

prog_datasci_2_python

August 7, 2019

1 Programación para *Data Science*

1.1 Unidad 2: Breve introducción a la programación en Python

1.1.1 Instrucciones de uso

A continuación se irá presentando la sintaxis básica del lenguaje de programación Python junto a ejemplos interactivos.

1.2 Variables y tipos de variables

Podemos entender una variable como un contenedor en el que podemos poner nuestros datos a fin de guardarlos y tratarlos más adelante. En Python, las variables no tienen tipo, es decir, no tenemos que indicar si la variable será un número, un carácter, una cadena de caracteres o una lista, por ejemplo. Además, las variables pueden ser declaradas e inicializadas en cualquier momento, a diferencia de otros lenguajes de programación.

Para declarar una variable, utilizamos la expresión *nombre_de_variable = valor*. Se recomienda repasar el documento [PEP-8](#) para definir nombres de variables correctamente, pero *grosso modo*, evitaremos utilizar mayúscula en la inicial, separaremos las diferentes palabras con el carácter `_` y no utilizaremos acentos ni caracteres específicos de nuestra condificación como el símbolo del `€` o la `ñ`, por ejemplo.

Veamos unos cuantos ejemplos de declaraciones de variables y cómo usarlas:

```
[1]: # Declaramos una variable de nombre 'variable_numerica' que contiene el valor
      ↪ entero 12.
variable_numerica = 12

# Declaramos una variable de nombre 'monstruo' que contiene el valor 'Godzilla'.
monstruo = 'Godzilla'

# Declaramos una variable de nombre 'planetas' que es una lista de cadenas de
      ↪ caracteres.
planetas = ['Mercurio', 'Venus', 'Tierra', 'Marte']

[2]: mi_edad = 25
      mi_edad_en_5 = mi_edad + 5
      # 'Imprimimos' el valor calculado que será, efectivamente, 30.
      print(mi_edad_en_5)
```

Los tipos nativos de datos que una variable en Python puede contener son: números enteros (int), números decimales (float), números complejos (complex), cadena de caracteres (string), listas (list), tuplas (tuple) y diccionarios (dict). Veamos uno por uno cada uno de estos tipos:

```
[3]: # Un número entero
int_var = 1
another_int_var = -5
# Podemos sumarlos, restarlos, multiplicarlos o dividirlos.
print(int_var + another_int_var)
print(int_var - another_int_var)
print(int_var * another_int_var)
print(int_var / another_int_var)

# También podemos realizar la división entera.
# Como solo tratamos con números enteros, no tendrá parte decimal.
print(int_var // another_int_var)
```

```
-4
6
-5
-0.2
-1
```

El comportamiento del operador / es una de las diferencias entre Python 2 y Python 3. Mientras que en Python 3, el operador / realiza la división real entre dos números enteros (fijaros que 1 / -5 da como resultado 0.2), en Python 2 realizaba la división entera (por lo que el resultado de ejecutar 1 / -5 en Python 2 sería -1). Notad que usamos el operador // para expresar la división entera en Python 3.

```
[4]: # Un número decimal o 'float'
float_var = 2.5
another_float_var = .7
# Convertimos un número entero en uno decimal mediante la función 'float()'.
encore_float = float(7)
# Podemos hacer lo mismo en sentido contrario con la función 'int()'.
new_int = int(encore_float)

# Podemos hacer las mismas operaciones que en el caso de los números enteros:
print(another_float_var + float_var)
print(another_float_var - float_var)
print(another_float_var * float_var)
print(another_float_var / float_var)
print(another_float_var // float_var)
```

```
3.2
-1.8
1.75
```

```
0.27999999999999997
0.0
```

```
[5]: # Un número complejo
complex_var = 2+3j
# Podemos acceder a la parte imaginaria o a la parte real:
print(complex_var.imag)
print(complex_var.real)
```

```
3.0
2.0
```

```
[6]: # Cadena de caracteres
my_string = 'Hello, Bio! ñç'
print(my_string)
```

```
Hello, Bio! ñç
```

Fijaros que podemos incluir caracteres unicode (como ño ç) en las cadenas. Esto también es una novedad de Python 3 (las variables de tipo str son ahora UTF-8).

```
[7]: # Podemos concatenar dos cadenas utilizando el operador '+'.
same_string = 'Hello, ' + 'Bio' + '!' + ' ñç'
print(same_string)

# En Python también podemos utilizar wildcards como en la función sprintf de C.
→Por ejemplo:
name = "Guido"
num_emails = 5
print("Hello, %s! You've got %d new emails" % (name, num_emails))
```

```
Hello, Bio! ñç
```

```
Hello, Guido! You've got 5 new emails
```

En el ejemplo anterior, hemos sustituido en el string la cadena %s por el contenido de la variable name, que es un string, y %d por num_emails, que es un número entero. Podríamos también utilizar %f para números decimales (podríamos indicar la precisión por ejemplo con %5.3f, el número tendría un tamaño total de cinco cifras y tres serían para la parte decimal). Hay muchas otras posibilidades, pero deberemos tener en cuenta el tipo de variable que queremos sustituir. Por ejemplo, si utilizamos %d y el contenido es string, Python devolverá un mensaje de error. Para evitar esta situación, será recomendado el uso de la función str() para convertir el valor a string.

También podemos mostrar el contenido de las variables sin especificar su tipo, usando format:

```
[8]: print("Hello, {}! You've got {} new emails".format(name, num_emails))
```

```
Hello, Guido! You've got 5 new emails
```

Ahora vamos a presentar otros tipos de datos nativos más complejos: listas, tuplas y diccionarios:

```
[9]: # Definimos una lista con el nombre de los planetas (string).
planets = ['Mercury', 'Venus', 'Earth', 'Mars',
           'Jupiter', 'Saturn', 'Uranus', 'Neptune']
# También puede contener números.
prime_numbers = [2, 3, 5, 7]

# Una lista vacía
empty_list = []

# O una mezcla de cualquier tipo:
sandbox = ['3', 'a string', ['a list inside another list', 'second item'], 7.5]
print(sandbox)
```

```
['3', 'a string', ['a list inside another list', 'second item'], 7.5]
```

```
[10]: # Podemos añadir elementos a una lista.
planets.append('Pluto')
print(planets)
```

```
['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune',
'Pluto']
```

```
[11]: # O podemos eliminar elementos.
planets.remove('Pluto')
print(planets)
```

```
['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
```

```
[12]: # Podemos eliminar cualquier elemento de la lista.
planets.remove('Venus')
print(planets)
```

```
['Mercury', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
```

```
[13]: # Siempre que añadamos, será al final de la lista. Una lista está ordenada.
planets.append('Venus')
print(planets)
```

```
['Mercury', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune', 'Venus']
```

```
[14]: # Si queremos ordenarla alfabéticamente, podemos utilizar la función 'sorted()'.
print(sorted(planets))
```

```
['Earth', 'Jupiter', 'Mars', 'Mercury', 'Neptune', 'Saturn', 'Uranus', 'Venus']
```

```
[15]: # Podemos concatenar dos listas:
monsters = ['Godzilla', 'King Kong']
more_monsters = ['Cthulu']
print(monsters + more_monsters)
```

```
['Godzilla', 'King Kong', 'Cthulu']
```

```
[16]: # Podemos concatenar una lista a otra y guardarla en la misma lista:
monsters.extend(more_monsters)
print(monsters)
```

```
['Godzilla', 'King Kong', 'Cthulu']
```

```
[17]: # Podemos acceder a un elemento en concreto de la lista:
print(monsters[0])
# El primer elemento de una lista es el 0, por lo tanto, el segundo será el 1:
print(monsters[1])
# Podemos acceder al último elemento mediante números negativos:
print(monsters[-1])
# Penúltimo:
print(monsters[-2])
```

```
Godzilla
King Kong
Cthulu
King Kong
```

```
[18]: # También podemos obtener partes de una lista mediante la técnica de 'slicing'.
planets = ['Mercury', 'Venus', 'Earth', 'Mars',
           'Jupiter', 'Saturn', 'Uranus', 'Neptune']

# Por ejemplo, los dos primeros elementos:
print(planets[:2])
```

```
['Mercury', 'Venus']
```

```
[19]: # O los elementos entre las posiciones 2 y 4
print(planets[2:5])
```

```
['Earth', 'Mars', 'Jupiter']
```

Fijaros en este último ejemplo: en la posición 2 encontramos el tercer elemento de la lista ('Earth') ya que la lista empieza a indexarse en 0. Además, el último elemento indicado (la posición 5) no se incluye.

```
[20]: # O los elementos del segundo al penúltimo:
print(planets[1:-1])
```

```
['Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus']
```

La técnica de **slicing** es muy importante y nos permite manejar listas de una forma muy sencilla y potente. Será imprescindible dominarla para muchos de los problemas que tendremos que resolver en el campo de la Ciencia de los Datos.

```
[21]: # Podemos modificar un elemento en concreto de una lista.
monsters = ['Godzilla', 'King Kong', 'Cthulu']
monsters[-1] = 'Kraken'
print(monsters)
```

```
['Godzilla', 'King Kong', 'Kraken']
```

```
[22]: # Una tupla es un tipo muy parecido a una lista, pero es immutable, es decir,
      ↪ una vez declarada no podemos añadir
      # ni eliminar elementos:
      birth_year = ('Stephen Hawking', 1942)
      # Si ejecutamos la siguiente línea, obtendremos un error de tipo 'TypeError'.
      birth_year[1] = 1984
```

↳ `TypeError` `Traceback (most recent call last)`

```
<ipython-input-22-de118f1db30e> in <module>
    3 birth_year = ('Stephen Hawking', 1942)
    4 # Si ejecutamos la siguiente línea, obtendremos un error de tipo
↳ 'TypeError'.
----> 5 birth_year[1] = 1984
```

```
TypeError: 'tuple' object does not support item assignment
```

Los errores en Python suelen ser muy informativos. Una búsqueda en Internet nos ayudará en la gran mayoría de problemas que podamos tener.

```
[23]: # Un string también es considerado una lista de caracteres
      # Así pues, podemos acceder a una posición determinada (aunque no modificarla):
      name = 'Albert Einstein'
      print(name[5])
```

```

# Podemos usar slicing también en cadenas de caracteres
print(name[7:15])

# Podemos separar por el carácter que consideremos un string. En este caso, por
→ el espacio en blanco, utilizando
# la función split().
n, surname = name.split()
print(surname)

# Y podemos convertir un determinado string en una lista de caracteres
→ fácilmente:
chars = list(surname)
print(chars)

# Para unir los diferentes elementos de una lista mediante un carácter, podemos
→ utilizar la función join():
print(''.join(chars))
print('.'.join(chars))

```

```

t
Einstein
Einstein
['E', 'i', 'n', 's', 't', 'e', 'i', 'n']
Einstein
E.i.n.s.t.e.i.n

```

[24]:

```

# El operador ',' es el creador de tuplas. Por ejemplo, el típico problema de
→ intercambiar los valores de dos variables
# en Python puede ser resuelto en una línea de forma muy elegante utilizando
→ tuplas (se trata de un idiom):
a = 5
b = -5
a, b = b, a
print(a)
print(b)

```

```

-5
5

```

El anterior ejemplo es un *idiom* típico de Python. En la tercera línea, creamos una tupla (a,b) a la que asignamos los valores uno por uno de la tupla (b,a). Los paréntesis no son necesarios y por eso queda una notación tan reducida.

Para acabar, presentaremos los diccionarios, una estructura de datos muy útil en la que asignamos un valor a una clave en el diccionario:

[25]:

```

# Códigos internacionales de algunos países. La clave o 'key' es el código de
→ país y el valor, su nombre:
country_codes = {34: 'Spain', 376: 'Andorra', 41: 'Switzerland', 424: None}

```

```
# Podemos buscar
my_code = 34
country = country_codes[my_code]
print(country)
```

Spain

```
[26]: # Podemos obtener todas las claves:
print(country_codes.keys())
```

dict_keys([34, 376, 41, 424])

```
[27]: # O los valores:
print(country_codes.values())
```

dict_values(['Spain', 'Andorra', 'Switzerland', None])

Es muy importante notar que los valores que obtenemos de las claves o al imprimir un **diccionario no están ordenados**. Es un error muy común suponer que el diccionario se guarda internamente en el mismo orden en el que fue definido y será una fuente de error habitual no tenerlo en cuenta.

```
[28]: # Podemos modificar valores en el diccionario o añadir nuevas claves.

# Definimos un diccionario vacío. country_codes = dict() es una notación
→equivalente:
country_codes = {}

# Añadimos un elemento:
country_codes[34] = 'Spain'

# Añadimos otro:
country_codes[81] = 'Japan'

print(country_codes)
```

{34: 'Spain', 81: 'Japan'}

```
[29]: # Modificamos el diccionario:
country_codes[81] = 'Andorra'

print(country_codes)
```

{34: 'Spain', 81: 'Andorra'}


```
[30]: # Podemos asignar el valor vacío a un elemento:
country_codes[81] = None

print(country_codes)
```

```
{34: 'Spain', 81: None}
```

Los valores vacíos nos serán útiles para declarar una variable de la que no sepamos qué valor o qué tipo de valor contendrá y para hacer comparaciones entre variables. Típicamente, los valores vacíos son `None` o `' '` en el caso de las cadenas de caracteres.

```
[31]: # Podemos asignar el valor de una variable a otra. Es importante que se
      ↪entiendan las siguientes líneas:
a = 5
b = 1
print(a, b)
# b contiene la 'dirección' del contenedor al que apunta 'a'.
b = a
print(a, b)
```

```
5 1
5 5
```

```
[32]: # Veamos ahora qué pasa si modificamos el valor de a o b:
a = 6
print(a, b)
b = 7
print(a, b)
```

```
6 5
6 7
```

Hasta aquí hemos presentado cómo declarar y utilizar variables. Recomendamos la lectura de la introducción [de la documentación oficial en línea](#) para fijar los conocimientos explicados.