Q. Explain about Python Lists and its operations with examples.

List is one of the built-in data types in Python.

List is an ordered collection of items. Each item in a list has a unique position index, starting from 0.

In Python, a list is a built-in data structure that can hold an ordered collection of items. Unlike arrays in some languages, Python lists are very flexible:

- Can contain duplicate items
- Mutable: items can be modified, replaced, or removed
- Ordered: maintains the order in which items are added
- Index-based: items are accessed using their position (starting from 0)
- Can store mixed data types (integers, strings, booleans, even other lists)

a)Creating a List

Lists can be created in several ways, such as using square brackets, the list() constructor or by repeating elements. Let's look at each method one by one with example:

1. Using Square Brackets

We use square brackets [] to create a list directly.

```
a = [1, 2, 3, 4, 5] # List of integers
b = ['apple', 'banana', 'cherry'] # List of strings
c = [1, 'hello', 3.14, True] # Mixed data types
print(a)
print(b)
print(c)
```

2. Using list() Constructor

We can also create a list by passing an iterable (like a tuple, string or another list) to the list() function.

```
a = list((1, 2, 3, 'apple', 4.5))
print(a)
b = list("GFG")
print(b)
```

3. Creating List with Repeated Elements

We can use the multiplication operator * to create a list with repeated items.

Ex:-

```
a = [2] * 5
```

```
b = [0] * 7
print(a)
print(b)
```

b)Accessing List Elements

Elements in a list are accessed using indexing. Python indexes start at 0, so a[0] gives the first element. Negative indexes allow access from the end (e.g., -1 gives the last element).

Ex:-

```
a = [10, 20, 30, 40, 50]
print(a[0])
print(a[-1])
print(a[1:4])
```

c) Adding Elements into List

We can add elements to a list using the following methods:

- append(): Adds an element at the end of the list.
- extend(): Adds multiple elements to the end of the list.
- insert(): Adds an element at a specific position.
- clear(): removes all items.

Ex:-

```
a = []
a.append(10)
print("After append(10):", a)
a.insert(0, 5)
print("After insert(0, 5):", a)
a.extend([15, 20, 25])
print("After extend([15, 20, 25]):", a)
a.clear()
```

d)Updating Elements into List

Since lists are mutable, we can update elements by accessing them via their index.

Ex:-

```
a = [10, 20, 30, 40, 50]
a[1] = 25
print(a)print("After clear():", a)
```

Removing Elements from List

We can remove elements from a list using:

- remove(): Removes the first occurrence of an element.
- pop(): Removes the element at a specific index or the last element if no index is specified.
- del statement: Deletes an element at a specified index.

Ex:-

```
a = [10, 20, 30, 40, 50]
a.remove(30)
print("After remove(30):", a)
popped_val = a.pop(1)
print("Popped element:", popped_val)
print("After pop(1):", a)
del a[0]
print("After del a[0]:", a)
```

Iterating Over Lists

We can iterate over lists using loops, which is useful for performing actions on each item.

Ex:-

```
a = ['apple', 'banana', 'cherry']
for item in a:
   print(item)
```

Nested Lists

A nested list is a list within another list, which is useful for representing matrices or tables. We can access nested elements by chaining indexes.

```
matrix = [ [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9] ]
print(matrix[1][2])
```

How Python Stores List Elements?

In Python, a list doesn't store actual values directly. Instead, it stores references (pointers) to objects in memory. This means numbers, strings and booleans are separate objects in memory and the list just keeps their addresses.

That's why modifying a mutable element (like another list or dictionary) can change the original object, while immutables remain unaffected.

Q. Explain about Python strings and its operations with examples.

In Python, a string is a sequence of characters enclosed in quotes. It can include letters, numbers, symbols or spaces. Since Python has no separate character type, even a single character is treated as a string with length one. Strings are widely used for text handling and manipulation.

Creating a String

Strings can be created using either single ('...') or double ("...") quotes. Both behave the same.

Example: Creating two equivalent strings one with single and other with double quotes.

```
s1 = 'GfG'  # single quote
s2 = "GfG"  # double quote
print(s1)
print(s2)
```

Multi-line Strings

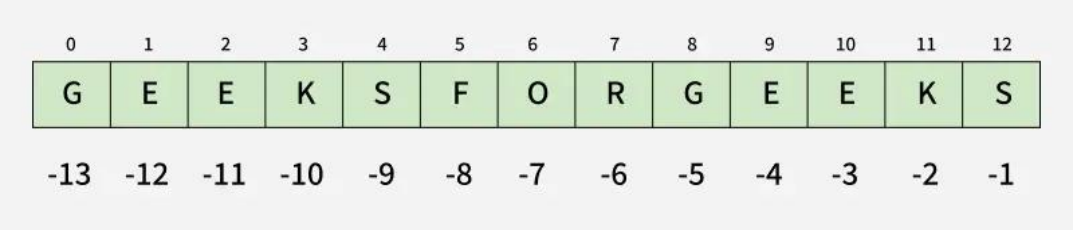
Use triple quotes ('''...''') or ("""...""") for strings that span multiple lines. Newlines are preserved.

Example: Define and print multi-line strings using both styles.

```
s = """I am Learning
Python String on GeeksforGeeks"""
print(s)
s = '''I'm a
Geek'''
print(s)
```

Accessing characters in String

Strings are indexed sequences. Positive indices start at **0** from the **left**; negative indices start at **-1** from the **right** as represented in below image:




```
s = "GeeksforGeeks"
print(s[0])   # first character
print(s[4])   # 5th character
```

String Slicing

Slicing is a way to extract a portion of a string by specifying the start and end indexes. The syntax for slicing is string[start:end], where start starting index and end is stopping index (excluded).

Ex:-

```
S=”PYTHON PROGRAMMING”
print(s[1:4])   # characters from index 1 to 3
print(s[:3])    # from start to index 2
print(s[3:])    # from index 3 to end
print(s[::-1])  # reverse string
```

String Iteration

Strings are iterable; you can loop through characters one by one

```
s = "Python"
for char in s:
  print(char)
```

Deleting a String

In Python, it is not possible to delete individual characters from a string since strings are immutable. However, we can delete an entire string variable using the del keyword.

```
s = "GfG"
del s
```

Updating a String

As strings are immutable, “updates” create new strings using slicing or methods such as replace().

Example: This code fix the first letter and replace a word.

```
s = "hello geeks"
s1 = "H" + s[1:]              # update first character
s2 = s.replace("geeks", "GeeksforGeeks")  # replace word
print(s1)
print(s2)
```

Common String Methods

Python provides various built-in methods to manipulate strings. Below are some of the most useful methods:

1. len(): The len() function returns the total number of characters in a string (including spaces and punctuation).

Example:

```
s = "GeeksforGeeks"
print(len(s))
```

2. upper() and lower(): upper() method converts all characters to uppercase whereas, lower() method converts all characters to lowercase.

Example:

```
s = "Hello World"
print(s.upper())
print(s.lower())
s = "Hello World"
print(s.upper())
print(s.lower())
```

3. strip() and replace(): strip() removes leading and trailing whitespace from the string and replace() replaces all occurrences of a specified substring with another.

Example:

```
s = "   Gfg   "
print(s.strip())
s = "Python is fun"
print(s.replace("fun", "awesome"))
```

4.swapcase()

Inverts case for all letters in string.

5.title()

Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.

6.strip() (lstrip,rstrip)

Removes both left and right trailing blanks in string.

7.isalnum()

Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.

8.isalpha()

Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.

9.isdigit()

Returns true if the string contains only digits and false otherwise.

Concatenating and Repeating Strings

We can concatenate strings using + operator and repeat them using * operator.

1. Strings can be combined by using + operator.

Example: Join two words with a space.

```
s1 = "Hello"
s2 = "World"
```

```
print(s1 + " " + s2)
```

2. We can repeat a string multiple times using * operator.

Example: Repeat a greeting three times.

```
s = "Hello "
print(s * 3)
```

Using format() method

It is a built-in method of str class. The format() method works by defining placeholders within a string using curly braces "{}". These placeholders are then replaced by the values specified in the method's arguments.

Example

In the below example, we are using format() method to insert values into a string dynamically.

```
str = "Welcome to {}"
print(str.format("Tutorialspoint"))
```

Using f-string

The f-strings, also known as formatted string literals, is used to embed expressions inside string literals. The "f" in f-strings stands for formatted and prefixing it with strings creates an f-string. The curly braces "{}" within the string will then act as placeholders that is filled with variables, expressions, or function calls.

Ex:-

```
item1_price = 2500
item2_price = 300
total = f'Total: {item1_price + item2_price}'
print(total)
```

Q. Explain about Set Operations in Python with examples.

A **Set** is another one of the built-in data types in Python.

A Set is an **unordered collection** of unique items. Unlike a list, a set does **not** allow duplicate elements and is **not** indexed.

In Python, a set is a built-in data structure that can hold a collection of distinct items. Sets are designed to perform mathematical set operations like union, intersection, and difference.

- **No Duplicates:** Automatically removes duplicate items.
- **Mutable:** Items can be added or removed, but the items themselves must be immutable (e.g., you can't put a list inside a set).
- **Unordered:** Does **not** maintain the order in which items are added.
- **Non-indexed:** Items are accessed using iteration, not position (indexing is not supported).
- **Can store mixed data types** (as long as they are immutable).

a) Creating a Set

Sets can be created using curly braces {} or the set() constructor.

1. Using Curly Braces {}

   We use curly braces to create a set directly. Note: To create an empty set, you must use set(), as {} creates an empty dictionary.

   Python

```
# Set of integers - duplicates are automatically removed
a = {1, 2, 3, 4, 1, 2}
# Set of strings
b = {'apple', 'banana', 'cherry'}
# Mixed data types (must be immutable)
c = {1, 'hello', 3.14, True}

print(a) # Output: {1, 2, 3, 4}
print(b) # Output: {'apple', 'banana', 'cherry'}
print(c) # Output: {1, 'hello', 3.14} (True and 1 are considered the same in a set)
```

2. Using set() Constructor

   We can create a set by passing an iterable (like a list, tuple, or string) to the set() function.

   Python

```
# Creating a set from a list
a = set([1, 2, 3, 'apple', 4.5])
print(a) # Output: {1, 2, 3, 4.5, 'apple'} (order may vary)

# Creating a set from a string
b = set("GFG")
print(b) # Output: {'G', 'F'} (order may vary, single 'G' in string)
```

---

b) Accessing Set Elements

Since sets are **unordered** and **non-indexed**, you cannot access elements using square brackets like in lists. You typically iterate over a set or check for membership.

Checking Membership:

Python

```
a = {'apple', 'banana', 'cherry'}

# Check if an element is in the set
print('banana' in a)  # Output: True
print('grape' in a)   # Output: False
```

---

c) Adding Elements to a Set

We can add elements to a set using the following methods:

- add(): Adds a single element to the set.
- update(): Adds multiple elements from an iterable (like a list, tuple, or another set) to the set.

Example:-

```
a = {10, 20}

a.add(30)
print("After add(30):", a)  # Output: {10, 20, 30} (order may vary)

a.update([40, 50, 60])
print("After update([40, 50, 60]):", a)
# Output: {40, 10, 50, 20, 60, 30} (order may vary)

# Attempting to add a duplicate has no effect
a.add(30)
print("After add(30) again:", a)
```

---

d) Removing Elements from a Set

We can remove elements from a set using:

- remove(): Removes the specified element. **Raises a KeyError if the element is not found.**
- discard(): Removes the specified element. **Does nothing if the element is not found.**
- pop(): Removes and returns an arbitrary element from the set (since sets are unordered).
- clear(): Removes all elements from the set.

Example:

```
a = {10, 20, 30, 40, 50}

a.remove(30)
print("After remove(30):", a) # Output: {10, 20, 40, 50}

a.discard(10)
print("After discard(10):", a) # Output: {20, 40, 50}

# Try to discard an item that doesn't exist (no error)
a.discard(100)

popped_val = a.pop() # Removes an arbitrary element
print("Popped element:", popped_val)
print("After pop():", a)

a.clear()
print("After clear():", a) # Output: set()
```

---

e) Set Operations (Mathematics)

Sets are powerful for performing mathematical set operations:

Operator	Method	Description
$\cup$	union() or `	`
$\cap$	intersection() or &	Returns a new set with only elements common to both sets.
$-$	difference() or -	Returns a new set with elements from the first set that are NOT in the second set.
$\triangle$	symmetric_difference() or ^	Returns a new set with all elements except the common ones.

Example:

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}

# Union
union_set = set1.union(set2)
print("Union:", union_set) # Output: {1, 2, 3, 4, 5, 6}

# Intersection
intersection_set = set1 & set2
print("Intersection:", intersection_set) # Output: {3, 4}

# Difference (Elements in set1 but not in set2)
difference_set = set1 - set2
print("Difference (set1 - set2):", difference_set) # Output: {1, 2}

# Symmetric Difference (Elements in either set, but not both)
sym_diff_set = set1 ^ set2
print("Symmetric Difference:", sym_diff_set) # Output: {1, 2, 5, 6}
```

---

f) Iterating Over Sets

You can iterate over a set using a loop, which is useful for processing each unique item.

Python

```
a = {'apple', 'banana', 'cherry'}

for item in a:
    print(item)
# Output (order may vary):
# apple
# banana
# cherry
```

---

g) Frozensets

A **frozenset** is an **immutable** version of a set. Once created, you cannot add or remove elements. Frozensets are hashable, which means they can be used as **keys in a dictionary** or as **elements in another set**.

Python

```
a = frozenset([1, 2, 3])
# a.add(4) # This would raise an AttributeError
print(a) # Output: frozenset({1, 2, 3})
```

Q. Explain about Python Dictionaries and Operations

A **Dictionary** (often called a 'dict') is a built-in data type in Python that stores data in **key-value pairs**.

It is an **unordered collection** of items where each item is a mapping from a **unique key** to a **value**. Dictionaries are optimized for retrieving values when the key is known.

- **Key-Value Pairs:** Data is stored as $Key: Value$.
- **Mutable:** The values associated with keys can be changed, and new key-value pairs can be added or removed.
- **Ordered (since Python 3.7):** Dictionaries maintain the insertion order of the keys.
- **Keys Must Be Unique and Immutable:** Keys must be hashable (e.g., strings, numbers, or tuples). Values can be any data type (including lists, other dictionaries, etc.).
- **Index-based (by key):** Values are accessed using their unique key, not their position index.

a) Creating a Dictionary

Dictionaries can be created using curly braces {} or the dict() constructor.

1. Using Curly Braces {}
   We use curly braces with colons : separating the key and value to create a dictionary.
   Ex:-

```
# Dictionary with string keys and mixed values
student = {
    'name': 'Alice',
    'age': 25,
    'courses': ['Math', 'Science'],
    'is_enrolled': True
}

# Dictionary with integer keys
lookup_table = {1: 'one', 2: 'two', 3: 'three'}

print(student)
print(lookup_table)
```

2. Using dict() Constructor
   We can create a dictionary using the dict() function with keyword arguments or by passing an iterable of key-value pairs.
   Ex:-

```
# Using keyword arguments
a = dict(name='Bob', score=95)
print(a) # Output: {'name': 'Bob', 'score': 95}

# Using a list of tuples (key, value)
b = dict([('apple', 1), ('banana', 2)])
print(b) # Output: {'apple': 1, 'banana': 2}
```

---

b) Accessing Dictionary Elements

Elements in a dictionary are accessed using their **key**.

1. Using Square Brackets []
   This is the most common way, but it raises a KeyError if the key doesn't exist.
   Ex:-

```
student = {'name': 'Alice', 'age': 25}

# Accessing value by key
print(student['name'])   # Output: Alice

# print(student['score']) # This would raise a KeyError
```

2. Using the get() Method

   This method is safer as it returns None (or a specified default value) if the key is not found, instead of raising an error.

   Ex:-

```
print(student.get('age'))    # Output: 25
print(student.get('score'))  # Output: None
print(student.get('score', 'Key not found')) # Output: Key not found
```

---

c) Adding and Updating Elements

Since dictionaries are mutable, adding a new key-value pair or updating an existing value is done using the same simple assignment syntax.

1. Adding a New Key-Value Pair
   Assign a value to a new key.
   Ex:-

```
my_dict = {'fruit': 'apple'}
my_dict['color'] = 'red' # Adds a new pair
print("After adding:", my_dict) # Output: {'fruit': 'apple', 'color': 'red'}
```

2. Updating an Existing Value
   Assign a new value to an existing key.
   Ex:-

```
my_dict['fruit'] = 'banana' # Updates the value for 'fruit'
print("After updating:", my_dict) # Output: {'fruit': 'banana', 'color': 'red'}
```

3. Using update()

   Merges another dictionary or an iterable of key-value pairs into the current dictionary. If keys overlap, the values from the argument override the original values.

   Ex:

```
my_dict.update({'color': 'yellow', 'size': 'large'})
print("After update():", my_dict)
# Output: {'fruit': 'banana', 'color': 'yellow', 'size': 'large'}
```

---

d) Removing Elements from a Dictionary

We can remove elements using:

- pop(key): Removes the item with the specified *key* and returns its *value*. **Raises a KeyError if the key is not found.**
- popitem(): Removes and returns an arbitrary key-value pair (in Python 3.7+ it removes the last inserted item).
- del statement: Deletes the key-value pair.
- clear(): Removes all items.

Ex:-

```
data = {'a': 10, 'b': 20, 'c': 30}

# Using pop()
popped_val = data.pop('b')
print(f"Popped value: {popped_val}") # Output: 20
print("After pop('b'):", data) # Output: {'a': 10, 'c': 30}

# Using del
del data['a']
print("After del data['a']:", data) # Output: {'c': 30}


# Using clear()
data.clear()
print("After clear():", data) # Output: {}
```

---

e) Dictionary Views (Keys, Values, Items)

Dictionaries provide methods that return **view objects** of their contents. These views reflect any changes made to the original dictionary.

- keys(): Returns a view object of all keys.
- values(): Returns a view object of all values.
- items(): Returns a view object of all key-value pairs (as tuples).

Python

```
user = {'id': 101, 'role': 'admin', 'status': 'active'}

print("Keys:", user.keys())     # Output: dict_keys(['id', 'role', 'status'])
print("Values:", user.values()) # Output: dict_values([101, 'admin', 'active'])
print("Items:", user.items())   # Output: dict_items([('id', 101), ('role', 'admin'), ('status', 'active')])
```

f) Iterating Over Dictionaries

You can iterate over the keys, values, or items in a dictionary.

Ex:-

```
user = {'id': 101, 'role': 'admin'}

print("--- Iterating over keys (Default) ---")
for key in user:
    print(key)

print("--- Iterating over values ---")
for value in user.values():
    print(value)

print("--- Iterating over items (Key and Value) ---")
for key, value in user.items():
```

```
print(f"{key} is {value}")
```

Q.Explain about Python Tuples and Operations

A **Tuple** is another one of the built-in data types in Python.

A Tuple is an **ordered collection** of items, similar to a list, but the key difference is that a tuple is **immutable** (cannot be changed after creation).

- **Ordered:** Maintains the order in which items are added.
- **Immutable:** Items cannot be modified, replaced, or removed after the tuple is created.
- **Index-based:** Items are accessed using their position (starting from 0).
- **Can store mixed data types** (integers, strings, lists, etc.).
- **Faster and Safer:** Because they are immutable, tuples are generally faster than lists and are used to ensure data integrity (data that shouldn't change).

a) Creating a Tuple

Tuples can be created using parentheses () or the tuple() constructor. They can often be created just by separating items with commas, which is known as **tuple packing**.

1. Using Parentheses () (Tuple Packing)

   We use parentheses () to create a tuple directly.

   Python

   ```
   # Tuple of integers
   a = (1, 2, 3, 4, 5)
   # Tuple of strings
   b = ('apple', 'banana', 'cherry')
   # Mixed data types
   c = (1, 'hello', 3.14, True)

   # Tuple packing (parentheses are often optional)
   d = 1, 2, 3

   # IMPORTANT: Creating a single-item tuple requires a trailing comma
   single_item_tuple = ('hello',)
   not_a_tuple = ('hello') # This is just a string!

   print(a)
   print(single_item_tuple)
   ```

2. Using tuple() Constructor

   We can create a tuple by passing an iterable (like a list, string, or another tuple) to the tuple() function.

   Python

   ```
   # Creating a tuple from a list
   ```

```
a = tuple([1, 2, 3, 'apple', 4.5])
print(a) # Output: (1, 2, 3, 'apple', 4.5)

# Creating a tuple from a string
b = tuple("GFG")
print(b) # Output: ('G', 'F', 'G')
```

---

b) Accessing Tuple Elements

Elements in a tuple are accessed using **indexing** and **slicing**, identical to how they are used in lists. Python indexes start at 0.

Python
```
a = (10, 20, 30, 40, 50)

# Positive indexing (start from 0)
print(a[0])   # Output: 10 (First element)

# Negative indexing (start from -1 for the last element)
print(a[-1])  # Output: 50 (Last element)

# Slicing (elements from index 1 up to (but not including) 4)
print(a[1:4]) # Output: (20, 30, 40)
```

---

c) Immutability (The Key Difference)

The defining feature of a tuple is that it is **immutable**. You cannot change, add, or remove elements after it has been created.

Python
```
my_tuple = (10, 20, 30)

# Attempting to change an element will result in an error
# my_tuple[1] = 25 # TypeError: 'tuple' object does not support item assignment

# Attempting to add or remove elements will also result in an error
# my_tuple.append(40) # AttributeError: 'tuple' object has no attribute 'append'
# my_tuple.pop()      # AttributeError: 'tuple' object has no attribute 'pop'
```

Note: A tuple's contents can include **mutable objects** (like a list). While you cannot replace the list object *itself* within the tuple, you *can* modify the contents of that nested list.

Python
```
nested_tuple = (1, 2, ['a', 'b'])
print("Original nested list:", nested_tuple[2]) # Output: ['a', 'b']

# Modifying the list *inside* the tuple is possible:
nested_tuple[2].append('c')
print("After modification:", nested_tuple)
# Output: (1, 2, ['a', 'b', 'c']) -> The list's content changed, but the tuple's structure didn't.
```

---

d) Tuple Operations and Utility Methods

Since tuples are immutable, they have fewer methods than lists, focusing on querying the data rather than manipulating it.

Method	Description
count(item)	Returns the number of times a specified value occurs in the tuple.
index(item)	Searches the tuple for a specified value and returns the index of its first occurrence. **Raises a ValueError if the item is not found.**

Concatenation and Repetition:

Tuples can be combined using the + operator and repeated using the * operator, both of which **create a new tuple** (since tuples are immutable).

Python
```
a = (1, 2, 3)
b = (4, 5)

# Concatenation
c = a + b
print("Concatenation:", c) # Output: (1, 2, 3, 4, 5)

# Repetition
d = a * 2
print("Repetition:", d) # Output: (1, 2, 3, 1, 2, 3)

# Utility methods
e = (1, 5, 2, 1, 3, 1)
print("Count of 1:", e.count(1))     # Output: 3
print("Index of 2:", e.index(2))     # Output: 2
# print(e.index(9)) # ValueError: tuple.index(x): x not in tuple
```

---

e) Tuple Unpacking (Sequence Unpacking)

One of the most common and powerful uses of tuples is **unpacking**, where the values in a tuple are assigned to multiple variables in a single statement.

Python
```
coordinates = (10, 20)

# Unpacking the tuple into two separate variables
x, y = coordinates

print(f"X coordinate: {x}") # Output: 10
print(f"Y coordinate: {y}") # Output: 20

# This is common for function returns or looping over items
for index, item in enumerate(['a', 'b', 'c']):
    # enumerate returns a tuple (index, item) in each iteration
    print(f"Index {index} has value {item}")
```

Q. Explain about Functions in Python.

A Python function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

A top-to-down approach towards building the processing logic involves defining blocks of independent reusable functions. A Python function may be invoked from any other

function by passing required data (called parameters or arguments). The called function returns its result back to the calling environment.

Syntax:

```
def function_name(parameter1,param2....):

        function code block
        .....
        .....
    [return(value/expression)]
```

- Function blocks begin with the keyword **def** followed by the function name and parentheses ().
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement; the documentation string of the function or docstring.
- The code block within every function starts with a colon (:) and is indented.
- The statement **return [expression]** exits a function, optionally passing back an expression to the caller. A **return** statement with no arguments is the same as return None.

Example:

```
def greetings():
    "This is docstring of greetings function"
     print ("Hello World") return
    return
```

When this function is called, **Hello world** message will be printed.

Calling a Python Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code. Once the basic structure of a function is finalized, you can call it by using the function name itself. If the function requires any parameters, they should be passed within parentheses.

Pass by Reference vs Value

In programming languages like C and C++, there are two main ways to pass variables to a function, which are **Call by Value** and **Call by Reference** (also known as pass by reference and pass by value). However, the way we pass variables to functions in Python differs from others.

- **call by value –** When a variable is passed to a function while calling, the value of actual arguments is copied to the variables representing the formal arguments.

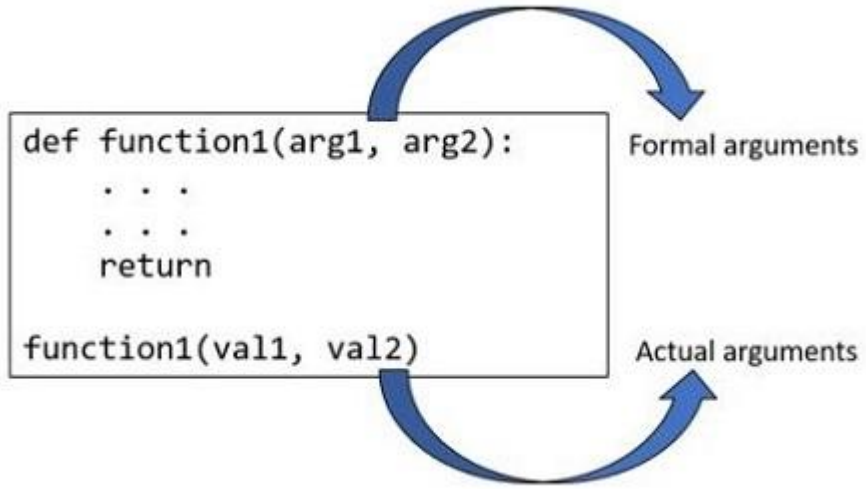
Thus, any changes in formal arguments does not get reflected in the actual argument. This way of passing variable is known as call by value.

- **call by reference –** In this way of passing variable, a reference to the object in memory is passed. Both the formal arguments and the actual arguments (variables in the calling code) refer to the same object. Hence, any changes in formal arguments does get reflected in the actual argument.

Python Function Arguments

Function arguments are the values or variables passed into a function when it is called. The behavior of a function often depends on the arguments passed to it.

While defining a function, you specify a list of variables (known as formal parameters) within the parentheses. These parameters act as placeholders for the data that will be passed to the function when it is called. When the function is called, value to each of the formal arguments must be provided. Those are called actual arguments.



Ex:-

```
def greetings(name):
  "This is docstring of greetings function"
  print ("Hello {}".format(name))
  return

greetings("Samay")
greetings("Pratima")
greetings("Steven")
```

Types of Python Function Arguments

Based on how the arguments are declared while defining a Python function, they are classified into the following categories –

- Positional or Required Arguments

- Keyword Arguments
- Default Arguments
- Arbitrary or Variable-length Arguments

Positional or Required Arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition, otherwise the code gives a syntax error.

```
# Function definition is here
def printme( str ):
  "This prints a passed string into this function"
  print (str)
  return;

# Now you can call printme function
printme() ----------→Error as no arguments passed to function as it requires one argument.
```

Keyword Arguments:

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name. This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

Ex:-

```
# Function definition is here
def printme( str ):
  "This prints a passed string into this function"
  print (str)
  return;

# Now you can call printme function
printme( str = "My string")
```

Default Arguments

A **default argument** is an argument that assumes a default value if a value is not provided in the function call for that argument.

Example

The following example gives an idea on default arguments, it prints default age if it is not passed −

```
# Function definition is here
def printinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print ("Name: ", name)
   print ("Age ", age)
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

Arbitrary or Variable-length Arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called **variable-length arguments** and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this –

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
   return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all non-keyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.

Example

Following is a simple example of Python variable-length arguments

```
# Function definition is here
def printinfo( arg1, *vartuple ):
   "This prints a variable passed arguments"
```

```
  print ("Output is: ")
  print (arg1)
  for var in vartuple:
    print (var)
  return;

# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

#------------------------------ BANK APPLICATION ------USING FUNCTIONS

```
# Initialize the account details storage as an empty dictionary
current_account = {}

def create_account():
  """
  Prompts the user for account details and updates the global
current_account.
  """
  global current_account
  print("\n--- 📝 Account Creation ---")

  # 1. Account Number (Input Validation)
  while True:
    try:
      accno = int(input("Enter Account Number (e.g., 12345): "))
      break
    except ValueError:
      print("❌ Invalid input. Please enter a valid whole number for the
Account Number.")

  # 2. Account Holder Name
  name = input("Enter Account Holder Name: ")
```

```
    # 3. Account Type (Input Validation)
    while True:
        account_type = input("Enter Account Type (Savings or Current): ").strip().title()
        if account_type in ["Savings", "Current"]:
            break
        else:
            print("❌ Invalid account type. Please enter 'Savings' or 'Current'.")

    # 4. Initial Balance (Input Validation)
    while True:
        try:
            balance = float(input("Enter Initial Deposit Amount (must be $0 or more): $"))
            if balance >= 0:
                break
            else:
                print("❌ Initial balance cannot be negative. Please enter $0 or more.")
        except ValueError:
            print("❌ Invalid input. Please enter a valid number for the balance.")

    # Store all details in the global dictionary
    current_account['accno'] = accno
    current_account['name'] = name
    current_account['account_type'] = account_type
    current_account['balance'] = balance

    print("\n✅ Account successfully created!")
    check_balance()

# --- Transaction Functions (Modified to use the dictionary) ---
```

```
def check_balance():
    """Prints the current account balance and details."""
    if not current_account:
        print("\n ⚠️ Please create an account first (Option 1).")
        return

    print("\n 💰 Account Details:")
    print(f"   Account Holder: {current_account['name']}")
    print(f"   Account Type: {current_account['account_type']}")
    print(f"   Balance: ${current_account['balance']:.2f}")

def deposit(amount):
    """Deposits the specified amount into the account."""
    if not current_account:
        print("\n ⚠️ Please create an account first (Option 1).")
        return

    if amount > 0:
        current_account['balance'] += amount
        print(f"\n ✅ Deposit successful.")
        print(f"   ${amount:.2f} has been added to the account.")
        check_balance()
    else:
        print("\n ❌ Error: Deposit amount must be a positive number.")

def withdraw(amount):
    """Withdraws the specified amount from the account, if sufficient funds
are available."""
    if not current_account:
        print("\n ⚠️ Please create an account first (Option 1).")
        return

    current_balance = current_account['balance']

    if amount <= 0:
```

```
        print("\n❌ Error: Withdrawal amount must be a positive number.")
    elif amount > current_balance:
        print("\n❌ Error: Insufficient funds.")
        print(f"   You tried to withdraw ${amount:.2f}, but your balance is only ${current_balance:.2f}.")
    else:
        current_account['balance'] -= amount
        print(f"\n✅ Withdrawal successful.")
        print(f"   ${amount:.2f} has been withdrawn from the account.")
        check_balance()

# --- Main Application Loop ---
def run_bank_app():
    """Main function to run the menu-driven application."""
    print("---------------------------------------")
    print(" 🏦 Welcome to the Simple Python Bank App")
    print("---------------------------------------")

    while True:
        print("\n### Main Menu ###")
        print("1. Create New Account")
        print("2. Check Balance")
        print("3. Deposit Amount")
        print("4. Withdraw Amount")
        print("5. Exit")

        choice = input("Enter your choice (1-5): ")

        if choice == '1':
            create_account()

        elif choice == '2':
            check_balance()

        elif choice == '3':
```

```
        # Get deposit amount from user
        if current_account:
            while True:
                try:
                    amt = float(input("Enter amount to deposit: $"))
                    deposit(amt)
                    break
                except ValueError:
                    print("❌ Invalid input. Please enter a valid number.")
        else:
            print("\n⚠️ Please create an account first (Option 1).")

    elif choice == '4':
        # Get withdrawal amount from user
        if current_account:
            while True:
                try:
                    amt = float(input("Enter amount to withdraw: $"))
                    withdraw(amt)
                    break
                except ValueError:
                    print("❌ Invalid input. Please enter a valid number.")
        else:
            print("\n⚠️ Please create an account first (Option 1).")

    elif choice == '5':
        print("\n👋 Thank you for using the Simple Python Bank App.
Goodbye!")
        break
    else:
        print("\n❌ Invalid choice. Please enter a number between 1 and 5.")

# Run the application
if __name__ == "__main__":
    run_bank_app()
```