W3school Python 教程

制作者：张恒涛

2021-07-03

说明：
1、实现爬虫爬取 w3shool 里 Python 教程（爬取 1 次内容），存为 word 作为电子书；
2、包含 Python 教程、文件处理、Python Numpy、机器学习、Python MySQL、Python MongoDB、Python 参考手册、模块参考手册、Python How To；
3、使用 Python，requests 库，pycharm 工具实现；
4、静态网页，非动态网页；（实际证明网页信息为动态加载！）

备注：https://www.w3school.com.cn/python/index.asp

Sat Jul 3 16:00:46 2021

Python 是一门编程语言。
您可以在服务器上使用 Python 来创建 Web 应用程序。

通过实例学习
我们的 TIY 编辑器使学习 Python 变得简单，它能够同时显示代码和结果。
实例
```
print("Hello, World!")
```
运行实例
单击“运行实例”按钮来查看它如何运行。

Python 文件处理
在我们的文件处理章节，您将学习如何打开、读取、写入和删除文件。
Python 文件处理

Python 数据库处理
在我们的数据库章节，您将学习如何访问和使用 MySQL 和 MongoDB 数据库：
Python MySQL 教程
Python MongoDB 教程

Python 实例
通过实例学习！本教程为您提供清晰的实例以及相应的解释。
查看所有 python 实例

Python 测验
通过测验来学习！这个测验会测试您对 Python 的掌握程度。
python 测验

Python 参考手册
我们提供完成的函数和方法参考手册：
参考概述
内建函数
字符串方法
列表/数组方法
字典方法
元组方法
集合方法
文件方法
Python 关键字

下载 Python

从 Python 的官方网站下载 Python：https://python.org

Python 教程
Python 简介

什么是 Python?
Python 是一门流行的编程语言。它由 Guido van Rossum 创建，于 1991 年发布。
它用于：
Web 开发（服务器端）
软件开发
数学
系统脚本

Python 可以做什么？
可以在服务器上使用 Python 来创建 Web 应用程序。
Python 可以与软件一起使用来创建工作流。
Python 可以连接到数据库系统。它还可以读取和修改文件。
Python 可用于处理大数据并执行复杂的数学运算。
Python 可用于快速原型设计，也可用于生产就绪的软件开发。

为何选择 Python?
Python 适用于不同的平台（Windows、Mac、Linux、Raspberry Pi 等）。
Python 有一种类似于英语的简单语法。
Python 的语法允许开发人员用比其他编程语言更少的代码行编写程序。
Python 在解释器系统上运行，这意味着代码可以在编写后立即执行。这也意味着原型设计可以非常快。
Python 可以以程序方式、面向对象的方式或功能方式来处理。

请您知晓
Python 的最新主要版本是 Python 3，我们将在本教程中使用它。但是，Python 2 虽然没有更新安全更新以外的任何东西，但仍然非常受欢迎。
在本教程中，我们将在在文本编辑器中编写 Python。您也可以在集成开发环境中编写 Python，例如 Thonny、Pycharm、Netbeans 或 Eclipse，这一点当您在管理大量 Python 文件时特别有用。

Python 语法与其他编程语言比较
Python 是为可读性设计的，与英语有一些相似之处，并受到数学的影响。
Python 使用新行来完成命令，而不像通常使用分号或括号的其他编程语言。

Python 依赖缩进，使用空格来定义范围；例如循环、函数和类的范围。其他编程语言通常使用花括号来实现此目的。

Python 教程
Python 入门

Python 安装
许多 PC 和 Mac 都已经安装了 python。
要检查是否已在 Windows PC 上安装了 python，请在开始栏中寻找 Python 或在命令行（cmd.exe）上运行以下命令：
C:\Users\Your Name>python --version
要检查您是否在 Linux 或 Mac 上安装了 python，请在 Linux 上打开命令行或在 Mac 上打开终端并键入：
python --version
如果您发现计算机上没有安装 python，则可以从以下网站免费下载：
https://www.python.org/

Python 快速入门
Python 是一门解释型编程语言，这意味着作为开发人员，您可以在文本编辑器中编写 Python（.py）文件，然后将这些文件放入 python 解释器中执行。
在命令行上运行 python 文件的方式如下：
C:\Users\Your Name>python helloworld.py
其中 “helloworld.py” 是 python 的文件名。
让我们编写第一个 Python 文件，名为 helloworld.py，它可以在任何文本编辑器中完成。
helloworld.py
```
print("Hello, World!")
```
运行实例
就那么简单。保存文件。打开命令行，导航到保存文件的目录，然后运行：
C:\Users\Your Name>python helloworld.py
输出：
Hello, World!
恭喜，您已经编写并执行了第一个 Python 程序。

Python 命令行
要在 python 中测试少量代码，在文件中写代码有时不是最快最简单的。把 Python 作为命令行来运行是可能的。
在 Windows、Mac 或 Linux 命令行上键入以下内容：
C:\Users\Your Name>python

在此，您可以编写任何 python，包括本教程前面的 hello world 例子：

```
C:\Users\Your Name>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
```

这将在命令行中输出 "Hello, World!"：

```
C:\Users\Your Name>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
Hello, World!
```

无论何时，您都可以通过键入如下命令来退出 python 命令行界面：

```
exit()
```

Python 简介

Python 语法

执行 Python 语法

正如我们在上一节中学习到的，可以直接在命令行中编写执行 Python 的语法：

```
>>> print("Hello, World!")
Hello, World!
```

或者通过在服务器上创建 python 文件，使用 .py 文件扩展名，并在命令行中运行它：

```
C:\Users\Your Name>python myfile.py
```

Python 缩进

缩进指的是代码行开头的空格。

在其他编程语言中，代码缩进仅出于可读性的考虑，而 Python 中的缩进非常重要。

Python 使用缩进来指示代码块。

实例

```
if 5 > 2:
  print("Five is greater than two!")
```

运行实例

如果省略缩进，Python 会出错：

实例

语法错误：

```
if 5 > 2:
print("Five is greater than two!")
```

运行实例

空格数取决于程序员，但至少需要一个。

实例

```
if 5 > 2:
 print("Five is greater than two!")
if 5 > 2:
        print("Five is greater than two!")
```

运行实例

您必须在同一代码块中使用相同数量的空格，否则 Python 会出错：

实例

语法错误：

```
if 5 > 2:
 print("Five is greater than two!")
        print("Five is greater than two!")
```

运行实例

Python 变量

在 Python 中，变量是在为其赋值时创建的：

实例

Python 中的变量：

```
x = 5
y = "Hello, World!"
```

运行实例

Python 没有声明变量的命令。

您将在 Python 变量 章节中学习有关变量的更多知识。

注释

Python 拥有对文档内代码进行注释的功能。

注释以 # 开头，Python 将其余部分作为注释呈现：

实例

Python 中的注释：

```
#This is a comment.
print("Hello, World!")
```

运行实例

Python 入门

Python 注释

注释可用于解释 Python 代码。
注释可用于提高代码的可读性。
在测试代码时，可以使用注释来阻止执行。

创建注释
注释以 # 开头，Python 将忽略它们：
实例
```
#This is a comment
print("Hello, World!")
```
运行实例
注释可以放在一行的末尾，Python 将忽略该行的其余部分：
实例
```
print("Hello, World!") #This is a comment
```
运行实例
注释不必是解释代码的文本，它也可以用来阻止 Python 执行代码：
实例
```
#print("Hello, World!")
print("Cheers, Mate!")
```
运行实例

多行注释
Python 实际上没有多行注释的语法。
要添加多行注释，您可以为每行插入一个 #：
实例
```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```
运行实例
或者，以不完全符合预期的方式，您可以使用多行字符串。
由于 Python 将忽略未分配给变量的字符串文字，因此您可以在代码中添加多行字符串（三引号），并在其中添加注释：
实例
```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```
运行实例
只要字符串未分配给变量，Python 就会读取代码，然后忽略它，这样您就已经完成了多行注释。

Python 语法

Python 变量

创建变量
变量是存放数据值的容器。
与其他编程语言不同，Python 没有声明变量的命令。
首次为其赋值时，才会创建变量。
实例
```
x = 10
y = "Bill"
print(x)
print(y)
```
运行实例
变量不需要使用任何特定类型声明，甚至可以在设置后更改其类型。
实例
```
x = 5 # x is of type int
x = "Steve" # x is now of type str
print(x)
```
运行实例
字符串变量可以使用单引号或双引号进行声明：
实例
```
x = "Bill"
# is the same as
x = 'Bill'
```
运行实例

变量名称
变量可以使用短名称（如 x 和 y）或更具描述性的名称（age、carname、total_volume）。
Python 变量命名规则：
变量名必须以字母或下划线字符开头
变量名称不能以数字开头
变量名只能包含字母数字字符和下划线（A-z、0-9 和 _）
变量名称区分大小写（age、Age 和 AGE 是三个不同的变量）
请记住，变量名称区分大小写

向多个变量赋值
Python 允许您在一行中为多个变量赋值：
实例
```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
```

```
print(z)
```
运行实例
您可以在一行中为多个变量分配相同的值：
实例
```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```
运行实例

输出变量
Python 的 print 语句通常用于输出变量。
如需结合文本和变量，Python 使用 + 字符：
实例
```
x = "awesome"
print("Python is " + x)
```
运行实例
您还可以使用 + 字符将变量与另一个变量相加：
实例
```
x = "Python is "
y = "awesome"
z =  x + y
print(z)
```
运行实例
对于数字，+ 字符用作数学运算符：
实例
```
x = 5
y = 10
print(x + y)
```
运行实例
如果您尝试组合字符串和数字，Python 会给出错误：
实例
```
x = 10
y = "Bill"
print(x + y)
```
运行实例

全局变量
在函数外部创建的变量（如上述所有实例所示）称为全局变量。
全局变量可以被函数内部和外部的每个人使用。
实例
在函数外部创建变量，并在函数内部使用它：
```
x = "awesome"
```

```
def myfunc():
  print("Python is " + x)

myfunc()
```

运行实例

如果在函数内部创建具有相同名称的变量，则该变量将是局部变量，并且只能在函数内部使用。具有相同名称的全局变量将保留原样，并拥有原始值。

实例

在函数内部创建一个与全局变量同名的变量：

```
x = "awesome"

def myfunc():
  x = "fantastic"
  print("Python is " + x)

myfunc()

print("Python is " + x)
```

运行实例

global 关键字

通常，在函数内部创建变量时，该变量是局部变量，只能在该函数内部使用。

要在函数内部创建全局变量，您可以使用 global 关键字。

实例

如果您用了 global 关键字，则该变量属于全局范围：

```
def myfunc():
  global x
  x = "fantastic"

myfunc()

print("Python is " + x)
```

运行实例

另外，如果要在函数内部更改全局变量，请使用 global 关键字。

实例

要在函数内部更改全局变量的值，请使用 global 关键字引用该变量：

```
x = "awesome"

def myfunc():
  global x
  x = "fantastic"
```

```
myfunc()

print("Python is " + x)
```
运行实例

Python 注释
Python 数据类型

内置数据类型
在编程中，数据类型是一个重要的概念。
变量可以存储不同类型的数据，并且不同类型可以执行不同的操作。
在这些类别中，Python 默认拥有以下内置数据类型：
文本类型： str
数值类型： int, float, complex
序列类型： list, tuple, range
映射类型： dict
集合类型： set, frozenset
布尔类型： bool
二进制类型： bytes, bytearray, memoryview

获取数据类型
您可以使用 type() 函数获取任何对象的数据类型：
实例
打印变量 x 的数据类型：
```
x = 10
print(type(x))
```
运行实例

设置数据类型
在 Python 中，当您为变量赋值时，会设置数据类型：
示例 数据类型
```
x = "Hello World" str
x = 29 int
x = 29.5 float
x = 1j complex
x = ["apple", "banana", "cherry"] list
x = ("apple", "banana", "cherry") tuple
x = range(6) range
x = {"name" : "Bill", "age" : 63} dict
x = {"apple", "banana", "cherry"} set
x = frozenset({"apple", "banana", "cherry"}) frozenset
```

```
x = True bool
x = b"Hello" bytes
x = bytearray(5) bytearray
x = memoryview(bytes(5)) memoryview
```

设定特定的数据类型
如果希望指定数据类型，则您可以使用以下构造函数：
示例 数据类型

```
x = str("Hello World") str
x = int(29) int
x = float(29.5) float
x = complex(1j) complex
x = list(("apple", "banana", "cherry")) list
x = tuple(("apple", "banana", "cherry")) tuple
x = range(6) range
x = dict(name="Bill", age=36) dict
x = set(("apple", "banana", "cherry")) set
x = frozenset(("apple", "banana", "cherry")) frozenset
x = bool(5) bool
x = bytes(5) bytes
x = bytearray(5) bytearray
x = memoryview(bytes(5)) memoryview
```

Python 变量
Python 数字

Python 数字
Python 中有三种数字类型：
int
float
complex
为变量赋值时，将创建数值类型的变量：
实例

```
x = 10    # int
y = 6.3   # float
z = 2j    # complex
```

如需验证 Python 中任何对象的类型，请使用 type() 函数：
实例

```
print(type(x))
print(type(y))
print(type(z))
```

运行实例

Int
Int 或整数是完整的数字，正数或负数，没有小数，长度不限。
实例
整数：

```
x = 10
y = 37216654545182186317
z = -465167846

print(type(x))
print(type(y))
print(type(z))
```

运行实例

Float
浮动或“浮点数”是包含小数的正数或负数。
实例
浮点：

```
x = 3.50
y = 2.0
z = -63.78

print(type(x))
print(type(y))
print(type(z))
```

运行实例
浮点数也可以是带有“e”的科学数字，表示 10 的幂。
实例
浮点：

```
x = 27e4
y = 15E2
z = -49.8e100

print(type(x))
print(type(y))
print(type(z))
```

运行实例

复数
复数用 "j" 作为虚部编写：
实例
复数：

```
x = 2+3j
```

```
y = 7j
z = -7j

print(type(x))
print(type(y))
print(type(z))
```

运行实例

类型转换

您可以使用 int()、float() 和 complex() 方法从一种类型转换为另一种类型：

实例

从一种类型转换为另一种类型：

```
x = 10 # int
y = 6.3 # float
z = 1j # complex

# 把整数转换为浮点数

a = float(x)

# 把浮点数转换为整数

b = int(y)

# 把整数转换为复数：

c = complex(x)

print(a)
print(b)
print(c)

print(type(a))
print(type(b))
print(type(c))
```

运行实例

注释：您无法将复数转换为其他数字类型。

随机数

Python 没有 random() 函数来创建随机数，但 Python 有一个名为 random 的内置模块，可用于生成随机数：

实例

导入 random 模块，并显示 1 到 9 之间的随机数：

```
import random

print(random.randrange(1,10))
```

运行实例

在 Random 模块参考手册 中，您将了解有关 Random 模块的更多信息。

Python 数据类型

Python Casting

指定变量类型

有时您可能需要为变量指定类型。这可以通过 casting 来完成。Python 是一门面向对象的语言，因此它使用类来定义数据类型，包括其原始类型。

因此，使用构造函数完成在 python 中的转换：

int() - 用整数字面量、浮点字面量构造整数（通过对数进行下舍入），或者用表示完整数字的字符串字面量

float() - 用整数字面量、浮点字面量，或字符串字面量构造浮点数（提供表示浮点数或整数的字符串）

str() - 用各种数据类型构造字符串，包括字符串，整数字面量和浮点字面量

实例

整数：

```
x = int(1)   # x 将是 1
y = int(2.5) # y 将是 2
z = int("3") # z 将是 3
```

运行实例

实例

浮点数：

```
x = float(1)     # x 将是 1.0
y = float(2.5)   # y 将是 2.5
z = float("3")   # z 将是 3.0
w = float("4.6") # w 将是 4.6
```

运行实例

实例

字符串：

```
x = str("S2") # x 将是 'S2'
y = str(3)    # y 将是 '3'
z = str(4.0)  # z 将是 '4.0'
```

运行实例

Python 数字

Python 字符串

字符串字面量
python 中的字符串字面量由单引号或双引号括起。
'hello' 等同于 "hello"。
您可以使用 print() 函数显示字符串字面量：
实例
```
print("Hello")
print('Hello')
```
运行实例

用字符串向变量赋值
通过使用变量名称后跟等号和字符串，可以把字符串赋值给变量：
实例
```
a = "Hello"
print(a)
```
运行实例

多行字符串
您可以使用三个引号将多行字符串赋值给变量：
实例
您可以使用三个双引号：
```
a = """Python is a widely used general-purpose, high level programming language.
It was initially designed by Guido van Rossum in 1991
and developed by Python Software Foundation.
It was mainly developed for emphasis on code readability,
and its syntax allows programmers to express concepts in fewer lines of code."""
print(a)
```
运行实例
或三个单引号：
实例
```
a = '''Python is a widely used general-purpose, high level programming language.
It was initially designed by Guido van Rossum in 1991
and developed by Python Software Foundation.
It was mainly developed for emphasis on code readability,
and its syntax allows programmers to express concepts in fewer lines of code.'''
print(a)
```

运行实例
注释：在结果中，换行符插入与代码中相同的位置。

字符串是数组
像许多其他流行的编程语言一样，Python 中的字符串是表示 unicode 字符的字节数组。
但是，Python 没有字符数据类型，单个字符就是长度为 1 的字符串。
方括号可用于访问字符串的元素。
实例
获取位置 1 处的字符（请记住第一个字符的位置为 0）：
```
a = "Hello, World!"
print(a[1])
```
运行实例
裁切
您可以使用裁切语法返回一定范围的字符。
指定开始索引和结束索引，以冒号分隔，以返回字符串的一部分。
实例
获取从位置 2 到位置 5（不包括）的字符：
```
b = "Hello, World!"
print(b[2:5])
```
运行实例
负的索引
使用负索引从字符串末尾开始切片：
实例
获取从位置 5 到位置 1 的字符，从字符串末尾开始计数：
```
b = "Hello, World!"
print(b[-5:-2])
```
运行实例
字符串长度
如需获取字符串的长度，请使用 len() 函数。
实例
len() 函数返回字符串的长度：
```
a = "Hello, World!"
print(len(a))
```
运行实例

字符串方法
Python 有一组可用于字符串的内置方法。
实例
strip() 方法删除开头和结尾的空白字符：
```
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"
```
运行实例
实例

lower() 返回小写的字符串：

```
a = "Hello, World!"
print(a.lower())
```

运行实例

实例

upper() 方法返回大写的字符串：

```
a = "Hello, World!"
print(a.upper())
```

运行实例

实例

replace() 用另一段字符串来替换字符串：

```
a = "Hello, World!"
print(a.replace("World", "Kitty"))
```

运行实例

实例

split() 方法在找到分隔符的实例时将字符串拆分为子字符串：

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

运行实例

请使用我们的字符串方法参考手册，学习更多的字符串方法。

检查字符串

如需检查字符串中是否存在特定短语或字符，我们可以使用 in 或 not in 关键字。

实例

检查以下文本中是否存在短语 "ina"：

```
txt = "China is a great country"
x = "ina" in txt
print(x)
```

运行实例

实例

检查以下文本中是否没有短语 "ina"：

```
txt = "China is a great country"
x = "ain" not in txt
print(x)
```

运行实例

字符串级联（串联）

如需串联或组合两个字符串，您可以使用 + 运算符。

实例

将变量 a 与变量 b 合并到变量 c 中：

```
a = "Hello"
b = "World"
c = a + b
```

```
print(c)
```
运行实例
实例
在它们之间添加一个空格：
```
a = "Hello"
b = "World"
c = a + " " + b
print(c)
```
运行实例

字符串格式
正如在 Python 变量一章中所学到的，我们不能像这样组合字符串和数字：
实例
```
age = 63
txt = "My name is Bill, I am " + age
print(txt)
```
运行实例
但是我们可以使用 format() 方法组合字符串和数字！
format() 方法接受传递的参数，格式化它们，并将它们放在占位符 {} 所在的字符串中：
实例
使用 format() 方法将数字插入字符串：
```
age = 63
txt = "My name is Bill, and I am {}"
print(txt.format(age))
```
运行实例
format() 方法接受不限数量的参数，并放在各自的占位符中：
实例
```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```
运行实例
您可以使用索引号 {0} 来确保参数被放在正确的占位符中：
实例
```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```
运行实例

字符串方法
Python 有一组可以在字符串上使用的内建方法。
注释：所有字符串方法都返回新值。它们不会更改原始字符串。
方法 描述
capitalize() 把首字符转换为大写。
casefold() 把字符串转换为小写。
center() 返回居中的字符串。
count() 返回指定值在字符串中出现的次数。
encode() 返回字符串的编码版本。
endswith() 如果字符串以指定值结尾，则返回 true。
expandtabs() 设置字符串的 tab 尺寸。
find() 在字符串中搜索指定的值并返回它被找到的位置。
format() 格式化字符串中的指定值。
format_map() 格式化字符串中的指定值。
index() 在字符串中搜索指定的值并返回它被找到的位置。
isalnum() 如果字符串中的所有字符都是字母数字，则返回 True。
isalpha() 如果字符串中的所有字符都在字母表中，则返回 True。
isdecimal() 如果字符串中的所有字符都是小数，则返回 True。
isdigit() 如果字符串中的所有字符都是数字，则返回 True。
isidentifier() 如果字符串是标识符，则返回 True。
islower() 如果字符串中的所有字符都是小写，则返回 True。
isnumeric() 如果字符串中的所有字符都是数，则返回 True。
isprintable() 如果字符串中的所有字符都是可打印的，则返回 True。
isspace() 如果字符串中的所有字符都是空白字符，则返回 True。
istitle() 如果字符串遵循标题规则，则返回 True。
isupper() 如果字符串中的所有字符都是大写，则返回 True。
join() 把可迭代对象的元素连接到字符串的末尾。
ljust() 返回字符串的左对齐版本。
lower() 把字符串转换为小写。
lstrip() 返回字符串的左修剪版本。
maketrans() 返回在转换中使用的转换表。
partition() 返回元组，其中的字符串被分为三部分。
replace() 返回字符串，其中指定的值被替换为指定的值。
rfind() 在字符串中搜索指定的值，并返回它被找到的最后位置。
rindex() 在字符串中搜索指定的值，并返回它被找到的最后位置。
rjust() 返回字符串的右对齐版本。
rpartition() 返回元组，其中字符串分为三部分。
rsplit() 在指定的分隔符处拆分字符串，并返回列表。
rstrip() 返回字符串的右边修剪版本。
split() 在指定的分隔符处拆分字符串，并返回列表。
splitlines() 在换行符处拆分字符串并返回列表。
startswith() 如果以指定值开头的字符串，则返回 true。
strip() 返回字符串的剪裁版本。

swapcase() 切换大小写，小写成为大写，反之亦然。
title() 把每个单词的首字符转换为大写。
translate() 返回被转换的字符串。
upper() 把字符串转换为大写。
zfill() 在字符串的开头填充指定数量的 0 值。
注释：所有字符串方法都返回新值。它们不会更改原始字符串。

Python Casting
Python 布尔

布尔表示两值之一：True 或 False。

布尔值
在编程中，您通常需要知道表达式是 True 还是 False。
您可以计算 Python 中的任何表达式，并获得两个答案之一，即 True 或 False。
比较两个值时，将对表达式求值，Python 返回布尔值答案：
实例
```
print(8 > 7)
print(8 == 7)
print(8 > 7)
```
运行实例
当在 if 语句中运行条件时，Python 返回 True 或 False：
实例
根据条件是对还是错，打印一条消息：
```
a = 200
b = 33

if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```
运行实例

评估值和变量
bool() 函数可让您评估任何值，并为您返回 True 或 False。
实例
评估字符串和数字：
```
print(bool("Hello"))
print(bool(10))
```
运行实例

实例
评估两个变量：

```
x = "Hello"
y = 10

print(bool(x))
print(bool(y))
```

运行实例

大多数值都为 True
如果有某种内容，则几乎所有值都将评估为 True。
除空字符串外，任何字符串均为 True。
除 0 外，任何数字均为 True。
除空列表外，任何列表、元组、集合和字典均为 True。
实例
下例将返回 True：

```
bool("abc")
bool(123)
bool(["apple", "cherry", "banana"])
```

运行实例

某些值为 False
实际上，除空值（例如 ()、[]、{}、""、数字 0 和值 None）外，没有多少值会被评估为 False。当然，值 False 的计算结果为 False。
实例
下例会返回 False：

```
bool(False)
bool(None)
bool(0)
bool("")
bool(())
bool([])
bool({})
```

运行实例
在这种情况下，一个值或对象的计算结果为 False，即如果对象由带有 __len__ 函数的类生成的，且该函数返回 0 或 False：
实例

```
class myclass():
  def __len__(self):
    return 0

myobj = myclass()
print(bool(myobj))
```

运行实例

函数可返回布尔
Python 还有很多返回布尔值的内置函数，例如 isinstance() 函数，该函数可用于确定对象是否具有某种数据类型：
实例
检查对象是否是整数：

```
x = 200
print(isinstance(x, int))
```

运行实例

Python 字符串
Python 运算符

Python 运算符
运算符用于对变量和值执行操作。
Python 在以下组中划分运算符：
算术运算符
赋值运算符
比较运算符
逻辑运算符
身份运算符
成员运算符
位运算符

Python 算术运算符
算术运算符与数值一起使用来执行常见的数学运算：

运算符	名称	实例
+	加	x + y
-	减	x - y
*	乘	x * y
/	除	x / y
%	取模	x % y
**	幂	x ** y
//	地板除（取整除）	x // y

Python 赋值运算符
赋值运算符用于为变量赋值：

运算符	实例	等同于
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3

*= x *= 3 x = x * 3
/= x /= 3 x = x / 3
%= x %= 3 x = x % 3
//= x //= 3 x = x // 3
**= x **= 3 x = x ** 3
&= x &= 3 x = x & 3
|= x |= 3 x = x | 3
^= x ^= 3 x = x ^ 3
>>= x >>= 3 x = x >> 3
<<= x <<= 3 x = x << 3

Python 比较运算符
比较运算符用于比较两个值：
运算符 名称 实例
== 等于 x == y
!= 不等于 x != y
> 大于 x > y
< 小于 x < y
>= 大于或等于 x >= y
<= 小于或等于 x <= y

Python 逻辑运算符
逻辑运算符用于组合条件语句：
运算符 描述 实例
and 如果两个语句都为真，则返回 True。 x > 3 and x < 10
or 如果其中一个语句为真，则返回 True。 x > 3 or x < 4
not 反转结果，如果结果为 true，则返回 False not(x > 3 and x < 10)

Python 身份运算符
身份运算符用于比较对象，不是比较它们是否相等，但如果它们实际上是同一个对象，则具有相同的内存位置：
运算符 描述 实例
is 如果两个变量是同一个对象，则返回 true。 x is y
is not 如果两个变量不是同一个对象，则返回 true。 x is not y

Python 成员运算符
成员资格运算符用于测试序列是否在对象中出现：
运算符 描述 实例
in 如果对象中存在具有指定值的序列，则返回 True。 x in y
not in 如果对象中不存在具有指定值的序列，则返回 True。 x not in y

Python 位运算符

位运算符用于比较（二进制）数字：
运算符 描述 实例
& AND 如果两个位均为 1，则将每个位设为 1。
| OR 如果两位中的一位为 1，则将每个位设为 1。
^ XOR 如果两个位中只有一位为 1，则将每个位设为 1。
~ NOT 反转所有位。
<< Zero fill left shift 通过从右侧推入零来向左移动，推掉最左边的位。
>> Signed right shift 通过从左侧推入最左边的位的副本向右移动，推掉最右边的位。

Python 布尔
Python 列表

Python 集合（数组）
Python 编程语言中有四种集合数据类型：
列表（List）是一种有序和可更改的集合。允许重复的成员。
元组（Tuple）是一种有序且不可更改的集合。允许重复的成员。
集合（Set）是一个无序和无索引的集合。没有重复的成员。
词典（Dictionary）是一个无序，可变和有索引的集合。没有重复的成员。
选择集合类型时，了解该类型的属性很有用。
为特定数据集选择正确的类型可能意味着保留含义，并且可能意味着提高效率或安全性。

列表
列表是一个有序且可更改的集合。在 Python 中，列表用方括号编写。
实例
创建列表：
```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```
运行实例

访问项目
您可以通过引用索引号来访问列表项：
实例
打印列表的第二项：
```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])
```
运行实例
负的索引

负索引表示从末尾开始，-1 表示最后一个项目，-2 表示倒数第二个项目，依此类推。

实例

打印列表的最后一项：

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

运行实例

索引范围

您可以通过指定范围的起点和终点来指定索引范围。

指定范围后，返回值将是包含指定项目的新列表。

实例

返回第三、第四、第五项：

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

运行实例

注释：搜索将从索引 2（包括）开始，到索引 5（不包括）结束。

请记住，第一项的索引为 0。

负索引的范围

如果要从列表末尾开始搜索，请指定负索引：

实例

此例将返回从索引 -4（包括）到索引 -1（排除）的项目：

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[-4:-1])
```

运行实例

更改项目值

如需更改特定项目的值，请引用索引号：

实例

更改第二项：

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

运行实例

遍历列表

您可以使用 for 循环遍历列表项：

实例

逐个打印列表中的所有项目：

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

运行实例

您将在 Python For 循环 这一章中学习有关 for 循环的更多知识。

检查项目是否存在
如需确定列表中是否存在指定的项，请使用 in 关键字：
实例
检查列表中是否存在 “apple”：
```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
  print("Yes, 'apple' is in the fruits list")
```
运行实例

列表长度
如需确定列表中有多少项，请使用 len() 方法：
实例
打印列表中的项目数：
```
thislist = ["apple", "banana", "cherry"]
print(len(thislist))
```
运行实例

添加项目
如需将项目添加到列表的末尾，请使用 append() 方法：
实例
使用 append() 方法追加项目：
```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```
运行实例
要在指定的索引处添加项目，请使用 insert() 方法：
实例
插入项目作为第二个位置：
```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```
运行实例

删除项目
有几种方法可以从列表中删除项目：
实例
remove() 方法删除指定的项目：
```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```
运行实例
实例

pop() 方法删除指定的索引（如果未指定索引，则删除最后一项）：

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

运行实例

实例

del 关键字删除指定的索引：

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

运行实例

实例

del 关键字也能完整地删除列表：

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

运行实例

实例

clear() 方法清空列表：

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

运行实例

复制列表

您只能通过键入 list2 = list1 来复制列表，因为：list2 将只是对 list1 的引用， 中所做的更改也将自动在 list2 中进行。

有一些方法可以进行复制，一种方法是使用内置的 List 方法 copy()。

实例

使用 copy() 方法来复制列表：

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

运行实例

制作副本的另一种方法是使用内建的方法 list()。

实例

使用 list() 方法复制列表：

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

运行实例

合并两个列表

在 Python 中，有几种方法可以连接或串联两个或多个列表。

最简单的方法之一是使用 + 运算符。

实例
合并两个列表：

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

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

运行实例
连接两个列表的另一种方法是将 list2 中的所有项一个接一个地追加到 list1 中：
实例
把 list2 追加到 list1 中：

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

for x in list2:
  list1.append(x)

print(list1)
```

运行实例
或者，您可以使用 extend() 方法，其目的是将一个列表中的元素添加到另一列表中：
实例
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)
```

运行实例

list() 构造函数
也可以使用 list() 构造函数创建一个新列表。
实例
使用 list() 构造函数创建列表：

```
thislist = list(("apple", "banana", "cherry")) # 请注意双括号
print(thislist)
```

运行实例

列表方法
Python 有一组可以在列表上使用的内建方法。

方法	描述
append()	在列表的末尾添加一个元素
clear()	删除列表中的所有元素

copy() 返回列表的副本
count() 返回具有指定值的元素数量。
extend() 将列表元素（或任何可迭代的元素）添加到当前列表的末尾
index() 返回具有指定值的第一个元素的索引
insert() 在指定位置添加元素
pop() 删除指定位置的元素
remove() 删除具有指定值的项目
reverse() 颠倒列表的顺序
sort() 对列表进行排序

Python 运算符
Python 元组

元组（Tuple）
元组是有序且不可更改的集合。在 Python 中，元组是用圆括号编写的。
实例
创建元组：
```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```
运行实例

访问元组项目
您可以通过引用方括号内的索引号来访问元组项目：
实例
打印元组中的第二个项目：
```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```
运行实例
负索引
负索引表示从末尾开始，-1 表示最后一个项目，-2 表示倒数第二个项目，依此类推。
实例
打印元组的最后一个项目：
```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```
运行实例
索引范围
您可以通过指定范围的起点和终点来指定索引范围。
指定范围后，返回值将是带有指定项目的新元组。
实例
返回第三、第四、第五个项目：

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

运行实例

注释：搜索将从索引 2（包括）开始，到索引 5（不包括）结束。

请记住，第一项的索引为 0。

负索引范围

如果要从元组的末尾开始搜索，请指定负索引：

实例

此例将返回从索引 -4（包括）到索引 -1（排除）的项目：

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi",
"melon", "mango")
print(thistuple[-4:-1])
```

运行实例

更改元组值

创建元组后，您将无法更改其值。元组是不可变的，或者也称为恒定的。

但是有一种解决方法。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。

实例

把元组转换为列表即可进行更改：

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

print(x)
```

运行实例

遍历元组

您可以使用 for 循环遍历元组项目。

实例

遍历项目并打印值：

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

运行实例

您将在 Python For 循环 这一章中学习有关 for 循环的更多知识。

检查项目是否存在

要确定元组中是否存在指定的项，请使用 in 关键字：

实例

检查元组中是否存在 "apple"：

```
thistuple = ("apple", "banana", "cherry")
```

```
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")
```
运行实例

元组长度
要确定元组有多少项，请使用 len() 方法：
实例
打印元组中的项目数量：
```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```
运行实例

添加项目
元组一旦创建，您就无法向其添加项目。元组是不可改变的。
实例
您无法向元组添加项目：
```
thistuple = ("apple", "banana", "cherry")
thistuple[3] = "orange" # 会引发错误
print(thistuple)
```
运行实例

创建有一个项目的元组
如需创建仅包含一个项目的元组，您必须在该项目后添加一个逗号，否则 Python 无法将变量识别为元组。
实例
单项元组，别忘了逗号：
```
thistuple = ("apple",)
print(type(thistuple))

#不是元组
thistuple = ("apple")
print(type(thistuple))
```
运行实例

删除项目
注释：您无法删除元组中的项目。
元组是不可更改的，因此您无法从中删除项目，但您可以完全删除元组：
实例
del 关键字可以完全删除元组：
```
thistuple = ("apple", "banana", "cherry")
del thistuple

print(thistuple) # 这会引发错误，因为元组已不存在。
```
运行实例

合并两个元组
如需连接两个或多个元组，您可以使用 + 运算符：
实例
合并这个元组：

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

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

运行实例

tuple() 构造函数
也可以使用 tuple() 构造函数来创建元组。
实例
使用 tuple() 方法来创建元组：

```
thistuple = tuple(("apple", "banana", "cherry")) # 请注意双括号
print(thistuple)
```

运行实例

元组方法
Python 提供两个可以在元组上使用的内建方法。

方法	描述
count()	返回元组中指定值出现的次数。
index()	在元组中搜索指定的值并返回它被找到的位置。

Python 列表
Python 集合

集合（Set）
集合是无序和无索引的集合。在 Python 中，集合用花括号编写。
实例
创建集合：

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

运行实例
注释：集合是无序的，因此您无法确定项目的显示顺序。

访问项目
您无法通过引用索引来访问 set 中的项目，因为 set 是无序的，项目

没有索引。
但是您可以使用 for 循环遍历 set 项目，或者使用 in 关键字查询集合中是否存在指定值。
实例
遍历集合，并打印值：

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)
```

运行实例
实例
检查 set 中是否存在 “banana”：

```
thisset = {"apple", "banana", "cherry"}

print("banana" in thisset)
```

运行实例

更改项目
集合一旦创建，您就无法更改项目，但是您可以添加新项目。

添加项目
要将一个项添加到集合，请使用 add() 方法。
要向集合中添加多个项目，请使用 update() 方法。
实例
使用 add() 方法向 set 添加项目：

```
thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)
```

运行实例
实例
使用 update() 方法将多个项添加到集合中：

```
thisset = {"apple", "banana", "cherry"}

thisset.update(["orange", "mango", "grapes"])

print(thisset)
```

运行实例

获取 Set 的长度
要确定集合中有多少项，请使用 len() 方法。
实例
获取集合中的项目数：

```
thisset = {"apple", "banana", "cherry"}

print(len(thisset))
```

运行实例

删除项目

要删除集合中的项目，请使用 remove() 或 discard() 方法。

实例

使用 remove() 方法来删除 “banana”：

```
thisset = {"apple", "banana", "cherry"}

thisset.remove("banana")

print(thisset)
```

运行实例

注释：如果要删除的项目不存在，则 remove() 将引发错误。

实例

使用 discard() 方法来删除 “banana”：

```
thisset = {"apple", "banana", "cherry"}

thisset.discard("banana")

print(thisset)
```

运行实例

注释：如果要删除的项目不存在，则 discard() 将引发错误。

您还可以使用 pop() 方法删除项目，但此方法将删除最后一项。请记住，set 是无序的，因此您不会知道被删除的是什么项目。

pop() 方法的返回值是被删除的项目。

实例

使用 pop() 方法删除最后一项：

```
thisset = {"apple", "banana", "cherry"}

x = thisset.pop()

print(x)

print(thisset)
```

运行实例

注释：集合是无序的，因此在使用 pop() 方法时，您不会知道删除的是哪个项目。

实例

clear() 方法清空集合：

```
thisset = {"apple", "banana", "cherry"}
```

```
thisset.clear()

print(thisset)
```

运行实例

实例

del 彻底删除集合：

```
thisset = {"apple", "banana", "cherry"}

del thisset

print(thisset)
```

运行实例

合并两个集合

在 Python 中，有几种方法可以连接两个或多个集合。

您可以使用 union() 方法返回包含两个集合中所有项目的新集合，也可以使用 update() 方法将一个集合中的所有项目插入另一个集合中：

实例

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)
```

运行实例

实例

update() 方法将 set2 中的项目插入 set1 中：

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

set1.update(set2)
print(set1)
```

运行实例

注释：union() 和 update() 都将排除任何重复项。

还有其他方法将两个集合连接起来，并且仅保留重复项，或者永远不保留重复项，请查看此页面底部的集合方法完整列表。

set() 构造函数

也可以使用 set() 构造函数来创建集合。

实例

使用 set() 构造函数来创建集合：

```
thisset = set(("apple", "banana", "cherry")) # 请留意这个双括号
print(thisset)
```

运行实例

Set 方法
Python 拥有一套能够在集合（set）上使用的内建方法。
方法 描述
add() 向集合添加元素。
clear() 删除集合中的所有元素。
copy() 返回集合的副本。
difference() 返回包含两个或更多集合之间差异的集合。
difference_update() 删除此集合中也包含在另一个指定集合中的项目。
discard() 删除指定项目。
intersection() 返回为两个其他集合的交集的集合。
intersection_update() 删除此集合中不存在于其他指定集合中的项目。
isdisjoint() 返回两个集合是否有交集。
issubset() 返回另一个集合是否包含此集合。
issuperset() 返回此集合是否包含另一个集合。
pop() 从集合中删除一个元素。
remove() 删除指定元素。
symmetric_difference() 返回具有两组集合的对称差集的集合。
symmetric_difference_update() 插入此集合和另一个集合的对称差集。
union() 返回包含集合并集的集合。
update() 用此集合和其他集合的并集来更新集合。

Python 元组
Python 字典

字典（Dictionary）
字典是一个无序、可变和有索引的集合。在 Python 中，字典用花括号编写，拥有键和值。
实例
创建并打印字典：

```
thisdict = {
  "brand": "Porsche",
  "model": "911",
  "year": 1963
}
print(thisdict)
```

运行实例

访问项目
您可以通过在方括号内引用其键名来访问字典的项目：
实例
获取 "model" 键的值：
```
x = thisdict["model"]
```
运行实例
还有一个名为 get() 的方法会给你相同的结果：
实例
获取 "model" 键的值：
```
x = thisdict.get("model")
```
运行实例

更改值
您可以通过引用其键名来更改特定项的值：
实例
把 "year" 改为 2019：
```
thisdict = {
  "brand": "Porsche",
  "model": "911",
  "year": 1963
}
thisdict["year"] = 2019
```
运行实例

遍历字典
您可以使用 for 循环遍历字典。
循环遍历字典时，返回值是字典的键，但也有返回值的方法。
实例
逐个打印字典中的所有键名：
```
for x in thisdict:
  print(x)
```
运行实例
实例
逐个打印字典中的所有值：
```
for x in thisdict:
  print(thisdict[x])
```
运行实例
实例
您还可以使用 values() 函数返回字典的值：
```
for x in thisdict.values():
  print(x)
```
运行实例
实例

通过使用 items() 函数遍历键和值：

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

运行实例

检查键是否存在

要确定字典中是否存在指定的键，请使用 in 关键字：

实例

检查字典中是否存在 "model"：

```
thisdict = {
  "brand": "Porsche",
  "model": "911",
  "year": 1963
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

运行实例

字典长度

要确定字典有多少项目（键值对），请使用 len() 方法。

实例

打印字典中的项目数：

```
print(len(thisdict))
```

运行实例

添加项目

通过使用新的索引键并为其赋值，可以将项目添加到字典中：

实例

```
thisdict = {
  "brand": "Porsche",
  "model": "911",
  "year": 1963
}
thisdict["color"] = "red"
print(thisdict)
```

运行实例

删除项目

有几种方法可以从字典中删除项目：

实例

pop() 方法删除具有指定键名的项：

```
thisdict = {
  "brand": "Porsche",
```

```
  "model": "911",
  "year": 1963
}
thisdict.pop("model")
print(thisdict)
```

运行实例

实例

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

```
thisdict = {
  "brand": "Porsche",
  "model": "911",
  "year": 1963
}
thisdict.popitem()
print(thisdict)
```

运行实例

实例

del 关键字删除具有指定键名的项目：

```
thisdict = {
  "brand": "Porsche",
  "model": "911",
  "year": 1963
}
del thisdict["model"]
print(thisdict)
```

运行实例

实例

del 关键字也可以完全删除字典：

```
thisdict = {
  "brand": "Porsche",
  "model": "911",
  "year": 1963
}
del thisdict

print(thisdict) #this 会导致错误，因为 "thisdict" 不再存在。
```

运行实例

实例

clear() 关键字清空字典：

```
thisdict = {
  "brand": "Porsche",
  "model": "911",
  "year": 1963
```

```
}
thisdict.clear()
print(thisdict)
```

运行实例

复制字典

您不能通过键入 dict2 = dict1 来复制字典，因为：dict2 只是对 dict1 的引用，而 中的更改也将自动在 dict2 中进行。

有一些方法可以进行复制，一种方法是使用内建的字典方法 copy()。

实例

使用 copy() 方法来复制字典：

```
thisdict = {
  "brand": "Porsche",
  "model": "911",
  "year": 1963
}
mydict = thisdict.copy()
print(mydict)
```

运行实例

制作副本的另一种方法是使用内建方法 dict()。

实例

使用 dict() 方法创建字典的副本：

```
thisdict = {
  "brand": "Porsche",
  "model": "911",
  "year": 1963
}
mydict = dict(thisdict)
print(mydict)
```

运行实例

嵌套字典

词典也可以包含许多词典，这被称为嵌套词典。

实例

创建包含三个字典的字典：

```
myfamily = {
  "child1" : {
    "name" : "Phoebe Adele",
    "year" : 2002
  },
  "child2" : {
    "name" : "Jennifer Katharine",
    "year" : 1996
  },
```

```
  "child3" : {
    "name" : "Rory John",
    "year" : 1999
  }
}
```

运行实例

或者，如果您想嵌套三个已经作为字典存在的字典：

实例

创建三个字典，然后创建一个包含其他三个字典的字典：

```
child1 = {
  "name" : "Phoebe Adele",
  "year" : 2002
}
child2 = {
  "name" : "Jennifer Katharine",
  "year" : 1996
}
child3 = {
  "name" : "Rory John",
  "year" : 1999
}

myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}
```

运行实例

dict() 构造函数

也可以使用 dict() 构造函数创建新的字典：

实例

```
thisdict = dict(brand="Porsche", model="911", year=1963)
# 请注意，关键字不是字符串字面量
# 请注意，使用了等号而不是冒号来赋值
print(thisdict)
```

运行实例

字典方法

Python 提供一组可以在字典上使用的内建方法。

方法	描述
clear()	删除字典中的所有元素
copy()	返回字典的副本
fromkeys()	返回拥有指定键和值的字典

get() 返回指定键的值
items() 返回包含每个键值对的元组的列表
keys() 返回包含字典键的列表
pop() 删除拥有指定键的元素
popitem() 删除最后插入的键值对
setdefault() 返回指定键的值。如果该键不存在，则插入具有指定值的键。
update() 使用指定的键值对字典进行更新
values() 返回字典中所有值的列表

Python 集合
Python If Else

Python 条件和 If 语句
Python 支持来自数学的常用逻辑条件：
等于：a == b
不等于：a != b
小于：a < b
小于等于：a <= b
大于：a > b
大于等于：a >= b
这些条件能够以多种方式使用，最常见的是“if 语句”和循环。
if 语句使用 if 关键词来写。
实例
If 语句：

```
a = 66
b = 200
if b > a:
  print("b is greater than a")
```

运行实例
在这个例子中，我们使用了两个变量，a 和 b，作为 if 语句的一部分，它们用于测试 b 是否大于 a。因为 a 是 66，而 b 是 200，我们知道 200 大于 66，所以我们将“b 大于 a”打印到屏幕。

缩进
Python 依赖缩进，使用空格来定义代码中的范围。其他编程语言通常使用花括号来实现此目的。
实例
没有缩进的 If 语句（会引发错误）：

```
a = 66
b = 200
```

```
if b > a:
print("b is greater than a") # 会报错
```

运行实例

Elif

elif 关键字是 python 对“如果之前的条件不正确，那么试试这个条件”的表达方式。

实例

```
a = 66
b = 66
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

运行实例

在这个例子中，a 等于 b，所以第一个条件不成立，但 elif 条件为 true，所以我们打印屏幕“a 和 b 相等”。

Else

else 关键字捕获未被之前的条件捕获的任何内容。

实例

```
a = 200
b = 66
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

运行实例

在这个例子中，a 大于 b，所以第一个条件不成立，elif 条件也不成立，所以我们转到 条件并打印到屏幕“a 大于 b”。

您也可以使用没有 elif 的 else：

实例

```
a = 200
b = 66
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

运行实例

简写 If

如果只有一条语句要执行，则可以将其与 if 语句放在同一行。

实例
单行 if 语句：

```
a = 200
b = 66
if a > b: print("a is greater than b")
```

运行实例

简写 If ... Else
如果只有两条语句要执行，一条用于 if，另一条用于 else，则可以将它们全部放在同一行：
实例
单行 if else 语句：

```
a = 200
b = 66
print("A") if a > b else print("B")
```

运行实例
您还可以在同一行上使用多个 else 语句：
实例
单行 if else 语句，有三个条件：

```
a = 200
b = 66
print("A") if a > b else print("=") if a == b else print("B")
```

运行实例

And
and 关键字是一个逻辑运算符，用于组合条件语句：
实例
测试 a 是否大于 b，且 c 是否大于 a：

```
a = 200
b = 66
c = 500
if a > b and c > a:
  print("Both conditions are True")
```

运行实例

Or
or 关键字也是逻辑运算符，用于组合条件语句：
实例
测试 a 是否大于 b，或者 a 是否大于 c：

```
a = 200
b = 66
c = 500
if a > b or a > c:
  print("At least one of the conditions is True")
```

运行实例

嵌套 If
您可以在 if 语句中包含 if 语句，这称为嵌套 if 语句。
实例
```
x = 52

if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
```
运行实例

pass 语句
if 语句不能为空，但是如果您处于某种原因写了无内容的 if 语句，请使用 pass 语句来避免错误。
实例
```
a = 66
b = 200

if b > a:
  pass
```
运行实例

Python 字典
Python While 循环

Python 循环
Python 有两个原始的循环命令：
while 循环
for 循环

while 循环
如果使用 while 循环，只要条件为真，我们就可以执行一组语句。
实例
只要 i 大于 7，打印 i：
```
i = 1
while i < 7:
  print(i)
```

```
  i += 1
```

运行实例

注释：请记得递增 i，否则循环会永远继续。

while 循环需要准备好相关的变量。在这个实例中，我们需要定义一个索引变量 i，我们将其设置为 1。

break 语句

如果使用 break 语句，即使 while 条件为真，我们也可以停止循环：

实例

在 i 等于 3 时退出循环：

```
i = 1
while i < 7:
  print(i)
  if i == 3:
    break
  i += 1
```

运行实例

continue 语句

如果使用 continue 语句，我们可以停止当前的迭代，并继续下一个：

实例

如果 i 等于 3，则继续下一个迭代：

```
i = 0
while i < 7:
  i += 1
  if i == 3:
    continue
  print(i)
```

运行实例

else 语句

通过使用 else 语句，当条件不再成立时，我们可以运行一次代码块：

实例

条件为假时打印一条消息：

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

运行实例

Python If Else

Python For 循环

Python For 循环

for 循环用于迭代序列（即列表，元组，字典，集合或字符串）。

这与其他编程语言中的 for 关键字不太相似，而是更像其他面向对象编程语言中的迭代器方法。

通过使用 for 循环，我们可以为列表、元组、集合中的每个项目等执行一组语句。

实例

打印 fruits 列表中的每种水果：

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

运行实例

提示：for 循环不需要预先设置索引变量。

循环遍历字符串

甚至连字符串都是可迭代的对象，它们包含一系列的字符：

实例

循环遍历单词 "banana" 中的字母：

```
for x in "banana":
  print(x)
```

运行实例

break 语句

通过使用 break 语句，我们可以在循环遍历所有项目之前停止循环：

实例

如果 x 是 "banana"，则退出循环：

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

运行实例

实例

当 x 为 "banana" 时退出循环，但这次在打印之前中断：

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

运行实例

continue 语句
通过使用 continue 语句，我们可以停止循环的当前迭代，并继续下一个：
实例
不打印香蕉：
```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```
运行实例

range() 函数
如需循环一组代码指定的次数，我们可以使用 range() 函数，
range() 函数返回一个数字序列，默认情况下从 0 开始，并递增 1（默认地），并以指定的数字结束。
实例
使用 range() 函数：
```
for x in range(10):
  print(x)
```
运行实例
注意：range(10) 不是 0 到 10 的值，而是值 0 到 9。
range() 函数默认 0 为起始值，不过可以通过添加参数来指定起始值：range(3, 10)，这意味着值为 3 到 10（但不包括 10）：
实例
使用起始参数：
```
for x in range(3, 10):
  print(x)
```
运行实例
range() 函数默认将序列递增 1，但是可以通过添加第三个参数来指定增量值：range(2, 30, 3)：
实例
使用 3 递增序列（默认值为 1）：
```
for x in range(3, 50, 6):
  print(x)
```
运行实例

For 循环中的 Else
for 循环中的 else 关键字指定循环结束时要执行的代码块：
实例
打印 0 到 9 的所有数字，并在循环结束时打印一条消息：
```
for x in range(10):
  print(x)
```

```
else:
  print("Finally finished!")
```
运行实例

嵌套循环
嵌套循环是循环内的循环。
“外循环”每迭代一次，“内循环”将执行一次：
实例
打印每个水果的每个形容词：
```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)
```
运行实例

pass 语句
for 语句不能为空，但是如果您处于某种原因写了无内容的 for 语句，请使用 pass 语句来避免错误。
实例
```
for x in [0, 1, 2]:
  pass
```
运行实例

Python While 循环
Python 函数

函数是一种仅在调用时运行的代码块。
您可以将数据（称为参数）传递到函数中。
函数可以把数据作为结果返回。

创建函数
在 Python 中，使用 def 关键字定义函数：
实例
```
def my_function():
  print("Hello from a function")
```

调用函数
如需调用函数，请使用函数名称后跟括号：
实例

```
def my_function():
  print("Hello from a function")

my_function()
```

运行实例

参数

信息可以作为参数传递给函数。

参数在函数名后的括号内指定。您可以根据需要添加任意数量的参数，只需用逗号分隔即可。

下面的例子有一个带参数（fname）的函数。当调用此函数时，我们传递一个名字，在函数内部使用它来打印全名：

实例

```
def my_function(fname):
  print(fname + " Gates")

my_function("Rory John")
my_function("Jennifer Katharine")
my_function("Phoebe Adele")
```

运行实例

默认参数值

下面的例子展示如何使用默认参数值。

如果我们调用了不带参数的函数，则使用默认值：

实例

```
def my_function(country = "China"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

运行实例

以 List 传参

您发送到函数的参数可以是任何数据类型（字符串、数字、列表、字典等），并且在函数内其将被视为相同数据类型。

例如，如果您将 List 作为参数发送，它到达函数时仍将是 List（列表）：

实例

```
def my_function(food):
  for x in food:
    print(x)
```

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

my_function(fruits)
```

运行实例

返回值

如需使函数返回值，请使用 return 语句：

实例

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

运行实例

关键字参数

您还可以使用 key = value 语法发送参数。

参数的顺序无关紧要。

实例

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Phoebe", child2 = "Jennifer", child3 =
"Rory")
```

运行实例

在 Python 文档中，“关键字参数”一词通常简称为 kwargs。

任意参数

如果您不知道将传递给您的函数多少个参数，请在函数定义的参数名称前添加 *。

这样，函数将接收一个参数元组，并可以相应地访问各项：

实例

如果参数数目未知，请在参数名称前添加 *：

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Phoebe", "Jennifer", "Rory")
```

运行实例

pass 语句

函数定义不能为空，但是如果您出于某种原因写了无内容的函数定义，请使用 pass 语句来避免错误。

实例

```
def myfunction:
  pass
```

运行实例

递归

Python 也接受函数递归，这意味着定义的函数能够调用自身。

递归是一种常见的数学和编程概念。它意味着函数调用自身。这样做的好处是可以循环访问数据以达成结果。

开发人员应该非常小心递归，因为它可以很容易地编写一个永不终止的，或者使用过量内存或处理器能力的函数。但是，在被正确编写后，递归可能是一种非常有效且数学上优雅的编程方法。

在这个例子中，tri_recursion() 是我们定义为调用自身 ("recurse") 的函数。我们使用 k 变量作为数据，每次递归时递减（-1）。当条件不大于 0 时（比如当它为 0 时），递归结束。

对于新的开发人员来说，可能需要一些时间来搞清楚其工作原理，最好的方法是测试并修改它。

实例

递归的例子：

```
def tri_recursion(k):
  if(k>0):
    result = k+tri_recursion(k-1)
    print(result)
  else:
    result = 0
  return result

print("\n\nRecursion Example Results")
tri_recursion(6)
```

运行实例

Python For 循环

Python Lambda

lambda 函数是一种小的匿名函数。

lambda 函数可接受任意数量的参数，但只能有一个表达式。

语法

```
lambda arguments : expression
```

执行表达式并返回结果：

实例

一个 lambda 函数，它把作为参数传入的数字加 10，然后打印结果：

```
x = lambda a : a + 10
print(x(5))
```

运行实例

lambda 函数可接受任意数量的参数：

实例

一个 lambda 函数，它把参数 a 与参数 b 相乘并打印结果：

```
x = lambda a, b : a * b
print(x(5, 6))
```

运行实例

实例

一个 lambda 函数，它把参数 a、b 和 c 相加并打印结果：

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

运行实例

为何使用 Lambda 函数？

当您把 lambda 用作另一个函数内的匿名函数时，会更好地展现 lambda 的强大能力。

假设您有一个带一个参数的函数定义，并且该参数将乘以未知数字：

```
def myfunc(n):
  return lambda a : a * n
```

使用该函数定义来创建一个总是使所发送数字加倍的函数：

实例

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```

运行实例

或者，使用相同的函数定义来创建一个总是使您发送的数字增加三倍的函数：

实例

```
def myfunc(n):
  return lambda a : a * n

mytripler = myfunc(3)

print(mytripler(11))
```

运行实例

或者，在同一程序中使用相同的函数定义来生成两个函数：

实例

```
def myfunc(n):
  return lambda a : a * n
```

```
mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

运行实例

如果在短时间内需要匿名函数，请使用 lambda 函数。

Python 函数
Python 数组

请注意，Python 没有内置对数组的支持，但可以使用 Python 列表代替。

数组

数组用于在单个变量中存储多个值：

实例

创建一个包含汽车品牌的数组：

```
cars = ["Porsche", "Volvo", "BMW"]
```

运行实例

什么是数组？

数组是一种特殊变量，能够一次包含多个值。

如果您有一个项目列表（例如，汽车品牌列表），将牌子存储在单个变量中可能如下所示：

```
car1 = "Porsche"
car2 = "Volvo"
car3 = "BMW"
```

但是，如果您想遍历这些品牌并找到特定的汽车品牌怎么办？如果不是 3 辆车，而是 300 辆怎么办？

解决方案是数组！

数组可以在单个名称下保存多个值，您可以通过引用索引号来访问这些值。

访问数组元素

通过索引号来引用数组元素。

实例

获取首个数组项目的值：

```
x = cars[0]
```

运行实例

实例
修改首个数组项目的值：
```
cars[0] = "Audi"
```
运行实例

数组长度
使用 len() 方法来返回数组的长度（数组中的元素数量）。
实例
返回 cars 数组中的元素数量：
```
x = len(cars)
```
运行实例
注释：数组长度总是比最高的数组索引大一个。

循环数组元素
您可以使用 for in 循环遍历数组的所有元素。
实例
打印 cars 数组中的每个项目：
```
for x in cars:
  print(x)
```
运行实例

添加数组元素
您可以使用 append() 方法把元素添加到数组中。
实例
向 cars 数组再添加一个元素：
```
cars.append("Audi")
```
运行实例

删除数组元素
您可以使用 pop() 方法从数组中删除元素。
实例
删除 cars 数组的第二个元素：
```
cars.pop(1)
```
运行实例
您也可以使用 remove() 方法从数组中删除元素。
实例
删除值为 "Volvo" 的元素：
```
cars.remove("Volvo")
```
运行实例
注释：列表的 remove() 方法仅删除首次出现的指定值。

数组方法
Python 提供一组可以在列表或数组上使用的内建方法。
方法 描述

append() 在列表的末尾添加一个元素
clear() 删除列表中的所有元素
copy() 返回列表的副本
count() 返回具有指定值的元素数量。
extend() 将列表元素（或任何可迭代的元素）添加到当前列表的末尾
index() 返回具有指定值的第一个元素的索引
insert() 在指定位置添加元素
pop() 删除指定位置的元素
remove() 删除具有指定值的项目
reverse() 颠倒列表的顺序
sort() 对列表进行排序
注释：Python 没有内置对数组的的支持，但可以使用 Python 列表代替。

Python Lambda
Python 类/对象

Python 类/对象
Python 是一种面向对象的编程语言。
Python 中的几乎所有东西都是对象，拥有属性和方法。
类（Class）类似对象构造函数，或者是用于创建对象的“蓝图”。

创建类
如需创建类，请使用 class 关键字：
实例
使用名为 x 的属性，创建一个名为 MyClass 的类：

```
class MyClass:
  x = 5
```

运行实例

创建对象
现在我们可以使用名为 myClass 的类来创建对象：
实例
创建一个名为 p1 的对象，并打印 x 的值：

```
p1 = MyClass()
print(p1.x)
```

运行实例

__init__() 函数
上面的例子是最简单形式的类和对象，在实际应用程序中并不真正有用。
要理解类的含义，我们必须先了解内置的 __init__() 函数。

所有类都有一个名为 __init__() 的函数，它始终在启动类时执行。
使用 __init__() 函数将值赋给对象属性，或者在创建对象时需要执行的其他操作：
实例
创建名为 Person 的类，使用 __init__() 函数为 name 和 age 赋值：

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("Bill", 63)

print(p1.name)
print(p1.age)
```

运行实例
注释：每次使用类创建新对象时，都会自动调用 __init__() 函数。

对象方法
对象也可以包含方法。对象中的方法是属于该对象的函数。
让我们在 Person 类中创建方法：
实例
插入一个打印问候语的函数，并在 p1 对象上执行它：

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("Bill", 63)
p1.myfunc()
```

运行实例
提示：self 参数是对类的当前实例的引用，用于访问属于该类的变量。

self 参数
self 参数是对类的当前实例的引用，用于访问属于该类的变量。
它不必被命名为 self，您可以随意调用它，但它必须是类中任意函数的首个参数：
实例
使用单词 mysillyobject 和 abc 代替 self：

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
```

```
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("Bill", 63)
p1.myfunc()
```

运行实例

修改对象属性

您可以这样修改对象的属性：

实例

把 p1 的年龄设置为 40：

```
p1.age = 40
```

运行实例

删除对象属性

您可以使用 del 关键字删除对象的属性：

实例

删除 p1 对象的 age 属性：

```
del p1.age
```

运行实例

删除对象

使用 del 关键字删除对象：

实例

删除 p1 对象：

```
del p1
```

运行实例

pass 语句

类定义不能为空，但是如果您处于某种原因写了无内容的类定义语句，请使用 pass 语句来避免错误。

实例

```
class Person:
  pass
```

运行实例

Python 数组

Python 继承

Python 继承
继承允许我们定义继承另一个类的所有方法和属性的类。
父类是继承的类，也称为基类。
子类是从另一个类继承的类，也称为派生类。

创建父类
任何类都可以是父类，因此语法与创建任何其他类相同：
实例
创建一个名为 Person 的类，其中包含 firstname 和 lastname 属性以及 printname 方法：

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

# 使用 Person 来创建对象，然后执行 printname 方法：

x = Person("Bill", "Gates")
x.printname()
```

运行实例

创建子类
要创建从其他类继承功能的类，请在创建子类时将父类作为参数发送：
实例
创建一个名为 Student 的类，它将从 Person 类继承属性和方法：

```
class Student(Person):
  pass
```

注释：如果您不想向该类添加任何其他属性或方法，请使用 pass 关键字。
现在，Student 类拥有与 Person 类相同的属性和方法。
实例
使用 Student 类创建一个对象，然后执行 printname 方法：

```
x = Student("Elon", "Musk")
x.printname()
```

运行实例

添加 __init__() 函数
到目前为止，我们已经创建了一个子类，它继承了父类的属性和方法。
我们想要把 __init__() 函数添加到子类（而不是 pass 关键字）。
注释：每次使用类创建新对象时，都会自动调用 __init__() 函数。
实例

为 Student 类添加 __init__() 函数：

```
class Student(Person):
  def __init__(self, fname, lname):
    # 添加属性等
```

当您添加 __init__() 函数时，子类将不再继承父的 __init__() 函数。

注释：子的 __init__() 函数会覆盖对父的 __init__() 函数的继承。

如需保持父的 __init__() 函数的继承，请添加对父的 __init__() 函数的调用：

实例

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

运行实例

现在，我们已经成功添加了 __init__() 函数，并保留了父类的继承，我们准备好在 __init__() 函数中添加功能了。

使用 super() 函数

Python 还有一个 super() 函数，它会使子类从其父继承所有方法和属性：

实例

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
```

运行实例

通过使用 super() 函数，您不必使用父元素的名称，它将自动从其父元素继承方法和属性。

添加属性

实例

把名为 graduationyear 的属性添加到 Student 类：

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019
```

运行实例

在这例子中，2019 年应该是一个变量，并在创建 student 对象时传递到 Student 类。为此，请在 __init__() 函数中添加另一个参数：

实例

添加 year 参数，并在创建对象时传递正确的年份：

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year
```

```
x = Student("Elon", "Musk", 2019)
```
运行实例

添加方法
实例
把名为 welcome 的方法添加到 Student 类：
```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
```
运行实例
提示：如果您在子类中添加一个与父类中的函数同名的方法，则将覆盖父方法的继承。

Python 类/对象
Python 迭代

Python 迭代器
迭代器是一种对象，该对象包含值的可计数数字。
迭代器是可迭代的对象，这意味着您可以遍历所有值。
从技术上讲，在 Python 中，迭代器是实现迭代器协议的对象，它包含方法 __iter__() 和 。

迭代器 VS 可迭代对象（Iterable）
列表、元组、字典和集合都是可迭代的对象。它们是可迭代的容器，您可以从中获取迭代器（Iterator）。
所有这些对象都有用于获取迭代器的 iter() 方法：
实例
从元组返回一个迭代器，并打印每个值：
```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```
运行实例

甚至连字符串都是可迭代的对象，并且可以返回迭代器：
实例
字符串也是可迭代的对象，包含一系列字符：

```
mystr = "banana"
myit = iter(mystr)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

运行实例

遍历迭代器
我们也可以使用 for 循环遍历可迭代对象：
实例
迭代元组的值：

```
mytuple = ("apple", "banana", "cherry")

for x in mytuple:
  print(x)
```

运行实例
实例
迭代字符串中的字符：

```
mystr = "banana"

for x in mystr:
  print(x)
```

运行实例
提示：for 循环实际上创建了一个迭代器对象，并为每个循环执行 next() 方法。

创建迭代器
要把对象/类创建为迭代器，必须为对象实现 __iter__() 和 __next__() 方法。
正如您在 Python 类/对象 一章中学到的，所有类都有名为 __init__() 的函数，它允许您在创建对象时进行一些初始化。
__iter__() 方法的作用相似，您可以执行操作（初始化等），但必须始终返回迭代器对象本身。
__next__() 方法也允许您执行操作，并且必须返回序列中的下一个项目。
实例
创建一个返回数字的迭代器，从 1 开始，每个序列将增加 1（返回 1、

2、3、4、5 等）：

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

运行实例

StopIteration

如果你有足够的 next() 语句，或者在 for 循环中使用，则上面的例子将永远进行下去。

为了防止迭代永远进行，我们可以使用 StopIteration 语句。

在 __next__() 方法中，如果迭代完成指定的次数，我们可以添加一个终止条件来引发错误：

实例

在 20 个迭代之后停止：

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)
```

```
for x in myiter:
  print(x)
```

运行实例

Python 继承
Python 作用域

变量仅在创建区域内可用。这称为作用域。

局部作用域
在函数内部创建的变量属于该函数的局部作用域，并且只能在该函数内部使用。
实例
在函数内部创建的变量在该函数内部可用：

```
def myfunc():
  x = 100
  print(x)

myfunc()
```

运行实例
函数内部的函数
如上例中所示，变量 x 在函数外部不可用，但对于函数内部的任何函数均可用：
实例
能够从函数内的一个函数访问局部变量：

```
def myfunc():
  x = 100
  def myinnerfunc():
    print(x)
  myinnerfunc()

myfunc()
```

运行实例

全局作用域
在 Python 代码主体中创建的变量是全局变量，属于全局作用域。
全局变量在任何范围（全局和局部）中可用。
实例
在函数外部创建的变量是全局变量，任何人都可以使用：

```
x = 100
```

```
def myfunc():
  print(x)

myfunc()

print(x)
```

运行实例

命名变量

如果在函数内部和外部操作同名变量，Python 会将它们视为两个单独的变量，一个在全局范围内可用（在函数外部），而一个在局部范围内可用（在函数内部）：

实例

该函数将打印局部变量 x，然后代码还会打印全局变量 x：

```
x = 100

def myfunc():
  x = 200
  print(x)

myfunc()

print(x)
```

运行实例

Global 关键字

如果您需要创建一个全局变量，但被卡在本地作用域内，则可以使用 global 关键字。

global 关键字使变量成为全局变量。

实例

如果使用 global 关键字，则该变量属于全局范围：

```
def myfunc():
  global x
  x = 100

myfunc()

print(x)
```

运行实例

另外，如果要在函数内部更改全局变量，也请使用 global 关键字。

实例

要在函数内部更改全局变量的值，请使用 global 关键字引用该变量：

```
x = 100
```

```
def myfunc():
  global x
  x = 200

myfunc()

print(x)
```

运行实例

Python 迭代
Python 模块

什么是模块?
请思考与代码库类似的模块。
模块是包含一组函数的文件，希望在应用程序中引用。

创建模块
如需创建模块，只需将所需代码保存在文件扩展名为 .py 的文件中：
实例
在名为 mymodule.py 的文件中保存代码：

```
def greeting(name):
  print("Hello, " + name)
```

使用模块
现在，我们就可以用 import 语句来使用我们刚刚创建的模块：
实例
导入名为 mymodule 的模块，并调用 greeting 函数：

```
import mymodule

mymodule.greeting("Bill")
```

运行实例
注释：如果使用模块中的函数时，请使用以下语法：

```
module_name.function_name
```

模块中的变量
模块可以包含已经描述的函数，但也可以包含各种类型的变量（数组、字典、对象等）：
实例
在文件 mymodule.py 中保存代码：

```
person1 = {
  "name": "Bill",
```

```
  "age": 63,
  "country": "USA"
}
```

实例

导入名为 mymodule 的模块，并访问 person1 字典：

```
import mymodule

a = mymodule.person1["age"]
print(a)
```

运行实例

为模块命名

您可以随意对模块文件命名，但是文件扩展名必须是 .py。

重命名模块

您可以在导入模块时使用 as 关键字创建别名：

实例

为 mymodule 创建别名 mx：

```
import mymodule as mx

a = mx.person1["age"]
print(a)
```

运行实例

内建模块

Python 中有几个内建模块，您可以随时导入。

实例

导入并使用 platform 模块：

```
import platform

x = platform.system()
print(x)
```

运行实例

使用 dir() 函数

有一个内置函数可以列出模块中的所有函数名（或变量名）。dir() 函数：

实例

列出属于 platform 模块的所有已定义名称：

```
import platform

x = dir(platform)
print(x)
```

运行实例

注释：dir() 函数可用于所有模块，也可用于您自己创建的模块。

从模块导入
您可以使用 from 关键字选择仅从模块导入部件。
实例
名为 mymodule 的模块拥有一个函数和一个字典：

```
def greeting(name):
  print("Hello, " + name)

person1 = {
  "name": "Bill",
  "age": 63,
  "country": "USA"
}
```

实例
仅从模块导入 person1 字典：

```
from mymodule import person1

print (person1["age"])
```

运行实例
提示：在使用 from 关键字导入时，请勿在引用模块中的元素时使用模块名称。示例：person1["age"]，而不是 mymodule.person1["age"]。

Python 作用域
Python 日期

Python 日期
Python 中的日期不是其自身的数据类型，但是我们可以导入名为 datetime 的模块，把日期视作日期对象进行处理。
实例
导入 datetime 模块并显示当前日期：

```
import datetime

x = datetime.datetime.now()
print(x)
```

运行实例

日期输出
如果我们执行上面的代码，结果将是：

```
2019-08-14 12:52:55.817273
```

日期包含年、月、日、小时、分钟、秒和微秒。

datetime 模块有许多方法可以返回有关日期对象的信息。
以下是一些例子，您将在本章稍后详细学习它们：
实例
返回 weekday 的名称和年份：

```
import datetime

x = datetime.datetime.now()

print(x.year)
print(x.strftime("%A"))
```

运行实例

创建日期对象
如需创建日期，我们可以使用 datetime 模块的 datetime() 类（构造函数）。
datetime() 类需要三个参数来创建日期：年、月、日。
实例
创建日期对象：

```
import datetime

x = datetime.datetime(2020, 5, 17)

print(x)
```

运行实例
datetime() 类还接受时间和时区（小时、分钟、秒、微秒、tzone）的参数，不过它们是可选的，默认值为 ，（时区默认为 None）。

strftime() 方法
datetime 对象拥有把日期对象格式化为可读字符串的方法。
该方法称为 strftime()，并使用一个 format 参数来指定返回字符串的格式：
实例
显示月份的名称：

```
import datetime

x = datetime.datetime(2019, 10, 1)

print(x.strftime("%B"))
```

运行实例
所有合法格式代码的参考：
指令 描述 实例
%a Weekday，短版本 Wed
%A Weekday，完整版本 Wednesday
%w Weekday，数字 0-6，0 为周日 3

%d 日，数字 01-31 31
%b 月名称，短版本 Dec
%B 月名称，完整版本 December
%m 月，数字 01-12 12
%y 年，短版本，无世纪 18
%Y 年，完整版本 2018
%H 小时，00-23 17
%I 小时，00-12 05
%p AM/PM PM
%M 分，00-59 41
%S 秒，00-59 08
%f 微妙，000000-999999 548513
%z UTC 偏移 +0100
%Z 时区 CST
%j 天数，001-366 365
%U 周数，每周的第一天是周日，00-53 52
%W 周数，每周的第一天是周一，00-53 52
%c 日期和时间的本地版本 Mon Dec 31 17:41:00 2018
%x 日期的本地版本 12/31/18
%X 时间的本地版本 17:41:00
%% A % character %

Python 模块
Python JSON

JSON 是用于存储和交换数据的语法。
JSON 是用 JavaScript 对象表示法（JavaScript object notation）编写的文本。

Python 中的 JSON
Python 有一个名为 json 的内置包，可用于处理 JSON 数据。
实例
导入 json 模块：
```
import json
```

解析 JSON - 把 JSON 转换为 Python
若有 JSON 字符串，则可以使用 json.loads() 方法对其进行解析。
结果将是 Python 字典。
实例
把 JSON 转换为 Python：
```
import json
```

```
# 一些 JSON:
x =  '{ "name":"Bill", "age":63, "city":"Seatle"}'

# 解析 x:
y = json.loads(x)

# 结果是 Python 字典：
print(y["age"])
```

运行实例

把 Python 转换为 JSON

若有 Python 对象，则可以使用 json.dumps() 方法将其转换为 JSON 字符串。

实例

把 Python 转换为 JSON：

```
import json

# Python 对象（字典）：
x = {
  "name": "Bill",
  "age": 63,
  "city": "Seatle"
}

# 转换为 JSON：
y = json.dumps(x)

# 结果是 JSON 字符串：
print(y)
```

运行实例

您可以把以下类型的 Python 对象转换为 JSON 字符串：

```
dict
list
tuple
string
int
float
True
False
None
```

实例

将 Python 对象转换为 JSON 字符串，并打印值：

```
import json
```

```
print(json.dumps({"name": "Bill", "age": 63}))
print(json.dumps(["apple", "bananas"]))
print(json.dumps(("apple", "bananas")))
print(json.dumps("hello"))
print(json.dumps(42))
print(json.dumps(31.76))
print(json.dumps(True))
print(json.dumps(False))
print(json.dumps(None))
```

运行实例

当 Python 转换为 JSON 时，Python 对象会被转换为 JSON（JavaScript）等效项：

Python	JSON
dict	Object
list	Array
tuple	Array
str	String
int	Number
float	Number
True	true
False	false
None	null

实例

转换包含所有合法数据类型的 Python 对象：

```
import json

x = {
  "name": "Bill",
  "age": 63,
  "married": True,
  "divorced": False,
  "children": ("Jennifer","Rory","Phoebe"),
  "pets": None,
  "cars": [
    {"model": "Porsche", "mpg": 38.2},
    {"model": "BMW M5", "mpg": 26.9}
  ]
}

print(json.dumps(x))
```

运行实例

格式化结果

上面的实例打印一个 JSON 字符串，但它不是很容易阅读，没有缩进和换行。
json.dumps() 方法提供了令结果更易读的参数：
实例
使用 indent 参数定义缩进数：
```
json.dumps(x, indent=4)
```
运行实例
您还可以定义分隔符，默认值为（", ", ": "），这意味着使用逗号和空格分隔每个对象，使用冒号和空格将键与值分开：
实例
使用 separators 参数来更改默认分隔符：
```
json.dumps(x, indent=4, separators=(". ", " = "))
```
运行实例

对结果排序
json.dumps() 方法提供了对结果中的键进行排序的参数：
实例
使用 sort_keys 参数来指定是否应对结果进行排序：
```
json.dumps(x, indent=4, sort_keys=True)
```
运行实例

Python 日期
Python RegEx

RegEx 或正则表达式是形成搜索模式的字符序列。
RegEx 可用于检查字符串是否包含指定的搜索模式。

RegEx 模块
Python 提供名为 re 的内置包，可用于处理正则表达式。
导入 re 模块：
```
import re
```

Python 中的 RegEx
导入 re 模块后，就可以开始使用正则表达式了：
实例
检索字符串以查看它是否以 "China" 开头并以 "country" 结尾：
```
import re

txt = "China is a great country"
x = re.search("^China.*country$", txt)
```
运行实例

RegEx 函数
re 模块提供了一组函数，允许我们检索字符串以进行匹配：
函数 描述
findall 返回包含所有匹配项的列表
search 如果字符串中的任意位置存在匹配，则返回 Match 对象
split 返回在每次匹配时拆分字符串的列表
sub 用字符串替换一个或多个匹配项

元字符
元字符是具有特殊含义的字符：
字符 描述
[] 一组字符
\ 示意特殊序列（也可用于转义特殊字符）
. 任何字符（换行符除外）
^ 起始于
$ 结束于
* 零次或多次出现
+ 一次或多次出现
{} 确切地指定的出现次数
| 两者任一
() 捕获和分组

特殊序列
特殊序列指的是 \ 后跟下表中的某个字符，拥有特殊含义：
字符 描述
\A 如果指定的字符位于字符串的开头，则返回匹配项
\b 返回指定字符位于单词的开头或末尾的匹配项

\B 返回指定字符存在的匹配项，但不在单词的开头（或结尾处）

\d 返回字符串包含数字的匹配项（数字 0-9）
\D 返回字符串不包含数字的匹配项
\s 返回字符串包含空白字符的匹配项
\S 返回字符串不包含空白字符的匹配项
\w 返回一个匹配项，其中字符串包含任何单词字符
（从 a 到 Z 的字符，从 0 到 9 的数字和下划线 _ 字符）
\W 返回一个匹配项，其中字符串不包含任何单词字符
\Z 如果指定的字符位于字符串的末尾，则返回匹配项

集合（Set）
集合（Set）是一对方括号 [] 内的一组字符，具有特殊含义：
集合 描述
[arn] 返回一个匹配项，其中存在指定字符（a，r 或 n）之一

[a-n] 返回字母顺序 a 和 n 之间的任意小写字符匹配项
[^arn] 返回除 a、r 和 n 之外的任意字符的匹配项
[0123] 返回存在任何指定数字（0、1、2 或 3）的匹配项
[0-9] 返回 0 与 9 之间任意数字的匹配
[0-5][0-9] 返回介于 0 到 9 之间的任何数字的匹配项
[a-zA-Z] 返回字母顺序 a 和 z 之间的任何字符的匹配，小写或大写
[+] 在集合中，+、*、.、|、()、$、{} 没有特殊含义，因此 [+] 表示：返回字符串中任何 + 字符的匹配项

findall() 函数
findall() 函数返回包含所有匹配项的列表。
实例
打印所有匹配的列表：

```
import re

str = "China is a great country"
x = re.findall("a", str)
print(x)
```

运行实例
这个列表以被找到的顺序包含匹配项。
如果未找到匹配项，则返回空列表：
实例
如果未找到匹配，则返回空列表：

```
import re

str = "China is a great country"
x = re.findall("USA", str)
print(x)
```

运行实例

search() 函数
search() 函数搜索字符串中的匹配项，如果存在匹配则返回 Match 对象。
如果有多个匹配，则仅返回首个匹配项：
实例
在字符串中搜索第一个空白字符：

```
import re

str = "China is a great country"
x = re.search("\s", str)

print("The first white-space character is located in position:",
x.start())
```

运行实例

如果未找到匹配，则返回值 None：
实例
进行不返回匹配的检索：

```
import re

str = "China is a great country"
x = re.search("USA", str)
print(x)
```

运行实例

split() 函数
split() 函数返回一个列表，其中字符串在每次匹配时被拆分：
实例
在每个空白字符处进行拆分：

```
import re

str = "China is a great country"
x = re.split("\s", str)
print(x)
```

运行实例
您可以通过指定 maxsplit 参数来控制出现次数：
实例
仅在首次出现时拆分字符串：

```
import re

str = "China is a great country"
x = re.split("\s", str, 1)
print(x)
```

运行实例

sub() 函数
sub() 函数把匹配替换为您选择的文本：
实例
用数字 9 替换每个空白字符：

```
import re

str = "China is a great country"
x = re.sub("\s", "9", str)
print(x)
```

运行实例
您可以通过指定 count 参数来控制替换次数：
实例
替换前两次出现：

```
import re
```

```
str = "China is a great country"
x = re.sub("\s", "9", str, 2)
print(x)
```

运行实例

Match 对象

Match 对象是包含有关搜索和结果信息的对象。

注释：如果没有匹配，则返回值 None，而不是 Match 对象。

实例

执行会返回 Match 对象的搜索：

```
import re

str = "China is a great country"
x = re.search("a", str)
print(x) # 将打印一个对象
```

运行实例

Match 对象提供了用于取回有关搜索及结果信息的属性和方法：

span() 返回的元组包含了匹配的开始和结束位置

.string 返回传入函数的字符串

group() 返回匹配的字符串部分

实例

打印首个匹配出现的位置（开始和结束位置）。

正则表达式查找以大写 "C" 开头的任何单词：

```
import re

str = "China is a great country"
x = re.search(r"\bC\w+", str)
print(x.span())
```

运行实例

实例

打印传入函数的字符串：

```
import re

str = "China is a great country"
x = re.search(r"\bC\w+", str)
print(x.string)
```

运行实例

实例

打印匹配的字符串部分。

正则表达式查找以大写 "C" 开头的任何单词：

```
import re

str = "China is a great country"
```

```
x = re.search(r"\bC\w+", str)
print(x.group())
```

运行实例

注释：如果没有匹配项，则返回值 None，而不是 Match 对象。

Python JSON

Python PIP

什么是 PIP?

PIP 是 Python 包或模块的包管理器。

注释：如果您使用的是 Python 3.4 或更高版本，则默认情况下会包含 PIP。

什么是包（Package）？

包中包含模块所需的所有文件。

模块是您可以包含在项目中的 Python 代码库。

检查是否已安装 PIP

将命令行导航到 Python 脚本目录所在的位置，然后键入以下内容：

实例

检查 PIP 版本：

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip --version
```

安装 PIP

如果尚未安装 PIP，可以从此页面下载并安装：https://pypi.org/project/pip/

下载包

下载包非常容易。

打开命令行界面并告诉 PIP 下载您需要的软件包。

将命令行导航到 Python 脚本目录的位置，然后键入以下内容：

实例

下载名为 "camelcase" 的包：

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip install camelcase
```

现在，您已经下载并安装了第一个包！

使用包

安装包后，即可使用。

把 "camelcase" 包导入您的项目中。

实例
导入并使用 "camelcase"：

```
import camelcase

c = camelcase.CamelCase()

txt = "hello world"

print(c.hump(txt))
```

运行实例

查找包
在 https://pypi.org/，您可以找到更多的包。

删除包
请使用 uninstall 命令来删除包：
实例
卸载名为 "camelcase" 的包：

```
C:\Users\Your     Name\AppData\Local\Programs\Python\Python36-
32\Scripts>pip uninstall camelcase
PIP 包管理器会要求您确认是否需要删除 camelcase 包：
Uninstalling camelcase-02.1:
  Would remove:
    c:\...\python\python36-32\lib\site-packages\camecase-0.2-
py3.6.egg-info
    c:\...\python\python36-32\lib\site-packages\camecase\*
Proceed (y/n)?
```

按 y 键，包就会被删除。

列出包
请使用 list 命令列出系统上安装的所有软件包：
实例
列出已安装的包：

```
C:\Users\Your     Name\AppData\Local\Programs\Python\Python36-
32\Scripts>pip list
```

结果：

```
Package         Version
-----------------------
camelcase       0.2
mysql-connector 2.1.6
pip             18.1
pymongo         3.6.1
setuptools      39.0.1
```

Python RegEx
Python Try Except

try 块允许您测试代码块以查找错误。
except 块允许您处理错误。
finally 块允许您执行代码，无论 try 和 except 块的结果如何。

异常处理
当我们调用 Python 并发生错误或异常时，通常会停止并生成错误消息。
可以使用 try 语句处理这些异常：
实例
try 块将生成异常，因为 x 未定义：

```
try:
  print(x)
except:
  print("An exception occurred")
```

运行实例
由于 try 块引发错误，因此会执行 except 块。
如果没有 try 块，程序将崩溃并引发错误：
实例
该语句将引发错误，因为未定义 x：

```
print(x)
```

运行实例

多个异常
您可以根据需要定义任意数量的 exception 块，例如，假如您要为特殊类型的错误执行特殊代码块：
实例
如果 try 块引发 NameError，则打印一条消息，如果是其他错误则打印另一条消息：

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

运行实例

Else
如果没有引发错误，那么您可以使用 else 关键字来定义要执行的代码块：

实例
在本例中，try 块不会生成任何错误：

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

运行实例

Finally
如果指定了 finally 块，则无论 try 块是否引发错误，都会执行 finally 块。
实例

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

运行实例
这对于关闭对象并清理资源非常有用：
实例
试图打开并写入不可写的文件：

```
try:
  f = open("demofile.txt")
  f.write("Lorum Ipsum")
except:
  print("Something went wrong when writing to the file")
finally:
  f.close()
```

运行实例
程序可以继续，而且不会打开文件对象。

引发异常
作为 Python 开发者，您可以选择在条件发生时抛出异常。
如需抛出（引发）异常，请使用 raise 关键词。
实例
假如 x 小于 0，则引发异常并终止程序：

```
x = -1

if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

运行实例

raise 关键字用于引发异常。
您能够定义所引发异常的类型、以及打印给用户的文本。
实例
如果 x 不是整数，则引发 TypeError：

```
x = "hello"

if not type(x) is int:
  raise TypeError("Only integers are allowed")
```

运行实例

Python PIP
Python 命令输入

命令行输入
Python 允许命令行输入。
这意味着我们能够要求用户输入。
Python 3.6 中的方法与 Python 2.7 略有不同。
Python 3.6 使用 input() 方法。
Python 2.7 使用 raw_input() 方法。
下面的例子会询问用户的姓名，当您输入名字时，名字将打印到屏幕上：
Python 3.6

```
print("Enter your name:")
x = input()
print("Hello ", x)
```

Python 2.7

```
print("Enter your name:")
x = raw_input()
print("Hello ", x)
```

将此文件另存为 demo_string_input.py，并通过命令行加载它：

```
C:\Users\Your Name>python demo_string_input.py
```

我们的程序会提示用户输入一个字符串：

```
Enter your name:
```

现在用户输入姓名：

```
Bill
```

然后，程序会打印一段消息：

```
Hello, Bill
```

Python Try Except
Python 字符串格式化

为了确保字符串按预期显示，我们可以使用 format() 方法对结果进行格式化。

字符串 format()
format() 方法允许您格式化字符串的选定部分。
有时文本的一部分是你无法控制的，也许它们来自数据库或用户输入？
要控制此类值，请在文本中添加占位符（花括号 {}），然后通过 format() 方法运行值：
实例
添加要显示价格的占位符：
```
price = 52
txt = "The price is {} dollars"
print(txt.format(price))
```
运行实例
您可以在花括号内添加参数以指定如何转换值：
实例
将价格格式化为带有两位小数的数字：
```
txt = "The price is {:.2f} dollars"
```
运行实例
查看字符串 format() 参考手册中的所有格式类型。

多个值
如需使用更多值，只需向 format() 方法添加更多值：
```
print(txt.format(price, itemno, count))
```
并添加更多占位符：
实例
```
quantity = 3
itemno = 567
price = 52
myorder = "I want {} pieces of item number {} for {:.2f} dollars."
print(myorder.format(quantity, itemno, price))
```
运行实例

索引号
您可以使用索引号（花括号 {0} 内的数字）来确保将值放在正确的占位符中：
实例
```
quantity = 3
itemno = 567
price = 52
myorder = "I want {0} pieces of item number {1} for {2:.2f}
```

```
dollars."
print(myorder.format(quantity, itemno, price))
```
运行实例
此外，如果要多次引用相同的值，请使用索引号：
实例
```
age = 63
name = "Bill"
txt = "His name is {1}. {1} is {0} years old."
print(txt.format(age, name))
```
运行实例

命名索引
您还可以通过在花括号 {carname} 中输入名称来使用命名索引，但是在传递参数值 txt.format(carname = "Ford") 时，必须使用名称：
实例
```
myorder = "I have a {carname}, it is a {model}."
print(myorder.format(carname = "Porsche", model = "911"))
```
运行实例

Python 命令输入
Python 文件打开

文件处理是任何 Web 应用程序的重要组成部分。
Python 有几个用于创建、读取、更新和删除文件的函数。

文件处理
在 Python 中使用文件的关键函数是 open() 函数。
open() 函数有两个参数：文件名和模式。
有四种打开文件的不同方法（模式）：
"r" - 读取 - 默认值。打开文件进行读取，如果文件不存在则报错。
"a" - 追加 - 打开供追加的文件，如果不存在则创建该文件。
"w" - 写入 - 打开文件进行写入，如果文件不存在则创建该文件。
"x" - 创建 - 创建指定的文件，如果文件存在则返回错误。
此外，您可以指定文件是应该作为二进制还是文本模式进行处理。
"t" - 文本 - 默认值。文本模式。
"b" - 二进制 - 二进制模式（例如图像）。

语法
此外，您可以指定文件是应该作为二进制还是文本模式进行处理：
```
f = open("demofile.txt")
```
以上代码等同于：

```
f = open("demofile.txt", "rt")
```

因为 "r" (读取)和 "t" (文本)是默认值，所以不需要指定它们。
注释：请确保文件存在，否则您将收到错误消息。

Python 字符串格式化
Python 文件读取

在服务器上打开文件
假设我们有以下文件，位于与 Python 相同的文件夹中：
demofile.txt

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

如需打开文件，请使用内建的 open() 函数。
open() 函数返回文件对象，此对象有一个 read() 方法用于读取文件的内容：
实例

```
f = open("demofile.txt", "r")
print(f.read())
```

运行实例

只读取文件的一部分
默认情况下，read() 方法返回整个文本，但您也可以指定要返回的字符数：
实例
返回文件中的前五个字符：

```
f = open("demofile.txt", "r")
print(f.read(5))
```

运行实例

读行
您可以使用 readline() 方法返回一行：
实例
读取文件中的一行：

```
f = open("demofile.txt", "r")
print(f.readline())
```

运行实例
通过两次调用 readline()，您可以读取前两行：
实例
读取文件中的两行：

```
f = open("demofile.txt", "r")
```

```
print(f.readline())
print(f.readline())
```

运行实例

通过循环遍历文件中的行，您可以逐行读取整个文件：

实例

逐行遍历文件：

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

运行实例

关闭文件

完成后始终关闭文件是一个好习惯。

实例

完成后关闭文件：

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

运行实例

注释：在某些情况下，由于缓冲，您应该始终关闭文件，在关闭文件之前，对文件所做的更改可能不会显示。

Python 文件打开

Python 文件写入/创建

写入已有文件

如需写入已有的文件，必须向 open() 函数添加参数：

"a" - 追加 - 会追加到文件的末尾

"w" - 写入 - 会覆盖任何已有的内容

实例

打开文件 "demofile2.txt" 并将内容追加到文件中：

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

# 追加后，打开并读取该文件：
f = open("demofile2.txt", "r")
print(f.read())
```

运行实例

实例

打开文件 "demofile3.txt" 并覆盖内容：

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

# 写入后，打开并读取该文件：
f = open("demofile3.txt", "r")
print(f.read())
```

运行实例

注释："w" 方法会覆盖全部内容。

创建新文件

如需在 Python 中创建新文件，请使用 open() 方法，并使用以下参数之一：

"x" - 创建 - 将创建一个文件，如果文件存在则返回错误

"a" - 追加 - 如果指定的文件不存在，将创建一个文件

"w" - 写入 - 如果指定的文件不存在，将创建一个文件

实例

创建名为 "myfile.txt" 的文件：

```
f = open("myfile.txt", "x")
```

结果：已创建新的空文件！

实例

如果不存在，则创建新文件：

```
f = open("myfile.txt", "w")
```

Python 文件读取
Python 文件删除

删除文件

如需删除文件，必须导入 OS 模块，并运行其 os.remove() 函数：

实例

删除文件 "demofile.txt"：

```
import os
os.remove("demofile.txt")
```

检查文件是否存在

为避免出现错误，您可能需要在尝试删除文件之前检查该文件是否存在：

实例

检查文件是否存在，然后删除它：

```
import os
if os.path.exists("demofile.txt"):
```

```
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

删除文件
如需删除整个文件夹，请使用 os.rmdir() 方法：
实例
删除文件夹 "myfolder"：

```
import os
os.rmdir("myfolder")
```

提示：您只能删除空文件夹。

Python 文件写入/创建
NumPy 简介

什么是 NumPy?
NumPy 是用于处理数组的 python 库。
它还拥有在线性代数、傅立叶变换和矩阵领域中工作的函数。
NumPy 由 Travis Oliphant 于 2005 年创建。它是一个开源项目，您可以自由使用它。
NumPy 指的是数值 Python（Numerical Python）。

为何使用 NumPy?
在 Python 中，我们有满足数组功能的列表，但是处理起来很慢。
NumPy 旨在提供一个比传统 Python 列表快 50 倍的数组对象。
NumPy 中的数组对象称为 ndarray，它提供了许多支持函数，使得利用 ndarray 非常容易。
数组在数据科学中非常常用，因为速度和资源非常重要。
数据科学：计算机科学的一个分支，研究如何存储、使用和分析数据以从中获取信息。

为什么 NumPy 比列表快?
与列表不同，NumPy 数组存储在内存中的一个连续位置，因此进程可以非常有效地访问和操纵它们。
这种行为在计算机科学中称为引用的局部性。
这是 NumPy 比列表更快的主要原因。它还经过了优化，可与最新的 CPU 体系结构一同使用。

NumPy 用哪种语言编写?
NumPy 是一个 Python 库，部分用 Python 编写，但是大多数需要快速计算的部分都是用 C 或 C ++ 编写的。

NumPy 代码库在哪里？
NumPy 的源代码位于这个 github 资料库中：
https://github.com/numpy/numpy
github：使许多人可以在同一代码库上工作。

Python 文件删除
NumPy 入门

安装 NumPy
如果您已经在系统上安装了 Python 和 PIP，那么安装 NumPy 非常容易。
请使用这条命令安装它：
C:\Users\Your Name>pip install numpy
如果此命令失败，请使用已经安装了 NumPy 的 python 发行版，例如 Anaconda、Spyder 等。

导入 NumPy
安装 NumPy 后，通过添加 import 关键字将其导入您的应用程序：
import numpy
现在，Numpy 已导入并可以使用。
实例
```
import numpy

arr = numpy.array([1, 2, 3, 4, 5])

print(arr)
```
运行实例

NumPy as np
NumPy 通常以 np 别名导入。
别名：在 Python 中，别名是用于引用同一事物的替代名称。
请在导入时使用 as 关键字创建别名：
import numpy as np
现在，可以将 NumPy 包称为 np 而不是 numpy。
实例
```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)
```

运行实例

检查 NumPy 版本
版本字符串存储在 __version__ 属性中。
实例
```
import numpy as np

print(np.__version__)
```
运行实例

NumPy 简介
NumPy 数组创建

创建 NumPy ndarray 对象
NumPy 用于处理数组。 NumPy 中的数组对象称为 ndarray。
我们可以使用 array() 函数创建一个 NumPy ndarray 对象。
实例
```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)

print(type(arr))
```
运行实例
type(): 这个内置的 Python 函数告诉我们传递给它的对象的类型。像上面的代码一样，它表明 是 类型。
要创建 ndarray，我们可以将列表、元组或任何类似数组的对象传递给 array() 方法，然后它将被转换为 ：
实例
使用元组创建 NumPy 数组：
```
import numpy as np

arr = np.array((1, 2, 3, 4, 5))

print(arr)
```
运行实例

数组中的维
数组中的维是数组深度（嵌套数组）的一个级别。
嵌套数组：指的是将数组作为元素的数组。

0-D 数组
0-D 数组，或标量（Scalars），是数组中的元素。数组中的每个值都是一个 0-D 数组。
实例
用值 61 创建 0-D 数组：

```
import numpy as np

arr = np.array(61)

print(arr)
```

运行实例

1-D 数组
其元素为 0-D 数组的数组，称为一维或 1-D 数组。
这是最常见和基础的数组。
实例
创建包含值 1、2、3、4、5、6 的 1-D 数组：

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6])

print(arr)
```

运行实例

2-D 数组
其元素为 1-D 数组的数组，称为 2-D 数组。
它们通常用于表示矩阵或二阶张量。
NumPy 有一个专门用于矩阵运算的完整子模块 numpy.mat。
实例
创建包含值 1、2、3 和 4、5、6 两个数组的 2-D 数组：

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

print(arr)
```

运行实例

3-D 数组
其元素为 2-D 数组的数组，称为 3-D 数组。
实例
用两个 2-D 数组创建一个 3-D 数组，这两个数组均包含值 1、2、3 和 4、5、6 的两个数组：

```
import numpy as np
```

```
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(arr)
```

运行实例

检查维数？

NumPy 数组提供了 ndim 属性，该属性返回一个整数，该整数会告诉我们数组有多少维。

实例

检查数组有多少维：

```
import numpy as np

a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

运行实例

更高维的数组

数组可以拥有任意数量的维。

在创建数组时，可以使用 ndmin 参数定义维数。

实例

创建一个有 5 个维度的数组，并验证它拥有 5 个维度：

```
import numpy as np

arr = np.array([1, 2, 3, 4], ndmin=5)

print(arr)
print('number of dimensions :', arr.ndim)
```

运行实例

在此数组中，最里面的维度（第 5 个 dim）有 4 个元素，第 4 个 dim 有 1 个元素作为向量，第 3 个 dim 具有 1 个元素是与向量的矩阵，第 2 个 dim 有 1 个元素是 3D 数组，而第 1 个 dim 有 1 个元素，该元素是 4D 数组。

NumPy 入门

NumPy 数组索引

访问数组元素
数组索引等同于访问数组元素。
您可以通过引用其索引号来访问数组元素。
NumPy 数组中的索引以 0 开头，这意味着第一个元素的索引为 0，第二个元素的索引为 1，以此类推。
实例
从以下数组中获取第一个元素：

```
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr[0])
```

运行实例
实例
从以下数组中获取第二个元素：

```
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr[1])
```

运行实例
实例
从以下数组中获取第三和第四个元素并将其相加：

```
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr[2] + arr[3])
```

运行实例

访问 2-D 数组
要访问二维数组中的元素，我们可以使用逗号分隔的整数表示元素的维数和索引。
实例
访问第一维中的第二个元素：

```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('2nd element on 1st dim: ', arr[0, 1])
```

运行实例

实例
访问第二维中的第五个元素：

```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('5th element on 2nd dim: ', arr[1, 4])
```

运行实例

访问 3-D 数组
要访问 3-D 数组中的元素，我们可以使用逗号分隔的整数来表示元素的维数和索引。
实例
访问第一个数组的第二个数组的第三个元素：

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

print(arr[0, 1, 2])
```

运行实例
例子解释
arr[0, 1, 2] 打印值 6。
工作原理：
第一个数字代表第一个维度，其中包含两个数组：
[[1, 2, 3], [4, 5, 6]]
然后：
[[7, 8, 9], [10, 11, 12]]
由于我们选择了 0，所以剩下第一个数组：
[[1, 2, 3], [4, 5, 6]]
第二个数字代表第二维，它也包含两个数组：
[1, 2, 3]
然后：
[4, 5, 6]
因为我们选择了 1，所以剩下第二个数组：
[4, 5, 6]
第三个数字代表第三维，其中包含三个值：
4
5
6
由于我们选择了 2，因此最终得到第三个值：
6

负索引

使用负索引从尾开始访问数组。
实例
打印第二个维中的的最后一个元素：

```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('Last element from 2nd dim: ', arr[1, -1])
```

运行实例

NumPy 数组创建
NumPy 数组裁切

裁切数组
python 中裁切的意思是将元素从一个给定的索引带到另一个给定的索引。
我们像这样传递切片而不是索引：[start：end]。
我们还可以定义步长，如下所示：[start：end：step]。
如果我们不传递 start，则将其视为 0。
如果我们不传递 end，则视为该维度内数组的长度。
如果我们不传递 step，则视为 1。
实例
从下面的数组中裁切索引 1 到索引 5 的元素：

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5])
```

运行实例
注释：结果包括了开始索引，但不包括结束索引。
实例
裁切数组中索引 4 到结尾的元素：

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[4:])
```

运行实例
实例
裁切从开头到索引 4（不包括）的元素：

```
import numpy as np
```

```
arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[:4])
```

运行实例

负裁切

使用减号运算符从末尾开始引用索引：

实例

从末尾开始的索引 3 到末尾开始的索引 1，对数组进行裁切：

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[-3:-1])
```

运行实例

STEP

请使用 step 值确定裁切的步长：

实例

从索引 1 到索引 5，返回相隔的元素：

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5:2])
```

运行实例

实例

返回数组中相隔的元素：

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[::2])
```

运行实例

裁切 2-D 数组

实例

从第二个元素开始，对从索引 1 到索引 4（不包括）的元素进行切片：

```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[1, 1:4])
```

运行实例
注释：请记得第二个元素的索引为 1。
实例
从两个元素中返回索引 2：

```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[0:2, 2])
```

运行实例
实例
从两个元素裁切索引 1 到索引 4（不包括），这将返回一个 2-D 数组：

```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[0:2, 1:4])
```

运行实例

NumPy 数组索引
NumPy 数据类型

Python 中的数据类型
默认情况下，Python 拥有以下数据类型：
strings - 用于表示文本数据，文本用引号引起来。例如 "ABCD"。
integer - 用于表示整数。例如 -1, -2, -3。
float - 用于表示实数。例如 1.2, 42.42。
boolean - 用于表示 True 或 False。
complex - 用于表示复平面中的数字。例如 1.0 + 2.0j，1.5 + 2.5j。

NumPy 中的数据类型
NumPy 有一些额外的数据类型，并通过一个字符引用数据类型，例如 i 代表整数，u 代表无符号整数等。
以下是 NumPy 中所有数据类型的列表以及用于表示它们的字符。
i - 整数
b - 布尔
u - 无符号整数
f - 浮点
c - 复合浮点数
m - timedelta
M - datetime

O - 对象
S - 字符串
U - unicode 字符串
V - 固定的其他类型的内存块 (void)

检查数组的数据类型
NumPy 数组对象有一个名为 dtype 的属性，该属性返回数组的数据类型：
实例
获取数组对象的数据类型：

```
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr.dtype)
```

运行实例
实例
获取包含字符串的数组的数据类型：

```
import numpy as np

arr = np.array(['apple', 'banana', 'cherry'])

print(arr.dtype)
```

运行实例

用已定义的数据类型创建数组
我们使用 array() 函数来创建数组，该函数可以使用可选参数：dtype，它允许我们定义数组元素的预期数据类型：
实例
用数据类型字符串创建数组：

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='S')

print(arr)
print(arr.dtype)
```

运行实例
对于 i、u、f、S 和 U，我们也可以定义大小。
实例
创建数据类型为 4 字节整数的数组：

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='i4')
```

```
print(arr)
print(arr.dtype)
```
运行实例

假如值无法转换会怎样？
如果给出了不能强制转换元素的类型，则 NumPy 将引发 ValueError。
ValueError：在 Python 中，如果传递给函数的参数的类型是非预期或错误的，则会引发 ValueError。
实例
无法将非整数字符串（比如 'a'）转换为整数（将引发错误）：
```
import numpy as np

arr = np.array(['a', '2', '3'], dtype='i')
```
运行实例

转换已有数组的数据类型
更改现有数组的数据类型的最佳方法，是使用 astype() 方法复制该数组。
astype() 函数创建数组的副本，并允许您将数据类型指定为参数。
数据类型可以使用字符串指定，例如 'f' 表示浮点数，'i' 表示整数等。或者您也可以直接使用数据类型，例如 表示浮点数，int 表示整数。
实例
通过使用 'i' 作为参数值，将数据类型从浮点数更改为整数：
```
import numpy as np

arr = np.array([1.1, 2.1, 3.1])

newarr = arr.astype('i')

print(newarr)
print(newarr.dtype)
```
运行实例
实例
通过使用 int 作为参数值，将数据类型从浮点数更改为整数：
```
import numpy as np

arr = np.array([1.1, 2.1, 3.1])

newarr = arr.astype(int)

print(newarr)
print(newarr.dtype)
```
运行实例

实例
将数据类型从整数更改为布尔值：

```
import numpy as np

arr = np.array([1, 0, 3])

newarr = arr.astype(bool)

print(newarr)
print(newarr.dtype)
```

运行实例

NumPy 数组裁切
NumPy 副本/视图

副本和视图之间的区别
副本和数组视图之间的主要区别在于副本是一个新数组，而这个视图只是原始数组的视图。
副本拥有数据，对副本所做的任何更改都不会影响原始数组，对原始数组所做的任何更改也不会影响副本。
视图不拥有数据，对视图所做的任何更改都会影响原始数组，而对原始数组所做的任何更改都会影响视图。

副本：
实例
进行复制，更改原始数组并显示两个数组：

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
arr[0] = 61

print(arr)
print(x)
```

运行实例
该副本不应受到对原始数组所做更改的影响。

视图：
实例
创建视图，更改原始数组，然后显示两个数组：

```
import numpy as np
```

```
arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
arr[0] = 61

print(arr)
print(x)
```

运行实例

视图应该受到对原始数组所做更改的影响。

在视图中进行更改：

实例

创建视图，更改视图，并显示两个数组：

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
x[0] = 31

print(arr)
print(x)
```

运行实例

原始数组应该受到对视图所做更改的影响。

检查数组是否拥有数据

如上所述，副本拥有数据，而视图不拥有数据，但是我们如何检查呢？

每个 NumPy 数组都有一个属性 base，如果该数组拥有数据，则这个 base 属性返回 None。

否则，base 属性将引用原始对象。

实例

打印 base 属性的值以检查数组是否拥有自己的数据：

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

x = arr.copy()
y = arr.view()

print(x.base)
print(y.base)
```

运行实例

副本返回 None。

视图返回原始数组。

NumPy 数据类型

NumPy 数组形状

数组的形状
数组的形状是每个维中元素的数量。

获取数组的形状
NumPy 数组有一个名为 shape 的属性，该属性返回一个元组，每个索引具有相应元素的数量。
实例
打印 2-D 数组的形状：

```
import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

print(arr.shape)
```

运行实例
上面的例子返回 (2, 4)，这意味着该数组有 2 个维，每个维有 4 个元素。
实例
利用 ndmin 使用值 1,2,3,4 的向量创建有 5 个维度的数组，并验证最后一个维度的值为 4：

```
import numpy as np

arr = np.array([1, 2, 3, 4], ndmin=5)

print(arr)
print('shape of array :', arr.shape)
```

运行实例

元组的形状代表什么？
每个索引处的整数表明相应维度拥有的元素数量。
上例中的索引 4，我们的值为 4，因此可以说第 5 个 (4 + 1 th) 维度有 4 个元素。

NumPy 副本/视图
NumPy 数组重塑

数组重塑

重塑意味着更改数组的形状。
数组的形状是每个维中元素的数量。
通过重塑，我们可以添加或删除维度或更改每个维度中的元素数量。

从 1-D 重塑为 2-D
实例
将以下具有 12 个元素的 1-D 数组转换为 2-D 数组。
最外面的维度将有 4 个数组，每个数组包含 3 个元素：

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(4, 3)

print(newarr)
```

运行实例

从 1-D 重塑为 3-D
实例
将以下具有 12 个元素的 1-D 数组转换为 3-D 数组。
最外面的维度将具有 2 个数组，其中包含 3 个数组，每个数组包含 2 个元素：

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(2, 3, 2)

print(newarr)
```

运行实例

我们可以重塑成任何形状吗？
是的，只要重塑所需的元素在两种形状中均相等。
我们可以将 8 元素 1D 数组重塑为 2 行 2D 数组中的 4 个元素，但是我们不能将其重塑为 3 元素 3 行 2D 数组，因为这将需要 3x3 = 9 个元素。
实例
尝试将具有 8 个元素的 1D 数组转换为每个维度中具有 3 个元素的 2D 数组（将产生错误）：

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

newarr = arr.reshape(3, 3)
```

```
print(newarr)
```
运行实例

返回副本还是视图？
实例
检查返回的数组是副本还是视图：
```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

print(arr.reshape(2, 4).base)
```
运行实例
上面的例子返回原始数组，因此它是一个视图。

未知的维
您可以使用一个“未知”维度。
这意味着您不必在 reshape 方法中为维度之一指定确切的数字。
传递 -1 作为值，NumPy 将为您计算该数字。
实例
将 8 个元素的 1D 数组转换为 2x2 元素的 3D 数组：
```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

newarr = arr.reshape(2, 2, -1)

print(newarr)
```
运行实例
注释：我们不能将 -1 传递给一个以上的维度。

展平数组
展平数组（Flattening the arrays）是指将多维数组转换为 1D 数组。
我们可以使用 reshape(-1) 来做到这一点。
实例
把数组转换为 1D 数组：
```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

newarr = arr.reshape(-1)

print(newarr)
```
运行实例

注释：有很多功能可以更改 numpy flatten、ravel 中数组形状，还可以重新排列元素 rot90、flip、fliplr、flipud 等。这些功能属于 numpy 的中级至高级部分。

NumPy 数组形状
NumPy 数组迭代

数组迭代
迭代意味着逐一遍历元素。
当我们在 numpy 中处理多维数组时，可以使用 python 的基本 for 循环来完成此操作。
如果我们对 1-D 数组进行迭代，它将逐一遍历每个元素。
实例
迭代以下一维数组的元素：

```
import numpy as np

arr = np.array([1, 2, 3])

for x in arr:
  print(x)
```

运行实例

迭代 2-D 数组
在 2-D 数组中，它将遍历所有行。
实例
迭代以下二维数组的元素：

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

for x in arr:
  print(x)
```

运行实例
如果我们迭代一个 n-D 数组，它将逐一遍历第 n-1 维。
如需返回实际值、标量，我们必须迭代每个维中的数组。
实例
迭代 2-D 数组的每个标量元素：

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])
```

```
for x in arr:
  for y in x:
    print(y)
```

运行实例

迭代 3-D 数组

在 3-D 数组中，它将遍历所有 2-D 数组。

实例

迭代以下 3-D 数组的元素：

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

for x in arr:
  print(x)
```

运行实例

要返回实际值、标量，我们必须迭代每个维中的数组。

实例

迭代到标量：

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

for x in arr:
  for y in x:
    for z in y:
      print(z)
```

运行实例

使用 nditer() 迭代数组

函数 nditer() 是一个辅助函数，从非常基本的迭代到非常高级的迭代都可以使用。它解决了我们在迭代中面临的一些基本问题，让我们通过例子进行介绍。

迭代每个标量元素

在基本的 for 循环中，迭代遍历数组的每个标量，我们需要使用 n 个 for 循环，对于具有高维数的数组可能很难编写。

实例

遍历以下 3-D 数组：

```
import numpy as np

arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
```

```
for x in np.nditer(arr):
  print(x)
```

运行实例

迭代不同数据类型的数组

我们可以使用 op_dtypes 参数，并传递期望的数据类型，以在迭代时更改元素的数据类型。

NumPy 不会就地更改元素的数据类型（元素位于数组中），因此它需要一些其他空间来执行此操作，该额外空间称为 buffer，为了在 nditer() 中启用它，我们传参 flags=['buffered']。

实例

以字符串形式遍历数组：

```
import numpy as np

arr = np.array([1, 2, 3])

for x in np.nditer(arr, flags=['buffered'], op_dtypes=['S']):
  print(x)
```

运行实例

以不同的步长迭代

我们可以使用过滤，然后进行迭代。

实例

每遍历 2D 数组的一个标量元素，跳过 1 个元素：

```
import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

for x in np.nditer(arr[:, ::2]):
  print(x)
```

运行实例

使用 ndenumerate() 进行枚举迭代

枚举是指逐一提及事物的序号。

有时，我们在迭代时需要元素的相应索引，对于这些用例，可以使用 ndenumerate() 方法。

实例

枚举以下 1D 数组元素：

```
import numpy as np

arr = np.array([1, 2, 3])

for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

运行实例
实例
枚举以下 2D 数组元素：

```
import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

运行实例

NumPy 数组重塑
NumPy 数组连接

连接 NumPy 数组
连接意味着将两个或多个数组的内容放在单个数组中。
在 SQL 中，我们基于键来连接表，而在 NumPy 中，我们按轴连接数组。
我们传递了一系列要与轴一起连接到 concatenate() 函数的数组。如果未显式传递轴，则将其视为 0。
实例
连接两个数组：

```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.concatenate((arr1, arr2))

print(arr)
```

运行实例
实例
沿着行 (axis=1) 连接两个 2-D 数组：

```
import numpy as np

arr1 = np.array([[1, 2], [3, 4]])

arr2 = np.array([[5, 6], [7, 8]])

arr = np.concatenate((arr1, arr2), axis=1)
```

```
print(arr)
```
运行实例

使用堆栈函数连接数组
堆栈与级联相同，唯一的不同是堆栈是沿着新轴完成的。
我们可以沿着第二个轴连接两个一维数组，这将导致它们彼此重叠，即，堆叠（stacking）。
我们传递了一系列要与轴一起连接到 concatenate() 方法的数组。如果未显式传递轴，则将其视为 0。
实例
```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.stack((arr1, arr2), axis=1)

print(arr)
```
运行实例

沿行堆叠
NumPy 提供了一个辅助函数：hstack() 沿行堆叠。
实例
```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.hstack((arr1, arr2))

print(arr)
```
运行实例

沿列堆叠
NumPy 提供了一个辅助函数：vstack() 沿列堆叠。
实例
```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])
```

```
arr = np.vstack((arr1, arr2))

print(arr)
```

运行实例

沿高度堆叠（深度）

NumPy 提供了一个辅助函数：dstack() 沿高度堆叠，该高度与深度相同。

实例

```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.dstack((arr1, arr2))

print(arr)
```

运行实例

NumPy 数组迭代

NumPy 数组拆分

拆分 NumPy 数组

拆分是连接的反向操作。

连接（Joining）是将多个数组合并为一个，拆分（Spliting）将一个数组拆分为多个。

我们使用 array_split() 分割数组，将要分割的数组和分割数传递给它。

实例

将数组分为 3 部分：

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6])

newarr = np.array_split(arr, 3)

print(newarr)
```

运行实例

注释：返回值是一个包含三个数组的数组。

如果数组中的元素少于要求的数量，它将从末尾进行相应调整。

实例
将数组分为 4 部分：

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6])

newarr = np.array_split(arr, 4)

print(newarr)
```

运行实例
提示：我们也有 split() 方法可用，但是当源数组中的元素较少用于拆分时，它将不会调整元素，如上例那样，array_split() 正常工作，但 split() 会失败。

拆分为数组
array_split() 方法的返回值是一个包含每个分割的数组。
如果将一个数组拆分为 3 个数组，则可以像使用任何数组元素一样从结果中访问它们：
实例
访问拆分的数组：

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6])

newarr = np.array_split(arr, 3)

print(newarr[0])
print(newarr[1])
print(newarr[2])
```

运行实例

分割二维数组
拆分二维数组时，请使用相同的语法。
使用 array_split() 方法，传入要分割的数组和想要分割的数目。
实例
把这个 2-D 拆分为三个 2-D 数组。

```
import numpy as np

arr = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])

newarr = np.array_split(arr, 3)

print(newarr)
```

运行实例

上例返回三个 2-D 数组。

让我们看另一个例子，这次 2-D 数组中的每个元素包含 3 个元素。

实例

把这个 2-D 拆分为三个 2-D 数组。

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12],
[13, 14, 15], [16, 17, 18]])

newarr = np.array_split(arr, 3)

print(newarr)
```

运行实例

上例返回三个 2-D 数组。

此外，您可以指定要进行拆分的轴。

下面的例子还返回三个 2-D 数组，但它们沿行 (axis=1) 分割。

实例

沿行把这个 2-D 拆分为三个 2-D 数组。

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12],
[13, 14, 15], [16, 17, 18]])

newarr = np.array_split(arr, 3, axis=1)

print(newarr)
```

运行实例

另一种解决方案是使用与 hstack() 相反的 hsplit()。

实例

使用 hsplit() 方法将 2-D 数组沿着行分成三个 2-D 数组。

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12],
[13, 14, 15], [16, 17, 18]])

newarr = np.hsplit(arr, 3)

print(newarr)
```

运行实例

提示：vsplit() 和 dsplit() 可以使用与 vstack() 和 dstack() 类似的替代方法。

NumPy 数组连接

NumPy 数组搜索

搜索数组
您可以在数组中搜索（检索）某个值，然后返回获得匹配的索引。
要搜索数组，请使用 where() 方法。
实例
查找值为 4 的索引：

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 4, 4])

x = np.where(arr == 4)

print(x)
```

运行实例
上例会返回一个元组：(array([3, 5, 6],)
意思就是值 4 出现在索引 3、5 和 6。
实例
查找值为偶数的索引：

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

x = np.where(arr%2 == 0)

print(x)
```

运行实例
实例
查找值为奇数的索引：

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

x = np.where(arr%2 == 1)

print(x)
```

运行实例

搜索排序
有一个名为 searchsorted() 的方法，该方法在数组中执行二进制搜索，并返回将在其中插入指定值以维持搜索顺序的索引。

假定 searchsorted() 方法用于排序数组。
实例
查找应在其中插入值 7 的索引：

```
import numpy as np

arr = np.array([6, 7, 8, 9])

x = np.searchsorted(arr, 7)

print(x)
```

运行实例
例子解释：应该在索引 1 上插入数字 7，以保持排序顺序。
该方法从左侧开始搜索，并返回第一个索引，其中数字 7 不再大于下一个值。
从右侧搜索
默认情况下，返回最左边的索引，但是我们可以给定 side='right'，以返回最右边的索引。
实例
从右边开始查找应该插入值 7 的索引：

```
import numpy as np

arr = np.array([6, 7, 8, 9])

x = np.searchsorted(arr, 7, side='right')

print(x)
```

运行实例
例子解释：应该在索引 2 上插入数字 7，以保持排序顺序。
该方法从右边开始搜索，并返回第一个索引，其中数字 7 不再小于下一个值。

多个值
要搜索多个值，请使用拥有指定值的数组。
实例
查找应在其中插入值 2、4 和 6 的索引：

```
import numpy as np

arr = np.array([1, 3, 5, 7])

x = np.searchsorted(arr, [2, 4, 6])

print(x)
```

运行实例
返回值是一个数组：[1 2 3] 包含三个索引，其中将在原始数组中插入

2、4、6 以维持顺序。

NumPy 数组拆分
NumPy 数组排序

数组排序
排序是指将元素按有序顺序排列。
有序序列是拥有与元素相对应的顺序的任何序列，例如数字或字母、升序或降序。
NumPy ndarray 对象有一个名为 sort() 的函数，该函数将对指定的数组进行排序。
实例
对数组进行排序：

```
import numpy as np

arr = np.array([3, 2, 0, 1])

print(np.sort(arr))
```

运行实例
注释：此方法返回数组的副本，而原始数组保持不变。
您还可以对字符串数组或任何其他数据类型进行排序：
实例
对数组以字母顺序进行排序：

```
import numpy as np

arr = np.array(['banana', 'cherry', 'apple'])

print(np.sort(arr))
```

运行实例
实例
对布尔数组进行排序：

```
import numpy as np

arr = np.array([True, False, True])

print(np.sort(arr))
```

运行实例

对 2-D 数组排序
如果在二维数组上使用 sort() 方法，则将对两个数组进行排序：
实例

对 2-D 数组排序

```
import numpy as np

arr = np.array([[3, 2, 4], [5, 0, 1]])

print(np.sort(arr))
```

运行实例

NumPy 数组搜索

NumPy 数组过滤

数组过滤

从现有数组中取出一些元素并从中创建新数组称为过滤（filtering）。

在 NumPy 中，我们使用布尔索引列表来过滤数组。

布尔索引列表是与数组中的索引相对应的布尔值列表。

如果索引处的值为 True，则该元素包含在过滤后的数组中；如果索引处的值为 False，则该元素将从过滤后的数组中排除。

实例

用索引 0 和 2、4 上的元素创建一个数组：

```
import numpy as np

arr = np.array([61, 62, 63, 64, 65])

x = [True, False, True, False, True]

newarr = arr[x]

print(newarr)
```

运行实例

上例将返回 [61, 63, 65]，为什么？

因为新过滤器仅包含过滤器数组有值 True 的值，所以在这种情况下，索引为 0 和 2、4。

创建过滤器数组

在上例中，我们对 True 和 False 值进行了硬编码，但通常的用途是根据条件创建过滤器数组。

实例

创建一个仅返回大于 62 的值的过滤器数组：

```
import numpy as np

arr = np.array([61, 62, 63, 64, 65])
```

```
# 创建一个空列表
filter_arr = []

# 遍历 arr 中的每个元素
for element in arr:
  # 如果元素大于 62，则将值设置为 True，否则为 False：
  if element > 62:
    filter_arr.append(True)
  else:
    filter_arr.append(False)

newarr = arr[filter_arr]

print(filter_arr)
print(newarr)
```

运行实例

实例

创建一个过滤器数组，该数组仅返回原始数组中的偶数元素：

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

# 创建一个空列表
filter_arr = []

# 遍历 arr 中的每个元素
for element in arr:
  # 如果元素可以被 2 整除，则将值设置为 True，否则设置为 False
  if element % 2 == 0:
    filter_arr.append(True)
  else:
    filter_arr.append(False)

newarr = arr[filter_arr]

print(filter_arr)
print(newarr)
```

运行实例

直接从数组创建过滤器

上例是 NumPy 中非常常见的任务，NumPy 提供了解决该问题的好方法。我们可以在条件中直接替换数组而不是 iterable 变量，它会如我们期望地那样工作。

实例
创建一个仅返回大于 62 的值的过滤器数组：

```
import numpy as np

arr = np.array([61, 62, 63, 64, 65])

filter_arr = arr > 62

newarr = arr[filter_arr]

print(filter_arr)
print(newarr)
```

运行实例
实例
创建一个过滤器数组，该数组仅返回原始数组中的偶数元素：

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

filter_arr = arr % 2 == 0

newarr = arr[filter_arr]

print(filter_arr)
print(newarr)
```

运行实例

NumPy 数组排序
NumPy 随机

什么是随机数？
随机数并不意味着每次都有不同的数字。随机意味着无法在逻辑上预测的事物。

伪随机和真随机
计算机在程序上工作，程序是权威的指令集。因此，这意味着必须有某种算法来生成随机数。
如果存在生成随机数的程序，则可以预测它，因此它就不是真正的随机数。
通过生成算法生成的随机数称为伪随机数。
我们可以生成真正的随机数吗？

是的。为了在我们的计算机上生成一个真正的随机数，我们需要从某个外部来源获取随机数据。外部来源通常是我们的击键、鼠标移动、网络数据等。
我们不需要真正的随机数，除非它与安全性（例如加密密钥）有关或应用的基础是随机性（例如数字轮盘赌轮）。
在本教程中，我们将使用伪随机数。

生成随机数
NumPy 提供了 random 模块来处理随机数。
实例
生成一个 0 到 100 之间的随机整数：

```
from numpy import random

x = random.randint(100)

print(x)
```

运行实例

生成随机浮点
random 模块的 rand() 方法返回 0 到 1 之间的随机浮点数。
实例
生成一个 0 到 100 之间的随机浮点数：

```
from numpy import random

x = random.rand()

print(x)
```

运行实例

生成随机数组
在 NumPy 中，我们可以使用上例中的两种方法来创建随机数组。
整数
randint() 方法接受 size 参数，您可以在其中指定数组的形状。
实例
生成一个 1-D 数组，其中包含 5 个从 0 到 100 之间的随机整数：

```
from numpy import random

x=random.randint(100, size=(5))

print(x)
```

运行实例
实例
生成有 3 行的 2-D 数组，每行包含 5 个从 0 到 100 之间的随机整数：

```
from numpy import random

x = random.randint(100, size=(3, 5))

print(x)
```

运行实例

浮点数

rand() 方法还允许您指定数组的形状。

实例

生成包含 5 个随机浮点数的 1-D 数组：

```
from numpy import random

x = random.rand(5)

print(x)
```

运行实例

实例

生成有 3 行的 2-D 数组，每行包含 5 个随机数：

```
from numpy import random

x = random.rand(3, 5)

print(x)
```

运行实例

从数组生成随机数

choice() 方法使您可以基于值数组生成随机值。

choice() 方法将数组作为参数，并随机返回其中一个值。

实例

返回数组中的值之一：

```
from numpy import random

x = random.choice([3, 5, 7, 9])

print(x)
```

运行实例

choice() 方法还允许您返回一个值数组。

请添加一个 size 参数以指定数组的形状。

实例

生成由数组参数（3、5、7 和 9）中的值组成的二维数组：

```
from numpy import random

x = random.choice([3, 5, 7, 9], size=(3, 5))
```

```
print(x)
```
运行实例

NumPy 数组过滤
NumPy ufuncs

什么是 ufuncs?
ufuncs 指的是“通用函数”(Universal Functions),它们是对 ndarray 对象进行操作的 NumPy 函数。

为什么要使用 ufuncs?
ufunc 用于在 NumPy 中实现矢量化，这比迭代元素要快得多。
它们还提供广播和其他方法，例如减少、累加等，它们对计算非常有帮助。
ufuncs 还接受其他参数，比如：
where 布尔值数组或条件，用于定义应在何处进行操作。
dtype 定义元素的返回类型。
out 返回值应被复制到的输出数组。

什么是向量化?
将迭代语句转换为基于向量的操作称为向量化。
由于现代 CPU 已针对此类操作进行了优化，因此速度更快。
对两个列表的元素进行相加：
list 1: [1, 2, 3, 4]
list 2: [4, 5, 6, 7]
一种方法是遍历两个列表，然后对每个元素求和。
实例
如果没有 ufunc，我们可以使用 Python 的内置 zip() 方法：

```
x = [1, 2, 3, 4]
y = [4, 5, 6, 7]
z = []

for i, j in zip(x, y):
  z.append(i + j)
print(z)
```
运行实例
对此，NumPy 有一个 ufunc，名为 add(x, y)，它会输出相同的结果。
实例
通过 ufunc，我们可以使用 add() 函数：

```
import numpy as np
```

```
x = [1, 2, 3, 4]
y = [4, 5, 6, 7]
z = np.add(x, y)

print(z)
```

运行实例

NumPy 随机
入门

机器学习使计算机能够从研究数据和统计信息中学习。
机器学习是迈向人工智能（AI）方向的其中一步。
机器学习是一种程序，可以分析数据并学习预测结果。

从何处开始？
在本教程中，我们将回到数学并研究统计学，以及如何根据数据集计算重要数值。
我们还将学习如何使用各种 Python 模块来获得所需的答案。
并且，我们将学习如何根据所学知识编写能够预测结果的函数。

数据集
在计算机中，数据集指的是任何数据集合。它可以是从数组到完整数据库的任何内容。
一个数组的例子：
[99,86,87,88,111,86,103,87,94,78,77,85,86]
一个数据库的例子：

Carname	Color	Age	Speed
BMW	red	5	99
Volvo	black	7	86
VW	gray	8	87
VW	white	7	88
Ford	white	2	111
VW	white	17	86
Tesla	red	2	103
BMW	black	9	87
Volvo	gray	4	94
Ford	white	11	78
Toyota	gray	12	77
VW	white	9	85
Toyota	blue	6	86

通过查看数组，我们可以猜测平均值可能约为 80 或 90，并且我们还

可以确定最大值和最小值，但是我们还能做什么？
通过查看数据库，我们可以看到最受欢迎的颜色是白色，最老的车龄是 17 年，但是如果仅通过查看其他值就可以预测汽车是否具有 AutoPass，该怎么办？
这就是机器学习的目的！分析数据并预测结果！
在机器学习中，通常使用非常大的数据集。在本教程中，我们会尝试让您尽可能容易地理解机器学习的不同概念，并将使用一些易于理解的小型数据集。

数据类型
如需分析数据，了解我们要处理的数据类型非常重要。
我们可以将数据类型分为三种主要类别：
数值（Numerical）
分类（Categorical）
序数（Ordinal）
数值数据是数字，可以分为两种数值类别：
离散数据（Discrete Data）
- 限制为整数的数字。例如：经过的汽车数量。
连续数据（Continuous Data）
- 具有无限值的数字。例如：一件商品的价格或一件商品的大小。
分类数据是无法相互度量的值。例如：颜色值或任何 yes/no 值。
序数数据类似于分类数据，但可以相互度量。示例：A 优于 B 的学校成绩，依此类推。
通过了解数据源的数据类型，您就能够知道在分析数据时使用何种技术。
在下一章中，您将学习有关统计和分析数据的更多知识。

NumPy ufuncs
平均中位数模式

均值、中值和众数
从一组数字中我们可以学到什么？
在机器学习（和数学）中，通常存在三中我们感兴趣的值：
均值（Mean） - 平均值
中值（Median） - 中点值，又称中位数
众数（Mode） - 最常见的值
例如：我们已经登记了 13 辆车的速度：
speed = [99,86,87,88,111,86,103,87,94,78,77,85,86]
什么是平均，中间或最常见的速度值？

均值
均值就是平均值。

要计算平均值，请找到所有值的总和，然后将总和除以值的数量：
(99+86+87+88+111+86+103+87+94+78+77+85+86) / 13 = 89.77
NumPy 模块拥有用于此目的的方法：
实例
请使用 NumPy mean() 方法确定平均速度：

```
import numpy

speed = [99,86,87,88,111,86,103,87,94,78,77,85,86]

x = numpy.mean(speed)

print(x)
```

运行实例

中值
中值是对所有值进行排序后的中间值：
77, 78, 85, 86, 86, 86, 87, 87, 88, 94, 99, 103, 111
在找到中位数之前，对数字进行排序很重要。
NumPy 模块拥有用于此目的的方法：
实例
请使用 NumPy median() 方法找到中间值：

```
import numpy

speed = [99,86,87,88,111,86,103,87,94,78,77,85,86]

x = numpy.median(speed)

print(x)
```

运行实例
如果中间有两个数字，则将这些数字之和除以 2。
77, 78, 85, 86, 86, 86, 87, 87, 94, 98, 99, 103

(86 + 87) / 2 = 86.5
实例
使用 NumPy 模块：

```
import numpy

speed = [99,86,87,88,86,103,87,94,78,77,85,86]

x = numpy.median(speed)

print(x)
```

运行实例

众数
众值是出现次数最多的值：
99, 86, 87, 88, 111, 86, 103, 87, 94, 78, 77, 85, 86 = 86
SciPy 模块拥有用于此目的的方法：
实例
请使用 SciPy mode() 方法查找出现次数最多的数字：

```
from scipy import stats

speed = [99,86,87,88,111,86,103,87,94,78,77,85,86]

x = stats.mode(speed)

print(x)
```

运行实例

章节总结
均值、中值和众数是机器学习中经常使用的技术，因此了解它们背后的概念很重要。

入门
标准差

什么是标准差？
标准差（Standard Deviation，又常称均方差）是一个数字，描述值的离散程度。
低标准偏差表示大多数数字接近均值（平均值）。
高标准偏差表示这些值分布在更宽的范围内。
例如：这次我们已经登记了 7 辆车的速度：

```
speed = [86,87,88,86,87,85,86]
```

标准差是：

```
0.9
```

意味着大多数值在平均值的 0.9 范围内，即 86.4。
让我们对范围更广的数字集合进行处理：

```
speed = [32,111,138,28,59,77,97]
```

标准差是：

```
37.85
```

这意味着大多数值都在平均值（平均值为 77.4）的 37.85 范围内。
如您所见，较高的标准偏差表示这些值分布在较宽的范围内。
NumPy 模块有一种计算标准差的方法：
实例
请使用 NumPy std() 方法查找标准差：

```
import numpy

speed = [86,87,88,86,87,85,86]

x = numpy.std(speed)

print(x)
```

运行实例

实例

```
import numpy

speed = [32,111,138,28,59,77,97]

x = numpy.std(speed)

print(x)
```

运行实例

方差

方差是另一种数字，指示值的分散程度。

实际上，如果采用方差的平方根，则会得到标准差！

或反之，如果将标准偏差乘以自身，则会得到方差！

如需计算方差，您必须执行以下操作：

1. 求均值：

(32+111+138+28+59+77+97) / 7 = 77.4

2. 对于每个值：找到与平均值的差：

 32 - 77.4 = -45.4

111 - 77.4 = 33.6

138 - 77.4 = 60.6

 28 - 77.4 = -49.4

 59 - 77.4 = -18.4

 77 - 77.4 = - 0.4

 97 - 77.4 = 19.6

3. 对于每个差异：找到平方值：

(-45.4)2 = 2061.16

 (33.6)2 = 1128.96

 (60.6)2 = 3672.36

(-49.4)2 = 2440.36

(-18.4)2 = 338.56

(- 0.4)2 =   0.16

 (19.6)2 = 384.16

4. 方差是这些平方差的平均值：

(2061.16+1128.96+3672.36+2440.36+338.56+0.16+384.16) / 7 = 1432.2

幸运的是，NumPy 有一种计算方差的方法：
实例
使用 NumPy var() 方法确定方差：

```
import numpy

speed = [32,111,138,28,59,77,97]

x = numpy.var(speed)

print(x)
```

运行实例

标准差
如我们所知，计算标准差的公式是方差的平方根：
√1432.25 = 37.85
或者，如上例所示，使用 NumPy 计算标准差：
实例
请使用 NumPy std() 方法查找标准差：

```
import numpy

speed = [32,111,138,28,59,77,97]

x = numpy.std(speed)

print(x)
```

运行实例

符号
标准差通常用 Sigma 符号表示： σ
方差通常由 Sigma Square 符号 σ2 表示

章节总结
标准差和方差是机器学习中经常使用的术语，因此了解如何获取它们以及它们背后的概念非常重要。

平均中位数模式
百分位数

什么是百分位数？
统计学中使用百分位数（Percentiles）为您提供一个数字，该数字描述了给定百分比值小于的值。

例如：假设我们有一个数组，包含住在一条街上的人的年龄。

```
ages =
[5,31,43,48,50,41,7,11,15,39,80,82,32,2,8,6,25,36,27,61,31]
```

什么是 75 百分位数？答案是 43，这意味着 75% 的人是 43 岁或以下。

NumPy 模块有一种用于找到指定百分位数的方法：

实例

使用 NumPy percentile() 方法查找百分位数：

```
import numpy

ages =
[5,31,43,48,50,41,7,11,15,39,80,82,32,2,8,6,25,36,27,61,31]

x = numpy.percentile(ages, 75)

print(x)
```

运行实例

实例

90% 的人口年龄是多少？

```
import numpy

ages =
[5,31,43,48,50,41,7,11,15,39,80,82,32,2,8,6,25,36,27,61,31]

x = numpy.percentile(ages, 90)

print(x)
```

运行实例

标准差

数据分布

数据分布（Data Distribution）

在本教程稍早之前，我们仅在例子中使用了非常少量的数据，目的是为了了解不同的概念。

在现实世界中，数据集要大得多，但是至少在项目的早期阶段，很难收集现实世界的数据。

我们如何获得大数据集？

为了创建用于测试的大数据集，我们使用 Python 模块 NumPy，该模块附带了许多创建任意大小的随机数据集的方法。

实例

创建一个包含 250 个介于 0 到 5 之间的随机浮点数的数组：

```
import numpy

x = numpy.random.uniform(0.0, 5.0, 250)

print(x)
```

运行实例

直方图

为了可视化数据集，我们可以对收集的数据绘制直方图。

我们将使用 Python 模块 Matplotlib 绘制直方图：

实例

绘制直方图：

```
import numpy
import matplotlib.pyplot as plt

x = numpy.random.uniform(0.0, 5.0, 250)

plt.hist(x, 5)
plt.show()
```

结果：

运行实例

直方图解释

我们使用上例中的数组绘制 5 条柱状图。

第一栏代表数组中有多少 0 到 1 之间的值。

第二栏代表有多少 1 到 2 之间的数值。

等等。

我们得到的结果是：

```
52 values are between 0 and 1
48 values are between 1 and 2
49 values are between 2 and 3
51 values are between 3 and 4
50 values are between 4 and 5
```

注释：数组值是随机数，不会在您的计算机上显示完全相同的结果。

大数据分布

包含 250 个值的数组被认为不是很大，但是现在您知道了如何创建一个随机值的集，并且通过更改参数，可以创建所需大小的数据集。

实例

创建一个具有 100000 个随机数的数组，并使用具有 100 栏的直方图显示它们：

```
import numpy
import matplotlib.pyplot as plt
```

```
x = numpy.random.uniform(0.0, 5.0, 100000)

plt.hist(x, 100)
plt.show()
```

运行实例

百分位数

正态数据分布

正态数据分布（Normal Data Distribution）

在上一章中，我们学习了如何创建给定大小且在两个给定值之间的完全随机数组。

在本章中，我们将学习如何创建一个将值集中在给定值周围的数组。

在概率论中,在数学家卡尔·弗里德里希·高斯(Carl Friedrich Gauss)提出了这种数据分布的公式之后，这种数据分布被称为正态数据分布或高斯数据分布。

实例

典型的正态数据分布：

```
import numpy
import matplotlib.pyplot as plt

x = numpy.random.normal(5.0, 1.0, 100000)

plt.hist(x, 100)
plt.show()
```

结果：

运行实例

注释：由于正态分布图具有钟形的特征形状，因此也称为钟形曲线。

直方图解释

我们使用 numpy.random.normal() 方法创建的数组（具有 100000 个值）绘制具有 100 栏的直方图。

我们指定平均值为 5.0，标准差为 1.0。

这意味着这些值应集中在 5.0 左右，并且很少与平均值偏离 1.0。

从直方图中可以看到，大多数值都在 4.0 到 6.0 之间，最高值大约是 5.0。

数据分布

散点图

散点图（Scatter Plot）
散点图是数据集中的每个值都由点表示的图。

Matplotlib 模块有一种绘制散点图的方法，它需要两个长度相同的数组，一个数组用于 x 轴的值，另一个数组用于 y 轴的值：

```
x = [5,7,8,7,2,17,2,9,4,11,12,9,6]

y = [99,86,87,88,111,86,103,87,94,78,77,85,86]
```

x 数组代表每辆汽车的年龄。
y 数组表示每个汽车的速度。
实例
请使用 scatter() 方法绘制散点图：

```
import matplotlib.pyplot as plt

x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]

plt.scatter(x, y)
plt.show()
```

结果：

运行实例
散点图解释
x 轴表示车龄，y 轴表示速度。
从图中可以看到，两辆最快的汽车都使用了 2 年，最慢的汽车使用了 12 年。
注释：汽车似乎越新，驾驶速度就越快，但这可能是一个巧合，毕竟我们只注册了 13 辆汽车。

随机数据分布
在机器学习中，数据集可以包含成千上万甚至数百万个值。
测试算法时，您可能没有真实的数据，您可能必须使用随机生成的值。
正如我们在上一章中学到的那样，NumPy 模块可以帮助我们！
让我们创建两个数组，它们都填充有来自正态数据分布的 1000 个随机数。
第一个数组的平均值设置为 5.0，标准差为 1.0。
第二个数组的平均值设置为 10.0，标准差为 2.0：
实例
有 1000 个点的散点图：

```
import numpy
import matplotlib.pyplot as plt
```

```
x = numpy.random.normal(5.0, 1.0, 1000)
y = numpy.random.normal(10.0, 2.0, 1000)

plt.scatter(x, y)
plt.show()
```

结果：

运行实例

散点图解释

我们可以看到，点集中在 x 轴上的值 5 和 y 轴上的 10 周围。

我们还可以看到，在 y 轴上扩散得比在 x 轴上更大。

正态数据分布

线性回归

回归

当您尝试找到变量之间的关系时，会用到术语“回归”（regression）。

在机器学习和统计建模中，这种关系用于预测未来事件的结果。

线性回归

线性回归使用数据点之间的关系在所有数据点之间画一条直线。

这条线可以用来预测未来的值。

在机器学习中，预测未来非常重要。

工作原理

Python 提供了一些方法来查找数据点之间的关系并绘制线性回归线。

我们将向您展示如何使用这些方法而不是通过数学公式。

在下面的示例中，x 轴表示车龄，y 轴表示速度。我们已经记录了 13 辆汽车通过收费站时的车龄和速度。让我们看看我们收集的数据是否可以用于线性回归：

实例

首先绘制散点图：

```
import matplotlib.pyplot as plt

x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]

plt.scatter(x, y)
plt.show()
```

结果：

运行实例
实例
导入 scipy 并绘制线性回归线：

```
import matplotlib.pyplot as plt
from scipy import stats

x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]

slope, intercept, r, p, std_err = stats.linregress(x, y)

def myfunc(x):
  return slope * x + intercept

mymodel = list(map(myfunc, x))

plt.scatter(x, y)
plt.plot(x, mymodel)
plt.show()
```

结果：

运行实例
例子解释
导入所需模块：

```
import matplotlib.pyplot as plt
from scipy import stats
```

创建表示 x 和 y 轴值的数组：

```
x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]
```

执行一个方法，该方法返回线性回归的一些重要键值：

```
slope, intercept, r, p, std_err = stats.linregress(x, y)
```

创建一个使用 slope 和 intercept 值的函数返回新值。这个新值表示相应的 x 值将在 y 轴上放置的位置：

```
def myfunc(x):
  return slope * x + intercept
```

通过函数运行 x 数组的每个值。这将产生一个新的数组，其中的 y 轴具有新值：

```
mymodel = list(map(myfunc, x))
```

绘制原始散点图：

```
plt.scatter(x, y)
```

绘制线性回归线：

```
plt.plot(x, mymodel)
```

显示图：
```
plt.show()
```

R-Squared
重要的是要知道 x 轴的值和 y 轴的值之间的关系有多好，如果没有关系，则线性回归不能用于预测任何东西。
该关系用一个称为 r 平方（r-squared）的值来度量。
r 平方值的范围是 0 到 1，其中 0 表示不相关，而 1 表示 100% 相关。
Python 和 Scipy 模块将为您计算该值，您所要做的就是将 x 和 y 值提供给它：
实例
我的数据在线性回归中的拟合度如何？
```
from scipy import stats

x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]

slope, intercept, r, p, std_err = stats.linregress(x, y)

print(r)
```
运行实例
注释：结果 -0.76 表明存在某种关系，但不是完美的关系，但它表明我们可以在将来的预测中使用线性回归。

预测未来价值
现在，我们可以使用收集到的信息来预测未来的值。
例如：让我们尝试预测一辆拥有 10 年历史的汽车的速度。
为此，我们需要与上例中相同的 myfunc() 函数：
```
def myfunc(x):
  return slope * x + intercept
```
实例
预测一辆有 10 年车龄的汽车的速度：
```
from scipy import stats

x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]

slope, intercept, r, p, std_err = stats.linregress(x, y)

def myfunc(x):
  return slope * x + intercept

speed = myfunc(10)
```

```
print(speed)
```

运行实例

该例预测速度为 85.6，我们也可以从图中读取：

糟糕的拟合度？

让我们创建一个实例，其中的线性回归并不是预测未来值的最佳方法。

实例

x 和 y 轴的这些值将导致线性回归的拟合度非常差：

```
import matplotlib.pyplot as plt
from scipy import stats

x = [89,43,36,36,95,10,66,34,38,20,26,29,48,64,6,5,36,66,72,40]
y = [21,46,3,35,67,95,53,72,58,10,26,34,90,33,38,20,56,2,47,15]

slope, intercept, r, p, std_err = stats.linregress(x, y)

def myfunc(x):
  return slope * x + intercept

mymodel = list(map(myfunc, x))

plt.scatter(x, y)
plt.plot(x, mymodel)
plt.show()
```

结果：

运行实例

以及 r-squared 值？

实例

您应该得到了一个非常低的 r-squared 值。

```
import numpy
from scipy import stats

x = [89,43,36,36,95,10,66,34,38,20,26,29,48,64,6,5,36,66,72,40]
y = [21,46,3,35,67,95,53,72,58,10,26,34,90,33,38,20,56,2,47,15]

slope, intercept, r, p, std_err = stats.linregress(x, y)

print(r)
```

运行实例

结果：0.013 表示关系很差，并告诉我们该数据集不适合线性回归。

散点图

多项式回归

多项式回归（Polynomial Regression）
如果您的数据点显然不适合线性回归（穿过数据点之间的直线），那么多项式回归可能是理想的选择。
像线性回归一样，多项式回归使用变量 x 和 y 之间的关系来找到绘制数据点线的最佳方法。

工作原理
Python 有一些方法可以找到数据点之间的关系并画出多项式回归线。我们将向您展示如何使用这些方法而不是通过数学公式。
在下面的例子中，我们注册了 18 辆经过特定收费站的汽车。
我们已经记录了汽车的速度和通过时间（小时）。
x 轴表示一天中的小时，y 轴表示速度：
实例
首先绘制散点图：

```
import matplotlib.pyplot as plt

x = [1,2,3,5,6,7,8,9,10,12,13,14,15,16,18,19,21,22]
y = [100,90,80,60,60,55,60,65,70,70,75,76,78,79,90,99,99,100]

plt.scatter(x, y)
plt.show()
```

结果：

运行实例
实例
导入 numpy 和 matplotlib，然后画出多项式回归线：

```
import numpy
import matplotlib.pyplot as plt

x = [1,2,3,5,6,7,8,9,10,12,13,14,15,16,18,19,21,22]
y = [100,90,80,60,60,55,60,65,70,70,75,76,78,79,90,99,99,100]

mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))

myline = numpy.linspace(1, 22, 100)

plt.scatter(x, y)
plt.plot(myline, mymodel(myline))
plt.show()
```

结果：

运行实例
例子解释
导入所需模块：

```
import numpy
import matplotlib.pyplot as plt
```

创建表示 x 和 y 轴值的数组：

```
x = [1,2,3,5,6,7,8,9,10,12,13,14,15,16,18,19,21,22]
y = [100,90,80,60,60,55,60,65,70,70,75,76,78,79,90,99,99,100]
```

NumPy 有一种方法可以让我们建立多项式模型：

```
mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))
```

然后指定行的显示方式，我们从位置 1 开始，到位置 22 结束：

```
myline = numpy.linspace(1, 22, 100)
```

绘制原始散点图：

```
plt.scatter(x, y)
```

画出多项式回归线：

```
plt.plot(myline, mymodel(myline))
```

显示图表：

```
plt.show()
```

R-Squared
重要的是要知道 x 轴和 y 轴的值之间的关系有多好，如果没有关系，则多项式回归不能用于预测任何东西。
该关系用一个称为 r 平方（ r-squared）的值来度量。
r 平方值的范围是 0 到 1，其中 0 表示不相关，而 1 表示 100% 相关。
Python 和 Sklearn 模块将为您计算该值，您所要做的就是将 x 和 y 数组输入：
实例
我的数据在多项式回归中的拟合度如何？

```
import numpy
from sklearn.metrics import r2_score

x = [1,2,3,5,6,7,8,9,10,12,13,14,15,16,18,19,21,22]
y = [100,90,80,60,60,55,60,65,70,70,75,76,78,79,90,99,99,100]

mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))

print(r2_score(y, mymodel(x)))
```

运行实例
注释：结果 0.94 表明存在很好的关系，我们可以在将来的预测中使用多项式回归。

预测未来值
现在，我们可以使用收集到的信息来预测未来的值。
例如：让我们尝试预测在晚上 17 点左右通过收费站的汽车的速度：
为此，我们需要与上面的实例相同的 mymodel 数组：

```
mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))
```

实例
预测下午 17 点过车的速度：

```
import numpy
from sklearn.metrics import r2_score

x = [1,2,3,5,6,7,8,9,10,12,13,14,15,16,18,19,21,22]
y = [100,90,80,60,60,55,60,65,70,70,75,76,78,79,90,99,99,100]

mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))

speed = mymodel(17)
print(speed)
```

运行实例
该例预测速度为 88.87，我们也可以在图中看到：

糟糕的拟合度？
让我们创建一个实例，其中多项式回归不是预测未来值的最佳方法。
实例
x 和 y 轴的这些值会导致多项式回归的拟合度非常差：

```
import numpy
import matplotlib.pyplot as plt

x = [89,43,36,36,95,10,66,34,38,20,26,29,48,64,6,5,36,66,72,40]
y = [21,46,3,35,67,95,53,72,58,10,26,34,90,33,38,20,56,2,47,15]

mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))

myline = numpy.linspace(2, 95, 100)

plt.scatter(x, y)
plt.plot(myline, mymodel(myline))
plt.show()
```

结果：

运行实例
r-squared 值呢？
实例
您应该得到一个非常低的 r-squared 值。

```
import numpy
```

```
from sklearn.metrics import r2_score

x = [89,43,36,36,95,10,66,34,38,20,26,29,48,64,6,5,36,66,72,40]
y = [21,46,3,35,67,95,53,72,58,10,26,34,90,33,38,20,56,2,47,15]

mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))

print(r2_score(y, mymodel(x)))
```

运行实例

结果：0.00995 表示关系很差，并告诉我们该数据集不适合多项式回归。

线性回归

多元回归

多元回归（Multiple Regression）

多元回归就像线性回归一样，但是具有多个独立值，这意味着我们试图基于两个或多个变量来预测一个值。

请看下面的数据集，其中包含了一些有关汽车的信息。

```
Car Model Volume Weight
Toyota Aygo 1000 790
Mitsubishi Space Star 1200 1160
Skoda Citigo 1000 929
Fiat 500 900 865
Mini Cooper 1500 1140
VW Up! 1000 929
Skoda Fabia 1400 1109
Mercedes A-Class 1500 1365
Ford Fiesta 1500 1112
Audi A1 1600 1150
Hyundai I20 1100 980
Suzuki Swift 1300 990
Ford Fiesta 1000 1112
Honda Civic 1600 1252
Hundai I30 1600 1326
Opel Astra 1600 1330
BMW 1 1600 1365
Mazda 3 2200 1280
Skoda Rapid 1600 1119
Ford Focus 2000 1328
Ford Mondeo 1600 1584
```

```
Opel Insignia 2000 1428
Mercedes C-Class 2100 1365
Skoda Octavia 1600 1415
Volvo S60 2000 1415
Mercedes CLA 1500 1465
Audi A4 2000 1490
Audi A6 2000 1725
Volvo V70 1600 1523
BMW 5 2000 1705
Mercedes E-Class 2100 1605
Volvo XC70 2000 1746
Ford B-Max 1600 1235
BMW 2 1600 1390
Opel Zafira 1600 1405
Mercedes SLK 2500 1395
```

我们可以根据发动机排量的大小预测汽车的二氧化碳排放量，但是通过多元回归，我们可以引入更多变量，例如汽车的重量，以使预测更加准确。

工作原理

在 Python 中，我们拥有可以完成这项工作的模块。首先导入 Pandas 模块：

```
import pandas
```

Pandas 模块允许我们读取 csv 文件并返回一个 DataFrame 对象。

此文件仅用于测试目的，您可以在此处下载：cars.csv

```
df = pandas.read_csv("cars.csv")
```

然后列出独立值，并将这个变量命名为 X。

将相关值放入名为 y 的变量中。

```
X = df[['Weight', 'Volume']]
y = df['CO2']
```

提示：通常，将独立值列表命名为大写 X，将相关值列表命名为小写 y。

我们将使用 sklearn 模块中的一些方法，因此我们也必须导入该模块：

```
from sklearn import linear_model
```

在 sklearn 模块中，我们将使用 LinearRegression() 方法创建一个线性回归对象。

该对象有一个名为 fit() 的方法，该方法将独立值和从属值作为参数，并用描述这种关系的数据填充回归对象：

```
regr = linear_model.LinearRegression()
regr.fit(X, y)
```

现在，我们有了一个回归对象，可以根据汽车的重量和排量预测 CO2 值：

```
# 预测重量为 2300kg、排量为 1300ccm 的汽车的二氧化碳排放量：

predictedCO2 = regr.predict([[2300, 1300]])
```

实例

请看完整实例：

```
import pandas
from sklearn import linear_model

df = pandas.read_csv("cars.csv")

X = df[['Weight', 'Volume']]
y = df['CO2']

regr = linear_model.LinearRegression()
regr.fit(X, y)

# 预测重量为 2300kg、排量为 1300ccm 的汽车的二氧化碳排放量：

predictedCO2 = regr.predict([[2300, 1300]])

print(predictedCO2)
```

结果：

```
[107.2087328]
```

运行实例

我们预测，配备 1.3 升发动机，重量为 2300 千克的汽车，每行驶 1 公里，就会释放约 107 克二氧化碳。

系数

系数是描述与未知变量的关系的因子。

例如：如果 x 是变量，则 2x 是 x 的两倍。x 是未知变量，数字 2 是系数。

在这种情况下，我们可以要求重量相对于 CO2 的系数值，以及体积相对于 CO2 的系数值。我们得到的答案告诉我们，如果我们增加或减少其中一个独立值，将会发生什么。

实例

打印回归对象的系数值：

```
import pandas
from sklearn import linear_model

df = pandas.read_csv("cars.csv")

X = df[['Weight', 'Volume']]
y = df['CO2']

regr = linear_model.LinearRegression()
regr.fit(X, y)

print(regr.coef_)
```

结果：
[0.00755095 0.00780526]
运行实例
结果解释
结果数组表示重量和排量的系数值。
Weight: 0.00755095
Volume: 0.00780526
这些值告诉我们，如果重量增加 1g，则 CO2 排放量将增加 0.00755095g。
如果发动机尺寸（容积）增加 1 ccm，则 CO2 排放量将增加 0.00780526g。
我认为这是一个合理的猜测，但还是请进行测试！
我们已经预言过，如果一辆配备 1300ccm 发动机的汽车重 2300 千克，则二氧化碳排放量将约为 107 克。
如果我们增加 1000g 的重量会怎样？
实例
复制之前的例子，但是将车重从 2300 更改为 3300：

```
import pandas
from sklearn import linear_model

df = pandas.read_csv("cars.csv")

X = df[['Weight', 'Volume']]
y = df['CO2']

regr = linear_model.LinearRegression()
regr.fit(X, y)

predictedCO2 = regr.predict([[3300, 1300]])

print(predictedCO2)
```

结果：
[114.75968007]
运行实例
我们已经预测，配备 1.3 升发动机，重量为 3.3 吨的汽车，每行驶 1 公里，就会释放约 115 克二氧化碳。
这表明 0.00755095 的系数是正确的：
107.2087328 + (1000 * 0.00755095) = 114.75968

多项式回归
缩放

特征缩放（Scale Features）
当您的数据拥有不同的值，甚至使用不同的度量单位时，可能很难比较它们。与米相比，公斤是多少？或者海拔比较时间呢？
这个问题的答案是缩放。我们可以将数据缩放为易于比较的新值。
请看下表，它与我们在多元回归一章中使用的数据集相同，但是这次，Volume 列包含的单位是升，而不是 ccm（1.0 而不是 1000）。

```
Car Model Volume Weight
Toyota Aygo 1.0 790
Mitsubishi Space Star 1.2 1160
Skoda Citigo 1.0 929
Fiat 500 0.9 865
Mini Cooper 1.5 1140
VW Up! 1.0 929
Skoda Fabia 1.4 1109
Mercedes A-Class 1.5 1365
Ford Fiesta 1.5 1112
Audi A1 1.6 1150
Hyundai I20 1.1 980
Suzuki Swift 1.3 990
Ford Fiesta 1.0 1112
Honda Civic 1.6 1252
Hundai I30 1.6 1326
Opel Astra 1.6 1330
BMW 1 1.6 1365
Mazda 3 2.2 1280
Skoda Rapid 1.6 1119
Ford Focus 2.0 1328
Ford Mondeo 1.6 1584
Opel Insignia 2.0 1428
Mercedes C-Class 2.1 1365
Skoda Octavia 1.6 1415
Volvo S60 2.0 1415
Mercedes CLA 1.5 1465
Audi A4 2.0 1490
Audi A6 2.0 1725
Volvo V70 1.7 1523
BMW 5 2.0 1705
Mercedes E-Class 2.1 1605
Volvo XC70 2.0 1746
Ford B-Max 1.6 1235
BMW 2 1.6 1390
Opel Zafira 1.6 1405
Mercedes SLK 2.5 1395
```

很难将排量 1.0 与车重 790 进行比较，但是如果将它们都缩放为可比较的值，我们可以很容易地看到一个值与另一个值相比有多少。
缩放数据有多种方法，在本教程中，我们将使用一种称为标准化（standardization）的方法。
标准化方法使用以下公式：
z = (x - u) / s
其中 z 是新值，x 是原始值，u 是平均值，s 是标准差。
如果从上述数据集中获取 weight 列，则第一个值为 790，缩放后的值为：
(790 - 1292.23) / 238.74 = -2.1
如果从上面的数据集中获取 volume 列，则第一个值为 1.0，缩放后的值为：
(1.0 - 1.61) / 0.38 = -1.59
现在，您可以将 -2.1 与 -1.59 相比较，而不是比较 790 与 1.0。
您不必手动执行此操作，Python sklearn 模块有一个名为 StandardScaler() 的方法，该方法返回带有转换数据集方法的 Scaler 对象。
实例
缩放 Weight 和 Volume 列中的所有值：

```
import pandas
from sklearn import linear_model
from sklearn.preprocessing import StandardScaler
scale = StandardScaler()

df = pandas.read_csv("cars2.csv")

X = df[['Weight', 'Volume']]

scaledX = scale.fit_transform(X)

print(scaledX)
```

结果：
请注意，前两个值是 -2.1 和 -1.59，与我们的计算相对应：

```
[[-2.10389253 -1.59336644]
 [-0.55407235 -1.07190106]
 [-1.52166278 -1.59336644]
 [-1.78973979 -1.85409913]
 [-0.63784641 -0.28970299]
 [-1.52166278 -1.59336644]
 [-0.76769621 -0.55043568]
 [ 0.3046118  -0.28970299]
 [-0.7551301  -0.28970299]
 [-0.59595938 -0.0289703 ]
 [-1.30803892 -1.33263375]
```

```
 [-1.26615189 -0.81116837]
 [-0.7551301  -1.59336644]
 [-0.16871166 -0.0289703 ]
 [ 0.14125238 -0.0289703 ]
 [ 0.15800719 -0.0289703 ]
 [ 0.3046118  -0.0289703 ]
 [-0.05142797  1.53542584]
 [-0.72580918 -0.0289703 ]
 [ 0.14962979  1.01396046]
 [ 1.2219378  -0.0289703 ]
 [ 0.5685001   1.01396046]
 [ 0.3046118   1.27469315]
 [ 0.51404696 -0.0289703 ]
 [ 0.51404696  1.01396046]
 [ 0.72348212 -0.28970299]
 [ 0.8281997   1.01396046]
 [ 1.81254495  1.01396046]
 [ 0.96642691 -0.0289703 ]
 [ 1.72877089  1.01396046]
 [ 1.30990057  1.27469315]
 [ 1.90050772  1.01396046]
 [-0.23991961 -0.0289703 ]
 [ 0.40932938 -0.0289703 ]
 [ 0.47215993 -0.0289703 ]
 [ 0.4302729   2.31762392]]
```

运行实例

预测 CO2 值

多元回归一章的任务是在仅知道汽车的重量和排量的情况下预测其排放的二氧化碳。

缩放数据集后，在预测值时必须使用缩放比例：

实例

预测一辆重 2300 公斤的 1.3 升汽车的二氧化碳排放量：

```
import pandas
from sklearn import linear_model
from sklearn.preprocessing import StandardScaler
scale = StandardScaler()

df = pandas.read_csv("cars2.csv")

X = df[['Weight', 'Volume']]
y = df['CO2']

scaledX = scale.fit_transform(X)
```

```
regr = linear_model.LinearRegression()
regr.fit(scaledX, y)

scaled = scale.transform([[2300, 1.3]])

predictedCO2 = regr.predict([scaled[0]])
print(predictedCO2)
```

结果：

```
[107.2087328]
```

运行实例

多元回归

训练/测试

评估模型

在机器学习中，我们创建模型来预测某些事件的结果，就像在上一章中当我们了解重量和发动机排量时，预测了汽车的二氧化碳排放量一样。

要衡量模型是否足够好，我们可以使用一种称为训练/测试的方法。

什么是训练/测试

训练/测试是一种测量模型准确性的方法。

之所以称为训练/测试，是因为我们将数据集分为两组：训练集和测试集。

80% 用于训练，20% 用于测试。

您可以使用训练集来训练模型。

您可以使用测试集来测试模型。

训练模型意味着创建模型。

测试模型意味着测试模型的准确性。

从数据集开始

从要测试的数据集开始。

我们的数据集展示了商店中的 100 位顾客及其购物习惯。

实例

```
import numpy
import matplotlib.pyplot as plt
numpy.random.seed(2)

x = numpy.random.normal(3, 1, 100)
y = numpy.random.normal(150, 40, 100) / x
```

```
plt.scatter(x, y)
plt.show()
```

结果：

x 轴表示购买前的分钟数。

y 轴表示在购买上花费的金额。

运行实例

拆分训练/测试

训练集应该是原始数据的 80% 的随机选择。

测试集应该是剩余的 20%。

```
train_x = x[:80]
train_y = y[:80]

test_x = x[80:]
test_y = y[80:]
```

显示训练集

显示与训练集相同的散点图：

实例

```
plt.scatter(train_x, train_y)
plt.show()
```

结果：

它看起来像原始数据集，因此似乎是一个合理的选择：

运行实例

显示测试集

为了确保测试集不是完全不同，我们还要看一下测试集。

实例

```
plt.scatter(test_x, test_y)
plt.show()
```

结果：

测试集也看起来像原始数据集：

运行实例

拟合数据集

数据集是什么样的？我认为最合适拟合的是多项式回归，因此让我们画一条多项式回归线。

要通过数据点画一条线，我们使用 matplotlib 模块的 plott() 方法：

实例

绘制穿过数据点的多项式回归线：

```
import numpy
```

```
import matplotlib.pyplot as plt
numpy.random.seed(2)

x = numpy.random.normal(3, 1, 100)
y = numpy.random.normal(150, 40, 100) / x

train_x = x[:80]
train_y = y[:80]

test_x = x[80:]
test_y = y[80:]

mymodel = numpy.poly1d(numpy.polyfit(train_x, train_y, 4))

myline = numpy.linspace(0, 6, 100)

plt.scatter(train_x, train_y)
plt.plot(myline, mymodel(myline))
plt.show()
```

结果：

运行实例

此结果可以支持我们对数据集拟合多项式回归的建议，即使如果我们尝试预测数据集之外的值会给我们带来一些奇怪的结果。例如：该行表明某位顾客在商店购物 6 分钟，会完成一笔价值 200 的购物。这可能是过拟合的迹象。

但是 R-squared 分数呢？ R-squared score 很好地指示了我的数据集对模型的拟合程度。

R2

还记得 R2，也称为 R 平方（R-squared）吗？

它测量 x 轴和 y 轴之间的关系，取值范围从 0 到 1，其中 0 表示没有关系，而 1 表示完全相关。

sklearn 模块有一个名为 rs_score() 的方法，该方法将帮助我们找到这种关系。

在这里，我们要衡量顾客在商店停留的时间与他们花费多少钱之间的关系。

实例

我们的训练数据在多项式回归中的拟合度如何？

```
import numpy
from sklearn.metrics import r2_score
numpy.random.seed(2)

x = numpy.random.normal(3, 1, 100)
```

```
y = numpy.random.normal(150, 40, 100) / x

train_x = x[:80]
train_y = y[:80]

test_x = x[80:]
test_y = y[80:]

mymodel = numpy.poly1d(numpy.polyfit(train_x, train_y, 4))

r2 = r2_score(train_y, mymodel(train_x))

print(r2)
```

运行实例

注释：结果 0.799 显示关系不错。

引入测试集

现在，至少在训练数据方面，我们已经建立了一个不错的模型。

然后，我们要使用测试数据来测试模型，以检验是否给出相同的结果。

实例

让我们在使用测试数据时确定 R2 分数：

```
import numpy
from sklearn.metrics import r2_score
numpy.random.seed(2)

x = numpy.random.normal(3, 1, 100)
y = numpy.random.normal(150, 40, 100) / x

train_x = x[:80]
train_y = y[:80]

test_x = x[80:]
test_y = y[80:]

mymodel = numpy.poly1d(numpy.polyfit(train_x, train_y, 4))

r2 = r2_score(test_y, mymodel(test_x))

print(r2)
```

运行实例

注释：结果 0.809 表明该模型也适合测试集，我们确信可以使用该模型预测未来值。

预测值

现在我们已经确定我们的模型是不错的，可以开始预测新值了。
实例
如果购买客户在商店中停留 5 分钟，他/她将花费多少钱？
print(mymodel(5))
运行实例
该例预测客户花费了 22.88 美元，似乎与图表相对应：

缩放
决策树

决策树（Decision Tree）
在本章中，我们将向您展示如何制作“决策树”。决策树是一种流程图，可以帮助您根据以前的经验进行决策。
在这个例子中，一个人将尝试决定他/她是否应该参加喜剧节目。
幸运的是，我们的例中人物每次在镇上举办喜剧节目时都进行注册，并注册一些关于喜剧演员的信息，并且还登记了他/她是否去过。
Age Experience Rank Nationality
36 10 9 UK
42 12 4 USA
23 4 6 N
52 4 4 USA
43 21 8 USA
44 14 5 UK
66 3 7 N
35 14 9 UK
52 13 7 N
35 5 9 N
24 3 5 USA
18 3 7 UK
45 9 9 UK
现在，基于此数据集，Python 可以创建决策树，这个决策树可用于决定是否值得参加任何新的演出。

工作原理
首先，导入所需的模块，并使用 pandas 读取数据集：
实例
读取并打印数据集：
import pandas
from sklearn import tree

```
import pydotplus
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt
import matplotlib.image as pltimg

df = pandas.read_csv("shows.csv")

print(df)
```

运行实例

如需制作决策树，所有数据都必须是数字。

我们必须将非数字列 “Nationality” 和 “Go” 转换为数值。

Pandas 有一个 map() 方法，该方法接受字典，其中包含有关如何转换值的信息。

```
{'UK': 0, 'USA': 1, 'N': 2}
```

表示将值 'UK' 转换为 0，将 'USA' 转换为 1，将 'N' 转换为 2。

实例

将字符串值更改为数值：

```
d = {'UK': 0, 'USA': 1, 'N': 2}
df['Nationality'] = df['Nationality'].map(d)
d = {'YES': 1, 'NO': 0}
df['Go'] = df['Go'].map(d)

print(df)
```

运行实例

然后，我们必须将特征列与目标列分开。

特征列是我们尝试从中预测的列，目标列是具有我们尝试预测的值的列。

实例

X 是特征列，y 是目标列：

```
features = ['Age', 'Experience', 'Rank', 'Nationality']

X = df[features]
y = df['Go']

print(X)
print(y)
```

运行实例

现在，我们可以创建实际的决策树，使其适合我们的细节，然后在计算机上保存一个 .png 文件：

实例

创建一个决策树，将其另存为图像，然后显示该图像：

```
dtree = DecisionTreeClassifier()
dtree = dtree.fit(X, y)
data = tree.export_graphviz(dtree, out_file=None,
feature_names=features)
```

```
graph = pydotplus.graph_from_dot_data(data)
graph.write_png('mydecisiontree.png')

img=pltimg.imread('mydecisiontree.png')
imgplot = plt.imshow(img)
plt.show()
```

运行实例

结果解释

决策树使用您先前的决策来计算您是否愿意去看喜剧演员的几率。

让我们阅读决策树的不同方面：

Rank

Rank <= 6.5 表示排名在 6.5 以下的喜剧演员将遵循 True 箭头（向左），其余的则遵循 箭头（向右）。

gini = 0.497 表示分割的质量，并且始终是 0.0 到 0.5 之间的数字，其中 0.0 表示所有样本均得到相同的结果，而 0.5 表示分割完全在中间进行。

samples = 13 表示在决策的这一点上还剩下 13 位喜剧演员，因为这是第一步，所以他们全部都是喜剧演员。

value = [6, 7] 表示在这 13 位喜剧演员中，有 6 位将获得 "NO"，而 7 位将获得 "GO"。

Gini

分割样本的方法有很多，我们在本教程中使用 GINI 方法。

基尼方法使用以下公式：

Gini = 1 - (x/n)2 - (y/n)2

其中，x 是肯定答案的数量 ("GO")，n 是样本数量，y 是否定答案的数量 ("NO")，使用以下公式进行计算：

1 - (7 / 13)2 - (6 / 13)2 = 0.497

下一步包含两个框，其中一个框用于喜剧演员，其 'Rank' 为 6.5 或更低，其余为一个框。

True - 5 名喜剧演员在这里结束：

gini = 0.0 表示所有样本均得到相同的结果。

samples = 5 表示该分支中还剩下 5 位喜剧演员（5 位的等级为 6.5 或更低的喜剧演员）。

value = [5, 0] 表示 5 得到 "NO" 而 0 得到 "GO"。

False - 8 位戏剧演员继续：

Nationality（国籍）

Nationality <= 0.5 表示国籍值小于 0.5 的喜剧演员将遵循左箭头（这表示来自英国的所有人），其余的将遵循右箭头。

gini = 0.219 意味着大约 22% 的样本将朝一个方向移动。

samples = 8 表示该分支中还剩下 8 个喜剧演员（8 个喜剧演员的等级高于 6.5）。

value = [1, 7] 表示在这 8 位喜剧演员中，1 位将获得 "NO"，而 7 位将获得 "GO"。

True - 4 名戏剧演员继续：
Age（年龄）
Age <= 35.5 表示年龄在 35.5 岁或以下的喜剧演员将遵循左箭头，其余的将遵循右箭头。
gini = 0.375 意味着大约 37.5% 的样本将朝一个方向移动。
samples = 4 表示该分支中还剩下 4 位喜剧演员（来自英国的 4 位喜剧演员）。
value = [1, 3] 表示在这 4 位喜剧演员中，1 位将获得 "NO"，而 3 位将获得 "GO"。
False - 4 名喜剧演员到这里结束：
gini = 0.0 表示所有样本都得到相同的结果。
samples = 4 表示该分支中还剩下 4 位喜剧演员（来自英国的 4 位喜剧演员）。
value = [0, 4] 表示在这 4 位喜剧演员中，0 将获得 "NO"，而 4 将获得 "GO"。

True - 2 名喜剧演员在这里结束：
gini = 0.0 表示所有样本都得到相同的结果。
samples = 2 表示该分支中还剩下 2 名喜剧演员（2 名 35.5 岁或更年轻的喜剧演员）。
value = [0, 2] 表示在这 2 位喜剧演员中，0 将获得 "NO"，而 2 将获得 "GO"。
False - 2 名戏剧演员继续：
Experience（经验）
Experience <= 9.5 表示具有 9.5 年或以上经验的喜剧演员将遵循左侧的箭头，其余的将遵循右侧的箭头。
gini = 0.5 表示 50% 的样本将朝一个方向移动。
samples = 2 表示此分支中还剩下 2 个喜剧演员（2 个年龄超过 35.5 的喜剧演员）。
value = [1, 1] 表示这两个喜剧演员中，1 将获得 "NO"，而 1 将获得 "GO"。

True - 1 名喜剧演员在这里结束：
gini = 0.0 表示所有样本都得到相同的结果。
samples = 1 表示此分支中还剩下 1 名喜剧演员（1 名具有 9.5 年或以下经验的喜剧演员）。
value = [0, 1] 表示 0 表示 "NO"，1 表示 "GO"。
False - 1 名喜剧演员到这里为止：
gini = 0.0 表示所有样本都得到相同的结果。
samples = 1 表示此分支中还剩下 1 位喜剧演员（其中 1 位具有超过 9.5 年经验的喜剧演员）。

value = [1, 0] 表示 1 表示 "NO"，0 表示 "GO"。

预测值
我们可以使用决策树来预测新值。
例如：我是否应该去看一个由 40 岁的美国喜剧演员主演的节目，该喜剧演员有 10 年的经验，喜剧排名为 7？
实例
使用 predict() 方法来预测新值：
```
print(dtree.predict([[40, 10, 7, 1]]))
```
运行实例
实例
如果喜剧等级为 6，答案是什么？
```
print(dtree.predict([[40, 10, 6, 1]]))
```
运行实例

不同的结果
如果运行足够多次，即使您输入的数据相同，决策树也会为您提供不同的结果。
这是因为决策树无法给我们 100% 的肯定答案。它基于结果的可能性，答案会有所不同。

训练/测试
MySQL 入门

Python 可以在数据库应用程序中使用。
MySQL 是最受欢迎的数据库之一。

MySQL 数据库
为了能够试验本教程中的代码示例，您应该在计算机上安装 MySQL。
请在这里下载免费的 MySQL 数据库：https://www.mysql.com/downloads/。

安装 MySQL 驱动程序
Python 需要 MySQL 驱动程序来访问 MySQL 数据库。
在本教程中，我们将使用驱动程序 "MySQL Connector"。
我们建议您使用 PIP 安装 "MySQL Connector"。
PIP 很可能已经安装在 Python 环境中。
将命令行导航到 PIP 的位置，然后键入以下内容：
下载并安装 "MySQL Connector"：
```
C:\...\AppData\Local\Programs\Python\Python36-32\Scripts>python -m pip install mysql-connector
```

现在，您已下载并安装 MySQL 驱动程序。

测试 MySQL Connector
如需测试安装是否成功，或者您是否已安装 "MySQL Connector"，请创建一张包含以下内容的 Python 页面：
demo_mysql_test.py:
```
import mysql.connector
```
运行实例
如果执行上述代码没有错误，则 "MySQL Connector" 已安装并待用。

创建连接
首先创建与数据库的连接。
使用 MySQL 数据库中的用户名和密码：
demo_mysql_connection.py:
```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword"
)

print(mydb)
```
运行实例
现在，您可以开始使用 SQL 语句查询数据库了。

决策树
MySQL Create Database

创建数据库
如需在 MySQL 中创建数据库，请使用 "CREATE DATABASE" 语句：
实例
创建名为 "mydatabase" 的数据库：
```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword"
)
```

```
mycursor = mydb.cursor()

mycursor.execute("CREATE DATABASE mydatabase")
```

运行实例

如果执行上面的代码没有错误，则您已成功创建数据库。

检查数据库是否存在

您可以通过使用 "SHOW DATABASES" 语句列出系统中的所有数据库，检查数据库是否存在：

实例

返回系统中的数据库列表：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword"
)

mycursor = mydb.cursor()

mycursor.execute("SHOW DATABASES")

for x in mycursor:
  print(x)
```

运行实例

或者您可以在建立连接时尝试访问数据库：

实例

尝试连接数据库 "mydatabase"：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)
```

运行实例

如果数据库不存在，会收到错误。

MySQL 入门

MySQL Create Table

创建表
如需在 MySQL 中创建表，请使用 "CREATE TABLE" 语句。
请确保在创建连接时定义数据库的名称。
实例
创建表 "customers"：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

mycursor.execute("CREATE TABLE customers (name VARCHAR(255),
address VARCHAR(255))")
```

运行实例
如果执行上面的代码没有错误，那么您现在已经成功创建了一个表。

检查表是否存在
您可以通过使用 "SHOW TABLES" 语句列出数据库中的所有表，来检查表是否存在：
实例
返回系统中的数据库列表：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

mycursor.execute("SHOW TABLES")

for x in mycursor:
  print(x)
```

运行实例

主键
创建表时，您还应该为每条记录创建一个具有唯一键的列。
这可以通过定义 PRIMARY KEY 来完成。
我们使用语句 "INT AUTO_INCREMENT PRIMARY KEY"，它将为每条记录插入唯一的编号。从 1 开始，每个记录递增 1。
实例
创建表时创建主键：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

mycursor.execute("CREATE TABLE customers (id INT
AUTO_INCREMENT PRIMARY KEY,
name VARCHAR(255), address VARCHAR(255))")
```

运行实例
如果表已存在，请使用 ALTER TABLE 关键字：
实例
在已有的表上创建主键：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

mycursor.execute("ALTER TABLE customers ADD COLUMN id INT
AUTO_INCREMENT PRIMARY KEY")
```

运行实例

MySQL Create Database
MySQL Insert

插入表
如需填充 MySQL 中的表，请使用 "INSERT INTO" 语句。
实例
在表 "customers" 中插入记录：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = ("John", "Highway 21")
mycursor.execute(sql, val)

mydb.commit()

print(mycursor.rowcount, "record inserted.")
```

运行实例
重要：请注意语句 mydb.commit()。需要进行更改，否则表不会有任何改变。

插入多行
要在表中插入多行，请使用 executemany() 方法。
executemany() 方法的第二个参数是元组列表，包含要插入的数据：
实例
用数据填充 "customers" 表：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()
```

```
sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = [
  ('Peter', 'Lowstreet 4'),
  ('Amy', 'Apple st 652'),
  ('Hannah', 'Mountain 21'),
  ('Michael', 'Valley 345'),
  ('Sandy', 'Ocean blvd 2'),
  ('Betty', 'Green Grass 1'),
  ('Richard', 'Sky st 331'),
  ('Susan', 'One way 98'),
  ('Vicky', 'Yellow Garden 2'),
  ('Ben', 'Park Lane 38'),
  ('William', 'Central st 954'),
  ('Chuck', 'Main Road 989'),
  ('Viola', 'Sideway 1633')
]

mycursor.executemany(sql, val)

mydb.commit()

print(mycursor.rowcount, "was inserted.")
```

运行实例

获取已插入 ID

您可以通过询问 cursor 对象来获取刚插入的行的 id。

注释：如果插入不止一行，则返回最后插入行的 id。

实例

插入一行，并返回 id：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = ("Michelle", "Blue Village")
mycursor.execute(sql, val)
```

```
mydb.commit()

print("1 record inserted, ID:", mycursor.lastrowid)
```

运行实例

```
MySQL Create Table
MySQL Select
```

从表中选取

如需从 MySQL 中的表中进行选择，请使用 "SELECT" 语句：

实例

从表 "customers" 中选取所有记录，并显示结果：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

mycursor.execute("SELECT * FROM customers")

myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```

运行实例

注释：我们用了 fetchall() 方法，该方法从最后执行的语句中获取所有行。

选取列

如需只选择表中的某些列，请使用 "SELECT" 语句，后跟列名：

实例

仅选择名称和地址列：

```
import mysql.connector

mydb = mysql.connector.connect(
```

```
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

mycursor.execute("SELECT name, address FROM customers")

myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```

运行实例

使用 fetchone() 方法

如果您只对一行感兴趣，可以使用 fetchone() 方法。

fetchone() 方法将返回结果的第一行：

实例

仅获取一行：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

mycursor.execute("SELECT * FROM customers")

myresult = mycursor.fetchone()

print(myresult)
```

运行实例

MySQL Insert

MySQL Where

使用筛选器来选取
从表中选择记录时，可以使用 "WHERE" 语句对选择进行筛选：
实例
选择记录为 "Park Lane 38" 的记录，结果：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "SELECT * FROM customers WHERE address ='Park Lane 38'"

mycursor.execute(sql)

myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```

运行实例

通配符
您也可以选择以给定字母或短语开头、包含或结束的记录。
请使用 % 表示通配符：
实例
选择地址中包含单词 "way" 的记录：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "SELECT * FROM customers WHERE address LIKE '%way%'"
```

```
mycursor.execute(sql)

myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```

运行实例

防止 SQL 注入

当用户提供查询值时，您应该转义这些值。

此举是为了防止 SQL 注入，这是一种常见的网络黑客技术，可以破坏或滥用您的数据库。

mysql.connector 模块拥有转义查询值的方法：

实例

使用占位符 %s 方法来转义查询值：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "SELECT * FROM customers WHERE address = %s"
adr = ("Yellow Garden 2", )

mycursor.execute(sql, adr)

myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```

运行实例

MySQL Select

MySQL Order By

结果排序

请使用 ORDER BY 语句按升序或降序对结果进行排序。
ORDER BY 关键字默认按升序对结果进行排序。若要按降序对结果进行排序，请使用 DESC 关键字。
实例
以字符顺序对姓名进行排序，结果：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "SELECT * FROM customers ORDER BY name"

mycursor.execute(sql)

myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```

运行实例

降序排序
请使用 DESC 关键字按降序对结果进行排序。
实例
按反转字母顺序对姓名的结果进行排序：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "SELECT * FROM customers ORDER BY name DESC"

mycursor.execute(sql)
```

```
myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```

运行实例

```
MySQL Where
MySQL Delete
```

删除记录

您可以使用 "DELETE FROM" 语句从已有的表中删除记录：

实例

删除地址为 "Mountain 21" 的任何记录：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "DELETE FROM customers WHERE address = 'Mountain 21'"

mycursor.execute(sql)

mydb.commit()

print(mycursor.rowcount, "record(s) deleted")
```

运行实例

重要：请注意语句 mydb.commit()。需要进行更改，否则表不会有任何改变。

请注意 DELETE 语法中的 WHERE 子句：WHERE 子句指定应删除哪些记录。如果省略 WHERE 子句，则将删除所有记录！

防止 SQL 注入

在 delete 语句中，转义任何查询的值也是一种好习惯。

此举是为了防止 SQL 注入，这是一种常见的网络黑客技术，可以破坏

或滥用您的数据库。
mysql.connector 模块使用占位符 %s 来转义 delete 语句中的值：
实例
使用占位符 %s 方法来转义值：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "DELETE FROM customers WHERE address = %s"
adr = ("Yellow Garden 2", )

mycursor.execute(sql, adr)

mydb.commit()

print(mycursor.rowcount, "record(s) deleted")
```

运行实例

MySQL Order By
MySQL Drop Table

删除表
您可以使用 "DROP TABLE" 语句来删除已有的表：
实例
删除 "customers" 表：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)
```

```
mycursor = mydb.cursor()

sql = "DROP TABLE customers"

mycursor.execute(sql)
```

运行实例

只在表存在时删除

如果要删除的表已被删除，或者由于任何其他原因不存在，那么可以使用 IF EXISTS 关键字以避免出错。

实例

删除表 "customers" （如果存在）：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "DROP TABLE IF EXISTS customers"

mycursor.execute(sql)
```

运行实例

MySQL Delete

MySQL Update

更新表

您可以使用 "UPDATE" 语句来更新表中的现有记录：

实例

把地址列中的 "Valley 345" 覆盖为 "Canyoun 123"：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
```

```
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "UPDATE customers SET address = 'Canyon 123' WHERE address
= 'Valley 345'"

mycursor.execute(sql)

mydb.commit()

print(mycursor.rowcount, "record(s) affected")
```

运行实例

重要：请注意语句 mydb.commit()。需要进行更改，否则不会表不会有任何改变。

请注意 UPDATE 语法中的 WHERE 子句：WHERE 子句指定应更新的记录。如果省略 WHERE 子句，则所有记录都将更新！

防止 SQL 注入

在 update 语句中，转义任何查询的值都是个好习惯。

此举是为了防止 SQL 注入，这是一种常见的网络黑客技术，可以破坏或滥用您的数据库。

mysql.connector 模块使用占位符 %s 来转义 delete 语句中的值：

实例

使用占位符 %s 方法来转义值：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "UPDATE customers SET address = %s WHERE address = %s"
val = ("Valley 345", "Canyon 123")

mycursor.execute(sql, val)

mydb.commit()
```

```
print(mycursor.rowcount, "record(s) affected")
```

运行实例

MySQL Drop Table
MySQL Limit

限定结果

您可以使用 "LIMIT" 语句限制从查询返回的记录数：

实例

选取 "customers" 表中的前五条记录：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

mycursor.execute("SELECT * FROM customers LIMIT 5")

myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```

运行实例

从另一个位置开始

如果希望从第三条记录开始返回五条记录，您可以使用 "OFFSET" 关键字：

实例

从位置 3 开始返回 5 条记录：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
```

```
)

mycursor = mydb.cursor()

mycursor.execute("SELECT * FROM customers LIMIT 5 OFFSET 2")

myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```

运行实例

```
MySQL Update
MySQL Join
```

组合两张或更多表

您可以使用 JOIN 语句，根据它们之间的相关列组合两个或多个表中的行。

假设您有 "users" 表和 "products" 表：

```
users
{ id: 1, name: 'John', fav: 154},
{ id: 2, name: 'Peter', fav: 154},
{ id: 3, name: 'Amy', fav: 155},
{ id: 4, name: 'Hannah', fav:},
{ id: 5, name: 'Michael', fav:}
products
{ id: 154, name: 'Chocolate Heaven' },
{ id: 155, name: 'Tasty Lemons' },
{ id: 156, name: 'Vanilla Dreams' }
```

可以使用 users 的 fav 字段和 products 的 id 字段来组合这两个表。

实例

组合用户和产品，查看用户最喜欢的产品名称：

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)
```

```
mycursor = mydb.cursor()

sql = "SELECT \
  users.name AS user, \
  products.name AS favorite \
  FROM users \
  INNER JOIN products ON users.fav = products.id"

mycursor.execute(sql)

myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```

运行实例

注释：您可以使用 JOIN 而不是 INNER JOIN。您都会得到相同的结果。

LEFT JOIN

在上例中，Hannah 和 Michael 被排除在结果之外，这是因为 INNER JOIN 只显示匹配的记录。

如果希望显示所有用户，即使他们没有喜欢的产品，请使用 LEFT JOIN 语句：

实例

选择所有用户及其喜爱的产品：

```
sql = "SELECT \
  users.name AS user, \
  products.name AS favorite \
  FROM users \
  LEFT JOIN products ON users.fav = products.id"
```

运行实例

RIGHT JOIN

如果您想要返回所有产品以及喜欢它们的用户，即使没有用户喜欢这些产品，请使用 RIGHT JOIN 语句：

实例

选择所有产品以及喜欢它们的用户：

```
sql = "SELECT \
  users.name AS user, \
  products.name AS favorite \
  FROM users \
  RIGHT JOIN products ON users.fav = products.id"
```

运行实例

注释：不对任何产品感兴趣的 Hannah 和 Michael 不包括在结果中。

MySQL Limit
MongoDB 入门

Python 可以在数据库应用程序中使用。
最受欢迎的 NoSQL 数据库之一是 MongoDB。

MongoDB
MongoDB 将数据存储在类似 JSON 的文档中，这使得数据库非常灵活和可伸缩。
为了能够测试本教程中的代码示例，您需要访问 MongoDB 数据库。
您可以在 https://www.mongodb.com 下载免费的 MongoDB 数据库。

PyMongo
Python 需要 MongoDB 驱动程序来访问 MongoDB 数据库。
在本教程中，我们会使用 MongoDB 驱动程序 "PyMongo"。
我们建议您使用 PIP 安装 "PyMongo"。
PIP 很可能已经安装在 Python 环境中。
将命令行导航到 PIP 的位置，然后键入以下内容：
下载并安装 "PyMongo"：
C:\Users\...\AppData\Local\Programs\Python\Python36-32\Scripts>python -m pip install pymongo
现在您已经下载并安装了 mongoDB 驱动程序。

测试 PyMongo
如需测试安装是否成功，或者您是否已安装 "pymongo"，请创建一张包含以下内容的 Python 页面：
demo_mongodb_test.py:
import pymongo
运行实例
如果执行上述代码没有错误，则 "pymongo" 已安装就绪。

MySQL Join
MongoDB 创建数据库

创建数据库
要在 MongoDB 中创建数据库，首先要创建 MongoClient 对象，然后使用正确的 IP 地址和要创建的数据库的名称指定连接 URL。

如果数据库不存在，MongoDB 将创建数据库并建立连接。
实例
创建名为 "mydatabase" 的数据库：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")

mydb = myclient["mydatabase"]
```

运行实例
重要说明：在 MongoDB 中，数据库在获取内容之前不会创建！
在实际创建数据库（和集合）之前，MongoDB 会一直等待您创建至少有一个文档（记录）的集合（表）。

检查数据库是否存在
请记住：在 MongoDB 中，数据库在获取内容之前不会创建，因此如果这是您第一次创建数据库，则应在检查数据库是否存在之前完成接下来的两章（创建集合和创建文档）！
您可以通过列出系统中的所有数据库来检查数据库是否存在：
实例
返回系统中的数据库列表：

```
print(myclient.list_database_names())
```

运行实例
或者您可以按名称检查特定数据库：
实例
检查 "mydatabase" 是否存在：

```
dblist = myclient.list_database_names()
if "mydatabase" in dblist:
  print("The database exists.")
```

运行实例

MongoDB 入门
MongoDB 创建集合

MongoDB 中的集合与 SQL 数据库中的表相同。

创建集合
要在 MongoDB 中创建集合，请使用数据库对象并指定要创建的集合的名称。
如果它不存在，MongoDB 会创建该集合。
实例
创建名为 "customers" 的集合：

```
import pymongo
```

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]

mycol = mydb["customers"]
```

运行实例

重要提示：在 MongoDB 中，集合在获得内容之前不会被创建！

在实际创建集合之前，MongoDB 会等待直到您已插入文档。

检查集合是否存在

请记住：在 MongoDB 中，集合在获取内容之前不会创建，因此如果这是您第一次创建集合，则应在检查集合是否存在之前完成下一章（创建文档）！

您可以通过列出所有集合来检查数据库中是否存在集合：

实例

返回数据库中所有集合的列表：

```
print(mydb.list_collection_names())
```

运行实例

或者您可以按名称检查特定集合：

实例

检查 "customers" 集合是否存在：

```
collist = mydb.list_collection_names()
if "customers" in collist:
  print("The collection exists.")
```

运行实例

MongoDB 创建数据库

MongoDB Insert

MongoDB 中的文档与 SQL 数据库中的记录相同。

插入集合

要在 MongoDB 中把记录或我们所称的文档插入集合，我们使用 insert_one() 方法。

insert_one() 方法的第一个参数是字典，其中包含希望插入文档中的每个字段名称和值。

实例

在 "customers" 集合中插入记录：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
```

```
mycol = mydb["customers"]

mydict = { "name": "Bill", "address": "Highway 37" }

x = mycol.insert_one(mydict)
```

运行实例

返回 _id 字段

insert_one() 方法返回 InsertOneResult 对象，该对象拥有属性 inserted_id，用于保存插入文档的 id。

实例

在 "customers" 集合中插入另一条记录，并返回 _id 字段的值：

```
mydict = { "name": "Peter", "address": "Lowstreet 27" }

x = mycol.insert_one(mydict)

print(x.inserted_id)
```

运行实例

如果您没有指定 _id 字段，那么 MongoDB 将为您添加一个，并为每个文档分配一个唯一的 ID。

在上例中，没有指定 _id 字段，因此 MongoDB 为记录（文档）分配了唯一的 _id。

插入多个文档

要将多个文档插入 MongoDB 中的集合，我们使用 insert_many() 方法。

insert_many() 方法的第一个参数是包含字典的列表，其中包含要插入的数据：

实例

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mylist = [
  { "name": "Amy", "address": "Apple st 652"},
  { "name": "Hannah", "address": "Mountain 21"},
  { "name": "Michael", "address": "Valley 345"},
  { "name": "Sandy", "address": "Ocean blvd 2"},
  { "name": "Betty", "address": "Green Grass 1"},
  { "name": "Richard", "address": "Sky st 331"},
  { "name": "Susan", "address": "One way 98"},
  { "name": "Vicky", "address": "Yellow Garden 2"},
```

```
  { "name": "Ben", "address": "Park Lane 38"},
  { "name": "William", "address": "Central st 954"},
  { "name": "Chuck", "address": "Main Road 989"},
  { "name": "Viola", "address": "Sideway 1633"}
]

x = mycol.insert_many(mylist)

# 打印被插入文档的  _id 值列表：
print(x.inserted_ids)
```

运行实例

insert_many() 方法返回 InsertManyResult 对象，该对象拥有属性 inserted_ids，用于保存被插入文档的 id。

插入带有指定 ID 的多个文档

如果您不希望 MongoDB 为您的文档分配唯一 id，则可以在插入文档时指定 _id 字段。

请记住，值必须是唯一的。两个文件不能有相同的 _id。

实例

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mylist = [
  { "_id": 1, "name": "John", "address": "Highway 37"},
  { "_id": 2, "name": "Peter", "address": "Lowstreet 27"},
  { "_id": 3, "name": "Amy", "address": "Apple st 652"},
  { "_id": 4, "name": "Hannah", "address": "Mountain 21"},
  { "_id": 5, "name": "Michael", "address": "Valley 345"},
  { "_id": 6, "name": "Sandy", "address": "Ocean blvd 2"},
  { "_id": 7, "name": "Betty", "address": "Green Grass 1"},
  { "_id": 8, "name": "Richard", "address": "Sky st 331"},
  { "_id": 9, "name": "Susan", "address": "One way 98"},
  { "_id": 10, "name": "Vicky", "address": "Yellow Garden 2"},
  { "_id": 11, "name": "Ben", "address": "Park Lane 38"},
  { "_id": 12, "name": "William", "address": "Central st 954"},
  { "_id": 13, "name": "Chuck", "address": "Main Road 989"},
  { "_id": 14, "name": "Viola", "address": "Sideway 1633"}
]

x = mycol.insert_many(mylist)
```

```
# 打印被插入文档的  _id 值列表：
print(x.inserted_ids)
```

运行实例

MongoDB 创建集合
MongoDB Find

在 MongoDB 中，我们使用 find 和 findOne 方法来查找集合中的数据。
就像 SELECT 语句用于查找 MySQL 数据库中的表中的数据一样。

查找一项
如需在 MongoDB 中的集合中选取数据，我们可以使用 find_one() 方法。
find_one() 方法返回选择中的第一个匹配项。
实例
查找 customers 集合中的首个文档：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

x = mycol.find_one()

print(x)
```

运行实例

查找全部
如需从 MongoDB 中的表中选取数据，我们还可以使用 find() 方法。
find() 方法返回选择中的所有匹配项。
find() 方法的第一个参数是 query 对象。在这个例子中，我们用了一个空的 query 对象，它会选取集合中的所有文档。
find() 方法没有参数提供与 MySQL 中的 SELECT * 相同的结果。
实例
返回 "customers" 集合中的所有文档，并打印每个文档：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
```

```
for x in mycol.find():
  print(x)
```

运行实例

只返回某些字段

find() 方法的第二个参数是描述包含在结果中字段的对象。

此参数是可选的，如果省略，则所有字段都将包含在结果中。

实例

只返回姓名和地址，而不是 _ids：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

for x in mycol.find({},{ "_id": 0, "name": 1, "address": 1 }):
  print(x)
```

运行实例

不允许在同一对象中同时指定 0 和 1 值（除非其中一个字段是 _id 字段）。如果指定值为 0 的字段，则所有其他字段的值为 1，反之亦然：

实例

这个例子从结果中排出 "address"：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

for x in mycol.find({},{ "address": 0 }):
  print(x)
```

运行实例

实例

如果在同一对象中同时指定 0 和 1 值，则会出现错误（除非其中一个字段是 _id 字段）：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

for x in mycol.find({},{ "name": 1, "address": 0 }):
  print(x)
```

MongoDB Insert
MongoDB Query

筛选结果
在集合中查找文档时，您能够使用 query 对象过滤结果。
find() 方法的第一个参数是 query 对象，用于限定搜索。
实例
查找地址为 "Park Lane 38" 的文档：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": "Park Lane 38" }

mydoc = mycol.find(myquery)

for x in mydoc:
  print(x)
```

运行实例

高级查询
如需进行高级查询，可以使用修饰符作为查询对象中的值。
例如，要查找 "address" 字段以字母 "S" 或更高（按字母顺序）开头的文档，请使用大于修饰符::
实例
查找地址以字母 "S" 或更高开头的文档：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": { "$gt": "S" } }

mydoc = mycol.find(myquery)

for x in mydoc:
  print(x)
```

运行实例

使用正则表达式来筛选
您也可以将正则表达式用作修饰符。
正则表达式只能用于查询字符串。
如果只查找 "address" 字段以字母 "S" 开头的文档，请使用正则表达式 {"$regex": "^S"}：
实例
查找地址以字母 "S" 开头的文档：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": { "$regex": "^S" } }

mydoc = mycol.find(myquery)

for x in mydoc:
  print(x)
```

运行实例

```
MongoDB Find
MongoDB Sort
```

结果排序
请使用 sort() 方法按升序或降序对结果进行排序。
sort() 方法为 "fieldname"（字段名称）提供一个参数，为 "direction"（方向）提供一个参数（升序是默认方向）。
实例
按姓名的字母顺序对结果进行排序：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mydoc = mycol.find().sort("name")

for x in mydoc:
  print(x)
```

运行实例

降序排序
使用值 -1 作为第二个参数进行降序排序。

```
sort("name", 1) # 升序
sort("name", -1) # 降序
```

实例
按名称的逆向字母顺序对结果进行排序：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mydoc = mycol.find().sort("name", -1)

for x in mydoc:
  print(x)
```

运行实例

MongoDB Query
MongoDB Delete

删除文档
要删除一个文档，我们使用 delete_one() 方法。
delete_one() 方法的第一个参数是 query 对象，用于定义要删除的文档。
注释：如果查询找到了多个文档，则仅删除第一个匹配项。
实例
删除地址为 "Mountain 21" 的文档：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": "Mountain 21" }

mycol.delete_one(myquery)
```

运行实例

删除多个文档
要删除多个文档，请使用 delete_many() 方法。
delete_many() 方法的第一个参数是一个查询对象，用于定义要删除的文档。
实例
删除地址以字母 S 开头的所有文档：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": {"$regex": "^S"} }

x = mycol.delete_many(myquery)

print(x.deleted_count, " documents deleted.")
```

运行实例

删除集合中的所有文档
要删除集合中的所有文档，请把空的查询对象传递给 delete_many() 方法：
实例
删除 "customers" 集合中的所有文档：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

x = mycol.delete_many({})

print(x.deleted_count, " documents deleted.")
```

运行实例

MongoDB Sort
MongoDB 删除集合

删除集合
您可以使用 drop() 方法删除在 MongoDB 中调用的表或集合。
实例

删除 "customers" 集合：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mycol.drop()
```

运行实例

如果成功删除集合，则 drop() 方法返回 true，如果集合不存在则返回 false。

MongoDB Delete

MongoDB Update

更新集合

您可以使用 update_one() 方法来更新 MongoDB 中调用的记录或文档。

update_one() 方法的第一个参数是 query 对象，用于定义要更新的文档。

注释：如果查询找到多个记录，则仅更新第一个匹配项。

第二个参数是定义文档新值的对象。

实例

把地址 "Valley 345" 改为 "Canyon 123"：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": "Valley 345" }
newvalues = { "$set": { "address": "Canyon 123" } }

mycol.update_one(myquery, newvalues)

#print "customers" after the update:
for x in mycol.find():
  print(x)
```

运行实例

更新多个

如需更新符合查询条件的所有文档，请使用 update_many() 方法。

实例
更新地址以字母 "S" 开头的所有文档：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": { "$regex": "^S" } }
newvalues = { "$set": { "name": "Minnie" } }

x = mycol.update_many(myquery, newvalues)

print(x.modified_count, "documents updated.")
```

运行实例

MongoDB 删除集合
MongoDB Limit

限定结果
要限制 MongoDB 中的结果，我们使用 limit() 方法。
limit() 方法接受一个参数，定义的数字表示返回的文档数。
假设你有一个 "customers" 集合：

```
Customers
{'_id': 1, 'name': 'John', 'address': 'Highway37'}
{'_id': 2, 'name': 'Peter', 'address': 'Lowstreet 27'}
{'_id': 3, 'name': 'Amy', 'address': 'Apple st 652'}
{'_id': 4, 'name': 'Hannah', 'address': 'Mountain 21'}
{'_id': 5, 'name': 'Michael', 'address': 'Valley 345'}
{'_id': 6, 'name': 'Sandy', 'address': 'Ocean blvd 2'}
{'_id': 7, 'name': 'Betty', 'address': 'Green Grass 1'}
{'_id': 8, 'name': 'Richard', 'address': 'Sky st 331'}
{'_id': 9, 'name': 'Susan', 'address': 'One way 98'}
{'_id': 10, 'name': 'Vicky', 'address': 'Yellow Garden 2'}
{'_id': 11, 'name': 'Ben', 'address': 'Park Lane 38'}
{'_id': 12, 'name': 'William', 'address': 'Central st 954'}
{'_id': 13, 'name': 'Chuck', 'address': 'Main Road 989'}
{'_id': 14, 'name': 'Viola', 'address': 'Sideway 1633'}
```

实例
把结果限定为只返回 5 个文档：

```
import pymongo
```

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myresult = mycol.find().limit(5)

# 打印结果：
for x in myresult:
  print(x)
```

运行实例

MongoDB Update
Python 参考概览

本节包含 Python 参考文档。

Python 参考手册
内建函数
字符串方法
列表方法
字典方法
元组方法
集合方法
文件方法
关键字

MongoDB Limit
Python 内建函数

Python 有一组内建函数。

函数 描述
abs() 返回数的绝对值
all() 如果可迭代对象中的所有项均为 true，则返回 True。
any() 如果可迭代对象中的任何项为 true，则返回 True。
ascii() 返回对象的可读版本。用转义字符替换 none-ascii 字符。
bin() 返回数的二进制版本。

bool() 返回指定对象的布尔值。
bytearray() 返回字节数组。
bytes() 返回字节对象。
callable() 如果指定的对象是可调用的，则返回 True，否则返回 False。
chr() 返回指定 Unicode 代码中的字符。
classmethod() 把方法转换为类方法。
compile() 把指定的源作为对象返回，准备执行。
complex() 返回复数。
delattr() 从指定的对象中删除指定的属性（属性或方法）。
dict() 返回字典（数组）。
dir() 返回指定对象的属性和方法的列表。
divmod() 当参数 1 除以参数 2 时，返回商和余数。
enumerate() 获取集合（例如元组）并将其作为枚举对象返回。
eval() 评估并执行表达式。
exec() 执行指定的代码（或对象）。
filter() 使用过滤器函数排除可迭代对象中的项目。
float() 返回浮点数。
format() 格式化指定值。
frozenset() 返回 frozenset 对象。
getattr() 返回指定属性的值（属性或方法）。
globals() 以字典返回当前全局符号表。
hasattr() 如果指定的对象拥有指定的属性（属性/方法），则返回 True。
hash() 返回指定对象的哈希值。
help() 执行内建的帮助系统。
hex() 把数字转换为十六进制值。
id() 返回对象的 id。
input() 允许用户输入。
int() 返回整数。
isinstance() 如果指定的对象是指定对象的实例，则返回 True。
issubclass() 如果指定的类是指定对象的子类，则返回 True。
iter() 返回迭代器对象。
len() 返回对象的长度。
list() 返回列表。
locals() 返回当前本地符号表的更新字典。
map() 返回指定的迭代器，其中指定的函数应用于每个项目。
max() 返回可迭代对象中的最大项目。
memoryview() 返回内存视图（memory view）对象。
min() 返回可迭代对象中的最小项目。
next() 返回可迭代对象中的下一项。
object() 返回新对象。
oct() 把数转换为八进制。
open() 打开文件并返回文件对象。
ord() 转换表示指定字符的 Unicode 的整数。

pow() 返回 x 的 y 次幂的值。
print() 打印标准输出设备。
property() 获取、设置、删除属性。
range() 返回数字序列，从 0 开始且以 1 为增量（默认地）。
repr() 返回对象的可读版本。
reversed() 返回反转的迭代器。
round() 对数进行舍入。
set() 返回新的集合对象。
setattr() 设置对象的属性（属性/方法）。
slice() 返回 slice 对象。
sorted() 返回排序列表。
@staticmethod() 把方法转换为静态方法。
str() 返回字符串对象。
sum() 对迭代器的项目进行求和。
super() 返回表示父类的对象。
tuple() 返回元组。
type() 返回对象的类型。
vars() 返回对象的 __dict__ 属性。
zip() 从两个或多个迭代器返回一个迭代器。

Python 参考概览
Python 字符串方法

Python 有一组可以在字符串上使用的内建方法。
注释：所有字符串方法都返回新值。它们不会更改原始字符串。

方法 描述
capitalize() 把首字符转换为大写。
casefold() 把字符串转换为小写。
center() 返回居中的字符串。
count() 返回指定值在字符串中出现的次数。
encode() 返回字符串的编码版本。
endswith() 如果字符串以指定值结尾，则返回 true。
expandtabs() 设置字符串的 tab 尺寸。
find() 在字符串中搜索指定的值并返回它被找到的位置。
format() 格式化字符串中的指定值。
format_map() 格式化字符串中的指定值。
index() 在字符串中搜索指定的值并返回它被找到的位置。
isalnum() 如果字符串中的所有字符都是字母数字，则返回 True。
isalpha() 如果字符串中的所有字符都在字母表中，则返回 True。
isdecimal() 如果字符串中的所有字符都是小数，则返回 True。

isdigit() 如果字符串中的所有字符都是数字，则返回 True。
isidentifier() 如果字符串是标识符，则返回 True。
islower() 如果字符串中的所有字符都是小写，则返回 True。
isnumeric() 如果字符串中的所有字符都是数，则返回 True。
isprintable() 如果字符串中的所有字符都是可打印的，则返回 True。
isspace() 如果字符串中的所有字符都是空白字符，则返回 True。
istitle() 如果字符串遵循标题规则，则返回 True。
isupper() 如果字符串中的所有字符都是大写，则返回 True。
join() 把可迭代对象的元素连接到字符串的末尾。
ljust() 返回字符串的左对齐版本。
lower() 把字符串转换为小写。
lstrip() 返回字符串的左修剪版本。
maketrans() 返回在转换中使用的转换表。
partition() 返回元组，其中的字符串被分为三部分。
replace() 返回字符串，其中指定的值被替换为指定的值。
rfind() 在字符串中搜索指定的值，并返回它被找到的最后位置。
rindex() 在字符串中搜索指定的值，并返回它被找到的最后位置。
rjust() 返回字符串的右对齐版本。
rpartition() 返回元组，其中字符串分为三部分。
rsplit() 在指定的分隔符处拆分字符串，并返回列表。
rstrip() 返回字符串的右边修剪版本。
split() 在指定的分隔符处拆分字符串，并返回列表。
splitlines() 在换行符处拆分字符串并返回列表。
startswith() 如果以指定值开头的字符串，则返回 true。
strip() 返回字符串的剪裁版本。
swapcase() 切换大小写，小写成为大写，反之亦然。
title() 把每个单词的首字符转换为大写。
translate() 返回被转换的字符串。
upper() 把字符串转换为大写。
zfill() 在字符串的开头填充指定数量的 0 值。
注释：所有字符串方法都返回新值。它们不会更改原始字符串。
请在 Python 字符串教程 中学习更多有关字符串的知识。

Python 内建函数
Python 列表方法

Python 有一组可以在列表/数组上使用的内置方法。

方法 描述
append() 在列表的末尾添加一个元素
clear() 删除列表中的所有元素

copy() 返回列表的副本
count() 返回具有指定值的元素数量。
extend() 将列表元素（或任何可迭代的元素）添加到当前列表的末尾
index() 返回具有指定值的第一个元素的索引
insert() 在指定位置添加元素
pop() 删除指定位置的元素
remove() 删除具有指定值的项目
reverse() 颠倒列表的顺序
sort() 对列表进行排序
注释：Python 没有内置对数组的支持，但可以使用 Python 列表。
在 Python 列表教程 中学习有关列表的更多知识。
在 Python 数组教程 中学习有关数组的更多知识。

Python 字符串方法
Python 字典方法

Python 有一组可以在字典上使用的内建方法。

方法 描述
clear() 删除字典中的所有元素
copy() 返回字典的副本
fromkeys() 返回拥有指定键和值的字典
get() 返回指定键的值
items() 返回包含每个键值对的元组的列表
keys() 返回包含字典键的列表
pop() 删除拥有指定键的元素
popitem() 删除最后插入的键值对
setdefault() 返回指定键的值。如果该键不存在，则插入具有指定值的键。
update() 使用指定的键值对字典进行更新
values() 返回字典中所有值的列表
在我们的 Python 字典教程 中学习更多有关字典的知识。

Python 列表方法
Python 元组方法

Python 有两个可以在元组上使用的内建方法。

方法 描述
count() 返回元组中指定值出现的次数。
index() 在元组中搜索指定的值并返回它被找到的位置。
在我们的 Python 元组教程 中学习更多有关元组的知识。

Python 字典方法
Python 集合方法

Python 有一组可以在集合上使用的内建方法。

方法 描述
add() 向集合添加元素。
clear() 删除集合中的所有元素。
copy() 返回集合的副本。
difference() 返回包含两个或更多集合之间差异的集合。
difference_update() 删除此集合中也包含在另一个指定集合中的项目。
discard() 删除指定项目。
intersection() 返回为两个其他集合的交集的集合。
intersection_update() 删除此集合中不存在于其他指定集合中的项目。
isdisjoint() 返回两个集合是否有交集。
issubset() 返回另一个集合是否包含此集合。
issuperset() 返回此集合是否包含另一个集合。
pop() 从集合中删除一个元素。
remove() 删除指定元素。
symmetric_difference() 返回具有两组集合的对称差集的集合。
symmetric_difference_update() 插入此集合和另一个集合的对称差集。
union() 返回包含集合并集的集合。
update() 用此集合和其他集合的并集来更新集合。
在我们的 Python 集合教程 中学习更多有关集合的知识。

Python 元组方法
Python 文件方法

Python 有一组可用于文件对象的方法。

方法 描述
close() 关闭文件。
detach() 从缓冲区返回分离的原始流（raw stream）。
fileno() 从操作系统的角度返回表示流的数字。
flush() 刷新内部缓冲区。
isatty() 返回文件流是否是交互式的。
read() 返回文件内容。
readable() 返回是否能够读取文件流。
readline() 返回文件中的一行。
readlines() 返回文件中的行列表。
seek() 更改文件位置。
seekable() 返回文件是否允许我们更改文件位置。
tell() 返回当前的文件位置。
truncate() 把文件调整为指定的大小。
writeable() 返回是否能够写入文件。
write() 把指定的字符串写入文件。
writelines() 把字符串列表写入文件。
请在我们的 Python 文件处理教程 学习更多有关文件对象的知识。

Python 集合方法
Python 关键字

Python 有一组关键字，这些关键字是保留字，不能用作变量名、函数名或任何其他标识符：

关键字 描述
and 逻辑运算符。
as 创建别名。
assert 用于调试。
break 跳出循环。
class 定义类。
continue 继续循环的下一个迭代。
def 定义函数。
del 删除对象。
elif 在条件语句中使用，等同于 else if。
else 用于条件语句。
except 处理异常，发生异常时如何执行。
False 布尔值，比较运算的结果。
finally 处理异常，无论是否存在异常，都将执行一段代码。
for 创建 for 循环。
from 导入模块的特定部分。

global 声明全局变量。
if 写一个条件语句。
import 导入模块。
in 检查列表、元组等集合中是否存在某个值。
is 测试两个变量是否相等。
lambda 创建匿名函数。
None 表示 null 值。
nonlocal 声明非局部变量。
not 逻辑运算符。
or 逻辑运算符。
pass null 语句，一条什么都不做的语句。
raise 产生异常。
return 退出函数并返回值。
True 布尔值，比较运算的结果。
try 编写 try...except 语句。
while 创建 while 循环。
with 用于简化异常处理。
yield 结束函数，返回生成器。

Python 文件方法
随机模块

Python 有一个可用于制作随机数的内建模块。
random 模块有一组如下的方法：

方法 描述
seed() 初始化随机数生成器。
getstate() 返回随机数生成器的当前内部状态。
setstate() 恢复随机数生成器的内部状态。
getrandbits() 返回表示随机位的数字。
randrange() 返回给定范围之间的随机数。
randint() 返回给定范围之间的随机数。
choice() 返回给定序列中的随机元素。
choices() 返回一个列表，其中包含给定序列中的随机选择。
shuffle() 接受一个序列，并以随机顺序返回此序列。
sample() 返回序列的给定样本。
random() 返回 0 与 1 之间的浮点数。
uniform() 返回两个给定参数之间的随机浮点数。
triangular() 返回两个给定参数之间的随机浮点数，您还可以设置模式参数以指定其他两个参数之间的中点。
betavariate() 基于 Beta 分布（用于统计学）返回 0 到 1 之间的随

机浮点数。
expovariate() 基于指数分布（用于统计学），返回 0 到 1 之间的随机浮点数，如果参数为负，则返回 0 到 -1 之间的随机浮点数。
gammavariate() 基于 Gamma 分布（用于统计学）返回 0 到 1 之间的随机浮点数。
gauss() 基于高斯分布（用于概率论）返回 0 到 1 之间的随机浮点数。
lognormvariate() 基于对数正态分布（用于概率论）返回 0 到 1 之间的随机浮点数。
normalvariate() 基于正态分布（用于概率论）返回 0 到 1 之间的随机浮点数。
vonmisesvariate() 基于 von Mises 分布（用于定向统计学）返回 0 到 1 之间的随机浮点数。
paretovariate() 基于 Pareto 分布（用于概率论）返回 0 到 1 之间的随机浮点数。
weibullvariate() 基于 Weibull 分布（用于统计学）返回 0 到 1 之间的随机浮点数。

Python 关键字
请求模块

实例
向网页发出请求，并打印响应文本：

```
import requests

x = requests.get('https://w3school.com.cn/python/demopage.htm')

print(x.text)
```

运行实例

定义和用法
requests 模块允许您使用 Python 发送 HTTP 请求。
HTTP 请求返回响应对象，其中包含所有响应数据（内容、编码、状态等）。

下载并安装请求模块
将命令行导航到 PIP 的位置，然后键入以下内容：

```
C:\Users\Your     Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip install requests
```

语法

```
requests.methodname(params)
```

方法
方法 描述
delete(url, args) 向指定的 URL 发送 DELETE 请求。
get(url, params, args) 向指定的 URL 发送 GET 请求。
head(url, args) 向指定的 URL 发送 HEAD 请求。
patch(url, data, args) 向指定的 URL 发送 PATCH 请求。
post(url, data, json, args) 向指定的 URL 发送 POST 请求。
put(url, data, args) 向指定的 URL 发送 PUT 请求。
request(method, url, args) 将指定方法的请求发送到指定的 URL。

随机模块
删除列表重复项

学习如何从 Python 中的 List 中删除重复项。

实例
从列表中删除任何重复项：
```
mylist = ["a", "b", "a", "c", "c"]
mylist = list(dict.fromkeys(mylist))
print(mylist)
```
运行实例

实例解释
首先，我们有一个包含重复项的 List：
包含重复项的列表
```
mylist = ["a", "b", "a", "c", "c"]
mylist = list(dict.fromkeys(mylist))
print(mylist)
```
使用列表项作为键创建字典。这将自动删除任何重复项，因为词典不能有重复的键。
创建字典
```
mylist = ["a", "b", "a", "c", "c"]
mylist = list( dict.fromkeys(mylist) )
print(mylist)
```
然后，将字典转换回列表：
转换为 List
```
mylist = ["a", "b", "a", "c", "c"]
mylist = list( dict.fromkeys(mylist) )
print(mylist)
```
现在我们有一个没有任何重复的 List，它与原始 List 拥有相同的顺

序。
打印列表以演示结果：
打印 List
```
mylist = ["a", "b", "a", "c", "c"]
mylist = list(dict.fromkeys(mylist))
print(mylist)
```

创建函数
如果您希望有一个函数可以发送列表，然后它们返回的无重复项，则可以创建函数并插入上例中的代码。
实例
```
def my_function(x):
  return list(dict.fromkeys(x))

mylist = my_function(["a", "b", "a", "c", "c"])

print(mylist)
```
运行实例

例子解释
创建一个以 List 作为参数的函数。
创建函数
```
def my_function(x):
  return list(dict.fromkeys(x))

mylist = my_function(["a", "b", "a", "c", "c"])

print(mylist)
```
使用此 List 项作为键创建字典。
创建字典
```
def my_function(x):
  return list( dict.fromkeys(x) )

mylist = my_function(["a", "b", "a", "c", "c"])

print(mylist)
```
将字典转换为列表：
转换为列表
```
def my_function(x):
  return list( dict.fromkeys(x) )

mylist = my_function(["a", "b", "a", "c", "c"])

print(mylist)
```

返回列表：
返回列表
```
def my_function(x):
  return list(dict.fromkeys(x))

mylist = my_function(["a", "b", "a", "c", "c"])

print(mylist)
```
使用列表作为参数来调用该函数：
调用函数
```
def my_function(x):
  return list(dict.fromkeys(x))

mylist = my_function(["a", "b", "a", "c", "c"])

print(mylist)
```
打印结果：
打印结果
```
def my_function(x):
  return list(dict.fromkeys(x))

mylist = my_function(["a", "b", "a", "c", "c"])

print(mylist)
```

请求模块
反转字符串

学习如何在 Python 中反转字符串。

在 Python 中没有内置函数来反转字符串。
最快（也是最简单？）的方法是使用向后退步的切片，-1。
实例
反转字符串 "Hello World"：
```
txt = "Hello World"[::-1]
print(txt)
```
运行实例

例子解释
我们有个字符串，"Hello World"，我们要反转它：
要反转的字符串

```
txt = "Hello World" [::-1]
print(txt)
```

创建一个从字符串末尾开始的切片，然后向后移动。
在这个特定的例子中，slice 语句 [::-1] 等同于 [11:0:-1]，这意味着从位置 11 开始（因为 "Hello "World" 有 11 个字符），结束于位置 0，移动步长 -1，负一意味着向后退一步。
裁切字符串

```
txt = "Hello World" [::-1]
print(txt)
```

现在我们有一个向后读取 "Hello World" 字符串 txt。
打印字符串以演示结果
打印列表

```
txt = "Hello World"[::-1]
print(txt)
```

创建函数
如果你想要一个可以发送字符串并向后返回它们的函数，那么可以创建一个函数并插入上例中的代码
实例

```
def my_function(x):
  return x[::-1]

mytxt = my_function("I wonder how this text looks like backwards")

print(mytxt)
```

运行实例

例子解释
创建以字符串作为参数的函数。
创建函数

```
def my_function(x):
  return x[::-1]

mytxt = my_function("I wonder how this text looks like backwards")

print(mytxt)
```

从字符串末尾开始裁切字符串并向后移动。
裁切字符串

```
def my_function(x):
  return x [::-1]

mytxt = my_function("I wonder how this text looks like
```

```
backwards")

print(mytxt)
返回向后的字符串。
返回字符串
def my_function(x):
  return x[::-1]

mytxt = my_function("I wonder how this text looks like
backwards")

print(mytxt )
使用字符串作为参数来调用函数：
调用函数
def my_function(x):
  return x[::-1]

mytxt = my_function("I wonder how this text looks like
backwards")

print(mytxt)
打印结果：
打印结果
def my_function(x):
  return x[::-1]

mytxt = my_function("I wonder how this text looks like
backwards")

print(mytxt)

删除列表重复项
Python 实例
```

Python 语法
打印 "Hello World"
Python 中的注释
文档字符串（Docstrings）
例子解释：语法

Python 变量

创建变量
同时输出文本和变量
把变量添加到另一个变量
例子解释：变量

Python 数字
验证对象的类型
创建整数
创建浮点数
创建带有“e”的科学数字以表示 10 的幂
创建复数
例子解释：数字

Python Casting
Casting - 整数
Casting - 浮点
Casting - 字符串
例子解释：Casting

Python 字符串
获取字符串位置 1 处的字符
子串。获取从位置 2 到位置 5 （不包括）的字符。
删除字符串的开头或结尾的空格
返回字符串的长度
把字符串转换为小写
把字符串转换为大写
把字符串替换为另一个字符串
把字符串拆分为子串
例子解释：字符串

Python 运算符
加运算符
减运算符
乘运算符
除运算符
取模运算符
赋值运算符
例子解释：运算符

Python 列表
创建列表
访问列表项
更改列表项目的值
遍历列表

检查列表项是否存在
获取列表的长度
把一个项目添加到列表末端
把一个项目添加到指定索引
删除项目
删除最后一项
删除指定索引的项目
清空列表
使用 list() 构造函数来生成列表
例子解释：列表

Python 元组
创建元组
访问元组项目
更改元组值
遍历元组
检查某个元组项目是否存在
获取元组的长度
删除元组
使用 tuple() 构造函数创建元组
例子解释：元组

Python 集合
创建集合
遍历集合
检查项目是否存在
向集合添加一个项目
向集合添加多个项目
获取集合的长度
删除集合中的一个项目
使用 discard() 方法删除集合中的一个项目
使用 pop() 方法删除集合中的最后一项
清空集合
删除集合
使用 set() 构造函数创建集合
例子解释：集合

Python 字典
创建字典
访问字典中的项目
更改字典中某个具体项目的值
逐一打印字典中的所有键名
逐一打印字典中的所有值
使用 values() 函数返回字典的值

使用 items() 遍历键和值
检查某个键是否存在
获取字典的长度
向字典添加一个项目
从字典删除一个项目
清空字典
使用 dict() 构造函数创建字典
例子解释：字典

Python If ... Else
if 语句
elif 语句
else 语句
简写 if
简写 if ... else
and 关键字
or 关键字
例子解释：If ... Else

Python While 循环
while 循环
在 while 循环中使用 break 语句
在 while 循环中使用 continue 语句
例子解释：While 循环

Python For 循环
for 循环
遍历字符串
在 for 循环中使用 break 语句
在 for 循环中使用 continue 语句
在 for 循环中使用 range() 函数
for 循环中的 Else
嵌套 for 循环
例子解释：For 循环

Python 函数
创建并调用函数
函数参数
默认参数值
使函数返回值
递归
例子解释：函数

Python Lambda

将作为参数传递的数字加 10 的 lambda 函数
将参数 a 与参数 b 相乘的 lambda 函数
将参数 a、b 和 c 相加 lambda 函数
例子解释：Lambda

Python 数组
创建数组
访问数组元素
更改数组元素的值
获取数组的长度
遍历数组中的所有元素
向数组添加元素
从数组删除元素
例子解释：数组

Python 类和对象
创建类
创建对象
__init__() 函数
创建对象方法
self 参数
修改对象属性
删除对象属性
删除对象
例子解释：类/对象

Python 迭代器
从元组返回迭代器
从字符串返回迭代器
遍历迭代器
创建迭代器
停止迭代器
例子解释：迭代器

Python 模块
使用模块
模块中的变量
对模块重命名
内置模块
使用 dir() 函数
从模块导入
例子解释：模块

Python 日期

导入 datetime 模块并显示当前日期
返回年份和 weekday 的名称
创建 date 对象
strftime() 方法
例子解释：日期

Python JSON
把 JSON 转换为 Python
把 Python 转换为 JSON
把 Python 对象转换为 JSON 字符串
转换包含所有合法数据类型的 Python 对象
使用 indent 参数来定义缩进量
使用 separators 参数来更改默认分隔符
使用 sort_keys 参数指定结果是否应该排序
例子解释：JSON

Python RegEx
检索字符串是否以 "China" 开头并以 "country" 结尾
使用 findall() 函数
使用 search() 函数
使用 split() 函数
使用 sub() 函数
例子解释：RegEx

Python PIP
使用包
例子解释：PIP

Python Try Except
当发生错误时，打印一条消息。
多个异常
使用 else 关键字定义没有出现错误时执行的代码块
使用 finally 块执行代码，不论 try 块是否引发了错误
例子解释：Try Except

Python 文件
读取文件
只读取文件的一部分
读取文件中的一行
遍历文件的所有行，逐行读取整个文件
例子解释：文件处理

Python MySQL
创建与数据库的连接

按降序对结果排序（反向字母顺序）
删除文档
删除多个文档
删除集合中的所有文档
删除集合
更新文档
更新多个/所有文档
限制结果
例子解释：MongoDB

反转字符串
Python 测验

通过 W3School 测验，测试您的 Python 技能。

测验
此测验包含 25 道题，每道题的最长答题时间是 20 分钟。
本测验是非官方的测试，它仅仅提供了一个了解您对 Python 的掌握程度的工具。

记分
每道题的分值是 1 分。在您完成全部的 25 道题之后，系统会为您的测验打分。
测验结束后，系统会提供您做错的题目的正确答案。其中，绿色为正确答案，而红色为错误答案。
现在就开始测验！(即将上线)
祝您好运！
提示：如果您不懂 Python，建议您从头开始学习 Python 教程。

Python 实例
Python 教程