Full Stack Checkpoint 5

Condicionales en Python

Un condicional es una sentencia de programación que permite controlar el flujo del código de programación en función de si se cumple o no una condición. La sentencia básica de ejecución condicional en Python se denomina *if*. Esta permite ejecutar un bloque de código si se cumple una condición dada, lo cual significa que la expresión es evaluada como verdadera. En el siguiente ejemplo, el código de programación comprueba si el valor de una variable *x* es menor de 20. Si la condición se cumple significara que la expresión "x < 20" será verdadera y se ejecutara el bloque de código que sigue a los ":", en este caso la función *print*,

In [1]:
```
x = 10

if x < 20:
    print("La variable x es menor de 20.")
```

```
La variable x es menor de 20.
```

La sentencia *if* se usa habitualmente con las sentencias *elif* y *else*, formando lo que se denomina una sentencia compuesta. *elif* comprueba una condición alternativa a la señalada por el *if* previo que es evaluada en caso de que no se cumpla la condición previa. La sentencia *else*, por el contrario, representa el caso por defecto que se ejecutará si ninguna de las condiciones anteriores ha resultado ser verdadera. Desarollando el ejemplo anterior, pero con el añadido de una sentencia *elif* podriamos escribir:

In [ ]:
```
if x < 5:
    print("La variable x es menor de 5")
elif x < 10:
    print("La variable x es menor de 10")
```

En este caso, ninguno de los segmentos de programación encapsulados después de las dos condiciones va a ejecutarse puesto que las dos expresiones son falsas. No obstante, si añadimos un caso por defecto, este si sería ejecutado tal y como muestra el ejemplo siguiente:

In [3]:
```
if x < 5:
    print("La variable x es menor de 5")
elif x < 10:
    print("La variable x es menor de 10")
else:
    print("La variable x es mayor de 9")
```

```
La variable x es mayor de 9
```

Nótese que, en Python, las sentencias *if*, *elif* y *else* tienen el mismo nivel de indentación mientras que los bloques de programación que encierran se encuentran indentados un nivel.

¿Cuáles son los diferentes tipos de bucles en Python? ¿Por qué son útiles?

Un bucle es una estructura de programación que te permite ejecutar un bloque de código de manera repetitiva. En Python existen dos tipos principales de bucles, los denominados bucles *for* y los bucles *while*. Estos dos bucles, aunque pueden usarse para realizar tareas equivalentes, normalmente se usan en casos distintos.

El bucle *for* se usa siempre con un objeto iterable, como podría ser una lista o un rango, donde se conoce el número de iteraciones que serán necesarias. Por ejemplo, el código siguiente imprimirá todos los números dentro del rango 0 a 10, los cuales van incrementando a cada iteración del bucle:

In [4]:
```
for x in range(0,11):
    print(x)
```

```
0
1
2
3
4
5
6
7
8
9
10
```

En este caso, el objeto iterable es un rango y el bloque de código que se ejecuta está formado únicamente la función *print*, la cual, como es habitual en Python, prosigue al símbolo ":" y se encuentra indentada un nivel.

El bucle *while* representa una iteración condicional donde no se conoce a priori el número de iteraciones necesarias para su finalización. En este caso, el bucle se repetirá mientras la condición siendo evaluada sea cierta. Por ejemplo, el bucle *while* siguiente se ejecutará inicialmente puesto que, al ser la variable *x* menor a 2, la condición será cierta inicialmente. Aun así, puesto que el bloque de código dentro del bucle usa números aleatorios para cambiar el valor de la variable implicada en la evaluación (función *randint* del paquete random), no se conoce el número de iteraciones que serán necesarias para abandonar el bucle.

In [2]:
```
from random import randint
x = 1
```

```
while x < 10:
  print(x)
  x = randint(0, 10)
```

```
1
7
```

A modo de resumen, los bucles *for* son útiles principalmente cuando se conoce el número de iteraciones necesarias para la realización de la tarea programada, como podría ser la evaluación de una función en un intervalo de valores. En cambio, los bucles *while* se usan prioritariamente cuando no podemos conocer el número de iteraciones que serán necesarias, como, por ejemplo, cuando necesitamos que el usuario introduzca un usuario y contraseña válidos, lo cual puede requerir un número de intentos que no podemos avanzar en el momento de la escritura del código.

¿Qué es una lista por comprensión en Python?

Una lista por comprensión es una estructura de datos que permite usar las funcionalidades de un bucle "for" pero en una sola línea de código con el propósito principal de crear una lista de manera dinámica. Esta estructura de datos se compone de los elementos principales que se muestran en el ejemplo:

In [ ]:
```
new_list = [ return_value for_in_code condition ]
```

donde *new_list* es la lista siendo creada, *return_value* representa el valor siendo retornado de manera iterativa y que formará las entradas de la lista, *for_in_code un bucle* for in* válido, y *condition* es una expresión condicional usada para filtrar que valores son añadidos a la lista. Este último es opcional y solo se añade en caso de ser necesario.

Desarrollando los elementos mencionados, podemos escribir el siguiente código a modo de ejemplo:

In [3]:
```
squared_even_numbers = [num**2 + 1 for num in range(11) if num % 2 == 0]

print(squared_even_numbers)
```

```
[1, 5, 17, 37, 65, 101]
```

El cual crea una lista con el cuadrado más uno de los números pares de 0 a 10.

¿Qué es un argumento en Python?

Un argumento en Python es un parámetro que se usa en un bloque de código, normalment una función. Los argumentos normalmente se pasan por posición, lo que significa que se asignan siguiendo el orden usado al usar función o método. En este caso, estos son obligatorios y debemos introducirlos forzosamente al llamar a la función. Si queremos evitar que los argumentos sean obligatorios debemos proporcionarles un valor por defecto u

omisión (estos también se denominan nombrados). Como ejemplo de los conceptos desarrollados podemos definir la función siguiente:

In [6]:
```
def my_foo(num_one, num_two, user = None):
    if user:
        print(f'Hi {user}, how are you doing?')

    return num_one + num_two


my_foo(1,2)
```

Out[6]: 3

In [7]:
```
my_foo(5,2, 'Juan')
```

```
Hi Juan, how are you doing?
```

Out[7]: 7

Tal y como muestra el ejemplo, al llamar la function my_foo, debemos introducir dos argumentos numéricos, pero podemos omitir el argumento *user*, el cual sí que introducimos en el segundo llamamiento a la función.
Existen técnicas para poder usar un numero variable de argumentos no nombrados y nombrados. Los argumentos nombrados se declaran en la definición de la función usando un asterisco seguidos de un nombre de argumento que, por convención, acostumbra a ser denominado *args* (del inglés, "arguments"). Veamos un ejemplo:

In [11]:
```
def my_foo2(num_one, num_two, *args):
    print('Hi ' + ' '.join(args) + ', how are you doing?')

    return num_one + num_two


my_foo2(2,5, 'Juan', 'Palomo')
```

```
Hi Juan Palomo, how are you doing?
```

Out[11]: 7

Como se puede apreciar en el ejemplo, *args* almacena dos argumentos *Juan* y *Palomo*. Al usar esta técnica, es necesario desempacar las distintas componentes de *args*. En este caso, usando el método *join*.

Otra posibilidad es el uso de un número variable de argumentos nombrados. La técnica es parecida a la descrita anteriormente, pero usando dos asteriscos seguidos de un nombre de argumento que, por convención, acostumbra a ser *kwargs* (del inglés, "keyword arguments"), como se muestra en el ejemplo:

In [16]:
```
def my_foo3(num_one, num_two, **kwargs):
    print(f'Hi  {kwargs["nombre"]} {kwargs["apellido"]}, How are you doing?')
```

```
    return num_one + num_two

my_foo3(2,5, nombre = 'Juan', apellido = 'Palomo')
```

```
Hi  Juan Palomo, How are you doing?
```

Out[16]: 7

Como se aprecia en el ejemplo, en este caso debemos introducir primero los argumentos posicionales, seguidos de tantos argumentos nombrados como queramos.

Los argumentos posicionales o optativos pueden utilizarse en conjunción con las dos técnicas descritas para pasar un número variable de argumentos no nombrados y nombrados. En este caso, los posicionales deben introducirse primero, seguidos de un número variable de argumentos no nombrados, seguidos a su vez de un numero variable de argumentos nombrados.

¿Qué es una función Lambda en Python?

Las funciones Lambda son funciones sencillas que se crean usando la palabra reservada *lambda*. A diferencia de las funciones declaradas con la palabra *def*, las funciones lambda no requieren asignar un nombre a la función que se crea, con lo cual también se las denomina funciones anónimas. La estructura general de estas funciones es la siguiente:

In [ ]:
```
lambda argumentos : expression
```

donde variables hace referencia a cualquier numero de argumentos y expresión a una única línea de código que define las operaciones que hace la función. Un ejemplo especifico podría ser el siguiente:

In [ ]:
```
foo_object = lambda num_one, n: num_one**n

foo_object(2,3)
```

el cual retorna el primer argumento elevado al segundo. Como puede verse en el ejemplo, la función se ha almacenado en el objeto *foo_object*.

La principal utilidad de las funciones lambda está en su simplicidad y su facilidad de declaración. Sus usos más habituales son para crear funciones simples que van a usarse pocas veces, por lo que no es necesario crear una función con la palabra reservada *def* que ocupara varias líneas de código. El segundo uso, es para pasar pequeñas funciones como argumente a funciones que aceptan funciones como argumento (denominadas funciones de orden superior). Un ejemplo de este uso seria la combinación de una función lambda con la función *map*:

In [ ]:
```
simple_list = range(0,11)
isinstance(simple_list, list)
new_list = list(map(lambda x: x + 10, simple_list))
```

```
print(new_list)
```

el cual usa la función *map* para añadir 10 a los valores contenidos en *simple_list*.

¿Qué es un paquete pip?

El instalador de paquetes de Python se llama *pip*. El nombre *pip* es un acrónimo de "Pip Installs Packages" o "Pip Installs Python", lo que denota claramente su función: instalar paquetes que otros programadores han creado. Estos paquetes *pip* se encuentran indexados, principalmente, en el listado de paquetes oficial de Python, denominado "Python Package Index". El instalador *pip* es probable que se encuentre ya instalado con Python pero, si es necesario, se puede instalar independientemente. Un método popular de instalación es usar el script *get-pip.py* descargable des de la url: https://bootstrap.pypa.io/get-pip.py. Una vez descargado, hay que ejecutar el fichero usando una terminal. Una vez instalado *pip* podemos usar la terminal para instalar los paquetes que deseemos. Para ello, hay que escribir *pip install "nombrepaquete"*, donde "*nombrepaquete*" hace referencia al nombre del paquete que queramos instalar, como, por ejemplo, el paquete de análisis de datos denominado *pandas*:

In [1]:
```
pip install pandas
```

```
Requirement already satisfied: pandas in c:\users\xavier\anaconda3\lib\site-packages
(2.1.4)
Requirement already satisfied: numpy<2,>=1.23.2 in c:\users\xavier\anaconda3\lib\sit
e-packages (from pandas) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in c:\users\xavier\anaconda3\l
ib\site-packages (from pandas) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in c:\users\xavier\anaconda3\lib\site-pa
ckages (from pandas) (2023.3.post1)
Requirement already satisfied: tzdata>=2022.1 in c:\users\xavier\anaconda3\lib\site-
packages (from pandas) (2023.3)
Requirement already satisfied: six>=1.5 in c:\users\xavier\anaconda3\lib\site-packag
es (from python-dateutil>=2.8.2->pandas) (1.16.0)
Note: you may need to restart the kernel to use updated packages.
```

Como se aprecia en el ejemplo, el paquete *pandas* ya se encuentra instalado, tal y como nos informa la terminal.