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

① **while loops** -: Allows us to executes a block of code several times as long as the condition is True

```
a = 1
while a < 3:
    a = a + 1
    print(a) → 2
               3
```

② **For loop** -: for statement iterates over each item of a sequence.

Syntax -:

```
for each_item in sequance:
    block of code.
```

→ Range Generate a sequance of integers starting from 0. Stop before n (n is not included):

```
for num in range(3):
    Print(num) → 0, 1, 2
```

Syntax of range -:

```
range(start, end)
```

```
for num in range(5, 8):
    Print(num)
        ↳ 5, 6, 7
```

Nested loops -: An inner loop within the representing block of an outer loop is called a Nested loop. The inner loop will be executed one time for each iteration of the outerloop.

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

Syntax of while loop in for loop -:

```
for item in sequance:
    Block 1
    while condition:
        Block 2
```

Syntax of for in while loop -:

```
while condition:
    Block 1
    for item in sequance:
        Block 2
```

Loop Constrol Statement =:

Break -: break statement makes the program exist a loop early.

Continue -: continue is used to skip the remaining statement in the current iteration when a condition is satisfied.

Pass -: pass statement is used as a syntatic placehdder when it is executed, nothing happens.

break (in nested loop) -: break in the inner loop stop the execution of the inner loop.

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", 2, 8.2]
for item in list_a:
    Print (item)
```
↳ 5, "six", 2, 8.2

List Concatination -: similar to string. + operator concatenats 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)
```
→ last index.

[5, "six"]

Extended slicing -: Similer to string extended slicing, we can extract alternate items using the steapes.

```
list_a = ["R", "B", "G", "O", "W"]
list_b = list_a [0:5:3]
Point ( 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]
Point (list_b)
```
→ [1, 2, 3, 4, 5]

Slicing with negative index -:

You can also specify negative indices while scling a list.

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

Negative steape size -: Negative steape size determine 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 list

in -: By using in operator, one can determine if a value is present in a sequance or not

not in -: By using the, not in operator, one can determine if a value is non present in a sequance or not.

Nested list -: 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 item of Nested list -:

```
list_a = [5, "six", [8, 6], 8.2]
Print (list_a [2] [0]) → 8
```

List method

Name	Syntax	Useage
① append()	list.append(value)	Add an element to the end of the list.
② extend()	list_a.extend(list_b)	Add all the elements of a sequance to the end of the list.
③ insert()	list.insert(index, value)	Element is inserted to the list at specified index.
④ pop()	list.pop()	Removes last element
⑤ remove()	list.remove(v)	Removes the first matching element from the list.
⑥ clear()	list.clear()	Removes all the elements from the list.
⑦ index()	list.index(value)	Returns the index at the first occurance of the specific value.
⑧ count()	list.count(value)	Returns the number of elements with the specific value.
⑨ sort()	list.sort()	Sort the list.
⑩ copy()	list.copy()	Return a new list, it doesnit modify the orginal list.

function

Block of reusable code to perform a specific action

Define a function -:

function is uniquely ~~definied~~ 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+5) → 5
sum_of_two_number (2,3)
```

function with argeguments -:

We can pass values to a function using an arguments,

```
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.

Passing Immutable objects -:

```
def increment(a):
    a += 1
a = int(input()) → 5
increment(a)
print(a) → 5
```

→ Even through variable names are same, they are referring to two different objects.

→ Changint 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 to by both list_a and list_x.

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

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

```
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 value by their name.

```
def greet ( arg_1 , arg_2):
        Print (arg_1 + "  " + arg_2)
                    → Good Morning Ram
greet (arg_1 = "Good Morning" , arg_2 = "Ram")
```

Positional arguments –:

Value can be passed without using argument name. These value get assigned according to their position. order of the argument matters here.

```
def greet(arg1 = "Hi", arg2 = "Ram"):
    Print(arg1 + " " + arg2)

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

Multiple keyword argument –:

we can define a function to receive any number of keyword argument. variable length kwargs are packed as dictionary.

```
def more_args(**kwargs):
    Print(kwargs)
more_args(a=1, b=2)
```
↳ {'a': 1, 'b': 2}

function Call stack –:

Stack is a data structure that store item in an last-in/fist-out manner. function call stack keeps track of fumction call in progress.

```
def function_1()
    Pass

def function_2()
    function_1()
```

Recursion –: A function call itself is called a Recursion,

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

```
def greet (arg_1 , arg_2):
    Print (arg_1 + " " + arg_2)
                    ↳ Good morning Ram.
greeting = input( ) → Good morning
name = input( ) → Ram
greet (greeting, name) .
```

① Default values :-

```
def greet (arg_1 = "Hi", arg_2 = "Ram"):
    Print (arg_1 + " " + arg_2)
                    ↳ Hi Ram
greeting = input() → Hello
name = input() → Teja
greet ()
```

Arbitary function Arguments :-

We can define a function to recive any number of argument.

Variable length argument :-

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