CHECKPOINT 5 – EJERCICIO TEÓRICO

1. ¿Qué es un condicional?

Las sentencias condicionales son aquellas que permiten que dentro del flujo del programa se tomen diferentes caminos para ejecutar bloques de código en función de condiciones establecidas.

Un condicional es una manera de generar un sistema de decisiones en el código basado en condiciones lógicas (cuando hay que comprobar más de un requisito a la vez) y en condiciones de comparación (cuando la condición es única).

Python entiende cada condición como verdadera o falsa, y actúa según dicha evaluación.

Un operador de comparación es, lo que colocamos dentro de nuestra condición para que busque elementos, así como igualdad o desigualdad, lo cual significa, ¿el objeto A es igual al B, o no es igual al B? Y también podemos experimentar con rangos de valores, así como ver si un elemento es mayor que, o igual que un valor, o si es menor que, o igual que otro valor. Esto nos permite no solo añadir comportamientos dinámicos, sino además nos brinda una gran cantidad de resultados verdaderos o falsos.

Lista completa de operadores de comparación:

== Igual

!= Desigual / Distinto a

<> Desigual → Obsoleto

> Mayor que

>= Mayor o igual que

< Menor que

<= Menor o igual que

} Únicamente con números

Los principales condicionales en Python son **if, else** y **elif:**

A continuación, vamos a ver más en detalle las principales condicionales con sus correspondientes ejemplos.

Single condition

Ejemplo:

```
edad = 16


if edad < 18:
  print("Lo siento, necesitas tener al menos 18 años")
```

if/else

Tiene la siguiente estructura:

```
if CONDICIÓN:
        Bloque de código cuando la condición es verdadera
else:
        Bloque de código cuando la condición no se cumple
```

Ejemplo:

```
edad = 55
if edad < 18:
  print("Lo siento, necesitas tener al menos 18 años")
else:
  print("Tienes la edad correcta, puedes continuar")
```

if/elif/else

Tiene la siguiente estructura:

```
if CONDICIÓN:
        Bloque de código cuando la condición es verdadera
elif:
        Bloque de código que permite agregar una nueva condición
else:
        Bloque de código cuando las condiciones anteriores no se cumplen
```

Ejemplo:

```
edad = 55


if edad < 18:
  print("Lo siento, necesitas tener al menos 18 años")
elif edad > 100:
  print("Lo siento, necesitas tener menos de 100 años")
else:
  print("Tienes la edad correcta, puedes continuar")
```

Condicionales compuestos

Las declaraciones con condicionales compuestos, significa tener múltiples condiciones dentro de un programa Python.

Las declaraciones compuestas contienen (grupos de) otras declaraciones; afectan o controlan la ejecución de esas otras declaraciones de alguna manera. En general, las declaraciones compuestas abarcan varias líneas, aunque si son declaraciones simples, pueden estar contenidas en una única línea.

Tenemos varias opciones para realizar las declaraciones.

'and'

Si se quiere que sean dos parámetros correctos, en este caso el username y la password.

Si ambas condiciones se cumplen entonces el operador 'and' retornara un 'True', si no se cumple un 'False'.

```
username = 'sara'
email = 'sara@gmail.com'
password = 'contraseña'


if username == 'sara' and password == 'contraseña':
  print('Acceso permitido')
else:
  print('Acceso no permitido')
```

Otra manera de realizar lo mismo, pero menos recomendado de usar sería:

```
username = 'sara'
email = 'sara@gmail.com'
password = 'contraseña'

if username == 'sara' :
        if password == 'contraseña':
                print('Acceso permitido')
else:
  print('Acceso no permitido')
```

'or'

Si se quiere que únicamente uno de los dos parámetros facilitados sea correcto, en este caso o el username o la password. Le estas diciendo al programa que ejecute la función si alguna de las dos condiciones es verdadera 'True'.

El operador 'or' funciona diferente que 'and', este retornará 'True' siempre y cuando una de las dos partes de la función sea verdadera.

```
username = 'sara'
email = 'sara@gmail.com'
password = 'contraseña'

if username == 'sara' or password == 'contraseña':
  print('Acceso permitido')
else:
  print('No permitido')
```

Compuesto **'and'** y **'or'**

También se aplica el orden de las operaciones por lo que, lo primero que hará es entrar en el paréntesis y lo tratará como una sola expresión. Mientras uno u otro de la expresión sea correcto, entonces tratara el objeto completo como verdadero pasando al siguiente punto el "and"

```
username = 'sara'
email = 'sara@gmail.com'
```

```
password = 'contraseña'


if (username == 'sara' or email == 'sara@gmail.com') and password == 'contrase
ña':
  print('Acceso permitido')
else:
  print('No permitido')
```

'and not':

El operador 'and not' esencialmente lo que hace es que invierte el 'boolean' que le da una función, es decir, Si una expresión se evalúa como True, si se usa 'not' devolverá False, y viceversa. Usamos "and not" cuando queremos que la primera condición sea verdadera y la segunda condición sea de forma obligatoria falsa

```
carnet_b = True
multas = False


if carnet_b == True and not multas == True:
  print('Puedes alquilar un vehículo')
else:
  print('No se te permite alquilar un vehículo')
```

2. ¿Qué tipo de bucles hay en JS?

Un bucle es una instrucción que se repite varias veces mientras se cumpla una condición. Cuando se deje de cumplir, se saldrá del bucle y se continuará la ejecución normal.

Los bucles son una herramienta muy importante en la programación, ya que nos permiten ejecutar un bloque de código varias veces seguidas. Se trata de uno de los conceptos básicos en programación.

Hay dos tipos principales de bucles: los bucles «while» y los bucles «for...in». Ambos pueden usarse para iterar colecciones, un rango de números, listas, cualquier cosa parecida. Pero hay una diferencia clave:

“for...in

Es la capacidad de realizar cierta tarea tantas veces como elementos haya dentro de la lista, tupla o diccionario. Sirve para cuando sabemos el número de repeticiones que vamos a tener que hacer basándonos en un número definido de datos, por ejemplo, si tenemos una lista con 50 datos, sabremos que el bucle “for...in” va a hacer 50 repeticiones.

Es algo muy positivo, porque como desarrolladores, no hay que registrar ni saber cuántos ítems tiene la lista, tupla o diccionario, ya que sea cual sea el número, las repeticiones van a ser una por cada ítem y una vez termine con los ítems, entonces el bucle “for...in” finalizará. Debido a esto, es el bucle más utilizado eligiéndolo un 95% de las ocasiones en general.

```
asignaturas = ['lengua', 'matemáticas', 'informática']
for asignatura in asignaturas:
  print(asignatura)
```

“while”

Este bucle funciona con condiciones, es decir, se ejecutará y realizará repeticiones siempre que se cumpla una condición, una vez que la condición se ha cumplido, el bucle terminara.

Se utiliza sobre todo cuando no se conoce de antemano el número de repeticiones del programa. Un bucle while continuara tantas veces como queramos.

```
num = 6
while num > 0:
 num -=1
 print(num)
```

Un ejemplo de un caso en el cual no se podría utilizar un bucle “for...in” porque no se sabe cuando debe terminar el bucle, sería en el caso de un juego en el cual se tiene que acertar con un número en concreto:

```
def acertar_numero():
  while True:
    print('Introduce un número del 1 al 10')
    num = input()

    if num == '7':
```

```
        print('Felicidades, has acertado')
        return False
    else:
        print(f"No, el {num} no es el número correcto, por favor pruebe de nuevo\n"
)


acertar_numero()
```

Si le realizamos hacer una tarea y no le dijera al bucle cuando terminar, incluso después de pasar por todos los elementos de la lista, tupla o diccionario continuaría ejecutándose. Si no se implementa un while apropiadamente nos toparemos con un bucle infinito y el programa nunca se detendrá generando que con el tiempo, el pc o servidor acabe dando fallos. A la hora de trabajar con un bucle “while”, como desarrollador es imprescindible decirle cuando detenerse. Hay que definir cual es el punto en el que tiene que finalizar, el cual se llama valor centinela. Por lo cual hay que establecer un valor centinela para poder decirle a tu bucle while cuando detenerse. Se usa sobre todo cuando no se tiene claro el valor final, cuando no sabemos cuándo tiene que terminar.

Ejemplo de un bucle infinito:

```
nums = list(range(1, 21))
while len(nums) > 0:
  print(nums)
```

Ejemplo de bucle while con valor centinela:

```
nums = list(range(1, 21))
while len(nums) > 0:
  print(nums.pop())
```

Mediante la función pop() itera la lista, sacando un elemento. Va a imprimir un valor y lo elimina de la lista nums, por lo tanto, va a crear el valor centinela reduciendo continuamente la longitud de la lista hasta que llegue a 0. Una vez que llegue a 0 ya no cumplirá la condición por lo que el bucle se detendrá.

3. ¿Cuáles son las diferencias entre const, let y var?

Las variables son un concepto fundamental en cualquier lenguaje de programación. En JavaScript, puedes declarar variables usando las palabras clave var, const o let.

Las variables son la manera como los programadores le dan nombre a un valor para poder reusarlo, actualizarlo o simplemente registrarlo. Las variables se pueden usar para guardar cualquier tipo de dato en JavaScript.

Sin embargo, hay diferencias importantes.

Var

Cuando usas la palabra clave var, le estás diciendo a JavaScript que vas a declarar una variable.

Al usar var, las variables pueden ser reasignadas.

```
var nombre = 'Sara';
var nombre = 'Eva';
console.log(nombre);
```

En este ejemplo, declaramos la variable una segunda vez, procediendo a modificar la variable.

Al usar la palabra clave var, las variables también pueden ser declaradas sin valor inicial, para posteriormente mediante el operador de asignación, asignarle un valor.

```
var nombre;
nombre= 'Sara';
```

var, puede tener un ámbito global o local; puede ser redeclarada y/o modificada y se elevan. Será global si la variable se declara fuera de una función, es decir, se puede hacer referencia a ella en cualquier momento del código, y local si se declara dentro de una función, es decir, sólo se podrá utilizar dentro de esa función.

```
var nombre = 'Eva';


function prueba() {
  var apellido = 'López';
  if (nombre == 'Sara') {
    console.log(nombre + '' + apellido);
  }
  else {
    console.log('No es Sara');
  }
```

En este caso se observa que nombre = 'Sara' se declara de manera global porque está fuera de la función y apellido = 'López' es local porque se declara dentro. Pero esta flexibilidad puede crear problemas si, por ejemplo, declaramos una variable en un ámbito global y sin darnos cuenta de ello, la redeclaramos y la modificamos, perdiendo así el valor asignado en la primera declaración.

El hoisting es un mecanismo en el que las variables y declaraciones de funciones se mueven a la parte superior de su ámbito antes de la ejecución del código. Ejemplo:

Si escribimos esto:

```
console.log (nombre);
var nombre = 'Sara';
```

JS interpretará esto:

```
var nombre;
console.log(nombre);
nombre = 'Sara';
```

Let

Cuando usas la palabra clave let, le estás diciendo a JavaScript que vas a declarar una variable.

La palabra clave let tiene un ámbito de bloque (código que está delimitado por {}).

```
let nombre = "Sara";
function prueba() {
let apellido = “López”;
console.log(apellido);
}
console.log(apellido);
```

El último console.log producirá un error ya que apellido únicamente se encuentra dentro de la función, así como nombre se encuentra fuera y no se podría utilizar dentro de la función.

let puede modificarse, pero no declararse más de una vez dentro de un mismo ámbito.

```
let nombre = "Sara";
nombre = "Eva";
```

En este ejemplo daría error, diciendo que la variable ya está declarada.

Sin embargo, sí podemos en diferentes ámbitos aprovechando global y local:

```
let nombre = "Sara";
if (true){
let nombre = "Eva";
let apellido = "López";
console.log(nombre); // “Eva”
console.log(apellido); // “López”
 }
console.log(nombre); // ”Sara”
```

let se eleva a la parte superior, aunque se comportará de manera diferente a var ya que no se inicializará como undefined. Si se intenta usar una variable let antes de declararla, se obtendrá un Reference Error.

Const

Cuando usas la palabra clave const, le estás diciendo a JavaScript que vas a declarar una variable.

const, tiene alcance de bloque(código que está delimitado por {}) por lo que tienen que ser utilizadas dentro del bloque en el que han sido declaradas, como en el caso de let.

No pueden ser ni modificadas ni declaradas varias veces; y aunque se elevan no se inicializan. Es necesario aclarar que, aunque la variable guardada como const no se puede modificar, si se trata de un objeto, las variables de este sí se pueden actualizar.

```
const estudiante = {nombre: "Eva", departamento: “psicología”}
estudiante.departamento = “matemáticas”;
```

4. ¿Qué es una función de flecha?

Una función de flecha es una forma más concisa de escribir funciones anónimas, es decir que no tienen nombre.

Para construir una función de flecha seguimos los siguientes pasos:

a) Quitamos la palabra function y se coloca la flecha entre el argumento y el corchete de apertura.

```
var sumarValores = (x, y) => {
  return x + y
}
```

b) Se quitan los corchetes del cuerpo y la palabra "return" — el return está implícito

```
let sumar = (a,b) => a+b;
```

Si la función es multilínea, se mantienen los corchetes y el "return".

```
let reserva = (habitacion) => {
  if (habitacion === "libre" {
    return "Habitación disponible";
  }
  else {
    return "Habitación no disponible";
    }
```

```
}
```

c) Se suprimen los paréntesis de los argumentos si hay sólo uno o dos. Si no hay argumentos o si son más de dos se deben mantener.

```
let sumar = a,b => a+b;
```

Es importante notar que las funciones flecha son anónimas, lo que significa que no tienen nombre. Este anonimato puede crear algunos problemas:

- Más difíciles de depurar: Cuando haya algún error, no se puede rastrear el nombre de la función o el nº de línea donde ocurrió.
- Sin autorreferencia: si la función necesita tener autorreferencia, no funcionará.

Aunque las funciones flecha pueden ser útiles, pero no reemplazan a las funciones clásicas. Hay que tener en cuenta que mantienen una relación diferente con ciertos parámetros, como por ejemplo this.

En las funciones clásicas, this está vinculada a diferentes valores en función del contexto mientras que las funciones flecha no tienen this y por tanto acceden al valor léxico anterior.

5. ¿Qué es la deconstrucción variable?

La deconstrucción o desestructuración de una variable es una expresión de JS que permite extraer datos o elementos de arrays u objetos y asignarlos a variables.

```
let introduccion = [‘Hola’, ‘soy’, ‘Eva’];
let saludo = introduccion[0];
let nombre = introduccion[2];


console.log(saludo); // "Hola"
console.log(nombre); // "Eva"
```

Esto se puede realizar de otra manera:

```
let [saludo,,,nombre] = ["Hola", "yo" , "soy", "Eva"];


console.log(saludo); // "Hola"
console.log(nombre); // "Eva"
```

Como se puede observar, en al lado izquierdo de la asignación de variables del arreglo, en lugar de tener solo una coma, tenemos tres. El separador de coma se utiliza para omitir valores en un arreglo. Pudiendo omitir también el primer elemento mediante el uso de comas.

```
let [,pronombre,,nombre] = ["Hola", "Yo" , "soy", "Eva"];


console.log(pronombre); // "Yo"
console.log(nombre); // "Eva"
```

6. ¿Qué hace el operador de spread JS?

El operador spread son los puntos suspensivos ("...") que aparecen en el llamado de una función o similar. Ejemplo:

```
function sum(x, y, z) {
  return x + y + z;
}
var numbers = 1, 2, 3;
console.log(sum(...numbers)); // 6
```

En este ejemplo se están pasando los valores (1,2,3) de manera dinámica a una función

Además, también puede ser utilizado para:

a) Concatenar arrays:

```
let array1 = 1, 2, 3;
let array2 = 4, 5, 6;
let concatenarArray = ...arr1, ...arr2;
console.log(concatenarArray); // 1, 2, 3, 4, 5, 6
```

b) Desglosar los caracteres de una cadena (string) en un array

```
var str = "Hola";

var charArray = ...str; console.log(charArray); // 'H', 'o', 'l',
'a'
```

c) Se puede fusionar las propiedades de dos objetos.

```
var obj1 = { a: 1, b: 2 };

var obj2 = { b: 3, c: 4 };

const fusionarObj = { ...obj1, ...obj2 }; console.log(fusionarObj)
; // { a: 1, b: 3, c: 4 }
```

d) Se puede clonar las propiedades de un objeto.

```
var obj1 = { a: 1, b: 2 };

var obj2= { ...obj1 };

console.log(obj2); // { a: 1, b: 2 }
```

7. ¿Qué es OOP?

OOP (Object Oriented Programing o Programación orientada a objetos) es un modelo de programación informática que organiza el diseño del software en torno a datos u objetos, en lugar de funciones y lógica. Un objeto se puede definir como un campo de datos que tiene atributos y comportamientos únicos.

Se centra en los objetos que los desarrolladores quieren manipular en lugar de enfocarse en la lógica necesaria para manipularlos. Este enfoque es adecuado para programas que son grandes, complejos y se actualizan o mantienen activamente, siendo también beneficioso para el desarrollo colaborativo, donde los proyectos se dividen en grupos.

Se basa en los siguientes principios:

- Encapsulación: la implementación y el estado de cada objeto se mantiene de forma privada dentro de un límite definido o clase.
- Abstracción: Los objetos solo revelan mecanismos internos que son relevantes para el uso de otros objetos, ocultando cualquier código de implementación innecesario.

- Herencia: Se pueden asignar relaciones y subclases entre objetos, lo que permite a los desarrolladores mantener una lógica común y reutilizarla sin dejar de mantener una jerarquía única.
- Polimorfismo: Los objetos pueden adoptar más de una forma según el contexto.

8. ¿Qué es una promesa JS?

Una promesa es un objeto que representa un valor que puede estar disponible ahora, en el futuro o nunca. Como no se sabe cuando va a estar disponible, todas las operaciones dependientes de ese valor, tendrán que posponerse en el tiempo. Esta representación encapsula el resultado (éxito o error) de una operación asíncrona.

En otras palabras, proporcionan una forma más estructurada y legible de manejar operaciones asíncronas, permitiendo que el código sea más claro y más fácil de mantener al escribir acciones de JavaScript.

Una operación asincrónica permite al programa iniciar una tarea de larga duración y seguir respondiendo a otros eventos mientras esa tarea se ejecuta, en lugar de tener que esperar hasta que esa tarea haya terminado. Una vez que dicha tarea ha finalizado, el programa presenta el resultado.

Los métodos más comunes que se pueden realizar son:

- .then(): este método se usa para manejar el resultado exitoso de una promesa.
- .catch(): Se utiliza para manejar errores que puedan ocurrir durante la ejecución de la promesa.
- .finally(): Se utiliza para ejecutar una función después de que la promesa se resuelva o se rechace, independientemente del resultado.
- Promise.all(iterable): permite manejar múltiples promesas al mismo tiempo y resuelve una promesa una vez que todas las promesas del iterable se hayan resuelto o alguna de ellas se haya rechazado.
- Promise.race(iterable): resuelve una promesa tan pronto como una de las promesas en el iterable se resuelva o se rechace.

9. ¿Qué hacen async y await por nosotros?

Async y await nos permiten transformar un código asíncrono para que parezca ser síncrono, es decir, un código donde cada instrucción espera a la anterior para ejecutarse.

La palabra clave async se pone delante de una declaración de función para convertirla en una función asincrónica, es decir, para que devuelvan una promesa en vez de un valor. Dentro de esta función podemos usar la palabra clave wait para invocar el código asíncrono.

La palabra await nos permite definir una sección de la función a la cual el resto del código debe esperar, es decir, definimos que el código restante de la función debe esperar a que esta sección se resuelva.

Ejemplo:

```
async function queue(){ var book = await getBook();
var main = await getMain(book);
console.log(main); } queue();
```

En este ejemplo, la función getBook() se conecta a un hipotético servicio web que devuelve un libro al azar, y getMain() devuelve el nombre del protagonista del libro. En vez de usar then(function(){}) usamos await para que parezca que las funciones devuelven valores síncronos en vez de promesas.

El código es más legible y cómodo de escribir.