1) ¿Qué es un condicional?

Los condicionales en Python son declaraciones que se utilizan para controlar el flujo del programa que estamos creando a partir de determinadas condiciones que establezcamos. Estos se implementan mediante las declaraciones 'if', 'elif' y 'else'.

- if: nos devuelve un valor True or False en función de si la condición es verdadera o falsa. En caso de ser verdadera el condicional se detiene en esa operación. En caso de ser falsa pasa al siguiente condicional.

- elif: nos sirve para establecer múltiples condicionales tras el primer 'if'. Al igual que el 'if' devuelve un valor True or False, siguiendo la misma lógica que el condicional anterior.

- else: establece lo que tiene que hacer el programa que estamos creando en caso de que los condicionales establecidos anteriormente no sean verdaderos.

Ejemplo básico:

```
x = 5

if x > 5:
    print('x es mayor que 5') #Devuelve False por lo que sigue recorriendo
los condicionales.
elif x < 5:
    print('x es menor que 5') #Devuelve False por lo que sigue recorriendo
los condicionales.
else:
    print('x es igual a 5') #Al no cumplirse los dos primeros condicionales
ejecuta el 'else'.
```

salida del terminal → x es igual a 5

1.1. Operadores ternarios:

Dentro de los condicionales existe lo que denominamos operadores ternarios que nos permiten expresar de manera más concisa los condicionales, a través de una única línea. Nos es de gran utilidad cuando queremos almacenar el resultado del condicional (valor) en una variable; aunque no conviene abusar de ellos ya que su legibilidad es menos clara. El ejemplo siguiente contiene la misma lógica que el anterior, simplemente cambia el modo en el que estructuramos el condicional.

Ejemplo básico:

```
x = 5

resultado = 'x es mayor que 5' if x > 5 else 'x es menor que 5' if x < 5
else 'x es igual a 5'
print(resultado)
```

salida del terminal → x es igual a 5

1.2. Condicionales compuestos:

Los condicionales también pueden ser compuestos, es decir, podemos combinar múltiples condiciones de manera simultánea. Los operadores que nos permiten esto son: 'and', 'or', 'not'.

- Operador 'and': devuelve True si todas las condiciones combinadas son True.

```
nombre_usuario = 'Jon'
contraseña = '12345'

if nombre_usuario == 'Jon' and contraseña == '12345':
    print('Acceso permitido')
else: #En caso de que no se cumpla la primera condición.
    print('Acceso denegado')
```

salida del terminal → Acceso permitido

- Operador 'or': si una de las condiciones devuelve True el condicional devuelve True.

```
nombre_usuario = 'Jon'
correo = 'otro@gmail.com'
contraseña = '12345'

if nombre_usuario == 'Jon' or correo == 'jon@gmail.com':
    print('Acceso permitido')
else:
    print('Acceso denegado')
```

salida del terminal → Acceso permitido

- Operador 'not': el operador niega la condición, devuelve True si la condición es False.

```
valor = 100

if not valor == 0:
    print('El valor no es igual a 0')
else:
    print('El valor es igual a 0')
```

salida del terminal → El valor no es igual a 0

- Condicionales con múltiples operadores: los condicionales también nos permiten combinar múltiples operadores.

  Imaginemos que estamos creando nuestro 'Log in' para la website y queremos que este sea algo flexible, permitiendo reconocer tanto nombre de usuario como correo electrónico como parte de las credenciales de acceso, además de la contraseña. El flujo del condicional sería el siguiente:

```
nombre_usuario = 'JonMada'
correo = 'otro@gmail.com'
contraseña = '12345'
usuario_premium = True

if (nombre_usuario == 'JonMada' or correo == 'jon@gmail.com') and
contraseña == '12345' and not usuario_premium:
    print('Acceso permitido con restricciones')
elif (nombre_usuario == 'JonMada' or correo == 'jon@gmail.com') and
contraseña == '12345' and usuario_premium:
    print('Acceso total a todos los servicios de la página')
else:
    print('Acceso denegado')
```

  El condicional establece que si el nombre de usuario o el correo devuelven True (alguno de los dos), la contraseña devuelve True, y que si 'usuario_premium' devuelve True, se transforma en False: imprimiendo acceso con restricciones. El segundo condicional es igual que el primero salvo que si 'usuario_premium' es True, da acceso total a todos los servicios de la página.

salida del terminal → Acceso total a todos los servicios de la página

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

Los loops o bucles en Python son estructuras de control que permiten ejecutar un bloque de código repetidamente a lo largo de los datos de una colección o mientras una condición se cumpla. Su utilidad reside en que automatiza el proceso de iteración (de repetición), ahorrándonos escribir el mismo código para ejecutar la misma acción sobre cada elemento de una determinada colección.

2.1. Bucle ‘for’

- El bucle for se utiliza para iterar (repetir una operación n veces) sobre una secuencia (lista, tupla, string...).

- La estructura básica es:

'for elemento in secuencia'

  - 'elemento' : representa una variable que toma el valor de cada elemento de una secuencia en cada iteración.

  - 'secuencia' : se refiere a la secuencia de elementos sobre los que se está iterando.

- Ejemplo aplicado:

```
marcas_coches = ['Opel','Mercedes','Audi','Seat','Aston Martin']

for marca_coche in marcas_coches:
    print(marca_coche) #Imprimirá cada elemento de la lista
```

salida del terminal →

Opel
Mercedes
Audi
Seat
Aston Martin

2.1.1. Continue

Dentro de los loops podemos incorporar el 'continue' para que cada vez que un determinado condicional se cumpla, el loop salte a la siguiente iteración, ignorando el código restante.

```
marcas_coches = ['Opel','Mercedes','Audi','Seat','Aston Martin']

for marca_coche in marcas_coches:
    if marca_coche == 'Mercedes': #Omitirá 'Mercedes' y no ejecutará
print(marca_coche) para ese elemento
        continue
    print(marca_coche)
```

salida del terminal →

Opel
Audi
Seat
Aston Martin

2.1.2. Break

‘Break’ detiene el loop si una determinada condición se cumple.

```
marcas_coches = ['Opel','Mercedes','Audi','Seat','Aston Martin']

for marca_coche in marcas_coches:
    if marca_coche == 'Mercedes': #Cuando el elemento es igual al
string 'Mercedes' el loop se detiene.
        break
    print(marca_coche)
```

salida del terminal →

Opel

2.2. Bucle ‘while’

El loop while establece un bucle infinito. Esto significa que el código dentro del bucle se ejecutará repetidamente mientras la condición sea 'True’.

- Ejemplo aplicado:

```
# Construimos un contador hasta 10
contador = 0

while contador <= 10: #Mientras el contador sea menor o igual a 10, itera sobre el código
    print(f'El contador es: {contador}')
    contador += 1 #Esto suma al contador una unidad en cada iteración
```

salida del terminal →

El contador es: 0
El contador es: 1
El contador es: 2
El contador es: 3
El contador es: 4
El contador es: 5
El contador es: 6
El contador es: 7
El contador es: 8
El contador es: 9
El contador es: 10

3) ¿Qué es una lista por comprensión en Python?

Es una forma de crear nuevas listas (estructura de datos) aplicando una expresión a cada elemento de la secuencia o iterando sobre la secuencia y filtrando los elementos según una condición.

- La sintaxis básica es:

nueva_lista = [expresión for elemento in secuencia if condición]

- Ejemplo aplicado:

```
#Creamos una lista que almacene los números del 1 al 30, y cuando el
número sea múltiplo de 3, lo ignore.

numbs = range(1,31)
multiplos_de_tres_deleted = [num for num in numbs if num % 3 != 0]
print(multiplos_de_tres_deleted)
```

salida del terminal →
[1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19, 20, 22, 23, 25, 26, 28, 29]

4) ¿Qué es un argumento en Python?

Los argumentos se utilizan para proporcionar datos de entrada a la función, los cuales pueden ser procesados y utilizados por la función para realizar alguna operación y producir un resultado. Asimismo, cabe remarcar que una función, no necesariamente tiene por que tener argumentos. Dentro de los argumentos podemos encontrarnos con diferentes tipologías.

4.1. Argumentos por defecto

A estos argumentos se les asigna un valor por defecto para determinada función. De este modo, en caso de que el desarrollador, no haya proporcionado un valor 'x' para el argumento de la función a la que está llamando, se establecerá ese valor previamente definido.

```
def saludo (name = 'Invitado'): #Aquí definimos el valor por defecto del
argumento 'name'
    print(f'Hola {name}')



saludo() # Salida --> Hola Invitado
saludo('Jon') # Salida --> Hola Jon
```

4.2. Argumentos posicionales

Cuando proporcionamos argumentos posicionales, Python los asigna a los parámetros de la función basándose en su posición en la llamada. Como veremos en el ejemplo, esta tipología puede llevar a error, en caso de que nuestra función contenga una gran cantidad de argumentos.

```
def saludar(saludo, nombre, apellido):
    print(f'{saludo}, {nombre} {apellido}')
```

```
saludar('Buenos días', 'Jon', 'Madariaga') # Salida --> Buenos días, Jon Madariaga
saludar('Jon','Buenos días', 'Madariaga') # Salida errónea --> Jon, Buenos días Madariaga
```

4.3. Named function arguments

Para resolver el problema de los argumentos posicionales, podemos especificar su nombre al llamar a la función, permitiendo que independientemente del orden en el que introducimos los argumentos, la función siga funcionando como deseamos.

```
def saludar(saludo, nombre, apellido):
    print(f'{saludo}, {nombre} {apellido}')


#El orden de entrada de los argumentos ya no afecta al funcionamiento
saludar(apellido='Madariaga', saludo='Buenos días', nombre='Jon')
```

salida del terminal → Buenos días, Jon Madariaga

4.4. Function argument unpacking

El operador '*' en Python nos permite desempaquetar los elementos de una secuencia durante una llamada a una función. Los argumentos serán desempaquetados y pasados individualmente a la función. La convención para nombrar dichos argumentos es 'args'.

```
def saludo (momento_día, *args):
      print(f'Hola {' '.join(args)}. Espero que estés teniendo una buena {momento_día}')


saludo('mañana', 'Jon', 'Madariaga')
```

'args' nos devuelve una tupla '('Jon', 'Madariaga') que estamos uniendo a través de espacios con la función join().

salida del terminal → Hola Jon Madariaga

4.5. Function keywords argument

Es una alternativa que nos permite el empaquetado de argumentos y a su vez utilizar los named arguments a la hora de llamarlos a la función, ya que crea un diccionario con su estructura clásica clave-valor. La convención determina que para nombrar dichos argumentos usemos la expresión 'kwargs'.

```
def saludo (momento_día, *args, **kwargs):
      print(f'Hola {" ".join(args)}. Espero que estés teniendo una buena
{momento_día}.')

    if kwargs: #Si hay kwargs...
        print('Esta es tu lista de tareas para hoy:')
        for key, value in kwargs.items():
            print(f'{key} : {value}.')



saludo('mañana',
        'Jon','Madariaga',
        primera_tarea = 'Realizar el cuestionario',
        segunda_tarea = 'Resolver los ejercicios prácticos')
```

Atendiendo al ‘for’, aplicamos un loop ‘for’ a dos elementos (key y value) a partir de los ítems extraídos (clave-valor) del diccionario generado por **kwargs.

salida del terminal →

Hola Jon Madariaga. Espero que estés teniendo una buena mañana.
Esta es tu lista de tareas para hoy:
primera_tarea : Realizar el cuestionario.
segunda_tarea : Resolver los ejercicios prácticos.

5. Funciones lambda

Son funciones anónimas y pequeñas que son definibles en una única línea. Son muy útiles cuando necesitamos una función rápida para operaciones sencillas. Se definen a través de la palabra clave 'lambda' seguida de los parámetros separados por comas, para posteriormente definir la expresión

- Ejemplo aplicado:

```
#Esta función lambda une en una string 'nombre' y 'apellido'
nombre_completo = lambda nombre, apellido: f'{nombre} {apellido}'

#Creamos una función para saludar que toma como argumento 'name'
def saludo (name):
    print(f'Buenos días {name}')


#Llamamos a la función saludo y le introducimos como argumento la
función lambda anteriormente creada.
saludo(nombre_completo('Jon','Madariaga'))
```

salida del terminal →

Buenos días Jon Madariaga

6. ¿Qué es un paquete pip?

Pip es una herramienta que nos permite instalar paquetes de Python Package Index y de otros repositorios creados por otros desarrolladores. Los paquetes contienen funciones que podemos llamar y que nos permiten realizar de forma simple y rápida sin apenas codificar, tareas que han sido programadas por otros desarrolladores.

- Para instalar un paquete que no se encuentre en nuestro entorno de trabajo la sintaxis es la siguiente (previamente tendremos que haber instalado PIP):

Desde el terminal --> pip install 'nombre del paquete'

- Si luego queremos trabajar con él, nuestro proyecto tendremos que importarlo:

import 'nombre del paquete'

¡CUIDADO! Hay situaciones en las que el nombre de la librería no coincide con el paquete a importar, por lo que es interesante revisar la documentación para comprender cómo funciona.

- Ejemplo de uso:

```
import numpy as np #Importamos la librería numpy y le otorgamos el alias 'np' para su llamada (convención y simplifica la llamada a las funciones)

num_range = np.arange(16) #Crea un array con 16 elementos
num_median = np.median(num_range) #Calcula la mediana de nuestro array 'num_range'
print(num_median)
```

salida del terminal →

7.5