7. 输入与输出

程序输出有几种显示方式；数据既可以输出供人阅读的形式，也可以写入文件备用。本章探讨一些可用的方式。

7.1. 更复杂的输出格式

至此，我们已学习了两种写入值的方法：*表达式语句* 和 `print()` 函数。第三种方法是使用文件对象的 `write()` 方法；标准输出文件称为 `sys.stdout`。详见标准库参考。

对输出格式的控制不只是打印空格分隔的值，还需要更多方式。格式化输出包括以下几种方法。

- 使用 格式化字符串字面值 ，要在字符串开头的引号/三引号前添加 `f` 或 `F` 。在这种字符串中，可以在 `{` 和 `}` 字符之间输入引用的变量，或字面值的 Python 表达式。

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- 字符串的 `str.format()` 方法需要更多手动操作。该方法也用 `{` 和 `}` 标记替换变量的位置，虽然这种方法支持详细的格式化指令，但需要提供格式化信息。

```
>>> yes_votes = 42_572_654
>>> no_votes = 43_132_495
>>> percentage = yes_votes / (yes_votes + no_votes)
>>> '{:-9} YES votes  {:2.2%}'.format(yes_votes, percentage)
' 42572654 YES votes  49.67%'
```

- 最后，还可以用字符串切片和合并操作完成字符串处理操作，创建任何排版布局。字符串类型还支持将字符串按给定列宽进行填充，这些方法也很有用。

如果不需要花哨的输出，只想快速显示变量进行调试，可以用 `repr()` 或 `str()` 函数把值转化为字符串。

`str()` 函数返回供人阅读的值，`repr()` 则生成适于解释器读取的值（如果没有等效的语法，则强制执行 `SyntaxError`）。对于没有支持供人阅读展示结果的对象， `str()` 返回与 `repr()` 相同的值。一般情况下，数字、列表或字典等结构的值，使用这两个函数输出的表现形式是一样的。字符串有两种不同的表现形式。

示例如下：

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

`string` 模块包含 `Template` 类，提供了将值替换为字符串的另一种方法。该类使用 `$x` 占位符，并用字典的值进行替换，但对格式控制的支持比较有限。

7.1.1. 格式化字符串字面值

格式化字符串字面值 （简称为 f-字符串）在字符串前加前缀 `f` 或 `F`，通过 `{expression}` 表达式，把 Python 表达式的值添加到字符串内。

格式说明符是可选的，写在表达式后面，可以更好地控制格式化值的方式。下例将 pi 舍入到小数点后三位：

```
>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.
```

在 `':'` 后传递整数，为该字段设置最小字符宽度，常用于列对齐：

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print(f'{name:10} ==> {phone:10d}')
...
Sjoerd     ==>       4127
Jack       ==>       4098
Dcab       ==>       7678
```

还有一些修饰符可以在格式化前转换值。 `'!a'` 应用 `ascii()` ， `'!s'` 应用 `str()`， `'!r'` 应用 `repr()`：

```
>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.
```

格式规范参考详见参考指南 格式规格迷你语言。

7.1.2. 字符串 format() 方法

`str.format()` 方法的基本用法如下所示：

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

花括号及之内的字符（称为格式字段）被替换为传递给 `str.format()` 方法的对象。花括号中的数字表示传递给 `str.format()` 方法的对象所在的位置。

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

`str.format()` 方法中使用关键字参数名引用值。

```
>>> print('This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

位置参数和关键字参数可以任意组合：

```
>>> print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',
...                                                   other='Georg'))
The story of Bill, Manfred, and Georg.
```

如果不想分拆较长的格式字符串，最好按名称引用变量进行格式化，不要按位置。这项操作可以通过传递字典，并用方括号 `'[]'` 访问键来完成。

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...       'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This could also be done by passing the `table` dictionary as keyword arguments with the `**` notation.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

与内置函数 `vars()` 结合使用时，这种方式非常实用，可以返回包含所有局部变量的字典。

As an example, the following lines produce a tidily aligned set of columns giving integers and their squares and cubes:

```
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

str.format() 进行字符串格式化的完整概述详见 格式字符串语法 。

7.1.3. 手动格式化字符串

下面是使用手动格式化方式实现的同一个平方和立方的表：

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Note use of 'end' on previous line
...     print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

（注意，每列之间的空格是通过使用 print() 添加的：它总在其参数间添加空格。）

字符串对象的 str.rjust() 方法通过在左侧填充空格，对给定宽度字段中的字符串进行右对齐。同类方法还有 str.ljust() 和 str.center() 。这些方法不写入任何内容，只返回一个新字符串，如果输入的字符串太长，它们不会截断字符串，而是原样返回；虽然这种方式会弄乱列布局，但也比另一种方法好，后者在显示值时可能不准确（如果真的想截断字符串，可以使用 x.ljust(n)[:n] 这样的切片操作 。）

另一种方法是 str.zfill() ，该方法在数字字符串左边填充零，且能识别正负号：

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

7.1.4. 旧式字符串格式化方法

% 运算符（求余符）也可用于字符串格式化。给定 'string' % values，则 string 中的 % 实例会以零个或多个 values 元素替换。此操作被称为字符串插值。例如：

```
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.
```

printf 风格的字符串格式化 小节介绍更多相关内容。

7.2. 读写文件

open() 返回一个 file object ，最常使用的是两个位置参数和一个关键字参数：open(filename, mode, encoding=None)

```
>>> f = open('workfile', 'w', encoding="utf-8")
```

第一个实参是文件名字符串。第二个实参是包含描述文件使用方式字符的字符串。*mode* 的值包括 'r' ，表示文件只能读取；'w' 表示只能写入（现有同名文件会被覆盖）；'a' 表示打开文件并追加内容，任何写入的数据会自动添加到文件末尾。'r+' 表示打开文件进行读写。*mode* 实参是可选的，省略时的默认值为 'r'。

通常情况下，文件是以 *text mode* 打开的，也就是说，你从文件中读写字符串，这些字符串是以特定的 *encoding* 编码的。如果没有指定 *encoding*，默认的是与平台有关的（见 `open()`）。因为 UTF-8 是现代事实上的标准，除非你知道你需要使用一个不同的编码，否则建议使用 `encoding="utf-8"`。在模式后面加上一个 `'b'`，可以用 *binary mode* 打开文件。二进制模式的数据是以 `bytes` 对象的形式读写的。在二进制模式下打开文件时，你不能指定 *encoding*。

在文本模式下读取文件时，默认把平台特定的行结束符（Unix 上为 `\n`, Windows 上为 `\r\n`）转换为 `\n`。在文本模式下写入数据时，默认把 `\n` 转换回平台特定结束符。这种操作方式在后台修改文件数据对文本文件来说没有问题，但会破坏 `JPEG` 或 `EXE` 等二进制文件中的数据。注意，在读写此类文件时，一定要使用二进制模式。

在处理文件对象时，最好使用 `with` 关键字。优点是，子句体结束后，文件会正确关闭，即便触发异常也可以。而且，使用 `with` 相比等效的 `try-finally` 代码块要简短得多：

```
>>> with open('workfile', encoding="utf-8") as f:
...     read_data = f.read()

>>> # We can check that the file has been automatically closed.
>>> f.closed
True
```

如果没有使用 `with` 关键字，则应调用 `f.close()` 关闭文件，即可释放文件占用的系统资源。

警告： 调用 `f.write()` 时，未使用 `with` 关键字，或未调用 `f.close()`，即使程序正常退出，也**可能** 导致 `f.write()` 的参数没有完全写入磁盘。

通过 `with` 语句，或调用 `f.close()` 关闭文件对象后，再次使用该文件对象将会失败。

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

7.2.1. 文件对象的方法

本节下文中的例子假定已创建 `f` 文件对象。

`f.read(size)` 可用于读取文件内容，它会读取一些数据，并返回字符串（文本模式），或字节串对象（在二进制模式下）。*size* 是可选的数值参数。省略 *size* 或 *size* 为负数时，读取并返回整个文件的内容；文件大小是内存的两倍时，会出现问题。*size* 取其他值时，读取并返回最多 *size* 个字符（文本模式）或 *size* 个字节（二进制模式）。如已到达文件末尾，`f.read()` 返回空字符串（`''`）。

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` 从文件中读取单行数据；字符串末尾保留换行符（`\n`），只有在文件不以换行符结尾时，文件的最后一行才会省略换行符。这种方式让返回值清晰明确；只要 `f.readline()` 返回空字符串，就表示已经到达了文件末尾，空行使用 `'\n'` 表示，该字符串只包含一个换行符。

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

从文件中读取多行时，可以用循环遍历整个文件对象。这种操作能高效利用内存，快速，且代码简单：

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

如需以列表形式读取文件中的所有行，可以用 `list(f)` 或 `f.readlines()`。

`f.write(string)` 把 *string* 的内容写入文件，并返回写入的字符数。

```
>>> f.write('This is a test\n')
15
```

写入其他类型的对象前，要先把它们转化为字符串（文本模式）或字节对象（二进制模式）：

```
>>> value = ('the answer', 42)
>>> s = str(value)  # convert the tuple to string
>>> f.write(s)
18
```

`f.tell()` 返回整数，给出文件对象在文件中的当前位置，表示为二进制模式下时从文件开始的字节数，以及文本模式下的意义不明的数字。

`f.seek(offset, whence)` 可以改变文件对象的位置。通过向参考点添加 *offset* 计算位置；参考点由 *whence* 参数指定。 *whence* 值为 0 时，表示从文件开头计算，1 表示使用当前文件位置，2 表示使用文件末尾作为参考点。省略 *whence* 时，其默认值为 0，即使用文件开头作为参考点。

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

在文本文件（模式字符串未使用 `b` 时打开的文件）中，只允许相对于文件开头搜索（使用 `seek(0, 2)` 搜索到文件末尾是个例外），唯一有效的 *offset* 值是能从 `f.tell()` 中返回的，或 0。其他 *offset* 值都会产生未定义的行为。

文件对象还支持 `isatty()` 和 `truncate()` 等方法，但不常用；文件对象的完整指南详见库参考。

7.2.2. 使用 `json` 保存结构化数据

从文件写入或读取字符串很简单，数字则稍显麻烦，因为 `read()` 方法只返回字符串，这些字符串必须传递给 `int()` 这样的函数，接受 `'123'` 这样的字符串，并返回数字值 123。保存嵌套列表、字典等复杂数据类型时，手动解析和序列化的操作非常复杂。

Python 支持 JSON (JavaScript Object Notation) 这种流行数据交换格式，用户无需没完没了地编写、调试代码，才能把复杂的数据类型保存到文件。`json` 标准模块采用 Python 数据层次结构，并将之转换为字符串表示形式；这个过程称为 *serializing*（序列化）。从字符串表示中重建数据称为 *deserializing*（解序化）。在序列化和解序化之间，表示对象的字符串可能已经存储在文件或数据中，或通过网络连接发送到远方 的机器。

注解： JSON 格式通常用于现代应用程序的数据交换。程序员早已对它耳熟能详，可谓是交互操作的不二之选。

只需一行简单的代码即可查看某个对象的 JSON 字符串表现形式：

```
>>> import json
>>> x = [1, 'simple', 'list']
>>> json.dumps(x)
'[1, "simple", "list"]'
```

`dumps()` 函数还有一个变体， `dump()` ，它只将对象序列化为 text file 。因此，如果 `f` 是 text file 对象，可以这样做：

```
json.dump(x, f)
```

要再次解码对象，如果 `f` 是已打开、供读取的 binary file 或 text file 对象：

```
x = json.load(f)
```

注解： JSON文件必须以UTF-8编码。当打开JSON文件作为一个 text file 用于读写时，使用 `encoding="utf-8"` 。

这种简单的序列化技术可以处理列表和字典，但在 JSON 中序列化任意类的实例，则需要付出额外努力。`json` 模块的参考包含对此的解释。

参见： `pickle` - 封存模块

与 JSON 不同，*pickle* 是一种允许对复杂 Python 对象进行序列化的协议。因此，它为 Python 所特有，不能用于与其他语言编写的应用程序通信。默认情况下它也是不安全的：如果解序化的数据是由手段高明的攻击者精心设计的，这种不受信任来源的 pickle 数据可以执行任意代码。

© 版权所有 2001-2022, Python Software Foundation.
This page is licensed under the Python Software Foundation License Version 2.
Examples, recipes, and other code in the documentation are additionally licensed under the Zero Clause BSD License.
See History and License for more information.

The Python Software Foundation is a non-profit corporation. Please donate.

最后更新于 9月 15, 2022. Found a bug?
Created using Sphinx 3.4.3.