CheckPoint 5 – Preguntas teóricas

¿Qué es un condicional?

En programación, el concepto de condicional significa la posibilidad de decidir qué acción ejecutará nuestro programa, en función de la situación.

La sintaxis para implementar condicionales nos permitirá crear la lógica condicional y dotar a nuestro programa de dinamismo.

El caso más simple es la condición única en la que tenemos una variable con un valor determinado y necesitamos que si cumple, '**if**', con una condición entonces ejecute una acción.

```
# Single condition

age = 25
if age < 25:
  print(f"I'm sorry, you need to be at least 25 years old")
# No se imprime nada porque la condición no se cumple

age = 15
if age < 25:
  print(f"I'm sorry, you need to be at least 25 years old")
I'm sorry, you need to be at least 25 years old
# Se imprime el mensaje porque la condición se cumple
```

Es muy importante tener en cuenta que después de ':' la siguiente línea, que contendrá el bloque de código, debe tener sangría o Python dará error.

Si se añade una segunda condición '**else**' haremos que el funcionamiento sea más dinámico ya que si no se cumple la primera condición, entonces se realizará la acción indicada en la segunda condición. En el ejemplo anterior de condición única no ocurría nada si no se cumplía la condición. Ahora, en ambos casos tendremos respuesta. Si 'if' es True, se realiza la acción dentro del bloque, pero si es False entonces, 'else', se ejecuta el bloque siguiente.

```
# if/else
age = 55

if age < 25:
  print(f"I'm sorry, {age} is under 25 years old")
else:
  print(f"You're good to go, {age} fits in the range to rent a car")

You're good to go, 55 fits in the range to rent a car
```

Si lo que necesitamos es concatenar condiciones adicionales, entonces recurrimos a la estructura **if/elif/else**.

```
# if/elif/else
```

```
age = 110

if age < 25:
  print(f"I'm sorry, {age} is under 25 years old")
elif age > 100:
  print(f"I'm sorry, {age} is over 100 years old")
else:
  print(f"You're good to go, {age} fits in the range to rent a car")

I'm sorry, 110 is over 100 years old
```

Existe la posibilidad de utilizar el **operador ternario**. Este operador permite crear una declaración if/else en una sola línea de código.

La ventaja de usar el operador ternario es que la variable que almacena la salida del operador ternario es asignada una sola vez. La desventaja es que si no implementa correctamente se corre el riesgo de hacer nuestro código demasiado complejo de leer.

En este ejemplo, dependiendo de si el usuario es administrador o invitado, tendrá acceso o no tendrá acceso a la aplicación.

```
role = 'admin'
auth = 'can access' if role == 'admin' else 'cannot access'
print(auth)
can access

role = 'guest'
auth = 'can access' if role == 'admin' else 'cannot access'
print(auth)
cannot access
```

Como se puede ver, el resultado es equivalente en el caso de usar una estructura if/else tradicional:

```
role = 'admin'
if role == 'admin':
  auth = 'can access'
else:
  auth = 'cannot access'
print(auth)
can access
```

Los **operadores de comparación** se utilizan para comparar dos valores y devolver un resultado True o False.

Los operadores de comparación son:

== Operador de igualdad determina si dos valores son iguales.
!= Operador de desigualdad determina si dos valores no son iguales.
> Operador mayor que, determina si un valor es mayor que otro.
< Operador menor que, determina si un valor es menor que otro.
>= Operador mayor o igual que, determina si un valor es mayor o igual que otro.

<= Operador menor o igual que, determina si un valor es menor o igual que otro.

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

Los bucles o loops son herramientas que nos permiten repetir un bloque de código las veces que queramos. En Python tenemos dos tipos de bucles. El bucle 'for...in' y el bucle 'while'.

El bucle **'for..in'** se utiliza para iterar sobre una secuencia de elementos y ejecutar un bloque de código sobre cada elemento de la secuencia. La ventaja de 'for...in' es que se detiene cuando termina de ejecutar el bucle.

En el ejemplo, utilizamos el bucle 'for...in' para imprimir los números del 1 al 5. Para cada elemento 'i' dentro del rango del 1 al 5 no incluido, imprime 'i' por pantalla.

```
for i in range(1, 6):
    print(i)

1
2
3
4
5
```

Los tipos de datos que pueden ser iterados con 'for...in' son las colecciones de datos, listas, tuplas y diccionarios y los strings, o cualquier objeto que pueda ser reiterado, como un rango.

El bucle **'while'** realiza la misma función mientras una condición determinada sea verdadera y se sale del bucle cuando la condición es falsa. El problema que puede surgir es que si la condición no se vuelve falsa, entonces se debe indicar, mediante un valor centinela, cuándo detenerse o el bucle 'while' seguirá ejecutándose indefinidamente.

En el ejemplo, inicializamos la variable a igual a 1. Mientras sea verdadero que a es menor que cinco se ejecutará el bloque de código que imprime ¡Hola, mundo! La variable se actualiza sumándole 1 y se retoma el bucle hasta que la condición a menor que 5 sea falsa.

```
a = 1
while a < 5:
    print("¡Hola, mundo!")
    a = a + 1

¡Hola, mundo!
¡Hola, mundo!
¡Hola, mundo!
¡Hola, mundo!
```

Tenemos la posibilidad de parar o alterar el comportamiento del bucle mediante dos tipos de operadores lógicos de control de flujo, 'Continue' y 'Break'

El operador 'Continue' hace que el programa siga adelante hasta el final de la secuencia.

```
for i in range(5):
    if i == 3:
        continue
    print(i)
0
1
2
4
```

El operador 'Break' hace que el programa se detenga y no siga hasta el final de la secuencia. Esto es interesante para ahorrar recursos.

```
for i in range(5):
    if i == 3:
        break
    print(i)
0
1
2
```

¿Qué es una lista por comprensión en Python?

Una lista por comprensión sirve para crear listas a partir de listas, iterables o rangos. Se configura en una sola línea de código, la acción que queremos ejecutar, cuyo resultado se colocará en la lista de salida, el bucle 'for...in' con el elemento del iterable y el iterable, y, una condición opcional que se puede usar para filtrar los elementos. Todo envuelto por corchetes.

La sintaxis es [expresión **for** elemento **in** iterable **if** condición]

Hay que tener en cuenta que la sintaxis tradicional es mas legible

Por ejemplo, dada una lista nums, creamos una nueva lista squared con los cuadrados de los números de la lista nums solo si el cuadrado es un numero par.

```
nums = [2, 3, 4, 5, 6]

squared = [num ** 2 for num in nums if num % 2 == 0]

print(squared)

[4, 16, 36]
```

¿Qué es un argumento en Python?

Para comprender lo que es un argumento en Python hay que conocer la sintaxis de una función.

La sintaxis que define una función con parámetros es:

```
def nombre_función(parametro1, parametro2,…):
```

```
    # Bloque de código de la función que opera con los argumentos introducidos como
parámetro(s) de la función
```

Un argumento en Python es el valor que pasamos a un parámetro de una función.

Hay tres tipos de argumentos: los argumentos posicionales, los argumentos con nombre y los argumentos de longitud variable *args y **kwargs.

Los **argumentos posicionales** se asignan a un parámetro en función de su posición. Por ejemplo, cuando llamamos a la función sum con los argumentos 4 y 6, estos se asignan al parámetro 1 y 2 de la función respectivamente.

```
def sum(a, b):
    result = a + b
    return result

print(sum(4,6))
10
```

Hay que tener en cuenta que cuando se trabaja con funciones mas complejas, con más argumentos, los argumentos posicionales pueden generar confusión porque la posición determina la asignación y corremos el riesgo de cometer errores. En este caso conviene usar **argumentos con nombre**. Es decir que al llamar a la función asignamos directamente el valor al argumento independientemente de la posición que ocupe.

```
def sum(a, b, c, d, e):
    result = a + b + c + d + e
    return result

print(sum(a = 4, b = 6, c = 1, d = 9, e = 10))
30
```

Los **argumentos de longitud variable** son colecciones comprimidas. Para desempaquetar los elementos de las colecciones y pasarlos como argumentos, se usa la sintaxis * y luego, la convención habitual es nombrar a la lista de argumentos con la palabra args. Entonces, el **argumento *args** es la versión descomprimida o lista de elementos que van a ser pasados a la función como argumentos.

Por ejemplo, tenemos una función llamada descompresión que, al ser llamada con cualquier número de argumentos, imprime esas frutas en forma de **tuplas**.

```
def descompresion(*args):
  print(args)


descompresion('Orange', 'Melon', 'Apple')
descompresion('car', 'bike')

('Orange', 'Melon', 'Apple')
('car', 'bike')
```

El **argumento **kwargs** se utiliza para desempaquetar los elementos de un diccionario y pasarlos a la función como argumentos. Por ejemplo, si llamamos a la función llamada descompresión y le

pasamos argumentos con nombre, la salida será un **diccionario** donde la clave es fruit1 y el valor asociado el que hemos pasado, Orange. Y así sucesivamente con los demás elementos.

```
def descompresion(**kwargs):
  print(kwargs)

descompresion(fruit1 = 'Orange', fruit2 = 'Melon', fruit3 = 'Apple')
{'fruit1': 'Orange', 'fruit2': 'Melon', 'fruit3': 'Apple'}
```

¿Qué es una función Lambda en Python?

La función Lambda es una función anónima que nos permite empaquetar una expresión o función más sencilla, para introducirla en otras funciones o en otras partes del programa. La sintaxis de la función lambda es:

```
lambda arg1, arg2, arg3, …, argn: expresion
```

Podemos almacenar la función lambda en una variable para, por ejemplo, multiplicar un argumento por diez, o multiplicar dos argumentos:

```
multip_ten = lambda x: x * 10
print(multip_ten(5))
50
```

```
multip_ten = lambda x, y: x * y
print(multip_ten(5, 6))
30
```

Podemos almacenar la función lambda en una variable para luego usarla como argumento en otra función:

```
nombre_completo = lambda nombre, apellido: f'{nombre} {apellido}'

def saludo(alumno):
  print(f'Hi there {alumno}')


saludo(nombre_completo('Cecilia', 'Cesbron'))
Hi there Cecilia Cesbron
```

La ventaja de usar funciones lambda es que se pueden crear para usarlas una sola vez, por lo que se ahorran recursos.

¿Qué es un paquete pip?

Pip es un sistema que permite gestiona e instalar paquetes desde el Python Package Index (PyPI). En el PyPI hay muchas librerías o paquetes creados por desarrolladores que no podemos importar desde el Core Python Library porque no están incluido. Para ello es necesario establecer una conexión entre el core y el PyPI, y el conector a la base de datos PyPI es el pip. Los paquetes pip que contienen las librerías son instalados manualmente en nuestro sistema para luego poder importarlos y hacer uso de ellos.