WEEK 6 – DISCUSSION PROMPT

Activity Content

---

1. What makes a function reusable, and how does its design affect reusability?

Reusability means a function can be used in multiple programs or parts of the same program without modification.

What makes a function reusable:

- **Generality:** The function performs a single, well-defined task instead of being tied to specific data.
- **Parameters:** Uses input parameters instead of hardcoded values.
- **Return values:** Returns results rather than printing them directly.
- **Documentation and naming:** Clear names and comments make it easier for others to understand and reuse.

Design effect:
A well-designed function that follows the **DRY (Don't Repeat Yourself)** principle and **modular design** is easier to test, maintain, and reuse across projects.

---

2. In what scenarios is recursion preferable to iteration, and what are the limitations?

Recursion is preferable when:

- The problem is **naturally hierarchical or self-similar**, such as:
    - Tree traversal (e.g., binary trees)
    - Divide and conquer algorithms (e.g., QuickSort, MergeSort)
    - Mathematical problems (e.g., factorial, Fibonacci)
- Recursive solutions can make the code **cleaner and easier to understand**.

Limitations of recursion:

- **Performance overhead:** Each recursive call adds a new frame to the call stack, consuming more memory.
- **Risk of stack overflow:** Deep recursion can cause runtime errors if the base case isn't reached quickly.
- **Slower execution:** Compared to iteration, recursion can be less efficient unless optimized (e.g., with tail recursion).

---

3. How do decorators improve the modularity or structure of a program?

Decorators allow adding functionality to existing functions or classes **without modifying their code**.

How they improve modularity:

- Promote **code reuse** by separating cross-cutting concerns (e.g., logging, authentication, timing).
- Maintain **clean structure** by keeping the core logic and extra functionality apart.
- Enable **flexible and dynamic behavior**, letting you add or remove features easily.

  Example:

  ```
  def log_decorator(func):
      def wrapper():
          print("Function started")
          func()
          print("Function ended")
      return wrapper


  @log_decorator
  def greet():
      print("Hello, Anusha!")
  ```

This keeps logging logic separate from the greet() function itself.

---

4. What are the advantages and trade-offs of using lambda functions instead of named functions?

Advantages:

- **Compact syntax:** One-liners for simple operations.
- **Useful for short, throwaway functions** used once (e.g., in map(), filter(), or sorting).
- **Improves readability** when the logic is very short and straightforward.

Trade-offs:

- **No name:** Harder to debug or reuse.

- **Limited functionality:** Can only contain a single expression (no statements or annotations).
- **Readability issues:** Overuse can make code harder to understand.

  Example:

  ```
  # Lambda
  square = lambda x: x*x


  # Named function
  def square(x):
      return x*x
  ```

---

5. When and why should a developer measure the execution time of code in a program?

When to measure:

- During **performance testing or optimization**.
- When comparing **different algorithms or data structures**.
- Before deploying a system where **speed or scalability** matters (e.g., web servers, ML pipelines).

Why:

- To **identify bottlenecks** in the code.
- To ensure the program meets **performance requirements**.
- To **validate improvements** after optimization.

  Example using Python's time module:

  ```
  import time
  start = time.time()
  # Code block to test
  end = time.time()
  print("Execution time:", end - start, "seconds")
  ```
