Loops

What are loops?

Let's start with an everyday example that you're all familiar with. Have you ever been to a playground with a swing?

Have you ever been to a playground with a swing?

If yes then imagine you're on that swing, moving back and forth. Now, you want to keep swinging for a specific number of times – let's say ten swings.

So, instead of having someone count each swing for you, you can use a loop – a concept that helps you perform repetitive tasks automatically. You'll simply tell yourself, "I'm going to swing ten times," and you'll keep swinging until you reach that count.

"I'm going to swing ten times" and you'll keep swinging until you reach that count.

Loops work similarly in programming. They help us repeat a set of instructions for a specified number of times or until a particular condition is met.

Let's consider an example where we need to print a number 100 times. Instead of writing the 'print' statement 100 times, which would be tedious, we can utilize the concept of a loop to automate the process efficiently.

we can utilize the **concept of a loop** to automate the process efficiently.
So it's an essential concept in coding.

So it's an essential concept in coding, and today, we'll explore different types of loops and how they're used.

Now, let's explore the first type of loop, which is the **for loop**. To understand it better, let's use a simple example that everyone can relate to, like counting the number of candies in a jar.

Imagine you have a jar filled with candies, and you want to count each candy one by one.

Imagine you have a jar filled with candies, and you want to count each candy one by one

Instead of doing it manually, you can use a for loop, which acts like your helpful assistant, to count the candies for you.

In Python, a for loop is a way to repeat a task for each item in a collection. It goes through the elements one by one and performs an action on each of them.

In Python, a for loop is a way to repeat a task for each item in a collection. It goes through the elements one by one and performs an action on each of them.

The syntax of a for loop is as follows:

- for item in collection :

Do something with the item

So, in the loop, "item" is like a temporary container that holds each toy (or element) from your collection, one at a time. And the "collection" is the whole group of toys you want to play with.

The for loop keeps going until it has gone through each item in the collection and performed the action you specified. It's a fantastic way to work with lots of things without having to do the same task over and over manually.

In Python, the "item" and "collection" in a for loop can take various forms depending on the specific programming task you want to accomplish. Let's break it down:

The "**item**" is a temporary variable that represents each element or item in the "collection" during each iteration of the loop.

It can be any valid variable name you choose. For example, you can use "toy" if you are working with a collection of toys, "number" if you are processing a list of numbers, or "word" if you are dealing with a list of strings.

The "**collection**" is a group of elements or items that you want to iterate over using the for loop.

It can be various data structures in Python, such as:

Lists: A collection of items enclosed in square brackets, like [1, 2, 3].

Strings: A sequence of characters, like "Hello, World!".

Tuples: Similar to lists but enclosed in parentheses, like (10, 20, 30).

Sets : An unordered collection of unique elements, like {1, 2, 3}.

Dictionaries: A collection of key-value pairs, like {"name": "John", "age": 30}.

Sequences: A general term for ordered collections of elements, which includes strings, lists, and tuples. A "sequence" is a more general term that covers ordered collections where each element has a specific position or index. In Python, strings, lists, and tuples are all examples of sequences.

The colon: in Python is like a signal to the computer that says, "Hey, something important is coming up!" When you see a colon, it means you are starting a new block of code, such as a loop or a function. It's like the beginning of a new section that the computer needs to pay special attention to.

The colon (**:**) after the "collection" line tells the computer that you are starting a for loop. The next line and all the lines after it, which are indented, will be part of the loop's code block. In other words, they belong to the for loop, and they will be repeated for each item in the "collection."

A simple example in Python that prints each letter of a name which is a string using a for loop:

- name = "John"

  for letter in name:

  print(letter)

Here The "collection" is the string "John", which is a sequence of characters.

And The "item" is represented by the variable name letter, which takes on each individual character in the string during each iteration of the loop. The output of the code will be:

- J

  o

  h

  n

The loop iterates over each character in the string "John," and the print(letter) statement prints each character on a new line.

Note: here letter is a variable; instead of using the code line for letter in name, even if we use the code line for a in name; the output would be the same.

Now, let's print numbers from 1 to 10:

But first let’s understand about **range()** function.

The range() function in Python is a built-in function that generates a sequence of numbers. It is commonly used in for loops to iterate over a specific range of values. The basic syntax of the range() function is as follows:

- range(start, stop, step)

  start: The starting value of the sequence which is optional. If not provided, it defaults to 0.

stop: The ending value of the sequence which is required. The sequence will go up to, but not include, this value.

step: The difference between each consecutive number in the sequence which is also optional. So If not provided, it defaults to 1.

So getting back to our example, we use the range(1, 11) function to generate a sequence of numbers from 1 to 10.

- for number in range(1, 11):

  print(number)

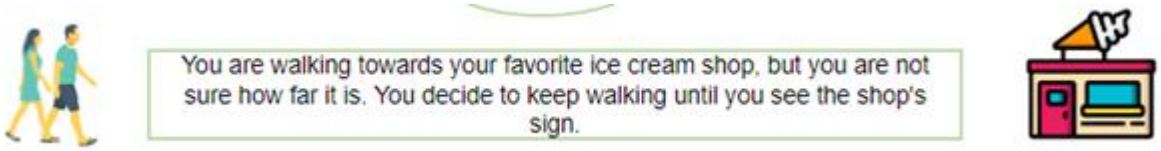
The for loop takes each number from this sequence, one by one, and prints it on the screen.

By using the for loop, we can easily perform repetitive tasks, like printing numbers, without writing the same code multiple times.

While loop

What is a while loop?

Imagine you are walking towards your favorite ice cream shop, but you are not sure how far it is. You decide to keep walking until you see the shop's sign.



As long as you haven't reached the ice cream shop, you will keep walking. This idea of 'keep doing something until a condition is met' is precisely what a 'while loop' is all about.

A 'while loop' is a programming method that allows us to execute a block of code repeatedly as long as a specified condition remains true. It continuously checks the condition before each iteration. If the condition is still true, it executes the code block, and the loop continues. However, once the condition becomes false, the loop stops, and the program moves on to the next statement after the while loop.

The syntax of a 'while loop' in Python is straightforward:

- while condition:

  # Code block to be executed during each iteration

The condition is a boolean expression i.e If the condition is true, the loop keeps running. If the condition becomes false, the loop stops, and that determines whether the loop should continue or stop. It is checked before each iteration.

The code block indented under the 'while' statement will be executed repeatedly as long as the condition remains true. The colon ":" after the condition indicates the start of a code block that belongs to the while loop.

So, the while loop works like this:

1) It checks the "condition."

2) If the condition is true, it enters the code block and executes the code inside it.

3) After completing the code block, it goes back to step 1 and checks the condition again.

- It keeps repeating steps 1 to 3 until the condition becomes false.

Here's an example of a while loop in Python:

- count = 1

  while count <= 5:

    print("Eating a cookie...")

    count = count + 1

In this example, the while loop will keep executing the code block (printing "Eating a cookie...") as long as the condition count <= 5 is true. The variable count starts at 1 and increments by 1 in each iteration by ‘count = count + 1’.

So, here's what happens at each iteration:

Iteration 1: count = 1

Iteration 2: count = 2

Iteration 3: count = 3

Iteration 4: count = 4

Iteration 5: count = 5

Once count becomes 6, the condition count <= 5 becomes false, and the loop exits.

The output of the above example is:

Eating a cookie...

Eating a cookie...

Eating a cookie...

Eating a cookie...

Eating a cookie...

This way, you can use a while loop to perform actions repeatedly until a specific condition is no longer met.

Be cautious to ensure that the condition eventually becomes false, if the condition never becomes false, the loop can keep running forever, so be careful with your conditions!

For example in the previous example, the loop will run indefinitely, if we just keep the first two lines of the code; which are count = 1 and while count <= 5 because the value of count always remains 1, and the condition count <= 5 is always satisfied as 1 always remains less than 5 since it is not incremented in each step. Without the line count = count + 1, the loop becomes an infinite loop, continuously printing "Eating a cookie..." without ever exiting, as the value of count is never updated to progress towards the termination condition.

break and continue statement

Two powerful keywords in Python loops.

Imagine you are shopping for fruits at the grocery store, and you have a shopping list of various fruits you want to buy.

As you walk through the aisles, you start picking fruits from your shopping list. Suddenly, you see your favorite fruit, which is an exotic mango, on a special display.

You are shopping for fruits at the grocery store, and you have a shopping list of various fruits you want to buy.
As you walk through the aisles, you start picking fruits from your shopping list.

Excitedly, you grab the mango and decide to stop shopping for other fruits.

In this case, you used "break" to exit the shopping loop early because you found what you wanted - the exotic mango.

In this case, you used "**break**" to **exit the shopping loop** early because you found what you wanted - the exotic mango

You don't need to continue searching for other fruits on your list since you are happy with your find.

On another day, you're shopping for fruits again, and you spot a big basket filled with various fruits. As you go through your shopping list, you notice there are some fruits you don't like, such as kiwi.

As you go through your shopping list, you notice there are some fruits you don't like, such as kiwi

You decide to skip those fruits and only pick the ones you enjoy.

You use "continue" to move to the next fruit on your list without adding the kiwi to your cart.

You decide to skip those fruits and only pick the ones you enjoy.
You use "**continue**" to move to the next fruit on your list without adding the kiwi to your cart.

In both examples, "break" and "continue" helped you control your shopping experience. "Break" allowed you to stop shopping once you found your favorite fruit, and "continue" helped you skip fruits you didn't like while continuing with the rest of your shopping. Similarly, in programming, "break" and "continue" are valuable tools that allow you to manage the flow of your code based on certain conditions, making your code more efficient and flexible.

break statement

The break statement is used to exit a loop prematurely. When a break statement is encountered inside a loop (such as a for or while loop), the loop is immediately terminated.

Python code to understand the "break" statement.

- for number in range(1, 7):

  print(number)

  if number == 5:

    print("Reached 5, breaking the loop.")

    break

  So, the loop is broken after the number 5 is printed because the break statement is inside the if block that checks if number is equal to 5. Once the condition is true, which is ‘if number == 5’. When this block is executed, it includes the print("Reached 5, breaking the loop.") statement and the ‘break’ statement. The break statement immediately exits the loop, preventing any further iterations from taking place, even though it hasn't reached 6 yet.

  Output:

  1

  2

  3

  4

  5

Reached 5, breaking the loop.

As a result, the loop stops after printing numbers 1 to 5, and the message indicates that the loop was broken when the iteration was 6.

continue statement

The continue statement is used to skip the rest of the code inside a loop for the current iteration and jump to the next iteration. It's handy for avoiding specific parts of the loop based on a condition.

Python code to understand the "continue" statement.

Where it uses a "for" loop to print numbers from 1 to 5, and we'll add a condition to skip the iteration when the number is 6 in the range from 1 to 10.

- for number in range(1, 11):

  if number == 6:

  print("Skipping 6, continuing the loop.")

  continue

  print(number)

  Output:

  1

  2

  3

  4

  5

  Skipping 6, continuing the loop.

  7

  8

9

10

During each iteration, the condition if number == 6: is checked.

When the loop reaches the number 6, the condition is met, and the "Skipping 6, continuing the loop." message is printed.

The "continue" statement is executed, which skips the current iteration (6) and proceeds to the next iteration without executing the print(number) statement for that iteration.

The loop continues and prints numbers 1 to 5, then skips printing 6, and proceeds to print numbers 7 to 10.

As a result, the number 6 is skipped in the loop, and the message indicates that the loop continued without printing it.

A break statement is used to exit a loop prematurely, terminating the loop and transferring control to the next statement outside the loop. Whereas the continue statement skips the rest of the code inside a loop for the current iteration and jumps to the next iteration, effectively bypassing the remaining code in the loop for that specific iteration.

Nested loop

A nested loop is a loop inside another loop.

In programming, a loop is a control structure that allows a certain block of code to be repeated multiple times. When one loop is placed inside another loop, it creates a nested loop structure. This means that the inner loop will execute its entire cycle each time the outer loop iterates once.

A nested loop is like making a grid of things. Imagine you have rows and columns, like in a table or a checkerboard.

Outer Loop: This is like moving through each row. You're saying, "Okay, let's look at the first row, then the second, and so on."

Inner Loop: Now, for each row you're on, the inner loop is like going through each column. It says, "In this row, let's look at the first column, then the second, and so on."

Nested loops are like going through a grid. The outer loop picks a row, and the inner loop goes through each column in that row. It helps you organize things and deal with a set of information in a systematic way. Let’s understand this with an example:

- for row in range(1, 4): # Outer loop for three rows

  for col in range(1, 4): # Inner loop for three columns in each row

  print(f'({row}, {col})')

  Here's what each part does:

  Outer Loop (for row in range(1, 4):):

  The outer loop uses the variable row to iterate over a range of values from 1 to 3 (inclusive).

  This loop is responsible for controlling the rows in the grid or table.

  Inner Loop (for col in range(1, 4):):

  Inside the outer loop, there's another loop, the inner loop.

  The inner loop uses the variable col to iterate over a range of values from 1 to 3 (inclusive).

  This loop is responsible for controlling the columns in the grid or table.

  Print Statement (print(f'({row}, {col})')):

  Inside the inner loop, there's a print statement.

  It prints the current values of row and col enclosed in parentheses.

This line effectively prints each coordinate in a grid, representing a combination of row and column.

So the workflow of the code is:

The outer loop runs for each value of row from 1 to 3.

For each value of row, the inner loop runs, iterating over values of col from 1 to 3.

Inside the inner loop, the print statement prints the current coordinates of the grid.

The output looks like this:

(1, 1)

(1, 2)

(1, 3)

(2, 1)

(2, 2)

(2, 3)

(3, 1)

(3, 2)

(3, 3)

Each line in the output corresponds to a unique combination of ‘row’ and ‘col’ values in the nested loops.

In conclusion, loops are essential tools in programming that enable the repetition of tasks. The "for" loop simplifies iterating over a range or collection, the "while" loop offers flexibility for indefinite repetition based on conditions, and nested loops combine these concepts for more intricate scenarios. Mastering loops is key to writing efficient

and dynamic code, providing a foundation for solving diverse problems in programming.