



Python Tutorial

Python HOME

[Python Intro](#)

[Python Get Started](#)

[Python Syntax](#)

[Python Comments](#)

[Python Variables](#)

[Python Data Types](#)

[Python Numbers](#)

[Python Casting](#)

[Python Strings](#)

[Python Booleans](#)

[Python Operators](#)

[Python Lists](#)

[Python Tuples](#)

[Python Sets](#)

[Python Dictionaries](#) •

[Python If...Else](#) •

[Python While Loops](#)

[Python For Loops](#)

[Python Functions](#) •

[Python Lambda](#)

[Python Map](#)

[Python Filter](#)

[Python Reduce](#)

[Python Scope](#)

[Python Modules](#)

[Python Dates](#)

[Python Time](#)

[Python Arrays](#)

[Python Classes, Objects](#)

[Python Inheritance](#)

[Python Iterators](#)

[Python Generators](#)

[Python Properties](#)

[Python Decorators](#)

[Python Lambda](#)

[Python Map](#)

[Python Filter](#)

[Python Reduce](#)

[Python Scope](#)

[Python Modules](#)

[Python Dates](#)

[Python Time](#)

[Python Arrays](#)

[Python Classes, Objects](#)

[Python Inheritance](#)

[Python Iterators](#)

[Python Generators](#)

[Python Properties](#)

[Python Lambda](#)

[Python Map](#)

[Python Filter](#)

[Python Reduce](#)

[Python Scope](#)

[Python Modules](#)

[Python Dates](#)

[Python Time](#)

[Python Arrays](#)

[Python Classes, Objects](#)

[Python Inheritance](#)

[Python Iterators](#)

[Python Generators](#)

[Python Properties](#)

[Python Lambda](#)

[Python Map](#)

[Python Filter](#)

[Python Reduce](#)

[Python Scope](#)

[Python Modules](#)

[Python Dates](#)

[Python Time](#)

[Python Arrays](#)

[Python Classes, Objects](#)

[Python Inheritance](#)

[Python Iterators](#)

[Python Generators](#)

[Python Properties](#)

[Python Lambda](#)

[Python Map](#)

[Python Filter](#)

[Python Reduce](#)

[Python Scope](#)

[Python Modules](#)

[Python Dates](#)

[Python Time](#)

[Python Arrays](#)

[Python Classes, Objects](#)

[Python Inheritance](#)

[Python Iterators](#)

[Python Generators](#)

[Python Properties](#)

[Python Lambda](#)

[Python Map](#)

[Python Filter](#)

[Python Reduce](#)

[Python Scope](#)

[Python Modules](#)

[Python Dates](#)

[Python Time](#)

[Python Arrays](#)

[Python Classes, Objects](#)

[Python Inheritance](#)

[Python Iterators](#)

[Python Generators](#)

[Python Properties](#)

[Python Lambda](#)

[Python Map](#)

[Python Filter](#)

[Python Reduce](#)

[Python Scope](#)

[Python Modules](#)

[Python Dates](#)

[Python Time](#)

[Python Arrays](#)

[Python Classes, Objects](#)

[Python Inheritance](#)

[Python Iterators](#)

[Python Generators](#)

[Python Properties](#)

[Python Lambda](#)

[Python Map](#)

[Python Filter](#)

[Python Reduce](#)

[Python Scope](#)

[Python Modules](#)

[Python Dates](#)

[Python Time](#)

[Python Arrays](#)

[Python Classes, Objects](#)

[Python Inheritance](#)

[Python Iterators](#)

[Python Generators](#)

[Python Properties](#)

[Python Lambda](#)

[Python Map](#)

[Python Filter](#)

[Python Reduce](#)

[Python Scope](#)

[Python Modules](#)

[Python Dates](#)

[Python Time](#)

[Python Arrays](#)

[Python Classes, Objects](#)

[Python Inheritance](#)

[Python Iterators](#)

[Python Generators](#)

[Python Properties](#)

[Python Lambda](#)

[Python Map](#)

[Python Filter](#)

[Python Reduce](#)

[Python Scope](#)

[Python Modules](#)

[Python Dates](#)

[Python Time](#)

[Python Arrays](#)

[Python Classes, Objects](#)

[Python Inheritance](#)

[Python Iterators](#)

[Python Generators](#)

[Python Properties](#)

[Python Lambda](#)

[Python Map](#)

[Python Filter](#)

[Python Reduce](#)

[Python Scope](#)

[Python Modules](#)

[Python Dates](#)

[Python Time](#)

[Python Arrays](#)

[Python Classes, Objects](#)

[Python Inheritance](#)

[Python Iterators](#)

[Python Generators](#)

[Python Properties](#)

[Python Lambda](#)

[Python Map](#)

[Python Filter](#)

[Python Reduce](#)

[Python Scope](#)

[Python Modules](#)

[Python Dates](#)

[Python Time](#)

[Python Arrays](#)

[Python Classes, Objects](#)

[Python Inheritance](#)

[Python Iterators](#)

[Python Generators](#)

[Python Properties](#)

[Python Lambda](#)

[Python Map](#)

[Python Filter](#)

[Python Reduce](#)

[Python Scope](#)

[Python Modules](#)

[Python Dates](#)

[Python Time](#)

[Python Arrays](#)

[Python Classes, Objects](#)

Variables and Simple Data Types



Commenting Your Code

Variables

Strings

Numbers

Input and Output

Type Casting

python school

1.comment code

单行 (#后面输入文字内容)

多行 (连续单行评论)

用“”

简洁明了 保持更新 避免过度使用

2.variables

只需要为变量命名 并且用等号赋值

命名变量的规则

- 字母和下划线：它们可以以字母（A-Z, a-z）或下划线（_）开头。
- 数字：在第一个字符之后，您可以使用数字（0-9），但变量名不能以数字开头。
- 区分大小写：变量名称区分大小写（`age`、`Age` 和 `AGE` 是三个不同的变量）。
- 避免关键字：不要使用Python的保留词或关键字，如 `if`、`else`、`for` 等。

可能会遇到命名错误

为了避免此错误，您必须：

- 双重检查拼写：始终仔细检查变量名是否有拼写错误，特别是如果您遇到 `NameError`。
- 一致的命名约定：使用一致的命名约定，以便更容易记住您是如何命名变量的。
- 代码编辑器：利用具有语法高亮显示和自动完成功能的代码编辑器或集成开发环境（IDE）。这些工具可以帮助您在键入时捕获拼写错误的变量名。

不同命名规则

Multi Words Variable Names

Variable names with more than one word can be difficult to read.

There are several techniques you can use to make them more readable:

Camel Case

Each word, except the first, starts with a capital letter:

```
myVariableName = "John"
```

Pascal Case

Each word starts with a capital letter:

```
MyVariableName = "John"
```

Snake Case

Each word is separated by an underscore character:

```
my_variable_name = "John"
```

In the `print()` function, you output multiple variables, separated by a comma:

Example

```
x = "Python"  
y = "is"  
z = "awesome"  
print(x, y, z)
```

[Try it Yourself »](#)

You can also use the `*` operator to output multiple variables:

Example

```
x = "Python "  
y = "is "  
z = "awesome"  
print(x + y + z)
```

[Try it Yourself »](#)

Notice the space character after "Python " and "is ", without them the result would be "Pythonisawesome".

多重赋值

```
1 | x, y, z = 1, 2, 3
```

在本例中，`x`被分配值1，`y`被分配值2，`z`被分配值3。

多重赋值可用于轻松交换两个变量的值：

```
1 | x, y = y, x
```

这交换了`x`和`y`的值，而不需要临时变量。

常数 Constants

在编程中，常量是一种值不变的变量。Python没有内置常量类型，但它遵循命名约定来表示变量应被视为常量。按照惯例，常量通常在Python文件的顶部定义，并使用所有大写字母命名，下划线分隔单词。

示例：

```
1 | PI = 3.14
2 | MAX_SIZE = 100
```

尽管这些在技术上仍然是可变的（可变的），但命名约定向其他程序员发出信号，这些值旨在保持不变，不应被修改。虽然与其他一些编程语言不同，Python并不强制常量的不变性。程序员应该尊重这些惯例，而不是改变常量变量的值。

最佳实践

- **描述性名称：**使用描述变量是什么或做什么的名称，如`student_name`或`total_price`。
- **一致命名约定：**坚持命名风格，如变量名的`lower_case_with_underscores`。

3.strings

- 串联：将字符串组合在一起（“`"Hello, " + "world!"` 结果为 `"Hello, world!"`）。
- 重复：重复字符串（“`"Ha" * 3` 结果为 `"HaHaHa"`）。
- 索引：访问单个字符（“`"Hello"[1]` 给出 `'e'`）。
- 切片：获取子字符串（“`"Hello"[1:4]` 给出 `'ell'`）。

- `upper()`, `lower()` 更改大小写。
- `strip()`: 删除空格。
- `find()`, `replace()` 搜索和替换子字符串。
- `split()`, `join()` 拆分字符串并加入字符串列表。

格式化的字符串文字 (F字符串)

- 在Python 3.6中引入的F字符串是一种更易读、更简洁、更高效的字符串格式化方式。
- 用 `f` 或 `F` 在字符串前缀，并使用包含将替换为其值的表达式的花括号 `{}`。
- 大括号内的表达式在运行时计算，这允许嵌入变量、表达式甚至函数调用。
- 示例：

```
1 | name = "Alice"
2 | age = 25
3 | message = f"Hello, {name}. You are {age} years old."
4 | print(message) # Output: Hello, Alice. You are 25 years old.
```

处理语法 (syntax) 错误

注意打完整的引号 统一单引号和双引号

(如果字符串里包括单引号 请打双引号)

转义字符

/



SQL

PYTHON

JAVA

PHP

HOW TO

W3.CSS

C

C++

C#

BO

英语

中文 (简体)



要在字符串中插入非法字符，请使用转义字符。转义字符是反斜杠 \ 后跟要插入的字符。非法字符的一个示例是双引号包围的字符串内的双引号

[翻译整页内容](#)[Google Translate](#)

To insert characters that are illegal in a string, use an escape character.

An escape character is a backslash \ followed by the character you want to insert.

An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

```
txt = "We are the so-called "Vikings" from the north."
```

[Try it Yourself »](#)[Get your own Python Server](#)

Example

You will get an error if you use double quotes inside a string that is surrounded by double quotes:

```
txt = "We are the so-called "Vikings" from the north."
```

[Try it Yourself »](#)

To fix this problem, use the escape character \" :

Example

The escape character allows you to use double quotes when you normally would not be allowed:

```
txt = "We are the so-called \"Vikings\" from the north."
```

[Try it Yourself »](#)

Replace the character H with a J .

```
txt = "Hello World"  
txt = txt.replace("H", "J")
```

String Concatenation

To concatenate, or combine, two strings you can use the `+` operator.

Example

Merge variable `a` with variable `b` into variable `c`:

```
a = "Hello"  
b = "World"  
c = a + b  
print(c)
```

[Try it Yourself »](#)

Example

To add a space between them, add a `" "`:

```
a = "Hello"  
b = "World"  
c = a + " " + b  
print(c)
```

[Try it Yourself »](#)

But we can combine strings and numbers by using the `format()` method!

The `format()` method takes the passed arguments, formats them, and places them in the string where the placeholders `{}` are:

Example

Use the `format()` method to insert numbers into strings:

```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age))
```

[Try it Yourself »](#)

The `format()` method takes unlimited number of arguments, and are placed into the respective placeholders:

Example

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

4. number

Integers (int)

整数

- Description: Integers are whole numbers without a decimal point.
- Example: `42`, `-3`, `0`

```
1 | positive_integer = 10
2 | negative_integer = -5
```

Floating-Point Numbers (float)

小数

- Description: Floating-point numbers, or floats, represent real numbers.
- Example: `3.14`, `-0.001`, `2e10`

- 加法 (+) : 添加两个数字。
- 减法 (-) 从另一个数字中减去一个数字。
- 乘法 (*) 乘以两个数字。
- 除法 (/) 将一个数字除以另一个数字。总是导致浮点数。
- 楼层除法 (//) 除法并返回不大于结果的最大整数。
- 模量 (%) : 返回除法的余数。
- 指数 (**): 将一个数字提高到另一个数字的功率。

```
1 | 5 + 2    # 7
2 | 5 - 2    # 3
3 | 5 * 2    # 10
4 | 5 / 2    # 2.5
5 | 5 // 2   # 2
6 | 5 % 2    # 1
7 | 5 ** 2   # 25
```

隐式类型转换

Python在适当的情况下自动将数字从一种类型转换为另一种类型。例如，当您在整数和浮点数之间执行操作时，Python会自动将整数转换为浮点数，以保持精度。这种操作的结果总是浮点数。

```
1 | result = 4 + 2.0 # Integer 4 is converted to float, result is 6.0 (float)
```

在Python中，除法运算符（`/`）总是产生一个浮点数，即使除法可以表示为整数。

```
1 | result = 8 / 4 # Result is 2.0, not 2
```

拷贝

如果您需要除法的结果为整数，您可以使用楼层除法（`//`）。此操作将数字除以四舍五入到最接近的整数。虽然地板除法对获得整数很有用，但重要的是要注意，在这个过程中你可能会失去精确度。这是因为底除会截断（切断）十进制部分，无论结果是否接近下一个更高或更低的整数。

```
1 | result = 7 // 2 # Result is 3, not 3.5
```

为了提高可读性，特别是对于长数字，Python允许使用下划线来分隔数字。

```
1 | billion = 1_000_000_000
```

数字函数

Python为处理数字提供了丰富的函数集，增强了执行各种数字操作的能力。以下是对这些功能的一些更详细的了解：

- `round()`：将浮点数四舍五入到最近的整数，或指定小数位。
- `int()`：将一个值转换为整数。这截断了浮点数中的任何小数部分。
- `float()`：将一个值转换为浮点数。
- `abs()`：返回一个数字的绝对值，即数字与零的距离。

```
1 | rounded_number = round(3.14159, 2) # Rounds to 3.14
```

```
1 | integer_number = int(5.7) # Converts to 5
```

```
1 | float_number = float(7) # Converts to 7.0
```

```
1 | absolute_value = abs(-5.5) # Returns 5.5
```

对于更高级的数学能力，Python提供了`math`、`statistics`和`random`等专业模块，这些模块提供了核心语言中没有的广泛数学函数、统计计算和随机数生成。Python中的模块是包含一组函数、类和变量的文件，它们可以导入到其他Python脚本中以扩展功能。

```
1 # Calculate the square root of a number.  
2 import math  
3 square_root = math.sqrt(16) # Returns 4.0
```

拷贝

```
1 # Generate a random integer within a specified range.  
2 import random  
3 random_integer = random.randint(1, 10) # Random integer between 1 and 10
```

Example

Get the characters from position 2 to position 5 (not included):

```
b = "Hello, World!"  
print(b[2:5])
```

[Try it Yourself »](#)

Note: The first character has index 0.

Slice From the Start

By leaving out the start index, the range will start at the first character:

Example

Get the characters from the start to position 5 (not included):

```
b = "Hello, World!"  
print(b[:5])
```

[Try it Yourself »](#)

5.input

print()函数

```
1 | print("Hello, World!")  
2 | # Output: Hello, World!
```

打印变量

```
1 | x = 10  
2 | print("The value of x is", x)  
3 | # Output: The value of x is 10
```

使用f-String进行字符串格式化

```
1 | name = "Alice"  
2 | age = 25  
3 | print(f"{name} is {age} years old")  
4 | # Output: Alice is 25 years old
```

使用`end`和`sep`参数

```
1 | print("Hello", "World", sep="-", end="!")  
2 | print(" This is on the same line.")  
3 | # Output: Hello-World! This is on the same line.
```

以最简单的形式，`input()` 可用于提示用户并将输入存储在变量中。

```
1 | user_name = input("Enter your name: ")  
2 | print("Hello,", user_name)
```

数字输入处理

由于 `input()` 返回一个字符串，要将输入用作数字，您需要使用类型转换来转换它。

```
1 | age = input("Enter your age: ")  
2 | age = int(age) # Converting string to integer  
3 | print("You are", age, "years old.")
```

多个输入

您可以通过多次调用 `input()` 来请求多个输入。

```
1 | first_name = input("Enter your first name: ")  
2 | last_name = input("Enter your last name: ")  
3 | print("Your full name is", first_name, last_name)
```

6.type casting

Python为不同数据类型之间的显式转换或类型转换提供了几个函数。以下是Python为我们已经介绍过的数据类型提供的类型转换函数：

- `str()`：将值转换为字符串。此功能对于创建数据的文本表示至关重要，特别是当您想将非字符串类型与字符串连接时。
- `int()`：将一个值转换为整数。当您需要执行需要整数的数学运算或处理需要整数输入的函数时，会使用它。
- `float()`：将一个值转换为浮点数。当您需要执行涉及十进制数的操作时，或者当函数期望浮点数时，请使用此功能。

转换为字符串

```
1 | number = 42
2 | converted_to_string = str(number)
3 | print(f"Number {converted_to_string} is now a string.")
4 | print(type(converted_to_string))
5 | # Output: Number 42 is now a string.
6 | # <class 'str'>
```

转换为整数

```
1 | float_number = 3.6
2 | converted_to_int = int(float_number)
3 | print(f"Float {float_number} is converted to integer {converted_to_int}.")
4 | # Output: Float 3.6 is converted to integer 3.
5 | # <class 'int'>
```

转换为浮动：

```
1 | string_number = "10.5"
2 | converted_to_float = float(string_number)
3 | print(f"String '{string_number}' is converted to float {converted_to_float}.")
4 | print(type(converted_to_float))
5 | # Output: String '10.5' is converted to float 10.5.
6 | # <class 'float'>
```

Conditionals and Control Flow



Booleans

Conditional Tests

If Statements

While Loops

For Loops

1. Booleans

Truthy Values

- Definition: A truthy value is one that, when evaluated in a Boolean context, is considered `True`.
- Common Examples: Non-zero numbers, non-empty strings, non-empty lists, and generally most objects are truthy.

Example:

```
1 | print(bool(0)) # Output: True
2 | print(bool('hello')) # Output: True
3 | print(bool([1, 2, 3])) # Output: True
```

In these examples, the conditions are truthy, and will return `True` when converted to Booleans.

Falsy Values

- Definition: A falsy value is one that, when evaluated in a Boolean context, is considered `False`.
- Common Examples: `0`, `None`, empty strings `''`, empty lists `[]`, empty tuples `()`, and empty dictionaries `{}` are falsy.

Example:

```
1 | print(bool(0)) # Output: False
2 | print(bool('')) # Output: False
3 | print(bool([])) # Output: False
```

Here, the conditions are falsy, so will return `False` when converted to Booleans.

2. Conditional tests

1) comparison operator

1. 比较运算符

标准比较运算符可用于比较值：

- 等于 (`==`)
- 不等于 (`!=`)
- 大于 (`>`)
- 小于 (`<`)
- 大于或等于 (`>=`)
- 小于或等于 (`<=`)

示例：

```
1 | x = 10
2 | print(x == 10) # True
3 | print(x != 5) # True
4 | print(x > 5) # True
5 | print(x < 15) # True
```

具体运用

- 平等和不平等：`==` 和 `!=` 运算符用于检查值是否相等，无论其数据类型如何。
- 适用性：这些运算符不限于数字；它们可以与字符串、列表、元组、字典和其他数据类型一起使用。

示例：

```
1 | print("hello" == "hello") # True
2 | print("hello" == "Hello") # False (case-sensitive)
```

字符串之间的比较区分大小写，并检查逐个字符的相等性。

```
1 | print([1, 2, 3] == [1, 2, 3]) # True
2 | print([1, 2, 3] == [3, 2, 1]) # False (order matters)
```

列表根据顺序和内容进行比较。每个元素的顺序和值都很重要。

2) Logical Operator

逻辑运算符（`and`, `or` `not`）用于组合条件测试：

- `and`：如果两个条件都成立，则为真
- `or`：如果任一条件为真，则为真
- `not`：反转测试结果

示例：

```
1 | x, y = 10, 20
2 | print(x < 15 and y > 15) # True
3 | print(x > 15 or y > 15) # True
4 | print(not x == 10) # False
```

这些测试检查序列中是否存在值（如列表、元组或字符串）：

3.) Membership Tests

- `in`：如果在序列中找到该值，则为真
- `not in`：如果在序列中找不到该值，则为真

示例：

```
1 | fruits = ["apple", "banana", "cherry"]
2 | print("banana" in fruits)      # True
3 | print("mango" not in fruits)  # True
```

4.) Identity Tests

4. 身份测试

身份运算符（`is`, `is not`）检查两个变量是否指向同一对象：

- `is`：如果两个变量都引用同一个对象，则为真
- `is not`：如果他们指的是不同的对象，那就正确了

示例：

```
1 | a = [1, 2, 3]
2 | b = a
3 | c = [1, 2, 3]
4 | print(a is b)      # True
5 | print(a is c)      # False
6 | print(a is not c) # True
```

3. If statement

添加Else和Elif

要处理不同的场景，您可以使用 `elif`（“else if”的缩写）扩展if语句，然后：

- `elif`: 如果之前的条件是 `False` 则用于检查其他条件。
- `else`: 捕获任何未由上述if和elif条件处理的情况。

示例：

```
1 | number = 10
2 |
3 | if number > 10:
4 |     print("Number is greater than 10.")
5 | elif number > 5:
6 |     print("Number is greater than 5 but not greater than 10.")
7 | else:
8 |     print("Number is 5 or less.")
```

在这个扩展示例中，程序检查多个条件：

- 如果 `number` 大于10，它将执行第一个代码块。
- 如果没有，但 `number` 大于5，它将执行第二个块。
- If none of these conditions are met, it executes the code in the `else` block.

If-Else语句的基本结构

If-else

`else` 语句遵循 `if` 语句，并在相同的缩进级别对齐。它的基本语法是：

```
1 | if condition:  
2 |     # code to execute if the condition is True  
3 | else:  
4 |     # code to execute if the condition is False
```

- `if` 块检查条件，如果条件为 `True`，则执行其代码。
- 如果条件为 `False`，则执行 `else` 块。

示例：

```
1 | age = 18  
2 |  
3 | if age >= 18:  
4 |     print("Adult Ticket.")  
5 | else:  
6 |     print("Childrens Ticket.")
```

在本例中，如果 `age` 为18岁或以上，将打印“成人票”。否则，将打印“儿童票”。

If-Elif-Else语句的基本结构

If-elif-else

`elif` 语句适合在 `if` 和 `else` 语句之间，允许更复杂和更细致的决策过程。它在条件块中的基本语法是：

```
1 | if condition1:  
2 |     # code to execute if condition1 is True  
3 | elif condition2:  
4 |     # code to execute if condition1 is False but condition2 is True  
5 | else:  
6 |     # code to execute if both condition1 and condition2 are False
```

- `if` 块检查条件，如果条件为 `True`，则执行其代码。
- 如果 `if` 块中的条件为 `False`，则检查 `elif` 条件。如果它是 `True`，则执行其代码块。
- 您可以有多个 `elif` 语句来按顺序检查各种条件。
- 最后，如果 `if` 或 `elif` 条件都不是 `True`，则执行 `else` 块。

```
1 | score = 85  
2 |  
3 | if score >= 70:  
4 |     grade = 'Distinction'  
5 | elif score >= 60:  
6 |     grade = 'Merit'  
7 | elif score >= 50:  
8 |     grade = 'Pass'  
9 | else:  
10|     grade = 'Fail'  
11| print(f"Your grade is {grade}.")
```

在本例中，程序根据多个条件检查 `score`，以确定 `grade`。`elif` 语句按顺序计算，一旦一个条件为 `True`，其相应块就会被执行，其余块被跳过。

使用场景

`elif` 在需要单独评估多个不同条件的场景中特别有用，例如：

- 分级系统
- 将数据分类到不同的类别中
- 多步骤决策过程

重要注意事项

- 执行流程：根据条件，在 `if`、`elif` 和 `else` 块中只执行一个块。
- 顺序检查：`if`、`elif` 语句中的条件按其出现的顺序进行检查。一旦发现条件为 `True`，其余的 `elif` 块将不进行评估。
- 单个其他：只有一个 `else` 块可以与 `if` 语句一起使用。
- 可选用法：`else` 块是可选的。您可以有一个没有另一个的 `if` 一个 `if-elif` 语句，但是，如果这些条件都不是 `True`，则不会执行任何代码块。
- 多个Elif块：您可以在条件块中拥有尽可能多的 `elif` 语句。

基本而循环

4.while loops

示例：

```
1 | count = 0
2 | while count < 5:
3 |     print("Count is:", count)
4 |     count += 1
```

这个循环从0打印数字到4。`count += 1` 语句至关重要——它更新 `count` 变量，确保循环最终结束。至关重要的是，`while` 后，Loop的情况最终会变成 `False`。因为如果条件永远不会变成 `False`，循环将永远持续下去。这被称为无限循环。

无限循环

当循环的条件永远不会变成 `False`，就会出现无限循环。这意味着循环将无限期地运行，这通常不是你想要的。

示例：

```
1 | while True:
2 |     print("This will never end!")
```

如果您的程序卡在无限循环中，您可以通过在Windows上按 `Ctrl + C` 或在Mac上按 `Command + C` 来中断执行。

退出循环

使用休息

无论循环的条件如何，`break` 语句都用于立即退出循环。当您需要根据环体内的特定条件停止循环时，这很有用。

示例：

```
1 | count = 0
2 | while True: # This would be an infinite loop without the break
3 |     print(count)
4 |     count += 1
5 |     if count >= 5:
6 |         break # Exits the loop when count is 5
7 | print("Loop ended.")
```

修改循环条件

退出循环的另一种方法是确保循环的条件最终变成 `False`。这种方法更直接，通常在已知或可以确定迭代次数时使用。

示例：

```
1 | count = 0
2 | while count < 5:
3 |     print(count)
4 |     count += 1 # Incrementing count so that the loop condition becomes False
5 | print("Loop ended.")
```

跳出循环

继续关键词

虽然不用于退出循环，但 `continue` 语句值得一提。它跳过当前迭代的循环内的其余代码，并移动到下一个迭代。

示例：

```
1 | count = 0
2 | while count < 5:
3 |     count += 1
4 |     if count == 3:
5 |         continue
6 |     print(count)
```

当循环条件变为 `False` 时，While Loop中的 `else` 块执行，但当循环被 `break` 语句终止时不会执行。

其他

示例：

```
1 | count = 0
2 | while count < 3:
3 |     print(count)
4 |     count += 1
5 | else:
6 |     print("Loop ended, count is no longer less than 3.")
```

最佳实践

- 始终确保你的循环有结束的方法。测试可能没有的边缘情况。
- 谨慎对待 `while True:`。始终有一个明确的退出策略。
- 易于阅读和理解的写入循环。避免过于复杂的条件。
- 编写复杂的循环时，在构建时测试它们，以避免惊喜。

Python中 `for` 循环的基本语法是：

5. for loop

```
1 | for element in sequence:  
2 |     # do something with element
```

Here, `element` is a variable that takes the value of each item in the `sequence` as the loop iterates over it. The `# do something with element` part represents the block of code that you want to execute for each item.

```
1 | fruits = ["apple", "banana", "cherry"]  
2 | for fruit in fruits:  
3 |     print(fruit)  
4 | # Output: apple  
5 | # banana  
6 | # cherry
```

为什么要使用for Loop?

1. 简单性：它提供了一种清晰简洁的序列进行遍演的方式。
2. Readability: Code written with `for` loops is generally more readable and easier to understand.
3. 灵活性：它可以与各种数据结构一起使用，并支持循环内的复杂操作。
4. 安全：减少无限循环等错误的可能性，这在 `while` 循环中更常见。

缩进

在Python中，缩进用于创建代码块。对于 `for` 循环，您希望作为循环每个迭代的一部分执行的所有代码都应在 `for` 语句下缩进。

以下是正确缩进的示例：

```
1 | fruits = ["apple", "banana", "cherry"]  
2 |  
3 | for fruit in fruits:  
4 |     print(fruit) # This line is part of the for loop  
5 |     print("Fruit is delicious!") # This line is also part of the for loop
```

在本例中，两个 `print` 语句的缩进量相同，并且是循环的一部分。它们将针对 `fruits` 中的每个项目执行。

for循环后的代码（未缩进的代码）

在 `for` 循环之后，任何未缩进的代码都不是循环的一部分，在循环完成所有迭代后，只会执行一次。

```
1 | for fruit in fruits:  
2 |     print(fruit) # This line is part of the for loop  
3 |  
4 | print("All fruits have been printed.") # This line is NOT part of the for loop
```

在这个例子中 `"All fruits have been printed."` 在循环完成 `fruits` 中所有元素的循环遍历完成后，只打印一次。

有关缩进的错误

1. 忘记在循环内缩进第一行

```
1 | fruits = ["apple", "banana", "cherry"]
2 | for fruit in fruits:
3 |     print(fruit) # Incorrect: This line should be indented
```

为什么它不正确:

- Python期望在 `for` 循环声明后出现缩进块。
- 这将导致 `IndentationError`。

2. 忘记在循环内缩进后续行

```
1 | fruits = ["apple", "banana", "cherry"]
2 | for fruit in fruits:
3 |     print(fruit)
4 |     print(f"{fruit} is delicious!") # Incorrect: This should be part of the loop
```

为什么它不正确:

- 第二个 `print` 语句旨在成为循环的一部分，但没有缩进，因此它只在循环完成后执行一次。
- 这是一个逻辑错误，而不是语法错误，但它可能会导致意想不到的结果。

3. 不应成为循环一部分的缩进线

```
1 | fruits = ["apple", "banana", "cherry"]
2 | for fruit in fruits:
3 |     print(fruit)
4 |     print("All fruits have been printed.") # Incorrect: This should not be part of the loop
```

为什么它不正确:

- 第二个 `print` 语句是缩进的，因此被认为是循环的一部分。
- 它将在每次迭代中执行，这可能不是预期的行为。这是一个逻辑错误。

4. 忘记声明末尾的冒号

```
1 | fruits = ['apple', 'banana', 'cherry']
2 | for fruit in fruits
3 |     print(fruit) # Incorrect: Missing colon at the end of the 'for' statement
```

为什么它不正确:

- `for` 语句末尾的冒号 (`:`) 在Python中是强制性的。
- 省略冒号会导致 `SyntaxError`。

Basic Data Structures



Lists

Working with Lists

Tuples

Sets

1.list

(有序的 可更改的 可重复的)

```
mylist = ["apple", "banana", "cherry"]
```

列表

列表用于在单个变量中存储多个项目。

列表是 Python 中用于存储数据集合的 4 种内置数据类型之一，其他 3 种是[Tuple](#)、[Set](#)和[Dictionary](#)，它们都具有不同的质量和用途。

列表是使用方括号创建的：

例子

[获取您自己的 Python 服务器](#)

创建一个列表：

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

The list() Constructor

It is also possible to use the `list()` constructor when creating a new list.

Example

Using the `list()` constructor to make a List:

```
thislist = list(("apple", "banana", "cherry")) # note the double round-brackets
print(thislist)
```

```
['apple', 'banana', 'cherry']
```

Python Collections (Arrays)

There are four collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered** and changeable. No duplicate members.

*Set items are unchangeable, but you can remove and/or add items whenever you like.

**As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

Python 集合 (数组)

Python 编程语言中有四种集合数据类型：

- **列表**是一个有序且可变的集合。允许重复的成员。
- **元组**是一个有序且不可更改的集合。允许重复的成员。
- **Set**是一个无序、不可更改*且无索引的集合。没有重复的成员。
- **字典**是一个有序的、可变的集合。没有重复的成员。

Print the second item of the list:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])
```

[Try it Yourself »](#)

access list items

banana

Note: The first item has index 0.

Negative Indexing

Negative indexing means start from the end

-1 refers to the last item, -2 refers to the second last item etc.

cherry

Example

Print the last item of the list:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])
```

Range of Indexes

range

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new list with the specified items.

Example

Return the third, fourth, and fifth item:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:5])
```

Try it Yourself »

```
['cherry', 'orange', 'kiwi']
```

1 2 3 4 .

Note: The search will start at index 2 (included) and end at index 5 (not included).

Remember that the first item has index 0.

By leaving out the start value, the range will start at the first item:

Example

This example returns the items from the beginning to, but NOT including, "kiwi":

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[:4])
```

Try it Yourself »

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[:4])
['apple', 'banana', 'cherry', 'orange']
```

#This will return the items from index 0 to index 4.

#Remember that index 0 is the first item, and index 4 is the fifth item

#Remember that the item in index 4 is NOT included

By leaving out the end value, the range will go on to the end of the list:

Example

This example returns the items from "cherry" to the end:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:])
```

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:])
```

```
['cherry', 'orange', 'kiwi', 'melon', 'mango']
```

#This will return the items from index 2 to the end.

#Remember that index 0 is the first item, and index 2 is the third

Range of Negative Indexes

negative

Specify negative indexes if you want to start the search from the end of the list:

Example

This example returns the items from "orange" (-4) to, but NOT including "mango" (-1):

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[-4:-1])
thislist = ['apple', 'banana', 'cherry', 'orange', 'kiwi', 'melon', 'mango']
print(thislist[-4:-1])
Try it #Negative indexing means starting from the end of the list.
#This example returns the items from index -4 (included) to index -1
(excluded)
#Remember that the last item has the index -1,
```

['orange', 'kiwi', 'melon']

Check If Item Exists

check

To determine if a specified item is present in a list use the `in` keyword:

Example

Check if "apple" is present in the list:

```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
    print("Yes, 'apple' is in the fruits list")
```

```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
    print("Yes, 'apple' is in the fruits list")
```

Yes, 'apple' is in the fruits list

change

Example

Get your own Python Server

Change the second item:

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
```

[Try it Yourself »](#)

Change a Range of Item Values

To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values:

Example

Change the values "banana" and "cherry" with the values "blackcurrant" and "watermelon":

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)
```

[Try it Yourself »](#)

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"

print(thislist)
```

['apple', 'blackcurrant', 'cherry']

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]

print(thislist)
```

['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(2, "watermelon")

print(thislist)
```

['apple', 'banana', 'watermelon', 'cherry']

append

Append Items

To add an item to the end of the list, use the `append()` method:

Example

Using the `append()` method to append an item:

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

[Try it Yourself »](#)

```
thislist = ["apple", "banana", "cherry"]

thislist.append("orange")

print(thislist)
```

```
['apple', 'banana', 'cherry', 'orange']
```

extend

Extend List

To append elements from *another list* to the current list, use the `extend()` method.

Example

Add the elements of `tropical` to `thislist`:

```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
thislist.extend(tropical)
print(thislist)
```

[Try it Yourself »](#)

```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]

thislist.extend(tropical)

print(thislist)
```

```
['apple', 'banana', 'cherry', 'mango', 'pineapple', 'papaya']
```

insert

Insert Items

To insert a list item at a specified index, use the `insert()` method.

The `insert()` method inserts an item at the specified index:

Example

Insert an item as the second position:

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```

[Try it Yourself »](#)

remove

(Item

```
thislist = ["apple", "banana", "cherry"]  
thislist.remove("banana")  
print(thislist)
```

```
['apple', 'cherry']
```

(index

Remove Specified Index

The `pop()` method removes the specified index.

Example

Remove the second item:

```
thislist = ["apple", "banana", "cherry"]  
thislist.pop(1)  
print(thislist)
```

[Try it Yourself »](#)

If you do not specify the index, the `pop()` method removes the last item.

```
['apple', 'cherry']
```

Loop Through a List

loop lists

You can loop through the list items by using a `for` loop:

Example

Print all items in the list, one by one:

```
thislist = ["apple", "banana", "cherry"]  
for x in thislist:  
    print(x)
```

[Try it Yourself »](#)



apple
banana
cherry

Loop Through the Index Numbers

You can also loop through the list items by referring to their index number.

`len`
`range`

Use the `range()` and `len()` functions to create a suitable iterable.

Example

Print all items by referring to their index number:

```
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
    print(thislist[i])
```

Using a While Loop

You can loop through the list items by using a `while` loop.

Use the `len()` function to determine the length of the list, then start at 0 and loop your way through the list items by referring to their indexes.

Remember to increase the index by 1 after each iteration.

Example

Print all items, using a `while` loop to go through all the index numbers

```
thislist = ["apple", "banana", "cherry"]
i = 0
while i < len(thislist):
    print(thislist[i])
    i = i + 1
```

Looping Using List Comprehension

List Comprehension offers the shortest syntax for looping through lists:

Example

A short hand `for` loop that will print all items in a list:

```
thislist = ["apple", "banana", "cherry"]
[print(x) for x in thislist]
```

List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Example:

Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.

Without list comprehension you will have to write a `for` statement with a conditional test inside:

Example

[Get your own Python Server](#)

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
    if "a" in x:
        newlist.append(x)

print(newlist)
```

```
['apple', 'banana', 'mango']
```

The Syntax

```
newlist = [expression for item in iterable if condition == True]
```

The return value is a new list, leaving the old list unchanged.

Condition

The *condition* is like a filter that only accepts the items that evaluate to `True`.

Example

Only accept items that are not "apple":

```
newlist = [x for x in fruits if x != "apple"]
```

```
['banana', 'cherry', 'kiwi', 'mango']
```

The condition `if x != "apple"` will return `True` for all elements other than "apple", making the new list contain all fruits except "apple".

The *condition* is optional and can be omitted:

Example

Accept only numbers lower than 5:

```
newlist = [x for x in range(10) if x < 5]
```

```
[0, 1, 2, 3, 4]
```

Expression

The `expression` is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list:

Example

Set the values in the new list to upper case:

```
newlist = [x.upper() for x in fruits]
```

[Try it Yourself »](#)

```
['APPLE', 'BANANA', 'CHERRY', 'KIWI', 'MANGO']
```

You can set the outcome to whatever you like:

Example

Return "orange" instead of "banana":

```
newlist = [x if x != "banana" else "orange" for x in fruits]
```

[Try it Yourself »](#)

```
['apple', 'orange', 'cherry', 'kiwi', 'mango']
```

Sort List Alphanumerically

List objects have a `sort()` method that will sort the list alphanumerically, ascending, by default:

Example

[Get y](#)

Sort the list alphabetically:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)
```

```
['banana', 'kiwi', 'mango', 'orange', 'pineapple']
```

Example

Sort the list numerically:

```
thislist = [100, 50, 65, 82, 23]
thislist.sort()
print(thislist)
```

[Try it Yourself »](#)

Sort Descending

降序

To sort descending, use the keyword argument `reverse = True`:

Example

Sort the list descending:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort(reverse = True)
print(thislist)
```

```
['pineapple', 'orange', 'mango', 'kiwi', 'banana']
```

不区分大小写排序

默认情况下，该 `sort()` 方法区分大小写，导致所有大写字母都排在小写字母之前：

例子

区分大小写的排序可能会产生意想不到的结果：

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort()
print(thislist)
```

自己尝试一下»

幸运的是，在对列表进行排序时，我们可以使用内置函数作为关键函数。

因此，如果您想要一个不区分大小写的排序函数，请使用 `str.lower` 作为关键函数：

例子

对列表执行不区分大小写的排序：

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key = str.lower)
print(thislist)
```

相反的顺序

如果您想要反转列表的顺序（无论字母表是什么）怎么办？

该 `reverse()` 方法反转元素的当前排序顺序。

例子

反转列表项的顺序：

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.reverse()
print(thislist)
```

您不能仅通过键入 来复制列表 `list2 = list1`，因为： `list2` 只会是对的 引用 `list1`，并且在 中所做的更改 `list1` 也会自动在 中进行 `list2`。

有多种方法可以进行复制，一种方法是使用内置的 List 方法 `copy()`。

例子

[获取您自己的 Python 服务器](#)

使用以下方法复制列表 `copy()`：

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

[自己尝试一下»](#)

另一种制作副本的方法是使用内置方法 `list()`。

例子

使用以下方法复制列表 `list()`：

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

在 Python 中，有多种方法可以连接两个或多个列表。

最简单的方法之一是使用 `+` 运算符。

例子

连接两个列表：

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

list3 = list1 + list2
print(list3)
```

自己尝试一下»

连接两个列表的另一种方法是将 `list2` 中的所有项目逐一附加到 `list1` 中：

例子

将 `list2` 追加到 `list1` 中：

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

for x in list2:
    list1.append(x)
```

或者您可以使用该 `extend()` 方法，其目的是将一个列表中的元素添加到另一个列表：

例子

使用 `extend()` 方法将 `list2` 添加到 `list1` 的末尾：

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

list1.extend(list2)
print(list1)
```

自己尝试一下»

2.tuples

有序 且不可更改的集合 允许有重复值 用圆括号书写

确定长度 用len函数

```
thistuple = tuple(("apple", "banana", "cherry"))
print(len(thistuple))
```

3

注意用逗号隔开

创建包含一项的元组

要创建只有一项的元组，必须在该项后面添加一个逗号，否则 Python 不会将其识别为元组。

例子

一项元组，记住逗号：

```
thistuple = ("apple",)
print(type(thistuple))

#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

tuple() 构造函数

也可以使用 `tuple()` 构造函数来创建元组。

例子

使用 `tuple()` 方法创建元组：

```
thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets
print(thistuple)
```

access tuple

用方括号的索引号来访问元组

Access Tuple Items

You can access tuple items by referring to the index number, inside square brackets:

Example

Print the second item in the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

其他的和前面list同理 只是方括号换成了圆括号

但由于不可更改 还是有一些不同

更改元组值

元组一旦创建，就无法更改其值。元组是不可更改的，或者也称为不可变的。

但有一个解决方法。您可以将元组转换为列表，更改列表，然后将列表转换回元组。

例子

将元组转换为列表以便能够更改它：

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

由于元组是不可变的，因此它们没有内置 `append()` 方法，但还有其他方法可以向元组添加项目。

1.转换为列表：就像更改元组的解决方法一样，您可以将其转换为列表，添加项目，然后将其转换回元组。

例子

将元组转换为列表，添加“orange”，然后将其转换回元组：

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
```

[自己尝试一下»](#)

2.将元组添加到元组中。您可以将元组添加到元组，因此如果您想添加一项（或多项），请使用该项创建一个新元组，并将其添加到现有元组中：

例子

创建一个值为“orange”的新元组，并添加该元组：

```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y

print(thistuple)
```

元组是不可更改的，因此您无法从中删除项目，但您可以使用与我们用于更改和添加元组项目相同的解决方法：

例子

将元组转换为列表，删除“apple”，然后将其转换回元组：

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.remove("apple")
thistuple = tuple(y)
```

[自己尝试一下»](#)

或者您可以完全删除元组：

例子

该 `del` 关键字可以完全删除元组：

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

但是，在 Python 中，我们也可以将值提取回变量中。这称为“拆包”：

例子

解压元组：

```
fruits = ("apple", "banana", "cherry")  
  
(green, yellow, red) = fruits  
  
print(green)  
print(yellow)  
print(red)
```

使用星号*

如果变量的数量少于值的数量，您可以 * 在变量名称中添加，并且值将以列表的形式分配给变量：

例子

将其余值分配为名为“red”的列表：

```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")  
  
(green, yellow, *red) = fruits  
  
print(green)  
print(yellow)  
print(red)
```

loop

遍历项目并打印值：

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
    print(x)
```

自己尝试一下»

for 在[Python For 循环](#)章节中了解有关循环的更多信息。

循环遍历索引号

您还可以通过引用元组项的索引号来循环遍历元组项。

使用 `range()` 和 `len()` 函数创建合适的可迭代对象。

例子

通过引用索引号打印所有项目：

```
thistuple = ("apple", "banana", "cherry")
for i in range(len(thistuple)):
    print(thistuple[i])
```

While

使用 While 循环

您可以使用循环来遍历元组项 `while`。

使用该 `len()` 函数确定元组的长度，然后从 0 开始并通过引用元组项的索引来循环遍历元组项。

请记住在每次迭代后将索引增加 1。

例子

打印所有项目，使用 `while` 循环遍历所有索引号：

```
thistuple = ("apple", "banana", "cherry")
i = 0
while i < len(thistuple):
    print(thistuple[i])
    i = i + 1
```

```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2
print(tuple3)
```

自己尝试一下»

join

元组相乘

如果要将元组的内容乘以给定次数，可以使用 `*` 运算符：

例子

将水果元组乘以 2：

```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2

print(mytuple)
```

3.set

无序 不可更改 不允许重复 但可以添加删除新项目 但无法更改项目 大括号书写

True 并被 1 认为是相同的值：

```
thisset = {"apple", "banana", "cherry", True, 1, 2}  
print(thisset)
```

自己尝试一下»

注意：值 False 和 0 被视为集合中的相同值，并被视为重复项：

例子

False 并被 0 认为是相同的值：

```
thisset = {"apple", "banana", "cherry", False, True, 0}  
print(thisset)
```

集合的数据类型是什么？

```
myset = {"apple", "banana", "cherry"}  
print(type(myset))
```

自己尝试一下»

set() 构造函数

也可以使用 set() 构造函数来创建集合。

例子

使用 set() 构造函数创建一个集合：

```
thisset = set(("apple", "banana", "cherry")) # note the double round-brackets  
print(thisset)
```

access

循环遍历集合并打印值：

```
thisset = {"apple", "banana", "cherry"}  
  
for x in thisset:  
    print(x)
```

自己尝试一下»

例子

检查集合中是否存在“banana”：

```
thisset = {"apple", "banana", "cherry"}  
  
print("banana" in thisset)
```

无法更改

add

要将一项添加到集合中，请使用该 `add()` 方法。

例子

使用以下方法将项目添加到集合中 `add()`：

```
thisset = {"apple", "banana", "cherry"}  
  
thisset.add("orange")  
  
print(thisset)
```

添加集合

要将另一个集合中的项目添加到当前集合中，请使用 `update()` 方法。

例子

添加元素 from `tropical` into `thisset` :

```
thisset = {"apple", "banana", "cherry"}  
tropical = {"pineapple", "mango", "papaya"}  
  
thisset.update(tropical)  
  
print(thisset)
```

添加任何可迭代对象

方法中的对象 `update()` 不一定是集合，它可以是任何可迭代对象（元组、列表、字典等）。

例子

将列表的元素添加到集合中：

```
thisset = {"apple", "banana", "cherry"}  
mylist = ["kiwi", "orange"]  
  
thisset.update(mylist)  
  
print(thisset)
```

remove

去除“香蕉”的方法如下 `discard()` :

```
thisset = {"apple", "banana", "cherry"}  
thisset.discard("banana")  
print(thisset)
```

自己尝试一下»

注意：如果要删除的项目不存在，`discard()` 则 不会引发错误。

您还可以使用该 `pop()` 方法删除项目，但此方法将删除随机项目，因此您无法确定删除了哪些项目。

该方法的返回值 `pop()` 是被删除的项目。

例子

使用以下 `pop()` 方法删除随机项：

```
thisset = {"apple", "banana", "cherry"}  
x = thisset.pop()  
print(x)
```

该 `clear()` 方法清空集合：

```
thisset = {"apple", "banana", "cherry"}  
thisset.clear()  
print(thisset)
```

自己尝试一下»

例子

关键字 `del` 将完全删除集合：

```
thisset = {"apple", "banana", "cherry"}  
del thisset  
print(thisset)
```

join

这两个方法都会排除任何重复的项目

该 `union()` 方法返回一个新集合，其中包含两个集合中的所有项目：

```
set1 = {"a", "b", "c"}  
set2 = {1, 2, 3}  
  
set3 = set1.union(set2)  
print(set3)
```

自己尝试一下»

例子

该 `update()` 方法将 `set2` 中的项插入到 `set1` 中：

```
set1 = {"a", "b", "c"}  
set2 = {1, 2, 3}  
  
set1.update(set2)  
print(set1)
```

4. dictionary

有序 可更改 不允许重复 大括号书写 有key, value

(重复值将覆盖现有值)

字典项目是有序的、可更改的并且不允许重复。

字典项以键:值对的形式呈现，可以通过键名来引用。

例子

打印字典的“品牌”值：

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
print(thisdict["brand"])
```

```
thisdict = dict(name = "John", age = 36, country = "Norway")  
print(thisdict)
```

```
{'name': 'John', 'age': 36, 'country': 'Norway'}
```

access

获取“model”键的值：

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
x = thisdict["model"]
```

自己尝试一下»

还有一个方法 `get()` 可以给你相同的结果：

例子

获取“model”键的值：

```
x = thisdict.get("model")
```

```
car = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
x = car.keys()  
  
print(x) #before the change  
  
car["color"] = "white"  
  
print(x) #after the change
```

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
x = thisdict.items()  
  
print(x)
```

```
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
```

要确定字典中是否存在指定的键，请使用关键字 `in`：

例子

检查字典中是否存在“model”：

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
if "model" in thisdict:  
    print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

change

Change the "year" to 2018:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict["year"] = 2018
```

[Try it Yourself »](#)

update

Update Dictionary

The `update()` method will update the dictionary with the items from the given argument.

The argument must be a dictionary, or an iterable object with key:value pairs.

Example

Update the "year" of the car by using the `update()` method:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.update({"year": 2020})
```

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict["color"] = "red"  
print(thisdict)
```

自己尝试一下»

更新字典

该 `update()` 方法将使用给定参数中的项目更新字典。如果该项目不存在，则会添加该项目。

参数必须是字典或具有键：值对的可迭代对象。

例子

使用以下方法将颜色项添加到字典中 `update()` :

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.update({"color": "red"})
```

remove

该 `pop()` 方法删除具有指定键名称的项目：

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.pop("model")  
print(thisdict)
```

自己尝试一下»

例子

该 `popitem()` 方法删除最后插入的项目（在 3.7 之前的版本中，会删除随机项目）：

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.popitem()  
print(thisdict)
```

例子

关键字 `del` 删除具有指定键名称的项目：

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
del thisdict["model"]  
print(thisdict)
```

loop

您还可以使用该 **values()** 方法返回字典的值：

```
for x in thisdict.values():
    print(x)
```

自己尝试一下»

例子

您可以使用该 **keys()** 方法返回字典的键：

```
for x in thisdict.keys():
    print(x)
```

自己尝试一下»

例子

使用以下 方法循环遍历键和值 **items()**：

```
for x, y in thisdict.items():
    print(x, y)
```

使用以下方法复制字典 `copy()` :

`copy`

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
mydict = thisdict.copy()  
print(mydict)
```

自己尝试一下»

另一种进行复制的方法是使用内置函数 `dict()`。

例子

使用以下函数复制字典 `dict()` :

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
mydict = dict(thisdict)  
print(mydict)
```