

✓ DSML-Python Refresher 2

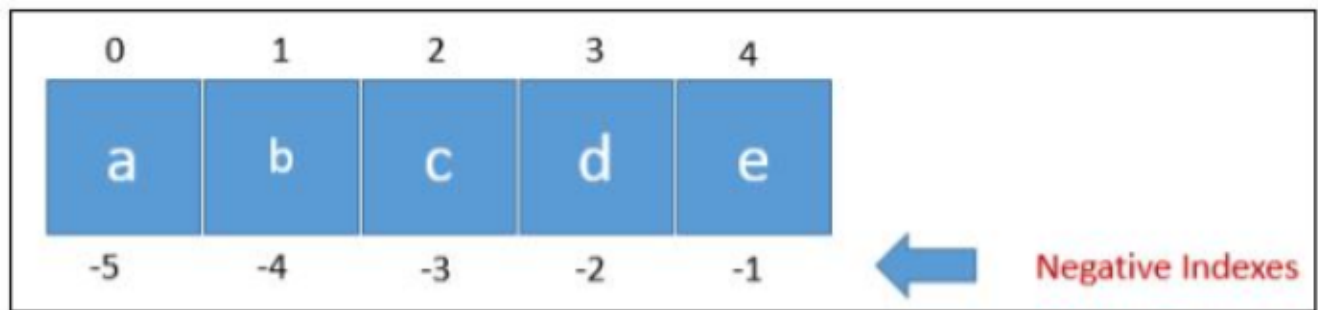
Agenda

1. Slicing
2. Control Flow
3. Loops
4. Lists
5. Strings

✓ Slicing

✓ What Is an Index?

An index is a single character or element's location within a string, tuple, or list. Regardless of the number of items, the index value always begins at zero and ends at one less.

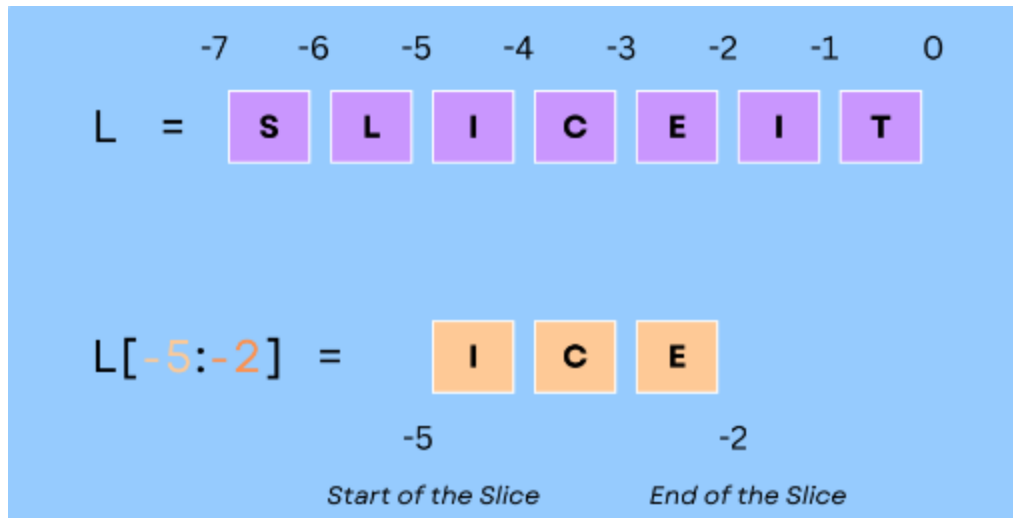


Negative indexes enable users to index a list, tuple, or other indexable containers from the end of the container, rather than the start.

✓ What Is Slicing?

Slicing is a powerful feature in Python that allows you to extract specific parts of sequences such as strings, lists, or tuples by specifying a range of indices. The basic syntax for slicing is:

Syntax: object[start:stop:step]



- **start:** The starting index of the slice. The element at this index is included in the slice.
- **stop:** The ending index of the slice. The element at this index is not included in the slice.
- **step:** The interval between elements in the slice. This parameter is optional and defaults to 1.

Here are some examples demonstrating how slicing works:

```
my_list = [0, 1, 2, 3, 4, 5, 6]
sliced_list = my_list[1:5]
print(sliced_list) # Output: [1, 2, 3, 4]
```

⇒ [1, 2, 3, 4]

✓ Slicing with steps

```
my_list = [0, 1, 2, 3, 4, 5, 6]
sliced_list = my_list[1:6:2]
print(sliced_list) # Output: [1, 3, 5]
```

⇒ [1, 3, 5]

✓ Omitting Start and Stop:

```
my_list = [0, 1, 2, 3, 4, 5, 6]
sliced_list = my_list[:4]
print(sliced_list) # Output: [0, 1, 2, 3]
```

```
sliced_list = my_list[2:]  
print(sliced_list) # Output: [2, 3, 4, 5, 6]
```

```
sliced_list = my_list[::2]  
print(sliced_list) # Output: [0, 2, 4, 6]
```

```
⇒ [0, 1, 2, 3]  
   [2, 3, 4, 5, 6]  
   [0, 2, 4, 6]
```

✓ Negative Indices

```
my_list = [0, 1, 2, 3, 4, 5, 6]  
sliced_list = my_list[-5:-2]  
print(sliced_list) # Output: [2, 3, 4]
```

```
sliced_list = my_list[::-1]  
print(sliced_list) # Output: [6, 5, 4, 3, 2, 1, 0]
```

```
⇒ [2, 3, 4]  
   [6, 5, 4, 3, 2, 1, 0]
```

Slicing can be very useful for various tasks such as extracting subarrays, manipulating substrings, and more. The ability to use negative indices and steps provides additional flexibility in accessing and manipulating sequence data in Python.

✓ Common Mistakes to Avoid

- **Exclusive End Index:** Remember that the end index is exclusive. For instance, `my_list[1:5]` includes elements from index 1 to 4, not 5.
- **Negative Indices Misunderstanding:** Confusion may arise when mixing positive and negative indices. Always double-check your indices to ensure they reference the intended elements.

✓ Practical Use Cases

- **Extracting Subarrays:** Useful in data manipulation, especially in libraries like NumPy for slicing arrays.
- **Manipulating Substrings:** For string operations, slicing can be used to extract parts of a string, such as domain names from URLs.

- **Creating Resized Lists:** Use slicing to create sublists for analysis in data processing tasks.

✓ Interactive Exercises

- **Exercise 1:**

- Given the list `my_list = [10, 20, 30, 40, 50, 60, 70]`, write a slicing operation to extract the elements from index 2 to 5.
- Expected Output: `[30, 40, 50]`

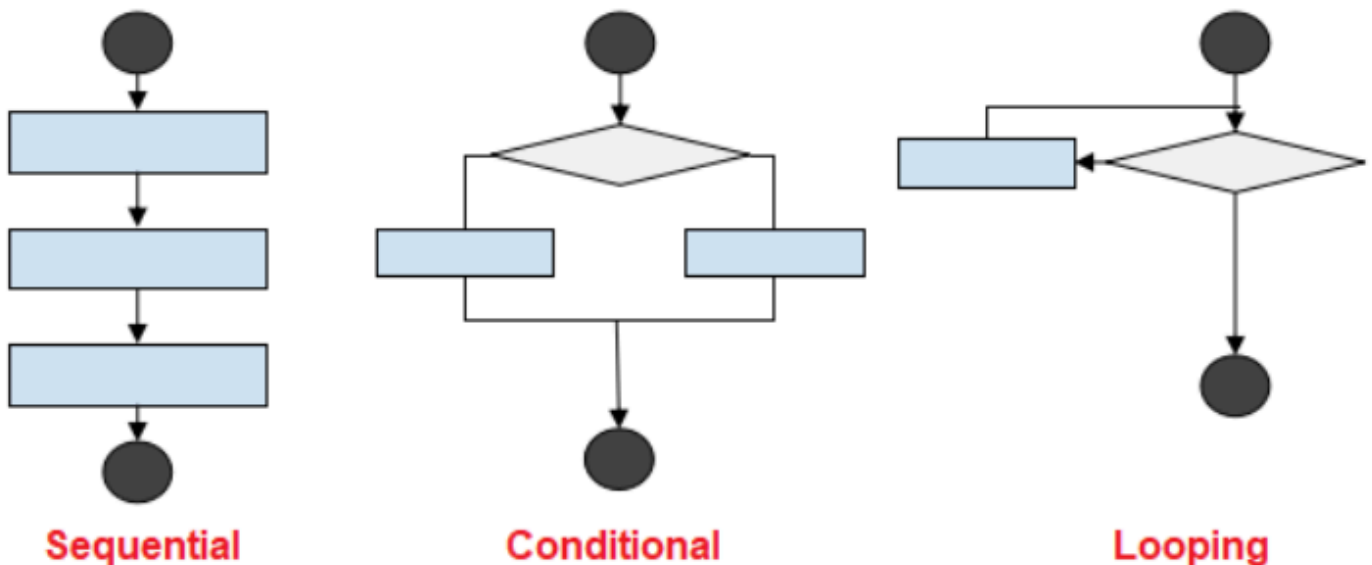
- **Exercise 2:**

- How would you reverse the list `my_list` using slicing?
- Expected Code: `my_list[::-1]`

- **Exercise 3:**

- What will be the output of `my_list[1:6:3]` for the list `my_list = [1, 2, 3, 4, 5, 6, 7, 8]`?
- Expected Output: `[2, 5]`


✓ Control Flow



✓ 1. Sequential Execution

- In this type of execution flow, the statements are executed one after the other sequentially.
- By using sequential statements, we can develop simple programs.

```
print("111")  
print("111")  
print("111")
```

 111
111
111

✓ Conditional Execution

- Statements are executed based on the condition. As shown above in the flow graph, if the condition is true then one set of statements are executed, and if false then the other set. Conditional statements are used much in complex programs.

✓ Basic Structure:

if :

```
# IF BLOCK executes if the condition is true
```

else:

```
# ELSE BLOCK executes if the condition is false
```

```
x = 60  
if x < 50:  
    print("x is less than 50")  
else:  
    print("x is 50 or more")
```

✓ Practical Use Cases

- **User Input Validation:** Checking if user-provided data meets certain criteria (e.g., password strength).
- **Menu Selection:** Executing different functions based on user choices.

✓ Common Mistakes

- **Incorrect Indentation:** Python relies on indentation to define blocks of code. Ensure consistent indentation to avoid errors.
- **Using = instead of ==:** When comparing values, use == for equality, not = which is for assignment.

✓ Examples of Conditional Statements

```
x = 60
if x < 50:
    print("random")
    y = 10
    y += 5
    print(y)

print("After the if block!")
```

⇒ After the if block!

```
print("0")

if 50 > 40:
    print("1")
    print("2")
    print("3")
```

```
print("4")
```

⇒ 0
1
2
3
4

```
print("0")

if 50 < 40:
    print("1")
    print("2")
    print("3")
```

```
print("4")
```

⇒ 0
4

```
...
```

LINES BEFORE THE IF-ELSE BLOCK WILL ALWAYS EXECUTE.

```
if <condition>:
    IF BLOCK EXECUTES IF THE CONDITION IS TRUE.
    SECOND LINE.
    THIRD LINE.
else:
    ELSE BLOCK EXECUTES IF THE CONDITION IS FALSE.
    SECOND LINE.
    THIRD LINE.
```

LINES AFTER THE IF-ELSE BLOCK WILL ALWAYS EXECUTE.

```
...
```

```

↳ '\nLINES BEFORE THE IF-ELSE BLOCK WILL ALWAYS EXECUTE.\n\nif <condition>:\n    IF BLOCK\n    EXECUTES IF THE CONDITION IS TRUE.\n    SECOND LINE.\n    THIRD LINE.\nelse:\n    ELSE\n    BLOCK EXECUTES IF THE CONDITION IS FALSE.\n    SECOND LINE.\n    THIRD LINE.\n\nLINES A\n    FTER THE IF-ELSE BLOCK WILL ALWAYS EXECUTE.\n'
```

```
...
```

Question 1 -

1. Take an integer as input from the user.
2. Calculate cube of the input.
3. If result > 50 print "Massive Number" else print "Small Number".
4. After this, print "EXECUTION COMPLETE!"

```
...
```

```

↳ '\nQuestion 1 -\n1. Take an integer as input from the user.\n2. Calculate cube of the i\n    nput.\n3. If result > 50 print "Massive Number" else print "Small Number".\n4. After th\n    is, print "EXECUTION COMPLETE!"\n'
```

```
n = int(input())
```

```
result = n * 3
```

```
if result > 50:
    print("Massive Number!")
else:
    print("Small Number!")
```

```
print("EXECUTION COMPLETE!")
```

```

↳ 44
    Massive Number!
    EXECUTION COMPLETE!
```

```
...
```

Question 2 -

1. Take a password as input from User.
2. Take another input to verify password.
3. Print "Successfully Verified!" if both passwords are same.

```
4. Print "Verification Failed!" otherwise.
...
```

```
➞ '\nQuestion 2 -\n1. Take a password as input from User.\n2. Take another input to verify password.\n3. Print "Successfully Verified!" if both passwords are same.\n4. Print "Verification Failed!" otherwise \n'
```

```
password_1 = input("Enter your password - ")
password_2 = input("Enter your password again - ")
```

```
if password_1 == password_2:
    print("Successfully Verified!")
else:
    print("Verification Failed!")
```

```
➞ Enter your password - 456456
Enter your password again - 456456
Successfully Verified!
```

```
time = int(input())
```

```
if time > 9 and time <= 12:
    print("Good Morning :)")
else:
    if time > 12 and time <= 17:
        print("Good Afternoon :|")
    else:
        if time > 17 and time <= 20:
            print("Good Evening :(")
        else:
            print("Good Night >:(")
```

```
➞ 5
Good Night >:(
```

```
time = int(input())
```

```
if time > 9 and time <= 12:
    print("Good Morning :)")
elif time > 12 and time <= 17:
    print("Good Afternoon :)")
elif time > 17 and time <= 20:
    print("Good Evening :)")
else:
    print("Good Night :)")
```

```
➞ 8
Good Night :)
```


✓ Pass statement

pass in Python is a null statement which the interpreter simply 'moves over' while executing the code. It can be used in a variety of scenarios where you want to leave parts of your code undefined (such as if-else, loops, classes or methods) but the Python syntax doesn't allow it.

```
x = 10
```

```
if x % 4 == 0:
```

```
File "<ipython-input-27-502165dfd7bf>", line 3
    if x % 4 == 0:
        ^
SyntaxError: incomplete input
```

```
x = 10
```

```
if x % 4 == 0:
    pass
```

```
for i in range(10):
```

```
File "<ipython-input-29-e61162350cc4>", line 1
    for i in range(10):
        ^
SyntaxError: incomplete input
```

```
for i in range(10):
    pass
```

✓ Error Handling

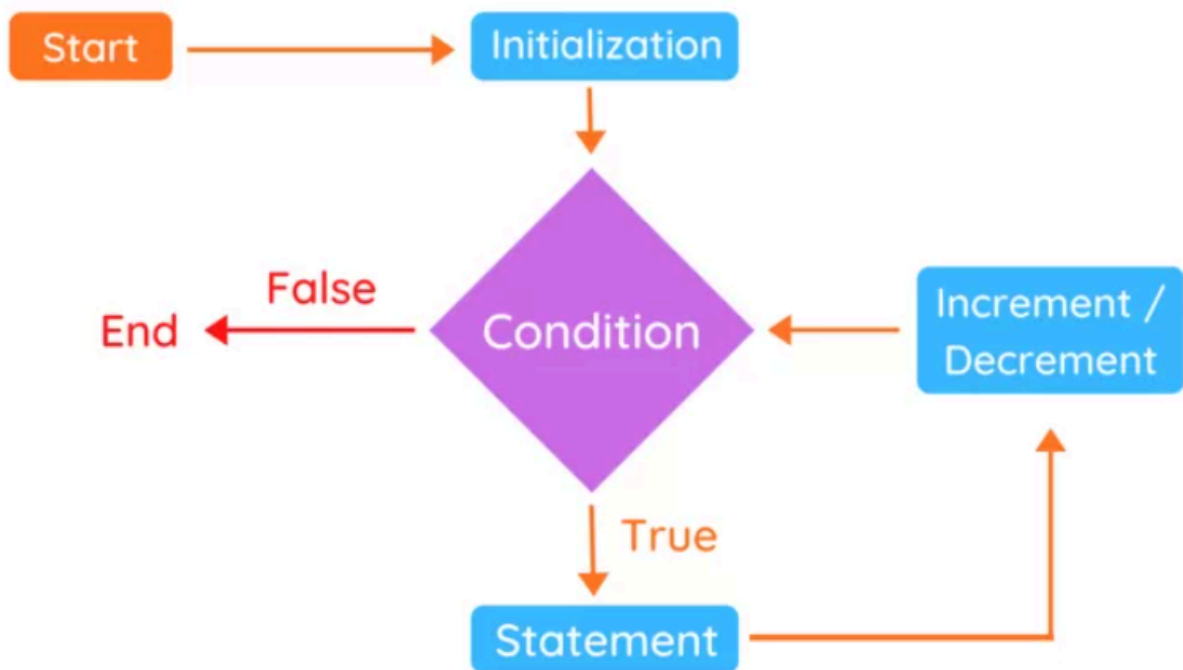
- Control flow can be combined with error handling using try and except statements. This allows programs to manage exceptions gracefully.

✓ Example of Error Handling:

```
try:  
    n = int(input("Enter an integer: "))  
except ValueError:  
    print("That's not an integer!")
```

✓ Loops

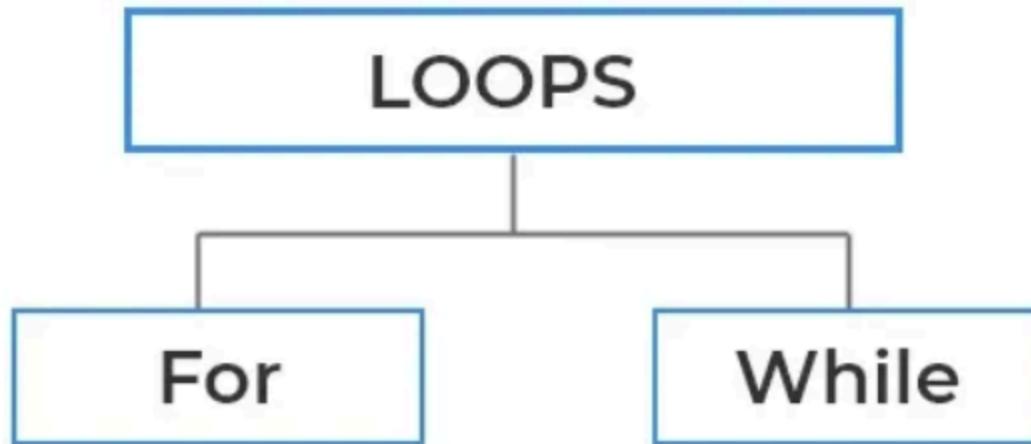
Loops are tools that allow you to repeat a block of code multiple times. They are helpful when you need to perform the same action or set of actions many times, such as iterating over items in a list or repeating an operation until a certain condition is met.



✓ Types of Loops

Python has two main types of loops: **for loops** and **while loops**.

LOOPS IN PYTHON

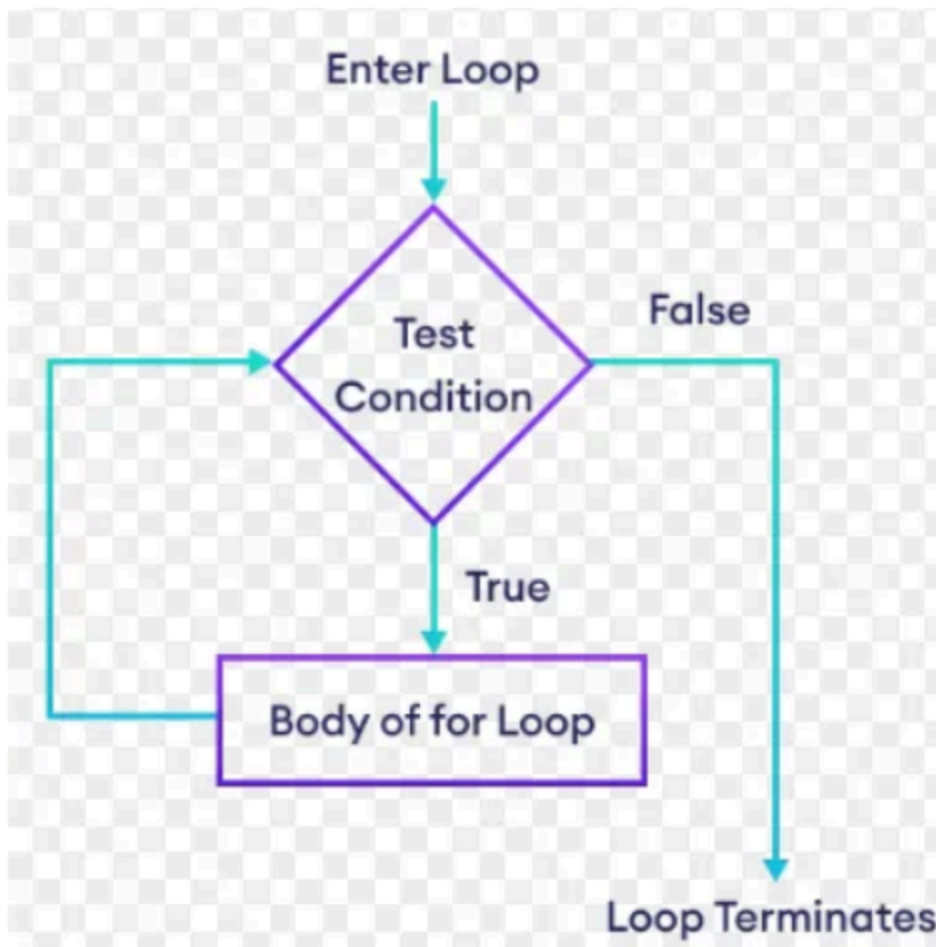


✓ For Loops

A for loop is a type of loop that is used to iterate over a sequence of values. In Python, a sequence can be any iterable object, such as a list, tuple, or string. The basic syntax for a for loop in Python is as follows:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

```
↪ apple
   banana
   cherry
```

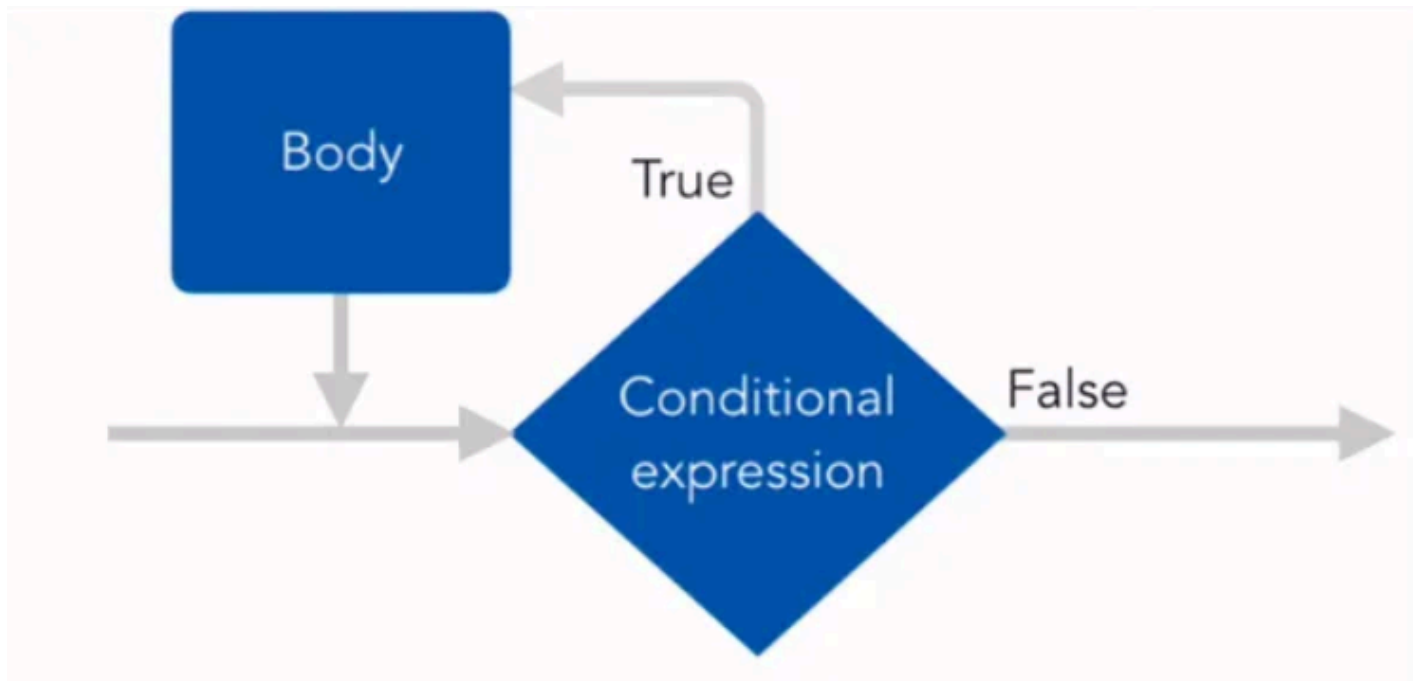


✓ While loops

A while loop is another type of loop in Python. Unlike a for loop, which iterates over a sequence of values, a while loop repeatedly executes a block of code as long as a certain condition is true. The basic syntax for a while loop in Python is as follows:

```
count = 1
while count <= 3:
    print(count)
    count += 1
```

```
➡ 1
   2
   3
```



✓ For loop Vs While loop

For vs While Loop

Comparison Chart

| For Loop | While Loop |
|---|--|
| The for loop is used for definite loops when the number of iterations is known. | The while loop is used when the number of iterations is not known. |
| For loops can have their counter variables declared in the declaration itself. | There is no built-in loop control variable with a while loop. |
| This is preferable when we know exactly how many times the loop will be repeated. | The while loop will continue to run infinite number of times until the condition is met. |
| The loop iterates infinite number of times if the condition is not specified. | If the condition is not specified, it shows a compilation error. |

✓ Nested loop

A nested loop in Python is a loop inside another loop. The "outer" loop controls the number of times the "inner" loop executes. Nested loops are useful when you need to perform repeated actions on multiple dimensions, such as iterating over a matrix or grid.

Example of a Nested Loop

Here's an example of a nested loop using for loops:

```
# Nested for loop example
for i in range(3): # Outer loop
    for j in range(2): # Inner loop
        print(f"i: {i}, j: {j}")
```

```
⇒ i: 0, j: 0
   i: 0, j: 1
   i: 1, j: 0
   i: 1, j: 1
   i: 2, j: 0
   i: 2, j: 1
```

Example of a Nested Loop with a List of Lists (Matrix)

Here's an example where nested loops are used to iterate over a list of lists (a matrix):

```
# Define a 2D list (matrix)
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Nested for loop to iterate over the matrix
for row in matrix: # Outer loop iterates over each row
    for element in row: # Inner loop iterates over each element in the row
        print(element, end=" ")
    print() # Print a new line after each row
```

```
⇒ 1 2 3
   4 5 6
   7 8 9
```

Example of a Nested while Loop

Here's an example using nested while loops:

```
# Nested while loop example
i = 0
while i < 3: # Outer loop
    j = 0
    while j < 2: # Inner loop
        print(f"i: {i}, j: {j}")
        j += 1
    i += 1
```

```
⇒ i: 0, j: 0  
  i: 0, j: 1  
  i: 1, j: 0  
  i: 1, j: 1  
  i: 2, j: 0  
  i: 2, j: 1
```

✓ Break Statement

The break statement in Python is used to exit a loop prematurely. It allows you to terminate the current loop and resume execution at the next statement following the loop. The break statement can be used in both for and while loops.

```
for val in sequence:
```

```
    # code
```

```
    if condition:
```

```
        break
```

```
    # code
```

```
while condition:
```

```
    # code
```

```
    if condition:
```

```
        break
```

```
    # code
```

Usage: The break statement is utilized within a loop, often accompanied by a conditional statement (such as an if statement) that checks a specific condition.

Purpose: When the `break` statement is executed, the loop is immediately terminated, and the program proceeds with the next statement after the loop.

Example:

```
for i in range(5):  
    if i == 3:  
        break
```

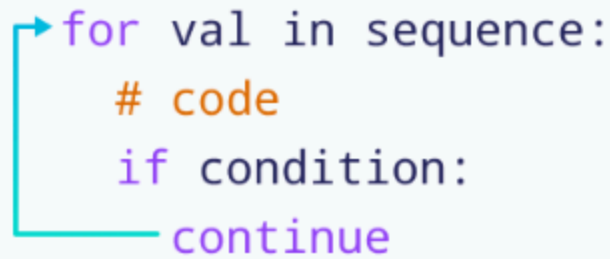
In this example, the loop stops when `i` equals 3, and the remaining numbers (4 and 5) are not iterated over. The `break` statement is a valuable tool for controlling the flow of loops and prematurely exiting a loop when a certain condition is met.

```
for i in range(10):  
    print(i)  
  
    if i == 5:  
        break  
  
    print("After the if")  
  
print("\nOutside the loop")
```

```
⇒ 0  
   After the if  
   1  
   After the if  
   2  
   After the if  
   3  
   After the if  
   4  
   After the if  
   5  
  
   Outside the loop
```

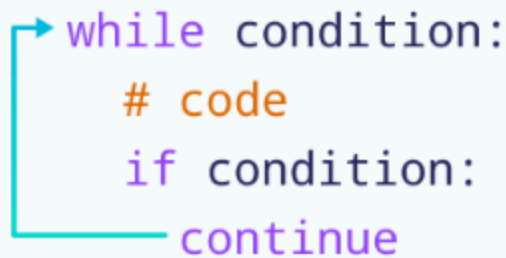
✓ Continue Statement

Skips an iteration. Works with both for loops and while loops



```
for val in sequence:
    # code
    if condition:
        continue
```

```
# code
```



```
while condition:
    # code
    if condition:
        continue
```

```
# code
```

Usage: The `continue` statement is employed within a loop, typically within an `if` statement that checks a specific condition.

Purpose: When the `continue` statement is executed, the current iteration is prematurely terminated, and the loop immediately proceeds to the next iteration.

Example:

```
for i in range(5):
    if i == 3:
        continue
```

In this example, when `i` equals 3, the current iteration is skipped, and the loop continues with the next value of `i` (4 and 5). The `continue` statement is a useful tool for controlling the flow of loops and skipping specific iterations based on a condition.

```
for i in range(1, 10):
    if i == 7:
        continue
```

```
print(i, end=' ')
```

```
➡ 1 2 3 4 5 6 8 9
```

✓ Example Question

- Write a program that continuously asks the user to provide an input number.
- Print the square of that input number.
- The program should stop if the user enters 5.
- In case user enters any multiple of 4, skip it.

```
while True:
    x = int(input())

    if x == 5:
        break

    if x % 4 == 0:
        continue

    print(x ** 2)
```

✓ Interactive Exercise

• Exercise 1:

- Write a program that continuously asks the user to provide an input number.
- Print the square of that input number.
- The program should stop if the user enters 5. In case the user enters any multiple of 4, skip it.

• Exercise 2:

- Write a for loop that iterates over a list of numbers and prints only the even numbers.

✓ Error Handling in Loops

Combining loops with error handling using try and except can help manage exceptions and ensure the program continues to run smoothly.

Example:

```
while True:
    try:
        num = int(input("Enter a number: "))
        print("Square:", num ** 2)
    except ValueError:
        print("Invalid input, please enter a number.")
```

✓ Pattern Problem (Nested Loops)

```
for j in range(5):
    print("*", end='')

```

⇒ *****

```
for j in range(5):
    print("*", end='')

```

```
print()
```

```
for j in range(5):
    print("*", end='')

```

```
print()
```

```
for j in range(5):
    print("*", end='')

```

⇒ *****


```
for i in range(3):
    for j in range(5):
        print("*", end='')
    print()
```

⇒ *****

Question: Write a program to print a N x M matrix of *

N = 4 # no of rows M = 2 # no of cols

★★

★★

**

**

```
N = int(input())
M = int(input())

for i in range(N):
    for j in range(M):
        print('*', end='')
    print()
```

⇒ 6
10


```
M = 2
for j in range(M):
    print('*', end='')
```

⇒ **

✓ Stair Pattern

N = 5

```
for i in range(1, N+1):
    for j in range(i):
        print("*", end='')
    print()
```

⇒ *
**

✓ Reverse Stair Pattern

N = 15

```
for i in range(1, N+1):
```


✓ Hollow Square Pattern:

- Create a hollow square where only the borders are *.

```
*****
*   *
*   *
*   *
*****
```

✓ Python Lists

A list is a versatile data structure that allows you to store a collection of items in a single variable. Lists can hold items of different types and are mutable, meaning their contents can be changed after creation. Here's a comprehensive guide to Python lists:

| List Operation | Description | Example | Output |
|-----------------|---|---|-------------------------------|
| append() | Used to add an element to the end of the list. | <code>my_list = [1, 2, 3]</code> <code>my_list.append(4)</code> | <code>[1, 2, 3, 4]</code> |
| pop() | Used to remove and return the last element of the list. | <code>my_list = [1, 2, 3]</code> <code>last_element = my_list.pop()</code> | <code>last_element = 3</code> |
| len() | Used to get the length of the list. | <code>my_list = [1, 2, 3]</code> <code>length = len(my_list)</code> | <code>length = 3</code> |
| sort() | Used to sort the list in ascending order. | <code>my_list = [3, 1, 2]</code> <code>my_list.sort()</code> | <code>[1, 2, 3]</code> |

✓ Creating a List

```
# Empty list
empty_list = []

# List of integers
int_list = [1, 2, 3, 4, 5]

# List of strings
str_list = ["apple", "banana", "cherry"]

# List of mixed data types
mixed_list = [1, "apple", 3.14, True]
```

✓ Property of List

- **1. Ordered**

Lists maintain the order of elements. The order in which elements are added to the list is preserved.

- **2. Mutable**

Lists are mutable, meaning that their elements can be changed after the list has been created.

- **3. Dynamic Size**

Lists can grow or shrink in size as elements are added or removed.

- **4. Allows Duplicate Elements**

Lists can contain duplicate elements.

- **5. Indexed**

Elements in a list are accessed by their index, starting from 0.

- **6. Allows Mixed Data Types**

Lists can contain elements of different data types.

- **7. Supports Nesting**

Lists can contain other lists (or other complex objects), allowing for nested data structures.

- **8. Supports Slicing**

Lists support slicing, which allows you to extract a portion of the list.

- **9. Iterable**

Lists are iterable, meaning you can loop through the elements using a for loop or other iteration methods.

- **10. Supports List Comprehensions**

Lists support list comprehensions, which provide a concise way to create lists.

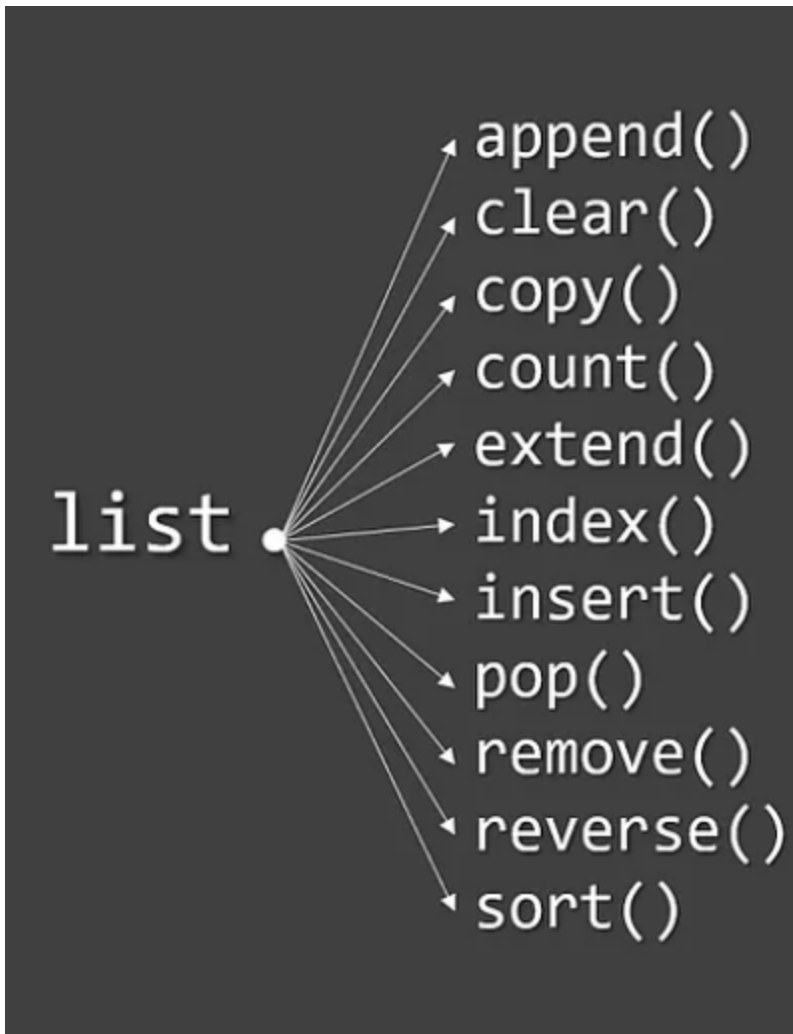
- **11. Variable Length**

Lists can have any number of elements, from zero to many.

- **12. Supports Insertion and Deletion**

You can insert or delete elements at specific positions in a list.

✓ List Method



Python provides various built-in methods to work with lists:

- **append():** Adds an item to the end of the list.

```
fruits = ["apple", "banana", "cherry"]
```

```
fruits.append("date")
```

```
print(fruits) # Output: ["apple", "blueberry", "cherry", "date"]
```

```
➞ ['apple', 'banana', 'cherry', 'date']
```

- **insert():** Inserts an item at a specified position.

```
fruits.insert(1, "banana")
```

```
print(fruits) # Output: ["apple", "banana", "blueberry", "cherry", "date"]
```

```
➞ ['apple', 'banana', 'banana', 'cherry', 'date']
```

- **remove()**: Removes the first occurrence of an item.

```
fruits.remove("banana")  
print(fruits) # Output: ["apple", "blueberry", "cherry", "date"]
```

```
➞ ['apple', 'banana', 'cherry', 'date']
```

- **pop()**: Removes and returns the item at a specified position (or the last item if no index is specified).

```
fruit = fruits.pop(1)  
print(fruit) # Output: blueberry  
print(fruits) # Output: ["apple", "cherry", "date"]
```

```
➞ banana  
['apple', 'cherry', 'date']
```

- **clear()**: Removes all items from the list.

```
fruits.clear()  
print(fruits) # Output: []
```

- **sort()**: Sorts the list in ascending order.

```
numbers = [5, 2, 9, 1]  
numbers.sort()  
print(numbers) # Output: [1, 2, 5, 9]
```

```
➞ [1, 2, 5, 9]
```

- **reverse()**: Reverses the order of the list.

```
numbers.reverse()  
print(numbers) # Output: [9, 5, 2, 1]
```

```
➞ [9, 5, 2, 1]
```

- **index()**: Returns the index of the first occurrence of an item.

```
index = fruits.index("cherry")  
print(index) # Output: 1
```

➞ 1

- **count()**: Returns the number of occurrences of an item.

```
count = fruits.count("apple")  
print(count) # Output: 1
```

➞ 1

- **extend()**: Extends the list by appending elements from another list.

```
fruits.extend(["fig", "grape"])  
print(fruits) # Output: ["apple", "cherry", "date", "fig", "grape"]
```

➞ ['apple', 'cherry', 'date', 'fig', 'grape']

✓ Nested List

Lists can contain other lists:

```
nested_list = [[1, 2, 3], ["apple", "banana"], [True, False]]  
print(nested_list[1][0]) # Output: apple
```

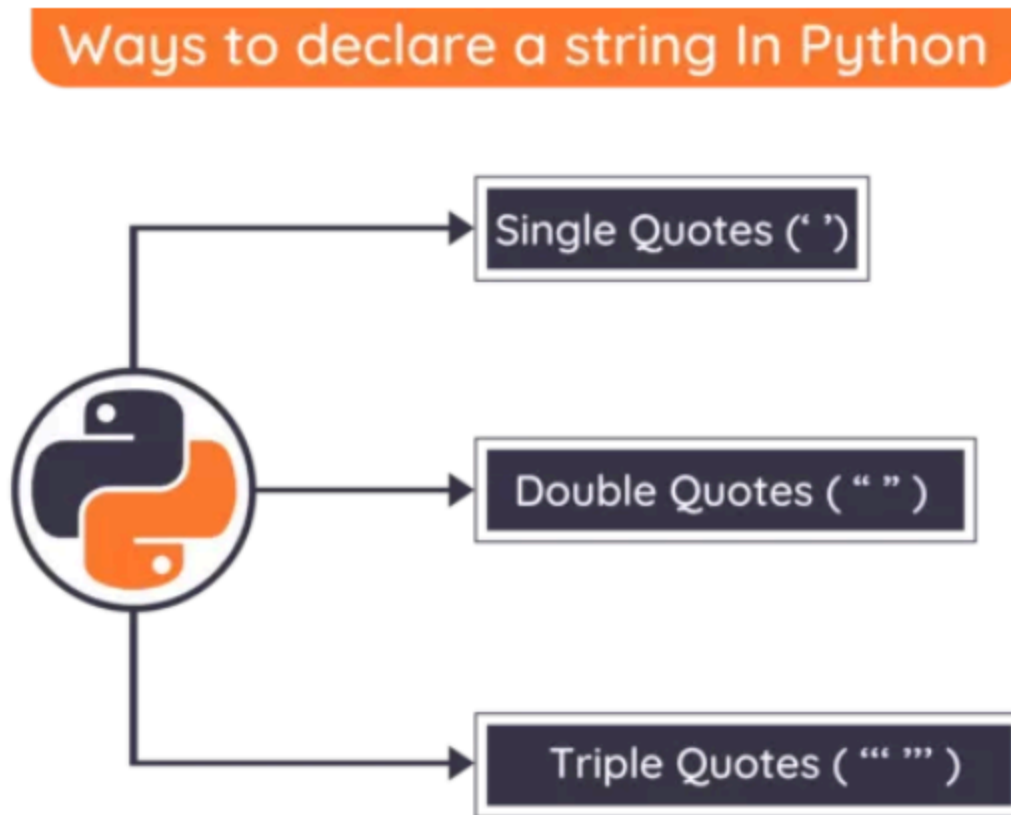
➞ apple

✓ Python Strings

Strings are a sequence of characters used to store and manipulate text. Here are the key properties and operations associated with strings in Python:

Creating a String

Strings can be created using single quotes, double quotes, or triple quotes (for multi-line strings):



```
single_quote_str = 'Hello, World!'
double_quote_str = "Hello, World!"
triple_quote_str = '''Hello,
World!'''
```

✓ Properties of Strings

- **Immutable**

Strings cannot be changed after they are created. Any operation that modifies a string will create a new string.

- **Ordered**

Strings maintain the order of characters. The order in which characters are added to the string is preserved.

- **Indexed**

Characters in a string can be accessed by their index, starting from 0.

- **Iterable**

Strings are iterable, meaning you can loop through each character in the string.

✓ Common String Methods

- **len()** - Returns the length of the string.

```
len("Hello") # Output: 5
```

↩ 5