**Coding Standards (Python)**  
**v2.0**  
**October 26, 2024**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Version Description

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Who | Comment |
| v0.0 | 10/06/24 | RM and LA | Document Created |
| V1.0 | 10/14/24 | RM and LA | Python Documentation Included |
| v2.0 | 10/26/24 | RM | Add Error Handling Section |
| v3.0 |  |  |  |

# Contents of this Document

* General
* Naming Conventions
* Commenting Code
* Parenthesis
* Constants
* Line Spacing
* Braces
* Declarations
* Error Handling
* Testing

# General

Our project’s functionality will include many services which will be needed for several Classes to ensure efficient implementation. Our scope will include the use of Python, an Object-Oriented Language. Coding standards will follow normal Python syntax and will make use of appropriate whitespace for enhanced readability.

Goals for code guidelines:  
1. All source code should be easily readable and understandable.

2. The resulting documentation should be easy to maintain.

3. Code changes should be relatively easy with no need for complete code refactoring.

# Naming Conventions

**Class Names:** Concrete classes should use natural descriptive names, begin with a capital letter with capitalization of the first letter of every subsequent word. For example: DataBaseManager  
  
**Function Names:** Function names should begin with a lowercase letter with each new subsequent word separated by an underscore. This is commonly known as snake case. All member functions should have “self” as the first parameter. If it is not a member function, then “self” will not be required.  
**Example:**  
class UserManager:  
 def login(self):  
 def logout(self):

def non\_member\_function(parameter\_1, parameter\_2):

# Commenting Code

Classes: Each class should have a concise docstring using triple double quotes above it. The docstring must describe the class's role and include @requires to specify preconditions and @ensures to outline postconditions. Focus on the purpose of the class and its expected behavior. Annotations help clarify class-level requirements without listing individual attributes.

Functions: Each function should have a brief docstring that describes its purpose and lists the parameters with their types. Include @requires for preconditions and @ensures for postconditions. For functions that return a value, specify the return type. Use single-line comments for brief notes, and multi-line comments for more detailed explanations when necessary. Keep descriptions concise while ensuring clarity on function behavior.

**Example:**

"""

Represents an Animal with basic attributes.

@requires Valid input for name and age

@ensures Creates an Animal instance with the given name and age attributes

"""

class Animal:

"""

Constructs all the necessary attributes for the Animal object.

name: str - The name of the animal

age: str - The age of the animal

@requires name and age should be non-empty strings

@ensures Initializes the animal instance with the provided name and age

"""

def \_\_init\_\_(self, name: str, age: str):

self.name = name

self.age = age

# Parenthesis

Parenthesis should always be used to make use of conditionals more obvious if necessary.

**Example:**  
if x + y:

NOTE: This example is really easy to interpret, so no parenthesis is needed.

if ((x\*\*2 \* (variable // value)) % (17 – variable\_two):

NOTE: This example is more complex, which is why we need the use of parenthesis to make the order of statements more obvious.

# Constants

Constants should be all uppercase with a comment #CONSTANT VALUE indicating the variable will be used as a constant.

**Example:**

#CONSTANT VALUE

DAYS = 7

# Line Spacing

Use blank lines to separate different logical blocks of code, such as function definitions, loops, conditional statements, and imports. Use the default tab size of 4 for loops, conditionals, functions and classes. Blank lines should also be used between logical blocks of code. This will ensure good readability. Place a blank line before function definitions, a blank line before and after class definitions, and use blank lines to separate logical sections within a function, like between variable declarations and processing logic, or before returning a result.

**Example:**

import math

class Circle:

def \_\_init\_\_(self, radius):

self.radius = max(0, radius)

def area(self):

return math.pi \* self.radius \*\* 2

# Declarations

Per Python syntax, all variables must be initialized upon creation. Any indeterminate variable should receive a default value and a comment stating the reason for the value.

**Example:**

# float(‘inf’) is a default value to be used for the min variable since it is the largest value to start

minimum = float(‘inf”)

# Error Handling

Error handling is an essential part of ensuring the robustness of any application. In our project, we will make use of Python's built-in exception handling mechanisms, such as try-except blocks, to handle predictable runtime errors. The goal is to provide clear, user-friendly error messages and avoid crashes during unexpected situations, all while maintaining clean and readable code.

**Key Guidelines:**

1. **Use try-except blocks** around code that can potentially raise exceptions. When possible, handle specific exceptions (e.g., ValueError, KeyError) rather than generic ones to ensure clarity and precision.
2. **Log meaningful error messages** that can help with debugging while being concise enough for users or developers to understand what went wrong. This can be done using flash() to display errors to users in a web application context or print() for debugging.
3. **Always ensure resources are properly released.** For example, use finally to close database connections or clean up temporary files to prevent resource leaks.
4. **Use flash() for web application errors** where the user needs to be notified, such as form submission errors or validation issues. For example, when a user tries to create a quiz beyond the allowed limit, flash an error message.

Example:

def create\_quiz(self):

try:

user\_email = session['email']

if self.has\_reached\_quiz\_limit(user\_email):

flash("You have reached the maximum number of quizzes allowed.", "error")

return redirect(url\_for('create\_quiz\_route'))

except KeyError:

flash("Session expired. Please log in again.", "error")

return redirect(url\_for('login'))

except Exception as e:

flash(f"An unexpected error occurred: {str(e)}", "error")

return redirect(url\_for('home'))

# Testing

All test cases must follow a structured format to ensure consistency, readability, and maintainability. Each test should be clear about its purpose, preconditions, and expected outcomes.

General Guidelines

* **@requires**: Specifies any preconditions that need to be met before the test can be run (e.g., the user must be signed in).
* **@ensures**: Describes the postconditions, i.e., what the system should achieve or return after the test runs successfully.

Structure for Test Functions

Each test should have:

* A clear **docstring** that uses @requires to define the preconditions and @ensures to describe the expected outcome.
* Error handling using try-except blocks to manage test failures gracefully.
* Meaningful test assertions to validate correct behavior.

Example:  
"""

@requires The user is not signed in.

@ensures The index route renders the homepage.

"""

def test\_index():

"""Test the index route to ensure it works without a signed-in user."""

with app.test\_client() as client:

try:

response = client.get('/')

assert response.status\_code == 200

except Exception as e:

print(f"Test failed with error: {str(e)}")