Cloud Counselage

GIFT A CAREER FOUNDATION

INTERNSHIP PROGRAM 2023

CODING STANDARDS

Artificial Intelligence

AI Chatbot

Created By:	Snehil Sharma	**Approved By:**	Eesha Kiran Patro
Created On:	25-Aug-2023	**Approved On:**	25-Aug-2023

Page left blank intentionally

INDEX

1 PURPOSE

The Coding Standards are the guidelines for software Developers to create uniform coding habits that eases the reading, checking and maintaining code. The intent of these standards is to define a natural style and consistency, yet leave to the authors, the freedom to practice their craft without unnecessary burden.

The coding standards shall enable the following:

- **Improve Code Quality**: Coding standards ensure that code is written consistently, readably, and maintainable manner. This makes it easier for developers to understand and work with the code, leading to higher-quality software.
- **Increase Efficiency**: By following coding standards, developers can save time by avoiding common mistakes and implementing proven solutions.
- **Facilitate Collaboration**: It creates a common language that all developers can understand and allows teams to collaborate, share code, and communicate effectively.
- **Ensure Compatibility**: It ensures that code is compatible with different platforms, browsers, and OS-device combinations.
- **Reduce Maintenance Costs**: By following established standards, developers can avoid introducing new bugs and make changes to code more quickly and easily.

The coding standards should follow the below best practices:

1. Focus on code readability
2. Enable Commenting
3. Formalising Exception Handling

2 SCOPE

This document describes general software coding standards for code written for python specific to AI Chatbot and shall be implemented while developing the code for the said project.

3 FILE STRUCTURE

The 'File Structure' allow developers to know where files are, when to use specific code, and locate associated results. Not only do file structures streamline productivity, but they also increase code consistency and shareability

3.1 Standard File Conventions

1. **File Naming:**
   - Use descriptive names that indicate the purpose of the file.
   - Separate words with underscores (_) or use CamelCase for readability.
   - For example: **chat.py**, **app.py**, **base.html**, **app.js**, **speechRecognition.js**.
2. **Folder Structure:**
   - Organize files into logical folders like **src**, **static**, **templates**, etc.
   - Keep related files together to enhance maintainability.
3. **Code Files:**
   - Use the **.py** extension for Python code files.
   - Use the **.html** extension for HTML files.
   - Use the **.js** extension for JavaScript files.
   - Use the **.css** extension for CSS files.
4. **Configuration Files:**
   - Use the **.json** extension for JSON configuration files.
   - Name configuration files descriptively, like **index.json**.
5. **Libraries and Dependencies:**
   - Include a **requirements.txt** file listing project dependencies.
   - Specify versions to ensure consistency across environments.
6. **Images and Assets:**
   - Create a separate folder, like **images** or **assets**, to store images and other media.
   - Use lowercase letters and underscores for image filenames, e.g., **logo.png**, **mic.png**.

3.2 Markdown Files

Include a **README.md** file at the project root for an overview and setup instructions.

3.3 Common Conventions

1. **Code Formatting:**
   - Use consistent indentation (e.g., 4 spaces per level) for better readability.
   - Follow PEP 8 guidelines for Python code.
2. **Naming Conventions:**
   - Use descriptive and meaningful names for variables, functions, classes, and files.
   - Use lowercase letters and underscores for Python variable and function names.
   - Use CamelCase for class names.
   - Use lowercase letters and hyphens for HTML, CSS, and JavaScript filenames.
   - Avoid using reserved keywords as identifiers.
3. **Commenting and Documentation:**
   - Provide comments to explain complex logic or sections of code.
   - Include inline documentation for functions, classes, and modules.
   - Use clear and concise comments that add value to the understanding of the code.
4. **Error Handling:**
   - Implement appropriate error handling to catch and handle exceptions.
   - Provide meaningful error messages that help users and developers understand issues.
5. **Version Control:**
   - Use a version control system (e.g., Git) to track changes and collaborate.
   - Commit messages should be descriptive and summarize the changes made.
6. **File Organization:**
   - Organize code files into meaningful folders (e.g., **src**, **static**, **templates**).
   - Keep related files together for easier navigation.
7. **Modularization:**
   - Break down the code into modular components for better maintainability.
   - Separate concerns and functionalities into different modules or classes.
8. **Consistent Styling:**
   - Maintain consistent styling across the entire codebase.
   - Keep the visual appearance of UI elements consistent with the design.
9. **Testing and Validation:**
   - Test the code thoroughly to ensure proper functionality.
   - Validate input data to prevent errors or vulnerabilities.
10. **Code Reusability:**
    - Avoid duplicating code whenever possible.
    - Use functions, classes, and modules to reuse code segments.
11. **Resource Management:**
    - Clean up and release resources (e.g., file handles, network connections) appropriately.
    - Dispose of resources that are no longer needed.
12. **Documentation and README:**
    - Provide a clear README.md with project details, setup instructions, and usage guidelines.
    - Document any third-party libraries, APIs, or dependencies used.

4 FORMATTING CONVENTIONS

These conventions are all about the positions of line breaks, how many characters should go on a line, and everything in between.

4.1 Indentation

- Use consistent indentation of 4 spaces per level.
- Indent code blocks such as loops, conditionals, and function definitions.
- Maintain alignment within code blocks for better readability.

4.2 Using Capitalization to Aid Readability

- Use lowercase letters for variable and function names (e.g., user_input, get_response).
- Use CamelCase for class names (e.g., Chatbox, SpeechRecognition).
- Use UPPERCASE for constants (e.g., API_KEY, MAX_ATTEMPTS).

4.3 Formatting Single Statements

- Place single-line control statements and loops on the same line if they fit comfortably.
- Example: `if condition: print("True")`

4.4 Formatting Declarations

- Separate imports, function and class declarations, and main program logic with two blank lines.
- Use a single space on both sides of assignment operators (=) and comparison operators (==, <, >, etc.).
- Example:

```
import openai
def get_response(input_text):
    user_msg = detected_text + " " + input_text
```

4.5 Formatting Multi-line Statements

- For function or method calls with multiple arguments, break lines after commas.
- For lists, dictionaries, and sets, break lines after each item and maintain consistent indentation.
- Example:

```
response = openai.ChatCompletion.create(
    model = "gpt-3.5-turbo",
    messages = [
        {"role": "system", "content":
  system_msg},
        {"role": "user", "content": user_msg },
    ],
)
```

5 NAMING CONVENTIONS

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier-for example, whether it's a constant, package, or class-which can be helpful in understanding the code.

Naming conventions result in improvements in terms of "four Cs": communication, code integration, consistency, and clarity. The idea is that "code should explain itself"

Naming convention is applicable to constants, variables, functions, modules, packages and files. In object-oriented languages, it's applicable to classes, objects, methods and instance variables.

With regard to scope, global names may have a different convention compared to local names; such as, Pascal Case for globals: Optind rather than optind in gawk. Private or protected attributes may be named differently: _secret or __secret rather than secret. Some may want to distinguish local variables from method arguments using prefixes.

For naming conventions, please refer to
https://www.pluralsight.com/blog/software-development/programming-naming-conventions-explained

6 SCOPING CONVENTIONS

Scoping is generally divided into two types:

6.1 Lexical/Static Scoping

A variable in this scope always refers to its top-level environment. This characteristic of the program text has nothing to do with the call stack at runtime. Static scoping makes it considerably easier to write modular code because a programmer can find out the scope by looking at the code.

6.2 Dynamic Scoping

With dynamic scope, a global identifier directs to the identifier associated with the most current environment and is unusual in modern languages. In technical terms, each identifier has a global stack of bindings, and the most current binding is explored for events of the identifier. In another way, the Compiler successfully explores the current block and all calling functions first in dynamic scoping.

7 COMPILE ERRORS & WARNINGS

7.1 Errors

Errors report problems that make it impossible to compile your program.

When developing programs there are three types of error that can occur:

- **Syntax error** occurs when the code given does not follow the syntax rules of the programming language. A program cannot run if it has syntax errors. Examples include:
  - misspelling a statement, e.g. writing pint instead of print
  - using a variable before it has been declared
  - missing brackets, e.g. opening a bracket, but not closing it

  Any such errors must be fixed first. A good integrated development environment (IDE) usually points out any syntax errors to the programmer.

- **Logic error** is an error in the way a program works. The program can run but does not do what it is expected to do. Logic errors can be caused by the programmer:
  - incorrectly using logical operators, e.g. expecting a program to stop when the value of a variable reaches 5, but using <5 instead of <=5
  - incorrectly using Boolean operators
  - unintentionally creating a situation where an infinite loop may occur
  - incorrectly using brackets in calculations
  - unintentionally using the same variable name at different points in the program for different purposes
  - using incorrect program design

- **Runtime error** is an error that takes place during the running of a program. An example is writing a program that tries to access the sixth item in an array that only contains five items. A runtime error is likely to crash the program.

7.2 Warnings

Warnings report other unusual conditions in your code that may indicate a danger points where you should check to make sure that your program really does what you intend**.**

Compiler warnings are useful, but they are highly unreliable. In addition, they are no substitute for language subsetting.

Please refer this article for understanding working with compiler warnings.

8 ENFORCING CODING STANDARD

Please refer this article to enforce coding standard across different tools and platform.