Daryl D. Silva

BSCpE-2B2

Laboratory Exercise No. 3 CH2

Title: Exploring Programming Paradigms

Brief Introduction

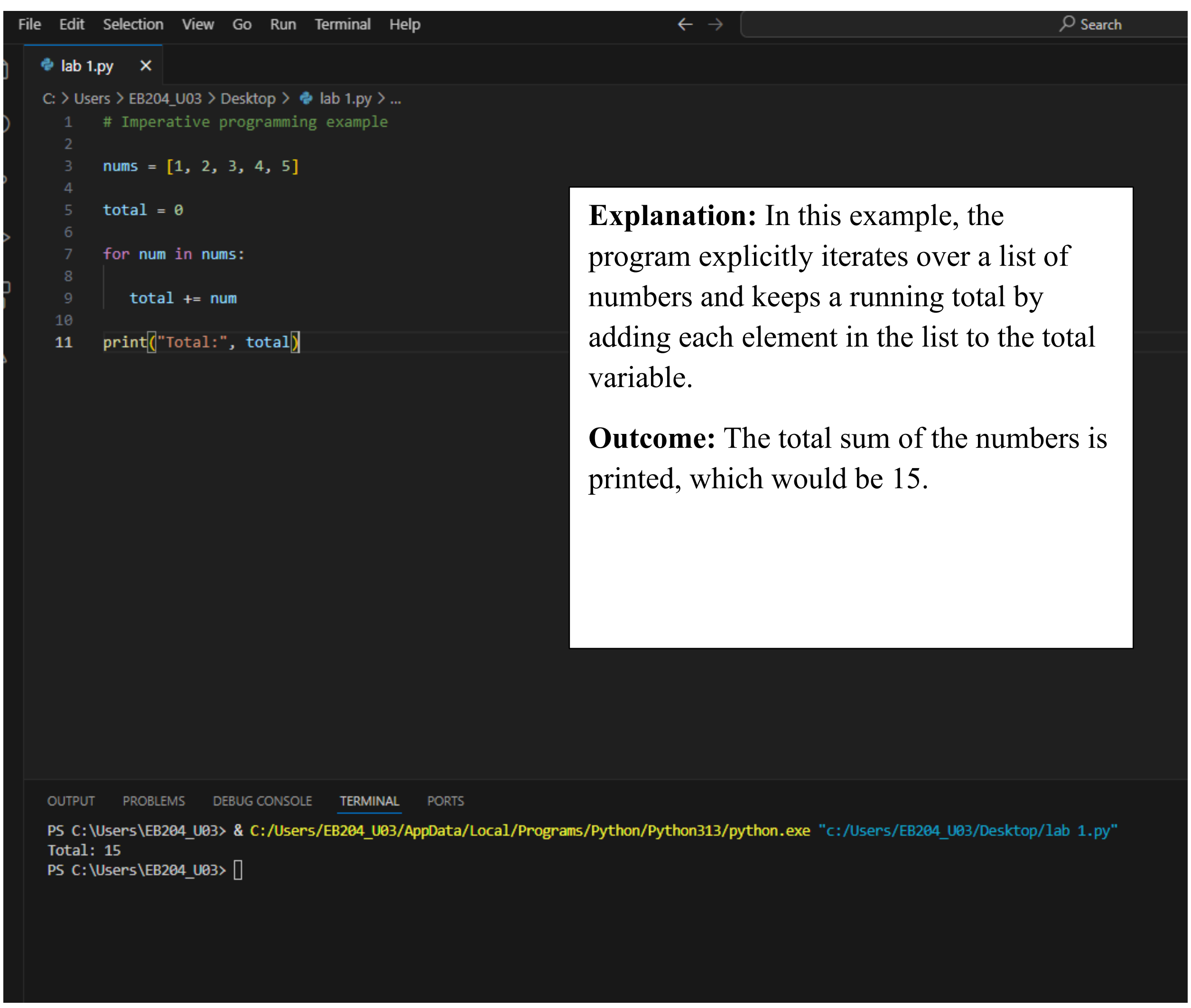
Programming paradigms define the style and structure of writing software programs. This exercise introduces imperative, object-oriented, functional, declarative, event-driven, and concurrent programming paradigms and their applications.

Procedure

1. **Implement imperative programming in Python:**

```
# Imperative programming example

nums = [1, 2, 3, 4, 5]

total = 0

for num in nums:

  total += num

print("Total:", total)
```

result:



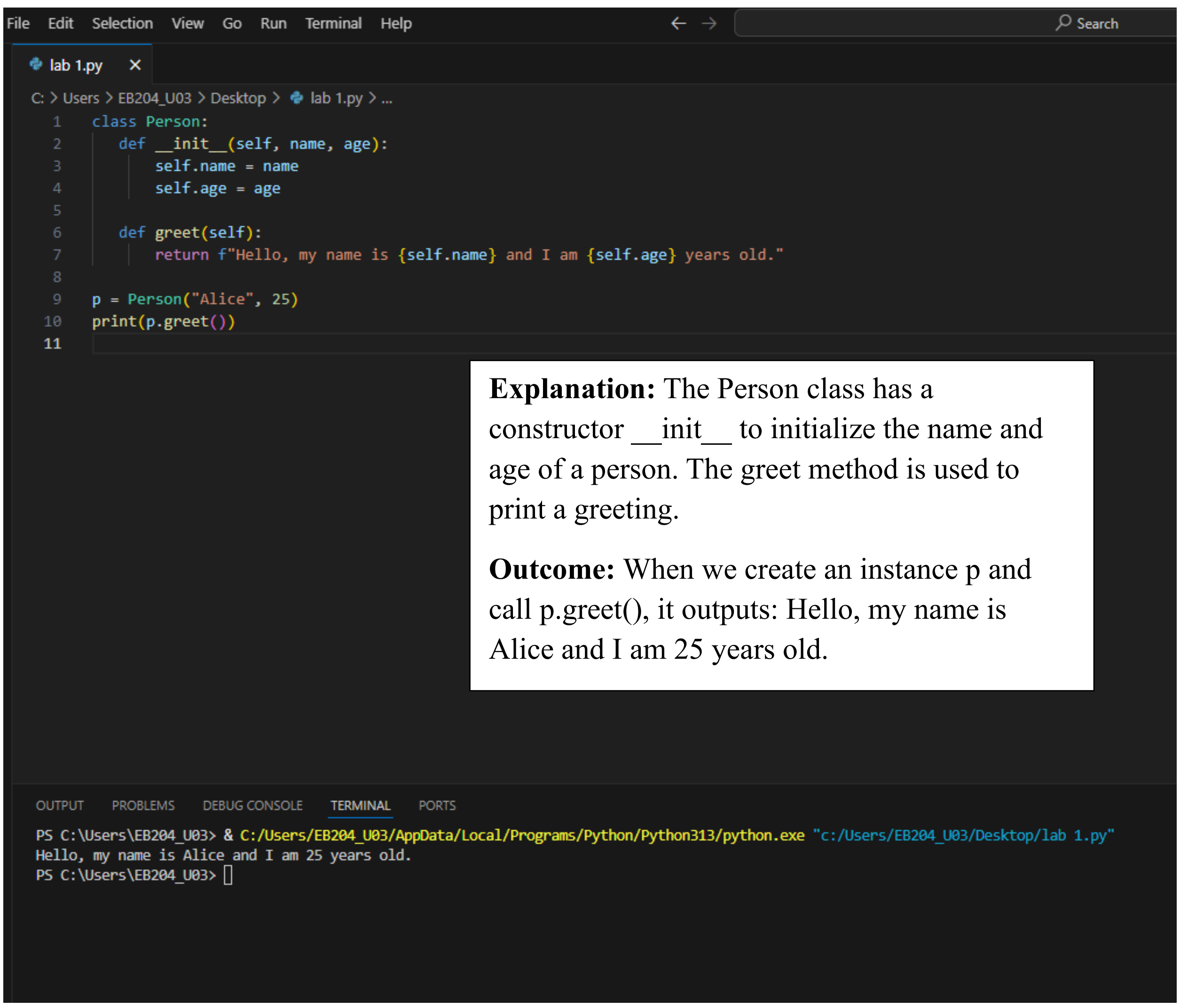
Explanation: In this example, the program explicitly iterates over a list of numbers and keeps a running total by adding each element in the list to the total variable.

Outcome: The total sum of the numbers is printed, which would be 15.

1. **Create a simple object-oriented program:**

```
class Person:
  def __init__(self, name, age):
     self.name = name
     self.age = age

  def greet(self):
     return f"Hello, my name is {self.name} and I am {self.age} years old."

p = Person("Alice", 25)
print(p.greet())
```

result:



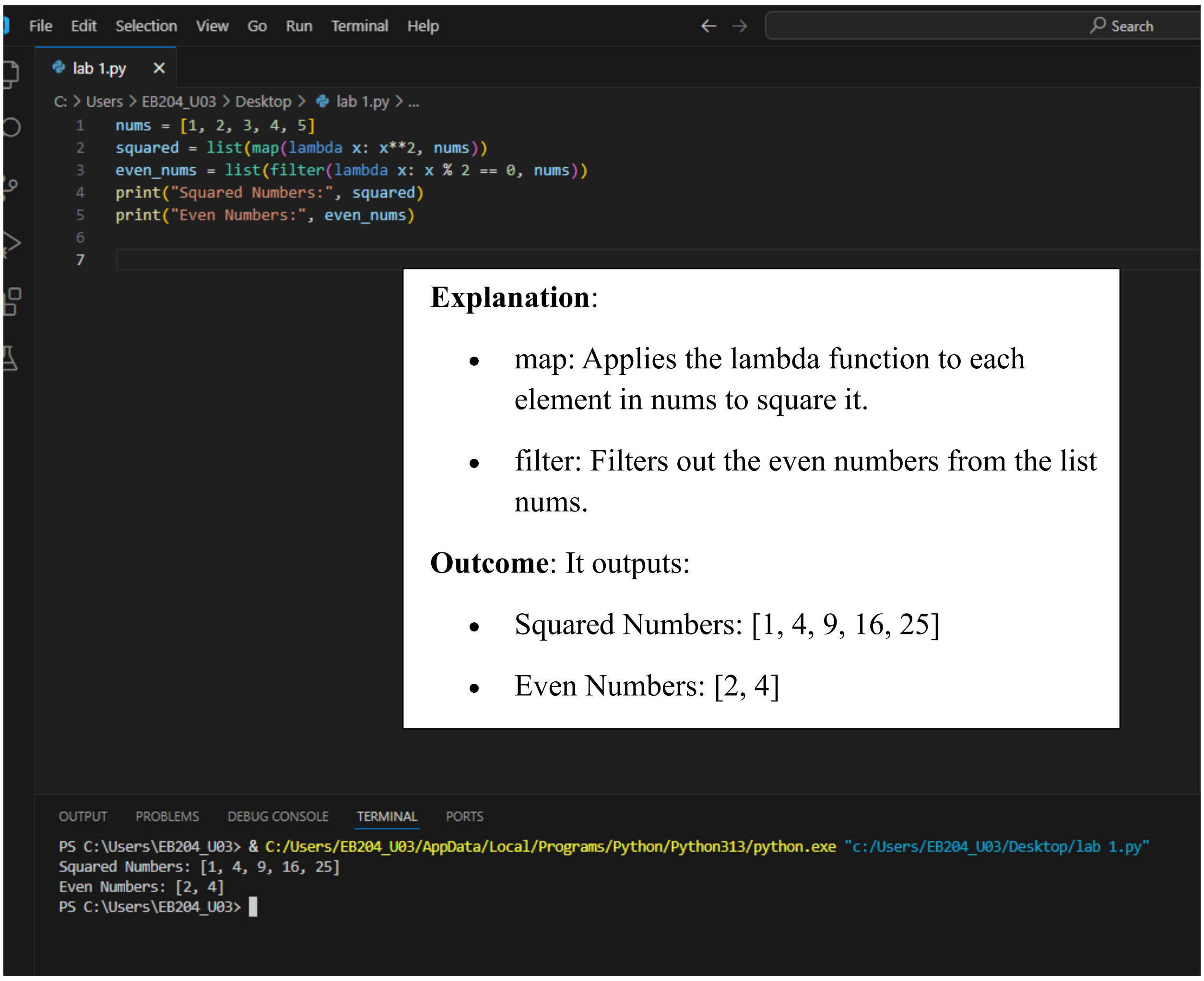
Explanation: The Person class has a constructor __init__ to initialize the name and age of a person. The greet method is used to print a greeting.

Outcome: When we create an instance p and call p.greet(), it outputs: Hello, my name is Alice and I am 25 years old.

1. **Write a functional programming example using Python's map and filter:**

```
nums = [1, 2, 3, 4, 5]

squared = list(map(lambda x: x**2, nums))

even_nums = list(filter(lambda x: x % 2 == 0, nums))

print("Squared Numbers:", squared)

print("Even Numbers:", even_nums)
```

result:



Explanation:

- map: Applies the lambda function to each element in nums to square it.
- filter: Filters out the even numbers from the list nums.

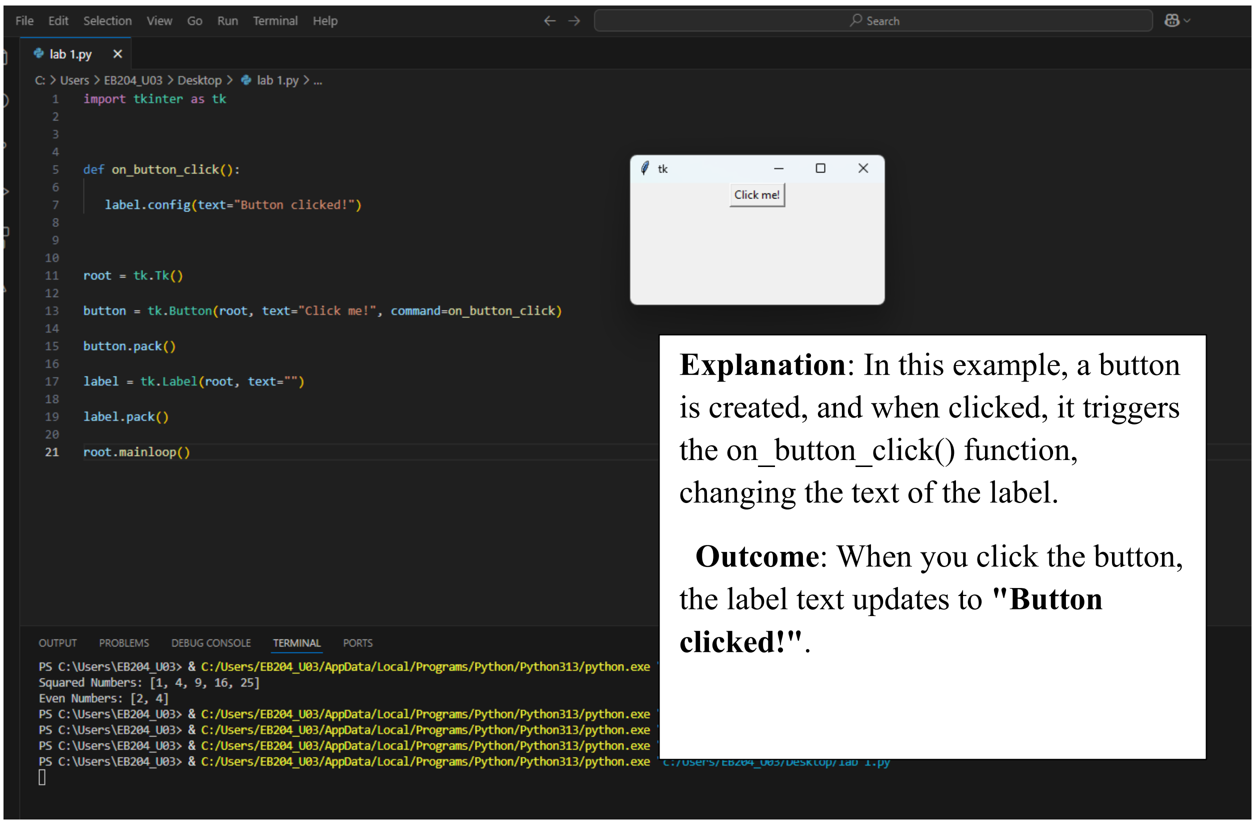
Outcome: It outputs:

- Squared Numbers: [1, 4, 9, 16, 25]
- Even Numbers: [2, 4]

1. **Showcase event-driven programming using Tkinter:**

```
import tkinter as tk


def on_button_click():
  label.config(text="Button clicked!")


root = tk.Tk()

button = tk.Button(root, text="Click me!", command=on_button_click)

button.pack()

label = tk.Label(root, text="")

label.pack()

root.mainloop()
```

result:



Explanation: In this example, a button is created, and when clicked, it triggers the on_button_click() function, changing the text of the label.

Outcome: When you click the button, the label text updates to **"Button clicked!"**.

1. Discuss concurrency with the threading module.

Follow-Up Questions

1. What are the key differences between imperative and declarative programming?

   Answer: **Key Differences Between Imperative and Declarative Programming:**

- **Imperative Programming**:
  - Focuses on *how* to perform tasks step-by-step.
  - Involves specifying the control flow (loops, conditionals) and the sequence of operations.
  - Provides explicit instructions for managing the program state.
  - Example: Using a for loop to sum a list of numbers.
- **Declarative Programming**:
  - Focuses on *what* needs to be done, rather than how.
  - Describes the desired outcome without specifying the control flow.
  - Often uses expressions or declarations rather than statements.
  - Example: Using map or filter to apply a function to a list.

2. In which scenarios would you prefer functional programming?

   Answer: **Scenarios Where You Would Prefer Functional Programming:**

- **Statelessness**: When you need to avoid mutable state and side effects. Functional programming encourages immutability, making it ideal for applications that need predictable behavior without unintended changes in state.

- **Parallelism**: Functional programming is well-suited for parallel execution because functions can be executed independently without worrying about shared state.
- **Complex Transformations**: When you need to perform complex transformations on data (e.g., mapping, filtering, and reducing), functional programming provides a concise, declarative approach to express these operations.
- **Concurrency**: As functional programming avoids side effects, it simplifies writing concurrent code that works in parallel, as there are fewer concerns about shared mutable states.
- **Data Processing**: Functional programming excels in scenarios like data manipulation, ETL pipelines, and transformations, where operations are applied to datasets.

3. How can concurrency improve software performance?

   Answer: **Parallel Execution**: Concurrency allows multiple tasks to be executed simultaneously, making use of multi-core processors, which can significantly speed up I/O-bound or CPU-bound operations.

   Non-blocking I/O: By running multiple tasks concurrently, applications can perform non-blocking I/O operations, enabling more efficient handling of tasks like reading from or writing to databases, files, or network sockets.

   Resource Utilization: Concurrency can lead to better resource utilization by ensuring that while one task is waiting for I/O operations, other tasks can continue executing, avoiding idle CPU time.

   Improved Responsiveness: In user interfaces, concurrency allows the application to remain responsive by running background tasks without freezing the main thread (e.g., loading data or processing while the user interacts with the app).

   Scalability: Concurrency allows systems to handle a higher volume of tasks simultaneously, which is crucial for building scalable systems, especially in web services, cloud computing, and distributed systems.