¿Qué es un condicional?

La declaración condicional **if-elif-else** en Python es una forma de escribir código de programa para que produzca un resultado dependiendo de si una determinada condición es verdadera o no.

Cuando tenga varias condiciones, puede usar **elif** (abreviatura de **else if**) para probarlas una a la vez. Si ninguna de las condiciones es verdadera, use el bloque **else** para ejecutar el código predeterminado.

La sintaxis se ve así:

```
if condición_1:
    # bloque de código que se ejecuta si la condición_1 es verdadera
elif condición_2:
    # bloque de código que se ejecuta si la condición_2 es verdadera
elif condición_3:
    # bloque de código que se ejecuta si la condición_3 es verdadera
else:
    # bloque de código que se ejecuta si ninguna de las condiciones es verdadera
```

Las condiciones se verifican en orden. Si alguno de ellos es verdadero, se ejecuta el bloque de código correspondiente y el resto se ignora. Si ninguna de las condiciones es verdadera, se ejecuta el bloque de código de la sección **else**, si está presente.

Imaginemos que el programa te ayuda a elegir ropa en función del clima:

```
weather = "soleado"

if weather == "lluvioso":
    print("Toma un paraguas")
elif weather == "soleado":
    print("Ponte las gafas de sol y gorra")
else:
    print("Puedes ponerte lo que quieras")
```

En este ejemplo:

- Si el clima es lluvioso, el programa te dará una recomendación para llevar paraguas.
- Si el clima es soleado, usa gafas de sol y gorra.
- Si no llueve ni hace sol, no hay recomendaciones específicas.

Declaración condicional if

En Python, una expresión se considera verdadera (**true**) si su resultado no es cero o la expresión no es un objeto vacío. En consecuencia, se considera falso (**false**) si el resultado es cero o un objeto vacío, incluido el valor **none**. Cuando se utilizan operadores de comparación, el resultado de la expresión es **true** si la condición es verdadera y **false** en caso contrario.

La declaración **if** en Python se usa para ejecutar un bloque de código si una condición determinada es verdadera.

Esquemáticamente, la construcción con **if** se puede escribir así:

```
if expresión_condicional:
    instrucciones
    instrucciones
    . . .
```

- **if** es una declaración que introduce una condición;

- dos puntos después de una expresión condicional indican que lo que sucederá a continuación es una acción o un conjunto de acciones secuenciales cuando se cumple la condición;

- instrucciones - acción con cada elemento.

Puede realizar múltiples comparaciones dentro de una condición, lo que amplía las posibilidades de utilizar construcciones condicionales. Esto se puede implementar usando los operadores lógicos **and, not** y **or**:

- **and** - significa "SÍ" para dos condiciones. Devuelve **true** si ambas condiciones son verdaderas y **false** si ambas condiciones son falsas;

- **or** - significa "O" para dos condiciones. Devuelve **true** si al menos una de las condiciones es verdadera y **false** en caso contrario;

- **not** - significa "NO" por una condición. Devuelve **true** si la condición es falsa y **false** si es verdadera.

Estas operaciones se utilizan para construir expresiones lógicas complejas.

Ejemplo:

```
x = 5
y = 10

result = (x > 0) and (y < 15)

print(result)  # Salida: True, ya que ambas condiciones (x > 0 e y < 15) son verdaderas
```

Declaración if-else

Una declaración **if-else** en Python es una construcción que permite ejecutar uno de dos bloques de código dependiendo de si una condición (la expresión después de **if**) se evalúa como verdadera o falsa. Si la condición es verdadera, se ejecuta el bloque de código bajo **if**. Si la condición es falsa — **else**.

Ejemplo:

```
x = 10

if x > 5:
    print("x > 5")
else:
    print("x <= 5")
```

Si el valor de la variable **x** es más que cinco, se ejecuta el bloque de código bajo **if** y se imprime “x > 5”. De lo contrario, se ejecuta el bloque de código de **else** y se imprime "x <= 5".

Tomemos como ejemplo el proceso de autenticación de usuarios.

```
username = input("Ingrese su nombre de usuario: ")
password = input("Ingrese su contraseña: ")
```

Supongamos que las credenciales de autenticación correctas son "username123" y "password123".

```
correct_username = "username123"
correct_password = "password123"

if username == correct_username and password == correct_password:
    print("Autenticación exitosa. Acceso permitido.")
else:
    print("Error de autenticación. Verifique su nombre de usuario y contraseña.")
```

Declaraciones anidadas if y if-else

Puede utilizar algunas declaraciones condicionales dentro de otras. Con tales construcciones, puede construir estructuras lógicas más complejas dependiendo de muchas condiciones. Estos operadores son útiles cuando es necesario probar varias condiciones con pasos lógicos adicionales dentro de cada una.

Una declaración if dentro de otra declaración if

```
x = 10
if x > 0:
    print("x - un número positivo")
    if x % 2 == 0:
        print("x - un número par")
    else:
        print("x - un numero impar")
else:
    print("x no es un número positivo")
```

En este ejemplo, si x es un número positivo, el programa comprueba si es par o no. Un bloque interno con una instrucción **if** anidada se ejecuta solo si la condición externa (x > 0) es verdadera.

La declaración if-else dentro de la condición else

```
x = 15
if x > 10:
    print("x > 10")
else:
    print("x < 10")
    if x % 2 == 0:
        print("x es un número par")
    else:
        print("x es un número impar")
```

En este ejemplo similar, si **x** no es más que diez, el programa comprueba si es par o no. Se ejecuta un bloque interno con una declaración **if-else** anidada dependiendo de la paridad del número. En el caso de x > 10, no se comprueba la paridad.

Declaración if-elif-else

Digamos que necesitas determinar el tipo de triángulo.

```
side1 = 3
side2 = 4
side3 = 5

if side1 == side2 == side3:
    print("Triángulo equilátero")
elif side1 == side2 or side1 == side3 or side2 == side3:
    print("Triángulo isósceles")
else:
    print("Triángulo escaleno")
```

En este ejemplo, dependiendo de la longitud de los lados del triángulo, el programa determina su tipo. Si los tres lados son iguales, es un triángulo equilátero. Si dos lados cualesquiera son iguales, es un triángulo isósceles. De lo contrario, el triángulo se considera escaleno.

Cómo se hace **if-elif-else:**

El programa comprueba la condición_1.

Si condition_1 es verdadera, el bloque de código bajo **if** se ejecuta y los bloques restantes (**elif** y **else**) se ignoran.

Si la condición_1 es falsa, el programa verifica la condición_2.

Si condition_2 es verdadera, se ejecuta el bloque de código bajo el primer **elif** y los bloques restantes (**elifs** y **elses** posteriores) se ignoran.

El proceso se repite para cada **elif** en el siguiente orden.

Si ninguna de las condiciones es verdadera, se ejecuta el bloque de código de **else**, si está presente.

Conceptos básicos sobre la declaración condicional if-elif-else en Python

- La declaración condicional **if-elif-else** en Python te ayuda a escribir código para que produzca un resultado dependiendo de si una determinada condición es verdadera o no. Cuando hay varias condiciones, puedes usar **elif** para probarlas una por una. Si ninguna de las condiciones es verdadera, utilice **else** para ejecutar el código predeterminado.

- En Python, una expresión se considera verdadera (**true)** si su resultado no es cero o la expresión no es un objeto vacío. Y se considera falso **(false)** si el resultado es cero o un objeto vacío, incluido el valor ninguno. Cuando se utilizan operadores de comparación, el resultado de la expresión es **true** si la condición es verdadera y **false** en caso contrario.

- Se pueden realizar varias comparaciones dentro de una condición, lo que permite ampliar las posibilidades de utilizar estructuras condicionales. Esto se puede implementar usando operadores lógicos: **and** devuelve **true** si ambas condiciones son verdaderas y **false** si son falsas; **or** devuelve **true** si al menos una de las condiciones es verdadera y **false** en caso contrario; **not** devuelve **true** si la condición es falsa y **false** si es verdadera.

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

Los bucles en Python, como cualquier otro lenguaje de programación, te permiten realizar una acción varias veces seguidas. Por supuesto, cada operación se puede escribir por separado en código, pero esto requerirá mucho tiempo y espacio y dicha construcción será difícil de leer. Puedes simplificar la tarea usando un bucle.

Por ejemplo, para imprimir los números del uno al cinco en la consola, puede utilizar cinco llamadas independientes a la función **print()**.

```
print(1)
print(2)
print(3)
print(4)
print(5)

>>> 1
>>> 2
>>> 3
>>> 4
>>> 5
```

O puedes envolver todo en un bucle **for**, llamando a la función **print()** solo una vez. En lugar de cinco líneas de código, resultaron ser dos. Si tiene que reescribir todo para imprimir los números del uno al diez, entonces solo necesitará arreglar los límites del bucle y no reescribir diez llamadas.

```
for number in range(1, 6):
    print(number)

>>> 1
>>> 2
>>> 3
>>> 4
>>> 5
```

Para qué se necesitan bucles en Python

En Python, los bucles se utilizan para automatizar procesos tanto en programas simples como en proyectos con grandes bases de código. Normalmente se utilizan bucles:

- **para repetir tareas.** Con los bucles, puedes llamar a funciones varias veces y los bucles mismos pueden depender de las condiciones;
- **trabajando con datos.** Puedes sumar todos los números de la lista, imprimir el resultado, guardarlo o pasarlo como argumento a otra función;
- **iteraciones**. El bucle **for** le permite recorrer todos los caracteres de una cadena, contando su número en total o según una condición determinada, por ejemplo, solo vocales o consonantes;

- **procesamiento de archivos**. Puede cargar un documento y utilizar un bucle para reemplazar todas las apariciones de una determinada palabra por otra.

El bucle WHILE

El bucle **while** le permite realizar acciones repetidas hasta que se alcance una determinada condición. Este es el tipo de bucle más fácil de entender en Python. En general, la construcción **while** tiene este aspecto:

```
while condición:
    bloque_código
```

El bucle **while** incluye:

- **condición**. Una expresión booleana que especifica cuándo debe interrumpirse el bucle. El bloque de código se ejecutará siempre que la condición sea verdadera;
- **bloque de código.** Un conjunto de operaciones y llamadas a funciones que se ejecuta con cada nueva iteración.

Implementemos una verificación simple de que el usuario haya ingresado la contraseña correcta. Para ello crearemos una variable *password* y colocaremos en ella la propia contraseña; para nosotros será *qwerty*; El bucle **while** se ejecutará siempre que el valor del usuario difiera del contenido de la variable *password*.

```
password = "qwerty"
input_password = input("Ingrese la contraseña: ")

while input_password  ! = password:
        print("❌ Contraseña incorrecta. Inténtalo de nuevo.")
        input_password = input("Ingrese la contraseña: ")

print("✅ Acceso permitido. Ha ingresado la contraseña correcta.")
```

El bucle **while** es muy simple, pero aun así puede causar problemas. Por ejemplo, puedes crear un bucle infinito que nunca deje de ejecutarse. Solo puede interrumpirse mediante la finalización forzada del programa, desbordamiento de la memoria o un cambio en la condición del código.

```
while True:
    print("Este es un bucle infinito.")
```

En el ejemplo anterior, la condición **True** siempre es verdadera y no cambia, lo que significa que nada puede finalizar la ejecución del bloque de código. Como resultado, obtenemos una salida de líneas infinitas a la consola. Se debe tener cuidado para garantizar que siempre exista una condición para salir del bucle.

El bucle FOR

Con bucle **for** en Python es conveniente de realizar acciones repetidas en cada elemento de una colección (lista, tupla o cadena). El número de repeticiones depende de cuántos elementos deben atravesarse y el bucle sale una vez que han finalizado los elementos a iterar.

Veamos un ejemplo del uso de un bucle **for** para imprimir los elementos de una lista:

```
fruits = ["manzana", "plátano", "cereza"]

for fruit in fruits:
    print(fruit)
```

En este ejemplo, la variable *fruits* toma valores de la lista *fruits* uno por uno y luego los imprime en la pantalla.

Ahora escribamos un bucle que recorra un conjunto de números y determine si son pares o no. Usaremos **for** y condiciones dentro del bloque de código.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
for num in numbers:
    if num % 2 == 0:
        print(f"{num} - número par")
    else:
        print(f"{num} - número impar")

>>> 1 - número impar
>>> 2 - un número par
>>> 3 - un número impar
>>> 4 - un número par
>>> 5 - un número impar
>>> 6 - un número par
>>> 7 - un número impar
>>> 8 - un número par
>>> 9 - un número impar
```

Rangos de números

A menudo es necesario establecer el número de iteraciones, pero para ello resulta inconveniente crear una nueva variable con la colección. En este caso, puede crear un rango de números para la iteración, que vivirán solo dentro del bucle. Esto se hace usando la función **range(),** a la que se le pueden pasar hasta tres argumentos.

Un argumento. En este caso, puede crear un rango de números de cero a n-1. Por ejemplo, si especifica el *range(5)*, la salida será [0, 1, 2, 3, 4].

```
for i  in range(5):
    print(i)

>>>0
>>>1
>>>2
>>>3
>>>4
```

Dos argumentos. Si pasa dos argumentos a **range()** a la vez, obtendrá un rango del primer

número al segundo. Debemos recordar que durante la salida el segundo argumento se reducirá en uno.

```
for i  in range(5, 10):
    print(i)

>>>5
>>>6
>>>7
>>>8
>>>9
```

Tres argumentos. Puede agregar un tercer argumento y luego especificar el paso de la secuencia. Por ejemplo, si solo necesita números pares, comience el rango con un número par y establezca el paso en 2.

```
for i  in range(2, 11, 2):
    print(i)

>>>2
>>>4
>>>6
>>>8
>>>10
```

Salir de un bucle con BREAK

A veces es necesario salir de un bucle antes de que termine de ejecutarse, por ejemplo, cuando se alcanza una determinada condición que afecta la lógica posterior del programa. En Python, esto se puede hacer usando la palabra clave **break**.

Imaginemos que necesitamos revisar todos los números de la colección usando un bucle **for**, pero salir si encontramos uno.

```
numbers = [2, 3, 4, 1, 5, 6, 7, 8, ]
for num in numbers:
    if num ==1:
        print("Salir del bucle")
        break
    print(num)

>>>2
>>>3
>>>4
>>> Salir del bucle
```

La ejecución del bucle se interrumpió antes de que el programa tuviera tiempo de revisar todos los elementos de la colección. Todo porque alcanzamos la condición y se activó la palabra clave **break**. Si elimina uno de la colección, el bucle imprimirá todos los números.

```
numbers = [2, 3, 4, 5, 6, 7, 8, ]
for num in numbers:
    if num ==1:
        print("Salir del bucle")
        break
     print(num)

>>>2
>>>3
>>>4
>>>5
>>>6
>>>7
>>>8
>>>9
```

Pasar a la siguiente iteración

No sólo puedes salir forzosamente del bucle, sino también omitir sus iteraciones por condición usando **continue**. Ya hemos usado **range()** para imprimir solo los números pares de la colección. Ahora implementemos este algoritmo omitiendo iteraciones.

```
for i  in range(1, 11):
    if i % 2 != 0:
        continue
    print(i)

>>>2
>>>4
>>>6
>>>8
>>>10
```

En la condición, comprobamos el resto al dividir un número entre dos. Si no es cero, nos saltamos el número y no lo imprimimos. Por lo tanto, no interrumpimos la ejecución del bucle, pero nos saltamos iteraciones innecesarias.

Generador de colección

Python te permite escribir bucles **for** en una línea. Esto hace que el código sea más compacto y pueda usarse en tareas adicionales. Por ejemplo, las colecciones se generan utilizando dicho registro.

Imaginemos que necesitamos crear rápidamente una lista de números del uno al diez. Puedes declarar una variable y hacerlo manualmente.

```
num = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(num)

>>>[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Esto lleva mucho tiempo, especialmente si hay más números en la lista. Por lo tanto, puede automatizar la tarea utilizando un bucle **for** de una línea.

```
num = [i for i in range(1, 11)]
print(num)

>>>[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

También puede utilizar condiciones; por ejemplo, generar una lista de números impares únicamente.

```
num = [i for i in range(1, 11) if i % 2 != 0]
print(num)

>>>[1, 3, 5, 7, 9]
```

Bucles anidados

En Python, puedes llamar a otro bucle desde un bucle. En este caso, el código se vuelve más complicado, pero resulta útil en muchas tareas. Por ejemplo, debe revisar todos los valores de una tabla. En un bucle revisamos los valores de las celdas de la fila y en el segundo cambiamos a una nueva fila.

Por ejemplo, imaginemos que tenemos una tabla con dos filas y cuatro columnas. Repasemos cada una de sus celdas usando un bucle anidado, generando el índice de cada celda.

```
for i  in range(1, 3):
    for j  in range(1, 5):
        print(f"Celda {i}:{j}")

>>> Celda 1:1
>>> Celda 1:2
>>> Celda 1:3
>>> Celda 1:4
>>> Celda 2:1
>>> Celda 2:2
>>> Celda 2:3
>>> Celda 2:4
```

- Los bucles son una herramienta básica de programación de Python. Con su ayuda, los desarrolladores pueden realizar rápidamente acciones repetitivas, automatizando el proceso.
- Cada bucle tiene una condición y un bloque de código que se ejecuta mientras la condición sea verdadera.
- Puede utilizar bucles **for** para iterar a través de colecciones de datos y realizar transformaciones.
- Los bucles **while** establecen una condición explícita.
- Puede utilizar la palabra clave **break** para interrumpir la ejecución de un bucle en cualquier momento.
- **Continue** le permite omitir iteraciones que no cumplen la condición.

¿Qué es una lista por comprensión en Python?

Una lista (**list**) es un conjunto ordenado de elementos, cada uno de los cuales tiene su propio número o índice, lo que permite un acceso rápido a él. La numeración de elementos en una lista comienza desde 0.

Una lista puede contener simultáneamente datos de diferentes tipos, por ejemplo, cadenas y números. Y puedes poner otro en una lista y nada se romperá.

Todos los elementos de la lista están numerados. Podemos encontrar fácilmente el índice del elemento y acceder a él.

Las listas se denominan estructuras de datos dinámicas porque se pueden cambiar sobre la marcha: eliminar uno o más elementos, reemplazar o agregar otros nuevos.

Cuando creamos un objeto **list**, se reserva espacio para él en la memoria de la computadora. No tenemos que preocuparnos por cuánto espacio se asigna y cuándo se libera: Python hará todo por sí mismo. Por ejemplo, cuando agregamos nuevos elementos, asigna memoria y cuando eliminamos los antiguos, la libera.

Sin embargo, las listas de Python pueden almacenar objetos de diferentes tamaños y tipos. Además, el tamaño de una matriz es limitado, pero el tamaño de una lista en Python no lo es. Pero todavía sabemos cuántos elementos tenemos, lo que significa que podemos acceder a cualquiera de ellos mediante índices.

Y aquí hay un pequeño truco: las listas en Python son una serie de referencias. Cada elemento de dicha matriz no almacena los datos en sí, sino un enlace a su ubicación en la memoria de la computadora.

Cómo crear una lista en Python

Para crear un objeto **list,** Python usa corchetes - **[]**. Dentro de ellos, se listan los elementos separados por comas:

```
a = [1, 2, 3]
```

Creamos una lista **a** y le pusimos tres números, separados por comas. Imprimamos usando la función **print()**:

```
print(a)

>>> [1, 2, 3]
```

Python muestra elementos entre corchetes para indicar que es **list** y también coloca comas entre elementos.

Ya hemos dicho que las listas pueden almacenar cualquier tipo de datos. En el siguiente ejemplo, el objeto **b** almacena: una cadena - **cat**, un número - **123** y un valor booleano - **True**:

```
b = ['cat', 123, True]

print(b)

>>> ['cat', 123, True]
```

También puedes crear listas anidadas en Python:

```
c = [1, 2, [3, 4]]

print(c)

>>> [1, 2, [3, 4]]
```

Recibimos un objeto que consta de dos números (**1** y **2**) y **list** anidada con dos elementos (**3**, **4**).

Operaciones con listas

Si simplemente almacena datos en listas, serán de poca utilidad. Veamos qué operaciones le permiten realizar.

Indexación

Se accede a los elementos de la lista mediante índices, a través de corchetes []:

```
a = [1, 2, 3]

print(a[1])

>>> 2
```

Accedimos al segundo elemento y lo imprimimos usando **print()**.
Es importante recordar dos cosas aquí:

- cada elemento tiene su propio índice;
- los índices comienzan desde 0.

```
a = [1, 2, 3, 4]

a[0]  #  Se refiere a 1
a[2]  #  Se refiere a 3
a[3]  #  Se refiere a 4
a[4]  #  arrojará un error
```

En la última línea accedimos a un índice inexistente, por lo que Python arrojó un error.

Además, Python admite el acceso a varios elementos a la vez, a intervalos. Esto se hace usando dos puntos - :.

```
a = [1, 2, 3, 4]

a[0:2]  #  Obtener [1, 2]
```

Los dos puntos le permiten obtener un segmento de la lista. La forma completa del operador es: **start_index:end_index:step.**

Aquí indicamos en qué índice comienza el "sector", en qué termina y en qué paso se toman los elementos; por defecto **1**. La única advertencia con el índice final: aunque podemos pensar que terminaremos allí, de hecho Python se detendrá en el elemento con índice **final_index - 1.**

En el ejemplo anterior, comenzamos en el índice **0** y terminamos en **1** porque el último índice no está incluido. Nuestro paso fue el **1**, es decir, repasamos cada elemento.

Compliquemos el ejemplo:

```
a = [1, 2, 3, 4, 5]

a[1:6:2]  #  Obtener [2, 4]
```

Aquí repasamos los elementos en pasos de **2**. Comenzamos en el índice **1** (este es el primer número entre paréntesis) y terminamos en el índice **6**, sin incluirlo. Avanzamos con el paso **2**, es decir, a través de un elemento, y obtuvimos **[2, 4]**.

Elementos cambiantes

Las listas son una estructura de datos dinámica. Esto significa que podemos cambiarlos después de la creación.

Por ejemplo, puedes reemplazar un elemento por otro:

```
a = [1, 2, 3]
a[1] = 4

print(a)

>>> [1, 4, 3]
```

Accedimos al elemento por índice y lo reemplazamos por el número **4**. Todo salió bien, la lista ha cambiado.

Pero hay que tener cuidado porque puede pasar esto:

```
a = [1, 2]
b = a
a[0] = 5

print(a)
print(b)

>> [5, 2]
>> [5, 2]
```

Primero creamos una lista **a** con dos elementos: **1** y **2**. Luego declaramos una variable **b** y le asignamos el contenido de **a**. Luego reemplazamos el primer elemento en **a** y... nos sorprendió que también fuera reemplazado en **b**.

El problema es que **a** es una referencia al área de la memoria de la computadora donde se almacena el primer elemento de la lista, así como el siguiente elemento.

Cada elemento de la lista tiene cuatro secciones: su dirección, datos, la dirección del siguiente elemento y la dirección del anterior. Si tenemos acceso a algún elemento, podemos avanzar y retroceder fácilmente por esta lista y cambiar sus datos.

Entonces, cuando asignamos la lista **b** a la lista **a**, en realidad le asignamos una referencia al primer elemento, esencialmente convirtiéndolos en una lista.

Fusionar listas

A veces resulta útil combinar dos listas. Para hacer esto, use el operador **+**:

```
a = [1, 2]
b = [3, 4]
c = a + b
a[0] = 5

print(c)

>>> [1, 2, 3, 4]
```

Hemos creado dos listas: **a** y **b**. Luego reasignamos **a** con una nueva lista, que se convirtió en la unión de las antiguas **a** y **b**.

Descomposición de listas

Los elementos de la lista se pueden asignar a variables individuales:

```
a = [1, 2, 3]
d1, d2, d3 = a

print(d1)
print(d2)
print(d3)

>>> 1
>>> 2
>>> 3
```

Aquí, los elementos de la lista **a** se toman uno por uno, comenzando con el índice **0**, y se asignan a variables. Y a diferencia de asignar una lista a otra, en este caso Python creará tres números enteros separados que no tienen nada que ver con los elementos de la lista y los asignará a tres variables. Por tanto, si cambiamos, por ejemplo, la variable **d2**, a la lista **a** no le pasará nada.

Iterando sobre elementos

Podemos iterar sobre los elementos de una lista usando bucles **for** y **while**.
Así es como se ve una búsqueda por el bucle **for**:

```
animals = ['cat', 'dog', 'bat']
for animal in  animals:
  print(animal)

>>> cat
>>> dog
>>> bat
```

Aquí iteramos sobre cada elemento de la lista y los imprimimos usando la función **print().**

Y así es como se ve al pasar por un bucle **while**:

```
animals = [‘cat’, ‘dog’, ‘bat’]
i = 0
while i < len(animals):
  print(animals[i])
  i += 1

>>> cat
>>> dog
>>> bat
```

Esta iteración es un poco más complicada porque usamos una variable adicional **i** para acceder a los elementos de la lista. También utilizamos la función **len()** incorporada para averiguar el tamaño de nuestra lista. Y en la condición del bucle **while**, indicamos el signo "menor que" **(<),** porque la indexación de elementos llega hasta el valor **del número de elementos de la lista - 1.** Como en el ejemplo anterior, todos los elementos se imprimen a su vez utilizando la función **print()**.

Comparación de listas

Python admite la comparación de listas. Dos listas se consideran iguales si contienen los mismos elementos. La función devuelve un valor booleano: **True** o **False**:

```
a = [1, 2, 3]
b = [1, 2, 3]

print(a == b)

>>> True
```

Encontramos que las listas son iguales.

En algunos idiomas, la igualdad también se verifica si las variables se refieren al mismo objeto. Esto generalmente se hace usando el operador ===. En Python, esto se puede hacer mediante el operador **is**, que prueba si dos variables tienen la misma dirección en la memoria:

```
a = [1, 2, 3]
b = a

print(a is b)

>>> True
```

Descubrimos que dos variables se refieren a la misma dirección en la memoria.

Funciones integradas para listas de Python

Python tiene cuatro funciones que te permiten averiguar la longitud de una lista, ordenarla y devolver los valores máximo y mínimo.

len()

Devuelve la longitud de la lista:

```
a = [5, 3, 1]
len(a) # 3
```

sorted()

Devuelve una lista ordenada:

```
a = [8, 1, 3, 2]
sorted(a) # [1, 2, 3, 8]
```

min() y max()

Devuelve el elemento más pequeño y más grande de la lista:

```
a = [1, 9, -2, 3]
min(a) # -2
max(a) # 9
```

Métodos de lista de Python

Para facilitar la gestión de elementos de la lista, la biblioteca estándar de Python tiene un conjunto de métodos populares para listas.

append()

Agrega un nuevo elemento al final de la lista:

```
a = [1, 2, 3]
a.append(4)
print(a)

>>> [1, 2, 3, 4]
```

extend()

Agrega un conjunto de elementos al final de la lista:

```
a = [1, 2, 3]
a.extend([4, 5])
print(a)

>>> [1, 2, 3, 4, 5]
```

Dentro del método **extend()**, debe pasar un objeto iterable, por ejemplo, otra lista o una cadena.

Así es como el método **extend()** agregará una cadena:

```
a = [‘cat’, ‘dog’, ‘bat’]
a.extend(‘mouse’)
print(a)

>>> [‘cat’, ‘dog’, ‘bat’, ‘m’, ‘o’, ‘u’, ‘s’, ‘e’]
```

insert()
Agrega un nuevo elemento en el índice:

```
a = [1, 2, 3]
a.insert(0, 4)
print(a)

>>> [4, 1, 2, 3]
```

Primero pasamos el índice en el que queremos insertar un nuevo elemento y luego el elemento en sí.

remove()
Elimina un elemento de la lista:

```
a = [1, 2, 3, 1]
a.remove(1)
print(a)

>>> [2, 3, 1]
```

El método elimina sólo la primera aparición del elemento. El resto permanece intacto.

clear()
Elimina todos los elementos de la lista y la deja vacía:

```
a = [1, 2, 3]
a.clear()
print(a)

>>> []
```

index()
Devuelve el índice de un elemento de lista en Python:

```
a = [1, 2, 3]
print(a.index(2))

>>> 1
```

Si el elemento no está en la lista, se mostrará un error.

pop()
Elimina un elemento en el índice y lo devuelve como resultado:

```
a = [1, 2, 3]
print(a.pop())
print(a)

>>> 3
>>> [1, 2]
```

No pasamos un índice al método, por lo que eliminó el último elemento de la lista. Si pasas el índice, se verá así:

```
a = [1, 2, 3]
print(a.pop(1))
print(a)

>>> 2
>>> [1, 3]
```

count()
Cuenta el número de veces que se repite un elemento en una lista:

```
a = [1, 1, 1, 2]
print(a.count(1))
print(a)

>>> 3
```

sort()
Ordena la lista:

```
a = [4, 1, 5, 2]
a.sort() # [1, 2, 4, 5]
```

Si necesitamos ordenar en orden inverso, de mayor a menor, el método tiene un parámetro **reverse**:

```
a = [1, 3, 2, 4]
a.sort(reverse=True) # [5, 4, 2, 1]
```

reverse()
Reorganiza los elementos en orden inverso:

```
a = [4, 1, 5, 2]
a.reverse() # [4, 2, 3, 1]
```

copy()
Copia la lista:

```
a = [1, 2, 3]
b = a.copy()

print(b)

 >>> [1, 2, 3]
```

¿Qué es un argumento en Python?

Un argumento es un valor que se pasa a una función cuando el programa la llama.

Nombrar argumentos en Python

Los argumentos se nombran con un par nombre-valor que se pasa a la función. Si el significado está directamente relacionado con el argumento, entonces, al pasar el argumento al argumento, el orden no tiene la culpa. Por ejemplo:

```
def display_info(first_name, last_name):
    print("First Name: ", first_name)
    print("Last Name: ", last_name)

display_info(last_name = "Cartman", first_name = "Eric")

>>> First Name: Eric
>>> Last Name: Cartman
```

Volver respecto a la función wiki:

```
display_info(last_name = "Cartman", first_name = "Eric")
```

Aquí indicamos tanto los nombres de los argumentos como sus valores al llamar a la función.

Aparentemente, el argumento **first_name** de la función llamada se convierte en el parámetro **first_name** de la función designada. Entonces, el argumento **last_name** de la función en la que se hizo clic se convierte en el parámetro **last_name** de la función designada.

En tales escenarios, el orden en que se presentan los argumentos no importa mucho.

Más argumentos en Python

A veces no sabemos de antemano cuántos argumentos se pasarán a la función. Para hacer frente a tal situación, podemos usar algunos argumentos adicionales en Python.

Los argumentos adicionales le permiten pasar una cantidad diferente de valores por hora a la llamada de función. Para insertar este tipo de argumento, se coloca una estrella (*****) antes del nombre del parámetro de la función asignada. Por ejemplo:

```
def find_sum(*numbers):
    result = 0

    for num in numbers:
        result = result + num

    print("Sum = ", result)

# llamar a una función con 3 argumentos

find_sum(1, 2, 3)
```

```
# llamar a una función con 2 argumentos

find_sum(4, 9)

>>> Sum = 6
>>> Sum = 13
```

Aquí creamos una función **find_sum(),** que acepta una gran cantidad de argumentos.

¿Qué es una función Lambda en Python?

Las funciones lambda son funciones anónimas que solo pueden contener una expresión. Por lo general, se utilizan para realizar operaciones simples que no requieren una definición de función completa utilizando la palabra clave **def**. Las funciones Lambda pueden tomar cualquier número de argumentos, pero solo pueden devolver un valor.

Crear funciones Lambda

La creación de funciones lambda en Python se realiza utilizando la palabra clave **lambda**. La sintaxis para crear funciones lambda es la siguiente:

```
lambda arguments: expression
```

Un ejemplo de creación de una función lambda que toma dos argumentos y devuelve su suma:

```
sum = lambda x, y: x + y
```

Usando funciones Lambda

Las funciones Lambda se pueden utilizar en cualquier lugar donde se utilicen funciones habituales. Son especialmente útiles cuando se usan con funciones de orden superior como **map(), filter()** y **reduce()**.

Ejemplo de uso de una función lambda con la función map()

La función **map()** aplica la función especificada a todos los elementos de un objeto iterable (como una lista) y devuelve un iterador con los resultados. En el siguiente ejemplo, usamos una función lambda para elevar al cuadrado cada elemento de una lista:

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(lambda x: x**2, numbers)
print(list(squared_numbers)) # Output: [1, 4, 9, 16, 25]
```

Ejemplo de uso de una función lambda con la función filter()

La función **filter()** filtra los elementos de un iterable según la función especificada y devuelve solo aquellos elementos para los cuales la función devuelve **True**. En el siguiente ejemplo, utilizamos una función lambda para filtrar números pares de una lista:

```
numbers = [1, 2, 3, 4, 5]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers)) # Output: [2, 4]
```

Las funciones Lambda proporcionan una manera conveniente de crear funciones anónimas para realizar tareas simples en Python. Son especialmente útiles cuando se trabaja con funciones de orden superior y pueden simplificar enormemente su código.

¿Qué es un paquete pip?

Pip es una herramienta de administración de paquetes estándar que proporciona una manera conveniente de instalar, actualizar y eliminar paquetes de Python. Le permite administrar de manera eficiente las dependencias del proyecto al admitir un fácil acceso a miles de paquetes desde el Índice de paquetes de Python (PyPI).

Comandos básicos:

- **pip install package_name:** Instala el paquete especificado.
- **pip uninstall package_name:** desinstala el paquete especificado.
- **pip list:** muestra los paquetes instalados.
- **pip show package_name:** muestra información sobre un paquete específico.
- **pip search query:** busca paquetes asociados con una consulta.

Ejemplo de uso:

1. Instalación del paquete:
   pip install requests
2. Quitar un paquete:
   pip uninstall requests
3. Ver paquetes instalados:
   pip list
4. Ver información del paquete:
   pip show Flask
5. Buscar paquetes:
   pip search matplotlib

El uso de **pip** hace que sea mucho más fácil administrar las dependencias del proyecto y también facilita agregar nuevas bibliotecas y herramientas a su proyecto Python. Asegúrese de tener **pip** instalado en su entorno Python para que pueda administrar fácilmente los paquetes para sus aplicaciones.