Usando el intérprete de Python

Invocando al intérprete

Por lo general, el intérprete de Python se instala en `/usr/local/bin/python3.5` en las máquinas dónde está disponible; poner `/usr/local/bin` en el camino de búsqueda de tu intérprete de comandos Unix hace posible iniciarlo ingresando la orden:

```
python3.5
```

...en la terminal. [1] Ya que la elección del directorio dónde vivirá el intérprete es una opción del proceso de instalación, puede estar en otros lugares; consultá a tu Gurú Python local o administrador de sistemas. (Por ejemplo, `/usr/local/python` es una alternativa popular).

En máquinas con Windows, la instalación de Python por lo general se encuentra en `C:\Python35`, aunque se puede cambiar durante la instalación. Para añadir este directorio al camino, podes ingresar la siguiente orden en el prompt de DOS:

```
set path=%path%;C:\python35
```

Se puede salir del intérprete con estado de salida cero ingresando el carácter de fin de archivo (Control-D en Unix, Control-Z en Windows) en el prompt primario. Si esto no funciona, se puede salir del intérprete ingresando: `quit()`.

Las características para editar líneas del intérprete incluyen edición interactiva, sustitución usando el historial y completado de código en sistemas que soportan readline. Tal vez la forma más rápida de detectar si las características de edición están presentes es ingresar Control-P en el primer prompt de Python que aparezca. Si se escucha un beep, las características están presentes; ver Apéndice *Edición de entrada interactiva y sustitución de historial* para una introducción a las teclas. Si no pasa nada, o si aparece `^P`, estas características no están disponibles; solo vas a poder usar backspace para borrar los caracteres de la línea actual.

La forma de operar del intérprete es parecida a la línea de comandos de Unix: cuando se la llama con la entrada estándar conectada a una terminal lee y ejecuta comandos en forma interactiva; cuando es llamada con un nombre de archivo como argumento o con un archivo como entrada estándar, lee y ejecuta un *script* del archivo.

Una segunda forma de iniciar el intérprete es `python -c comando [arg] ...`, que ejecuta las sentencias en *comando*, similar a la opción *-c* de la línea de comandos. Ya que las sentencias de Python suelen tener espacios en blanco u otros caracteres que son especiales en la línea de comandos, es normalmente recomendado citar *comando* entre comillas dobles.

Algunos módulos de Python son también útiles como scripts. Pueden invocarse usando `python -m module [arg] ...`, que ejecuta el código de *module* como si se hubiese ingresado su nombre completo en la línea de comandos.

Cuando se usa un script, a veces es útil correr primero el script y luego entrar al modo interactivo. Esto se puede hacer pasándole la opción *-i* antes del nombre del script.

Todas las opciones de línea de comandos están se descriptas en *Línea de comandos y entorno*.

Pasaje de argumentos

Cuando son conocidos por el intérprete, el nombre del script y los argumentos adicionales son entonces convertidos a una lista de cadenas de texto asignada a la variable `argv` del módulo `sys`. Podés acceder a esta lista haciendo `import sys`. El largo de esta lista es al menos uno; cuando ningún script o argumentos son pasados, `sys.argv[0]` es una cadena vacía. Cuando se pasa el nombre del script con `'-'` (lo que significa la entrada estándar), `sys.argv[0]` vale `'-'`. Cuando se usa *-c command*, `sys.argv[0]` vale `'-c'`. Cuando se usa *-m module*, `sys.argv[0]` toma el valor del nombre completo del módulo. Las opciones encontradas luego de *-c command* o *-m module* no son consumidas por el procesador de opciones de Python pero de todas formas almacenadas en `sys.argv` para ser manejadas por el comando o módulo.

Modo interactivo

Se dice que estamos usando el intérprete en modo interactivo, cuando los comandos son leídos desde una terminal. En este modo espera el siguiente comando con el *prompt primario*, usualmente tres signos mayor-que (>>>); para las líneas de continuación espera con el *prompt secundario*, por defecto tres puntos (`...`). Antes de mostrar el prompt primario, el intérprete muestra un mensaje de bienvenida reportando su número de versión y una nota de copyright:

```
$ python3.5
Python 3.5 (default, Sep 16 2014, 09:25:04)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Las líneas de continuación son necesarias cuando queremos ingresar un constructor multilínea. Como en el ejemplo, mirá la sentencia **`if`**:

```
>>> el_mundo_es_plano = True
>>> if el_mundo_es_plano:
...     print("¡Tené cuidado de no caerte!")
...
¡Tené cuidado de no caerte!
```

Para más información sobre el modo interactivo, ve a *Modo interactivo*.

El intérprete y su entorno

Codificación del código fuente

Por default, los archivos fuente de Python son tratados como codificados en UTF-8. En esa codificación, los caracteres de la mayoría de los lenguajes del mundo pueden ser usados simultáneamente en literales, identificadores y comentarios, a pesar de que la biblioteca estándar usa solamente caracteres ASCII para los identificadores, una convención que debería seguir cualquier código que sea portable. Para mostrar estos caracteres correctamente, tu editor debe reconocer que el archivo está en UTF-8 y usar una tipografía que soporte todos los careacteres del archivo.

También es posible especificar una codificación distinta para los archivos fuente. Para hacer esto, poné una o más lineas de comentarios especiales luego de la linea del `#!` para definir la codificación del archivo fuente:

```
# -*- coding: encoding -*-
```

Con esa declaración, todo en el archivo fuente será tratado utilizando la codificación *encoding* en lugar de UTF-8. La lista de posibles codificaciones se puede encontrar en la Referencia de la Biblioteca de Python, en la sección sobre **`codecs`**.

Por ejemplo, si tu editor no soporta la codificación UTF-8 e insiste en usar alguna otra, digamos Windows-1252, podés escribir:

```
# -*- coding: cp-1252 -*-
```

y usar todos los caracteres del conjunto de Windows-1252 en los archivos fuente. El comentario especial de la codificación debe estar en la *primera o segunda* linea del archivo.

---

1 En Unix, el intérprete de Python 3.x no se instala por default con el ejecutable llamado `python` para que no conflictúe con un ejecutable de Python 2.x que esté instalado simultaneamente.

Una introducción informal a Python

En los siguientes ejemplos, las entradas y salidas son distinguidas por la presencia o ausencia de los prompts (`>>>` y `...`): para reproducir los ejemplos, debés escribir todo lo que esté después del prompt, cuando este aparezca; las líneas que no comiencen con el prompt son las salidas del intérprete. Tené en cuenta que el prompt secundario que aparece por si sólo en una línea de un ejemplo significa que debés escribir una línea en blanco; esto es usado para terminar un comando multilínea.

Muchos de los ejemplos de este manual, incluso aquellos ingresados en el prompt interactivo, incluyen comentarios. Los comentarios en Python comienzan con el carácter numeral, `#`, y se extienden hasta el final físico de la línea. Un comentario quizás aparezca al comienzo de la línea o seguidos de espacios blancos o código, pero no dentro de una cadena de caracteres. Un carácter numeral dentro de una cadena de caracteres es sólo un carácter numeral. Ya que los comentarios son para aclarar código y no son interpretados por Python, pueden omitirse cuando se escriben los ejemplos.

Algunos ejemplos:

```
# este es el primer comentario
spam = 1                 # y este es el segundo comentario
                         # ... y ahora un tercero!
text = "# Este no es un comentario".
```

Usar Python como una calculadora

Vamos a probar algunos comandos simples en Python. Iniciá un intérprete y esperá por el prompt primario, `>>>`. (No debería demorar tanto).

Números

El intérprete actúa como una simple calculadora; podés ingresar una expresión y este escribirá los valores. La sintaxis es sencilla: los operadores `+`, `-`, `*` y `/` funcionan como en la mayoría de los lenguajes (por ejemplo, Pascal o C); los paréntesis (`(` `)`) pueden ser usados para agrupar. Por ejemplo:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # la división simpre retorna un número de punto flotante
1.6
```

Los números enteros (por ejemplo `2`, `4`, `20`) son de tipo **`int`**, aquellos con una parte fraccional (por ejemplo `5.0`, `1.6`) son de tipo **`float`**. Vamos a ver más sobre tipos de números luego en este tutorial.

La división (`/`) siempre retorna un punto flotante. Para hacer *floor division* y obtener un resultado entero (descartando cualquier resultado fraccional) podés usar el operador `//`; para calcular el resto podés usar `%`:

```
>>> 17 / 3  # la división clásica retorna un punto flotante
5.666666666666667
>>>
>>> 17 // 3  # la división entera descarta la parte fraccional
5
```

```
>>> 17 % 3  # el operado % retorna el resto de la división
2
>>> 5 * 3 + 2  # resultado * divisor + resto
17
```

Con Python, es posible usar el operador `**` para calcular potencias [2]:

```
>>> 5 ** 2  # 5 al cuadrado
25
>>> 2 ** 7  # 2 a la potencia de 7
128
```

El signo igual (=) es usado para asignar un valor a una variable. Luego, ningún resultado es mostrado antes del próximo prompt:

```
>>> ancho = 20
>>> largo = 5 * 9
>>> ancho * largo
900
```

Si una variable no está "definida" (con un valor asignado), intentar usarla producirá un error:

```
>>> n  # tratamos de acceder a una variable no definida
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

En el modo interactivo, la última expresión impresa es asignada a la variable _. Esto significa que cuando estés usando Python como una calculadora de escritorio, es más fácil seguir calculando, por ejemplo:

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

Esta variable debería ser tratada como de sólo lectura por el usuario. No le asignes explícitamente un valor; crearás una variable local independiente con el mismo nombre enmascarando la variable con el comportamiento mágico.

Además de **`int`** y **`float`**, Python soporta otros tipos de números, como ser **`Decimal`** y **`Fraction`**. Python también tiene soporte integrado para *números complejos*, y usa el sufijo `j` o `J` para indicar la parte imaginaria (por ejemplo `3+5j`).

Cadenas de caracteres

Además de números, Python puede manipular cadenas de texto, las cuales pueden ser expresadas de distintas formas. Pueden estar encerradas en comillas simples (`'...'`) o dobles (`"..."`) con el mismo resultado [3]. `\` puede ser usado para escapar comillas:

```
>>> 'huevos y pan'  # comillas simples
'huevos y pan'
>>> 'doesn\'t'  # usa \' para escapar comillas simples...
"doesn't"
>>> "doesn't"  # ...o de lo contrario usa comillas doblas
"doesn't"
>>> '"Si," le dijo.'
'"Si," le dijo.'
>>> "\"Si,\" le dijo."
```

```
'"Si," le dijo.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

En el intéprete interactivo, la salida de cadenas está encerrada en comillas y los caracteres especiales son escapados con barras invertidas. Aunque esto a veces luzca diferente de la entrada (las comillas que encierran pueden cambiar), las dos cadenas son equivalentes. La cadena se encierra en comillas dobles si la cadena contiene una comilla simple y ninguna doble, de lo contrario es encerrada en comillas simples. La función **`print()`** produce una salida más legible, omitiendo las comillas que la encierran e imprimiendo caracteres especiales y escapados:

```
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
>>> print('"Isn\'t," she said.')
"Isn't," she said.
>>> s = 'Primerea línea.\nSegunda línea.'  # \n significa nueva línea
>>> s  # sin print(), \n es incluído en la salida
'Primera línea.\nSegunda línea.'
>>> print(s)  # con print(), \n produce una nueva línea
Primera línea.
Segunda línea.
```

Si no querés que los caracteres antepuestos por `\` sean interpretados como caracteres especiales, podés usar *cadenas crudas* agregando una `r` antes de la primera comilla:

```
>>> print('C:\algun\nombre')  # aquí \n significa nueva línea!
C:\algun
ombre
>>> print(r'C:\algun\nombre')  # nota la r antes de la comilla
C:\algun\nombre
```

Las cadenas de texto literales pueden contener múltiples líneas. Una forma es usar triple comillas: `"""..."""` o `'''...'''`. Los fin de línea son incluídos automáticamente, pero es posible prevenir esto agregando una `\` al final de la línea. Por ejemplo:

```
print("""\
Uso: algo [OPTIONS]
     -h                        Muestra el mensaje de uso
     -H nombrehost             Nombre del host al cual conectarse
""")
```

produce la siguiente salida: (nota que la línea inicial no está incluída)

```
Uso: algo [OPTIONS]
     -h                        Muestra el mensaje de uso
     -H nombrehost             Nombre del host al cual conectarse
```

Las cadenas de texto pueden ser concatenadas (pegadas juntas) con el operador `+` y repetidas con `*`:

```
>>> # 3 veces 'un', seguido de 'ium'
>>> 3 * 'un' + 'ium'
'ununium'
```

Dos o más *cadenas literales* (aquellas encerradas entre comillas) una al lado de la otra son automáticamente concatenadas:

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

Esto solo funciona con dos literales, no con variables ni expresiones:

```
>>> prefix = 'Py'
>>> prefix 'thon'  # no se puede concatenar una variable y una cadena literal
  ...
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  ...
SyntaxError: invalid syntax
```

Si querés concatenar variables o una variable con un literal, usá +:

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

Esta característica es particularmente útil cuando querés separar cadenadas largas:

```
>>> texto = ('Poné muchas cadenas dentro de paréntesis '
             'para que ellas sean unidas juntas.')
>>> texto
'Poné muchas cadenas dentro de paréntesis para que ellas sean unidas juntas.'
```

Las cadenas de texto se pueden *indexar* (subíndices), el primer carácter de la cadena tiene el índice 0. No hay un tipo de dato para los caracteres; un carácter es simplemente una cadena de longitud uno:

```
>>> palabra = 'Python'
>>> palabra[0]  # caracter en la posición 0
'P'
>>> palabra[5]  # caracter en la posición 5
'n'
```

Los índices quizás sean números negativos, para empezar a contar desde la dereche:

```
>>> palabra[-1]  # último caracter
'n'
>>> palabra[-2]  # ante último caracter
'o'
>>> palabra[-6]
'P'
```

Nota que -0 es lo mismo que 0, los índice negativos comienzan desde -1.

Además de los índices, las *rebanadas* también están soportadas. Mientras que los índices son usados para obtener caracteres individuales, las *rebanadas* te permiten obtener sub-cadenas:

```
>>> palabra[0:2]  # caracteres desde la posición 0 (incluída) hasta la 2 (excluída)
'Py'
>>> palabra[2:5]  # caracteres desde la posición 2 (incluída) hasta la 5 (excluída)
'tho'
```

Nota como el primero es siempre incluído, y que el último es siempre excluído. Esto asegura que `s[:i] + s[i:]` siempre sea igual a `s`:

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

Los índices de las rebanadas tienen valores por defecto útiles; el valor por defecto para el primer índice es cero, el valor por defecto para el segundo índice es la longitud de la cadena a rebanar.

```
>>> palabra[:2]  # caracteres desde el principio hasta la posición 2 (excluída)
'Py'
```

```
>>> palabra[4:]  # caracterrs desde la posición 4 (incluída) hasta el final
'on'
>>> palabra[-2:] # caracteres desde la ante-última (incluída) hasta el final
'on'
```

Una forma de recordar cómo funcionan las rebanadas es pensar en los índices como puntos *entre* caracteres, con el punto a la izquierda del primer carácter numerado en 0. Luego, el punto a la derecha del último carácter de una cadena de *n* caracteres tienen índice *n*, por ejemplo:

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

La primer fila de números da la posición de los índices 0...6 en la cadena; la segunda fila da los correspondientes índices negativos. La rebanada de *i* a *j* consiste en todos los caracteres entre los puntos etiquetados *i* y *j*, respectivamente.

Para índices no negativos, la longitud de la rebanada es la diferencia de los índices, si ambos entran en los límites. Por ejemplo, la longitud de `palabra[1:3]` es 2.

Intentar usar un índice que es muy grande resultará en un error:

```
>>> palabra[42]  # la palabra solo tiene 6 caracteres
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Sin embargo, índices fuera de rango en rebanadas son manejados satisfactoriamente:

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

Las cadenas de Python no pueden ser modificadas -- son *immutable*. Por eso, asignar a una posición indexada de la cadena resulta en un error:

```
>>> palabra[0] = 'J'
  ...
TypeError: 'str' object does not support item assignment
>>> palabra[2:] = 'py'
  ...
TypeError: 'str' object does not support item assignment
```

Si necesitás una cadena diferente, deberías crear una nueva:

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

La función incorporada `len()` devuelve la longitud de una cadena de texto:

```
>>> s = 'supercalifrastilisticoespialidoso'
>>> len(s)
33
```