[译] PythonGuru 中文系列教程

Python 是 Guido Van Rossum 创建的通用编程语言。 如果您刚刚开始编程生涯，Python 因其优雅的语法和可读代码而广受赞誉。Python 最适合您。使用 Python，您可以完成 GUI 开发，Web 应用，系统管理任务，财务计算…

目　录

致谢

当前文档 《[译] PythonGuru 中文系列教程》 由 进击的皇虫 使用 书栈网 (BookStack.CN) 进行构建，生成于 2020-07-22。

书栈网仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈网难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到书栈网，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到书栈网获取最新的文档，以跟上知识更新换代的步伐。

内容来源：ApacheCN https://github.com/apachecn/pythonguru-zh

文档地址：http://www.bookstack.cn/books/pythonguru-zh

书栈官网：https://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

PythonGuru 中文系列教程

> 原文：PythonGuru
>
> 协议：CC BY-NC-SA 4.0
>
> 欢迎任何人参与和完善：一个人可以走的很快，但是一群人却可以走的更远。

- 在线阅读
- ApacheCN 学习资源

贡献指南

本项目需要校对，欢迎大家提交 Pull Request。

> 请您勇敢地去翻译和改进翻译。虽然我们追求卓越，但我们并不要求您做到十全十美，因此请不要担心因为翻译上犯错——在大部分情况下，我们的服务器已经记录所有的翻译，因此您不必担心会因为您的失误遭到无法挽回的破坏。（改编自维基百科）

联系方式

负责人

- 飞龙：562826179

其他

- 在我们的 apachecn/pythonguru-zh github 上提 issue.
- 发邮件到 Email: `apachecn@163.com`.
- 在我们的 组织学习交流群 中联系群主/管理员即可.

赞助我们

微信	支付宝
	

初级 Python

python 入门

> 原文: https://thepythonguru.com/getting-started-with-python/

于 2020 年 1 月 7 日更新

什么是 Python？

Python 是 Guido Van Rossum 创建的通用编程语言。 如果您刚刚开始编程生涯，Python 因其优雅的语法和可读代码而广受赞誉。Python 最适合您。 使用 Python，您可以完成 GUI 开发，Web 应用，系统管理任务，财务计算，数据分析，可视化以及列表等所有工作。

Python 是一种解释语言

是的，Python 是解释语言，当您运行 python 程序时，解释器将逐行解析 python 程序，而 C 或 C++ 等已编译的语言则先编译该程序然后开始运行。

现在您可能会问，那有什么区别？

区别在于，与编译语言相比，解释语言要慢一些。 是的，如果使用 C 或 C++ 等编译语言编写代码，则绝对会获得一些性能上的好处。

但是用这种语言编写代码对于初学者来说是艰巨的任务。 同样在这种语言中，您甚至需要编写大多数基本功能，例如计算数组的长度，分割字符串等。对于更高级的任务，有时您需要创建自己的数据结构以将数据封装在程序中。 因此，在 C/C++ 中，在真正开始解决业务问题之前，您需要注意所有次要细节。 这就是 Python 来的地方。 在 Python 中，您不需要定义任何数据结构，也不需要定义小型工具函数，因为 Python 可以帮助您入门。

此外，Python 在 https://pypi.python.org/ 上提供了数百个可用的库，您可以在项目中使用它们而无需重新设计轮子。

Python 是动态类型的

Python 不需要您提前定义变量数据类型。 Python 根据其包含的值的类型自动推断变量的数据类型。

例如：

```
1.  myvar = "Hello Python"
```

上面的代码行将字符串 `"Hello Python"` 分配给变量 `myvar` ，因此 `myvar` 的类型为字符串。

请注意，与 C，C++ 和 Java 之类的语言不同，在 Python 中，您不需要以分号（ `;` ）结尾的语句。

假设稍后在程序中我们为变量 `myvar` 分配了 `1` 的值，即

```
1.  myvar = 1
```

现在 `myvar` 变量的类型为 `int` 。

Python 是强类型的

如果您使用 PHP 或 javascript 编程。 您可能已经注意到，它们都将一种类型的数据自动转换为另一种类型。

For e.g:

在 JavaScript 中

```
1.  1 + "2"
```

将是 `'12'`

在这里，在进行加法（ `+` ）之前， `1` 将转换为字符串并连接到 `"2"` ，这将导致 `'12'` 成为字符串。 但是，在 Python 中，不允许进行此类自动转换，因此

```
1.  1 + "2"
```

会产生一个错误。

试试看：

```
1.  # run this code to see the error
2.  1 + "2"
```

编写更少的代码，做更多的事情

用 Python 编写的程序通常是 Java 代码的 1/3 或 1/5。 这意味着我们可以用 Python 编写更少的代码来实现与 Java 相同的功能。

要在 Python 中读取文件，您只需要两行代码：

```
1. with open("myfile.txt") as f:
2.     print(f.read())
```

试一试：

```
1. # these two lines create a file "myfile.txt" with data "Learning Python"
2. with open("myfile.txt", "w") as f:
3.     f.write("Learning Python")
4.
5. # these two lines read data from myfile.txt
6. with open("myfile.txt") as f:
7.     print(f.read())
```

不要太在意用于读写文件的命令。 我们将在以后的文章中学习所有内容。

谁使用 Python

许多大型组织（例如 Google，NASA，Quora，HortonWorks 等）都使用 Python。

好的，我可以开始用 Python 构建什么？

您想要的几乎任何东西。 例如：

- GUI 应用
- 网络应用
- 从网站抓取数据
- 分析数据
- 系统管理工具
- 游戏开发
- 数据科学

还有很多 …

在下一篇文章中，我们将学习如何安装 Python 。

安装 Python3

> 原文： https://thepythonguru.com/installing-python3/

于 2020 年 1 月 7 日更新

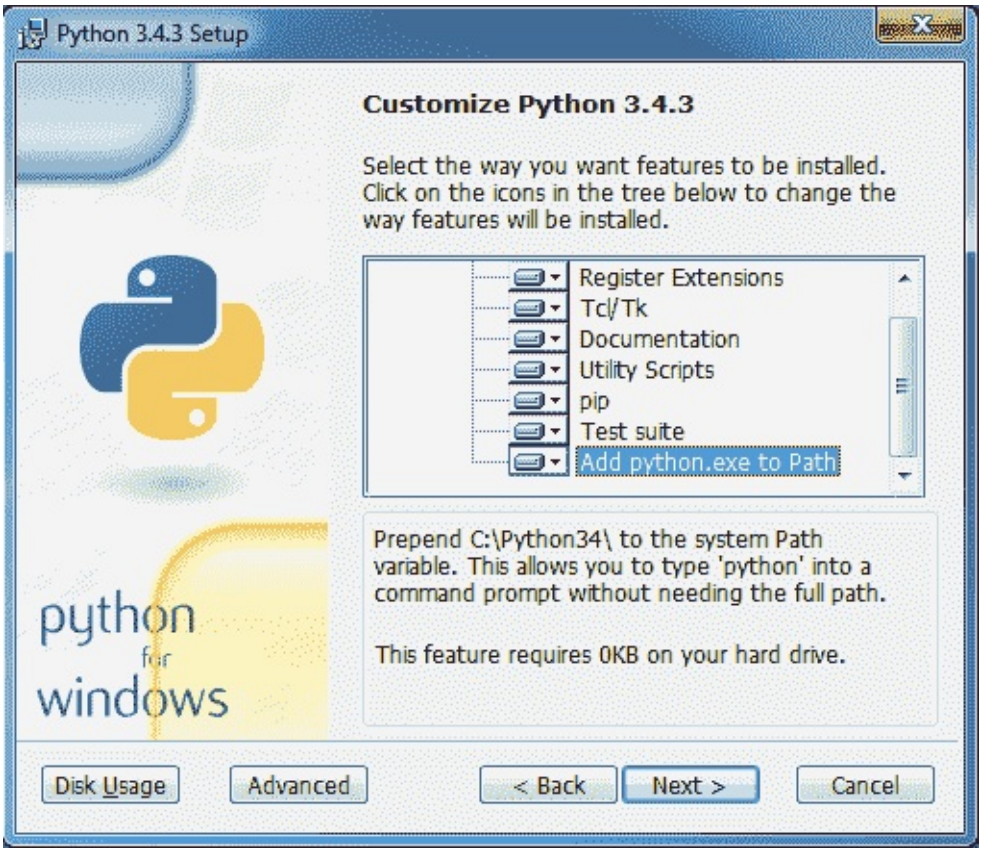
本教程重点介绍 Python3。大多数 Linux 发行版，例如 Ubuntu 14.04，都安装了 python 2 和 3，这是下载链接。 如果您使用其他 Linux 发行版，请参阅此链接以获取安装说明。 Mac 还随附安装了 python 2 和 python 3（如果未查看此链接以获取说明），但 Windows 并非如此。

注意：

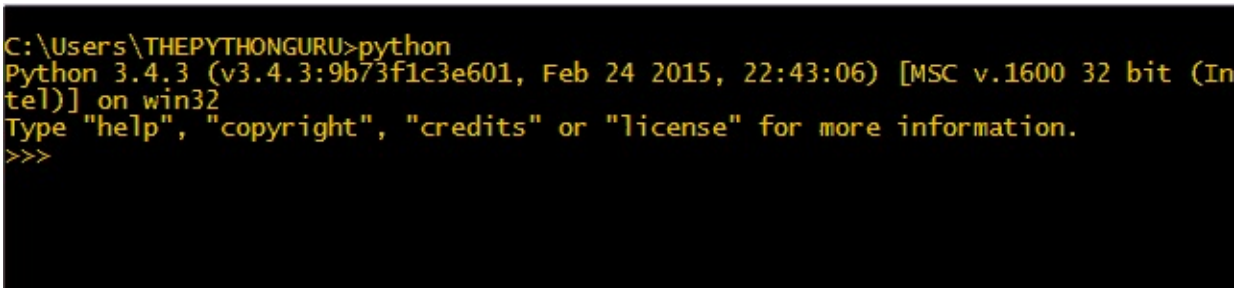
注意：在本教程中，我将仅在 Windows 和 Ubuntu 14.04 上提供必要的说明。

在 Windows 中安装 Python 3

要安装 python，您需要从 https://www.python.org/downloads/ 下载 python 二进制文件，特别是我们将使用 python 3.4.3，您可以在此处从下载 。 安装时，请记住检查“将 `Python.exe` 添加到路径”（请参见下图）。

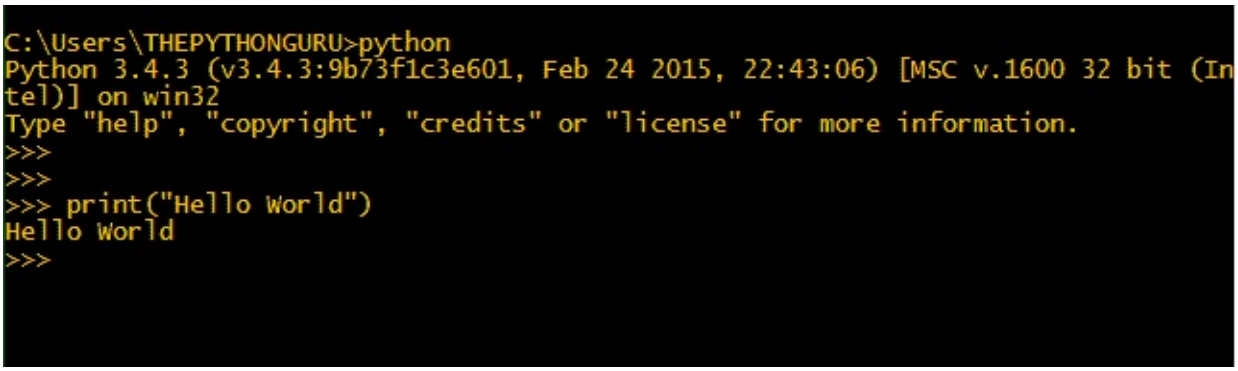


现在您已经安装了 python，打开命令提示符或终端并输入 `python` 。 现在您在 python shell 中。



要测试一切正常，请在 python shell 中键入以下命令。

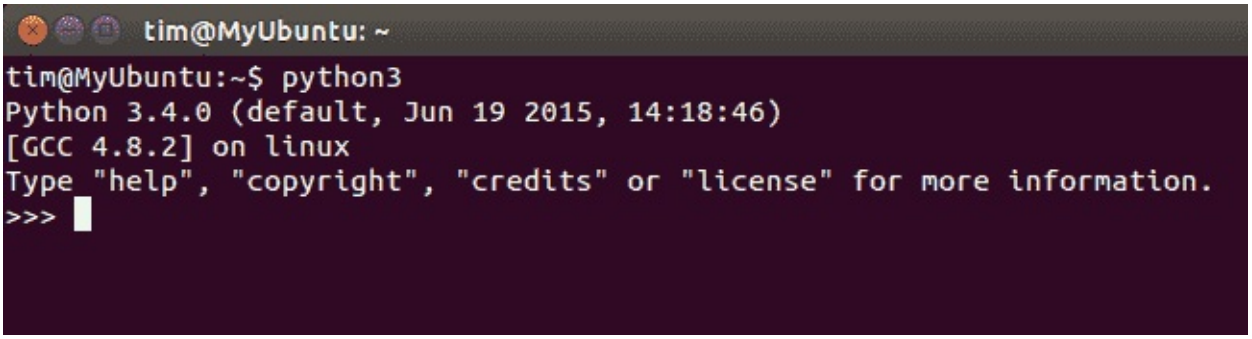
```
1.  print("Hello World")
```



预期输出：

```
1.  Hello World
```

如果您使用的是已随附 python 2 和 python 3 的 Ubuntu 14.04，则需要输入 `python3` 而不是仅 `python` 才能输入 python 3 shell。



安装文本编辑器要编写 python 程序，您将需要一个文本编辑器，您可以使用文本编辑器（如记事本）。 如果要使用完整的文本编辑器，请使用 Notepad++ 或 SublimeText。 下载并安装您选择的文本编辑器。

现在您已经成功安装了 python 3 和文本编辑器，并准备继续进行下一章，在此我们将学习运行 python 程序的不同方法。

运行 python 程序

> 原文: https://thepythonguru.com/running-python-programs/

于 2020 年 1 月 7 日更新

您可以通过两种方式运行 python 程序，首先通过直接在 python shell 中键入命令或运行存储在文件中的程序。 但是大多数时候您想运行存储在文件中的程序。

让我们在记事本目录中创建一个名为 `hello.py` 的文件，即使用记事本（或您选择的任何其他文本编辑器）创建一个 `C:\Users\YourUserName\Documents` ，记住 python 文件具有 `.py` 扩展名，然后在文件中编写以下代码。

```
1.  print("Hello World")
```

在 python 中，我们使用 `print` 函数将字符串显示到控制台。 它可以接受多个参数。 当传递两个或多个参数时， `print()` 函数显示每个参数，并用空格分隔。

```
1.  print("Hello", "World")
```

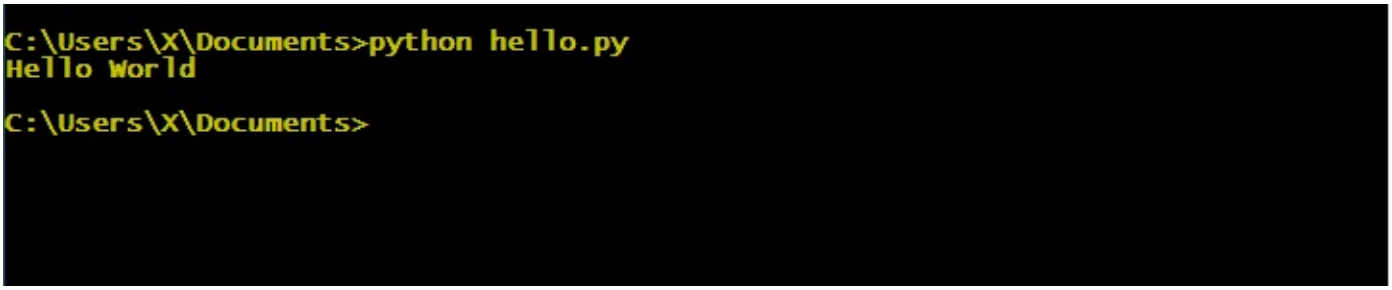
预期输出：

```
1.  Hello World
```

现在打开终端，并使用 `cd` 命令将当前工作目录更改为 `C:\Users\YourUserName\Documents` 。

要运行该程序，请键入以下命令。

```
1.  python hello.py
```



如果一切顺利，您将获得以下输出。

```
Hello World
```

获得帮助

使用 python 迟早会遇到一种情况，您想了解更多有关某些方法或函数的信息。 为了帮助您，Python 具有 `help()` 函数，这是如何使用它。

语法：

要查找有关类别的信息： `help(class_name)`

要查找有关方法的更多信息，属于类别： `help(class_name.method_name)`

假设您想了解更多关于 `int` 类的信息，请转到 Python shell 并键入以下命令。

```
>>> help(int)

Help on class int in module builtins:

class int(object)
 | int(x=0) -> integer
 | int(x, base=10) -> integer
 |
 | Convert a number or string to an integer, or return 0 if no arguments
 | are given. If x is a number, return x.__int__(). For floating point
 | numbers, this truncates towards zero.
 |
 | If x is not a number or if base is given, then x must be a string,
 | bytes, or bytearray instance representing an integer literal in the
 | given base. The literal can be preceded by '+' or '-' and be surrounded
 | by whitespace. The base defaults to 10. Valid bases are 0 and 2-36.
 | Base 0 means to interpret the base from the string as an integer literal.
```

```
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |  abs(self)
 |
 |  __add__(self, value, /)
 |  Return self+value.
```

如您所见， `help()` 函数使用所有方法吐出整个 `int` 类，它还在需要的地方包含说明。

现在，假设您想知道 `str` 类的 `index()` 方法所需的参数，要找出您需要在 python shell 中键入以下命令。

```
>>> help(str.index)

Help on method_descriptor:

index(...)
S.index(sub[, start[, end]]) -> int

Like S.find() but raise ValueError when the substring is not found.
```

在下一篇文章中，我们将学习 python 中的数据类型和变量。

数据类型和变量

> 原文: https://thepythonguru.com/datatype-varibles/

于 2020 年 1 月 7 日更新

变量是命名位置，用于存储对内存中存储的对象的引用。 我们为变量和函数选择的名称通常称为标识符。 在 Python 中，标识符必须遵守以下规则。

1. 所有标识符都必须以字母或下划线（ `_` ）开头，您不能使用数字。 例如： `my_var` 是有效的标识符，但 `1digit` 不是。
2. 标识符可以包含字母，数字和下划线（ `_` ）。 例如： `error_404` ， `_save` 是有效的标识符，但 `$name$` （不允许 `$` ）和 `#age` （不允许 `#` ）是无效的标识符。
3. 它们可以是任何长度。
4. 标识符不能是关键字。 关键字是 Python 用于特殊目的的保留字）。 以下是 Python 3 中的关键字。

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
pass       else       import     assert
break      except     in         raise
```

给变量赋值

值是程序可以使用的基本东西。 例如： `1` ， `11` ， `3.14` 和 `"hello"` 均为值。 在编程术语中，它们通常也称为字面值。 字面值可以是不同的类型，例如 `1` ， `11` 是 `int` 类型， `3.14` 是 `float` 和 `"hello"` 是 `string` 。 记住，在 Python 中，所有东西都是对象，甚至是基本数据类型，例如 int，float，string，我们将在后面的章节中对此进行详细说明。

在 Python 中，您不需要提前声明变量类型。 解释器通过包含的数据自动检测变量的类型。 要将值赋给变量等号（ `=` ）。 `=` 符号也称为赋值运算符。

以下是变量声明的一些示例：

```
x = 100                         # x is integer
```

```
pi = 3.14                    # pi is float
empname = "python is great"   # empname is string

a = b = c = 100 # this statement assign 100 to c, b and a.
```

试试看：

```
x = 100                      # x is integer
pi = 3.14                    # pi is float
empname = "python is great"   # empname is string

a = b = c = 100 # this statement assign 100 to c, b and a.

print(x)  # print the value of variable x
print(pi)  # print the value of variable pi
print(empname)  # print the value of variable empname
print(a, b, c)  # print the value of variable a, b, and c, simultaneously
```

提示：

将值分配给变量后，该变量本身不存储值。 而是，变量仅将对象的引用（地址）存储在内存中。 因此，在上面的清单中，变量 `x` 存储对 `100` （ `int` 对象）的引用（或地址）。 变量 `x` 本身不存储对象 `100` 。

注释

注释是说明程序目的或程序工作方式的注释。 注释不是 Python 解释器在运行程序时执行的编程语句。 注释也用于编写程序文档。 在 Python 中，任何以井号（ `#` ）开头的行均被视为注释。 例如：

```
# This program prints "hello world"

print("hello world")
```

试一试：

```
# This program prints "hello world"

print("hello world")
```

在此清单中，第 1 行是注释。 因此，在运行程序时，Python 解释器将忽略它。

我们还可以在语句末尾写注释。 例如：

```
1.  # This program prints "hello world"
2.
3.  print("hello world")  # display "hello world"
```

当注释以这种形式出现时，它们称为最终注释。

试一试：

```
1.  # This program prints "hello world"
2.
3.  print("hello world")  # display hello world
```

同时赋值

同时赋值或多重赋值允许我们一次将值赋给多个变量。 同时分配的语法如下：

```
1.  var1, var2, ..., varn = exp1, exp2, ..., expn
```

该语句告诉 Python 求值右边的所有表达式，并将它们分配给左边的相应变量。 例如：

```
1.  a, b = 10, 20
2.
3.  print(a)
4.  print(b)
```

试一试：

```
1.  a, b = 10, 20
2.
3.  print(a)
4.  print(b)
```

当您想交换两个变量的值时，同时分配非常有帮助。 例如：

```
1.  >>> x = 1   # initial value of x is 1
2.  >>> y = 2   # initial value of y is 2
```

```
3.
4.  >>> y, x = x, y # assign y value to x and x value to y
```

预期输出：

```
1.  >>> print(x)  # final value of x is 2
2.  2
3.  >>> print(y)  # final value of y is 1
4.  1
```

试一试：

```
1.  x = 1   # initial value of x is 1
2.  y = 2   # initial value of y is 2
3.
4.  y, x = x, y # assign y value to x and x value to y
5.
6.  print(x)  # final value of x is 2
7.  print(y)  # final value of y is 1
```

Python 数据类型

Python 即有 5 种标准数据类型。

1. 数字
2. 字符串
3. 列表
4. 元组
5. 字典
6. 布尔值 - 在 Python 中，`True` 和 `False` 是布尔字面值。 但是以下值也被认为是 `False` 。
    - `0` - `0` , `0.0`
    - `[]` - 空列表, `()` -空元组, `{}` -空字典, `''`
    - `None`

从控制台接收输入

`input()` 函数用于从控制台接收输入。

语法： `input([prompt]) -> string`

`input()` 函数接受一个名为 `prompt` 的可选字符串参数，并返回一个字符串。

```
>>> name = input("Enter your name: ")
>>> Enter your name: tim
>>> name
'tim'
```

试一试：

```
name = input("Enter your name: ")
print(name)
```

请注意，即使输入了数字， `input()` 函数也始终返回字符串。 要将其转换为整数，可以使用 `int()` 或 `eval()` 函数。

```
>>> age = int(input("Enter your age: "))
Enter your age: 22
>>> age
22
>>> type(age)
<class 'int'>
```

试一试：

```
age = int(input("Enter your age: "))

print(age)
print(type(age))
```

导入模块

Python 使用模块组织代码。 Python 随附了许多内置模块，可用于例如数学相关函数的 `math` 模块，正则表达式的 `re` 模块，与操作系统相关的函数的 `os` 模块等。

要使用模块，我们首先使用 `import` 语句将其导入。 其语法如下：

```
import module_name
```

我们还可以使用以下语法导入多个模块：

```
import module_name_1, module_name_2
```

这是一个例子

```
>>> import math, os
>>>
>>> math.pi
3.141592653589793
>>>
>>> math.e
2.718281828459045
>>>
>>>
>>> os.getcwd()  # print current working directory
>>> '/home/user'
>>>
```

试一试：

```
import math, os

print(math.pi)
print(math.e)

print(os.getcwd())
```

在此清单中，第一行导入了 `math` 和 `os` 模块中定义的所有函数，类，变量和常量。 要访问模块中定义的对象，我们首先编写模块名称，后跟点（ `.` ），然后编写对象本身的名称。 （即类或函数或常量或变量）。 在上面的示例中，我们从 `math` 数学访问两个常见的数学常数 `pi` 和 `e` 。 在下一行中，我们将调用 `os` 模块的 `getcwd()` 函数，该函数将打印当前工作目录。

在下一章中，我们将介绍 Python 中的数字。

Python 数字

> 原文: https://thepythonguru.com/python-numbers/

于 2020 年 5 月 7 日更新

此数据类型仅支持诸如 `1` , `31.4` , `-1000` , `0.000023` 和 `88888888` 之类的数值。

Python 支持 3 种不同的数字类型。

1. `int` -用于整数值，例如 `1` , `100` , `2255` , `-999999` , `0` 和 `12345678` 。
2. `float` -用于像 `2.3` , `3.14` , `2.71` , `-11.0` 之类的浮点值。
3. `complex` -适用于 `3+2j` , `-2+2.3j` , `10j` , `4.5+3.14j` 等复数。

整数

python 中的整数字面值属于 `int` 类。

```
>>> i = 100
>>> i
100
```

浮点数

浮点数是带有小数点的值。

```
>>> f = 12.3
>>> f
12.3
```

需要注意的一点是，当数字运算符的操作数之一是浮点值时，结果将是浮点值。

```
>>> 3 * 1.5
4.5
```

复数

如您所知，复数由实部和虚部两部分组成，用 `j` 表示。 您可以这样定义复数：

```
>>> x = 2 + 3j # where 2 is the real part and 3 is imaginary
```

确定类型

Python 具有 `type()` 内置函数，可用于确定变量的类型。

```
>>> x = 12
>>> type(x)
 <class 'int'>
```

Python 运算符

Python 具有不同的运算符，可让您在程序中执行所需的计算。

`+` ， `-` 和 `*` 可以正常工作，其余的运算符需要一些解释。

名称	含义	示例	结果
`+`	加法	`15+20`	`35`
`-`	减法	`24.5-3.5`	`21.0`
`*`	乘法	`15*4`	`60`
`/`	浮点除法	`4/5`	`0.8`
`//`	整数除法	`4//5`	`0`
`**`	求幂	`4**2`	`16`
`%`	余数	`27%4`	`3`

浮点除法（ `/` ）： `/` 运算符进行除法并以浮点数形式返回结果，这意味着它将始终返回小数部分。 例如

```
>>> 3/2
1.5
```

整数除法（ `//` ）： `//` 执行整数除法，即它将截断答案的小数部分并仅返回整数。

```
>>> 3//2
1
```

幂运算符（`）**：此运算符有助于计算 ^b`（a 的 b 次幂）。 让我们举个例子：

```
>>> 2 ** 3 # is same as 2 * 2 * 2
8
```

余数运算符（ % ）： % 运算符也称为余数或模数运算符。 该运算符除法后返回余数。 例如：

```
>>> 7 % 2
1
```

运算符优先级

在 python 中，每个表达式都使用运算符优先级进行求值。 让我们以一个例子来阐明它。

```
>>> 3 * 4 + 1
```

在上面的表达式中，将对哪个运算进行第一个加法或乘法运算？ 为了回答这样的问题，我们需要在 python 中引用运算符优先级列表。 下图列出了 python 优先级从高到低的顺序。

Operator	Description
`()`	Parentheses (grouping)
`f(args...)`	Function call
`x[index:index]`	Slicing
`x[index]`	Subscription
`x.attribute`	Attribute reference
`**`	Exponentiation
`~x`	Bitwise not
`+x, -x`	Positive, negative
`*, /, %`	Multiplication, division, remainder
`+, -`	Addition, subtraction
`<<, >>`	Bitwise shifts
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`in, not in, is, is not, <, <=, >, >=, <>, !=, ==`	Comparisons, membership, identity
`not x`	Boolean NOT
`and`	Boolean AND
`or`	Boolean OR
`lambda`	Lambda expression

如您在上表中所见 `*` 在 `+` 之上，因此 `*` 将首先出现，然后加法。 因此，以上表达式的结果将为 `13` 。

```
1.  >>> 3 * 4 + 1
2.  >>> 13
```

让我们再举一个例子来说明另一个概念。

```
1.  >>> 3 + 4 - 2
```

在以上表达式中，将首先进行加法或减法。 从表 `+` 和 `-` 的优先级相同，然后将它们从左到右进行求值，即先加法，然后减法。

```
1.  >>> 3 + 4 - 2
2.  >>> 5
```

该规则的唯一例外是赋值运算符（ `=` ），它从右到左出现。

```
1.  a = b = c
```

您可以使用括号 `()` 更改优先级，例如：

```
1.  >> 3 * (4 + 1)
2.  >> 15
```

从优先级表中可以看出， `()` 具有最高优先级，因此在表达式 `3 * (4 + 1)` 中，先求值 `(4 + 1)` ，然后相乘。 因此，您可以使用 `()` 更改优先级。

复合赋值

这些运算符使您可以编写快捷方式分配语句。 例如：

```
1.  >>> count = 1
2.  >>> count = count + 1
3.  >>> count
4.  2
```

通过使用增强分配运算符，我们可以将其编写为：

```
1.  >>> count = 1
2.  >>> count += 1
3.  >>> count
4.  2
```

类似地，您可以将 `-` ， `%` ， `//` ， `/` ， `*` 和 `**` 与赋值运算符一起使用以构成扩展赋值运算符。

运算符	名称	示例	等价于
`+=`	加法赋值	`x += 4`	`x = x + 4`
`-=`	减法赋值	`x -= 2`	`x = x - 2`
`*=`	乘法赋值	`x *= 5`	`x = x * 5`
`/*=`	除法赋值	`x /= 5`	`x = x / 5`
`//*=`	整数除法赋值	`x //= 5`	`x = x // 5`
`%*=`	余数赋值	`x %= 5`	`x = x % 5`
`**=`	指数赋值	`x **= 5`	`x = x ** 5`

在下一篇文章中，我们将学习 python 字符串。

Python 字符串

> 原文: https://thepythonguru.com/python-strings/

于 2020 年 1 月 10 日更新

python 中的字符串是由单引号或双引号分隔的连续字符系列。 Python 没有任何单独的字符数据类型，因此它们表示为单个字符串。

创建字符串

```
>>> name = "tom" # a string
>>> mychar = 'a' # a character
```

您还可以使用以下语法创建字符串。

```
>>> name1 = str() # this will create empty string object
>>> name2 = str("newstring") # string object containing 'newstring'
```

```
name = "tom" # a string
mychar = 'a' # a character

print(name)
print(mychar)

name1 = str() # this will create empty string object
name2 = str("newstring") # string object containing 'newstring'

print(name1)
print(name2)
```

Python 中的字符串是不可变的

这对您而言意味着，一旦创建了字符串，便无法对其进行修改。 让我们以一个例子来说明这一点。

```
>>> str1 = "welcome"
>>> str2 = "welcome"
```

这里 `str1` 和 `str2` 指的是存储在内存中某个位置的相同字符串对象“ `welcome` ”。 您可以使用 `id()` 函数测试 `str1` 是否与 `str2` 引用相同的对象。

什么是身份证？

python 中的每个对象都存储在内存中的某个位置。 我们可以使用 `id()` 获得该内存地址。

```
>>> id(str1)
78965411
>>> id(str2)
78965411
```

由于 `str1` 和 `str2` 都指向相同的存储位置，因此它们都指向同一对象。

让我们尝试通过向其添加新字符串来修改 `str1` 对象。

```
>>> str1 += " mike"
>>> str1
welcome mike
>>> id(str1)
>>> 78965579
```

如您现在所见， `str1` 指向完全不同的内存位置，这证明了并置不会修改原始字符串对象而是创建一个新的字符串对象这一点。 同样，数字（即 `int` 类型）也是不可变的。

试试看：

```
str1 = "welcome"
str2 = "welcome"

print(id(str1),  id(str2))

str1 += " mike"

print(str1)

print(id(str1))
```

字符串操作

字符串索引从 `0` 开始，因此要访问字符串类型中的第一个字符：

```
1.  >>> name[0] #
2.  t
```

试一试：

```
1.  name = "tom"
2.
3.  print(name[0])
4.  print(name[1])
```

`+` 运算符用于连接字符串，而 `*` 运算符是字符串的重复运算符。

```
1.  >>> s = "tom and " + "jerry"
2.  >>> print(s)
3.  tom and jerry
```

```
1.  >>> s = "spamming is bad " * 3
2.  >>> print(s)
3.  'spamming is bad spamming is bad spamming is bad '
```

试一试：

```
1.  s = "tom and " + "jerry"
2.  print(s)
3.
4.  s = "spamming is bad " * 3
5.  print(s)
```

字符串切片

您可以使用 `[]` 运算符（也称为切片运算符）从原始字符串中提取字符串的子集。

语法： `s[start:end]`

这将从索引 `start` 到索引 `end - 1` 返回字符串的一部分。

让我们举一些例子。

```
>>> s = "Welcome"
>>> s[1:3]
el
```

一些更多的例子。

```
>>> s = "Welcome"
>>>
>>> s[:6]
'Welcom'
>>>
>>> s[4:]
'ome'
>>>
>>> s[1:-1]
'elcom'
```

试一试：

```
s = "Welcome"

print(s[1:3])
print(s[:6])
print(s[4:])
print(s[1:-1])
```

注意：

`start` 索引和 `end` 索引是可选的。 如果省略，则 `start` 索引的默认值为 `0` ，而 `end` 的默认值为字符串的最后一个索引。

`ord()` 和 `chr()` 函数

`ord()` -函数返回字符的 ASCII 码。

`chr()` -函数返回由 ASCII 数字表示的字符。

```
>>> ch = 'b'
>>> ord(ch)
```

```
98
>>> chr(97)
'a'
>>> ord('A')
65
```

试一试：

```
ch = 'b'

print(ord(ch))

print(chr(97))

print(ord('A'))
```

Python 中的字符串函数

函数名称	函数说明
`len ()`	返回字符串的长度
`max()`	返回具有最高 ASCII 值的字符
`min()`	返回具有最低 ASCII 值的字符

```
>>> len("hello")
5
>>> max("abc")
'c'
>>> min("abc")
'a'
```

试一试：

```
print(len("hello"))

print(max("abc"))

print(min("abc"))
```

`in` 和 `not in` 运算符

您可以使用 `in` 和 `not in` 运算符检查另一个字符串中是否存在一个字符串。 他们也被称为会员运营商。

```
>>> s1 = "Welcome"
>>> "come" in s1
True
>>> "come" not in s1
False
>>>
```

试一试:

```
s1 = "Welcome"

print("come" in s1)

print("come" not in s1)
```

字符串比较

您可以使用[`>` , `<` , `<=` , `<=` , `==` , `!=`) 比较两个字符串。 Python 按字典顺序比较字符串，即使用字符的 ASCII 值。

假设您将 `str1` 设置为 `"Mary"` 并将 `str2` 设置为 `"Mac"` 。 比较 `str1` 和 `str2` 的前两个字符（ `M` 和 `M` ）。 由于它们相等，因此比较后两个字符。 因为它们也相等，所以比较了前两个字符（ `r` 和 `c` ）。 并且因为 `r` 具有比 `c` 更大的 ASCII 值，所以 `str1` 大于 `str2` 。

这里还有更多示例:

```
>>> "tim" == "tie"
False
>>> "free" != "freedom"
True
>>> "arrow" > "aron"
True
>>> "right" >= "left"
True
>>> "teeth" < "tee"
False
```

```
>>> "yellow" <= "fellow"
False
>>> "abc" > ""
True
>>>
```

试一试：

```
print("tim" == "tie")

print("free" != "freedom")

print("arrow" > "aron")

print("right" >= "left")

print("teeth" < "tee")

print("yellow" <= "fellow")

print("abc" > "")
```

使用 `for` 循环迭代字符串

字符串是一种序列类型，也可以使用 `for` 循环进行迭代（要了解有关 `for` 循环的更多信息，请单击此处）。

```
>>> s = "hello"
>>> for i in s:
...     print(i, end="")
hello
```

注意：

默认情况下， `print()` 函数用换行符打印字符串，我们通过传递名为 `end` 的命名关键字参数来更改此行为，如下所示。

```
print("my string", end="\n")  # this is default behavior
print("my string", end="")    # print string without a newline
print("my string", end="foo") # now print() will print foo after every string
```

试一试：

```
s = "hello"
for i in s:
    print(i, end="")
```

测试字符串

python 中的字符串类具有各种内置方法，可用于检查不同类型的字符串。

方法名称	方法说明
`isalnum()`	如果字符串是字母数字，则返回 `True`
`isalpha()`	如果字符串仅包含字母，则返回 `True`
`isdigit()`	如果字符串仅包含数字，则返回 `True`
`isidentifier()`	返回 `True` 是字符串是有效的标识符
`islower()`	如果字符串为小写，则返回 `True`
`isupper()`	如果字符串为大写则返回 `True`
`isspace()`	如果字符串仅包含空格，则返回 `True`

```
>>> s = "welcome to python"
>>> s.isalnum()
False
>>> "Welcome".isalpha()
True
>>> "2012".isdigit()
True
>>> "first Number".isidentifier()
False
>>> s.islower()
True
>>> "WELCOME".isupper()
True
>>> " \t".isspace()
True
```

试一试：

```
s = "welcome to python"

```

```
print(s.isalnum())
print("Welcome".isalpha())
print("2012".isdigit())
print("first Number".isidentifier())
print(s.islower())
print("WELCOME".isupper())
print("  \t".isspace())
```

搜索子串

方法名称	方法说明
`endwith(s1: str): bool`	如果字符串以子字符串 `s1` 结尾，则返回 `True`
`startswith(s1: str): bool`	如果字符串以子字符串 `s1` 开头，则返回 `True`
`count(s: str): int`	返回字符串中子字符串出现的次数
`find(s1): int`	返回字符串中 `s1` 起始处的最低索引，如果找不到字符串则返回 `-1`
`rfind(s1): int`	从字符串中 `s1` 的起始位置返回最高索引，如果找不到字符串则返回 `-1`

```
>>> s = "welcome to python"
>>> s.endswith("thon")
True
>>> s.startswith("good")
False
>>> s.find("come")
3
>>> s.find("become")
-1
>>> s.rfind("o")
15
>>> s.count("o")
3
>>>
```

试一试：

```
s = "welcome to python"

print(s.endswith("thon"))

print(s.startswith("good"))
```

```
6.
7.  print(s.find("come"))
8.
9.  print(s.find("become"))
10.
11. print(s.rfind("o"))
12.
13. print(s.count("o"))
```

转换字符串

方法名称	方法说明
`capitalize(): str`	返回此字符串的副本，仅第一个字符大写。
`lower(): str`	通过将每个字符转换为小写来返回字符串
`upper(): str`	通过将每个字符转换为大写来返回字符串
`title(): str`	此函数通过大写字符串中每个单词的首字母来返回字符串
`swapcase(): str`	返回一个字符串，其中小写字母转换为大写，大写字母转换为小写
`replace(old, new): str`	此函数通过用新字符串替换旧字符串的出现来返回新字符串

```
1.  s = "string in python"
2.  >>>
3.  >>> s1 = s.capitalize()
4.  >>> s1
5.  'String in python'
6.  >>>
7.  >>> s2 = s.title()
8.  >>> s2
9.  'String In Python'
10. >>>
11. >>> s = "This Is Test"
12. >>> s3 = s.lower()
13. >>> s3
14. 'this is test'
15. >>>
16. >>> s4 = s.upper()
17. >>> s4
18. 'THIS IS TEST'
19. >>>
20. >>> s5 = s.swapcase()
21. >>> s5
```

```
'tHIS iS tEST'
>>>
>>> s6 = s.replace("Is", "Was")
>>> s6
'This Was Test'
>>>
>>> s
'This Is Test'
>>>
```

试一试:

```
s = "string in python"

s1 = s.capitalize()
print(s1)

s2 = s.title()
print(s2)

s = "This Is Test"
s3 = s.lower()

print(s3)

s4 = s.upper()
print(s4)

s5 = s.swapcase()
print(s5)

s6 = s.replace("Is", "Was")
print(s6)

print(s)
```

在下一章中，我们将学习 python 列表

Python 列表

> 原文: https://thepythonguru.com/python-lists/

于 2020 年 1 月 7 日更新

列表类型是 python 的列表类定义的另一种序列类型。 列表允许您以非常简单的方式添加，删除或处理元素。 列表与数组非常相似。

在 python 中创建列表

您可以使用以下语法创建列表。

```
1.  >>> l = [1, 2, 3, 4]
```

在此，列表中的每个元素都用逗号分隔，并用一对方括号（ `[]` ）包围。 列表中的元素可以是相同类型或不同类型。 例如：

```
1.  l2 = ["this is a string", 12]
```

创建列表的其他方式。

```
1.  list1 = list() # Create an empty list
2.  list2 = list([22, 31, 61]) # Create a list with elements 22, 31, 61
3.  list3 = list(["tom", "jerry", "spyke"]) # Create a list with strings
4.  list5 = list("python") # Create a list with characters p, y, t, h, o, n
```

注意：

列表是可变的。

访问列表中的元素

您可以使用索引运算符（ `[]` ）访问列表中的各个元素。 列表索引从 `0` 开始。

```
1.  >>> l = [1,2,3,4,5]
```

```
>>> l[1] # access second element in the list
2
>>> l[0] # access first element in the list
1
```

常用列表操作

方法名称	描述
`x in s`	如果元素 `x` 在序列 `s` 中，则为 `True` ，否则为 `False`
`x not in s`	如果元素 `x` 不在序列 `s` 中，则为 `True` ，否则为 `False`
`s1 + s2`	连接两个序列 `s1` 和 `s2`
`s * n` ， `n * s`	连接序列 `s` 的 `n` 个副本
`s[i]`	序列 `s` 的第 `i` 个元素。
`len(s)`	序列 `s` 的长度，也就是元素数量。
`min(s)`	序列 `s` 中最小的元素。
`max(s)`	序列 `s` 中最大的元素。
`sum(s)`	序列 `s` 中所有数字的总和。
`for` 循环	在 `for` 循环中从左到右遍历元素。

列表函数示例

```
>>> list1 = [2, 3, 4, 1, 32]
>>> 2 in list1
True
>>> 33 not in list1
True
>>> len(list1) # find the number of elements in the list
5
>>> max(list1) # find the largest element in the list
32
>>> min(list1) # find the smallest element in the list
1
>>> sum(list1) # sum of elements in the list
42
```

列表切片

切片运算符（ `[start:end]` ）允许从列表中获取子列表。 它的工作原理类似于字符串。

```
>>> list = [11,33,44,66,788,1]
>>> list[0:5] # this will return list starting from index 0 to index 4
[11,33,44,66,788]
```

```
>>> list[:3]
[11,33,44]
```

类似于字符串 `start` 的索引是可选的，如果省略，它将为 `0` 。

```
>>> list[2:]
[44,66,788,1]
```

`end` 索引也是可选的，如果省略，它将被设置为列表的最后一个索引。

注意：

如果为 `start >= end` ，则 `list[start : end]` 将返回一个空列表。 如果 `end` 指定的位置超出列表的 `end` ，则 Python 将使用 `end` 的列表长度。

列表中的 `+` 和 `*` 运算符

`+` 运算符加入两个列表。

```
>>> list1 = [11, 33]
>>> list2 = [1, 9]
>>> list3 = list1 + list2
>>> list3
[11, 33, 1, 9]
```

`*` 操作符复制列表中的元素。

```
>>> list4 = [1, 2, 3, 4]
>>> list5 = list4 * 3
>>> list5
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
```

`in` 或 `not in` 运算符

`in` 运算符用于确定列表中是否存在元素。 成功则返回 `True` ；失败则返回 `False` 。

```
>>> list1 = [11, 22, 44, 16, 77, 98]
>>> 22 in list1
True
```

同样， `not in` 与 `in` 运算符相反。

```
>>> 22 not in list1
False
```

使用 `for` 循环遍历列表

如前所述，列表是一个序列并且也是可迭代的。 意味着您可以使用 `for` 循环遍历列表的所有元素。

```
>>> list = [1,2,3,4,5]
>>> for i in list:
... print(i, end=" ")
1 2 3 4 5
```

常用列表方法和返回类型

方法	描述
`append(x: object): None`	在列表的末尾添加元素 `x` 并返回 `None` 。
`count(x: object): int`	返回元素 `x` 在列表中出现的次数。
`append(l: list): None`	将 `l` 中的所有元素附加到列表中并返回 `None` 。
`index(x: object): int`	返回列表中第一次出现的元素 `x` 的索引
`insert(index: int, x: object): None`	在给定索引处插入元素 `x` 。 请注意，列表中的第一个元素具有索引 `0` 并返回 `None` 。
`remove(x: object): None`	从列表中删除第一次出现的元素 `x` 并返回 `None`
`reverse(): None`	反转列表并返回 `None`
`sort(): None`	按升序对列表中的元素进行排序并返回 `None` 。
`pop(i): object`	删除给定位置的元素并返回它。 参数 `i` 是可选的。 如果未指定，则 `pop()` 删除并返回列表中的最后一个元素。

```
>>> list1 = [2, 3, 4, 1, 32, 4]
```

```
>>> list1.append(19)
>>> list1
[2, 3, 4, 1, 32, 4, 19]
>>> list1.count(4) # Return the count for number 4
2
>>> list2 = [99, 54]
>>> list1.extend(list2)
>>> list1
[2, 3, 4, 1, 32, 4, 19, 99, 54]
>>> list1.index(4) # Return the index of number 4
2
>>> list1.insert(1, 25) # Insert 25 at position index 1
>>> list1
[2, 25, 3, 4, 1, 32, 4, 19, 99, 54]
>>>
>>> list1 = [2, 25, 3, 4, 1, 32, 4, 19, 99, 54]
>>> list1.pop(2)
3
>>> list1
[2, 25, 4, 1, 32, 4, 19, 99, 54]
>>> list1.pop()
54
>>> list1
[2, 25, 4, 1, 32, 4, 19, 99]
>>> list1.remove(32) # Remove number 32
>>> list1
[2, 25, 4, 1, 4, 19, 99]
>>> list1.reverse() # Reverse the list
>>> list1
[99, 19, 4, 1, 4, 25, 2]
>>> list1.sort() # Sort the list
>>> list1
[1, 2, 4, 4, 19, 25, 99]
>>>
```

列表推导式

注意：

本主题需要具有 Python 循环的使用知识。

列表理解为创建列表提供了一种简洁的方法。 它由包含表达式的方括号组成，后跟 `for` 子句，然后是零个或多个 `for` 或 `if` 子句。

这里有些例子：

```
>>> list1 = [ x for x in range(10) ]
>>> list1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>>
>>> list2 = [ x + 1 for x in range(10) ]
>>> list2
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>>
>>>
>>> list3 = [ x for x in range(10) if x % 2 == 0 ]
>>> list3
[0, 2, 4, 6, 8]
>>>
>>>
>>> list4 = [ x *2 for x in range(10) if x % 2 == 0 ]
[0, 4, 8, 12, 16]
```

在下一个教程中，我们将学习 python 字典。

Python 字典

> 原文: https://thepythonguru.com/python-dictionaries/

于 2020 年 1 月 7 日更新

字典是一种 python 数据类型，用于存储键值对。 它使您可以使用键快速检索，添加，删除，修改值。 字典与我们在其他语言上称为关联数组或哈希的非常相似。

注意：

字典是可变的。

创建字典

可以使用一对大括号（ `{}` ）创建字典。 字典中的每个项目都由一个键，一个冒号，一个值组成。 每个项目都用逗号（ `,` ）分隔。 让我们举个例子。

```
1.  friends = {
2.  'tom' : '111-222-333',
3.  'jerry' : '666-33-111'
4.  }
```

这里 `friends` 是有两个项目的字典。 需要注意的一点是，键必须是可哈希的类型，但是值可以是任何类型。 字典中的每个键都必须是唯一的。

```
1.  >>> dict_emp = {} # this will create an empty dictionary
```

检索，修改和向字典中添加元素

要从字典中获取项目，请使用以下语法：

```
1.  >>> dictionary_name['key']
```

```
1.  >>> friends['tom']
2.  '111-222-333'
```

如果字典中存在键，则将返回值，否则将引发 `KeyError` 异常。 要添加或修改项目，请使用以下语法：

```
>>> dictionary_name['newkey'] = 'newvalue'
```

```
>>> friends['bob'] = '888-999-666'
>>> friends
 {'tom': '111-222-333', 'bob': '888-999-666', 'jerry': '666-33-111'}
```

从字典中删除项目

```
>>> del dictionary_name['key']
```

```
>>>  del friends['bob']
>>>  friends
{'tom': '111-222-333', 'jerry': '666-33-111'}
```

如果找到键，则该项目将被删除，否则将抛出 `KeyError` 异常。

遍历字典中的项目

您可以使用 `for` 循环遍历字典中的元素。

```
>>> friends = {
... 'tom' : '111-222-333',
... 'jerry' : '666-33-111'
...}
>>>
>>> for key in friends:
...     print(key, ":", friends[key])
...
tom : 111-222-333
jerry : 666-33-111
>>>
>>>
```

查找字典的长度

您可以使用 `len()` 函数查找字典的长度。

```
1.  >>> len(friends)
2.  2
```

`in` 和 `not in` 运算符

`in` 和 `not in` 运算符检查字典中是否存在键。

```
1.  >>> 'tom' in friends
2.  True
3.  >>> 'tom' not in friends
4.  False
```

字典中的相等测试

`==` 和 `!=` 运算符告诉字典是否包含相同的项目。

```
1.  >>> d1 = {"mike":41, "bob":3}
2.  >>> d2 = {"bob":3, "mike":41}
3.  >>> d1 == d2
4.  True
5.  >>> d1 != d2
6.  False
7.  >>>
```

注意：

您不能使用 `<` ， `>` ， `>=` ， `<=` 等其他关系运算符来比较字典。

字典方法

Python 提供了几种内置的方法来处理字典。

方法	描述
`popitem()`	返回字典中随机选择的项目，并删除所选项目。

`clear()`	删除字典中的所有内容
`keys()`	以元组形式返回字典中的键
`values()`	以元组形式返回字典中的值
`get(key)`	键的返回值，如果找不到键，则返回 `None`，而不是引发 `KeyError` 异常
`pop(key)`	从字典中删除该项目，如果找不到该键，则会抛出 `KeyError`

```
>>> friends = {'tom': '111-222-333', 'bob': '888-999-666', 'jerry': '666-33-111'}
>>>
>>> friends.popitem()
('tom', '111-222-333')
>>>
>>> friends.clear()
>>>
>>> friends
{}
>>>
>>> friends = {'tom': '111-222-333', 'bob': '888-999-666', 'jerry': '666-33-111'}
>>>
>>> friends.keys()
dict_keys(['tom', 'bob', 'jerry'])
>>>
>>> friends.values()
dict_values(['111-222-333', '888-999-666', '666-33-111'])
>>>
>>> friends.get('tom')
'111-222-333'
>>>
>>> friends.get('mike', 'Not Exists')
'Not Exists'
>>>
>>> friends.pop('bob')
'888-999-666'
>>>
>>> friends
{'tom': '111-222-333', 'jerry': '666-33-111'}
```

在下一篇文章中，我们将学习 Python 元组。

Python 元组

> 原文: https://thepythonguru.com/python-tuples/

于 2020 年 1 月 7 日更新

在 Python 中，元组与列表非常相似，但是一旦创建了元组，就无法添加，删除，替换和重新排序元素。

注意:

元组是不可变的。

创建一个元组

```
>>> t1 = () # creates an empty tuple with no data
>>>
>>> t2 = (11,22,33)
>>>
>>> t3 = tuple([1,2,3,4,4]) # tuple from array
>>>
>>> t4 = tuple("abc") # tuple from string
```

元组函数

元组也可以使用 `max()` ， `min()` ， `len()` 和 `sum()` 之类的函数。

```
>>> t1 = (1, 12, 55, 12, 81)
>>> min(t1)
1
>>> max(t1)
81
>>> sum(t1)
161
>>> len(t1)
5
```

元组迭代

元组可使用 `for` 循环进行迭代，在此处了解有关 `for` 循环的更多信息。

```
>>> t = (11,22,33,44,55)
>>> for i in t:
...     print(i, end=" ")
>>> 11 22 33 44 55
```

元组切片

切片运算符在元组中的作用与在列表和字符串中的作用相同。

```
>>> t = (11,22,33,44,55)
>>> t[0:2]
(11,22)
```

`in` 和 `not in` 运算符

您可以使用 `in` 和 `not in` 运算符检查元组中项的存在，如下所示。

```
>>> t = (11,22,33,44,55)
>>> 22 in t
True
>>> 22 not in t
False
```

在下一章中，我们将学习 python 数据类型转换。

数据类型转换

> 原文: https://thepythonguru.com/datatype-conversion/

于 2020 年 1 月 7 日更新

偶尔，您会希望将一种类型的数据类型转换为另一种类型。 数据类型转换也称为类型转换。

将 `int` 转换为 `float`

要将 `int` 转换为 `float` ，可以使用 `float()` 函数。

```
>>> i = 10
>>> float(i)
10.0
```

将 `float` 转换为 `int`

要将 `float` 转换为 `int` ，您需要使用 `int()` 函数。

```
>>> f = 14.66
>>> int(f)
14
```

将字符串转换为 `int`

要将 `string` 转换为 `int` ，请使用 `int()` 函数。

```
>>> s = "123"
>>> int(s)
123
```

提示：

如果字符串包含非数字字符，则 `int()` 将引发 `ValueError` 异常。

将数字转换为字符串

要将数字转换为字符串，请使用 `str()` 函数。

```
>>> i = 100
>>> str(i)
"100"
>>> f = 1.3
str(f)
'1.3'
```

舍入数字

四舍五入数字是通过 `round()` 函数完成的。

语法： `round(number[, ndigits])`

```
>>> i = 23.97312
>>> round(i, 2)
23.97
```

接下来，我们将介绍控制语句。

Python 控制语句

> 原文: https://thepythonguru.com/python-control-statements/

于 2020 年 1 月 7 日更新

程序根据某些条件执行语句是很常见的。 在本节中，我们将了解 Python 中的 `if else` 语句。

但是在我们需要了解关系运算符之前。 关系运算符使我们可以比较两个对象。

符号	描述
`<=`	小于或等于
`<`	小于
`>`	大于
`>=`	大于或等于
`==`	等于
`!=`	不等于

比较的结果将始终为布尔值，即 `True` 或 `False` 。 请记住， `True` 和 `False` 是用于表示布尔值的 python 关键字。

让我们举一些例子：

```
>>> 3 == 4
False
>>> 12 > 3
True
>>> 12 == 12
True
>>> 44 != 12
True
```

现在您可以处理 `if` 语句了。 `if` 语句的语法如下所示：

```
if boolean-expression:
   #statements
else:
   #statements
```

注意：

`if` 块中的每个语句都必须使用相同数量的空格缩进，否则将导致语法错误。 这与 Java，C，C# 等使用花括号（ `{}` ）的语言完全不同。

现在来看一个例子

```
1.  i = 10
2.
3.  if i % 2 == 0:
4.     print("Number is even")
5.  else:
6.     print("Number is odd")
```

在这里您可以看到，如果数字为偶数，则将打印 `"Number is even"` 。 否则打印 `"Number is odd"` 。

注意：

`else` 子句是可选的，您可以根据需要仅使用 `if` 子句，如下所示：

```
1.  if today == "party":
2.      print("thumbs up!")
```

在此，如果 `today` 的值为 `"party"` ，则将打印 `thumbs up!` ，否则将不打印任何内容。

如果您的程序需要检查一长串条件，那么您需要使用 `if-elif-else` 语句。

```
1.  if boolean-expression:
2.     #statements
3.  elif boolean-expression:
4.     #statements
5.  elif boolean-expression:
6.     #statements
7.  elif boolean-expression:
8.     #statements
9.  else:
10.    #statements
```

您可以根据程序要求添加 `elif` 条件。

这是一个说明 `if-elif-else` 语句的示例。

```
today = "monday"

if today == "monday":
    print("this is monday")
elif today == "tuesday":
    print("this is tuesday")
elif today == "wednesday":
    print("this is wednesday")
elif today == "thursday":
    print("this is thursday")
elif today == "friday":
    print("this is friday")
elif today == "saturday":
    print("this is saturday")
elif today == "sunday":
    print("this is sunday")
else:
    print("something else")
```

嵌套 `if` 语句

您可以将 `if` 语句嵌套在另一个 `if` 语句中，如下所示：

```
today = "holiday"
bank_balance = 25000
if today == "holiday":
    if bank_balance > 20000:
        print("Go for shopping")
    else:
        print("Watch TV")
else:
    print("normal working day")
```

在下一篇文章中，我们将学习 Python 函数。

Python 函数

> 原文: https://thepythonguru.com/python-functions/

于 2020 年 1 月 7 日更新

函数是可重用的代码段，可帮助我们组织代码的结构。 我们创建函数，以便我们可以在程序中多次运行一组语句，而无需重复自己。

创建函数

Python 使用 `def` 关键字启动函数，以下是语法：

```
def function_name(arg1, arg2, arg3, .... argN):
     #statement inside function
```

注意：

函数内的所有语句均应使用相等的空格缩进。 函数可以接受零个或多个括在括号中的参数（也称为参数）。 您也可以使用 `pass` 关键字来省略函数的主体，如下所示：

```
def myfunc():
    pass
```

让我们来看一个例子。

```
def sum(start, end):
    result = 0
    for i in range(start, end + 1):
        result += i
    print(result)

sum(10, 50)
```

预期输出：

```
1230
```

上面我们定义了一个名为 `sum()` 的函数，它具有两个参数 `start` 和 `end` 。 该函数计算从 `start` 到 `end` 开始的所有数字的总和。

具有返回值的函数。

上面的函数只是将结果打印到控制台，如果我们想将结果分配给变量以进行进一步处理怎么办？ 然后，我们需要使用 `return` 语句。 `return` 语句将结果发送回调用方并退出该函数。

```
1.  def sum(start, end):
2.      result = 0
3.      for i in range(start, end + 1):
4.          result += i
5.      return result
6.
7.  s = sum(10, 50)
8.  print(s)
```

预期输出：

```
1.  1230
```

在这里，我们使用 `return` 语句返回数字总和并将其分配给变量 `s` 。

您也可以使用 `return` 陈述式而没有返回值。

```
1.  def sum(start, end):
2.      if(start > end):
3.          print("start should be less than end")
            return    # here we are not returning any value so a special value None
4.  is returned
5.      result = 0
6.      for i in range(start, end + 1):
7.          result += i
8.      return result
9.
10. s = sum(110, 50)
11. print(s)
```

预期输出：

```
start should be less than end
None
```

在 python 中，如果您未从函数显式返回值，则始终会返回特殊值 `None` 。 让我们举个例子：

```
def test():   # test function with only one statement
    i = 100
print(test())
```

预期输出：

```
None
```

如您所见， `test()` 函数没有显式返回任何值，因此返回了 `None` 。

全局变量与局部变量

全局变量：未绑定到任何函数但可以在函数内部和外部访问的变量称为全局变量。

局部变量：在函数内部声明的变量称为局部变量。

让我们看一些例子来说明这一点。

示例 **1** ：

```
global_var = 12         # a global variable

def func():
    local_var = 100     # this is local variable
    print(global_var)   # you can access global variables in side function

func()                  # calling function func()

#print(local_var)        # you can't access local_var outside the function, because as soon as function ends local_var is destroyed
```

预期输出：

```
12
```

示例 **2**：

```
xy = 100

def cool():
    xy = 200     # xy inside the function is totally different from xy outside
the function
    print(xy)    # this will print local xy variable i.e 200

cool()

print(xy)        # this will print global xy variable i.e 100
```

预期输出：

```
200
100
```

您可以使用 `global` 关键字后跟逗号分隔的变量名称（ `,` ）在全局范围内绑定局部变量。

```
t = 1

def increment():
    global t   # now t inside the function is same as t outside the function
    t = t + 1
    print(t) # Displays 2

increment()
print(t) # Displays 2
```

预期输出：

```
2
2
```

请注意，在全局声明变量时不能为变量赋值。

```
t = 1

def increment():
    #global t = 1   # this is error
    global t
    t = 100        # this is okay
```

```
    t = t + 1
    print(t) # Displays 101

increment()
print(t) # Displays 101
```

预期输出：

```
101
101
```

实际上，无需在函数外部声明全局变量。 您可以在函数内部全局声明它们。

```
def foo():
    global x   # x is declared as global so it is available outside the
function
    x = 100

foo()
print(x)
```

预期输出： 100

具有默认值的参数

要指定参数的默认值，您只需要使用赋值运算符分配一个值。

```
def func(i, j = 100):
    print(i, j)
```

以上函数具有两个参数 `i` 和 `j` 。 参数 `j` 的默认值为 `100` ，这意味着我们可以在调用函数时忽略 `j` 的值。

```
func(2) # here no value is passed to j, so default value will be used
```

预期输出：

```
2 100
```

再次调用 `func()` 函数，但这一次为 `j` 参数提供一个值。

```
    func(2, 300) # here 300 is passed as a value of j, so default value will not be
1.  used
```

预期输出：

```
1.  2 300
```

关键字参数

有两种方法可以将参数传递给方法：位置参数和关键字参数。 在上一节中，我们已经看到了位置参数的工作方式。 在本节中，我们将学习关键字参数。

关键字参数允许您使用像 `name=value` 这样的名称值对来传递每个参数。 让我们举个例子：

```
1.  def named_args(name, greeting):
2.      print(greeting + " " + name )
```

```
1.  named_args(name='jim', greeting='Hello')
2.
3.  named_args(greeting='Hello', name='jim') # you can pass arguments this way too
```

期望值：

```
1.  Hello jim
2.  Hello jim
```

混合位置和关键字参数

可以混合使用位置参数和关键字参数，但是对于此位置参数必须出现在任何关键字参数之前。 我们来看一个例子。

```
1.  def my_func(a, b, c):
2.      print(a, b, c)
```

您可以通过以下方式调用上述函数。

```
1.  # using positional arguments only
```

```
my_func(12, 13, 14)

# here first argument is passed as positional arguments while other two as keyword argument
my_func(12, b=13, c=14)

# same as above
my_func(12, c=13, b=14)

# this is wrong as positional argument must appear before any keyword argument
# my_func(12, b=13, 14)
```

预期输出：

```
12 13 14
12 13 14
12 14 13
```

从函数返回多个值

我们可以使用 `return` 语句从函数中返回多个值，方法是用逗号（ `,` ）分隔它们。 多个值作为元组返回。

```
def bigger(a, b):
    if a > b:
        return a, b
    else:
        return b, a

s = bigger(12, 100)
print(s)
print(type(s))
```

预期输出：

```
(100, 12)
<class 'tuple'>
```

在下一篇文章中，我们将学习 Python 循环

Python 循环

> 原文: https://thepythonguru.com/python-loops/

于 2020 年 1 月 7 日更新

Python 只有两个循环:

1. `for` 循环
2. `while` 循环

`for` 循环

`for` 循环语法:

```
for i in iterable_object:
    # do something
```

注意:

`for` 和 `while` 循环内的所有语句必须缩进相同的空格数。 否则，将抛出 `SyntaxError` 。

让我们举个例子

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

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

预期输出:

```
1
2
3
4
```

这是 `for` 循环的工作方式:

在第一次迭代中，为 `i` 分配了值 `1` ，然后执行了 `print` 语句。 在第二次迭代中，为 `i` 赋

值 `2` ，然后再次执行 `print` 语句。 此过程将继续进行，直到列表中没有其他元素并且存在 `for` 循环为止。

`range(a, b)` 函数

`range(a, b)` 函数从 `a` ， `a + 1` ， `a+ 2` …., `b - 2` 和 `b - 1` 返回整数序列。 例如：

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

预期输出：

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
```

您还可以通过仅提供一个参数来使用 `range()` 函数，如下所示：

```
1. >>> for i in range(10):
2. ...         print(i)
3.
4. 0
5. 1
6. 2
7. 3
8. 4
9. 5
10. 6
11. 7
12. 8
13. 9
```

循环打印的范围是 `0` 到 `9`。

`range(a, b)` 函数具有可选的第三个参数，用于指定步长。 例如：

```
1.  for i in range(1, 20, 2):
2.      print(i)
```

预期输出：

```
1.  1
2.  3
3.  5
4.  7
5.  9
6.  11
7.  13
8.  15
9.  17
10. 19
```

`While` 循环

句法：

```
1.  while condition:
2.      # do something
```

`while` 循环在其中继续执行语句，直到条件变为假。 在每次迭代条件检查之后，如果其条件为 `True` ，则会在 `while` 循环中再次执行语句。

让我们举个例子：

```
1.  count = 0
2.
3.  while count < 10:
4.      print(count)
5.      count += 1
```

预期输出：

```
1.  0
2.  1
```

```
2
3
4
5
6
7
8
9
```

在此处，将继续打印，直到 `count` 小于 `10` 为止。

`break` 语句

`break` 语句允许突破循环。

```
count = 0

while count < 10:
    count += 1
    if count == 5:
        break
    print("inside loop", count)

print("out of while loop")
```

当 `count` 等于 `5` 时，如果条件求值为 `True` ，并且 `break` 关键字跳出循环。

预期输出：

```
inside loop 1
inside loop 2
inside loop 3
inside loop 4
out of while loop
```

`continue` 语句

当在循环中遇到 `continue` 语句时，它将结束当前迭代，并且程序控制将转到循环主体的末尾。

```
count = 0

while count < 10:
    count += 1
    if count % 2 == 0:
           continue
    print(count)
```

预期输出：

```
1
3
5
7
9
```

如您所见，当 `count % 2 == 0` 时，将执行 `continue` 语句，该语句导致当前迭代结束，并且控件继续进行下一个迭代。

在下一课中，我们将学习 Python 数学函数。

Python 数学函数

> 原文: https://thepythonguru.com/python-mathematical-function/

于 2020 年 1 月 7 日更新

Python 具有许多内置函数。

方法	描述
`round(number[, ndigits])`	四舍五入数字，也可以在第二个参数中指定精度
`pow(a, b)`	将 `a` 提升到 `b` 的幂
`abs(x)`	返回 `x` 的绝对值
`max(x1, x2, ..., xn)`	返回提供的参数中的最大值
`min(x1, x2, ..., xn)`	返回提供的参数中的最小值

下面提到的函数位于 `math` 模块中，因此您需要使用以下行首先导入 `math` 模块。

```
1.  import math
```

方法	描述
`ceil(x)`	此函数将数字四舍五入并返回其最接近的整数
`floor(x)`	此函数将向下取整并返回其最接近的整数
`sqrt(x)`	返回数字的平方根
`sin(x)`	返回 `x` 的正弦，其中 `x` 以弧度表示
`cos(x)`	返回 `x` 的余弦值，其中 `x` 为弧度
`tan(x)`	返回 `x` 的切线，其中 `x` 为弧度

让我们举一些例子来更好地理解

```
1.  >>> abs(-22)              # Returns the absolute value
2.  22
3.  >>>
4.  >>> max(9, 3, 12, 81)     # Returns the maximum number
5.  81
6.  >>>
7.  >>> min(78, 99, 12, 32)   # Returns the minimum number
8.  12
9.  >>>
```

```
>>> pow(8, 2)              # can also be written as 8 ** 2
64
>>>
>>> pow(4.1, 3.2)          # can also be written as 4.1 ** 3.2
91.39203368671122
>>>
>>> round(5.32)            # Rounds to its nearest integer
5
>>>
>>> round(3.1456875712, 3) # Return number with 3 digits after decimal point
3.146
```

```
>>> import math
>>> math.ceil(3.4123)
4
>>> math.floor(24.99231)
24
```

在下一篇文章中，我们将学习如何在 python 中生成随机数。

Python 生成随机数

> 原文: https://thepythonguru.com/python-generating-random-numbers/

于 2020 年 1 月 7 日更新

Python `random` 模块包含生成随机数的函数。 因此，首先需要使用以下行导入 `random` 模块。

```
1. import random
```

`random()` 函数

`random()` 函数返回随机数 `r` ，使得 `0 <= r < 1.0` 。

```
1. >>> import random
2. >>> for i in range(0, 10):
3. ...     print(random.random())
4. ...
```

预期输出：

```
1. 0.9240468209780505
2. 0.14200320177446257
3. 0.8647635207997064
4. 0.23926674191769448
5. 0.4436673317102027
6. 0.09211695432442013
7. 0.2512541244937194
8. 0.7279402864974873
9. 0.3864708801092763
10. 0.08450122561765672
```

`randint(a, b)` 生成 `a` 和 `b` （含）之间的随机数。

```
1. >>> import random
2. >>> for i in range(0, 10):
3. ...   print(random.randint(1, 10))
```

```
4.  ...
5.
6.  8
7.  3
8.  4
9.  7
10. 1
11. 5
12. 3
13. 7
14. 3
15. 3
```

下一章将介绍 python 中的文件处理技术。

Python 文件处理

> 原文: https://thepythonguru.com/python-file-handling/

于 2020 年 1 月 7 日更新

我们可以使用文件处理来读写文件中的数据。

打开文件

在读/写之前，您首先需要打开文件。 打开文件的语法是。

```
1.  f = open(filename, mode)
```

`open()` 函数接受两个参数 `filename` 和 `mode` 。 `filename` 是一个字符串参数，用于指定文件名及其路径，而 `mode` 也是一个字符串参数，用于指定文件的使用方式，即用于读取或写入。 `f` 是文件处理器对象，也称为文件指针。

关闭文件

读/写完文件后，您需要使用 `close()` 这样的方法关闭文件，

```
1.  f.close()  # where f is a file pointer
```

打开文件的不同模式

模式	描述
`"r"`	打开文件以只读
`"w"`	打开文件进行写入。 如果文件已经存在，则在打开之前将清除其数据。 否则将创建新文件
`"a"`	以附加模式打开文件，即将数据写入文件末尾
`"wb"`	打开文件以二进制模式写入
`"rb"`	打开文件以二进制模式读取

现在让我们看一些示例。

将数据写入文件

```
>>> f = open('myfile.txt', 'w')    # open file for writing
>>> f.write('this first line\n')   # write a line to the file
>>> f.write('this second line\n')  # write one more line to the file
>>> f.close()                      # close the file
```

注意：

`write()` 方法不会像 `print()` 函数那样自动插入新行（ `'\n'` ），需要显式添加 `'\n'` 来编写 `write()` 方法。

从文件读取数据

要从文件读回数据，您需要以下三种方法之一。

方法	描述
`read([number])`	从文件中返回指定数量的字符。 如果省略，它将读取文件的全部内容。
`readline()`	返回文件的下一行。
`readlines()`	读取所有行作为文件中的字符串列表

一次读取所有数据

```
>>> f = open('myfile.txt', 'r')
>>> f.read() # read entire content of file at once
"this first line\nthis second line\n"
>>> f.close()
```

将所有行读取为数组。

```
>>> f = open('myfile.txt', 'r')
>>> f.readlines() # read entire content of file at once
["this first line\n", "this second line\n"]
>>> f.close()
```

只读一行。

```
>>> f = open('myfile.txt', 'r')
>>> f.readline() # read the first line
"this first line\n"
>>> f.close()
```

附加数据

要附加数据，您需要以 `'a'` 模式打开文件。

```
>>> f = open('myfile.txt', 'a')
>>> f.write("this is third line\n")
19
>>> f.close()
```

使用 `for` 循环遍历数据

您可以使用文件指针遍历文件。

```
>>> f = open('myfile.txt', 'r')
>>> for line in f:
...     print(line)
...
this first line
this second line
this is third line

>>> f.close()
```

二进制读写

要执行二进制 I/O，您需要使用一个名为 `pickle` 的模块。 `pickle` 模块允许您分别使用 `load` 和 `dump` 方法读取和写入数据。

写入二进制数据

```
>> import pickle
>>> f = open('pick.dat', 'wb')
>>> pickle.dump(11, f)
>>> pickle.dump("this is a line", f)
>>> pickle.dump([1, 2, 3, 4], f)
>>> f.close()
```

读取二进制数据

```
>> import pickle
>>> f = open('pick.dat', 'rb')
>>> pickle.load(f)
11
>>> pickle.load(f)
"this is a line"
>>> pickle.load(f)
[1,2,3,4]
>>> f.close()
```

如果没有更多数据要读取，则 `pickle.load()` 会引发 `EOFError` 或文件结尾错误。

在下一课中，我们将学习 python 中的类和对象。

Python 对象和类

> 原文: https://thepythonguru.com/python-object-and-classes/

于 2020 年 1 月 7 日更新

创建对象和类

Python 是一种面向对象的语言。 在 python 中，所有东西都是对象，即 `int` ， `str` ， `bool` 甚至模块，函数也是对象。

面向对象的编程使用对象来创建程序，这些对象存储数据和行为。

定义类

python 中的类名以 `class` 关键字开头，后跟冒号（ `:` ）。 类通常包含用于存储数据的数据字段和用于定义行为的方法。 python 中的每个类还包含一个称为初始化器的特殊方法（也称为构造器），该方法在每次创建新对象时自动被调用。

让我们来看一个例子。

```
1.  class Person:
2.
3.         # constructor or initializer
4.        def __init__(self, name):
               self.name = name # name is data field also commonly known as
5.  instance variables
6.
7.        # method which returns a string
8.       def whoami(self):
9.             return "You are " + self.name
```

在这里，我们创建了一个名为 `Person` 的类，其中包含一个名为 `name` 和方法 `whoami()` 的数据字段。

什么是 `self` ？

python 中的所有方法（包括一些特殊方法，如初始化器）都具有第一个参数 `self` 。 此参数引用调用该方法的对象。 创建新对象时，会自动将 `__init__` 方法中的 `self` 参数设置为引用刚创建的对象。

从类创建对象

```
1.  p1 = Person('tom') # now we have created a new person object p1
2.  print(p1.whoami())
3.  print(p1.name)
```

预期输出：

```
1.  You are tom
2.  tom
```

注意：

当您调用一个方法时，您无需将任何内容传递给 `self` 参数，python 就会在后台自动为您完成此操作。

您也可以更改 `name` 数据字段。

```
1.  p1.name = 'jerry'
2.  print(p1.name)
```

预期输出：

```
1.  jerry
```

尽管在类之外授予对您的数据字段的访问权是一种不好的做法。 接下来，我们将讨论如何防止这种情况。

隐藏数据字段

要隐藏数据字段，您需要定义私有数据字段。 在 python 中，您可以使用两个前划线来创建私有数据字段。 您还可以使用两个下划线定义私有方法。

让我们看一个例子

```
class BankAccount:

     # constructor or initializer
    def __init__(self, name, money):
         self.__name = name
         self.__balance = money   # __balance is private now, so it is only
accessible inside the class

    def deposit(self, money):
         self.__balance += money

    def withdraw(self, money):
         if self.__balance > money :
             self.__balance -= money
             return money
         else:
             return "Insufficient funds"

    def checkbalance(self):
         return self.__balance

b1 = BankAccount('tim', 400)
print(b1.withdraw(500))
b1.deposit(500)
print(b1.checkbalance())
print(b1.withdraw(800))
print(b1.checkbalance())
```

预期输出：

```
Insufficient funds
900
800
100
```

让我们尝试访问类外部的 `__balance` 数据字段。

```
print(b1.__balance)
```

预期输出：

```
AttributeError: 'BankAccount' object has no attribute '__balance'
```

如您所见，现在在类外部无法访问 `__balance` 字段。

在下一章中，我们将学习运算符重载。

Python 运算符重载

> 原文: https://thepythonguru.com/python-operator-overloading/

于 2020 年 1 月 7 日更新

您已经看到可以使用 `+` 运算符添加数字，并同时连接字符串。 这是可能的，因为 `int` 类和 `str` 类都重载了 `+` 运算符。 运算符实际上是在各个类中定义的方法。 运算符的定义方法称为运算符重载。 例如：要对自定义对象使用 `+` 运算符，您需要定义一个名为 `__add__` 的方法。

让我们举个例子来更好地理解

```
1.  import math
2.
3.  class Circle:
4.
5.      def __init__(self, radius):
6.          self.__radius = radius
7.
8.      def setRadius(self, radius):
9.          self.__radius = radius
10.
11.     def getRadius(self):
12.         return self.__radius
13.
14.     def area(self):
15.         return math.pi * self.__radius ** 2
16.
17.     def __add__(self, another_circle):
18.         return Circle( self.__radius + another_circle.__radius )
19.
20. c1 = Circle(4)
21. print(c1.getRadius())
22.
23. c2 = Circle(5)
24. print(c2.getRadius())
25.
    c3 = c1 + c2 # This became possible because we have overloaded + operator by
26. adding a    method named __add__
27. print(c3.getRadius())
```

预期输出：

```
4
5
9
```

```
import math

class Circle:

    def __init__(self, radius):
        self.__radius = radius

    def setRadius(self, radius):
        self.__radius = radius

    def getRadius(self):
        return self.__radius

    def area(self):
        return math.pi * self.__radius ** 2

    def __add__(self, another_circle):
        return Circle( self.__radius + another_circle.__radius )

c1 = Circle(4)
print(c1.getRadius())

c2 = Circle(5)
print(c2.getRadius())

c3 = c1 + c2 # This became possible because we have overloaded + operator by adding a    method named __add__
print(c3.getRadius())
```

在上面的示例中，我们添加了 `__add__()` 方法，该方法允许使用 `+` 运算符添加两个圆形对象。在 `__add__()` 方法内部，我们正在创建一个新对象并将其返回给调用者。

Python 还有许多其他特殊方法，例如 `__add__()` ，请参见下面的列表。

运算符	函数	方法说明

`+`	`__add__(self, other)`	加法
`*`	`__mul__(self, other)`	乘法
`-`	`__sub__(self, other)`	减法
`%`	`__mod__(self, other)`	余数
`/`	`__truediv__(self, other)`	除法
`<`	`__lt__(self, other)`	小于
`<=`	`__le__(self, other)`	小于或等于
`==`	`__eq__(self, other)`	等于
`!=`	`__ne__(self, other)`	不等于
`>`	`__gt__(self, other)`	大于

`>=` ， `__ge__(self, other)` ，大于或等于 `[index]` ， `__getitem__(self, index)` ，索引运算符 `in` ， `__contains__(self, value)` ，检查成员资格 `len` ， `__len__(self)` ，元素数 `str` ， `__str__(self)` 的字符串表示形式

下面的程序使用上面提到的一些函数来重载运算符。

```
1.  import math
2.
3.  class Circle:
4.
5.      def __init__(self, radius):
6.          self.__radius = radius
7.
8.      def setRadius(self, radius):
9.          self.__radius = radius
10.
11.     def getRadius(self):
12.         return self.__radius
13.
14.     def area(self):
15.         return math.pi * self.__radius ** 2
16.
17.     def __add__(self, another_circle):
18.         return Circle( self.__radius + another_circle.__radius )
19.
20.     def __gt__(self, another_circle):
21.         return self.__radius > another_circle.__radius
22.
23.     def __lt__(self, another_circle):
24.         return self.__radius < another_circle.__radius
```

```
25.
26.     def __str__(self):
27.         return "Circle with radius " + str(self.__radius)
28.
29. c1 = Circle(4)
30. print(c1.getRadius())
31.
32. c2 = Circle(5)
33. print(c2.getRadius())
34.
35. c3 = c1 + c2
36. print(c3.getRadius())
37.
38. print( c3 > c2) # Became possible because we have added __gt__ method
39.
40. print( c1 < c2) # Became possible because we have added __lt__ method
41.
42. print(c3) # Became possible because we have added __str__ method
```

预期输出：

```
1. 4
2. 5
3. 9
4. True
5. True
6. Circle with radius 9
```

```
1. import math
2.
3. class Circle:
4.
5.     def __init__(self, radius):
6.         self.__radius = radius
7.
8.     def setRadius(self, radius):
9.         self.__radius = radius
10.
11.     def getRadius(self):
12.         return self.__radius
13.
14.     def area(self):
```

```
        return math.pi * self.__radius ** 2

    def __add__(self, another_circle):
        return Circle( self.__radius + another_circle.__radius )

    def __gt__(self, another_circle):
        return self.__radius > another_circle.__radius

    def __lt__(self, another_circle):
        return self.__radius < another_circle.__radius

    def __str__(self):
        return "Circle with radius " + str(self.__radius)

c1 = Circle(4)
print(c1.getRadius())

c2 = Circle(5)
print(c2.getRadius())

c3 = c1 + c2
print(c3.getRadius())

print( c3 > c2) # Became possible because we have added __gt__ method

print( c1 < c2) # Became possible because we have added __lt__ method

print(c3) # Became possible because we have added __str__ method
```

下一课是继承和多态。

Python 继承与多态

> 原文: https://thepythonguru.com/python-inheritance-and-polymorphism/

于 2020 年 1 月 7 日更新

继承允许程序员首先创建一个通用类，然后再将其扩展为更专业的类。 它还允许程序员编写更好的代码。

使用继承，您可以继承所有访问数据字段和方法，还可以添加自己的方法和字段，因此继承提供了一种组织代码的方法，而不是从头开始重写代码。

在面向对象的术语中，当 `X` 类扩展了 `Y` 类时，则 `Y` 被称为超类或基类，而 `X` 被称为子类或派生类。 还有一点要注意，子类只能访问非私有的数据字段和方法，私有数据字段和方法只能在该类内部访问。

创建子类的语法是：

```
class SubClass(SuperClass):
  # data fields
  # instance methods
```

让我们以一个例子来说明这一点。

```
class Vehicle:

    def __init__(self, name, color):
        self.__name = name      # __name is private to Vehicle class
        self.__color = color

    def getColor(self):         # getColor() function is accessible to class
Car
        return self.__color

    def setColor(self, color):  # setColor is accessible outside the class
        self.__color = color

    def getName(self):          # getName() is accessible outside the class
        return self.__name

```

```
class Car(Vehicle):

    def __init__(self, name, color, model):
        # call parent constructor to set name and color
        super().__init__(name, color)
        self.__model = model

    def getDescription(self):
        return self.getName() + self.__model + " in " + self.getColor() + "
color"

# in method getDescrition we are able to call getName(), getColor() because
they are
# accessible to child class through inheritance

c = Car("Ford Mustang", "red", "GT350")
print(c.getDescription())
print(c.getName()) # car has no method getName() but it is accessible through
class Vehicle
```

预期输出：

```
Ford MustangGT350 in red color
Ford Mustang
```

在这里，我们创建了基类 `Vehicle` 及其子类 `Car` 。 注意，我们没有在 `Car` 类中定义 `getName()` ，但是我们仍然可以访问它，因为类 `Car` 从 `Vehicle` 类继承。 在上面的代码中， `super()` 方法用于调用基类的方法。 这是 `super()` 的工作方式

假设您需要在子类的基类中调用名为 `get_information()` 的方法，则可以使用以下代码进行调用。

```
super().get_information()
```

同样，您可以使用以下代码从子类构造器中调用基类构造器。

```
super().__init__()
```

多重继承

与 Java 和 C# 等语言不同，python 允许多重继承，即您可以同时从多个类继承，

```
class Subclass(SuperClass1, SuperClass2, ...):
   # initializer
   # methods
```

让我们举个例子：

```
class MySuperClass1():

    def method_super1(self):
        print("method_super1 method called")

class MySuperClass2():

    def method_super2(self):
        print("method_super2 method called")

class ChildClass(MySuperClass1, MySuperClass2):

    def child_method(self):
        print("child method")

c = ChildClass()
c.method_super1()
c.method_super2()
```

预期输出：

```
method_super1 method called
method_super2 method called
```

如您所见，因为 `ChildClass` 继承了 `MySuperClass1` ， `MySuperClass2` ，所以 `ChildClass` 的对象现在可以访问 `method_super1()` 和 `method_super2()` 。

覆盖方法

要覆盖基类中的方法，子类需要定义一个具有相同签名的方法。 （即与基类中的方法相同的方法名称和相同数量的参数）。

```
class A():

```

```
    def __init__(self):
        self.__x = 1

    def m1(self):
        print("m1 from A")

class B(A):

    def __init__(self):
        self.__y = 1

    def m1(self):
        print("m1 from B")

c = B()
c.m1()
```

预期输出：

```
m1 from B
```

在这里，我们从基类中重写 `m1()` 方法。 尝试在 `B` 类中注释 `m1()` 方法，现在将运行 `Base` 类中的 `m1()` 方法，即 `A` 类。

预期输出：

```
m1 from A
```

`isinstance()` 函数

`isinstance()` 函数用于确定对象是否为该类的实例。

语法： `isinstance(object, class_type)`

```
>>> isinstance(1, int)
True

>>> isinstance(1.2, int)
False

>>> isinstance([1,2,3,4], list)
```

```
8.  True
```

下一章异常处理。

Python 异常处理

> 原文: https://thepythonguru.com/python-exception-handling/

于 2020 年 1 月 7 日更新

异常处理使您能够优雅地处理错误并对其进行有意义的处理。 如果未找到所需文件，则向用户显示一条消息。 Python 使用 `try` 和 `except` 块处理异常。

语法：

```
try:
    # write some code
    # that might throw exception
except <ExceptionType>:
    # Exception handler, alert the user
```

如您在 `try` 块中看到的那样，您需要编写可能引发异常的代码。 当发生异常时，将跳过 `try` 块中的代码。 如果 `except` 子句中存在匹配的异常类型，则执行其处理器。

让我们举个例子：

```
try:
    f = open('somefile.txt', 'r')
    print(f.read())
    f.close()
except IOError:
    print('file not found')
```

上面的代码如下：

1. 执行 `try` 和 `except` 块之间的第一条语句。
2. 如果没有异常发生，则将跳过 `except` 子句下的代码。
3. 如果文件不存在，则会引发异常，并且 `try` 块中的其余代码将被跳过
4. 发生异常时，如果异常类型与 `except` 关键字后的异常名称匹配，则将执行该 `except` 子句中的代码。

注意：

上面的代码仅能处理 `IOError` 异常。 要处理其他类型的异常，您需要添加更多的 `except` 子句。

`try` 语句可以具有多个 `except` 子句，也可以具有可选的 `else` 和/或 `finally` 语句。

```
try:
    <body>
except <ExceptionType1>:
    <handler1>
except <ExceptionTypeN>:
    <handlerN>
except:
    <handlerExcept>
else:
    <process_else>
finally:
    <process_finally>
```

`except` 子句类似于 `elif` 。 发生异常时，将检查该异常以匹配 `except` 子句中的异常类型。 如果找到匹配项，则执行匹配大小写的处理器。 另请注意，在最后的 `except` 子句中， `ExceptionType` 被省略。 如果异常不匹配最后一个 `except` 子句之前的任何异常类型，则执行最后一个 `except` 子句的处理器。

注意：

`else` 子句下的语句仅在没有引发异常时运行。

注意：

无论是否发生异常， `finally` 子句中的语句都将运行。

现在举个例子。

```
try:
    num1, num2 = eval(input("Enter two numbers, separated by a comma : "))
    result = num1 / num2
    print("Result is", result)

except ZeroDivisionError:
    print("Division by zero is error !!")

except SyntaxError:
    print("Comma is missing. Enter numbers separated by comma like this 1, 2")

except:
    print("Wrong input")
```

```
14.
15. else:
16.     print("No exceptions")
17.
18. finally:
19.     print("This will execute no matter what")
```

注意：

`eval()` 函数允许 python 程序在其内部运行 python 代码， `eval()` 需要一个字符串参数。

要了解有关 `eval()` 的更多信息，请访问 Python 中的 `eval()` 。

引发异常

要从您自己的方法引发异常，您需要像这样使用 `raise` 关键字

```
1. raise ExceptionClass("Your argument")
```

让我们举个例子

```
1. def enterage(age):
2.     if age < 0:
3.         raise ValueError("Only positive integers are allowed")
4.
5.     if age % 2 == 0:
6.         print("age is even")
7.     else:
8.         print("age is odd")
9.
10. try:
11.     num = int(input("Enter your age: "))
12.     enterage(num)
13.
14. except ValueError:
15.     print("Only positive integers are allowed")
16. except:
17.     print("something is wrong")
```

运行程序并输入正整数。

预期输出：

```
Enter your age: 12
age is even
```

再次运行该程序并输入一个负数。

预期输出：

```
Enter your age: -12
Only integers are allowed
```

使用异常对象

现在您知道如何处理异常，在本节中，我们将学习如何在异常处理器代码中访问异常对象。 您可以使用以下代码将异常对象分配给变量。

```
try:
    # this code is expected to throw exception
except ExceptionType as ex:
    # code to handle exception
```

如您所见，您可以将异常对象存储在变量 `ex` 中。 现在，您可以在异常处理器代码中使用此对象。

```
try:
    number = eval(input("Enter a number: "))
    print("The number entered is", number)
except NameError as ex:
    print("Exception:", ex)
```

运行程序并输入一个数字。

预期输出：

```
Enter a number: 34
The number entered is 34
```

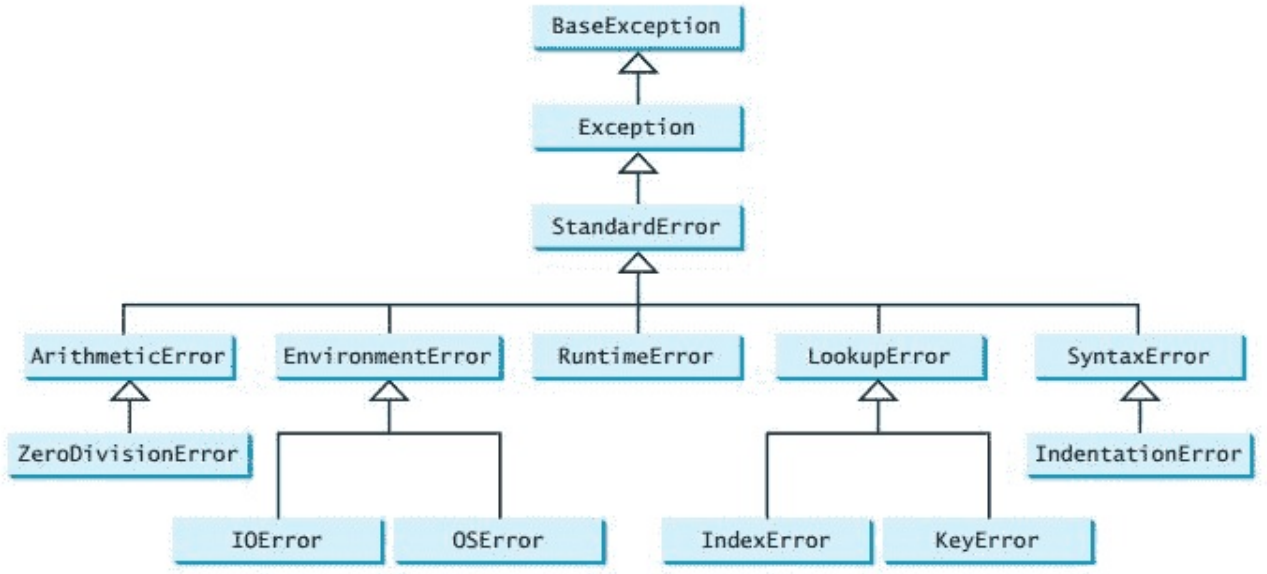
再次运行程序并输入一个字符串。

预期输出：

```
Enter a number: one
Exception: name 'one' is not defined
```

创建自定义异常类

您可以通过扩展 `BaseException` 类或 `BaseException` 的子类来创建自定义异常类。



如您所见，python 中的大多数异常类都是从 `BaseException` 类扩展而来的。 您可以从 `BaseException` 类或 `BaseException` 的子类（例如 `RuntimeError` ）派生自己的异常类。

创建一个名为 `NegativeAgeException.py` 的新文件，并编写以下代码。

```
class NegativeAgeException(RuntimeError):
    def __init__(self, age):
        super().__init__()
        self.age = age
```

上面的代码创建了一个名为 `NegativeAgeException` 的新异常类，该异常类仅由使用 `super().__init__()` 调用父类构造器并设置 `age` 的构造器组成。

使用自定义异常类

```
def enterage(age):
    if age < 0:
        raise NegativeAgeException("Only positive integers are allowed")
```

```
4.
5.      if age % 2 == 0:
6.          print("age is even")
7.      else:
8.          print("age is odd")
9.
10. try:
11.     num = int(input("Enter your age: "))
12.     enterage(num)
13. except NegativeAgeException:
14.     print("Only positive integers are allowed")
15. except:
16.     print("something is wrong")
```

在下一篇文章中，我们将学习 Python 模块。

Python 模块

> 原文: https://thepythonguru.com/python-modules/

于 2020 年 1 月 7 日更新

Python 模块是一个普通的 python 文件，可以存储函数，变量，类，常量等。模块帮助我们组织相关代码。 例如，python 中的 `math` 模块具有与数学相关的函数。

创建模块

创建一个名为 `mymodule.py` 的新文件并编写以下代码。

```
1.  foo = 100
2.
3.  def hello():
4.      print("i am from mymodule.py")
```

如您所见，我们在模块中定义了全局变量 `foo` 和函数 `hello()` 。 现在要在程序中使用此模块，我们首先需要使用 `import` 语句将其导入

```
1.  import mymodule
```

现在您可以使用以下代码在 `mymodule.py` 中使用变量和调用函数。

```
1.  import mymodule
2.
3.  print(mymodule.foo)
4.  print(mymodule.hello())
```

预期输出：

```
1.  100
2.  i am from mymodule.py
```

请记住，您需要先指定模块名称才能访问其变量和函数，否则将导致错误。

结合使用 `from` 和 `import`

使用 `import` 语句会导入模块中的所有内容，如果只想访问特定的函数或变量该怎么办？ 这是 `from` 语句的来源，这里是如何使用它。

```
    from mymodule import foo # this statement import only foo variable from
1.  mymodule
2.  print(foo)
```

预期输出：

```
1.  100
```

注意：

在这种情况下，您无需指定模块名称即可访问变量和函数。

`dir()` 方法

`dir()` 是一种内置方法，用于查找对象的所有属性（即所有可用的类，函数，变量和常量）。 正如我们已经在 python 中讨论的所有对象一样，我们可以使用 `dir()` 方法来查找模块的属性，如下所示：

```
1.  dir(module_name)
```

`dir()` 返回包含可用属性名称的字符串列表。

```
1.  >>> dir(mymodule)
2.  ['__builtins__', '__cached__', '__doc__', '__file__',
3.  '__loader__', '__name__', '__package__', '__spec__', 'foo', 'hello']
```

如您所见，除了 `foo` 和 `hello` 之外， `mymodule` 中还有其他属性。 这些是 python 自动提供给所有模块的内置属性。

恭喜您已经完成了掌握 Python 所需的所有构建基块！！

高级 Python

Python `*args` 和 `**kwargs`

> 原文: https://thepythonguru.com/python-args-and-kwargs/

于 2020 年 1 月 7 日更新

什么是 `*args` ？

`*args` 允许我们将可变数量的参数传递给函数。 让我们以一个例子来阐明这一点。

假设您创建了一个将两个数字相加的函数。

```
1.  def sum(a, b):
2.      print("sum is", a+b)
```

如您所见，该程序仅接受两个数字，如果您要传递两个以上的参数，这就是 `*args` 起作用的地方。

```
1.  def sum(*args):
2.      s = 0
3.      for i in args:
4.          s += i
5.      print("sum is", s)
```

现在，您可以像这样将任意数量的参数传递给函数，

```
1.  >>> sum(1, 2, 3)
2.  6
3.  >>> sum(1, 2, 3, 4, 5, 7)
4.  22
5.  >>> sum(1, 2, 3, 4, 5, 7, 8, 9, 10)
6.  49
7.  >>> sum()
8.  0
```

注意：

`*args` 的名称只是一个约定，您可以使用任何有效标识符。 例如 `*myargs` 是完全有效的。

什么是 `**kwargs` ？

`**kwargs` 允许我们传递可变数量的关键字参数，例如 `func_name(name='tim', team='school')`

```
def my_func(**kwargs):
    for i, j in kwargs.items():
        print(i, j)

my_func(name='tim', sport='football', roll=19)
```

预期输出：

```
sport football
roll 19
name tim
```

在函数调用中使用 `*args` 和 `**kwargs`

您可以使用 `*args` 将可迭代变量中的元素传递给函数。 以下示例将清除所有内容。

```
def my_three(a, b, c):
    print(a, b, c)

a = [1,2,3]
my_three(*a) # here list is broken into three elements
```

注意：

仅当参数数量与可迭代变量中的元素数量相同时，此方法才有效。

同样，您可以使用 `**kwargs` 来调用如下函数：

```
def my_three(a, b, c):
    print(a, b, c)

a = {'a': "one", 'b': "two", 'c': "three" }
my_three(**a)
```

请注意，要使此工作有效，需要做两件事：

1. 函数中的参数名称必须与字典中的键名称匹配。
2. 参数的数量应与字典中的键的数量相同。

Python 生成器

> 原文: https://thepythonguru.com/python-generators/

于 2020 年 1 月 7 日更新

生成器是用于创建迭代器的函数，因此可以在 `for` 循环中使用它。

创建生成器

生成器的定义类似于函数，但只有一个区别，我们使用 `yield` 关键字返回用于 `for` 循环的每次迭代的值。 让我们看一个示例，其中我们试图克隆 python 的内置 `range()` 函数。

```
1.  def my_range(start, stop, step = 1):
2.      if stop <= start:
3.          raise RuntimeError("start must be smaller than stop")
4.      i = start
5.      while i < stop:
6.          yield i
7.          i += step
8.
9.  try:
10.     for k in my_range(10, 50, 3):
11.         print(k)
12. except RuntimeError as ex:
13.     print(ex)
14. except:
15.     print("Unknown error occurred")
```

预期输出：

```
1.  10
2.  13
3.  16
4.  19
5.  22
6.  25
7.  28
```

```
31
34
37
40
43
46
49
```

```python
def my_range(start, stop, step = 1):
    if stop <= start:
        raise RuntimeError("start must be smaller than stop")
    i = start
    while i < stop:
        yield i
        i += step

try:
    for k in my_range(10, 50, 3):
        print(k)
except RuntimeError as ex:
    print(ex)
except:
    print("Unknown error occurred")
```

`my_range()` 的工作方式如下：

在 `for` 循环中，调用 `my_range()` 函数，它将初始化三个参数（ `start` ， `stop` 和 `step` ）的值，并检查 `stop` 是否小于或等于 `start` 。 `i` 被分配了 `start` 的值。 此时， `i` 为 `10` ，因此 `while` 条件的值为 `True` ，而 `while` 循环开始执行。 在下一个语句 `yield` 中，将控制转移到 `for` 循环，并将 `i` 的当前值分配给变量 `k` ，在 `for` 循环打印语句中执行该语句，然后该控件再次传递到函数 `my_range()` 内的第 7 行 `i` 递增。 此过程一直重复进行，直到 `i < stop` 为止。

Python 正则表达式

> 原文: https://thepythonguru.com/python-regular-expression/

于 2020 年 1 月 7 日更新

正则表达式广泛用于模式匹配。 Python 具有对常规功能的内置支持。 要使用正则表达式，您需要导入 `re` 模块。

```
1. import re
```

现在您可以使用正则表达式了。

`search` 方法

`re.search()` 用于查找字符串中模式的第一个匹配项。

语法： `re.search(pattern, string, flags[optional])`

`re.search()` 方法接受模式和字符串，并在成功时返回 `match` 对象；如果找不到匹配项，则返回 `None` 。 `match` 对象具有 `group()` 方法，该方法在字符串中包含匹配的文本。

您必须使用原始字符串来指定模式，即像这样用 `r` 开头的字符串。

```
1. r'this \n'
```

所有特殊字符和转义序列在原始字符串中均失去其特殊含义，因此 `\n` 不是换行符，它只是一个反斜杠 `\` 后跟一个 `n` 。

```
1. >>> import re
2. >>> s = "my number is 123"
3. >>> match = re.search(r'\d\d\d', s)
4. >>> match
5. <_sre.SRE_Match object; span=(13, 16), match='123'>
6. >>> match.group()
7. '123'
```

上面我们使用 `\d\d\d` 作为模式。 `\d` 正则表达式匹配一位数字，因此

`\d\d\d` 将匹配 `111` ， `222` 和 `786` 之类的数字。 它与 `12` 和 `1444` 不匹配。

正则表达式中使用的基本模式

符号	描述
`.`	点匹配除换行符以外的任何字符
`\w`	匹配任何单词字符，即字母，字母数字，数字和下划线（ `_` ）
`\W`	匹配非单词字符
`\d`	匹配一个数字
`\D`	匹配不是数字的单个字符
`\s`	匹配任何空白字符，例如 `\n` ， `\t` ，空格
`\S`	匹配单个非空白字符
`[abc]`	匹配集合中的单个字符，即匹配 `a` ， `b` 或 `c`
`[^abc]`	匹配 `a` ， `b` 和 `c` 以外的单个字符
`[a-z]`	匹配 `a` 至 `z` 范围内的单个字符。
`[a-zA-Z]`	匹配 `a-z` 或 `A-Z` 范围内的单个字符
`[0-9]`	匹配 `0` - `9` 范围内的单个字符
`^`	匹配从字符串开头开始
`$`	匹配从字符串末尾开始
`+`	匹配一个或多个前面的字符（贪婪匹配）。
`*`	匹配零个或多个前一个字符（贪婪匹配）。

再举一个例子：

```
1.  import re
2.  s = "tim email is tim@somehost.com"
3.  match = re.search(r'[\w.-]+@[\w.-]+', s)
4.
5.  # the above regular expression will match a email address
6.
7.  if match:
8.      print(match.group())
9.  else:
10.     print("match not found")
```

这里我们使用了 `[\w.-]+@[\w.-]+` 模式来匹配电子邮件地址。 成功后， `re.search()` 返回一个 `match` 对象，其 `group()` 方法将包含匹配的文本。

捕捉组

组捕获允许从匹配的字符串中提取部分。 您可以使用括号 `()` 创建组。 假设在上面的示例中，我们想从电子邮件地址中提取用户名和主机名。 为此，我们需要在用户名和主机名周围添加 `()` ，如下所示。

```
match = re.search(r'([\w.-]+)@([\w.-]+)', s)
```

请注意，括号不会更改模式匹配的内容。 如果匹配成功，则 `match.group(1)` 将包含第一个括号中的匹配， `match.group(2)` 将包含第二个括号中的匹配。

```
import re
s = "tim email is tim@somehost.com"
match = re.search('([\w.-]+)@([\w.-]+)', s)
if match:
    print(match.group()) ## tim@somehost.com (the whole match)
    print(match.group(1)) ## tim (the username, group 1)
    print(match.group(2)) ## somehost (the host, group 2)
```

`findall()` 函数

如您所知，现在 `re.search()` 仅找到模式的第一个匹配项，如果我们想找到字符串中的所有匹配项，这就是 `findall()` 发挥作用的地方。

语法： `findall(pattern, string, flags=0[optional])`

成功时，它将所有匹配项作为字符串列表返回，否则返回空列表。

```
import re
s = "Tim's phone numbers are 12345-41521 and 78963-85214"
match = re.findall(r'\d{5}', s)

if match:
    print(match)
```

预期输出：

```
['12345', '41521', '78963', '85214']
```

您还可以通过 `findall()` 使用组捕获，当应用组捕获时， `findall()` 返回一个元组列表，其中元组将包含匹配的组。 一个示例将清除所有内容。

```
import re
s = "Tim's phone numbers are 12345-41521 and 78963-85214"
match = re.findall(r'(\d{5})-(\d{5})', s)
print(match)

for i in match:
    print()
    print(i)
    print("First group", i[0])
    print("Second group", i[1])
```

预期输出：

```
[('12345', '41521'), ('78963', '85214')]

('12345', '41521')
First group 12345
Second group 41521

('78963', '85214')
First group 78963
Second group 85214
```

可选标志

`re.search()` 和 `re.findall()` 都接受可选参数称为标志。 标志用于修改模式匹配的行为。

标志	描述
`re.IGNORECASE`	忽略大写和小写
`re.DOTALL`	允许（ `.` ）匹配换行符，默认（ `.` ）匹配除换行符之外的任何字符
`re.MULTILINE`	这将允许 `^` 和 `$` 匹配每行的开始和结束

使用 `re.match()`

`re.match()` 与 `re.search()` 非常相似，区别在于它将在字符串的开头开始寻找匹配项。

```
1. import re
2. s = "python tuts"
3. match = re.match(r'py', s)
4. if match:
5.     print(match.group())
```

您可以通过使用 `re.search()` 将 `^` 应用于模式来完成同一件事。

```
1. import re
2. s = "python tuts"
3. match = re.search(r'^py', s)
4. if match:
5.     print(match.group())
```

这样就完成了您需要了解的有关 `re` 模块的所有内容。

使用 PIP 在 python 中安装包

> 原文: https://thepythonguru.com/installing-packages-in-python-using-pip

于 2020 年 1 月 7 日更新

PIP 是一个包管理系统，用于从存储库安装包。 您可以使用 `pip` 安装 http://pypi.python.org/pypi 上可用的各种包。 PIP 与 php 中的作曲家非常相似。 PIP 是递归的首字母缩写，代表 PIP 安装包。

安装 PIP

Python 2.7.9 及更高版本（python2 系列）和 Python 3.4 及更高版本（python 3 系列）已经带有 pip。

要检查您的 python 版本，您需要输入以下命令：

```
1.  python  -V
```

如果您的 python 版本不属于上述任何版本，则需要手动安装 `pip` （请参见下面的链接）。

- 单击此处以获取 Windows 安装说明。
- 单击此处以获取 Linux 指南。

安装包

假设您要安装一个名为 Requests 的包（用于发出 HTTP 请求）。 您需要发出以下命令。

```
1.  pip install requests # this will install latest request package
    pip install requests==2.6.0 # this will install requests 2.6.0 package not the
2.  latest package
    pip install requests>=2.6.0 # specify a minimum version if it's not available
3.  pip will install the latest version
```

注意：

`pip.exe` 存储在 `C:\Python34\Scripts` 下，因此您需要去那里安装包。 或者，将整个路径添加

到 `PATH` 环境变量。 这样，您可以从任何目录访问 `pip` 。

卸载包

要卸载包，请使用以下命令。

```
1.  pip uninstall package_name
```

升级包

```
1.  pip install --upgrade package_name
```

搜索包

```
1.  pip search "your query"
```

注意：

您无需在搜索字词前后添加引号。

列出已安装的包

```
1.  pip list
```

上面的命令将列出所有已安装的包。

列出过时的已安装包

```
1.  pip list --outdated
```

有关已安装包的详细信息

您可以使用以下命令来获取有关已安装包的信息，即包名称，版本，位置，依赖项。

```
1.  pip show package_name
```

Python `virtualenv` 指南

> 原文: https://thepythonguru.com/python-virtualenv-guide/

于 2020 年 1 月 7 日更新

注意:

本教程需要 `pip` ，如果您尚未这样做，请首先通过安装 `pip` 。

`virtualenv` 是用于分隔项目所需的不同依赖项的工具。 在处理多个项目时，一个项目需要一个与另一个项目完全不同的包版本是一个常见的问题， `virtualenv` 可帮助我们解决此类问题。 它还有助于防止污染全局站点包。

安装 `virtualenv`

`virtualenv` 只是 `pypi` 提供的包，您可以使用 `pip` 安装 `virtualenv` 。

```
1.  pip install virtualenv
```

安装后，您可能需要将 `C:\Python34\Scripts` 添加到 `PATH` 环境变量中。 这样，诸如 `pip` ， `virtualenv` 之类的命令将在任何目录级别可用。

创建虚拟环境

创建一个名为 `python_project` 的新目录，并将当前工作目录更改为 `python_project` 。

```
1.  mkdir python_project
2.  cd python_project
```

要在 `python_project` 中创建虚拟环境，您需要发出以下命令。

```
1.  virtualenv my_env
```

这将在 `python_project` 内创建一个新文件夹 `my_env` 。 此文件夹将包含用于安装包的 python 可执行文件和 `pip` 库的副本。 在这里，我们使用 `my_env` 作为名称，但是您可以使用任何您想要的名称。 现在您可以使用虚拟环境了，您只需要激活它即可。

在本教程中，我们有一点要使用 python 3.4 安装了 `virtualenv` ，假设您也有 python 2.7 并想创建一个使用 python 2.7 而不是 3.4 的虚拟环境，则可以使用以下命令进行操作。

```
1.  virtualenv -p c:\Python27/python.exe my_env
```

激活虚拟环境

如果您在 Windows 上，则需要执行以下命令。

```
1.  my_env\Scripts\activate.bat
```

在 Linux 上，请输入。

```
1.  source my_env/bin/activate
```

发出上述命令后，您的命令提示符字符串将发生变化，看起来像这样，

```
1.  ( my_env ) Path_to_the_project: $
```

注意 `( my_env )` ，这表明您现在正在虚拟环境下运行。

现在您的虚拟环境已激活。 您在此处安装的所有内容仅会被该项目使用。

让我们尝试安装请求包。

在 Windows 中，输入以下代码。

```
1.  my_env\Scripts\pip.exe install requests
```

您不能在 Windows 中仅使用 `pip` 安装请求，因为如果将 `C:\Python34\Scripts` 添加到 `PATH` 环境变量中，它将执行全局 `pip` 。 如果尚未添加，则会出现错误。

同样，在 Linux 中，您需要执行以下代码

```
1.  my_env\Scripts\pip install requests
```

停用虚拟环境

要停用虚拟环境，您需要使用以下命令。

```
1.  deactivate
```

此命令将使您返回系统的默认 python 解释器，我们可以在其中将包安装在全局站点包中。

您现在应该能够看到使用 `virtualenv` 的动机。 它可以帮助我们组织项目的需求而不会相互冲突。

Python 递归函数

> 原文: https://thepythonguru.com/python-recursive-functions/

于 2020 年 1 月 7 日更新

函数本身调用时称为递归。 递归的工作原理类似于循环，但有时使用递归比循环更有意义。 您可以将任何循环转换为递归。

这是递归的工作方式。 递归函数会自行调用。 如您所料，如果不因某种条件而停止，则此过程将无限期重复。 此条件称为基本条件。 每个递归程序中都必须有一个基本条件，否则它将像无限循环一样永远继续执行。

递归函数的工作原理概述：

1. 递归函数由一些外部代码调用。
2. 如果满足基本条件，则程序执行有意义的操作并退出。
3. 否则，函数将执行一些必需的处理，然后调用自身以继续递归。 这是用于计算阶乘的递归函数的示例。

阶乘由数字表示，后跟（ `!` ）符号，即 `4!` 。

例如：

```
4! = 4 * 3 * 2 * 1
2! = 2 * 1
0! = 1
```

这是一个例子

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

print(fact(0))
print(fact(5))
```

预期输出：

```
1
120
```

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

print(fact(0))
print(fact(5))
```

现在尝试执行上述函数：

```
print(fact(2000))
```

你会得到：

```
RuntimeError: maximum recursion depth exceeded in comparison
```

发生这种情况是因为默认情况下 python 在 `1000` 调用之后停止了调用递归函数。 若要更改此行为，您需要按如下所示修改代码。

```
import sys
sys.setrecursionlimit(3000)

def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

print(fact(2000))
```

```
import sys
sys.setrecursionlimit(3000)

def fact(n):
    if n == 0:
        return 1
```

```
    else:
        return n * fact(n-1)

print(fact(2000))
```

`__name__ == "__main__"` 是什么？

> 原文: https://thepythonguru.com/what-is-if-name-main/

于 2020 年 1 月 7 日更新

Python 中的每个模块都有一个称为 `__name__` 的特殊属性。 当模块作为主程序运行时， `__name__` 属性的值设置为 `'__main__'` 。 否则，将 `__name__` 的值设置为包含模块的名称。

考虑以下代码，以更好地理解。

```
1.  # file my_module.py
2.
3.  foo = 100
4.
5.  def hello():
6.      print("i am from my_module.py")
7.
8.  if __name__ == "__main__":
9.      print("Executing as main program")
10.     print("Value of __name__ is: ", __name__)
11.     hello()
```

在这里，我们定义了一个新模块 `my_module` 。 通过输入以下代码，我们可以将该模块作为主程序执行：

```
1.  python my_module.py
```

预期输出：

```
1.  Executing as main program
2.  Value of __name__ is: __main__
3.  i am from my_module.py
```

```
1.  # file my_module.py
2.
3.  foo = 100
4.
```

```
def hello():
    print("i am from my_module.py")

if __name__ == "__main__":
    print("Executing as main program")
    print("Value of __name__ is: ", __name__)
    hello()
```

在这里，我们正在创建一个新模块并将其作为主程序执行，因此 `__name__` 的值设置为 `'__main__'` 。 结果，如果条件满足，则函数 `hello()` 被调用。

现在创建一个名为 `module.py` 的新文件，并编写以下代码：

```
import my_module

print(my_module.foo)
my_module.hello()

print(my_module.__name__)
```

预期输出：

```
100
i am from my_module.py
my_module
```

如您现在所见，由于 `__name__` 的值设置为 `'my_module'` ，因此 `my_module` 中的 `if` 语句执行失败。

Python Lambda 函数

> 原文: https://thepythonguru.com/python-lambda-function/

于 2020 年 1 月 7 日更新

Python 允许您使用称为 lambda 函数的工具来创建匿名函数，即没有名称的函数。

Lambda 函数是小的函数，通常不超过一行。 就像普通函数一样，它可以具有任意数量的参数。 lambda 函数的主体非常小，仅包含一个表达式。 表达式的结果是将 lambda 应用于参数时的值。 另外，lambda 函数中无需任何 `return` 语句。

让我们举个例子:

考虑一个函数 `multiply()` :

```
1.  def multiply(x, y):
2.      return x * y
```

此函数太小，因此让我们将其转换为 lambda 函数。

要创建 lambda 函数，请首先编写关键字 `lambda` ，然后是多个以逗号分隔的参数（ `,` ），然后是冒号 a（ `:` ），然后是单行表达式。

```
1.  r = lambda x, y: x * y
2.  r(12, 3)   # call the lambda function
```

预期输出:

```
1.  36
```

```
1.  r = lambda x, y: x * y
2.  print(r(12, 3))   # call the lambda function
```

在这里，我们使用两个参数 `x` 和 `y` ，冒号后面的表达式是 lambda 函数的主体。 如您所见，lambda 函数没有名称，并通过分配给它的变量进行调用。

您无需将 lambda 函数分配给变量。

```
(lambda x, y: x * y)(3,4)
```

预期输出：

```
12
```

```
print( (lambda x, y: x * y)(3,4) )
```

请注意，lambda 函数不能包含多个表达式。

Python 字符串格式化

> 原文: https://thepythonguru.com/python-string-formatting

于 2020 年 1 月 7 日更新

`format()` 方法允许您以任何所需的方式格式化字符串。

语法: `template.format(p1, p1, .... , k1=v1, k2=v2)`

模板是一个包含格式代码的字符串， `format()` 方法使用它的参数替换每个格式代码的值。 例如:

```
1.  >>> 'Sam has {0} red balls and {1} yellow balls'.format(12, 31)
```

`{0}` 和 `{1}` 是格式代码。 格式代码 `{0}` 替换为 `format()` 的第一个参数，即 `12` ，而 `{1}` 替换为 `format()` 的第二个参数，即 `31` 。

预期输出:

```
1.  Sam has 12 red balls and 31 yellow balls
```

对于简单的格式化，该技术是可以的，但是如果要在浮点数中指定精度怎么办？ 对于这种情况，您需要了解有关格式代码的更多信息。 这是格式代码的完整语法。

语法: `{[argument_index_or_keyword]:[width][.precision][type]}`

`type` 可以与格式代码一起使用:

格式码	描述
`d`	用于整数
`f`	用于浮点数
`b`	用于二进制数
`o`	八进制数
`x`	八进制十六进制数
`s`	用于字符串
`e`	用于指数格式的浮点

以下示例将使事情更加清楚。

示例 **1:**

```
>>> "Floating point {0:.2f}".format(345.7916732)
```

在这里，我们指定精度的 `2` 位， `f` 用于表示浮点数。

预期输出：

```
Floating point 345.79
```

示例 **2**：

```
>>> import math
>>> "Floating point {0:10.3f}".format(math.pi)
```

在这里，我们指定 `3` 精度数字， `10` 表示宽度， `f` 表示浮点数。

预期输出：

```
Floating point 3.142
```

示例 **3**：

```
"Floating point pi = {0:.3f}, with {1:d} digit precision".format(math.pi, 3)
```

这里 `{1:d}` 中的 `d` 表示整数值。

预期输出：

```
Floating point pi = 3.142, with 3 digit precision
```

如果为整数 `ValueError` 指定精度，则仅在浮点数的情况下才需要指定精度。

示例 **5**：

```
'Sam has {1:d} red balls and {0:d} yellow balls'.format(12, 31)
```

预期输出：

```
Sam has 31 red balls and 12 yellow balls
```

示例 **6**：

```
"In binary 4 is {0:b}".format(4) # b for binary, refer to Fig 1.1
```

预期输出：

```
In binary 4 is 100
```

示例 **7**：

```
array = [34, 66, 12]
"A = {0}, B = {1}, C = {2}".format(*array)
```

预期输出：

```
A = 34, B = 66, C = 12
```

示例 **8**：

```
d = {
'hats' : 122,
'mats' : 42
}

"Sam had {hats} hats and {mats} mats".format(**d)
```

预期输出：

```
Sam had 122 hats and 42 mats
```

`format()` 方法还支持关键字参数。

```
'Sam has {red} red balls and {green} yellow balls'.format(red = 12, green = 31)
```

请注意，在使用关键字参数时，我们需要在 `{}` 内部使用参数，而不是数字索引。

您还可以将位置参数与关键字参数混合

```
'Sam has {red} red balls, {green} yellow balls \
and {0} bats'.format(3, red = 12, green = 31)
```

格式化字符串的 `format()` 方法是一个非常新的方法，它是在 Python 2.6 中引入的。 您将在旧

版代码中看到另一种古老的技术，它允许您使用 `%` 运算符而不是 `format()` 方法来格式化字符串。

让我们举个例子。

```
1.  "%d pens cost = %.2f" % (12, 150.87612)
```

在这里，我们使用 `%` 左侧的模板字符串。 我们使用 `%` 代替格式代码的 `{}` 。 在 `%` 的右侧，我们使用元组包含我们的值。 `%d` 和 `%.2f` 被称为格式说明符，它们以 `%` 开头，后跟代表数据类型的字符。 例如， `%d` 格式说明符是整数的占位符，类似地 `%.2f` 是浮点数的占位符。

因此， `%d` 被替换为元组的第一值，即 `12` ，而 `%.2f` 被替换为第二值，即 `150.87612` 。

预期输出：

```
1.  12 pens cost = 150.88
```

一些更多的例子。

示例 **1**：

新：

```
1.  "{0:d} {1:d} ".format(12, 31)
```

旧：

```
1.  "%d  %d" % (12, 31)
```

预期输出：

```
1.  12 31
```

示例 **2**：

New:

```
1.  "{0:.2f} {1:.3f}".format(12.3152, 89.65431)
```

Old:

```
1.  "%.2f  %.3f" % (12.3152, 89.65431)
```

预期输出：

```
12.32 89.654
```

示例 3：

New：

```
"{0:s} {1:o} {2:.2f} {3:d}".format("Hello", 71, 45836.12589, 45 )
```

Old：

```
"%s  %o  %.2f  %d" % ("Hello", 71, 45836.12589, 45 )
```

预期输出：

```
Hello 107 45836.13 45
```

Python 内置函数和方法

Python `abs()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/abs/

于 2020 年 1 月 7 日更新

`abs()` 函数返回数字的绝对值（大小无符号）。

其语法如下:

```
abs(x) -> absolute value
```

参数	描述
`x`	任何数值

这是一个例子:

```
>>>
>>> abs(-45)
45
>>>
>>>
>>> abs(-3.14)
3.14
>>>
>>>
>>> abs(10)
10
>>>
>>>
>>> abs(2+4j)
4.47213595499958
>>>
```

```
print(abs(-45))
print(abs(-3.14))
print(abs(10))
print(abs(2+4j))
```

对于整数和浮点数，结果非常明显。 如果是 `z = x + yi` 的复数，则 `abs()` 函数将按如下方式计算绝对值：

绝对值： `|z| = √x² + y²`

```
1.  => 2+4j
2.  => √2² + 4²
3.  => √4 + 16
4.  => √20
5.  => 2√5
6.  => 2*2.236
7.  => 4.472
```

Python `bin()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/bin/

于 2020 年 1 月 7 日更新

`bin()` 函数以字符串形式返回整数的二进制表示形式。

其语法如下:

```
bin(number) -> binary representation
```

参数	描述
`number`	任何数值

这是一个例子:

```
>>>
>>> bin(4) # convert decimal to binary
'0b100'
>>>
>>>
>>> bin(0xff) # convert hexadecimal to binary, 0xff is same decimal 255
'0b11111111'
>>>
>>>
>>> bin(0o24) # convert octacl to binary, 0o24 is same decimal 20
'0b10100'
>>>
```

```
print(bin(4))
print(bin(0xff))
print(bin(0o24))
```

`bin()` 与用户定义的对象

要将 `bin()` 与用户定义的对象一起使用，我们必须首先重载 `__index__()` 方法。 在切片和索引

的上下文中， `__index__()` 方法用于将对象强制为整数。 例如，考虑以下内容：

```
>>>
>>> l = [1, 2, 3, 4, 5]
>>>
>>> x, y = 1, 3
>>>
>>>
>>> l[x]
2
>>>
>>>
>>> l[y]
4
>>>
>>>
>>> l[x:y]
[2, 3]
>>>
```

```
l = [1, 2, 3, 4, 5]
x, y = 1, 3
print(l[x])
print(l[y])
print(l[x:y])
```

当我们使用索引和切片访问列表中的项目时，内部 Python 会调用 `int` 对象的 `__index__()` 方法。

```
>>>
>>> l[x.__index__()] # same as l[x]
2
>>>
>>>
>>> l[y.__index__()] # same as l[y]
4
>>>
>>>
>>> l[x.__index__():y.__index__()] # # same as l[x:y]
[2, 3]
>>>
```

```
l = [1, 2, 3, 4, 5]
x, y = 1, 3
print(l[x.__index__()])
print(l[y.__index__()])
print(l[x.__index__():y.__index__()])
```

除了 `bin()` 之外，在对象上调用 `hex()` 和 `oct()` 时也会调用 `__index__()` 方法。 这是一个例子：

```
>>>
>>> class Num:
...     def __index__(self):
...         return 4
...
>>>
>>> l = [1, 2, 3, 4, 5]
>>>
>>>
>>> n1 = Num()
>>>
>>>
>>> bin(n1)
0b100
>>>
>>>
>>> hex(n1)
0x4
>>>
>>>
>>> oct(n1)
0o4
>>>
>>>
>>> l[n1]
5
>>>
>>>
>>> l[n1.__index__()]
5
>>>
```

```
class Num:
    def __index__(self):
        return 4

l = [1, 2, 3, 4, 5]

n1 = Num()

print(bin(n1))
print(hex(n1))
print(oct(n1))
print(l[n1])
print(l[n1.__index__()])
```

Python `id()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/id/

于 2020 年 1 月 7 日更新

`id()` 函数返回与对象关联的唯一数字标识符。

在标准 Python（即 CPython）中，标识符表示对象的内存地址。 虽然，这在其他实现中可能会有所不同。

定义对象时，唯一标识符会自动存在，并且在程序运行之前不会更改。 我们可以使用此标识符来确定两个对象是否相同。

`id()` 函数的语法如下：

```
id(obj) -> unique identifier
```

这是一个例子：

```
>>>
>>> a = 10
>>>
>>> b = 5
>>>
>>>
>>> id(a), id(b)
(10919712, 10919552)
>>>
>>>
>>> a = b # a now references same object as b
>>>
>>>
>>> id(a), id(b)
(10919552, 10919552)
>>>
```

```
a = 10
b = 5
```

```
3.
4.  print(id(a), id(b))
5.
6.  a = b # a now references same object as b
7.
8.  print(id(a), id(b))
```

最初，变量 `a` 和 `b` 引用两个不同的对象。 结果， `id()` 调用返回两个唯一标识符。 接下来，我们将对象 `b` 分配给 `a` 。 现在， `a` 和 `b` 引用相同的对象（ `5` ）。 因此，下一个 `id()` 调用返回相同的标识符。

Python `map()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/map/

于 2020 年 1 月 7 日更新

将 `map()` 内置函数应用于序列中的每个项目后，它会返回一个迭代器。 其语法如下：

语法: `map(function, sequence[, sequence, ...]) -> map object`

Python 3

```
>>>
>>> map(ord, ['a', 'b', 'c', 'd'])
<map object at 0x7f36fac76dd8>
>>>
>>> list(map(ord, ['a', 'b', 'c', 'd']))
>>> [97, 98, 99, 100]
>>>
```

```
map_obj = map(ord, ['a', 'b', 'c', 'd'])
print(map_obj)
print(list(map_obj))
```

在此，列表中的项目一次被传递到 `ord()` 内置函数。

由于 `map()` 返回一个迭代器，因此我们使用了 `list()` 函数立即生成结果。

上面的代码在功能上等同于以下代码：

Python 3

```
>>>
>>> ascii = []
>>>
>>> for i in ['a', 'b', 'c', 'd']:
...   ascii.append(ord(i))
...
>>>
>>> ascii
```

```
[97, 98, 99, 100]
>>>
```

```
ascii = []

for i in ['a', 'b', 'c', 'd']:
    ascii.append(ord(i))

print(ascii)
```

但是，使用 `map()` 会导致代码缩短，并且通常运行速度更快。

在 Python 2 中， `map()` 函数返回一个列表，而不是一个迭代器（就内存消耗而言，这不是很有效），因此我们无需在 `list()` 调用中包装 `map()` 。

Python 2

```
>>>
>>> map(ord, ['a', 'b', 'c', 'd']) # in Python 2
[97, 98, 99, 100]
>>>
```

传递用户定义的函数

在下面的清单中，我们将用户定义的函数传递给 `map()` 函数。

Python 3

```
>>>
>>> def twice(x):
...     return x*2
...
>>>
>>> list(map(twice, [11,22,33,44,55]))
[22, 44, 66, 88, 110]
>>>
```

```
def twice(x):
    return x*2

```

```
print(list(map(twice, [11,22,33,44,55])))
```

在此，该函数将列表中的每个项目乘以 2。

传递多个参数

如果我们将 `n` 序列传递给 `map()` ，则该函数必须采用 `n` 个参数，并且并行使用序列中的项，直到用尽最短的序列。 但是在 Python 2 中，当最长的序列被用尽时， `map()` 函数停止，而当较短的序列被用尽时， `None` 的值用作填充。

Python 3

```
>>>
>>> def calc_sum(x1, x2):
...     return x1+x2
...
>>>
>>> list(map(calc_sum, [1, 2, 3, 4, 5], [10, 20, 30]))
[11, 22, 33]
>>>
```

```
def calc_sum(x1, x2):
    return x1+x2

map_obj = list(map(calc_sum, [1, 2, 3, 4, 5], [10, 20, 30]))
print(map_obj)
```

Python 2

```
>>>
>>> def foo(x1, x2):
...     if x2 is None:
...         return x1
...     else:
...         return x1+x2
...
>>>
>>> list(map(foo, [1, 2, 3, 4, 5], [10, 20, 30]))
[11, 22, 33, 4, 5]
>>>
```

传递 Lambda

如果您的函数不打算被重用，则可以传递 lambda（内联匿名函数）而不是函数。

Python 3

```
>>>
>>> list(map(lambda x1:x1*5, [1, 2, 3]))
[5, 10, 15]
>>>
```

```
map_obj = map(lambda x1:x1*5, [1, 2, 3])
print(list(map_obj))
```

在此，该函数将列表中的每个项目乘以 5。

配对项目（仅在 Python 2 中）

最后，您可以通过传递 `None` 代替函数来配对多个序列中的项目：

Python 2

```
>>>
>>> map(None, "hello", "pi")
[('h', 'p'), ('e', 'i'), ('l', None), ('l', None), ('o', None)]
>>>
```

请注意，当较短的序列用尽时，将使用 `None` 填充结果。

这种形式的 `map()` 在 Python 3 中无效。

```
>>>
>>> list(map(None, "hello", "pi"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'NoneType' object is not callable
>>>
```

```
print(list(map(None, "hello", "pi")))
```

如果要配对多个序列中的项目，请使用 `zip()` 函数。

Python `zip()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/zip/

于 2020 年 1 月 7 日更新

`zip()` 函数采用一个或多个序列，并将序列中的对应项组合成一个元组。 最短序列用完时它将停止。 在 Python 2 中， `zip()` 返回一个实际的列表，如果您处理大量数据则效率不高。 因此，在 Python 3 中， `zip()` 返回一个可迭代的对象，该对象可按需生成结果。

语法: `zip(iter1 [,iter2 [...]]) --> zip object`

Python 3

```
>>>
>>> zip([1, 2, 3, 4], "pow")
<zip object at 0x7f3c1ceb51c8>
>>>
```

要产生结果，请在 `list()` 调用中包装 `zip()` 。

```
>>>
>>> list(zip([1, 2, 3, 4], "pow"))
[(1, 'p'), (2, 'o'), (3, 'w')]
>>>
```

试试看:

```
zip_obj = zip([1, 2, 3, 4], "pow")
print(list(zip_obj))
```

Python 2

```
>>>
>>> zip([1, 2, 3, 4], "pow") # In Python 2, list() call is not required
[(1, 'p'), (2, 'o'), (3, 'w')]
>>>
```

这是一个实际示例，其中 `zip()` 用于并行迭代多个序列。

```
>>>
>>> for i, j, k, l in zip([1, 2, 3], "foo", ("one", "two", "three"), {"alpha",
"beta", "gamma"}):
...   print(i, j, k, l)
...
1 f one alpha
2 o two gamma
3 o three beta
>>>
```

试一试：

```
for i, j, k, l in zip([1, 2, 3],  "foo",
                      ("one", "two", "three"),
                      {"alpha", "beta", "gamma"}
                     ):
    print(i, j, k, l)
```

这是另一个使用 `zip()` 函数创建字典的示例。

```
>>>
>>> keys = ['alpha', 'beta', 'gamma']
>>> values = [10, 20, 30]
>>>
>>> d = dict(zip(keys, values))
>>> d
{'alpha': 10, 'beta': 20, 'gamma': 30}
>>>
```

试一试：

```
keys = ['alpha', 'beta', 'gamma']
values = [10, 20, 30]

d = dict(zip(keys, values))
print(d)
```

Python `filter()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/filter/

于 2020 年 1 月 7 日更新

`filter()` 函数将一个函数和一个序列作为参数并返回一个可迭代的对象，仅按顺序产生要为其返回 `True` 的项目。 如果传递了 `None` 而不是函数，则将求值为 `False` 的序列中的所有项目删除。 `filter()` 的语法如下：

语法： `filter(function or None, iterable) --> filter object`

这是一个例子：

Python 3

```
>>>
>>> def is_even(x):
...     if x % 2 == 0:
...         return True
...     else:
...         return False
...
>>>
>>> f = filter(is_even, [1, 3, 10, 45, 6, 50])
>>>
>>> f
<filter object at 0x7fcd88d54eb8>
>>>
>>>
>>> for i in f:
...     print(i)
...
10
6
50
>>>
```

试试看：

```
def is_even(x):
    if x % 2 == 0:
        return True
    else:
        return False

f = filter(is_even, [1, 3, 10, 45, 6, 50])

print(f)

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

要立即产生结果，我们可以使用 `list()` 函数。

Python 3

```
>>>
>>> list(filter(is_even, [1, 3, 10, 45, 6, 50]))
[10, 6, 50]
>>>
>>>
>>> list(filter(None, [1, 45, "", 6, 50, 0, {}, False])) # function argument is None
[1, 45, 6, 50]
>>>
```

试一试：

```
def is_even(x):
    if x % 2 == 0:
        return True
    else:
        return False

print( list(filter(is_even, [1, 3, 10, 45, 6, 50])) )

# function argument is None
print( list(filter(None, [1, 45, "", 6, 50, 0, {}, False])) )
```

在 Python 2 中， `filter()` 返回实际列表（这不是处理大数据的有效方法），因此您无需将 `filter()` 包装在 `list()` 调用中。

Python 2

```
>>>
>>> filter(is_even, [1, 3, 10, 45, 6, 50])
[10, 6, 50]
>>>
```

这是其他一些例子。

Python 3

```
>>>
>>> filter(lambda x: x % 2 != 0, [1, 3, 10, 45, 6, 50]) # lambda is used in place of a function
[1, 3, 45]
>>>
>>>
>>> list(filter(bool, [10, "", "py"]))
[10, 'py']
>>>
>>>
>>> import os
>>>
>>> # display all files in the current directory (except the hidden ones)
>>> list(filter(lambda x: x.startswith(".") != True, os.listdir(".") ))
['Documents', 'Downloads', 'Desktop', 'Pictures', 'bin', 'opt', 'Templates', 'Public', 'Videos', 'Music']
>>>
```

试一试：

```
 # lambda is used in place of a function
print(filter(lambda x: x % 2 != 0, [1, 3, 10, 45, 6, 50]))

print(list(filter(bool, [10, "", "py"])))

import os

# display all files in the current directory (except the hidden ones)
print(list(filter(lambda x: x.startswith(".") != True, os.listdir(".") )) )
```

Python `reduce()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/reduce/

于 2020 年 1 月 7 日更新

`reduce()` 函数接受一个函数和一个序列并返回如下计算的单个值:

1. 最初，使用序列中的前两项调用该函数，然后返回结果。
2. 然后使用在步骤 1 中获得的结果和序列中的下一个值再次调用该函数。 这个过程一直重复，直到序列中有项目为止。

`reduce()` 函数的语法如下:

语法: `reduce(function, sequence[, initial]) -> value`

提供 `initial` 值时，将使用 `initial` 值和序列中的第一项调用该函数。

在 Python 2 中， `reduce()` 是一个内置函数。 但是，在 Python 3 中，它已移至 `functools` 模块。 因此，要使用它，必须先按以下步骤导入它:

```
1.  from functools import reduce # only in Python 3
```

这是添加列表中所有项目的示例。

```
1.  >>>
2.  >>> from functools import reduce
3.  >>>
4.  >>> def do_sum(x1, x2): return x1 + x2
5.  ...
6.  >>>
7.  >>> reduce(do_sum, [1, 2, 3, 4])
8.  10
9.  >>>
```

试试看:

```
1.  from functools import reduce
2.
3.  def do_sum(x1, x2):
```

```
    return x1 + x2

print(reduce(do_sum, [1, 2, 3, 4]))
```

此 `reduce()` 调用执行以下操作：

```
(((1 + 2) + 3) + 4) => 10
```

前面的 `reduce()` 调用在功能上等同于以下内容：

```
>>>
>>> def my_reduce(func, seq):
...     first = seq[0]
...     for i in seq[1:]:
...         first = func(first, i)
...     return first
...
>>>
>>> my_reduce(do_sum, [1, 2, 3, 4])
10
>>>
```

试一试：

```
def do_sum(x1, x2):
    return x1 + x2

def my_reduce(func, seq):
    first = seq[0]
    for i in seq[1:]:
        first = func(first, i)
    return first

print(my_reduce(do_sum, [1, 2, 3, 4]))
```

但是， `reduce()` 调用比 `for` 循环更简洁，并且性能明显更好。

Python `sorted()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/sorted/

于 2020 年 1 月 7 日更新

`sorted()` 内置函数允许我们对数据进行排序。 它接受一个可迭代对象，并返回一个包含来自可迭代对象的项目的排序列表。 默认情况下，它以升序排序。

`sorted()` 函数的语法如下：

语法： `sorted(iterable, key=None, reverse=False)`

参数	描述
`iterable`	（必需）可迭代地进行排序，例如字符串，列表，字典，元组等。
`key` ，（可选）	它引用单个参数函数以自定义排序顺序。 该函数应用于迭代器上的每个项目。 默认情况下，此参数设置为 `None` 。
`reverse`	（可选）布尔值标志以反转排序顺序。 默认为 `False` 。

如果可迭代对象中的项目是字符串，则将按字母顺序对其进行排序。 另一方面，如果它们是数字，则将按数字顺序对其进行排序。

这是一个例子：

```
>>>
>>> fruits = ['lime', 'blueberry', 'plum', 'avocado']
>>>
>>> sorted(fruits) # default sorting, in ascending order
['avocado', 'blueberry', 'lime', 'plum']
>>>
>>>
>>> sorted(fruits, reverse=True) # reverse the sorting
['plum', 'lime', 'blueberry', 'avocado']
>>>
>>>
>>> ages = [45, 11, 30, 20, 55]
>>>
>>> sorted(ages)
[11, 20, 30, 45, 55]
>>>
```

```
>>> sorted(ages, reverse=True)
[55, 45, 30, 20, 11]
>>>
```

试试看：

```
fruits = ['lime', 'blueberry', 'plum', 'avocado']

print(sorted(fruits)) # default sorting, in ascending order
print(sorted(fruits, reverse=True)) # reverse the sorting

ages = [45, 11, 30, 20, 55]

print(sorted(ages))
print(sorted(ages, reverse=True))
```

请注意， `sorted()` 返回一个包含可迭代项的新列表。 它不会更改过程中的原始可迭代项。

```
>>>
>>> fruits # fruit list is same as before
['lime', 'blueberry', 'plum', 'avocado']
>>>
>>>
>>> ages # ages list is same as before
[45, 11, 30, 20, 55]
>>>
```

试一试：

```
fruits = ['lime', 'blueberry', 'plum', 'avocado']
ages = [45, 11, 30, 20, 55]

print(sorted(fruits)) # default sorting, in ascending order
print(sorted(fruits, reverse=True)) # reverse the sorting

print(fruits)
print(ages)
```

以下是一些其他示例，显示 `sorted()` 如何与其他 Python 类型一起使用。

带字符串的 `sorted()`

```
>>>
>>> name = "Alfred Hajos"
>>>
>>> sorted(name)
[' ', 'A', 'H', 'a', 'd', 'e', 'f', 'j', 'l', 'o', 'r', 's']
>>>
>>>
>>> sorted(name, reverse=True)
['s', 'r', 'o', 'l', 'j', 'f', 'e', 'd', 'a', 'H', 'A', ' ']
>>>
```

试一试：

```
name = "Alfred Hajos"

print(sorted(name))
print(sorted(name, reverse=True))
```

请注意，在第一个 `sorted()` 调用 `A` 的结果出现在 `a` 之前。 这是因为 `A` 的 ASCII 值为 65， `a` 的 ASCII 值为 97。出于相同的原因，空格字符（ `' '` ）的 ASCII 值 32 出现在 `A` 之前。

带元组的 `sorted()`

```
>>>
>>> t = ( 'ff', 'xx', 'gg', 'aa')
>>>
>>> sorted(t)
['aa', 'ff', 'gg', 'xx']
>>>
>>> sorted(t, reverse=True)
['xx', 'gg', 'ff', 'aa']
>>>
```

试一试：

```
t = ( 'ff', 'xx', 'gg', 'aa')

print(sorted(t))
```

```
print(sorted(t, reverse=True))
```

带字典的 `sorted()`

```
>>>
>>> d = {'name': 'John', 'age': 25, 'designation': 'manager'}
>>>
>>>
>>> sorted(d)
['age', 'designation', 'name']
>>>
>>> sorted(d, reverse=True)
['name', 'designation', 'age']
>>>
>>>
>>> for k in sorted(d):
...     print(k, d[k])
...
age 25
designation manager
name John
>>>
>>>
>>> for k in sorted(d, reverse=True):
...     print(k, d[k])
...
name John
designation manager
age 25
>>>
```

```
d = {'name': 'John', 'age': 25, 'designation': 'manager'}

print(sorted(d))
print(sorted(d, reverse=True))

for k in sorted(d):
    print(k, d[k])

print('-'*10)
```

```
10.
11. for k in sorted(d, reverse=True):
12.     print(k, d[k])
```

使用命名参数 `key` 自定义排序顺序

从上一节中我们知道，如果将 `sorted()` 函数应用于字符串列表，则将获得按字母顺序排序的字符串列表。

如果我们想按字符串的长度而不是字母顺序排序怎么办？

这是 `key` 命名参数出现的地方。

要按字符串长度排序，请按以下所示将 `len()` 函数的键命名参数设置为：

```
1. >>>
2. >>> fruits
3. ['lime', 'blueberry', 'plum', 'avocado']
4. >>>
5. >>> sorted(fruits) # sort by alphabetical order
6. ['avocado', 'blueberry', 'lime', 'plum']
7. >>>
8. >>> sorted(fruits, key=len) # sort by string length
9. ['lime', 'plum', 'avocado', 'blueberry']
10. >>>
11. >>> sorted(fruits, key=len, reverse=True) # reverse sort order
12. ['blueberry', 'avocado', 'lime', 'plum']
13. >>>
```

试一试：

```
1. fruits = ['lime', 'blueberry', 'plum', 'avocado']
2.
3. print(fruits)
4.
5. print(sorted(fruits))
6.
7. print(sorted(fruits, key=len))  # sort by string length
8.
9. print(sorted(fruits, key=len, reverse=True)) # reverse sort order
```

有时您可能希望使排序不区分大小写。 我们可以通过将 `key` 的命名参数设置为 `str.lower` 函数来轻松实现此目的。

```
>>>
>>> t = ( 'AA', 'aa', 'ZZ', 'cc', 'bb')
>>>
>>> sorted(t)
['AA', 'ZZ', 'aa', 'bb', 'cc']
>>>
>>> sorted(t, key=str.lower)
['AA', 'aa', 'bb', 'cc', 'ZZ']
>>>
```

```
t = ( 'AA', 'aa', 'ZZ', 'cc', 'bb')

print(sorted(t))
print(sorted(t, key=str.lower))
```

这是另一个示例，该示例使用自定义函数根据其包含的元音数量对字符串列表进行排序。

```
>>>
>>> fruits
['lime', 'blueberry', 'plum', 'avocado']
>>>
>>>
>>> def count_vowel(s):
...     vowels = ('a', 'e', 'i', 'o', 'u')
...     count = 0
...
...     for i in s:
...         if i in vowels:
...         count = count + 1
...
...     return count
...
>>>
>>>
>>> sorted(fruits)
['avocado', 'blueberry', 'lime', 'plum']
>>>
>>> sorted(fruits, key=count_vowel)
```

```
['plum', 'lime', 'blueberry', 'avocado']
>>>
```

试一试：

```
fruits = ['lime', 'blueberry', 'plum', 'avocado']

def count_vowel(s):
    vowels = ('a', 'e', 'i', 'o', 'u')
    count = 0

    for i in s:
        if i in vowels:
            count = count + 1

    return count

print(sorted(fruits))

print(sorted(fruits, key=count_vowel))
```

您还可以在用户定义的对象上使用 `sorted()` 。

```
>>>
>>> class Employee:
...     def __init__(self, name, salary, age):
...         self.name = name
...         self.salary = salary
...         self.age = age
...
...     def __repr__(self):
...         return self.__str__()
...
...     def __str__(self):
...         return "{0}:{1}:{2}".format(self.name, self.salary, self.age)
...
>>>
>>>
>>> e1 = Employee("Tom", 20000, 32)
>>> e2 = Employee("Jane", 50000, 36)
>>> e3 = Employee("Bob", 45000, 40)
>>>
```

```
>>>
>>> emp_list = [e2, e3, e1]
>>>
>>> print(emp_list)
[Jane:50000:36, Bob:45000:40, Tom:20000:32]
>>>
```

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

    def __repr__(self):
        return self.__str__()

    def __str__(self):
        return "{0}:{1}:{2}".format(self.name, self.salary, self.age)

e1 = Employee("Tom", 20000, 32)
e2 = Employee("Jane", 50000, 36)
e3 = Employee("Bob", 45000, 40)

emp_list = [e2, e3, e1]

print(emp_list)

# print(sorted(emp_list))
```

如果现在在 `emp_list` 上调用 `sorted()` ，则会出现如下错误：

```
>>>
>>> sorted(emp_list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Employee() < Employee()
>>>
```

提示：

上面的代码不会在 Python 2 中引发任何错误。相反，它将根据 `id()` 内置函数返回的 ID 对 `Employee` 对象进行排序。

发生这种情况是因为 Python 不知道如何比较 `Employee` 对象。 我们可以通过在 `Employee` 类中实现特殊方法（例如 `__lt__()` ， `__gt__()` 等）来告诉 Python 如何比较对象。

代替定义特殊方法，我们可以显式告诉 `sorted()` 函数如何使用 `key` 命名参数对 `Employee` 对象进行排序。

```
>>>
>>> sorted(emp_list, key=lambda x: x.name) # sort Employee objects by name
[Bob:45000:40, Jane:50000:36, Tom:20000:32]
>>>
>>>
>>> sorted(emp_list, key=lambda x: x.age) # sort Employee objects by age
[Tom:20000:32, Jane:50000:36, Bob:45000:40]
>>>
>>>
>>> print(sorted(emp_list, key=lambda x: x.salary)) # sort Employee objects by
salary
[Tom:20000:32, Bob:45000:40, Jane:50000:36]
>>>
```

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

    def __repr__(self):
        return self.__str__()

    def __str__(self):
        return "{0}:{1}:{2}".format(self.name, self.salary, self.age)

e1 = Employee("Tom", 20000, 32)
e2 = Employee("Jane", 50000, 36)
e3 = Employee("Bob", 45000, 40)

emp_list = [e2, e3, e1]

print(sorted(emp_list, key=lambda x: x.name)) # sort Employee objects by name

print(sorted(emp_list, key=lambda x: x.age)) # sort Employee objects by age

```

```
    print(sorted(emp_list, key=lambda x: x.salary)) # sort Employee objects by
23. salary
```

Python `enumerate()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/enumerate/

于 2020 年 1 月 7 日更新

`enumerate()` 函数采用一个可迭代对象，并返回一个枚举对象（一个迭代器），该对象生成一个格式为 `(index, item)` 的元组，其中 `index` 指该项目的偏移量， `item` 指的是可迭代对象中的对应的项目。

`enumerate()` 函数的语法如下：

语法：

```
enumerate(iterable[, start=0]) -> iterator for index, value of iterable
```

参数	描述
`iterable`	（必需）任何可迭代的对象，例如字符串，列表，字典等。
`start` （可选）	`index` 的初始值。 默认为 `0` 。

这是一个例子：

```
>>>
>>> list(enumerate("hello"))
[(0, 'h'), (1, 'e'), (2, 'l'), (3, 'l'), (4, 'o')]
>>>
>>>
>>> for index, value in enumerate("hello"):
...     print(index, value)
...
0 h
1 e
2 l
3 l
4 o
>>>
```

试试看：

```
print( list(enumerate("hello")) )

for index, value in enumerate("hello"):
    print(index, value)
```

以下清单显示 `enumerate()` 如何与清单，字典和元组一起使用：

```
>>>
>>> for index, value in enumerate([110, 45, 12, 891, "one"]):
...     print(index, value)
...
0 110
1 45
2 12
3 891
4 one
>>>
>>>
>>> for index, value in enumerate({'name': 'Jane', 'age': 26, 'salary':
40000}):
...     print(index, value)
...
0 name
1 salary
2 age
>>>
>>>
>>> for index, value in enumerate({1, 290, -88, 10}):
...     print(index, value)
...
0 -88
1 1
2 10
3 290
>>>
```

试一试：

```
for index, value in enumerate([110, 45, 12, 891, "one"]):
    print(index, value)

print("-"*20)
```

```
5.
6.  for index, value in enumerate({'name': 'Jane', 'age': 26, 'salary': 40000}):
7.      print(index, value)
8.
9.  print("-"*20)
10.
11. for index, value in enumerate({1, 290, -88, 10}):
12.     print(index, value)
```

设置索引的初始值

要设置索引的初始值，我们使用 `start` 关键字参数。

```
1.  >>>
2.  >>> list(enumerate("hello", start=2))
3.  [(2, 'h'), (3, 'e'), (4, 'l'), (5, 'l'), (6, 'o')]
4.  >>>
5.  >>>
6.  >>> for index, value in enumerate("hello", start=2):
7.  ... print(index, value)
8.  ...
9.  2 h
10. 3 e
11. 4 l
12. 5 l
13. 6 o
14. >>>
```

试一试：

```
1.  print( list(enumerate("hello", start=2)) )
2.
3.  for index, value in enumerate("hello", start=2):
4.      print(index, value)
```

Python `reversed()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/reversed/

于 2020 年 1 月 7 日更新

`reversed()` 函数允许我们以相反的顺序处理项目。 它接受一个序列并返回一个迭代器。

其语法如下:

语法:

```
reversed(sequence) -> reverse iterator
```

参数	描述
`sequence`	序列列表字符串，列表，元组等。

这里有些例子:

```
>>>
>>> reversed([44, 11, -90, 55, 3])
<list_reverseiterator object at 0x7f2aff2f91d0>
>>>
>>>
>>> list(reversed([44, 11, -90, 55, 3])) # reversing a list
[3, 55, -90, 11, 44]
>>>
>>>
>>> list(reversed((6, 1, 3, 9))) # reversing a tuple
[9, 3, 1, 6]
>>>
>>> list(reversed("hello")) # reversing a string
['o', 'l', 'l', 'e', 'h']
>>>
```

试试看:

```
print( reversed([44, 11, -90, 55, 3]) )

print(list(reversed([44, 11, -90, 55, 3]))) # reversing a list
```

```
4.
5.  print( list(reversed((6, 1, 3, 9)))) # reversing a tuple
6.
7.  print(list(reversed("hello"))) # reversing a string
```

为了立即产生结果，我们将 `reversed()` 包装在 `list()` 调用中。 Python 2 和 Python 3 都需要这样做。

传递给 `reversed()` 的参数必须是正确的序列。 尝试传递不保持其顺序（例如 `dict` 和 `set` ）的对象将导致 `TypeError` 。

```
1.  >>>
2.  >>> reversed({0, 4, -2, 12, 6})
3.  Traceback (most recent call last):
4.    File "", line 1, in
5.  TypeError: argument to reversed() must be a sequence
6.  >>>
7.  >>>
8.  >>> reversed({'name': 'John', 'age': 20})
9.  Traceback (most recent call last):
10.   File "", line 1, in
11. TypeError: argument to reversed() must be a sequence
12. >>>
```

反转用户定义的对象

若要反转用户定义的对象，该类必须执行下列操作之一：

1. 实现 `__len__()` 和 `__getitem__()` 方法； 要么
2. 实现 `__reversed__()` 方法

在下面的清单中， `CardDeck` 类实现 `__len__()` 和 `__getitem__()` 方法。 结果，我们可以将 `reversed()` 应用于 `CardDeck` 实例。

```
1.  >>>
2.  >>> from collections import namedtuple
3.  >>>
4.  >>> Card = namedtuple('Card', ['rank', 'suit'])
5.  >>>
6.  >>> class CardDeck:
7.  ...     suits = ('club', 'diamond', 'heart', 'spades')
```

```
...     ranks = tuple((str(i) for i in range(2, 11))) + tuple("JQKA")
...
...     def __init__(self):
...         self._cards = [Card(r, s) for s in self.suits for r in self.ranks ]
...
...     def __len__(self):
...         return len(self._cards)
...
...     def __getitem__(self, index):
...         return self._cards[index]
...
...     #  def __reversed__(self):   this is how you would define __reversed__() method
...     #     return self._cards[::-1]
...
...
>>>
>>> deck = CardDeck()
>>>
>>> deck
<__main__.CardDeck object at 0x7f2aff2feb00>
>>>
>>>
>>> deck[0], deck[-1] # deck before reversing
(Card(rank='2', suit='club'), Card(rank='A', suit='spades'))
>>>
>>>
>>> reversed_deck = list(reversed(deck))
>>>
>>>
>>> reversed_deck[0], reversed_deck[-1] # deck after reversing
(Card(rank='A', suit='spades'), Card(rank='2', suit='club'))
>>>
```

试一试:

```
from collections import namedtuple

Card = namedtuple('Card', ['rank', 'suit'])

class CardDeck:
    suits = ('club', 'diamond', 'heart', 'spades')
```

```
7.      ranks = tuple((str(i) for i in range(2, 11))) + tuple("JQKA")
8.
9.      def __init__(self):
10.         self._cards = [Card(r, s) for s in self.suits for r in self.ranks ]
11.
12.     def __len__(self):
13.         return len(self._cards)
14.
15.     def __getitem__(self, index):
16.         return self._cards[index]
17.
        #  def __reversed__(self):   this is how you would define __reversed__()
18. method
19.     #     return self._cards[::-1]
20.
21. deck = CardDeck()
22.
23. print(deck)
24.
25. print( deck[0], deck[-1] ) # deck before reversing
26.
27. reversed_deck = list(reversed(deck))
28.
29. print(reversed_deck[0], reversed_deck[-1] ) # deck after reversing
```

Python `range()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/range/

于 2020 年 1 月 7 日更新

`range()` 函数用于随时间生成一系列数字。 简单地说，它接受一个整数并返回一个范围对象（一种可迭代的类型）。 在 Python 2 中， `range()` 返回一个 `list` ，它对处理大数据不是很有效。

`range()` 函数的语法如下：

语法：

```
range([start,] stop [, step]) -> range object
```

参数	描述
`start`	（可选）序列的起点。 默认为 `0` 。
`stop` （必填）	序列的端点。 该项目将不包括在序列中。
`step` （可选）	序列的步长。 默认为 `1` 。

现在让我们看几个示例，以了解 `range()` 的工作方式：

示例 **1**：

```
>>>
>>> range(5)
range(0, 5)
>>>
>>> list(range(5)) # list() call is not required in Python 2
[0, 1, 2, 3, 4]
>>>
```

试试看：

```
print(range(5))

# list() call is not required in Python 2
print(list(range(5)))
```

当使用单个参数调用 `range()` 时，它将生成从 `0` 到指定参数（但不包括它）的数字序列。 因此，序列中不包含数字 `5` 。

示例 **2:**

```
>>>
>>> range(5, 10)
range(5, 10)
>>>
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>>
```

试一试:

```
print(range(5, 10))

print(list(range(5, 10)))
```

在这里 `range()` 用两个参数 `5` 和 `10` 调用。 结果，它将生成从 `5` 到 `10` （但不包括 `10` ）的数字序列。

您还可以指定负数:

```
>>>
>>> list(range(-2, 2))
[-2, -1, 0, 1]
>>>
>>> list(range(-100, -95))
[-100, -99, -98, -97, -96]
>>>
```

试一试:

```
print(list(range(-2, 2)))

print(list(range(-100, -95)))
```

示例 **3:**

```
>>>
>>> range(1, 20, 3)
```

```
range(1, 20, 3)
>>>
>>>
>>> list(range(1, 20, 3))
[1, 4, 7, 10, 13, 16, 19]
>>>
```

试一试：

```
print( range(1, 20, 3))

print(list(range(1, 20, 3)))
```

在这里 `range()` 函数被 `3` 的 `step` 参数调用，因此它将每隔三个元素从 `1` 返回到 `20` （当然不包括 `20` ）。

您也可以使用步骤参数来倒数。

```
>>>
>>> list(range(20, 10, -1))
[20, 19, 18, 17, 16, 15, 14, 13, 12, 11]
>>>
>>> list(range(20, 10, -5))
[20, 15]
>>>
```

试一试：

```
print(list(range(20, 10, -1)))

print(list(range(20, 10, -5)))
```

`range()` 函数通常与 `for` 循环一起使用以重复执行一定次数的操作。 例如，在下面的清单中，我们使用 `range()` 执行循环主体 5 次。

```
>>>
>>> for i in range(5):
...     print(i)
...
0
1
2
```

```
3
4
>>>
```

试一试：

```
for i in range(5):
    print(i)
```

该代码在功能上等同于以下代码：

```
>>>
>>> for i in [0, 1, 2, 3, 4]:
...     print(i)
...
0
1
2
3
4
>>>
```

但是，在实际代码中，应始终使用 `range()` ，因为它简洁，灵活且性能更好。

Python `sum()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/sum/

于 2020 年 1 月 7 日更新

`sum()` 函数采用一个可迭代的并返回其中的项目总数。

语法：

```
sum(iterable, [start]) -> number
```

参数	描述
`iterable` （必填）	可迭代项，例如字符串，列表，字典等。
`start` （可选）	一个可选的数值添加到最终结果中。 默认为 `0` 。

`sum()` 函数仅适用于数字值，尝试将其用于非数字类型将导致错误。

这是一个例子：

```
>>>
>>> sum([1, 2, 3, 4, 5]) # sum values in a list
15
>>>
>>> sum((1, 2, 3, 4, 5)) # sum values in a tuple
15
>>>
>>> sum({1, 2, 3, 4, 5}) # sum values in a set
15
>>>
>>> sum({1: "one", 2: "two", 3: "three"}) # sum values in a
6
>>>
```

试试看：

```
print(sum([1, 2, 3, 4, 5])) # sum values in a list

print(sum((1, 2, 3, 4, 5))) # sum values in a tuple
```

```
4.
5.  print(sum({1, 2, 3, 4, 5})) # sum values in a set
6.
7.  print(sum({1: "one", 2: "two", 3: "three"})) # sum values in a
```

在最后一个命令中， `sum()` 将字典中的键添加进去，而忽略其值。

这是另一个示例，它指定要添加到最终结果中的 `start` 值。

```
1.  >>>
2.  >>> sum([10, 20, 30], 100)
3.  160
4.  >>>
```

试一试：

```
1.  print(sum([10, 20, 30], 100))
```

Python `max()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/max/

于 2020 年 1 月 7 日更新

`max()` 函数返回最大的输入值。

其语法如下:

```
1.  max(iterable[, default=obj, key=func]) -> value
```

参数	描述
`iterable` （必填）	可迭代对象，例如字符串，列表，元组等。
`default` （可选）	如果可迭代对象为空，则返回默认值。
`key` （可选）	它引用单个参数函数以自定义排序顺序。 该函数应用于迭代器上的每个项目。

要么

```
1.  max(a,b,c, ...[, key=func]) -> value
```

参数	描述
`a, b, c ...`	比较项目
`key` （可选）	它引用单个参数函数以自定义排序顺序。 该函数应用于迭代器上的每个项目。

如果以可迭代方式调用 `max()` ，它将返回其中的最大项。 如果可迭代对象为空，则返回 `default` 值，否则引发 `ValueError` 异常。

如果使用多个参数调用 `max()` ，它将返回最大的参数。

让我们看一些例子:

示例 **1** ：以可迭代方式调用 `max()`

```
1.  >>>
2.  >>> max("abcDEF") # find largest item in the string
3.  'c'
4.  >>>
5.  >>>
6.  >>> max([2, 1, 4, 3]) # find largest item in the list
```

```
4
>>>
>>>
>>> max(("one", "two", "three")) # find largest item in the tuple
'two'
>>>
>>>
>>> max({1: "one", 2: "two", 3: "three"}) # find largest item in the dict
3
>>>
>>>
>>> max([]) # empty iterable causes ValueError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: max() arg is an empty sequence
>>>
>>>
>>> max([], default=0) # supressing the error with default value
0
>>>
```

试试看：

```
# find largest item in the string
print(max("abcDEF"))

# find largest item in the list
print(max([2, 1, 4, 3]))

# find largest item in the tuple
print(max(("one", "two", "three")))
'two'

# find largest item in the dict
print(max({1: "one", 2: "two", 3: "three"}))
3

# empty iterable causes ValueError
# print(max([]))

# supressing the error with default value
print(max([], default=0))
```

示例 2 ：使用多个参数调用 `max()`

```
>>>
>>> max(20, 10, 30, -5)
30
>>>
>>>
>>> max("c", "b", "a", "Y", "Z")
'c'
>>>
>>>
>>> max(3.14, -9.91, 2.41)
3.14
>>>
```

试图在不同类型的对象中找到最大值会导致错误。

```
>>>
>>> max(10, "pypi")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() > int()
>>>
>>>
>>> max(5, [-10, 55])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: list() > int()
>>>
```

自定义排序顺序

为了自定义排序顺序，我们使用 `key` 命名参数。 它的工作原理类似于 `sorted()` 函数的 `key` 命名参数。

这是一个使用键参数使字符串比较区分大小写的示例。

```
>>>
>>> max("c", "b", "a", "Y", "Z")
'c'
```

```
>>>
>>> max("c", "b", "a", "Y", "Z", key=str.lower)
'Z'
>>>
```

试一试：

```
print(max("c", "b", "a", "Y", "Z"))

print(max("c", "b", "a", "Y", "Z", key=str.lower))
```

以下是另一个示例，其中我们根据字符串的长度而不是其 ASCII 值比较字符串。

```
>>>
>>> max(("python", "lua", "ruby"))
'ruby'
>>>
>>>
>>> max(("python", "lua", "ruby"), key=len)
'python'
>>>
```

试一试：

```
print(max(("python", "lua", "ruby")))

print(max(("python", "lua", "ruby"), key=len))
```

还存在一个名为 `min()` 的互补函数，该函数查找最低的输入值。

Python `min()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/min/

于 2020 年 1 月 7 日更新

`min()` 函数返回最小的输入值。

其语法如下:

```
1.  min(iterable[, default=obj, key=func]) -> value
```

参数	描述
`iterable` （必填）	可迭代对象，例如字符串，列表，元组等。
`default` （可选）	如果可迭代对象为空，则返回默认值。
`key` （可选）	它引用单个参数函数以自定义排序顺序。 该函数应用于迭代器上的每个项目。

要么

```
1.  min(a, b, c, ...[, key=func]) -> value
```

参数	描述
`a, b, c ...`	比较项目
`key` （可选）	它引用单个参数函数以自定义排序顺序。 该函数适用于每个项目。

如果以可迭代方式调用 `min()` ，它将返回其中的最小项。 如果可迭代对象为空，则返回 `default` 值（假设已提供），否则引发 `ValueError` 异常。

如果使用多个参数调用 `min()` ，它将返回最小的参数。

让我们看一些例子:

示例 **1** ：以可迭代方式调用 `min()`

```
1.  >>>
2.  >>> min("abcDEF") # find smallest item in the string
3.  'D'
4.  >>>
5.  >>>
6.  >>> min([2, -1, 4, 3]) # find smallest item in the list
```

```
-1
>>>
>>>
>>> min(("one", "two", "three")) # find smallest item in the tuple
'one'
>>>
>>>
>>> min({1: "one", 2: "two", 3: "three"}) # find smallest item in the dict
1
>>>
>>>
>>> min([]) # empty iterable causes ValueError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: min() arg is an empty sequence
>>>
>>>
>>> min([], default=0) # supressing the error with default value
0
>>>
```

试试看：

```
# find smallest item in the string
print(min("abcDEF"))

# find smallest item in the list
print(min([2, -1, 4, 3]))

# find smallest item in the tuple
print(min(("one", "two", "three")))

# find smallest item in the dict
print(min({1: "one", 2: "two", 3: "three"}))

#print(min([]))  # empty iterable causes ValueError

# supressing the error with default value
print(min([], default=0))
```

示例 2 ：使用多个参数调用 `min()`

```
>>>
>>> min(20, 10, 30, -5)
-5
>>>
>>>
>>> min("c", "b", "a", "Y", "Z")
'Y'
>>>
>>>
>>> min(3.14, -9.91, 2.41)
-9.91
>>>
```

试一试：

```
print(min(20, 10, 30, -5))

print(min("c", "b", "a", "Y", "Z"))

print(min(3.14, -9.91, 2.41))
```

试图在不同类型的对象中找到最大值会导致错误。

```
>>>
>>> min(10, "pypi")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() > int()
>>>
>>>
>>> min(5, [-10, 55])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: list() > int()
>>>
```

自定义排序顺序

为了自定义排序顺序，我们使用 `key` 命名参数。 它的工作原理类似于 `sorted()` 函数的 `key` 命名参数。

这是一个使用键参数使字符串比较区分大小写的示例。

```
>>>
>>> min("c", "b", "a", "Y", "Z")
'Y'
>>>
>>> min("c", "b", "a", "Y", "Z", key=str.lower)
'a'
>>>
```

试一试：

```
print(min("c", "b", "a", "Y", "Z"))

print(min("c", "b", "a", "Y", "Z", key=str.lower))
```

以下是另一个示例，其中我们根据字符串的长度而不是其 ASCII 值比较字符串。

```
>>>
>>> min(("java", "python", "z++"))
'java'
>>>
>>> min(("java", "python", "z++"), key=len)
'z++'
>>>
```

试一试：

```
print(min(("java", "python", "z++")))

print(min(("java", "python", "z++"), key=len))
```

还存在一个互补函数，称为 `max()` ，可找到最大的输入值。

Python `eval()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/eval/

于 2020 年 1 月 7 日更新

`eval()` 允许我们执行任意字符串作为 Python 代码。 它接受源字符串并返回一个对象。

其语法如下:

语法:

```
eval(expr, globals=None, locals=None)
```

参数	描述
`expr` (必填)	`expr` 可以是任何有效的 Python 表达式
`globals` (可选)	执行源时要使用的全局名称空间。 它必须是字典。 如果未提供，则将使用当前的全局名称空间。
`locals` (可选)	执行源时要使用的本地名称空间。 它可以是任何映射。 如果省略，则默认为 `globals` 字典。

如果同时省略 `globals` 和 `locals` ，则使用当前的全局和局部名称空间。

这是一个演示 `eval()` 如何工作的示例:

```
>>>
>>> eval("5 == 5")
True
>>>
>>>
>>> eval("4 < 10")
True
>>>
>>>
>>> eval("8 + 4 - 2 * 3")
6
>>>
>>>
>>> eval("'py ' * 5")
'py py py py py '
```

```
>>>
>>>
>>> eval("10 ** 2")
100
>>>
>>>
>>> eval("'hello' + 'py'")
'hellopy'
>>>
```

试试看：

```
print(eval("5 == 5"))

print(eval("4 < 10"))

print(eval("8 + 4 - 2 * 3"))

print(eval("'py ' * 5"))

print(eval("10 ** 2"))

print(eval("'hello' + 'py'"))
```

`eval()` 不仅限于简单表达。 我们可以执行函数，调用方法，引用变量等。

```
>>>
>>> eval("abs(-11)")
11
>>>
>>>
>>> eval('"hello".upper()')
'HELLO'
>>>
>>>
>>> import os
>>>
>>>
>>> eval('os.getcwd()') # get current working directory
'/home/thepythonguru'
>>>
>>>
```

```
>>> x = 2
>>>
>>> eval("x+4") # x is referenced inside the expression
6
>>>
```

试一试：

```
print(eval("abs(-11)"))

print(eval('"hello".upper()'))

import os

# get current working directory
print(eval('os.getcwd()'))

x = 2

print(eval("x+4")) # x is referenced inside the expression
```

请注意， `eval()` 仅适用于表达式。 尝试传递语句会导致 `SyntaxError` 。

```
>>>
>>> eval('a=1') # assignment statement
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1
    a=1
    ^
SyntaxError: invalid syntax
>>>
>>>
>>> eval('import re') # import statement
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1
    import re
         ^
SyntaxError: invalid syntax
>>>
```

邪恶的 `eval()`

您永远不要直接将不受信任的源传递给 `eval()` 。 由于恶意用户很容易对您的系统造成破坏。 例如，以下代码可以用于从系统中删除所有文件。

```
>>>
eval('os.system("RM -RF /")') # command is deliberately capitalized
>>>
```

如果 `os` 模块在您当前的全局范围内不可用，则以上代码将失败。 但是我们可以通过使用 `__import__()` 内置函数轻松地避免这种情况。

```
>>>
>>> eval("__import__('os').system('RM -RF /')") # command is deliberately
capitalized
>>>
```

那么有什么方法可以使 `eval()` 安全吗？

指定命名空间

`eval()` 可选地接受两个映射，作为要执行的表达式的全局和局部名称空间。 如果未提供映射，则将使用全局和局部名称空间的当前值。

这里有些例子：

示例 **1:**

```
>>>
>>> globals = {
... 'a': 10,
... 'fruits': ['mangoes', 'peaches', 'bananas'],
... }
>>>
>>>
>>> locals = {}
>>>
>>>
>>> eval("str(a) + ' ' + fruits[0]", globals, locals)
'10 mangoes'
```

```
>>>
```

示例 **2:**

```
>>>
>>> eval('abs(-100)', {}, {})
100
>>>
```

即使我们已经将空字典作为全局和本地名称空间传递了， `eval()` 仍可以访问内置函数（即 `__builtins__` ）。

```
>>>
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
'ChildProcessError',
...
...
'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr',
'slice', 'sorted'
, 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
>>>
```

要从全局名称空间中删除内置函数，请传递一个字典，该字典包含一个值为 `None` 的键 `__builtins__` 。

示例 **3:**

```
>>>
>>> eval('abs(-100)', {'__builtins__':None}, {})
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<string>", line 1, in <module>
TypeError: 'NoneType' object is not subscriptable
>>>
```

即使删除对内置函数的访问权限后， `eval()` 仍然不安全。 考虑以下清单。

```
>>>
>>> eval("5**98765432111123", {'__builtins__':None}, {})
>>>
```

这个看似简单的外观表达式足以使您的 CPU 崩溃。

关键要点是仅将 `eval()` 与受信任的源一起使用。

Python `len()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/len/

于 2020 年 1 月 7 日更新

`len()` 函数计算对象中的项目数。

其语法如下:

```
len(obj) -> length
```

参数	描述
`obj`	`obj` 可以是字符串，列表，字典，元组等。

这是一个例子:

```
>>>
>>> len([1, 2, 3, 4, 5]) # length of list
5
>>>
>>> print(len({"spande", "club", "diamond", "heart"})) # length of set
4
>>>
>>> print(len(("alpha", "beta", "gamma"))) # length of tuple
3
>>>
>>> print(len({ "mango": 10, "apple": 40, "plum": 16 })) # length of dictionary
3
```

试试看:

```
# length of list
print(len([1, 2, 3, 4, 5]))

# length of set
print(len({"spande", "club", "diamond", "heart"}))

# length of tuple
```

```
print(len(("alpha", "beta", "gamma")))

# length of dictionary
print(len({ "mango": 10, "apple": 40, "plum": 16 }))
```

具有讽刺意味的是， `len()` 函数不适用于生成器。 尝试在生成器对象上调用 `len()` 将导致 `TypeError` 异常。

```
>>>
>>> def gen_func():
...     for i in range(5):
...         yield i
...
>>>
>>>
>>> len(gen_func())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'generator' has no len()
>>>
```

试一试：

```
def gen_func():
    for i in range(5):
        yield i

print(len(gen_func()))
```

`len()` 与用户定义的对象

要在用户定义的对象上使用 `len()` ，您将必须实现 `__len__()` 方法。

```
>>>
>>> class Stack:
...
...     def __init__(self):
...         self._stack = []
...
...     def push(self, item):
```

```
...         self._stack.append(item)
...
...     def pop(self):
...         self._stack.pop()
...
...     def __len__(self):
...         return len(self._stack)
...
>>>
>>> s = Stack()
>>>
>>> len(s)
0
>>>
>>> s.push(2)
>>> s.push(5)
>>> s.push(9)
>>> s.push(12)
>>>
>>> len(s)
4
>>>
```

试一试：

```
class Stack:
    def __init__(self):
        self._stack = []

    def push(self, item):
        self._stack.append(item)

    def pop(self):
        self._stack.pop()

    def __len__(self):
        return len(self._stack)

s = Stack()

print(len(s))

```

```
18.  s.push(2)
19.  s.push(5)
20.  s.push(9)
21.  s.push(12)
22.
23.  print(len(s))
```

Python `ord()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/ord/

于 2020 年 1 月 7 日更新

`ord()` 函数（缺少序数）返回一个整数，表示传递给它的字符。 对于 ASCII 字符，返回值是 7 位 ASCII 代码，对于 Unicode 字符，返回值是指 Unicode 代码点。

其语法如下：

```
ord(c) -> code point
```

参数	描述
`c`	`c` 是字符串字符串。

这是一个例子：

```
>>>
>>> ord("A")
65
>>>
>>>
>>> ord("f")
102
>>>
>>>
>>> ord("á") # accented a
225
>>>
>>>
>>> ord("卍") # swastika
21325
>>>
>>>
>>> ord("😀") # Grinning Face
128512
>>>
```

试试看：

```
print(ord("A"))

print(ord("f"))

print(ord("á")) # accented a

print(ord("卍")) # swastika

print(ord("😀")) # Grinning Face
```

要将 `ord()` 返回的整数转换回其等效字符，我们使用 `chr()` 函数。

Python `chr()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/chr/

于 2020 年 1 月 7 日更新

`chr()` 函数返回由整数序数值表示的单个字符串。

其语法如下:

```
chr(integer) -> single character string
```

这是一个例子:

```
>>>
>>> chr(65)
'A'
>>>
>>>
>>> chr(102)
'f'
>>>
>>>
>>> chr(225) # accented a
'á'
>>>
>>>
>>> chr(21325) # swastika
'卍'
>>>
>>>
>>> chr(128512) # Grinning Face
'😀'
>>>
```

试试看:

```
print(chr(65))

```

```
print(chr(102))

print(chr(225)) # accented a

print(chr(21325)) # swastika

print(chr(128512)) # Grinning Face
```

要将字符转换回整数，请使用 `ord()` 函数。

Python `any()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/any/

于 2020 年 1 月 7 日更新

`any()` 函数测试可迭代项中的任何项目是否求值为 `True` 。 它接受一个可迭代对象并返回 `True` ，如果可迭代对象中的至少一项为 `true` ，则返回 `False` 。

其语法如下：

```
any(iterable) -> boolean
```

这是一个例子：

```
>>>
>>> any([10, "", "one"])
True
>>>
>>>
>>> any(("", {}))
False
>>>
>>>
>>>
>>> any([])
False
>>>
>>>
>>> gen = (i for i in [5, 0, 0.0, 4]) # generator
>>>
>>> any(gen)
True
>>>
```

试试看：

```
print(any([10, "", "one"]))

```

```
print(any(("", {})))

print(any([]))

gen = (i for i in [5, 0, 0.0, 4]) # generator

print(any(gen))
```

Python `all()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/all/

于 2020 年 1 月 7 日更新

`all()` 函数测试可迭代项中的所有项目是否都等于 `True` 。 如果所有项目都为 `true` ，它将接受一个可迭代对象并返回 `True` ，否则返回 `False` 。

其语法如下：

```
all(iterable) -> boolean
```

这是一个例子：

```
>>>
>>> all(['alpha', 'beta', ''])
False
>>>
>>>
>>> all(['one', 'two', 'three'])
True
>>>
>>>
>>> all([])
True
>>>
>>>
>>> gen = (i for i in ['0', (), {}, 51, 89]) # generator
>>>
>>>
>>> all(gen)
False
>>>
```

试试看：

```
print(all(['alpha', 'beta', '']))

```

```
print(all(['one', 'two', 'three']))

print(all([]))

gen = (i for i in ['0', (), {}, 51, 89]) # generator

print(all(gen))
```

Python `globals()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/globals/

于 2020 年 1 月 7 日更新

`globals()` 函数返回一个字典，其中包含在全局命名空间中定义的变量。 当从函数或方法中调用 `globals()` 时，它将返回表示该函数或方法所定义的模块的全局命名空间的字典，而不是从其调用处。

其语法如下：

```
globals() -> dictionary
```

让我们举一些例子：

示例 **1**：

`module1.py`

```
#!/usr/bin/python3

from pprint import pprint

a = 100
b = 4

def foo():
    x = 100 # x is a local variable

pprint(globals())
```

预期输出：

```
{'__builtins__': <module 'builtins' (built-in)>,
'__cached__': None,
'__doc__': None,
'__file__': './module1.py',
'__loader__': <_frozen_importlib_external.SourceFileLoader object at
0x7f699cab37f0>,
```

```
'__name__': '__main__',
'__package__': None,
'__spec__': None,
'a': 100,
'b': 4,
'foo': <function foo at 0x7f699ca1e2f0>,
'pprint': <function pprint at 0x7f699ca1e6a8>}
```

试试看：

```
from pprint import pprint

a = 100
b = 4

def foo():
    x = 100 # x is a local variable

pprint(globals())
```

以双下划线开头和结尾的名称是特殊的，并且由 Python 解释器定义。 我们在模块中定义的变量最后出现。

请注意，在 `foo()` 函数内部定义的局部变量 `x` 不包含在结果中。 要访问本地名称空间，请使用 `locals()` 函数。

示例 2：

module1.py

```
#!/usr/bin/python3

from pprint import pprint

a = 100
b = 4

def foo():
    x = 100 # x is a local variable
    pprint(globals())
```

module2.py

```
#!/usr/bin/python3

import module1

x = 100
y = 2

module1.foo()
```

预期输出：

```
{'__builtins__': { ... }
'__cached__': '/home/overiq/tmp/__pycache__/module1.cpython-35.pyc',
'__doc__': None,
'__file__': '/home/overiq/tmp/module1.py',
'__loader__': <_frozen_importlib_external.SourceFileLoader object at
0x7f17b12305c0>,
'__name__': 'module1',
'__package__': '',
'__spec__': ModuleSpec(name='module1', loader=
<_frozen_importlib_external.SourceFileLoader object at 0x7f17b12305c0>,
origin='/home/overiq/tmp/module1.py'),
'a': 100,
'b': 4,
'foo': <function foo at 0x7f17b121d488>,
'pprint': <function pprint at 0x7f17b121d730>}
```

在这种情况下， `globals()` 调用位于 `foo()` 函数内部。 当从 `module2` 调用 `foo()` 函数时，它将打印在 `module1` 的全局命名空间中定义的变量。

Python `locals()` 函数

> 原文: https://thepythonguru.com/python-builtin-functions/locals/

于 2020 年 1 月 7 日更新

`locals()` 函数返回一个字典，其中包含在本地名称空间中定义的变量。 在全局名称空间中调用 `locals()` 与调用 `globals()` 相同，并返回代表模块全局名称空间的字典。

其语法如下：

```
locals() -> dictionary containg local scope variables
```

这是一个例子：

```
#!/usr/bin/python3

from pprint import pprint

a = 10
b = 20

def foo():
    x = 30 # x and y are local variables
    y = 40

print("locals() = {0}".format(locals()))

pprint(locals()) # same as calling globals()

print('*' * 80)

print("locals() == globals()? ", locals() == globals())

print('*' * 80)

foo()
```

预期输出：

```
{'__builtins__': <module 'builtins' (built-in)>,
'__cached__': None,
'__doc__': None,
'__file__': 'module1.py',
'__loader__': <_frozen_importlib_external.SourceFileLoader object at
0x7fa18790a828>,
'__name__': '__main__',
'__package__': None,
'__spec__': None,
'a': 10,
'b': 20,
'foo': <function foo at 0x7fa1878752f0>,
'pprint': <function pprint at 0x7fa1878756a8>}

********************************************************************************

locals() == globals()? True

********************************************************************************

locals() = {'y': 40, 'x': 30}
```

试试看:

```
from pprint import pprint

a = 10
b = 20

def foo():
    x = 30 # x and y are local variables
    y = 40

print("locals() = {0}".format(locals()))

pprint(locals()) # same as calling globals()

print('*' * 80)

print("locals() == globals()? ", locals() == globals())

print('*' * 80)
```

```
19.
20. foo()
```

数据库访问

安装 Python MySQLdb

> 原文: https://thepythonguru.com/installing-mysqldb/

于 2020 年 1 月 7 日更新

MySQLdb 是用于使用 python 访问 MySQL 数据库的 api。 它建立在 MySQL C API 之上。

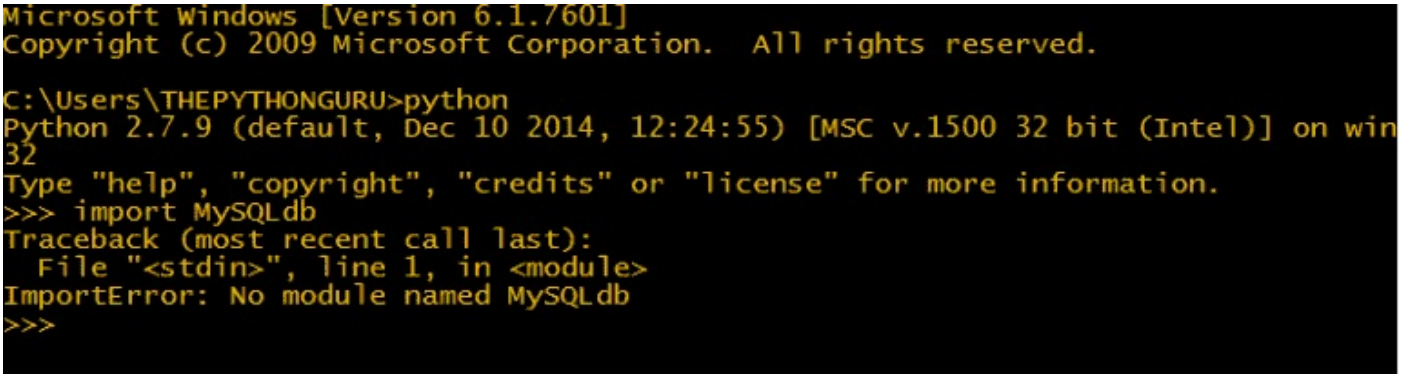
MySQLdb 尚不支持 python 3，它仅支持 python 2.4-2.7。 因此，您需要在本教程中使用 python 2。 我们将使用 python 2.7.9，您可以从此处下载。

安装 MySQLdb

在安装之前，必须首先检查系统上是否已经安装了 MySQLdb。 要测试打开命令提示符或终端，然后启动 python shell 并键入以下代码

```
1.  import MySQLdb
```

如果它像这样抛出 `ImportError` ：



那么您需要安装 MySQLdb。 否则，您已经安装了 MySQLdb。

如果您在 Windows 上，请下载 MySQLdb 并安装它。

在下一篇文章中，我们将讨论如何连接到访问数据库。

连接到数据库

> 原文: https://thepythonguru.com/connecting-to-the-database/

于 2020 年 1 月 7 日更新

在我们开始将数据库与 python 一起使用之前，必须先连接到数据库。 与 python 的数据库通信分为四个阶段：

1. 创建一个连接对象。
2. 创建一个游标对象以进行读取/写入。
3. 与数据库进行交互。
4. 关闭连接。

注意：

我们将使用世界 mysql 数据库，因此首先下载并导入数据库，如下所示：

首次登录到您的 mysql 服务器

```
1.  mysql -u root -p
```

此时将要求您输入密码，输入密码，然后按 `Enter` 键。

```
1.  source path/to/world.sql
```

连接到数据库

要连接到数据库，您需要使用 `connect()` 方法。

语法：

```
1.  MySQLdb.connect(
2.                  host="127.0.0.1",
3.                  user="username",
4.                  passwd="password",
5.                  db="database"
6.                )
```

成功后， `connect()` 方法将返回一个连接对象。 否则，将引发 `OperationalError` 异常。

```
from __future__ import print_function

import MySQLdb as my

db = my.connect(host="127.0.0.1",
user="root",
passwd="",
db="world"
)

print(db)
```

注意第一行 `import print_function from __future__` 中的 `import` 语句，这使我们能够在 Python 2 中使用 Python 3 版本的 `print()` 函数。

预期输出：

```
<_mysql.connection open to '127.0.0.1' at 21fe6f0>
```

创建游标对象

开始与数据库进行交互之前，需要创建游标对象。

语法： `connection_object.cursor()`

成功时，它将返回 `Cursor` 对象，否则将引发异常。

```
from __future__ import print_function

import MySQLdb as my

db = my.connect(host="127.0.0.1",
user="root",
passwd="",
db="world"
)

print(db)

```

```
cursor = db.cursor()

print(cursor)
```

预期输出：

```
<_mysql.connection open to '127.0.0.1' at 239e2c0>
<MySQLdb.cursors.Cursor object at 0x02444AD0>
```

与数据库交互

游标对象具有 `execute()` 方法，可用于执行 sql 查询。

语法： `cursor.execute(sql)`

成功后，它将返回受影响的行数，否则将引发异常。

```
from __future__ import print_function

import MySQLdb as my

db = my.connect(host="127.0.0.1",
user="root",
passwd="",
db="world"
)

print(db)

cursor = db.cursor()

number_of_rows = cursor.execute("select * from city");

print(number_of_rows)
```

预期输出：

```
4079
```

断开连接

与数据库进行交互之后，您需要关闭数据库连接以放弃资源。

语法： `connection_object.close()`

```
from __future__ import print_function

import MySQLdb as my

db = my.connect(host="127.0.0.1",
user="root",
passwd="",
db="world"
)

print(db)

cursor = db.cursor()

number_of_rows = cursor.execute("select * from city");

print(number_of_rows)

db.close()
```

现在您知道了如何与数据库连接，执行查询并关闭连接。 在下一篇文章中，我们讨论如何从表中获取行。

MySQLdb 获取结果

> 原文: https://thepythonguru.com/fetching-results/

于 2020 年 1 月 7 日更新

在上一篇文章中，我们看到了如何使用 `execute()` 方法执行 sql 查询。 `execute()` 方法返回受影响的行，但不返回结果。 要获取结果，我们使用游标对象的 `fetchall()` 方法。

语法： `cursor.fetchall()`

成功后，它将返回行的元组，其中每一行都是一个元组。

```
1.  from __future__ import print_function
2.
3.  import MySQLdb as my
4.
5.  db = my.connect(host="127.0.0.1",
6.  user="root",
7.  passwd="",
8.  db="world"
9.  )
10.
11.  cursor = db.cursor()
12.
13.  number_of_rows = cursor.execute("select * from city");
14.
15.  result = cursor.fetchall()
16.
17.  print(result)
18.
19.  db.close()
```

上面的代码将打印城市表中的所有行。

您也可以使用 `for` 循环遍历结果。

```
1.  from __future__ import print_function
2.
3.  import MySQLdb as my
```

```
4.
5.  db = my.connect(host="127.0.0.1",
6.  user="root",
7.  passwd="",
8.  db="world"
9.  )
10.
11.  cursor = db.cursor()
12.
13.  number_of_rows = cursor.execute("select * from city");
14.
15.  result = cursor.fetchall()
16.  for row in result:
17.    print(row)
18.
19.  db.close()
```

一些更多的例子。

示例 **1:**

```
1.  from __future__ import print_function
2.
3.  import MySQLdb as my
4.
5.  db = my.connect(host="127.0.0.1",
6.  user="root",
7.  passwd="",
8.  db="world"
9.  )
10.
11.  cursor = db.cursor()
12.
13.  id = 10
14.
15.  operation = ">"
16.
17.  sql = "select * from city where id {} {}".format(operation, id)
18.
19.  number_of_rows = cursor.execute(sql)
20.
21.  result = cursor.fetchall()
```

```
for row in result:
  print(row[0], row[1])

db.close()
```

示例 **2:**

```
from __future__ import print_function

import MySQLdb as my

db = my.connect(host="127.0.0.1",
user="root",
passwd="",
db="world"
)

cursor = db.cursor()
city = "%pur%"

sql = "select * from city where name like '{}'".format(city)

number_of_rows = cursor.execute(sql)

result = cursor.fetchall()
for row in result:
  print(row[0], row[1])

db.close()
```

在下一篇文章中，我们讨论如何将行插入数据库中。

插入行

> 原文: https://thepythonguru.com/inserting-rows/

于 2020 年 1 月 7 日更新

Insert 语句用于在 mysql 中插入记录。

语法: `INSERT INTO <some table> (<some column names>) VALUES("<some values>");`

示例 **1:**

```
1.  from __future__ import print_function
2.
3.  import MySQLdb as my
4.
5.  db = my.connect(host="127.0.0.1",
6.  user="root",
7.  passwd="",
8.  db="world"
9.  )
10.
11. cursor = db.cursor()
12.
13. sql = "insert into city VALUES(null, 'Mars City', 'MAC', 'MARC', 1233)"
14.
15. number_of_rows = cursor.execute(sql)
16. db.commit()   # you need to call commit() method to save
17.               # your changes to the database
18.
19. db.close()
```

该程序在城市表中插入一个新城市，注意对 `db.commit()` 的使用，该方法将您的更改保存到数据库中。

示例 **2:**

```
1.  from __future__ import print_function
2.
3.  import MySQLdb as my
```

```
4.
5.  db = my.connect(host="127.0.0.1",
6.  user="root",
7.  passwd="",
8.  db="world"
9.  )
10.
11.  cursor = db.cursor()
12.
13.  name = "Some new city"
14.  country_code = 'PSE'
15.  district = 'Someyork'
16.  population = 10008
17.
18.  sql = "insert into city VALUES(null, '%s', '%s', '%s', %d)" % \
19.   (name, country_code , district, population)
20.
21.  number_of_rows = cursor.execute(sql)
22.  db.commit()
23.
24.  db.close()
```

请注意，在第 18 行中使用了反斜杠（ `\` ）字符。 `\` 字符用于将 python 语句拆分为多行。

插入多行

要在表中插入多行，请使用游标对象的 `executemany()` 方法。

语法： `cursor_object.executemany(statement, arguments)`

`statement` ：包含要执行的查询的字符串。

`arguments` ：一个包含要在 `insert` 语句中使用的值的序列。

让我们举个例子。

```
1.  from __future__ import print_function
2.
3.  import MySQLdb as my
4.
5.  db = my.connect(host="127.0.0.1",
6.  user="root",
```

```
 7.  passwd="",
 8.  db="world"
 9.  )
10.
11.  cursor = db.cursor()
12.  name = "Some new city"
13.
14.  country_code = 'SNC'
15.
16.  district = 'Someyork'
17.
18.  population = 10008
19.
20.  data = [
21.  ('city 1', 'MAC', 'distrct 1', 16822),
22.  ('city 2', 'PSE', 'distrct 2', 15642),
23.  ('city 3', 'ZWE', 'distrct 3', 11642),
24.  ('city 4', 'USA', 'distrct 4', 14612),
25.  ('city 5', 'USA', 'distrct 5', 17672),
26.  ]
27.
28.  sql = "insert into city(name, countrycode, district, population)
29.  VALUES(%s, %s, %s, %s)"
30.
31.  number_of_rows = cursor.executemany(sql, data)
32.  db.commit()
33.
34.  db.close()
```

在下一篇文章中，我们讨论如何处理错误。

处理错误

> 原文: https://thepythonguru.com/handling-errors/

于 2020 年 1 月 7 日更新

与数据库交互是一个容易出错的过程，因此我们必须始终实现某种机制来优雅地处理错误。

MySQLdb 具有 `MySQLdb.Error` 异常，这是一个顶级异常，可用于捕获 `MySQLdb` 模块引发的所有数据库异常。

```
1.  from __future__ import print_function
2.
3.  import MySQLdb as my
4.
5.  try:
6.
7.      db = my.connect(host="127.0.0.1",
8.                      user="root",
9.                      passwd="",
10.                     db="world"
11.                   )
12.
13.     cursor = db.cursor()
14.
15.     sql = "select * from city"
16.     number_of_rows = cursor.execute(sql)
17.     print(number_of_rows)
18.     db.close()
19.
20. except my.Error as e:
21.     print(e)
22.
23. except :
24.     print("Unknown error occurred")
```

MySQLdb 中的两个主要错误

需要注意的是，MySQLdb 中有两类异常类：

1. `DatabaseError`
2. `InterfaceError`

---

1. **`DatabaseError`**：当数据处理中存在问题，sql 语法错误，mysql 内部问题时，引发此异常。 如果建立连接并且出现问题，则 `DatabaseError` 会捕获到它。
2. **`InterfaceError`**：当由于某种原因数据库连接失败时，MySQLdb 将引发 `InterfaceError`。 注意 `InterfaceError` 仅在与数据库连接存在内部问题时才引发，MySQLdb 不会因错误的数据库名称或密码而引发 `InterfaceError`。

`DatabaseError` 进一步分为 6 种类型：

1. `DataError`
2. `InternalError`
3. `IntegrityError`
4. `OperationalError`
5. `NotSupportedError`
6. `ProgrammingError`

---

1. **`DataError`**：当数据处理出现问题时，例如除以零，范围的数值，MySQLdb 会引发此错误。
2. **`InternalError`**：当 MySQL 数据库本身存在一些内部错误时，引发此异常。 例如无效的游标，事务不同步等。
3. **`IntegrityError`**：当外键检查失败时，引发此异常。
4. **`OperationalError`**：对于不受程序员控制的事情，会引发此异常。 例如，意外断开连接，内存分配错误等，所选数据库不存在。
5. **`NotSupportedError`**：当存在不支持的方法或 api 时，引发此异常。
6. **`ProgrammingError`**：引发此编程错误。 例如找不到表，mysql 语法错误，指定的参数数量错误等。

```
1. from __future__ import print_function
2.
3. import MySQLdb as my
4.
5. try:
6.
7.     db = my.connect(host="127.0.0.1",
```

```
                    user="root",
                    passwd="",
                    db="world"
                   )

    cursor = db.cursor()

    sql = "select * from city"
    number_of_rows = cursor.execute(sql)
    print(number_of_rows)
    db.close()

except my.DataError as e:
    print("DataError")
    print(e)

except my.InternalError as e:
    print("InternalError")
    print(e)

except my.IntegrityError as e:
    print("IntegrityError")
    print(e)

except my.OperationalError as e:
    print("OperationalError")
    print(e)

except my.NotSupportedError as e:
    print("NotSupportedError")
    print(e)

except my.ProgrammingError as e:
    print("ProgrammingError")
    print(e)

except :
    print("Unknown error occurred")
```

在下一篇文章中，我们讨论如何从数据库中获取特定的行数。

使用 `fetchone()` 和 `fetchmany()` 获取记录

> 原文: https://thepythonguru.com/fetching-records-using-fetchone-and-fetchmany/

于 2020 年 1 月 7 日更新

到目前为止，我们一直在使用游标对象的 `fetchall()` 方法来获取记录。 一次性访问所有记录的过程并非十分有效。 结果，MySQLdb 具有游标对象的 `fetchone()` 和 `fetchmany()` 方法来更有效地获取记录。

方法	描述
`fetchone()`	此方法以元组形式返回一个记录，如果没有更多记录，则返回 `None` 。
`fetchmany(number_of_records)`	此方法接受要提取的记录数，并返回元组，其中每个记录本身就是一个元组。 如果没有更多记录，则返回一个空元组。

使用 `fetchone()`

```
from __future__ import print_function

import MySQLdb as my

try:

    db = my.connect(host="127.0.0.1",
                    user="root",
                    passwd="",
                    db="world"
                    )

    cursor = db.cursor()

    sql = "select * from city where id < 10"
    number_of_rows = cursor.execute(sql)

    print(cursor.fetchone())  # fetch the first row only

    db.close()

```

```
except my.DataError as e:
    print("DataError")
    print(e)

except my.InternalError as e:
    print("InternalError")
    print(e)

except my.IntegrityError as e:
    print("IntegrityError")
    print(e)

except my.OperationalError as e:
    print("OperationalError")
    print(e)

except my.NotSupportedError as e:
    print("NotSupportedError")
    print(e)

except my.ProgrammingError as e:
    print("ProgrammingError")
    print(e)

except :
    print("Unknown error occurred")
```

使用 `fetchone()` 遍历结果

```
from __future__ import print_function

import MySQLdb as my

try:

    db = my.connect(host="127.0.0.1",
                    user="root",
                    passwd="",
                    db="world"
                    )
```

```
12.
13.     cursor = db.cursor()
14.
15.     sql = "select * from city where id < 10"
16.     number_of_rows = cursor.execute(sql)
17.
18.     while True:
19.         row = cursor.fetchone()
20.         if row == None:
21.             break
22.         print(row)
23.
24.     db.close()
25.
26. except my.DataError as e:
27.     print("DataError")
28.     print(e)
29.
30. except my.InternalError as e:
31.     print("InternalError")
32.     print(e)
33.
34. except my.IntegrityError as e:
35.     print("IntegrityError")
36.     print(e)
37.
38. except my.OperationalError as e:
39.     print("OperationalError")
40.     print(e)
41.
42. except my.NotSupportedError as e:
43.     print("NotSupportedError")
44.     print(e)
45.
46. except my.ProgrammingError as e:
47.     print("ProgrammingError")
48.     print(e)
49.
50. except :
51.     print("Unknown error occurred")
```

使用 `fetchmany()`

```
1.  from __future__ import print_function
2.
3.  import MySQLdb as my
4.
5.  try:
6.
7.      db = my.connect(host="127.0.0.1",
8.                      user="root",
9.                      passwd="",
10.                     db="world"
11.                    )
12.
13.     cursor = db.cursor()
14.
15.     sql = "select * from city where id < 10"
16.     number_of_rows = cursor.execute(sql)
17.
18.     print(cursor.fetchmany(2))  # fetch first 2 rows only
19.
20.     db.close()
21.
22. except my.DataError as e:
23.     print("DataError")
24.     print(e)
25.
26. except my.InternalError as e:
27.     print("InternalError")
28.     print(e)
29.
30. except my.IntegrityError as e:
31.     print("IntegrityError")
32.     print(e)
33.
34. except my.OperationalError as e:
35.     print("OperationalError")
36.     print(e)
37.
38. except my.NotSupportedError as e:
39.     print("NotSupportedError")
```

```
    print(e)

except my.ProgrammingError as e:
    print("ProgrammingError")
    print(e)

except :
    print("Unknown error occurred")
```

使用 `fetchmany()` 遍历结果

```
from __future__ import print_function

import MySQLdb as my

try:

    db = my.connect(host="127.0.0.1",
                    user="root",
                    passwd="",
                    db="world"
                   )

    cursor = db.cursor()

    sql = "select * from city where id < 10"
    number_of_rows = cursor.execute(sql)

    while True:
        two_rows = cursor.fetchmany(2)
        if not two_rows:
            break
        print(two_rows)

    db.close()

except my.DataError as e:
    print("DataError")
    print(e)

```

```
except my.InternalError as e:
    print("InternalError")
    print(e)

except my.IntegrityError as e:
    print("IntegrityError")
    print(e)

except my.OperationalError as e:
    print("OperationalError")
    print(e)

except my.NotSupportedError as e:
    print("NotSupportedError")
    print(e)

except my.ProgrammingError as e:
    print("ProgrammingError")
    print(e)

except :
    print("Unknown error occurred")
```

常见做法

Python: 如何读取和写入文件

> 原文: https://thepythonguru.com/python-how-to-read-and-write-files/

于 2020 年 1 月 7 日更新

在本文中，我们将学习如何在 Python 中读取和写入文件。

处理文件包括以下三个步骤：

1. 打开文件
2. 执行读或写操作
3. 关闭文件

让我们详细了解每个步骤。

文件类型

有两种类型的文件：

1. 文本文件
2. 二进制文件

文本文件只是使用 utf-8，latin1 等编码存储字符序列的文件，而对于二进制文件，数据以与计算机内存相同的格式存储。

以下是一些文本和二进制文件示例：

文本文件：Python 源代码，HTML 文件，文本文件，降价文件等。

二进制文件：可执行文件，图像，音频等

重要的是要注意，在磁盘内部，两种类型的文件都以 1 和 0 的顺序存储。 唯一的区别是，当打开文本文件时，将使用与编码相同的编码方案对数据进行解码。但是，对于二进制文件，不会发生这种情况。

打开文件 - `open()` 函数

`open()` 内置函数用于打开文件。 其语法如下：

```
1.  open(filename, mode) -> file object
```

成功时， `open()` 返回文件对象。 如果失败，它将引发 `IOError` 或它的子类。

参数	描述
`filename`	要打开的文件的绝对或相对路径。
`mode`	（可选）模式是一个字符串，表示处理模式（即读取，写入，附加等）和文件类型。

以下是 `mode` 的可能值。

模式	描述
`r`	打开文件进行读取（默认）。
`w`	打开文件进行写入。
`a`	以附加模式打开文件，即在文件末尾添加新数据。
`r+`	打开文件以进行读写
`x`	仅在尚不存在的情况下，打开文件进行写入。

我们还可以将 `t` 或 `b` 附加到模式字符串以指示将要使用的文件的类型。 `t` 用于文本文件， `b` 用于二进制文件。 如果未指定，则默认为 `t` 。

`mode` 是可选的，如果未指定，则该文件将作为文本文件打开，仅供读取。

这意味着对 `open()` 的以下三个调用是等效的：

```
1.  # open file todo.md for reading in text mode
2.
3.  open('todo.md')
4.
5.  open('todo.md', 'r')
6.
7.  open('todo.md', 'rt')
```

请注意，在读取文件之前，该文件必须已经存在，否则 `open()` 将引发 `FileNotFoundError` 异常。 但是，如果打开文件进行写入（使用 `w` ， `a` 或 `r+` 之类的模式），Python 将自动为您创建文件。 如果文件已经存在，则其内容将被删除。 如果要防止这种情况，请以 `x` 模式打开文件。

关闭文件 - `close()` 方法

处理完文件后，应将其关闭。 尽管程序结束时该文件会自动关闭，但是这样做仍然是一个好习惯。 无法在大型程序中关闭文件可能会出现问题，甚至可能导致程序崩溃。

要关闭文件，请调用文件对象的 `close()` 方法。 关闭文件将释放与其相关的资源，并将缓冲区中的数据刷新到磁盘。

文件指针

通过 `open()` 方法打开文件时。 操作系统将指向文件中字符的指针关联。 文件指针确定从何处进行读取和写入操作。 最初，文件指针指向文件的开头，并随着我们向文件读取和写入数据而前进。 在本文的后面，我们将看到如何确定文件指针的当前位置，并使用它来随机访问文件的各个部分。

使用 `read()` ， `readline()` 和 `readlines()` 读取文件

要读取数据，文件对象提供以下方法：

方法	参数
`read([n])`	从文件读取并返回 `n` 个字节或更少的字节（如果没有足够的字符读取）作为字符串。 如果未指定 `n` ，它将以字符串形式读取整个文件并将其返回。
`readline()`	读取并返回字符，直到以字符串形式到达行尾为止。
`readlines()`	读取并返回所有行作为字符串列表。

当到达文件末尾（EOF）时， `read()` 和 `readline()` 方法返回一个空字符串，而 `readlines()` 返回一个空列表（ `[]` ）。

这里有些例子：

`poem.txt`

```
The caged bird sings
with a fearful trill
of things unknown
but longed for still
```

示例 **1** ：使用 `read()`

```
>>>
>>> f = open("poem.txt", "r")
>>>
>>> f.read(3) # read the first 3 characters
'The'
>>>
>>> f.read() # read the remaining characters in the file.
```

```
    ' caged bird sings\nwith a fearful trill\nof things unknown\nbut longed for
 8. still\n'
 9. >>>
10. >>> f.read() # End of the file (EOF) is reached
11. ''
12. >>>
13. >>> f.close()
14. >>>
```

示例 **2** ：使用 `readline()`

```
 1. >>>
 2. >>> f = open("poem.txt", "r")
 3. >>>
 4. >>> f.read(4) # read first 4 characters
 5. 'The '
 6. >>>
 7. >>> f.readline() # read until the end of the line is reached
 8. 'caged bird sings\n'
 9. >>>
10. >>> f.readline() # read the second line
11. 'with a fearful trill\n'
12. >>>
13. >>> f.readline() # read the third line
14. 'of things unknown\n'
15. >>>
16. >>> f.readline() # read the fourth line
17. 'but longed for still'
18. >>>
19. >>> f.readline() # EOF reached
20. ''
21. >>>
22. >>> f.close()
23. >>>
```

示例 **3** ：使用 `readlines()`

```
1. >>>
2. >>> f = open("poem.txt", "r")
3. >>>
4. >>> f.readlines()
```

```
    ['The caged bird sings\n', 'with a fearful trill\n', 'of things unknown\n',
5.  'but longed for still\n']
6.  >>>
7.  >>> f.readlines() # EOF reached
8.  []
9.  >>>
10. >>> f.close()
11. >>>
```

批量读取文件

`read()`（不带参数）和 `readlines()` 方法立即将所有数据读入内存。 因此，请勿使用它们读取大文件。

更好的方法是使用 `read()` 批量读取文件，或使用 `readline()` 逐行读取文件，如下所示：

示例：读取文件块

```
1.  >>>
2.  >>> f = open("poem.txt", "r")
3.  >>>
4.  >>> chunk = 200
5.  >>>
6.  >>> while True:
7.  ...     data = f.read(chunk)
8.  ...     if not data:
9.  ...         break
10. ...     print(data)
11. ...
12. The caged bird sings
13. with a fearful trill
14. of things unknown
15. but longed for still
16. >>>
```

示例：逐行读取文件

```
1.  >>>
2.  >>> f = open("poem.txt", "r")
3.  >>>
4.  >>> while True:
```

```
...     line = f.readline()
...     if not line:
...         break
...     print(line)
...
The caged bird sings
with a fearful trill
of things unknown
but longed for still
>>>
```

除了使用 `read()` （带有参数）或 `readline()` 方法之外，您还可以使用文件对象一次遍历一行的文件内容。

```
>>>
>>> f = open("poem.txt", "r")
>>>
>>> for line in f:
...     print(line, end="")
...
The caged bird sings
with a fearful trill
of things unknown
but longed for still
>>>
```

该代码与前面的示例等效，但是更加简洁，易读且易于键入。

警告：

提防 `readline()` 方法，如果您在打开没有任何换行符的大文件时遇到不幸，那么 `readline()` 并不比 `read()` 好（无参数）。 当您使用文件对象作为迭代器时，也是如此。

使用 `write()` 和 `writelines()` 写入数据

为了写入数据，文件对象提供了以下两种方法：

方法	描述
`write(s)`	将字符串 `s` 写入文件并返回写入的数字字符。
`writelines(s)`	将序列 `s` 中的所有字符串写入文件。

以下是示例：

```
>>>
>>> f = open("poem_2.txt", "w")
>>>
>>> f.write("When I think about myself, ")
26
>>> f.write("I almost laugh myself to death.")
31
>>> f.close() # close the file and flush the data in the buffer to the disk
>>>
>>>
>>> f = open("poem_2.txt", "r") # open the file for reading
>>>
>>> data = f.read() # read entire file
>>>
>>> data
'When I think about myself, I almost laugh myself to death.'
>>>
>>> print(data)
When I think about myself, I almost laugh myself to death.
>>>
>>> f.close()
>>>
```

请注意，与 `print()` 函数不同， `write()` 方法不会在行尾添加换行符（ `\n` ）。 如果需要换行符，则必须手动添加它，如下所示：

```
>>>
>>>
>>> f = open("poem_2.txt", "w")
>>>
>>> f.write("When I think about myself, \n") # notice newline
27
>>> f.write("I almost laugh myself to death.\n") # notice newline
32
>>>
>>> f.close()
>>>
>>>
>>> f = open("poem_2.txt", "r") # open the file again
>>>
```

```
>>> data = f.read() # read the entire file
>>>
>>> data
'When I think about myself, \nI almost laugh myself to death.\n'
>>>
>>> print(data)
When I think about myself,
I almost laugh myself to death.

>>>
>>>
```

您还可以使用 `print()` 函数将换行符附加到该行，如下所示：

```
>>>
>>> f = open("poem_2.txt", "w")
>>>
>>> print("When I think about myself, ", file=f)
>>>
>>> print("I almost laugh myself to death.", file=f)
>>>
>>> f.close()
>>>
>>>
>>> f = open("poem_2.txt", "r") # open the file again
>>>
>>> data = f.read()
>>>
>>> data
'When I think about myself, \nI almost laugh myself to death.\n'
>>>
>>> print(data)
When I think about myself,
I almost laugh myself to death.

>>>
>>>
```

这是 `writelines()` 方法的示例。

```
>>>
>>> lines = [
```

```
... "Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod",
... "tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,"
... "quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo",
... "consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse",
... "cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non",
... "proident, sunt in culpa qui officia deserunt mollit anim id est laborum."
... ]
>>>
>>>
>>> f = open("lorem.txt", "w")
>>>
>>> f.writelines(lines)
>>>
>>> f.close()
>>>
```

`writelines()` 方法在内部调用 `write()` 方法。

```
def writelines(self, lines):
    self._checkClosed()
    for line in lines:
        self.write(line)
```

这是另一个以附加模式打开文件的示例。

```
>>>
>>> f = open("poem_2.txt", "a")
>>>
>>> f.write("\nAlone, all alone. Nobody, but nobody. Can make it out here alone.")
65
>>> f.close()
>>>
>>> data = open("poem_2.txt").read()
>>> data
'When I think about myself, \nI almost laugh myself to death.\n\nAlone, all alone. Nobody, but nobody. Can make it out here alone.'
>>>
>>> print(data)
```

```
When I think about myself,
I almost laugh myself to death.

Alone, all alone. Nobody, but nobody. Can make it out here alone.
>>>
```

假设文件 `poem_2.txt` 对于使用非常重要，并且我们不希望其被覆盖。 为了防止在 `x` 模式下打开文件

```
>>>
>>> f = open("poem_2.txt", "x")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'poem.txt'
>>>
```

如果 `x` 模式不存在，则仅打开该文件进行写入。

缓冲和刷新

缓冲是在将数据移到新位置之前临时存储数据的过程。

对于文件，数据不会立即写入磁盘，而是存储在缓冲存储器中。

这样做的基本原理是，将数据写入磁盘需要花费时间，而不是将数据写入物理内存。 想象一下，每当调用 `write()` 方法时一个程序正在写入数据。 这样的程序将非常慢。

当我们使用缓冲区时，仅当缓冲区已满或调用 `close()` 方法时，才将数据写入磁盘。 此过程称为刷新输出。 您也可以使用文件对象的 `flush()` 方法手动刷新输出。 请注意， `flush()` 仅将缓冲的数据保存到磁盘。 它不会关闭文件。

`open()` 方法提供了一个可选的第三个参数来控制缓冲区。 要了解更多信息，请访问官方文档。

读写二进制数据

通过将 `b` 附加到模式字符串来完成读写二进制文件。

在 Python 3 中，二进制数据使用称为 `bytes` 的特殊类型表示。

`bytes` 类型表示介于 0 和 255 之间的数字的不可变序列。

让我们通过阅读 `poem.txt` 文件来创建诗歌的二进制版本。

```
>>>
>>> binary_poem = bytes(open("poem.txt").read(), encoding="utf-8")
>>>
>>> binary_poem
b'The caged bird sings\nwith a fearful trill\nof things unknown\nbut longed for
still'
>>>
>>>
>>> binary_poem[0] # ASCII value of character T
84
>>> binary_poem[1] # ASCII value of character h
104
>>>
```

请注意，索引 `bytes` 对象将返回 `int` 。

让我们将二进制诗写在一个新文件中。

```
>>>
>>> f = open("binary_poem", "wb")
>>>
>>> f.write(binary_poem)
80
>>>
>>> f.close()
>>>
```

现在，我们的二进制诗已写入文件。 要读取它，请以 `rb` 模式打开文件。

```
>>>
>>> f = open("binary_poem", "rb")
>>>
>>> data = f.read()
>>>
>>> data
b'The caged bird sings\nwith a fearful trill\nof things unknown\nbut longed for
still'
>>>
>>> print(data)
```

```
    b'The caged bird sings\nwith a fearful trill\nof things unknown\nbut longed for
10. still'
11. >>>
12. >>> f.close()
13. >>>
```

重要的是要注意，在我们的情况下，二进制数据碰巧包含可打印的字符，例如字母，换行符等。但是，在大多数情况下并非如此。 这意味着对于二进制数据，由于文件中可能没有换行符，因此我们无法可靠地使用 `readline()` 和文件对象（作为迭代器）来读取文件的内容。 读取二进制数据的最佳方法是使用 `read()` 方法分块读取它。

```
 1. >>>
 2. >>> # Just as with text files, you can read (or write) binary files in chunks.
 3. >>>
 4. >>> f = open("binary_poem", "rb")
 5. >>>
 6. >>> chunk = 200
 7. >>>
 8. >>> while True:
 9. ...     data = f.read(chunk)
10. ...     if not data:
11. ...         break
12. ...     print(data)
13. ...
    b'The caged bird sings\nwith a fearful trill\nof things unknown\nbut longed for
14. still'
15. >>>
16. >>>
```

使用 `fseek()` 和 `ftell()` 的随机访问

在本文的前面，我们了解到打开文件时，系统将一个指针与它相关联，该指针确定从哪个位置进行读取或写入。

到目前为止，我们已经线性地读写文件。 但是也可以在特定位置进行读写。 为此，文件对象提供了以下两种方法：

方法	描述
`tell()`	返回文件指针的当前位置。
`seek(offset,`	将文件指针移动到给定的 `offset` 。 `offset` 引用字节计数， `whence` 确定 `offset` 将相对于文件指针移动的位置。 `whence` 的默认值为 `0`，这意味着 `offset` 将使文件指针从文

`[whence=0])`	件的开头移开。 如果 `wherece` 设置为 `1` 或 `2` ，则偏移量将分别将文件的指针从当前位置或文件的末尾移动。

现在让我们举一些例子。

```
>>>
>>> ###### binary poem at a glance #######
>>>
>>> for i in open("binary_poem", "rb"):
...     print(i)
...
b'The caged bird sings\n'
b'with a fearful trill\n'
b'of things unknown\n'
b'but longed for still'
>>>
>>> f.close()
>>>
>>> ######################################
>>>
>>> f = open('binary_poem', 'rb') # open binary_poem file for reading
>>>
>>> f.tell() # initial position of the file pointer
0
>>>
>>> f.read(5) # read 5 bytes
b'The c'
>>>
>>> f.tell()
5
>>>
```

读取 5 个字符后，文件指针现在位于字符 `a` （用字 `caged` 表示）。 因此，下一个读取（或写入）操作将从此处开始。

```
>>>
>>>
>>> f.read()
b'aged bird sings\nwith a fearful trill\nof things unknown\nbut longed for
still'
>>>
>>> f.tell()
80
```

```
>>>
>>> f.read() # EOF reached
b''
>>>
>>> f.tell()
80
>>>
```

现在，我们已到达文件末尾。 此时，我们可以使用 `fseek()` 方法将文件指针后退到文件的开头，如下所示：

```
>>>
>>> f.seek(0) # rewind the file pointer to the beginning, same as seek(0, 0)
0
>>>
>>> f.tell()
0
>>>
```

文件指针现在位于文件的开头。 从现在开始的所有读取和写入操作将从文件的开头再次进行。

```
>>>
>>> f.read(14) # read the first 14 characters
b'The caged bird'
>>>
>>>
>>> f.tell()
14
>>>
```

要将文件指针从当前位置从 12 个字节向前移动，请按以下步骤操作： `seek()` ：

```
>>>
>>> f.seek(12, 1)
26
>>>
>>> f.tell()
26
>>>
>>>
```

文件指针现在位于字符 `a` （在单词 `with` 之后），因此将从此处进行读取和写入操作。

```
>>>
>>>
>>> f.read(15)
b'a fearful trill'
>>>
>>>
```

我们还可以向后移动文件指针。 例如，以下对 `seek()` 的调用将文件指针从当前位置向后移 13 个字节。

```
>>>
>>> f.seek(-13, 1)
28
>>>
>>> f.tell()
28
>>>
>>> f.read(7)
b'fearful'
>>>
```

假设我们要读取文件的最后 16 个字节。 为此，将文件指针相对于文件末尾向后移 16 个字节。

```
>>>
>>> f.seek(-16, 2)
64
>>>
>>> f.read()
b'longed for still'
>>>
```

`fseek()` 的 `whence` 自变量的值在 `os` 模块中也定义为常量。

值	常量
`0`	`SEEK_SET`
`1`	`SEEK_CUR`
`2`	`SEEK_END`

`with` 语句

使用 `with` 语句可使我们在完成处理后自动关闭文件。 其语法如下：

```
with expression as variable:
    # do operations on file here.
```

必须像 `for` 循环一样，将 `with` 语句内的语句缩进相等，否则将引发 `SyntaxError` 异常。

这是一个例子：

```
>>>
>>> with open('poem.txt') as f:
...     print(f.read()) # read the entire file
...
The caged bird sings
with a fearful trill
of things unknown
but longed for still
>>>
```

Python: 如何读取和写入 CSV 文件

> 原文: https://thepythonguru.com/python-how-to-read-and-write-csv-files/

于 2020 年 1 月 7 日更新

什么是 CSV 文件？

CSV（逗号分隔值）是应用用来生成和使用数据的通用数据交换格式。 其他一些众所周知的数据交换格式是 XML，HTML，JSON 等。

CSV 文件是一个简单的文本文件，其中的每一行都包含用逗号分隔的值（或字段）列表。

尽管术语“逗号”本身出现在格式名称中，但是您会遇到 CSV 文件，其中使用制表符（ `\t` ）或管道（ `|` ）或任何其他可用作定界符的字符来分隔数据。

CSV 文件的第一行表示标题，该标题包含文件中的列名列表。 标头是可选的，但强烈建议使用。

CSV 文件通常用于表示表格数据。 例如，考虑下表：

ID	名称	国家代码	区域	人口
1	Kabul	AFG	Kabol	1780000
2	Qandahar	AFG	Qandahar	237500
3	Herat	AFG	Herat	186800
4	Mazar-e-Sharif	AFG	Balkh	127800
5	Amsterdam	NLD	Noord-Holland	731200

上表可以使用 CSV 格式表示如下：

```
1. "ID","Name","CountryCode","District","Population"
2. "1","Kabul","AFG","Kabol","1780000"
3. "2","Qandahar","AFG","Qandahar","237500"
4. "3","Herat","AFG","Herat","186800"
5. "4","Mazar-e-Sharif","AFG","Balkh","127800"
6. "5","Amsterdam","NLD","Noord-Holland","731200"
```

如果 CSV 文件中的值包含逗号，则必须将其用双引号引起来。 例如：

名称	年龄	地址

Jerry	10	2776 McDowell Street, Nashville, Tennessee
Tom	20	3171 Jessie Street, Westerville, Ohio
Mike	30	1818 Sherman Street, Hope, Kansas

要将逗号保留在“地址”字段中，请用双引号将其引起来，如下所示：

```
1. Name,Age,Address
2. Jerry,10,"2776 McDowell Street, Nashville, Tennessee"
3. Tom,20,"3171 Jessie Street, Westerville, Ohio"
4. Mike,30,"1818 Sherman Street, Hope, Kansas"
```

同样，如果您在字段中嵌入了双引号，则必须使用另一个双引号字符对其进行转义。 否则，将无法正确解释它们。 例如：

ID	用户	评论
1	Bob	John said “Hello World”
2	Tom	“The Magician”

要保留注释字段中的双引号，请使用两个双引号。

```
1. Id,User,Comment
2. 1,Bob,"John said ""Hello World"""
3. 2,Tom,""The Magician""
```

重要的是要注意，CSV 格式尚未完全标准化。 因此，我们刚才提到的规则不是通用的。 有时，您会遇到 CSV 文件，它们以不同的方式表示字段。

幸运的是，为了使我们更轻松，Python 提供了 `csv` 模块。

在开始读写 CSV 文件之前，您应该对一般文件的使用方法有很好的了解。 如果需要复习，请考虑阅读如何在 Python 中读写文件。

`csv` 模块用于读取和写入文件。 它主要提供以下类和函数：

1. `reader()`
2. `writer()`
3. `DictReader()`
4. `DictWriter()`

让我们从 `reader()` 函数开始。

使用 `reader()` 读取 CSV 文件

`reader()` 函数接受一个文件对象，并返回一个 `_csv.reader` 对象，该对象可用于遍历 CSV 文件的内容。 `reader()` 函数的语法如下：

语法： `reader(fileobj [, dialect='excel' [, **fmtparam] ]) -> _csv.reader`

参数	描述
`fileobj`	（必需）它引用文件对象
`dialect`	（可选）方言是指格式化 CSV 文档的不同方式。 默认情况下， `csv` 模块使用与 Microsoft Excel 相同的格式。 我们将在本文后面详细讨论方言。
`fmtparam`	（可选）它是指一组用于自定义方言的关键字参数（请参阅下一节）。

假设我们有以下 CSV 文件：

`employees.csv`

```
id,name,email,age,designation
1,John,john@mail.com,24,programmer
2,Bob,bob@mail.com,34,designer
3,Mary,mary@mail.com,43,sales
```

以下是读取此 CSV 文件的方法：

```
import csv

with open('employees.csv', 'rt') as f:
    csv_reader = csv.reader(f)

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

预期输出：

```
['id', 'name', 'email', 'age', 'designation']
['1', 'John', 'john@mail.com', '24', 'programmer']
['2', 'Bob', 'bob@mail.com', '34', 'designer']
['3', 'Mary', 'mary@mail.com', '43', 'sales']
```

请注意，CSV 文件中的每一行都作为字符串列表返回。

要从某些字段获取数据，可以使用索引。 例如：

```
import csv
```

```
2.
3.  with open('employees.csv', 'rt') as f:
4.      csv_reader = csv.reader(f)
5.
6.      for line in csv_reader:
7.          print(line[0], line[1], line[2])
```

预期输出：

```
1.  id name email
2.  1 John john@mail.com
3.  2 Bob bob@mail.com
4.  3 Mary mary@mail.com
```

如果要跳过标题，请调用 `_csv.reader` 对象上的 `next()` 内置函数，然后照常遍历其余行。

```
1.  import csv
2.
3.  with open('employees.csv', 'rt') as f:
4.      csv_reader = csv.reader(f)
5.
6.      next(csv_reader) # skip the heading
7.
8.      for line in csv_reader:
9.          print(line[0], line[1], line[2])
```

预期输出：

```
1.  1 John john@mail.com
2.  2 Bob bob@mail.com
3.  3 Mary mary@mail.com
```

自定义 `reader()`

默认情况下， `csv` 模块根据 Microsoft excel 使用的格式工作，但是您也可以使用方言定义自己的格式。

以下是一些其他参数，您可以将其传递给 `reader()` 函数以自定义其工作方式。

- `delimiter` -是指用于分隔 CSV 文件中的值（或字段）的字符。 默认为逗号（ `,` ）。

- `skipinitialspace` -控制分隔符后面的空格的解释方式。 如果为 `True` ，则将删除初始空格。 默认为 `False` 。
- `lineterminator` -指用于终止行的字符序列。 默认为 `\r\n` 。
- `quotechar` -指的是如果字段中出现特殊字符（如定界符），则将用于引用值的单个字符串。 默认为 `"` 。
- `quoting` -控制引号何时应由作者生成或由读者识别。 它可以采用以下常量之一：
  - `csv.QUOTE_MINIMAL` 表示仅在需要时（例如，当字段包含 `quotechar` 或定界符时）添加引号。 这是默认值。
  - `csv.QUOTE_ALL` 表示所有内容的引用，无论字段类型如何。
  - `csv.QUOTE_NONNUMERIC` 表示除整数和浮点数外的所有内容。
  - `csv.QUOTE_NONE` 表示在输出中不引用任何内容。 但是，在阅读报价时，字段值周围会包含引号。
- `escapechar` -引用设置为 `QUOTE_NONE` 时，用于转义分隔符的单字符字符串。 默认为 `None` 。
- `doublequote` -控制字段内引号的处理。 当 `True` 时，两个连续的引号在读取期间被解释为一个，而在写入时，嵌入在数据中的每个引号字符都被写入两个引号。 让我们通过一些示例来更好地理解这些参数的工作方式：

定界符参数

`employee_pipe.csv`

```
id|name|email|age|designation
1|John|john@mail.com|24|programmer
2|Bob|bob@mail.com|34|designer
3|Mary|mary@mail.com|43|sales
```

该文件使用竖线（ `|` ）字符作为分隔符。 以下是读取此 CSV 文件的方法：

```
import csv

with open('employees.csv', 'rt') as f:
    csv_reader = csv.reader(f, delimiter='|')

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

预期输出：

```
['id', 'name', 'email', 'age', 'designation']
```

```
['1', 'John', 'john@mail.com', '24', 'programmer']
['2', 'Bob', 'bob@mail.com', '34', 'designer']
['3', 'Mary', 'mary@mail.com', '43', 'sales']
```

`skipinitialspace` 参数

`Basketball_players.csv`

```
"Name", "Team", "Position", "Height(inches)", "Weight(lbs)", "Age"
"Adam Donachie", "BAL", "Catcher", 74, 180, 22.99
"Paul Bako", "BAL", "Catcher", 74, 215, 34.69
"Ramon Hernandez", "BAL", "Catcher", 72, 210, 30.78
"Kevin Millar", "BAL", "First Baseman", 72, 210, 35.43
"Chris Gomez", "BAL", "First Baseman", 73, 188, 35.71
"Brian Roberts", "BAL", "Second Baseman", 69, 176, 29.39
"Miguel Tejada", "BAL", "Shortstop", 69, 209, 30.77
"Melvin Mora", "BAL", "Third Baseman", 71, 200, 35.07
```

该 CSV 文件在逗号（ `,` ）后包含空格。 要正确读取此 CSV 文件，请将 `skipinitialspace` 设置为 `True` ，如下所示：

```
import csv

with open('baseball_players.csv', 'rt') as f:
    csv_reader = csv.reader(f, skipinitialspace=True)

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

预期输出：

```
['Name', 'Team', 'Position', 'Height(inches)', 'Weight(lbs)', 'Age']
['Adam Donachie', 'BAL', 'Catcher', '74', '180', '22.99']
['Paul Bako', 'BAL', 'Catcher', '74', '215', '34.69']
['Ramon Hernandez', 'BAL', 'Catcher', '72', '210', '30.78']
['Kevin Millar', 'BAL', 'First Baseman', '72', '210', '35.43']
['Chris Gomez', 'BAL', 'First Baseman', '73', '188', '35.71']
['Brian Roberts', 'BAL', 'Second Baseman', '69', '176', '29.39']
['Miguel Tejada', 'BAL', 'Shortstop', '69', '209', '30.77']
['Melvin Mora', 'BAL', 'Third Baseman', '71', '200', '35.07']
```

`quotechar` 参数

`address.csv`

```
Name, Age, Address
Jerry, 44, '2776 McDowell Street, Nashville, Tennessee'
Tom, 21, '3171 Jessie Street, Westerville, Ohio'
Mike, 32, '1818 Sherman Street, Hope, Kansas'
```

此文件中有两件事需要注意。 首先，使用单引号（ `'` ）而不是 `"` 双引号（这是默认值）包装地址字段。 其次，逗号（ `,` ）后面有空格。

如果尝试在不更改引号的情况下读取此文件，则将获得以下输出：

```
import csv

with open('addresses.csv', 'rt') as f:
    csv_reader = csv.reader(f, skipinitialspace=True)

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

预期输出：

```
['Name', 'Age', 'Address']
['Jerry', '44', "'2776 McDowell Street", 'Nashville', "Tennessee'"]
['Tom', '21', "'3171 Jessie Street", 'Westerville', "Ohio'"]
['Mike', '32', "'1818 Sherman Street", 'Hope', "Kansas'"]
```

请注意，地址分为三个字段，这当然是不正确的。 要解决此问题，只需使用 `quotechar` 参数将引号字符更改为单引号（ `'` ）：

```
import csv

with open('housing.csv', 'rt') as f:
    csv_reader = csv.reader(f, skipinitialspace=True, quotechar="'")

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

预期输出：

```
['Name', 'Age', 'Address']
['Jerry', '44', '2776 McDowell Street, Nashville, Tennessee']
['Tom', '21', '3171 Jessie Street, Westerville, Ohio']
['Mike', '32', '1818 Sherman Street, Hope, Kansas']
```

`escapechar` 参数

`comments.csv`

```
Id, User, Comment
1, Bob, "John said \"Hello World\""
2, Tom, "\"The Magician\""
3, Harry, "\"walk around the corner\" she explained to the child"
4, Louis, "He said, \"stop pulling the dog's tail\""
```

此文件使用反斜杠（ `\` ）字符转义嵌入的双引号。 但是，默认情况下，默认的 `csv` 模块使用双引号字符转义双引号字符。

如果尝试使用默认选项读取此文件，则将获得如下输出：

```
import csv

with open('employees.csv', 'rt') as f:
    csv_reader = csv.reader(f, skipinitialspace=True)

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

预期输出：

```
['Id', 'User', 'Comment']
['1', 'Bob', 'John said \\Hello World\\""']
['2', 'Tom', '\\The Magician\\""']
['3', 'Harry', '\\walk around the corner\\" she explained to the child"']
['4', 'Louis', 'He said, \\stop pulling the dog\'s tail\\""']
```

这个输出肯定是不希望的。 要获得正确的输出，请使用 `escapechar` 参数更改转义符，如下所示：

```
import csv

```

```
with open('employees.csv', 'rt') as f:
    csv_reader = csv.reader(f, skipinitialspace=True, escapechar='\\')

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

预期输出：

```
['Id', 'User', 'Comment']
['1', 'Bob', 'John said "Hello World"']
['2', 'Tom', '"The Magician"']
['3', 'Harry', '"walk around the corner" she explained to the child']
['4', 'Louis', 'He said, "stop pulling the dog\'s tail"']
```

`doublequote` 参数

`dialogs.csv`

```
Id, Actor, Dialogue
1, Harley Betts, "The suspect told the arresting officer, ""I was nowhere near the crime."""
2, Clyde Esparza, "John said, ""I have just finished reading Browning's 'My Last Duchess.'"""
3, Zack Campbell, "Bill asked Sandra, ""Will you marry me?"""
4, Keziah Chaney, "The librarian whispered to us, ""The sign on the wall says 'Quiet'"""
```

此文件使用双引号转义字段中嵌入的双引号字符。 默认情况下， `doublequote` 设置为 `True` 。 结果，在读取两个连续的双引号时会被解释为一个。

```
import csv

with open('employees.csv', 'rt') as f:
    # same as csv_reader = csv.reader(f, skipinitialspace=True)
    csv_reader = csv.reader(f, skipinitialspace=True, doublequote=True)

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

预期输出：

```
['Id', 'Actor', 'Dialogue']
['1', 'Harley Betts', 'The suspect told the arresting officer, "I was nowhere near the crime."']
['2', 'Clyde Esparza', 'John said, "I have just finished reading Browning\'s \'My Last Duchess.\'"']
['3', 'Zack Campbell', 'Bill asked Sandra, "Will you marry me?"']
['4', 'Keziah Chaney', 'The librarian whispered to us, "The sign on the wall says \'Quiet\'"']
```

但是，如果将 `doublequote` 设置为 `False` ，则连续的双引号将出现在输出中。

```
import csv

with open('employees.csv', 'rt') as f:
    csv_reader = csv.reader(f, skipinitialspace=True, doublequote=False)

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

预期输出：

```
['Id', 'Actor', 'Dialogue']
['1', 'Harley Betts', 'The suspect told the arresting officer, "I was nowhere near the crime."""']
['2', 'Clyde Esparza', 'John said, "I have just finished reading Browning\'s \'My Last Duchess.\'"""']
['3', 'Zack Campbell', 'Bill asked Sandra, "Will you marry me?"""']
['4', 'Keziah Chaney', 'The librarian whispered to us, "The sign on the wall says \'Quiet\'"""']
```

用 `writer()` 编写 CSV 文件

要将数据写入 CSV 文件，我们使用 `writer()` 函数。 它接受与 `reader()` 函数相同的参数，但返回 `writer` 对象（即 `_csv.writer` ）：

语法： `writer(fileobj [, dialect='excel' [, **fmtparam] ]) -> csv_writer`

参数	描述
`fileobj`	（必需）它引用文件对象
`dialect`	（可选）方言是指格式化 CSV 文档的不同方式。 默认情况下， `csv` 模块使用与 Microsoft Excel 相同的格式。 我们将在本文后面详细讨论方言。

`fmtparam`	（可选）格式化参数，与 `reader()` 的功能相同。

`writer` 实例提供以下两种写入数据的方法：

方法	描述
`writerow(row)`	写入一行数据并返回写入的字符数。 `row` 必须是字符串和数字的序列。
`writerows(rows)`	写入多行数据并返回 `None` 。 `rows` 必须是一个序列。

以下是示例：

示例 **1** ：使用 `writerow()`

```
import csv

header = ['id', 'name', 'address', 'zip']
rows = [
    [1, 'Hannah', '4891 Blackwell Street, Anchorage, Alaska', 99503 ],
    [2, 'Walton', '4223 Half and Half Drive, Lemoore, California', 97401 ],
    [3, 'Sam', '3952 Little Street, Akron, Ohio', 93704],
    [4, 'Chris', '3192 Flinderation Road, Arlington Heights, Illinois', 62677],
    [5, 'Doug', '3236 Walkers Ridge Way, Burr Ridge', 61257],
]

with open('customers.csv', 'wt') as f:
    csv_writer = csv.writer(f)

    csv_writer.writerow(header) # write header

    for row in rows:
        csv_writer.writerow(row)
```

示例 **2** ：使用 `writerows()`

```
import csv

header = ['id', 'name', 'address', 'zip']
rows = [
    [1, 'Hannah', '4891 Blackwell Street, Anchorage, Alaska', 99503 ],
    [2, 'Walton', '4223 Half and Half Drive, Lemoore, California', 97401 ],
    [3, 'Sam', '3952 Little Street, Akron, Ohio', 93704],
    [4, 'Chris', '3192 Flinderation Road, Arlington Heights, Illinois', 62677],
    [5, 'Doug', '3236 Walkers Ridge Way, Burr Ridge', 61257],
]
```

```
11.
12. with open('customers.csv', 'wt') as f:
13.     csv_writer = csv.writer(f)
14.
15.     csv_writer.writerow(header) # write header
16.
17.     csv_writer.writerows(rows)
```

这两个清单生成的输出将是相同的，看起来像这样：

`customer.csv`

```
1. id,name,address,zip
2. 1,Hannah,"4891 Blackwell Street, Anchorage, Alaska",99503
3. 2,Walton,"4223 Half and Half Drive, Lemoore, California",97401
4. 3,Sam,"3952 Little Street, Akron, Ohio",93704
5. 4,Chris,"3192 Flinderation Road, Arlington Heights, Illinois",62677
6. 5,Doug,"3236 Walkers Ridge Way, Burr Ridge",61257
```

注意，只有地址字段用双引号引起来。 这是因为默认情况下 `quoting` 参数设置为 `QUOTE_MINIMAL` 。 换句话说，仅当 `quotechar` 或定界符出现在数据中时，字段才会被引用。

假设您要在所有文本数据中使用双引号。 为此，请将 `quoting` 参数设置为 `QUOTE_NONNUMERIC` 。

```
1. import csv
2.
3. header = ['id', 'name', 'address', 'zip']
4. rows = [
5.     [1, 'Hannah', '4891 Blackwell Street, Anchorage, Alaska', 99503 ],
6.     [2, 'Walton', '4223 Half and Half Drive, Lemoore, California', 97401 ],
7.     [3, 'Sam', '3952 Little Street, Akron, Ohio', 93704],
8.     [4, 'Chris', '3192 Flinderation Road, Arlington Heights, Illinois', 62677],
9.     [5, 'Doug', '3236 Walkers Ridge Way, Burr Ridge', 61257],
10. ]
11.
12. with open('customers.csv', 'wt') as f:
13.     csv_writer = csv.writer(f, quoting=csv.QUOTE_NONNUMERIC)
14.
15.     csv_writer.writerow(header) # write header
16.
17.     csv_writer.writerows(rows)
```

预期输出：

`customers.csv`

```
"id","name","address","zip"
1,"Hannah","4891 Blackwell Street, Anchorage, Alaska",99503
2,"Walton","4223 Half and Half Drive, Lemoore, California",97401
3,"Sam","3952 Little Street, Akron, Ohio",93704
4,"Chris","3192 Flinderation Road, Arlington Heights, Illinois",62677
5,"Doug","3236 Walkers Ridge Way, Burr Ridge",61257
```

现在，所有名称和地址都用双引号引起来。

如果要在所有字段周围加双引号，而不管数据中是否出现 `quotechar` 或定界符，请将 `quoting` 设置为 `csv.QUOTE_ALL` 。

```
import csv

header = ['id', 'name', 'address', 'zip']
rows = [
    [1, 'Hannah', '4891 Blackwell Street, Anchorage, Alaska', 99503 ],
    [2, 'Walton', '4223 Half and Half Drive, Lemoore, California', 97401 ],
    [3, 'Sam', '3952 Little Street, Akron, Ohio', 93704],
    [4, 'Chris', '3192 Flinderation Road, Arlington Heights, Illinois', 62677],
    [5, 'Doug', '3236 Walkers Ridge Way, Burr Ridge', 61257],
]

with open('customers.csv', 'wt') as f:
    csv_writer = csv.writer(f, quoting=csv.QUOTE_ALL)

    csv_writer.writerow(header) # write header

    csv_writer.writerows(rows)
```

预期输出：

```
"id","name","address","zip"
"1","Hannah","4891 Blackwell Street, Anchorage, Alaska","99503"
"2","Walton","4223 Half and Half Drive, Lemoore, California","97401"
"3","Sam","3952 Little Street, Akron, Ohio","93704"
"4","Chris","3192 Flinderation Road, Arlington Heights, Illinois","62677"
"5","Doug","3236 Walkers Ridge Way, Burr Ridge","61257"
```

现在一切都被双引号了。

重要的是要注意，当引号打开时（即 `quoting` 参数的值不是 `csv.QUOTE_NONE` ）， `csv` 模块将使用 `quotechar` （默认为 `"` ）对字段进行引号。

下面的清单将引号字符从双引号（ `"` ）更改为单引号（ `'` ）。

```
import csv

header = ['id', 'name', 'address', 'zip']
rows = [
    [1, 'Hannah', '4891 Blackwell Street, Anchorage, Alaska', 99503 ],
    [2, 'Walton', '4223 Half and Half Drive, Lemoore, California', 97401 ],
    [3, 'Sam', '3952 Little Street, Akron, Ohio', 93704],
    [4, 'Chris', '3192 Flinderation Road, Arlington Heights, Illinois', 62677],
    [5, 'Doug', '3236 Walkers Ridge Way, Burr Ridge', 61257],
]

with open('customers.csv', 'wt') as f:
    csv_writer = csv.writer(f, quotechar="'")

    csv_writer.writerow(header) # write header

    csv_writer.writerows(rows)
```

预期输出：

```
id,name,address,zip
1,Hannah,'4891 Blackwell Street, Anchorage, Alaska',99503
2,Walton,'4223 Half and Half Drive, Lemoore, California',97401
3,Sam,'3952 Little Street, Akron, Ohio',93704
4,Chris,'3192 Flinderation Road, Arlington Heights, Illinois',62677
5,Doug,'3236 Walkers Ridge Way, Burr Ridge',61257
```

在这种情况下， `csv` 模块使用单引号（ `'` ）而不是（ `"` ）来引用包含 quotechar 或定界符的字段。

我们也可以通过将 `quoting` 设置为 `csv.QUOTE_NONE` 来关闭全部引用。 但是，如果这样做，并且分隔符出现在数据中，则将出现如下错误：

```
import csv

```

```
header = ['id', 'name', 'address', 'zip']
rows = [
    [1, 'Hannah', '4891 Blackwell Street, Anchorage, Alaska', 99503 ],
    [2, 'Walton', '4223 Half and Half Drive, Lemoore, California', 97401 ],
    [3, 'Sam', '3952 Little Street, Akron, Ohio', 93704],
    [4, 'Chris', '3192 Flinderation Road, Arlington Heights, Illinois', 62677],
    [5, 'Doug', '3236 Walkers Ridge Way, Burr Ridge', 61257],
]

with open('customers.csv', 'wt') as f:
    csv_writer = csv.writer(f, quoting=csv.QUOTE_NONE)

    csv_writer.writerow(header) # write header

    csv_writer.writerows(rows)
```

预期输出：

```
Traceback (most recent call last):
...
csv_writer.writerows(rows)
_csv.Error: need to escape, but no escapechar set
```

问题在于地址字段包含嵌入式逗号（ `,` ），并且由于我们已关闭了引用字段的功能，因此 `csv` 模块不知道如何正确对其进行转义。

这是 `escapechar` 参数起作用的地方。 它使用一个单字符字符串，当引号关闭时（即 `quoting=csv.QUOTE_NONE` ），该字符串将用于转义分隔符。

以下清单将 `escapechar` 设置为反斜杠（ `\` ）。

```
import csv

header = ['id', 'name', 'address', 'zip']
rows = [
    [1, 'Hannah', '4891 Blackwell Street, Anchorage, Alaska', 99503 ],
    [2, 'Walton', '4223 Half and Half Drive, Lemoore, California', 97401 ],
    [3, 'Sam', '3952 Little Street, Akron, Ohio', 93704],
    [4, 'Chris', '3192 Flinderation Road, Arlington Heights, Illinois', 62677],
    [5, 'Doug', '3236 Walkers Ridge Way, Burr Ridge', 61257],
]

```

```
with open('customers.csv', 'wt') as f:
    csv_writer = csv.writer(f, quoting=csv.QUOTE_NONE, escapechar='\\')

    csv_writer.writerow(header) # write header

    csv_writer.writerows(rows)
```

预期输出：

```
id,name,address,zip
1,Hannah,4891 Blackwell Street\, Anchorage\, Alaska,99503
2,Walton,4223 Half and Half Drive\, Lemoore\, California,97401
3,Sam,3952 Little Street\, Akron\, Ohio,93704
4,Chris,3192 Flinderation Road\, Arlington Heights\, Illinois,62677
5,Doug,3236 Walkers Ridge Way\, Burr Ridge,61257
```

请注意，地址字段中的逗号（ `,` ）使用反斜杠（ `\` ）字符进行转义。

现在，您应该对 `reader()` 和 `writer()` 函数使用的各种格式参数及其上下文有很好的了解。 在下一节中，将看到其他一些读取和写入数据的方式。

使用 `DictReader` 读取 CSV 文件

`DictReader` 的工作原理几乎与 `reader()` 相似，但是它不是将行重新调谐为列表，而是返回字典。 其语法如下：

语法：： `DictReader(fileobj, fieldnames=None, restkey=None, restval=None, dialect='excel', **fmtparam)`

参数	描述
`fileobj`	（必需）引用文件对象。
`fieldnames`	（可选）它是指将按顺序在返回的字典中使用的键的列表。 如果省略，则从 CSV 文件的第一行推断字段名称。
`restkey`	（可选）如果行中的字段比 `fieldnames` 参数中指定的字段多，则其余字段将作为由 `restkey` 参数的值键控的序列存储。
`restval`	（可选）它为输入中缺少的字段提供值。
`dialect`	（可选）方言是指格式化 CSV 文档的不同方式。 默认情况下， `csv` 模块使用与 Microsoft excel 相同的格式。 我们将在本文后面详细讨论方言。
`fmtparam`	它指的是格式化参数，并且与 `reader()` 和 `writer()` 完全一样。

让我们举一些例子：

示例 **1** ：

`customers.csv`

```
id,name,address,zip
1,Hannah,4891 Blackwell Street\, Anchorage\, Alaska,99503
2,Walton,4223 Half and Half Drive\, Lemoore\, California,97401
3,Sam,3952 Little Street\, Akron\, Ohio,93704
4,Chris,3192 Flinderation Road\, Arlington Heights\, Illinois,62677
5,Doug,3236 Walkers Ridge Way\, Burr Ridge,61257
```

```
import csv

with open('customers.csv', 'rt') as f:
    csv_reader = csv.DictReader(f, escapechar='\\')

    for row in csv_reader:
        print(row)
```

预期输出：

```
{'id': '1', 'name': 'Hannah', 'zip': '99503', 'address': '4891 Blackwell Street, Anchorage, Alaska'}
{'id': '2', 'name': 'Walton', 'zip': '97401', 'address': '4223 Half and Half Drive, Lemoore, California'}
{'id': '3', 'name': 'Sam', 'zip': '93704', 'address': '3952 Little Street, Akron, Ohio'}
{'id': '4', 'name': 'Chris', 'zip': '62677', 'address': '3192 Flinderation Road, Arlington Heights, Illinois'}
{'id': '5', 'name': 'Doug', 'zip': '61257', 'address': '3236 Walkers Ridge Way, Burr Ridge'}
```

注意：结果中键的顺序可能会有所不同。 由于字典不保留元素的顺序。

在这种情况下，字段名称是从 CSV 文件的第一行（或标题）推断出来的。

示例 **2** ：使用 `fieldnames` 参数

```
1,Hannah,4891 Blackwell Street\, Anchorage\, Alaska,99503
2,Walton,4223 Half and Half Drive\, Lemoore\, California,97401
3,Sam,3952 Little Street\, Akron\, Ohio,93704
4,Chris,3192 Flinderation Road\, Arlington Heights\, Illinois,62677
5,Doug,3236 Walkers Ridge Way\, Burr Ridge,61257
```

此 CSV 文件没有标题。 因此，我们必须通过 `fieldnames` 参数提供字段名称。

```
import csv

with open('customers.csv', 'rt') as f:
    fields = ['id', 'name', 'address', 'zip']

    csv_reader = csv.DictReader(f, fieldnames=fields, escapechar='\\')

    for row in csv_reader:
        print(row)
```

预期输出：

```
{'name': 'Hannah', 'zip': '99503', 'id': '1', 'address': '4891 Blackwell Street, Anchorage, Alaska'}
{'name': 'Walton', 'zip': '97401', 'id': '2', 'address': '4223 Half and Half Drive, Lemoore, California'}
{'name': 'Sam', 'zip': '93704', 'id': '3', 'address': '3952 Little Street, Akron, Ohio'}
{'name': 'Chris', 'zip': '62677', 'id': '4', 'address': '3192 Flinderation Road, Arlington Heights, Illinois'}
{'name': 'Doug', 'zip': '61257', 'id': '5', 'address': '3236 Walkers Ridge Way, Burr Ridge'}
```

示例 **3** ：使用 `restkey` 参数

```
1,Hannah,4891 Blackwell Street\, Anchorage\, Alaska,99503
2,Walton,4223 Half and Half Drive\, Lemoore\, California,97401
3,Sam,3952 Little Street\, Akron\, Ohio,93704
4,Chris,3192 Flinderation Road\, Arlington Heights\, Illinois,62677
5,Doug,3236 Walkers Ridge Way\, Burr Ridge,61257
```

```
import csv

with open('customers.csv', 'rt') as f:

    fields = ['id','name',]

    csv_reader = csv.DictReader(f, fieldnames=fields, restkey='extra', escapechar='\\')
```

```
8.
9.      for row in csv_reader:
10.         print(row)
```

预期输出：

```
    {'id': '1', 'name': 'Hannah', 'extra': ['4891 Blackwell Street, Anchorage,
1.  Alaska', '99503']}
    {'id': '2', 'name': 'Walton', 'extra': ['4223 Half and Half Drive, Lemoore,
2.  California', '97401']}
    {'id': '3', 'name': 'Sam', 'extra': ['3952 Little Street, Akron, Ohio',
3.  '93704']}
    {'id': '4', 'name': 'Chris', 'extra': ['3192 Flinderation Road, Arlington
4.  Heights, Illinois', '62677']}
    {'id': '5', 'name': 'Doug', 'extra': ['3236 Walkers Ridge Way, Burr Ridge',
5.  '61257']}
```

请注意，地址和邮政编码现在存储为由值 `extra` 键控的序列。

示例 **4** ：使用 `restval`

```
1.  1,Hannah,4891 Blackwell Street\, Anchorage\, Alaska,99503
2.  2,Walton,4223 Half and Half Drive\, Lemoore\, California,97401
3.  3,Sam,3952 Little Street\, Akron\, Ohio,93704
4.  4,Chris,3192 Flinderation Road\, Arlington Heights\, Illinois,62677
5.  5,Doug,3236 Walkers Ridge Way\, Burr Ridge,61257
```

```
1.  import csv
2.
3.  with open('customers.csv', 'rt') as f:
4.
        fields = ['id','name', 'address', 'zip', 'phone', 'email'] # two extra
5.  fields
6.
        csv_reader = csv.DictReader(f, fieldnames=fields, restkey='extra',
7.  restval='NA', escapechar='\\')
8.
9.      for row in csv_reader:
10.         print(row)
```

预期输出：

```
    {'id': '1', 'name': 'Hannah', 'email': 'NA', 'phone': 'NA', 'address': '4891
1.  Blackwell Street, Anchorage, Alaska', 'zip': '99503'}
    {'id': '2', 'name': 'Walton', 'email': 'NA', 'phone': 'NA', 'address': '4223
2.  Half and Half Drive, Lemoore, California', 'zip': '97401'}
    {'id': '3', 'name': 'Sam', 'email': 'NA', 'phone': 'NA', 'address': '3952
3.  Little Street, Akron, Ohio', 'zip': '93704'}
    {'id': '4', 'name': 'Chris', 'email': 'NA', 'phone': 'NA', 'address': '3192
4.  Flinderation Road, Arlington Heights, Illinois', 'zip': '62677'}
    {'id': '5', 'name': 'Doug', 'email': 'NA', 'phone': 'NA', 'address': '3236
5.  Walkers Ridge Way, Burr Ridge', 'zip': '61257'}
```

在这种情况下，我们为字段指定了两个额外的字段： `phone` 和 `email` 。 额外字段的值由 `restval` 参数提供。

用 `DictWriter()` 编写 CSV 文件

`DictWriter` 对象将字典写入 CSV 文件。 其语法如下：

语法： `DictWriter(fileobj, fieldnames, restval='', extrasaction='raise', dialect='excel', **fmtparam)`

参数	描述
`fileobj`	它引用文件对象
`fieldnames`	它指的是字段名称以及将它们写入文件的顺序。
`restval`	它提供了字典中不存在的键的缺失值。
`extrasaction`	它控制如果字典包含 `fieldnames` 参数中找不到的键，该采取的动作。 默认情况下， `extrasaction` 设置为 `raise` ，这意味着将在此类事件中引发异常。 如果要忽略其他值，请将 `extrasaction` 设置为 `ignore` 。

`DictWriter` 提供以下三种写入数据的方法。

方法	描述
`writeheader()`	将标头（即 `fieldnames` ）写入 CSV 文件并返回 `None` 。
`writerow(row)`	写入一行数据并返回写入的字符数。 `row` 必须是字符串和数字的序列。
`writerows(rows)`	写入多行数据并返回 `None` 。 `rows` 必须是一个序列。

Let’s take some examples:

示例 **1:**

```
1.  import csv
2.
```

```
header = ['id', 'name', 'address', 'zip']

rows = [
    {'id': 1, 'name': 'Hannah', 'address': '4891 Blackwell Street, Anchorage, Alaska', 'zip': 99503 },
    {'id': 2, 'name': 'Walton', 'address': '4223 Half and Half Drive, Lemoore, California', 'zip': 97401 },
    {'id': 3, 'name': 'Sam', 'address': '3952 Little Street, Akron, Ohio', 'zip': 93704 },
    {'id': 4, 'name': 'Chris', 'address': '3192 Flinderation Road, Arlington Heights, Illinois', 'zip': 62677},
    {'id': 5, 'name': 'Doug', 'address': '3236 Walkers Ridge Way, Burr Ridge', 'zip': 61257},
]

with open('dictcustomers.csv', 'wt') as f:

    csv_writer = csv.DictWriter(f, fieldnames=header)

    csv_writer.writeheader() # write header

    csv_writer.writerows(rows)
```

预期输出：

`dictcustomers.csv`

```
id,name,address,zip
1,Hannah,"4891 Blackwell Street, Anchorage, Alaska",99503
2,Walton,"4223 Half and Half Drive, Lemoore, California",97401
3,Sam,"3952 Little Street, Akron, Ohio",93704
4,Chris,"3192 Flinderation Road, Arlington Heights, Illinois",62677
5,Doug,"3236 Walkers Ridge Way, Burr Ridge",61257
```

示例 2 ：使用 `restval`

```
import csv

header = ['id', 'name', 'address', 'zip', 'email'] # an extra field email

rows = [
    {'id': 1, 'name': 'Hannah', 'address': '4891 Blackwell Street, Anchorage, Alaska', 'zip': 99503 },
```

```
        {'id': 2, 'name': 'Walton', 'address': '4223 Half and Half Drive, Lemoore,
 7. California', 'zip': 97401 },
        {'id': 3, 'name': 'Sam', 'address': '3952 Little Street, Akron, Ohio',
 8. 'zip': 93704 },
        {'id': 4, 'name': 'Chris', 'address': '3192 Flinderation Road, Arlington
 9. Heights, Illinois', 'zip': 62677},
        {'id': 5, 'name': 'Doug', 'address': '3236 Walkers Ridge Way, Burr Ridge',
10. 'zip': 61257},
11. ]
12.
13. with open('dictcustomers.csv', 'wt') as f:
14.
15.     csv_writer = csv.DictWriter(f, fieldnames=header, restval="NA")
16.
17.     csv_writer.writeheader() # write header
18.
19.     csv_writer.writerows(rows)
```

预期输出：

`dictcustomers.csv`

```
1. id,name,address,zip,email
2. 1,Hannah,"4891 Blackwell Street, Anchorage, Alaska",99503,NA
3. 2,Walton,"4223 Half and Half Drive, Lemoore, California",97401,NA
4. 3,Sam,"3952 Little Street, Akron, Ohio",93704,NA
5. 4,Chris,"3192 Flinderation Road, Arlington Heights, Illinois",62677,NA
6. 5,Doug,"3236 Walkers Ridge Way, Burr Ridge",61257,NA
```

在这种情况下，字典中缺少 `email` 字段的值。 结果， `restval` 的值将用于 `email` 字段。

示例 **3** ：使用 `extrasaction`

```
1. import csv
2.
3. header = ['id', 'name', 'address'] # notice zip is missing
4.
5. rows = [
        {'id': 1, 'name': 'Hannah', 'address': '4891 Blackwell Street, Anchorage,
6. Alaska', 'zip': 99503 },
        {'id': 2, 'name': 'Walton', 'address': '4223 Half and Half Drive, Lemoore,
7. California', 'zip': 97401 },
```

```
        {'id': 3, 'name': 'Sam', 'address': '3952 Little Street, Akron, Ohio',
 8. 'zip': 93704 },
        {'id': 4, 'name': 'Chris', 'address': '3192 Flinderation Road, Arlington
 9. Heights, Illinois', 'zip': 62677},
        {'id': 5, 'name': 'Doug', 'address': '3236 Walkers Ridge Way, Burr Ridge',
10. 'zip': 61257},
11. ]
12.
13. with open('dictcustomers.csv', 'wt') as f:
14.
15.     csv_writer = csv.DictWriter(
16.                 f,
17.                 fieldnames=header,
18.                 restval="NA",
19.                 extrasaction='ignore' # ignore extra values in the dictionary
20.     )
21.
22.     csv_writer.writeheader() # write header
23.
24.     csv_writer.writerows(rows)
```

在此，词典包含一个名为 `zip` 的附加键，该键不在 `header` 列表中。 为了防止引发异常，我们将 `extrasaction` 设置为 `ignore` 。

预期输出：

dictcustomers.csv

```
1. id,name,address
2. 1,Hannah,"4891 Blackwell Street, Anchorage, Alaska"
3. 2,Walton,"4223 Half and Half Drive, Lemoore, California"
4. 3,Sam,"3952 Little Street, Akron, Ohio"
5. 4,Chris,"3192 Flinderation Road, Arlington Heights, Illinois"
6. 5,Doug,"3236 Walkers Ridge Way, Burr Ridge"
```

创建方言

在本文的前面，我们学习了各种格式化参数，这些参数使我们可以自定义 `reader` 和 `writer` 对象，以适应 CSV 约定中的差异。

如果发现自己一次又一次地传递相同的格式参数集。 考虑创建自己的方言。

方言对象或（仅是方言）是对各种格式参数进行分组的一种方式。 一旦创建了方言对象，只需将其传递给阅读器或书写器，而不必分别传递每个格式参数。

要创建新的方言，我们使用 `register_dialect()` 函数。 它接受方言名称作为字符串，并接受一个或多个格式参数作为关键字参数。

下表列出了所有格式参数及其默认值：

参数	默认值	描述
`delimiter`	`,`	它是指用于分隔 CSV 文件中的值（或字段）的字符。
`skipinitialspace`	`False`	它控制定界符后面的空格的解释方式。 如果为 `True`，则将删除初始空格。
`lineterminator`	`\r\n`	它是指用于终止行的字符序列。
`quotechar`	`"`	它指的是如果字段中出现特殊字符（如定界符），则将用于引用值的单个字符串。
`quoting`	`csv.QUOTE_NONE`	控制引号由作者生成或由读者识别的时间（其他选项请参见上文）。
`escapechar`	`None`	引用设置为引号时，它用于转义定界符的一字符字符串。
`doublequote`	`True`	控制字段内引号的处理。 当 `True` 时，在读取期间将两个连续的引号解释为一个，而在写入时，将嵌入数据中的每个引号字符写入为两个引号。

让我们创建一个简单的方言。

```
1.  import csv
2.
3.  # create and register new dialect
4.  csv.register_dialect('psv', delimiter='|', quoting=csv.QUOTE_NONNUMERIC)
5.
6.  header = ['id', 'year', 'age', 'name', 'movie']
7.
8.  rows = [
        {'id': 1, 'year': 2013, 'age': 55, 'name': "Daniel Day-Lewis", 'movie':
9.  "Lincoln" },
        {'id': 2, 'year': 2014, 'age': 44, 'name': "Matthew McConaughey", 'movie':
10. "Dallas Buyers Club" },
        {'id': 3, 'year': 2015, 'age': 33, 'name': "Eddie Redmayne", 'movie': "The
11. Theory of Everything" },
        {'id': 4, 'year': 2016, 'age': 41, 'name': "Leonardo DiCaprio", 'movie':
12. "The Revenant" }
13. ]
14.
15. with open('oscars.csv', 'wt') as f:
16.
```

```
    csv_writer = csv.DictWriter(
                f,
                fieldnames=header,
                dialect='psv', # pass the new dialect
                extrasaction='ignore'
    )

    csv_writer.writeheader() # write header

    csv_writer.writerows(rows)
```

预期输出：

oscars.csv

```
"id"|"year"|"age"|"name"|"movie"
1|2013|55|"Daniel Day-Lewis"|"Lincoln"
2|2014|44|"Matthew McConaughey"|"Dallas Buyers Club"
3|2015|33|"Eddie Redmayne"|"The Theory of Everything"
4|2016|41|"Leonardo DiCaprio"|"The Revenant"
```

用 Python 读写 JSON

> 原文: https://thepythonguru.com/reading-and-writing-json-in-python/

于 2020 年 1 月 7 日更新

JSON（JavaScript 对象表示法）是与语言无关的数据交换格式。 它是由道格拉斯·克罗克福德（Douglas Crockford）创建和推广的。 在短短的历史中，JSON 已成为事实上的跨网络数据传输标准。

JSON 是从 JavaScript 对象语法派生的基于文本的格式。 但是，它完全独立于 JavaScript，因此您无需知道任何 JavaScript 即可使用 JSON。

Web 应用通常使用 JSON 在客户端和服务器之间传输数据。 如果您使用的是 Web 服务，则很有可能默认情况下以 JSON 格式将数据返回给您。

在 JSON 诞生之前，XML 主要用于在客户端和服务器之间发送和接收数据。 XML 的问题在于它冗长，繁重且不容易解析。 但是，JSON 并非如此，您将很快看到。

以下是描述人的 XML 文档的示例。

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
    <firstName>John</firstName>
    <lastName>Smith</lastName>
    <isAlive>true</isAlive>
    <age>27</age>
    <address>
        <streetAddress>21 2nd Street</streetAddress>
        <city>New York</city>
        <state>NY</state>
        <postalCode>10021-3100</postalCode>
    </address>
    <phoneNumbers>
        <type>home</type>
        <number>212 555-1234</number>
    </phoneNumbers>
    <phoneNumbers>
        <type>office</type>
        <number>646 555-4567</number>
```

```
    </phoneNumbers>
    <phoneNumbers>
        <type>mobile</type>
        <number>123 456-7890</number>
    </phoneNumbers>
    <spouse />
</root>
```

可以使用 JSON 表示相同的信息，如下所示：

```
{
    "firstName": "John",
    "lastName": "Smith",
    "isAlive": true,
    "age": 27,
    "address": {
    "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": "10021-3100"
    },
    "phoneNumbers": [
        {
            "type": "home",
            "number": "212 555-1234"
        },
        {
            "type": "office",
            "number": "646 555-4567"
        },
        {
            "type": "mobile",
            "number": "123 456-7890"
        }
    ],
    "children": [],
    "spouse": null
}
```

我相信您会同意 JSON 副本更容易读写。

另外，请注意，JSON 格式与 Python 中的字典非常相似。

序列化和反序列化

序列化：将对象转换为适合通过网络传输或存储在文件或数据库中的特殊格式的过程称为序列化。

反序列化：与序列化相反。 它将序列化返回的特殊格式转换回可用的对象。

在 JSON 的情况下，当我们序列化对象时，实际上是将 Python 对象转换为 JSON 字符串，反序列化则通过其 JSON 字符串表示形式构建 Python 对象。

Python 提供了一个称为 `json` 的内置模块，用于对对象进行序列化和反序列化。 要使用 `json` 模块，请按以下步骤导入它：

```
>>>
>>> import json
>>>
```

`json` 模块主要提供以下用于序列化和反序列化的函数。

1. `dump(obj, fileobj)`
2. `dumps(obj)`
3. `load(fileobj)`
4. `loads(s)`

让我们从 `dump()` 函数开始。

使用 `dump()` 进行序列化

`dump()` 函数用于序列化数据。 它需要一个 Python 对象，对其进行序列化，然后将输出（它是 JSON 字符串）写入对象之类的文件。

`dump()` 函数的语法如下：

语法： `dump(obj, fp)`

参数	描述
`obj`	要序列化的对象。
`fp`	一个类似文件的对象，将在其中写入序列化数据。

这是一个例子：

```
>>>
```

```
>>> import json
>>>
>>> person = {
...     'first_name': "John",
...     "isAlive": True,
...     "age": 27,
...     "address": {
...         "streetAddress": "21 2nd Street",
...         "city": "New York",
...         "state": "NY",
...         "postalCode": "10021-3100"
...     },
...     "hasMortgage": None
... }
>>>
>>>
>>> with open('person.json', 'w') as f:  # writing JSON object
...     json.dump(person, f)
...
>>>
>>>
>>> open('person.json', 'r').read()   # reading JSON object as string
'{"hasMortgage": null, "isAlive": true, "age": 27, "address": {"state": "NY",
"streetAddress": "21 2nd Street", "city": "New York", "postalCode": "10021-
3100"}, "first_name": "John"}'
>>>
>>>
>>> type(open('person.json', 'r').read())
<class 'str'>
>>>
>>>
```

请注意，在序列化对象时，Python 的 `None` 类型将转换为 JSON 的 `null` 类型。

下表列出了序列化数据时类型之间的转换。

Python 类型	JSON 类型
`dict`	`object`
`list` , `tuple`	`array`
`int`	`number`
`float`	`number`
`str`	`string`

`True`	`true`
`False`	`false`
`None`	`null`

当我们反序列化对象时，JSON 类型将转换回其等效的 Python 类型。 下表中描述了此操作：

JSON 类型	Python 类型
`object`	`dict`
`array`	`list`
`string`	`str`
`number (int)`	`int`
`number (real)`	`float`
`true`	`True`
`false`	`False`
`null`	`None`

这是另一个序列化两个人的列表的示例：

```
>>>
>>>
>>> persons = \
... [
...     {
...         'first_name': "John",
...         "isAlive": True,
...         "age": 27,
...         "address": {
...             "streetAddress": "21 2nd Street",
...             "city": "New York",
...             "state": "NY",
...             "postalCode": "10021-3100"
...         },
...         "hasMortgage": None,
...     },
...     {
...         'first_name': "Bob",
...         "isAlive": True,
...         "age": 32,
...         "address": {
...             "streetAddress": "2428 O Conner Street",
...             "city": " Ocean Springs",
...             "state": "Mississippi",
```

```
...             "postalCode": "20031-9110"
...         },
...         "hasMortgage": True,
...     }
... ]
>>>
>>> with open('person_list.json', 'w') as f:
...     json.dump(persons, f)
...
>>>
>>>
>>> open('person_list.json', 'r').read()
'[{"hasMortgage": null, "isAlive": true, "age": 27, "address": {"state": "NY", "streetAddress": "21 2nd Street", "city": "New York", "postalCode": "10021-3100"}, "first_name": "John"}, {"hasMortgage": true, "isAlive": true, "age": 32, "address": {"state": "Mississippi", "streetAddress": "2428 O Conner Street", "city": " Ocean Springs", "postalCode": "20031-9110"}, "first_name": "Bob"}]'
>>>
>>>
```

现在，我们的 Python 对象已序列化到文件。 要将其反序列化回 Python 对象，我们使用 `load()` 函数。

用 `load()` 反序列化

`load()` 函数从类似于对象的文件中反序列化 JSON 对象并返回它。

其语法如下：

```
load(fp) -> a Python object
```

参数	描述
`fp`	从中读取 JSON 字符串的类似文件的对象。

Here is an example:

```
>>>
>>> with open('person.json', 'r') as f:
...     person = json.load(f)
```

```
...
>>>
>>> type(person)  # notice the type of data returned by load()
<class 'dict'>
>>>
>>> person
{'age': 27, 'isAlive': True, 'hasMortgage': None, 'address': {'state': 'NY',
'streetAddress': '21 2nd Street', 'city': 'New York', 'postalCode': '10021-
3100'}, 'first_name': 'John'}
>>>
>>>
```

使用 `dumps()` 和 `loads()` 进行序列化和反序列化

`dumps()` 函数的工作原理与 `dump()` 完全相同，但是它不是将输出发送到类似文件的对象，而是将输出作为字符串返回。

同样， `loads()` 函数与 `load()` 相同，但是它不是从文件反序列化 JSON 字符串，而是从字符串反序列化。

这里有些例子：

```
>>>
>>> person = {
...     'first_name': "John",
...     "isAlive": True,
...     "age": 27,
...     "address": {
...         "streetAddress": "21 2nd Street",
...         "city": "New York",
...         "state": "NY",
...         "postalCode": "10021-3100"
...     },
...     "hasMortgage": None
... }
>>>
>>> data = json.dumps(person)   # serialize
>>>
>>> data
```

```
    '{"hasMortgage": null, "isAlive": true, "age": 27, "address": {"state": "NY",
    "streetAddress": "21 2nd Street", "city": "New York", "postalCode": "10021-
18. 3100"}, "first_name": "John"}'
19. >>>
20. >>>
21. >>> person = json.loads(data)  # deserialize from string
22. >>>
23. >>> type(person)
24. <class 'dict'>
25. >>>
26. >>> person
    {'age': 27, 'isAlive': True, 'hasMortgage': None, 'address': {'state': 'NY',
    'streetAddress': '21 2nd Street', 'city': 'New York', 'postalCode': '10021-
27. 3100'}, 'first_name': 'John'}
28. >>>
29. >>>
```

注意：

由于字典不保留元素的顺序，因此获取键的顺序可能会有所不同。

自定义序列化器

以下是一些可选的关键字参数，可以将这些参数传递给 `dumps` 或 `dump()` 函数以自定义串行器。

参数	描述
`indent`	一个正整数，用于确定每个级别的键值对的缩进量。 如果您具有深层嵌套的数据结构，则 `indent` 参数可以方便地美化输出。 `indent` 的默认值为 `None` 。
`sort_keys`	布尔值标志（如果设置为 `True` ）将返回按键排序的 JSON 字符串，而不是随机排序的。 其默认值为 `False` 。
`skipkeys`	JSON 格式期望键为字符串，如果您尝试使用无法转换为字符串的类型（如元组），则会引发 `TypeError` 异常。 为防止引发异常并跳过非字符串键，请将 `skipkeys` 参数设置为 `True` 。
`separators`	它是指形式为 `(item_separator, key_separator)` 的元组。 `item_separator` 是一个字符串，用于分隔列表中的项目。 `key_separator` 也是一个字符串，用于分隔字典中的键和值。默认情况下， `separators` 设置为 `(',', ': ')` 。

以下是一些示例，演示了如何在操作中使用这些参数：

示例 **1** **：** 使用 `indent`

```
1. >>>
2. >>> print(json.dumps(person))  # without indent
```

```
    {"age": 27, "isAlive": true, "hasMortgage": null, "address": {"state": "NY",
    "streetAddress": "21 2nd Street", "city": "New York", "postalCode": "10021-
3.  3100"}, "first_name": "John"}
4.  >>>
5.  >>>
6.  >>> print(json.dumps(person, indent=4))  # with 4 levels of indentation
7.  {
8.      "age": 27,
9.      "isAlive": true,
10.     "hasMortgage": null,
11.     "address": {
12.         "state": "NY",
13.         "streetAddress": "21 2nd Street",
14.         "city": "New York",
15.         "postalCode": "10021-3100"
16.     },
17.     "first_name": "John"
18. }
19. >>>
20. >>>
```

请记住，增加缩进量也会增加数据的大小。 因此，请勿在生产环境中使用 `indent` 。

示例 **2** ：使用 `sort_keys`

```
1.  >>>
2.  >>> print(json.dumps(person, indent=4))   # print JSON string in random order
3.  {
4.      "address": {
5.          "state": "NY",
6.          "postalCode": "10021-3100",
7.          "city": "New York",
8.          "streetAddress": "21 2nd Street"
9.      },
10.     "hasMortgage": null,
11.     "first_name": "John",
12.     "isAlive": true,
13.     "age": 27
14. }
15. >>>
16. >>>
    >>> print(json.dumps(person, indent=4, sort_keys=True))  # print JSON string in
17. order by keys
```

```
{
    "address": {
        "city": "New York",
        "postalCode": "10021-3100",
        "state": "NY",
        "streetAddress": "21 2nd Street"
    },
    "age": 27,
    "first_name": "John",
    "hasMortgage": null,
    "isAlive": true
}
>>>
>>>
```

示例 **3** ：使用 `skipkeys`

```
>>>
>>> data = {'one': 1, 'two': 2, (1,2): 3}
>>>
>>> json.dumps(data, indent=4)
Traceback (most recent call last):
  ...
TypeError: key (1, 2) is not a string
>>>
>>>
```

在这种情况下，键 `(1,2)` 无法转换为字符串，因此会引发 `TypeError` 异常。 为防止引发异常并跳过非字符串键，请使用 `skipkeys` 参数。

```
>>>
>>> print(json.dumps(data, indent=4, skipkeys=True))
{
    "two": 2,
    "one": 1
}
>>>
>>>
```

示例 **4** ：使用 `separators`

```
>>>
```

```
>>> employee = {
...     'first_name': "Tom",
...     "designation": 'CEO',
...     "Salary": '2000000',
...     "age": 35,
...     "cars": ['chevy cavalier', 'ford taurus', 'tesla model x']
... }
>>>
>>>
>>> print(json.dumps(employee, indent=4, skipkeys=True,))
{
    "designation": "CEO",
    "age": 35,
    "cars": [
        "chevy cavalier",
        "ford taurus",
        "tesla model x"
    ],
    "Salary": "2000000",
    "first_name": "Tom"
}
>>>
>>>
```

以上输出中需要注意三件事：

1. 每个键值对使用逗号（`,`）分隔。
2. 数组中的项（例如 `cars`）也使用逗号（`,`）分隔。
3. JSON 对象的键使用 `': '` 与值分开（即冒号后跟一个空格）。

前两种情况下的分隔符使用 `item_separator` 字符串控制，最后一种情况下使用 `key_separator` 控制。 以下示例将 `item_separator` 和 `key_separator` 分别更改为竖线（`|`）和破折号（`-`）字符

```
>>>
>>> print(json.dumps(employee, indent=4, skipkeys=True, separators=('|', '-')))
{
    "designation"-"CEO"|
    "age"-35|
    "cars"-[
        "chevy cavalier"|
        "ford taurus"|
```

```
        "tesla model x"
    ]|
    "Salary"-"2000000"|
    "first_name"-"Tom"
}
>>>
>>>
```

现在您知道 `separators` 的工作原理，我们可以通过从 `item_separator` 字符串中删除空格字符来使输出更紧凑。 例如：

```
>>>
>>> print(json.dumps(employee, indent=4, skipkeys=True, separators=(',', ':')))
{
    "designation":"CEO",
    "age":35,
    "cars":[
        "chevy cavalier",
        "ford taurus",
        "tesla model x"
    ],
    "Salary":"2000000",
    "first_name":"Tom"
}
>>>
>>>
```

序列化自定义对象

默认情况下， `json` 模块仅允许我们序列化以下基本类型：

- `int`
- `float`
- `str`
- `bool`
- `list`
- `tuple`
- `dict`
- `None`

如果您尝试序列化或反序列化自定义对象或任何其他内置类型，将引发 `TypeError` 异常。 例如：

```
>>>
>>> from datetime import datetime
>>>
>>> now = datetime.now()
>>>
>>> now
datetime.datetime(2018, 9, 28, 22, 16, 46, 16944)
>>>
>>> d  = {'name': 'bob', 'dob': now}
>>>
>>> json.dumps(d)
Traceback (most recent call last):
  ...
TypeError: datetime.datetime(2018, 9, 28, 22, 7, 0, 622242) is not JSON
serializable
>>>
>>>
>>>
>>>
>>> class Employee:
...
...     def __init__(self, name):
...             self.name = name
...
>>>
>>> e = Employee('John')
>>>
>>> e
<__main__.Employee object at 0x7f20c82ee4e0>
>>>
>>>
>>> json.dumps(e)
Traceback (most recent call last):
  ...
TypeError: <__main__.Employee object at 0x7f20c82ee4e0> is not JSON
serializable
>>>
>>>
```

要序列化自定义对象或内置类型，我们必须创建自己的序列化函数。

```
1.  def serialize_objects(obj):
2.
3.      # serialize datetime object
4.
5.      if isinstance(obj, datetime):
6.          return {
7.              '__class__': datetime.__name__,
8.              '__value__': str(obj)
9.          }
10.
11.     # serialize Employee object
12.     #
13.     # if isinstance(obj, Employee):
14.     #     return {
15.     #         '__class__': 'Employee',
16.     #         '__value__': obj.name
17.     #     }
18.     raise TypeError(str(obj) + ' is not JSON serializable')
```

以下是有关该函数的一些注意事项。

1. 该函数采用一个名为 `obj` 的参数。
2. 在第 5 行中，我们使用 `isinstance()` 函数检查对象的类型。 如果您的函数仅序列化单个类型，则严格地不必检查类型，但是可以轻松添加其他类型的序列化。
3. 在 6-9 行中，我们使用两个键创建一个字典： `__class__` 和 `__value__` 。 `__class__` 键存储该类的原始名称，并将用于反序列化数据。 `__value__` 键存储对象的值，在这种情况下，我们仅需使用内置的 `str()` 函数将 `datetime.datetime` 对象转换为其字符串表示形式。
4. 在第 18 行中，我们引发了 `TypeError` 异常。 这是必要的，否则我们的序列化函数不会为无法序列化的对象报告错误。

我们的序列化函数现在可以序列化 `datetime.datetime` 对象。

下一个问题是-我们如何将自定义序列化函数传递给 `dumps()` 或 `dump()` 。

我们可以使用 `default` 关键字参数将自定义序列化函数传递给 `dumps()` 或 `dump()` 。 这是一个例子：

```
1.  >>>
2.  >>> def serialize_objects(obj):
```

```
...     if isinstance(obj, datetime):
...         return {
...             '__class__': datetime.__name__,
...             '__value__': str(obj)
...         }
...     raise TypeError(str(obj) + ' is not JSON serializable')
...
>>>
>>> employee = {
...     'first_name': "Mike",
...     "designation": 'Manager',
...     "doj": datetime(year=2016, month=5, day=2),  # date of joining
... }
>>>
>>>
>>> emp_json = json.dumps(employee, indent=4, default=serialize_objects)
>>>
>>>
>>> print(emp_json)
{
    "designation": "Manager",
    "doj": {
        "__value__": "2016-05-02 00:00:00",
        "__class__": "datetime"
    },
    "first_name": "Mike"
}
>>>
>>>
```

注意 `datetime.datetime` 对象如何被序列化为带有两个键的字典。

重要的是要注意，将仅调用 `serialize_objects()` 函数来序列化不是 Python 基本类型之一的对象。

现在，我们已经成功地序列化了 `datetime.datetime` 对象。 让我们看看如果尝试反序列化会发生什么。

```
>>>
>>> emp_dict = json.loads(emp_json)
>>>
>>> type(emp_dict)
<class 'dict'>
```

```
>>>
>>> emp_dict
{'designation': 'Manager', 'doj': {'__value__': '2016-05-02 00:00:00',
'__class__': 'datetime'}, 'first_name': 'Mike'}
>>>
>>> emp_dict['doj']
{'__value__': '2016-05-02 00:00:00', '__class__': 'datetime'}
>>>
>>>
```

请注意， `doj` 键的值作为字典而不是 `datetime.datetime` 对象返回。

发生这种情况是因为 `loads()` 函数对首先将 `datetime.datetime` 对象序列化的 `serialize_objects()` 函数一无所知。

我们需要的是 `serialize_objects()` 函数的反面-该函数接受字典对象，检查 `__class__` 键的存在，并根据 `__value__` 键中存储的字符串表示形式构建 `datetime.datetime` 对象。

```
def deserialize_objects(obj):
    if '__class__' in obj:
        if obj['__class__'] == 'datetime':
            return datetime.strptime(obj['__value__'], "%Y-%m-%d %H:%M:%S")

        # if obj['__class__'] == 'Employee':
        #     return Employee(obj['__value__'])

    return obj
```

这里唯一需要注意的是，我们正在使用 `datetime.strptime` 函数将日期时间字符串转换为 `datetime.datetime` 对象。

要将自定义反序列化函数传递给 `loads()` 方法，我们使用 `object_hook` 关键字参数。

```
>>>
>>> def deserialize_objects(obj):
...     if '__class__' in obj:
...         if obj['__class__'] == 'datetime':
...             return datetime.strptime(obj['__value__'], "%Y-%m-%d %H:%M:%S")
...         # if obj['__class__'] == 'Employee':
...         #     return Employee(obj['__value__'])
...     return obj
...
>>>
```

```
>>>
>>> emp_dict = json.loads(emp_json, object_hook=deserialize_objects)
>>>
>>> emp_dict
{'designation': 'Manager', 'doj': datetime.datetime(2016, 5, 2, 0, 0),
'first_name': 'Mike'}
>>>
>>> emp_dict['doj']
datetime.datetime(2016, 5, 2, 0, 0)
>>>
>>>
```

不出所料，这次 `doj` 键的值是 `datetime.datetime` 对象而不是字典。

用 Python 转储对象

> 原文: https://thepythonguru.com/pickling-objects-in-python/

于 2020 年 1 月 7 日更新

在用 Python 阅读和编写 JSON 一文中，我们了解了如何在 Python 中处理 JSON 数据。 如果您还没有看完这篇文章，我建议您这样做，然后再回到这里。

事实证明， `json` 模块不是序列化数据的唯一方法。 Python 提供了另一个名为 `pickle` 的模块来对数据进行序列化和反序列化。

这是 `json` 和 `pickle` 模块之间的主要区别。

1. `pickle` 模块是特定于 Python 的，这意味着对象被序列化后，就不能使用其他语言（如 PHP，Java，Perl 等）反序列化。如果需要互操作性，则请坚持使用 `json` 模块。
2. 与 `json` 模块将对象序列化为人类可读的 JSON 字符串不同， `pickle` 模块以二进制格式序列化数据。
3. `json` 模块允许我们仅序列化基本的 Python 类型（例如 `int` ， `str` ， `dict` ， `list` 等）。 如果需要序列化自定义对象，则必须提供自己的序列化函数。 但是， `pickle` 模块可立即使用多种 Python 类型，包括您定义的自定义对象。
4. `pickle` 模块的大多数代码都是用 C 编码的。因此，与 `json` 模块相比，它在处理大型数据集时性能得到了极大的提高。

`pickle` 模块提供的接口与 `json` 模块相同，并且由 `dump()` / `load()` 和 `dumps()` / `loads()` 函数组成。

要使用 `pickle` 模块，请按以下步骤导入它：

```
1. >>>
2. >>> import pickle
3. >>>
```

现在让我们看看如何使用 `pickle` 模块来序列化和反序列化对象。

注意：

序列化和反序列化有时也分别称为转储和加载。

用 `dump()` 转储

转储数据通过 `dump()` 函数完成。 它接受数据和文件对象。 然后， `dump()` 函数将数据序列化并将其写入文件。 `dump()` 的语法如下：

语法： `dump(obj, file)`

参数	描述
`obj`	要转储的对象。
`file`	将写入转储数据的文件对象。

这是一个例子：

```
>>>
>>> import pickle
>>>
>>> from datetime import datetime
>>>
>>>
>>> f = open("my_pickle", "wb") # remember to open the file in binary mode
>>>
>>> pickle.dump(10, f)
>>> pickle.dump("a string", f)
>>> pickle.dump({'a': 1, 'b': 2}, f)
>>> pickle.dump(datetime.now(), f) # serialize datetime.datetime object
>>>
>>> f.close()
>>>
>>>
```

这里有两件事要注意：

1. 首先，我们以二进制模式而不是文本模式打开文件。 这是必要的，否则在写入时数据将被破坏。
2. 其次， `dump()` 函数能够对 `datetime.datetime` 对象进行序列化，而无需提供任何自定义序列化函数。

显然，我们不仅限于 `datetime.datetime` 对象。 举一个例子，以下清单对 Python 中可用的其他一些类型进行了序列化。

```
>>>
>>> class My_class:
...     def __init__(self, name):
...         self.name = name
...
>>>
>>>
>>> def func(): return "func() called"
...
>>>
>>>
>>> f = open("other_pickles", "wb")
>>>
>>> pickle.dump(My_class, f) # serialize class object
>>>
>>> pickle.dump(2 + 3j, f) # serialize complex number
>>>
>>> pickle.dump(func, f) # serialize function object
>>>
>>> pickle.dump(bytes([1, 2, 3, 4, 5]), f) # serialize bytes object
>>>
>>> pickle.dump(My_class("name"), f) # serialize class instance
>>>
>>> f.close()
>>>
>>>
```

我们现在转储了一些数据。 此时，如果您尝试从文件中读取数据，则会将数据作为 `bytes` 对象获得。

```
>>>
>>> open("my_pickle", "rb").read()
b'\x80\x03K\n.\x80\x03X\x08\x00\x00\x00a
stringq\x00.\x80\x03}q\x00(X\x01\x00\x00\x00bq\x01K\x02X\x01\x00\x00\x00aq\x02K\x
>>>
>>>
>>> open("other_pickles", "rb").read()
b'\x80\x03c__main__\nMy_Class\nq\x00.\x80\x03cbuiltins\ncomplex\nq\x00G@\x00\x00\
>>>
>>>
```

这不是很可读。 对？

要恢复拾取的对象，我们使用 `load()` 函数

用 `load()` 加载

`load()` 函数获取一个文件对象，从转储的表示中重建对象，然后将其返回。

其语法如下：

参数	描述
`file`	从中读取序列化数据的文件对象。

现在，让我们尝试阅读我们在本文前面创建的 `my_pickle` 文件。

```
>>>
>>> f = open("my_pickle", "rb")
>>>
>>> pickle.load(f)
10
>>> pickle.load(f)
'a string'
>>>
>>> pickle.load(f)
{'b': 2, 'a': 1}
>>>
>>> pickle.load(f)
datetime.datetime(2018, 9, 30, 16, 46, 30, 866706)
>>>
>>> pickle.load(f)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
EOFError: Ran out of input
>>>
>>> f.close()
>>>
```

注意，对象的返回顺序与我们首先对其进行转储的顺序相同。 另外，请注意，该文件以二进制模式打开以进行读取。 当没有更多数据要返回时， `load()` 函数将引发 `EOFError` 。

同样，我们可以从 `other_pickles` 文件中读取转储的数据。

```
>>>
>>>
>>> f = open("other_pickles", "rb") # open the file for reading in binary mode
>>>
>>> My_class = pickle.load(f)
<class '__main__.My_class'>
>>>
>>>
>>> c = pickle.load(f)
>>>
>>> c
(2+3j)
>>>
>>>
>>> func = pickle.load(f)
>>>
>>> func
<function func at 0x7f9aa6ab6488>
>>>
>>>
>>> b = pickle.load(f)
>>>
>>> b
b'\x01\x02\x03\x04\x05'
>>>
>>>
>>> my_class_obj = pickle.load(f)
>>> my_class_obj
<__main__.My_Class object at 0x7f9aa74e61d0>
>>>
>>>
>>> pickle.load(f)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
EOFError: Ran out of input
>>>
>>>
>>> f.close()
>>>
>>>
```

加载数据后，就可以像普通的 Python 对象一样使用它了。

```
>>>
>>> func()
'func() called'
>>>
>>>
>>> c.imag, c.real
(3.0, 2.0)
>>>
>>>
>>> My_class("Tom")
<__main__.My_Class object at 0x7f9aa74e6358>
>>>
>>>
>>> my_class_obj.name
'name'
>>>
>>>
```

使用 `dumps()` 和 `load()` 进行转储和加载

`dumps()` 的工作方式与 `dump()` 完全相同，但是它不是将输出发送到文件，而是将转储的数据作为字符串返回。 其语法如下：

语法： `dumps(obj) -> pickled_data`

参数	描述
`obj`	要序列化的对象

同样， `loads()` 函数与 `load()` 相同，但是它不是从文件中读取转储的数据，而是从字符串中读取数据。 其语法如下：

语法： `loads(pickled_data) -> obj`

参数	描述
`pickled_data`	转储数据

Here is an example:

```
>>>
>>> employee = {
...     "first_name": "Mike",
```

```
...     "designation": 'Manager',
...     "doj": datetime(year=2016, month=5, day=2), # date of joining
... }
>>>
>>>
>>> pickled_emp = pickle.dumps(employee) # pickle employee dictionary
>>>
>>> pickled_emp
b'\x80\x03}q\x00(X\x0b\x00\x00\x00designationq\x01X\x07\x00\x00\x00Managerq\x02X\
>>>
>>>
>>> pickle.loads(pickled_emp) # unpickle employee dictionary
{'designation': 'Manager', 'doj': datetime.datetime(2016, 5, 2, 0, 0),
'first_name': 'Mike'}
>>>
>>>
```

请记住，当您释放数据时，对象会浮现，因此切勿尝试处理来自不受信任来源的转储数据。 恶意用户可以使用这种技术在系统上执行任意命令。