Checkpoint 5

1.-¿Qué es un condicional?

Las **condicionales** son estructuras que permiten elegir entre la ejecución de una acción u otra. Son una condición, como bien indica su nombre, así que podemos pensar en ellos como si fueran el "si" condicional que usamos dentro de una frase. Por eso, utilizando la palabra en inglés, podemos expresar un condicional dentro de un código como **IF**

Son las declaraciones que permiten que el *software* tome decisiones dinámicamente en función de la evaluación de expresiones lógicas donde se ejecutará un conjunto de instrucciones si se cumple una condición, y otro conjunto si no se cumple.

la sintaxis básica de una declaración condicional sigue la estructura «si-entonces» Ejemplo:

```
if condición: # Bloque de código si la condición es verdadera

else: # Bloque de código si la condición es falsa

elif : # condiciones adicionales a verificar
```

Aquí, el programa evalúa la condición y ejecuta el bloque de código dentro del «if» si la condición es verdadera; de lo contrario, ejecuta el bloque dentro del «else».

1.1.-Tipos de condicionales

a) Sentencia *if*

La sentencia condicional más básica en Python es la sentencia if, la cual se expresa de la siguiente forma:

```
if condicion:
    # ejecutar un código
```

En la expresión previa:

- La condición es una expresión booleana que se evalúa como verdadera (True) o falsa (False).
- Se requiere el uso de dos puntos (:) al final de la condición.
- Todas las líneas de código a ejecutar si se cumple la condición tienen que estar indentadas respecto la sentencia if.

La **indentación** es una característica que diferencia Python de otros lenguajes de programación, dónde el código a ejecutar de cumplirse la condición se encierra entre llaves. Esta característica tiene el propósito de mejorar la legibilidad de los programas.

Veamos el uso de la sentencia if con un ejemplo:

```
x = 15

if x > 10:

  print('x es mayor que 10')     output :  x es mayor que 10
```

En el ejemplo empezamos asignando a la variable x el valor 15. A continuación evaluamos si cumple la condición de que la variable x es mayor a 10. Como se da el caso, la condición se evalúa como verdadera. Es por ello que el programa ejecuta la sentencia print().

b) Sentencia *else*

A la sentencia if se le puede añadir opcionalmente una sentencia else. Esta sentencia contiene el código a ejecutar en caso de que no se cumpla la condición de la sentencia if. Esta estructura se expresa del siguiente modo:

```
if condicion:
# ejecutar un código
else:
# ejecutar un código distinto
```

Veamos un ejemplo parecido al anterior al que se le ha añadido una sentencia else:

```
x = 5
if x > 10:
  print('x es mayor que 10')
else:
  print('x es menor o igual que 10')  ->  Output x es menor o igual que 10
```

En este caso a la variable x se le asigna el valor 5. Como ahora no es verdad que x sea mayor que 10, la condición de la sentencia if se evalúa como falsa. Por tanto, el programa ejecuta la instrucción bajo la sentencia else.

c) Sentencia *elif*

A una sentencia if else se pueden añadir un número indefinido de condiciones adicionales a verificar. Estas condiciones se definen mediante la sentencia elif , la cual es un abreviación de else if. Ésta se define así:

```
if condición:
# ejecutar un código
elif otra condición:
# ejecutar otro código
else:
# ejecutar un código distinto
```

Siguiendo con el ejemplo de los números, podemos añadir una sentencia elif del siguiente modo:

```
x = 7
if x > 10:
  print('x es mayor que 10')
elif x < 10:
  print('x es menor que 10')
else:
  print('x es 10')                    Output : x es menor que 10
```

En este caso como x tiene asignado el valor 7, la condición de la sentencia elif se evalúa como verdadera. Por este motivo el programa ejecuta la instrucción print() asociada a esta sentencia.

Un punto importante de las sentencias if elif es que una cuando una condición es evaluada como verdadera se ignoran el resto de condiciones.

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

En Python, el código se ejecuta de manera secuencial. Es decir, si lo programado se compone de varias líneas, la ejecución de lo que se ha programado comenzará por la primera, pasará a la segunda y así sucesivamente.

Sin embargo, existen casos en los que no se quiere esto, sino que es necesario que un bloque determinado se repita.

Gracias a los bucles, podemos hacer que estos **fragmentos de código** se repitan en la ejecución del programa desarrollado en Python.

Existen distintos tipos de bucles en programación, Python cuenta con **dos tipos de bucles fundamentales:**

- **El bucle *while*** ejecutan los bloques de código a los que afecta siempre y cuando la

condición del bucle sea verdadera. El bucle while se repite hasta que la condición se ha cumplido un número determinado de veces.

Es similar a la función if, sin embargo, el bucle *while* en Python se ejecuta varias veces mientras que los if tan solo se ejecutan una vez.

Sintaxis:

while <condición>:

<bloque de código>

En este bucle hay que verificar la expresión ya que es cuando esta se vuelve verdadera cuando se ejecuta el bucle. En cada repetición del bucle, se comprobará esa expresión y cuando sea falsa, el bucle llegará a su fin.

- **El bucle *for*** se basan en iteradores que recorren todos los elementos de aquellos objetos que sean iterables: listas, tuplas, cadenas. Ejecuta una pieza de código repetidamente en función de la **cantidad de elementos** que contenga el objeto iterable.

Sintaxis:

for <variable> **in** <secuencia>:

<bloque de código>

. En este caso, *elem* hace referencia a la variable de la que parte el iterador mientras que el *iterable* es el elemento sobre el que variable aplica el loop. En este caso, el bucle se repetirá hasta que se completen todos los valores de la secuencia de elementos del *iterable*.

Además, los bucles en Python se pueden controlar. Para ello se utilizan las siguientes expresiones:

- ***Break:*** Pone fin al bucle, lo rompe.

- ***Continue:*** Evita todo el código que existe debajo y vuelve al inicio del bucle.

En el caso de que no se indique adecuadamente la ruptura de los bucles en Python se crea un loop infinito. Es decir, el programa ejecuta continuamente el mismo fragmento de código. Esto puede ser un error o pueden programarse **loops infinitos con finalidades concretas.**

3.-¿Qué es una lista por comprensión en Python?

La comprensión de listas, del inglés **list comprehensions**, es una funcionalidad que nos permite crear listas avanzadas en una misma línea de código.

```
# Método tradicional
lista = []
for letra in 'casa':
   lista.append(letra)
print(lista)                Output:  ['c', 'a', 's', 'a']
```

```
# Con comprensión de listas
lista = [letra for letra in 'casa']
print(lista) )              Output:  ['c', 'a', 's', 'a']
```

Como vemos, gracias a la comprensión de listas podemos indicar directamente cada elemento que va a formar la lista, en este caso la letra, a la vez que definimos el for:

```
# La lista está formada por cada letra que recorremos en el for
lista = [letra for letra in 'casa']
```

4.-¿Qué es un argumento en Python?

Los argumentos son los valores que pasamos a una función con un parámetro definido.

Una función de un lenguaje de programación como Python está representada por la **palabra def, un parámetro que la nombre y argumentos** que se escriben entre paréntesis. Esta función, además, necesita de un comando que la imprima, este es **print( ).**

```
Sintaxis de una Función
def un_parámetro («argumentos»)
Representación de una función
```

Python permite el uso de argumentos posicionales y de palabras clave.
Los **argumentos posicionales** son aquellos que son pasados en el orden en el que se definen en la función. Los **argumentos de palabras clave** son aquellos en los que se especifica el nombre del argumento antes de su valor

- **argumentos posicionales:** Son los argumentos que se envían a una función en el orden en que se definieron, es decir, el primer argumento será el primero que se envíe, el segundo será el segundo, y así sucesivamente.

Utilizar argumentos posicionales puede ser muy útil cuando se requiere una flexibilidad de argumentos, ya que, al enviar los argumentos en orden, no es necesario preocuparse por escribir la lista completa de argumentos y sus valores en la llamada a la función.

Ejemplo argumento posicional:

```
def resta(a, b):
   return a - b
resta(30, 10)  # argumento 30 => posición 0 => parámetro a
               # argumento 10 => posición 1 => parámetro b
```

Output: 20

- **Argumentos de nombre :** es posible evadir el orden de los parámetros si indicamos durante la llamada que valor tiene cada parámetro a partir de su nombre:

Ejemplo argumento de nombre:

Continuando con el anterior ejemplo

```
resta(b=30, a=10)
```

Output: -20

5.-¿Qué es una función Lambda en Python?

En Python, una función lambda es una función anónima y de una sola línea que se puede definir en el lugar donde se necesita, sin tener que asignarle un nombre. Estas pequeñas pero poderosas funciones pueden tener argumentos y devolver resultados, lo que las convierte en una herramienta muy versátil.

A diferencia de las funciones regulares que se definen con la palabra clave def, las funciones lambda se definen utilizando la palabra clave lambda.

La **sintaxis básica** de una función lambda es la siguiente:

```
lambda argumentos: expresión
```

Aquí, «argumentos» son los parámetros que la función lambda puede recibir, y «expresión» es el código que se ejecuta y devuelve un resultado.

Ejemplo:

```
sumar = lambda x, y: x + y
print(sumar(5, 3))
```

Output: 8

En este caso, *lambda x*, y define una función anónima que toma dos argumentos, x e y, y devuelve su suma. La belleza de las funciones lambda reside en su simplicidad y en la capacidad de escribir funciones pequeñas de manera rápida y eficiente.

6.-¿Qué es un paquete pip?

pip es una herramienta esencial para cualquier desarrollador de Python. Permite instalar, actualizar y administrar paquetes de software con facilidad, lo que a su vez facilita el desarrollo y la distribución de aplicaciones y proyectos de Python.

Además, **pip** hace que sea fácil compartir código y colaborar con otros desarrolladores en la comunidad de Python

Los paquetes de software son conjuntos de módulos y bibliotecas que pueden ser reutilizados en diferentes proyectos. Los paquetes pueden proporcionar funcionalidades que van desde operaciones matemáticas hasta acceso a bases de datos y análisis de datos, entre otros.

La ventaja de usar un gestor de paquetes como **pip** es que simplifica el proceso de instalación y actualización de paquetes, así como la gestión de dependencias entre ellos.
Además, **pip** permite a los desarrolladores compartir sus propios paquetes con la comunidad y descargar paquetes creados por otros desarrolladores para usarlos en sus proyectos.

Usos principales:

1. **Creación y distribución de paquetes propios**: Además de ser una herramienta para la instalación y administración de paquetes, **pip** también permite a los desarrolladores crear y distribuir sus propios paquetes. Esto permite a la comunidad de Python compartir sus proyectos y código con otros.

2. **Instalación de paquetes en un entorno virtual**: **pip** se puede usar junto con herramientas de entorno virtual, como **virtualenv** o **venv**, para instalar paquetes en un entorno aislado, lo que es útil para evitar conflictos de versiones entre paquetes o para mantener separados los paquetes de diferentes proyectos.

3. **Especificación de versiones de paquetes**: Al instalar o actualizar paquetes, puedes especificar una versión particular del paquete que deseas instalar utilizando el operador **==**. Por ejemplo: **pip install nombre_del_paquete==1.0.0**.

4. **Instalación de paquetes desde un archivo de requerimientos**: Puedes crear un archivo de requerimientos (usualmente llamado **requirements.txt**) que liste todos los paquetes y sus versiones que tu proyecto necesita. Luego, puedes usar **pip** para instalar todos esos paquetes de una sola vez: **pip install -r requirements.txt**.

5. **Mostrar información sobre un paquete**: Puedes usar el comando **pip show** para mostrar información detallada sobre un paquete instalado, como su versión, ubicación, dependencias y más.

6. **Buscar paquetes**: **pip** tiene un comando de búsqueda que permite buscar paquetes en el Índice de Paquetes de Python (PyPI) directamente desde la línea de comandos: **pip search nombre_del_paquete**.