Python Coding Style Guidelines

This document outlines the coding style guidelines that will be used to grade your final project for this course. Your coding style counts for 10% of your final project grade, and we will use the following criteria to evaluate it:

Abstraction (2%)

Functions should be properly abstracted and not too long. This means that your functions should be designed to perform a single task and be of reasonable length (i.e., not too long or too short). Functions that are too long or perform multiple tasks can be difficult to read and debug, so we expect you to use proper abstraction in your code.

Correct example:

```python
def calculate_average(numbers):
    total = sum(numbers)
    count = len(numbers)
    return total / count
```

- This function is abstracted and performs a single task: calculating the average of a list of numbers.
- The function is also not too long and is easy to read.

Incorrect example:

```python
def do_something():
    # 50 lines of code
    # multiple tasks performed
    # no clear abstraction
    pass
```

- This function is too long and performs multiple tasks.
- The function is not properly abstracted and is difficult to read and debug.

Comments (2%)

Your code should be properly documented using comments. This includes function and class definitions, as well as any code blocks that are not self-explanatory. Comments should be concise, clear, and relevant. They should explain what the code does, why it does it, and how it does it.

Correct example:

```python
def calculate_average(numbers):
    """Calculate the average of a list of numbers.

    Args:
        numbers (list of int or float): A list of numbers.

    Returns:
        float: The average of the numbers.

    """
    total = sum(numbers)
    count = len(numbers)
    return total / count
```

- This function is well-documented using a docstring.
- The docstring includes a description of what the function does, what arguments it takes, and what it returns.

Incorrect example:

```python
def calculate_average(numbers):
    # divide sum of numbers by length
    total = sum(numbers)
    count = len(numbers)
    return total / count
```

- This function lacks documentation and does not include any comments explaining what it does.

Tests (2%)

Your code should include test cases that demonstrate the functionality of your program. This includes unit tests that verify that your functions and classes work as intended, as well as any other tests that demonstrate the correctness and robustness of your code.

Correct example:

```python
import unittest

class TestCalculateAverage(unittest.TestCase):
    def test_average(self):
        self.assertEqual(calculate_average([1, 2, 3]), 2.0)
        self.assertEqual(calculate_average([2, 4, 6]), 4.0)
        self.assertEqual(calculate_average([-1, 0, 1]), 0.0)
```

- This code includes unit tests that verify that the `calculate_average` function works as intended.
- The tests are well-written and demonstrate the correctness of the function.

Incorrect example:

```python
import unittest

class TestCalculateAverage(unittest.TestCase):
    def test_average(self):
        assert calculate_average([1, 2, 3]) == 2.0
        assert calculate_average([2, 4, 6]) == 4.0
        assert calculate_average([-1, 0, 1]) == 0.0
```

- These tests are not written using the
- built-in `assert` statement or the `assertEqual` method from the `unittest` module.
    - The tests are not well-documented and do not clearly demonstrate the correctness of the function.

Variable/Function Naming (1%)

Your variable and function names should be descriptive, concise, and follow standard Python naming conventions. Variable names should be in lowercase with words separated by underscores (`snake_case`). Function names should also be in lowercase with words separated by underscores. Class names should be in PascalCase (also known as camel case with initial capitalization).

Correct example:

```python
def calculate_average(numbers_list):
    """Calculate the average of a list of numbers.

    Args:
        numbers_list (list of int or float): A list of numbers.

    Returns:
        float: The average of the numbers.

    """
    total = sum(numbers_list)
    count = len(numbers_list)
    return total / count
```

- This code uses descriptive variable and function names that follow standard Python naming conventions.
- The names are concise and clearly indicate what the variables and functions do.

Incorrect example:

```python
def avg(num_lst):
    total = sum(num_lst)
    count = len(num_lst)
    return total / count
```

- This code uses non-descriptive variable and function names that do not follow standard Python naming conventions.
- The names are not concise and do not clearly indicate what the variables and functions do.

Global Variables (1%)

Global variables should be avoided in your code whenever possible. Global variables can lead to unexpected behavior, and can make it difficult to debug and test your code. Instead of using global variables, you should use function arguments and return values to pass data between functions.

Correct example:

```python
def calculate_average(numbers_list):
    """Calculate the average of a list of numbers.

    Args:
        numbers_list (list of int or float): A list of numbers.

    Returns:
        float: The average of the numbers.

    """
    total = sum(numbers_list)
    count = len(numbers_list)
    return total / count
```

- This code does not use any global variables.

Incorrect example:

```python
numbers_list = [1, 2, 3, 4, 5]

def calculate_average():
    total = sum(numbers_list)
    count = len(numbers_list)
    return total / count
```

- This code uses a global variable (`numbers_list`) instead of passing the list as an argument to the function.
- The use of a global variable makes the function less modular and more difficult to test and debug.

Code Formatting (2%)

Proper code formatting is important for readability and maintainability of your code. Here are ten common formatting issues to avoid:

1. Missing whitespace around operators:
    - Correct: `total = x + y`, Incorrect: `total=x+y`
2. Inconsistent indentation:
    - All lines within a block should be indented the same amount.
3. Inconsistent spacing after commas:
    - Correct: `values = [1, 2, 3]`, Incorrect: `values=[1,2,3]`
4. Line length:
    - Keep lines less than 79 characters whenever possible.
5. Incorrect use of semicolons:
    - Avoid using semicolons to separate statements in Python.
6. Incorrect spacing around function arguments:
    - Correct: `print("Hello, world!")`, Incorrect: `print( "Hello, world!" )`
7. Inconsistent capitalization:
    - Follow consistent capitalization conventions for variable and function names.
8. Mixed use of tabs and spaces for indentation:
    - Choose one or the other for consistency.
9. Unnecessary parentheses:
    - Only use parentheses where necessary for clarity or to modify operator precedence.
10. Trailing whitespace:
    - Remove any trailing whitespace at the end of lines.
      Sure, here are 10 more formatting tips to help improve code readability:
11. Use descriptive variable names:
    - Variable names should be descriptive and convey their purpose.

12. Use blank lines to separate logical sections:
    - Use blank lines to group related statements and separate them from unrelated statements.

13. Use consistent line spacing:
    - Use a consistent number of blank lines between sections of code to improve readability.

14. Use lowercase for function names:

    - Function names should be in lowercase with words separated by underscores.

15. Use single quotes for string literals:

    - Use single quotes for string literals instead of double quotes to improve consistency and readability.

16. Use whitespace around assignment operators:

    - Use whitespace before and after assignment operators to improve readability.

17. Use parentheses to group complex expressions:

    - Use parentheses to group complex expressions and improve readability.

18. Use consistent naming conventions:

    - Use consistent naming conventions throughout the code for variables, functions, and classes.

19. Use docstrings to document code:

    - Use docstrings to provide clear and concise documentation for functions, classes, and modules.

20. Avoid using backslashes for line continuation:

    - Use parentheses or backslashes at the end of a line for line continuation only when necessary.

Correct example:

```python
def calculate_average(numbers_list):
    """Calculate the average of a list of numbers.

    Args:
        numbers_list (list of int or float): A list of numbers.

    Returns:
        float: The average of the numbers.

    """
    total = sum(numbers_list)
    count = len(numbers_list)
    return total / count


values = [1, 2, 3]
average = calculate_average(values)
print(f"The average of {values} is {average}.")
```

- This code follows proper formatting guidelines, with consistent spacing, indentation, and capitalization.
- The line length is within the recommended limit of 79 characters.

Incorrect example:

```
def calculate_average(numbers_list):
    """Calculate the average of a list of numbers.
    Args: numbers_list(list of int or float): A list of numbers.
    Returns: float, the average of the numbers.
    """
    total=sum(numbers_list)
    count=len(numbers_list)
    return total/count

values=[1,2,3]
average=calculate_average(values)
print(f"the average of {values} is {average}.")
```

- This code violates several formatting guidelines, with inconsistent spacing, indentation, and capitalization.
- The line length is also longer than recommended, which can make the code less readable.