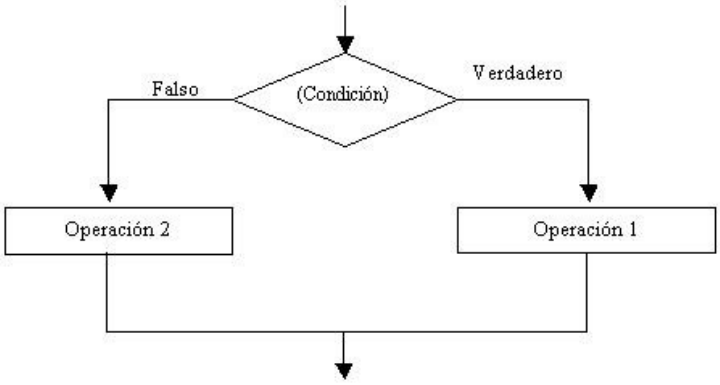
¿Qué es un condicional?

Son las declaraciones que permiten la toma de decisiones dinámicamente en función de la evaluación de expresiones lógicas. Dicho de otra forma, las condicionales en programación definen bifurcaciones en el código, donde si se cumple una condición, se ejecuta un conjunto de instrucciones y si no se cumple, se ejecuta otro conjunto de instrucciones



Sintaxis básica de las condiciones

En la mayoría de los lenguajes de programación, la sintaxis básica de una declaración condicional sigue la estructura «si-entonces». Se representa en la mayoría de lenguajes con la declaración «if» (si) ejemplo en Python:

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

Aquí, el programa evalúa la condición y ejecuta el bloque de código dentro del «if» si la condición es verdadera (True); de lo contrario, ejecuta el bloque dentro del «else», que representa False. Podemos controlar el flujo del programa de manera sencilla.

Para realizar condicionales, la mayoría de veces, involucran operadores de comparación, como igualdad (==), desigualdad (!=), mayor que (>), menor que (<) y otros. Estos operadores permiten evaluar las expresiones y tomar las decisiones en función de los resultados. Además, las condiciones pueden ser compuestas, para ello se utilizan operadores lógicos como «and» y «or», permiten construir lógica más compleja.

```
edad = 18
if edad >= 18 and edad < 21:
   print("Eres mayor de edad, pero aún no puedes beber alcohol en algunos lugares.")
elif edad >= 21:
   print("¡Bienvenido a la edad adulta! Puedes disfrutar de todas las bebidas.")
else:
   print("Eres menor de edad. No puedes acceder a ciertos privilegios.")
```

En este ejemplo, la condición compuesta evalúa la edad y determina el mensaje que se va a imprimir en función de las restricciones de la edad.

En el caso de Python tenemos el operador ternario. Este operador permite asignar un valor a una variable en función de una condición, lo que a menudo conduce a un código más limpio y legible. Esto solo se usa para casos sencillos que no necesiten un conjunto de instrucciones para continuar con la ejecución del programa.

```
evaluacion = "Aprobado" if puntos >= 60 else "Suspendido"
```

En este caso, la variable «evaluacion» se establece en «Aprobado» si la puntuación es igual o mayor a 60, de lo contrario, se establece en «Suspendido».

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

Los bucles se utilizan para repetir varias veces la ejecución de una parte de un programa. Hay dos tipos de bucles:

- Los bucles **while**: Cuando no se conoce de antemano el número de repeticiones del programa
- Los bucles **for**: Cuando se sabe de antemano cuántas veces se va a repetir el programa

Bucles **while:**

La palabra clave **while** significa **mientras**. Por lo que como su nombre indica, el bloque de instrucciones que está dentro de un bucle **while** se ejecutará mientras se cumplan las condiciones. Esto será un booleano con los siguientes valores:

- **True**: se ejecuta el bloque de instrucciones que hay dentro del bucle.
- **False**: no se ejecuta el bloque de instrucciones y se finaliza el bucle continuando con la ejecución del programa.

```
While condición:
    bloque instrucciones
```

Por lo general, la condición que se aplica en el bucle depende de un objeto, una variable definida previamente y que se actualizará al pasar el bucle. Pero si no cambia nada inherente a la condición, el bucle se ejecutará sin fin. Tomemos un ejemplo de bucle **«infinito»**

```
i = 0
while i < 10 :
  print(i)
```

El valor del parámetro **i** no cambia una vez que está en el bucle. De este modo, cuando volvamos a la condición impuesta, se comprobará una y otra vez que se cumple. De ahí la importancia de variar la variable **i** como en el ejemplo siguiente:

```
i = 0
while i < 10 :
   print(i**2)
   i = i + 1
print('fin del programa')
```

Este código devuelve la secuencia de números de 0 a 10 al cuadrado. En este caso la variable que interviene en la condición se actualiza al pasar por el bucle. Aquí, la variable i se fija inicialmente en 0 y se incrementa en 1 cada vez que pasa por el bucle. Este incremento nos permite establecer un límite a la ejecución del bucle.

Bucles f**or**:

El bucle for en Python se emplea para poder recorrer los elementos que componen un objeto iterable (listas, tuplas, conjuntos o diccionarios, por ejemplo) para ejecutar repetidamente un bloque de código determinado. Durante cada paso de esta iteración se tiene en cuenta un solo elemento del objeto iterable y, sobre él, se aplican una serie o **conjunto de operaciones.**

A nivel de sintaxis, el bucle en Python se expresa de la siguiente manera. Desgranando esta expresión, nos encontramos con que: *elem* es la variable de la que parte el iterador durante cada paso del bucle.

```
 for<elem> in <iterable>:
   <bloque de código>
```

un ejemplo:

```
numeros = [1, 2, 3, 4, 5]
suma = 0

for numero in numeros:
    suma += numero

print("La suma de los números es:", suma)
```

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.

Los bucles son útiles en programación porque permiten ejecutar un bloque de código repetidamente. Por ejemplo, podemos usar un bucle "for" para procesar todos los elementos de una lista, o un bucle "while" para repetir una acción hasta que se cumpla una condición.

¿Qué es una lista por comprensión en Python?

Es una característica única de Python que permite crear y manipular listas de manera eficiente con menos código (en otros lenguajes necesitamos más líneas de código para hacer lo mismo), aplicando una expresión a cada elemento de una secuencia iterable. Es una forma de transformar una lista en otra lista, aplicando una expresión a cada uno de sus elementos, y filtrando los resultados.

La sintaxis básica de una comprensión de listas es:

```
nueva_lista = [exp for element in lista_original if condición]
```

La **exp** es el resultado que queremos obtener y se aplica a cada elemento de la lista. La parte **for elemento in lista_original** es similar a un bucle for normal. **element** es una variable temporal que toma cada valor de **lista_original** durante la iteración. La **condición** es opcional y se puede usar para filtrar los elementos que se incluirán en la nueva lista. Todo esto con ejemplos se entiende mejor.

De un rango del 1 a 30 obtendremos los valores que sean múltiplos de 3

```
Metodo tradicional:

multiples = []
for i in range(30):
  if i % 3 == 0]
     multiples.append(i)

print(multiples)
# Salida: [ 3, 6, 9, 12, 15, 18, 21, 24, 27]


Metodo por lista por comprensión:

multiples = [i for i in range(30) if i % 3 == 0]
print(multiples)
# Salida: [ 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

De una lista dada obtenemos el número elevado al cuadrado de esa lista.

```
Metodo tradicional:

lista1=[8, 5, 4, 10, 2]
lista2=[]
for elemento in lista1:
   lista2.append(elemento*elemento
lista1=[8, 5, 4, 10, 2]
```

```
lista2=[elemento*elemento for elemento in lista1]
print("Lista 1")
print(lista1)
# Salida: =[8, 5, 4, 10, 2]
print("Nueva lista")
```

Metodo por lista por comprensión:

```
lista1=[8, 5, 4, 10, 2]
lista2=[elemento*elemento for elemento in lista1]
print("Lista 1")
print(lista1)
# Salida: =[8, 5, 4, 10, 2]
print("Nueva lista")
print(lista2)
# Salida: [64,25,16,100,4]
```

¿Qué es un argumento en Python?

Es el valor que se recibe en la definición de una función y se denominan parámetros, pero durante la llamada a la función esos valores se envían, y se denominan argumentos.

En Python existen distintos tipos de argumentos que debemos de conocer ya que entenderlos nos ayudará muchísimo a la hora de programar y de consultar la documentación. En concreto debemos de distinguir entre:

- **Los argumentos posicionales:** son argumentos que se pueden llamar por su posición en la definición de la función.
- **Argumentos por nombre:** clave son argumentos que se pueden llamar por su nombre.
- **Los argumentos obligatorios**: son argumentos que se deben pasar a la función**.** Si no pasamos los argumentos provocará un error.
- **Los argumentos opcionales:** son argumentos que no es necesario especificar. En Python, los argumentos opcionales son argumentos que tienen un valor predeterminado.

La forma en que se pasa el valor a la función determina si son argumentos posicionales o argumentos de palabra clave

Argumentos por posición

```
def resta(a, b):
   return a - b


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

Argumentos por nombre

```
def resta(a, b):
   return a - b


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

Llamada sin argumentos

```
def resta(a, b):
   return a – b


resta()



---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-78c8f433960e> in <module>()
----> 1 resta()


TypeError: resta() missing 2 required positional arguments: 'a' and 'b'
```

Parametros por defecto

```
def resta(a=None, b=None):
   if a == None or b == None:
      print("Error, debes enviar dos números a la función")
      return   # indicamos el final de la función aunque no devuelva nada
   return a-b


resta()



Error, debes enviar dos números a la función
```

Parámetros indeterminados

En alguna ocasión no sabremos de antemano cuantos elementos vamos a enviar a una función. En estos casos podemos utilizar los **parámetros indeterminados**. Tenemos 2 tipos: por posición y por nombre.

Por posición

Para recibir un número indeterminado de parámetros por posición, debemos crear una lista dinámica de argumentos (una tupla) definiendo el parámetro con un **asterisco**:

```
def indeterminados_posicion(*args):
    for arg in args:
        print(arg)


indeterminados_posicion(5,"Hola",[1,2,3,4,5])


#salida por pantalla:
#5
#Hola
#[1, 2, 3, 4, 5]
```

Por nombre

Para recibir un número indeterminado de parámetros por nombre (clave-valor o en inglés ***keyword args***), debemos crear un diccionario dinámico de argumentos definiendo el parámetro con dos asteriscos:

```
def indeterminados_nombre(**kwargs):
    print(kwargs)


indeterminados_nombre(n=5, c="Hola", l=[1,2,3,4,5])

#salida por pantalla
#{'n': 5, 'c': 'Hola', 'l': [1, 2, 3, 4, 5]}
```

Si se recibe como un diccionario, podemos iterarlo y mostrar la clave y valor de cada argumento:

```
def indeterminados_nombre(**kwargs):
    for kwarg in kwargs:
        print(kwarg, "=>", kwargs[kwarg])


indeterminados_nombre(n=5, c="Hola", l=[1,2,3,4,5])

#salida por pantalla
#n => 5
#c => Hola
#l => [1, 2, 3, 4, 5]
```

Por posición y nombre

Si queremos aceptar ambos tipos de parámetros simultáneamente, entonces debemos crear ambas colecciones dinámicas. Primero los argumentos indeterminados por valor y luego los que son por clave y valor:

```
def super_funcion(*args,**kwargs):
    total = 0
    for arg in args:
        total += arg
    print("sumatorio => ", total)
    for kwarg in kwargs:
        print(kwarg, "=>", kwargs[kwarg])


super_funcion(10, 50, -1, 1.56, 10, 20, 300, nombre="Hector", edad=27)
#salida por pantalla
#sumatorio => 390.56
#nombre => Hector
#edad => 27
```

Los argumentos **args** y **kwargs** no son nombres obligatorios, pero se suelen utilizar por convención. En muchos frameworks y librerías se utilizan, por lo que es una buena práctica usar estos nombres.

¿Qué es una función Lambda en Python?

Las expresiones lambda se usan idealmente cuando necesitamos hacer algo simple y estamos más interesados en hacer el trabajo rápidamente en lugar de nombrar formalmente la función. Las expresiones lambda también se conocen como funciones anónimas.

Las expresiones lambda en Python son una forma corta de declarar funciones pequeñas y anónimas (no es necesario proporcionar un nombre para las funciones lambda).

Las funciones Lambda se comportan como funciones normales declaradas con la palabra clave **def.** Resultan útiles cuando se desea definir una función pequeña de forma concisa. Pueden contener solo una expresión, por lo que no son las más adecuadas para funciones con instrucciones de flujo de control.

Sintaxis de una función Lambda

```
lambda argumentos: expresión
```

Las funciones Lambda pueden tener cualquier número de argumentos, pero solo una expresión.

Código de ejemplo:

```
# Función Lambda para calcular el cuadrado de un número
square = lambda x: x ** 2
print(square(3)) # Resultado: 9


# Funcion tradicional para calcular el cuadrado de un numero
def square1(num):
  return num ** 2
print(square(5)) # Resultado: 25
```

En el ejemplo de lambda anterior, **lambda x: x ** 2** produce un objeto de función anónimo que se puede asociar con cualquier nombre. Entonces, asociamos el objeto de función con **square.** De ahora en adelante, podemos llamar al objeto **square** como cualquier función tradicional, por ejemplo, **square(10)** que nos devolvería un valor igual a 100.

Ejemplos de funciones lambda

Principiante

```
lambda_func = lambda x: x**2 # Funcion que recoge un número entero y devuelve su cuadrado
lambda_func(3) # Retorna 9
```

Intermedio

```
lambda_func = lambda x: True if x**2 >= 10 else False
lambda_func(3) # Retorna False
lambda_func(4) # Retorna True
```

Complejos

```
mi_dicc = {"A": 1, "B": 2, "C": 3}
sorted(mi_dicc, key=lambda x: my_dict[x]%3) # Retorna ['C', 'A', 'B']
```

Casos de uso

Supongamos que desea filtrar los números impares de una lista. Podrías usar un bucle for:

```
mi_lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
filtrado = []

for num in mi_lista:
    if num % 2 != 0:
        filtrado.append(num)

print(filtrado)     # Python 2: impresión de filtrado
# [1, 3, 5, 7, 9]
```

O podría escribir esto en una línea con comprensión de listas (list-comprehensions). La compresión de listas ofrece una sintaxis más corta cuando deseas crear una nueva lista basada en los valores de una lista existente:

```
filtrado = [x for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] if x % 2 != 0]
```

En Python 3, las funciones integradas devuelven objetos generadores, por lo que se debe llamar la función list. La función list creara un objecto de lista.

¿Qué es un paquete pip?

Pip es un sistema de gestión de paquetes utilizado para instalar y administrar paquetes de software escritos en Python. Muchos de los paquetes pueden ser encontrados en el Python Package Index (PyPI). Python 2.7.9 y posteriores (en la serie Python2) y Python 3.4 y posteriores incluyen pip (pip3 para Python3) por defecto. Pip es un acrónimo recursivo que se puede interpretar como **Pip Instalador de Paquetes** o **Pip Instalador de Python**.

Estos paquetes pueden ser instalados en cualquier sistema operativo que soporte Python, como Windows, macOS o Linux.

PIP proporciona una forma sencilla de instalar paquetes, permitiendo a los usuarios acceder y utilizar rápidamente el código escrito por otros desarrolladores. Una ventaja importante de **pip** es la facilidad de su interfaz a través de la línea de comandos, el cual permite instalar paquetes de software de Python fácilmente con una orden:

```
pip install nombre-paquete
```

Los usuarios también pueden fácilmente desinstalar algún paquete:

```
pip uninstall nombre-paquete
```

Para comprobar si tenemos **pip** instalado, ejecutaremos el siguiente comando:

```
pip --version
```

En caso de no estar, realizaremos los siguientes pasos:

- Descargue el script del instalador **get-pip.py**. Si estás en Python 3.2, necesitarás esta versión de **get-pip.py**. En caso de tener Python 3.3 o 3.4 usar estas versiones de PiP correspondientemente Python 3.3 get-pip.py o Python 3.4 get-pip.py. De cualquier manera, clic botón derecho en el enlace y seleccionamos **Guardar como** y lo guardamos en cualquier carpeta del pc, como por ejemplo Descargas.
- Abrimos el símbolo del sistema con privilegios de administrador y navegamos hasta el archivo get-pip.py.
- Ejecutamos el siguiente comando: **python get-pip.py**

Para instala paquetes escribiremos el siguiente comando:

```
pip install nombre_del_paquete
```

Si queremos instalar por ejemplo el repositorio Numpy ejecutaremos el siguiente comando:

```
pip install numpy
```

Instalará la librería numpy para que podamos hacer uso de ella en nuestro programa.

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.