Guía de Estudios PCFI161 202010

CL, TC, y JP, Universidad Adnrés Bello 05/20/2020

Imprimir en pantalla con Python 3

Imprimir en pantalla en Python, también conocido como `stdout` (*standard output*), es uno de los elementos principales para *depurar* (limpiar de *bugs*, o errores) nuestro código. Es por ello que acá revisaremos algunos de los modelos más utilizados para imprimir en pantalla mediante el uso de Python.

El siguiente código define unas variables y las muestra en pantalla.

```
#Definimos variables de distinto tipo,
a = 5
b = 6.0
c = "un texto cualquiera"
d = 1E-3

#Imprimimos en pantalla estas variables
print(a)
print(b)
print(c)
print(d)
```

Podemos notar que la isntrucción `print` automáticamente genera una nueva linea luego de ser ejecutada. Además, la variable asignada es convertida automaticamente a un `string` –un texto–, que es lo que realmente vemos nosotros en la pantalla.

Si queremos que la impresión en pantalla no termine con una línea nueva, podemos utilizar la opción `sep`, dentro de nuestra función `print`. Como se muestra en el siguiente código

```
#Definimos variables de distinto tipo,
a, b = 5, 6.0
c = "un texto cualquiera"
d = 1E-3

#Imprimimos en pantalla estas variables
print(a, end='-')
print(b, end='-')
print(c, end='\t')
print(d, end='\n')
```

Acá el fin de la línea es asignado por el usuario, `'-'` corresponde al string `-`, el valor `\t` es un tabulador, y `\n` es el valor por *default* en `print` que corresponde una línea nueva.

Imprimiendo frases más complejas

Si bien existen muchas variantes para escribir frases más complejas en Python, en esta guía pondremos énfasis al estilo `f-string` de Python. El estilo `f-string` es un estilo moderno que se encuentra disponible desde Python 3.6 y corresponde a un formato moderno con ventajas sobre `%-format` y `str.format()` [1].

Para escribir un texto simple, uno no necesita usar `f-string`, el siguiente código muestra la misma linea de texto en dos ocasiones en la pantalla, en la primera se utiliza `f-string` y la segunda no, pero cumplen la misma función.

```
print(f'Este es un texto simple')
print('Este es un texto simple')
```

La ventaja principal de los formatos se observa principalmente cuando involucramos variables en el texto. Consideremos ahora el siguiente código que involucra variables en la escritura de contenido en pantalla.

```
#Definimos variables de distinto tipo,
a, b = 5, 6.0
c, d = "un texto cualquiera", 1E-3

#Imprimimos en pantalla estas variables
print(f'El valor de a es {a}')
print(f'El valor de b es {b}')
print(f'El contenido de c es {c}')
print(f'El valor de d es {d}')
```

Es evidente la ventaja en no convertir las variables a string, y a la simplesa de describir las variables simplemente entre corchetes cursivos. Pero además es posible combinar disstintos tipos, así como también realizar operaciones directas entre las variables en la sintaxis. Por ejemplo consideremos el siguiente código:

```
#Definimos variables de distinto tipo,
a, b = 5, 6.0
c, d = "un texto cualquiera", 1E-3

#Imprimimos en pantalla estas variables
print(f'El valor de a*b es {a * b}')
print(f'El valor de d es {d} y el contenido de c es {c}')
```

Uno además puede generar sistemas más complejos o definir textos a partir de variables en un código. En el siguiente código usted puede observar la definición de variables, y la construccion de una nueva variable (un *string*) utilizando el formato `f-string` de Python.

[1] https://realpython.com/python-f-strings/

```python
#Definimos variables de distinto tipo,
a, b = 5.0, 6.0
c, d = 7.0, 8.0

#Generamos una nueva variable
s = f'Este es un texto que muestra {a} * {b} = {a*b}, y {c} * {d} = {c*d}'

#Imprimimos en pantalla la variable definida
print(s)
```

Slicing de listas y arreglos

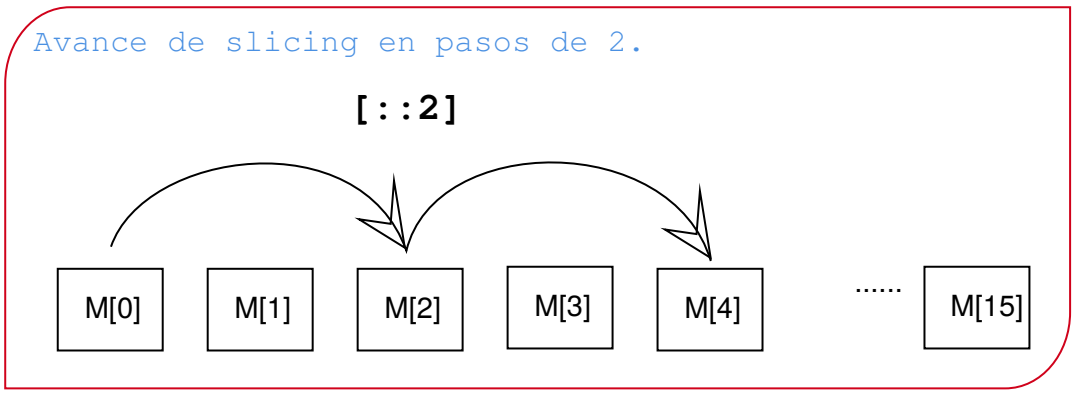
Si bien hemos cubierto lo básico en el slicing de listas y arreglos en Python, existen un par de *hints* que siempre es bueno tener en consideración a la hora de trabajar con slicing. Es importante hacer notar que *slicing* es una herramienta que podemos utilizar tanto en listas como en arreglos de Python. Para el contexto de est curso, hemos decidido que los ejemplos que se muestran en esta guía serán realizados con arreglos (libreria `numpy`).

Veamos el caso simple de un arreglo unidimensional del cual extraemos sub-arreglos.

```python
import numpy as np

#Definimos una matriz de 4x4
M = np.array([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]], dtype=float)

#Las indices son numerados de 0 a 15.

s1 = M[0:5] #Arreglo [1,2,3,4,5]
s2 = M[3:6] #Arreglo [4,5,6]
s3 = M[3:] #Arreglo desde el indice 3 (M[3]) hasta al final
s4 = M[:5] #Arreglo desde el inicio hasta el indice 4 (M[4])
s5 = M[-1] #El ultimo valor del arreglo
s6 = M[0:17:2] # Desde M[0] hasta M[16] en intervalos de 2
s7 = M[::2] #Igual que s6
```

Quizás inicialmente cuesta reconocer el patrón, pero luego de revisar el código en detalle y cada una de las variables (use `print` para ver las variables en su código), es posible observar algo como `[m:n:o]` como la base para cualquier slicing.

Los slicing suelen ser descritos de su forma general `[m:n:o]`. En este formato, `m` corresponde al índice inicial del arreglo. `n` corresponde al valor más uno del índice final del arreglo, es decir nuestra extracción de datos llegará hasta el índice `n-1`. El valor `o` corresponde a un paso iterador en el arreglo, en muchas ocasiones este no se utiliza y tiene un valor por *default* de 1.



El diagrama muestra como serían los avances de nuestro *slicing* para el caso en que el iterador sea ajustado a dos. Una recomendación inicial es probar el código anterior y realizar variados *slices* antes de pasar a los slicing bidimensionales (*slices* en matrices).

Slicing en más de una dimensión

Consideremos una matriz construida a partir de listas, en donde realizaremos extracción de algunos elementos fundamentales mediante *slicing*. Como se muestra en el siguiente código

```
1  import numpy as np
2
3  #Definimos una matriz de 4x4
4  M = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12], [13,14,15,16]], dtype=float)
5
6  #Las filas y columnas son numeradas de 0 a 3 -> (0, 1, 2, 3)
7
8  f1 = M[0,:] #Fila 0 completa
9  f2 = M[1,:] #Fila 1 completa
10 c1 = M[:,0] #Columna 0 completa
11 c2 = M[:,1] #Columna 1 completa
12 m = M[2:3,2:3] #submatriz
```