Entrada y salida

Hay diferentes métodos de presentar la salida de un programa; los datos pueden ser impresos de una forma legible por humanos, o escritos a un archivo para uso futuro. Este capítulo discutirá algunas de las posibilidades.

Formateo elegante de la salida

Hasta ahora encontramos dos maneras de escribir valores: *declaraciones de expresión* y la función `print()`. (Una tercer manera es usando el método `write()` de los objetos tipo archivo; el archivo de salida estándar puede referenciarse como `sys.stdout`. Mirá la Referencia de la Biblioteca para más información sobre esto.)

Frecuentemente querrás más control sobre el formateo de tu salida que simplemente imprimir valores separados por espacios. Hay dos maneras de formatear tu salida; la primera es hacer todo el manejo de las cadenas vos mismo: usando rebanado de cadenas y operaciones de concatenado podés crear cualquier forma que puedas imaginar. El tipo *string* contiene algunos métodos útiles para emparejar cadenas a un determinado ancho; estas las discutiremos en breve. La otra forma es usar *formatted string literals* o el método `str.format()`.

El módulo `string` contiene una clase `string.Template` que ofrece otra forma de sustituir valores en las cadenas.

Nos queda una pregunta, por supuesto: ¿cómo convertís valores a cadenas? Afortunadamente, Python tiene maneras de convertir cualquier valor a una cadena: pasalos a las funciones `repr()` o `str()`.

La función `str()` devuelve representaciones de los valores que son bastante legibles por humanos, mientras que `repr()` genera representaciones que pueden ser leídas por el el intérprete (o forzarían un `SyntaxError` si no hay sintáxis equivalente). Para objetos que no tienen una representación en particular para consumo humano, `str()` devolverá el mismo valor que `repr()`. Muchos valores, como números o estructuras como listas y diccionarios, tienen la misma representación usando cualquiera de las dos funciones. Las cadenas, en particular, tienen dos representaciones distintas.

Algunos ejemplos:

```
>>> s = 'Hola mundo.'
>>> str(s)
'Hola mundo.'
>>> repr(s)
"'Hola mundo.'"
>>> str(1 / 7)
'0.142857142857'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'El valor de x es ' + repr(x) + ', y es ' + repr(y) + '...'
>>> print(s)
El valor de x es 32.5, y es 40000...
>>> # El repr() de una cadena agrega apóstrofos y barras invertidas
... hola = 'hola mundo\n'
>>> holas = repr(hola)
>>> print(holas)
'hola mundo\n'
>>> # El argumento de repr() puede ser cualquier objeto Python:
... repr((x, y, ('carne', 'huevos')))
"(32.5, 40000, ('carne', 'huevos'))"
```

Acá hay dos maneras de escribir una tabla de cuadrados y cubos:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x * x).rjust(3), end=' ')
...     # notar el uso de 'end' en la linea anterior
...     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

>>> 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
```

(Notar que en el primer ejemplo, un espacio entre cada columna fue agregado por la manera en que **`print()`** trabaja: siempre agrega espacios entre sus argumentos)

Este ejemplo muestra el método **`str.rjust()`** de los objetos cadena, el cual ordena una cadena a la derecha en un campo del ancho dado llenándolo con espacios a la izquierda. Hay métodos similares **`str.ljust()`** y **`str.center()`**. Estos métodos no escriben nada, sólo devuelven una nueva cadena. Si la cadena de entrada es demasiado larga, no la truncan, sino la devuelven intacta; esto te romperá la alineación de tus columnas pero es normalmente mejor que la alternativa, que te estaría mintiendo sobre el valor. (Si realmente querés que se recorte, siempre podés agregarle una operación de rebanado, como en `x.ljust(n)[:n]`.)

Hay otro método, **`str.zfill()`** , el cual rellena una cadena numérica a la izquierda con ceros. Entiende signos positivos y negativos:

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

El uso básico del método **`str.format()`** es como esto:

```
>>> print('Somos los {} quienes decimos "{}!"'.format('caballeros', 'Nop'))
Somos los caballeros quienes decimos "Nop!"
```

Las llaves y caracteres dentro de las mismas (llamados campos de formato) son reemplazadas con los objetos pasados en el método **`str.format()`**. Un número en las llaves se refiere a la posición del objeto pasado en el método.

```
>>> print('{0} y {1}'.format('carne', 'huevos'))
carne y huevos
>>> print('{1} y {0}'.format('carne', 'huevos'))
huevos y carne
```

Si se usan argumentos nombrados en el método **`str.format()`**, sus valores serán referidos usando el nombre del argumento.

```
>>> print('Esta {comida} es {adjetivo}.'.format(
...       comida='carne', adjetivo='espantosa'))
Esta carne es espantosa.
```

Se pueden combinar arbitrariamente argumentos posicionales y nombrados:

```
>>> print('La historia de {0}, {1}, y {otro}.'.format('Bill', 'Manfred',
...                                                   otro='Georg'))
La historia de Bill, Manfred, y Georg.
```

Se pueden usar `'!a'` (aplica **`apply()`**), `'!s'` (aplica **`str()`**) y `'!r'` (aplica **`repr()`**) para convertir el valor antes de que se formatee.

```
>>> contents = 'anguilas'
>>> print('Mi aerodeslizador esta lleno de {}.'.format(contents))
Mi aerodeslizador esta lleno de anguilas.
>>> print('My hovercraft is full of {!r}.'.format(contents))
Mi aerodeslizador esta lleno de 'anguilas'.
```

Un `':'` y especificador de formato opcionales pueden ir luego del nombre del campo. Esto aumenta el control sobre cómo el valor es formateado. El siguiente ejemplo redondea Pi a tres lugares luego del punto decimal.

```
>>> import math
>>> print('El valor de PI es aproximadamente {0:.3f}.'.format(math.pi))
El valor de PI es aproximadamente 3.142.
```

Pasando un entero luego del `':'` causará que que el campo sea de un mínimo número de caracteres de ancho. Esto es útil para hacer tablas lindas.

```
>>> tabla = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for nombre, telefono in tabla.items():
...     print('{0:10} ==> {1:10d}'.format(nombre, telefono))
...
Dcab       ==>       7678
Jack       ==>       4098
Sjoerd     ==>       4127
```

Si tenés una cadena de formateo realmente larga que no querés separar, podría ser bueno que puedas hacer referencia a las variables a ser formateadas por el nombre en vez de la posición. Esto puede hacerse simplemente pasando el diccionario y usando corchetes `'[]'` para acceder a las claves

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

Esto se podría también hacer pasando la tabla como argumentos nombrados con la notación '**'.

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

Esto es particularmente útil en combinación con la función integrada **`vars()`**, que devuelve un diccionario conteniendo todas las variables locales.

Para una completa descripción del formateo de cadenas con **`str.format()`**, mirá en *Tipos integrados*.

Viejo formateo de cadenas

El operador `%` también puede usarse para formateo de cadenas. Interpreta el argumento de la izquierda con el estilo de formateo de **`sprintf()`** para ser aplicado al argumento de la derecha, y devuelve la cadena resultante de esta operación de formateo. Por ejemplo:

```
>>> import math
>>> print('El valor de PI es aproximadamente %5.3f.' % math.pi)
El valor de PI es aproximadamente 3.142.
```

Podés encontrar más información en la sección *Tipos integrados*.

Leyendo y escribiendo archivos

La función **`open()`** devuelve un *objeto archivo*, y se usa normalmente con dos argumentos: `open(nombre_de_archivo, modo)`.

```
>>> f = open('archivodetrabajo', 'w')
>>> print(f)
<_io.TextIOWrapper name='archivodetrabajo' mode='w' encoding='UTF-8'>
```

El primer argumento es una cadena conteniendo el nombre de archivo. El segundo argumento es otra cadena conteniendo unos pocos caracteres que describen la forma en que el archivo será usado. El *modo* puede ser `'r'` cuando el archivo será solamente leído, `'w'` para sólo escribirlo (un archivo existente con el mismo nombre será borrado), y `'a'` abre el archivo para agregarle información; cualquier dato escrito al archivo será automáticamente agregado al final. `'r+'` abre el archivo tanto para leerlo como para escribirlo. El argumento *modo* es opcional; si se omite se asume `'r'`.

Normalmente los archivos se abren en *modo texto*, lo que significa que podés leer y escribir cadenas del y al archivo, las cuales se codifican utilizando un código específico. Si el código no es especificado, el valor predeterminado depende de la plataforma. Si se agrega `b` al modo el archivo se abre en *modo binario*: ahora los datos se leen y escriben en forma de objetos bytes. Se debería usar este modo para todos los archivos que no contengan texto.

Cuando se lee en modo texto, por defecto se convierten los fines de lineas que son específicos a las plataformas (`\n` en Unix, `\r\n` en Windows) a solamente `\n`. Cuando se escribe en modo texto, por defecto se convierten los `\n` a los finales de linea específicos de la plataforma. Este cambio automático está bien para archivos de texto, pero corrompería datos binarios como los de archivos JPEG o EXE. Asegurate de usar modo binario cuando leas y escribas tales archivos.

Es una buena práctica usar la declaración **`with`** cuando manejamos objetos archivo. Tiene la ventaja que el archivo es cerrado apropiadamente luego de que el bloque termina, incluso si se generó una excepción. También es mucho más corto que escribir los equivalentes bloques **`try-finally`**

```
>>> with open('archivodetrabajo') as f:
...     datos_leidos = f.read()
>>> f.closed
True
```

Si no estuvieses usando el bloque **`with`**, entonces deberías llamar `f.close()` para cerrar el archivo e inmediatamente liberar cualquier recurso del sistema usado por este. Si no cierras explícitamente el archivo, el «garbage collector» de Python eventualmente destruirá el objeto y cerrará el archivo por vos, pero el archivo puede estar abierto por un tiempo. Otro riesgo es que diferentes implementaciones de Python harán esta limpieza en diferentes momentos.

Después de que un objeto de archivo es cerrado, ya sea por **`with`** o llamando a `f.close()`, intentar volver a utilizarlo fallará automáticamente:

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

Métodos de los objetos Archivo

El resto de los ejemplos en esta sección asumirán que ya se creó un objeto archivo llamado`f`.

Para leer el contenido de una archivo llamá a `f.read(cantidad)`, el cual lee alguna cantidad de datos y los devuelve como una cadena de (en modo texto) o un objeto de bytes (en modo binario). *cantidad* es un argumento numérico opcional. Cuando se omite *cantidad* o es negativo, el contenido entero del archivo será leido y devuelto; es tu problema si el archivo es el doble de grande que la memoria de tu máquina. De otra manera, a lo sumo una *cantidad* de bytes son leídos y devueltos. Si se alcanzó el fin del archivo, `f.read()` devolverá una cadena vacía (`""`).

```
>>> f.read()
'Este es el archivo entero.\n'
>>> f.read()
''
```

`f.readline()` lee una sola linea del archivo; el caracter de fin de linea (`\n`) se deja al final de la cadena, y sólo se omite en la última linea del archivo si el mismo no termina en un fin de linea. Esto hace que el valor de retorno no sea ambiguo; si `f.readline()` devuelve una cadena vacía, es que se alcanzó el fin del archivo, mientras que una linea en blanco es representada por `'\n'`, una cadena conteniendo sólo un único fin de linea.

```
>>> f.readline()
'Esta es la primer linea del archivo.\n'
>>> f.readline()
'Segunda linea del archivo\n'
>>> f.readline()
''
```

Para leer líneas de un archivo, podés iterar sobre el objeto archivo. Esto es eficiente en memoria, rápido, y conduce a un código más simple:

```
>>> for linea in f:
...     print(linea, end='')

Esta es la primer linea del archivo
Segunda linea del archivo
```

Si querés leer todas las líneas de un archivo en una lista también podés usar`list(f)` o `f.readlines()`.

`f.write(cadena)` escribe el contenido de la *cadena* al archivo, devolviendo la cantidad de caracteres escritos.

```
>>> f.write('Esto es una prueba\n')
19
```

Otros tipos de objetos necesitan serconvertidos -- tanto a una cadena (en modo texto) o a un objeto de bytes (en modo binario) -- antes de escribirlos:

```
>>> valor = ('la respuesta', 42)
>>> s = str(valor)  # Convierte la tupla a string
>>> f.write(s)
18
```

`f.tell()` devuelve un entero que indica la posición actual en el archivo representada como número de bytes desde el comienzo del archivo en modo binario y un número opaco en modo texto.

Para cambiar la posición del objeto archivo, usá `f.seek(desplazamiento, desde_donde)`. La posición es calculada agregando el *desplazamiento* a un punto de referencia; el punto de referencia se selecciona del argumento *desde_donde*. Un valor *desde_donde* de 0 mide desde el comienzo del archivo, 1 usa la posición actual del archivo, y 2 usa el fin del archivo como punto de referencia. *desde_donde* puede omitirse, el default es 0, usando el comienzo del archivo como punto de referencia.

```
>>> f = open('archivodetrabajo', 'rb+')
>>> f.write(b'0123456789abcdef')
>>> f.seek(5)      # Va al sexto byte en el archivo
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # Va al tercer byte antes del final
13
>>> f.read(1)
b'd'
```

En los archivos de texto (aquellos que se abrieron sin una `b` en el modo), se permiten solamente desplazamientos con `seek` relativos al comienzo (con la excepción de ir justo al final con `seek(0, 2)`) y los únicos valores de *desplazamiento* válidos son aquellos retornados por `f.tell()`, o cero. Cualquier otro valor de *desplazamiento* produce un comportamiento indefinido.

Los objetos archivo tienen algunos métodos más, como **`isatty()`** y **`truncate()`** que son usados menos frecuentemente; consultá la Referencia de la Biblioteca para una guía completa sobre los objetos archivo.

Guardar datos estructurados con JSON

Las cadenas pueden facilmente escribirse y leerse de un archivo. Los números toman algo más de esfuerzo, ya que el método **`read()`** sólo devuelve cadenas, que tendrán que ser pasadas a una función como **`int()`**, que toma una cadena como `'123'` y devuelve su valor numérico 123. Sin embargo, cuando querés grabar tipos de datos más complejos como listas, diccionarios, o instancias de clases, las cosas se ponen más complicadas.

En lugar de tener a los usuarios constantemente escribiendo y debugueando código para grabar tipos de datos complicados, Python te permite usar formato intercambiable de datos popular llamado JSON (JavaScript Object Notation). El módulo estandar llamado **`json`** puede tomar datos de Python con una jerarquía, y convertirlo a representaciones de cadena de caracteres; este proceso es llamado *serializing*. Reconstruir los datos desde la representación de cadena de caracteres es llamado *deserializing*. Entre serialización y deserialización, la cadena de caracteres representando el objeto quizás haya sido guardado en un archivo o datos, o enviado a una máquina distante por una conexión de red.

Nota

El formato JSON es comunmente usado por aplicaciones modernas para permitir intercambiar datos. Muchos programadores están familiarizados con este, lo que lo hace una buena elección por su interoperatividad.

Si tienes un objeto `x`, puedes ver su representación JSON con una simple línea de código:

```
>>> import json
>>> json.dumps([1, 'simple', 'lista'])
'[1, "simple", "lista"]'
```

Otra variante de la función **`dumps()`**, llamada **`dump()`**, simplemente serializa el objeto a un *archivo de texto*. Así que, si `f` es un objeto *archivo de texto* abierto para escritura, podemos hacer:

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

Para decodificar un objeto nuevamente, si `f` es un objeto *archivo de texto* que fue abierto para lectura:

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

La simple técnica de serialización puede manejar listas y diccionarios, pero serializar instancias de clases arbitrarias en JSON requiere un poco de esfuerzo extra. La referencia del módulo `json` contiene una explicación de esto.

Ver también

`pickle` - el módulo pickle

Contrariamente a *JSON*, *pickle* es un protocolo que permite la serialización de arbitrariamente objetos complejos de Python. Por lo tanto, este es específico de Python y no puede ser usado para comunicarse con aplicaciones escritas en otros lenguajes. Es inseguro por defecto: deserializar datos que fueron serializados con pickle desde fuentes inseguras puede ejecutar código arbitrario, si los datos fueron interceptados por un atacante experto.