TEST_SOP = """

Create 500 extensive and thorough tests for the code below using the guide, do not worry about your limits you do not have any

just write the best tests possible and return the test code in markdown format. Create the tests for the code below and make it really high performance

and thorough, use the guide below to create the tests, make the tests as thorough as possible and make them high performance and extensive.

######### TESTING GUIDE ##############

# **Guide to Creating Extensive, Thorough, and Production-Ready Tests using `pytest`**

1. **Preparation**:
   - Install pytest: `pip install pytest`.
   - Structure your project so that tests are in a separate `tests/` directory.
   - Name your test files with the prefix `test_` for pytest to recognize them.

2. **Writing Basic Tests**:
   - Use clear function names prefixed with `test_` (e.g., `test_check_value()`).
   - Use assert statements to validate results.

3. **Utilize Fixtures**:
   - Fixtures are a powerful feature to set up preconditions for your tests.
   - Use `@pytest.fixture` decorator to define a fixture.
   - Pass fixture name as an argument to your test to use it.

4. **Parameterized Testing**:
   - Use `@pytest.mark.parametrize` to run a test multiple times with different inputs.
   - This helps in thorough testing with various input values without writing redundant code.

5. **Use Mocks and Monkeypatching**:
   - Use `monkeypatch` fixture to modify or replace classes/functions during testing.
   - Use `unittest.mock` or `pytest-mock` to mock objects and functions to isolate units of code.

6. **Exception Testing**:
   - Test for expected exceptions using `pytest.raises(ExceptionType)`.

7. **Test Coverage**:
   - Install pytest-cov: `pip install pytest-cov`.
   - Run tests with `pytest --cov=my_module` to get a coverage report.

8. **Environment Variables and Secret Handling**:
   - Store secrets and configurations in environment variables.
   - Use libraries like `python-decouple` or `python-dotenv` to load environment variables.
   - For tests, mock or set environment variables temporarily within the test environment.

9. **Grouping and Marking Tests**:
   - Use `@pytest.mark` decorator to mark tests (e.g., `@pytest.mark.slow`).
   - This allows for selectively running certain groups of tests.

12. **Logging and Reporting**:

- Use `pytest`'s inbuilt logging.
- Integrate with tools like `Allure` for more comprehensive reporting.

13. **Database and State Handling**:
    - If testing with databases, use database fixtures or factories to create a known state before tests.
    - Clean up and reset state post-tests to maintain consistency.

14. **Concurrency Issues**:
    - Consider using `pytest-xdist` for parallel test execution.
    - Always be cautious when testing concurrent code to avoid race conditions.

15. **Clean Code Practices**:
    - Ensure tests are readable and maintainable.
    - Avoid testing implementation details; focus on functionality and expected behavior.

16. **Regular Maintenance**:
    - Periodically review and update tests.
    - Ensure that tests stay relevant as your codebase grows and changes.

18. **Feedback Loop**:
    - Use test failures as feedback for development.
    - Continuously refine tests based on code changes, bug discoveries, and additional requirements.

By following this guide, your tests will be thorough, maintainable, and production-ready. Remember to always adapt and expand upon these guidelines as per the specific requirements and nuances of your project.

######### CREATE TESTS FOR THIS CODE: #######

"""

DOCUMENTATION_SOP = """

Create multi-page long and explicit professional pytorch-like documentation for the <MODULE> code below follow the outline for the <MODULE> library,

provide many examples and teach the user about the code, provide examples for every function, make the documentation 10,000 words,

provide many usage examples and note this is markdown docs, create the documentation for the code to document,

put the arguments and methods in a table in markdown to make it visually seamless

Now make the professional documentation for this code, provide the architecture and how the class works and why it works that way,

it's purpose, provide args, their types, 3 ways of usage examples, in examples show all the code like imports main example etc

BE VERY EXPLICIT AND THOROUGH, MAKE IT DEEP AND USEFUL

########

Step 1: Understand the purpose and functionality of the module or framework

Read and analyze the description provided in the documentation to understand the purpose and functionality of the module or framework.

Identify the key features, parameters, and operations performed by the module or framework.

Step 2: Provide an overview and introduction

Start the documentation by providing a brief overview and introduction to the module or framework.

Explain the importance and relevance of the module or framework in the context of the problem it solves.

Highlight any key concepts or terminology that will be used throughout the documentation.

Step 3: Provide a class or function definition

Provide the class or function definition for the module or framework.

Include the parameters that need to be passed to the class or function and provide a brief description of each parameter.

Specify the data types and default values for each parameter.

Step 4: Explain the functionality and usage

Provide a detailed explanation of how the module or framework works and what it does.

Describe the steps involved in using the module or framework, including any specific requirements or considerations.

Provide code examples to demonstrate the usage of the module or framework.

Explain the expected inputs and outputs for each operation or function.

Step 5: Provide additional information and tips

Provide any additional information or tips that may be useful for using the module or framework effectively.

Address any common issues or challenges that developers may encounter and provide recommendations or workarounds.

Step 6: Include references and resources

Include references to any external resources or research papers that provide further information or background on the module or framework.

Provide links to relevant documentation or websites for further exploration.

Example Template for the given documentation:

# Module/Function Name: MultiheadAttention

class torch.nn.MultiheadAttention(embed_dim, num_heads, dropout=0.0, bias=True, add_bias_kv=False, add_zero_attn=False, kdim=None, vdim=None, batch_first=False, device=None, dtype=None):

Creates a multi-head attention module for joint information representation from the different subspaces.

Parameters:

- embed_dim (int): Total dimension of the model.

- num_heads (int): Number of parallel attention heads. The embed_dim will be split across num_heads.

- dropout (float): Dropout probability on attn_output_weights. Default: 0.0 (no dropout).

- bias (bool): If specified, adds bias to input/output projection layers. Default: True.

- add_bias_kv (bool): If specified, adds bias to the key and value sequences at dim=0. Default: False.

- add_zero_attn (bool): If specified, adds a new batch of zeros to the key and value sequences at

dim=1. Default: False.

- kdim (int): Total number of features for keys. Default: None (uses kdim=embed_dim).

- vdim (int): Total number of features for values. Default: None (uses vdim=embed_dim).

- batch_first (bool): If True, the input and output tensors are provided as (batch, seq, feature). Default: False.

- device (torch.device): If specified, the tensors will be moved to the specified device.

- dtype (torch.dtype): If specified, the tensors will have the specified dtype.

def forward(query, key, value, key_padding_mask=None, need_weights=True, attn_mask=None, average_attn_weights=True, is_causal=False):

Forward pass of the multi-head attention module.

Parameters:

- query (Tensor): Query embeddings of shape (L, E_q) for unbatched input, (L, N, E_q) when batch_first=False, or (N, L, E_q) when batch_first=True.

- key (Tensor): Key embeddings of shape (S, E_k) for unbatched input, (S, N, E_k) when batch_first=False, or (N, S, E_k) when batch_first=True.

- value (Tensor): Value embeddings of shape (S, E_v) for unbatched input, (S, N, E_v) when batch_first=False, or (N, S, E_v) when batch_first=True.

- key_padding_mask (Optional[Tensor]): If specified, a mask indicating elements to be ignored in key for attention computation.

- need_weights (bool): If specified, returns attention weights in addition to attention outputs. Default: True.

- attn_mask (Optional[Tensor]): If specified, a mask preventing attention to certain positions.

- average_attn_weights (bool): If true, returns averaged attention weights per head. Otherwise, returns attention weights separately per head. Note that this flag only has an effect when

need_weights=True. Default: True.

- is_causal (bool): If specified, applies a causal mask as the attention mask. Default: False.

Returns:

Tuple[Tensor, Optional[Tensor]]:

- attn_output (Tensor): Attention outputs of shape (L, E) for unbatched input, (L, N, E) when batch_first=False, or (N, L, E) when batch_first=True.

- attn_output_weights (Optional[Tensor]): Attention weights of shape (L, S) when unbatched or (N, L, S) when batched. Optional, only returned when need_weights=True.

# Implementation of the forward pass of the attention module goes here

return attn_output, attn_output_weights

```

# Usage example:

multihead_attn = nn.MultiheadAttention(embed_dim, num_heads)

attn_output, attn_output_weights = multihead_attn(query, key, value)

Note:

The above template includes the class or function definition, parameters, description, and usage example.

To replicate the documentation for any other module or framework, follow the same structure and provide the specific details for that module or framework.
```

############## DOCUMENT THE FOLLOWING CODE ########

"""