Practical 3

3.1 Functions

Aim:
Basics of function

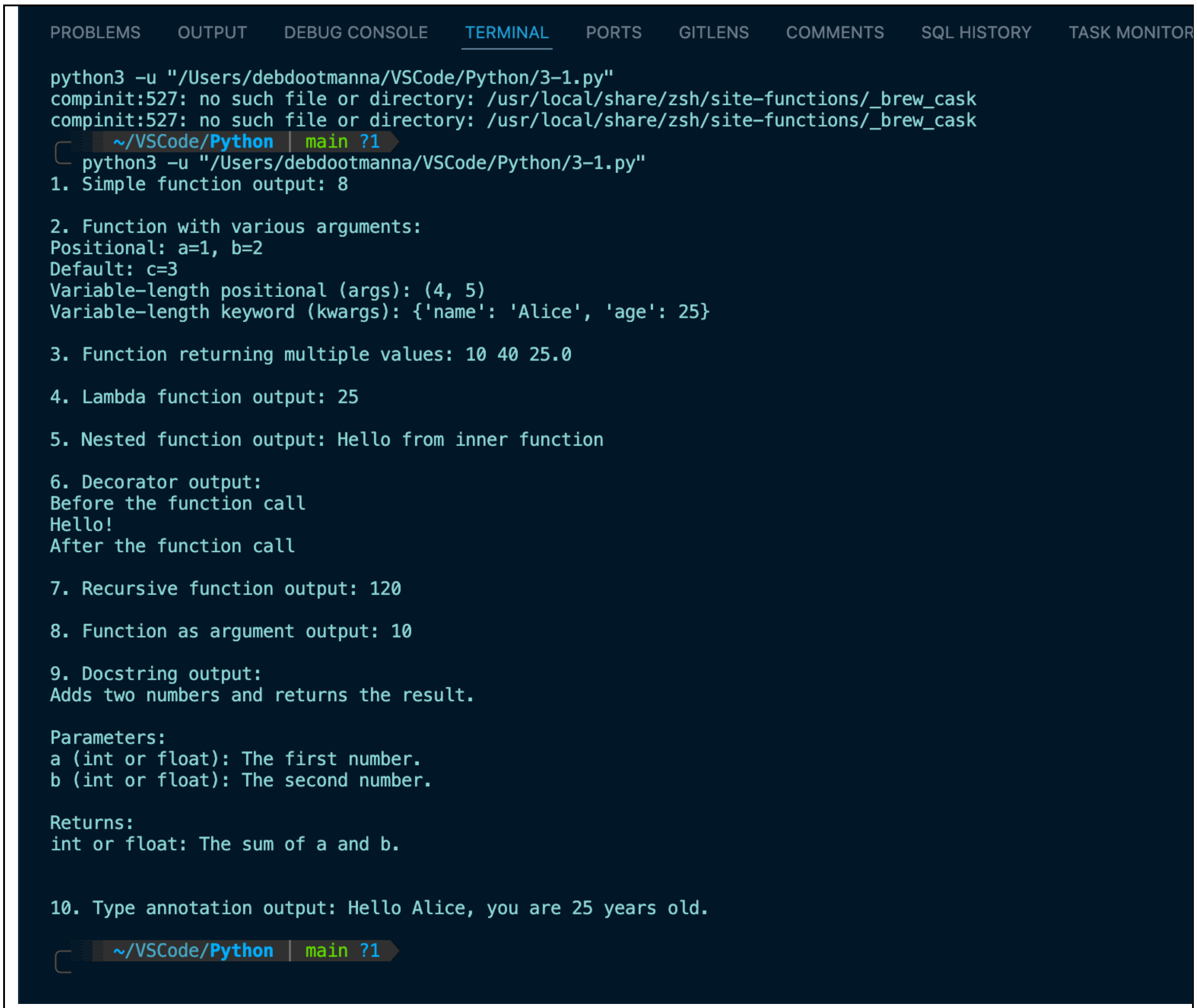
- ☐ Define a simple function that takes inputs and returns an output.
- ☐ Define a function with positional, keyword, default, and variable-length arguments
- ☐ Define a function that returns the multiple values using tuple.
- ☐ Define an anonymous function using lambda keyword
- ☐ Define a function inside another function.
- ☐ Create and use decorators to modify the behavior of functions.
- ☐ Define a function that calls itself to solve a problem recursively.
- ☐ Define functions that take other functions as arguments or return functions as results.
- ☐ Add docstrings to functions to document their purpose and usage.
- ☐ Use type annotations to specify the expected types of function arguments and return values.

Code:

```python
# 1. Define a simple function that takes inputs and returns an output.
def add(a, b):
    return a + b

result = add(3, 5)
print("1. Simple function output:", result)  # Output: 8

# 2. Define a function with positional, keyword, default, and variable-length arguments.
def example_function(a, b, c=10, *args, **kwargs):
    print("\n2. Function with various arguments:")
    print(f"Positional: a={a}, b={b}")
    print(f"Default: c={c}")
    print(f"Variable-length positional (args): {args}")
    print(f"Variable-length keyword (kwargs): {kwargs}")

example_function(1, 2, 3, 4, 5, name="Alice", age=25)

# 3. Define a function that returns multiple values using a tuple.
def get_stats(numbers):
    return min(numbers), max(numbers), sum(numbers) / len(numbers)

min_val, max_val, avg_val = get_stats([10, 20, 30, 40])
print("\n3. Function returning multiple values:", min_val, max_val,
avg_val)  # Output: 10 40 25.0

# 4. Define an anonymous function using the `lambda` keyword.
square = lambda x: x ** 2
print("\n4. Lambda function output:", square(5))  # Output: 25
```

```python
# 5. Define a function inside another function.
def outer_function():
    def inner_function():
        return "Hello from inner function"
    return inner_function()

print("\n5. Nested function output:", outer_function())  # Output:
Hello from inner function

# 6. Create and use decorators to modify the behavior of functions.
def my_decorator(func):
    def wrapper():
        print("\n6. Decorator output:")
        print("Before the function call")
        func()
        print("After the function call")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()

# 7. Define a function that calls itself to solve a problem
recursively.
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)

print("\n7. Recursive function output:", factorial(5))  # Output: 120

# 8. Define functions that take other functions as arguments or return
functions as results.
def apply_function(func, value):
    return func(value)

def multiply_by_two(x):
    return x * 2

print("\n8. Function as argument output:",
apply_function(multiply_by_two, 5))  # Output: 10

# 9. Add docstrings to functions to document their purpose and usage.
def add(a, b):
    """
    Adds two numbers and returns the result.

    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("\n9. Docstring output:", add.__doc__)

# 10. Use type annotations to specify the expected types of function
arguments and return values.
def greet(name: str, age: int) -> str:
    return f"Hello {name}, you are {age} years old."

print("\n10. Type annotation output:", greet("Alice", 25))
```

Output Screenshot:



```
python3 -u "/Users/debdootmanna/VSCode/Python/3-1.py"
compinit:527: no such file or directory: /usr/local/share/zsh/site-functions/_brew_cask
compinit:527: no such file or directory: /usr/local/share/zsh/site-functions/_brew_cask
   ~/VSCode/Python  main ?1
  python3 -u "/Users/debdootmanna/VSCode/Python/3-1.py"
1. Simple function output: 8

2. Function with various arguments:
Positional: a=1, b=2
Default: c=3
Variable-length positional (args): (4, 5)
Variable-length keyword (kwargs): {'name': 'Alice', 'age': 25}

3. Function returning multiple values: 10 40 25.0

4. Lambda function output: 25

5. Nested function output: Hello from inner function

6. Decorator output:
Before the function call
Hello!
After the function call

7. Recursive function output: 120

8. Function as argument output: 10

9. Docstring output:
Adds two numbers and returns the result.

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.


10. Type annotation output: Hello Alice, you are 25 years old.

   ~/VSCode/Python  main ?1
```

3.2 Loops **Date: **/**/******

Aim:

Basics of loops

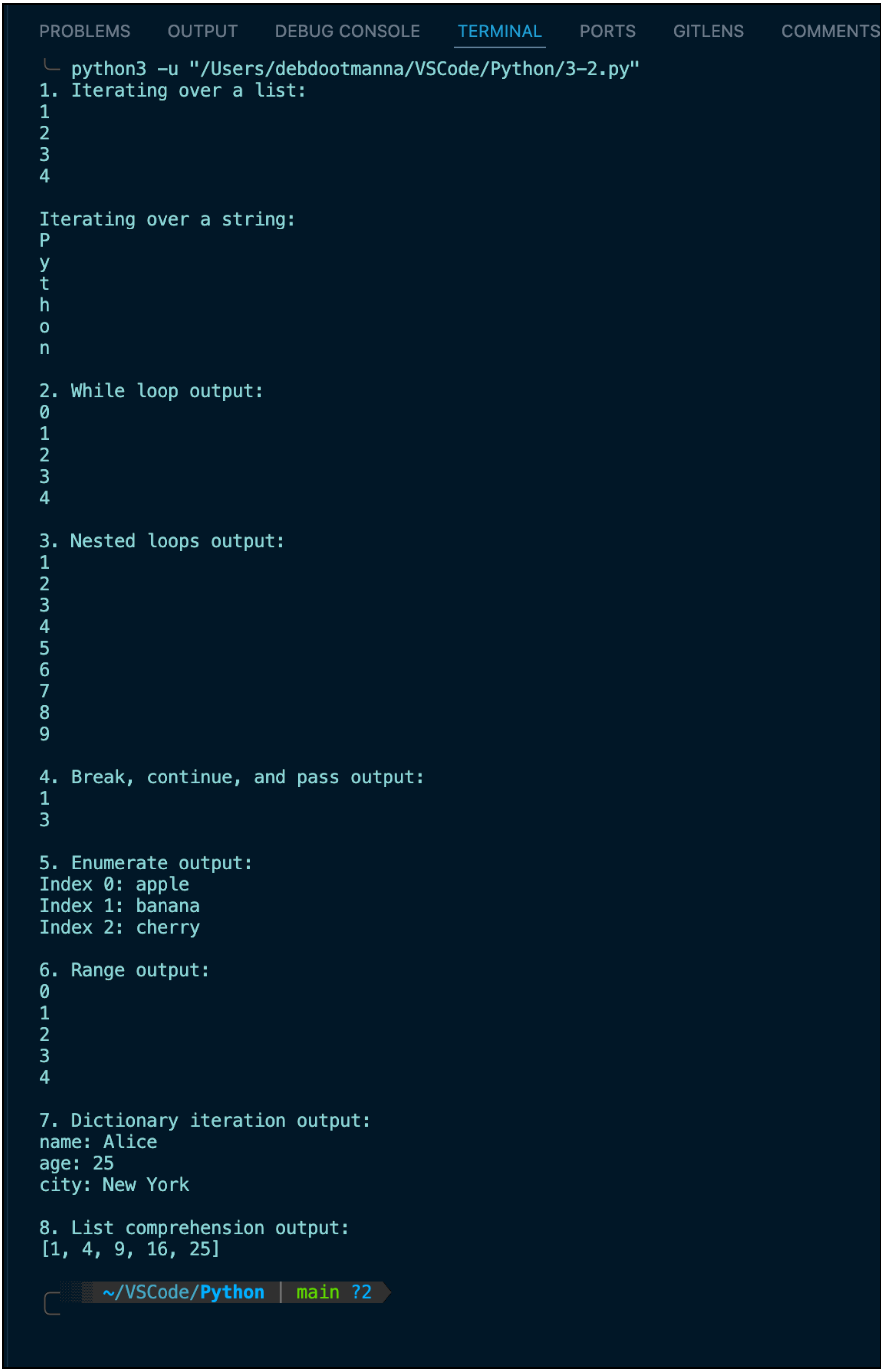
 Iterate over a sequence (list, tuple, string, or range) using a for loop.
 Repeat a block of code as long as a condition is true using a while loop.
 Use loops inside other loops to handle multi-dimensional data structures.
 Use break, continue, and pass to control the flow of loops.
 Use the enumerate function to get both the index and value while iterating over a sequence.
 Use the range function to generate a sequence of numbers for iteration.
 Iterate over the key-value pairs of a dictionary using a for loop.
 Use list comprehensions to create new lists by applying an expression to each item in an existing list.

Code:

```
x# 1. Iterate over a sequence (list, tuple, string, or range) using a
`for` loop.
print("1. Iterating over a list:")
for item in [1, 2, 3, 4]:
    print(item)

print("\nIterating over a string:")
for char in "Python":
    print(char)

# 2. Repeat a block of code as long as a condition is true using a
`while` loop.
print("\n2. While loop output:")
count = 0
while count < 5:
    print(count)
    count += 1

# 3. Use loops inside other loops to handle multi-dimensional data
structures.
print("\n3. Nested loops output:")
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for row in matrix:
    for item in row:
        print(item)

# 4. Use `break`, `continue`, and `pass` to control the flow of loops.
print("\n4. Break, continue, and pass output:")
for i in range(10):
    if i == 5:
        break  # Exit the loop
    if i % 2 == 0:
        continue  # Skip the rest of the loop body
```

```python
    print(i)

# 5. Use the `enumerate` function to get both the index and value while
iterating over a sequence.
print("\n5. Enumerate output:")
fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(fruits):
    print(f"Index {index}: {fruit}")

# 6. Use the `range` function to generate a sequence of numbers for
iteration.
print("\n6. Range output:")
for i in range(5):
    print(i)  # Output: 0 1 2 3 4

# 7. Iterate over the key-value pairs of a dictionary using a `for`
loop.
print("\n7. Dictionary iteration output:")
person = {"name": "Alice", "age": 25, "city": "New York"}
for key, value in person.items():
    print(f"{key}: {value}")

# 8. Use list comprehensions to create new lists by applying an
expression to each item in an existing list.
print("\n8. List comprehension output:")
numbers = [1, 2, 3, 4, 5]
squares = [x ** 2 for x in numbers]
print(squares)  # Output: [1, 4, 9, 16, 25]
```

Output Screenshot:



```
╰ python3 -u "/Users/debdootmanna/VSCode/Python/3-2.py"
1. Iterating over a list:
1
2
3
4

Iterating over a string:
P
y
t
h
o
n

2. While loop output:
0
1
2
3
4

3. Nested loops output:
1
2
3
4
5
6
7
8
9

4. Break, continue, and pass output:
1
3

5. Enumerate output:
Index 0: apple
Index 1: banana
Index 2: cherry

6. Range output:
0
1
2
3
4

7. Dictionary iteration output:
name: Alice
age: 25
city: New York

8. List comprehension output:
[1, 4, 9, 16, 25]

╭  ~/VSCode/Python │ main ?2
```

Conclusion/Summary:

This practical exercise provided a comprehensive overview of **functions** and **loops** in Python, two fundamental concepts that form the backbone of programming. Through hands-on examples, we explored defining and using functions with various argument types, recursion, decorators, and lambda functions. Additionally, we practiced iterating over sequences, controlling loop flow, and leveraging advanced techniques like list comprehensions and dictionary iterations. By organizing the code into two files (`2-3.py` and `3-2.py`), we reinforced modular programming practices, making the code more readable and maintainable. This practical serves as a strong foundation for writing efficient and structured Python programs.

Student Signature & Date	**Marks:**	**Evaluator Signature & Date**