PYTHON INTERNSHIP MANUAL

Batch A - 2025 **Week#2 (Functions in Python)**

Important Instructions:

- Read the manual carefully and understand every topics before starting the assignment.
- Follow the deadlines properly; marks will be deducted for late submissions.
- Avoid to use AI Tools and make your own logic.
- Completing all tasks in the assignment is compulsory to receive full marks.

- **Manual Posted Date:**

  11 March, 2025 (Wednesday)

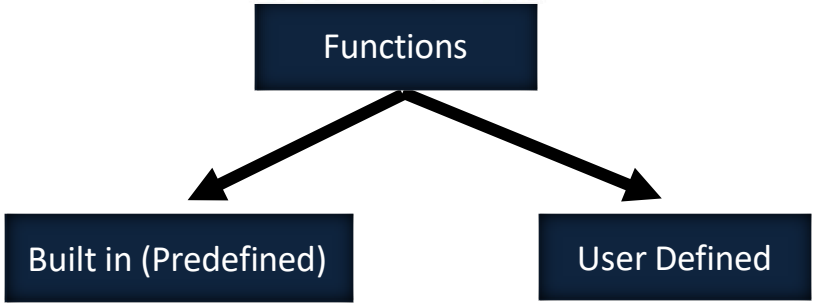
- **Assignment-2 Deadline:**

  17th March, 2025 (Monday at 11:59pm)

Functions

Afunctionisablockoforganized, reusablecodethat isusedto performasingle,relatedaction. Functions provide better modularity for your application and a high degree of code reusing.

Asyoualreadyknow,Pythongivesyou manybuilt-infunctionslikeprint(),etc.butyoucanalso create your own functions. These functions are called user-defined functions.



Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- 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.

```
# Function definition is here
def function_name(Parameters):
  """This is DocString"""
  # Function Body
  a=12
  b-=13
  return a+ b  #Return Statement
```

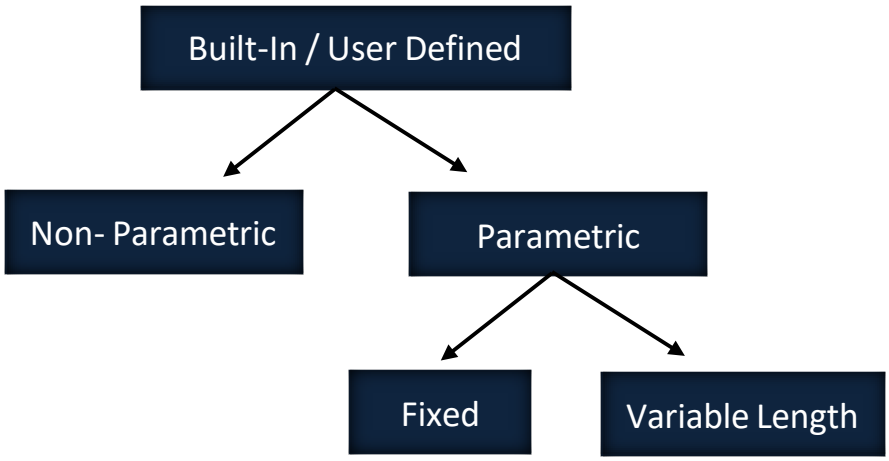
Calling a Function

Defining a function 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 execute it by calling it from another function or directly from the Python prompt. Following is an example to call the printme() function.

```
# Function definition is here
def email(name):
  """This function takes a name as input and generates an email
address."""

  return name + '@gmail.com'


#Here we are calling a Function
email('mehak')
#Calling Function 2nd Time
email('ayesha')
```



Non-Parametric Function	Parametric Function

`# Function definition is here`
`def non_paraFunc ():`
`  """This is Non_parametric Function"""`
`  # Function Body`
`  a=12`
`  b=13`
`  return a+ b`
 	
`# Calling Non_parametric Function`
`non_paraFunc()`
	`# Function definition is here`
	`def paraFunc(a,b):`
	`  """This is parametric Function"""`
	`  #Takes the value of a and b parameters provided by the function user`
	`  return a + b`
	
	`# Calling Parametric Function`
	`paraFunc(1,2)`

Practice Task1: You are building a command-line application that needs to greet the user every time it starts. This function should be called automatically when the application begins to run. Write a function called `print_greeting()` that prints a welcome message to the user.

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

```
# Function definition is here
def changeme( mylist ):
   '''This changes a passed list into this function'''
   print("Values inside the function before change: ", mylist)
   mylist[2]=50
   print ("Values inside the function after change: ", mylist)
   return

#Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

```
OUTPUT:
Values inside the function before change:  [10, 20, 30]
Values inside the function after change:  [10, 20, 50]
Values outside the function:  [10, 20, 50]
```

The Return Statement

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.

The return statement is used to exit a function and pass a value back to the caller. It allows the function to produce a result that can be used elsewhere in the program

Return with no Argument	**Return with Argument**

`def print_message(message):`
`  print('Print Statement : ', message)`
`  return  # This is equivalent to return None`
 	
`result = print_message("Hello, World!")`
`#We cannot store data if nothing is returned from the statement`
 	
`print(result)  # Output: None`
	`def print_message(message):`
	`  print('Print Statement : ', message)`
	`  return message`
	
	`result = print_message("Hello, World!")`
	`#We cannot store data if soemthing is returned from the statement`
	
	`print(result)  # Output: Message`

Fixed Parametric Function:

A function with fixed parameters has a set number of parameters specified when the function is defined. Each parameter must be provided with a value when the function is called.

```
#Function with Fixed Parameters
def display_names(names):
  '''This is Fixed Parametric Function'''
  return names

display_names('mehak')
```

We cannot pass more or fewer parameters than specified; doing so will result in an error.

Passing Additional Parameter

```
#Function with Fixed Parameters
def add(a, b):
  '''A and B are Fixed Parameters'''
  return a + b

add(1,2,3)
```

Error

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-13-623dbc791c39> in <cell line: 6>()
      4   return a + b
      5
----> 6 add(1,2,3)

TypeError: add() takes 2 positional arguments but 3 were given
```

When a user wants to add more than two items, what can we do? In such cases, variable-length parameters come into play. Variable-length parameters allow a function to accept an arbitrary number of arguments, providing flexibility in handling inputs. This approach eliminates the need to specify a fixed number of parameters and simplifies the function's usage.

Practice Task 2:

Create a Python function named calculate_area that calculates the area of a rectangle. The function should take two parameters, length and width, and return the area.

Variable Length Parametric Function:

A function with variable length parameters can accept a varying number of arguments. This flexibility allows for different numbers of inputs without changing the function definition.

```
def display_names(*names):
  '''This is Variable-Length Parametric Function'''
  return names

display_names('mehak','ayesha','ali')
```

Function Arguments

An **argument** is a value that you pass to a function or method when you call it. Arguments are used to provide input data to functions so that they can perform their tasks using this data.
You can call a function by using the following types of formal arguments.

- Positioal arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Positional Argument:

Positional argument is a type of argument that is passed to a function based on its position in the function call. The order of positional arguments matters because it determines which parameter each argument is assigned to.

Positional arguments	**Swapping the Position**

`#Positional Argument`
 	
`def details(name,age):`
`  print('Name : ', name)`
`  print('Age : ', age)`
 	
`details('Mehak', 77)`
Output
`Name :  Mehak`
`Age :  77`
	`#Positional Argument`
	
	`def details(name,age):`
	`  print('Name : ', name)`
	`  print('Age : ', age)`
	
	`details(77,'Mehak')`
	Output
	`Name :  77`
	`Age :  Mehak`

Keyword Argument

These arguments are passed to a function by explicitly specifying the parameter names. This allows you to provide arguments in any order

Keyword arguments

`#Keyword Argument`
 	
`def details(name,age):`
`  print('Name : ', name)`
`  print('Age : ', age)`
 	
`details(age=77, name='Mehak')`
 	
Output
`Name :  Mehak`
`Age :  77`

Default Argument

These arguments have default values assigned to them. If the caller does not provide a value for these arguments, the default value is used.

Default arguments
`#Default Argument`

```
def details(name,age=13):
  print('Name : ', name)
  print('Age : ', age)

details(name='Mehak')
```

Output

```
Name :  Mehak
Age :  13
```

How to Pass the Arguments for Variable-length arguments

Positional Variable-Length Parameters (*args)

Positional Variable-Length Parameters (*args)

```
#Variable Length Args Argument

def details(*name):
  print('Names : ' ,name)

details('Mehak','Ali','Zainab')
```

Output

```
Names :  ('Mehak', 'Ali', 'Zainab')
```

Practice Task 3:

You are working on a statistics module that needs to handle an unknown number of inputs to determine the maximum value. This function will help you handle cases where the number of inputs can vary.

Write a function called `find_maximum(*numbers)` that takes a variable number of numerical arguments and returns the maximum value among them

But how can we assign Keyword to each argument we passed in Variable Length, Here come
Keyword Variable-Length Parameters (kwargs):**
This allows a function to accept a variable number of keyword arguments. The arguments are collected into a dictionary where the keys are the argument names and the values are the corresponding argument values.

Keyword Variable-Length Parameters (*args)

```
#Keyword Variable Length Args Argument

def details(**name):
  print('Names : ' ,name)

details(n1= 'Mehak', n2= 'Ali', n3= 'Zainab')
```

Output

```
Names :  {'n1': 'Mehak', 'n2': 'Ali', 'n3': 'Zainab'}
```

Practice Task 4

You are developing a basic tool that needs to display user-provided information in a simple, readable format.

Write a function called show_info(**details) that takes a variable number of keyword arguments representing information about a person (e.g., name, age). The function should return a string that shows the key-value pairs in a readable format.

Lambda Functions

A lambda function is a small anonymous function. A lambda function can take any number of arguments, but can only have one expression.

```
lambda arguments: expression
```

Here is the Lambda Function

```
add = lambda x, y: x + y
print(add(5, 3))  # Output: 8
```

The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
    return lambda a : a * n
```

Use that function definition to make a function that always doubles the number you send in:

```
def myfunc(n):
    return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```

Built-in Functions in Python

String Methods

1) capitalize() - Converts the first character to upper case.
2) casefold() - Converts string into lower case.
3) center() - Returns a centered string.
4) count() - Returns the number of times a specified value occurs in a string.
5) endswith() - Returns true if the string ends with the specified value.
6) find() - Searches the string for a specified value and returns the position of where it was found.

7) format() - Formats specified values in a string.
8) format_map() – This method is used to format specified values in a string by using a mapping (usually a dictionary) to replace placeholders with corresponding values. It is similar to the **format()** method, but instead of positional or keyword arguments, it takes a mapping object as its argument.
9) index() - Searches the string for a specified value and returns the position of where it was found.
10) isalnum() - Returns True if all characters in the string are alphanumeric.
11) isalpha() - Returns True if all characters in the string are in the alphabet.
12) isdigit() - Returns True if all characters in the string are digits.
13) islower() - Returns True if all characters in the string are lower case.
14) isupper() - Returns True if all characters in the string are upper case.
15) isnumeric() - Returns True if all characters in the string are numeric.
16) isspace() - Returns True if all characters in the string are whitespaces.
17) istitle() - Returns True if the string follows the rules of a title.
18) join() - Joins the elements of an iterable to the end of the string.
19) lower() - Converts a string into lower case.
20) upper() - Converts a string into upper case.
21) replace() - Returns a string where a specified value is replaced with a specified value.
22) rfind()- Searches the string for a specified value and returns the last position of where it was found.
23) split() - Splits the string at the specified separator, and returns a list.
24) splitlines() - Splits the string at line breaks and returns a list.
25) startswith() - Returns true if the string starts with the specified value.
26) title() - Converts the first character of each word to upper case.
27) zfill() Fills the string with a specified number of 0 values at the beginning.

Here are few Examples

```
text = "  Mehak mazhar  "

# Convert to uppercase
upper_text = text.upper()
print(upper_text)  # Output: "  MEHAK MAZHAR  "

# Convert to lowercase
lower_text = text.lower()
print(lower_text)  # Output: "  mehak mazhar  "

# Strip leading and trailing whitespace
stripped_text = text.strip()
print(stripped_text)  # Output: "Mehak mazhar"

# Replace a substring
replaced_text = text.replace("mazhar", "Khan")
print(replaced_text)  # Output: "  Mehak Khan  "
```

```
# Split the string into a list
split_text = text.split()
print(split_text)  # Output: ['Mehak', 'mazhar']

# Check if the string starts with a substring
starts_with_mehak = text.startswith(" Mehak")
print(starts_with_mehak)  # Output: True

# Check if the string ends with a substring
ends_with_mazhar = text.endswith("mazhar ")
print(ends_with_mazhar)  # Output: True

# Find the position of a substring
position = text.find("mazhar")
print(position)  # Output: 9
```

Practice Task 2:

1. Ask user name **(your good name!)**
   Hello! (User name) .. Should be titled
   I hope you are fine, let me know how I can help you! (Start asking)
   If user say **yes**
   Then ask '**share your problem with us**" after user share its problem then say '**Thanks for your feedback, we will resolve your problem**
   If user say anything other than yes take it as No and
   Then print,
   Okay! Good to see you , stay connected (should be in center)

2. Ask user to enter his full name with cast or Father Name After then make a first and last name from this information

List Methods

- append(x): Adds an item to the end of the list.
- extend(iterable): Extends the list by appending elements from an iterable.
- insert(i, x): Inserts an item at a given position.
- remove(x): Removes the first item from the list whose value is equal to x.
- pop([i]): Removes and returns the item at the given position in the list.
- clear(): Removes all items from the list.
- index(x[, start[, end]]): Returns the index of the first item whose value is equal to x.
- count(x): Returns the number of times x appears in the list.
- sort(key=None, reverse=False): Sorts the items of the list in place.
- reverse(): Reverses the elements of the list in place.
- copy(): Returns a shallow copy of the list.
-

```python
# List of common list methods in Python

# Sample list
my_list = [10, 20, 30, 40, 50]

# 1. append() - Adds an element at the end of the list
my_list.append(60)
print("After append(60):", my_list)

# 2. extend() - Adds elements from another list at the end
my_list.extend([70, 80])
print("After extend([70, 80]):", my_list)

# 3. insert() - Inserts an element at the specified position
my_list.insert(2, 25)  # Inserts 25 at index 2
print("After insert(2, 25):", my_list)

# 4. remove() - Removes the first item with the specified value
my_list.remove(30)
print("After remove(30):", my_list)

# 5. pop() - Removes the element at the specified position
popped_element = my_list.pop(3)  # Removes element at index 3
print("After pop(3):", my_list)
print("Popped element:", popped_element)

# 6. clear() - Removes all the elements from the list
my_list.clear()
print("After clear():", my_list)
```

Tuple Methods:

- count(x): Returns the number of times x appears in the tuple.
- index(x[, start[, end]]): Returns the index of the first item whose value is equal to x.

```python
# Tuple of sample elements
my_tuple = (10, 20, 30, 40, 50, 20, 30, 20)

# 1. count() - Returns the number of occurrences of a value
count_of_20 = my_tuple.count(20)
print("Count of 20 in tuple:", count_of_20)

# 2. index() - Returns the index of the first occurrence of a value
index_of_30 = my_tuple.index(30)
print("Index of 30 in tuple:", index_of_30)
```

Set Methods:

- add(elem): Adds an element to the set.
- clear(): Removes all elements from the set.
- copy(): Returns a shallow copy of the set.
- difference(*s): Returns a new set with elements in the set that are not in the others.
- difference_update(*s): Removes elements found in the specified sets.
- discard(elem): Removes an element from the set if it is a member.
- intersection(*s): Returns a new set with elements common to all specified sets.
- intersection_update(*s): Updates the set with the intersection of itself and others.
- isdisjoint(s): Returns True if the set has no elements in common with s.
- issubset(s): Returns True if the set is a subset of s.
- issuperset(s): Returns True if the set is a superset of s.
- pop(): Removes and returns an arbitrary element from the set.
- remove(elem): Removes an element from the set; raises KeyError if the element is not present.
- discard(elem): Removes an element from the set if it is present.
- union(*s): Returns a new set with elements from the set and all others.
- update(*s): Updates the set with the union of itself and others.

```
#Add element in a Set
my_set.add(60)
print("After add(60):", my_set)

#Updating Set
my_set.update({60, 70, 80})
print("After update({60, 70, 80}):", my_set)

#Union
new_set = my_set.union({80, 90, 100})
print("Union with {80, 90, 100}:", new_set)

#Intersection
intersection_set = my_set.intersection({40, 50, 60, 100})
print("Intersection with {40, 50, 60, 100}:", intersection_set)

```

Dictionary Methods

- clear(): Removes all items from the dictionary.
- copy(): Returns a shallow copy of the dictionary.
- fromkeys(iterable, value=None): Creates a new dictionary with keys from iterable and values set to value.
- get(key, default=None): Returns the value for the key if it exists, otherwise default.
- items(): Returns a view object that displays a list of a dictionary’s key-value tuple pairs.
- keys(): Returns a view object that displays a list of all the dictionary's keys.
- pop(key, default=None): Removes the specified key and returns the corresponding value.
- popitem(): Removes and returns a (key, value) pair as a tuple.
- setdefault(key, default=None): Returns the value for key if key is in the dictionary; otherwise, inserts key

with a default value.

- update(*args, **kwargs): Updates the dictionary with elements from another dictionary or from an iterable of key-value pairs.
- values(): Returns a view object that displays a list of all the dictionary's values.

```
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}
#  get() - Returns the value for a specified key
name = my_dict.get('name')
print("Value of 'name':", name)

#  items() - Returns a view object with key-value pairs
items = my_dict.items()
print("Items in dictionary:", list(items))

# keys() - Returns a view object with all the keys
keys = my_dict.keys()
print("Keys in dictionary:", list(keys))

# pop() - Removes and returns the value for the specified key
age = my_dict.pop('age')
print("After pop('age'):", my_dict)
print("Popped value:", age)
```