AI Assistant Coding

Name: CH. Arush Nandhan

HTNO: 2303A51880

Batch-14

Date: 05-02-2026

Assignment - 9.4

Problem statements with solution:

Task 1: Auto-Generating Function Documentation in a Shared

Codebase

Scenario

You have joined a development team where several utility functions are already implemented, but the code lacks proper documentation. New team members are struggling to understand how these functions should be used.

Task Description

You are given a Python script containing multiple functions without any docstrings.

Using an AI-assisted coding tool:

• Ask the AI to automatically generate Google-style function docstrings for each function

• Each docstring should include:

o A brief description of the function

o Parameters with data types

o Return values

o At least one example usage (if applicable)

Experiment with different prompting styles (zero-shot or context-based)

to observe quality differences.

Expected Outcome

• A Python script with well-structured Google-style docstrings

• Docstrings that clearly explain function behavior and usage

• Improved readability and usability of the codebase

—---

Prompt: Generate Google-style Python docstrings for each function in the code, including a brief description, parameters with types, return values, and at least one example usage if applicable.

Ai coding > assignment 9.4 > task1.py > ...

```python
import random
import string

def generate_password(length):
    chars = string.ascii_letters + string.digits + string.punctuation
    return ''.join(random.choice(chars) for _ in range(length))

def is_strong_password(password):
    has_upper = any(c.isupper() for c in password)
    has_lower = any(c.islower() for c in password)
    has_digit = any(c.isdigit() for c in password)
    has_special = any(c in string.punctuation for c in password)
    return len(password) >= 8 and has_upper and has_lower and has_digit and has_special

def hash_password(password):
    value = 0
    for char in password:
        value = (value * 31 + ord(char)) % 1000000007
    return value

def main():
    pwd = generate_password(12)
    strength = is_strong_password(pwd)
    hashed = hash_password(pwd)
    print("Password:", pwd)
    print("Strong:", strength)
    print("Hash:", hashed)

if __name__ == "__main__":
    main()

```

Code after Adding Docstrings:

```python
import random
import string

def generate_password(length):
    """Generate a random password with specified length.

    Creates a password containing uppercase letters, lowercase letters,
    digits, and special characters randomly selected from the available
    character set.

    Args:
        length (int): The desired length of the generated password.

    Returns:
        str: A randomly generated password of the specified length.

    Example:
        >>> password = generate_password(12)
        >>> len(password)
        12
    """
    chars = string.ascii_letters + string.digits + string.punctuation
    return ''.join(random.choice(chars) for _ in range(length))

def is_strong_password(password):
    """Check if a password meets strong security criteria.

    A password is considered strong if it contains at least 8 characters
    and includes at least one uppercase letter, one lowercase letter,
    one digit, and one special character.

    Args:
        password (str): The password string to validate.

    Returns:
        bool: True if the password is strong, False otherwise.

    Example:
        >>> is_strong_password('Weak')
        False
        >>> is_strong_password('StrongP@ss123')
        True
    """
    has_upper = any(c.isupper() for c in password)
    has_lower = any(c.islower() for c in password)
    has_digit = any(c.isdigit() for c in password)
    has_special = any(c in string.punctuation for c in password)
    return len(password) >= 8 and has_upper and has_lower and has_digit and
has_special
def hash_password(password):
    """Generate a hash value for a password using polynomial rolling hash.

    Applies a polynomial rolling hash algorithm to convert a password
    into a numeric hash value. Uses a base of 31 and modulo 1000000007
    to keep the result manageable.

    Args:
        password (str): The password string to hash.

    Returns:
        int: A numeric hash value of the password.

    Example:
        >>> hash_val = hash_password('test123')
        >>> isinstance(hash_val, int)
        True
    """
    value = 0
    for char in password:
        value = (value * 31 + ord(char)) % 1000000007
    return value

def main():
    """Main function to demonstrate password generation and validation.

    Generates a random password, checks its strength, and computes its hash.
    Outputs the generated password, its strength status, and hash value.

    Returns:
        None

    Example:
        >>> main()
        Password: <generated_password>
        Strong: True
        Hash: <hash_value>
    """
    pwd = generate_password(12)
    strength = is_strong_password(pwd)
    hashed = hash_password(pwd)
    print("Password:", pwd)
    print("Strong:", strength)
    print("Hash:", hashed)

if __name__ == "__main__":
    main()
```

Task 2: Enhancing Readability Through AI-Generated Inline

Comments

Scenario

A Python program contains complex logic that works correctly but is difficult to understand at first glance. Future maintainers may find it hard to debug or extend this code.

Task Description

You are provided with a Python script containing:

• Loops

• Conditional logic

• Algorithms (such as Fibonacci sequence, sorting, or searching)

Use AI assistance to:

• Automatically insert inline comments only for complex or non-obvious logic

• Avoid commenting on trivial or self-explanatory syntax

The goal is to improve clarity without cluttering the code.

Expected Outcome

• A Python script with concise, meaningful inline comments

• Comments that explain why the logic exists, not what Python syntax does

• Noticeable improvement in code readability

—-------------------------------

Prompt: Automatically insert inline comments only for complex or non-obvious logic in the code, avoiding trivial or self-explanatory syntax, to improve clarity without cluttering the code.

```python
from functools import lru_cache

@lru_cache(None)
def x(a, b):
    return a + 1 if b == 0 else x(a - 1, x(a, b - 1)) if a > 0 else x(1, b - 1)

def y(n):
    return (n ^ (n >> 1)) & 0xffffffff

def z(s):
    r = 0
    for i, c in enumerate(s):
        r ^= y(ord(c) + x(i % 2, ord(c) % 3))
        r = ((r << 5) | (r >> 27)) & 0xffffffff
    return r

def main():
    import sys
    data = sys.stdin.read().strip()
    print(hex(z(data)))

if __name__ == "__main__":
    main()
```

```python
from functools import lru_cache

@lru_cache(None)
def x(a, b):
    # Ackermann function: recursive computation with base cases and caching
    # Base case: when b=0, return a+1
    # Recursive case: computes nested function calls or reduces parameters
    return a + 1 if b == 0 else x(a - 1, x(a, b - 1)) if a > 0 else x(1, b - 1)

def y(n):
    # Gray code conversion: XOR with right-shifted version to produce reflected binary code
    # Mask to 32-bit unsigned integer to maintain consistent bit width
    return (n ^ (n >> 1)) & 0xffffffff

def z(s):
    r = 0
    for i, c in enumerate(s):
        # Transform character using Gray code based on Ackermann function result
        # XOR accumulates the encoded values
        r ^= y(ord(c) + x(i % 2, ord(c) % 3))
        # Rotate left by 5 bits: shift left 5, OR with bits shifted out on right
        # Mask to keep result in 32-bit range
        r = ((r << 5) | (r >> 27)) & 0xffffffff
    return r

def main():
    import sys
    data = sys.stdin.read().strip()
    print(hex(z(data)))

if __name__ == "__main__":
    main()
```

Task 3: Generating Module-Level Documentation for a Python

Package

Scenario

Your team is preparing a Python module to be shared internally (or uploaded to a repository). Anyone opening the file should immediately understand its purpose and structure.

Task Description

Provide a complete Python module to an AI tool and instruct it to automatically generate a module-level docstring at the top of the file that includes:

- The purpose of the module
- Required libraries or dependencies

• A short example of how the module can be used

Focus on clarity and professional tone.

Expected Outcome

• A well-written multi-line module-level docstring

• Clear overview of what the module does and how to use it

• Documentation suitable for real-world projects or repositories

—---------------------

Prompt: Generate a module-level docstring at the top of the file that describes the module’s purpose, required libraries or dependencies, key functions and classes, and includes a short usage example."

Ai coding > assignment 9.4 > task3.py > ...

```python
import math
import random

class Vector:
    def __init__(self, *coords):
        self._c = tuple(float(x) for x in coords)

    def __add__(self, other):
        return Vector(*(a + b for a, b in zip(self._c, other._c)))

    def __sub__(self, other):
        return Vector(*(a - b for a, b in zip(self._c, other._c)))

    def scale(self, k):
        return Vector(*(k * x for x in self._c))

    def dot(self, other):
        return sum(a * b for a, b in zip(self._c, other._c))

    def magnitude(self):
        return math.sqrt(self.dot(self))

    def normalize(self):
        m = self.magnitude()
        return self.scale(1 / m) if m else self

    def __repr__(self):
        return f"Vector{self._c}"

def random_vector(dim, low=-1.0, high=1.0):
    return Vector(*(random.uniform(low, high) for _ in range(dim)))

def distance(a, b):
    return (a - b).magnitude()
```

Documented Code:

```python
"""Vector mathematics module for multi-dimensional geometric operations.

This module provides a Vector class and utility functions for performing
mathematical operations on vectors in arbitrary dimensions. It supports
vector arithmetic (addition, subtraction), scaling, dot product computation,
magnitude calculation, normalization, and distance measurements between vectors.

Dependencies:
    - math: For mathematical operations (sqrt)
    - random: For generating random vectors

Key Components:
    Vector: A class representing a multi-dimensional vector with support for
            arithmetic operations, dot products, magnitude, and normalization.
    random_vector: Function to generate random vectors in a specified dimension
                   with optional bounds.
    distance: Function to calculate the Euclidean distance between two vectors.

Example Usage:
    >>> v1 = Vector(3, 4)
    >>> v2 = Vector(1, 2)
    >>> result = v1 + v2
    >>> print(result)
    Vector(4.0, 6.0)
    >>> mag = v1.magnitude()
    >>> print(mag)
    5.0
    >>> dist = distance(v1, v2)
    >>> print(dist)
    2.23606797749979
    >>> rand_vec = random_vector(3, low=0, high=1)
    >>> print(rand_vec.normalize())
    Vector(...)
"""

import math
import random

class Vector:
    def __init__(self, *coords):
        self._c = tuple(float(x) for x in coords)

    def __add__(self, other):
        return Vector(*(a + b for a, b in zip(self._c, other._c)))

    def __sub__(self, other):
        return Vector(*(a - b for a, b in zip(self._c, other._c)))

    def scale(self, k):
        return Vector(*(k * x for x in self._c))

    def dot(self, other):
        return sum(a * b for a, b in zip(self._c, other._c))

    def magnitude(self):
        return math.sqrt(self.dot(self))

    def normalize(self):
        m = self.magnitude()
        return self.scale(1 / m) if m else self

    def __repr__(self):
        return f"Vector{self._c}"

def random_vector(dim, low=-1.0, high=1.0):
    return Vector(*(random.uniform(low, high) for _ in range(dim)))

def distance(a, b):
    return (a - b).magnitude()
```

Task 4: Converting Developer Comments into Structured Docstrings

Scenario

In a legacy project, developers have written long explanatory comments inside functions instead of proper docstrings. The team now wants to standardize documentation.

Task Description

You are given a Python script where functions contain detailed inline comments explaining their logic.

Use AI to:

• Automatically convert these comments into structured Google-style or NumPy-style docstrings

• Preserve the original meaning and intent of the comments

• Remove redundant inline comments after conversion

Expected Outcome

• Functions with clean, standardized docstrings

• Reduced clutter inside function bodies

• Improved consistency across the codebase

—--

Prompt: Automatically convert existing comments into structured Google-style or NumPy-style docstrings, preserving their original meaning and intent, and remove any redundant inline comments after conversion.

```python
def analyze_numbers(nums):
    # Alright, so here’s what’s happening in this function, but explained in
    # the kind of ridiculously long-winded and verbose way that makes it
    # a little harder to read at first glance. The function’s purpose, in a
    # nutshell, is to take a list of numeric values (we’re going to assume
    # they are valid numbers and that the list is non-empty because we are
    # ignoring all error checking on purpose, which is itself a subtle
    # design decision that affects how the rest of the code behaves) and
    # produce a set of statistics about them. But instead of using Python’s
    # built-in sum(), min(), max(), or statistics.mean(), we are going to
    # track everything manually so that multiple calculations are being done
    # at once in a single pass. This creates a mental load for someone trying
    # to follow the function, because the state is being mutated in multiple
    # ways as we loop over the data.
    #
    # We start by initializing several variables. `total` is set to 0 so that
    # it can accumulate the sum of all numbers. `smallest` and `largest` are
    # initialized to the first number in the list, which conveniently avoids
    # having to think about infinities or other sentinels but also implicitly
    # assumes the list has at least one element. `index` will serve both as a
    # counter for the number of items (to compute the average later) and also
    # as a factor in a slightly arbitrary derived metric that will make the
    # function look more complicated. Finally, `derived` is initialized to 0
    # and will accumulate a running combination of value, position, and current
    # minimum in a way that doesn’t have a clear real-world interpretation
    # but increases the cognitive overhead of reading this function.

    total = 0
    smallest = largest = nums[0]
    index = 0
    derived = 0

    # Now we enter the main loop, iterating over each number in the list.
    # During this loop, multiple state variables are updated simultaneously.
    # This means that the meaning of each variable at any given moment is
    # dependent on all previous iterations, which requires the reader to hold
    # a lot in memory to understand what the final result will be.
    for n in nums:
        # incrementally add the current number to the running total
        total += n

        # update smallest if the current number is smaller than the previous smallest
        if n < smallest:
            smallest = n

        # update largest if the current number is larger than the previous largest
        if n > largest:
            largest = n

        # compute a derived value that combines the current number, the running
        # index, and the current minimum, purely for the sake of creating a
        # non-trivial computation that isn’t immediately obvious
        derived += (n - smallest) * (index + 1)

        # increment the index for both average calculation and derived metric
        index += 1

    # after the loop ends, compute the average by dividing the total by the
    # number of elements (index now equals the length of the list)
    average = total / index

    # finally, return a dictionary with all the computed statistics so that
    # the caller can easily access them using descriptive keys rather than
    # remembering positional return values
    return {
        "sum": total,
        "avg": average,
        "min": smallest,
        "max": largest,
        "derived": derived
    }


def main():
    nums = list(map(float, input("Enter numbers separated by space: ").split()))
    print(analyze_numbers(nums))


if __name__ == "__main__":
    main()
```

```python
def analyze_numbers(nums):
    """Compute comprehensive statistics for a list of numbers in a single pass.

    This function calculates multiple statistics (sum, average, min, max, and a
    derived metric) by iterating through the list once instead of using multiple
    passes. All state variables are updated simultaneously during iteration.

    The function initializes smallest and largest to the first element, avoiding
    sentinels but implicitly assuming a non-empty list. A derived metric combines
    each value's offset from minimum with its position for additional analysis.

    Args:
        nums (list): A non-empty list of numeric values. Function assumes valid
                     numbers and performs no error checking.

    Returns:
        dict: Dictionary containing computed statistics:
            - 'sum' (numeric): Total sum of all numbers
            - 'avg' (float): Arithmetic mean of the numbers
            - 'min' (numeric): Smallest value in the list
            - 'max' (numeric): Largest value in the list
            - 'derived' (numeric): Cumulative metric combining each value's offset
                                   from minimum with its position index

    Example:
        >>> analyze_numbers([1, 2, 3, 4, 5])
        {'sum': 15, 'avg': 3.0, 'min': 1, 'max': 5, 'derived': 20}
        >>> analyze_numbers([10.0, 20.0])
        {'sum': 30.0, 'avg': 15.0, 'min': 10.0, 'max': 20.0, 'derived': 10.0}
    """
    total = 0
    smallest = largest = nums[0]
    index = 0
    derived = 0

    for n in nums:
        total += n

        if n < smallest:
            smallest = n

        if n > largest:
            largest = n

        derived += (n - smallest) * (index + 1)

        index += 1

    average = total / index

    return {
        "sum": total,
        "avg": average,
        "min": smallest,
        "max": largest,
        "derived": derived
    }


def main():
    """Main entry point for the number analysis tool.

    Prompts the user to enter space-separated numbers and displays
    comprehensive statistics including sum, average, min, max, and derived metrics.
    """
    nums = list(map(float, input("Enter numbers separated by space: ").split()))
    print(analyze_numbers(nums))


if __name__ == "__main__":
    main()
```

Task 5: Building a Mini Automatic Documentation Generator

Scenario

Your team wants a simple internal tool that helps developers start documenting new Python files quickly, without writing documentation from scratch.

Task Description

Design a small Python utility that:

- Reads a given .py file
- Automatically detects:
  - o Functions
  - o Classes
- Inserts placeholder Google-style docstrings for each detected function or class

AI tools may be used to assist in generating or refining this utility.

Note: The goal is documentation scaffolding, not perfect documentation.

Expected Outcome

- A working Python script that processes another .py file
- Automatically inserted placeholder docstrings
- Clear demonstration of how AI can assist in documentation automation

Note: Report should be submitted a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots

—---------------------------------

Prompt: Design a small Python utility that reads a .py file, detects all functions and classes, and inserts placeholder Google-style docstrings for each, serving as documentation scaffolding rather than complete documentation. AI tools may be used to assist.

```python
"""Automatic Google-style docstring injector for Python files."""

import ast
import sys
import os


class DocstringInjector(ast.NodeVisitor):
    """Find functions and classes without docstrings."""

    def __init__(self):
        self.missing = []

    def visit_FunctionDef(self, node):
        if not ast.get_docstring(node):
            self.missing.append((node.lineno, node.name, 'func', node))
        self.generic_visit(node)

    def visit_ClassDef(self, node):
        if not ast.get_docstring(node):
            self.missing.append((node.lineno, node.name, 'class',
node))  self.generic_visit(node)


def get_indent(line):
    """Get indentation of a line."""
    return line[:len(line) - len(line.lstrip())]


def generate_func_docstring(node, indent):
    """Generate function docstring."""
    args = [arg.arg for arg in node.args.args]
    args_str = ", ".join(args) if args else ""

    doc = f'{indent}"""\n'
    doc += f'{indent}{node.name}({args_str}).\n'
    doc += f'{indent}\n'
    doc += f'{indent}[Brief description].\n'

    if args:
        doc += f'{indent}\n'
        doc += f'{indent}Args:\n'
        for arg in args:
            doc += f'{indent}    {arg} ([type]): [Description].\n'

    doc += f'{indent}\n'
    doc += f'{indent}Returns:\n'
    doc += f'{indent}    [type]: [Description].\n'
    doc += f'{indent}"""\n'

    return doc


def generate_class_docstring(node, indent):
    """Generate class docstring."""
    doc = f'{indent}"""\n'
    doc += f'{indent}{node.name}.\n'
    doc += f'{indent}\n'
    doc += f'{indent}[Brief description].\n'
    doc += f'{indent}\n'
    doc += f'{indent}Attributes:\n'
    doc += f'{indent}    [attr] ([type]): [Description].\n'
    doc += f'{indent}"""\n'

    return doc
```

```python
    doc += f'{indent}Attributes:\n'
    doc += f'{indent}    [attr] ([type]): [Description].\n'
    doc += f'{indent}"""\n'

    return doc


def process_file(filepath, output=None):
    """Process Python file and inject docstrings."""

    if not os.path.exists(filepath):
        print(f"Error: File not found: {filepath}")
        return

    if not filepath.endswith('.py'):
        print("Error: Must be a .py file")
        return

    # Read file
    with open(filepath, 'r', encoding='utf-8') as f:
        lines = f.readlines()

    # Parse and find missing docstrings
    try:
        tree = ast.parse(''.join(lines))
    except SyntaxError as e:
        print(f"Syntax error: {e}")
        return

    visitor = DocstringInjector()
    visitor.visit(tree)

    # Sort by line number (reverse) to insert from bottom up
    missing = sorted(visitor.missing, key=lambda x: x[0], reverse=True)

    # Insert docstrings
    added = {'func': 0, 'class': 0}

    for line_num, name, node_type, node in missing:
        indent = get_indent(lines[line_num - 1])

        if node_type == 'class':
            doc = generate_class_docstring(node, indent)
            added['class'] += 1
        else:
            doc = generate_func_docstring(node, indent)
            added['func'] += 1

        lines.insert(line_num, doc)

    # Save result
    save_path = output or filepath
    with open(save_path, 'w', encoding='utf-8') as f:
        f.writelines(lines)

    print(f"✓ Processed: {filepath}")
    print(f"  Functions: {added['func']}")
    print(f"  Classes: {added['class']}")
    print(f"✓ Saved to: {save_path}")


if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python task5.py <file.py> [output.py]")
        sys.exit(1)

    input_file = sys.argv[1]
    output_file = sys.argv[2] if len(sys.argv) > 2 else None

    process_file(input_file, output_file)
```