PYTHON

<u>UNIT 1</u>

Q.1. What is List in Python? Explain the following method with syntax and example.
i. append()
ii. extend()
iii. pop()
iv. remove()

Ans. In Python, a list is a mutable, ordered collection of elements. Lists can contain elements of different data types, and you can modify them by adding, removing, or updating elements. Lists are versatile and commonly used in Python for storing and manipulating data. Lists are defined using square brackets [ ] and elements are separated by commas.

Example:
my_list = [1, 2, 3, 'hello', 5.0]
Here, `my_list` is a list containing integers, a string, and a floating-point number.

i. <u>`append()`:</u> The `append()` method in Python is used to add a single element to the end of a list. It modifies the original list in place.
Syntax: `list.append(element)`
Example:

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)
# Output: [1, 2, 3, 4]
```

In this example, `append(4)` adds the element 4 to the end of the `my_list`.

ii. <u>`extend()`:</u>
The `extend()` method is used to append the elements of an iterable (e.g., another list) to the end of the list. It also modifies the original list in place.
Syntax: `list.extend(iterable)`
Example:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list1.extend(list2)
print(list1)
# Output: [1, 2, 3, 4, 5, 6]
```

Here, `extend(list2)` adds the elements of `list2` to the end of `list1`.

iii. `pop()`: The `pop()` method removes and returns the element at the specified index. If no index is provided, it removes and returns the last element. It modifies the original list.
Syntax: `list.pop(index)`
Example:
```
my_list = [1, 2, 3, 4]
popped_element = my_list.pop(2)
print(popped_element)  # Output: 3
print(my_list)
# Output: [1, 2, 4]
```
Here, `pop(2)` removes the element at index 2 (which is 3) and returns it.

iv. `remove()`: The `remove()` method removes the first occurrence of a specified value from the list. It modifies the original list.
Syntax: `list.remove(value)`
Example:
```
my_list = [1, 2, 3, 2, 4]
my_list.remove(2)
print(my_list)
# Output: [1, 3, 2, 4]
```
The `remove(2)` call removes the first occurrence of the value 2 from `my_list`.

Q.2. What is dictionary in Python?
Consider the dictionary
Dict = { "Name" : "IICC", "University" : "RTMNU","Estt Year": 1987}
Perform the following operation on dictionary with their syntax and output.
i. Get the University using appropriate method and store it in a variable Univ.
ii. Add the item Course "MCA" in Dict
iii. Update the "Estt Year " to 1997 in Dict
iv. Add nested dictionary with key other course and values "PGDCS" and "PGDCCA"
v. Remove an item "Estt Year" in a Dict
vi. Copy the Dict to Dict1
Ans.
In Python, a dictionary is an unordered collection of key-value pairs. It is a mutable data type that allows you to store and retrieve values based on unique keys. Dictionaries are versatile and commonly used for various purposes, such as

representing data structures, configurations, and more. Dictionaries are defined using curly braces `{}` and consist of comma-separated key-value pairs.

```
my_dict = {"name": "John", "age": 25, "city": "New York"}
```
In this example, `"name"`, `"age"`, and `"city"` are keys, and `"John"`, `25`, and `"New York"` are their corresponding values. Keys must be unique within a dictionary, and they are used to access the associated values.

Given dictionary
```
Dict = {"Name": "IICC", "University": "RTMNU", "Estt Year": 1987}
```

i. Get the University using appropriate method and store it in a variable Univ.
```
Univ = Dict.get("University")
print(Univ)
 # Output: RTMNU
```

ii. Add the item Course “MCA” in Dict.
```
Dict["Course"] = "MCA"
print(Dict)
# Output: {"Name": "IICC", "University": "RTMNU", "Estt Year": 1987, "Course": "MCA"}
```

iii. Update the “Estt Year“ to 1997 in Dict.
```
Dict["Estt Year"] = 1997
print(Dict)
# Output: {"Name": "IICC", "University": "RTMNU", "Estt Year": 1997, "Course": "MCA"}
```

iv. Add a nested dictionary with key other course and values “PGDCS” and “PGDCCA”.
```
Dict["Other Courses"] = {"PGDCS": None, "PGDCCA": None}
print(Dict)
# Output: {"Name": "IICC", "University": "RTMNU", "Estt Year":1997, "Course": "MCA", "Other Courses": {"PGDCS": None, "PGDCCA": None}}
```

v. Remove an item “Estt Year” in a Dict.
```
Dict.pop("Estt Year")
print(Dict)
# Output: {"Name": "IICC", "University": "RTMNU", "Course": "MCA", "Other Courses": {"PGDCS": None, "PGDCCA": None}}
```

vi. Copy the Dict to Dict1.

```
Dict1 = Dict.copy()
print(Dict1)
# Output: {"Name": "IICC", "University": "RTMNU", "Course": "MCA", "Other Courses": {"PGDCS": None, "PGDCCA": None}}
```

Q.3. What is tuple in Python? Explain the count() and index() methods with syntax and example.

Ans.

__Tuple__: In Python, a tuple is an ordered, immutable collection of elements. This means that once a tuple is created, its elements cannot be changed, added, or removed. Tuples are defined using parentheses () and may contain elements of different data types. Tuples are commonly used when you want to represent a collection of values that should remain constant throughout the program. They are often used for functions that return multiple values, as the elements of a tuple can be unpacked easily.

```
my_tuple = (1, "hello", 3.14)
```

In this example, my_tuple contains three elements: an integer 1, a string "hello", and a floating-point number 3.14.

__Count():__ The `count()` method in Python is a built-in method for tuples (as well as lists and strings). It is used to count the number of occurrences of a specified element in a tuple. It required a parameter which is to be counted. It returns error if the parameter is missing.

Syntax: tuple.count(element)

- tuple: The tuple on which the `count()` method is called.
- element: The element whose occurrences you want to count in the tuple.

Example:

```
# Define a tuple
my_tuple = (1, 2, 3, 2, 4, 2, 5)
# Use count() to count occurrences of the element 2
count_of_2 = my_tuple.count(2)
# Print the result
print(count_of_2)

# Output: 3
```

In this example, `my_tuple.count(2)` returns the count of occurrences of the element `2` in the tuple `my_tuple`. The result, `3`, indicates that the element `2` appears three times in the tuple.

Index(): The `index()` method in Python is used to find the index of the first occurrence of a specified value within a tuple. It returns the index of the first occurrence of the value. If the value is not found, it raises a `ValueError`.

Syntax: tuple.index(value, start, end)
- `value`: The value to search for in the tuple.
- `start` (optional): The index from where the search begins. Default is 0.
- `end` (optional): The index where the search ends. Default is the end of the tuple.

Example: my_tuple = (10, 20, 30, 20, 40)

```
# Using index() to find the index of the first occurrence of 20
index_of_20 = my_tuple.index(20)
print("Index of 20:", index_of_20)
# Output: Index of 20: 1

# Specifying start and end parameters
index_of_20_after_index_1 = my_tuple.index(20, 2)
print("Index of 20 after index 1:", index_of_20_after_index_1)
# Output: Index of 20 after index 1: 3
```

In this example:
- `index_of_20` returns the index of the first occurrence of the value `20` in the tuple `my_tuple`, which is `1`.
- `index_of_20_after_index_1` starts the search from index `2` and finds the next occurrence of `20`, which is at index `3`.

Q.4. What are the applications of Python? Explain.
Ans. Python is a versatile programming language with a wide range of applications across various domains. The versatility and ease of learning make Python a go-to language for a diverse range of applications, contributing to its popularity across industries.

1. Web Development: Python is widely used for building dynamic and scalable web applications. Frameworks like Django and Flask simplify web development tasks, making it easier to create robust web applications.

2. Data Science and Machine Learning: Python is a popular choice for data analysis, machine learning, and artificial intelligence. Libraries like NumPy, pandas, scikit-learn, and TensorFlow provide powerful tools for working with data and implementing machine learning algorithms.

3. Scientific and Numeric Computing: Python is extensively used in scientific and numeric computing. Libraries such as SciPy and NumPy provide support for scientific calculations, simulations, and data analysis.

4. Automation and Scripting: Python's simplicity and readability make it an excellent choice for automation and scripting tasks. It is often used for writing scripts to automate repetitive tasks and manage system operations.

5. Game Development: Python is utilized in game development, especially for prototyping and scripting within game engines. Pygame is a popular library for creating 2D games.

6. Desktop GUI Applications: Python supports the development of graphical user interface (GUI) applications. Tkinter is a built-in library for creating desktop applications with graphical interfaces.

7. Network Programming: Python's networking capabilities make it suitable for developing network applications and protocols. Libraries like Requests and Twisted are commonly used in networking.

8. Database Systems: Python is used for interacting with various database systems. Libraries such as SQLAlchemy facilitate database operations and integration.

9. Cybersecurity: Python is employed in cybersecurity for tasks like penetration testing, scripting security tools, and analyzing vulnerabilities. Its simplicity and extensive libraries make it useful for security professionals.

10. Education: Python's readability and simplicity make it an ideal language for teaching programming concepts. It is widely used in educational institutions to introduce programming to beginners.

Q.5. Explain if-else statement in Python with syntax and example.

Ans. The **<u>if-else statement</u>** is used to execute both the true part and the false part of a given condition. If the condition is true, the if block code is executed and if the condition is false, the else block code is executed.

Syntax:

```
if condition:
   # Code to be executed if the condition is true
else:
   # Code to be executed if the condition is false
```

First, the test expression is checked. If it is true, the statements present in the body of the if block will execute. Next, the statements present below the if block is executed. In case the test expression has false results, the statements present in the else body are executed, and then the statements below the if-else are executed.

Example:

```
number = int(input("Enter a number: "))
if number > 0:
   print("The number is positive.")
else:
   print("The number is non-positive (zero or negative).")
```

The input() function takes user input, and int() converts it to an integer.
The if statement checks whether the entered number is greater than 0.
If the condition is true, it executes the block of code under the if statement.
If the condition is false, it executes the block of code under the else statement.

Q.6. What are the different operators in Python? Explain with example.

Ans. In Python, **<u>operators</u>** are special symbols or keywords that carry out operations on values and python variables. They serve as a basis for expressions, which are used to modify data and execute computations.
Python supports various types of operators, which are special symbols or keywords that perform operations on variables and values.

<u>1. Arithmetic Operators:</u>

- Addition (+): Adds two operands.

```
result = 5 + 3
print(result) # Output: 8
```

- **Subtraction (-):** Subtracts the right operand from the left operand.
```
result = 10 - 4
print(result)  # Output: 6
```

- **Multiplication (*):** Multiplies two operands.
```
result = 3 * 4
print(result)  # Output: 12
```

- **Division (/):** Divides the left operand by the right operand.
```
result = 20 / 5
print(result)  # Output: 4.0
```

- **Modulus (%):** Returns the remainder of the division of the left operand by the right operand.
```
result = 17 % 5
print(result)  # Output: 2
```

- **Exponentiation (**):** Raises the left operand to the power of the right operand.
```
result = 2 ** 3
print(result)  # Output: 8
```

2. Comparison Operators:

- **Equal to (==):** Checks if the values of two operands are equal.
```
result = (5 == 5)
print(result)  # Output: True
```

- **Not equal to (!=):** Checks if the values of two operands are not equal.
```
result = (5 != 3)
print(result)  # Output: True
```

- **Greater than (>):** Checks if the left operand is greater than the right operand.
```
result = (8 > 5)
print(result)  # Output: True
```

- **Less than (<):** Checks if the left operand is less than the right operand.
```
result = (2 < 4)
print(result)  # Output: True
```

- **Greater than or equal to (>=):** Checks if the left operand is greater than or equal to the right operand.

```
result = (7 >= 7)
print(result)  # Output: True
```

- Less than or equal to (<=): Checks if the left operand is less than or equal to the right operand.

```
result = (3 <= 5)
print(result)  # Output: True
```

3. Logical Operators:

- Logical AND (and): Returns True if both operands are true.

```
result = (True and False)
print(result)  # Output: False
```

- Logical OR (or): Returns True if at least one operand is true.

```
result = (True or False)
print(result)  # Output: True
```

- Logical NOT (not): Returns True if the operand is false and vice versa.

```
result = not True
print(result)  # Output: False
```

4. Assignment Operators:

- Assignment (=): Assigns the value on the right to the variable on the left.

```
x = 10
```

- Addition Assignment (+=): Adds the right operand to the left operand and assigns the result to the left operand.

```
x += 5  # equivalent to x = x + 5
```

- Subtraction Assignment (-=): Subtracts the right operand from the left operand and assigns the result to the left operand.

```
x -= 3  # equivalent to x = x - 3
```

5. Membership Operators:

- in: Returns True if a value is found in the sequence.

```
result = 2 in [1, 2, 3]
print(result)  # Output: True
```

- not in: Returns True if a value is not found in the sequence.

```
result = 4 not in [1, 2, 3]
```

```
  print(result)  # Output: True
```

6. Identity Operators:

- is: Returns True if both operands are the same object.

```
  x = [1, 2, 3]
  y = [1, 2, 3]
  result = x is y
  print(result)  # Output: False
```

- is not: Returns True if both operands are not the same object.

```
  x = [1, 2, 3]
  y = [1, 2, 3]
  result = x is not y
  print(result)  # Output: True
```

<u>**7. Bitwise Operators:**</u>

- Bitwise AND (&): Performs a bitwise AND operation.

```
  result = 5 & 3
  print(result)  # Output: 1
```

- Bitwise OR (|): Performs a bitwise OR operation.

```
  result = 5 | 3
  print(result)  # Output: 7
```

- Bitwise XOR (^): Performs a bitwise XOR (exclusive OR) operation.

```
  result = 5 ^ 3
  print(result)  # Output: 6
```

- Bitwise NOT (~): Performs a bitwise NOT operation.

```
  result = ~5
  print(result)  # Output: -6
```

- Bitwise Left Shift (<<): Shifts the bits of the left operand to the left by a specified number of positions.

```
  result = 5 << 1
  print(result)  # Output: 10
```

- Bitwise Right Shift (>>): Shifts the bits of the left operand to the right by a specified number of positions.

```
  result = 5 >> 1
  print(result)  # Output: 2
```

Q.7. Explain for and while statements in Python with syntax and example.
Ans.
for loop: A for loop is a control flow statement in Python that allows you to iterate over a sequence of elements such as a list, tuple, or string.
On each iteration, the loop variable in Python will take on the value of the next item in the sequence.
Syntax:
```
for variable in sequence:
   # Code to be executed for each item in the sequence
```
Example:
```
# Iterating over a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
   print(fruit)
# Output:
# apple
# banana
# cherry
```
We have assigned a variable, x, which is going to be a placeholder for every item in our iterable object. In this case, the variable “x” actually represents the elements in that list. Then, we print our variable, x. This process continues until all items are printed.

while loop: In Python, a while loop is a control flow statement that allows you to execute a block of code repeatedly while a given condition is true.
The condition is evaluated at the start of each loop iteration, and if it is true, the loop body is executed.
The loop is terminated if the condition is false.
Syntax:
```
while condition:
   # Code to be executed as long as the condition is true
```
Example:
```
# Using a while loop to print numbers 1 to 5
count = 1
while count <= 5:
   print(count)
   count += 1
# Output:
```

```
# 1
# 2
# 3
# 4
# 5
```

In the example, the while loop continues as long as the condition count <= 5 is true. The count variable is incremented inside the loop, ensuring that the loop will eventually exit when count becomes greater than 5.

Q.8. Explain break, continue and pass in Python with suitable example.
Ans. In Python, `break`, `continue`, and `pass` are control flow statements that are used within loops (such as `for` and `while`) to modify their behavior.

1. `break` Statement:
The `break` statement is used to exit a loop prematurely. When a `break` statement is encountered inside a loop, the loop is terminated, and the program continues with the next statement outside the loop.
Syntax: for / while loop:

```
  # statement(s)
  if condition:
     break
  # statement(s)
# loop end
```

Example:

```
for i in range(5):
 if i == 3:
 break
 print(i)
Output:
0
1
2
```

In this example, the `for` loop iterates from 0 to 4. When `i` becomes 3, the `break` statement is encountered, and the loop is terminated. As a result, only the values 0, 1, and 2 are printed.

2. `continue` Statement:
The `continue` statement is used to skip the rest of the code inside a loop for the current iteration and move to the next iteration.

Syntax: for / while loop:
```
   # statement(s)
   if condition:
      continue
   # statement(s)
```

Example:
```
for i in range(5):
 if i == 2:
 continue
 print(i)
```
Output:
```
0
1
3
4
```
In this example, when `i` becomes 2, the `continue` statement is encountered, and the remaining code inside the loop for that iteration is skipped. The loop then continues with the next iteration.

<u>**3. `pass` Statement:**</u>

The `pass` statement is a no-operation statement. It serves as a placeholder where syntactically some code is required but no action is desired.

Syntax: function/ condition / loop:
```
  pass
```

Example:
```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
   if num == 3:
      pass
   else:
      print(num)
# Output:
# 1
# 2
# 4
# 5
```
In this example, the pass statement is used to do nothing when the element is 3, and the loop continues to the next iteration.

Q.9. Write a program in Python to check whether a string is palindrome or not.
Ans. A **palindrome** is a string that reads the same forwards and backward. To check whether a string is a palindrome in Python, you can compare the original string with its reverse.

```
def is_palindrome(input_str):
   # Remove spaces and convert to lowercase for case-insensitive comparison
   cleaned_str = input_str.replace(" ", "").lower()

   # Compare the original string with its reverse
   return cleaned_str == cleaned_str[::-1]

# Get input from the user
user_input = input("Enter a string: ")

# Check if the input string is a palindrome
if is_palindrome(user_input):
   print(f"{user_input} is a palindrome.")
else:
   print(f"{user_input} is not a palindrome.")
```

Explanation:
1. The `is_palindrome` function takes an input string, removes spaces, and converts it to lowercase for a case-insensitive comparison.
2. The function uses string slicing (`[::-1]`) to reverse the cleaned string.
3. It then checks if the cleaned string is equal to its reverse. If they are equal, the input string is a palindrome.
4. The main part of the program takes input from the user, calls the `is_palindrome` function, and prints whether the entered string is a palindrome or not.

Q.10. What is dictionary in Python? Explain with syntax and example.
Ans. Dictionary: In Python, a dictionary is an unordered collection of key-value pairs. Each key in a dictionary must be unique, and it maps to a specific value. Dictionaries are defined using curly braces { }. Dictionaries are versatile and widely used in Python for storing and managing data in a structured way. Dictionaries are mutable, meaning you can modify their content by adding, updating, or removing key-value pairs.

Syntax:

```
my_dict = {"key1": value1, "key2": value2, ...}
```

- `my_dict`: The name of the dictionary.
- `"key1"`, `"key2"`, ...: Unique keys.
- `value1`, `value2`, ...: Corresponding values associated with the keys.

Example:

```
# Creating a dictionary
student = {
   "name": "Alice",
   "age": 20,
   "grade": "A",
   "courses": ["Math", "History", "English"]
}

# Accessing values using keys
print("Student name:", student["name"])  # Output: Student name: Alice
print("Student courses:", student["courses"])  # Output: Student courses:
['Math', 'History', 'English']

# Modifying values
student["age"] = 21
print("Updated age:", student["age"])  # Output: Updated age: 21

# Adding a new key-value pair
student["gender"] = "Female"
print("Updated dictionary:", student)
# Output: Updated dictionary: {'name': 'Alice', 'age': 21, 'grade': 'A', 'courses':
['Math', 'History', 'English'], 'gender': 'Female'}

# Removing a key-value pair
del student["grade"]
print("Dictionary after removing 'grade':", student)
# Output: Dictionary after removing 'grade': {'name': 'Alice', 'age': 21, 'courses':
['Math', 'History', 'English'], 'gender': 'Female'}
```

In this example:
- The `student` dictionary has keys like `"name"`, `"age"`, `"grade"`, and `"courses"`, each associated with corresponding values.

- Values can be accessed, modified, added, and removed using dictionary operations.

Q.11. What is function in Python? Explain with syntax and example.
Ans. **Function**: In Python, a function is a block of reusable code that performs a specific task. Functions help in organizing code, making it modular, and promoting code reuse. A function is defined using the `def` keyword, followed by the function name, parameters in parentheses, a colon, and the function body. Functions are crucial for structuring code, improving code readability, and making it easier to maintain and reuse code segments. They allow you to encapsulate functionality and execute it whenever needed by calling the function with appropriate arguments.

Syntax:
```
def function_name(parameter1, parameter2, ...):
 # Function body
 # Code to be executed when the function is called
 return result # Optional: The function can return a value
```

Example:
```
def calculate_square(number):
 square = number ** 2
 return square
def greet_person(name):
 greeting = f"Hello, {name}! How are you today?"
 return greeting
result1 = calculate_square(5)
print("Square of 5:", result1)
result2 = greet_person("Alice")
print(result2)
```

Output:
```
Square of 5: 25
Hello, Alice! How are you today?
```

In this example, we define two functions:
1. `calculate_square`: This function takes a parameter `number` and calculates its square. The result is then returned.
2. `greet_person`: This function takes a parameter `name` and generates a greeting message. The result is returned as a string.

We then call these functions with specific arguments (`calculate_square(5)` and `greet_person("Alice")`) and store the results in variables (`result1` and `result2`).

Q.12. Explain LEGB Rule with example.
Ans. The **LEGB rule** is a concept in Python that defines the order in which the interpreter looks for names (identifiers or variables). LEGB stands for Local, Enclosing, Global, and Built-in, representing the four scopes in which Python searches for a name. The LEGB rule guides Python in determining the value of a name during execution. If a name is not found in the local scope, Python searches in the enclosing, then global, and finally the built-in scope.

1. Local (L):
- Local scope refers to the current function or block where a variable is declared.
- Variables defined inside a function are in the local scope and are accessible only within that function.

Example:
```
def my_function():
    local_variable = "I am local"
    print(local_variable)

my_function()
# Output: I am local

# The following line would raise an error since local_variable is not accessible outside the function.
# print(local_variable)
# NameError: name 'local_variable' is not defined
```

2. Enclosing (E):
- Enclosing scope refers to the scope of the containing function when dealing with nested functions.
- It allows inner functions to access variables from the outer (enclosing) function.

Example:
```
def outer_function():
    outer_variable = "I am outer"

    def inner_function():
        print(outer_variable)

    inner_function()

outer_function()
# Output: I am outer
```

3. Global (G):
- Global scope refers to the top-level scope of the script or module.
- Variables defined outside of any function or block are in the global scope and are accessible throughout the script.

Example:

```
global_variable = "I am global"

def my_function():
    print(global_variable)

my_function()
# Output: I am global

# Accessing the global variable outside the function
print(global_variable)
# Output: I am global
```

4. Built-in (B):
- Built-in scope refers to the built-in names that Python provides, such as `print()`, `len()`, etc.
- These names are always available without the need for import.

Example:

```
# Using the built-in function len()
my_list = [1, 2, 3, 4, 5]
length = len(my_list)
print(length)
# Output: 5
```

Q.13. Explain arguments as objects and argument calling by keyword in python.

Ans. In Python, **arguments** are the values that you pass to a function when you call it. There are two main types of arguments: positional arguments and

keyword arguments. Additionally, Python treats function arguments as objects, meaning they
can be assigned to variables, passed around, and manipulated like any other object.

1. Arguments as Objects:

In Python, function arguments are treated as objects, and they are passed by object reference. This means that when you pass an argument to a function, you are passing a reference to the object, not a copy of the object. Changes made to the object inside the function may affect the original object outside the function.

Example:

```
def modify_list(my_list):
 my_list.append(4)
 my_list = [10, 20, 30] # This assignment creates a new local reference

my_numbers = [1, 2, 3]
modify_list(my_numbers)
print(my_numbers) # Output: [1, 2, 3, 4]
```

In this example, the `modify_list` function modifies the original list by appending the value 4. However, the assignment `my_list = [10, 20, 30]` inside the function creates a new local reference, and it doesn't affect the original list outside the function.

2. Argument Calling by Keyword:

When calling a function in Python, you can pass arguments by position or by keyword. Passing arguments by keyword allows you to specify which parameter each argument corresponds to, regardless of their order in the function definition.

Example:

```
def display_info(name, age, city):
 print(f"Name: {name}, Age: {age}, City: {city}")
# Calling the function with arguments by keyword
display_info(age=25, name="Alice", city="Wonderland")
```

In this example, the function `display_info` is called with arguments specified by their parameter names. This allows you to pass the arguments in any order, making the code more readable and self-explanatory.

Q.14. What is default arguments? Explain.

Ans. **<u>Default arguments</u>**: In Python, default arguments are parameters in a function that have a predefined value. When calling a function, if a value for a default argument is not provided, the function uses the default value specified in the function definition. Default arguments allow you to define functions with a level of flexibility, as they provide default values for certain parameters. Default arguments must come after non-default arguments in the function definition. The default values are evaluated only once when the function is defined, not each time the function is called.
Default arguments are especially useful when you want to provide a commonly used value for a parameter in most cases but still allow the flexibility to customize it when necessary. They make functions more versatile and easier to use in different situations.

Syntax:

```
def function_name(param1, param2=default_value2, param3=default_value3, ...):
   # Code to be executed
```

param1: A required parameter without a default value.
param2, param3, ...: Parameters with default values.

Example:

```
def greet(name, greeting="Hello", punctuation="!"):
   """
   Greets a person with a customizable greeting and punctuation.
   """
   message = f"{greeting}, {name}{punctuation}"
   return message

# Calling the function with and without specifying default arguments
print(greet("Alice"))  # Output: Hello, Alice!
print(greet("Bob", greeting="Hi"))  # Output: Hi, Bob!
print(greet("Charlie", punctuation="!!!"))  # Output: Hello, Charlie!!!
```

In this example:
The greet function has three parameters: name, greeting with a default value of "Hello", and punctuation with a default value of "!".
When calling the function, you can provide values for name, greeting, and/or punctuation. If not provided, the default values are used.

Q.15. Write a note on Argument tuples and argument dictionaries in python.

Ans. <u>Argument tuples</u>: In Python, argument tuples, often denoted as `*args`, provide a way to handle a variable number of positional arguments in a function. This feature allows you to create functions that can accept and process any number of input values without explicitly specifying the number of parameters. The term "args" is a convention, and you could use any name preceded by an asterisk. Using argument tuples with `*args` enhances the flexibility and generality of functions, allowing them to accommodate a dynamic number of input values.

Syntax:

```
def function_name(*args):
   # Code to be executed
   # Access elements in 'args' as a tuple
```

Example:

```
def sum_values(*args):
   total = 0
   for value in args:
      total += value
   return total

result = sum_values(1, 2, 3, 4, 5)
print(result)  # Output: 15
```

In this example, the `sum_values` function takes any number of arguments and calculates their sum. The `*args` syntax collects the positional arguments into a tuple named `args`.

- Using `*args` allows a function to accept any number of positional arguments.
- The collected arguments are treated as a tuple within the function, and you can iterate over them or access individual elements.
- Positional arguments specified in the function call are matched to the parameters in the same order as they appear in the function definition.
- You can use `*args` along with regular parameters, but `*args` must come after the regular parameters.

<u>Argument dictionary</u>: In Python, the term "argument dictionary" usually refers to using the double-asterisk notation `**kwargs` in a function definition to handle a variable number of keyword arguments. This allows you to pass a dictionary of named parameters to a function, providing flexibility and extensibility in function design. Using an argument dictionary with `**kwargs` in Python functions provides a powerful and flexible way to work with named parameters, allowing for dynamic and extensible function designs.

Using Argument Dictionary (kwargs):**

```
def process_student(**kwargs):
    """
    Process information about a student using a dictionary of parameters.
    """
    if "name" in kwargs:
        print(f"Student Name: {kwargs['name']}")
    if "age" in kwargs:
        print(f"Student Age: {kwargs['age']}")
    if "grade" in kwargs:
        print(f"Student Grade: {kwargs['grade']}")

# Calling the function with an argument dictionary
student_info = {"name": "Alice", "age": 20, "grade": "A"}
process_student(**student_info)
```

In this example, the `process_student` function uses `**kwargs` to accept a variable number of keyword arguments. The function checks for specific keys in the `kwargs` dictionary and processes the corresponding values.

Benefits of Using Argument Dictionary:

1. Flexibility:
  - Allows functions to handle a dynamic set of named parameters without explicitly defining them in the function signature.
2. Extensibility:
  - Enables easy extension of function parameters without modifying existing function calls.
3. Readability:
  - Makes function calls more readable and self-explanatory when passing a dictionary of parameters.

Q.16. List function rules about defining, calling.

Ans. When defining and calling functions in Python, there are certain rules and conventions to follow.

Defining Functions:

1. Function Definition Syntax:
- Use the `def` keyword to define a function.
- Followed by the function name and parentheses containing parameters.
- End the line with a colon.

```
def function_name(parameter1, parameter2, ...):
    # Function body
    # ...
```

2. Indentation:
- Indent the code within the function body.
- Use a consistent number of spaces or tabs.

```
def example_function():
    # Indented code
    print("Hello, World!")
```

3. Docstring (Optional):
- Add a docstring to describe the purpose of the function.
- Helps in providing documentation for the function.

```
def example_function():
    """
    This is a docstring describing the function.
    """
    print("Hello, World!")
```

4. Return Statement (Optional):
- Use the `return` statement to specify the value to be returned by the function (optional).

```
def add_numbers(a, b):
    return a + b
```

<u>**Calling Functions:**</u>

1. Function Name and Parentheses:
- Call a function by using its name followed by parentheses.
- Provide values for parameters inside the parentheses.

```
result = add_numbers(5, 3)
```

2. Arguments:
- Match the number and order of arguments when calling a function with the function definition.
- Positional arguments are matched based on their order.

```
  greet("Alice", age=25)
```

3. Keyword Arguments:
  - Use keyword arguments to explicitly specify values for parameters by name.

```
  print_info(name="Bob", age=30)
```

4. Default Values:
  - If a function has default parameter values, you can skip providing values for those parameters.

```
  def greet(name, greeting="Hello"):
      print(f"{greeting}, {name}!")
```

5. Return Values:
  - Capture the return value of a function using the assignment statement.

```
  result = add_numbers(5, 3)
```

6. Variable Scope:
  - Understand the scope of variables within a function.
  - Variables defined inside a function are local by default.

```
  def example_function():
      local_variable = 10
      print(local_variable)
```

Q.17. Explain apply and map statements.

Ans. In Python, both `apply` and `map` are used for applying a function to elements in an iterable (e.g., list, tuple) but have different use cases and behaviors. However, it's important to note that the `apply` function is not a built-in function in Python. If you're referring to applying a function to elements in an iterable, `map` is a more appropriate term. `map` is a built-in function in Python used for applying a function to each item in an iterable, while `apply` is a Pandas DataFrame method used for applying a function along the axis of the DataFrame. The choice between them depends on the specific use case and data structure.

`map` Function:

The `map` function in Python is used to apply a specified function to all the items in an iterable (e.g., list) and returns an iterator that produces the results.

Syntax:

```
map(function, iterable, ...)
```

- `function`: The function to apply to each item in the iterable.

- `iterable`: The iterable (e.g., list) whose elements will be processed.

Example:

```
# Function to square a number
def square(x):
   return x ** 2

# Using map to apply the square function to a list of numbers
numbers = [1, 2, 3, 4]
squared_numbers = map(square, numbers)

# Converting the iterator to a list
result_list = list(squared_numbers)
print(result_list)  # Output: [1, 4, 9, 16]
```

In this example, the `square` function is applied to each element in the `numbers` list using `map`.

<u>`apply` (Pandas DataFrame):</u>

If you are referring to `apply` in the context of Pandas, it is a method available for DataFrames. The `apply` method is used to apply a function along the axis of a DataFrame (either rows or columns).

Syntax:

```
DataFrame.apply(func, axis=0, raw=False, result_type=None, args=(), **kwds)
```

- `func`: The function to apply.
- `axis`: The axis along which the function will be applied (0 for columns, 1 for rows).
- `raw`: Boolean, determines if the function receives data as ndarray.
- `result_type`: Specify the dtype of the result.
- `args`: Additional positional arguments to pass to `func`.
- `**kwds`: Additional keyword arguments to pass to `func`.

Example:

```
import pandas as pd

# Creating a DataFrame
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}
df = pd.DataFrame(data)

# Function to square a number
```

```
def square(x):
   return x ** 2

# Applying the square function to each column using apply
result_df = df.apply(square)
print(result_df)
# Output:
#    A  B
# 0  1  16
# 1  4  25
# 2  9  36
```

In this Pandas example, the `square` function is applied to each column of the DataFrame using `apply` with the default `axis=0`.

Q.18. Explain anonymous function in python.

Ans. Anonymous function: In Python, an anonymous function is a function defined without a name. These functions are created using the `lambda` keyword. They are also referred to as lambda functions. Lambda functions are useful for short, simple operations where a full function definition with a name may seem unnecessary. While lambda functions offer conciseness, they are limited in functionality compared to regular functions. They are most suitable for short, simple operations. For more complex tasks or functions requiring multiple expressions, a regular function with a name is often preferred.

Syntax of Lambda Function:

```
lambda arguments: expression
```

- `lambda`: Keyword indicating the creation of a lambda function.
- `arguments`: Parameters the lambda function takes.
- `expression`: The operation or computation to be performed.

Example of Lambda Function:

```
# Regular function
def add(x, y):
   return x + y

# Equivalent lambda function
add_lambda = lambda x, y: x + y

# Using the lambda function
```

```
result = add_lambda(3, 5)
print(result)  # Output: 8
```

In this example, the lambda function `lambda x, y: x + y` performs the same operation as the regular function `add`. Lambda functions are often used for short-lived operations and can be handy in situations where a function is required for a brief period.

Use Cases of Lambda Functions:

1. As Arguments to Higher-Order Functions:

- Lambda functions can be passed as arguments to functions that accept functions (e.g., `map`, `filter`, `sorted`).

```
  numbers = [1, 4, 2, 7, 5]
  squared_numbers = map(lambda x: x ** 2, numbers)
  print(list(squared_numbers))  # Output: [1, 16, 4, 49, 25]
```

2. In List Comprehensions:

- Lambda functions can be used in list comprehensions for concise transformations

```
  numbers = [1, 2, 3, 4]
  squared_numbers = [(lambda x: x ** 2)(num) for num in numbers]
  print(squared_numbers)  # Output: [1, 4, 9, 16]
```

3. On-the-Fly Functions:

- When a simple, one-time-use function is needed without explicitly defining a full function.

```
  # Sorting a list of tuples based on the second element
  pairs = [(1, 5), (3, 2), (2, 8)]
  sorted_pairs = sorted(pairs, key=lambda x: x[1])
  print(sorted_pairs)  # Output: [(3, 2), (1, 5), (2, 8)]
```

UNIT 2

Q.1. What is Module in Python? How to create a module?

Ans.

Module: In Python, a module is a file containing Python definitions, functions, and statements. It serves as a way to organize code and make it reusable across different programs. A module can include variables, functions, and classes that can be imported and used in other Python scripts or modules. Modules are a fundamental concept in Python programming, contributing to the language's readability and scalability. A module has a name specified by the filename without the .py extension. For example, if you have a file called pricing.py, the module name is pricing.

Creating a Module:
To create a module in Python, follow these steps:
1. Create a Python File:
- Start by creating a new Python file with a `.py` extension. This file will serve as your module.

```
# mymodule.py

def greet(name):
   return f"Hello, {name}!"

def add_numbers(a, b):
   return a + b

my_variable = 42
```

2. Save the Module:
- Save the file with a meaningful name, for example, `mymodule.py`.

3. Using the Module:
- In another Python script or interactive session, you can use the module by importing it.

```
 # main_script.py
import mymodule

# Using functions from the module
greeting = mymodule.greet("Alice")
print(greeting)

# Using variables from the module
result = mymodule.add_numbers(3, 5)
print(result)

print(mymodule.my_variable)
```

In this example, the `main_script.py` script imports the `mymodule` module and uses its functions and variables.

Module Search Path:
When you import a module, Python looks for it in certain directories. The directories are determined by the `sys.path` list, and Python searches for modules in these directories. You can check the current search path using:

```
import sys
print(sys.path)
```

By default, Python includes the current working directory and the standard library directories in the search path.

Module Namespaces:
When you import a module, its contents become part of a namespace. The module name is used as a prefix to access its contents. For example:

```
import mymodule
mymodule.greet("Bob")
```

Here, `mymodule` is the namespace, and `greet` is a function within that namespace.

Q.2. Explain packages in Python.
Ans.
Package: In Python, a package is a way of organizing related modules into a single directory hierarchy. This provides a modular and organized structure to large Python projects. A package is essentially a directory that contains a special file named `__init__.py` (which can be empty) and other Python modules or subpackages. Packages in Python provide a structured and organized way to manage and distribute code. They play a crucial role in creating scalable and maintainable projects by promoting modularity, encapsulation, and code reuse.

Structure of a Package:

```
my_package/
|-- __init__.py
|-- module1.py
|-- module2.py
|-- subpackage/
|   |-- __init__.py
|   |-- submodule1.py
|   |-- submodule2.py
```

In this example:
- The `my_package` directory is the package.
- The `__init__.py` files indicate that the directory should be treated as a package or subpackage.
- `module1.py` and `module2.py` are modules within the package.
- The `subpackage` is a subpackage containing its own modules.

Creating a Package:

To create a package, follow these steps:

1. Create a Directory: Create a directory for your package.

```
  mkdir my_package
```

2. Add `__init__.py`: Add an empty `__init__.py` file to the package directory.

```
  touch my_package/__init__.py
```

3. Add Modules: Add Python modules or subpackages to the package directory.

```
  touch my_package/module1.py
  touch my_package/module2.py
  mkdir my_package/subpackage
  touch my_package/subpackage/__init__.py
  touch my_package/subpackage/submodule1.py
  touch my_package/subpackage/submodule2.py
```

Using a Package:

To use modules or subpackages from a package, you can import them in your Python scripts:

```
# main_script.py

from my_package import module1
from my_package.subpackage import submodule1

module1.function_from_module1()
submodule1.function_from_submodule1()
```

Benefits of Packages:

1. Modularity: Packages provide a way to organize code into separate modules, improving modularity and code readability.

2. Namespace Management: Packages help in managing namespaces by avoiding naming conflicts between modules from different packages.

3. Encapsulation: Modules within a package can encapsulate related functionality and data, providing a cleaner and more structured design.

4. Code Reusability: Packages support code reusability by allowing you to import and use modules across different parts of your project.

5. Distribution: Packages facilitate the distribution of code, making it easier to share and reuse code among different projects.

Importing Modules and Subpackages:

You can import modules and subpackages from a package using relative or absolute imports:

```
# Relative import
from . import module1
from .subpackage import submodule1

# Absolute import
from my_package import module1
from my_package.subpackage import submodule1
```

Q.3. Explain class and objects in Python with example.

Ans. Classes in Python:

In Python, a class is a user-defined data type that contains both the data itself and the methods that may be used to manipulate it. In a sense, classes serve as a template to create objects. They provide the characteristics and operations that the objects will employ.

In Python, a class can be created by using the keyword class, followed by the class name. The syntax to create a class is given below.

Syntax:

```
class ClassName:
   #statement_suite
```

Example:

```
class Person:
   def __init__(self, name, age):
      # This is the constructor method that is called when creating a new Person
object
      # It takes two parameters, name and age, and initializes them as attributes
of the object
      self.name = name
      self.age = age
   def greet(self):
      # This is a method of the Person class that prints a greeting message
      print("Hello, my name is " + self.name)
```

Here, Name and age are the two properties of the Person class. Additionally, it has a function called greet that prints a greeting.

<u>**Objects in Python:**</u>

An object is a particular instance of a class with unique characteristics and functions. After a class has been established, you may make objects based on it. By using the class constructor, you may create an object of a class in Python. The object's attributes are initialised in the constructor, which is a special procedure with the name __init__.

Syntax: #Declare an object of a class

```
object_name = Class_Name(arguments)
```

Example:

```
class Person:
   def __init__(self, name, age):
      self.name = name
      self.age = age
   def greet(self):
      print("Hello, my name is " + self.name)

# Create a new instance of the Person class and assign it to the variable
person1
person1 = Person("Ayan", 25)
person1.greet()
```

Q.4. Write a note on Object Constructors and Object Destructors.
Ans. **Object constructor** is a special method called when an object is created from a class. It initializes the object's attributes or performs any setup necessary for the object to function correctly. In Python, the constructor method is named __init__.

Purpose of Constructors:
Initialization: Constructors initialize the attributes of an object with specified values or default values. This ensures that the object starts with a known state.
Setup: Constructors can perform setup tasks, such as opening a connection, allocating resources, or configuring the object to be in a usable state.
Parameterized Initialization: Constructors can take parameters, allowing for parameterized initialization based on the values provided during object creation

Syntax:
```
class MyClass:
  def __init__(self, param1, param2, ...):
    # Initialization code here
    self.attribute1 = param1
    self.attribute2 = param2
    # ...
```
The __init__ method is called automatically when an object is created from the class.
The self parameter represents the instance of the class (the object being created) and is used to refer to its attributes.

Object destructors are special methods used for cleaning up resources and performing necessary cleanup operations before an object is deleted or goes out of scope. The destructor method is named __del__ and is automatically called when an object is no longer in use or is being explicitly deleted.
Destructors are useful for releasing resources like file handles, network connections, or other external resources when an object is no longer needed.
While Python uses automatic garbage collection for memory management, destructors provide a way to clean up additional resources beyond memory.

Syntax:
```
def __del__(self):
  # Cleanup code and resource release
```

Example:
```
class MyClass:
    # Constructor
    def __init__(self, name):
        self.name = name
        print(f"{self.name} created.")

    # Instance method
    def display_info(self):
        print(f"Object {self.name} is active.")

    # Destructor
    def __del__(self):
        print(f"{self.name} is being destroyed.")

# Creating objects
obj1 = MyClass("Object1")
obj2 = MyClass("Object2")

# Calling instance method
obj1.display_info()  # Output: Object Object1 is active.

# Deleting objects explicitly
del obj1
del obj2

# Creating another object
obj3 = MyClass("Object3")

# Object3 will be automatically destroyed when it goes out of scope
```

```
Output: Object Object1 created.
Object Object2 created.
Object Object1 is active.
Object1 is being destroyed.
Object2 is being destroyed.
Object Object3 created.
Object3 is being destroyed.
```

Q.5. Explain class inheritance in python with example.
Ans. **<u>Class inheritance</u>** is a mechanism that allows a new class (subclass or derived class) to inherit attributes and methods from an existing class

(superclass or base class). This promotes code reuse and allows for the creation of more specialized classes based on existing ones. Inheritance in Python facilitates the creation of a hierarchy of classes, allowing for the reuse and extension of code. It supports the development of more specialized classes while maintaining a clear and organized structure.

Example:

```
# Base class (superclass)
class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        pass  # Placeholder method, to be overridden by subclasses

# Derived class (subclass)
class Dog(Animal):
    def make_sound(self):
        return "Woof!"

# Another derived class (subclass)
class Cat(Animal):
    def make_sound(self):
        return "Meow!"

# Creating instances of the derived classes
dog_instance = Dog("Buddy")
cat_instance = Cat("Whiskers")

# Calling methods of the base class and derived classes
print(f"{dog_instance.name} says: {dog_instance.make_sound()}")
print(f"{cat_instance.name} says: {cat_instance.make_sound()}")
```

- The `Animal` class is the base class with a constructor (`__init__`) and a placeholder method (`make_sound`).
- The `Dog` and `Cat` classes are derived classes that inherit from the `Animal` class.
- Both `Dog` and `Cat` override the `make_sound` method, providing their own implementation.

```
Output:
Buddy says: Woof!
Whiskers says: Meow!
```

1. The `Dog` and `Cat` classes inherit the `__init__` method from the `Animal` class, allowing them to initialize the `name` attribute.
2. Both `Dog` and `Cat` override the `make_sound` method to provide specific implementations for the sound each animal makes.
3. Instances of `Dog` and `Cat` can be created, and their methods can be called.

<u>**Types of Inheritance:**</u>

1. Single Inheritance: A class inherits from only one base class.

```
class DerivedClass(BaseClass):
    # class body
```

2. Multiple Inheritance: A class can inherit from multiple base classes.

```
class DerivedClass(BaseClass1, BaseClass2, ...):
    # class body
```

3. Multilevel Inheritance: A derived class can act as a base class for another class.

```
class BaseClass:
    # class body

class IntermediateClass(BaseClass):
    # class body

class DerivedClass(IntermediateClass):
    # class body
```

Q.6. Explain __init__() method with an example.

Ans. The `__init__()` method in Python is a special method, also known as the constructor. It is automatically called when an object is created from a class. The primary purpose of the `__init__()` method is to initialize the attributes of the object. The `__init__()` method is a fundamental part of the object initialization process in Python classes. It helps ensure that objects are properly configured when they are created, providing a way to set up initial states and attributes.

Example:

```
class Car:
    def __init__(self, make, model, year):
        # Initializing instance variables
        self.make = make
        self.model = model
        self.year = year
        # Additional setup code can go here

    def display_info(self):
        return f"{self.year} {self.make} {self.model}"

# Creating instances of the Car class
car1 = Car("Toyota", "Camry", 2020)
car2 = Car("Honda", "Accord", 2021)

# Accessing instance variables and calling methods
print(car1.display_info())  # Output: 2020 Toyota Camry
print(car2.display_info())  # Output: 2021 Honda Accord
```

In this example:
- The `Car` class has an `__init__()` method that takes three parameters (`make`, `model`, and `year`) in addition to the `self` parameter, which refers to the instance being created.
- Inside the `__init__()` method, instance variables (`self.make`, `self.model`, `self.year`) are assigned values based on the provided arguments.
- When you create instances of the `Car` class (`car1` and `car2`), the `__init__()` method is automatically called, initializing the attributes for each object.

Usage of `__init__()`:

1. Attribute Initialization: The `__init__()` method is commonly used to initialize instance variables with values passed during object creation.

```
   def __init__(self, parameter1, parameter2, ...):
       self.attribute1 = parameter1
       self.attribute2 = parameter2
       # ...
```

2. Additional Setup: You can include additional setup code or perform computations inside the `__init__()` method.

```
  def __init__(self, parameter1, parameter2, ...):
      # Initialization code
      # Additional setup or computations
```

3. Default Values: You can provide default values for parameters to make them optional during object creation.

```
  def __init__(self, parameter1, parameter2="default_value"):
      # ...
```

Q.7. Write a Python program for instantiating a class.

Ans.
```
class Student:
    # Constructor (__init__ method)
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade

    # Instance method
    def display_info(self):
        print(f"Student Name: {self.name}")
        print(f"Age: {self.age}")
        print(f"Grade: {self.grade}")

# Instantiating the Student class
student1 = Student(name="Alice", age=18, grade="A")
student2 = Student(name="Bob", age=17, grade="B")

# Calling instance method to display information
print("Information for Student 1:")
student1.display_info()

print("\nInformation for Student 2:")
student2.display_info()
```

In this program:
- We define a `Student` class with an `__init__` method for initializing attributes (`name`, `age`, `grade`) and a `display_info` method for displaying student information.

- We create two instances (`student1` and `student2`) of the `Student` class with different attribute values.
- We call the `display_info` method for each instance to print student information.

```
Output:
Information for Student 1:
Student Name: Alice
Age: 18
Grade: A

Information for Student 2:
Student Name: Bob
Age: 17
Grade: B
```

Q.8. What is destructor? Write a program to demonstrate destructor.
Ans. **Destructor** is a special method called `__del__` that is automatically invoked when an object is about to be destroyed or deallocated. It is used for cleaning up resources or performing finalization tasks before an object is removed from memory. However, it's important to note that the `__del__` method is not guaranteed to be called immediately upon object destruction due to Python's garbage collection mechanism.

```
class Book:
    # Constructor (__init__ method)
    def __init__(self, title, author):
        self.title = title
        self.author = author

    # Instance method
    def display_info(self):
        print(f"Book Title: {self.title}")
        print(f"Author: {self.author}")

    # Destructor (__del__ method)
    def __del__(self):
        print(f"Book {self.title} by {self.author} is being destroyed.")

# Creating an instance of the Book class
```

```
book1 = Book(title="The Great Gatsby", author="F. Scott Fitzgerald")

# Calling the display_info method
print("Book Information:")
book1.display_info()

# Deleting the object explicitly (triggering the __del__ method)
del book1
```

In this program:
- The `Book` class has an `__init__` method for initializing attributes (`title`, `author`), a `display_info` method for displaying book information, and a `__del__` method as the destructor.
- An instance of the `Book` class (`book1`) is created with specific attribute values.
- The `display_info` method is called to print information about the book.
- The `del` statement is used to explicitly delete the object, triggering the `__del__` method.

Output:

```
Book Information:
Book Title: The Great Gatsby
Author: F. Scott Fitzgerald
Book The Great Gatsby by F. Scott Fitzgerald is being destroyed.
```

In this example, the `__del__` method is called when the `book1` object is explicitly deleted using the `del` statement. Keep in mind that relying solely on `__del__` for resource cleanup is not always recommended, and alternative approaches such as using context managers (`with` statement) or the `__enter__` and `__exit__` methods are often preferred for managing resources more explicitly.

Q.9. What is an exception? What happens when an exception occurs?
Ans. <u>**Exception**</u> is an event that occurs during the execution of a program and disrupts the normal flow of instructions. Exceptions are raised when an error or unexpected condition occurs, and they allow the program to handle such situations gracefully. Python provides a mechanism to catch and handle exceptions using the `try`, `except`, `else`, and `finally` blocks.

<u>**When an exception occurs:**</u>

1. Exception is Raised: An exception is raised when an error or exceptional condition occurs during the execution of the program. This could be due to various reasons, such as a division by zero, trying to access an index that doesn't exist, or attempting to open a non-existent file.

2. Program Execution Halts: The normal flow of the program is interrupted, and the Python interpreter looks for an exception handler to handle the raised exception.

3. Exception Handlers: The program searches for an appropriate `except` block that can handle the specific type of exception raised. If a matching `except` block is found, its code is executed.

4. Exception Propagation: If no suitable `except` block is found in the current scope, the exception propagates up the call stack, looking for an outer `try` block or a global exception handler.

5. Termination or Error Message: If the exception is not handled anywhere in the program, the program terminates, and an error message is displayed, including information about the type of exception and its traceback.

Example:

```
try:
   # Code that may raise an exception
   num1 = int(input("Enter a number: "))
   num2 = int(input("Enter another number: "))
   result = num1 / num2
   print(f"Result: {result}")

except ZeroDivisionError:
   # Handling division by zero exception
   print("Error: Cannot divide by zero.")

except ValueError:
   # Handling invalid input (non-integer) exception
   print("Error: Please enter valid integers.")

else:
   # Code to execute if no exception is raised
   print("Division operation successful.")
```

```
finally:
    # Code to execute regardless of whether an exception occurred or not
    print("Execution complete.")
```

In this example:
- The `try` block contains code that may raise exceptions, such as dividing by zero or encountering non-integer input.
- The `except` blocks catch specific types of exceptions and provide custom error messages.
- The `else` block contains code that is executed if no exception is raised in the `try` block.
- The `finally` block contains code that is executed regardless of whether an exception occurred or not.

Q.10. What is exception handling? Explain try…except…else.
Ans. **Exception handling** in Python involves using the `try`, `except`, `else`, and optionally, the `finally` blocks to manage and respond to exceptions that may occur during the execution of a program. Exception handling in this manner helps in gracefully handling errors, preventing the program from terminating abruptly, and providing feedback to users or developers about the nature of the encountered issues.

`try` Block: The `try` block encloses the code that might raise an exception. It is the portion of code where you anticipate the possibility of an exception.

`except` Block: The `except` block follows the `try` block and contains the code that will be executed if a specified exception occurs during the execution of the `try` block. You can have multiple `except` blocks to handle different types of exceptions.

`else` Block: The `else` block is executed if no exceptions occur in the `try` block. It is optional and typically contains code that should run when the `try` block completes successfully.

Syntax:
```
try:
    # Code that may raise an exception
    # ...
except ExceptionType1 as e1:
    # Code to handle exception of type ExceptionType1
    # ...
except ExceptionType2 as e2:
    # Code to handle exception of type ExceptionType2
    # ...
else:
    # Code to execute if no exception is raised in the try block
    # ...
f inally:
    # Code that will be executed regardless of whether an exception occurred or
not
    # ...
```

Example:

```
try:
    num1 = int(input("Enter a number: "))
    num2 = int(input("Enter another number: "))
    result = num1 / num2
    print(f"Result: {result}")

except ZeroDivisionError as zde:
    print(f"Error: Cannot divide by zero. ({zde})")

except ValueError as ve:
    print(f"Error: Please enter valid integers. ({ve})")

else:
    print("Division operation successful.")

finally:
    print("Execution complete.")
```

In this example:
- The `try` block contains code that might raise a `ZeroDivisionError` or `ValueError`.
- The `except` blocks catch these specific exceptions, providing custom error messages along with the exception objects (`zde` and `ve`).
- The `else` block contains code that will run if no exception occurs in the `try` block.

- The `finally` block contains code that will be executed regardless of whether an exception occurred or not. It is commonly used for cleanup operations.

Q.11. Write a note on try…finally statement.
Ans. The `try...finally` statement in Python is used for exception handling and cleanup operations. It combines the `try` and `finally` blocks to ensure that a specified block of code in the `finally` block is executed, whether an exception occurs in the `try` block or not. This is particularly useful for performing cleanup tasks, such as closing files or releasing resources, regardless of the program's flow.
`try...finally` is a valuable construct for scenarios where certain operations must be performed, regardless of whether an exception occurs or not. It plays a crucial role in maintaining program integrity and ensuring proper resource cleanup.

Syntax:

```
try:
   # Code that may raise an exception
   # ...
finally:
   # Code that will be executed regardless of whether an exception occurred or
not
   # ...
```

Example:

```
try:
   file = open("example.txt", "r")
   content = file.read()
   print("File content:", content)

except FileNotFoundError:
   print("Error: The file does not exist.")

finally:
   if file:
      file.close()
   print("File closed.")
```

In this example:
- The `try` block attempts to open a file, read its content, and print it.

- The `except` block catches a `FileNotFoundError` if the specified file does not exist and prints an error message.
- The `finally` block contains code that ensures the file is closed, regardless of whether an exception occurred or not. The `file.close()` statement is placed in the `finally` block to guarantee execution.

Use Cases:
1. Resource Cleanup: It is commonly used for releasing resources like closing files, database connections, or network connections. This ensures that resources are properly released, even if an exception occurs.

2. Cleanup Operations: Any operation that needs to be executed regardless of the outcome of the `try` block can be placed in the `finally` block. This is useful for cleanup tasks.

3. Guaranteeing Execution: The `finally` block guarantees that its code will be executed, providing a way to maintain the integrity of the program even in the presence of exceptions.

- While `try...finally` ensures the `finally` block is executed, it doesn't catch or handle exceptions. If an exception occurs in the `try` block, it will propagate to the outer scope after the `finally` block is executed.
- It's important to use `try...finally` judiciously, especially when handling resources, to avoid potential issues like resource leaks. Consider using context managers and the `with` statement for more explicit and cleaner resource management.

Q.12. Explain raise and assert statement with syntax.
Ans. `raise` and `assert` statements are used to handle exceptional cases and perform sanity checks in your code. Both `raise` and `assert` statements are tools for handling exceptional cases and enforcing certain conditions in your code. While `raise` is more general-purpose and allows you to raise custom exceptions, `assert` is typically used for debugging and catching programming errors du ring development. It's important to use them judiciously and provide informative error messages to aid in debugging.

`raise` Statement: The `raise` statement is used to explicitly raise an exception. This can be useful when you want to signal that a certain condition has occurred that should be treated as an error.

Syntax:
```
raise ExceptionType("Optional error message")
```

- `ExceptionType`: The type of exception to be raised (e.g., `ValueError`, `TypeError`).
- `"Optional error message"`: An optional human-readable error message associated with the raised exception.

Example:
```
def divide(a, b):
   if b == 0:
      raise ValueError("Cannot divide by zero")
   return a / b

try:
   result = divide(10, 0)
except ValueError as ve:
   print(f"Error: {ve}")
```

In this example, the `divide` function raises a `ValueError` if an attempt is made to divide by zero.

<u>`assert` Statement:</u> The `assert` statement is used for debugging purposes to check if a given condition is `True`. If the condition is `False`, it raises an `AssertionError` exception with an optional error message.

Syntax:
```
assert condition, "Optional error message"
```

- `condition`: The expression to be evaluated. If it evaluates to `False`, the `assert` statement triggers an `AssertionError`.
- `"Optional error message"`: An optional human-readable error message associated with the failed assertion.

Example:
```
def calculate_discount(price, discount_rate):
   assert 0 <= discount_rate <= 1, "Discount rate should be between 0 and 1"
   discounted_price = price * (1 - discount_rate)
   return discounted_price
```

```
try:
    discounted_price = calculate_discount(100, 1.5)
except AssertionError as ae:
    print(f"Assertion Error: {ae}")
```

In this example, the `assert` statement checks if the discount rate is between 0 and 1. If not, an `AssertionError` is raised.