Lectura 4: El tutorial de Python (Páginas 8 a 15)

Este material de lectura corresponde a las primeras páginas del libro "El tutorial de Python". En la sección de recursos del curso encontrarás el libro completo.

En estas páginas se muestran a través de ejemplos el uso del lenguaje. Se explica cómo se pueden poner comentarios al código, como realizar las operaciones con números enteros y decimales, como manipular cadenas de caracteres y el uso de listas.

Además se muestra un ejemplo de un pequeño programa explicando las distintas expresiones utilizadas.

No te preocupes si hay muchas cosas que aún no comprendes en detalle, ya que se irán explicando a lo largo del curso.

Cantidad de páginas: 8.

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
```

Hay soporte completo de punto flotante; operadores con operando mezclados convertirán los enteros a punto flotante:

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

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 *númreos 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 derecha:

```
>>> 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
```

Ver también

Tipos integrados

Las cadenas de texto son ejemplos de *tipos secuencias*, y soportan las operaciones comunes para esos tipos.

Tipos integrados

Las cadenas de texto soportan una gran cantidad de métodos para transformaciones básicas y búsqueda.

f-strings

Literales de cadena que tienen expresiones embebidas.

formatstrings

Aquí se da información sobre formateo de cadenas de texto con `str.format()`.

Tipos integrados

Aquí se describe con más detalle las operaciones viejas para formateo usadas cuando una cadena de texto están a la izquierda del operador `%`.

Listas

Python tiene varios tipos de datos *compuestos*, usados para agrupar otros valores. El más versátil es la *lista*, la cual puede ser escrita como una lista de valores separados por coma (ítems) entre corchetes. Las listas pueden contener ítems de diferentes tipos, pero usualmente los ítems son del mismo tipo:

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

Como las cadenas de caracteres (y todos los otros tipos *sequence* integrados), las listas pueden ser indexadas y rebanadas:

```
>>> cuadrados[0]  # índices retornan un ítem
1
>>> cuadrados[-1]
25
>>> cuadrados[-3:]  # rebanadas retornan una nueva lista
[9, 16, 25]
```

Todas las operaciones de rebanado devuelven una nueva lista conteniendo los elementos pedidos. Esto significa que la siguiente rebanada devuelve una copia superficial de la lista:

```
>>> cuadrados[:]
[1, 4, 9, 16, 25]
```

Las listas también soportan operaciones como concatenación:

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

A diferencia de las cadenas de texto, que son *immutable*, las listas son un tipo *mutable*, es posible cambiar un su contenido:

```
>>> cubos = [1, 8, 27, 65, 125]  # hay algo mal aquí
>>> 4 ** 3  # el cubo de 4 es 64, no 65!
64
```

```
>>> cubos[3] = 64  # reemplazar el valor incorrecto
>>> cubos
[1, 8, 27, 64, 125]
```

También podés agregar nuevos ítems al final de la lista, usando el *método* **`append()`** (vamos a ver más sobre los métodos luego):

```
>>> cubos.append(216)  # agregar el cubo de 6
>>> cubos.append(7 ** 3)  # y el cubo de 7
>>> cubos
[1, 8, 27, 64, 125, 216, 343]
```

También es posible asignar a una rebanada, y esto incluso puede cambiar la longitud de la lista o vaciarla totalmente:

```
>>> letras = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letras
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # reemplazar algunos valores
>>> letras[2:5] = ['C', 'D', 'E']
>>> letras
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # ahora borrarlas
>>> letras[2:5] = []
>>> letras
['a', 'b', 'f', 'g']
>>> # borrar la lista reemplzando todos los elementos por una lista vacía
>>> letras[:] = []
>>> letras
[]
```

La función predefinida **`len()`** también sirve para las listas:

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

Es posible anidar listas (crear listas que contengan otras listas), por ejemplo:

```
>>> 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'
```

Primeros pasos hacia la programación

Por supuesto, podemos usar Python para tareas más complicadas que sumar dos y dos. Por ejemplo, podemos escribir una subsecuencia inicial de la serie de *Fibonacci* así:

```
>>> # Series de Fibonacci:
... # la suma de dos elementos define el siguiente
... a, b = 0, 1
>>> while b < 10:
...     print(b)
...     a, b = b, a+b
...
```

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

Este ejemplo introduce varias características nuevas.

- La primer línea contiene una *asignación múltiple*: las variables `a` y `b` toman en forma simultanea los nuevos valores 0 y 1. En la última linea esto se vuelve a usar, demostrando que las expresiones a la derecha son evaluadas antes de que suceda cualquier asignación. Las expresiones a la derecha son evaluadas de izquierda a derecha.
- El bucle **`while`** se ejecuta mientras la condición (aquí: `b < 10`) sea verdadera. En Python, como en C, cualquier entero distinto de cero es verdadero; cero es falso. La condición también puede ser una cadena de texto o una lista, de hecho cualquier secuencia; cualquier cosa con longitud distinta de cero es verdadero, las secuencias vacías son falsas. La prueba usada en el ejemplo es una comparación simple. Los operadores estándar de comparación se escriben igual que en C: `<` (menor qué), `>` (mayor qué), `==` (igual a), `<=` (menor o igual qué), `>=` (mayor o igual qué) y `!=` (distinto a).
- El *cuerpo* del bucle está *sangrado*: la sangría es la forma que usa Python para agrupar declaraciones. En el intérprete interactivo debés teclear un tab o espacio(s) para cada línea sangrada. En la práctica vas a preparar entradas más complicadas para Python con un editor de texto; todos los editores de texto decentes tienen la facilidad de agregar la sangría automáticamente. Al ingresar una declaración compuesta en forma interactiva, debés finalizar con una línea en blanco para indicar que está completa (ya que el analizador no puede adivinar cuando tecleaste la última línea). Notá que cada línea de un bloque básico debe estar sangrada de la misma forma.
- La función **`print()`** escribe el valor de el o los argumentos que se le pasan. Difiere de simplemente escribir la expresión que se quiere mostrar (como hicimos antes en los ejemplos de la calculadora) en la forma en que maneja múltiples argumentos, cantidades en punto flotante, y cadenas. Las cadenas de texto son impresas sin comillas, y un espacio en blanco es insertado entre los elementos, así podés formatear cosas de una forma agradable:

  ```
  >>> i = 256*256
  >>> print('El valor de i es', i)
  El valor de i es 65536
  ```

  El parámetro nombrado *end* puede usarse para evitar el salto de linea al final de la salida, o terminar la salida con una cadena diferente:

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

---

2 Debido a que `**` tiene mayor precedencia que `-`, `-3**2` será interpretado como `-(3**2)` y eso da como resultado `-9`. Para evitar esto y obtener 9, podés usar `(-3)**2`.

3 A diferencia de otros lenguajes, caracteres especiales como `\n` tiene el mismo significado con simple (`'...'`) y doble (`"..."`) comillas. La única diferencia entre las dos es que dentro de las comillas simples no tenés la necesitada de escapar `"` (pero tenés que escapar `\'`) y viceversa.