# 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')
```

¿Qué es un condicional?
¿Cuáles son los diferentes tipos de bucles en Python? ¿Por qué son útiles?
¿Qué es una lista por comprensión en Python?
¿Qué es un argumento en Python?
¿Qué es una función Lambda en Python?
¿Qué es un paquete pip?

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

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. ¿Qué es una lista por comprensión en Python?

La compresión de listas nos permite crear listas de elementos en una sola línea de código, es esencialmente un conjunto de bucles for y condicionales que se pueden colocar dentro de una sola línea de código.

```
num_list = range(1, 11)
cubed_nums = [num ** 3 for num in num_list]


print(cubed_nums)
```

Lo cual es lo mismo que:

```
num_list = range(1, 11)
cubed_nums = [] →


for num in num_list:
  cubed_nums.append(num ** 3)


print(cubed_nums)
```

Es importante remarcar que si se intenta ejecutar la comprensión de listas y no se envuelve el código entre corchetes [ ], Python indicará un error de sintaxis.

Por otro lado, destacar que hay que tener cuidado con su uso, ya que las compresiones muy largas pueden ser muy difíciles de leer y en muchas ocasiones puede llevar a confusión. Debido a ello es raro su uso, con el fin de facilitar su lectura y comprensión.

### 4. ¿Qué es un argumento en Python?

Los términos parámetro y argumento son utilizados para el mismo concepto: información que se pasa a una función.

Desde la perspectiva de una función:

- Un parámetro es la variable que aparece entre paréntesis en la definición de la función.
- Un argumento es el valor que se envía a la función cuando ésta es llamada.

**Argumentos de posición**

```
def suma(a, b, c):
return a + b + c
```

En este caso hemos definido la función suma. Función que posee 3 parámetros (a, b y c). Estos parámetros no poseen valores por default, esto quiere decir que, al hacer el llamado a la función, será necesario definir sus valores. Y para ello la forma más sencilla es utilizar argumentos.

Simplemente, al hacer el llamado a la función, colocamos los argumentos. En este caso 3.

```
suma(10, 5, 15) // 30
```

Este sería un ejemplo de argumento por posición ya que a corresponde al primer argumento añadido (10), b al segundo (5) y así sucesivamente.

Sin embargo, cuando se trata de trabajar con programas más grandes y una funcionalidad más avanzada, los argumentos posicionales pueden generar cierta confusión, ya que puedes pensar que estas llamando y transmitiendo valores en el orden correcto, pero puedes omitir uno y de repente todo el programa se desactiva. Y eso es algo que a evitar.

**Argumentos con nombre**

En Phyton los argumentos con nombre te dan la habilidad de ser mucho más explícitos con este mapeo. Po lo que cuando se llama a la función en vez de pasar el valor, se puede le puede llamar mediante el nombre que se le ha dado.

```
def suma(a, b, c):
  return a + b + c


suma(b=5, c=15, a=10)
```

El orden no cambia el comportamiento debido a que los argumentos con nombre declaran claramente el mapeo, entonces permite pasar cualquier valor en el orden que prefieras.

**Argumentos por defecto**

La sintaxis de un argumento predeterminado es dentro de la línea de definición, decimos name = y entonces pasamos lo que sea el argumento por defecto.

Es una manera muy buena y limpia de hacer que tu sistema sea bastante dinámico, poder llamar a la función y poder pasarle argumentos, o pueden dejar esos argumentos en blanco.

```
def greeting(name = 'Guest'):
  print(f'Hi {name}!')


greeting() // Hi Guest!
greeting('Sara') // Sobreescribe el argumento predeterminado devolviendo Hi Sara!
```

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

Una función Lambda sirve para poder encerrar en una variable los argumentos que se quieran pasar a una función y de esta forma poder reutilizarse y además no se necesita que aparezca en la función todos los argumentos, pueden ser dinámicos con lo que se puede aprovechar mejor las funciones ya creadas.

Lambda es una herramienta que nos permite empaquetar una función, en general una función más pequeña y luego introducirla en otras funciones. Las funciones dentro de Phyton son lo que llamamos “ciudadanos de primera clase” es decir, que podemos tratar una función como cualquier tipo de objeto. Entonces cuando trabajamos con lambdas, podemos empaquetar un comportamiento, en general un comportamiento pequeño y luego introducirlo en otras funciones.

Son muy móviles y fáciles de usar. Pueden pensar que una lambda es muy similar de una variable que se puede introducir, en lugar de un valor básico como una cadena, diccionario o algo parecido. Un lambda nos permite empaquetar un proceso.

Sintaxis para crear una lambda: escribir lambda y seguido por la lista de argumentos después la llamada (:) y tras eso lo que va a devolver:

```
full_name = lambda first, last: f'{first} {last}'


def greeting(name):
  print(f'Hi there {name}')


greeting(full_name('Kristine', 'Hudgens')) // “Hi there Tiffany Hudgens”
```

En este ejemplo, lo primero buscara la lambda, ingresara los dos valores, y luego lambda ejecutara su proceso. En este caso está formateando una cadena, se almacena en la variable full_name, luego podemos usarla como cualquier otra variable.

## 6. ¿Qué es un paquete pip?

PIP es un acrónimo recurrente que significa “pip installs package” o “pip installs Phyton”, y lo que nos permite hacer es introducir paquetes externos almacenados en la tienda PyPI o en CheeseShop, ahorrando mucho tiempo y esfuerzo. Nos permite introducir código Phyton que otros desarrolladores crearon para poder usarlo en nuestra propia aplicación. Por lo tanto, gracias a PIP no es necesario escribir código desde cero, sino que hay muchas funciones ya existentes que se pueden importar al código propio y poder crear la aplicación o web más rápido y con más estabilidad, ya que es código que funciona correctamente por haberse probado y usado por la comunidad.

Para poder usarlo, primero hay que instalarlo en el sistema y el proceso de instalación utilizará Phyton puro.

Para instalarlo guardar get-pip-py y luego en cmd meterse en el directorio y ejecutar python get-pip.py para comprobar que se ha instalado bien ejecutar pip --version donde nos aparecerá la versión de pip y la de Python con la que trabajamos.