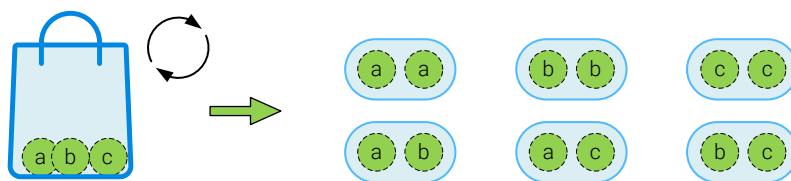


图 49. 3 个元素有放回抽取 2 个，结果有 6 个组合



除了练习本章给出的代码示例之外，请大家完成下面几道题目。

- Q1. 给定一个整数列表 [3, 5, 2, 7, 1]，找到其中的最大值和最小值，并打印两者之和。
- Q2. 使用 `while` 循环输出 1 到 10 的所有奇数。
- Q3. 输入一个数字并将其转换为整数，如果输入的不是数字，则提示用户重新输入直到输入数字为止。
- Q4. 求 100 以内的素数。
- Q5. 请用至少两种不同办法计算 1-100 中奇数之和。
- Q6. 写两个函数分别计算矩阵行、列方向元素之和。
- Q7. 写两个函数分别计算矩阵行、列方向元素平均值。

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。

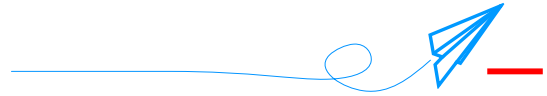
版权归清华大学出版社所有，请勿商用，引用请注明出处。

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)

\* 题目答案在 Bk1\_Ch07\_29.ipynb 中。



本章介绍了常用控制结构的用法，比如条件语句、`while` 循环、`for` 循环、异常处理语句、列表生成式、迭代器 `itertools` 等。特别地，本章还和大家写代码自行完成向量内积、矩阵乘法、等差数列、网格坐标、矩阵转置、逐项积、笛卡儿积相关运算。

当然，本书后续会介绍各种其他函数高效完成上述计算。本章这样安排的意图一方面是让大家理解 Python 控制结构用法；另外一方面，这些例子可以帮助我们理解这些数学工具背后的思想。

请大家格外注意，在实践中我们要尽量“向量化”运算，避免使用循环。