prog_datasci_3_apython

September 24, 2022

1 Fundamentos de programación

1.1 Unidad 3: Estructuras de control y funciones en Python

---

2 Instrucciones de uso

Este documento es un *notebook* interactivo que intercala explicaciones más bien teóricas de conceptos de programación con fragmentos de código ejecutables. Para aprovechar las ventajas que aporta este formato, os recomendamos que, en primer lugar, leáis las explicaciones y el código que os proporcionamos. De este modo, tendréis un primer contacto con los conceptos que exponemos. Ahora bien, **la lectura es solo el principio!** Una vez hayáis leído el contenido, no os olvidéis de ejecutar el código proporcionado y modificarlo para crear variantes que os permitan comprobar que habéis entendido la funcionalidad y explorar los detalles de implementación. En último lugar, os recomendamos también consultar la documentación enlazada para explorar con más profundidad las funcionalidades de los módulos presentados.

Para guardar posibles modificaciones que hagáis sobre este notebook, os aconsejamos que montéis la unidad de Drive en Google Colaboratory (colab). Tenéis que ejecutar las instrucciones siguientes:

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

[ ]:
```
%cd /content/drive/MyDrive/Colab_Notebooks/prog_datasci_3
```

3 Introducción

Esta unidad sigue presentando conceptos básicos de programación en Python. En concreto, presentaremos las operaciones lógicas y veremos cómo se puede alterar el flujo de ejecución de los programas con estructuras iterativas (*for* y *while*), condicionales (*if*) y funciones. Adicionalmente,

explicaremos cómo podemos interactuar con ficheros desde Python, y cómo podemos importar otros módulos para incorporar funcionalidades adicionales en nuestros programas.

A continuación, se incluye la tabla de contenidos. Para navegar por el documento usad `Table of contents` (parte superior izquierda) de Google colab.

1 ¿Qué es una estructura de control?

Los programas en Python ejecutan las instrucciones secuencialmente. Cada instrucción se ejecuta de manera ordenada y en el mismo orden que están escritas, como hemos visto en la unidad 2. En el ejemplo siguiente se puede apreciar cómo se ejecutan las instrucciones secuencialmente.

[1]:
```
x = 3
y = x + 7
print("El valor de x es {}".format(x))
print("El valor de y es {}".format(y))
```

```
El valor de x es 3
El valor de y es 10
```

Primero asignamos la variable x igual a 3 y la variable y como la suma de x más el valor numérico 7. A continuación, mostramos por pantalla los valores de x y y. Cada instrucción se ejecuta una detrás de la otra. El flujo de ejecución de un conjunto de instrucciones puede ser modificado mediante las estructuras de control. Las principales estructuras de control son las estructuras de control condicionales (**if-elif-else**) y las estructuras de control iterativas (**for y while**).

2 Condicional

La estructura de control condicional es aquella estructura de control que permite implementar acciones según la evaluación de una condición simple, ya sea falsa o verdadera.

2.1 If

La instrucción if nos permite ejecutar un bloque de código si se cumple una determinada condición.

[2]:
```
a = 10
if a >= 5:
    print('a es mayor o igual a 5')
```

```
a es mayor o igual a 5
```

El objetivo del código anterior es evaluar si la variable a es mayor o igual a 5. Si la variable a es mayor o igual a 5, se cumple la condición de `if`, de forma que se imprime por pantalla la instrucción **a es mayor o igual a 5**. En cambio, en la celda siguiente a es menor que 5. Por lo tanto, la condición de `if` no se cumple, y el bloque de código dentro de `if` no se ejecuta.

[2]:
```
a = 2
if a >= 5:
    print('a es mayor o igual a 5')
```

En los dos ejemplos anteriores podemos apreciar la estructura inherente de `if`. Esta estructura tiene dos partes muy diferenciadas:

- **Condición** que se tiene que cumplir para que el bloque de código se ejecute, en el ejemplo anterior `a >= 5`.
- **Bloque de código** que se tiene que ejecutar si se cumple la condición anterior. Es importante remarcar que el bloque de código siempre debe contener una instrucción, si no es así se genera un error de sintaxis `SyntarError`.

La instrucción `if` debe terminarse con `:` y el bloque de código debe estar tabulado. Toda sentencia insertada después de `if` y correctamente tabulada, forma parte del bloque de código que se ejecutará si la condición se cumple. En el ejemplo siguiente, la instrucción `print('Fuera if')` se está ejecutando siempre porque no forma parte del bloque de código de `if`.

[3]:
```
a = 2
if a >= 5:
    print('a es mayor o igual a 5')
print('Fuera if')
```

```
Fuera if
```

2.2 If … else

Si la condición no se cumple, se puede ejecutar un segundo bloque de código especificado dentro de la instrucción `else`.

[4]:
```
a = 9
if a % 2 == 0:
    print('a es par')
else:
    print('a no es par')
```

```
a no es par
```

La cláusula `else` especifica qué hay que hacer si la condición no se cumple. En este ejemplo, la primera condición no se cumple porque a no es par, por lo tanto, se ejecuta la instrucción que hay dentro de `else`.

2.3 If…elif…else

Si queremos ejecutar condiciones adicionales podemos usar la instrucción `elif`.

[5]:
```
a = 9
if a % 2 == 0:
    print('a es par')
elif a % 3 == 0:
    print('a es multiple de 3')
```

```
a es multiple de 3
```

En el ejemplo anterior, si se cumple la primera condición, se imprime por pantalla **a es par**. Si se cumple la segunda condición, se imprime por pantalla **a es múltiple de 3**. También podemos incluir cláusulas `else` dentro de la estructura`if-elif` si no se cumple ninguna de las condiciones anteriores.

[6]:
```
a = 7
if a % 2 == 0:
    print('a es par')
elif a % 3 == 0:
    print('a es múltiple de 3')
else:
    print('a no es par ni múltiple 3')
```

```
a no es par ni múltiple 3
```

Veamos unos ejemplos más para ver diferentes estructuras condicionales. Os aconsejamos que variéis los valores de las variables de código de los ejemplos siguientes e id ejecutando el código para comprobar cómo se comporta en cada situación.

[7]:
```
# Ejemplo 1
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
```

[8]:
```
# Ejemplo 2
a = -1
b = -2
if a > b:
    if b >= 0:
        print('a es mayor que b y positivo')
    elif a*-1 > 0:
        print('a es mayor que b y negativo')
    else:
        print('a es mayor que b y positivo')
else:
    print('b es mayor o igual que a')
```

```
a es mayor que b y negativo
```

3 Iteración

Las estructuras de control iterativas permiten ejecutar un mismo bloque de código tantas veces como sea necesario. En la mayoría de los lenguajes de programación, hay dos formas de iterar una secuencia: mediante `for` o mediante `while`.

3.1 For

La instrucción for nos permite crear bucles sobre un número de iteraciones definido inicialmente.

[10]:
```
monsters = ['Kraken', 'Leviathan', 'Uroborus', 'Hydra']

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

```
Kraken
Leviathan
Uroborus
```

En el ejemplo anterior, estamos recorriendo cada elemento de la lista `monsters` e imprimiéndolo por pantalla. La estructura de iteración `for` sigue la estructura siguiente:

[28]:
```
# for <variable> in <iterable>:
#<codi>
```

donde la `variable` toma cada uno de los valores del iterable. `iterable` son aquellos objetos que pueden ser iterados o que pueden ser indexados. Algunos ejemplos de iterables son las `listas`, `tuples`, `cadenas` o `diccionarios`. Es importante remarcar que toda sentencia insertada después del `for` tiene que estar correctamente tabulada para que se ejecute. Como podéis ver en la celda siguiente, el `print` no está tabulado y crea un error `IdentationError`.

[9]:
```
for monster in monsters:
print(monster)
```

```
  File "<ipython-input-9-baf8f65b9206>", line 2
    print(monster)
    ^
IndentationError: expected an indented block
```

En el ejemplo siguiente, veremos diferentes tipos de iterables.

[10]:
```
# Definimos una lista, una tupla, un diccionario y una cadena de caracteres
planetas_lista = ['Mercurio', 'Venus', 'Tierra']
planetas_tupla = ('Mercurio', 'Venus', 'Tierra')
planetas_dict = {"Mercurio": 4880, "Venus": 12105, "Tierra": 12745}
planetas_str = 'Mercurio'

# Recorremos las estructuras con un *for* y mostramos su contenido
print("**Lista**")
for i in planetas_lista:
    print(i, end=" ")
print()

print("**Tupla**")
for i in planetas_tupla:
    print(i, end=" ")
print()

print("**dict**")
for i in planetas_dict:
    print(i, end=" ")
print()
```

```
print("**cadena**")
for i in planetas_str:
    print(i, end=" ")
```

```
**Lista**
Mercurio Venus Tierra
**Tupla**
Mercurio Venus Tierra
**dict**
Mercurio Venus Tierra
**cadena**
M e r c u r i o
```

Es importante remarcar que un valor numérico no es un objeto iterable porque no puede ser indexado. Si ejecutamos el ejemplo siguiente se obtiene el error `TypeError: 'int' object is not iterable`.

[11]:
```
a = 5

for i in a:
    print(a)
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-11-7938d9b1b2cb> in <module>
  1 a = 5
  2
----> 3 for i in a:
  4     print(a)


TypeError: 'int' object is not iterable
```

Si queremos iterar una secuencia de valores numéricos, tendremos que crear una lista de valores e iterar sobre esta lista.

[12]:
```
# Primero creamos la lista de valores numéricos del 0 al 5 (inclusivos)
lista_valores = [0, 1, 2, 3, 4, 5]

# Recorremos la lista con un *for* y mostramos su contenido
for i in lista_valores:
    print(i, end=" ")
```

```
0 1 2 3 4 5
```

En la sección 3.5 veremos cómo podemos crear una secuencia de valores numéricos mediante la función `range()` sin tener que definir ninguna lista antes.

La instrucción `for` permite incluir otras estructuras de control dentro de su bloque de código, por ejemplo `for`. Esto permite iterar un objeto que en cada elemento tiene un objeto iterable.

[13]:
```
planetas_lista = ['Mercurio', 'Venus', 'Tierra']

# Recorremos la lista con un *for* y mostramos su contenido

for i in planetas_lista:
    print(i, end="")
    # Recorremos cada elemento de la cadena y contamos el número de letras que␣
 ↪forman la palabra
    print()
    k = 0
    for j in i:
        # Se actualiza el valor de k sumándole 1
        k += 1
    print("La palabra {} está formada por {} letras".format((i), (k)))
```

```
Mercurio
La palabra Mercurio está formada por 8 letras
Venus
La palabra Venus está formada por 5 letras
Tierra
La palabra Tierra está formada por 6 letras
```

3.2 While

La instrucción `while` permite iterar un objeto mientras se cumpla una condición determinada. Cuando la condición deja de cumplirse, se sale del bucle y continúa la ejecución del bloque de código siguiente. En el ejemplo que sigue, el código dentro de `while` se ejecutará mientras `a` sea mayor que `b`.

[16]:
```
a = 5
b = 0

while a > b:
    # Se actualiza el valor de b sumándole 1
    b += 1
print("b es igual {}".format(b))
```

```
b es igual 5
```

También se puede iterar una lista mediante `while`, pero es una manera mucho menos idiomática en Python y es preferible la opción de `for`.

[14]:
```
# Mientras el índice 'i' sea menor que la longitud de la lista 'planetas':
i = 0
```

```
while i < len(planetas_lista):
    # Imprime el valor de la lista en la posición 'i'.
    print(i, planetas_lista[i])
    # No nos olvidamos de actualizar el valor de 'i' sumándole 1 o tendremos un␣
 ↪bucle infinito.
    i += 1
```

```
0 Mercurio
1 Venus
2 Tierra
```

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

[15]:
```
# 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
```

3.3 Break y Continue

Las instrucciones `break` y `continue` nos permiten alterar el comportamiento de las instrucciones `for` y `while`.

- **break**: permite salir de un bucle en un momento dado y parar la ejecución.
- **continue**: permite saltarse el código restante en la iteración actual volviendo al principio del bucle.

[16]:
```
for planeta in planetas_lista:
    if planeta == "Venus":
        break
    print(planeta)
```

```
Mercurio
```

[17]:
```
for planeta in planetas_lista:
    if planeta == "Venus":
        continue
    print(planeta)
```

```
Mercurio
Tierra
```

En los ejemplos de las celdas anteriores, se puede apreciar la diferencia en el funcionamiento de las instrucciones `break` y `continue`. En el primer ejemplo, `break` interrumpe la ejecución del bucle cuando *planeta* es igual a *Venus*. En cambio, `continue` no interrumpe el bucle, sinó que pasa a la iteración siguiente saltando la instrucción pendiente `print`.

En el ejemplo siguiente, cambiad la instrucción `break` por `continue`. Ejecutad el código en las dos tesituras para comprobar cómo varía el output final según la instrucción que se haya usado.

[18]:
```
a = 20
#
while a > 0:
    a -= 1
    if a % 2 == 0:
        continue  # break
    print(a, end=" ")
```

```
19 17 15 13 11 9 7 5 3 1
```

3.4 Herramientas de Python para iterar eficientemente

Antes hemos definido dos estructuras de control, `for` y `while`, generales de la mayoría de los lenguajes de programación. Python tiene herramientas específicas para optimizar el proceso de iteración. Entre estas herramientas están:`range()`, `enumerate()` y `zip()`.

3.4.1 Range

`Range` es una función muy útil de Python para generar una secuencia de números. La sintaxis de `range` es la siguiente:

[27]:
```
# range(start,end,step)
```

`Range`es una función que permite pasar tres parámetros separados por coma, en que **start** es el valor inicial de la secuencia, **end** es el valor final y **step** el salto entre números. Es muy importante remarcar que 'range' devuelve una lista de números hasta la posición final menos 1. Por defecto, el valor inicial es 0 y el salto es 1.

[19]:
```
# La función 'range' nos devuelve una lista de números:
list(range(10))
```

[19]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

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

[20]:
```
# 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`:

[21]:
```
# Podemos usarla para iterar:
for i in range(10):
```

```
    print(i, end=" ")
print()
```

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

[22]:
```
# Podemos definir el rango de acción.

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

print()

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

print()

# o especificando también el salto entre cada valor:
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), puesto 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.

[26]:
```
# También es posible iterar en un diccionario:
pais_codigos = {34: 'Spain', 376: 'Andorra', 41: 'Switzerland', 424: None}

# Por clave:
for pais_codigo in pais_codigos.keys():
    print(pais_codigo)
print()

# Por valor:
for pais in pais_codigos.values():
    print(pais)
print()

# Por ambos a la vez:
for pais_codigo, pais in pais_codigos.items():
    print(pais_codigo, pais)
```

```
34
376
41
424

Spain
Andorra
Switzerland
None

34 Spain
376 Andorra
41 Switzerland
424 None
```

3.4.2 Enumerate

Enumerate es una función de Python para acceder a los índices de una colección. Esta función devuelve una tupla en que el primer elemento es un índice que empieza por 0 y aumenta de 1 a 1, y el segundo elemento es el valor de la posición en la lista.

En el ejemplo siguiente, accedemos al índice y al valor de la lista mediante `for` y un contador:

[23]:
```python
# Recuperamos el ejemplo de los monstruos
monsters = ['Kraken', 'Leviathan', 'Uroborus', 'Hydra']

# Inicializamos el contador
i = 0

for monster in monsters:
    print(i, monster, end=" ")
    i += 1
```

```
0 Kraken 1 Leviathan 2 Uroborus 3 Hydra
```

Podemos obtener el mismo resultado usando la función `enumerate`:

[24]:
```python
for index, monster in enumerate(monsters):
    print(index, monster, end=" ")
```

```
0 Kraken 1 Leviathan 2 Uroborus 3 Hydra
```

3.4.3 Zip

`Zip` es una función que, combinada con un `for`, permite iterar tantas listas como sea necesario en paralelo.

[27]:
```python
planetas = ["Mercurio", "Venus", "Tierra"]
diametro = ["4880 km", "12105 km", "12750 km"]
distancia_sol = ["57910000 km", "108200000 km", "146600000 km"]
```

```
for pla, dist, ds in zip(planetas, diametro, distancia_sol):
    print("{} tiene un diámetro de {} y se encuentra a una distancia del Sol de␣
 ↪{}".format(
        (pla), (dist), (ds)))
```

```
Mercurio tiene un diámetro de 4880 km y se encuentra a una distancia del Sol de
57910000 km
Venus tiene un diámetro de 12105 km y se encuentra a una distancia del Sol de
108200000 km
Tierra tiene un diámetro de 12750 km y se encuentra a una distancia del Sol de
146600000 km
```

Si las listas tienen longitudes diferentes, la iteración se detiene cuando la lista más pequeña se acaba.

[28]:
```
planetas = ["Mercurio", "Venus", "Tierra"]
diametro = ["4880 km", "12105 km", "12750 km"]
distancia_sol = ["57910000 km", "108200000 km"]

for pla, dist, ds in zip(planetas, diámetro, distancia_sol):
    print("{} tiene un diámetro de {} y se encuentra a una distancia del Sol de␣
 ↪{}".format(
        (pla), (dist), (ds)))
```

```
Mercurio tiene un diámetro de 4880 km y se encuentra a una distancia del Sol de
57910000 km
Venus tiene un diámetro de 12105 km y se encuentra a una distancia del Sol de
108200000 km
```

Esta función está definida por cualquier objeto iterable. Esta característica hace que se pueda usar `zip` con diccionarios. Para acceder al valor y la clave de cada elemento se tiene que usar la función `items`.

[29]:
```
pais_codigo = {34: 'Spain', 376: 'Andorra', 41: 'Switzerland', 424: None}
pais_hab = {'Spain': 46000000, 'Andorra': 77000, 'Switzerland': 8000000}

for (a, b), (c, d) in zip(pais_codigo.items(), pais_hab.items()):
    print(a, b, d)
```

```
34 Spain 46000000
376 Andorra 77000
41 Switzerland 8000000
```

4 List Comprehensions

List comprehension es una expresión idiomática de Python que permite crear listas de elementos en una sola línea de código de forma sencilla y elegante sin usar estructuras de control muy complejas.

[30]:
```
# Queremos seleccionar todos aquellos planetas que tienen una *r* en su nombre
planetas = ["Mercurio", "Venus", "Tierra"]
subset = []
for x in planetas:
    if "r" in x:
        subset.append(x)
print(subset)
```

```
['Mercurio', 'Tierra']
```

En el ejemplo anterior, hemos implementado una estructura de control clásica con `for` e `if` para seleccionar los elementos de la lista que tienen una *r* en su nombre. En la celda siguiente haremos lo mismo usando *list comprehension*.

[31]:
```
subset = [x for x in planetas if "r" in x]

print(subset)
```

```
['Mercurio', 'Tierra']
```

La sintaxis general de una *list comprehension* es la siguiente:

[115]:
```
# lista = [expresión for element in iterable]
```

Esta sintaxis tiene dos partes muy diferenciadas.

- `for element in iterable` iteración de un determinado iterable y se guarda cada uno de los elementos en `elemento`. El iterable puede ser una lista, secuencia, generador o una estructura de control (como por ejemplo, `if`) que devuelva sus elementos uno por uno.
- `expresión` acción sobre `elemento` que se añadirá a la lista a cada iteración.

[35]:
```
# Aplicaremos una list comprehension para calcular el cuadrado de los números␣
↪pares presentes hasta el 9
resultado = [i**2 for i in range(10) if i % 2 == 0]
print(resultado)
```

```
[0, 4, 16, 36, 64]
```

5 Funciones

Otra manera muy importante de organizar el flujo de ejecución es encapsulando una cierta porción de código en una función reutilizable. Se tiene que enfatizar que el conjunto de instrucciones que se ejecutan dentro de la función deben estar correctamente tabuladas respecto a `def()`. Si no es así, se obtiene un error de tabulación.

Una función en Python utiliza un concepto similar al de una función matemática. Por ejemplo, imaginémonos la función matemática:

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

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

[36]:
```
# La función 'suma' se define mediante la palabra especial 'def' y tiene dos␣
↪argumentos: 'x' e 'y':
def suma(x, y):
    # Mostramos por pantalla los valores de x e y
    print("El valor de x es {}".format(x))
    print("El valor de y es {}".format(y))
    # Devolvemos el valor de la suma
    return x + y
```

La definición de una función Python tiene las características siguientes:

- Palabra clave def().
- Nombre de la función.
- Paréntesis y dentro de los paréntesis los parámetros de entrada (a pesar de que pueden ser opcionales).
- Dos puntos :.
- Bloc de código.
- Sentencia de retorno, `return`, del resultado (opcional).

En la definición de una función no se ejecuta el bloque de código que contiene la función. Para ejecutarlo, se tiene que escribir el nombre de la función con los parámetros de entrada correspondientes, como podemos ver en los ejemplos siguientes. En el primer ejemplo, estamos ejecutando la función `suma()` usando el nombre de los parámetros.

[37]:
```
# Ejecución de la función suma
suma(x=5, y=-5)
```

```
El valor de x es 5
El valor de y es -5
```

[37]:
```
0
```

Si usamos el nombre de los parámetros, podemos cambiar la orden de los parámetros sin modificar el resultado de la función.

[37]:
```
# Ejecución de la función suma cambiando el orden de los parámetros
suma(y=-5, x=5)
```

```
El valor de x es 5
El valor de y es -5
```

[37]:
```
0
```

Si no usamos el nombre de los parámetros, cada posición corresponde a un parámetro tal como hemos definido dentro de la función. En nuestro caso, la primera posición es `x`, y la segunda, `y`.

[38]:
```
# Ejecución de la función con los argumentos posicionales
suma(4, 2)
```

```
El valor de x es 4
El valor de y es 2
```

[38]: 6

[39]:
```
# Ejecución de la función cambiando el orden de los argumentos posicionales
suma(2, 4)
```

```
El valor de x es 2
El valor de y es 4
```

[39]: 6

[40]:
```
# Podemos definir una función que no haga nada usando la palabra especial 'pass':

def dummy():
    pass
```

[41]:
```
dummy()
```

[38]:
```
# 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
```

[39]:
```
fibonacci(10)
```

```
0 1 1 2 3 5 8
```

[40]:
```
# En el ejemplo anterior, hemos definido que el argumento n tenga un valor por␣
 ↪defecto. Esto es muy útil
# en los casos en que usamos la función siempre con un mismo valor, y queramos␣
 ↪dejar constancia de un
# caso de ejemplo o por defecto. En este caso, podemos ejecutar la función sin␣
 ↪pasarle ningún valor:
fibonacci()
```

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

[41]:
```
# Podemos definir parte de los argumentos con valores por defecto y otra parte␣
 ↪sin.
# Los argumentos sin valor por defecto siempre tienen que estar más a la␣
 ↪izquierda a la definición de la función:
```

```
def potencia(a, b=2):
    # Por defecto, elevaremos al cuadrado.
    return a**b
```

[42]:
```
print(potencia(3))
print(potencia(2, 3))
```

```
9
8
```

5.1 Args y kargs

Tal como hemos visto en el apartado anterior, hemos definido las funciones considerando un número concreto de argumentos *(0,1,2,…,N)*. Pero nos podemos encontrar que, a priori, no sepamos cuántas variables necesitamos. Para solucionar esto, en Python se puede definir un conjunto de argumentos variable con `argsy kargs`.

Args nos permite definir funciones genéricas que acepten un número variable de argumentos.

[47]:
```
def suma_args(*args):
    # Devolvemos el valor de la suma
    return sum(args)
```

[49]:
```
suma_args(3, 4, 5)
```

[49]:
```
12
```

La función `suma_args` corresponde a la misma función `suma`, que hemos definido en el apartado anterior, pero usando un número variable de argumentos. Si comparamos las dos funciones, veremos que la función `suma` solo permite sumar dos números. En cambio, `suma_args` permite sumar tantos números como quiera el usuario.

[50]:
```
print("Suma con un número variable de argumentos es {}".formato(suma_args(4, 5)))
print("Suma de dos argumentos es {}".formato(suma(4, 5)))
```

```
Suma con un número variable de argumentos es 9
El valor de x es 4
El valor de y es 5
Suma de dos argumentos es 9
```

[51]:
```
print("Suma con un número variable de argumentos es {}".format(
    suma_args(4, 5, 7, 8, 9)))
#print("Suma de dos argumentos es {}".format(suma(4,5,7,8,9)))
```

```
Suma con un número variable de argumentos es 33
```

Si se quiere sumar *5* números mediante la función `suma` definida con dos variables, la ejecución da error porque estás usando más argumentos de los definidos.

Si, además de trabajar con un número variable de argumentos, queremos tener la capacidad de definir el nombre de cada uno de los argumentos de entrada, tenemos que usar **kargs**. Debemos trabajar **kargs** como si se tratara de un diccionario y acceder a los correspondientes valores y claves mediante la función `items`.

[44]:
```
def suma_kargs(**kargs):
    r = 0
    # Devolvemos el valor de la suma
    for k, v in kargs.items():
        print(k, "=", v)
        r += v
    return r
```

[45]:
```
print("Suma con un número variable de argumentos es {}".format(
    suma_kargs(a=4, b=5, c=7, d=8, e=9)))
```

```
a = 4
b = 5
c = 7
d = 8
e = 9
Suma con un número variable de argumentos es 33
```

5.2 Función Lambda

La función *Lambda* o anónima es un tipo de función de Python que se emplea para simplificar la escritura de funciones en una sola línea. En el ejemplo siguiente queremos obtener todos los números múltiples de 5.

[54]:
```
# Definimos la función myfun(), en que el parámetro de entrada es el valor hasta␣
↪donde queremos estudiar

def myfun(xmax):
    # Definimos la lista
    mylist = []
    for k in range(0, xmax):
        # Comprobem si el valor es multiple de 5
        if k % 5 == 0:
            # Añadimos el valor a la lista con append()
            mylist.append(k)
    # Mostramos por pantalla
    print(mylist)
```

[55]:
```
myfun(101)
```

```
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95,
100]
```

Podemos expresar la función `myfun()` usando la función *Lambda* como:

[56]:
```
# La función filter() permite extraer los valores que cumplen una determina␣
 ↪condición
print(list(filter(lambda x: x % 5 == 0,  range(0, 101))))
```

```
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95,
100]
```

Como se puede ver en el ejemplo anterior, la función `lambda` (en combinación con la función `filter`) nos permite obtener el mismo resultado que la función `myfun` de manera más simplificada y compacta.

6 Leer y escribir desde ficheros

Una tarea habitual es leer líneas de un fichero o escribir líneas en un fichero. A continuación, os explicamos cómo podemos escribir y leer un fichero:

[55]:
```
# Antes que nada, escribiremos en un fichero:

# Abrimos un fichero de nombre 'a_file.txt' para escritura (de aquí la 'w',␣
 ↪'writing').
# Lo asignamos a un objeto para gestionar el fichero de nombre 'out':
out = open('a_file.txt', 'w')
# Escribiremos 10 líneas, cada una con un número del 0 al 9.
for i in range(10):
    # La línea siguiente escribe en el fichero todo lo que ponemos dentro de␣
 ↪'out.write()'
    # En nuestro caso, es un string del tipo '0\ , '1\ , etc. Esto lo conseguimos
    # usando formato con un parámetro, que es el número de línea.
    # Añadimos el \n al final de cada línea, para incluir el salto de línea
    out.write("Línea {}\n".format(i))

    # Alternativamente, podríamos usar los wildcards %d y %s.
    # %s representa un string o una 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 a Linux
    # a '\n':
    # out.write("Línea %d%s" % (i, os.linesep))

# Una vez hemos terminado de escribir el fichero, se tiene que cerrar con la␣
 ↪instrucción close()
out.close()
```

Ahora leeremos el fichero que acabamos de escribir de tres maneras diferentes. En los dos primeros métodos, una vez hemos acabado de trabajar con el archivo, se tiene que cerrar con la instrucción `close()`. En cambio, el tercer método,`with()`, cierra el archivo automáticamente.

[56]:
```
# 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
```

6.1 Pandas

Otra manera de leer y escribir ficheros en Python es mediante la librería *Pandas*. Para poder hacer uso de las funciones de una librería, antes que nada se tiene que importar la librería mediante la instrucción *import*.

[58]:
```
import pandas as pd
```

Pandas nos permite cargar los datos de un fichero *CSV* directamente a un dataframe por medio de la función `read_csv`. Esta función es muy versátil y dispone de muchos parámetros para configurar con todo detalle cómo llevar a cabo la importación. En muchos casos, la configuración por defecto ya nos ofrecerá los resultados deseados.

Ahora cargaremos los datos del fichero `marvel-wiki-data.csv`, que contiene datos sobre personajes de cómic de Marvel. El conjunto de datos fue creado por la web FiveThirtyEight, que escribe artículos basados en datos sobre deportes y noticias, y que pone a disposición pública los conjuntos de datos que recoge para sus artículos.

[59]:
```
data = pd.read_csv("data/marvel-wiki-data.csv")
print(type(data))
```

```
<class 'pandas.core.frame.DataFrame'>
```

De una manera análoga a como hemos cargado los datos de un fichero en un *dataframe*, podemos escribir los datos de un *dataframe* en un fichero CSV.

[60]:
```
# Seleccionamos las primeras 20 líneas del dataframe data
new_data = data.head(n=20)
# Guardamos el dataframe new_data en un CSV
new_data.to_csv("data/marvel-wiki-data-head20.csv", encoding='utf-8')
```

En la unidad 4 veremos más funcionalidades de la librería Pandas y presentaremos con más detalle los tipos de datos que devuelve `read_csv`.

7 Errores y Excepciones

La codificación está sujeta a errores. Conocer y corregir estos errores forma parte del aprendizaje de cualquier lenguaje de programación. En Python, hay dos tipos diferentes de errores: errores de sintaxis y excepciones. La principal diferencia entre los dos tipos de errores es que el primero está asociado a un error de sintaxis, y el segundo, a un error de ejecución.

- **Errores de sintaxis o de interpretación**: Son errores provocados por un error en la codificación. Cuando hay un error de este tipo, el intérprete reproduce la línea responsable del error y muestra con una flecha dónde ha habido el error.

[61]:
```
i = 0
while i == 0
    print("hola")
```

```
    i += 1
```

```
      File "<ipython-input-61-92300af304f3>", line 2
    while i == 0
                ^
SyntaxError: invalid syntax
```

El error nos está indicando que ha habido un error de sintaxis en la segunda línea en la instrucción `while`. En este caso, faltan dos puntos (`':'`) al final.

[62]:
```
i = 0
while i == 0:
    print("hola")
    i += 1
```

```
hola
```

- **Excepciones**: Los errores detectados durante la ejecución, a pesar de que la sintaxis sea correcta, se denominan **excepciones**. La mayoría de las excepciones dan un mensaje de error. El ejemplo siguiente muestra un error de ejecución:

[63]:
```
a = 10
b = 0
print(a/b)
```

```
    ␣
↪---------------------------------------------------------------------------

    ZeroDivisionError                         Traceback (most recent call last)

    <ipython-input-63-0ca4df5e1f05> in <module>
      1 a = 10
      2 b = 0
----> 3 print(a/b)


    ZeroDivisionError: division by zero
```

La última línea del mensaje de error informa de aquello que pasa. Hay diferentes tipos de excepciones, las principales son **ZeroDivisionError**, **NameError**, **TypeError**. En las celdas siguientes veremos un ejemplo de las excepciones **NameError** y **TypeError**.

[64]:
```
# Excepción NameError
x = 3 + parametre
```

```
↪---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-64-392d838d537f> in <module>
      1 # Excepción NameError
----> 2 x = 3 + parametre


NameError: name 'parametre' is not defined
```

La excepción **NameError** se produce porque estamos ejecutando una orden, `suma`, con una variable que no está definida. En cambio, el error siguiente, **TypeError**, se origina porque estamos intentando sumar dos tipos de magnitudes que no se pueden sumar, número entero con una cadena de caracteres.

[65]:
```
# Excepción TypeError
x = 3 + '2'
```

```
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-65-f6730bb87b1a> in <module>
      1 # Excepción TypeError
----> 2 x = 3 + '2'


TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

7.1 Handling Exceptions

La gestión de las excepciones es muy importante para el buen funcionamiento del programa. Si no es así, la aparición de una excepción puede provocar que el programa se pare. Las excepciones pueden ser capturadas y gestionadas adecuadamente sin que el programa se pare haciendo uso de las instrucciones `try` y `except`. Si recuperamos el ejemplo anterior:

[46]:
```
a = 10
b = 0
while b < 5:
    try:
        c = a/b
        print(c)
```

```
        break
    except ZeroDivisionError:
        b += 1
        print("No se puede hacer la división")
```

```
No se puede hacer la división
10.0
```

En este ejemplo, el programa no se para y puede continuar ejecutándose. Cuando hay una *excepción* se ejecuta el código que hay dentro de la instrucción `except`, pero el programa no se para. La instrucción `except` también permite usar excepciones genéricas mediante `Exception` cuando se desconoce la excepción que se crea.

[66]:
```
a = 10
b = 0
while b < 5:
    try:
        c = a/b
        print(c)
        break
    except Exception:
        b += 1
        print("No se puede hacer la división")
```

```
No se puede hacer la división
10.0
```

Otras instrucciones para añadir a los bloques de código `try` y `except`son `else` y `finally`.

- `else`: Se ejecuta si no ha habido ninguna excepción.
- `finally`: Se ejecuta siempre aunque haya habido una excepción.

[67]:
```
a = 10
b = 0
while b < 5:
    try:
        c = a/b
    except ZeroDivisionError:
        b += 1
        print("No se puede hacer la división")
    else:
        print("Se ha ejecutado la división y el resultado ha sido {}".format(c))
        break
```

```
No se puede hacer la división
Se ha ejecutado la división y el resultado ha sido 10.0
```

[68]:
```
a = 10
b = 1
```

```
while b < 2:
    try:
        c = a/b
    except ZeroDivisionError:
        b += 1
        c = "No se puede hacer la división"
    finally:
        print(c)
        break
```

```
10.0
```

7.2 Raising Exceptions

Finalmente, también se pueden lanzar excepciones específicas con la instrucción `raise`. Esta instrucción permite definir qué tipo de error se genera y el texto que se tiene que imprimir al usuario.

[69]:
```
x = -1

if x < 0:
    raise Exception("Error, tienen que ser valores positivos")
```

```
␣
↪---------------------------------------------------------------------------

Exception                                 Traceback (most recent call last)

<ipython-input-69-72c3bb899d4a> in <module>
  2
  3 if x  4     raise Exception("Error, tienen que ser valores positivos")


Exception: Error, tienen que ser valores positivos
```

Recomendamos la lectura de la documentación oficial para terminar de fijar conocimientos.

8 Ejercicios y preguntas teóricas

La parte evaluable de esta unidad consiste en la entrega de un fichero IPython Notebook con extensión IPYNB que contendrá los diferentes ejercicios y las preguntas teóricas que se tienen que contestar. Encontraréis el fichero (`prog_datasci_3_python_entrega.ipynb`) con las actividades en la misma carpeta que este notebook que estáis leyendo.

8.1 Instrucciones importantes

Es muy importante que a la hora de entregar el fichero Notebook con vuestras actividades os aseguréis de que:

1. Vuestras soluciones sean originales. Esperamos no detectar copia directa entre estudiantes.

2. Todo el código esté correctamente documentado. El código sin documentar equivaldrá a un 0.
3. El fichero comprimido que entregáis es correcto (contiene las actividades de la PEC que tenéis que entregar).

Para hacer la entrega, tenéis que ir a la carpeta del drive `Colab Notebooks`, clicando con el botón derecho en la PEC en cuestión y haciendo `Download`. De este modo, os bajaréis la carpeta de la PEC comprimida en `zip`. Este es el archivo que tenéis que subir al campus virtual de la asignatura.

9 Bibliografía

Os recomendamos que consultéis los ejemplos siguientes del libro *Learn Python 3 the hard way* de la bibliografía de la asignatura, y que intentéis hacer las actividades que se proponen para acabar de entender y practicar los conceptos explicados en esta unidad.

Operaciones lógicas: * Exercise 27. Memorizing Logic * Exercise 28. Boolean Practice

Condicionales: * Exercise 29. What If * Exercise 30. Else and If * Exercise 31. Making Decisions

Iteración: * Exercise 32. Loops and Lists * Exercise 33. While-Loops

4 Autores

- Autor original Brian Jiménez Garcia, 2016.
- Actualizado por Cristina Pérez Solà, 2017 y 2019.
- Actualizado por Joan Maynou Fernández, 2022.

```
<div style="width:0%;"> </div>
    <img src="https://mirrors.creativecommons.org/presskit/buttons/88x31/png/by-sa.png">
```