3. Python 速览

下面的例子以是否显示提示符（>>> 与 ...）区分输入与输出：输入例子中的代码时，要键入以提示符开头的行中提示符后的所有内容；未以提示符开头的行是解释器的输出。注意，例子中的某行出现的第二个提示符是用来结束多行命令的，此时，要键入一个空白行。

你可以使用 "Copy" 按钮（它会在悬停或点击代码示例时显示于右上角），它会去除提示符并省略输出，以将输入行复制粘贴到你的解释器。

本手册中的许多例子，甚至交互式命令都包含注释。Python 注释以 # 开头，直到该物理行结束。注释可以在行开头，或空白符与代码之后，但不能在字符串里面。字符串中的 # 号就是 # 号。注释用于阐明代码，Python 不解释注释，键入例子时，可以不输入注释。

示例如下：

```
# 这是第一条注释
spam = 1  # 而这是第二条注释
          # ... 而这是第三条！
text = "# 这不是注释因为它是在引号之内。"
```

3.1. Python 用作计算器

现在，尝试一些简单的 Python 命令。启动解释器，等待主提示符（>>> ）出现。

3.1.1. 数字

解释器像一个简单的计算器：你可以输入一个表达式，它将给出结果值。 表达式语法很直观：运算符 +, -, * 和 / 可被用来执行算术运算；圆括号 (()) 可被用来进行分组。 例如:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # 除法运算总是返回一个浮点数
1.6
```

整数（如，2、4、20 ）的类型是 int，带小数（如，5.0、1.6 ）的类型是 float。本教程后半部分将介绍更多数字类型。

除法运算 (/) 总是返回浮点数。 如果要做 floor division 得到一个整数结果你可以使用 // 运算符；要计算余数你可以使用 %:

```
>>> 17 / 3  # 经典除法运算返回一个浮点数
5.666666666666667
```

```
>>>
>>> 17 // 3  # 向下取整除法运算会丢弃小数部分
5
>>> 17 % 3  # % 运算返回相除的余数
2
>>> 5 * 3 + 2  # 向下取整的商 * 除数 + 余数
17
```

Python 用 `**` 运算符计算乘方 [1]：

```
>>> 5 ** 2  # 5 的平方
25
>>> 2 ** 7  # 2 的 7 次方
128
```

等号（`=`）用于给变量赋值。赋值后，下一个交互提示符的位置不显示任何结果：

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

如果变量未定义（即，未赋值），使用该变量会提示错误：

```
>>> n  # 试图访问一个未定义的变量
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Python 全面支持浮点数；混合类型运算数的运算会把整数转换为浮点数：

```
>>> 4 * 3.75 - 1
14.0
```

交互模式下，上次输出的表达式会赋给变量 `_`。把 Python 当作计算器时，用该变量实现下一步计算更简单，例如：

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

最好把该变量当作只读类型。不要为它显式赋值，否则会创建一个同名独立局部变量，该变量会用它的魔法行为屏蔽内置变量。

除了 `int` 和 `float`，Python 还支持其他数字类型，例如 `Decimal` 或 `Fraction`。Python 还内置支持复数，后缀 `j` 或 `J` 用于表示虚数（例如 `3+5j` ）。

3.1.2. 文本

除了数字 Python 还可以操作文本（由 `str` 类型表示，称为“字符串”）。 这包括字符 `"!"`, 单词 `"rabbit"`, 名称 `"Paris"`, 句子 `"Got your back."` 等等. `"Yay! :)"`。 它们可以用成对的单引号 (`'...'`) 或双引号 (`"..."`) 来标示，结果完全相同 [2]。

```
>>> 'spam eggs'  # 单引号
'spam eggs'
>>> "Paris rabbit got your back :)! Yay!"  # 双引号
'Paris rabbit got your back :)! Yay!'
>>> '1975'  # 引号中的数字也是字符串
'1975'
```

要标示引号本身，我们需要对它进行“转义”，即在前面加一个 `\`。 或者，我们也可以使用不同类型的引号:

```
>>> 'doesn\'t'  # 使用 \' 来转义单引号...
"doesn't"
>>> "doesn't"  # ...或者改用双引号
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

在 Python shell 中，字符串定义和输出字符串看起来可能不同。 `print()` 函数会略去标示用的引号，并打印经过转义的特殊字符，产生更为易读的输出:

```
>>> s = 'First line.\nSecond line.'  # \n 表示换行符
>>> s  # 不用 print()，特殊字符将包括在字符串中
'First line.\nSecond line.'
>>> print(s)  # 用 print()，特殊字符会被转写，因此 \n 将产生一个新行
First line.
Second line.
```

如果不希望前置 `\` 的字符转义成特殊字符，可以使用 *原始字符串*，在引号前添加 `r` 即可:

```
>>> print('C:\some\name')  # 这里 \n 表示换行符!
C:\some
ame
>>> print(r'C:\some\name')  # 请注意引号前的 r
C:\some\name
```

原始字符串还有一个微妙的限制：一个原始字符串不能以奇数个 `\` 字符结束；请参阅 此 FAQ 条目 了解更多信息及绕过的办法。

字符串字面值可以跨越多行。 一种做法是使用三重引号: `"""..."""` 或 `'''...'''`。 行结束符会自动包括在字符串中，但可以通过在行尾添加 `\` 来避免此行为。 在下面的例子中，开头的换行符将不会被包括:

```
>>> print("""\
... Usage: thingy [OPTIONS]
...      -h                        Display this usage message
```

```
...       -H hostname               Hostname to connect to
... """)
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to

>>>
```

字符串可以用 `+` 合并（粘到一起），也可以用 `*` 重复：

```
>>> # 3 乘以 'un'，再加 'ium'
>>> 3 * 'un' + 'ium'
'ununium'
```

相邻的两个或多个 *字符串字面值*（引号标注的字符）会自动合并：

```
>>> 'Py' 'thon'
'Python'
```

拼接分隔开的长字符串时，这个功能特别实用：

```
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

这项功能只能用于两个字面值，不能用于变量或表达式：

```
>>> prefix = 'Py'
>>> prefix 'thon'  # 不能拼接变量和字符串字面值
  File "<stdin>", line 1
    prefix 'thon'
           ^^^^^^
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  File "<stdin>", line 1
    ('un' * 3) 'ium'
               ^^^^^
SyntaxError: invalid syntax
```

合并多个变量，或合并变量与字面值，要用 `+`：

```
>>> prefix + 'thon'
'Python'
```

字符串支持 *索引*（下标访问），第一个字符的索引是 0。单字符没有专用的类型，就是长度为一的字符串：

```
>>> word = 'Python'
>>> word[0]  # 0 号位的字符
'P'
>>> word[5]  # 5 号位的字符
'n'
```

索引还支持负数，用负数索引时，从右边开始计数：

```
>>> word[-1]  # 最后一个字符
'n'
>>> word[-2]  # 倒数第二个字符
'o'
>>> word[-6]
'P'
```

注意，-0 和 0 一样，因此，负数索引从 -1 开始。

除了索引操作，还支持 *切片*。 索引用来获取单个字符，而 *切片* 允许你获取子字符串：

```
>>> word[0:2]  # 从 0 号位（含）到 2 号位（不含）的字符
'Py'
>>> word[2:5]  # 从 2 号位（含）到 5 号位（不含）的字符
'tho'
```

切片索引的默认值很有用；省略开始索引时，默认值为 0，省略结束索引时，默认为到字符串的结尾：

```
>>> word[:2]   # 从开头到 2 号位（不含）的字符
'Py'
>>> word[4:]   # 从 4 号位（含）到末尾
'on'
>>> word[-2:]  # 从倒数第二个（含）到末尾
'on'
```

注意，输出结果包含切片开始，但不包含切片结束。因此，`s[:i] + s[i:]` 总是等于 `s`：

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

还可以这样理解切片，索引指向的是字符 *之间*，第一个字符的左侧标为 0，最后一个字符的右侧标为 *n*，*n* 是字符串长度。例如：

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

第一行数字是字符串中索引 0…6 的位置，第二行数字是对应的负数索引位置。*i* 到 *j* 的切片由 *i* 和 *j* 之间所有对应的字符组成。

对于使用非负索引的切片，如果两个索引都不越界，切片长度就是起止索引之差。例如，`word[1:3]` 的长度是 2。

索引越界会报错：

```
>>> word[42]  # word 只有 6 个字符
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

但是，切片会自动处理越界索引：

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Python 字符串不能修改，是 immutable 的。因此，为字符串中某个索引位置赋值会报错：

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

要生成不同的字符串，应新建一个字符串：

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

内置函数 `len()` 返回字符串的长度：

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

参见:

文本序列类型 --- str

字符串是 *序列类型* ，支持序列类型的各种操作。

字符串的方法

字符串支持很多变形与查找方法。

f-字符串

内嵌表达式的字符串字面值。

格式字符串语法

使用 `str.format()` 格式化字符串。

printf 风格的字符串格式化

这里详述了用 `%` 运算符格式化字符串的操作。

3.1.3. 列表

Python 支持多种 *复合* 数据类型，可将不同值组合在一起。最常用的 *列表* ，是用方括号标注，逗号分隔的一组值。*列表* 可以包含不同类型的元素，但一般情况下，各个元素的类型相同：

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

和字符串（及其他内置 sequence 类型）一样，列表也支持索引和切片：

```
>>> squares[0]  # 索引操作将返回条目
1
>>> squares[-1]
25
>>> squares[-3:]  # 切片操作将返回一个新列表
[9, 16, 25]
```

列表还支持合并操作：

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

与 immutable 字符串不同, 列表是 mutable 类型，其内容可以改变：

```
>>> cubes = [1, 8, 27, 65, 125]  # 这里有点问题
>>> 4 ** 3  # 4 的立方是 64，不是 65！
64
>>> cubes[3] = 64  # 替换错误的值
>>> cubes
[1, 8, 27, 64, 125]
```

你也可以通过使用 `list.append()` *方法*，在列表末尾添加新条目（我们将在后文看到有关方法的更多介绍）：

```
>>> cubes.append(216)  # 添加 6 的立方
>>> cubes.append(7 ** 3)  # 和 7 的立方
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Python 中的简单赋值绝不会复制数据。 当你将一个列表赋值给一个变量时，该变量将引用 *现有的列表*。你通过一个变量对列表所做的任何更改都会被引用它的所有其他变量看到。：

```
>>> rgb = ["Red", "Green", "Blue"]
>>> rgba = rgb
>>> id(rgb) == id(rgba)  # 它们指向同一个对象
True
>>> rgba.append("Alph")
>>> rgb
["Red", "Green", "Blue", "Alph"]
```

切片操作返回包含请求元素的新列表。以下切片操作会返回列表的 浅拷贝：

```
>>> correct_rgba = rgba[:]
>>> correct_rgba[-1] = "Alpha"
>>> correct_rgba
["Red", "Green", "Blue", "Alpha"]
>>> rgba
["Red", "Green", "Blue", "Alph"]
```

为切片赋值可以改变列表大小，甚至清空整个列表：

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # 替换一些值
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # 现在移除它们
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # 通过用一个空列表替代所有元素来清空列表
>>> letters[:] = []
>>> letters
[]
```

内置函数 `len()` 也支持列表：

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

还可以嵌套列表（创建包含其他列表的列表），例如：

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2. 走向编程的第一步

当然，我们还能用 Python 完成比二加二更复杂的任务。 例如，我们可以像下面这样写出 斐波那契数列 初始部分的子序列：

```
>>> # 斐波那契数列：
>>> # 前两项之和即下一项的值
>>> a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
...
```

```
0
1
1
2
3
5
8
```

本例引入了几个新功能。

- 第一行中的 *多重赋值*：变量 `a` 和 `b` 同时获得新值 0 和 1。最后一行又用了一次多重赋值，体现了，等号右边的所有表达式的值，都是在这一语句对任何变量赋新值之前求出来的——求值顺序为从左到右。

- `while` 循环只要条件（这里是 `a < 10`）为真就会一直执行。Python 和 C 一样，任何非零整数都为真，零为假。这个条件也可以是字符串或列表类型的值，事实上，任何序列都可以：长度非零就为真，空序列则为假。示例中的判断只是最简单的比较。比较操作符的写法和 C 语言一样：`<`（小于）、`>`（大于）、`==`（等于）、`<=`（小于等于）、`>=`（大于等于）及 `!=`（不等于）。

- *循环体* 是 *缩进的*：缩进是 Python 组织语句的方式。在交互式命令行里，得为每个缩进的行输入空格（或制表符）。使用文本编辑器可以实现更复杂的输入方式；所有像样的文本编辑器都支持自动缩进。交互式输入复合语句时，要在最后输入空白行表示完成（因为解析器不知道哪一行代码是代码块的最后一行）。注意，同一块语句的每一行的缩进相同。

- `print()` 函数输出给定参数的值。 除了可以以单一的表达式作为参数（比如，前面的计算器的例子），它还能处理多个参数，包括浮点数与字符串。 它输出的字符串不带引号，且各参数项之间会插入一个空格，这样可以实现更好的格式化操作，就像这样:

  ```
  >>> i = 256*256
  >>> print('The value of i is', i)
  The value of i is 65536
  ```

  关键字参数 *end* 可以取消输出后面的换行, 或用另一个字符串结尾：

  ```
  >>> a, b = 0, 1
  >>> while a < 1000:
  ...     print(a, end=',')
  ...     a, b = b, a+b
  ...
  0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
  ```

备注

[1] `**` 比 `-` 的优先级更高, 所以 `-3**2` 会被解释成 `-(3**2)`，因此，结果是 `-9`。要避免这个问题，并且得到 `9`, 可以用 `(-3)**2`。

[2] 与其他语言不同，特殊字符如 `\n` 在单引号（`'...'`）和双引号（`"..."`）里的意义一样。这两种引号唯一的区别是，不需要在单引号里转义双引号 `"`（但此时必须把单引号转义成 `\'`），反之亦然。