Python Assignment

Task 1 : A Jupyter Notebook is created to document the code and rename it as per the specification.

Taks 2: Python Functions

2.1 Create functions with different numbers of parameters and return types.

Functions can have different numbers of parameters, and they can return different types of values like strings, integers, or even other collections like lists.

```
# Example 1: No parameters, returns a string
def greet_world():
    return "Hello, World!"
print(greet_world())
```

```
Hello, World!
```

```
# Example 2: One parameter, returns an integer
def double_value(x):
    return x * 2
print(double_value(10))
```

```
20
```

```
# Example 3: Two parameters, returns a concatenated string
    def full_name(first_name, last_name):
        return f"{first_name} {last_name}"
    print(full_name("John", "Doe"))

    John Doe

# Example 4: Multiple parameters, returns a list
    def create_list(a, b, c):
        return [a, b, c]
    print(create_list(1, 2, 3))

    [1, 2, 3]

# Example 5: Three parameters, returns a dictionary
    def person_info(name, age, city):
        return {"Name": name, "Age": age, "City": city}
    print(person_info("Alice", 25, "New York"))

    {'Name': 'Alice', 'Age': 25, 'City': 'New York'}
```

2.2 Explore function scope and variable accessibility.

Variables declared inside functions are local to that function and cannot be accessed outside of it. Global variables, on the other hand, can be accessed inside a function but need to be explicitly declared if modified.

```
# Example 1: Local scope
def local_variable_example():
    local_var = 10
    return local_var

print(local_variable_example())
```

```
 10
```

```
# Example 2: Global variable usage
global_var = 20

def use_global():
    return global_var

print(use_global())
```

```
 20
```

```
# Example 3: Modify global variable inside a function
def modify_global():
    global global_var
    global_var += 10
    return global_var
print(modify_global())
```

```
30

# Example 4: Nonlocal variable (used in nested functions)
def outer_function():
    nonlocal_var = "I am outer"

    def inner_function():
        nonlocal nonlocal_var
        nonlocal_var = "I am modified by inner"
        return nonlocal_var

    inner_function()
    return nonlocal_var

print(outer_function())

I am modified by inner

# Example 5: Parameter shadowing
x = 50

def shadow_example(x):
    return x + 5  # Uses the local x

print(shadow_example(10))

15
```

2.3 Implement functions with default argument values.

A function can have parameters with default values. These default values will be used if the function is called without specifying those parameters.

```
 # Example 1: Single default argument
def greet(name="World"):
    return f"Hello, {name}!"

print(greet())
```

```
 Hello, World!
```

```
# Example 2: Two parameters, one with a default
def calculate_area(length, width=5):
    return length * width

print(calculate_area(10))
```

```
50
```

```
# Example 3: Multiple default arguments
def introduce(name="John", age=30, city="New York"):
    return f"My name is {name}, I am {age} years old, and I live in
{city}."
print(introduce())
```

```
 My name is John, I am 30 years old, and I live in New York.
```

```
# Example 4: Default value based on another argument
def add_with_offset(a, b=10):
    return a + b
print(add_with_offset(5))
```

```
 15
```

```
# Example 5: Default argument that changes dynamically
def append_item_to_list(item, items=None):
    if items is None:
        items = []
```

```
    items.append(item)
    return items

print(append_item_to_list("apple"))
```

```
['apple']
```

2.4 Write recursive functions.

A recursive function is a function that calls itself in order to solve a problem. An example is a factorial function, which multiplies a number by all the numbers below it

```
# Example 1: Factorial function
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))
```

```
120
```

```
# Example 2: Fibonacci sequence
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(6))
```

```
8
```

```
# Example 3: Sum of list elements
def sum_list(lst):
    if len(lst) == 0:
        return 0
    else:
        return lst[0] + sum_list(lst[1:])

print(sum_list([1, 2, 3, 4]))
```

```
 10
```

```
# Example 4: Count down to zero
def countdown(n):
    if n == 0:
        print("Blast off!")
    else:
        print(n)
        countdown(n - 1)

countdown(5)
```

```
 5 4 3 2 1 Blast off!
```

```
# Example 5: Reverse a string
def reverse_string(s):
    if len(s) == 0:
        return s
    else:
        return reverse_string(s[1:]) + s[0]

print(reverse_string("hello"))
```

```
 olleh
```

2.5 Demonstrate how to use docstrings to document functions.

Docstrings are used to document functions, explaining what they do, their parameters, and what they return. This is helpful for both the developer and others using the function.

```
# Example 1: Simple docstring

def greet(name):
    """
    Greets a person by their name.


    Parameters:

    name (str): The name of the person to greet.


    Returns:

    str: A greeting message.
    """
    return f"Hello, {name}!"


print(greet.__doc__)
```

```
# Example 2: Docstring with multiple parameters
def add_numbers(a, b):
    """
    Adds two numbers together.

    Parameters:
    a (int or float): The first number.
    b (int or float): The second number.

    Returns:
    int or float: The sum of a and b.
    """
    return a + b

print(add_numbers.__doc__)

# Example 3: Docstring for a function with default arguments
def describe_person(name, age=30):
    """
    Provides a description of a person.
```

```
    Parameters:
    name (str): The person's name.
    age (int, optional): The person's age. Defaults to 30.

    Returns:
    str: A description of the person.
    """
    return f"{name} is {age} years old."

print(describe_person.__doc__)

# Example 4: Docstring with a recursive function
def factorial(n):
    """
    Calculates the factorial of a number using recursion.

    Parameters:
    n (int): The number to calculate the factorial of.

    Returns:
    int: The factorial of n.
```

```
    """
    if n == 1:
        return 1
    return n * factorial(n - 1)

print(factorial.__doc__)

# Example 5: Docstring with a return type of None
def print_message():
    """
    Prints a simple message.

    Returns:
    None
    """
    print("Hello, world!")

print(print_message.__doc__)
```

Task 3 : Lambda Functions

3.1 Create simple lambda functions for various operations

```
# Example 1: Lambda function for adding two numbers
add = lambda a, b: a + b
print(add(5, 3))
 Output: 8

# Example 2: Lambda function for squaring a number
square = lambda x: x ** 2
print(square(4))
Output: 16

# Example 3: Lambda function for finding the maximum of two
numbers
maximum = lambda a, b: a if a > b else b
print(maximum(10, 15))
 Output: 15

# Example 4: Lambda function for checking if a number is even
is_even = lambda x: x % 2 == 0
print(is_even(7))
 Output: False

# Example 5: Lambda function for concatenating two strings
concat = lambda s1, s2: s1 + s2
print(concat("Hello, ", "World!"))
 Output: Hello, World!
```

3.2 Use lambda functions with built-in functions like map, filter, and reduce.

Lambda functions are commonly used with Python's built-in functions like `map`, `filter`, and `reduce` for operations on lists and other collections.

```
from functools import reduce

# Example 1: Using lambda with map (to square all numbers in a list)
numbers = [1, 2, 3, 4]
squares = list(map(lambda x: x ** 2, numbers))
print(squares)
Output: [1, 4, 9, 16]

# Example 2: Using lambda with filter (to filter even numbers from a list)
numbers = [1, 2, 3, 4, 5, 6]
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)
 Output: [2, 4, 6]

# Example 3: Using lambda with reduce (to find the product of all numbers in a list)
numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product)
 Output: 24

# Example 4: Using lambda with map (to convert temperatures from Celsius to Fahrenheit)
celsius = [0, 10, 20, 30]
fahrenheit = list(map(lambda c: (c * 9/5) + 32, celsius))
print(fahrenheit)
 Output: [32.0, 50.0, 68.0, 86.0]
```

```
# Example 5: Using lambda with filter (to find words longer than 3 characters)
words = ["hi", "hello", "sun", "cat", "elephant"]
long_words = list(filter(lambda word: len(word) > 3, words))
print(long_words)
 Output: ['hello', 'elephant']
```

3.3 Compare lambda functions with regular functions in terms of syntax and use cases.

Lambda functions are typically used for short, simple operations, while regular functions are better suited for more complex logic.

```
# Example 1: Regular function for squaring a number

def square_function(x):

    return x ** 2


# Lambda equivalent

square_lambda = lambda x: x ** 2


# Use

print(square_function(4))  # Output: 16

print(square_lambda(4))    # Output: 16
```

```
# Example 2: Regular function for checking if a number is positive
def is_positive(n):
    return n > 0

# Lambda equivalent
is_positive_lambda = lambda n: n > 0

# Use
print(is_positive(5))          # Output: True
print(is_positive_lambda(5))    # Output: True

# Example 3: Regular function for adding two numbers
def add_function(a, b):
    return a + b

# Lambda equivalent
add_lambda = lambda a, b: a + b

# Use
print(add_function(10, 20))      # Output: 30
print(add_lambda(10, 20))        # Output: 30
```

```
# Example 4: Regular function for filtering even numbers from a list
def filter_even(numbers):
    return [n for n in numbers if n % 2 == 0]

# Lambda with filter equivalent
numbers = [1, 2, 3, 4, 5, 6]
filter_even_lambda = list(filter(lambda x: x % 2 == 0, numbers))

# Use
print(filter_even(numbers))      # Output: [2, 4, 6]
print(filter_even_lambda)        # Output: [2, 4, 6]

# Example 5: Regular function for sorting a list of tuples by the second element
def sort_by_second_element(tuples):
    return sorted(tuples, key=lambda x: x[1])

# Equivalent lambda directly in sorted
tuples = [(1, 2), (3, 1), (5, 4)]
sorted_tuples = sorted(tuples, key=lambda x: x[1])
```

```
# Use
print(sort_by_second_element(tuples))  # Output: [(3, 1), (1, 2), (5, 4)]
print(sorted_tuples)
```

Output: [(3, 1), (1, 2), (5, 4)]

Task 4: NumPy

4.1 Create different types of NumPy arrays (1D, 2D, 3D).

```
import numpy as np

# Example 1: 1D Array
arr_1d = np.array([1, 2, 3, 4, 5])
print("1D Array:", arr_1d)

# Example 2: 2D Array (Matrix)
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
print("2D Array:\n", arr_2d)

# Example 3: 3D Array
arr_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print("3D Array:\n", arr_3d)

# Example 4: Array with zeros
arr_zeros = np.zeros((3, 3))
print("Array with Zeros:\n", arr_zeros)

# Example 5: Array with a range of numbers
arr_range = np.arange(1, 10)
print("Array with Range:\n", arr_range)
```

4.2 Perform basic arithmetic operations on arrays.

```
# Example 1: Adding a scalar to an array
arr = np.array([1, 2, 3, 4])
arr_add = arr + 10
print("Add 10 to each element:", arr_add)

# Example 2: Element-wise addition between two arrays
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr_sum = arr1 + arr2
print("Element-wise addition:", arr_sum)

# Example 3: Element-wise multiplication
arr_mul = arr1 * arr2
print("Element-wise multiplication:", arr_mul)

# Example 4: Array division by a scalar
arr_div = arr1 / 2
print("Array divided by 2:", arr_div)

# Example 5: Matrix multiplication
matrix1 = np.array([[1, 2], [3, 4]])
matrix2 = np.array([[5, 6], [7, 8]])
matrix_mul = np.dot(matrix1, matrix2)
print("Matrix multiplication:\n", matrix_mul)
```

4.3 Use indexing and slicing to access elements.

```
# Example 1: Access a specific element (2D array)
arr_2d = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])
element = arr_2d[1, 2]  # Row 2, Column 3
```

```
print("Access element:", element)

# Example 2: Slice a portion of a 1D array
arr_1d = np.array([10, 20, 30, 40, 50])
slice_1d = arr_1d[1:4]
print("Sliced 1D array:", slice_1d)

# Example 3: Slice a portion of a 2D array
slice_2d = arr_2d[0:2, 1:3]
print("Sliced 2D array:\n", slice_2d)

# Example 4: Reverse a 1D array
reversed_arr = arr_1d[::-1]
print("Reversed 1D array:", reversed_arr)

# Example 5: Use Boolean indexing
bool_index = arr_1d > 30
filtered_arr = arr_1d[bool_index]
print("Filtered array (elements > 30):", filtered_arr)
```

4.4 Explore array manipulation functions (reshape, transpose, concatenate).

```
# Example 1: Reshape a 1D array to a 2D array
arr = np.array([1, 2, 3, 4, 5, 6])
reshaped_arr = arr.reshape(2, 3)
print("Reshaped array:\n", reshaped_arr)

# Example 2: Transpose of a 2D array
arr_2d = np.array([[1, 2], [3, 4], [5, 6]])
transposed_arr = arr_2d.T
print("Transposed array:\n", transposed_arr)

# Example 3: Concatenate two 1D arrays
arr1 = np.array([1, 2, 3])
```

```
arr2 = np.array([4, 5, 6])
concatenated_arr = np.concatenate((arr1, arr2))
print("Concatenated array:", concatenated_arr)

# Example 4: Concatenate along a new axis (stacking)
stacked_arr = np.stack((arr1, arr2))
print("Stacked array:\n", stacked_arr)

# Example 5: Flatten a 2D array to 1D
flattened_arr = arr_2d.flatten()
print("Flattened array:", flattened_arr)
```

4.5 Create and use NumPy random number generators.

```
# Example 1: Generate an array of random numbers (uniform distribution)
random_arr = np.random.rand(3, 3)
print("Random array (uniform distribution):\n", random_arr)

# Example 2: Generate random integers within a specific range
random_ints = np.random.randint(0, 10, size=(2, 3))
print("Random integers:\n", random_ints)

# Example 3: Generate random numbers from a normal distribution
random_normal = np.random.randn(3, 3)
print("Random normal distribution array:\n", random_normal)

# Example 4: Set a random seed for reproducibility
np.random.seed(42)
random_seeded = np.random.rand(3)
print("Random array with seed:\n", random_seeded)

# Example 5: Random choice from an array
```

```
arr = np.array([10, 20, 30, 40, 50])
random_choice = np.random.choice(arr, size=3)
print("Random choice from array:", random_choice)
```

Task 5 : Pandas

5.1 Create Pandas Series and DataFrames.

```
import pandas as pd

# Example 1: Create a Pandas Series from a list
data = [10, 20, 30, 40]
series = pd.Series(data)
print("Pandas Series:\n", series)

# Example 2: Create a Pandas DataFrame from a dictionary
data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35]}
df = pd.DataFrame(data)
print("Pandas DataFrame:\n", df)

# Example 3: Create a DataFrame with a custom index
data = {'Product': ['A', 'B', 'C'], 'Price': [100, 150, 200]}
df_custom_index = pd.DataFrame(data, index=['x1', 'x2', 'x3'])
print("DataFrame with custom index:\n", df_custom_index)

# Example 4: Create a DataFrame from a NumPy array
import numpy as np
data = np.random.rand(3, 3)
df_numpy = pd.DataFrame(data, columns=['A', 'B', 'C'])
print("DataFrame from NumPy array:\n", df_numpy)

# Example 5: Create a Series with a custom index
series_custom_index = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
```

```
print("Series with custom index:\n", series_custom_index)
```

5.2 Load data from various file formats (CSV, Excel, etc.).

```
# Example 1: Load data from a CSV file
df_csv = pd.read_csv('data.csv')
print("Data loaded from CSV:\n", df_csv.head())

# Example 2: Load data from an Excel file
df_excel = pd.read_excel('data.xlsx', sheet_name='Sheet1')
print("Data loaded from Excel:\n", df_excel.head())

# Example 3: Load data from a JSON file
df_json = pd.read_json('data.json')
print("Data loaded from JSON:\n", df_json.head())

import pandas as pd

# Load JSON data into a DataFrame
json_file = "data.json"
df_json = pd.read_json(json_file)

# Display the first 5 rows
df_json.head()



# Example 4: Load data from a URL (CSV format)
url = 'https://people.sc.fsu.edu/~jburkardt/data/csv/airtravel.csv'
df_url = pd.read_csv(url)
print("Data loaded from URL:\n", df_url.head())

# Example 5: Load data from a text file with custom delimiters
df_txt = pd.read_csv('data.txt', delimiter='\t')
```

```
print("Data loaded from text file:\n", df_txt.head())
```

5.3 Perform data cleaning and manipulation tasks.

```
# Example 1: Handling missing values
df = pd.DataFrame({'A': [1, 2, None], 'B': [4, None, 6]})
df_cleaned = df.fillna(0)  # Replace missing values with 0
print("DataFrame with missing values handled:\n", df_cleaned)

# Example 2: Drop missing values
df_dropped = df.dropna()  # Drop rows with missing values
print("Dropped missing values:\n", df_dropped)

# Example 3: Renaming columns
df_renamed = df.rename(columns={'A': 'Column_A', 'B': 'Column_B'})
print("Renamed columns:\n", df_renamed)

# Example 4: Filtering rows based on a condition
df = pd.DataFrame({'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35]})
df_filtered = df[df['Age'] > 30]
print("Filtered DataFrame:\n", df_filtered)

# Example 5: Adding a new column
df['Salary'] = [50000, 60000, 70000]
print("DataFrame with new column:\n", df)
```

5.4 Explore data analysis and visualization using Pandas.

```
import matplotlib.pyplot as plt

# Example 1: Descriptive statistics
```

```
df = pd.DataFrame({'Age': [23, 45, 31, 50, 29], 'Salary': [50000, 70000,
60000, 80000, 45000]})
print("Descriptive statistics:\n", df.describe())

# Example 2: Grouping data and calculating aggregate values
df_grouped = df.groupby('Age').sum()
print("Grouped DataFrame:\n", df_grouped)

# Example 3: Visualizing data with a bar plot
df.plot(kind='bar', x='Age', y='Salary')
plt.title('Age vs Salary')
plt.show()

# Example 4: Visualizing data with a line plot
df.plot(kind='line', x='Age', y='Salary')
plt.title('Age vs Salary Line Plot')
plt.show()

# Example 5: Plotting a histogram
df['Age'].plot(kind='hist', bins=5)
plt.title('Age Distribution')
plt.show()
```

5.5 Create pivot tables and group data for analysis.

```
# Example 1: Creating a pivot table
df = pd.DataFrame({'Product': ['A', 'B', 'A', 'B'], 'Sales': [100, 200, 150, 250],
'Region': ['North', 'South', 'North', 'South']})
pivot_table = pd.pivot_table(df, values='Sales', index='Product',
columns='Region', aggfunc='sum')
print("Pivot Table:\n", pivot_table)

# Example 2: Grouping data by multiple columns and calculating sum
```

```
df_grouped = df.groupby(['Product', 'Region']).sum()
print("Grouped DataFrame:\n", df_grouped)

# Example 3: Grouping data and calculating mean
df_grouped_mean = df.groupby('Product').mean()
print("Grouped by Product (mean):\n", df_grouped_mean)

# Example 4: Creating a pivot table with multiple aggregate functions
pivot_table_multi = pd.pivot_table(df, values='Sales', index='Product',
columns='Region', aggfunc=[sum, len])
print("Pivot Table with multiple aggregate functions:\n", pivot_table_multi)

# Example 5: Grouping data by one column and counting occurrences
df_count = df.groupby('Product').size()
print("Count of occurrences by Product:\n", df_count)
```

Task 6: If Statements

6.1 Demonstrate conditional logic using if, else, and elif statements.

```
        # Example 1: Basic if statement
x = 10
if x > 5:
    print("x is greater than 5")


# Example 2: if-else statement
x = 3
if x > 5:
```

```
    print("x is greater than 5")
else:
    print("x is less than or equal to 5")

# Example 3: if-elif-else statement
x = 7
if x > 10:
    print("x is greater than 10")
elif x > 5:
    print("x is greater than 5 but less than or equal to 10")
else:
    print("x is less than or equal to 5")

# Example 4: if statement with a string condition
name = "Alice"
if name == "Alice":
    print("Hello, Alice!")

#Example 5 :  Login Status Check
user_logged_in = False
admin_logged_in = True
```

```
if user_logged_in:
    print("Welcome, user!")
elif admin_logged_in:
    print("Welcome, admin!")
else:
    print("Please log in.")
```

6.2 Create complex conditional expressions.

```
        # Example 1: Multiple conditions with logical AND
age = 25
income = 40000
if age > 18 and income > 30000:
    print("Eligible for loan")

# Example 2: Multiple conditions with logical OR
x = 5
if x < 0 or x > 10:
    print("x is outside the range 0-10")
else:
    print("x is within the range 0-10")
```

```
# Example 3: Using not operator in condition
is_sunny = False
if not is_sunny:
    print("It is not sunny today")

# Example 4: Combining multiple logical operators
x = 7
if (x > 5 and x < 10) or x == 15:
    print("x is between 5 and 10 or equal to 15")

# Example 5: Complex condition using comparison chaining
y = 15
if 10 < y < 20:
    print("y is between 10 and 20")
```

6.3 Implement nested if statements.

```
        # Example 1: Nested if statement (checking multiple conditions)

x = 20

if x > 10:

    print("x is greater than 10")

    if x > 15:

        print("x is also greater than 15")

    else:

        print("x is less than or equal to 15")
```

```
# Example 2: Nested if-else statement (evaluating within another condition)

age = 25

if age > 18:

    if age >= 21:

        print("You can legally drink alcohol")

    else:

        print("You are an adult but can't drink yet")

else:

    print("You are not an adult")


# Example 3: Nested conditions with multiple logical operators

x = 30

if x > 10:

    print("x is greater than 10")

    if x % 2 == 0:

        print("x is also even")


# Example 4: Nested if within an elif block

num = 50

if num < 30:

    print("num is less than 30")
```

```
elif num >= 30:
    print("num is greater than or equal to 30")
    if num == 50:
        print("num is exactly 50")


# Example 5: Deeply nested if conditions
marks = 85
if marks > 40:
    if marks >= 60:
        if marks >= 75:
            print("You passed with distinction")
        else:
            print("You passed with first class")
    else:
        print("You passed")
else:
    print("You failed")
```

Task 7: Loops

7.1 Use for loops to iterate over sequences.

```
# Example 1: Iterating over a list
```

```
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    print(fruit)

# Example 2: Iterating over a range of numbers
for i in range(5):
    print(i)

# Example 3: Iterating over a string
word = "hello"
for letter in word:
    print(letter)

# Example 4: Iterating over a dictionary
person = {'name': 'Alice', 'age': 25}
for key, value in person.items():
    print(f"{key}: {value}")

# Example 5: Iterating over a list with index
numbers = [10, 20, 30]
for index, number in enumerate(numbers):
```

```
    print(f"Index: {index}, Number: {number}")
```

7.2 Employ while loops for indefinite iteration.

```
        # Example 1: Basic while loop
count = 0
while count < 5:
    print(count)
    count += 1

# Example 2: While loop with a break condition
x = 0
while True:
    print(x)
    x += 1
    if x == 3:
        break

# Example 3: Using a while loop to prompt user input
user_input = ''
while user_input != 'exit':
    user_input = input("Type 'exit' to stop: ")

# Example 4: Counting down with a while loop
n = 5
while n > 0:
    print(n)
    n -= 1

# Example 5: While loop with a conditional check
balance = 100
while balance > 0:
    print(f"Balance: {balance}")
    balance -= 20
```

7.3 Implement nested loops.

```
        # Example 1: Nested for loops
for i in range(3):
    for j in range(2):
        print(f"i = {i}, j = {j}")

# Example 2: Nested loops for multiplication table
for i in range(1, 4):
    for j in range(1, 4):
        print(f"{i} x {j} = {i * j}")

# Example 3: Nested loop with a list of lists
matrix = [[1, 2], [3, 4], [5, 6]]
for row in matrix:
    for element in row:
        print(element)

# Example 4: Nested loop to print a triangle pattern
n = 5
for i in range(1, n + 1):
    for j in range(i):
        print('*', end="")
    print()

# Example 5: Nested loop with if condition inside
for i in range(1, 4):
    for j in range(1, 4):
        if i == j:
            print(f"{i} is equal to {j}")
```

7.4 Utilize break and continue statements.

```
        # Example 1: Break statement in a loop
for i in range(5):
    if i == 3:
        break
    print(i)

# Example 2: Continue statement in a loop
for i in range(5):
    if i == 3:
        continue
    print(i)

# Example 3: Break statement in a while loop
n = 0
while n < 5:
    print(n)
    if n == 2:
        break
```

```
    n += 1


# Example 4: Continue statement in a while loop
n = 0
while n < 5:
    n += 1
    if n == 3:
        continue
    print(n)


# Example 5: Nested loop with break statement
for i in range(5):
    for j in range(5):
        if j == 2:
            break
        print(f"i = {i}, j = {j}")
```

Task 8 : Lists, Tuples, Sets, Dictionaries

8.1 Create and manipulate lists, tuples, sets, and dictionaries.

Here are examples demonstrating how to work with **Lists**, **Tuples**, **Sets**, and **Dictionaries** in Python. The examples cover creation, manipulation, differences, operations like indexing and slicing, and built-in methods for each data structure.

Lists

1. Create and Manipulate Lists

```
# Example 1: Create a list

fruits = ['apple', 'banana', 'cherry']

print(fruits)


# Example 2: Add an element to the list

fruits.append('orange')

print(fruits)


# Example 3: Remove an element from the list

fruits.remove('banana')

print(fruits)
```

```
# Example 4: Indexing and slicing in lists
print(fruits[1])  # Access the second element
print(fruits[0:2])  # Slice first two elements

# Example 5: Insert an element at a specific index
fruits.insert(1, 'mango')
print(fruits)
```

2. Built-in Methods for Lists

```
# Example 1: Sort the list
fruits.sort()
print(fruits)

# Example 2: Reverse the list
fruits.reverse()
print(fruits)

# Example 3: Pop an element (removes the last item
by default)
```

```
popped_item = fruits.pop()
print(popped_item)
print(fruits)

# Example 4: Count occurrences of an element
count = fruits.count('apple')
print(f"Number of 'apple' in the list:", count)

# Example 5: Extend a list with another list
more_fruits = ['pineapple', 'grapes']
fruits.extend(more_fruits)
print(fruits)
```

Tuples

1. Create and Manipulate Tuples

```
# Example 1: Create a tuple
numbers = (10, 20, 30)
print(numbers)
```

```
# Example 2: Access elements in a tuple (indexing)
print(numbers[1])

# Example 3: Slicing a tuple
print(numbers[:2])

# Example 4: Concatenating tuples
new_tuple = numbers + (40, 50)
print(new_tuple)

# Example 5: Unpacking tuples
a, b, c = numbers
print(a, b, c)
```

2. Built-in Methods for Tuples

```
# Example 1: Get the length of a tuple
print(len(numbers))

# Example 2: Count occurrences of an element
print(numbers.count(20))
```

```
# Example 3: Find the index of an element

print(numbers.index(30))


# Example 4: Nested tuple access

nested_tuple = (1, (2, 3), 4)

print(nested_tuple[1][0])


# Example 5: Immutable nature of tuples (can't
change values)

# numbers[0] = 100  # This would throw an error,
since tuples are immutable
```

Sets

1. Create and Manipulate Sets

```
# Example 1: Create a set

my_set = {1, 2, 3, 4}

print(my_set)
```

```
# Example 2: Add an element to the set
my_set.add(5)
print(my_set)

# Example 3: Remove an element from the set
my_set.remove(3)
print(my_set)

# Example 4: Check if an element is in the set
print(2 in my_set)

# Example 5: Set union and intersection
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print("Union:", set1.union(set2))
print("Intersection:", set1.intersection(set2))
```

2. Built-in Methods for Sets

```
# Example 1: Difference between sets
```

```
print(set1.difference(set2))  # Elements in set1 but not in set2


# Example 2: Symmetric difference (elements in either set1 or set2, but not both)

print(set1.symmetric_difference(set2))


# Example 3: Check if one set is a subset of another

print(set1.issubset({1, 2, 3, 4}))


# Example 4: Discard an element (won't raise an error if the element is not found)

my_set.discard(10)  # No error if 10 is not in the set

print(my_set)


# Example 5: Clear all elements in a set

my_set.clear()

print(my_set)  # Output: set()
```

Dictionaries

1. Create and Manipulate Dictionaries

```
# Example 1: Create a dictionary

person = {'name': 'Alice', 'age': 25, 'city': 'New
York'}

print(person)


# Example 2: Access values using keys

print(person['name'])


# Example 3: Add or update a key-value pair

person['job'] = 'Engineer'

print(person)


# Example 4: Remove a key-value pair

del person['age']

print(person)


# Example 5: Check if a key exists in a dictionary
```

```
print('name' in person)
```

2. Built-in Methods for Dictionaries

```
# Example 1: Get all keys in a dictionary

print(person.keys())


# Example 2: Get all values in a dictionary

print(person.values())


# Example 3: Get all key-value pairs as tuples

print(person.items())


# Example 4: Use get method to retrieve a value
(with a default if key doesn't exist)

age = person.get('age', 'Not available')

print(age)


# Example 5: Iterate over dictionary key-value
pairs

for key, value in person.items():
```

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

Differences Between Lists, Tuples, Sets, and Dictionaries

- **Lists**: Ordered, mutable, and allow duplicate elements.
- **Tuples**: Ordered, immutable, and allow duplicate elements.
- **Sets**: Unordered, mutable, and do not allow duplicate elements.
- **Dictionaries**: Key-value pairs, mutable, keys are unique, unordered.

Summary:

- **Lists**: Examples show how to create, manipulate, and use built-in methods such as sorting, appending, and slicing.
- **Tuples**: Demonstrated their immutability, indexing, and unpacking.
- **Sets**: Showcased the uniqueness of elements, set operations like union, intersection, and built-in methods like discard and clear.
- **Dictionaries**: Focused on creating key-value pairs, adding/removing keys, and using built-in methods like `items()`, `get()`, and iteration over key-value pairs

8.2 Understand the differences between these data structures.

In Python, lists, tuples, sets, and dictionaries are fundamental data structures, each designed to store collections of items. However, they differ significantly in terms of behavior, use cases, and efficiency. Let's break down their differences in a theoretical way.

1. Lists

- **Definition: A list is a mutable, ordered collection of items that can hold elements of any type.**
- **Mutability: Lists are mutable, meaning elements can be changed, added, or removed after the list is created.**
- **Ordering: Lists maintain order, meaning the items have a defined sequence, and you can access elements via indexing.**
- **Duplicates: Lists can hold duplicate values.**
- **Use Case: Lists are used when you need an ordered collection of items that can be modified (e.g., for maintaining a to-do list).**

2. Tuples

- **Definition: A tuple is an immutable, ordered collection of items, similar to a list but immutable.**
- **Mutability: Tuples are immutable, meaning that once created, you cannot modify, add, or remove elements.**
- **Ordering: Tuples are also ordered, so elements can be accessed via indexing.**
- **Duplicates: Like lists, tuples allow duplicate values.**
- **Use Case: Tuples are used when you need a collection of items that should not be changed (e.g., representing geographic coordinates or data that is constant).**

3. Sets

- **Definition: A set is an unordered collection of unique items. Sets do not allow duplicates.**

- **Mutability: Sets are mutable, meaning that you can add or remove elements, but they do not support indexing or slicing since they are unordered.**
- **Ordering: Sets are unordered, meaning there is no guarantee that elements will maintain any specific order.**
- **Duplicates: Sets do not allow duplicates. Every element in a set is unique.**
- **Use Case: Sets are used when you need to maintain a collection of unique elements or perform set operations like union, intersection, and difference (e.g., finding common items between two sets of data).**

4. Dictionaries

- **Definition: A dictionary is a collection of key-value pairs. Each key must be unique, but values can be duplicated.**
- **Mutability: Dictionaries are mutable, meaning you can add, modify, or remove key-value pairs.**
- **Ordering: In Python 3.7+, dictionaries are ordered by insertion order, but in earlier versions, they were unordered.**
- **Duplicates: Keys in a dictionary must be unique, but values can be duplicated.**
- **Use Case: Dictionaries are used when you need a mapping between unique keys and values, such as storing user information with unique IDs or using key-value pairs for fast lookups (e.g., a phone book).**

Lists: Ordered and mutable. You can modify elements, add or remove items, and they allow duplicates.

- **Tuples: Ordered but immutable. Once created, they cannot be changed, making them more efficient for read-only operations.**
- **Sets: Unordered and mutable, but only store unique items. They are useful for set operations like union, intersection, and difference.**

- **Dictionaries: Store key-value pairs. The keys must be unique, and values are accessed by the keys.**

8.3 Perform operations like indexing, slicing, adding, removing Elements.

Lists

```
# Indexing and Slicing
my_list = ['a', 'b', 'c', 'd']
print(my_list[1])  # Output: 'b' (indexing)
print(my_list[1:3])  # Output: ['b', 'c'] (slicing)

# Adding elements
my_list.append('e')  # Adds 'e' to the end
my_list.insert(2, 'z')  # Inserts 'z' at index 2
print(my_list)  # ['a', 'b', 'z', 'c', 'd', 'e']

# Removing elements
my_list.remove('b')  # Removes 'b'
my_list.pop(2)  # Removes element at index 2 ('z')
print(my_list)  # ['a', 'c', 'd', 'e']
```

Tuples

```
# Indexing and Slicing
my_tuple = ('a', 'b', 'c', 'd')
print(my_tuple[1])  # Output: 'b' (indexing)
print(my_tuple[1:3])  # Output: ('b', 'c') (slicing)

# Tuples are immutable, so you cannot add or remove elements directly
# If you need to modify a tuple, you can convert it to a list first:
temp_list = list(my_tuple)
temp_list.append('e')
my_tuple = tuple(temp_list)
print(my_tuple)  # Output: ('a', 'b', 'c', 'd', 'e')
```

Sets

```
# Adding elements
my_set = {1, 2, 3}
my_set.add(4)  # Adds 4 to the set
print(my_set)  # Output: {1, 2, 3, 4}

# Removing elements
my_set.remove(2)  # Removes 2 from the set
print(my_set)  # Output: {1, 3, 4}

# No indexing or slicing since sets are unordered
```

Dictionaries

```
# Adding and Accessing elements
my_dict = {'name': 'Alice', 'age': 25}
my_dict['city'] = 'New York'  # Adding a new key-value pair
print(my_dict['name'])  # Accessing the value by key 'name' -> Output:
'Alice'

# Removing elements
del my_dict['age']  # Removes the key 'age'
print(my_dict)  # Output: {'name': 'Alice', 'city': 'New York'}

# No indexing or slicing since dictionaries use keys for access
```

8.4 Explore built-in methods for each data structure.

Lists

```
my_list = [1, 2, 3, 4]

# append(): Adds an element to the end of the list
my_list.append(5)
print(my_list)  # Output: [1, 2, 3, 4, 5]
```

```
# extend(): Extend the list by appending elements from another list
my_list.extend([6, 7])
print(my_list)  # Output: [1, 2, 3, 4, 5, 6, 7]

# pop(): Removes and returns the last element (or the element at the
specified index)
removed_element = my_list.pop()
print(removed_element)  # Output: 7
print(my_list)  # Output: [1, 2, 3, 4, 5, 6]

# sort(): Sorts the list in ascending order
my_list.sort()
print(my_list)  # Output: [1, 2, 3, 4, 5, 6]

# reverse(): Reverses the order of the list
my_list.reverse()
print(my_list)  # Output: [6, 5, 4, 3, 2, 1]
```

Tuples

```
my_tuple = (1, 2, 3, 2, 4)

# count(): Counts occurrences of an element
print(my_tuple.count(2))  # Output: 2

# index(): Returns the index of the first occurrence of an element
print(my_tuple.index(3))  # Output: 2

# Tuples have fewer methods since they are immutable, unlike lists
```

Sets

```
my_set = {1, 2, 3, 4}

# add(): Adds an element to the set
my_set.add(5)
print(my_set)  # Output: {1, 2, 3, 4, 5}
```

```
# remove(): Removes an element from the set (raises an error if not
found)
my_set.remove(2)
print(my_set)  # Output: {1, 3, 4, 5}

# union(): Returns the union of two sets
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print(set1.union(set2))  # Output: {1, 2, 3, 4, 5}

# intersection(): Returns the intersection of two sets
print(set1.intersection(set2))  # Output: {3}

# difference(): Returns the difference between two sets
print(set1.difference(set2))  # Output: {1, 2}
```

Dictionaries

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

# keys(): Returns all keys in the dictionary
print(my_dict.keys())  # Output: dict_keys(['name', 'age'])

# values(): Returns all values in the dictionary
print(my_dict.values())  # Output: dict_values(['Alice', 25])

# items(): Returns all key-value pairs in the dictionary
print(my_dict.items())  # Output: dict_items([('name', 'Alice'),
('age', 25)])

# get(): Returns the value for a key (returns None if key is not
found)
print(my_dict.get('name'))  # Output: 'Alice'
print(my_dict.get('city', 'Not Found'))  # Output: 'Not Found'

# update(): Updates the dictionary with another dictionary or
key-value pairs
my_dict.update({'city': 'New York'})
```

```
print(my_dict)  # Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

Task 9 : Operators

9.1 Use Arithmetic, Comparison, Logical, and Assignment Operators

Arithmetic Operators

These are used to perform basic mathematical operations.

Operator	Description	Example	Result
+	Addition	5 + 3	8
-	Subtraction	5 - 3	2
*	Multiplication	5 * 3	15
/	Division	5 / 2	2.5
//	Floor Division	5 // 2	2
%	Modulus (remainder)	5 % 2	1

```
**        Exponentiation  5 ** 2    25
```

```
a = 10
b = 3
print("Addition:", a + b)     # 13
print("Subtraction:", a - b)  # 7
print("Multiplication:", a * b)  # 30
print("Division:", a / b)     # 3.333...
print("Floor Division:", a // b) # 3
print("Modulus:", a % b)      # 1
print("Exponent:", a ** b)    # 1000
```

Comparison Operators

These compare two values and return True or False.

```
Operator  Description     Example  Result
==        Equal to        5 == 3   False
!=        Not equal to    5 != 3   True
>         Greater than    5 > 3    True
```

<	Less than	5 < 3	False
>=	Greater than or equal to	5 >= 3	True
<=	Less than or equal to	5 <= 3	False

```
print("Equal:", a == b)       # False
print("Not Equal:", a != b)   # True
print("Greater than:", a > b) # True
print("Less than:", a < b)    # False
print("Greater or equal:", a >= b) # True
print("Less or equal:", a <= b)   # False
```

Logical Operators

These are used to combine conditional statements.

Operator	Description	Example	Result
and	True if both conditions are true	(5 > 3) and (5 < 10)	True

or	True if at least one condition is true	(5 > 3) or (5 > 10)	True
not	Reverses the result	not(5 > 3)	False

```
x = True
y = False
print("x and y:", x and y)  # False
print("x or y:", x or y)  # True
print("not x:", not x)    # False
```

Assignment Operators

These are used to assign values to variables.

Operator	Description	Example	Result
=	Assignment	a = 5	a = 5
+=	Add and assign	a += 3	a = 8

-=	Subtract and assign	a -= 3	a = 5
*=	Multiply and assign	a *= 3	a = 15
/=	Divide and assign	a /= 3	a = 5.0
//=	Floor divide and assign	a //= 3	a = 5
%=	Modulus and assign	a %= 3	a = 2
**=	Exponent and assign	a **= 3	a = 1000

```
a = 5

a += 3  # equivalent to a = a + 3

print("After +=:", a)  # 8
```

```
a *= 2  # equivalent to a = a * 2

print("After *=:", a)  # 16
```

9.2 Understand Operator Precedence

Operator precedence defines the order in which operations are evaluated in an expression. Higher precedence operators are evaluated first. Here's the precedence of common operators from highest to lowest:

1. Exponentiation (**)
2. Unary plus/minus (+x, -x)
3. Multiplication, Division, Modulus, Floor Division (*, /, %, //)
4. Addition, Subtraction (+, -)
5. Comparison Operators (<, >, <=, >=, ==, !=)
6. Logical NOT (not)
7. Logical AND (and)
8. Logical OR (or)

Examples of Operator Precedence:

```
# Example 1: Exponentiation has higher precedence than multiplication

result = 2 ** 3 * 4

print(result)  # Output: 32 (2^3 = 8, then 8 * 4 = 32)


# Example 2: Parentheses have the highest precedence
```

```
result = (2 + 3) * 4

print(result)  # Output: 20 (2 + 3 = 5, then 5 * 4 = 20)


# Example 3: Logical operators and comparison

result = (5 > 3) and (2 < 4) or not (3 == 3)

print(result)  # Output: True (because not (3 == 3) is False, and (5 > 3 and 2 < 4) is True)

Using Parentheses to Override Precedence:

You can use parentheses to explicitly specify the order of operations, overriding the default precedence.

result = 2 + 3 * 4  # Without parentheses: 2 + (3 * 4) = 14

print(result)  # 14


result = (2 + 3) * 4  # With parentheses: (2 + 3) * 4 = 20

print(result)  # 20
```

9.3 Apply Operators in Expressions and Calculations

You can use operators in various expressions and calculations, often combining different types of operators within the same expression.

```
# Example 1: Combining arithmetic and assignment operators

x = 10

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

x *= 2  # equivalent to x = x * 2

print(x)  # Output: 30


# Example 2: Applying logical and comparison operators

age = 20

income = 50000


is_eligible = (age > 18) and (income > 30000)

print(is_eligible)  # Output: True


    # Example 3:Modulo and Exponentiation Operators

x = 7
```

```
y = 2

# Modulo (Remainder)
result_mod = x % y
# Exponentiation
result_exp = x ** y
print("Modulo:", result_mod)
print("Exponentiation:", result_exp)
```

Output:

```
makefile
Copy code
Modulo: 1
Exponentiation: 49
```

 # Example 4:Comparison Operators

```
a = 15
b = 10
```

```
# Greater than
print(a > b)
# Less than
print(a < b)
# Equal to
print(a == b)
# Not equal to
print(a != b)
```

 # Example 5: Logical Operators

```
x = True
y = False

# AND operator
result_and = x and y
# OR operator
result_or = x or y
# NOT operator
```

```
result_not = not x


print("AND:", result_and)

print("OR:", result_or)

print("NOT:", result_not)
```

Task 10 : Reading CSV files

10.1 Read CSV Files into Pandas DataFrames

```
import pandas as pd

# Example 1: Basic CSV reading
df1 = pd.read_csv('data1.csv')
print(df1.head())

# Example 2: Reading a CSV file with specific column names
df2 = pd.read_csv('data2.csv', names=['A', 'B', 'C'])
print(df2.head())

# Example 3: Reading a CSV from a URL
url = 'https://people.sc.fsu.edu/~jburkardt/data/csv/hw_200.csv'
df3 = pd.read_csv(url)
print(df3.head())

# Example 4: Reading a CSV with index column
df4 = pd.read_csv('data3.csv', index_col=0)  # Setting the first
column as the index
print(df4.head())

# Example 5: Reading a CSV with specific data types
df5 = pd.read_csv('data4.csv', dtype={'A': int, 'B': float})
```

```
print(df5.head())
```

10.2 Explore Different CSV Reading Options and Parameters

```
# Example 1: Reading a CSV with a different delimiter
(semicolon-separated)
df1 = pd.read_csv('data5.csv', delimiter=';')
print(df1.head())

# Example 2: Skipping a specific number of rows
df2 = pd.read_csv('data6.csv', skiprows=2)  # Skip first 2 rows
print(df2.head())

# Example 3: Reading only specific columns
df3 = pd.read_csv('data7.csv', usecols=['A', 'C'])
print(df3.head())

# Example 4: Reading CSV with custom NA values
df4 = pd.read_csv('data8.csv', na_values=['N/A', 'missing', '-'])
print(df4.head())

# Example 5: Reading a large CSV file in chunks
chunksize = 100
for chunk in pd.read_csv('data9.csv', chunksize=chunksize):
    print(chunk.head())
```

10.3 Handle Missing Values and Data Cleaning

```
# Example 1: Checking for missing values
df1 = pd.read_csv('data10.csv')
print(df1.isnull().sum())  # Checking how many missing values each
column has

# Example 2: Filling missing values with a specific value
df2 = df1.fillna(0)  # Fill missing values with 0
print(df2.head())
```

```
# Example 3: Dropping rows with missing values
df3 = df1.dropna()  # Drop rows with any missing values
print(df3.head())

# Example 4: Replacing missing values with the mean of a column
df4 = df1.copy()
df4['B'] = df4['B'].fillna(df4['B'].mean())
print(df4.head())

# Example 5: Removing duplicate rows
df5 = pd.read_csv('data11.csv')
df5_cleaned = df5.drop_duplicates()
print(df5_cleaned.head())
```

Task 11: Python String Methods

11.1 Manipulate Strings Using Various Built-in Methods

```
# Example 1: Replace a substring in a string
text = "Hello, World!"
new_text = text.replace("World", "Python")
print(new_text)  # Output: Hello, Python!

# Example 2: Join a list of strings into a single string
words = ['Python', 'is', 'awesome']
sentence = ' '.join(words)
print(sentence)  # Output: Python is awesome

# Example 3: Counting occurrences of a substring
text = "banana"
count = text.count('a')
print(count)  # Output: 3

# Example 4: Checking if a string starts with a specific substring
print(text.startswith('ban'))  # Output: True
```

```
# Example 5: Finding the position of a substring
position = text.find('ana')
print(position)  # Output: 1
```

11.2 Perform Operations Like Concatenation, Slicing, and Finding Substrings

```
# Example 1: Concatenate two strings
str1 = "Hello"
str2 = "World"
result = str1 + " " + str2
print(result)  # Output: Hello World

# Example 2: Slice a string
text = "Python programming"
sliced_text = text[0:6]  # Extract 'Python'
print(sliced_text)  # Output: Python

# Example 3: Find if a substring exists
print("programming" in text)  # Output: True

# Example 4: Get a substring from the end
last_word = text[-11:]
print(last_word)  # Output: programming

# Example 5: Extract every second character from a string
every_second_char = text[::2]
print(every_second_char)  # Output: Pto rgamn
```

11.3 Convert Strings to Uppercase, Lowercase, and Title Case

```
# Example 1: Convert to uppercase
text = "hello world"
uppercase_text = text.upper()
print(uppercase_text)  # Output: HELLO WORLD
```

```
# Example 2: Convert to lowercase
lowercase_text = text.lower()
print(lowercase_text)  # Output: hello world

# Example 3: Convert to title case
title_text = text.title()
print(title_text)  # Output: Hello World

# Example 4: Swap case of a string (convert uppercase to lowercase and
vice versa)
swapped_case = text.swapcase()
print(swapped_case)  # Output: HELLO WORLD

# Example 5: Capitalize only the first letter of the string
capitalized_text = text.capitalize()
print(capitalized_text)  # Output: Hello world
```

11.4 Remove Whitespace and Split Strings

```
# Example 1: Remove leading and trailing whitespace
text = "   Hello, World!   "
trimmed_text = text.strip()
print(trimmed_text)  # Output: Hello, World!

# Example 2: Remove only leading whitespace
leading_trimmed_text = text.lstrip()
print(leading_trimmed_text)  # Output: "Hello, World!   "

# Example 3: Remove only trailing whitespace
trailing_trimmed_text = text.rstrip()
print(trailing_trimmed_text)  # Output: "   Hello, World!"

# Example 4: Split a string into a list by spaces
text = "Python is awesome"
split_text = text.split()
print(split_text)  # Output: ['Python', 'is', 'awesome']
```

```
# Example 5: Split a string using a specific delimiter
csv_text = "apple,banana,cherry"
split_csv = csv_text.split(',')
print(split_csv)  # Output: ['apple', 'banana', 'cherry']
```