"Heaven's Light is Our Guide"

Rajshahi University of Engineering & Technology

Department of Electrical and Computer Engineering

Software Engineering & Information System Design

ECE 3117

Submitted To	Submitted By
Oishi Joyti Assistant Professor, Dept of ECE	Md. Jonayed Dewan Roll: 2010006

Table of Contents

1. Experiment Number:

1

2. Experiment Name:

Basic documentation using Markdown language

3. Objectives:

- To understand the basic syntax of Markdown.
- To learn how to create structured documentation using Markdown.
- To implement Markdown for experiment documentation.

4. Source Code and Output:

Types of header

Header 1

Header 2

Header 3

Header 4

Header 5

Header 6

Emphasis

This is italic writing using underscore
This is italic writing using star

This is bold writing using dash
This is bold writing using star

This is bold and italic writing using star
This is bold and italic writing using dash

Strike through

~~A line through the text~~

Links

[https://dimikoj.com/]

[https://codeforces.com/]

table

Name	Age	Occupation
Junayed Dewan	30	Software Engineer
Joy Jahan	29	Data Scientist
Shahriaar aabdur Rahman	28	Product Manager

image

Source Code:

```python
def display_message():
    print("HELLO WORLD")

# Call the function
display_message()
```

```c
#include<stdio.h>
int main(){
  printf("My name is Junayed");

  return 0;
}
```

Discussion:

In this experiment, the focus was on creating well-structured documentation using Markdown language. The document demonstrated various Markdown features such as headers, emphasis, tables, images, links, and code blocks.

1. **Headers**: Different levels of headers were used, ranging from `Header 1` to `Header 6`, to organize the content in a hierarchical manner. This is important for readability and structuring documents logically.

2. **Emphasis**: Markdown offers multiple ways to emphasize text. Italics were applied using both underscores (_) and stars (*), while bold text was demonstrated using double underscores (__) and double stars (**). Additionally, bold and italic combinations were shown to highlight key points within the text.

3. **Strike-through**: The example also illustrated how to apply strike-through to text using ~~. This can be helpful in situations where one wants to indicate outdated or irrelevant information without removing it.

4. **Links**: Links to external websites were embedded, showing how Markdown simplifies adding hyperlinks without complex HTML tags. This enhances navigation, making the document interactive and resourceful.

5. **Tables**: The inclusion of a table demonstrated how Markdown can organize data effectively in a tabular format. This is especially useful for presenting structured information like comparisons, datasets, or lists.

6. **Images**: Adding images within Markdown was shown by embedding a link to a remote image. This capability allows for visual content to be integrated seamlessly into documents, enriching the user experience.

7. **Source Code**: The Python code block showcased how Markdown supports syntax highlighting for various programming languages. This feature is useful in technical documentation, allowing code to be easily readable and formatted.

Overall, this experiment demonstrated the versatility and simplicity of Markdown for creating professional and structured documents. It is particularly useful in environments like GitHub, technical blogs, or collaborative documentation, where quick formatting and readability are key. Markdown's lightweight syntax allows for swift adoption while maintaining clarity across different types of content.

1. Experiment No: 2

2. Experiment Name:

Understanding and Implementing Best Coding Practices

3. Objectives:

- To explore the fundamental principles of best coding practices.
- To learn how to structure code for readability, maintainability, and efficiency.

4. Source Code and Description

Below is a sample Python code that demonstrates the application of best coding practices, including different naming conventions. The code calculates the factorial of a number using a recursive function and is well-documented with consistent naming styles.

Source Code

```
# Filename: factorial_calculator.py

# PascalCase for Class Names
class FactorialCalculator:
    """
    A class to calculate the factorial of a given number using different naming
conventions.
    """

    # snake_case for function and variable names
    def calculate_factorial(self, number):
        """
        Recursively calculates the factorial of a given number.

        Args:
            number (int): A non-negative integer whose factorial is to be
calculated.

        Returns:
            int: Factorial of the given number.

        Raises:
            ValueError: If the input number is negative.
        """
        if number < 0:
            raise ValueError("Factorial is not defined for negative numbers.")

        # Base case: factorial of 0 is 1
        if number == 0:
            return 1
        else:
            # Recursive case: factorial(n) = n * factorial(n-1)
```

```
            return number * self.calculate_factorial(number - 1)

    def run_calculator(self):
        """
        Main function to execute the factorial calculation.

        Prompts the user for input and calculates the factorial
        of the provided number using the calculate_factorial function.
        """
        try:
            # camelCase for variable used locally
            userInput = int(input("Enter a non-negative integer: "))

            # Calculate factorial
            result = self.calculate_factorial(userInput)

            # Display the result
            print(f"The factorial of {userInput} is: {result}")

        except ValueError as e:
            print(f"Error: {e}")
        except Exception as e:
            print(f"An unexpected error occurred: {e}")

# ALL_CAPS for constants
MAX_NUMBER = 1000

if __name__ == "__main__":
    calculator = FactorialCalculator()
    calculator.run_calculator()
```

Description of Naming Conventions:

PascalCase: This style capitalizes the first letter of each word without spaces or underscores, typically used for class names. Example: FactorialCalculator.

snake_case: Uses lowercase letters with words separated by underscores, commonly applied to function and variable names in Python. Example: calculate_factorial, run_calculator.

camelCase: Starts with a lowercase letter, with each subsequent word capitalized. It's often used for local variables or properties within a class. Example: userInput.

ALL_CAPS: All letters are capitalized with underscores separating words, typically reserved for constants. Example: MAX_NUMBER.

5. Discussion

Implementing different naming conventions based on the context of the code can enhance clarity and consistency:

PascalCase for Classes: Using PascalCase for class names makes it easy to identify the class in the code. It distinguishes class types from functions and variables, making the structure clearer.

snake_case for Functions and Variables: This is the standard convention in Python for function and variable names, which enhances readability. The consistent use of lowercase letters and underscores provides a clear separation between words, making code easier to understand.

camelCase for Local Variables: While less common in Python, camelCase is sometimes used for variables within a small, localized scope. This convention can help differentiate between class-level attributes (often snake_case) and local variables.

ALL_CAPS for Constants: Using ALL_CAPS for constants immediately indicates that the value should remain unchanged. It's a clear visual signal that the variable holds a fixed value throughout the program.

By following these naming conventions, the code becomes more readable and maintainable, which is particularly important in collaborative environments. Consistent naming helps developers quickly understand the role of a variable or function, reducing the chances of errors and improving overall code quality. Adopting these standards is a key aspect of best coding practices, ensuring that code is clean, well-structured, and accessible to any developer who may interact with it.

1. Experiment No:

3

2. Experiment Name:

Study of Different Git Commands

3. Objectives:

- To understand the purpose and usage of Git commands.
- To learn how to use Git for version control and collaboration.
- To explore different Git commands for managing repositories.

4. Theory:

Git is a distributed version control system that tracks changes in source code during software development. It allows multiple developers to work on the same project without overwriting each other's contributions. Key concepts in Git include repositories, commits, branches, and merges. This experiment focuses on studying different Git commands, their uses, and how they contribute to efficient version control.

Key Git Commands:

1. `git init`: Initializes a new Git repository.
2. `git clone`: Clones an existing repository into a new directory.
3. `git status`: Displays the state of the working directory and the staging area.
4. `git add`: Adds changes to the staging area for the next commit.
5. `git commit`: Records the changes in the repository.
6. `git log`: Shows the commit history of the repository.
7. `git push`: Uploads local repository content to a remote repository.
8. `git pull`: Fetches changes from a remote repository and merges them into the local repository.
9. `git branch`: Lists, creates, or deletes branches in a repository.
10. `git merge`: Merges changes from one branch into another.

5. Procedure:

1. **Initialize a Git repository**:
    - Use `git init` to initialize a new Git repository in the current directory.
    - Example:

```
git init
```

2. **Clone an existing repository**:
    - Use `git clone` to clone a remote repository.
    - Example:

```
git clone https://github.com/username/repository.git
```

3. **Check repository status**:
   - Use `git status` to check the current status of the repository.
   - Example:

```
git status
```

4. **Add files to the staging area**:
   - Use `git add` to add specific files or all changes to the staging area.
   - Example:

```
git add filename.txt
git add .
```

5. **Commit changes**:
   - Use `git commit` to save changes to the repository with a message.
   - Example:

```
git commit -m "Initial commit"
```

6. **View commit history**:
   - Use `git log` to view the commit history.
   - Example:

```
git log
```

7. **Push changes to a remote repository**:
   - Use `git push` to upload changes to a remote repository (like GitHub).
   - Example:

```
git push origin main
```

8. **Pull updates from a remote repository**:

- Use `git pull` to fetch and merge changes from the remote repository.
- Example:

```
git pull origin main
```

9. **Branch management**:

- Use `git branch` to create and list branches.
- Example:

```
git branch new-feature
git checkout new-feature
```

10. **Merge branches**:

- Use `git merge` to merge changes from one branch into another.
- Example:

```
git checkout main
git merge new-feature
```

6. Source Code and Output:

Commands:

```
# Initialize a new repository
git init
# Clone a repository
git clone https://github.com/username/repository.git
# Check status
git status
# Add files
git add .
# Commit changes
git commit -m "Initial commit"
# Push changes
git push origin main
# Pull updates
git pull origin main
# Create a new branch
git branch feature-branch
# Merge branches
git merge feature-branch
### Discussion:
In this experiment, we studied key Git commands essential for managing code.
```

1. **Repository Setup**: We used `git init` to start a new repository and `git clone` to copy an existing one. This allows us to either begin a new project or work with an existing codebase.
2. **Tracking and Staging**: `git status` helps check the current state of our files, and `git add` stages changes for the next commit. This keeps our work organized and ready for version control.
3. **Committing and Sharing**: With `git commit`, we save our changes with a message. `git push` uploads these changes to a remote repository, while `git pull` updates our local copy with changes from others.
4. **Branching**: We used `git branch` to manage different lines of development and `git merge` to combine changes from different branches.

Overall, these commands help us keep track of changes, collaborate with others, and manage different versions of our code efficiently.