Python 3 Tutorial

https://www.sololearn.com/Play/python

翻译：Vincent

2018/11/1

linqinghong@email.szu.edu.cn

1.基本概念

1.1 什么是 Python？

✧ 欢迎使用 Python！

• **Python** 是一种高级编程语言，其应用涉及多个领域，包括 Web 编程，脚本编写，科学计算和人工智能。

• 它非常受欢迎，并被 Google，NASA，CIA 和迪士尼等机构采用。
Python 在运行时由解释器处理。在执行程序之前无需编译程序。

✧ 欢迎使用 Python！

• Python 的三个主要版本是 1.x，2.x 和 3.x. 它们被细分为次要版本，例如 2.7 和 3.3。
• 为 Python 3.x 编写的代码保证可以被所有的未来版本使用。
• 目前使用 Python 版本 2.x 和 3.x。
• 本课程涵盖 **Python 3.x**，但从一个版本更改为另一个版本并不难。

• Python 有几种不同的实现，用各种语言编写。
• 本课程中使用的版本 **CPython** 是迄今为止最受欢迎的版本。
解释器是一个运行用 Python 等解释语言编写的脚本的程序。

1.2 你的第一个程序

✧ 你的第一个程序

• 让我们从创建一个显示"Hello world!"的短程序开始。
• 在 Python 中，我们使用 **print** 语句输出文本：

```
>>> print('Hello world!')
Hello world!
```

• 恭喜！你已经写了第一个程序。

在我们的 Code Playground 上运行，保存和共享您的 Python 代码，无需安装任何其他软件。
使用计算机时，您需要从 www.python.org 下载并安装 Python。
请注意上面代码中的>>>。它们是 Python 控制台的提示符号。Python 是一种解释型语言，这意味着每一行都在输入时执行。Python 还包括 **IDLE**，即集成开发环境，其中包括用于编写和调试整个程序的工具。

✧ 打印文字

print 语句还可用于输出多行文本。
例如：

```
>>> print('Hello world!')
Hello world!
>>> print('Hello world!')
Hello world!
>>> print('Spam and eggs...')
Spam and eggs...
```

Python 代码通常包含对喜剧团体 **Monty Python** 的引用。这就是为什么"Spam"和"eggs"这两个词经常被用作 Python 中的占位符变量，其中"foo"和"bar"将用于其他编程语言。

1.3.简单操作

✧ 简单操作

- Python 具有执行计算的能力。
- 直接在 Python 控制台中输入计算，它将输出答案。

```
>>> 2 + 2
4
>>> 5 + 4 - 3
6
```

这里的加号和减号周围的空格是**可选**的（代码可以在没有它们的情况下工作），但它们使它更容易阅读。

✧ 简单操作

- Python 还执行乘法和除法，使用**星号**表示乘法，使用**正斜杠**表示除法。

- 使用**括号**来确定首先执行哪些操作。

```
>>> 2 * (3 + 4)
14
>>> 10/2
5
```

使用单个斜杠来划分数字会产生十进制（或**浮点数**，因为它在编程中调用）。我们将在后面的课程中详细介绍 **floats**。

✧ 简单操作

- 减号表示**负数**。
- 对负数进行操作，就像在正数上一样。

```
>>> -7
-7
>>>(-7 + 2) * (-4)
20
```

加号也可以放在数字前面，但这没有作用，只强调一个数字有利于提高代码的可读性。

✧ 简单操作

- 在 Python 中除以零会产生**错误**，因为无法计算答案。

```
>>> 11 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

在 Python 中，错误消息的最后一行表示错误的类型。
仔细阅读错误消息，因为它们经常告诉您如何修复程序！

1.4.浮点数

✧ 浮点数

• **Floats** 在 Python 中用于表示非整数的数字。
• 表示为浮点数的一些数字示例是 0.5 和-7.8237591。
可以通过输入带小数点的数字或使用整数除法等操作直接创建它们。数字末尾的额外零被忽略。

```
>>> 3/4
0.75
>>> 9.8765000
9.8765
```

计算机不能完全准确地存储浮点数，就像我们无法写下 1/3 的完整十进制扩展(0.3333333333333333 ...)一样。记住这一点，因为它经常会导致令人激怒的错误！

✧ 浮点数

• 如前所述，除以任意两个整数会产生**浮点数**。
• 在两个浮点数上运行操作，或者在浮点数和整数上，也可以生成浮点数。

```
>>> 8 / 2
4.0
>>> 6 * 7.0
42.0
>>> 4 + 1.65
5.65
```

浮点数可以添加到整数，因为 Python 会默认地将整数转换为浮点数。但是，这种隐式转换是异常而不是 Python 中的规则 - 如果要对它们进行操作，通常必须手动转换值。

1.5.其他数值运算

✧ 幂

• 除了加法，减法，乘法和除法之外，Python 还支持取**幂**，即将一个数字提到另一个数字的幂。使用两个星号执行该操作。

```
>>> 2**5
32
>>> 9 ** (1/2)
3.0
```

您可以将指数链接在一起。换句话说，您可以将数字增加到多个幂。例如，2**3**2。

✧ 商和求余

• 要确定除法的**商**和**余数**，请分别使用**取整除**和**取模**运算符。
• 使用两个正斜杠完成取整除。
• 取模运算符使用百分号(%)执行。
• 这些运算符可以与浮点数和整数一起使用。

• 该代码表明 20 取整除 6 等于 3，而 1.25 除以 0.5 的余数为 0.25。

```
>>> 20 // 6
3
>>> 1.25 % 0.5
0.25
```

在上面的示例中，20 % 6 将返回 2，因为 3 * 6 + 2 等于 20。

1.6.字符串

✧ 字符串

• 如果要在 Python 的中使用文本，则必须使用**字符串**。
• 通过在**两个单引号或双引号**之间输入文本来创建**字符串**。

• 当 Python 控制台显示字符串时，它通常使用单引号。用于字符串的分隔符不会以任何方式影响其行为。

```
>>> "Python is fun!"
'Python is fun!'
>>> 'Always look on the bright side of life'
'Always look on the bright side of life'
```

Python 中有另一种名为 docstrings 的字符串类型，用于块注释，但它实际上是一个字符串。您将在以后的课程中了解这一点。

✧ 字符串

• 某些字符不能直接包含在字符串中。例如，双引号不能直接包含在双引号字符串中；这会导致它过早结束。

• 必须通过在它们之前放置**反斜杠**来转义这些字符。
• 必须转义的其他常见字符有换行符和反斜杠。
• 双引号只需要在双引号字符串中进行转义，单引号字符串也是如此。

```
>>> 'Brian\'s mother: He\'s not the Messiah. He\'s a very naughty boy!'
'Brian's mother: He's not the Messiah. He's a very naughty boy!'
```

• **\ n** 表示新行。
反斜杠也可用于转义制表符，任意 Unicode 字符以及无法可靠打印的各种其他内容。这些字符称为转义字符。

✧ 换行

• Python 提供了一种简单的方法来避免手动编写"\n"来转义字符串中的换行符。创建一个包含**三组引号**的字符串，按 Enter 键创建的换行符将自动为您转义。

```
>>> """Customer: Good morning.
Owner: Good morning, Sir. Welcome to the National Cheese Emporium."""

'Customer: Good morning.\nOwner: Good morning, Sir. Welcome to the National Cheese Emporium.'
```

如您所见，**\n** 在我们按 Enter 键的地方自动输出。

1.7.简单输入和输出

✧ 输出

• 通常，程序接受**输入**并处理它以产生**输出**。
• 在 Python 中，您可以使用 **print** 函数来生成输出。这会在屏幕上显示某些内容的文本表示。

```
>>> print(1 + 1)
2
>>> print("Hello\nWorld!")
Hello
World!
```

打印字符串时，不会显示其周围的引号。

✧ 输入

• 要在 Python 中获得用户的输入，您可以使用直观命名的**输入**函数。
• 该功能提示用户输入，并返回它们作为字符串输入的内容（内容自动转义）。

```
>>> input("Enter something please: ")
Enter something please: This is what\nthe user enters!

'This is what\\nthe user enters!'
```

在 Python 控制台上，**打印**和**输入**功能不是很有用，它自动进行输入和输出。但是，它们在实际程序中非常有用。

1.8.字符串操作

✧ 连接

• 与整数和浮点数一样，可以通过**连接**添加 Python 中的字符串，这可以在任何两个字符串上完成。
• 连接字符串时，它们是用单引号还是双引号创建并不重要。

```
>>> "Spam" + 'eggs'
'Spameggs'

>>> print("First string" + ", " + "second string")
First string, second string
```

您不能使用数字（整数）连接字符串。在下一课中找出原因。

✧ 连接

• 即使您的字符串包含数字，它们仍然作为字符串而不是整数添加。将字符串添加到数字会产生错误，即使它们看起来相似，但它们是两个不同的实体。

```
>>> "2" + "2"
'22'
>>> 1 + '2' + 3 + '4'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

在将来的课程中，只会显示最后一行错误消息，因为它是唯一一个提供有关已发生错误类型的详细信息的消息。

✧ 字符串操作

• 字符串也可以**乘**以整数。这会生成原始字符串的重复版本。字符串和整数的顺序无关紧要，但字符串通常是第一位的。

•字符串不能与其他字符串相乘。即使浮点数是整数，字符串也不能乘以浮点数。

```
>>> print("spam" * 3)
spamspamspam

>>> 4 * '2'
'2222'

>>> '17' * '87'
TypeError: can't multiply sequence by non-int of type 'str'

>>> 'pythonisfun' * 7.0
TypeError: can't multiply sequence by non-int of type 'float'
```

尝试将字符串乘以 0，看看会发生什么。

1.9.类型转换

✧ 类型转换

• 在 Python 中，由于涉及的类型，不可能完成某些操作。例如，您不能将包含数字 2 和 3 的两个字符串一起添加以生成整数 5，因为操作将在字符串上执行，结果为'23'。
• 解决方案是**类型转换**。
• 在该示例中，您将使用 **int** 函数。

```
>>> "2" + "3"
'23'
>>> int("2") + int("3")
5
```

在 Python 中，到目前为止我们使用的类型是**整数**，**浮点数**和**字符串**。 用于转换为这些的函数分别是 **int**，**float** 和 **str**。

✧ 类型转换

• 类型转换的另一个例子是将用户输入（**字符串**）转换为数字（**整数**或**浮点数**），以便进行计算。

```
>>> float(input("Enter a number: ")) + float(input("Enter another number: "))
Enter a number: 40
Enter another number: 2
42.0
```

传递非整数或浮点值将导致错误。

1.10.变量

✧ 变量

• **变量**在大多数编程语言中起着非常重要的作用，Python 也不例外。变量允许您通过将值分配给名称来存储值，该名称可用于在之后的程序中引用该值。

• 要分配变量，请使用**一个等号**。与我们迄今为止所看到的大多数代码行不同，它不会在 Python 控制台上产生任何输出。

```
>>> x = 7
>>> print(x)
7
>>> print(x + 3)
10
>>> print(x)
7
```

您可以使用变量执行相应的操作，就像使用数字和字符串一样。如您所见，变量在整个程序中存储其值。

✧ 变量

• 可以根据需要多次重新分配变量，以便更改其值。
• 在 Python 中，变量没有特定的类型，因此您可以将字符串分配给变量，然后将整数分配给同一个变量。

```
>>> x = 123.456
>>> print(x)
123.456
>>> x = "This is a string"
>>> print(x + "!")
This is a string!
```

但是，这不是好习惯。为避免错误，请尽量避免使用不同的数据类型覆盖相同的变量。

✧ 变量名称

• Python 变量名称中限制了使用的字符。允许的字符包括字母，数字和下划线。而且，他们不能从数字开始。
• 不遵循这些规则会导致错误。

```
>>> this_is_a_normal_name = 7

>>> 123abc = 7
SyntaxError: invalid syntax

>>> spaces are not allowed
SyntaxError: invalid syntax
```

Python 是一种区分大小写的编程语言。因此，**Lastname** 和 **lastname** 是 Python 中的两个不同的变量名。

✧ 变量

• 尝试引用未分配的变量会导致**错误**。
• 您可以使用 **del** 语句删除变量，这意味着将删除从名称到值的引用，并尝试使用该变量会导致错误。删除的变量可以正常重新分配。

```
>>> foo = "a string"
>>> foo
'a string'
>>> bar
NameError: name 'bar' is not defined
>>> del foo
>>> foo
NameError: name 'foo' is not defined
```

• 您还可以从用户输入中获取变量的值。

```
>>> foo = input("Enter a number: ")
Enter a number: 7
>>> print(foo)
7
```

变量 **foo** 和 **bar** 称为**伪**变量，这意味着它们在示例代码中用作占位符名称来演示某些内容。

1.11.原位操作

✧ 原位操作

• 原位操作允许您更简洁地编写类似'x = x + 3'的代码，如'x + = 3'。
• 其他运算符也可以使用相同的功能，例如 -、*、/和%。

```
>>> x = 2
>>> print(x)
2
>>> x += 3
>>> print(x)
5
```

✧ 原位操作

• 这些运算符也可以用于数字以外的类型，例如**字符串**。

```
>>> x = "spam"
>>> print(x)
spam

>>> x += "eggs"
>>> print(x)
spameggs
```

许多其他语言都有特殊的运算符，例如'++'作为'x + = 1'的快捷方式。Python **没有**这些。

1.12.使用编辑器

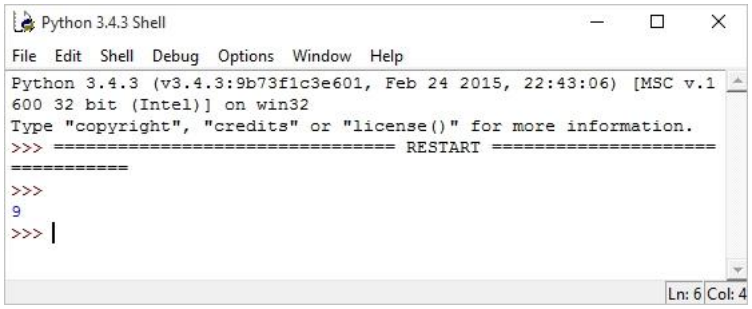
• 到目前为止，我们只使用 Python 与控制台，一次输入和运行一行代码。
• 实际程序的创建方式不同；许多代码行都写在一个文件中，然后用 Python 解释器执行。

• 在 **IDLE** 中，可以通过创建新文件，输入一些代码，保存文件和运行它来完成。这可以通过菜单或键盘快捷键 Ctrl-N，Ctrl-S 和 F5 完成。

• 文件中的每行代码都被解释为您在控制台上一次输入一行代码。

```
x = 7
x = x + 2
print(x)
```

结果：



• Python 源文件的扩展名为**.py**

您可以在我们的 **Code Playground** 上运行，保存和共享您的 Python 代码，而无需安装任何其他软件。

如果需要在计算机上安装软件，请参阅本课程。

2.控制结构

2.1.布尔与比较

✧ 布尔

• Python 中的另一种类型是**布尔**类型。有两个布尔值：**True** 和 **False**。
• 它们可以通过比较值来创建，例如通过使用等于运算符==。

```
>>> my_boolean = True
>>> my_boolean
True

>>> 2 == 3
False
>>> "hello" == "hello"
True
```

注意不要将**赋值**（一个等号）与**比较**混淆（两个等号）。

✧ 比较

• 另一个比较运算符，即**不等于**运算符（**!=**），如果被比较的项不相等则求值为 **True**，如果是，则求值为 **False**。

```
>>> 1 != 1
False
>>> "eleven" != "seven"
True
>>> 2 != 10
True
```

✧ 比较

• Python 还有一些运算符，用于确定一个数字（浮点数或整数）是否大于或小于另一个数字。这些运算符分别是>和<。

```
>>> 7 > 5
True
>>> 10 < 10
False
```

✧ 比较

- 大于或等于，小于或等于运算符是> =和<=。
- 它们与严格大于和小于运算符相同，除了它们在比较**相等**数字时返回 True。

```
>>> 7 <= 8
True
>>> 9 >= 9.0
True
```

大于和小于运算符也可用于按**字典顺序**比较字符串（单词的字母顺序基于其组成字母的字母顺序）。

2.2.if 语句

✧ if 语句

- 如果某个条件成立，您可以使用 **if** 语句来运行代码。
- 如果表达式的计算结果为 **True**，则执行某些语句。否则，它们不会被执行。
- if 语句如下所示：

```
if expression:
    statements
```

Python 使用**缩进**（行开头的空格）来分隔代码块。其他语言，如 C，使用花括号来完成此任务，但在 Python 中缩进是必需的；没有它，程序将无法运作。如您所见，**if** 中的语句应缩进。

✧ if 语句

- 以下是 **if** 语句的示例：

```
if 10 > 5:
```

```
    print("10 greater than 5")

print("Program ended")
```

• 表达式确定 10 是否大于 5。因为它是，缩进语句运行，并输出“10 大于 5”。然后，运行未缩进的语句，该语句不是 **if** 语句的一部分，并显示“Program ended”。

结果：

```
>>>
10 greater than 5
Program ended
>>>
```

• 注意 **if** 语句中表达式末尾的**冒号**。
由于程序包含多行代码，您应该将其创建为单独的文件并运行它。

✧ if 语句

• 要执行更复杂的检查，**if** 语句可以嵌套，一个在另一个内。
•这意味着内部 **if** 语句是外部语句的语句部分。这是检查是否满足多个条件的一种方法。
例如：

```
num = 12
if num > 5:
    print("Bigger than 5")
    if num <=47:
        print("Between 5 and 47")
```

结果：

```
>>>
Bigger than 5
Between 5 and 47
>>>
```

2.3.else 语句

✧ else 语句

• **else** 语句跟在 **if** 语句之后，包含 if 语句求值为 **False** 时调用的代码。
• 与 **if** 语句一样，块内的代码应缩进。

```
x = 4
if x == 5:
    print("Yes")
else:
    print("No")
```

结果：

```
>>>
No
>>>
```

✧ else 语句

• 您可以链接 **if** 和 **else** 语句来确定一系列可能性中的哪个选项为真。
例如：

```
num = 7
if num == 5:
   print("Number is 5")
else:
   if num == 11:
      print("Number is 11")
   else:
      if num == 7:
         print("Number is 7")
      else:
         print("Number isn't 5, 11 or 7")
```

结果：

```
>>>
Number is 7
>>>
```

✧ elif 语句

• **elif**（else if 的缩写）语句是链接 **if** 和 **else** 语句时使用的快捷方式。
•一系列 **if elif** 语句可以有一个最后的 **else** 块，如果 **if** 或 **elif** 表达式都不是 True，则调用它。
例如：

```
num = 7
if num == 5:
    print("Number is 5")
elif num == 11:
    print("Number is 11")
elif num == 7:
    print("Number is 7")
else:
    print("Number isn't 5, 11 or 7")
```

结果：

```
>>>
Number is 7
>>>
```

在其他编程语言中，**elif** 语句的等价物具有不同的名称，包括 **if**，**elseif** 或 **elsif**。

2.4.布尔逻辑

✧ 与

• **布尔逻辑**用于为依赖于多个条件的 **if** 语句创建更复杂的条件。
• Python 的布尔运算符是**与**、**或**和**非**。
•**与**运算符接受两个参数，并且当且仅当两个参数都为 **True** 时才计算为 **True**。否则，它的计算结果为 **False**。

```
>>> 1 == 1 and 2 == 2
True
>>> 1 == 1 and 2 == 3
False
>>> 1 != 1 and 2 == 2
False
>>> 2 < 1 and 3 > 6
False
```

Python 对其布尔运算符使用单词，而大多数其他语言使用诸如&&，||和！之类的符号。

✧ 或

• **or** 运算符也有两个参数。 如果其参数中的任何一个（或两个）为 **True**，则计算结果为 **True**;如果两个参数均为 **False**，则计算结果为 **False**。

```
>>> 1 == 1 or 2 == 2
True
>>> 1 == 1 or 2 == 3
True
>>> 1 != 1 or 2 == 2
True
>>> 2 < 1 or 3 > 6
False
```

✧ 非

• 与我们到目前为止看到的其他运算符不同，**非**仅需要一个参数，而且反转它。**not True** 的结果为 **False**，而 **False** 的结果为 **True**。

```
>>> not 1 == 1
False
>>> not 1 > 7
True
```

您可以使用布尔运算符在 **if** 语句中链接多个条件语句。

2.5.运算符优先级

✧ 运算符优先级

• **运算符优先级**是编程中非常重要的概念。它是操作顺序（在加法之前执行乘法等）的数学概念的扩展，以包括其他运算符，例如布尔逻辑中的运算符。

• 以下代码显示**==**的优先级高于 **or**：

```
>>> False == False or True
True
>>> False == (False or True)
False
>>> (False == False) or True
True
```

Python 的操作顺序与普通数学的顺序相同：首先是括号，然后是取幂，然后是乘法/除法，然后是加法/减法。

✧ 运算符优先级

• 下表列出了所有 Python 的运算符，从最高优先级到最低优先级。

Operator	Description
**	Exponentiation (raise to the power)
~, +, -	Complement, unary plus and minus (method names for the last two are +@ and -@)
*, /, %, //	Multiply, divide, modulo and floor division
+, -	Addition and subtraction
>>, <<	Right and left bitwise shift
&	Bitwise 'AND'
^	Bitwise exclusive 'OR'
|	Bitwise 'OR'
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparison operators, equality operators, membership and identity operators
not	Boolean 'NOT'
and	Boolean 'AND'
or	Boolean 'OR'
=, %=, /=, //=, -=, +=, *=, **=	Assignment operators

同一个框中的运算符具有相同的优先级。

2.6.while 循环

✧ while 循环

• 如果 **if** 语句的条件计算结果为 **True**，则运行 if 语句，如果计算结果为 **False**，则运行 if 语句。
• **while** 语句类似，只是它可以运行多次。只要条件成立，其中的语句就会重复执行。一旦评估为 **False**，就会执行下一部分代码。
• 下面是一个 **while** 循环，包含一个从 1 到 5 计数的变量，此时循环终止。

```
i = 1
while i <=5:
    print(i)
    i = i + 1

print("Finished!")
```

结果：

```
>>>
1
2
3
4
5
Finished!
>>>
```

while 循环体中的代码重复执行。这称为**迭代**。

✧ while 循环

•**无限循环**是一种特殊的 while 循环；它永远不会停止运行。它的状况始终是 **True** 的。
• 无限循环的一个例子：

```
while 1==1:
    print("In the loop")
```

• 该程序将无限期地打印“在循环中”。
您可以使用 Ctrl-C 快捷方式或关闭程序来停止程序的执行。

✧ break

- 要过早结束 **while** 循环，可以使用 **break** 语句。
- 当在循环中遇到 **break** 语句会使循环立即结束。

```
i = 0
while 1==1:
    print(i)
    i = i + 1
    if i >= 5:
        print("Breaking")
        break

print("Finished")
```

结果：

```
>>>
0
1
2
3
4
Breaking
Finished
>>>
```

在循环外使用 **break** 语句会导致错误。

✧ continue

- 可以在循环中使用的另一个语句是 **continue**。
- 与 **break** 不同，**continue** 会跳回到循环的顶部，而不是停止它。

```
i = 0
while True:
    i = i +1
    if i == 2:
        print("Skipping 2")
        continue
    if i == 5:
        print("Breaking")
        break
```

```
    print(i)

print("Finished")
```

结果：

```
>>>
1
Skipping 2
3
4
Breaking
Finished
>>>
```

• 基本上，**continue** 语句会停止当前迭代并继续下一个迭代。
在循环外使用 **continue** 语句会导致错误。

2.7.列表

✧ 列表

• **列表**是 Python 中的另一种对象。它们用于存储索引的项目列表。
• 使用带**逗号**的**方括号**创建列表。
• 可以使用方括号中的索引访问列表中的特定项。
例如：

```
words = ["Hello", "world", "!"]
print(words[0])
print(words[1])
print(words[2])
```

结果：

```
>>>
Hello
world
!
>>>
```

第一个列表项的索引是 **0**，而不是 **1**，正如预料的那样。

✧ 列表

• 使用一对空方括号创建一个空列表。

```
empty_list = []
print(empty_list
```

结果：

```
>>>
[]
>>>
```

大多数情况下，逗号不会跟随列表中的最后一项。但是，在那里放置一个是完全有效的，在某些情况下鼓励它。

✧ 列表

• 通常，列表将包含单个项类型的项，但也可以包含多个不同类型。
• 列表也可以**嵌套**在其他列表中。

```
number = 3
things = ["string", 0, [1, 2, number], 4.56]
print(things[1])
print(things[2])
print(things[2][2])
```

结果：

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

列表列表通常用于表示 2D 网格，因为 Python 缺少将在其他语言中使用的多维数组。

✧ 列表

• 索引超出列表的范围会导致 IndexError。
• 某些类型（如**字符串**）可以像列表一样编制索引。索引**字符串**的行为就像索引

包含字符串中每个字符的列表一样。
• 对于其他类型，例如整数，不可能索引它们，并且它会导致 TypeError。

```
str = "Hello world!"
print(str[6])
```

结果：

```
>>>
w
>>>
```

2.8.列表操作

✧ 列表操作

• 可以重新分配列表中某个索引处的元素。
例如：

```
nums = [7, 7, 7, 7, 7]
nums[2] = 5
print(nums)
```

结果：

```
>>>
[7, 7, 5, 7, 7]
>>>
```

✧ 列表操作

• 列表可以字符串的方式加和乘。
例如：

```
nums = [1, 2, 3]
print(nums + [4, 5, 6])
print(nums * 3)
```

结果：

```
>>>
[1, 2, 3, 4, 5, 6]
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>>
```

列表和字符串在很多方面类似 - 字符串可以被认为是无法更改的字符列表。

✧ 列表操作

• 要检查项目是否在列表中，可以使用 **in** 运算符。如果项目在列表中出现一次或多次，则返回 **True**，否则返回 **False**。

```
words = ["spam", "egg", "spam", "sausage"]
print("spam" in words)
print("egg" in words)
print("tomato" in words)
```

结果：

```
>>>
True
True
False
>>>
```

in 运算符还用于确定字符串是否是另一个字符串的子字符串。

✧ 列表操作

• 要检查项目是否不在列表中，您可以通过以下方式之一使用 **not** 运算符：

```
nums = [1, 2, 3]
print(not 4 in nums)
print(4 not in nums)
print(not 3 in nums)
print(3 not in nums)
```

结果：

```
>>>
True
True
False
False
>>>
```

2.9.列表函数

✧ 列表函数

• 改变列表的另一种方法是使用 **append** 方法。这会将项目添加到现有列表的末尾。

```
nums = [1, 2, 3]
nums.append(4)
print(nums)
```

结果：

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

附加前的**点**是因为它是列表类的**方法**。方法将在后面的课程中解释。

✧ 列表函数

• 要获取列表中的元素数，可以使用 **len** 函数。

```
nums = [1, 3, 5, 2, 4]
print(len(nums))
```

结果：

```
>>>
5
>>>
```

与 **append** 不同，**len** 是一个正常的函数，而不是一个方法。这意味着它在被调用的列表之前写入，没有点。

✧ 列表函数

• **insert** 方法类似于 **append**，除了它允许您在列表中的任何位置插入新元素，而不是在最后。

```
words = ["Python", "fun"]
index = 1
words.insert(index, "is")
print(words)
```

结果：

```
>>>
['Python', 'is', 'fun']
>>>
```

✧ 列表函数

- **index** 方法查找列表项的第一个匹配项并返回其索引。
- 如果该项不在列表中，则会引发 ValueError。

```
letters = ['p', 'q', 'r', 's', 'p', 'u']
print(letters.index('r'))
print(letters.index('p'))
print(letters.index('z'))
```

结果：

```
>>>
2
0
ValueError: 'z' is not in list
>>>
```

列表还有一些更有用的功能和方法。
max(list)：返回具有最大值的列表项
min(list)：返回值最小的列表项
list.**count**(obj)：返回项目在列表中出现次数的计数
list.**remove**(obj)：从列表中删除对象
list.**reverse**()：反转列表中的对象

2.10.范围

✧ Range

- **range** 函数创建一个连续的数字列表。
- 下面的代码生成一个包含所有整数的列表，最多 10 个。

```
numbers = list(range(10))
print(numbers)
```

结果：

```
>>>
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

对**列表**的调用是必要的，因为**范围**本身会创建一个**范围对象**，如果要将其用作列表对象，则必须将其转换为**列表**。

✧ Range

- 如果使用一个参数调用 **range**，则会生成一个值为 0 到该参数的对象。
- 如果使用两个参数调用它，它将生成从第一个到第二个的值。

例如：

```
numbers = list(range(3, 8))
print(numbers)

print(range(20) == range(0, 20))
```

结果：

```
>>>
[3, 4, 5, 6, 7]

True
>>>
```

✧ Range

- **range** 可以有第三个参数，它决定了生成序列的间隔。第三个参数必须是整数。

```
numbers = list(range(5, 20, 2))
print(numbers)
```

结果：

```
>>>
[5, 7, 9, 11, 13, 15, 17, 19]
>>>
```

2.11.for 循环

✧ 循环

• 有时，您需要在列表中的每个项目上执行代码。这称为迭代，可以使用 **while** 循环和计数器变量来完成。

例如：

```
words = ["hello", "world", "spam", "eggs"]
counter = 0
max_index = len(words) - 1

while counter <= max_index:
    word = words[counter]
    print(word + "!")
    counter = counter + 1
```

结果：

```
>>>
hello!
world!
spam!
eggs!
>>>
```

上面的示例遍历列表中的所有项目，使用它们的索引访问它们，并使用感叹号打印它们。

✧ for 循环

• 使用 **while** 循环遍历列表需要相当多的代码，因此 Python 提供 **for** 循环作为实现相同功能的快捷方式。
• 可以使用 **for** 循环编写前一个示例中的相同代码，如下所示：

```
words = ["hello", "world", "spam", "eggs"]
for word in words:
    print(word + "!")
```

结果：

```
>>>
hello!
world!
spam!
eggs!
>>>
```

Python 中的 **for** 循环就像其他语言中的 **foreach** 循环一样。

✧ for 循环

• **for** 循环通常用于重复某些代码一定次数。这是通过将 for 循环与 **range** 对象组合来完成的。

```
for i in range(5):
    print("hello!")
```

结果：

```
>>>
hello!
hello!
hello!
hello!
hello!
>>>
```

当在 for 循环中使用它时，您不需要在 **range** 对象上调用 **list**，因为它没有被索引，因此不需要列表。

2.12.一个简易计算器

✧ 创建一个计算器

• 本课程是一个示例 Python 项目：一个简单的计算器。
• 每个部分都解释了该计划的不同部分。
• 第一部分是整体菜单。这继续接受用户输入，直到用户输入“退出”，因此使用 **while** 循环。

```
while True:
    print("Options:")
    print("Enter 'add' to add two numbers")
    print("Enter 'subtract' to subtract two numbers")
    print("Enter 'multiply' to multiply two numbers")
    print("Enter 'divide' to divide two numbers")
    print("Enter 'quit' to end the program")
    user_input = input(": ")

    if user_input == "quit":
        break
    elif user_input == "add":
        ...
    elif user_input == "subtract":
        ...
    elif user_input == "multiply":
        ...
    elif user_input == "divide":
        ...
    else:
        print("Unknown input")
```

上面的代码是我们程序的起点。它接受用户输入，并将其与 **if / elif** 语句中的选项进行比较。
break 语句用于停止 while 循环，以防用户输入“quit”。

✧ 创建一个计算器

• 该计划的下一部分是获取用户想要做的事情的数字。
• 下面的代码显示了计算器的加法部分。必须为其他部分编写类似的代码。

```
elif user_input == "add":
```

```
    num1 = float(input("Enter a number: "))
    num2 = float(input("Enter another number: "))
```

• 现在，当用户输入“添加”时，程序会提示输入两个数字，并将它们存储在相应的变量中。
实际上，如果用户在提示输入数字时输入非数字输入，则此代码会崩溃。 我们将在后面的模块中讨论修复这样的问题。

✧ 创建一个计算器

• 程序的最后部分处理用户输入并显示它。
• 此处显示了添加部分的代码。

```
elif user_input == "add":
    num1 = float(input("Enter a number: "))
    num2 = float(input("Enter another number: "))
    result = str(num1 + num2)
    print("The answer is " + result)
```

• 我们现在有一个工作程序提示用户输入，然后计算并打印输入的总和。
必须为其他分支编写类似的代码（用于减法，乘法和除法）。
输出行可以放在 **if** 语句之外，以省略重复的代码。

3.函数和模块

3.1.代码重用

✧ 代码重用

•**代码重用**是任何语言编程的一个非常重要的部分。增加代码规模使其难以维护。
• 要使大型编程项目取得成功，必须遵守“**不要重复自己**”或“**枯燥**”原则。我们已经考虑过这样做的一种方法：使用循环。在本单元中，我们将探讨另外两个：功能和模块。
糟糕、重复的代码遵守 **WET** 原则，它代表 **Write Everything Twice**，或者**我们喜欢打字**。

✧ 函数

• 您已经在以前的课程中使用过函数。
• 任何由**括号**中的信息后跟单词组成的语句都是函数调用。
• 以下是您已经看过的一些示例：

```
print("Hello world!")
range(2, 20)
str(12)
range(10, 20, 3)
```

括号前面的单词是函数**名**，括号内的逗号分隔值是函数**参数**。

3.2.函数

✧ 函数

• 除了使用预定义函数外，还可以使用 **def** 语句创建自己的函数。
•以下是名为 **my_func** 的函数示例。它不需要任何参数，并打印三次“垃圾邮件”。它被定义，然后被调用。函数中的语句仅在调用函数时执行。

```
def my_func():
    print("spam")
    print("spam")
```

```
    print("spam")

my_func()
```

结果：

```
>>>
spam
spam
spam
>>>
```

每个函数中的代码块以冒号（:）开头并**缩进**。

✧ 函数

• 您必须在调用函数之前定义它们，就像在使用它们之前必须分配变量一样。

```
hello()

def hello():
      print("Hello world!")
```

结果：

```
>>>
NameError: name 'hello' is not defined
>>>
```

3.3.函数参数

✧ 参数

• 到目前为止，我们所看到的所有函数定义都是零参数的函数，这些参数用空括号调用。
• 但是，大多数函数都需要参数。
• 下面的示例定义了一个带有一个参数的函数：

```
def print_with_exclamation(word):
    print(word + "!")

print_with_exclamation("spam")
```

```
print_with_exclamation("eggs")
print_with_exclamation("python")
```

结果：

```
>>>
spam!
eggs!
python!
>>>
```

如您所见，参数在括号内定义。

✧ 参数

• 您还可以使用多个参数定义函数；用逗号分隔它们。

```
def print_sum_twice(x, y):
    print(x + y)
    print(x + y)

print_sum_twice(5, 8)
```

结果：

```
>>>
13
13
>>>
```

✧ 参数

• 函数参数可以用作函数定义中的变量。但是，它们不能在函数定义之外引用。这也适用于函数内部创建的其他变量。

```
def function(variable):
    variable += 1
    print(variable)

function(7)
print(variable)
```

结果：

```
>>>
8

NameError: name 'variable' is not defined
>>>
```

一般而言，**parameters（形参）**是函数定义中的变量，**arguments（实参）**是调用函数时放入参数的值。

3.4.从函数返回

✧ 从函数返回

• 某些函数（如 **int** 或 **str**）返回一个可以在以后使用的值。
• 要对定义的函数执行此操作，可以使用 **return** 语句。
例如:

```
def max(x, y):
    if x >=y:
        return x
    else:
        return y

print(max(4, 7))
z = max(8, 5)
print(z)
```

结果：

```
>>>
7
8
>>>
```

return 语句不能在函数定义之外使用。

✧ 从函数返回

•从函数返回值后，它会立即停止执行。**return** 语句之后的任何代码都不会发生。
例如：

```
def add_numbers(x, y):
    total = x + y
    return total
    print("This won't be printed")

print(add_numbers(4, 5))
```

结果：

```
>>>
9
>>>
```

3.5.注释和文档

✧ 注释

• **注释**是用于使其更易于理解的代码的注释。 它们不会影响代码的运行方式。
• 在 Python 中，通过插入 **octothorpe**（也称为数字符号或哈希符号：#）来创建注释。该行之后的所有文本都将被忽略。
例如：

```
x = 365
y = 7
# this is a comment

print(x % y) # find the remainder
# print (x // y)
# another comment
```

结果：

```
>>>
1
>>>
```

Python 没有通用的多行注释，C 等编程语言也是如此。

✧ 文档字符串

• **Docstrings**（文档字符串）与注释的用途相似，因为它们旨在解释代码。 但是，它们更具体，并且具有不同的语法。它们是通过在**函数的第一行**下面放置一个包含函数说明的多行字符串来创建的。

```
def shout(word):
    """
    Print a word with an
    exclamation mark following it.
    """
    print(word + "!")

shout("spam")
```

结果：

```
>>>
spam!
>>>
```

与传统注释不同，**文档字符串**在整个程序运行时期间保留。这允许程序员在运行时检查这些注释。

3.6.函数作为对象

✧ 函数

• 虽然它们的创建方式与普通变量不同，但**函数**与任何其他类型的值一样。
• 可以将它们分配并重新指定给变量，然后由这些名称引用。

```
def multiply(x, y):
    return x * y

a = 4
b = 7
operation = multiply
print(operation(a, b))
```

结果：

```
>>>
28
```

```
>>>
```

上面的例子将函数**乘**以一个变量 **operation**。现在，**operation** 也可用于调用该函数。

✧ 函数

• 函数也可以用作其他函数的**参数**。

```
def add(x, y):
    return x + y

def do_twice(func, x, y):
    return func(func(x, y), func(x, y))

a = 5
b = 10

print(do_twice(add,a,b))
```

结果：

```
>>>
30
>>>
```

正如您所看到的，函数 **do_twice** 将函数作为其参数并在内部调用它。

3.7.模块

✧ 模块

• **模块**是其他人为完成常见任务而编写的代码片段，例如生成随机数，执行数学运算等。

• 使用模块的基本方法是在代码顶部添加 **import module_name**，然后使用 **module_name.var** 访问模块中名称为 **var** 的函数和值。
• 例如，以下示例使用**随机**模块生成随机数：

```
import random
```

```
for i in range(5):
    value = random.randint(1, 6)
    print(value)
```

结果：

```
>>>
2
3
6
5
4
>>>
```

该代码使用**随机**模块中定义的 **randint** 函数来打印 1 到 6 范围内的 5 个随机数。

✧ 模块

• 如果您只需要模块中的某些功能，则可以使用另一种**导入**。
• 它们采用 **module_name import var** 的形式，然后可以使用 **var**，就像它在代码中正常定义一样。
• 例如，要从 **math** 模块仅导入 **pi** 常量：

```
from math import pi

print(pi)
```

结果：

```
>>>
3.141592653589793
>>>
```

• 使用逗号分隔列表导入多个对象。例如：

```
from math import pi, sqrt
```

*从模块导入所有对象。例如：**from math import***
通常不鼓励这样做，因为它会将代码中的变量与外部模块中的变量混淆。

✧ 模块

• 尝试导入不可用的模块会导致 ImportError。

```
import some_module
```

结果：

```
>>>
ImportError: No module named 'some_module'
>>>
```

尝试导入不可用的模块会导致 ImportError。

✧ 模块

• 您可以使用 **as** 关键字以不同的名称导入模块或对象。这主要用于模块或对象具有长名称或混淆名称的情况。
例如：

```
from math import sqrt as square_root
print(square_root(100))
```

结果：

```
>>>
10.0
>>>
```

3.8.标准库和 pip

✧ 模块

• Python 中有三种主要类型的模块，您自己编写的模块、从外部源安装的模块以及预装了 Python 的模块。
• 最后一种类型称为**标准库**，包含许多有用的模块。一些标准库的有用模块包括 **string，re，datetime，math，random，os，multiprocessing，subprocess，socket，email，json，doctest，unittest，pdb，argparse** 和 **sys**。

• 可以由标准库完成的任务包括字符串解析，数据序列化，测试，调试和操作日期，电子邮件，命令行参数等等！
Python 广泛的标准库是它作为一种语言的主要优势之一。

✧ 标准库

• 标准库中的一些模块是用 Python 编写的，有些是用 C 语言编写的。
• 大多数都可以在所有平台上使用，但有些是 Windows 或 Unix 特定的。

我们不会涵盖标准库中的所有模块；他们太多了。有关标准库的完整文档，请访问 www.python.org。

✧ 模块

• 许多第三方 Python 模块存储在 **Python Package Index（PyPI）**中。
•安装这些的最好方法是使用一个名为**pip**的程序。默认情况下，这是使用 Python 的现代发行版安装的。如果您没有，可以轻松在线安装。一旦你拥有它，从 PyPI 安装库很容易。查找要安装的库的名称，转到命令行（对于 Windows，它将是命令提示符），然后输入 **pip install library_name**。完成此操作后，导入库并在代码中使用它。

• 使用 **pip** 是在大多数操作系统上安装库的标准方法，但是一些库具有用于 Windows 的预构建二进制文件。这些是普通的可执行文件，允许您使用 GUI 安装库，就像安装其他程序一样。
在命令行输入 **pip** 命令很重要，而不是 Python 解释器。

4.异常和文件

4.1.异常

✧ 异常

• 您已经在以前的代码中看到过**异常**。由于代码或输入错误，它们会在出现问题时发生。发生异常时，程序立即停止。
• 以下代码通过尝试将 7 除以 0 来生成 ZeroDivisionError 异常。

```
num1 = 7
num2 = 0
print(num1/num2)
```

结果：

```
>>>
ZeroDivisionError: division by zero
>>>
```

✧ 异常

• 出于不同原因造成了不同的异常。
• 常见异常：
ImportError：导入失败；
IndexError：列表使用超出范围的数字编制索引；
NameError：使用未知变量；
SyntaxError：无法正确解析代码；
TypeError：在不合适类型的值上调用函数；
ValueError：函数在正确类型的值上调用，但值不合适。
Python 还有其他几个内置异常，例如 ZeroDivisionError 和 OSError。第三方库也经常定义自己的例外。

4.2.异常处理

✧ 异常处理

• 要处理异常，并在发生异常时调用代码，可以使用 **try / except** 语句。
• **try** 块包含可能引发异常的代码。如果发生该异常，则 **try** 块中的代码将停止执行，并且将运行 **except** 块中的代码。如果没有错误发生，则 **except** 块中的代码不会运行。
例如：

```
try:
    num1 = 7
    num2 = 0
    print (num1 / num2)
    print("Done calculation")
except ZeroDivisionError:
    print("An error occurred")
    print("due to zero division")
```

结果：

```
>>>
An error occurred
due to zero division
>>>
```

在上面的代码中，except 语句定义了要处理的异常类型（在我们的例子中，是 **ZeroDivisionError**）。

✧ 异常处理

• **try** 语句可以有多个不同的 **except** 块来处理不同的异常。
• 也可以使用括号将多个异常放入单个 **except** 块中，以使 **except** 块处理所有这些异常。

```
try:
    variable = 10
    print(variable + "hello")
    print(variable / 2)
except ZeroDivisionError:
    print("Divided by zero")
```

```
except (ValueError, TypeError):
    print("Error occurred")
```

结果：

```
>>>
Error occurred
>>>
```

✧ 异常处理

• 没有指定任何异常的 **except** 语句将捕获所有错误。这些应该谨慎使用，因为它们可以捕获意外错误并隐藏编程错误。

例如：

```
try:
    word = "spam"
    print(word / 0)
except:
    print("An error occurred")
```

结果：

```
>>>
An error occurred
>>>
```

处理用户输入时，异常处理特别有用。

4.3.finally

✧ finally

• 无论发生什么错误，都要确保某些代码运行，您可以使用 **finally** 语句。**finally** 语句放在 **try / except** 语句的底部。**finally** 语句中的代码总是在 **try** 中执行代码之后运行，并且可能在 **except** 块后。

```
try:
    print("Hello")
    print(1 / 0)
except ZeroDivisionError:
```

```
    print("Divided by zero")
finally:
    print("This code will run no matter what")
```

结果：

```
>>>
Hello
Divided by zero
This code will run no matter what
>>>
```

✧ finally

•如果在前面的一个块中发生未捕获的异常，则 **finally** 语句中的代码仍然会运行。

```
try:
    print(1)
    print(10 / 0)
except ZeroDivisionError:
    print(unknown_var)
finally:
    print("This is executed last")
```

结果：

```
>>>
1
This is executed last

ZeroDivisionError: division by zero
During handling of the above exception, another exception occurred:
NameError: name 'unknown_var' is not defined
>>>
```

4.4.抛出异常

✧ 抛出异常

• 您可以使用 **raise** 语句引发异常。

```
print(1)
raise ValueError
print(2)
```

结果：

```
>>>
1
ValueError
>>>
```

您需要指定引发的异常**类型**。

✧ 抛出异常

• 可以通过提供有关它们的详细信息的参数来引发异常。
例如：

```
name = "123"
raise NameError("Invalid name!")
```

结果：

```
>>>
NameError: Invalid name!
>>>
```

✧ 抛出异常

• 在 **except** 块中，可以在不使用参数的情况下使用 **raise** 语句来重新引发任何发生的异常。
例如：

```
try:
    num = 5 / 0
except:
    print("An error occurred")
    raise
```

结果：

```
>>>
An error occurred
```

```
ZeroDivisionError: division by zero
>>>
```

4.5.断言

✧ 断言

- **断言**是一种完整性检查，您可以在完成程序测试后打开或关闭它。
- 测试表达式，如果结果为 false，则引发异常。
- 断言是通过使用 **assert** 语句来执行的。

```
print(1)
assert 2 + 2 == 4
print(2)
assert 1 + 1 == 3
print(3)
```

结果：

```
>>>
1
2
AssertionError
>>>
```

程序员经常在函数的开头放置断言以检查有效输入，并在函数调用之后检查有效输出。

✧ 断言

- 如果**断言**失败，断言可以采用传递给 AssertionError 的第二个参数。

```
temp = -10
assert (temp >= 0), "Colder than absolute zero!"
```

结果：

```
>>>
AssertionError: Colder than absolute zero!
>>>
```

可以像使用 **try-except** 语句的任何其他异常一样捕获和处理 AssertionError 异常，但如果不处理，这种类型的异常将终止程序。

4.6.打开文件

✧ 打开文件

• 您可以使用 Python 来读取和写入**文件**的内容。
• 文本文件是最容易操作的。在编辑文件之前，必须使用 **open** 函数打开它。

```
myfile = open("filename.txt")
```

open 函数的参数是文件的**路径**。如果文件位于程序的当前工作目录中，则只需指定其名称。

✧ 打开文件

• 您可以通过向 open 函数应用第二个参数来指定用于打开文件的**模式**。
• 发送“r”表示在读取模式下打开，这是默认设置。
• 发送“w”表示写入模式，用于重写文件的内容。
• 发送“a”表示附加模式，用于将新内容添加到文件末尾。

• 将“b”添加到模式会以**二进制**模式打开它，该模式用于非文本文件（例如图像和声音文件）。
例如：

```
# write mode
open("filename.txt", "w")

# read mode
open("filename.txt", "r")
open("filename.txt")

# binary write mode
open("filename.txt", "wb")
```

您可以对上面的每种模式使用+号，以便为文件提供额外的访问权限。例如，r + 打开文件进行读写。

✧ 打开文件

• 打开并使用文件后，您应该关闭它。
• 这是通过文件对象的 **close** 方法完成的。

```
file = open("filename.txt", "w")
# do stuff to the file
file.close()
```

我们将在即将上课的课程中读/写文件内容。

4.7.读取文件

✧ 读取文件

• 可以使用 **read** 方法读取以文本模式打开的文件的内容。

```
file = open("filename.txt", "r")
cont = file.read()
print(cont)
file.close()
```

这将打印文件“filename.txt”的所有内容。

✧ 读取文件

• 要只读取一定数量的文件，可以提供一个数字作为 **read** 函数的参数。这决定了应该读取的**字节**数。
• 您可以进行更多调用以**读取**同一文件对象，以逐字节读取更多文件。如果没有参数，**read** 将返回文件的其余部分。

```
file = open("filename.txt", "r")
print(file.read(16))
print(file.read(4))
print(file.read(4))
print(file.read())
file.close()
```

就像不传递任何参数一样，负值将返回整个内容。

✧ 读取文件

•读取文件中的所有内容后，任何从该文件中进一步读取的尝试都将返回一个空字符串，因为您尝试从文件末尾读取。

```
file = open("filename.txt", "r")
file.read()
print("Re-reading")
print(file.read())
print("Finished")
file.close()
```

结果：

```
>>>
Re-reading

Finished
>>>
```

就像不传递任何参数一样，负值将返回整个内容。

✧ 读取文件

• 要检索文件中的每一行，可以使用 **readlines** 方法返回一个列表，其中每个元素都是文件中的一行。
例如：

```
file = open("filename.txt", "r")
print(file.readlines())
file.close()
```

结果：

```
>>>
['Line 1 text \n', 'Line 2 text \n', 'Line 3 text']
>>>
```

• 您还可以使用 **for** 循环遍历文件中的行：

```
file = open("filename.txt", "r")
```

```
for line in file:
        print(line)

file.close()
```

结果：

```
>>>
Line 1 text

Line 2 text

Line 3 text
>>>
```

在输出中，每行由空行分隔，因为**打印**功能会在输出结束时自动添加新行。

4.8.写文件

✧ 写文件

• 要写入文件，请使用 **write** 方法，该方法将字符串写入文件。

例如：

```
file = open("newfile.txt", "w")
file.write("This has been written to a file")
file.close()

file = open("newfile.txt", "r")
print(file.read())
file.close()
```

结果：

```
>>>
This has been written to a file
>>>
```

如果文件尚不存在，“w”模式将创建一个文件。

✧ 写文件

• 在写入模式下打开文件时，将删除文件的现有内容。

```
file = open("newfile.txt", "r")
print("Reading initial contents")
print(file.read())
print("Finished")
file.close()

file = open("newfile.txt", "w")
file.write("Some new text")
file.close()

file = open("newfile.txt", "r")
print("Reading new contents")
print(file.read())
print("Finished")
file.close()
```

结果：

```
>>>
Reading initial contents
some initial text
Finished
Reading new contents
Some new text
Finished
>>>
```

如您所见，文件的内容已被覆盖。

✧ 写文件

• 如果成功，**write** 方法返回写入文件的**字节**数。

```
msg = "Hello world!"
file = open("newfile.txt", "w")
amount_written = file.write(msg)
print(amount_written)
file.close()
```

结果：

```
>>>
12
>>>
```

要编写除字符串以外的内容，首先需要将其转换为字符串。

4.9.使用文件

✧ 使用文件

• 最好通过确保文件在使用后始终关闭来避免浪费资源。一种方法是使用 **try** 和 **finally**。

```
try:
    f = open("filename.txt")
    print(f.read())
finally:
    f.close()
```

这可确保即使发生错误，文件也始终处于关闭状态。

✧ 使用文件

• 另一种方法是使用 **with** 语句。这会创建一个临时变量（通常称为 **f**），只能在 with 语句的缩进块中访问。

```
with open("filename.txt") as f:
    print(f.read())
```

该文件在 **with** 语句结束时自动关闭，即使在其中发生异常也是如此。

5.更多类型

5.1.None

✧ None

- **None** 对象用于表示缺少值。
- 它在其他编程语言中类似于 **null**。
- 与其他“空”值（如 0，[]和空字符串）一样，转换为布尔变量时为 **False**。
- 在 Python 控制台中输入时，它将显示为空字符串。

```
>>> None == None
True
>>> None
>>> print(None)
None
>>>
```

✧ None

- **None** 对象由任何未显式返回任何其他内容的函数返回。

```
def some_func():
    print("Hi!")

var = some_func()
print(var)
```

结果：

```
>>>
Hi!
None
>>>
```

5.2.字典

✧ 字典

• **字典**是用于将任意键映射到值的数据结构。
• 列表可以被认为是具有特定范围内的整数键的字典。
• 可以使用包含键的**方括号**，以列表相同的方式对字典编制索引。
例：

```
ages = {"Dave": 24, "Mary": 42, "John": 58}
print(ages["Dave"])
print(ages["Mary"])
```

结果：

```
>>>
24
42
>>>
```

字典中的每个元素都由一个**键：值**对表示。

✧ 字典

• 尝试索引不属于字典的键会返回 **KeyError**。
例：

```
primary = {
    "red": [255, 0, 0],
    "green": [0, 255, 0],
    "blue": [0, 0, 255],
}

print(primary["red"])
print(primary["yellow"])
```

结果：

```
>>>
[255, 0, 0]

KeyError: 'yellow'
>>>
```

• 如您所见，字典可以将任何类型的数据存储为值。
空字典定义为**{}**。

✧ 字典

• 只有**不可变**对象才能用作字典的键。**不可变**对象是那些无法更改的对象。到目前为止，您遇到的唯一可变对象是**列表**和**字典**。尝试将可变对象用作字典键会导致 **TypeError**。

```
bad_dict = {
    [1, 2, 3]: "one two three",
}
```

结果：

```
>>>
TypeError: unhashable type: 'list'
>>>
```

5.3.字典函数

✧ 字典

• 就像列表一样，字典键可以分配给不同的值。
• 但是，与列表不同，新字典键也可以赋值，而不仅仅是已经存在的值

```
squares = {1: 1, 2: 4, 3: "error", 4: 16,}
squares[8] = 64
squares[3] = 9
print(squares)
```

结果

```
{8: 64, 1: 1, 2: 4, 3: 9, 4: 16}
```

✧ 字典

• 要确定某个键是否在字典中，您可以使用 **in** 和 **not in**，就像使用列表一样。
例：

```
nums = {
   1: "one",
   2: "two",
   3: "three",
}
print(1 in nums)
print("three" in nums)
print(4 not in nums)
```

结果：

```
>>>
True
False
True
>>>
```

✧ 字典

• **get** 是一个有用的字典方法。它与索引相同，但如果在字典中找不到该键，则返回另一个指定值（默认情况下为“None”）。

例：

```
pairs = {1: "apple",
   "orange": [2, 3, 4],
   True: False,
   None: "True",
}

print(pairs.get("orange"))
print(pairs.get(7))
print(pairs.get(12345, "not in dictionary"))
```

结果：

```
>>>
[2, 3, 4]
None
not in dictionary
>>>
```

5.4.元组

✧ 元组

• **元组**与列表非常相似，只是它们是不可变的（它们不能被更改）。
此外，它们是使用**括号**而不是方括号创建的。
例：

```
words = ("spam", "eggs", "sausages",)
```

• 您可以使用索引访问元组中的值，就像使用列表一样：

```
print(words[0])
```

• 尝试重新分配元组中的值会导致 TypeError。

```
words[1] = "cheese"
```

结果：

```
>>>
TypeError: 'tuple' object does not support item assignment
>>>
```

像列表和字典一样，元组可以互相嵌套。

✧ 元组

• 只需用逗号分隔值，就可以在没有括号的情况下创建元组。
例：

```
my_tuple = "one", "two", "three"
print(my_tuple[0])
```

结果：

```
>>>
one
>>>
```

• 使用空括号对创建空元组。

```
tql=()
```

元组比列表更快，但它们无法更改。

5.5.列表切片

✧ 列表切片

•**列表切片**提供了一种从列表中检索值的更高级方法。基本列表切片涉及使用**两个冒号分隔的整数**索引列表。这将返回一个新列表，其中包含索引之间旧列表中的所有值。

例：

```
squares = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
print(squares[2:6])
print(squares[3:8])
print(squares[0:1])
```

结果：

```
>>>
[4, 9, 16, 25]
[9, 16, 25, 36, 49]
[0]
>>>
```

与 **range** 的参数一样，切片中提供的第一个索引包含在结果中，但第二个索引不包含在结果中。

✧ 列表切片

- 如果省略切片中的第一个数字，则将其视为列表的开头。
- 如果省略第二个数字，则将其视为结束。

例：

```
squares = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
print(squares[:7])
print(squares[7:])
```

结果：

```
>>>
[0, 1, 4, 9, 16, 25, 36]
[49, 64, 81]
>>>
```

切片也可以在元组上完成。

✧ 列表切片

• 列表切片还可以具有表示步长的第三个数字，以替代切片中唯一的间隔值。

```
squares = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
print(squares[::2])
print(squares[2:8:3])
```

结果

```
>>>
[0, 4, 16, 36, 64]
[4, 25]
>>>
```

[2:8:3]将包含从第 2 个索引到第 8 个索引的元素，步长为 3。

✧ 列表切片

• 可以在列表切片（和正常列表索引）中使用**负值**。当负值用于切片（或普通索引）中的第一个和第二个值时，它们从列表的末尾开始计算。

```
squares = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
print(squares[1:-1])
```

结果：

```
>>>
[1, 4, 9, 16, 25, 36, 49, 64]
>>>
```

如果步长使用负值，则向后完成切片。
使用**[:: - 1]**作为切片是反转列表的常用和惯用方法。

5.6.列表生成式

✧ 列表生成式

• **列表生成式**是快速创建其内容遵循简单规则的列表的有用方式。
例如，我们可以执行以下操作：

```
# a list comprehension
cubes = [i**3 for i in range(5)]

print(cubes)
```

结果：

```
>>>
[0, 1, 8, 27, 64]
>>>
```

列表生成式的灵感来自数学中的集合构建符号。

✧ 列表生成式

• 列表推导还可以包含 **if** 语句，以对列表中的值强制执行条件。
例：

```
evens=[i**2 for i in range(10) if i**2 % 2 == 0]

print(evens)
```

结果：

```
>>>
[0, 4, 16, 36, 64]
>>>
```

✧ 列表生成式

• 尝试在非常广泛的范围内创建列表将导致 **MemoryError**。
• 此代码显示了列表解析耗尽内存的示例。

```
even = [2*i for i in range(10**100)]
```

结果：

```
>>>
MemoryError
>>>
```

这个问题由**生成器**解决，将在下一个模块中介绍。

5.7.字符串格式化

✧ 字符串格式化

• 到目前为止，要组合字符串和非字符串，您已将非字符串转换为字符串并添加它们。
• 字符串格式化提供了一种在字符串中嵌入非字符串的更强大的方法。字符串格式化使用字符串的**格式**方法替换字符串中的多个参数。
例：

```
# string formatting
nums = [4, 5, 6]
msg = "Numbers: {0} {1} {2}". format(nums[0], nums[1], nums[2])
print(msg)
```

结果

```
>>>
Numbers: 4 5 6
>>>
```

format 函数的每个参数都放在相应位置的字符串中，这是使用花括号{}确定的。

✧ 字符串格式化

• 字符串格式也可以使用命名参数完成。
例：

```
a = "{x}, {y}".format(x=5, y=12)
print(a)
```

结果：

```
>>>
5, 12
>>>
```

5.8 有用的函数

✧ 字符串函数

• Python 包含许多有用的内置函数和方法来完成常见任务。
join - 使用另一个字符串作为分隔符连接字符串列表。
replace - 将字符串中的一个子字符串替换为另一个子字符串。
startswith 和 **endswith** - 分别确定字符串的开头和结尾是否有子字符串。
• 要更改字符串的大小写，可以使用 **lower** 和 **upper**。
• 方法 **split** 与 **join** 相反，将具有特定分隔符的字符串转换为列表。
一些例子：

```
print(", ".join(["spam", "eggs", "ham"]))
#prints "spam, eggs, ham"

print("Hello ME".replace("ME", "world"))
#prints "Hello world"

print("This is a sentence.".startswith("This"))
# prints "True"

print("This is a sentence.".endswith("sentence."))
# prints "True"

print("This is a sentence.".upper())
# prints "THIS IS A SENTENCE."

print("AN ALL CAPS SENTENCE".lower())
#prints "an all caps sentence"

print("spam, eggs, ham".split(", "))
#prints "['spam', 'eggs', 'ham']"
```

✧ 数值函数

• 要查找某些数字或列表的最大值或最小值，可以使用 **max** 或 **min**。
• 要从零（其绝对值）中找到数字的距离，请使用 **abs**。
• 要将数字四舍五入到一定数量的小数位，请使用 **round**。
• 要查找列表的总数，请使用 **sum**。
一些例子：

```
print(min(1, 2, 3, 4, 0, 2, 1))
print(max([1, 4, 9, 2, 5, 6, 8]))
print(abs(-99))
print(abs(42))
print(sum([1, 2, 3, 4, 5]))
```

结果：

```
>>>
0
9
99
42
15
>>>
```

✧ 列表函数

• 通常在条件语句中使用，**all** 和 **any** 将列表作为参数，如果所有或任何（分别）的参数都计算为 **True**，则返回 **True**（否则返回 **False**）。
• 函数 **enumerate** 可用于同时迭代列表的值和索引。
例：

```
nums = [55, 44, 33, 22, 11]

if all([i > 5 for i in nums]):
    print("All larger than 5")

if any([i % 2 == 0 for i in nums]):
    print("At least one is even")

for v in enumerate(nums):
    print(v)
```

结果：

```
>>>
All larger than 5
At least one is even
(0, 55)
(1, 44)
(2, 33)
(3, 22)
(4, 11)
>>>
```

5.9.文本分析器

✧ 文本分析器

• 这是一个示例项目，显示了一个程序，该程序分析样本文件以查找每个字符占用的文本的百分比。
• 本节介绍如何打开和读取文件。

```
filename = input("Enter a filename: ")

with open(filename) as f:
    text = f.read()

print(text)
```

Result:

```
>>>
Enter a filename: test.txt
Ornhgvshy vf orggre guna htyl.
Rkcyvpvg vf orggre guna vzcyvpvg.
Fvzcyr vf orggre guna pbzcyvpngrq.
Syng vf orggre guna arfgrq.
Fcenfr fv orggre guna qrafr.
Ernqnovyvgl pbhagf.
Fcrpvny pnfrf nera'g fcrpvny rabthu gb oernx gur ehyrf.
Nygubhtu cenpgvpnyvgl orgnf chevgl.
Reebef fubhyq arire cnff fvyragyl.
Hayrff rkcyvpvgyl fvyraprq.
Va gur snpr bs nzovthvgl, ershfr gur grzcgngvba bg thrff.
Gurer fubhyq or bar-- naq cersrenoylbayl bar --boivbhf jnl gb qb vg.
Nygubhtu gung jnl znl abg or boivbhf ng svefg hayrff lbh'er Qhgpu.
Abj vf orggre guna arrire.
```

```
Nygubhtu arire vf bsgra orggre guna *evtug* abj.
Vs gur vzcyrzragngvba vf uneq gb rkcynva, vg'f n onq vqrn.
Vs gur vzcyrzragngvba vf rnfl gb rkcynva, vg znl or n tbbq vqrn.
Anzrfcnprf ner bar ubaxvat terng vqrn -- yrg'f qb zber bs gubfr!
```

此示例内容仅用于演示目的。

✧ 文本分析器

• 程序的这一部分显示了一个函数，该函数计算字符串中出现字符的次数。

```
def count_char(text, char):
    count = 0
    for c in text:
        if c == char:
            count += 1
    return count
```

•此函数将文件的文本和一个字符作为其参数，返回该字符在文本中出现的次数。
• 现在我们可以为我们的文件调用它。

```
filename = input("Enter a filename: ")
with open(filename) as f:
    text = f.read()

print(count_char(text, "r"))
```

结果：

```
>>>
Enter a filename: test.txt
83
>>>
```

字符“r”在文件中出现 83 次。

✧ 文本分析器

• 程序的下一部分查找字母表中每个字符所占的文本百分比。

```
for char in "abcdefghijklmnopqrstuvwxyz":
    perc = 100 * count_char(text, char) / len(text)
```

```
    print("{0} - {1}%".format(char, round(perc, 2)))
```

让我们把它们放在一起运行程序：

```
def count_char(text, char):
    count = 0
    for c in text:
        if c == char:
            count += 1
    return count

filename = input("Enter a filename: ")
with open(filename) as f:
    text = f.read()

for char in "abcdefghijklmnopqrstuvwxyz":
    perc = 100 * count_char(text, char) / len(text)
    print("{0} - {1}%".format(char, round(perc, 2)))
```

结果：

```
Enter a filename: test.txt
a - 4.68%
b - 4.94%
c - 2.28%
...
```

6.函数式编程

6.1.函数式编程

✧ 函数式编程

• **函数式编程**是一种编程风格（顾名思义）基于函数。
• 函数式编程的关键部分是**高阶函数**。我们在上一节关于函数作为对象的内容中简要地看到了这个想法。高阶函数将其他函数作为参数，或将它们作为结果返回。
例：

```
def apply_twice(func, arg):
    return func(func(arg))

def add_five(x):
    return x + 5

print(apply_twice(add_five, 10))
```

结果：

```
>>>
20
>>>
```

函数 **apply_twice** 将另一个函数作为其参数，并在其内部调用它两次。

✧ 纯函数

• 功能编程旨在使用**纯函数**。纯函数没有副作用，并返回**仅**依赖于其参数的值。
• 这就是数学函数的工作原理：例如，对于相同的 x 值，cos(x)将始终返回相同的结果。
• 以下是纯函数和非函数的示例。

纯函数：

```
def pure_function(x, y):
   temp = x + 2*y
   return temp / (2*x + y)
```

非纯函数：

```
some_list = []

def impure(arg):
    some_list.append(arg)
```

上面的函数不是纯粹的，因为它改变了 **some_list** 的状态。

✧ 纯函数

• 使用纯函数既有优点也有缺点。
• 纯功能是：
- 更容易推理和测试。
- 更高效。一旦为输入评估了函数，就可以存储结果并在下次需要该输入的函数时引用该结果，从而减少调用函数的次数。这称为 **memoization**。
- 更容易并行运行。

仅使用纯函数的主要缺点是它们使 I / O 的其他简单任务变得非常复杂，因为这似乎本身就需要副作用。
在某些情况下，它们也可能更难写。

6.2.Lambda 表达式

✧ Lambdas

• 正常创建函数（使用 **def**）会自动将其赋值给变量。
• 这与创建其他对象（例如字符串和整数）不同，后者可以动态创建，而无需将它们分配给变量。
• 只要使用 **lambda** 语法创建函数，函数也可以这样做。以这种方式创建的函数称为**匿名**。
• 将简单函数作为参数传递给另一个函数时，最常使用此方法。语法显示在下一个示例中，由 lambda 关键字后跟参数列表，冒号以及要计算和返回的表达式组成。

```
def my_func(f, arg):
    return f(arg)

my_func(lambda x: 2*x*x, 5)
```

Lambda 函数的名字来自 **lambda 演算**，这是 Alonzo Church 发明的计算模型。

✧ Lambdas

• Lambda 函数没有命名函数那么强大。
• 他们只能做需要单个表达式的事情 - 通常相当于一行代码。
例：

```
#named function
def polynomial(x):
    return x**2 + 5*x + 4
print(polynomial(-4))

#lambda
print((lambda x: x**2 + 5*x + 4) (-4))
```

结果：

```
>>>
0
0
>>>
```

在上面的代码中，我们动态创建了一个匿名函数，并使用参数调用它。

✧ Lambdas

• Lambda 函数可以分配给变量，并像普通函数一样使用。
例：

```
double = lambda x: x * 2
print(double(7))
```

结果：

```
>>>
14
>>>
```

但是，很少有充分的理由这样做 - 通常使用 **def** 来定义函数通常更好。

6.3.map 和 filter

✧ map

• 内置函数 **map** 和 **filter** 是非常有用的高阶函数，它们对列表（或称为 **iterables** 的类似对象）进行操作。
• 函数 **map** 将函数和 iterable 作为参数，并返回一个新的 iterable，函数应用于每个参数。
例：

```
def add_five(x):
    return x + 5

nums = [11, 22, 33, 44, 55]
result = list(map(add_five, nums))
print(result)
```

结果：

```
>>>
[16, 27, 38, 49, 60]
>>>
```

• 通过使用 **lambda** 语法，我们可以更轻松地实现相同的结果。

```
nums = [11, 22, 33, 44, 55]

result = list(map(lambda x: x+5, nums))
print(result)
```

要将结果转换为列表，我们明确使用了**列表**。

✧ filter

函数 **filter** 通过删除与谓词不匹配的项（返回布尔值的函数）来过滤 iterable。
例：

```
nums = [11, 22, 33, 44, 55]
res = list(filter(lambda x: x%2==0, nums))
print(res)
```

结果：

```
>>>
[22, 44]
>>>
```

与 **map** 一样，如果要打印结果，则必须将结果显式转换为列表。

6.4.生成器

✧ 生成器

• **生成器**是一种可迭代的类型，类似列表或元组。
• 与列表不同，它们不允许使用任意索引进行索引，但它们仍然可以使用 **for** 循环进行迭代。
• 可以使用函数和 **yield** 语句创建它们。
例：

```
def countdown():
    i=5
    while i > 0:
        yield i
        i -= 1

for i in countdown():
    print(i)
```

结果：

```
>>>
5
4
3
2
1
```

yield 语句用于定义生成器，替换函数的返回值以向调用者提供结果而不破坏局部变量。

✧ 生成器

• 由于它们一次**产生**一个元素，因此生成器没有列表的内存限制。
• 事实上，它们可以是**无限**的！

```
def infinite_sevens():
    while True:
        yield 7

for i in infinite_sevens():
    print(i)
```

结果：

```
>>>
7
7
7
7
7
7
7
...
```

简而言之，**生成器**允许您声明一个行为类似于迭代器的函数，即它可以在 **for** 循环中使用。

✧ 生成器

• 有限生成器可以通过将它们作为参数传递给 **list** 函数来转换为列表。

```
def numbers(x):
    for i in range(x):
        if i % 2 == 0:
            yield i

print(list(numbers(11)))
```

结果：

```
>>>
[0, 2, 4, 6, 8, 10]
>>>
```

使用**生成器**可以提高性能，这是延迟（按需）生成值的结果，这意味着更低的内存使用率。此外，在开始使用它们之前，我们不需要等到所有元素都已生成。

6.5.包装器

✧ 包装器

• **装饰器**提供了一种使用其他函数修改函数的方法。
• 当您需要扩展不想修改的函数的功能时，这是理想的选择。
例：

```
def decor(func):
    def wrap():
        print("============")
        func()
        print("============")
    return wrap

def print_text():
    print("Hello world!")

decorated = decor(print_text)
decorated()
```

• 我们定义了一个名为 **decor** 的函数，它具有单个参数 **func**。在内部装饰中，我们定义了一个名为 **wrap** 的嵌套函数。**wrap** 函数将打印一个字符串，然后调用 **func()**，并打印另一个字符串。**decor** 函数返回 **wrap** 函数作为结果。
• 我们可以说**包装的**变量是 **print_text** 的装饰版本 - 它是 **print_text** 加上一些东西。
• 事实上，如果我们写了一个有用的装饰器，我们可能想要将 **print_text** 替换为装饰版本，所以我们总是得到 **print_text** 的“附加后”版本。
• 这是通过重新分配包含我们的函数的变量来完成的：

```
print_text = decor(print_text)
print_text()
```

现在 **print_text** 对应于我们的装饰版本。现在 print_text 对应于我们的装饰版本。

✧ 包装器

• 在前面的示例中，我们通过将包含函数的变量替换为包装版本来修饰函数。

```
def print_text():
    print("Hello world!")
```

```
print_text = decor(print_text)
```

• 这种模式可以随时使用，以包装任何功能。
• Python 通过使用装饰器名称和@符号预先挂起函数定义，提供了在装饰器中包装函数的支持。
• 如果我们定义一个函数，我们可以用@符号“包装”它，如：

```
@decor
def print_text():
    print("Hello world!")
```

• 这与上面的代码具有相同的结果。
单个函数可以有多个装饰器。

6.6.递归

✧ 递归

• **递归**是函数式编程中非常重要的概念。
• 递归的基本部分是自引用 - 函数调用自身。它用于解决可以分解为相同类型的更容易的子问题的问题。

• 递归实现的函数的典型示例是**阶乘**函数，其发现低于指定数字的所有正整数的乘积。
• 例如，5!(5 阶乘)是 5 * 4 * 3 * 2 * 1(120)。要递归执行此操作，请注意 5!= 5 * 4!，4! = 4 * 3!，3! = 3 * 2!等等。一般来说，n!= n *(n-1)!
• 此外，1！这被称为**基本情况**，因为它可以在不执行任何因子的情况下进行计算。
• 下面是阶乘函数的递归实现。

```
def factorial(x):
    if x == 1:
        return 1
    else:
        return x * factorial(x-1)

print(factorial(5))
```

结果：

```
>>>
120
>>>
```

基本情况充当递归的退出条件。

✧ 递归

•递归函数可以是无限的，就像无限 **while** 循环一样。当您忘记实施基本案例时，通常会发生这种情况。
•以下是阶乘函数的错误版本。它没有基本情况，因此一直运行直到解释器耗尽内存并崩溃。

```
def factorial(x):
    return x * factorial(x-1)

print(factorial(5))
```

结果：

```
>>>
RuntimeError: maximum recursion depth exceeded
>>>
```

✧ 递归

•递归也可以是间接的。一个函数可以调用第二个函数，它调用第一个函数，调用第二个函数，依此类推。任何数量的功能都可能发生这种情况。
例：

```
def is_even(x):
    if x == 0:
        return True
    else:
        return is_odd(x-1)

def is_odd(x):
    return not is_even(x)


print(is_odd(17))
print(is_even(23))
```

结果：

```
>>>
True
False
>>>
```

6.7.集合

✧ 集合

• **集合**是数据结构，类似于列表或字典。它们是使用花括号或 **set** 函数创建的。它们与列表共享一些功能，例如使用 **in** 来检查它们是否包含特定项目。

```
num_set = {1, 2, 3, 4, 5}
word_set = set(["spam", "eggs", "sausage"])

print(3 in num_set)
print("spam" not in word_set)
```

结果：

```
>>>
True
False
>>>
```

要创建空集，必须使用 **set()**，因为**{}**创建一个空字典。

✧ 集合

• 集合在几个方面与列表不同，但共享多个列表操作，如 **len**。
• 它们是无序的，这意味着它们无法编入索引。
• 它们**不能**包含重复元素。
•由于它们的存储方式，检查项目是否是集合的一部分而不是列表的一部分**更快**。
• 不使用 **append** 添加到集合，而是使用 **add**。
• **remove** 方法从集合中删除特定元素; **pop** 删除任意元素。

```
nums = {1, 2, 1, 3, 1, 4, 5, 6}
print(nums)
nums.add(-7)
```

```
nums.remove(3)
print(nums)
```

结果：

```
>>>
{1, 2, 3, 4, 5, 6}
{1, 2, 4, 5, 6, -7}
>>>
```

集合的基本用途包括成员关系测试和消除重复条目。

✧ 集合

- 可以使用数学运算来组合集合。
- **并集**运算符 **|** 将两个集合组合成一个包含其中任何一个项目的新集合。
- **交集**运算符 **&** 仅在两者中获取项目。
- **差异**运算符 **-** 获取第一组中的项目但不获取第二组中的项目。
- **对称差分**运算符 **^** 获取任一集合中的项目，但不能同时获取两者。

```
first = {1, 2, 3, 4, 5, 6}
second = {4, 5, 6, 7, 8, 9}

print(first | second)
print(first & second)
print(first - second)
print(second - first)
print(first ^ second)
```

结果：

```
>>>
{1, 2, 3, 4, 5, 6, 7, 8, 9}
{4, 5, 6}
{1, 2, 3}
{8, 9, 7}
{1, 2, 3, 7, 8, 9}
>>>
```

✧ 数据结构

- 正如我们在前面的课程中看到的，Python 支持以下数据结构：**列表，字典，元组，集合**。

• 何时使用字典：
- 当您需要**键：值**对之间的逻辑关联时。
- 当您需要基于自定义密钥快速查找数据时。
- 不断修改数据时。请记住，词典是可变的。

• 何时使用其他类型：
- 如果您有一组不需要随机访问的数据，请使用**列表**。当您需要经常修改的简单可迭代集合时，尝试选择列表。
- 如果您需要元素的唯一性，请使用**集合**。
- 当数据无法更改时使用**元组**。

很多时候，**元组**与**字典**结合使用，例如，**元组**可能代表一个键，因为它是不可变的

6.8.itertools 模块

✧ itertools

• 模块 **itertools** 是一个标准库，包含几个在函数式编程中有用的函数。
• 它产生的一种函数是无限迭代器。
• 函数 **count** 从一个值无限计数。
• 函数 **cycle** 无限迭代迭代（例如列表或字符串）。
• 函数 **repeat** 重复一个对象，无限次或特定次数。

例：

```
from itertools import count

for i in count(3):
    print(i)
    if i >=11:
break
```

结果：

```
>>>
3
4
5
6
7
8
9
10
```

```
11
>>>
```

✧ itertools

• **itertools** 中有许多函数以迭代形式运行，与 **map** 和 **filter** 类似。
一些例子：
takewhile - 在谓词函数保持为真时从可迭代中获取项目;
chain - 将几个迭代组合成一个长的;
accumulate - 返回可迭代中的运行总值。

```
from itertools import accumulate, takewhile

nums = list(accumulate(range(8)))
print(nums)
print(list(takewhile(lambda x: x<= 6, nums)))
```

结果：

```
>>>
[0, 1, 3, 6, 10, 15, 21, 28]
[0, 1, 3, 6]
>>>
```

✧ itertools

• 在 **itertool** 中还有几个组合函数，例如 **product** 和 **permutation**。
• 当您想要完成具有某些项目的所有可能组合的任务时，可以使用这些。
例：

```
from itertools import product, permutations

letters = ("A", "B")
print(list(product(letters, range(2))))
print(list(permutations(letters)))
```

结果：

```
>>>
[('A', 0), ('A', 1), ('B', 0), ('B', 1)]
[('A', 'B'), ('B', 'A')]
>>>
```

7.面向对象编程

7.1.类

✧ 类

• 我们之前已经看过两种编程范式 - **命令式**（使用语句，循环和函数作为子例程）和**函数式**（使用纯函数，高阶函数和递归）。

• 另一个非常流行的范式是**面向对象编程**（OOP）。
• 使用**类**创建对象，这实际上是 OOP 的焦点。
• 该**类**描述了对象的内容，但与对象本身是分开的。换句话说，类可以被描述为对象的蓝图，描述或定义。
• 您可以使用相同的类作为蓝图来创建多个不同的对象。

• 使用关键字 **class** 和缩进块创建类，其中包含类**方法**（函数）。
• 下面是一个简单类及其对象的示例。

```
class Cat:
    def __init__(self, color, legs):
        self.color = color
        self.legs = legs

felix = Cat("ginger", 4)
rover = Cat("dog-colored", 4)
stumpy = Cat("brown", 3)
```

此代码定义了一个名为 **Cat** 的类，它有两个属性：**color** 和 **legs**。
然后该类用于创建该类的 3 个单独对象。

✧ __init__

• **__init__**方法是类中最重要的方法。
• 当使用类名作为函数创建类的实例（对象）时，将调用此方法。

• 所有方法都必须将 **self** 作为其第一个参数，尽管未明确传递，Python 会将 **self** 参数添加到列表中；调用方法时不需要包含它。在方法定义中，**self** 指的是调用该方法的实例。

• 类的实例具有**属性**，这些属性是与它们相关联的数据。
• 在此示例中，**Cat** 实例具有属性**颜色**和**腿**。可以通过在实例后面添加一个**点**和属性名来访问它们。
• 因此，在**__init__**方法中，**self.attribute** 可用于设置实例属性的初始值。
例：

```
class Cat:
    def __init__(self, color, legs):
        self.color = color
        self.legs = legs

felix = Cat("ginger", 4)
print(felix.color)
```

结果：

```
>>>
ginger
>>>
```

在上面的示例中，**__ init__**方法接受两个参数并将它们分配给对象的属性。
__init__方法称为类**构造函数**。

✧ 方法

• 方法类可以定义其他**方法**来为它们添加功能。请记住，所有方法都必须将 **self** 作为其第一个参数。使用与属性相同的**点**语法访问这些方法。
例：

```
class Dog:
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def bark(self):
        print("Woof!")

fido = Dog("Fido", "brown")
print(fido.name)
fido.bark()
```

Result:

```
>>>
Fido
```

```
Woof!
>>>
```

• 类也可以具有**类属性**，通过在类的主体内分配变量来创建。这些可以从类的实例或类本身访问。

例：

```
class Dog:
    legs = 4
    def __init__(self, name, color):
        self.name = name
        self.color = color

fido = Dog("Fido", "brown")
print(fido.legs)
print(Dog.legs)
```

Result:

```
>>>
4
4
>>>
```

类属性由类的所有实例共享。

✧ 类

• 类尝试访问未定义的实例的属性会导致 **AttributeError**。当您调用未定义的方法时，这也适用。

例：

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

rect = Rectangle(7, 8)
print(rect.color)
```

Result:

```
>>>
AttributeError: 'Rectangle' object has no attribute 'color'
>>>
```

7.2.继承

✧ 继承

• **继承**提供了一种在类之间共享功能的方法。
• 想象一下，**Cat**，**Dog**，**Rabbit** 等几个类。 虽然它们在某些方面可能有所不同（只有 **Dog** 可能有方法 **bark**），但它们在其他方面可能相似（都具有属性**颜色**和**名称**）。
• 这种相似性可以通过使它们全部继承包含共享功能的**父类 Animal** 来表达。
• 要从另一个类继承一个类，请将父类名称放在类名后面的括号中。例：

```
class Animal:
  def __init__(self, name, color):
    self.name = name
    self.color = color

class Cat(Animal):
  def purr(self):
    print("Purr...")

class Dog(Animal):
  def bark(self):
    print("Woof!")

fido = Dog("Fido", "brown")
print(fido.color)
fido.bark()
```

Result:

```
>>>
brown
Woof!
>>>
```

✧ 继承

• 从另一个类继承的类称为**子类**。
• 被继承的类称为**父类**。
• 如果一个类从具有相同属性或方法的另一个类继承，则它将覆盖它们。

```
class Wolf:
```

```
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def bark(self):
        print("Grr...")

class Dog(Wolf):
    def bark(self):
        print("Woof")

husky = Dog("Max", "grey")
husky.bark()
```

Result:

```
>>>
Woof
>>>
```

在上面的例子中，**Wolf** 是父类，**Dog** 是子类。

✧ 继承

•继承也可以是间接的。一个类可以从另一个类继承，该类可以从第三个类继承。

例：

```
class A:
    def method(self):
        print("A method")

class B(A):
    def another_method(self):
        print("B method")

class C(B):
    def third_method(self):
        print("C method")

c = C()
c.method()
c.another_method()
c.third_method()
```

Result:

```
>>>
A method
B method
C method
>>>
```

但是，循环继承是不可能的。

✧ 继承

• 函数 **super** 是一个有用的继承相关函数，它引用父类。它可用于在对象的父类中查找具有特定名称的方法。

例：

```
class A:
    def spam(self):
        print(1)

class B(A):
    def spam(self):
        print(2)
        super().spam()

B().spam()
```

Result:

```
>>>
2
1
>>>
```

super().spam()调用父类的 **spam** 方法。

7.3.魔术方法和操作符重载

✧ 魔术方法

• **魔术方法**是特殊方法，在名称的开头和结尾都有**双重下划线**。

• 它们也被称为 **dunders**。
• 到目前为止，我们遇到的唯一一个是**__init__**，但还有其他几个。
• 它们用于创建无法表示为普通方法的功能。

• 它们的一个常见用途是**运算符重载**。
• 这意味着为自定义类定义运算符，允许在其上使用+和*等运算符。
• 一个示例魔术方法是**__add__** for +。

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Vector2D(self.x + other.x, self.y + other.y)

first = Vector2D(5, 7)
second = Vector2D(3, 9)
result = first + second
print(result.x)
print(result.y)
```

结果：

```
>>>
8
16
>>>
```

__add__方法允许为我们类中的+运算符定义自定义行为。
如您所见，它添加了对象的相应属性并返回包含结果的新对象。
一旦定义，我们可以将类的两个对象一起添加。

✧ 魔术方法

• 普通运算符的更多魔术方法：
__sub__ 为 -
__mul__ 为 *
__truediv__ 为 /
__floordiv__ 为 //
__mod__ 为 %
__pow__ 为 **
__and__ 为 &
__xor__ 为 ^
__or__ 为 |

• 表达式 **x + y** 被转换为 **x .__ add __(y)**。
•但是，如果 x 未实现__add__，并且 x 和 y 的类型不同，则调用 **y .__ radd __(x)**。
• 对于刚刚提到的所有魔术方法，都有等价的方法。
例：

```
class SpecialString:
    def __init__(self, cont):
        self.cont = cont

    def __truediv__(self, other):
        line = "=" * len(other.cont)
        return "\n".join([self.cont, line, other.cont])

spam = SpecialString("spam")
hello = SpecialString("Hello world!")
print(spam / hello)
```

结果：

```
>>>
spam
============
Hello world!
>>>
```

在上面的例子中，我们为我们的 **SpecialString** 类定义了**除法运算**。

✧ 魔术方法

• Python 还提供了用于比较的魔术方法。
__lt__ 为<
__le__ 为 <=
__eq__ 为 ==
__ne__ 为 !=
__gt__ 为 >
__ge__ 表示 >=

• 如果未实现**__ne__**，则返回**__eq__**的反向。
• 其他运算符之间没有其他关系。
例：

```
class SpecialString:
    def __init__(self, cont):
```

```
        self.cont = cont

    def __gt__(self, other):
        for index in range(len(other.cont)+1):
            result = other.cont[:index] + ">" + self.cont
            result += ">" + other.cont[index:]
            print(result)

spam = SpecialString("spam")
eggs = SpecialString("eggs")
spam > eggs
```

结果：

```
>>>
>spam>eggs
e>spam>ggs
eg>spam>gs
egg>spam>s
eggs>spam>
>>>
```

如您所见，您可以为重载的运算符定义任何自定义行为。

✧ 魔术方法

• 有几种使类像容器一样的神奇方法。
__len__ 为 len()
__getitem__ 用于索引
__setitem__ 用于分配索引值
__delitem__ 用于删除索引值
__iter__ 用于对象的迭代（例如，在 for 循环中）
__contains__ 为 in

• 还有许多其他魔术方法，我们不会在这里讨论，例如**__call__**用于将对象作为函数调用，而**__int __**，**__str__**，等等，用于将对象转换为内置类型。
例：

```
import random

class VagueList:
    def __init__(self, cont):
        self.cont = cont
```

```
    def __getitem__(self, index):
        return self.cont[index + random.randint(-1, 1)]

    def __len__(self):
        return random.randint(0, len(self.cont)*2)

vague_list = VagueList(["A", "B", "C", "D", "E"])
print(len(vague_list))
print(len(vague_list))
print(vague_list[2])
print(vague_list[2])
```

Result:

```
>>>
6
7
D
C
>>>
```

我们重写了类 VagueList 的 len()函数以返回一个随机数。
索引函数还根据表达式返回列表范围内的随机项。

7.4.对象生命周期

✧ 对象生命周期

• 对象的生命周期由其**创建**，**操作**和**销毁**组成。

• 对象生命周期的第一个阶段是它所属的类的**定义**。
• 当调用**__init__**时，下一个阶段是实例的**实例化**。分配内存来存储实例。就在此之前，调用类的**__new__**方法。这通常仅在特殊情况下被覆盖。
• 发生这种情况后，该对象就可以使用了。

然后，其他代码可以通过调用其上的函数并访问其属性来与对象进行交互。
最终，它将被完成使用，并可以被**销毁**。

✧ 对象生命周期

• 当一个对象被**销毁**时，分配给它的内存被释放，并可用于其他目的。

• 当对象的**引用计数**达到零时，就会发生对象的破坏。引用计数是引用对象的变量和其他元素的数量。
• 如果没有任何东西引用它（它的引用计数为零）没有任何东西可以与它交互，那么可以安全地删除它。

• 在某些情况下，两个（或更多）对象只能彼此引用，因此也可以删除。
• **del** 语句将对象的引用计数减少一个，这通常会导致删除它。
• **del** 语句的魔术方法是**__del__**。
• 删除不再需要的对象的过程称为**垃圾回收**。
• 总之，对象的引用计数在分配新名称或放在容器（列表，元组或字典）时会增加。当删除 **del** 时，对象的引用计数减少，其引用被重新分配，或者引用超出范围。当对象的引用计数达到零时，Python 会自动删除它。
例：

```
a = 42 # Create object <42>
b = a # Increase ref. count of <42>
c = [a] # Increase ref. count of <42>

del a # Decrease ref. count of <42>
b = 100 # Decrease ref. count of <42>
c[0] = -1 # Decrease ref. count of <42>
```

像 C 这样的低级语言没有这种自动内存管理。

7.5.数据隐藏

✧ 数据隐藏

•面向对象编程的一个关键部分是**封装**，它涉及将相关变量和函数打包到一个易于使用的对象中 - 一个类的实例。
•一个相关的概念是**数据隐藏**，它声明应隐藏类的实现细节，并为想要使用该类的人提供干净的标准接口。
•在其他编程语言中，这通常使用私有方法和属性来完成，这些方法和属性阻止对类中某些方法和属性的外部访问。

• Python 哲学略有不同。它通常被称为“**我们都认可这里的成年人**”，这意味着你不应该对访问某个类的部分进行任意限制。因此，没有办法强制执行方法或属性严格私密。

但是，有一些方法可以阻止人们访问类的某些部分，例如通过表示它是一个实现细节，并且应该自担风险使用。

✧ 数据隐藏

• 弱私有方法和属性在开头有一个**下划线**。
• 这表示它们是私有的，不应被外部代码使用。但是，它主要只是一个约定，并不会阻止外部代码访问它们。
•它唯一的实际效果是来自 **module_name import *** 不会导入以单个下划线开头的变量。
例：

```
class Queue:
    def __init__(self, contents):
        self._hiddenlist = list(contents)

    def push(self, value):
        self._hiddenlist.insert(0, value)

    def pop(self):
        return self._hiddenlist.pop(-1)

    def __repr__(self):
        return "Queue({})".format(self._hiddenlist)

queue = Queue([1, 2, 3])
print(queue)
queue.push(0)
print(queue)
queue.pop()
print(queue)
print(queue._hiddenlist)
```

Result:

```
>>>
Queue([1, 2, 3])
Queue([0, 1, 2, 3])
Queue([0, 1, 2])
[0, 1, 2]
>>>
```

在上面的代码中，属性**_hiddenlist** 被标记为私有，但仍然可以在外部代码中访问它。
__repr__魔术方法用于实例的字符串表示。

✧ 数据隐藏

•强大的私有方法和属性在其名称的开头有一个**双下划线**。这会导致它们的名称被破坏，这意味着它们无法从类外部访问。
•这样做的目的不是确保它们保持私有，而是为了避免在存在具有相同名称的方法或属性的子类时出现错误。
• 名称损坏的方法仍然可以从外部访问，但使用不同的名称。可以使用 **_Spam__private** 方法在外部访问 **Spam** 类的方法**__private** 方法。
例：

```
class Spam:
    __egg = 7
    def print_egg(self):
        print(self.__egg)

s = Spam()
s.print_egg()
print(s._Spam__egg)
print(s.__egg)
```

Result:

```
>>>
7
7
AttributeError: 'Spam' object has no attribute '__egg'
>>>
```

基本上，Python 通过内部更改名称来保护这些成员以包含类名。

7.6.类和静态方法

✧ 类方法

• 到目前为止我们查看的对象方法由类的实例调用，然后接受方法的 **self** 参数。
• **类方法**是不同的 - 它们由类调用，它接受方法的 **cls** 参数。
•这些的常见用途是工厂方法，它使用跟通常传递给类构造函数的参数不同的参数来实例化。
• 类方法用 **classmethod 装饰器**标记。
例：

```
class Rectangle:
```

```
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height

    @classmethod
    def new_square(cls, side_length):
        return cls(side_length, side_length)

square = Rectangle.new_square(5)
print(square.calculate_area())
```

Result:

```
>>>
25
>>>
```

• **new_square** 是一个类方法，在类上调用，而不是在类的实例上调用。它返回类 **cls** 的新对象。
从技术上讲，参数 **self** 和 **cls** 只是约定；他们可以改成其他任何东西。然而，它们是普遍遵循的，所以坚持使用它们是明智的。

✧ 静态方法

• **静态方法**类似于类方法，除了它们不接收任何其他参数；它们与属于类的普通函数相同。
• 它们用 **staticmethod** 装饰器标记。
例：

```
class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings

    @staticmethod
    def validate_topping(topping):
        if topping == "pineapple":
            raise ValueError("No pineapples!")
        else:
            return True

ingredients = ["cheese", "onions", "spam"]
```

```
if all(Pizza.validate_topping(i) for i in ingredients):
    pizza = Pizza(ingredients)
```

静态方法的行为类似于普通函数，除了您可以从类的实例调用它们。

7.7.属性

✧ 属性

• **属性**提供了一种自定义实例属性访问的方法。
• 它们是通过将**属性**装饰器放在方法上方来创建的，这意味着当访问与方法同名的实例属性时，将调用该方法。
• 属性的一个常见用途是将属性设置为**只读**。
例：

```
class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings

    @property
    def pineapple_allowed(self):
        return False

pizza = Pizza(["cheese", "tomato"])
print(pizza.pineapple_allowed)
pizza.pineapple_allowed = True
```

Result:

```
>>>
False

AttributeError: can't set attribute
>>>
```

✧ 属性

• 属性也可以通过定义 **setter / getter** 函数来设置属性。
• **setter** 函数设置相应属性的值。
• **getter** 获取值。
• 要定义 **setter**，需要使用与属性同名的装饰器，后跟一个点和 **setter** 关键字。

• 这同样适用于定义 **getter** 函数。
例：

```
class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings
        self._pineapple_allowed = False

    @property
    def pineapple_allowed(self):
        return self._pineapple_allowed

    @pineapple_allowed.setter
    def pineapple_allowed(self, value):
        if value:
            password = input("Enter the password: ")
            if password == "Sw0rdf1sh!":
                self._pineapple_allowed = value
            else:
                raise ValueError("Alert! Intruder!")

pizza = Pizza(["cheese", "tomato"])
print(pizza.pineapple_allowed)
pizza.pineapple_allowed = True
print(pizza.pineapple_allowed)
```

Result:

```
>>>
False
Enter the password: Sw0rdf1sh!
True
```

7.8.一个简单的游戏

✧ 一个简单的游戏

• 在管理不同对象及其关系时，面向对象非常有用。当您开发具有不同角色和功能的游戏时，这尤其有用。

• 让我们看一个示例项目，该项目显示如何在游戏开发中使用类。
要开发的游戏是一种老式的基于文本的冒险游戏。
• 下面是函数处理输入和简单解析。

```
def get_input():
    command = input(": ").split()
    verb_word = command[0]
    if verb_word in verb_dict:
        verb = verb_dict[verb_word]
    else:
        print("Unknown verb {}". format(verb_word))
        return

    if len(command) >= 2:
        noun_word = command[1]
        print (verb(noun_word))
    else:
        print(verb("nothing"))

def say(noun):
    return 'You said "{}"'.format(noun)

verb_dict = {
    "say": say,
}

while True:
    get_input()
```

Result:

```
>>>
: say Hello!
You said "Hello!"
: say Goodbye!
You said "Goodbye!"

: test
Unknown verb test
```

上面的代码从用户获取输入，并尝试将第一个单词与 **verb_dict** 中的命令匹配。如果找到匹配项，则调用相应的函数。

✧ 一个简单的游戏

- 下一步是使用类来表示游戏对象。

```
class GameObject:
    class_name = ""
    desc = ""
    objects = {}

    def __init__(self, name):
        self.name = name
        GameObject.objects[self.class_name] = self

    def get_desc(self):
        return self.class_name + "\n" + self.desc

class Goblin(GameObject):
    class_name = "goblin"
    desc = "A foul creature"

goblin = Goblin("Gobbly")

def examine(noun):
    if noun in GameObject.objects:
        return GameObject.objects[noun].get_desc()
    else:
        return "There is no {} here.".format(noun)
```

- 我们创建了一个 **Goblin** 类，它继承自 **GameObjects** 类。
- 我们还创建了一个新的函数 **examine**，它返回对象描述。
- 现在我们可以在我们的字典中添加一个新的“检查”动词并试一试！

```
verb_dict = {
"say": say,
"examine": examine,
}
```

- 将此代码与前一个示例中的代码组合，然后运行该程序。

```
>>>
: say Hello!
You said "Hello!"

: examine goblin
goblin
A foul creature

: examine elf
```

```
There is no elf here.
:
```

将此代码与前一个示例中的代码组合，然后运行该程序。

✧ 一个简单的游戏

• 此代码为 **Goblin** 类添加了更多细节，并允许您与地精**战斗**。

```
class Goblin(GameObject):
    def __init__(self, name):
        self.class_name = "goblin"
        self.health = 3
        self._desc = " A foul creature"
        super().__init__(name)

    @property
    def desc(self):
        if self.health >=3:
            return self._desc
        elif self.health == 2:
            health_line = "It has a wound on its knee."
        elif self.health == 1:
            health_line = "Its left arm has been cut off!"
        elif self.health <= 0:
            health_line = "It is dead."
        return self._desc + "\n" + health_line

    @desc.setter
    def desc(self, value):
        self._desc = value

def hit(noun):
    if noun in GameObject.objects:
        thing = GameObject.objects[noun]
        if type(thing) == Goblin:
            thing.health = thing.health - 1
            if thing.health <= 0:
                msg = "You killed the goblin!"
            else:
                msg = "You hit the {}".format(thing.class_name)
    else:
        msg ="There is no {} here.".format(noun)
```

```
  return msg
```

结果：

```
>>>
: hit goblin
You hit the goblin

: examine goblin
goblin
  A foul creature
It has a wound on its knee.

: hit goblin
You hit the goblin

: hit goblin
You killed the goblin!

: examine goblin
A goblin

goblin
  A foul creature
It is dead.
:
```

这只是一个简单的例子。
你可以创建不同的类（例如精灵、兽人、人类），对抗它们，使它们相互战斗，等等。

8.正则表达式

8.1.正则表达式

✧ 正则表达式

• **正则表达式**是各种字符串操作的强大工具。
• 它们是一种领域特定语言（DSL），在大多数现代编程语言中作为库存在，而不仅仅是 Python。
• 它们对两个主要任务很有用：
- 验证字符串是否与**模式**匹配（例如，字符串具有电子邮件地址的格式），
- 在字符串中执行替换（例如将所有美国拼写更改为英国拼写）。

领域特定语言是高度专业化的迷你编程语言。
正则表达式是一个流行的例子，SQL（用于数据库操作）是另一个。
专用域专用语言通常用于特定的工业目的。

✧ 正则表达式

• 可以使用 **re** 模块访问 Python 中的正则表达式，re 模块是标准库的一部分。
• 在定义了正则表达式之后，可以使用 **re.match** 函数来确定它是否在字符串的**开头**匹配。
• 如果是，则 **match** 返回表示匹配的对象，否则返回 **None**。
• 为了避免在使用正则表达式时出现任何混淆，我们将原始字符串用作 **r"expression"**。
• 原始字符串不会转义任何内容，这使得使用正则表达式更容易。
例：

```
import re

pattern = r"spam"

if re.match(pattern, "spamspamspam"):
    print("Match")
else:
    print("No match")
```

Result:

```
>>>
Match
>>>
```

• 上面的示例检查模式“spam”是否与字符串匹配，如果匹配则打印“匹配”。这里的模式是一个简单的单词，但是有各种各样的字符，当它们用在正则表达式中时会有特殊的意义。

✧ 正则表达式

• 匹配模式的其他函数是 **re.search** 和 **re.findall**。
• 函数 **re.search** 在字符串中的任何位置找到模式的匹配项。
• 函数 **re.findall** 返回与模式匹配的所有子字符串的列表。

例：

```
import re

pattern = r"spam"

if re.match(pattern, "eggspamsausagespam"):
    print("Match")
else:
    print("No match")

if re.search(pattern, "eggspamsausagespam"):
    print("Match")
else:
    print("No match")

print(re.findall(pattern, "eggspamsausagespam"))
```

结果：

```
>>>
No match
Match
['spam', 'spam']
>>>
```

• 在上面的示例中，**match** 函数与模式不匹配，因为它查看字符串的开头。
• **search** 功能在字符串中找到匹配项。
函数 **re.finditer** 与 **re.findall** 做同样的事情，除了它返回迭代器而不是列表。

✧ 正则表达式

• 正则表达式搜索返回一个对象，其中包含几个提供有关它的详细信息的方法。
• 这些方法包括返回匹配的字符串的 **group**，返回第一个匹配的开始和结束位置的 **start** 和 **end**，以及将第一个匹配的开始和结束位置作为元组返回的 **span**。
例：

```
import re

pattern = r"pam"

match = re.search(pattern, "eggspamsausage")
if match:
    print(match.group())
    print(match.start())
    print(match.end())
    print(match.span())
```

结果：

```
>>>
pam
4
7
(4, 7)
>>>
```

✧ 搜索和替换

• 使用正则表达式的最重要的 re 方法之一是 **sub**。
语法：

```
re.sub(pattern, repl, string, max=0)
```

• 除非提供 **max**，否则此方法将把出现在字符串中的所有**模式**替换为 **repl**。此方法返回修改后的字符串。
例：

```
import re

str = "My name is David. Hi David."
pattern = r"David"
newstr = re.sub(pattern, "Amy", str)
print(newstr)
```

结果：

```
>>>
My name is Amy. Hi Amy.
>>>
```

8.2.简单的元字符

✧ 元字符

• **元字符**使正则表达式比普通字符串方法更强大。
• 它们允许您创建正则表达式来表示“一个或多个元音重复”等概念。

• 如果要创建与文字元字符匹配的正则表达式（或 **regex**），例如“$”，则元字符的存在会产生问题。您可以通过在元数据前面放一个**反斜杠**来转义元字符。
• 但是，这可能会导致问题，因为反斜杠在普通的 Python 字符串中也有一个转义函数。这可能意味着连续放入三个或四个反斜杠来完成所有的转义。
为避免这种情况，您可以使用原始字符串，它是一个普通字符串，前面带有“r”。我们在上一课中看到了原始字符串的使用。

✧ 元字符

我们将看到的第一个元字符是.（点）。
这匹配**任何字符**，而不是新行。
例：

```
import re

pattern = r"gr.y"

if re.match(pattern, "grey"):
    print("Match 1")

if re.match(pattern, "gray"):
    print("Match 2")

if re.match(pattern, "blue"):
    print("Match 3")
```

结果：

```
>>>
Match 1
Match 2
>>>
```

✧ 元字符

• 接下来的两个元字符是**^**和**$**。
• 它们分别匹配字符串的**开头**和**结尾**。
例：

```
import re

pattern = r"^gr.y$"

if re.match(pattern, "grey"):
    print("Match 1")

if re.match(pattern, "gray"):
    print("Match 2")

if re.match(pattern, "stingray"):
    print("Match 3")
```

结果：

```
>>>
Match 1
Match 2
>>>
```

模式“**^gr.y $**”表示字符串应以 **gr** 开头，然后跟随除换行符之外的任何字符，并以 **y** 结尾。

8.3.字符类

✧ 字符类

• **字符类**提供了一种仅匹配特定字符集之一的方法。
• 通过将匹配的字符放在**方括号**内来创建字符类。
例：

```
import re

pattern = r"[aeiou]"

if re.search(pattern, "grey"):
    print("Match 1")

if re.search(pattern, "qwertyuiop"):
    print("Match 2")

if re.search(pattern, "rhythm myths"):
    print("Match 3")
```

结果：

```
>>>
Match 1
Match 2
>>>
```

search 功能中的模式**[aeiou]**匹配包含任何一个定义字符的所有字符串。

✧ 字符类

• 字符类也可以匹配字符范围。
一些例子：
类**[a-z]**匹配任何小写字母字符。
类**[G-P]**匹配从 G 到 P 的任何大写字符。
类**[0-9]**匹配任何数字。
多个范围可以包含在一个类中。例如，**[A-Za-z]**匹配任何情况的字母。
例：

```
import re

pattern = r"[A-Z][A-Z][0-9]"

if re.search(pattern, "LS8"):
    print("Match 1")

if re.search(pattern, "E3"):
    print("Match 2")

if re.search(pattern, "1ab"):
    print("Match 3")
```

结果：

```
>>>
Match 1
>>>
```

上例中的模式匹配包含两个字母大写字母后跟数字的字符串。

✧ 字符类

- 在字符类的开头放置**^**以**反转**它。
- 这使它匹配除包含的字符以外的任何字符。
- 其他元字符（如**$**和**.**）在字符类中没有意义。
- 元字符**^**没有任何意义，除非它是类中的第一个字符。

例：

```
import re

pattern = r"[^A-Z]"

if re.search(pattern, "this is all quiet"):
    print("Match 1")

if re.search(pattern, "AbCdEfG123"):
    print("Match 2")

if re.search(pattern, "THISISALLSHOUTING"):
    print("Match 3")
```

结果：

```
>>>
Match 1
Match 2
>>>
```

模式**[^ A-Z]**不包括大写字符串。
注意，^应该在括号内，以反转字符类。

8.4.更多关于元字符

✧ 元字符

• 一些更多的元字符是*****，**+**，**？**，**{**和**}**。
• 这些指定了重复次数。
• 元字符*****表示“前一个事物的零次或多次重复”。它尝试匹配尽可能多的重复。“前一个事物”可以是**括号**中的单个字符，类或一组字符。
例：

```
import re

pattern = r"egg(spam)*"

if re.match(pattern, "egg"):
    print("Match 1")

if re.match(pattern, "eggspamspamegg"):
    print("Match 2")

if re.match(pattern, "spam"):
    print("Match 3")
```

结果：

```
>>>
Match 1
Match 2
>>>
```

上面的示例匹配以“**egg**”开头的字符串，并跟随零个或多个“**spam**”。

✧ 元字符

• 元字符**+**与*****非常相似，除了它表示“**一次**或多次重复”，而不是“零次或多次重复”。
例：

```
import re

pattern = r"g+"
```

```
if re.match(pattern, "g"):
    print("Match 1")

if re.match(pattern, "ggggggggggggg"):
    print("Match 2")

if re.match(pattern, "abc"):
    print("Match 3")
```

结果：

```
>>>
Match 1
Match 2
>>>
```

总结一下：
*匹配前面表达式的 0 次或更多次出现。
+匹配前一个表达式的 1 次或多次出现。

✧ 元字符

• 元字符**?**意思是“零或一次重复”。
例：

```
import re

pattern = r"ice(-)?cream"

if re.match(pattern, "ice-cream"):
    print("Match 1")

if re.match(pattern, "icecream"):
    print("Match 2")

if re.match(pattern, "sausages"):
    print("Match 3")

if re.match(pattern, "ice--ice"):
    print("Match 4")
```

结果：

```
>>>
Match 1
```

```
Match 2
>>>
```

✧ 花括号

• **花括号**可用于表示两个数字之间的重复次数。
• 正则表达式**{x,y}**表示“某事重复次数在 x 和 y 之间”。
• 因此**{0,1}**与**?**相同。
• 如果缺少第一个数字，则将其视为零。如果缺少第二个数字，则将其视为无穷大。
例：

```
import re

pattern = r"9{1,3}$"

if re.match(pattern, "9"):
    print("Match 1")

if re.match(pattern, "999"):
    print("Match 2")

if re.match(pattern, "9999"):
    print("Match 3")
```

结果：

```
>>>
Match 1
Match 2
>>>
```

“**9 {1,3} $**”匹配具有 1 到 3 个 9 的字符串。

8.5.组

✧ 组

• 可以通过用**括号**括起正则表达式的一部分来创建组。
• 这意味着可以将组作为元字符的参数给出，例如*和？。
例：

```
import re

pattern = r"egg(spam)*"

if re.match(pattern, "egg"):
    print("Match 1")

if re.match(pattern, "eggspamspamspamegg"):
    print("Match 2")

if re.match(pattern, "spam"):
    print("Match 3")
```

(spam)表示上面显示的示例模式中的一个组。
结果：

```
>>>
Match 1
Match 2
>>>
```

✧ 组

- 可以使用 **group** 函数访问匹配中的组内容。
- **group(0)**或 **group()**的调用返回整个匹配。
- **group(n)**的调用，其中 n 大于 0，从左边返回第 n 个组。
- 方法 **groups()**从 1 开始返回所有组。

例：

```
import re

pattern = r"a(bc)(de)(f(g)h)i"

match = re.match(pattern, "abcdefghijklmnop")
if match:
    print(match.group())
    print(match.group(0))
    print(match.group(1))
    print(match.group(2))
    print(match.groups())
```

结果：

```
>>>
abcdefghi
abcdefghi
bc
de
('bc', 'de', 'fgh', 'g')
>>>
```

从上面的示例中可以看出，组可以嵌套。

✧ 组

• 有几种特殊组。
• 两个有用的是**命名组**和**非捕获组**。
• **命名组**的格式为（**?P <name> ...**），其中 **name** 是组的名称，而...是内容。 它们的行为与普通组完全相同，除了它们的编号之外，它们可以通过**组（名称）**访问。
• **非捕获组**具有格式（**?:...**）。 组方法无法访问它们，因此可以将它们添加到现有正则表达式中而不会破坏编号。
例：

```
import re

pattern = r"(?P<first>abc)(?:def)(ghi)"

match = re.match(pattern, "abcdefghi")
if match:
    print(match.group("first"))
    print(match.groups())
```

结果：

```
>>>
abc
('abc', 'ghi')
>>>
```

✧ 元字符

另一个重要的元字符是**|**。
这意味着“或”，所以 **red|blue** 匹配“红色”或“蓝色”。
例：

```
import re

pattern = r"gr(a|e)y"

match = re.match(pattern, "gray")
if match:
    print ("Match 1")

match = re.match(pattern, "grey")
if match:
    print ("Match 2")

match = re.match(pattern, "griy")
if match:
print ("Match 3")
```

结果：

```
>>>
Match 1
Match 2
>>>
```

8.6.特殊序列

✧ 特殊序列

• 您可以在正则表达式中使用各种**特殊序列**。它们被写为反斜杠，后跟另一个字符。
• 一个有用的特殊序列是反斜杠和 1 到 99 之间的数字，例如\ 1 或\ 17。这以一定次数匹配组的表达式。
例：

```
import re

pattern = r"(.+) \1"

match = re.match(pattern, "word word")
if match:
    print ("Match 1")
```

```
match = re.match(pattern, "?! ?!")
if match:
    print ("Match 2")

match = re.match(pattern, "abc cde")
if match:
    print ("Match 3")
```

结果：

```
>>>
Match 1
Match 2
>>>
```

注意，**"(.+)\1"**与**"(.+)(.+)"**不同，因为**\ 1** 指的是第一个组的子表达式，它是匹配的表达式本身，而不是正则表达式模式。

✧ 特殊序列

• 更有用的特殊序列是**\ d**，**\ s** 和**\ w**。
• 这些分别匹配**数字**，**空格**和**单词字符**。
• 在 ASCII 模式下，它们相当于[0-9]，[\ t \ n \ r \ n \ f \ v]和[a-zA-Z0-9_]。
•在 Unicode 模式下，它们也匹配某些其他字符。 例如，**\ w** 匹配带重音的字母。
• 这些带有大写字母的特殊序列的版本 **- \ D**，**\ S** 和**\ W** - 表示与小写版本相反的版本。例如，**\ D** 匹配任何不是数字的东西。
例：

```
import re

pattern = r"(\D+\d)"

match = re.match(pattern, "Hi 999!")

if match:
    print("Match 1")

match = re.match(pattern, "1, 23, 456!")
if match:
    print("Match 2")

match = re.match(pattern, " ! $?")
if match:
print("Match 3")
```

结果：

```
>>>
Match 1
>>>
```

(\ D + \ d) match 一或多个非数字后跟一个数字。

✧ 特殊序列

• 其他特殊序列是**\ A**，**\ Z** 和**\ b**。
• 序列**\ A** 和**\ Z** 分别匹配字符串的开头和结尾。
• 序列**\ b** 匹配**\ w** 和**\ W** 字符之间的空字符串，或**\ w** 字符和字符串的开头或结尾。非正式地，它代表了单词之间的界限。
• 序列**\ B** 匹配其他任何地方的空字符串。

例：

```
import re

pattern = r"\b(cat)\b"

match = re.search(pattern, "The cat sat!")
if match:
    print ("Match 1")

match = re.search(pattern, "We s>cat<tered?")
if match:
    print ("Match 2")

match = re.search(pattern, "We scattered.")
if match:
    print ("Match 3")
```

结果：

```
>>>
Match 1
Match 2
>>>
```

"\b(cat)\b"基本上匹配由空格包围的单词**"cat"**。

8.7.邮件提取

✧ 邮件提取

• 为了演示正则表达式的示例用法，我们创建一个程序来从字符串中提取电子邮件地址。
• 假设我们有一个包含电子邮件地址的文本：

```
str = "Please contact info@sololearn.com for assistance"
```

• 我们的目标是提取子字符串“info@sololearn.com”。
• 基本电子邮件地址由单词组成，可包括点或短划线。 接下来是@符号和域名（名称，点和域名后缀）。
• 这是构建正则表达式的基础。

```
pattern = r"([\w\.-]+)@([\w\.-]+)(\.[\w\.]+)"
```

• **[\ w \ .-] +**匹配一个或多个单词字符，点或短划线。
• 上面的正则表达式说字符串应该包含一个单词（含点和短划线），然后是@符号，然后是另一个相似的单词，然后是一个点和另一个单词。
我们的正则表达式包含三组：
1 - 电子邮件地址的第一部分。
2 - 没有后缀的域名。
3 - 域后缀。

✧ 邮件提取

• 把它们放在一起：

```
import re

pattern = r"([\w\.-]+)@([\w\.-]+)(\.[\w\.]+)"
str = "Please contact info@sololearn.com for assistance"

match = re.search(pattern, str)
if match:
    print(match.group())
```

结果：

```
>>>
info@sololearn.com
>>>
```

• 如果字符串包含多个电子邮件地址，我们可以使用 **re.findall** 方法而不是 **re.search** 来提取所有电子邮件地址。
此示例中的正则表达式仅用于演示目的。
完全验证电子邮件地址需要更复杂的正则表达式。

9.Python 风格和打包

9.1.Python 之禅

✧ Python 之禅

•编写程序完成他们应该做的事情只是成为优秀 Python 程序员的一个组成部分。
• 编写易于理解的干净代码也很重要，即使在编写完几周后也是如此。

• 这样做的一种方法是遵循 **Python 之禅**，这是一套有点幽默的原则，可以作为 Python 程序员方式编程的指南。使用以下代码访问 Python 之禅。

```
import this
```

结果：
Python 之禅宗，by Tim Peters

优美胜于丑陋（*Python 以编写优美的代码为目标*）
明了胜于晦涩（*优美的代码应当是明了的，命名规范，风格相似*）
简洁胜于复杂（*优美的代码应当是简洁的，不要有复杂的内部实现*）
复杂胜于凌乱（*如果复杂不可避免，那代码间也不能有难懂的关系，要保持接口简洁*）
扁平胜于嵌套（*优美的代码应当是扁平的，不能有太多的嵌套*）
间隔胜于紧凑（*优美的代码有适当的间隔，不要奢望一行代码解决问题*）
可读性很重要（*优美的代码是可读的*）
即便假借特例的实用性之名，也不可违背这些规则（*这些规则至高无上*）
不要包容所有错误，除非你确定需要这样做（*精准地捕获异常，不写 except:pass 风格的代码*）
当存在多种可能，不要尝试去猜测
而是尽量找一种，最好是唯一一种明显的解决方案（*如果不确定，就用穷举法*）
虽然这并不容易，因为你不是 Python 之父（*这里的 Dutch 是指 Guido*）
做也许好过不做，但不假思索就动手还不如不做（*动手之前要细思量*）
如果你无法向人描述你的方案，那肯定不是一个好方案；反之亦然（*方案测评标准*）
命名空间是一种绝妙的理念，我们应当多加利用（*倡导与号召*）

✧ Python 之禅

• Python 之禅中的一些行可能需要更多解释。

• 明了胜于晦涩：最好明确说明代码正在做什么。这就是为什么将数字字符串添加到整数需要显式转换，而不是像在其他语言中那样在幕后发生。
• 扁平胜于嵌套：应避免使用重型嵌套结构（列表，列表，以及 on 和 on .....）。
错误永远不应该以沉默方式传递：通常，当发生错误时，您应该输出某种错误消息，而不是忽略它。

• 在 Python 之禅中有 20 条原则，但只有 19 行文本。
• 第 20 条原则是一个意见问题，但我们的解释是空白行意味着“使用空白”。
“而是尽量找一种，最好是唯一一种明显的解决方案”这一行引用并与 Perl 语言哲学相矛盾，即应该有不止一种方法来实现它。

9.2.PEP

✧ PEP

• **Python 增强建议（PEP）**是经验丰富的 Python 开发人员对语言进行改进的建议。
• **PEP 8** 是关于编写可读代码主题的样式指南。它包含许多引用变量名称的指南，这些指南总结如下：
- 模块应该有简短的全小写名称;
- 类名称应采用 CapWords 风格;
- 大多数变量和函数名称应为 lowercase_with_underscores;
- 常量（永远不会改变值的变量）应该是 CAPS_WITH_UNDERSCORES;
- 与 Python 关键字冲突的名称（例如'class'或'if'）应该有一个尾随下划线。

• **PEP 8** 还建议在操作符周围使用空格，并使用逗号后增加可读性。
但是，不应过度使用空格。例如，避免在任何类型的括号内直接有任何空格。

✧ PEP 8

• 其他 **PEP 8** 建议包括以下内容：
- 行不应超过 80 个字符;
- 应避免'from module import *';
- 每行应该只有一个语句。

• 它还建议您使用空格而不是制表符来缩进。但是，在某种程度上，这是个人偏好的问题。如果使用空格，则每行只使用 4 个。选择一个并坚持下去更为重要。

• PEP 中最重要的建议是在有意义的时候忽略它。当它会导致您的代码可读性降低时，请不要理会以下 PEP 建议；与周围的代码不一致；或不向后兼容。

• 但是，总的来说，遵循 PEP 8 将大大提高代码的质量。
其他一些着名的 PEP 涵盖了代码风格：
PEP 20：Python 之禅
PEP 257：Docstrings 的样式约定

9.3.更多关于函数参数

✧ 函数参数

• Python 允许具有**不同数量的参数**的函数。
• 使用*** args** 作为函数参数，可以将任意数量的参数传递给该函数。然后可以作为函数体中的元组 **args** 访问参数。
例：

```
def function(named_arg, *args):
    print(named_arg)
    print(args)

function(1, 2, 3, 4, 5)
```

结果：

```
>>>
1
(2, 3, 4, 5)
>>>
```

参数*** args** 必须在函数的命名参数之后。
args 这个名字只是一个惯例；你可以选择使用另一个。

✧ 默认值

• 通过为函数提供**默认值**，可以使函数的命名参数成为可选参数。
这些必须在没有默认值的命名参数之后。
例：

```
def function(x, y, food="spam"):
    print(food)

function(1, 2)
function(3, 4, "egg")
```

结果：

```
>>>
spam
egg
>>>
```

如果传入参数，则忽略默认值。
如果未传入参数，则使用默认值。

✧ 函数参数

• **** kwargs**（代表关键字参数）允许您处理事先未定义的命名参数。
• 关键字参数返回一个字典，其中键是参数名称，值是参数值。
例：

```
def my_func(x, y=7, *args, **kwargs):
    print(kwargs)

my_func(2, 3, 4, 5, 6, a=7, b=8)
```

结果：

```
>>>
{'a': 7, 'b': 8}
>>>
```

• **a** 和 **b** 是我们传递给函数调用的参数名称。
**** kwargs** 返回的参数不包含在**** args** 中。

9.4.元祖拆包

✧ 元组拆包

• **元组解包**允许您将可迭代（通常是元组）中的每个项目分配给变量。
例：

```
numbers = (1, 2, 3)
a, b, c = numbers
print(a)
```

```
print(b)
print(c)
```

结果：

```
>>>
1
2
3
>>>
```

这也可以通过执行 **a,b = b,a** 来交换变量，因为 **b,a** 在右边形成元组**(b,a)**然后解包。

✧ 元组拆包

• 以星号（*）开头的变量从迭代中获取其他变量遗留的所有值。

例：

```
a, b, *c, d = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(a)
print(b)
print(c)
print(d)
```

结果：

```
>>>
1
2
[3, 4, 5, 6, 7, 8]
9
>>>
```

9.5.三元运算符

✧ 三元运算符

• 条件表达式在使用较少代码时提供 if 语句的功能。它们不应过度使用，因为它们很容易降低可读性，但在分配变量时它们通常很有用。

• 条件表达式也称为**三元运算符**的应用程序。

例：

```
a = 7
b = 1 if a >= 5 else 42
print(b)
```

结果：

```
>>>
1
>>>
```

• 三元运算符检查条件并返回相应的值。
• 在上面的示例中，当条件为真时，**b** 被赋值为 1.如果 **a** 小于 5，则它将被赋值为 42。
另一个例子：

```
status = 1
msg = "Logout" if status == 1 else "Login"
```

所谓的三元运算符是因为与大多数运算符不同，它需要三个参数。

9.6.更多关于 else 语句

✧ else

• **else** 语句最常与 **if** 语句一起使用，但它也可以跟随 **for** 或 **while** 循环，这赋予它不同的含义。
• 使用 **for** 或 **while** 循环，如果循环正常结束（当 **break** 语句不导致从循环退出时），则调用其中的代码。
例：

```
for i in range(10):
    if i == 999:
        break
else:
    print("Unbroken 1")

for i in range(10):
    if i == 5:
        break
else:
    print("Unbroken 2")
```

结果：

```
>>>
Unbroken 1
>>>
```

第一个 **for** 循环正常执行，导致打印“Unbroken 1”。
第二个循环由于**中断**而退出，这就是为什么不执行 **else** 语句的原因。

✧ else

• **else** 语句也可以与 **try / except** 语句一起使用。
• 在这种情况下，只有在 **try** 语句中没有发生错误时才会执行其中的代码。
例：

```
try:
    print(1)
except ZeroDivisionError:
    print(2)
else:
    print(3)

try:
    print(1/0)
except ZeroDivisionError:
    print(4)
else:
    print(5)
```

结果：

```
>>>
1
3
4
>>>
```

9.7.__main__

✧ __main__

• 大多数 Python 代码要么是要导入的模块，要么是执行某些操作的脚本。
但是，有时创建一个既可以作为模块导入又可以作为脚本运行的文件也很有用。

为此，请在 **if __name__=="__main__"**内放置脚本代码。
这可确保在导入文件时不会运行它。
例：

```
def function():
    print("This is a module function")

if __name__=="__main__":
    print("This is a script")
```

结果：

```
>>>
This is a script
>>>
```

当 Python 解释器读取源文件时，它会执行它在文件中找到的所有代码。在执行代码之前，它定义了一些特殊变量。
例如，如果 Python 解释器将该模块（源文件）作为主程序运行，则它将特殊的 **__name__**变量设置为具有值**"__main__"**。如果从另一个模块导入此文件，则 **__name__**将设置为模块的名称。

✧ __main__

• 如果我们将前一个示例中的代码保存为名为 **sololearn.py** 的文件，我们可以使用名称 **sololearn** 将其作为模块导入另一个脚本。
sololearn.py

```
def function():
    print("This is a module function")

if __name__=="__main__":
    print("This is a script")
```

some_script.py

```
import sololearn

sololearn.function()
```

结果：

```
>>>
This is a module function
>>>
```

基本上，我们已经创建了一个名为 **sololearn** 的自定义模块，然后在另一个脚本中使用它。

9.8.主要的第三方库

✧ 主要的第三方库

• 仅 Python 标准库包含广泛的功能。
• 但是，某些任务需要使用第三方库。一些主要的第三方库：
• **Django**：用 Python 编写的最常用的 Web 框架，Django 支持包含 Instagram 和 Disqus 的网站。它有许多有用的功能，扩展包涵盖了它缺少的任何功能。
• **CherryPy** 和 **Flask** 也是流行的 Web 框架。

• 为了从网站上抓取数据，**BeautifulSoup** 库非常有用，与使用正则表达式构建自己的工具相比，可以获得更好的结果。
虽然 Python 确实提供了以编程方式访问网站的模块，例如 urllib，但它们使用起来非常麻烦。第三方库请求使得使用 HTTP 请求变得更加容易。

✧ 主要的第三方库

• 有许多第三方模块可以使用 Python 进行科学和数学计算变得更加容易。
• 模块 **matplotlib** 允许您基于 Python 中的数据创建图形。
• 模块 **NumPy** 允许使用比嵌套列表的本机 Python 解决方案快得多的多维数组。它还包含执行数学运算的函数，例如数组上的矩阵变换。
• **SciPy** 库包含许多 **NumPy** 功能的扩展。

• Python 也可以用于**游戏开发**。
• 通常，它被用作用其他语言编写的游戏的脚本语言，但它可以用来自己制作游戏。
对于 3D 游戏，可以使用 **Panda3D** 库。对于 2D 游戏，您可以使用 **pygame**。

9.9.打包

✧ 打包

• 在 Python 中，术语**打包**是指将您编写的模块放在标准格式中，以便其他程序员可以轻松地安装和使用它们。
• 这涉及使用模块 **setuptools** 和 **distutils**。

•打包的第一步是正确组织现有文件。将要放入库中的所有文件放在同一个父目录中。此目录还应包含名为**__init__.py** 的文件，该文件可以为空，但必须存在于目录中。
•该目录进入另一个包含自述文件和许可证的目录，以及一个名为 **setup.py** 的重要文件。
示例目录结构：

```
SoloLearn/
    LICENSE.txt
    README.txt
    setup.py
    sololearn/
        __init__.py
        sololearn.py
        sololearn2.py
```

您可以根据需要在目录中放置尽可能多的脚本文件。

✧ 打包

• 打包的下一步是编写 **setup.py** 文件。
• 这包含组装包所需的信息，因此可以将其上载到 **PyPI** 并使用 **pip**（名称，版本等）进行安装。
setup.py 文件的示例：

```
from distutils.core import setup

setup(
    name='SoloLearn',
    version='0.1dev',
    packages=['sololearn',],
    license='MIT',
    long_description=open('README.txt').read(),
)
```

• 创建 **setup.py** 文件后，将其上载到 **PyPI**，或使用命令行创建二进制分发（可执行安装程序）。
•要构建源代码分发，请使用命令行导航到包含 setup.py 的目录，然后运行命令 **python setup.py sdist**。
• 运行 **python setup.py bdist** 或者，对于 Windows，运行 **python setup.py bdist_wininst** 来构建二进制分发。
• 使用 **python setup.py** 寄存器，然后使用 **python setup.py sdist upload** 上传包。
最后，使用 **python setup.py install** 安装包。

9.10.打包给用户

✧ 打包

• 上一课涵盖了其他 Python 程序员使用的打包模块。 但是，许多非程序员的计算机用户没有安装 Python。因此，将脚本打包为相关平台（例如 Windows 或 Mac 操作系统）的可执行文件非常有用。这对于 Linux 来说不是必需的，因为大多数 Linux 用户都安装了 Python，并且能够按原样运行脚本。

• 对于 Windows，许多工具可用于将脚本转换为可执行文件。例如，py2exe 可用于将 Python 脚本及其所需的库打包到单个可执行文件中。
• **PyInstaller** 和 **cx_Freeze** 具有相同的用途。
对于 Mac，请使用 **py2app**，**PyInstaller** 或 **cx_Freeze**。

CERTIFICATE

SOLOLEARN
DISCOVER YOUR POTENTIAL

Issued 15 November, 2017

This is to certify that

Vincent

has successfully completed the

C++ Tutorial course

Yeva Hyusyan
Chief Executive Officer

Certificate #1051-6684149