Loops

Loops: Loops allow us to execute a block of code several times.

While Loop: Allows us to execute a block of code several times as long as the condition is True.

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

```
# Output is:
2
3
```

For Loop: for statement iterates over each item of a sequence.

Syntax:

```
for each_item in sequence:
    block of code
```

Range: Generates a sequence of integers starting from 0. Stops before n (n is not included).

Syntax: `range(n)`

```
for number in range(3):
    print(number)
```

```
# Output is:
0
1
2
```

Range with Start and End: Generates a sequence of numbers starting from the start. Stops before the end (the end is not included).
Syntax: `range(start, end)`

```
for number in range(5, 8):
    print(number)
```

```
# Output is:
5
6
7
```

Lists - Working with Lists

List: List is the most versatile python data structure. Holds an ordered sequence of items.

Accessing List Items: To access elements of a list, we use Indexing.

```
list_a = [5, "Six", 2, 8.2]
print(list_a[1]) # Six
```

Iterating Over a List:

```
list_a = [5, "Six", 8.2]
for item in list_a:
print(item)
```

```
# Output is:
5
Six
8.2
```

List Concatenation: Similar to strings, + operator concatenates lists.

```
list_a = [1, 2]
list_b = ["a", "b"]
list_c = list_a + list_b
print(list_c) # [1, 2, 'a', 'b']
```

List Slicing: Obtaining a part of a list is called List Slicing.

```
list_a = [5, "Six", 2]
list_b = list_a[:2]
print(list_b) # [5, 'Six']
```

Extended Slicing: Similar to string extended slicing, we can extract alternate items using the step.

```
list_a = ["R", "B", "G", "O", "W"]
list_b = list_a[0:5:3]
print(list_b) # ['R', 'O']
```

Reversing a List: `-1` for step will reverse the order of items in the list.

```
list_a = [5, 4, 3, 2, 1]
list_b = list_a[::-1]
print(list_b) # [1, 2, 3, 4, 5]
```

Slicing With Negative Index: You can also specify negative indices while slicing a List.

```
list_a = [5, 4, 3, 2, 1]
list_b = list_a[-3:-1]
print(list_b) # [3, 2]
```

Negative Step Size: Negative Step determines the decrement between each index for slicing. The start index should be greater than the end index in this case

```
list_a = [5, 4, 3, 2, 1]
list_b = list_a[4:2:-1]
print(list_b) # [1, 2]
```

Membership check-in lists:

Name Usage

in By using the `in` operator, one can determine if a value is present in a sequence or not. not in By using the, `not in` operator, one can determine if a value is not present in a sequence or not.

Nested Lists: A list as an item of another list.

Accessing Nested List:

```
list_a = [5, "Six", [8, 6], 8.2]
print(list_a[2]) # [8, 6]
```

Accessing Items of Nested List:

```
list_a = [5, "Six", [8, 6], 8.2]
print(list_a[2][0]) # 8
```

List Methods

Name	Syntax	Usage
append()	list.append(value)	Adds an element to the end of the list.
extend()	list_a.extend(list_b)	Adds all the elements of a sequence to the end of the list.
insert()	list.insert(index,value)	Element is inserted to the list at specified index.
pop()	list.pop()	Removes last element.
remove()	list.remove(value)	Removes the first matching element from the list.
clear()	list.clear()	Removes all the items from the list.
index()	list.index(value)	Returns the index at the first occurrence of the specified value.
count()	list.count(value)	Returns the number of elements with the specified value.
sort()	list.sort()	Sorts the list.
copy()	list.copy()	Returns a new list. It doesn't modify the original list.

Functions

Functions: Block of reusable code to perform a specific action.

Defining a Function: Function is uniquely identified by the `function_name`.

```
def function_name():
    reusable code
```

Calling a Function: The functional block of code is executed only when the function is called.

```
def function_name():
    reusable code
function_name()
```

```
def sum_of_two_number(a, b):
    print(a + b) # 5

sum_of_two_number(2, 3)
```

Function With Arguments: We can pass values to a function using an Argument.

```
def function_name(args):
    reusable code
function_name(args)
```

Returning a Value: To return a value from the function use return keyword. Exits from the function when return statement is executed.

```
def function_name(args):
    block of code
return msg
function_name(args)
```

```
def sum_of_two_number(a, b):
    total = a + b
return total

result = sum_of_two_number(2, 3)
print(result) # 5
```

Function Arguments: A function can have more than one argument.

```
def function_name(arg_1, arg_2):
    reusable code

function_name(arg_1, arg_2)
```

Keyword Arguments: Passing values by their names.

```
def greet(arg_1, arg_2):
    print(arg_1 + " " + arg_2) # Good Morning Ram

greet(arg_1="Good Morning", arg_2="Ram")
```

Positional Arguments: Values can be passed without using argument names. These values get assigned according to their position. Order of the arguments matters here.

```
def greet(arg_1, arg_2):
    print(arg_1 + " " + arg_2) # Good Morning Ram

greeting = input() # Good Morning
name = input() # Ram
greet(greeting, name)
```

Default Values:

```
def greet(arg_1="Hi", arg_2="Ram"):
    print(arg_1 + " " + arg_2) # Hi Ram

greeting = input() # Hello
name = input() # Teja
greet()
```

Arbitrary Function Arguments: We can define a function to receive any number of arguments.

Variable Length Arguments: Variable length arguments are packed as tuple.

```
def more_args(*args):
  print(args) # (1, 2, 3, 4)

more_args(1, 2, 3, 4)
```

Unpacking as Arguments: If we already have the data required to pass to a function as a sequence, we can unpack it with * while passing.

```
def greet(arg1="Hi", arg2="Ram"):
    print(arg1 + " " + arg2) # Hello Teja

data = ["Hello", "Teja"]
greet(*data)
```

Multiple Keyword Arguments: We can define a function to receive any number of keyword arguments. Variable length kwargs are packed as dictionary.

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

more_args(a=1, b=2)
```

Function Call Stack: Stack is a data structure that stores items in an Last-In/First-Out manner. Function Call Stack keeps track of function calls in progress.

```
def function_1():
    pass

def function_2():
    function_1()
```

Recursion: A function calling itself is called a Recursion.

```
def function_1():
    block of code
    function_1()
```

Passing Immutable Objects:

```
def increment(a):
    a += 1

a = int(input()) # 5
increment(a)
print(a) # 5
```

Even though variable names are same, they are referring to two different objects.
Changing the value of the variable inside the function will not affect the variable outside.

Passing Mutable Objects:

```
def add_item(list_x):
list_x += [3] list_a =
[1,2]



add_item(list_a)
print(list_a) # [1, 2, 3]
```

The same object in the memory is referred by both list_a and list_x

```
def add_item(list_x=[]):
    list_x += [3]
    print(list_x)

add_item()
add_item([1,2])
add_item()
```

```
# Output is:
[3]
[1, 2, 3]
[3, 3]
```

Default args are evaluated only once when the function is defined, not each time the function is called.

Nested Loops

Nested Loops: An inner loop within the repeating block of an outer loop is called a `Nested Loop`. The Inner Loop will be executed one time for each iteration of the Outer Loop.

Syntax:

```
for item in sequence A:
    Block 1
    for item in sequence B:
        Block 2
```

Syntax of while in for loop:

```
for item in sequence:
    Block 1
    while Condition:
        Block 2
```

Syntax of for in while loop:

```
while Condition:
    Block 1
    for item in sequence:
        Block 2
```

Loop Control Statements:

Name	Usage
Break	`break` statement makes the program exit a loop early.
Continue	`continue` is used to skip the remaining statements in the current iteration when a condition is satisfied.
Pass	`pass` statement is used as a syntactic placeholder. When it is executed, nothing happens.

Name	Usage
Break (In Nested Loop)	`break` in the inner loop stops the execution of the inner loop.