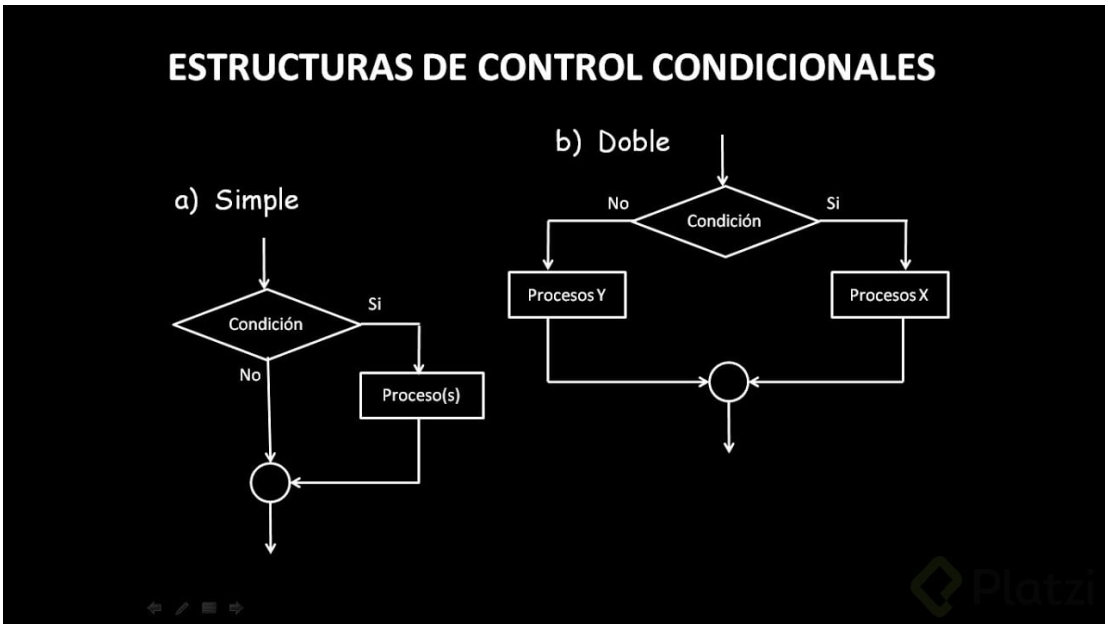
Checkpoint nº 5: teoría

1. ¿Qué es un condicional?

Un **condicional** en Python (y en general, en cualquier lenguaje de programación, ya que es una estructura universal en todos ellos) es una expresión sintáctica que permite ejecutar porciones (bloques) de código de Python dependiendo del cumplimiento de una o varias **condiciones**. Este tipo de estructuras es fundamental para el desarrollo de aplicaciones, ya que nos permiten controlar el flujo del programa: le damos 'inteligencia' a una aplicación, al hacer que tome diferentes caminos dependiendo de las condiciones. Esto hace que la aplicación que hayamos desarrollado se comporte diferente dependiendo de la entrada que le suministremos. Podemos decir que este tipo de estructuras son el primer paso de **inteligencia artificial** que se dió en programación. Representándolo con un diagrama de flujo:



La sintaxis de las estructuras condicionales existentes en Python son 3

Sintaxis 1

```
if condición :
    bloque a ejecutar si 'condición' es True
```

Esta es la estructura más sencilla de condicional: en ella, si *condición* se cumple (es decir es verdadera) se ejecuta el bloque de código. Si no se cumple, el flujo del programa salta a la siguiente línea después del bloque condicional.

Ejemplo

Jon es el usuario de tipo 'Superusuario' en una aplicación. Cuando un usuario introduce su nombre , este código comprueba que es 'Jon': en caso de que sea él cambia el valor de la variabla 'autorizado' a True y muestra un mensaje de saludo. Si no es 'Jon', salta a la siguiente línea después del bloque condicional.

```

```

```
name = 'Jon'
autorizado = False
if name == 'Jon' :
    autorizado = True
    saludo = f'Hola, {name}, puedes acceder como Superusuario'
    print(saludo)

# Salida: Hola, Jon, puedes acceder como Superusuario'
```

Sintaxis 2

```
    if condición :
        bloque1 (a ejecutar si 'condición' es True)
    else :
        bloque2 (a ejecutar si 'condición' es False)
```

En esta sintaxis, si se cumple la *condición*, se ejecuta el *bloque1* de código. Si no se cumple dicha condición, se ejecuta el *bloque2* de código.

Ejemplo
Jon es el usuario de tipo 'Superusuario' en una aplicación. Cuando un usuario introduce su nombre, este código comprueba que es 'Jon': en caso de que sea él, cambia el valor de la variabla 'autorizado' a True y muestra un mensaje de saludo. Si no es 'Jon', muestra un mensaje en pantalla señalando que no está autorizado.

```
autorizado = False
name = 'Jone'
if name == 'Jon' :
    autorizado = True
    saludo = f'Hola, {name}, puedes acceder como Superusuario'
    print(saludo)
else :
    print('Hola, no estás autorizado para acceder')

# Salida: Hola, no estás autorizado para acceder
```

Sintaxis 3

```
    if condición1 :
        bloque a ejecutar si 'condición1'es True
    elif condición2 :
        bloque a ejecutar si 'condición2'es True
    elif condición3 :
        bloque a ejecutar si 'condición3'es True
    .
```

```
    .
    .

    elif condición_N :
        bloque a ejecutar si 'condición_N'es True
    else :
        bloque a ejecutar si 'condición1', 'condición2', ... 'condición_N' son
False
```

En este caso, podemos usar tantos bloques **elif** como queramos, cada uno con su condición: esto nos permite definir tantas condiciones y bloques de código como queramos. Esto es útil en el caso de tener múltiples estados o situaciones en las que debamos aplicar diferentes bloques de código dependiendo de esas condiciones.

Ejemplo
Jon, Nerea y Txomin son los usuarios de tipo 'Superusuario' en una aplicación. Cuando un usuario introduce su nombre, este código comprueba que es alguno de los superusuarios: en caso de que lo sea, cambia el valor de la variable 'autorizado' a True y muestra un mensaje de saludo. Si no es ninguno de estos, muestra un mensaje en pantalla señalando que no está autorizado.

```
autorizado = False
name = 'Nerea'
if name == 'Jon' :
    autorizado = True
    saludo = f'Hola, {name}, puedes acceder como Superusuario'
    print(saludo)
elif name == 'Nerea' :
    autorizado = True
    saludo = f'Hola, {name}, puedes acceder como Superusuario'
    print(saludo)
elif name == 'Txomin' :
    autorizado = True
    saludo = f'Hola, {name}, puedes acceder como Superusuario'
    print(saludo)
else :
    print('Hola, no estás autorizado para acceder')

# Salida: Hola, Nerea, puedes acceder como Superusuario
```

En los 3 tipos de condicionales mencionados, si alguna de las condiciones es verdadera, no se evalúan el resto de las condiciones posteriores que pueda haber, saltando el flujo del programa a la primera línea después del bloque condicional: esto hace que la ejecución del programa sea más ágil que si tuviéramos que evaluarlas todas, ya que al encontrar la primera que es 'True', se salta el resto de las líneas del bloque condicional. En nuestro último ejemplo, si el usuario fuera *Jon*, el flujo del pograma saltaría a la siguiente línea después de este bloque, no evaluándose los **elif** ni el **else**.

Resumiendo, las estructuras condicionales son estructuras fundamentales en Python (y en cualquier lenguaje de programación) ya que nos permiten ejecutar diferentes bloques de código dependiendo de que se den

unas determinadas condiciones. Además, tenemos 3 tipos de estructuras condicionales en Python, lo que nos da mucha flexiblidad a la hora de elegir la más adecuada a la situación que queramos resolver.

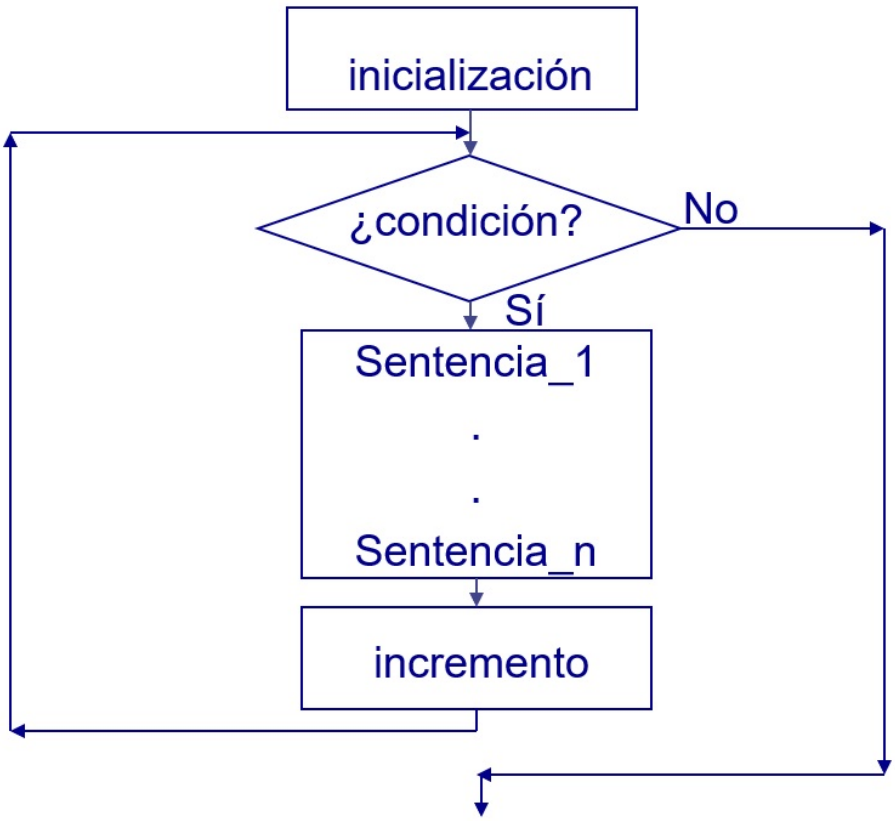
---

2. ¿Cuáles son los diferentes tipos de bucles en Python? ¿Por qué son útiles?
Los bucles son estructuras de control fundamentales en programación, que nos permiten repetir la ejecución de un bloque de código en diferentes situaciones (iteración).
Son muy útiles para:

- Automatización: ejecutan una tarea repetidamente sin tener que escribir el mismo código varias veces.
- Procesamiento de datos: permiten iterar sobre colecciones de datos (listas, tuplas, diccionarios, etc.) para realizar operaciones en cada elemento, realizando complejas operaciones con pocas líneas de código.
- Creación de programas dinámicos: hacen que el programa se adapte a diferentes entradas y situaciones, repitiendo acciones según sea necesario.

Existen 2 tipos de bucles en Python

1. **Bucle 'for'**
   El bucle **for** se usa para recorrer iterables (tipos de python compuestos de varios elementos) y ejecutar un bloque de código; iterables son los strings, las listas, las tuplas, los diccionarios, los rangos y los sets.
   El bucle **for** nos permite coger todos y cada uno de los elementos del iterable y aplicar un mismo bloque de código a cada uno de ellos.
   Representándolo con un diagrama de flujo:



Sintaxis

```
for variable in iterable :
   bloque de código a ejecutar
```

- variable: una variable que toma el valor de cada elemento del iterable en cada iteración del bucle.
- iterable: una lista, tupla, string, rango, o set (u otro objeto iterable).
- bloque: el código dentro del bloque for (indentado) se ejecuta una vez para cada elemento de la secuencia.

Ejemplo 1
Tenemos un diccionario, obtenido después de procesar una petición a una base de datos, con los datos de un usuario. Queremos mostrarlos por pantalla.

```
datos_usuario = {'Nombre': 'Inaki', 'Apellido': 'Mendigutxia', 'Edad': 39,
'Dirección': 'Plaza Basetxea 1 1º B', 'Municipio': 'Errenteria'}
n = 0
print('Datos del usuario\n')
for datos, valores in datos_usuario.items() :
   n += 1
   print(f'{n}. {datos}:  {valores}')


# Salida:

# Datos del usuario
# 1. Nombre:  Inaki
# 2. Apellido:  Mendigutxia
# 3. Edad:  39
# 4. Dirección:  Plaza Basetxea 1 1º B
# 5. Municipio:  Errenteria
```

En este caso, el iterable es el diccionario *datos_usuario*, compuesto por pares clave-valor. Las variables *datos*, *valores* van tomando consecutivamente los diferentes valores de cada par de valores de los elementos del diccionario, y se usan en la ejecución del bloque de código (la impresión en pantalla de los mismos). El blucle *for* recorre todos los elementos del diccionario, lo que nos permite de una forma sencilla (con apenas 2 líneas de código) mostrar toda la información. Nos daría lo mismo si el diccionario fuese de 500 pares key-value, en vez de 5, el programa se ejecutaría correctamente sobre esos 500 elementos.

Ejemplo 2

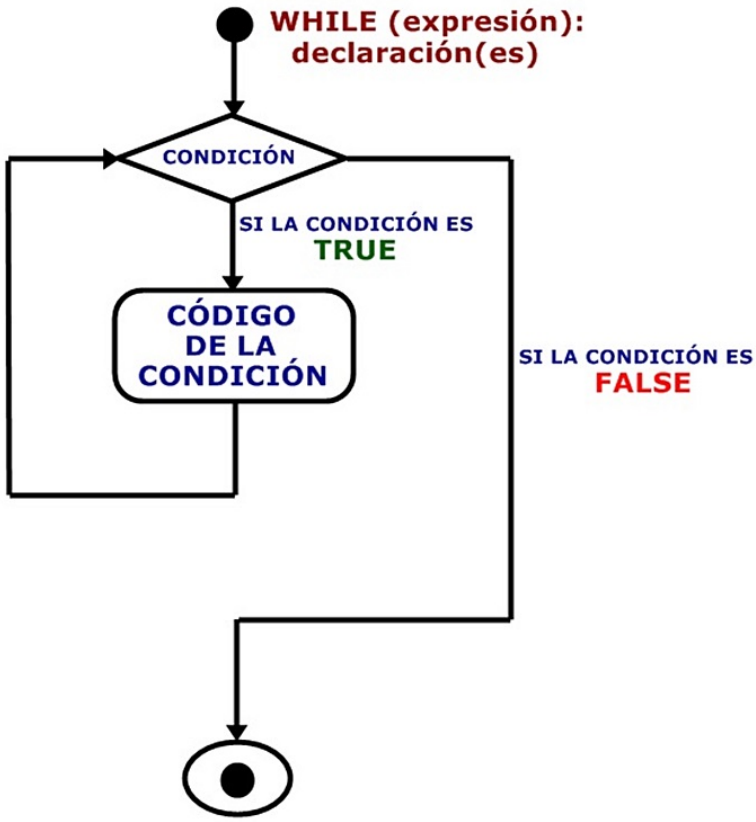
```
palabra = "Devcamp"
for letra in palabra:
    print(letra)
# Salida:

# D
# e
# v
```

```
# c
# a
# m
# p
```

En este caso el iterable es la palabra *Devcamp* y la variable es *letra*. El bucle **for** recorre la palabra letra por letra y para cada una de ellas realiza un salida por pantalla.

2. **Bucle 'while'**
   El bucle while nos permite ejecutar un bloque de código repetidamente mientras sea **verdadera** una cierta condición. Representándolo con un diagrama de flujo:



Sintaxis

```
while condición :
   bloque de código a ejecutar
```

- condición: una condición que si es True hace que se ejecute el bloque de código, y si es False finaliza el bucle y salta a la siguiente línea después del bucle.
- bloque de código: las líneas que se ejecutan en el caso de que *condición* sea True.

Una cuestión crítica en este tipo de bucles es que la *condición* se vuelva False en algún momento, porque de otra forma el bucle se vuelve infinito, lo que lleva al bloqueo de la aplicación.

Ejemplo 1
Tenemos una lista y queremos mostrar en pantalla solo los 5 primeros elementos de la lista

```
n = 0
frutas = ['naranja', 'pera',' manzana', 'mandarina', 'melocotón', 'aguacate']
while n < 5 :
   print(frutas[n])
   n += 1


# Salida:

# naranja
# pera
# manzana
# mandarina
# melocotón
```

La condición en este bucle es que la variable *n* sea menor que 5: en tal caso, se muestra en pantalla el elemento de la tupla *frutas* que se encuentra en esa posición. De esta manera, el último elemento de la tupla (aguacate), que es el nº 5, no se imprime. Aquí nos aseguramos que el bucle finalice alguna vez ya que la variable *n*, partiendo de 0, aumenta en una unidad en cada iteración.

Ejemplo 2
Hemos definido un bucle que pregunta un nombre de usuario, y hasta que no se introduce le nombre clave, continúa preguntan indefinidamente

```
nombre_clave = 'Pedro'
nombre_usuario = ''
while  nombre_usuario != nombre_clave :
   print('Teclea nombre de usuario:')
   nombre_usuario =input()
```

En este caso, la condición para que se ejecute el bloque de código es que el nombre que teclea el usuario NO sea *Pedro*. Mientras no acierte con ese nombre, el sistema seguirá pidiéndole que teclee uno. Este podría ser el caso de un bucle infinito: para que este bucle deje de ejecutarse y el programa no se quede de forma infinita en este punto, el usuario debe acertar con el nombre (Pedro)

3. Control de bucles

Existen 2 sentencias que nos permiten modificar el comportamiento de los bucles **for** y **while** en Python

- **break**: termina la ejecución del bucle inmediatamente.
- **continue**: salta la iteración actual del bucle y pasa a la siguiente iteración.

Ejemplos

break

```
frutas = ['manzana', 'banana', 'cereza', 'aguacate']
for fruta in frutas:
    if fruta == 'cereza' :
        break
    print(fruta)

# Salida:

# manzana
# banana
```

Cuando la variable *fruta* toma el valor *cereza*, se termina la ejecución del bucle y saltamos a la siguiente línea después del bucle.

continue

```
frutas = ['manzana', 'banana', 'cereza', 'aguacate']
for fruta in frutas:
    if fruta == 'cereza' :
        continue
    print(fruta)

# Salida:

# manzana
# banana
# aguacate
```

Cuando la variable *fruta* toma el valor *cereza*, se salta a la siguiente iteración: por tanto 'cereza' no aparece por pantalla.

4. Bucles anidados

Hemos visto las dos estructuras de bucle, pero esto se puede hacer más complejo usando bucles anidados, donde insertamos un bucle dentro de otro bucle. Aunque esto es posible hacerlo, se recomienda no abusar de este tipo de anidadaciones, ni de usar muchos niveles de anidación (bucle dentro de bucle dentro de bucle dentro de bucle...) ya que hacen los programas más difíciles de entender, y por tanto de modificar y depurar.

```
for n in range(4) :
    for m in range(2 ):
        print(f'{n} {m}')

# Output:

# 0 0
# 0 1
# 1 0
# 1 1
```

```
# 2 0
# 2 1
# 3 0
# 3 1
```

En resumen, los bucles **for** y **while** son herramientas esenciales en Python para ejecutar bloques de código de forma reiterada. Una diferencia muy importante entre el bucle **for** y el **while** es que el bucle **for** se ejecuta sobre **todos** los elementos de un iterable (salvo que usemos la setencia **break**), mientras que el bucle **while** se ejecuta dependiendo de una condición.

---

3. ¿Qué es una lista por comprensión en Python?
Una lista por comprensión en Python es una manera ágil y eficiente de crear nuevas listas a partir de iterables (una lista, tupla, string, rango, o set, u otro objeto iterable). Proporciona una sintaxis más compacta y legible que los bucles **for** tradicionales, para crear listas basadas en transformaciones o filtros aplicados a los elementos de un iterable. Se usan cuando la trasformación o el filtro a aplicar se pueden expresar en una línea, es decir, no son excesivamente complejos. Si la trasformación a realizar no podemos expresarla en una sola línea, es mejor usar un bucle **for** para crear la nueva lista.
Sintaxis

```
[expresión for elemento in iterable if condición]
```

- expresión: es una expresión que define la trasformación que queremos realizar para cada *elemento* del *iterable*.
- elemento: representa el *elemento* del *iterable* que estamos tratando.
- iterable: una lista, tupla, string, rango, o set (u otro objeto iterable).
- condición: es opcional y representa la *condición* que debe cumplir el *elemento* para que se le aplique la *expresión*

Ejemplo 1
Queremos crear una una nueva lista que suma 5 a cada elemento de una lista original

```
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
nueva_lista = [elemento + 5 for elemento in lista]
print(cuadrados)   # Salida: [6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

- expresión -> elemento + 5
- elemento -> cada uno de los elementos de la lista original
- iterable -> la *lista* [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
- condición -> no aplicada en este caso

Ejemplo 2
Creamos una una nueva lista con los elementos de la original en mayúsculas, solo si el elemento tiene más de 7 caracteres

```
lista = ['naranja', 'pera',' manzana', 'mandarina', 'melocotón', 'aguacate']
nueva_lista = [elemento.upper() for elemento in lista if len(elemento) > 7]
print(nueva_lista)   # Salida: [' MANZANA', 'MANDARINA', 'MELOCOTÓN', 'AGUACATE']
```

- expresión -> elemento.upper() convierte a mayúsculas un string
- elemento -> cada uno de los elementos de la lista original
- iterable -> la *lista* ['naranja', 'pera',' manzana', 'mandarina', 'melocotón', 'aguacate']
- condición -> len(elemento) > 7 (sólo si el nº de caracteres de la palabra es mayor de 7)

En resumen, esta estructura nos permite crear listas a partir de otras listas, usando una sola línea de código, de forma muy compacta y eficiente.

---

4. ¿Qué es un argumento en Python?

Un argumento en Python es un concepto que se usa relacionado con los métodos y las funciones. **Funciones** son bloques de código que encapsulamos y a las que les damos un nombre. De esta manera, podemos llamar a estos bloques para ejecutarlos en diferentes puntos de nuestra aplicación, sin tener que repetir ese código: a eso se le llama reutilización del código y nos permite construir aplicaciones más compactas y más fáciles de entender. Estas funciones pueden tener (de hecho, la mayor parte de las veces los tienen) **argumentos**, que nos permiten pasar a estos bloques encapsulados variables para que el bloque de código que contienen sea ejecutado con esos valores. Veámoslo con un ejemplo: vamos a definir una función sencilla que sume dos valores.

```
def funcion_suma(primer_sumando, segundo_sumando) :
    suma = primer_sumando + segundo_sumando
    return suma

print(funcion_suma(45, 15))   # Salida: 60
```

En este caso, los **argumentos** de 'funcion_suma' son **primer_sumando** y **segundo_sumando** y se encierran entre paréntesis. Estos argumentos nos permiten pasar a la función cualquier valor que deseemos. De esta manera, podemos usar esta función en cualquier parte de nuestra aplicación y con cualquier par de valores. Esto nos da una enorme flexibilidad.

Argumentos por posición o por nombre

Cuando pasamos los argumentos a una función podemos hacerlo empleando dos métodos distintos.

- **Por posición**. En este caso, al pasar los argumentos debemos asegurarnos que la posición en la que los pasamos a la función es la misma que en la definición. Veámoslo con un ejemplo.

```
def datos_usuario(nombre, apellido) :
    print(f'El nombre del usuario es {nombre} {apellido}')

print(datos_usuario('Iñaki','Gómez'))  #Salida: 'El nombre del usuario es Iñaki
Gómez' Uso correcto, hemos puesto cada elemento en su lugar
```

```
print(datos_usuario('Gómez','Iñaki'))  #Salida: 'El nombre del usuario es Gómez Iñaki' Uso incorrecto, no hemos puesto cada elemento en su lugar
```

- **Por nombre**. En este caso, al pasar los argumentos ponemos explícitamente qué valor asignamos a cada argumento, indicándolo con el nombre con el que se ha definido la función. En este caso, el orden de los argumentos es irrelevante, y podemos ponerlos como queramos. Veámoslo con un ejemplo.

```
def datos_usuario(nombre, apellido) :
    print(f'El nombre del usuario es {nombre} {apellido}')

print(datos_usuario(nombre = 'Iñaki', apellido = 'Gómez'))  #Salida: 'El nombre del usuario es Iñaki Gómez' Uso correcto.
print(datos_usuario(apellido = 'Gómez', nombre = 'Iñaki'))  #Salida: 'El nombre del usuario es Iñaki Gómez' Uso correcto.
```

Argumentos por defecto

Como hemos mencionado, al crear una función definimos los argumentos (si es que los tiene) necesarios para ejecutar esa función. Y al utilizarla debemos dar valores a esos argumentos para que todo funcione correctamente. Pero puede darse el caso en no tengamos un valor para uno o varios de esos argumentos y en tal caso, al llamar a la función se generaría un error. Aquí es donde entran en juego los valores por defecto. Los valores por defecto son valores que damos a los argumentos de una función al definirla, y son esos los valores que tomará la función al llamarla, si no le pasamos esos valores explícitamente. Veámoslo con un ejemplo.

Creamos una función y asignamos valores por defecto a ambos argumentos, mediante la asignación =: en el caso de 'nombre' es 'Jhon' y en el caso de 'apellido' es 'Doe'

```
def datos_usuario(nombre = 'Jhon', apellido = 'Doe') :
    print(f'El nombre del usuario es {nombre} {apellido}')

print(datos_usuario())  #Salida: 'El nombre del usuario es Jhon Doe'
print(datos_usuario(apellido = 'Gómez'))  #Salida: 'El nombre del usuario es Jhon Gómez. Ya que no hemos asignado un valor a 'nombre', usa 'Jhon' que es el valor por defecto
print(datos_usuario(,'Gómez'))  #Salida: Se genera un error sintáctico
```

Es importante señalar que si no vamos a pasar el primer valor (dejamos el de defecto) debemos pasar los argumentos por nombre.

Desempaquetado de argumentos

El desempaquetado de argumentos se usa para definir funciones en las que no sabemos a priori cuántos argumentos vamos a recibir y por tanto, no podemos explicitarlos en el momento de su definición. El desempaquetado de argumentos es una técnica poderosa en Python que permite expandir secuencias y diccionarios al pasar argumentos a funciones.

1. **Desempaquetado posicional** El desempaquetado de argumentos posicionales permite pasar los elementos de un iterable (lista, tupla, etc.) como argumentos posicionales a una función.

Sintaxis

```
def funcion(*args) :
    bloque a ejecutar
```

- args: es una expresión que representa los argumentos que desconocemos a priori. El '*' es necesario porque es quien indica que son una serie de argumentos que se deben desempaquetar.
- bloque a ejecutar : el bloque de código que se ejecutará al invocar la función.

Ejemplo

```
def suma_totales(*args) :
    total = 0
    for argumento in args :
        total += argumento
    return total

print(suma_totales(5,24,67,23))  # Salida: 119
```

En este ejemplo, pasamos a la función varios valores *integer* y la función se encarga mediante un bucle **for** sumarlos todos. No hemos definido en la función cuántos sumandos le vamos a pasar, por lo que podemos pasarle 5, 20 ó 1.000, y ella de encargará de desempaquetarlos y realizar la operación correctamente. Esto nos da unas posibilidades inmensas a la hora de definir funciones enormemente flexibles, en las que no sabemos a priori cuántos argumentos vamos a pasar.

2. **Desempaquetado por clave**
   El desempaquetado de argumentos de palabra clave permite pasar los pares clave-valor como argumentos de palabra clave a una función.

Sintaxis

```
def funcion(**kwargs) :
    bloque a ejecutar
```

- kwargs: es una expresión que representa los argumentos que desconocemos a priori. Los argumentos son pares que serán trasformados a pares clave-valor. El '**' es necesario porque es quien indica que son una serie de argumentos que deberemos desmpaquetar por clave.
- bloque a ejecutar : el bloque de código que se ejecutará al invocar la función.

Ejemplo

```
def listado(**kwargs) :
    for argumento in kwargs.values() :
        print(argumento)
```

```
listado(nombre = 'Iñaki', apellido = 'Mendigutxia')

# Salida:

# Iñaki
# Mendigutxia
```

3. **Desempaquetado por posición y clave**
   También podemos definir funciones en las que empleemos ambos tipos de desempaquetado, por posición y por clave. Esto nos da aún más flexibilidad a la hora de definir funciones ya que podemos sumar las ventajas de ambas opciones simultáneamente en una misma función.

Sintaxis

```
def funcion(*args,**kwargs) :
    bloque a ejecutar
```

- args: es una expresión que representa los argumentos que desconocemos a priori. El '*' es necesario porque es quien indica que son una serie de argumentos que deberemos desempaquetar.
- kwargs: es una expresión que representa los argumentos que desconocemos a priori. Los argumentos son pares que serán trasformados a pares clave-valor. El '**' es necesario porque es quien indica que son una serie de argumentos que deberemos desmpaquetar por clave.
- bloque a ejecutar : el bloque de código que se ejecutará al invocar la función.

Ejemplo

```
def listado(*args, **kwargs) :
    print(f"{' '.join(args)}, ")
    for argumento in kwargs.values() :
        print(argumento)

listado('Buenos', 'días', nombre = 'Iñaki', apellido = 'Mendigutxia')

# Salida:

# Buenos días,
# Iñaki
# Mendigutxia
```

En resumen, los **argumentos** en Python (y en todos los lenguajes funcionales) son parte esencial de las **funciones** que nos permiten pasar valores diferentes que dichas funciones que ejecutarán su código interno, generando una salida específica para esos valores de entrada. Esto nos permite una enorme flexibilidad, reduce el código de una aplicación ya que reutilizamos mlíneas en diferentes situaciones y hace que las aplicaciones sean mucha más fáciles de entender y de corregir errores. Tal co mo hemos visto, Python tiene diferentes opciones que se adaptan a multitud de situaciones, lo que hace que sea una herramienta de programación tan poderosa.

5. ¿Qué es una función Lambda en Python?
Una función lambda es una función especial que se define de forma sencilla en una sola línea y es anónima, aunque pueda ser asignada a una variable. Nos permite definir funciones sencillas y muy compactas.

Características principales

- Anónimas: no tienen un nombre asignado (aunque se pueden asignar a una variable).
- De una sola línea: solo pueden contener una única expresión.
- Pueden tener cualquier número de argumentos: al igual que las funciones regulares, las funciones lambda pueden aceptar cualquier número de argumentos.

Sintaxis

```
lambda argumentos : expresión
```

- lambda: palabra clave que indica la creación de una función lambda.
- argumentos: una lista de argumentos separados por comas (puede estar vacía).
- expresión: una única expresión que se evalúa y se retorna como resultado de la función.

Ejemplo
Vamos a crear una función lambda que sume 2 sumandos

```
suma = lambda sumando1, sumando2 : sumando1 + sumando2
print(suma(45, 15))   # Salida: 60
```

- argumentos: sumando1, sumando2
- expresión: sumando1 + sumando2

Como vemos, es la misma función que definimos en un punto anteriror, pero definida con un formato en una sola línea, mucho más compacto. Se recomiendan cuando la expresión es una operación sencilla que se puede expresar en una sola línea y al ser anónima no necesite usarse en otras partes de la aplicación. En nuestro ejemplo la hemos asignado a una variable y podríamos usarla en otra parte, pero no tiene por qué ser así.
Si el bloque de código de la función consta de varias líneas, o lo vamos a usar en otras partes del código, es mejor usar una función regular.

---

6. ¿Qué es un paquete pip?

Los paquetes pip son librerias desarrolladas por programadores que añaden funcionalidades a la base de Python. Cuando instalamos Python en una máquina viene con un conjunto básico de componentes (tipos de objetos, funciones, operadores, métodos, clases,...) que son los llamados objetos built-in. Pero con el tiempo, los desarrolladores de Python han ido desarrollando bibliotecas que aumentan enormemente las funcionalidades de Python. **PIP** es una aplicación de consola que nos permite gestionar los paquetes que necesitemos instalar en nuestra máquina para su uso. De esta manera, con el comando **pip** podemos instalar en nuestra máquina el paquete que necesitemos y así podemos usar sus funciones, métodos y clases sin tener

que crearlos desde cero. Existen infinitos paquetes desarrollados por la comunidad Python para múltiples áreas: machine learning, ciencia de datos, representaciones gráficas, deep learning, tratamieto de textos, reconocimieto de caracteres,... Esta enorme aportación de la comunidad Python hace que este lenguaje sea uno de los más potentes y desarrollados.

Sería una locura instalar todos los disponibles al existir cientos (si no miles de ellos) que ocuparían todo el disco duro de nuestro ordenador. El repositorio centralizado de paquetes Python se encuentra en la URL https://pypi.org/, donde podemos buscar el que se ajuste a nuestras necesidades. Por ese motivo cada programador instala y activa los que necesita para su trabajo, usando el comando **pip**.

Al instalar Python, si todo se instala correctamente, se debiera instalar la aplicación **pip** para la gestión de paquetes.

Para instalar un paquete, éste se instalaría localmente en nuestra máquina. Debemos abrir una consola de Windows (mediante PowerShell o command) y ejecutar el comando:

```
pip install nombre_del _paquete
```

Otra manera de usar **pip** es:

```
python -m pip nombre_del_paquete
```

Por ejemplo, para instalar SQLAlchemy, que es un paquete para gestionar bases de datos SQL, usaríamos el siguiente comando en la consola de Windows:

```
pip install sqlalchemy
```

Los commandos más usuales de pip son los siguientes:

Instalar un paquete

```
pip install package_name
```

Instalar una versión específica

```
pip install package_name==1.0.0
```

Actualizar un paquete

```
pip install --upgrade package_name
```

Desinstalar un paquete

```
pip uninstall package_name
```

Listar los paquetes instalados

```
pip list
```

Mostrar los detalles de un paquete

```
pip show package_name
```

Para facilitar la tarea de migrar las aplicaciones de Python, podemos guardar en un archivo de texto todos los paquetes usados en un desarrollo en particular. De esta manera si migramos la aplicación a otra máquina podemos instalar los paquetes necesarios sin tener que instalarlos de uno en uno, con el ahorro de tiempo que ello supone, sobre todo si el proyecto usa muchos paquetes o librerías distintas. Podemos incluir comentarios (usando #) y estructurarlo en bloques para hacer más fácil su comprensión a los humanos. El archivo que generamos lo llamaremos *requirements.txt* por convenio.

Ejemplo de archivo requirements.txt (debe ser formato de texto plano)

```
# Paquetes principales
requests==2.31.0  # HTTP para humanos
flask>=2.0.0     # Microframework web
django>=3.2,<4.0  # Framework web full-stack

# Ciencia de datos
numpy
pandas~=1.5.0

# Desarrollo
black            # Formateador de código
pytest==7.4.0    # Testing

# Desde un repositorio Git
git+https://github.com/user/repo.git@branch#egg=nombre_paquete
```

Una vez creado para instalar esas librerías en la nueva máquina debemos usar el comando:

```
pip install -r requirements.txt
```

Si queremos generar de forma automática el archivo requirements.txt, debemos ejecutar el siguiente comando en la máqueina en la que tenemos desarrollada la aplicación:

```
pip freeze > requirements.txt
```

Actualizar el mismo gestor de paqeutes **pip**

```
pip install --upgrade pip
```