5CS037 - Concepts and Technologies of AI. Worksheet - 0: Python Essentials for Machine Learning.

Prepared By: Siman Giri {Module Leader - 5CS037}

October 28, 2025

———————— Welcome - to - 5CS037. ————————-

1 Instructions

This is a Pre-requisite homework assignment to be completed on your own before your first workshop and is compulsory to submit.
{**Cautions!!!:**Failure to Submit this assignment might affect your future grades and ability to receive highest grades}.
Please answer the questions below using python in the Jupyter Notebook and follow the guidelines below:

- This worksheet must be completed individually.
- All the solutions must be written in Jupyter Notebook.
- You are allowed to use basic packages like time, collection etc. but do not use the packages to solve the problem directly.

2 Getting Started with Python.

This is NOT a problem, but you are highly recommended to run the following code with some of the input changed in order to understand the meaning of the operations.

- **Cautions!!!:**
  - This Guide doesnot contain sample output, as we expect you to re-write the code and observe the output.
  - If found: any error or bugs, please report to your instructor and Module leader.

  {Will hugely appreciate your effort.}

2.1 About a Python:

Python is a high-level, general-purpose programming language created by Guido van Rossum, first released in 1991. Its design focuses on making code easy to read and understand, using clear formatting and meaningful whitespace. Python's structure and support for objectoriented programming help programmers write organized, logical code adaptable for both small-scale and large-scale projects. Known for its flexibility and powerful libraries, Python has become a popular language for machine learning research and is the main language used in frameworks like Numpy, Pandas, Matplotlib, sickit learn and many more.

Python Version Check

```
import sys
# Check Python version if sys.version_info.major == 3 and sys.version_info.minor
>= 6:
    print("Hello, World!") print(f"Python version:
    {sys.version}")
else:
    print("Please use Python 3.6 or higher.")
```

3 Data Types in Python.

Mutable Data Types: Mutable objects can be changed after they are created. This means you can modify their contents, such as adding or removing items, or changing their values without creating a new object.
Immutable Data Types: Immutable objects cannot be changed once they are created. Any operation that seems to modify an immutable object will actually create a new object instead of changing the original.

3.1 Some Common Data Types in Python:

Numeric:

These data types are used to represent numerical values.

- int: Represents integer values (e.g., 10, -3).
- float: Represents floating-point (decimal) values (e.g., 10.5, -2.7).
- complex: Represents complex numbers (e.g., 3 + 4j).

Data Types - Numeric

```
age = 23 #int pi = 3.14 #float temperature = -5.5 #float print("data type of variable
age = ", type(age)) print("data type of variable pi = ", type(pi)) print("data type of
variable temperature = ", type(temperature))
```

Sequence:

These data types are ordered collections of items. You can access elements by their position (index).

- str: string (str) represents sequence of characters enclosed by double quotes or single quotes. (e.g., “Hello, World!”).It is an immutable sequence.

Data Types - Sequence - String

```
name = "Alice" greeting = 'Hello' address = "123 Main St" print("data type of
the variable name = ",type(name)) print("data type of the variable greeting =
",type(greeting)) print("data type of the variable address = ",type(address))
# slice only one element print("The first letter of the name is:",
name[0]) print("The last letter of the name is:", name[-1])
# slice a range of elements print("The second letter to the fourth of the name is:",
name[1:4]) print("The first two letters of the name are:", name[:2]) print("Substring
starting from the third letter is:", name[2:])
```

- list: Represents lists, which can contain mixed data types and are mutable (e.g., ["apple", "banana"]).

Data Types - Sequence - List

```
list1 = [1, 2, 3, 4] mixed_list = [12,
"Hello", True]
# List is mutable mixed_list[0] = False
mixed_list
```

- tuple: Represents tuples, which are ordered and immutable collections (e.g., (1, 2, 3)).

Data Types - Sequence - Tuple

```
colors = ('red', 'green', 'yellow', 'blue') print("First element:",
colors[0]) print("Last two elements:", colors[2:]) print("Middle two
elements:", colors[1:3]) colors[0] = 'purple' colors # will generate an
error as tuple is immutable.
```

Mapping:

This category includes data types that store key-value pairs, allowing for efficient retrieval based on keys.

- dict: Represents dictionaries, which can store various data types as values associated with unique keys (e.g., "name": "Alice", "age": 25).

Data Types - Sequence - Dict

```
person = {'name':'John','age':30,'city':'Pittsburgh'}
print(f"Hello my name is {person['name']}. I am {person['age']} years old and
       I live at {person['city']}.") print("All keys:",
list(person.keys())) print("All values:",
list(person.values()))
```

Set:

Sets are unordered collections of unique elements. They are useful for membership testing and eliminating duplicate entries.

- set: A mutable collection of unique items (e.g., 1, 2, 3).
- frozenset: An immutable version of a set (e.g., frozenset([1, 2, 3])).

Data Types - Set

```
unique_numbers = {1,2,3,3,3,3,4,5} print(unique_numbers)
```

Boolean:

This category contains types that represent truth values.

- bool: Represents boolean values (True or False).

Data Types - Boolean

```
is_student = True has_license = False print("data type of the variable is_student =
",type(is_student)) print("data type of the variable has_license = ",type(has_license))
```

Special:

This category is used for unique data types that do not fit into the other categories.

- NoneType: Represents the absence of a value or a null value (e.g., None).

Data Types - Boolean

```
result = None
```

Data - Types - Summary:

Data Type	**Category**	**Sample Code**
list	Sequence	fruits = ["apple", "banana", "cherry"]
dict	Mapping	person = {"name": "Bob", "age": 25}
set	Set	unique values = {1, 2, 3}

bytearray	Sequence	data = bytearray(b"hello")

Table 1: Mutable Data Types in Python

Data Type	**Category**	**Sample Code**
int	Numeric	x = 10
float	Numeric	y = 10.5
complex	Numeric	z = 3 + 4j
str	Sequence	name = "Alice"
tuple	Sequence	coordinates = (10, 20)
frozenset	Set	frozen values = frozenset([1, 2, 3])
bool	Boolean	is active = True
NoneType	Special	value = None

Table 2: Immutable Data Types in Python

4 Logical Statements and Loops.

Python code can be decomposed into packages, modules, statements, and expressions, as follows:

1. Expressions create and process objects: • Expressions are part of statements that return a value, such as variables, operators, or function calls.

2. Statements contain expressions:

   • Statements are sections of code that perform an action. The main groups of Python statements are: assignment statements, print statements, conditional statements (if, break, continue, try), and looping statements (for, while).

3. Module Contain statements:

   • Modules are Python files that contain Python statements, and are also called scripts.

4. Packages are composed of modules: • Packages are Python programs that collect related modules together within a single directory hierarchy.

4.1 Logical Statements:

Logical statements are used to perform conditional operations. The primary logical statements in Python are:

- if: Executes a block of code if the condition is true.

```
if condition:
    # Code to execute if condition is true
```

- elif: Short for ”else if,” allows for multiple conditions to be checked sequentially.

```
if condition1:
    # Code for condition1 elif
condition2:
    # Code for condition2
```

- else: Executes a block of code if none of the preceding conditions are true.

```
if condition:
    # Code if condition is true else:
    # Code if condition is false
```

Sample Code - if-else statement

```
num = 10 if
num > 0:
    print("Positive") elif num
== 0:
    print("Zero") else:
    print("Non-positive")
```

- Comparison operators: Used to compare values.
  - ==: Equal to
  - !=: Not equal to
  - <: Less than
  - >: Greater than
  - <=: Less than or equal to
  - >=: Greater than or equal to

Sample Code - Comparision Operator

```
x = 5 y = 10 if x > 0 and y
< 20:
    print("Both conditions are true") if x > 0 or y
> 20:
    print("At least one condition is true")
if not x == 0:
    print("x is not equal to 0")
```

- Logical operators: Used to combine multiple conditions.
  - and: True if both conditions are true.
  - or: True if at least one condition is true.
  - not: Inverts the truth value of the condition.

Sample Code - Logical Operator

```
# Define two boolean variables a =
True b = False
# Using the 'and' operator if a and b:
print("Both a and b are True")
else:
    print("Either a or b is False") # This will be printed
# Using the 'or' operator if a or b:
    print("At least one of a or b is True") # This will be printed else:
    print("Both a and b are False") # Using the 'not' operator if
not a: print("a is False") # This will not be printed else:
    print("a is True") # This will be printed
```

4.2 Loops:

Loops are used to execute a block of code multiple times. The primary loop types in Python are:

- for loop: Iterates over a sequence (like a list, tuple, or string).

```
for item in iterable:
    # Code to execute for each item
```

- while loop: Repeats as long as a condition is true.

```
while condition:
    # Code to execute while condition is true
```

- break: Exits the loop immediately.

```
for item in iterable:
        if some_condition: break #
            Exit loop
```

- continue: Skips the current iteration and continues with the next iteration of the loop.

```
    for item in iterable:
        if some_condition: continue # Skip to the next
            iteration
```

Sample Code - Various Loops

```
# for loop:
fruits = ["apple", "banana","cherry"] for fruit in
fruits:
    print(fruit) # While loop: count = 0 while
count < 5: print("Count is:", count) count += 1 #
Break fruits = ["apple", "banana", "orange"]
print("loop 1") for fruit in fruits: print(fruit) if
fruit == "apple":
        break #
Continiue print("loop 2")
for fruit in fruits:
    if fruit == "apple":
        continue else:
        print(fruit)
```

5 Functions.

Functions are fundamental building blocks in Python, similar to mathematical functions that define relationships between inputs and outputs. In mathematics, a function such as

$$z = f(x,y)$$

maps inputs x and y to an output z.

However, functions in programming languages are more generalized and versatile; they can process various data types and perform a wide range of operations on inputs beyond simple mathematical relationships.

A function in programming is a self-contained block of code that encapsulates a specific task or a set of related tasks. It defines how inputs relate to outputs through the operations performed within the function. When a function is called, input arguments are provided, the program executes the defined code, and then returns the function's output. This structure enables functions to manage complex processes in a structured and reusable way. Most programming languages offer support for pre-built and user-defined functions to promote following:

1. **Code Abstraction and Re-usability:** A key principles in software development. Rather than repeating the same code across multiple locations in an application, we can define a single function that performs the desired task and call it wherever needed. This approach not only reduces redundancy but also simplifies maintenance; if a change is required, we modify it in one place—the function itself—rather than updating every instance where the code appears.

2. **Code Modularity:** By breaking down complex processes into smaller, self-contained functions that focus on specific tasks, we create modular code that is both organized and easier to understand. This modular structure enhances readability and maintainability, making it simpler to update or expand the program without disrupting its overall structure.

5.1 Anatomy of a Function:

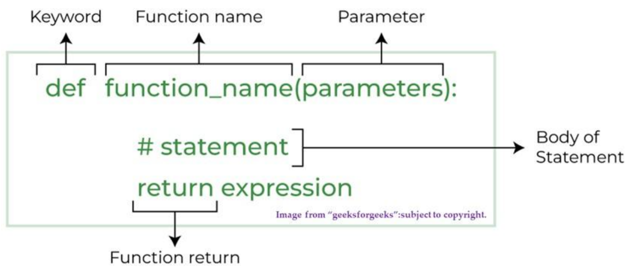


Figure 1: Anatomy of a Function

Here's a brief overview of the main components of a function:

- **Function Name:** This is the identifier by which the function is called. It should be descriptive, conveying the function's purpose or the task it performs. In Python, function names typically follow snake case formatting (e.g., calculate_area).

- **Input Parameters:** These are variables listed within the parentheses after the function name in the definition. Parameters allow functions to accept inputs, which are used within the function's body to perform calculations or operations. Parameters make functions more flexible and reusable, as they allow you to pass different values each time you call the function.

- **Docstring:** A docstring is an optional, multi-line comment placed just after the function definition. It provides documentation on the function's purpose, parameters, and output, helping others understand what the function does. In Python, a docstring is written within triple quotes (""" """) and can be accessed using the help() function.

- **Return Output:** The return statement specifies the output of the function, allowing it to send back a result to the part of the program where it was called. This enables further processing or storage of the function's output. If no return statement is specified, the function will return None by default.

How to write a Function Correctly? - Docstring are Most.

Correct way to Write Function

```
        def add_binary(a, b):
'''

Returns the sum of two decimal numbers in binary digits.

Parameters:
        a (int): A decimal integer b (int): Another decimal integer Returns:
binary_sum (str): Binary string of the sum of a and b
'''
binary_sum = bin(a+b)[2:] return
binary_sum
```

5.2 Built - in - Functions:

Python provides a wide range of built-in functions that are ready to use. These functions perform common tasks and are available without the need to import additional modules.Presented below is an example showcase; for more details, refer to the Python Reference on W3Schools

Example on Built - in - Function

```
print("Hello, World!") # Output: Hello, World! length =
len("Hello") # Output: 5 data_type = type(42) # Output:
<class 'int'> total = sum([1, 2, 3]) # Output: 6
```

5.3 Built - in - Methods:

Methods are similar to functions but are associated with objects. They act specifically on data types (e.g., strings, lists, dictionaries and are called using dot notation object.method(). Presented below is an example showcase; for more details, refer to the Python Reference on W3Schools

Example on Built - in - Methods

```
# str.upper(): Converts all characters in a string to uppercase. message = "hello".upper() #
Output: "HELLO"
# list.append(): Adds an element to the end of a list.
fruits = ["apple", "banana"] fruits.append("cherry") # Output: ["apple", "banana",
"cherry"] # dict.get(): Returns the value associated with a key in a dictionary.
info = {"name": "Alice", "age": 25} age =
info.get("age") # Output: 25
```

5.4 User - Defined - Function:

User-defined functions are created by the user to perform specific tasks. These functions are defined using the def keyword and may include parameters and return values.
Here is an example function python that converts Celsius to Fahrenheit and vice versa.

Example of well-structured user-defined function

```
def temperature_converter():
"""
Converts temperature between Celsius and Fahrenheit.
This function prompts the user to specify the conversion type (Celsius to Fahrenheit or Fahrenheit to Celsius)
and then asks for the temperature value. It calculates and returns the converted temperature.
Returns: float: The converted temperature value.
""" print("Choose conversion type:") print("1.
Celsius to Fahrenheit") print("2. Fahrenheit to
Celsius") # Get conversion choice from user
choice = input("Enter 1 or 2: ") if choice == "1":
    # Celsius to Fahrenheit conversion
    celsius = float(input("Enter temperature in Celsius: ")) fahrenheit = (celsius *
    9/5) + 32 print(f"{celsius}C is equal to {fahrenheit}F") return fahrenheit
elif choice == "2":
    # Fahrenheit to Celsius conversion
    fahrenheit = float(input("Enter temperature in Fahrenheit: ")) celsius = (fahrenheit - 32) * 5/9
    print(f"{fahrenheit}F is equal to {celsius}C") return celsius







    else:
        print("Invalid choice. Please enter 1 or 2.") return None #
Call the function temperature_converter()
```

5.5 Global and Local Variables:

1. Global Variables:

A **global variable** is defined outside of any function and is accessible from any part of the program, including within functions. Global variables maintain their values throughout the program's execution and can be accessed or modified by any function unless explicitly declared as nonlocal or redefined within the function.

Global Varaible

```
x = 10 # Global Variable.
def print_global():
print(x) # Accessing global variable
print_global() # Output: 10
```

In this example, the variable x is global and can be accessed inside the print _global() function.

2. Local Variables:

A **local variable** is defined within a function and can only be accessed within that function. It exists only during the function's execution, and once the function completes, the local variable is removed from memory.

Global Varaible

```
def print_local():
    y = 5 # Local variable
    print(y)

print_local() # Output: 5 print(y) # This would cause an error because y is not accessible outside the function
```

Here, y is a local variable within print local() and cannot be accessed outside the function.

3. Accessing and Modifying Global Variables Inside a Function

To modify a global variable within a function, we must use the global keyword. Otherwise, assigning a new value to a global variable within a function will create a new local variable with the same name, leaving the global variable unchanged.

Global Varaible

```
x = 10 # Global variable def
modify_global():
    global x # Declaring x as global x = 20 #
    Modifying global x
modify_global() print(x) #
Output: 20
```

Using global here allows the function to modify the global variable x directly.

6 Exception and Error Handling.

In programming, errors can interrupt the normal flow of execution, potentially causing the program to crash. **Exception and Error handling** is a method to detect and respond to such errors gracefully, ensuring the program can handle unexpected situations without failing.

1. Types of Error:

Errors in Python can be broadly classified into:

- **Syntax Errors**: Occur when the Python parser finds code that violates Python's syntax rules. These are usually detected before the program runs.

```
# Example of Syntax Error
```

```
print("Hello World) # Missing closing quote
```

- **Exceptions**: Occur at runtime and indicate unexpected conditions that disrupt program execution. Examples include division by zero, accessing an undefined variable, or trying to open a file that does not exist.

```
# Example of an Exception
result = 10 / 0 # Raises ZeroDivisionError
```

2. Handling Exceptions with try-except

To handle exceptions, Python uses the try-except structure. Code that may raise an exception is placed within the try block, and if an exception occurs, it is caught and handled in the except block.

- Example:

```
try:
    num = int(input("Enter a number: ")) print("Result:", 10 / num)
except ZeroDivisionError:
    print("Cannot divide by zero.")
except ValueError: print("Invalid input. Please enter a
    number.")
```

In this example, ZeroDivisionError and ValueError are handled specifically, ensuring that the program continues even if the user inputs invalid data.

3. Using finally Block

The finally block is optional and executes regardless of whether an exception occurs or not. It is typically used for cleanup tasks, like closing files or releasing resources.

- Example:

```
try:
    file = open("example.txt", "r") data =
    file.read()
except FileNotFoundError:
    print("File not found.")
finally: file.close() # Ensures the file is closed
```

The finally block here ensures that the file is closed, even if an exception occurs.

4. Raising Custom Exceptions

Python allows the creation of custom exceptions using the raise statement. This is useful for handling specific errors not covered by built-in exceptions.

- Example:

```
def check_age(age): if age
      < 0:
                  raise ValueError("Age cannot be negative.")
      else: print("Valid age") try:
      check_age(-1)
except ValueError as e:
      print(e)
```

This example raises a ValueError if an invalid age is provided.

6.1 To - Summarize:

- **Syntax Errors:** occur due to incorrect code structure and are identified before execution.
- **Exceptions:** occur at runtime and can be handled using try-except blocks.
- The finally block ensures that essential cleanup code is run, regardless of exceptions.
- Custom exceptions can be created with raise to handle specific scenarios.

Understanding and implementing exception handling is essential for writing resilient code that can gracefully handle unexpected situations.

7 TO - DO - Task

Please complete all the problem listed below.

7.1 Warming Up Exercise:

In this exercise, you'll work with daily time allocation data recorded by a group of students. Each student tracked the number of hours spent studying, watching entertainment, and sleeping for 15 consecutive days.

- **Dataset:** Each record represents one day's data in the format:

(study _hours,entertainment hours,sleep hours) • A

sample dataset is provided below.

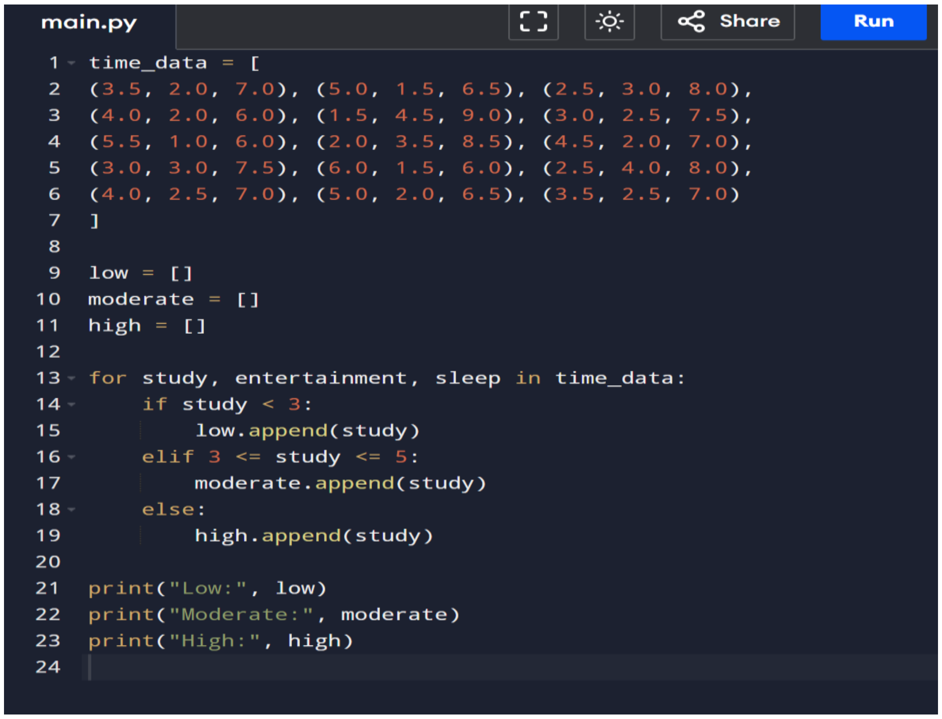
Daily Student Productivity Data

```
# Daily time (in hours): [study, entertainment, sleep] time_data = [
    (3.5, 2.0, 7.0), (5.0, 1.5, 6.5), (2.5, 3.0, 8.0),
    (4.0, 2.0, 6.0), (1.5, 4.5, 9.0), (3.0, 2.5, 7.5),
    (5.5, 1.0, 6.0), (2.0, 3.5, 8.5), (4.5, 2.0, 7.0),
    (3.0, 3.0, 7.5), (6.0, 1.5, 6.0), (2.5, 4.0, 8.0),
    (4.0, 2.5, 7.0), (5.0, 2.0, 6.5), (3.5, 2.5, 7.0)
]
```

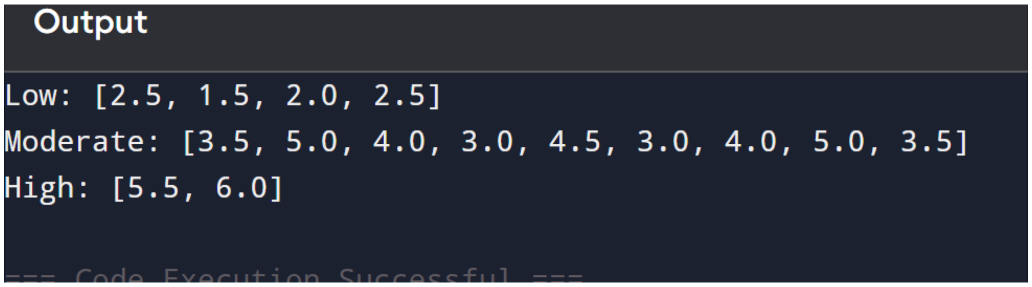
Complete all the tasks below:

Task 1. Classify Study Time:

1. Create empty lists for study time classifications:
   (a) Low: less than 3 hours.
   (b) Moderate: between 3 and 5 hours.
   (c) High: more than 5 hours.
2. Iterate over the time_data list and add each study hour to the appropriate category.
3. Print the lists to verify the classifications.



```python
time_data = [
(3.5, 2.0, 7.0), (5.0, 1.5, 6.5), (2.5, 3.0, 8.0),
(4.0, 2.0, 6.0), (1.5, 4.5, 9.0), (3.0, 2.5, 7.5),
(5.5, 1.0, 6.0), (2.0, 3.5, 8.5), (4.5, 2.0, 7.0),
(3.0, 3.0, 7.5), (6.0, 1.5, 6.0), (2.5, 4.0, 8.0),
(4.0, 2.5, 7.0), (5.0, 2.0, 6.5), (3.5, 2.5, 7.0)
]

low = []
moderate = []
high = []

for study, entertainment, sleep in time_data:
    if study < 3:
        low.append(study)
    elif 3 <= study <= 5:
        moderate.append(study)
    else:
        high.append(study)

print("Low:", low)
print("Moderate:", moderate)
print("High:", high)
```



Output

```
Low: [2.5, 1.5, 2.0, 2.5]
Moderate: [3.5, 5.0, 4.0, 3.0, 4.5, 3.0, 4.0, 5.0, 3.5]
High: [5.5, 6.0]

=== Code Execution Successful ===
```

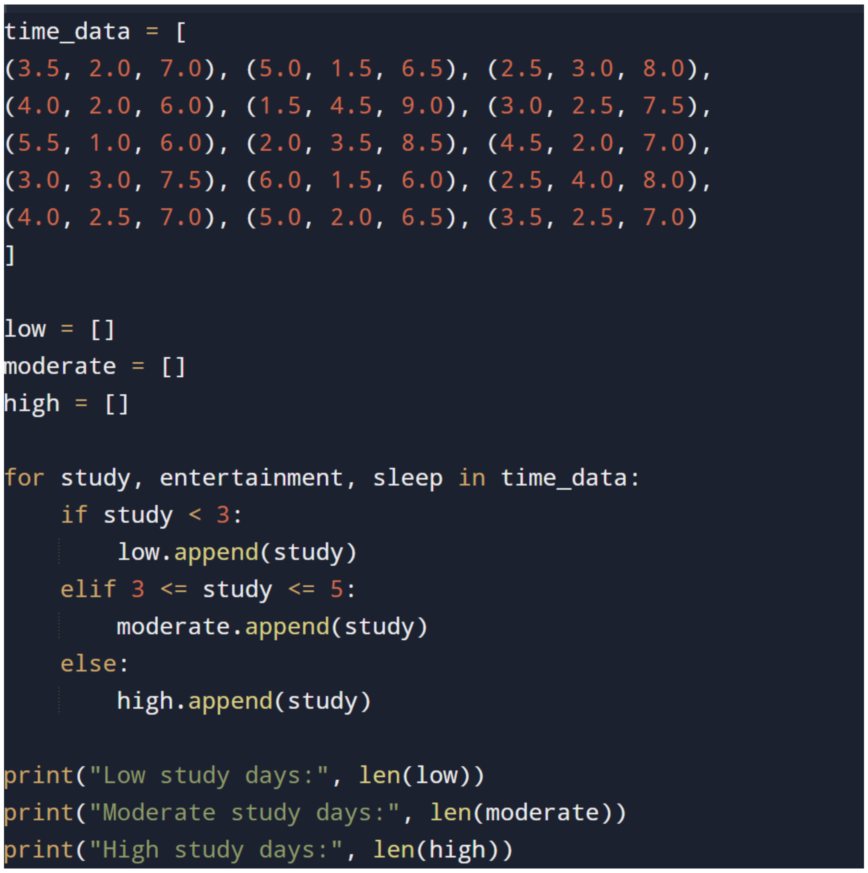
Task 2. Based on Data – Answer all the Questions:

1. How many days had **low study time**?

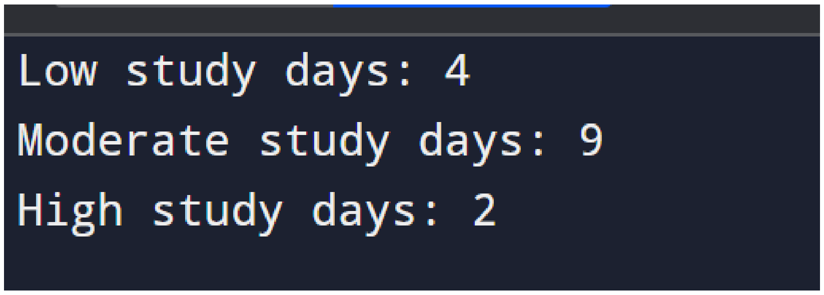
   (a) Hint: Count the number of items in the low study list and print the result.

2. How many days had **moderate study time**?

3. How many days had **high study time**?



```
time_data = [
(3.5, 2.0, 7.0), (5.0, 1.5, 6.5), (2.5, 3.0, 8.0),
(4.0, 2.0, 6.0), (1.5, 4.5, 9.0), (3.0, 2.5, 7.5),
(5.5, 1.0, 6.0), (2.0, 3.5, 8.5), (4.5, 2.0, 7.0),
(3.0, 3.0, 7.5), (6.0, 1.5, 6.0), (2.5, 4.0, 8.0),
(4.0, 2.5, 7.0), (5.0, 2.0, 6.5), (3.5, 2.5, 7.0)
]

low = []
moderate = []
high = []

for study, entertainment, sleep in time_data:
    if study < 3:
        low.append(study)
    elif 3 <= study <= 5:
        moderate.append(study)
    else:
        high.append(study)

print("Low study days:", len(low))
print("Moderate study days:", len(moderate))
print("High study days:", len(high))
```



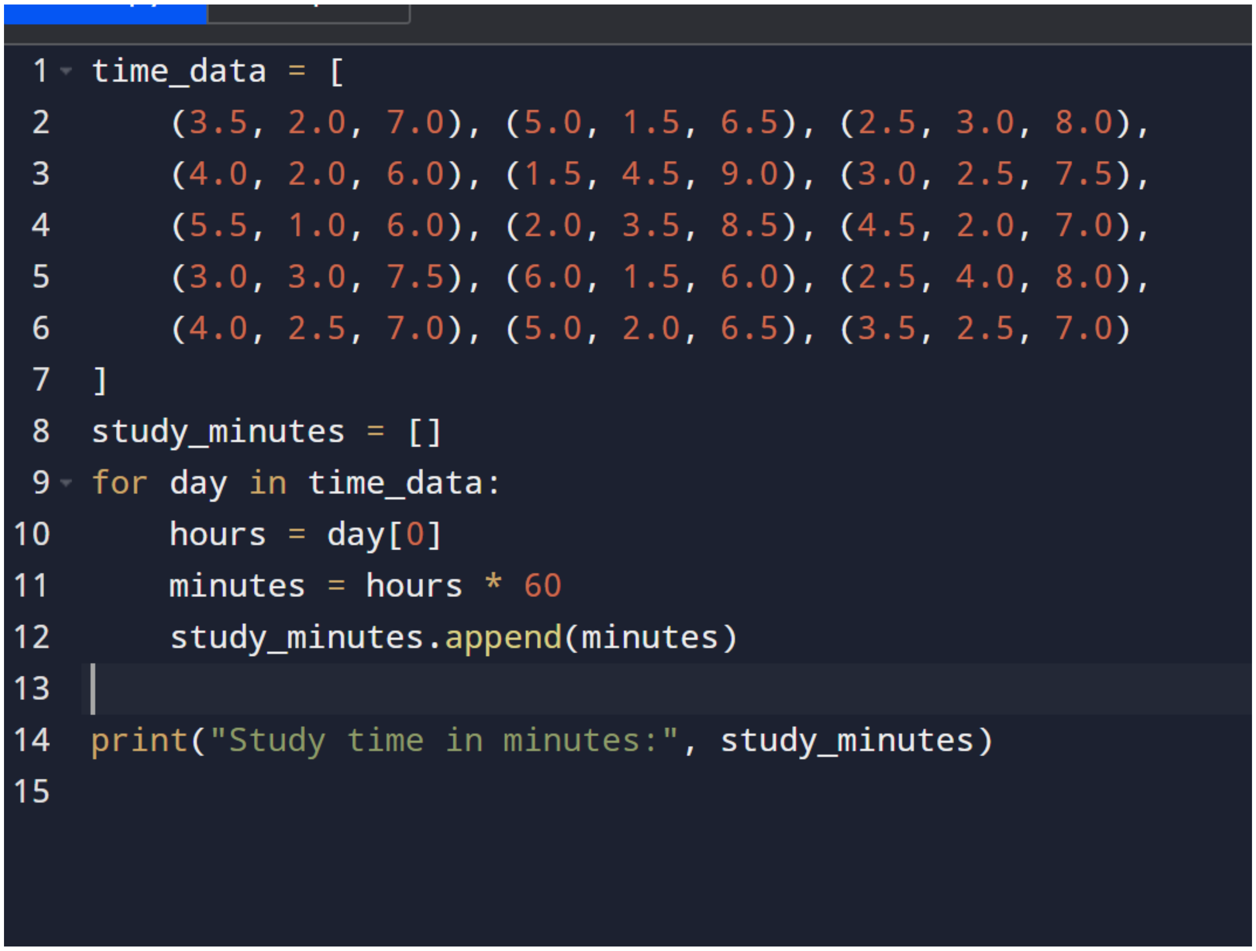
```
Low study days: 4
Moderate study days: 9
High study days: 2
```

Task 3. Convert Study Hours to Minutes:

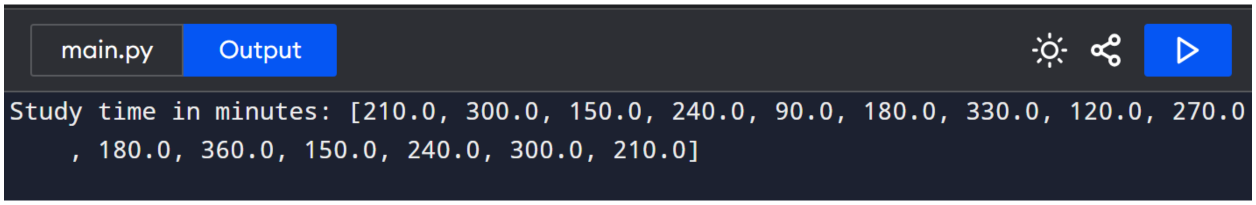
Convert each study hour value into minutes and store it in a new list called study_minutes.

Formula: Minutes = Hours× 60

1. Iterate over the time_data list and apply the formula to the study hours.
2. Store the results in the new list.
3. Print the converted values.



```python
time_data = [
    (3.5, 2.0, 7.0), (5.0, 1.5, 6.5), (2.5, 3.0, 8.0),
    (4.0, 2.0, 6.0), (1.5, 4.5, 9.0), (3.0, 2.5, 7.5),
    (5.5, 1.0, 6.0), (2.0, 3.5, 8.5), (4.5, 2.0, 7.0),
    (3.0, 3.0, 7.5), (6.0, 1.5, 6.0), (2.5, 4.0, 8.0),
    (4.0, 2.5, 7.0), (5.0, 2.0, 6.5), (3.5, 2.5, 7.0)
]
study_minutes = []
for day in time_data:
    hours = day[0]
    minutes = hours * 60
    study_minutes.append(minutes)

print("Study time in minutes:", study_minutes)

```

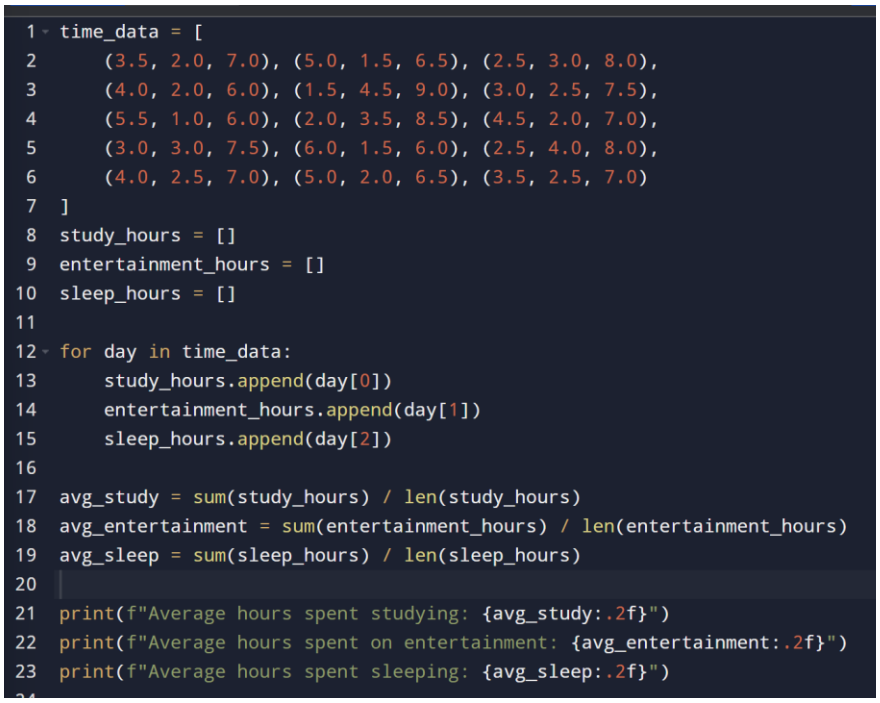


```
Study time in minutes: [210.0, 300.0, 150.0, 240.0, 90.0, 180.0, 330.0, 120.0, 270.0
    , 180.0, 360.0, 150.0, 240.0, 300.0, 210.0]
```

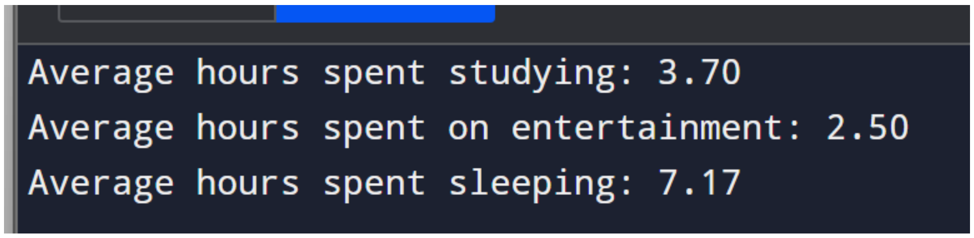
Task 4. Analyze Average Time Use:

Scenario: Each record contains daily hours of study, entertainment, and sleep.

1. Create empty lists for study_hours, entertainment_hours, and sleep_hours.
2. Iterate over time_data and extract values into each list.
3. Calculate and print:
   (a) Average hours spent studying.
   (b) Average hours spent on entertainment.
   (c) Average hours spent sleeping.



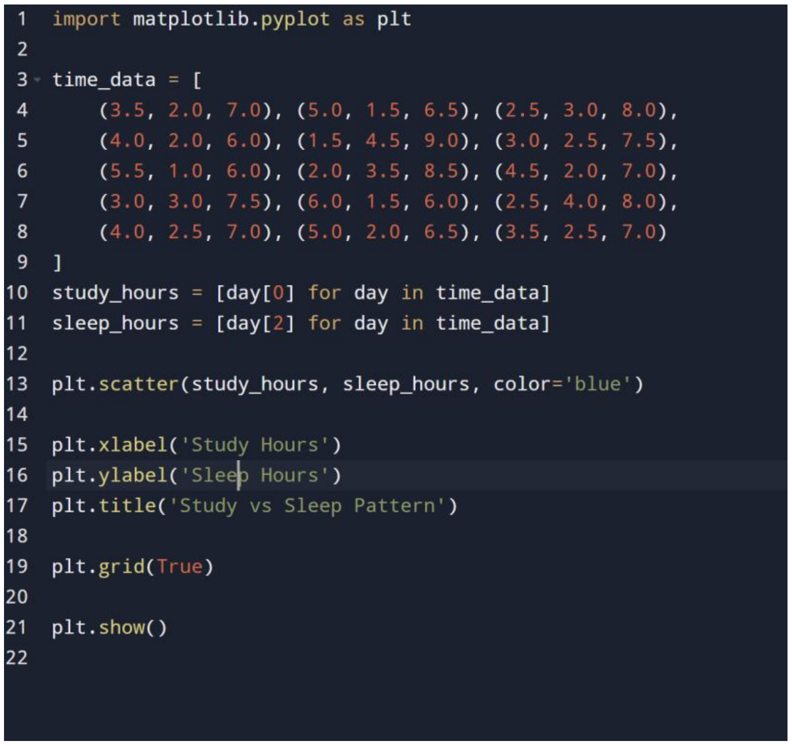
```python
time_data = [
    (3.5, 2.0, 7.0), (5.0, 1.5, 6.5), (2.5, 3.0, 8.0),
    (4.0, 2.0, 6.0), (1.5, 4.5, 9.0), (3.0, 2.5, 7.5),
    (5.5, 1.0, 6.0), (2.0, 3.5, 8.5), (4.5, 2.0, 7.0),
    (3.0, 3.0, 7.5), (6.0, 1.5, 6.0), (2.5, 4.0, 8.0),
    (4.0, 2.5, 7.0), (5.0, 2.0, 6.5), (3.5, 2.5, 7.0)
]
study_hours = []
entertainment_hours = []
sleep_hours = []

for day in time_data:
    study_hours.append(day[0])
    entertainment_hours.append(day[1])
    sleep_hours.append(day[2])

avg_study = sum(study_hours) / len(study_hours)
avg_entertainment = sum(entertainment_hours) / len(entertainment_hours)
avg_sleep = sum(sleep_hours) / len(sleep_hours)

print(f"Average hours spent studying: {avg_study:.2f}")
print(f"Average hours spent on entertainment: {avg_entertainment:.2f}")
print(f"Average hours spent sleeping: {avg_sleep:.2f}")
```



```
Average hours spent studying: 3.70
Average hours spent on entertainment: 2.50
Average hours spent sleeping: 7.17
```

Task 5. Visualization - Study vs Sleep Pattern:

1. Import matplotlib.pyplot as plt.
2. Plot a scatter plot with:
   - x-axis: Study hours
   - y-axis: Sleep hours
3. Add labels, title, and color.



```python
import matplotlib.pyplot as plt

time_data = [
    (3.5, 2.0, 7.0), (5.0, 1.5, 6.5), (2.5, 3.0, 8.0),
    (4.0, 2.0, 6.0), (1.5, 4.5, 9.0), (3.0, 2.5, 7.5),
    (5.5, 1.0, 6.0), (2.0, 3.5, 8.5), (4.5, 2.0, 7.0),
    (3.0, 3.0, 7.5), (6.0, 1.5, 6.0), (2.5, 4.0, 8.0),
    (4.0, 2.5, 7.0), (5.0, 2.0, 6.5), (3.5, 2.5, 7.0)
]
study_hours = [day[0] for day in time_data]
sleep_hours = [day[2] for day in time_data]

plt.scatter(study_hours, sleep_hours, color='blue')

plt.xlabel('Study Hours')
plt.ylabel('Sleep Hours')
plt.title('Study vs Sleep Pattern')

plt.grid(True)

plt.show()

```

8 Problem based on Popular Algorithm.

8.1 Recursion:

1. Definition:

Recursion is a programming technique where a function calls itself to solve a problem. It's typically used when a problem can be broken down into smaller, similar sub-problems. Some key concepts in Recursion:

- Base Case: This is the condition where the recursive calls stop. It prevents infinite loops by returning a value without further recursion.
- Recursive Case: This is the part where the function calls itself with a modified argument, gradually moving towards the base case.
- Stack Memory: Each recursive call adds a layer to the call stack. Too many recursive calls without reaching a base case can lead to a "stack overflow" error due to memory limits.

Example: A classic recursive example is the factorial of a number:

Sample Code - Reccursive Approach for finding factorial of a Number.

```
 def factorial(n):
"""
Calculate the factorial of a non-negative integer n.
The factorial of a number n (denoted n!) is the product of all positive integers less than or equal to n.
Specifically:
    -  If n is 0, the factorial is defined as 1 (base case).
    -  For any positive integer n, the factorial is calculated recursively as n * (n-1) * (n-2) * ... * 1.
Args: n (int): A non-negative integer for which to calculate the factorial.
Returns: int: The factorial of the input number n.
Raises:
    ValueError: If the input n is a negative integer, as factorial is only defined for non-negative integers.
"""
if n == 0: # Base case return 1
else: # Recursive case return n *
    factorial(n - 1)
```

2. General Algorithm for Recursion Problems: • Identify the Base Case: Determine the simplest form of the problem. This is crucial to avoid infinite recursion.

- Divide the Problem: Think about how to reduce the problem's size. Often, the problem should become simpler with each recursive call.

- Formulate the Recursive Case: Decide how each recursive step will bring you closer to the base case.
- Combine Results: If necessary, think about how to combine the results of recursive calls to form the final result.

3. How to Identify Recursion-Based Problems:

1. Problem Can Be Split into Smaller Versions of Itself: • If the problem can be divided into smaller sub-problems that resemble the original, recursion might be suitable.
   - Example: Summing all numbers in a nested list. Each sub-list can be treated as a smaller instance of the original list.
2. Tasks with Nested or Hierarchical Structures:
   (a) Problems involving nested data structures, like nested lists or tree structures, are often recursive by nature.
   (b) Example: Calculating the size of a directory with subdirectories.
3. Repetitive or Iterative Nature with Reducing Input: • Problems that involve performing an operation repeatedly with a progressively smaller (or simpler) input are ideal candidates.
   - Example: Calculating the power of a number, where the exponent decreases with each call

4. Problems Involving Dividing and Conquering: • When a problem can be divided into parts that can be solved independently, recursion is often useful, as seen in sorting and search algorithms like quicksort and binary search.

8.1.1 Exercise - Recursion:

Task 1 - Sum of Nested Lists:

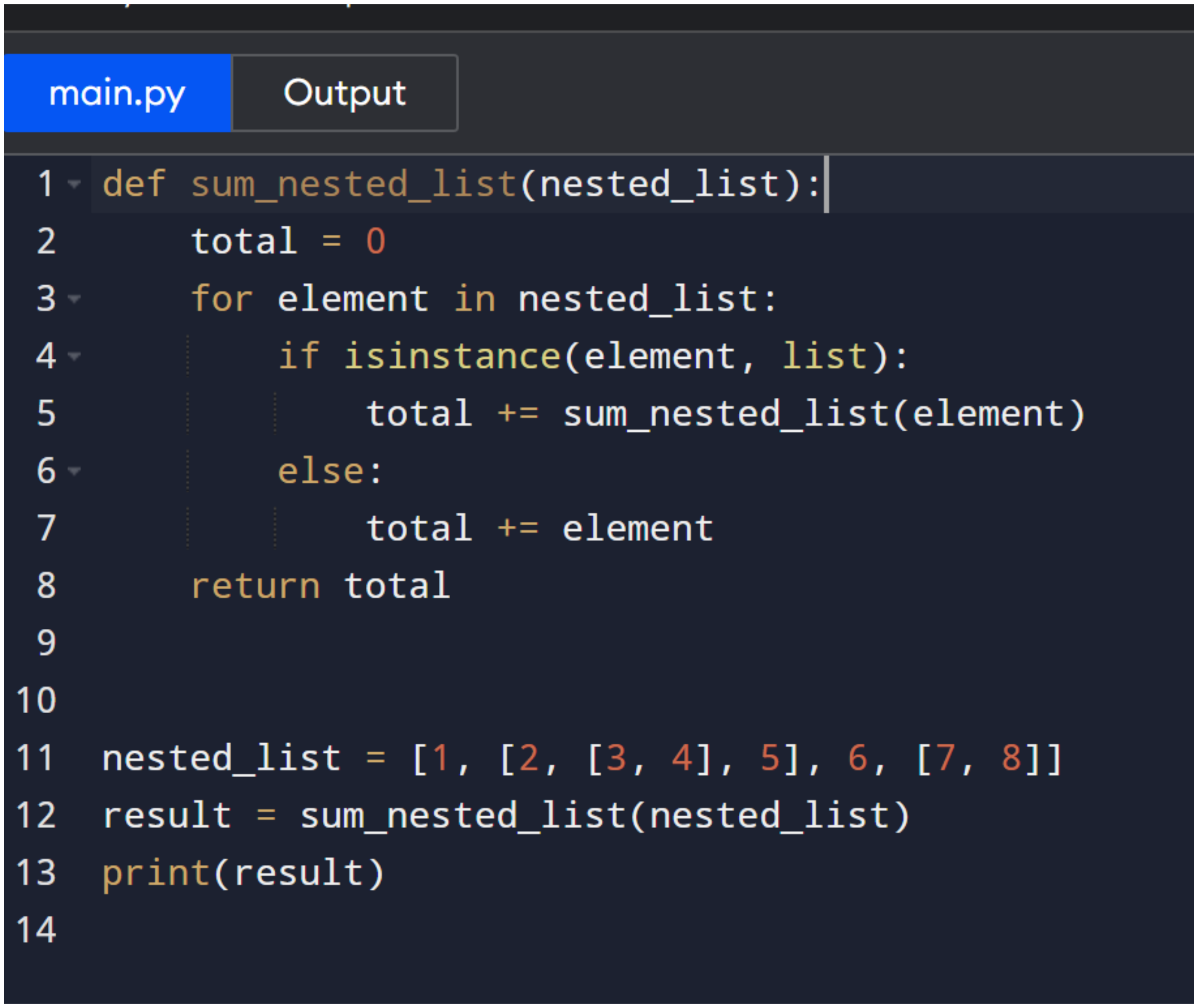
Scenario: You have a list that contains numbers and other lists of numbers (nested lists). You want to find the total sum of all the numbers in this structure. Task:

- Write a recursive function sum_nested_list(nested_list) that:
  1. Takes a nested list (a list that can contain numbers or other lists of numbers) as input.
  2. Sums all numbers at every depth level of the list, regardless of how deeply nested the numbers are.

- Test the function with a sample nested list, such as nested_list = [1, [2, [3, 4], 5], 6, [7, 8]].
  The result should be the total sum of all the numbers.

Sample Code - Sum of Nested lists.

```
def sum_nested_list(nested_list):
"""
Calculate the sum of all numbers in a nested list.
This function takes a list that may contain integers and other nested lists.
It recursively traverses the list and sums all the integers, no matter how deeply nested they are.
Args: nested_list (list): A list that may contain integers or other lists of integers.
Returns:
    int: The total sum of all integers in the nested list, including those in sublists . Example:
    >>> sum_nested_list([1, [2, [3, 4], 5], 6, [7, 8]])
    36
    >>> sum_nested_list([1, [2, 3], [4, [5]]]) 15
"""
total = 0 for element in
nested_list:
    if isinstance(element, list): # Check if the element is a list total +=
        sum_nested_list(element) # Recursively sum the nested list
    else: total += element # Add the number to the total return
total
```


main.py Output

```python
def sum_nested_list(nested_list):
    total = 0
    for element in nested_list:
        if isinstance(element, list):
            total += sum_nested_list(element)
        else:
            total += element
    return total


nested_list = [1, [2, [3, 4], 5], 6, [7, 8]]
result = sum_nested_list(nested_list)
print(result)
```

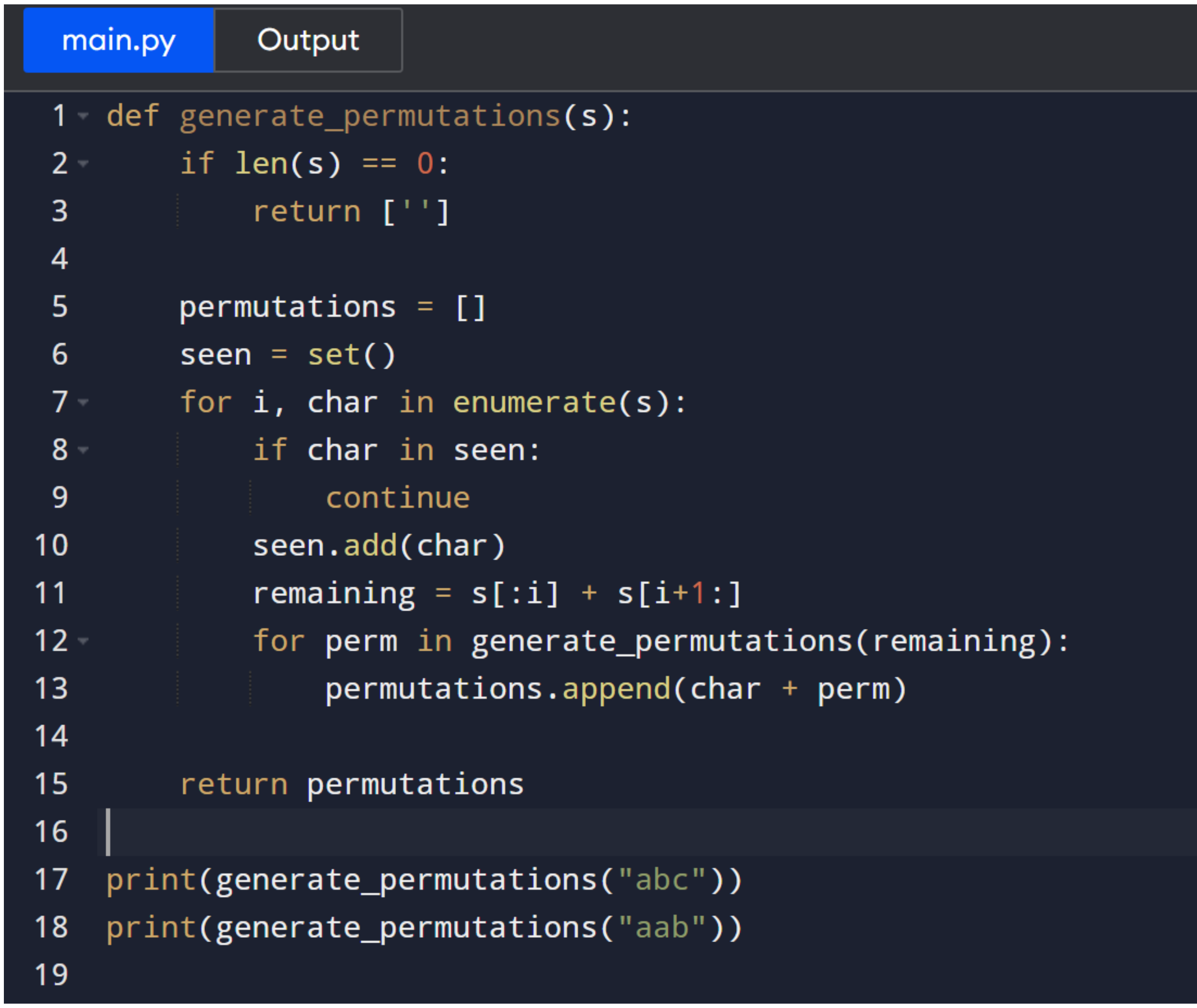
Result:36

Task 2 - Generate All Permutations of a String:

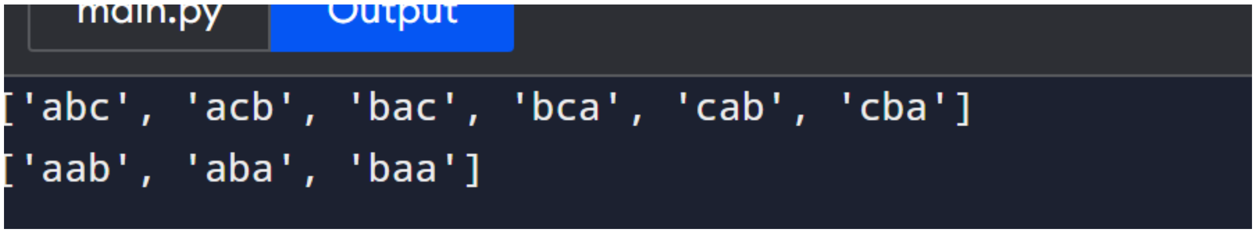
Scenario: Given a string, generate all possible permutations of its characters. This is useful for understanding backtracking and recursive depth-first search. Task:

- Write a recursive function generate_permutations(s) that:
  - Takes a string s as input and returns a list of all unique permutations.
- Test with strings like ”abc” and ”aab”.

```
print(generate_permutations("abc"))
# Should return [’abc’, ’acb’, ’bac’, ’bca’, ’cab’, ’cba’]
```



```python
def generate_permutations(s):
    if len(s) == 0:
        return ['']

    permutations = []
    seen = set()
    for i, char in enumerate(s):
        if char in seen:
            continue
        seen.add(char)
        remaining = s[:i] + s[i+1:]
        for perm in generate_permutations(remaining):
            permutations.append(char + perm)

    return permutations

print(generate_permutations("abc"))
print(generate_permutations("aab"))
```



```
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
['aab', 'aba', 'baa']
```

Task 3 - Directory Size Calculation:

Directory Size Calculation Scenario: Imagine a file system where directories can contain files (with sizes in KB) and other directories. You want to calculate the total size of a directory, including all nested files and subdirectories.

Sample directory structure.

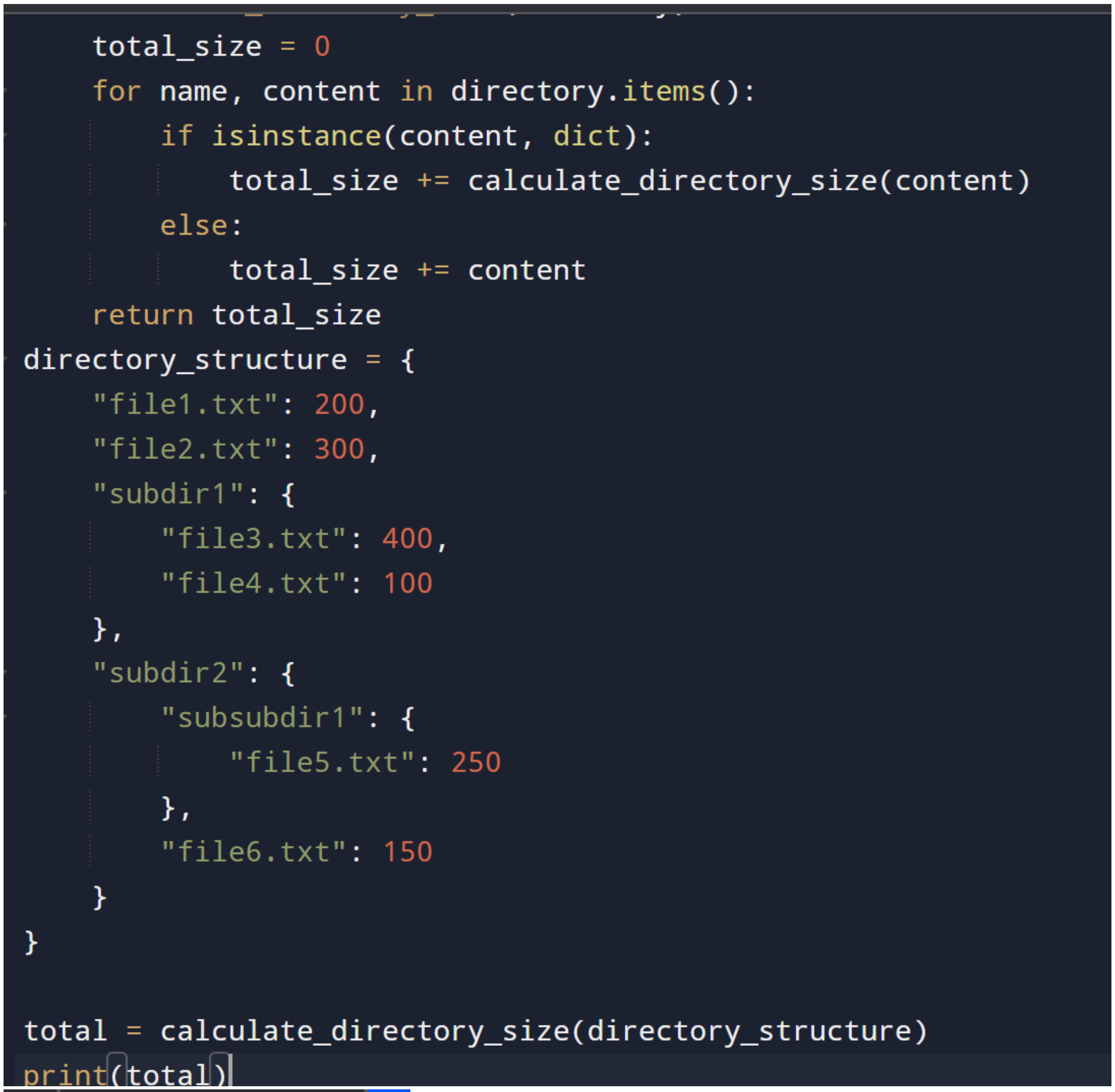
```
# Sample directory structure directory_structure = {
    "file1.txt": 200,
    "file2.txt": 300,
    "subdir1": {
        "file3.txt": 400,
        "file4.txt": 100
    },
    "subdir2": {
        "subsubdir1": {
            "file5.txt": 250
        },
        "file6.txt": 150
    }
}
```

Task:

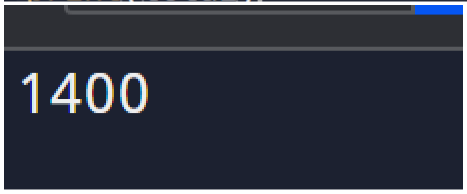
1. Write a recursive function calculate_directory_size(directory) where:

    - directory is a dictionary where keys represent file names (with values as sizes in KB) or directory names (with values as another dictionary representing a subdirectory).
    - The function should return the total size of the directory, including all nested subdirectories.

    3. Test the function with a sample directory structure.



```
    total_size = 0
    for name, content in directory.items():
        if isinstance(content, dict):
            total_size += calculate_directory_size(content)
        else:
            total_size += content
    return total_size
directory_structure = {
    "file1.txt": 200,
    "file2.txt": 300,
    "subdir1": {
        "file3.txt": 400,
        "file4.txt": 100
    },
    "subdir2": {
        "subsubdir1": {
            "file5.txt": 250
        },
        "file6.txt": 150
    }
}

total = calculate_directory_size(directory_structure)
print(total)
```



```
1400
```

8.2 Dynamic Programming:

1. Introduction to Dynamic Programming:

Dynamic Programming is a method used for solving problems that can be broken down into smaller sub-problems. It is particularly useful when the problem exhibits overlapping sub-problems and optimal substructure.

• Overlapping Sub problems: This property occurs when the problem can be divided into smaller sub problems, and these sub problems are solved multiple times. • Optimal Substructure: The optimal solution to the problem can be constructed from optimal solutions to its sub-problems.

2. Type of Dynamic Programming:

1. Memoization: A top-down approach where we solve the problem by breaking it into sub-problems, and store the results of sub-problems to avoid redundant computations.
2. Tabulation: A bottom-up approach that starts solving from the smallest subproblem, and progressively solves larger sub-problems while storing results in a table.

3. Steps in Dynamic Programming:

1. Define the subproblem: Break the original problem into smaller sub-problems.
2. Recursive Relation: Identify the relation between sub-problems (i.e., how to compute the solution of the problem based on the solutions to smaller sub-problems).
3. Store the results: Cache or store results of sub-problems to avoid recalculating them (memoization or tabulation).
4. Build the final solution: Combine the solutions of sub-problems to get the solution to the original problem.

4. Example:

Fibonacci Sequence: The Fibonacci sequence is defined as:

$$\begin{cases} F(0) = 0 \\ F(1) = 1 \\ F(n) = F(n-1) + F(n-2) & \text{for } n > 1 \end{cases} \quad (1)$$

Solving Fibonacci Sequnce with Memoization {Top - Down Approach}.

```
def fibonacci(n, memo={}):
        """
        Computes the nth Fibonacci number using memoization to optimize the recursive solution.

This function uses memoization to store previously computed Fibonacci numbers, reducing redundant calculations
         and improving performance.
Parameters:
n (int): The index in the Fibonacci sequence for which the value is to be computed. Must be a non-negative
integer. memo (dict, optional): A dictionary used to store previously computed Fibonacci values.
                                    It is initialized as an empty dictionary by default and is used during the recursive calls
to avoid recalculating results. Returns:
int: The nth Fibonacci number.
Example:
>>> fibonacci(5)
5
>>> fibonacci(10)
55
Time Complexity:
-   The time complexity is O(n) because each Fibonacci number is computed only once.
Space Complexity:
-   The space complexity is O(n) due to the memoization dictionary storing the results.
"""
if n in memo: return
memo[n] if n <= 1:
        return n
memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo) return memo[n]
```

Solving Fibonacci Sequnce with Tbulation {Bottom - Up Approach}.

```
def fibonacci(n): """
    Computes the nth Fibonacci number using dynamic programming to optimize the solution.
    This function uses a bottom-up dynamic programming approach to calculate the nth
        Fibonacci number by iteratively building up an array of Fibonacci numbers up to n, thus
    eliminating
    redundant calculations and optimizing performance.
    Parameters:
    n (int): The index in the Fibonacci sequence for which the value is to be computed.
             Must be a non-negative integer.
    Returns:
    int: The nth Fibonacci number.
    Example:
    >>> fibonacci(5)
    5
    >>> fibonacci(10)
    55
    Time Complexity:
    -  The time complexity is O(n), as the function iterates through the range 2 to n, calculating
       each Fibonacci number once.
    Space Complexity:
    -  The space complexity is O(n), due to the array used to store the Fibonacci numbers
       up to n.
"""
if n <= 1:
    return n
dp = [0] * (n + 1) dp[1] = 1 for i in
range(2, n + 1):
    dp[i] = dp[i - 1] + dp[i - 2]
return dp[n]
```

5. **Tips for Identifying DP Problems:** • Overlapping Sub-problems: If a problem requires solving the same subproblem multiple times, it is a good candidate for DP.

   - Optimal Substructure: If an optimal solution can be constructed from the optimal solutions to sub-problems, the problem might be solvable using DP.
   - Recursive Structure: If the problem can be expressed recursively, check if storing intermediate results helps in improving efficiency.

8.2.1 Exercises - Dynamic Programming:

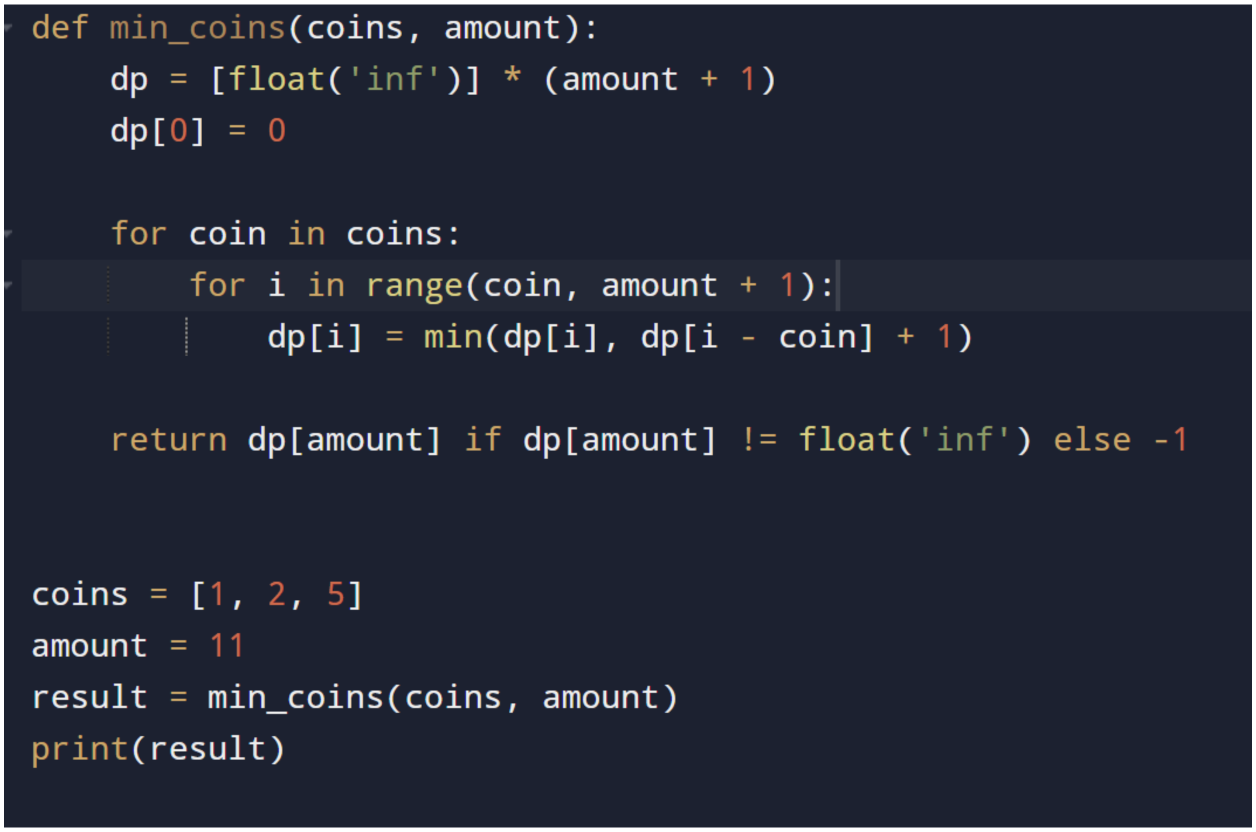
Task 1 - Coin Change Problem:

Scenario: Given a set of coin denominations and a target amount, find the minimum number of coins needed to make the amount. If it's not possible, return - 1. Task:

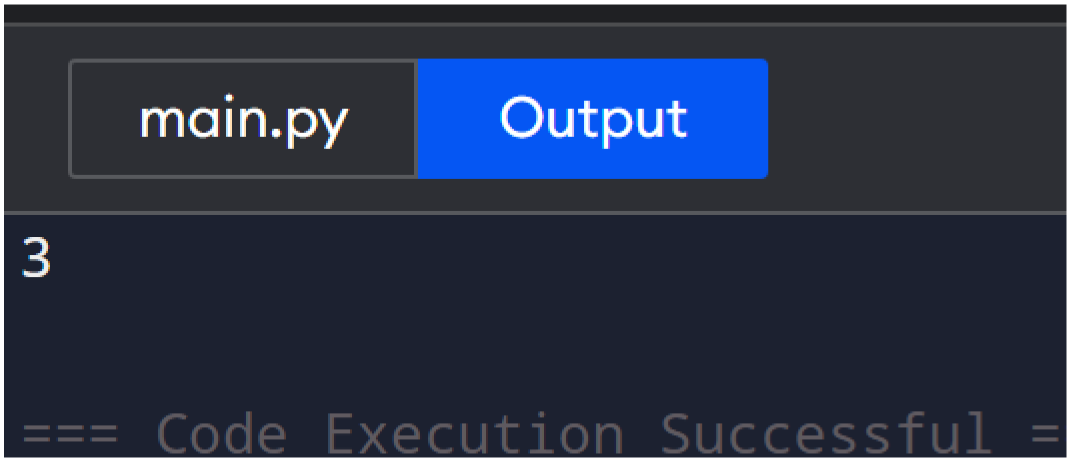
1. Write a function min_coins(coins, amount) that: • Uses DP to calculate the minimum number of coins needed to make up the amount.
2. Test with coins = [1, 2, 5] and amount = 11. The result should be 3 (using coins [5, 5, 1]).

Sample Code - Coin Change Problem.

```
def min_coins(coins, amount):
"""
Finds the minimum number of coins needed to make up a given amount using dynamic programming.
This function solves the coin change problem by determining the fewest number of coins from a given set of
    coin denominations that sum up to a target amount. The solution uses dynamic
    programming(tabulation) to iteratively build up the minimum number of coins required for each amount.
Parameters:
coins (list of int): A list of coin denominations available for making change. Each coin denomination is a
    positive integer.
amount (int): The target amount for which we need to find the minimum number of coins . It must be a
    non-negative integer.
Returns:
int: The minimum number of coins required to make the given amount.
     If it is not possible to make the amount with the given coins, returns -1. Example:
>>> min_coins([1, 2, 5], 11)
3
>>> min_coins([2], 3)
-1
""" dp = [float('inf')] * (amount + 1)
dp[0] = 0

for coin in coins:
    for i in range(coin, amount + 1):
        dp[i] = min(dp[i], dp[i - coin] + 1) return dp[amount] if
dp[amount] != float('inf') else -1
```



```python
def min_coins(coins, amount):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0

    for coin in coins:
        for i in range(coin, amount + 1):
            dp[i] = min(dp[i], dp[i - coin] + 1)

    return dp[amount] if dp[amount] != float('inf') else -1


coins = [1, 2, 5]
amount = 11
result = min_coins(coins, amount)
print(result)
```

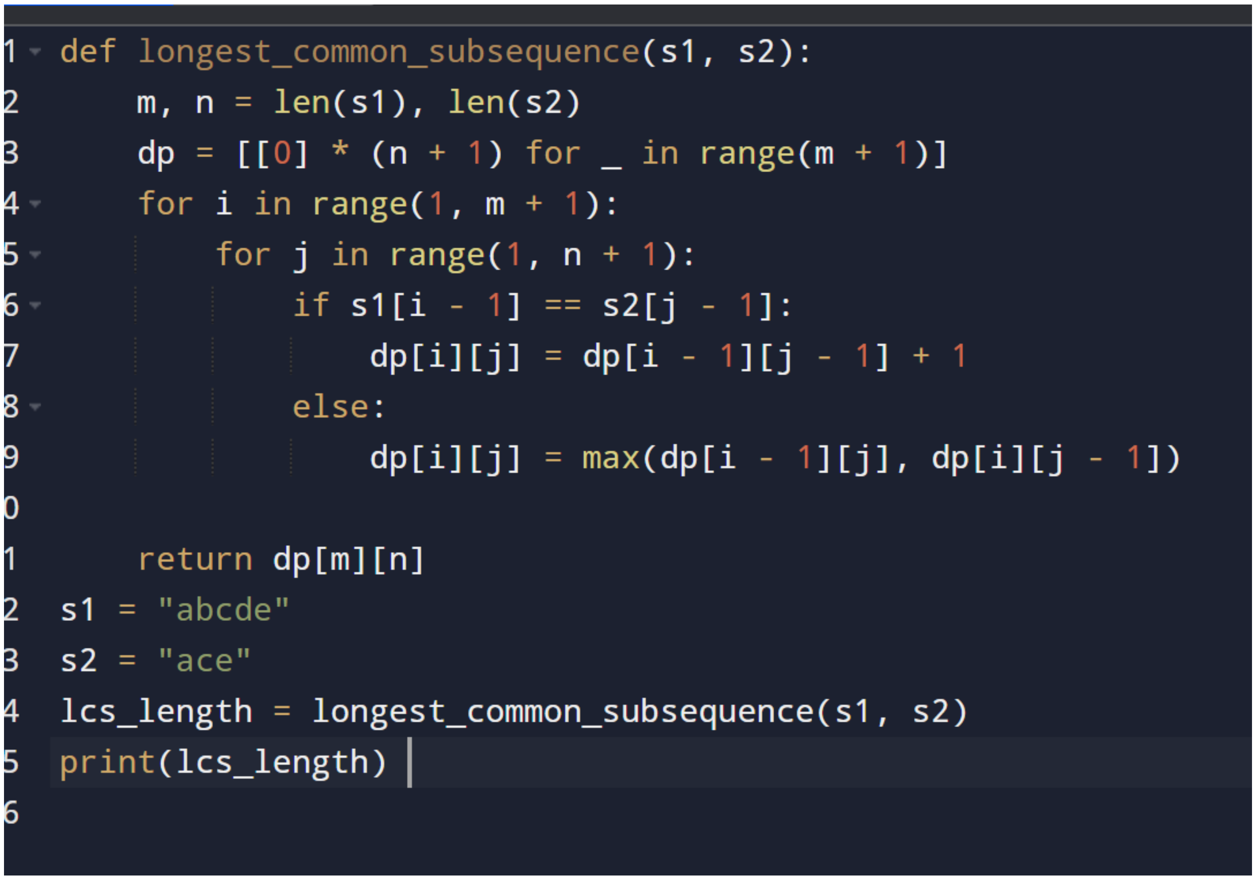


```
3


=== Code Execution Successful =
```

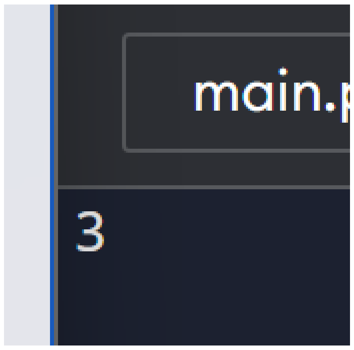
Task 2 - Longest Common Subsequence (LCS):

Scenario: Given two strings, find the length of their longest common subsequence (LCS). This is useful in text comparison. Task:

1. Write a function longest_common_subsequence(s1, s2) that:
   - Uses DP to find the length of the LCS of two strings s1 and s2.
2. Test with strings like "abcde" and "ace"; the LCS length should be 3 ("ace").



```python
def longest_common_subsequence(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i - 1] == s2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[m][n]
s1 = "abcde"
s2 = "ace"
lcs_length = longest_common_subsequence(s1, s2)
print(lcs_length)
```



```
3
```

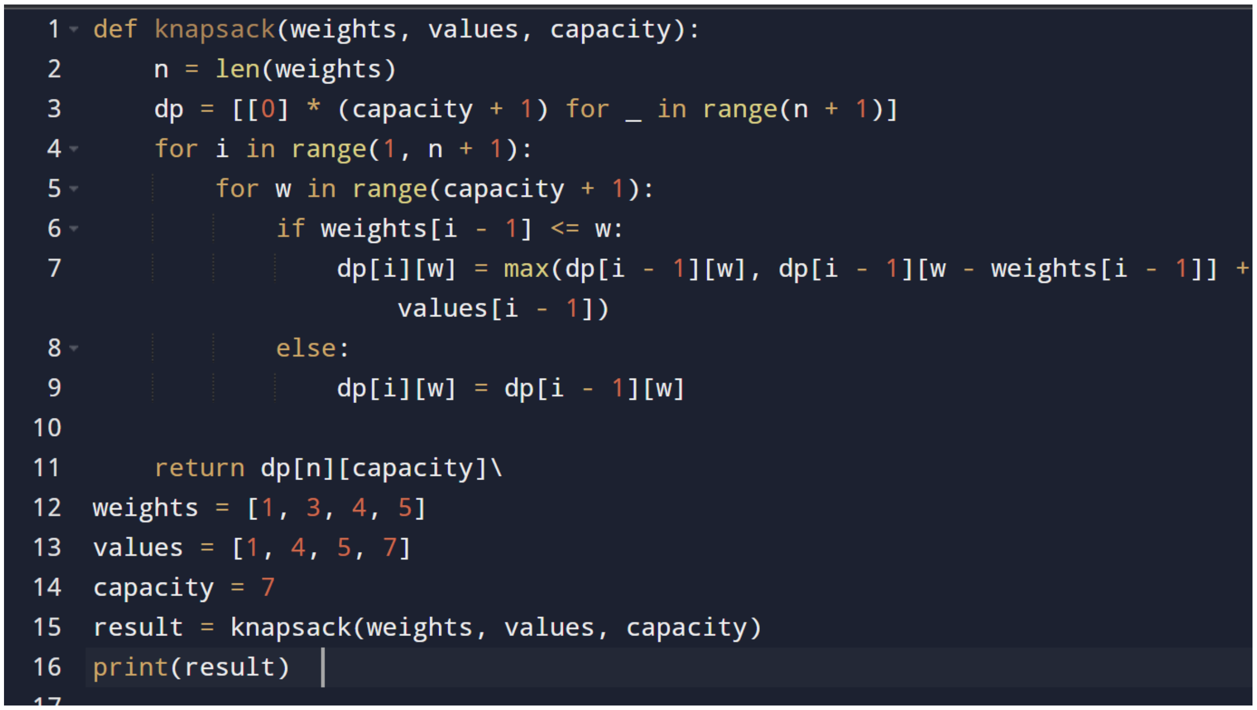
Task 3 - 0/1 Knapsack Problem:

Scenario: You have a list of items, each with a weight and a value. Given a weight capacity, maximize the total value of items you can carry without exceeding the weight capacity. Task:

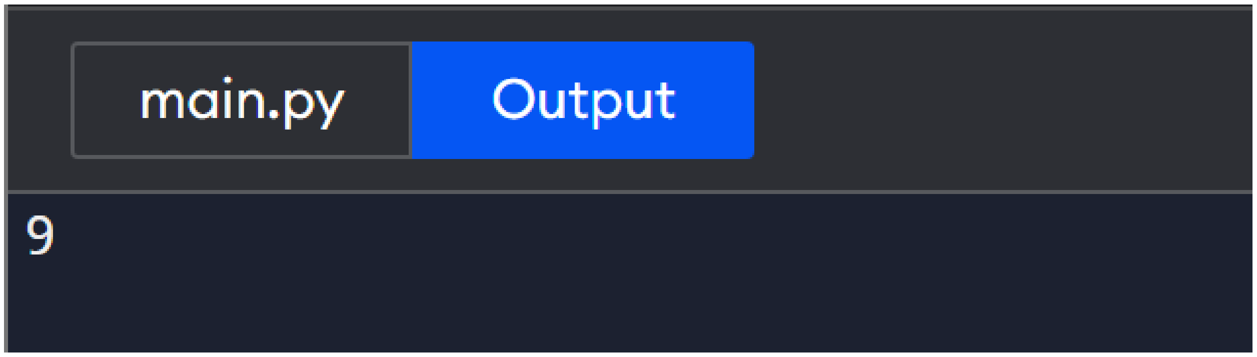
1. Write a function knapsack(weights, values, capacity) that: • Uses DP to determine the

   maximum value that can be achieved within the given weight capacity.

2. Test with weights [1, 3, 4, 5], values [1, 4, 5, 7], and capacity 7. The result should be 9.



```python
def knapsack(weights, values, capacity):
    n = len(weights)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        for w in range(capacity + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] +
                    values[i - 1])
            else:
                dp[i][w] = dp[i - 1][w]

    return dp[n][capacity]\
weights = [1, 3, 4, 5]
values = [1, 4, 5, 7]
capacity = 7
result = knapsack(weights, values, capacity)
print(result)
```



```
9
```

9 Closing Remarks on Recursive and Dynamic Algorithm.

Dynamic Programming (DP) and recursion are both techniques used to solve problems by breaking them down into smaller sub-problems. However, they differ significantly in how they approach problem-solving and how efficiently they execute. Here's a comparison of both approaches:

1. Problem Solving Approach:

- Recursion:
    1. Recursion is generally easier to implement when solving problems with a clear recursive structure, like tree traversal or calculating factorials.
    2. It is suitable for problems where each subproblem is independent and doesn't need to store its results for reuse.
- Dynamic Programming:
    1. DP is used for problems that have optimal substructure and overlapping subproblems. It's more complex to implement than simple recursion because it requires designing a table or memoization structure.
    2. DP is more suitable for optimization problems, like shortest path problems, knapsack problems, etc., where storing intermediate results is crucial for efficient computation.

2. When to Use Which?

- Use Recursion:
    1. When the problem naturally fits a recursive structure and does not involve overlapping sub-problems.
    2. When the problem has a small problem size (so the overhead of recursion is minimal) or when an elegant, simple solution is needed without worrying about performance.
    3. Examples: Tree traversal, divide-and-conquer algorithms (like merge sort).
- Use Dynamic Programming:
    1. When the problem involves overlapping sub-problems and optimal substructure.
    2. When you need to optimize recursive solutions to avoid redundant work.
    3. Examples: Knapsack problem, Fibonacci sequence, shortest path algorithms (e.g., Dijkstra's, Bellman-Ford).

——————————– Fasten Your Seat Belt and Enjoy the Ride. ——————————-