1. ¿Qué es un condicional?

1.1 Sintaxis de declaración condicional

Si no fuera por las instrucciones condicionales, los programas tendrían una estructura lineal: todas las instrucciones se ejecutaron secuencialmente una tras otra, cada instrucción escrita debe ejecutarse.

Digamos que queremos determinar su valor absoluto (módulo) a partir de un número x dado. El programa debe imprimir el valor de la variable x si x>0 o el valor -x en caso contrario. Se viola la estructura lineal del programa: dependiendo de la validez de la condición x>0, se debe emitir uno u otro valor. El fragmento correspondiente del programa Python tiene el siguiente aspecto:

```
x = int(input())
if x > 0:
    print(x)              # print x
else:
    print(-x)             # print -x
```

Este programa utiliza una declaración condicional if. Después de la palabra if, se indica la condición que se está probando (x > 0), terminando con dos puntos. Después de esto viene un bloque (secuencia) de instrucciones que se ejecutarán si la condición es verdadera; en nuestro ejemplo, esto muestra el valor x. Luego viene la palabra else (de lo contrario), que también termina con dos puntos, y un bloque de instrucciones que se ejecutarán si la condición que se está probando es falsa, en este caso se imprimirá el valor -x.

Entonces, una declaración condicional en Python tiene la siguiente sintaxis:

```
if condicional:
    block 1
else:
    block 2
```

El bloque de instrucciones 1 se ejecutará si la condición es verdadera. Si la condición es falsa, se ejecutará el bloque de instrucciones 2.

Una declaración condicional puede carecer de la palabra else y de un bloque posterior. Esta instrucción se llama ramificación incompleta. Por ejemplo, si dado un número x y queremos reemplazarlo con el valor absoluto de x, esto se puede hacer de la siguiente manera:

```
x = int(input())
if x < 0:
    x = -x
print(x)                  # print abs(x)
```

En este ejemplo, a la variable x se le asignará el valor -x, pero solo si x<0. Pero la instrucción print(x) siempre se ejecutará, independientemente de la condición que se esté verificando.

Python usa sangría para resaltar un bloque de instrucciones relacionadas con una declaración if o else. Todas las instrucciones que pertenecen al mismo bloque deben tener la misma cantidad de sangría, es decir, la misma cantidad de espacios al principio de la línea. Se recomienda utilizar una sangría de 4 espacios y no se recomienda utilizar un carácter de tabulación como sangría.

1.2 Declaraciones condicionales anidadas

Dentro de las instrucciones condicionales, puede utilizar cualquier instrucción del lenguaje Python, incluidas las instrucciones condicionales. Obtenemos una rama anidada: después de una bifurcación, aparece otra bifurcación durante la ejecución del programa. En este caso, los bloques anidados tienen un tamaño de sangría mayor (por ejemplo, 8 espacios). Demostremos esto usando el ejemplo de un programa que, dados números x e y distintos de cero, determina en qué cuarto del plano de coordenadas se encuentra el punto (x,y):

```
x = int(input())
y = int(input())
if x > 0:
    if y > 0:               # x > 0, y > 0
        print("Primer trimestre")
    else:                   # x > 0, y < 0
        print("Cuarto trimestre")
else:
    if y > 0:               # x < 0, y > 0
        print("Segundo trimestre")
    else:                   # x < 0, y < 0
        print("Tercer trimestre")
```

En este ejemplo, utilizamos comentarios: texto que el intérprete ignora. Los comentarios en Python son el carácter # y todo el texto después de ese carácter hasta el final de la línea.

1.3 Operadores de comparación

Normalmente, el resultado de uno de los siguientes operadores de comparación se utiliza como condición que se prueba:

<
Menor que: la condición es verdadera si el primer operando es menor que el segundo.
>
Mayor que: la condición es verdadera si el primer operando es mayor que el segundo.
<=
Menos o igual.
>=
Más o igual.
==
Igualdad. La condición es verdadera si los dos operandos son iguales.
!=

Desigualdad. La condición es verdadera si los dos operandos son desiguales.
Por ejemplo, la condición (x * x < 1000) significa “el valor de x * x es menor que 1000” y la condición (2 * x != y) significa “el doble del valor de x no es igual al valor de y."

Los operadores de comparación en Python se pueden combinar en cadenas (a diferencia de la mayoría de los otros lenguajes de programación, donde es necesario usar conectivos lógicos para esto), por ejemplo, a == b == c o 1 <= x <= 10.

1.4 Tipo de datos booleano

Los operadores de comparación devuelven valores del tipo booleano especial bool. Los valores booleanos pueden tomar uno de dos valores: True(Verdadero) o False(Falso). La conversión de un booleano True(Verdadero) a un int dará como resultado 1, y la conversión a False(Falso) dará como resultado un 0. La conversión inversa convertirá el número 0 en False(Falso) y cualquier número

distinto de cero en True(Verdadero). Al convertir str a bool, la cadena vacía se convierte en False y cualquier cadena que no esté vacía se convierte en True.

1.5 Operadores logicos

A veces es necesario comprobar no una, sino varias condiciones al mismo tiempo. Por ejemplo, puedes verificar si un número dado es par usando la condición (n % 2 == 0) (el resto de n dividido por 2 es 0), y si necesitas verificar que dos números enteros n y m dados sean pares , es necesario comprobar la validez de ambas condiciones: n % 2 == 0 y m % 2 == 0, para lo cual se deben combinar utilizando el operador and (Y lógico): n % 2 == 0 and m % 2 == 0.

Hay operadores lógicos estándar en Python: Y lógico, O lógico, negación lógica.

El AND lógico es un operador binario (es decir, un operador con dos operandos: izquierdo y derecho) y tiene la forma y. El operador and devuelve Verdadero si y sólo si ambos operandos son Verdaderos.

El OR lógico es un operador binario y devuelve Verdadero si y sólo si al menos un operando es Verdadero. El operador “lógico OR” tiene la forma o.

El NOT lógico (negación) es un operador unario (es decir, con un operando) y tiene la forma no seguida de un solo operando. Booleano NOT devuelve Verdadero si el operando es Falso y viceversa.

Ejemplo. Comprobemos que al menos uno de los números aob termina en 0:

```
a = int(input())
b = int(input())
if a % 10 == 0 or b % 10 == 0:
    print('YES')
else:
    print('NO')
```

Comprobemos que el número a es positivo y b no es negativo:
if a > 0 and not (b < 0):
O puedes escribir (b >= 0) en lugar de no (b < 0).

1.6 Instrucciones condicionales en cascada

Un programa de ejemplo que define un cuarto del plano de coordenadas se puede reescribir usando una secuencia en “cascada” con una operación if... elif... else:

```
x = int(input())
y = int(input())
if x > 0 and y > 0:
    print("Primer trimestre")
elif x > 0 and y < 0:
    print("Cuarto trimestre")
elif y > 0:
    print("Segundo trimestre")
else:
    print("Tercer trimestre")
```

En esta construcción se verifican una por una las condiciones if, ..., elif y se ejecuta el bloque correspondiente a la primera condición verdadera. Si todas las condiciones que se prueban son falsas, entonces se ejecuta el bloque else, si está presente.

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

Los bucles en Python son una herramienta central de desarrollo. Con su ayuda, puedes automatizar tareas, realizar varias acciones en un par de líneas de código y generar datos. Entendamos los principios del bucle en Python usando ejemplos.

2.1 ¿Qué son los bucles en Python?

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)
```

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)
```

2.2 Por 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 ciclos:
- para repetir tareas. Con los bucles, puedes llamar a funciones varias veces y los bucles mismos pueden depender de las condiciones;
- trabajando con datos. Puede 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.

2.3 Bucle While

Un 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 conditions:
    block
```

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 de contraseña 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 de contraseña.

```
password = "qwerty"
input_password = input("Enter password: ")

while input_password != password:
    print("Bad password")
    input_password = input("Enter password: ")

print("Good password")
```

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("Bucle While")
```

En el ejemplo anterior, la condición Verdadero 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.

2.4 Bucle for

Usar for en Python es una forma 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. En general, un bucle for tiene este aspecto:

```
my_list = [..., ..., ..., ...]
for temp_var in my_list:
    block
```

Incluye:
- colección iterable. La estructura de datos por la que pasa el bucle;
- variable temporal. Necesario trabajar con valores de recopilación en cada nuevo paso del ciclo;
- bloque de código. Acciones variables realizadas en cada círculo.

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")
```

2.5 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 desde cero hasta n-1. Por ejemplo, si especifica el rango (5), la salida será [0, 1, 2, 3, 4].

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

Dos argumentos. Si pasa dos argumentos a range() a la vez, obtendrá un rango desde el primer número hasta el segundo. Debemos recordar que durante la salida el segundo argumento se reducirá en uno.

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

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)
```

2.6 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, 9]
for num in numbers:
    if num == 1:
        print("Exit")
        break
    print(num)
```

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, 9]
for num in numbers:
    if num == 1:
        print('Exit')
        break
    print(num)
```

2.7 Pasar a la siguiente iteración

No sólo puedes salir forzosamente del bucle, sino también omitir sus iteraciones por condición usando continuar. 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)
```

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 ciclo, pero nos saltamos iteraciones innecesarias.

2.8 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. Con un bucle repasamos los valores de las celdas de la fila, y con el segundo cambiamos a una nueva fila. Una representación general de bucles anidados se ve así:

```
for condition:
    block
    for condition:
        block
```

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'Element {i}:{j}')
```

2.9 Línea de fondo

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.

Continuar le permite omitir iteraciones que no cumplen la condición.

3. ¿Qué es una lista por comprensión en Python?

3.1 Lista

La mayoría de programas no trabajan con variables individuales, sino con un conjunto de variables. Por ejemplo, un programa puede procesar información sobre los estudiantes de una clase leyendo una lista de estudiantes desde el teclado o desde un archivo, pero cambiar el número de estudiantes en una clase no debería requerir la modificación del código fuente del programa.

Anteriormente, nos enfrentábamos a la tarea de procesar elementos de una secuencia, por ejemplo, calcular el elemento más grande de la secuencia. Pero al mismo tiempo no guardamos la secuencia completa en la memoria de la computadora. Sin embargo, en muchas tareas es necesario almacenar la secuencia completa, por ejemplo, si necesitáramos mostrar todos los elementos de la secuencia en orden

ascendente (“ordenar la secuencia”).

Para almacenar dichos datos, puede utilizar una estructura de datos llamada lista en Python (la mayoría de los lenguajes de programación utilizan otro término, "matriz"). Una lista es una secuencia de elementos, numerados desde 0, como los caracteres de una cadena. Se puede especificar una lista enumerando los elementos de la lista entre corchetes; por ejemplo, una lista se puede especificar de esta manera:

```
Primes = [2, 3, 5, 7, 11, 13]
Rainbow = ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']
```

La lista de Primos tiene 6 elementos, a saber: Primes[0] == 2, Primes[1] == 3, Primes[2] == 5, Primes[3] == 7, Primes[4] == 11, Primes[5] == 13. La lista Rainbow consta de 7 elementos, cada uno de los cuales es una cadena.

Al igual que los caracteres en una cadena, los elementos de la lista se pueden indexar con números negativos desde el final, por ejemplo, Primes[-1] == 13, Primes[-6] == 2.

La longitud de la lista, es decir, el número de elementos que contiene, se puede encontrar usando la función len, por ejemplo, len(Primes) == 6.

A diferencia de las cadenas, los elementos de la lista se pueden modificar asignándoles nuevos valores.

```
Rainbow = ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']
Rainbow[0] = 'Red2'
```

Este ejemplo crea una lista vacía, lee el número de elementos de la lista, luego lee los elementos de la lista uno por uno y los agrega al final de la lista. Se puede escribir lo mismo guardando la variable n:

```
a = []
for i in range(int(input())):
    a.append(int(input()))
print(a)
```

Las siguientes operaciones están completamente definidas para las listas: concatenación de listas (suma de listas, es decir, agregar otra a una lista) y repetición de listas (multiplicar una lista por un número). Por ejemplo:

```
a = [1, 2, 3]
b = [4, 5]
c = a + b
d = b * 3
print([7, 8] + [9])
print([0, 1] * 3)
```

Como resultado, la lista c será igual a [1, 2, 3, 4, 5] y la lista d será igual a [4, 5, 4, 5, 4, 5]. Esto le permite organizar el proceso de lectura de listas de diferentes maneras: primero, cuente el tamaño de la lista y cree una lista a partir del número requerido de elementos, luego organice un bucle sobre la variable i, comenzando con el número 0, y dentro del bucle. , se lee el i-ésimo elemento de la lista:

```
a = [0] * int(input())
for i in range(len(a)):
    a[i] = int(input())
```

Puede imprimir los elementos de la lista a con una sola declaración print(a), que imprimirá corchetes alrededor de los elementos de la lista y comas entre los elementos de la lista. Esta salida es inconveniente; más a menudo solo necesita mostrar todos los elementos de la lista en una línea o un elemento por

línea. A continuación se muestran dos ejemplos que también difieren en la organización del ciclo:

```
a = [1, 2, 3, 4, 5]
for i in range(len(a)):
    print(a[i])
```

Aquí el índice del elemento i se cambia en un bucle, luego se muestra el elemento de la lista con el índice i.

```
a = [1, 2, 3, 4, 5]
for elem in a:
    print(elem, end=' ')
```

En este ejemplo, los elementos de la lista se muestran en una línea, separados por un espacio, mientras que en el bucle no se cambia el índice del elemento de la lista, sino el valor de la variable en sí.

3.2 Métodos split y join

Los elementos de la lista se pueden ingresar uno por línea, en cuyo caso la línea completa se puede tratar como una función input(). Luego puede usar el método split() de la cadena, que devuelve una lista de cadenas que se obtendrían si la cadena original se dividiera en partes en espacios en blanco. Ejemplo:

```
# 1 2 3
s = input()  # s == '1 2 3'
a = s.split()  # a == ['1', '2', '3']
```

Si ingresa la línea 1 2 3 cuando ejecuta este programa, entonces la lista a será igual a ['1', '2', '3']. Tenga en cuenta que la lista estará formada por cadenas, no por números. Si desea obtener una lista de números, puede convertir los elementos de la lista uno por uno en números:

```
a = input().split()
for i in range(len(a)):
    a[i] = int(a[i])
```

Usando magia especial de Python (generadores), se puede hacer lo mismo en una línea:

```
a = [int(s) for s in input().split()]
```

El método split() tiene un parámetro opcional que especifica qué cadena se utilizará como separador entre los elementos de la lista. Por ejemplo, llamar a split('.') devolverá una lista obtenida al dividir la cadena original en los caracteres '.':

```
a = '192.168.0.1'.split('.')
```

En Python, puedes enumerar cadenas usando un comando de una línea. Para ello se utiliza el método de cadena join. Este método tiene un parámetro: una lista de cadenas. Como resultado, se devuelve una cadena obtenida al concatenar los elementos de la lista pasada en una cadena, mientras que entre los elementos de la lista se inserta un separador igual a la cadena a la que se aplica el método. Sabemos que no entendiste la frase anterior la primera vez. Así que mira los ejemplos:

```
a = ['red', 'green', 'blue']
print(' '.join(a))
# return red green blue
print(''.join(a))
# return redgreenblue
```

```
print('***'.join(a))
# return red***green***blue
```

3.3 Generadores de listas

Para crear una lista llena de elementos idénticos, puede utilizar el operador de repetición de lista, por ejemplo:

```
n = 5
a = [0] * n
```

A continuación se muestran algunos ejemplos del uso de generadores.

También puedes crear una lista que consta de n ceros usando un generador:

```
a = [0 for i in range(5)]
```

Puedes crear una lista llena de cuadrados de números enteros como este:

```
n = 5
a = [i ** 2 for i in range(n)]
```

3.4 Rebanadas

Con las listas, al igual que con las cadenas, puedes hacer cortes. A saber:

A[i:j] = A[i], A[i+1], ..., A[j-1].
A[i:j:-1] = A[i], A[i-1], ..., A[j+1] (es decir, el orden de los elementos cambia)
A[i:j:k] = A[i], A[i+k], A[i+2*k]

3.5 Operaciones con listas

x in A - Compruebe si un elemento está en la lista. Devuelve True o False.
min(A) - Elemento de lista más pequeño
max(A) - Elemento de lista más grande
A.index(x) - El índice de la primera aparición del elemento x en la lista; si falta, genera una excepción ValueError
A.count(x) - Número de apariciones del elemento x en la lista

4. ¿Qué es un argumento en Python?

Al pasar argumentos a una función, se realizan una variedad de acciones, dependiendo del tipo de objetos que se envían a la función (objetos mutables o inmutables). Un llamador de función es una entidad que llama a una función y le pasa argumentos. Hablando de funciones de llamada, hay algunas cosas en las que vale la pena pensar y que discutiremos en un momento.

Los argumentos, cuyos nombres se especifican al declarar una función, contienen objetos que se pasan a las funciones cuando se llaman. Al mismo tiempo, si se asigna algo a las variables locales correspondientes de las funciones, sus parámetros, esta operación no afecta los objetos inmutables pasados a las funciones. Por ejemplo:

```
def foo(a):
    a = a+5
    print(a)            # print 15

a = 10
foo(a)
print(a)                # print 10
```

Como puede ver, la llamada a la función no tuvo ningún efecto en la variable a. Esto es exactamente lo que sucede cuando se pasa un objeto inmutable a una función.

Pero si a las funciones se les pasan objetos mutables, es posible que encuentre un comportamiento del sistema diferente al descrito anteriormente.

```
def foo(lst):
    lst = lst + ['new entry']
    print(lst)               # print ['Book', 'Pen', 'new entry']

lst = ['Book', 'Pen']
print(lst)                   # print ['Book', 'Pen']
foo(lst)
print(lst)                   # print ['Book', 'Pen']
```

¿Has notado algo nuevo aquí? Si responde “No”, estará en lo cierto. Pero si de alguna manera influyemos en los elementos del objeto mutable pasado a la función, seremos testigos de algo diferente.

```
def foo(lst):
    lst[1] = 'new entry'
    print(lst)               # print ['Book', 'new entry']

lst = ['Book', 'Pen']
print(lst)                    # print ['Book', 'Pen']
foo(lst)
print(lst)                    # print ['Book', 'new entry']
```

Como puede ver, el objeto del último parámetro se cambió después de llamar a la función. Esto sucedió porque estamos trabajando con una referencia al objeto almacenado en el último parámetro. Como resultado, cambiar el contenido de este objeto está fuera del alcance de la función. Puede evitar esto simplemente haciendo copias profundas de dichos objetos y escribiéndolas en variables de funciones locales.

```
def foo(lst):
    lst = lst[:]
    lst[1] = 'new entry'
    print(lst)                  # print ['Book', 'new entry']

lst = ['Book', 'Pen']
print(lst)                      # print ['Book', 'Pen']
foo(lst)
print(lst)                      # print ['Book', 'Pen']
```

Esto es lo que debes saber sobre los argumentos de función:
1. El orden en el que se pasan los argumentos posicionales a las funciones.
2. El orden en el que los argumentos nombrados se pasan a las funciones.
3. Asignación de valores de argumento predeterminados.
4. Organizar el procesamiento de conjuntos de argumentos de longitud variable.
5. Desempaquetando argumentos.
6. Usar argumentos que solo se pueden pasar por nombre (solo palabras clave).
Veamos cada uno de estos puntos.

4.1 El orden de pasar argumentos posicionales a funciones.

Los argumentos posicionales se procesan de izquierda a derecha. Es decir, resulta que la posición del argumento pasado a la función está en correspondencia directa con la posición del parámetro utilizado en el encabezado de la función al declararla.

```
def foo(d, e, f):
    print(d, e, f)

a, b, c = 1, 2, 3
foo(a, b, c)                  # print 1, 2, 3
foo(b, a, c)                  # print 2, 1, 3
foo(c, b, a)                  # print 3, 2, 1
```

Las variables a, b y c tienen respectivamente los valores 1, 2 y 3. Estas variables actúan como argumentos con los que se llama a la función foo. Estos, cuando se llama a la función por primera vez, corresponden a los parámetros d, e y f. Este mecanismo se aplica a casi todos los 6 puntos anteriores con respecto a lo que necesita saber sobre los argumentos de funciones en Python. La ubicación del argumento posicional pasado a la función cuando se llama juega un papel importante en la asignación de valores a los parámetros de la función.

4.2 Orden de paso de argumentos con nombre a funciones

Los argumentos con nombre se pasan a funciones con los nombres de estos argumentos correspondientes a los nombres que se les asignaron cuando se declaró la función.

```
def foo(arg1=0, arg2=0, arg3=0):
    print(arg1, arg2, arg3)

a, b, c = 1, 2, 3
foo(a,b,c)                          # print 1 2 3
foo(arg1=a, arg2=b, arg3=c)         # print 1 2 3
foo(arg3=c, arg2=b, arg1=a)         # print 1 2 3
foo(arg2=b, arg1=a, arg3=c)         # print 1 2 3
```

Como puedes ver, la función foo toma 3 argumentos. Estos argumentos se denominan arg1, arg2 y arg3. Preste atención a cómo cambiamos las posiciones de los argumentos al llamar a la función. Los argumentos con nombre se tratan de forma diferente a los argumentos posicionales, aunque el sistema continúa leyéndolos de izquierda a derecha. Python considera los nombres de los argumentos, no sus posiciones, al asignar valores apropiados a los parámetros de una función. Como resultado, resulta que la función genera lo mismo independientemente de las posiciones de los argumentos que se le pasan. Siempre es 1 2 3.

Tenga en cuenta que los mecanismos descritos en el punto N° 1 continúan aplicándose aquí.

4.3 Asignación de valores de argumento predeterminados

A los argumentos con nombre se les pueden asignar valores predeterminados. Cuando se utiliza este mecanismo en una función, ciertos argumentos se vuelven opcionales. Declarar tales funciones se parece a lo que consideramos en el punto 2. La única diferencia es cómo se llaman exactamente estas funciones.

```
def foo(arg1=0, arg2=0, arg3=0):
    print(arg1, arg2, arg3)

a, b, c = 1, 2, 3
foo(arg1=a)                         # print 1 0 0
foo(arg1=a, arg2=b )                # print 1 2 0
foo(arg1=a, arg2=b, arg3=c)         # print 1 2 3
```

Tenga en cuenta que en este ejemplo no pasamos todos los argumentos descritos al declarar la función. En estos casos, a los parámetros correspondientes se les asignan valores predeterminados. Sigamos con este ejemplo:

```
foo(arg2=b)                         # print 0 2 0
foo(arg2=b, arg3=c )                # print 0 2 3

foo(arg3=c)                         # print 0 0 3
foo(arg3=c, arg1=a )                # print 1 0 3
```

Estos son ejemplos simples y claros del uso de los mecanismos descritos anteriormente para llamar a funciones pasándoles argumentos con nombre. Ahora compliquemos nuestros experimentos combinando lo que hemos hablado hasta ahora en los puntos 1, 2 y 3:

```
foo(a, arg2=b)                      # print 1 2 0
foo(a, arg2=b, arg3=c)              # print 1 2 3
foo(a, b, arg3=c)                   # print 1 2 3

foo(a)                              # print 1 0 0
foo(a,b)                            # print 1 2 0
```

Aquí, al llamar a una función, se utilizan argumentos posicionales y con nombre. Cuando se utilizan argumentos posicionales, el orden en el que se especifican sigue desempeñando un papel fundamental para pasar correctamente la función de datos de entrada.

Aquí me gustaría llamar su atención sobre un detalle notable. El problema es que los argumentos posicionales no se pueden especificar después de argumentos con nombre. A continuación se muestra un ejemplo para ayudarle a comprender mejor esta idea:

```
foo(arg1=a, b)
>>>
foo(arg1=a, b)
          ^
SyntaxError: positional argument follows keyword argument
foo(a, arg2=b, c)
>>>
foo(a, arg2=b, c)
             ^
SyntaxError: positional argument follows keyword argument
```

Puedes tomar esto como una regla. Los argumentos posicionales no deben seguir a los argumentos con nombre al llamar a una función.

4.4 Organizar el procesamiento de conjuntos de argumentos de longitud variable.

Aquí hablaremos sobre las construcciones *args y **kwargs. Cuando estas construcciones se utilizan en una declaración de función, esperamos que conjuntos de argumentos de longitud arbitraria se representen como args y kwargs cuando se llama a la función. Cuando se utiliza la construcción *args, el parámetro args contiene argumentos posicionales representados como una tupla. Cuando se utiliza **kwargs, kwargs contiene argumentos con nombre, representados como un diccionario.

```
def foo(*args):
    print(args)

a, b, c = 1, 2, 3

foo(a, b, c)                # print (1, 2, 3)
foo(a, b)                   # print (1, 2)
foo(a)                      # print (1)
foo(b, c)                   # print (2, 3)
```

Este código demuestra que el parámetro args almacena una tupla que contiene lo que se pasa a la función cuando se llama.

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

foo(a=1, b=2, c=3)        # print {'a': 1, 'b': 2, 'c': 3}
foo(a=1, b=2)             # print {'a': 1, 'b': 2}
foo(a=1)                  # print {'a': 1}
foo(b=2, c=3)             # print {'b': 2, 'c': 3}
```

El código anterior muestra que el parámetro kwargs almacena un diccionario de pares clave-valor que representan los argumentos con nombre pasados a la función cuando se llama.

Pero cabe señalar que una función diseñada para aceptar argumentos posicionales no puede pasar argumentos con nombre (y viceversa).

```
def foo(*args):
    print(args)

foo(a=1, b=2, c=3)
>>>
foo(a=1, b=2, c=3)
TypeError: foo() got an unexpected keyword argument 'a'
############################################################
def foo(**kwargs):
    print(kwargs)

a, b, c = 1, 2, 3
foo(a, b, c)
>>>
TypeError: foo() takes 0 positional arguments but 3 were given
```

Ahora juntemos todo lo que discutimos en los puntos 1, 2, 3 y 4, y experimentemos con todo esto, explorando diferentes combinaciones de argumentos que se pueden pasar a funciones cuando se llaman.

```
def foo(*args,**kwargs):
    print(args, kwargs)

foo(a=1,)
# () {'a': 1}

foo(a=1, b=2, c=3)
# () {'a': 1, 'b': 2, 'c': 3}

foo(1, 2, a=1, b=2)
# (1, 2) {'a': 1, 'b': 2}

foo(1, 2)
# (1, 2) {}
```

Como puedes ver, tenemos a nuestra disposición una tupla de args y un diccionario de kwargs.

Aquí hay otra regla. El problema es que *args no se puede utilizar después de **kwargs.

```
def foo(**kwargs, *args):
    print(kwargs, args)
>>>
    def foo(**kwargs, *args):
```

```
                    ^
SyntaxError: invalid syntax
```

La misma regla se aplica al orden de especificación de los argumentos al llamar a funciones. Los argumentos posicionales no deben seguir a argumentos con nombre.

```
foo(a=1, 1)
>>>
    foo(a=1, 1)
           ^
SyntaxError: positional argument follows keyword argument
foo(1, a=1, 2)
>>>
    foo(1, a=1, 2)
              ^
SyntaxError: positional argument follows keyword argument
```

Al declarar funciones, puede combinar argumentos posicionales, *args y *kwagrs de la siguiente manera:

```
def foo(var, *args,**kwargs):
    print(var, args, kwargs)

foo(1, a=1,)                            # call 1
# 1 () {'a': 1}

foo(1, a=1, b=2, c=3)                   # call 2
# 1 () {'a': 1, 'b': 2, 'c': 3}

foo(1, 2, a=1, b=2)                     # call 3
# 1 (2,) {'a': 1, 'b': 2}
foo(1, 2, 3, a=1, b=2)                  # call 4
# 1 (2, 3) {'a': 1, 'b': 2}
foo(1, 2)                               # call 5
# 1 (2,) {}
```

Al declarar la función foo, asumimos que debe tener un argumento posicional requerido. A esto le sigue un conjunto de argumentos posicionales de longitud variable, y a este conjunto le sigue un conjunto de argumentos con nombre de longitud variable. Sabiendo esto, podemos "descifrar" fácilmente cada una de las llamadas a funciones anteriores.

En la Llamada 1, a la función se le pasan los argumentos 1 y a=1. Estos son argumentos posicionales y con nombre, respectivamente. La llamada 2 es una variación de la llamada 1. Aquí la longitud del conjunto de argumentos posicionales es cero.

En la Llamada 3 pasamos las funciones 1, 2 y a=1,b=2. Esto significa que ahora se necesitan dos argumentos posicionales y dos argumentos con nombre. Según la declaración de la función, resulta que 1 se trata como un argumento posicional requerido, 2 va al conjunto de argumentos posicionales de longitud variable y a=1 y b=2 van al conjunto de argumentos con nombre de longitud variable.

Para llamar a esta función correctamente, debemos pasarle al menos un argumento posicional. De lo contrario nos encontraremos con un error.

```
def foo(var, *args,**kwargs):
    print(var, args, kwargs)

foo(a=1)
>>>
foo(a=1)
```

```
TypeError: foo() missing 1 required positional argument: 'var'
```

Otra variación de dicha función sería una función cuya declaración indique que toma un argumento posicional requerido y un argumento con nombre, seguidos de conjuntos de argumentos posicionales y con nombre de longitud variable.

```
def foo(var, kvar=0, *args,**kwargs):
    print(var, kvar, args, kwargs)

foo(1, a=1,)                                # call 1
# 1 0 () {'a': 1}

foo(1, 2, a=1, b=2, c=3)                    # call 2
# 1 0 () {'a': 1, 'b': 2, 'c': 3}

foo(1, 2, 3, a=1, b=2)                      # call 3
# 1 2 () {'a': 1, 'b': 2}

foo(1, 2, 3, 4, a=1, b=2)                   # call 4
# 1 2 (3,) {'a': 1, 'b': 2}

foo(1, kvar=2)                              # call 5
# 1 2 () {}
```

Las llamadas a esta función se pueden "descifrar" de la misma forma que se hizo al analizar la función anterior.

Al llamar a esta función, se le debe pasar al menos un argumento posicional. De lo contrario nos encontraremos con un error:

```
foo()
>>>
foo()
TypeError: foo() missing 1 required positional argument: 'var'
foo(1)
# 1 0 () {}
```

Tenga en cuenta que llamar a foo(1) funciona bien. El punto aquí es que si se llama a una función sin especificar un valor para un argumento con nombre, el valor se le asigna automáticamente.

Aquí hay algunos errores más que puede encontrar al llamar incorrectamente a esta función:

```
foo(kvar=1)                             # call 1
>>>
TypeError: foo() missing 1 required positional argument: 'var'
foo(kvar=1, 1, a=1)                      # call 2
>>>
SyntaxError: positional argument follows keyword argument
foo(1, kvar=2, 3, a=2)                   # call 3
>>>
SyntaxError: positional argument follows keyword argument
```

Presta especial atención al error que se produce durante la Llamada 3.

4.5 Desempaquetando argumentos

En secciones anteriores, hablamos sobre cómo recopilar conjuntos de argumentos pasados a funciones en tuplas y diccionarios. Y aquí discutiremos la operación inversa. Es decir, analizaremos el mecanismo que le permite descomprimir los argumentos suministrados a la entrada de la función.

```
args = (1, 2, 3, 4)
print(*args)                  # print 1 2 3 4
print(args)                   # print (1, 2, 3, 4)

kwargs = { 'a':1, 'b':2}
print(kwargs)                 # print {'a': 1, 'b': 2}
print(*kwargs)                # print a b
```

Las variables se pueden descomprimir utilizando construcciones sintácticas * y **. Así es como se ve su uso al pasar tuplas, listas y diccionarios a funciones.

```
def foo(a, b=0, *args, **kwargs):
    print(a, b, args, kwargs)

tup = (1, 2, 3, 4)
lst = [1, 2, 3, 4]
d = {'e':1, 'f':2, 'g':'3'}

foo(*tup)             # foo(1, 2, 3, 4)
# 1 2 (3, 4) {}

foo(*lst)             # foo(1, 2, 3, 4)
# 1 2 (3, 4) {}

foo(1, *tup)          # foo(1, 1, 2, 3, 4)
# 1 1 (2, 3, 4) {}

foo(1, 5, *tup)       # foo(1, 5, 1, 2, 3, 4)
# 1 5 (1, 2, 3, 4) {}

foo(1, *tup, **d)     # foo(1, 1, 2, 3, 4 ,e=1 ,f=2, g=3)
# 1 1 (2, 3, 4) {'e': 1, 'f': 2, 'g': '3'}

foo(*tup, **d)         # foo(1, 1, 2, 3, 4 ,e=1 ,f=2, g=3)
# 1 2 (3, 4) {'e': 1, 'f': 2, 'g': '3'}
d['b'] = 45
foo(2, **d)             # foo(1, e=1 ,f=2, g=3, b=45)
# 2 45 () {'e': 1, 'f': 2, 'g': '3'}
```

Considere cada una de las llamadas a funciones que se muestran aquí y que se realizaron mediante el desempaquetado de argumentos y observe cómo se verían las llamadas correspondientes sin el uso de * y **. Intente comprender qué sucede cuando se realizan estas llamadas y cómo se descomprimen las distintas estructuras de datos.

Al experimentar con el descomprimido de argumentos, es posible que encuentre un nuevo error:

```
foo(1, *tup, b=5)
>>>
TypeError: foo() got multiple values for argument 'b'
foo(1, b=5, *tup)
>>>
TypeError: foo() got multiple values for argument 'b'
```

Este error se produce porque un argumento con nombre, b=5, y un argumento posicional entran en conflicto. Como descubrimos en la sección 2, al pasar argumentos con nombre, su orden no importa. Como resultado, se produce el mismo error en ambos casos.

4.6 Usar argumentos que solo se pueden pasar por nombre (solo palabras clave)

En algunos casos, es necesario hacer que una función tome los argumentos con nombre requeridos. Si, al declarar una función, describe argumentos que solo se pueden pasar por nombre, entonces dichos argumentos se le deben pasar cada vez que se llama.

```
def foo(a, *args, b):
    print(a, args, b)

tup = (1, 2, 3, 4)

foo(*tup, b=35)
# 1 (2, 3, 4) 35

foo(1, *tup, b=35)
# 1 (1, 2, 3, 4) 35

foo(1, 5, *tup, b=35)
# 1 (5, 1, 2, 3, 4) 35

foo(1, *tup, b=35)
# 1 (1, 2, 3, 4) 35

foo(1, b=35)
# 1 () 35

foo(1, 2, b=35)
# 1 (2,) 35

foo(1)
# TypeError: foo() missing 1 required keyword-only argument: 'b'

foo(1, 2, 3)
# TypeError: foo() missing 1 required keyword-only argument: 'b'
```

Como puede ver, se espera que a la función se le pase un argumento con nombre b, que aparece después de *args en la declaración de la función. En este caso, en una declaración de función, simplemente puede usar el símbolo *, después del cual, separados por una coma, hay identificadores de argumentos con nombre, que se pueden pasar a la función solo por su nombre. Una función de este tipo no estará diseñada para aceptar un conjunto de argumentos posicionales de longitud variable.

```
def foo(a, *, b, c):
    print(a, b, c)

tup = (1, 2, 3, 4)

foo(1, b=35, c=55)
# 1 35 55

foo(c= 55, b=35, a=1)
# 1 35 55

foo(1, 2, 3)
# TypeError: foo() takes 1 positional argument but 3 were given

foo(*tup, b=35)
# TypeError: foo() takes 1 positional argument but 4 positional arguments (and 1 keyword-only argument) were given
```

```
foo(1, b=35)
# TypeError: foo() takes 1 positional argument but 4 positional arguments (and 1
keyword-only argument) were given
```

La función declarada en el ejemplo anterior toma un argumento posicional y dos argumentos con nombre, que solo se pueden pasar por nombre. Esto da como resultado que a la función se le tengan que pasar ambos argumentos con nombre para poder llamarla correctamente. Después de * también puede describir argumentos con nombre a los que se les asignan valores predeterminados. Esto nos da cierta libertad al llamar a dichas funciones.

```
def foo(a, *, b=0, c, d=0):
    print(a, b, c, d)

foo(1, c=55)
# 1 0 55 0

foo(1, c=55, b=35)
# 1 35 55 0

foo(1)
# TypeError: foo() missing 1 required keyword-only argument: 'c'
```

Tenga en cuenta que la función se puede llamar normalmente sin pasar los argumentos b y d, ya que están configurados con valores predeterminados.

5. ¿Qué es una función Lambda en Python?

Las funciones Lambda en Python son anónimas. Esto significa que la función no tiene nombre. Como sabes, la palabra clave def se utiliza en Python para definir una función normal. A su vez, la palabra clave lambda se utiliza para definir una función anónima.

5.1 La función lambda tiene la siguiente sintaxis:

```
lambda args: expresión
```

Las funciones Lambda pueden tener cualquier número de argumentos, pero cada una solo puede tener una expresión. La expresión se evalúa y se devuelve. Estas funciones se pueden utilizar siempre que se requiera un objeto de función.

Ejemplo:

```
double = lambda x: x*2
print(double(5))         # print 10
```

En el código anterior, lambda x: x*2 es la función lambda. Aquí x es el argumento y x*2 es la expresión que se evalúa y devuelve.

Esta función no tiene nombre. Devuelve un objeto de función con identificador doble. Ahora podemos considerarla una función normal.

Instrucciones:

```
double = lambda x: x*2
```

Equivalente a:

```
def double(x):
            return x * 2
```

- Esta función puede tomar cualquier número de argumentos, pero evalúa y devuelve solo un valor.
- Las funciones Lambda son aplicables siempre que se requieran objetos de función.
- Debes recordar que la función lambda está limitada sintácticamente, permitiendo representar solo una expresión
- Tienen muchos usos en áreas específicas de programación, junto con otros tipos de expresiones utilizadas en funciones.

5.2 Diferencia entre función regular y función lambda

Veamos un ejemplo e intentemos comprender la diferencia entre la definición (Def) de una función regular y una función lambda. Este código devuelve el valor dado al cubo:

```
def defined_cube(y):
    return y*y*y

lambda_cube = lambda y: y*y*y
print(defined_cube(2))           # print 8
print(lambda_cube(2))           # print 8
```

Como se muestra en el ejemplo anterior, ambas funciones proporcionadas, define_cube() y lambda_cube(), se comportan igual que se esperaba.

Veamos el ejemplo anterior con más detalle:
- Sin usar lambda: aquí ambas funciones devuelven el valor dado al cubo. Pero cuando usamos def, tuvimos que definir una función con un nombre y define_cube() darle un valor de entrada. Después de la ejecución, también necesitábamos devolver el resultado desde donde se llamó la función, y lo hicimos usando la palabra clave return.
- Uso de una lambda: la definición lambda no incluye una declaración de devolución, pero siempre contiene la expresión devuelta. También podemos colocar la definición lambda en cualquier lugar donde se espere la función y no tenemos que asignarla a una variable. Así es como se ven las funciones lambda simples.

5.3 Funciones lambda y funciones de orden superior

Usamos una función lambda cuando necesitamos brevemente una función sin nombre.

En Python, a menudo los usamos como argumento para una función de orden superior (una función que toma otras funciones como argumentos). Las funciones Lambda se utilizan junto con funciones integradas como filter(), map(), reduce(), etc.
Veamos algunos usos más comunes de las funciones lambda.

Ejemplo con filtro()

La función filter() en Python toma una función y una lista como argumentos.

La función se llama con todos los elementos de la lista y el resultado es una nueva lista que contiene los elementos para los cuales la función devuelve Verdadero.

A continuación se muestra un ejemplo del uso de la función filter() para seleccionar números pares de una lista.

```
my_list = [1, 3, 4, 6, 10, 11, 15, 12, 14]
new_list = list(filter(lambda x: (x%2 == 0) , my_list))
print(new_list)                        # print [4, 6, 10, 12, 14]
```

Ejemplo con mapa()

La función map() toma una función y una lista como argumentos.

La función se llama con todos los elementos de la lista y el resultado es una nueva lista que contiene los elementos devueltos por la función para cada elemento original.

A continuación se muestra un ejemplo del uso de la función map() para duplicar todos los elementos de una lista.

```
current_list = [1, 3, 4, 6, 10, 11, 15, 12, 14]
new_list = list(map(lambda x: x*2 , current_list))
print(new_list)                    # print [2, 6, 8, 12, 20, 22, 30, 24, 28]
```

Ejemplo con reduce()

La función reduce() toma una función y una lista como argumentos. La función se llama usando una función lambda y se devuelve un iterable y un nuevo resultado minimizado. Esto realiza una operación repetida en pares de objetos iterables. La función reduce() es parte del módulo functools.

```
from functools import reduce


current_list = [5, 15, 20, 30, 50, 55, 75, 60, 70]
summa = reduce((lambda x, y: x + y), current_list)
print(summa)                      # print 380
```

Aquí los resultados de los dos elementos anteriores se suman con el siguiente elemento y esto continúa hasta el final de la lista.
(5+15+20+30+50+55+75+60+70)

5.4 Lambda y lista de inclusión

En este ejemplo, usaremos una función lambda habilitada para listas y una función lambda de bucle for. Mostraremos una tabla de 10 elementos.

```
tables = [lambda x = x: x*10 for x in range(1, 11)]
for table in tables:
    print(table())                # print 10 20 30 40 50 60 70 80 90 100
```

5.5 Lambda y declaraciones condicionales

Veamos el uso de condiciones if-else en una función lambda. Como sabes, Python nos permite usar condiciones de una línea, y estas son las que podemos poner en una función lambda para procesar el resultado devuelto.

Por ejemplo, hay dos dígitos y debes determinar cuál representa el número mayor.

```
max_number = lambda a, b: a if a > b else b
print(max_number(3, 5))           # print 5
```

Este método le permite agregar condiciones a funciones lambda.

5.6 Lambda y múltiples operadores

Las funciones lambda no permiten declaraciones múltiples; sin embargo, podemos crear dos funciones lambda y luego llamar a la segunda función lambda

como parámetro de la primera función. Intentemos encontrar el segundo elemento más grande usando lambda.

```
current_list = [[10,6,9],[0, 14, 16, 80],[8, 12, 30, 44]]
sorted_list = lambda x: (sorted(i) for i in x)
second_largest = lambda x, func: [y[len(y)-2] for y in func(x)]
result = second_largest(current_list, sorted_list)
print(result)                                # print [9, 16, 30]
```

En el ejemplo anterior, creamos una función lambda que ordena cada sublista dentro de una lista determinada. Luego, esta lista se pasa como parámetro a una segunda función lambda, que devuelve el elemento n-2 de la lista ordenada, donde n es la longitud de la lista anidada.

6. ¿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 nombre_paquete: Instala el paquete especificado.
pip uninstall nombre_paquete: desinstala el paquete especificado.
pip list: muestra los paquetes instalados.
pip show nombre_paquete: muestra información sobre un paquete específico.
pip search query: busca paquetes asociados con una consulta.
```

Ejemplo de uso:

Instalación del paquete:
```
pip install requests
```
Quitar un paquete:
```
pip uninstall requests
```
Ver paquetes instalados:
```
pip list
```
Ver información del paquete:
```
pip show Flask
```
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 que pip esté instalado en su entorno Python para que pueda administrar fácilmente los paquetes para sus aplicaciones.