**Introduction to Loops in Python**

**Estimated time needed:** 10 minutes

**Objectives**

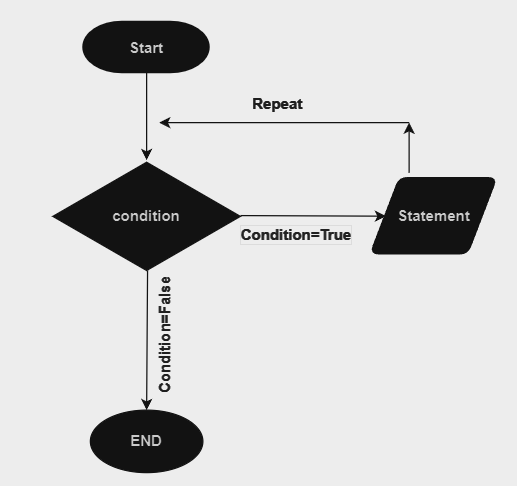
1. Understand Python loops.
2. How the loop Works
3. Learn about the needs for loop
4. Utilize Python's Range function.
5. Familiarize with Python's enumerate function.
6. Apply while loops for conditional tasks.
7. Distinguish appropriate loop selection.

**What is a Loop?**

In programming, a loop is like a magic trick that allows a computer to do something over and over again. Imagine you are a magician's assistant, and your magician friend asks you to pull a rabbit out of a hat, but not just once - they want you to keep doing it until they tell you to stop. That is what loops do for computers - they repeat a set of instructions as many times as needed.

**How Loop works?**

Here's how it works in Python:



* **Start:** The for loop begins with the keyword for, followed by a variable that will take on each value in a sequence.
* **Condition:** After the variable, you specify the keyword in and a sequence, such as a list or a range, that the loop will iterate through.
* If **Condition True**:

1. The loop takes the first value from the sequence and assigns it to the variable.
2. The indented block of code following the loop header is executed using this value.
3. The loop then moves to the next value in the sequence and repeats the process until all values have been used.

* **Statement:** Inside the indented block of the loop, you write the statements that you want to repeat for each value in the sequence.
* **Repeat:** The loop continues to repeat the block of code for each value in the sequence until there are no more values left.
* If **Condition False**:

1. Once all values in the sequence have been processed, the loop terminates automatically.
2. The loop completes its execution, and the program continues to the next statement after the loop.

**The Need for Loops**

Think about when you need to count from 1 to 10. Doing it manually is easy, but what if you had to count to a **million**? Typing all those numbers one by one would be a nightmare! This is where loops come in handy. They help computers repeat tasks quickly and accurately without getting tired.

**Main Types of Loops**

**For Loops**

For loops are like a superhero's checklist. A for loop in programming is a control structure that allows the repeated execution of a set of statements for each item in a sequence, such as elements in a list or numbers in a range, enabling efficient iteration and automation of tasks

**Syntax of for loop**

1. 1
2. 2
3. for val in sequence:
4. # statement(s) to be executed in sequence as a part of the loop.

Copied!

Here is an example of For loop.

Imagine you're a painter, and you want to paint a beautiful rainbow with seven colors. Instead of picking up each color one by one and painting the rainbow, you could tell a magical painter's assistant to do it for you. This is what a basic for loop does in programming.

**We have a list of colours.**

1. 1
2. colors = ["red", "orange", "yellow", "green", "blue", "indigo", "violet"]

Copied!

**Let's print the colour name in the new line using for loop.**

1. 1
2. 2
3. for color in colors:
4. print(color)

Copied!

In this example, the for loop picks each color from the colors list and prints it on the screen. You don't have to write the same code for each color - the loop does it automatically!

Sometimes you do not want to paint a rainbow, but *you want to count the number of steps to reach your goal.* A range-based for loop is like having a friendly step counter that helps you reach your target.  
Here is how you might use a for loop to count from 1 to 10:

1. 1
2. 2
3. for number in range(1, 11):
4. print(number)

Copied!

Here, the **range(1, 11)** generates a sequence from 1 to 10, and the for loop goes through each number in that sequence, printing it out. It's like taking 10 steps, and you're guided by the loop!

**Range Function**

The range function in Python generates an ordered sequence that can be used in loops. It takes one or two arguments:

* If given one argument (e.g., range(11)), it generates a sequence starting from 0 up to (but not including) the given number.

1. 1
2. 2
3. for number in range(11):
4. print(number)

Copied!

* If given two arguments (e.g., range(1, 11)), it generates a sequence starting from the first argument up to (but not including) the second argument.

1. 1
2. 2
3. for number in range(1, 11):
4. print(number)

Copied!

**The Enumerated For Loop**

Have you ever needed to keep track of both the item and its position in a list? An enumerated for loop comes to your rescue. It's like having a personal assistant who not only hands you the item but also tells you where to find it.

Consider this example:

1. 1
2. 2
3. 3
4. fruits = ["apple", "banana", "orange"]
5. for index, fruit in enumerate(fruits):
6. print(f"At position {index}, I found a {fruit}")

Copied!

With this loop, you not only get the fruit but also its position in the list. It's as if you have a magical guide pointing out each fruit's location!

**While Loops**

While loops are like a sleepless night at a friend's sleepover. Imagine you and your friends keep telling ghost stories until someone decides it's time to sleep. As long as no one says, "Let's sleep" you keep telling stories.  
A while loop works similarly - it repeats a task as long as a certain condition is true. It's like saying, "Hey computer, keep doing this until I say stop!"

**Basic syntax of While Loop.**

1. 1
2. 2
3. 3
4. while condition:
5. # Code to be executed while the condition is true
6. # Indentation is crucial to indicate the scope of the loop

Copied!

For example, here's how you might use a while loop to count from 1 to 10:

1. 1
2. 2
3. 3
4. 4
5. count = 1
6. while count <= 10:
7. print(count)
8. count += 1

Copied!

here's a breakdown of the above code.

1. There is a variable named **count** initialized with the value 1.
2. The while loop is used to repeatedly execute a block of code as long as a given condition is True. In this case, the condition is **count <= 10**, meaning the loop will continue as long as count is less than or equal to 10.
3. Inside the loop:
   * The **print(count)** statement outputs the current value of the count variable.
   * The **count += 1** statement increments the value of count by 1. This step ensures that the loop will eventually terminate when count becomes greater than 10.
4. The loop will continue executing as long as the condition count <= 10 is satisfied.
5. The loop will print the numbers 1 to 10 in consecutive order since the print statement is inside the loop block and executed during each iteration.
6. Once count reaches 11, the condition count <= 10 will evaluate to False, and the loop will terminate.
7. The output of the code will be the numbers 1 to 10, each printed on a separate line.

**The Loop Flow**

Both for and while loops have their special moves, but they follow a pattern:

* **Initialization:** You set up things like a starting point or conditions.
* **Condition:** You decide when the loop should keep going and when it should stop.
* **Execution:** You do the task inside the loop.
* **Update:** You make changes to your starting point or conditions to move forward.
* **Repeat:** The loop goes back to step 2 until the condition is no longer true.

**When to Use Each**

**For Loops:** Use for loops when you know the number of iterations in advance and want to process each element in a sequence. They are best suited for iterating over collections and sequences where the length is known.

**While Loops:** Use while loops when you need to perform a task repeatedly as long as a certain condition holds true. While loops are particularly useful for situations where the number of iterations is uncertain or where you're waiting for a specific condition to be met.

**Summary**

In this adventure into coding, we explored loops in Python - special tools that help us do things over and over again without getting tired. We met two types of loops: **"for loops"** and **"while loops."**

**For Loops** were like helpers that made us repeat tasks in order. We painted colors, counted numbers, and even got a helper to tell us where things were in a list. For loops made our job easier and made our code look cleaner.

**While Loops** were like detectives that kept doing something as long as a rule was true. They helped us take steps, guess numbers, and work until we were tired. While loops were like smart assistants that didn't stop until we said so.

**Functions**

**Introduction to functions**

**A function is a fundamental building block that encapsulates specific actions or computations. As in mathematics, where functions take inputs and produce outputs, programming functions perform similarly. They take inputs, execute predefined actions or calculations, and then return an output.**

**Purpose of functions**

**Functions promote code modularity and reusability. Imagine you have a task that needs to be performed multiple times within a program. Instead of duplicating the same code at various places, you can define a function once and call it whenever you need that task. This reduces redundancy and makes the code easier to manage and maintain.**

**Benefits of using functions**

**Modularity: Functions break down complex tasks into manageable components  
Reusability: Functions can be used multiple times without rewriting code  
Readability: Functions with meaningful names enhance code understanding  
Debugging: Isolating functions eases troubleshooting and issue fixing  
Abstraction: Functions simplify complex processes behind a user-friendly interface  
Collaboration: Team members can work on different functions concurrently  
Maintenance: Changes made in a function automatically apply wherever it's used**

**How functions take inputs, perform tasks, and produce outputs**

**Inputs (Parameters)**

**Functions operate on data, and they can receive data as input. These inputs are known as *parameters* or *arguments*. Parameters provide functions with the necessary information they need to perform their tasks. Consider parameters as values you pass to a function, allowing it to work with specific data.**

**Performing tasks**

**Once a function receives its input (parameters), it executes predefined actions or computations. These actions can include calculations, operations on data, or even more complex tasks. The purpose of a function determines the tasks it performs. For instance, a function could calculate the sum of numbers, sort a list, format text, or fetch data from a database.**

**Producing outputs**

**After performing its tasks, a function can produce an output. This output is the result of the operations carried out within the function. It's the value that the function “returns” to the code that called it. Think of the output as the end product of the function's work. You can use this output in your code, assign it to variables, pass it to other functions, or even print it out for display.**

**Example:**

**Consider a function named calculate\_total that takes two numbers as input (parameters), adds them together, and then produces the sum as the output. Here's how it works:**

1. **1**
2. **2**
3. **3**
4. **4**
5. **5**
6. **6**
7. **def calculate\_total(a, b): # Parameters: a and b**
8. **total = a + b # Task: Addition**
9. **return total # Output: Sum of a and b**
10. **result = calculate\_total(5, 7) # Calling the function with inputs 5 and 7**
11. **print(result) # Output: 12**

**Copied!**

**Python's built-in functions**

**Python has a rich set of built-in functions that provide a wide range of functionalities. These functions are readily available for you to use, and you don't need to be concerned about how they are implemented internally. Instead, you can focus on understanding what each function does and how to use it effectively.**

**Using built-in functions or Pre-defined functions**

**To use a built-in function, you simply call the function's name followed by parentheses. Any required arguments or parameters are passed into the function within these parentheses. The function then performs its predefined task and may return an output you can use in your code.**

**Here are a few examples of commonly used built-in functions:**

**len(): Calculates the length of a sequence or collection**

1. **1**
2. **2**
3. **string\_length = len("Hello, World!") # Output: 13**
4. **list\_length = len([1, 2, 3, 4, 5]) # Output: 5**

**Copied!**

**sum(): Adds up the elements in an iterable (list, tuple, and so on)**

1. **1**
2. **total = sum([10, 20, 30, 40, 50]) # Output: 150**

**Copied!**

**max(): Returns the maximum value in an iterable**

1. **1**
2. **highest = max([5, 12, 8, 23, 16]) # Output: 23**

**Copied!**

**min(): Returns the minimum value in an iterable**

1. **1**
2. **lowest = min([5, 12, 8, 23, 16]) # Output: 5**

**Copied!**

**Python's built-in functions offer a wide array of functionalities, from basic operations like len() and sum() to more specialized tasks.**

**Defining your functions**

**Defining a function is like creating your mini-program:**

1. **Use def followed by the function name and parentheses**

**Here is the syntax to define a function:**

1. **1**
2. **2**
3. **def function\_name():**
4. **pass**

**Copied!**

**A "pass" statement in a programming function is a placeholder or a no-op (no operation) statement. Use it when you want to define a function or a code block syntactically but do not want to specify any functionality or implementation at that moment.**

* **Placeholder: "pass" acts as a temporary placeholder for future code that you intend to write within a function or a code block.**
* **Syntax Requirement: In many programming languages like Python, using "pass" is necessary when you define a function or a conditional block. It ensures that the code remains syntactically correct, even if it doesn't do anything yet.**
* **No Operation: "pass" itself doesn't perform any meaningful action. When the interpreter encounters “pass”, it simply moves on to the next statement without executing any code.**

**Function Parameters:**

* **Parameters are like inputs for functions**
* **They go inside parentheses when defining the function**
* **Functions can have multiple parameters**

**Example:**

1. **1**
2. **2**
3. **3**
4. **4**
5. **5**
6. **def greet(name):**
7. **print("Hello, " + name)**
8. **result = greet("Alice")**
9. **print(result) # Output: Hello, Alice**

**Copied!**

**Docstrings (Documentation Strings)**

* **Docstrings explain what a function does**
* **Placed inside triple quotes under the function definition**
* **Helps other developers understand your function**

**Example:**

1. **1**
2. **2**
3. **3**
4. **4**
5. **5**
6. **6**
7. **7**
8. **8**
9. **def multiply(a, b):**
10. **"""**
11. **This function multiplies two numbers.**
12. **Input: a (number), b (number)**
13. **Output: Product of a and b**
14. **"""**
15. **print(a \* b)**
16. **multiply(2,6)**

**Copied!**

**Return statement**

* **Return gives back a value from a function**
* **Ends the function's execution and sends the result**
* **A function can return various types of data**

**Example:**

1. **1**
2. **2**
3. **3**
4. **4**
5. **def add(a, b):**
6. **return a + b**
7. **sum\_result = add(3, 5) # sum\_result gets the value 8**

**Copied!**

**Understanding scopes and variables**

**Scope is where a variable can be seen and used:**

* **Global Scope: Variables defined outside functions; accessible everywhere**
* **Local Scope: Variables inside functions; only usable within that function**

**Example:**

**Part 1: Global variable declaration**

1. **1**
2. **global\_variable = "I'm global"**

**Copied!**

**This line initializes a global variable called global\_variable and assigns it the value "I'm global".**

***Global variables are accessible throughout the entire program, both inside and outside functions.***

**Part 2: Function definition**

1. **1**
2. **2**
3. **3**
4. **4**
5. **def example\_function():**
6. **local\_variable = "I'm local"**
7. **print(global\_variable) # Accessing global variable**
8. **print(local\_variable) # Accessing local variable**

**Copied!**

**Here, you define a function called example\_function().**

**Within this function:**

* **A local variable named local\_variable is declared and initialized with the string value "I'm local." This variable is local to the function and can only be accessed within the function's scope.**
* **The function then prints the values of both the global variable (global\_variable) and the local variable (local\_variable). It demonstrates that you can access global and local variables within a function.**

**Part 3: Function call**

1. **1**
2. **example\_function()**

**Copied!**

**In this part, you call the example\_function() by invoking it. This results in the function's code being executed.  
As a result of this function call, it will print the values of the global and local variables within the function.**

**Part 4: Accessing global variable outside the function**

1. **1**
2. **print(global\_variable) # Accessible outside the function**

**Copied!**

**After calling the function, you print the value of the global variable global\_variable outside the function. This demonstrates that global variables are accessible inside and outside of functions.**

**Part 5: Attempting to access local variable outside the function**

1. **1**
2. **# print(local\_variable) # Error, local variable not visible here**

**Copied!**

**In this part, you are attempting to print the value of the local variable local\_variable outside of the function. However, this line would result in an error.**

***Local variables are only visible and accessible within the scope of the function where they are defined.***

**Attempting to access them outside of that scope would raise a "NameError".**

**Using functions with loops**

**Functions and loops together**

1. **Functions can contain code with loops**
2. **This makes complex tasks more organized**
3. **The loop code becomes a repeatable function**

**Example:**

1. **1**
2. **2**
3. **3**
4. **4**
5. **5**
6. **def print\_numbers(limit):**
7. **for i in range(1, limit+1):**
8. **print(i)**
9. **print\_numbers(5) # Output: 1 2 3 4 5**

**Copied!**

**Enhancing code organization and reusability**

1. **Functions group similar actions for easy understanding**
2. **Looping within functions keeps code clean**
3. **You can reuse a function to repeat actions**

**Example**

1. **1**
2. **2**
3. **3**
4. **4**
5. **5**
6. **def greet(name):**
7. **return "Hello, " + name**
8. **for \_ in range(3):**
9. **print(greet("Alice"))**

**Copied!**

**Modifying data structure using functions**

**You'll use Python and a list as the data structure for this illustration. In this example, you will create functions to add and remove elements from a list.**

**Part 1: Initialize an empty list**

1. **1**
2. **2**
3. **# Define an empty list as the initial data structure**
4. **my\_list = []**

**Copied!**

**In this part, you start by creating an empty list named my\_list. This empty list serves as the data structure that you will modify throughout the code.**

**Part 2: Define a function to add elements**

1. **1**
2. **2**
3. **3**
4. **# Function to add an element to the list**
5. **def add\_element(data\_structure, element):**
6. **data\_structure.append(element)**

**Copied!**

**Here, you define a function called add\_element. This function takes two parameters:**

* **data\_structure: This parameter represents the list to which you want to add an element**
* **element: This parameter represents the element you want to add to the list**

**Inside the function, you use the append method to add the provided element to the data\_structure, which is assumed to be a list.**

**Part 3: Define a function to remove elements**

1. **1**
2. **2**
3. **3**
4. **4**
5. **5**
6. **6**
7. **# Function to remove an element from the list**
8. **def remove\_element(data\_structure, element):**
9. **if element in data\_structure:**
10. **data\_structure.remove(element)**
11. **else:**
12. **print(f"{element} not found in the list.")**

**Copied!**

**In this part, you define another function called remove\_element. It also takes two parameters:**

* **data\_structure: The list from which we want to remove an element**
* **element: The element we want to remove from the list**

**Inside the function, you use conditional statements to check if the element is present in the data\_structure. If it is, you use the remove method to remove the first occurrence of the element. If it's not found, you print a message indicating that the element was not found in the list.**

**Part 4: Add elements to the list**

1. **1**
2. **2**
3. **3**
4. **4**
5. **# Add elements to the list using the add\_element function**
6. **add\_element(my\_list, 42)**
7. **add\_element(my\_list, 17)**
8. **add\_element(my\_list, 99)**

**Copied!**

**Here, you use the add\_element function to add three elements (42, 17, and 99) to the my\_list. These are added one at a time using function calls.**

**Part 5: Print the current list**

1. **1**
2. **2**
3. **# Print the current list**
4. **print("Current list:", my\_list)**

**Copied!**

**This part simply prints the current state of the my\_list to the console, allowing us to see the elements that have been added so far.**

**Part 6: Remove elements from the list**

1. **1**
2. **2**
3. **3**
4. **# Remove an element from the list using the remove\_element function**
5. **remove\_element(my\_list, 17)**
6. **remove\_element(my\_list, 55) # This will print a message since 55 is not in the list**

**Copied!**

**In this part, you use the remove\_element function to remove elements from the my\_list. First, you attempt to remove 17 (which is in the list), and then you try to remove 55 (which is not in the list). The second call to remove\_element will print a message indicating that 55 was not found.**

**Part 7: Print the updated list**

1. **1**
2. **2**
3. **# Print the updated list**
4. **print("Updated list:", my\_list)**

**Copied!**

**Finally, you print the updated my\_list to the console. This allows us to observe the modifications made to the list by adding and removing elements using the defined functions.**

**Exception Handling in Python**

What are exceptions?

Exceptions are alerts when something unexpected happens while running a program. It could be a mistake in the code or a situation that was not planned for. Python can raise these alerts automatically, but we can also trigger them on purpose using the raise command. The cool part is that we can prevent our program from crashing by handling exceptions.

Errors vs. Exceptions

Hold on, what is the difference between errors and exceptions? Well, errors are usually big problems that come from the computer or the system. They often make the program stop working completely. On the other hand, exceptions are more like issues we can control. They happen because of something we did in our code and can usually be fixed, so the program keeps going.

Here is the difference between Errors and exceptions:-

Aspect Errors Exceptions

Origin Errors are typically caused by the environment, hardware, or operating system. Exceptions are usually a result of problematic code execution within the program.

Nature Errors are often severe and can lead to program crashes or abnormal termination. Exceptions are generally less severe and can be caught and handled to prevent program termination.

Handling Errors are not usually caught or handled by the program itself. Exceptions can be caught using try-except blocks and dealt with gracefully, allowing the program to continue execution.

Examples Examples include “SyntaxError” due to incorrect syntax or “NameError” when a variable is not defined. Examples include “ZeroDivisionError” when dividing by zero, or “FileNotFoundError” when attempting to open a non-existent file.

Categorization Errors are not classified into categories. Exceptions are categorized into various classes, such as “ArithmeticError,” “IOError,” ValueError,” etc., based on their nature.

Common Exceptions in Python

Here are a few examples of exceptions we often run into and can handle using this tool:

ZeroDivisionError: This error arises when an attempt is made to divide a number by zero. Division by zero is undefined in mathematics, causing an arithmetic error. For instance:

For example:

1

2

3

result = 10 / 0

print(result)

# Raises ZeroDivisionError

Copied!

ValueError: This error occurs when an inappropriate value is used within the code. An example of this is when trying to convert a non-numeric string to an integer:

For example:

1

2

num = int("abc")

# Raises ValueError

Copied!

FileNotFoundError: This exception is encountered when an attempt is made to access a file that does not exist.

For example:

1

2

with open("nonexistent\_file.txt", "r") as file:

content = file.read() # Raises FileNotFoundError

Copied!

IndexError: An IndexError occurs when an index is used to access an element in a list that is outside the valid index range.

For example:

1

2

3

my\_list = [1, 2, 3]

value = my\_list[1] # No IndexError, within range

missing = my\_list[5] # Raises IndexError

Copied!

KeyError: The KeyError arises when an attempt is made to access a non-existent key in a dictionary.

For example:

1

2

3

my\_dict = {"name": "Alice", "age": 30}

value = my\_dict.get("city") # No KeyError, using .get() method

missing = my\_dict["city"] # Raises KeyError

Copied!

TypeError: The TypeError occurs when an object is used in an incompatible manner. An example includes trying to concatenate a string and an integer:

For example:

1

2

result = "hello" + 5

# Raises TypeError

Copied!

AttributeError: An AttributeError occurs when an attribute or method is accessed on an object that doesn't possess that specific attribute or method. For instance:

For example:

1

2

3

text = "example"

length = len(text) # No AttributeError, correct method usage

missing = text.some\_method() # Raises AttributeError

Copied!

ImportError: This error is encountered when an attempt is made to import a module that is unavailable. For example: import non\_existent\_module

Note: Please remember, the exceptions you will encounter are not limited to just these. There are many more in Python. However, there is no need to worry. By using the technique provided below and following the correct syntax, you will be able to handle any exceptions that come your way.

Handling Exceptions:

Python has a handy tool called try and except that helps us manage exceptions.

Try and Except : You can use the try and except blocks to prevent your program from crashing due to exceptions.

Here's how they work:

The code that may result in an exception is contained in the try block.

If an exception occurs, the code directly jumps to except block.

In the except block, you can define how to handle the exception gracefully, like displaying an error message or taking alternative actions.

After the except block, the program continues executing the remaining code.

Example: Attempting to divide by zero

# using Try- except

try:

# Attempting to divide 10 by 0

result = 10 / 0

except ZeroDivisionError:

# Handling the ZeroDivisionError and printing an error message

print("Error: Cannot divide by zero")

# This line will be executed regardless of whether an exception occurred

print("outside of try and except block")