Técnicas de Programación

Instituto de Formación Técnica Superior Nro. 11
Docente: Lic. Norberto A. Orlando

Indice

UNIDAD 3: Valores Booleanos, Ejecución Condicional, Bucles, Listas y su procesamiento, Operaciones Lógicas y de Bit a Bit

Aprenderemos sobre listas, operaciones lógicas y de bit a bit, cómo usar mecanismos de flujo de control para tomar decisiones y repetir la ejecución de código en Python

Toma de decisiones en Python

Aprenderemos sobre las sentencias condicionales y cómo usarlas para tomar decisiones en Python

Preguntas y Respuestas

Un programador escribe un programa y el **programa hace preguntas**.

Una computadora ejecuta el programa y **proporciona las respuestas**. El programa debe ser capaz de **reaccionar de acuerdo con las respuestas recibidas**.

Afortunadamente, las computadoras solo conocen dos tipos de respuestas:

- Si, es cierto.
- No, esto es falso.

Nunca obtendrás una respuesta como Déjame pensar...., no lo sé, o probablemente sí, pero no lo sé con seguridad.

Para hacer preguntas, Python utiliza un conjunto de operadores muy especiales. Revisemos uno tras otro, ilustrando sus efectos en algunos ejemplos simples.

Comparación: operador de igualdad

Pregunta: ¿son dos valores iguales?

Para hacer esta pregunta, se utiliza el **==** (igual igual) operador.

No olvides esta importante distinción:

- **=** es un **operador de asignación**, por ejemplo, **a = b** asigna a la variable a el valor de b;
- **==** es una pregunta ¿Son estos valores iguales? así que **a == b compara a y b**.

Es **un operador binario con enlazado del lado izquierdo**. Necesita dos argumentos y **verifica si son iguales**.

Operadores

Igualdad: El operador igual a (==)

El **operador ==** (igual a) compara los valores de dos operandos. Si son iguales, el resultado de la comparación es **True**. Si no son iguales, el resultado de la comparación es **False**.

Entonces, vamos a practicar la comprensión del operador ==

```
1  var = 0  # Asignando 0 a var
2  print(var == 0)
3
4  var = 1  # Asignando 1 a var
5  print(var == 0)
6
```

```
Console >_

True
False
```

Desigualdad: el operador no es igual a (!=)

El **operador != (no es igual a)** también compara los valores de dos operandos. Aquí está la diferencia: si son iguales, el resultado de la **comparación** es **False**. Si **no son iguales**, el resultado de la **comparación es True**.

Ahora echa un vistazo a la comparación de desigualdad a continuación

```
var = 0  # Asignando 0 a var
print(var != 0)

var = 1  # Asignando 1 a var
print(var != 0)
```

```
Console >_

False
True
```

Operadores de comparación: mayor que

También se puede hacer una pregunta de comparación usando el operador > (mayor que).

```
1  varA = 1 # Asignando 1 a varA
2  varB = 0 # Asignando 0 a varB
3
4  print(varA > varB)
5
6  varA = 0 # Asignando 0 a varA
7  varB = 1 # Asignando 1 a varB
8
9  print(varA > varB)
10
11 varA = 1 # Asignando 1 a varA
12 varB = 1 # Asignando 1 a varB
13
14 print(varA > varB)
```

```
Console >_

True
False
False
```

Operadores de comparación: mayor o igual que

El **operador mayor** que tiene otra variante especial, una variante no estricta, pero se denota de manera diferente que la notación aritmética clásica: **>= (mayor o igual que).**

Hay dos signos subsecuentes, no uno.

```
1  varA = 1 # Asignando 1 a varA
2  varB = 0 # Asignando 0 a varB
3
4  print(varA >= varB)
5
6  varA = 0 # Asignando 0 a varA
7  varB = 1 # Asignando 1 a varB
8
9  print(varA >= varB)
10
11 varA = 1 # Asignando 1 a varA
12 varB = 1 # Asignando 1 a varB
13
14 print(varA >= varB)
```

```
Console >_

True
False
True
```

Operadores de comparación: menor o igual que

Como probablemente ya hayas adivinado, los operadores utilizados en este caso son: El **operador < (menor que)** y su hermano no estricto**: <= (menor o igual que).**

Observa este ejemplo simple:

```python
varA = 1 # Asignando 1 a varA
varB = 0 # Asignando 0 a varB

print(varA <= varB)

varA = 0 # Asignando 0 a varA
varB = 1 # Asignando 1 a varB

print(varA <= varB)

varA = 1 # Asignando 1 a varA
varB = 1 # Asignando 1 a varB

print(varA <= varB)
```

```
Console >_

False
True
True
```

Haciendo uso de las respuestas

¿Qué puedes hacer con la respuesta (es decir, el resultado de una operación de comparación) que se obtiene de la computadora?

Existen al menos dos posibilidades: **primero, puedes memorizarlo (almacenarlo en una variable)** y utilizarlo más tarde y el contenido de la variable te dirá la respuesta a la pregunta.

La **segunda posibilidad** es más conveniente y mucho más común: **puedes utilizar la respuesta que obtengas para tomar una decisión sobre el futuro del programa**.

Ahora necesitamos actualizar nuestra tabla de prioridades, y poner todos los nuevos operadores en ella. Ahora se ve como a continuación:

Prioridad	Operador	
1	`+`, `-`	unario
2	`**`	
3	`*`, `/`, `//`, `%`	
4	`+`, `-`	binario
5	`<`, `<=`, `>`, `>=`	
6	`==`, `!=`	

Condiciones y ejecución condicional

Ya sabes como hacer preguntas a Python, pero aún no sabes como hacer un uso razonable de las respuestas. Se debe tener un mecanismo que le permita hacer algo **si se cumple una condición, y no hacerlo si no se cumple**.

Es como en la vida real: haces ciertas cosas o no cuando se cumple una condición específica, por ejemplo, sales a caminar si el clima es bueno, o te quedas en casa si está húmedo y frío.

Para tomar tales decisiones, Python ofrece una **instrucción** especial. Debido a su naturaleza y su aplicación, se denomina **instrucción condicional** (o sentencia condicional).

Existen varias variantes de la misma. Comenzaremos con la más simple, aumentando la dificultad lentamente.

La primera forma de una sentencia condicional:

```
if true_or_not:
    do_this_if_true
```

Esta sentencia condicional consta de los siguientes elementos, estrictamente necesarios en este orden:

- La palabra clave reservada **if**
- Uno o más espacios en blanco;
- **Una expresión** (una pregunta o una respuesta) cuyo valor se interpretar únicamente en términos de **True** (cuando su valor no sea cero) y **False** (cuando sea igual a cero);
- Unos **dos puntos** seguidos de una nueva línea
- **Una instrucción con sangría o un conjunto de instrucciones** (se requiere absolutamente al menos una instrucción); la sangría se puede lograr de dos maneras: insertando un número particular de espacios (la recomendación es usar cuatro espacios de sangría), o usando el tabulador; nota: si hay mas de una instrucción en la parte con sangría, la sangría debe ser la

misma en todas las líneas; aunque puede parecer lo mismo si se mezclan tabuladores con espacios, es importante que todas las sangrías sean exactamente iguales Python 3 no permite mezclar espacios y tabuladores para la sangría.

¿Cómo funciona esta sentencia?

- Si la expresión **true_or_not representa la verdad** (es decir, su valor no es igual a cero), las **sentencias con sangría se ejecutarán**;
- Si la expresión **true_or_not no representa la verdad** (es decir, su valor es igual a cero), las **sentencias con sangría se omitirán** (ignorado), y la siguiente instrucción ejecutada será la siguiente al nivel de la sangría original.

Ejecución condicional: la sentencia if-else

Comenzamos con una frase simple que decía: Si el clima es bueno, saldremos a caminar.

Nota: no hay una palabra sobre lo que sucederá si el clima es malo. Solo sabemos que no saldremos al aire libre, pero no sabemos que podríamos hacer. Es posible que también queramos planificar algo en caso de mal tiempo.

Podemos decir, por ejemplo: Si el clima es bueno, saldremos a caminar, de lo contrario, iremos al cine.

Ahora sabemos lo que haremos si se cumplen las condiciones, y sabemos lo que haremos si no todo sale como queremos. En otras palabras, tenemos un "Plan B".

Python nos permite expresar dichos planes alternativos. Esto se hace con una segunda forma, ligeramente mas compleja, de la sentencia condicional, **la sentencia if-else**:

```
1  if true_or_false_condition:
2      perform_if_condition_true
3  else:
4      perform_if_condition_false
5
```

Por lo tanto, hay una nueva palabra: **else - esta es una palabra clave reservada**.

La parte del código que comienza con **else** dice que hacer si no se cumple la condición especificada por el **if** (observa los **dos puntos después de la palabra**).

La ejecución de **if-else** es la siguiente:

- Si la condición se evalúa como **True** (su valor no es igual a cero), la instrucción **perform_if_condition_true se ejecuta**, y la sentencia condicional llega a su fin.

- Si la condición se evalúa como **False** (es igual a cero), la instrucción **perform_if_condition_false se ejecut**a, y la sentencia condicional llega a su fin.

Sentencias if-else anidadas

Ahora, analicemos dos casos especiales de la sentencia condicional.

Primero, considera el caso donde **la instrucción colocada después del if es otro if**.

Lee lo que hemos planeado para este Domingo. Si hay buen clima, saldremos a caminar. Si encontramos un buen restaurante, almorzaremos allí. De lo contrario, vamos a comer un sandwich. Si hay mal clima, iremos al cine. Si no hay boletos, iremos de compras al centro comercial más cercano.

Escribamos lo mismo en Python. Considera cuidadosamente el código siguiente:

```
1  if the_weather_is_good:
2      if nice_restaurant_is_found:
3          have_lunch()
4      else:
5          eat_a_sandwich()
6  else:
7      if tickets_are_available:
8          go_to_the_theater()
9      else:
10         go_shopping()
```

Aquí hay dos puntos importantes:

- Este uso de la sentencia **if** se conoce como **anidamiento**; recuerda que cada **else** se refiere al **if que se encuentra en el mismo nivel de sangría**; se necesita saber esto para determinar cómo se relacionan los **ifs** y los **else**;
- Considera como la sangría **mejora la legibilidad** y hace que el código sea más fácil de entender y rastrear.

La sentencia elif

El segundo caso especial presenta otra nueva palabra clave de Python: **elif**. Como probablemente sospechas**, es una forma más corta de else if**.

elif se usa para verificar más de una condición, y para **detener** cuando se encuentra la primera sentencia verdadera.

Nuestro siguiente ejemplo se parece a la anidación, pero las similitudes son muy leves. Nuevamente, cambiaremos nuestros planes y los expresaremos de la siguiente manera: si hay buen clima, saldremos a caminar, de lo contrario, si obtenemos entradas, iremos al cine, de lo contrario, si hay mesas libres en el restaurante, vamos a almorzar; si todo falla, regresaremos a casa y jugaremos ajedrez.

¿Has notado cuantas veces hemos usado las palabras de lo contrario? Esta es la etapa en la que la palabra clave reservada **elif** desempeña su función.

Escribamos el mismo escenario empleando Python:

```
1  if the_weather_is_good:
2      go_for_a_walk()
3  elif tickets_are_available:
4      go_to_the_theater()
5  elif table_is_available:
6      go_for_lunch()
7  else:
8      play_chess_at_home()
9
```

La forma de ensamblar las siguientes sentencias **if-elif-else** a veces se denomina **cascada**.

Observa de nuevo como la sangría mejora la legibilidad del código.

Se debe prestar atención adicional a este caso:

- **No debes usar else sin un if precedente**
- **else siempre es la última rama de la cascada**, independientemente de si has usado elif o no
- **else es una parte opcional de la cascada, y puede omitirse**
- Si hay una rama **else** en la cascada, solo se ejecuta una de todas las ramas
- Si no hay una rama **else**, es posible que no se ejecute ninguna de las opciones disponibles.

Bucles en Python

Aprenderás acerca de los bucles en Python y específicamente - los bucles **while y for**. Aprenderás cómo crear (y evitar caer en) bucles infinitos, cómo salir de bucles, y omitir ciertas iteraciones en los bucles.

Bucles en tu código con while

En general, en Python, un bucle se puede representar de la siguiente manera:

```
while
    instruction
```

Si observas algunas similitudes con la instrucción **if**, está bien. De hecho, la diferencia sintáctica es solo una: usa la palabra **while** en lugar de la palabra **if**.

La diferencia semántica es más importante: cuando se cumple la condición, **if** realiza sus sentencias **sólo una vez**; **while repite la ejecución siempre que la condición se evalúe como True**.

Nota: todas las reglas relacionadas con **sangría** también se aplican aquí.

Observa el algoritmo a continuación:

```
while conditional_expression:
    instruction_one
    instruction_two
    instruction_three
    :
    :
    instruction_n
```

Ahora, es importante recordar que:

- si deseas **ejecutar más de una sentencia dentro de un while**, debes (como con **if**) **poner sangría** a todas las instrucciones de la misma manera.
- una instrucción o conjunto de instrucciones ejecutadas dentro del **while** se llama el **cuerpo del bucle**.
- si la condición es **False** (igual a cero) tan pronto como se compruebe por primera vez, el cuerpo no se ejecuta ni una sola vez (ten en cuenta la analogía de no tener que hacer nada si no hay nada que hacer).
- el cuerpo debe poder cambiar el valor de la condición, porque si la condición es **True** al principio, el cuerpo podría funcionar continuamente hasta el infinito. Observa que hacer una cosa generalmente disminuye la cantidad de cosas por hacer.

Un bucle infinito

Un bucle infinito, también denominado **bucle sin fin**, es una secuencia de instrucciones en un programa que se repite indefinidamente (bucle sin fin).

Este es un ejemplo de un bucle que no puede finalizar su ejecución:

```
1  while True:
2      print("Estoy atrapado dentro de un bucle.")
3
```

Este bucle imprimirá infinitamente "Estoy atrapado dentro de un bucle." en la pantalla.

Te mostraremos como usar este bucle recién aprendido para encontrar el número más grande de un gran conjunto de datos ingresados.

Analiza el programa cuidadosamente. Localiza donde comienza el bucle (línea 8) y **descubre como se sale del cuerpo del bucle**:

```
1  # Almacena el actual número más grande aquí.
2  largest_number = -999999999
3
4  # Ingresa el primer valor.
5  number = int(input("Introduce un número o escribe -1 para detener: "))
6
7  # Si el número no es igual a -1, continuaremos
8  while number != -1:
9      # ¿Es el número más grande que el valor de largest_number?
10     if number > largest_number:
11         # Sí si, se actualiza largest_number.
12         largest_number = number
13     # Ingresa el siguiente número.
14     number = int(input("Introduce un número o escribe -1 para detener: "))
15
16 # Imprime el número más grande.
17 print("El número más grande es:", largest_number)
18
```

Veamos otro ejemplo utilizando el bucle **while**. Sigue los comentarios para descubrir la idea y la solución.

```python
# Un programa que lee una secuencia de números
# y cuenta cuántos números son pares y cuántos son impares.
# El programa termina cuando se ingresa un cero.

odd_numbers = 0
even_numbers = 0

# Lee el primer número.
number = int(input("Introduce un número o escribe 0 para detener: "))

# 0 termina la ejecución.
while number != 0:
    # Verificar si el número es impar.
    if number % 2 == 1:
        # Incrementar el contador de números impares odd_numbers.
        odd_numbers += 1
    else:
        # Incrementar el contador de números pares even_numbers.
        even_numbers += 1
    # Leer el siguiente número.
    number = int(input("Introduce un número o escribe 0 para detener: "))

# Imprimir resultados.
print("Conteo de números impares:", odd_numbers)
print("Conteo de números pares:", even_numbers)

```

Ciertas expresiones se pueden simplificar sin cambiar el comportamiento del programa.

Intenta recordar cómo Python interpreta la verdad de una condición y ten en cuenta que estas dos formas son equivalentes:

while number != 0: y while number:

La condición que verifica si un número es impar también puede codificarse en estas formas equivalentes:

if number % 2 == 1: y if number % 2:

Empleando una variable counter para salir del bucle

```python
counter = 5
while counter != 0:
    print("Dentro del bucle.", counter)
    counter -= 1
print("Fuera del bucle.", counter)

```

Este código está destinado a imprimir la cadena "Dentro del bucle." y el valor almacenado en la variable counter durante un bucle dado exactamente cinco veces. Una vez que la condición se haya cumplido (la variable counter ha alcanzado 0), se sale del bucle y aparece el mensaje "Fuera del bucle." así como tambien el valor almacenado en counter se imprime.

Pero hay una cosa que se puede escribir de forma más compacta - la condición del bucle while.

¿Puedes ver la diferencia?

```
counter = 5
while counter:
    print("Dentro del bucle.", counter)
    counter -= 1
print("Fuera del bucle.", counter)

```

Bucles en tu código con for

Otro tipo de bucle disponible en Python proviene de la observación de que a veces es más importante contar los "giros o vueltas" del bucle que verificar las condiciones.

Imagina que el cuerpo de un bucle debe ejecutarse exactamente cien veces. Si deseas utilizar el bucle **while** para hacerlo, puede tener este aspecto:

```
i = 0
while i < 100:
    # do_something()
    i += 1

```

Hay un bucle especial para este tipo de tareas, y se llama **for**.

En realidad, el bucle **for** está diseñado para realizar tareas más complicadas, **puede "explorar" grandes colecciones de datos elemento por elemento**. Te mostraremos como hacerlo pronto, pero ahora presentaremos una variante más sencilla de su aplicación.

```
for i in range(100):
    # do_something()
    pass

```

Existen algunos elementos nuevos;

- la palabra reservada **for** abre el **bucle for**; nota - No hay condición después de eso; no tienes que pensar en las condiciones, ya que se verifican internamente, sin ninguna intervención.
- cualquier variable después de la palabra reservada **for** es la variable de **control del bucle**; cuenta los giros del bucle y lo hace automáticamente.
- la palabra reservada **in** introduce un elemento de sintaxis que describe el rango de valores posibles que se asignan a la variable de control.
- la función **range()** (esta es una función muy especial) es responsable de generar todos los valores deseados de la variable de control; en nuestro ejemplo, la función creará (incluso podemos decir que alimentará el bucle con) valores subsiguientes del siguiente conjunto: 0, 1, 2 .. 97, 98, 99; nota: en este caso, la función **range()** comienza su trabajo desde 0 y lo finaliza un paso (un número entero) antes del valor de su argumento.
- nota la palabra clave **pass** dentro del cuerpo del bucle - no hace nada en absoluto; es una instrucción vacía - la colocamos aquí porque la sintaxis del bucle for exige al menos una instrucción dentro del cuerpo.

Veamos el siguiente ejemplo

```
1 for i in range(10):
2     print("El valor de i es", i)
3
```

```
Console >_
El valor de i es 0
El valor de i es 1
El valor de i es 2
El valor de i es 3
```

```
Console >_
El valor de i es 6
El valor de i es 7
El valor de i es 8
El valor de i es 9
```

Nota:

- El bucle se ha ejecutado diez veces (es el argumento de la función **range())**
- El valor de la última variable de control es 9 (no 10, ya que comienza desde 0 , no desde 1)

La invocación de la función **range**() puede estar equipada con **dos argumentos**, no solo uno:

```
1 for i in range(2, 8):
2     print("El valor de i es", i)
3
```

En este caso, el primer argumento determina el valor inicial (primero) de la variable de control.

El último argumento muestra el primer valor que no se asignará a la variable de control.

Nota: la función **range**() **solo acepta enteros como argumentos** y **genera secuencias de enteros**.

¿Puedes adivinar la output del programa? Ejecútalo para comprobar si ahora también estabas en lo cierto.

El primer valor mostrado es 2 (tomado del primer argumento de range()).

El último es 7 (aunque el segundo argumento de range() es 8).

Más acerca del bucle for y la función range() con tres argumentos

La función **range()** también puede aceptar **tres argumentos**: Echa un vistazo al código del editor.

```
1 for i in range(2, 8, 3):
2     print("El valor de i es", i)
3
4
```

```
Console >_

El valor de i es 2
El valor de i es 5
```

El tercer argumento es un **incremento** - es un valor agregado para controlar la variable en cada giro del bucle (como puedes sospechar, el valor predeterminado del incremento es 1).

El **primer argumento** pasado a la función **range**() nos dice cual es el **número de inicio de la secuencia** (por lo tanto 2 en la output). El **segundo argumento** le dice a la función **dónde detener la secuencia** (la función genera números hasta el número indicado por el segundo argumento, pero no lo incluye). Finalmente, el **tercer argumento** indica el paso, que en realidad significa la diferencia entre cada número en la secuencia de números generados por la función.

2 (número inicial) → **5** (2 incremento por 3 es igual a 5 - el número está dentro del rango de 2 a 8) → 8 (5 incremento por 3 es igual a 8 - el número no está dentro del rango de 2 a 8, porque el parámetro de parada no está incluido en la secuencia de números generados por la función).

Nota: el conjunto generado por **range()** debe ordenarse en un **orden ascendente**. No hay forma de forzar el range() para crear un conjunto en una forma diferente. Esto significa que el segundo argumento de range() debe ser mayor que el primero.

Las sentencias break y continue

Hasta ahora, hemos tratado el cuerpo del bucle como una secuencia indivisible e inseparable de instrucciones que se realizan completamente en cada giro del bucle. Sin embargo, como desarrollador, podrías enfrentar las siguientes opciones:

- parece que no es necesario continuar el bucle en su totalidad; se debe abstener de seguir ejecutando el cuerpo del bucle e ir más allá.
- parece que necesitas comenzar el siguiente giro del bucle sin completar la ejecución del turno actual.

Python proporciona dos instrucciones especiales para la implementación de estas dos tareas.

Estas dos instrucciones son:

- **break** - sale del bucle inmediatamente, e incondicionalmente termina la operación del bucle; el programa comienza a ejecutar la instrucción más cercana después del cuerpo del bucle.
- **continue** - se comporta como si el programa hubiera llegado repentinamente al final del cuerpo; el siguiente turno se inicia y la expresión de condición se prueba de inmediato.

Ambas palabras son **palabras clave reservadas**.

Ahora te mostraremos dos ejemplos simples para ilustrar como funcionan las dos instrucciones.

```
# break - ejemplo

print("La instrucción break:")
for i in range(1, 6):
    if i == 3:
        break
    print("Dentro del bucle.", i)
print("Fuera del bucle.")
```

```
Console >_

La instrucción break:
Dentro del bucle. 1
Dentro del bucle. 2
Fuera del bucle.
```

```python
# continue - ejemplo

print("\nLa instrucción continue:")
for i in range(1, 6):
    if i == 3:
        continue
    print("Dentro del bucle.", i)
print("Fuera del bucle.")
```

```
Console >_
La instrucción continue:
Dentro del bucle. 1
Dentro del bucle. 2
Dentro del bucle. 4
```

```
Console >_
Dentro del bucle. 2
Dentro del bucle. 4
Dentro del bucle. 5
Fuera del bucle.
```

Operaciones lógicas en Python

Aprenderás sobre los operadores lógicos en Python, conceptos como las tablas de verdad.

Lógica de computadoras

Te has dado cuenta de que las condiciones que hemos usado hasta ahora han sido muy simples, por no decir, bastante primitivas. Las condiciones que utilizamos en la vida real son mucho más complejas. Veamos este enunciado:

Si tenemos tiempo libre y el clima es bueno, saldremos a caminar.

Hemos utilizado **la conjunción and (y)**, lo que significa que salir a caminar depende del cumplimiento simultáneo de estas **dos condiciones**. En el lenguaje de la lógica, tal conexión de condiciones se denomina **conjunción**. Y ahora otro ejemplo:

Si tu estás en el centro comercial o yo estoy en el centro comercial, uno de nosotros le comprará un regalo a mamá.

La aparición de la **palabra or (o)** significa que la compra depende de al menos una de estas condiciones. En lógica, este compuesto se llama una **disyunción**.

Está claro que Python debe tener operadores para construir conjunciones y disyunciones. Sin ellos, el poder expresivo del lenguaje se debilitaría sustancialmente. Se llaman **operadores lógicos**.

El operador and

Un operador de **conjunción** lógica en Python es la palabra **and**. Es un operador binario **con una prioridad inferior a la expresada por los operadores de comparación**. Nos permite codificar condiciones complejas sin el uso de paréntesis como este:

```
counter > 0 and value == 100
```

El resultado proporcionado por el operador **and** se puede determinar sobre la base de **la tabla de verdad**.

Si consideramos la conjunción de **A and B**, el conjunto de valores posibles de argumentos y los valores correspondientes de conjunción se ve de la siguiente manera:

Argumento A	Argumento B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

El operador or

Un operador de **disyunción** es la palabra **or**. Es un operador binario con **una prioridad más baja que and (al igual que + en comparación con *)**. Su tabla de verdad es la siguiente:

Argumento A	Argumento B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

El operador not

Además, hay otro operador que se puede aplicar para condiciones de construcción. Es **un operador unario que realiza una negación lógica**. Su funcionamiento es simple: convierte la verdad en falso y lo falso en verdad.

Este operador se escribe como la palabra **not**, y **su prioridad es muy alta**: igual que el unario + y -. Su tabla de verdad es simple:

Argumento	not Argumento
False	True
True	False

Listas

Aquí aprenderás sobre las listas de Python y cómo realizar varias operaciones en ellas. Aprenderás a cómo indexar, actualizar y eliminar elementos de lista, cómo crear rodajas y cómo usar algunas de las funciones y métodos de lista más importantes.

¿Por qué necesitamos listas?

Puede suceder que tengas que leer, almacenar, procesar y, finalmente, imprimir docenas, quizás cientos, tal vez incluso miles de números. ¿Entonces qué? ¿Necesitas crear una variable separada para cada valor? ¿Tendrás que pasar largas horas escribiendo sentencias como la que se muestra a continuación?

```
var1 = int(input())
var2 = int(input())
var3 = int(input())
var4 = int(input())
var5 = int(input())
var6 = int(input())
:
:
```

Hasta ahora, has aprendido como declarar variables que pueden almacenar exactamente un valor dado a la vez. Tales variables a veces se denominan **escalares** por analogía con las matemáticas. Todas las variables que has usado hasta ahora son realmente escalares.

Piensa en lo conveniente que sería declarar una variable que podría almacenar más de un valor. Por ejemplo, cien, o mil o incluso diez mil. Todavía sería una y la misma variable, pero muy amplia y espaciosa. ¿Suena atractivo? Quizás, pero ¿cómo manejarías un contenedor así lleno de valores diferentes? ¿Cómo elegirías solo el que necesitas?

Te mostraremos como declarar tales **variables de múltiples valores**. Haremos esto con un ejemplo, escribiremos un programa que **ordene una secuencia de números**. No seremos particularmente ambiciosos - asumiremos que hay exactamente cinco números.

Vamos a crear una variable llamada **numbers**; se le asigna no solo un número, sino que se llena con una lista que consta de cinco valores (nota: **la lista comienza con un corchete abierto y termina con un corchete cerrado; el espacio entre los corchetes es llenado con cinco números separados por comas**).

```
numbers = [10, 5, 7, 2, 1]
```

Digamos lo mismo utilizando una terminología adecuada: **numbers es una lista que consta de cinco valores, todos ellos números**. También podemos decir que esta sentencia crea una lista de longitud igual a cinco (ya que contiene cinco elementos).

Los elementos dentro de una lista **pueden tener diferentes tipos**. Algunos de ellos pueden ser enteros, otros son flotantes y otros pueden ser listas.

Python ha adoptado una convención que indica que los elementos de una lista están siempre **numerados desde cero.** Esto significa que el elemento almacenado al principio de la lista tendrá el número cero. Como hay cinco elementos en nuestra lista, al último de ellos se le asigna el número cuatro. No olvides esto.

Antes de continuar con nuestra discusión, debemos indicar lo siguiente: **nuestra lista es una colección de elementos, pero cada elemento es un escalar**.

Indexación de listas

¿Cómo cambias el valor de un elemento elegido en la lista?

Vamos a asignar un nuevo valor de 111 al primer elemento en la lista. Lo hacemos de esta manera:

```
1  numbers = [10, 5, 7, 2, 1]
2  print("Contenido de la lista:", numbers)  # Imprimiendo contenido de la lista original.
3
4  numbers[0] = 111
5  print("Nuevo contenido de la lista: ", numbers)  # Contenido actual de la lista.
6
```

Y ahora queremos copiar el valor del quinto elemento al segundo elemento - ¿puedes adivinar cómo hacerlo?

```
1  numbers = [10, 5, 7, 2, 1]
2  print("Contenido de la lista original:", numbers)  # Imprimiendo contenido de la lista original.
3
4  numbers[0] = 111
5  print("\nPrevio contenido de la lista:", numbers)  # Imprimiendo contenido de la lista anterior.
6
7  numbers[1] = numbers[4]  # Copiando el valor del quinto elemento al segundo elemento.
8  print("Nuevo contenido de la lista:", numbers)  # Imprimiendo el contenido de la lista actual.
9
```

```
Console >_
Contenido de la lista original: [10, 5, 7, 2, 1]

Previo contenido de la lista: [111, 5, 7, 2, 1]
Nuevo contenido de la lista: [111, 1, 7, 2, 1]
```

El valor dentro de los corchetes que selecciona un elemento de la lista se llama un **índice**, mientras que **la operación de seleccionar un elemento de la lista se conoce como indexación**.

Vamos a utilizar la función **print**() para imprimir el contenido de la lista cada vez que realicemos los cambios. Esto nos ayudará a seguir cada paso con más cuidado y ver que sucede después de una modificación de la lista en particular.

Nota: todos los índices utilizados hasta ahora son literales. Sus valores se fijan en el tiempo de ejecución, pero cualquier expresión también puede ser un índice. Esto abre muchas posibilidades.

Acceso al contenido de las listas

Se puede acceder a cada uno de los elementos de la lista por separado. Por ejemplo, se puede imprimir:

```
1  print(numbers[0]) # Accediendo al primer elemento de la lista.
2
```

```
1  numbers = [10, 5, 7, 2, 1]
2  print("Contenido de la lista original:", numbers)  # Imprimiendo el contenido de la lista original.
3
4  numbers[0] = 111
5  print("\nContenido de la lista con cambio:", numbers)  # Imprimiendo contenido de la lista con 111.
6
7  numbers[1] = numbers[4]  # Copiando el valor del quinto elemento al segundo elemento.
8  print("Contenido de la lista con intercambio:", numbers)  # Imprimiendo contenido de la lista con intercambio.
9
10 print("\nLongitud de la lista:", len(numbers))  # Imprimiendo la longitud de la lista.
11
```

La función len()

La longitud de una lista puede variar durante la ejecución. Se pueden agregar nuevos elementos a la lista, mientras que otros pueden eliminarse de ella. Esto significa que la lista es una entidad muy dinámica.

Si deseas verificar la longitud actual de la lista, puedes usar una función llamada **len()** (su nombre proviene de length - longitud).

La función toma el nombre de la lista como un argumento y devuelve el número de elementos almacenados actualmente dentro de la lista (en otras palabras - la longitud de la lista).
Observa la última línea de código en el editor, ejecuta el programa y verifica que valor imprimirá en la consola. ¿Puedes adivinar?

```
numbers = [10, 5, 7, 2, 1]
print("Contenido de la lista original:", numbers)  # Imprimiendo el contenido de la lista original.

print("\nLongitud de la lista:", len(numbers))  # Imprimiendo la longitud de la lista.

```

```
Console >_

Contenido de la lista original: [10, 5, 7, 2, 1]


Longitud de la lista: 5
```

Eliminando elementos de una lista

Cualquier elemento de la lista puede ser eliminado en cualquier momento - esto se hace con una instrucción llamada **del** (eliminar). Nota: **es una instrucción, no una función.**

Tienes que apuntar al elemento que quieres eliminar - desaparecerá de la lista y la longitud de la lista se reducirá en uno.

Mira el fragmento de abajo. ¿Puedes adivinar qué output producirá? Ejecuta el programa en el editor y comprueba.

```
numbers = [10, 5, 7, 2, 1]
del numbers[1]
print(len(numbers))
print(numbers)
```

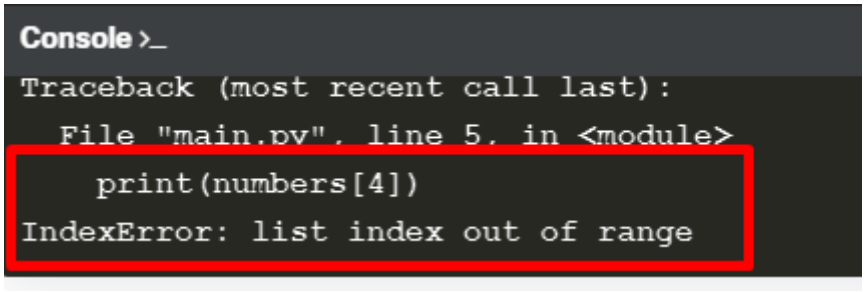
```
Console >_

4
[10, 7, 2, 1]
```

No puedes acceder a un elemento que no existe - no puedes obtener su valor ni asignarle un valor. Ambas instrucciones causarán ahora errores de tiempo de ejecución: comprueba.

```
numbers = [10, 5, 7, 2, 1]
del numbers[1]
print(len(numbers))
print(numbers)
print(numbers[4])

```



Los índices negativos son legales

Puede parecer extraño, pero los índices negativos son válidos, y pueden ser muy útiles.

Un elemento con un índice igual a **-1 es el último en la lista**.

```
1  numbers = [111, 7, 2, 1]
2  print(numbers[-1])
3
```

```
Console >_
1
```

Del mismo modo, el elemento con un índice igual a -2 es el anterior al último en la lista.
El fragmento de ejemplo dará como output un 2.

Funciones vs métodos

Un método **es un tipo específico de función** - se comporta como una función y se parece a una función, pero difiere en la forma en que actúa y en su estilo de invocación.

Una función **no pertenece a ningún dato** - obtiene datos, puede crear nuevos datos y (generalmente) produce un resultado.

Un método hace todas estas cosas, **pero también puede cambiar el estado de una entidad seleccionada.**

Un método es propiedad de los datos para los que trabaja, mientras que una función es propiedad de todo el código.

Esto también significa que invocar un método requiere alguna especificación de los datos a partir de los cuales se invoca el método.

En general, una invocación de función típica puede tener este aspecto:

```
result = function(arg)
```

La función toma un argumento, hace algo, y devuelve un resultado.

Una invocación de un método típico usualmente se ve así:

```
result = data.method(arg)
```

Nota: el nombre del método está precedido por el nombre de los datos que posee el método. A continuación, se agrega un **punto**, seguido del **nombre del método** y un par **de paréntesis que encierran los argumentos**.

El método se comportará como una función, pero puede hacer algo más - puede cambiar el estado interno de los datos a partir de los cuales se ha invocado.

Puedes preguntar: ¿por qué estamos hablando de métodos, y no de listas?

Este es un tema esencial en este momento, ya que le mostraremos como agregar nuevos elementos a una lista existente. Esto se puede hacer con métodos propios de las listas, no por funciones.

Agregando elementos a una lista: append() y insert()

Un nuevo elemento puede ser añadido al final de la lista existente:

```
list.append(value)
```

Dicha operación se realiza mediante un método llamado **append()**. Toma el valor de su argumento y lo coloca al **final de la lista** que posee el método.

La longitud de la lista aumenta en uno.

El método **insert()** es un poco más inteligente - puede agregar un nuevo elemento en **cualquier lugar de la lista**, no solo al final.

```
list.insert(location, value)
```

Toma dos argumentos:

- el primero muestra la ubicación requerida del elemento a insertar; nota: todos los elementos existentes que ocupan ubicaciones a la derecha del nuevo elemento (incluido el que está en la posición indicada) se desplazan a la derecha, para hacer espacio para el nuevo elemento;

- el segundo es el elemento a insertar.

Observa el código en el editor. Ve como usamos los métodos append() e insert(). Presta atención a lo que sucede después de usar insert(): el primer elemento anterior ahora es el segundo, el segundo el tercero, y así sucesivamente.

```
1  numbers = [111, 7, 2, 1]
2  print(len(numbers))
3  print(numbers)
4
5  ###
6  numbers.append(4)
7
8  print(len(numbers))
9  print(numbers)
10
```

```
Console >_
4
[111, 7, 2, 1]
5
[111, 7, 2, 1, 4]
```

```
1  numbers = [111, 7, 2, 1]
2  print(len(numbers))
3  print(numbers)
4
5  ###
6  numbers.insert(0, 222)
7  print(len(numbers))
8  print(numbers)
9
```

```
Console >_
4
[111, 7, 2, 1]
5
[222, 111, 7, 2, 1]
```

Puedes **iniciar la vida de una lista creándola vacía (esto se hace con un par de corchetes vacíos)** y luego agregar nuevos elementos según sea necesario.

Echa un vistazo al fragmento en el editor. Intenta adivinar su output después de la ejecución del bucle for. Ejecuta el programa para comprobar si tenías razón.

```
1  my_list = []  # Creando una lista vacía.
2
3  for i in range(5):
4      my_list.append(i + 1)
5
6  print(my_list)
7
```

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

Será una secuencia de números enteros consecutivos del 1 (luego agrega uno a todos los valores agregados) hasta 5.

Hemos modificado un poco el fragmento:

```
my_list = []  # Creando una lista vacía.

for i in range(5):
    my_list.insert(0, i + 1)

print(my_list)
```

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

¿Qué pasará ahora? Ejecuta el programa y comprueba.

Deberías obtener la misma secuencia, pero en orden inverso (este es el mérito de usar el método insert()).

Haciendo uso de las listas

El bucle for tiene una variante muy especial que puede procesar las listas de manera muy efectiva

```
my_list = [10, 1, 8, 3, 5]
total = 0

for i in range(len(my_list)):
    total += my_list[i]

print(total)
```

```
Console >_
27
```

Supongamos que deseas calcular la suma de todos los valores almacenados en la lista my_list.

Necesitas una variable cuya suma se almacenará y se le asignará inicialmente un valor de 0 - su nombre será total. (Nota: no la vamos a nombrar sum ya que Python utiliza el mismo nombre para una de sus funciones integradas: sum(). Utilizar ese nombre sería considerado una mala práctica.) Luego agrega todos los elementos de la lista usando el bucle for. Echa un vistazo al fragmento en el editor.

Comentemos este ejemplo:

- a la lista se le asigna una secuencia de cinco valores enteros
- la variable i toma los valores 0, 1,2,3, y 4, y luego indexa la lista, seleccionando los elementos siguientes: el primero, segundo, tercero, cuarto y quinto;
- cada uno de estos elementos se agrega junto con el operador += a la variable suma, dando el resultado final al final del bucle;
- observa la forma en que se ha empleado la función **len()** - hace que el código sea independiente de cualquier posible cambio en el contenido de la lista.

El segundo aspecto del bucle for

Pero el bucle **for** puede hacer mucho más. Puede ocultar todas las acciones conectadas a la indexación de la lista y entregar todos los elementos de la lista de manera práctica.

Este fragmento modificado muestra como funciona:

```
1  my_list = [10, 1, 8, 3, 5]
2  total = 0
3
4  for i in my_list:
5      total += i
6
7  print(total)
```

```
Console >_
27
```

¿Qué sucede aquí?

- la instrucción **for** especifica la variable utilizada para navegar por la lista (i) seguida de la palabra clave **in** y el nombre de la lista siendo procesado (**my_list**).
- a la variable **i** se le asignan los valores de todos los elementos de la lista subsiguiente, y el proceso ocurre tantas veces como hay elementos en la lista;
- esto significa que usa la variable **i** como una copia de los valores de los elementos, y no necesita emplear índices;
- la función **len()** tampoco es necesaria aquí.

Ordenando listas simples: el ordenamiento de burbuja

Aprenderás a cómo ordenar listas simples utilizando el algoritmo de ordenamiento burbuja.

Ordenamiento Burbuja

Ahora que puedes hacer malabarismos con los elementos de las listas, es hora de aprender como **ordenarlos**.

Digamos que una lista se puede ordenar de dos maneras:

- **ascendente** (o más precisamente - no descendente) - si en cada par de elementos adyacentes, el primer elemento no es mayor que el segundo;
- **descendente** (o más precisamente - no ascendente) - si en cada par de elementos adyacentes, el primer elemento no es menor que el segundo.

En las siguientes secciones, ordenaremos la lista en orden ascendente, de modo que los números se ordenen de menor a mayor.

Aquí está la lista:

8	10	6	2	4

Intentaremos utilizar el siguiente enfoque: tomaremos el primer y el segundo elemento y los compararemos; si determinamos que están en el orden incorrecto (es decir, el primero es mayor que el segundo), los intercambiaremos; si su orden es válido, no haremos nada. Un vistazo a nuestra lista confirma lo último - los elementos 01 y 02 están en el orden correcto, así como 8<10.

Ahora observa el segundo y el tercer elemento. Están en las posiciones equivocadas. Tenemos que intercambiarlos:

8	**6**	**10**	2	4

Vamos más allá y observemos los elementos tercero y cuarto. Una vez más, esto no es lo que se supone que es. Tenemos que intercambiarlos:

8	6	**2**	**10**	4

Ahora comprobemos los elementos cuarto y quinto. Si, ellos también están en las posiciones equivocadas. Ocurre otro intercambio:

8	6	2	**4**	**10**

El primer paso a través de la lista ya está terminado. Todavía estamos lejos de terminar nuestro trabajo, pero algo curioso ha sucedido mientras tanto. El elemento más grande, 10, ya ha llegado al final de la lista. Ten en cuenta que este es el lugar deseado para el. Todos los elementos restantes forman un lío pintoresco, pero este ya está en su lugar.

Ahora, por un momento, intenta imaginar la lista de una manera ligeramente diferente - es decir, de esta manera:

10
4
2
6
8

Observa - El **10** está en la parte superior. Podríamos decir que flotó desde el fondo hasta la superficie, al igual que las burbujas en una copa de champán. El método de clasificación deriva su nombre de la misma observación - se denomina ordenamiento burbuja.
Ahora comenzamos con el segundo paso a través de la lista. Miramos el primer y el segundo elemento - es necesario un intercambio:

6	**8**	2	4	10

Es hora del segundo y tercer elemento: también tenemos que intercambiarlos:

6	**2**	**8**	4	10

Ahora el tercer y cuarto elementos, y la segunda pasada, se completa, ya que **8** ya está en su lugar:

6	2	**4**	**8**	10

Comenzamos el siguiente pase inmediatamente. Observa atentamente el primer y el segundo elemento - se necesita otro cambio:

2	**6**	4	8	10

Ahora 6 necesita ir a su lugar. Cambiamos el segundo y el tercer elemento:

2	**4**	**6**	8	10

La lista ya está ordenada. No tenemos nada más que hacer. Esto es exactamente lo que queremos.

Como puedes ver, la esencia de este algoritmo es simple: comparamos los elementos adyacentes y, al intercambiar algunos de ellos, logramos nuestro objetivo.

Codifiquemos en Python todas las acciones realizadas durante un solo paso a través de la lista, y luego consideraremos cuántos pases necesitamos para realizarlo. No hemos explicado esto hasta ahora, pero lo haremos pronto.

Ordenando una lista

¿Cuántos pases necesitamos para ordenar la lista completa?

Resolvamos este problema de la siguiente manera: **introducimos otra variable**, su tarea es observar si se ha realizado algún intercambio durante el pase o no. Si no hay intercambio, entonces la lista ya está ordenada, y no hay que hacer nada más. Creamos una variable llamada **swapped**, y le asignamos un valor de **False** para indicar que no hay intercambios. De lo contrario, se le asignará **True**.

```
my_list = [8, 10, 6, 2, 4]  # lista a ordenar
swapped = True  # Lo necesitamos verdadero (True) para ingresar al bucle while.

while swapped:
    swapped = False  # no hay intercambios hasta ahora
    for i in range(len(my_list) - 1):
        if my_list[i] > my_list[i + 1]:
            swapped = True  # ¡ocurrió el intercambio!
            my_list[i], my_list[i + 1] = my_list[i + 1], my_list[i]

print(my_list)
```

```
Console >_

[2, 4, 6, 8, 10]
```

El ordenamiento burbuja – versión interactiva

En el editor, puedes ver un programa completo, enriquecido por una conversación con el usuario, y que permite ingresar e imprimir elementos de la lista: **El ordenamiento burbuja - versión final interactiva**.

```
my_list = []
swapped = True
num = int(input("¿Cuántos elementos deseas ordenar?: "))

for i in range(num):
    val = float(input("Ingresa un elemento de la lista: "))
    my_list.append(val)

while swapped:
    swapped = False
    for i in range(len(my_list) - 1):
        if my_list[i] > my_list[i + 1]:
            swapped = True
            my_list[i], my_list[i + 1] = my_list[i + 1], my_list[i]

print("\nOrdenada:")
print(my_list)
```

```
Console >_

¿Cuántos elementos deseas ordenar?: 3
Ingresa un elemento de la lista: 10
Ingresa un elemento de la lista: 2
Ingresa un elemento de la lista: 8
```

```
Ordenada:
[2.0, 8.0, 10.0]
```

Python, sin embargo, tiene sus propios mecanismos de clasificación. Nadie necesita escribir sus propias clases, ya que hay un número suficiente de **herramientas listas para usar**.

Te explicamos este sistema de clasificación porque es importante aprender como procesar los contenidos de una lista y mostrarte como puede funcionar la clasificación real.

Si quieres que Python ordene tu lista, puedes hacerlo de la siguiente manera:

```
1  my_list = [8, 10, 6, 2, 4]
2  my_list.sort()
3  print(my_list)
4
```

La output del fragmento es la siguiente:

```
[2, 4, 6, 8, 10]
```

Como puedes ver, todas las listas tienen un **método denominado sort()**, que las ordena lo más rápido posible. Ya has aprendido acerca de algunos de los métodos de lista anteriormente, y pronto aprenderás más sobre otros.

Puedes usar el **método sort()** para ordenar los elementos de una lista, por ejemplo:

```
1  lst = [5, 3, 1, 2, 4]
2  print(lst)
3
4  lst.sort()
5  print(lst)  # output: [1, 2, 3, 4, 5]
6
```

También hay **un método de lista llamado reverse()**, que puedes usar para invertir la lista, por ejemplo:

```
lst = [5, 3, 1, 2, 4]
print(lst)

lst.reverse()
print(lst)  # output: [4, 2, 1, 3, 5]

```

Operaciones con listas

Aprenderás a cómo procesar listas utilizando rebanadas y los operadores in y not in. También analizarás algunos programas simples que utilizan el concepto de listas para aprender a aplicarlas en proyectos más desafiantes.

La vida al interior de las listas

Ahora queremos mostrarte una característica importante y muy sorprendente de las listas, que las distingue de las variables ordinarias.

Queremos que lo memorices - ya que puede afectar tus programas futuros y causar graves problemas si se olvida o se pasa por alto.

Echa un vistazo al fragmento en el editor.

```
list_1 = [1]
list_2 = list_1
list_1[0] = 2
print(list_2)

```

```
Console >_

[2]
```

El programa:

- crea una lista de un elemento llamada **list_1**
- la asigna a una nueva lista llamada **list_2**
- cambia el único elemento de **list_1**
- imprime la **list_2**

La parte sorprendente es el hecho de que el programa mostrará como resultado: [2], no [1], que parece ser la solución obvia.

Las listas (y muchas otras entidades complejas de Python) se almacenan de diferentes maneras que las variables ordinarias (escalares).

Se podría decir que:

- el nombre de una variable ordinaria es el **nombre de su contenido**
- el nombre de una lista es el nombre de una ubicación de memoria **donde se almacena la lista**.

Lee estas dos líneas una vez más - la diferencia es esencial para comprender de que vamos a hablar a continuación.

La asignación: **list_2 = list_1** copia el nombre del arreglo no su contenido. En efecto, los dos nombres (list_1 y list_2) identifican la misma ubicación en la memoria de la computadora. Modificar uno de ellos afecta al otro, y viceversa.

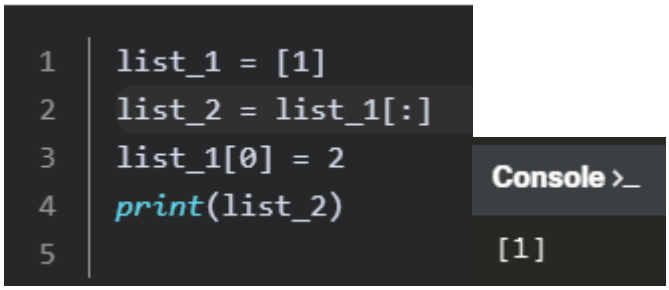
Rebanadas poderosas

Afortunadamente, la solución está al alcance de tu mano - su nombre es **rebanada**.

Una rebanada es un elemento de la sintaxis de Python que permite hacer **una copia nueva de una lista, o partes de una lista**.

En realidad, copia el contenido de la lista, no el nombre de la lista.

Esto es exactamente lo que necesitas. Echa un vistazo al fragmento de código a continuación:



Su output es **[1].**

Esta parte no visible del código descrito como **[:]** puede producir una lista completamente nueva.

Una de las formas más generales de la rebanada es la siguiente:

```
my_list[inicio:fin]
```

Como puedes ver, se asemeja a la indexación, pero los dos puntos en el interior hacen una gran diferencia.

Una rebanada de este tipo **crea una nueva lista (de destino), tomando elementos de la lista de origen - los elementos de los índices desde el principio hasta el fin fin - 1**.

Nota: no hasta el fin sino hasta fin-1. Un elemento con un índice igual a fin es el primer elemento el cual **no participa en la segmentación**.

Es posible utilizar valores negativos tanto para el inicio como para el fin(al igual que en la indexación).

Echa un vistazo al fragmento:

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[1:3]
print(new_list)

```

La lista **new_list** contendrá **fin - inicio (3 - 1 = 2) elementos** – los que tienen índices iguales a 1 y 2 (pero no 3).

```
Console >_
[8, 6]
```

La output del fragmento es: [8, 6]

Rebanadas – índices negativos

Observa el fragmento de código a continuación:

```
my_list[start:end]
```

Para confirmar:

- **start** es el índice del primer elemento **incluido en la rebanada**.
- **end** es el índice del primer elemento **no incluido en la rebanada**.

Así es como los **índices negativos** funcionan en las rebanadas:

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[1:-1]
print(new_list)

```

El resultado del fragmento es:

```
[8, 6, 4]
```

Si **start** especifica un elemento que se encuentra más allá del descrito por **end** (desde el punto de vista inicial de la lista), la rebanada estará vacía:

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[-1:1]
print(new_list)

```

La output del fragmento es:

```
[]
```

Si omites el **start** en tu rebanada, se supone que deseas obtener un segmento que comienza en el elemento con índice 0.

En otras palabras, la rebanada sería de esta forma:

```
my_list[:end]
```

es un equivalente más compacto de:

```
my_list[0:end]
```

Observa el fragmento de código a continuación:

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[:3]
print(new_list)

```

Es por esto que su output es: **[10, 8, 6].**

Del mismo modo, si omites el **end** en tu rebanada, se supone que deseas que el segmento termine en el elemento con el índice **len(my_list).**

En otras palabras, la rebanada sería de esta forma:

```
my_list[start:]
```

es un equivalente más compacto de:

```
my_list[start:len(my_list)]
```

Observa el siguiente fragmento de código:

```
1  my_list = [10, 8, 6, 4, 2]
2  new_list = my_list[3:]
3  print(new_list)
```

Por lo tanto, la output es**: [4, 2].**

Como hemos dicho anteriormente, el **omitir el inicio y fin crea una copia de toda la lista**

```
1  my_list = [10, 8, 6, 4, 2]
2  new_list = my_list[:]
3  print(new_list)
```

El resultado del fragmento es: **[10, 8, 6, 4, 2].**

Más sobre la instrucción del

La instrucción **del** descrita anteriormente **puede eliminar más de un elemento de la lista a la vez - también puede eliminar rebanadas**:

```
1  my_list = [10, 8, 6, 4, 2]
2  del my_list[1:3]
3  print(my_list)
```

Nota: En este caso, la rebanada **¡no produce ninguna lista nueva!**

La output del fragmento es: **[10, 4, 2].**

También es posible eliminar **todos los elementos** a la vez:

```
1  my_list = [10, 8, 6, 4, 2]
2  del my_list[:]
3  print(my_list)
```

La lista se queda vacía y la output es**: [].**

Al eliminar la rebanada del código, su significado cambia dramáticamente.

Echa un vistazo:

```
1  my_list = [10, 8, 6, 4, 2]
2  del my_list
3  print(my_list)
4
```

La instrucción **del eliminará la lista, no su contenido**.

La función **print()** de la última línea del código provocará un error de ejecución.

Los operadores in y not in

Python ofrece dos operadores muy poderosos, capaces de **revisar la lista para verificar si un valor específico está almacenado dentro de la lista o no.**

Estos operadores son:

```
1  elem in my_list
2  elem not in my_list
3
```

El primero de ellos **(in)** verifica si un elemento dado (el argumento izquierdo) está actualmente almacenado en algún lugar dentro de la lista (el argumento derecho) - el operador devuelve **True** en este caso.

El segundo **(not in)** comprueba si un elemento dado (el argumento izquierdo) está ausente en una lista - el operador devuelve **True** en este caso.

Observa el código en el editor. El fragmento muestra ambos operadores en acción. ¿Puedes adivinar su output? Ejecuta el programa para comprobar si tenías razón.

```
1  my_list = [0, 3, 12, 8, 2]
2
3  print(5 in my_list)
4  print(5 not in my_list)
5  print(12 in my_list)
6
```

```
Console >_
False
True
True
```

Listas - algunos programas simples

Ahora queremos mostrarte algunos programas simples que utilizan listas.

El primero de ellos intenta encontrar el valor mayor en la lista. Mira el código en el editor.

```
my_list = [17, 3, 11, 5, 1, 9, 7, 15, 13]
largest = my_list[0]

for i in range(1, len(my_list)):
    if my_list[i] > largest:
        largest = my_list[i]

print(largest)
```

```
Console >_
17
```

El concepto es bastante simple - asumimos temporalmente que el primer elemento es el más grande y comparamos la hipótesis con todos los elementos restantes de la lista.

El código da como resultado el 17 (como se espera).

El código puede ser reescrito para hacer uso de la forma recién introducida del bucle for:

```
my_list = [17, 3, 11, 5, 1, 9, 7, 15, 13]
largest = my_list[0]

for i in my_list:
    if i > largest:
        largest = i

print(largest)
```

Puedes usar una rebanada:

```
my_list = [17, 3, 11, 5, 1, 9, 7, 15, 13]
largest = my_list[0]

for i in my_list[1:]:
    if i > largest:
        largest = i

print(largest)
```

Ahora encontremos la ubicación de un elemento dado dentro de una lista:

```python
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
to_find = 5
found = False

for i in range(len(my_list)):
    found = my_list[i] == to_find
    if found:
        break

if found:
    print("Elemento encontrado en el índice", i)
else:
    print("ausente")
```

Nota:

- el valor buscado se almacena en la variable to_find;
- el estado actual de la búsqueda se almacena en la variable found (True/False).
- cuando found se convierte en True, se sale del bucle for.

Supongamos que has elegido los siguientes números en la lotería: 3, 7, 11, 42, 34, 49.
Los números que han salido sorteados son: 5, 11, 9, 42, 3, y 49.
La pregunta es: ¿A cuántos números has adivinado ?

Este programa te dará la respuesta:

```python
drawn = [5, 11, 9, 42, 3, 49]
bets = [3, 7, 11, 42, 34, 49]
hits = 0

for number in bets:
    if number in drawn:
        hits += 1

print(hits)
```

```
Console >_
4
```

Nota:

- la lista drawn almacena todos los números sorteados
- La lista bets almacena los números con que se juega
- la variable hits cuenta tus aciertos.

La output del programa es: 4.

Listas en aplicaciones avanzadas

Aprenderemos sobre arreglos, listas anidadas (matrices) y listas por comprensión.

Listas dentro de listas

Las listas pueden constar de escalares (es decir, números) y elementos de una estructura mucho más compleja (ya has visto ejemplos como cadenas, booleanos o incluso otras listas en las lecciones del Resumen de la Sección anterior). Veamos más de cerca el caso en el que **los elementos de una lista son listas**.

A menudo encontramos estos **arreglos** en nuestras vidas. Probablemente el mejor ejemplo de esto sea un tablero de ajedrez.

Un tablero de ajedrez está compuesto de filas y columnas. Hay ocho filas y ocho columnas. Cada columna está marcada con las letras de la A a la H. Cada línea está marcada con un número del uno al ocho.

La ubicación de cada campo se identifica por pares de letras y dígitos. Por lo tanto, sabemos que la esquina inferior derecha del tablero (la que tiene la torre blanca) es A1, mientras que la esquina opuesta es H8.

Supongamos que podemos usar los números seleccionados para representar cualquier pieza de ajedrez. También podemos asumir que cada fila en el tablero de ajedrez es una lista.

Observa el siguiente código:

```
1  row = []
2
3  for i in range(8):
4      row.append(WHITE_PAWN)
5
```

Crea una lista que contiene ocho elementos que representan la segunda fila del tablero de ajedrez: la que está llena de peones (supon que WHITE_PAWN es un **símbolo predefinido** que representa un peón blanco).

Comprensión de lista

El mismo efecto se puede lograr mediante una **comprensión de lista**, la sintaxis especial utilizada por Python para completar o llenar listas masivas.

Una comprensión de lista es en realidad una lista, pero **se creó sobre la marcha durante la ejecución del programa, y no se describe de forma estática.**

Echa un vistazo al fragmento:

```
row = [WHITE_PAWN for i in range(8)]

```

La parte del código colocada dentro de los paréntesis especifica:

- los datos que se utilizarán para completar la lista (**WHITE_PAWN**)
- la cláusula que especifica cuántas veces se producen los datos dentro de la lista (**for i in range(8)**)

Otros ejemplos de comprensión de lista:

Ejemplo 1

```
squares = [x ** 2 for x in range(10)]

```

El fragmento de código genera una lista de diez elementos y la rellena con cuadrados de diez números enteros que comienzan desde cero (0, 1, 4, 9, 16, 25, 36, 49, 64, 81)

Ejemplo 2:

```
twos = [2 ** i for i in range(8)]

```

El fragmento crea un arreglo de ocho elementos que contiene las primeras ocho potencias del numero dos (1, 2, 4, 8, 16, 32, 64, 128)

Ejemplo 3:

```
odds = [x for x in squares if x % 2 != 0 ]

```

El fragmento crea una lista con solo los elementos impares de la lista squares.

Arreglos de dos dimensiones

Supongamos también que un símbolo predefinido denominado EMPTY designa un campo vacío en el tablero de ajedrez.

Entonces, si queremos crear una lista de listas que representan todo el tablero de ajedrez, se puede hacer de la siguiente manera:

```
board = []

for i in range(8):
    row = [EMPTY for i in range(8)]
    board.append(row)

```

Nota:

- la parte interior del bucle crea una fila que consta de ocho elementos (cada uno de ellos es igual a EMPTY) y lo agrega a la lista del board
- la parte exterior se repite ocho veces
- en total, la lista board consta de 64 elementos (todos iguales a EMPTY).

Este modelo imita perfectamente el tablero de ajedrez real, que en realidad es una lista de elementos de ocho elementos, todos ellos en filas individuales. Resumamos nuestras observaciones:

- los elementos de las filas son campos, ocho de ellos por fila;
- los elementos del tablero de ajedrez son filas, ocho de ellos por tablero de ajedrez.

La variable board ahora es un arreglo **bidimensional**. También se le llama, por analogía a los términos algebraicos, una **matriz**.

Como las listas de comprensión puede ser **anidadas**, podemos acortar la creación del tablero de la siguiente manera:

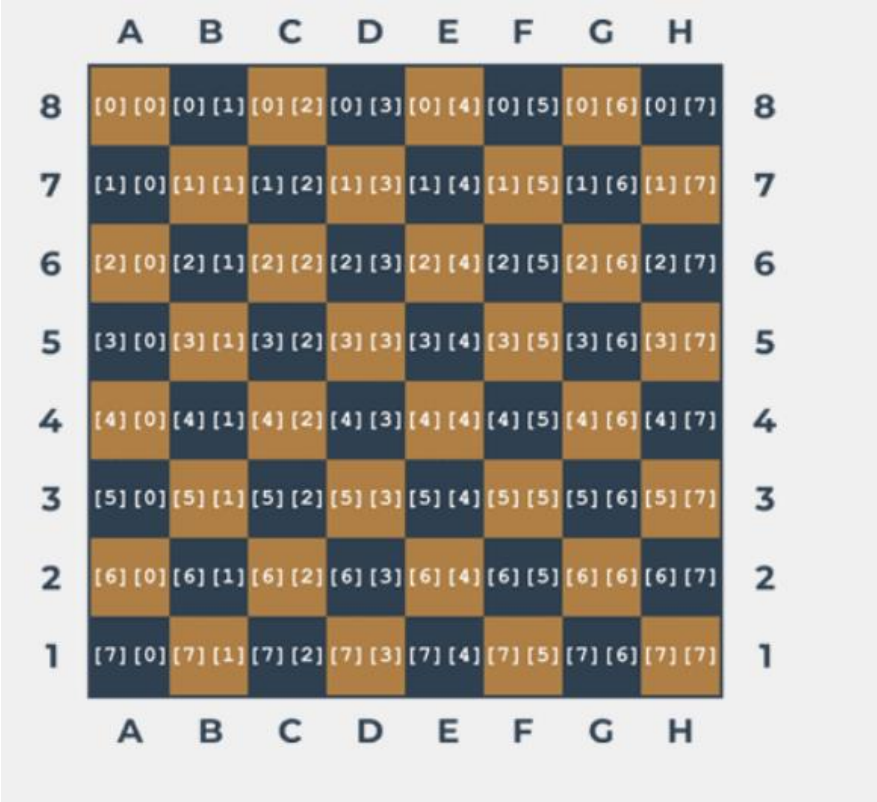
```
board = [[EMPTY for i in range(8)] for j in range(8)]

```

La parte interna crea una fila, y la parte externa crea una lista de filas.

El acceso al campo seleccionado del tablero requiere dos índices - el primero selecciona la fila; el segundo - el número del campo dentro de la fila, el cual es un número de columna.

Echa un vistazo al tablero de ajedrez. Cada campo contiene un par de índices que se deben dar para acceder al contenido del campo



Echando un vistazo a la figura que se muestra arriba, coloquemos algunas piezas de ajedrez en el tablero. Primero, agreguemos todas las torres:

```
board[0][0] = ROOK
board[0][7] = ROOK
board[7][0] = ROOK
board[7][7] = ROOK

```

Si deseas agregar un caballo a C4, hazlo de la siguiente manera:

```
board[4][2] = KNIGHT

```

Y ahora un peón a E5:

```
board[3][4] = PAWN

```

Naturaleza multidimensional de las listas: aplicaciones avanzadas

Profundicemos en la naturaleza multidimensional de las listas. Para encontrar cualquier elemento de una lista bidimensional, debes usar dos coordenadas:

- Una vertical (número de fila).

- Una horizontal (número de columna).

Imagina que desarrollas una pieza de software para una estación meteorológica automática. El dispositivo registra la temperatura del aire cada hora y lo hace durante todo el mes. Esto te da un total de 24 × 31 = 744 valores. Intentemos diseñar una lista capaz de almacenar todos estos resultados.

Primero, debes decidir qué tipo de datos sería adecuado para esta aplicación. En este caso, sería mejor un float, ya que este termómetro puede medir la temperatura con una precisión de 0.1 ℃.

Luego tomarás la decisión arbitraria de que las filas registrarán las lecturas cada hora exactamente (por lo que la fila tendrá 24 elementos) y cada una de las filas se asignará a un día del mes (supongamos que cada mes tiene 31 días, por lo que necesita 31 filas). Aquí está el par apropiado de comprensiones (**h** es para las horas, **d** para el día):

```
1  temps = [[0.0 for h in range(24)] for d in range(31)]
2
```

Toda la matriz está llena de ceros ahora. Puede suponer que se actualiza automáticamente utilizando agentes de hardware especiales. Lo que tienes que hacer es esperar a que la matriz se llene con las mediciones.

Ahora es el momento de determinar la temperatura promedio mensual del mediodía. Suma las 31 lecturas registradas al mediodía y divida la suma por 31. Puedes suponer que la temperatura de medianoche se almacena primero. Aquí está el código:

```
1  temps = [[0.0 for h in range(24)] for d in range(31)]
2  #
3  # La matriz se actualiza aquí.
4  #
5
6  total = 0.0
7
8  for day in temps:
9      total += day[11]
10
11 average = total / 31
12
13 print("Temperatura promedio al mediodía:", average)
14
```

Nota: La variable day utilizada por el bucle for no es un escalar - cada paso a través de la matriz temps lo asigna a la siguiente fila de la matriz; Por lo tanto, es una lista. Se debe indexar con 11 para acceder al valor de temperatura medida al mediodía.

Ahora encuentra la temperatura más alta durante todo el mes - analiza el código:

```
temps = [[0.0 for h in range(24)] for d in range(31)]
#
# La matriz se actualiza aquí.
#

highest = -100.0

for day in temps:
    for temp in day:
        if temp > highest:
            highest = temp

print("La temperatura más alta fue:", highest)

```

Note:

- la variable day itera en todas las filas de la matriz temps;
- la variable temp itera a través de todas las mediciones tomadas en un día.

Ahora cuenta los días en que la temperatura al mediodía fue de al menos 20 ℃:

```
temps = [[0.0 for h in range(24)] for d in range(31)]
#
# La matriz se actualiza aquí.
#

hot_days = 0

for day in temps:
    if day[11] > 20.0:
        hot_days += 1

print(hot_days, "fueron los días calurosos.")

```

Python no limita la profundidad de la inclusión lista en lista. Aquí puedes ver un ejemplo de un arreglo tridimensional:

```
rooms = [[[False for r in range(20)] for f in range(15)] for t in range(3)]

```

Imagina un hotel. Es un hotel enorme que consta de tres edificios, de 15 pisos cada uno. Hay 20 habitaciones en cada piso. Para esto, necesitas un arreglo que pueda recopilar y procesar información sobre las habitaciones ocupadas/libres.

Primer paso - El tipo de elementos del arreglo. En este caso, sería un valor Booleano (True/False).

Paso dos - análisis de la situación. Resume la información disponible: tres edificios, 15 pisos, 20 habitaciones.

Ahora puedes crear el arreglo:

```
rooms = [[[False for r in range(20)] for f in range(15)] for t in range(3)]

```

El primer índice (0 a 2) selecciona uno de los edificios; el segundo (0 a 14) selecciona el piso, el tercero (0 a 19) selecciona el número de habitación. Todas las habitaciones están inicialmente desocupadas.

Ahora ya puedes reservar una habitación para dos recién casados: en el segundo edificio, en el décimo piso, habitación 14:

```
rooms[1][9][13] = True

```

y desocupar el segundo cuarto en el quinto piso ubicado en el primer edificio:

```
rooms[0][4][1] = False

```

Verifica si hay disponibilidad en el piso 15 del tercer edificio:

```
vacancy = 0

for room_number in range(20):
    if not rooms[2][14][room_number]:
        vacancy += 1

```

La variable vacancy contiene 0 si todas las habitaciones están ocupadas, o en dado caso el número de habitaciones disponibles.