

# Programación para *Data Science*

## Unidad 3: Estructuras de control y funciones en Python

### Instrucciones de uso

En el módulo anterior introducimos conceptos básicos sobre variables y su uso en Python. En este módulo estudiaremos conceptos más avanzados como las instrucciones de flujo de ejecución (for, while, if), cómo definir y utilizar funciones, cómo leer y escribir ficheros y cómo organizar el código.

### Iteración y operaciones lógicas

En la gran mayoría de ocasiones, tendremos que manipular nuestros datos y para ello utilizaremos los conceptos de iteración y las operaciones lógicas. Las operaciones lógicas nos permiten comparar valores entre variables (mayor, menor, igualdad) y la iteración ir visitando uno a uno los elementos de una lista, tupla, diccionario o cualquier estructura de datos que sea susceptible de secuenciarse.

In [1]:

```
# Las operaciones lógicas tendrán como resultado un valor cierto (True) o falso (False):  
  
a = 5  
b = 1  
  
# ¿Es el valor de a mayor que b?  
print(a > b)
```

True

In [2]:

```
# ¿Es el valor de a menor que b?  
print(a < b)
```

False

In [3]:

```
b = 5  
# ¿Es el valor de b igual al de a?  
print(b == a)
```

True

In [4]:

```
# Otros operadores lógicos disponibles son menor o igual '<=', mayor o igual '>=' o la negación 'not'.  
print(a <= b)  
print(a >= b)  
  
a = False  
print(not a)
```

True

True

True

También podemos alterar el flujo de ejecución de nuestro programa utilizando las estructuras *if...else* o *if...elif...else*. Veamos unos cuantos ejemplos:

In [5]:

```
a = 5
b = 6
if a > b:
    print('a es mayor que b')
else:
    print('a es menor o igual que b')
```

a es menor o igual que b

In [6]:

```
a = 5
b = 5
if a > b:
    print('a es mayor que b')
elif a < b:
    print('a es menor que b')
else:
    print('a es igual a b')
```

a es igual a b

En Python existen solo dos formas de iterar una secuencia: mediante **for** o mediante **while**. La primera de las opciones, *for*, iterará uno por uno los elementos contenidos en una lista. En el caso de *while*, iteraremos mientras la condición de permanencia en el bucle se cumpla. Veamos unos ejemplos:

In [7]:

```
monsters = ['Kraken', 'Leviathan', 'Uroborus', 'Hydra']

# Primer método iterando mediante for:
for monster in monsters:
    print(monster)

print()

# Segundo método. La función especial 'enumerate' nos devuelve una tupla en la que el primer elemento es un
# índice que empieza en 0 y aumenta de 1 en 1 y el segundo elemento es el valor de la posición en la lista:
for i, monster in enumerate(monsters):
    print(i, monster)
```

Kraken  
Leviathan  
Uroborus  
Hydra

0 Kraken  
1 Leviathan  
2 Uroborus  
3 Hydra

In [8]:

```
# También podríamos iterar la lista mediante while, pero es una forma mucho menos idiomática en Python
# y preferiremos siempre la opción de for:
i = 0
# Mientras que el índice 'i' sea menor que la longitud de la lista 'monsters':
while i < len(monsters):
    # Imprime el valor de la lista en la posición 'i'.
    print(i, monsters[i])
    # No nos olvidemos de actualizar el valor de 'i' sumándole 1 o tendremos un bucle infinito.
    i += 1
```

0 Kraken  
1 Leviathan  
2 Uroborus  
3 Hydra

2 Uroborus  
3 Hydra

En este momento seríamos capaces de calcular la serie de Fibonacci hasta un determinado valor:

In [9]:

```
# Calculamos el valor de la serie hasta un valor n=100.
n = 100

a, b = 0, 1
while a < n:
    print(a, end = " ")
    a, b = b, a+b
```

0 1 1 2 3 5 8 13 21 34 55 89

En Python disponemos de una función muy útil para generar una secuencia de números, que podemos utilizar de diferentes formas:

In [10]:

```
# Podemos usar la función range para obtener una lista de números:
list(range(10))
```

Out[10]:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Fijaros que `range` no devuelve directamente una lista, sino que devuelve un tipo propio del mismo nombre. Por este motivo, para obtener una lista hay que hacer una conversión de tipo usando `list()`. Esto es una novedad en Python 3, ya que en Python 2 la función `range` devolvía una lista.

In [11]:

```
# Visualizamos el tipo de retorno de range
print(type(range(10)))
print(type(list(range(10))))
```

```
<class 'range'>
<class 'list'>
```

Veamos algunos de los usos más habituales de `range`:

In [12]:

```
# Podemos utilizar range para iterar:
for i in range(10):
    print(i, end = " ")

print()
```

0 1 2 3 4 5 6 7 8 9

In [13]:

```
# Podemos definir el rango de acción.

# Por ejemplo, especificando únicamente el final como hemos hecho anteriormente
for i in range(10):
    print(i, end = " ")

print()

# especificando inicio y final:
for i in range(5, 10):
    print(i, end = " ")
```

```
print()

# o especificar también el salto entre cada valor del resultado:
for i in range(5, 10, 3):
    print(i, end = " ")
```

```
0 1 2 3 4 5 6 7 8 9
5 6 7 8 9
5 8
```

Siempre que no necesitemos explícitamente una lista, usaremos directamente el tipo `range` (sin hacer la conversión a lista), ya que esto es en general más eficiente (ahorra memoria). Solo cuando necesitemos una lista, (por ejemplo, para visualizar el resultado como hemos hecho en el primer ejemplo) convertiremos el resultado de `range` a lista.

In [14]:

```
# También es posible iterar en un diccionario:
country_codes = {34: 'Spain', 376: 'Andorra', 41: 'Switzerland', 424: None}

# Por clave:
for country_code in country_codes.keys():
    print(country_code)
print()

# Por valor:
for country in country_codes.values():
    print(country)
print()

# Por los dos a la vez:
for country_code, country in country_codes.items():
    print(country_code, country)
```

```
34
376
41
424
```

```
Spain
Andorra
Switzerland
None
```

```
34 Spain
376 Andorra
41 Switzerland
424 None
```

## Funciones

Otra forma muy importante de organizar el flujo de ejecución es encapsulando cierta porción de código en una función reutilizable. Una función en Python utiliza el mismo concepto que una función matemática. Por ejemplo, imaginemos la función matemática:

$suma(x, y) = x + y$

En Python podemos definir la misma función de la siguiente forma:

In [15]:

```
# La función suma se define mediante la palabra especial 'def' y tiene dos argumentos: 'x' e 'y':
def suma(x, y):
    # Devolvemos el valor de la suma.
    return x + y

# En este momento, podemos llamarla con cualquier valor:
print(suma(2, 4))
print(suma(5, -5))
print(suma(3.5, 2.5))
```

```
6
n
```

v  
6.0

In [16]:

```
# Podemos definir una función que no haga nada utilizando la palabra especial 'pass':  
  
def dummy():  
    pass  
  
dummy()
```

In [17]:

```
# Podríamos volver a definir el trozo de código de la secuencia de Fibonacci como una función:  
  
def fibonacci(n=100):  
    a, b = 0, 1  
    while a < n:  
        print(a, end = " ")  
        a, b = b, a+b  
  
fibonacci(10)
```

0 1 1 2 3 5 8

In [18]:

```
# En el anterior ejemplo, hemos definido que el argumento n tenga un valor por defecto. Esto es mu  
y útil  
# en los casos en los que utilicemos la función siempre con un mismo valor, queramos dejar constan  
cia de un  
# caso de ejemplo o por defecto. En este caso, podemos ejecutar la función sin pasarle ningún valo  
r:  
fibonacci()
```

0 1 1 2 3 5 8 13 21 34 55 89

In [19]:

```
# Podemos definir parte de los argumentos con valores por defecto y otro sin.  
# Los argumentos sin valor por defecto siempre tienen que estar más a la izquierda en la definició  
n de la función:  
  
def potencia(a, b=2):  
    # Por defecto, elevaremos al cuadrado.  
    return a**b  
  
print(potencia(3))  
print(potencia(2, 3))
```

9  
8

Recomendamos la lectura de la [documentación oficial](#) para acabar de fijar conocimientos.

## Leer y escribir desde ficheros

Una tarea habitual es leer líneas de un fichero o escribir líneas en un fichero. En Python, leer y escribir ficheros se hace a través de la librería `os`. Una librería es un conjunto de código que tiene cierto sentido agrupar para ser usado por otra gente. En nuestro caso, `os` es una librería que agrupa funciones relacionadas con el sistema operativo ( `operating system`). Para cargar una librería, utilizaremos la palabra reservada **import**. Más adelante explicaremos algunas particularidades de utilizar e importar librerías. A continuación `os` explicamos cómo escribir y leer un fichero:

In [20]:

```
# Primero de todo, vamos a escribir en un fichero.  
import os
```

```

# Abrimos un fichero de nombre 'a_file.txt' para escritura (de ahí la 'w', 'writing').
# Lo asignamos a un objeto para manejar el fichero de nombre 'out':
out = open('a_file.txt', 'w')
# Vamos a escribir 10 líneas, cada una con un número de 0 a 9.
for i in range(10):
    # La siguiente línea escribe en el fichero todo lo que pongamos dentro de out.write()
    # En nuestro caso, es un string del tipo '0\n', '1\n', etc. Esto lo conseguimos utilizando las
    wildcards %d y %s.
    # %s representa un string o cadena de caracteres y %d un número entero.
    # Concatenamos en este caso un número con un string que se trata del salto de línea,
    os.linesep, que equivale en Linux
    # a '\n':
    out.write("Línea %d%s" % (i, os.linesep))

out.close()

```

In [21]:

```

# Ahora vamos a leer el fichero que acabamos de escribir de tres formas distintas:

# Primer método
f = open('a_file.txt')
for line in f:
    print(line, end = "")
f.close()
print()

# Segundo método
f = open('a_file.txt')
lines = f.readlines()
for line in lines:
    print(line, end = "")
f.close()
print()

# Tercer método
with open('a_file.txt') as f:
    for line in f:
        print(line, end = "")

```

Línea 0  
 Línea 1  
 Línea 2  
 Línea 3  
 Línea 4  
 Línea 5  
 Línea 6  
 Línea 7  
 Línea 8  
 Línea 9

Línea 0  
 Línea 1  
 Línea 2  
 Línea 3  
 Línea 4  
 Línea 5  
 Línea 6  
 Línea 7  
 Línea 8  
 Línea 9

Línea 0  
 Línea 1  
 Línea 2  
 Línea 3  
 Línea 4  
 Línea 5  
 Línea 6  
 Línea 7  
 Línea 8  
 Línea 9

## Organización del código

Un módulo de Python es cualquier fichero con extensión `.py` que esté bajo la ruta del path de Python. El path de Python puede consultarse importando la librería `sys`:

In [22]:

```
import sys
print(sys.path)

['/usr/lib/python36.zip', '/usr/lib/python3.6', '/usr/lib/python3.6/lib-dynload', '',
'/usr/local/lib/python3.6/dist-packages', '/usr/lib/python3/dist-packages',
'/usr/local/lib/python3.6/dist-packages/IPython/extensions', '/home/datasci/.ipython']
```

Por defecto, Python también mira las librerías que se hayan definido en la variable de entorno `$PYTHONPATH` (esto puede cambiar ligeramente en un [entorno Windows](#)).

Un paquete en Python es cualquier directorio que contenga un fichero especial de nombre `__init__.py` (este fichero estará vacío la mayoría de veces).

Un módulo puede contener diferentes funciones, variables u objetos. Por ejemplo, definamos un módulo de nombre `prog_datasci.py` que contenga:

In [23]:

```
# prog_datasci.py

PI = 3.14159265

def suma(x, y):
    return x + y

def resta(x, y):
    return x - y
```

Para utilizar desde otro módulo o script estas funciones, deberíamos escribir lo siguiente:

In [24]:

```
from prog_datasci import PI, suma, resta

# Y entonces podríamos utilizarlas normalmente.
siete = suma(2,5)
```

En Python también podemos utilizar la directiva `from prog_datasci import *`, pero su uso está **totalmente desaconsejado**. La razón es que estaríamos importando gran cantidad de código que no utilizaremos (con el consiguiente aumento del uso de la memoria), pero además, podríamos tener colisiones de nombres (funciones que se llamen de la misma forma en diferentes módulos) sin nuestro conocimiento. A no ser que sea imprescindible, no utilizaremos esa directiva e importaremos una a una las librerías y las funciones que vayamos a necesitar.

Podéis aprender más sobre importar librerías y definir vuestros propios módulos [aquí](#).