# Clean Code Development

Clean Code Development focuses on writing code that is easy to read, understand and maintain.

1. **Descriptive Variable Names:**
    - **In Code:**

      ```
      self.difficulty_scale_label = Label(self.root, text
      ="Set Difficulty Level:", foreground="black")
      ```

    - **Explanation:**
      Variable names like `difficulty_scale_label` provide clarity about their purpose, making the code more readable and maintainable.

2. **Modularization:**
    - **In Code:**

      ```
      from word_api import WordAPI
      ```

- **Explanation:**
  Importing the
  `WordAPI` class from a separate module promotes modularization, improving code organization and ease of maintenance.

3. **Handling User Input Gracefully:**

- **In Code:**

```
notification_time = simpledialog.askstring("Set Notif
ication Time", "Enter notification time (HH:MM:SS):")
```

- **Explanation:**
  The use of a dialog for user input improves the user experience, and the code handles input gracefully.

4. **Meaningful Function Name:**

- **In Code:**

```
def set_notification_time(self):
```

- **Explanation:**
  The function name
  `set_notification_time` clearly indicates its purpose, enhancing the code's readability.

5. **Error Handling:**

- **In Code:**

```
if notification_time
```

- **Explanation:**
  The code checks if
  `notification_time` exists before proceeding, demonstrating good error handling to prevent unexpected behavior.

# Clean Code Development Cheat Sheet

## 1. Descriptive Naming:

- Uses clear and descriptive names for variables, functions, and classes.
- Prioritises readability over brevity.

## 2. Modularization:

- Organises codes into modules and packages based on functionality.
- Aims for a modular and well-structured project layout.

## 3. User Input Handling:

- Provides clear prompts for user input.
- Implements robust error handling to gracefully manage input errors.

## 4. Meaningful Functions:

- Keeps functions focused on a single responsibility.
- Uses meaningful function names that convey their purpose.

## 5. Error Handling:

- Implements comprehensive error handling.
- Provides informative error messages to assist debugging.

## 6. Single Responsibility Principle (SRP):

- Ensures that each class and function has a single responsibility.
- Avoids functions that perform too many tasks.

## 7. Don't Repeat Yourself (DRY):

- Eliminates duplicated code by creating reusable functions and modules.

## 8. Use Constants:

- Replaces magic numbers or strings with named constants.
- Enhances code readability and maintainability.

## 9. Consistent Formatting:

- Maintains a consistent coding style throughout the project.
- Include clear and organized indentation.

## 10. Avoid Deep Nesting:

- Minimises nested structures to avoid code complexity.
- Aims for clean and readable code.

## 11. Encapsulate Conditional Logic:

- Places complex conditions in well-named functions.
- Improves code readability and maintainability.

## 12. Regular Refactoring:

- Regularly reviews and refactors code to improve its design.
- Prioritises simplicity and maintainability.

## 13. Version Control Best Practices:

- Makes meaningful commits with clear commit messages.
- Uses branches effectively for collaborative development.

## 14. Documentation:

- Provides clear and concise documentation for functions, classes, and modules.
- Keeps documentation up to date.

## 15. Minimize Global Variables:

- Limits the use of global variables to improve code maintainability.
- Prefers passing variables explicitly.