V.HRK SRI HARSHA VIGNESH

2303A52304

BATCH-42

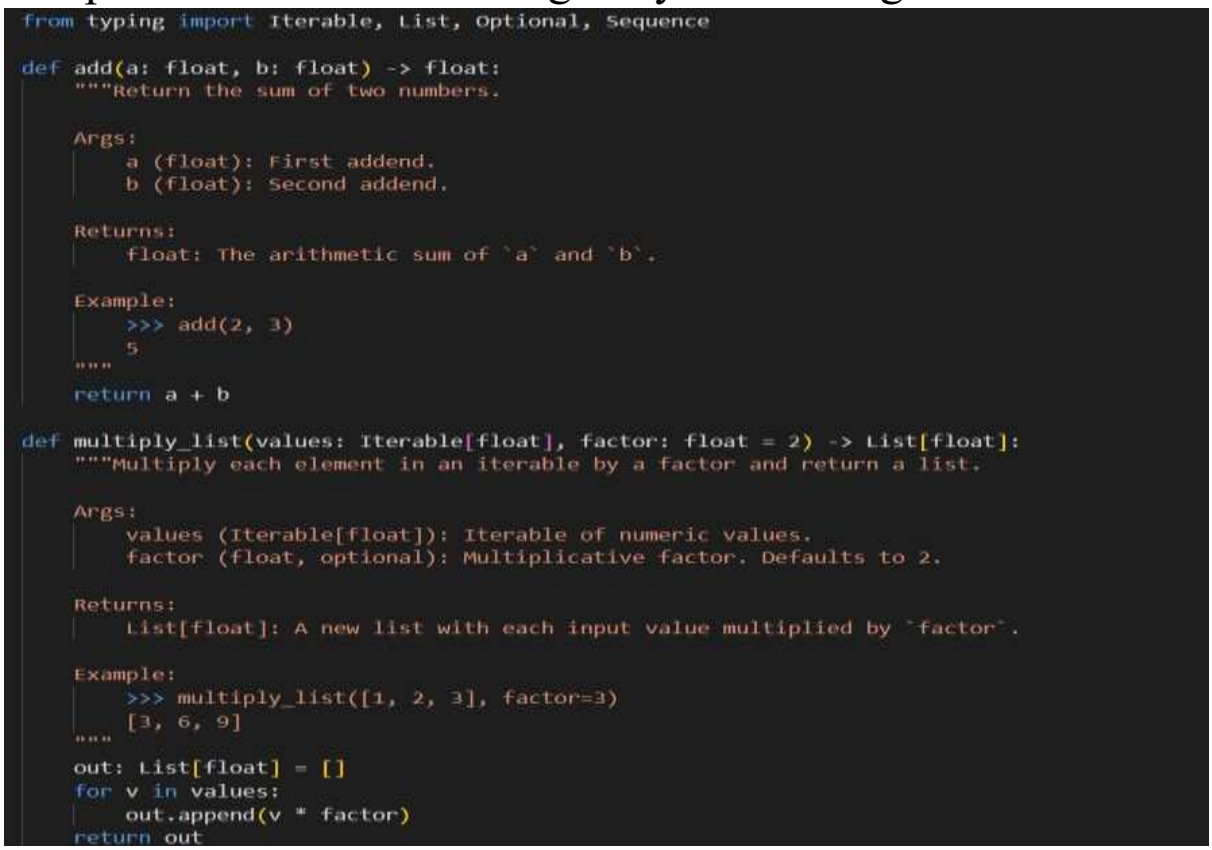
TASK-1: Auto-Generating Function Documentation in a Shared

Codebase

PROMPT:

Insert Google-style docstrings for every top-level function in this Python file that lacks one. Each docstring must include: a one-line summary, Args (with types), Returns (with type), Raises (if applicable), and at least one short Example. Do not change code behavior—only add docstrings. Return the full updated file.

Completed version with Google-style docstrings:



```python
from typing import Iterable, List, Optional, Sequence

def add(a: float, b: float) -> float:
    """Return the sum of two numbers.

    Args:
        a (float): First addend.
        b (float): Second addend.

    Returns:
        float: The arithmetic sum of `a` and `b`.

    Example:
        >>> add(2, 3)
        5
    """
    return a + b

def multiply_list(values: Iterable[float], factor: float = 2) -> List[float]:
    """Multiply each element in an iterable by a factor and return a list.

    Args:
        values (Iterable[float]): Iterable of numeric values.
        factor (float, optional): Multiplicative factor. Defaults to 2.

    Returns:
        List[float]: A new list with each input value multiplied by `factor`.

    Example:
        >>> multiply_list([1, 2, 3], factor=3)
        [3, 6, 9]
    """
    out: List[float] = []
    for v in values:
        out.append(v * factor)
    return out
```

```python
def find_max(values: Sequence[float]) -> Optional[float]:
    """Find the maximum value in a sequence.

    Args:
        values (Sequence[float]): A non-empty sequence of comparable numbers.

    Returns:
        Optional[float]: The largest value, or `None` if the sequence is empty.

    Example:
        >>> find_max([1, 7, 3])
        7
        >>> find_max([])
        None
    """
    if not values:
        return None
    m = values[0]
    for v in values:
        if v > m:
            m = v
    return m

def is_palindrome(s: str) -> bool:
    """Return True if the given string is a palindrome (alphanumeric, case-insensitive).

    Non-alphanumeric characters are ignored and comparison is case-insensitive.

    Args:
        s (str): Input string to test.

    Returns:
        bool: True if `s` is a palindrome after normalization, otherwise False.

    Example:
```

```python
    Example:
        >>> is_palindrome("A man, a plan, a canal: Panama")
        True
        >>> is_palindrome("hello")
        False
    """
    s2 = ''.join(ch.lower() for ch in s if ch.isalnum())
    return s2 == s2[::-1]
```

JUSTIFICATION:

- Speeds onboarding: new team members understand function purpose and usage immediately.
- Makes contracts explicit: documents parameter types, return values, and error conditions so callers know expectations.

- Improves developer UX: IDEs and editor tooltips show readable summaries and examples, reducing context switches.
- Enables automated docs and tooling: consistent docstrings are parsed by Sphinx, linters, and typechecking tools.
- Increases maintainability and review quality: clear docs reduce misunderstandings during refactors and code reviews.
- Low risk, high ROI: only adds documentation (no behavior changes) while delivering outsized longterm productivity gains.

TASK-2:Enhancing Readability Through AI-Generated Inline comments Prompt :

```text
Insert concise inline comments into this Python file. Only add comments for complex or non-obvious logic (algorithms, loop invariants, pivot/partition choices, performance/space trade-offs). Avoid commenting trivial syntax or restating the code. Explain why the code works or why choices were made, not how Python syntax works. Preserve behavior and formatting; return the full updated file.
```

```python
def fibonacci(n: int) -> int:
    # Use an iterative O(n) algorithm with O(1) extra memory.
    # Tuple-swap keeps only the last two values, avoiding list allocations.
    if n < 0:
        raise ValueError("n must be non-negative")
    a, b = 0, 1
    for _ in range(n):
        # update to next Fibonacci number; tuple swap is atomic and concise
        a, b = b, a + b
    return a


def quicksort(arr: list) -> list:
    # Return a new sorted list using an in-place quicksort helper.
    # We pick the middle element as pivot to reduce likelihood of worst-case
    # on already-sorted or reverse-sorted inputs.
    def _quicksort(lo: int, hi: int):
        if lo >= hi:
            return
        pivot = arr[(lo + hi) // 2]
        i, j = lo, hi
        # Hoare partitioning: moves pointers inward and swaps out-of-place items.
        # This style often performs fewer swaps than Lomuto and handles duplicates well.
        while i <= j:
            # Move left pointer until an element >= pivot is found
            while arr[i] < pivot:
                i += 1
            # Move right pointer until an element <= pivot is found
            while arr[j] > pivot:
                j -= 1
            if i <= j:
                # Elements at i and j are on the wrong sides; swap them.
                arr[i], arr[j] = arr[j], arr[i]
                i += 1
                j -= 1
        # Recur on partitions; note ranges because i and j crossed.
        _quicksort(lo, j)
        _quicksort(i, hi)

    _quicksort(0, len(arr) - 1)
    return arr


def binary_search(arr: list, target) -> int:
    # Classic binary search returning index or -1 if not found.
    # Maintain invariant: search range is lo..hi inclusive.
    lo, hi = 0, len(arr) - 1
    while lo <= hi:
        # mid computed to avoid potential overflow in other languages;
        # also keeps consistent partitioning when lo and hi are large.
        mid = lo + (hi - lo) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            # target must be strictly to the right
            lo = mid + 1
        else:
            # target must be strictly to the left
            hi = mid - 1
    return -1


# Example usage (for quick manual testing):
if __name__ == "__main__":
    print("fib(10) =", fibonacci(10))
    data = [3, 6, 1, 8, 4, 5]
    print("sorted:", quicksort(data.copy()))
    print("index of 8:", binary_search(sorted(data), 8))
```

JUSTIFICATION:

- Targets only non-obvious parts (algorithms, invariants, trade-offs) so comments help comprehension without clutter.

- "Why-focused" comments guide future maintainers on design decisions and edge cases, speeding debugging and safe refactors.

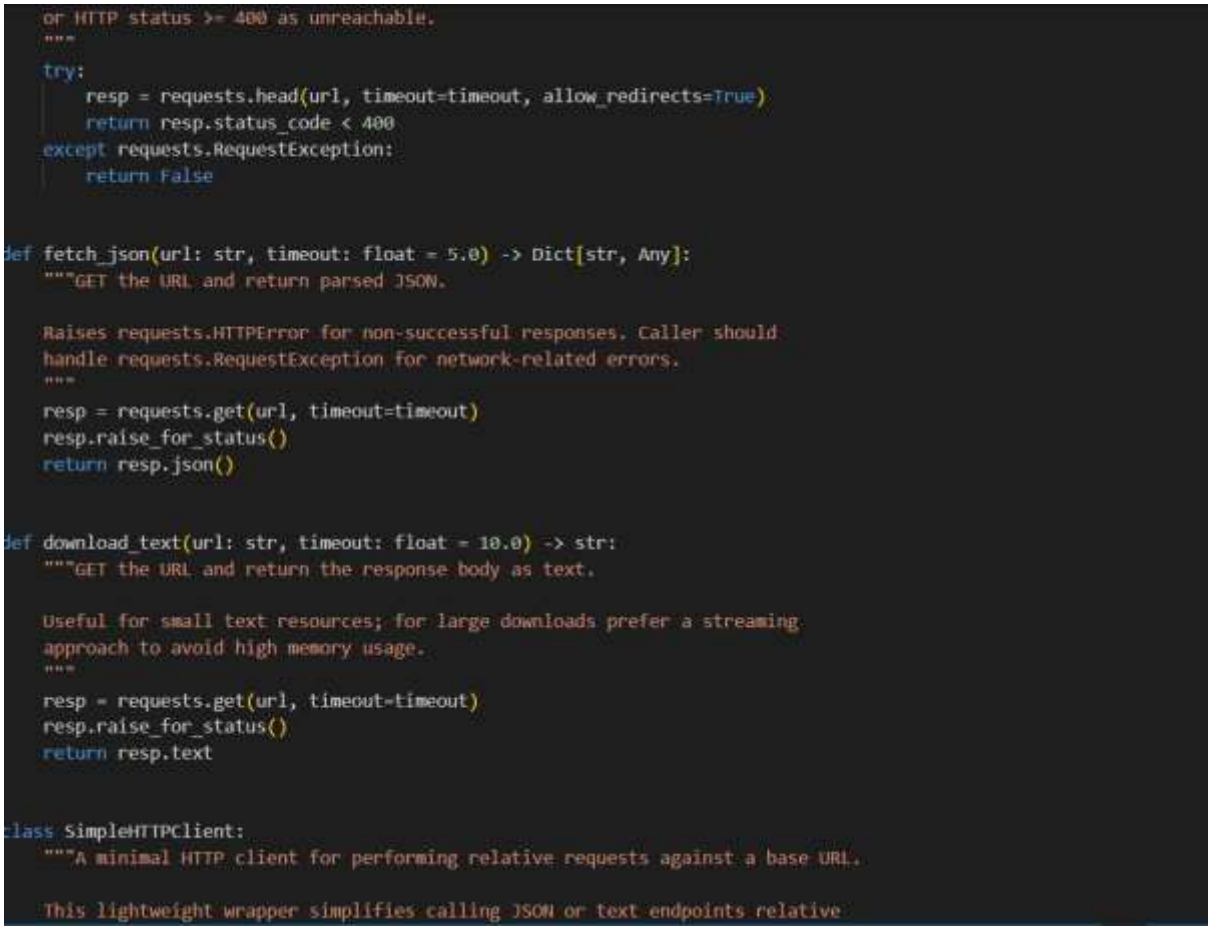
- Preserving code and minimal commenting keeps risk low while delivering clear, actionable context for readers and IDE tooltips.

TASK-3: Generating Module-Level Documentation for a Python package

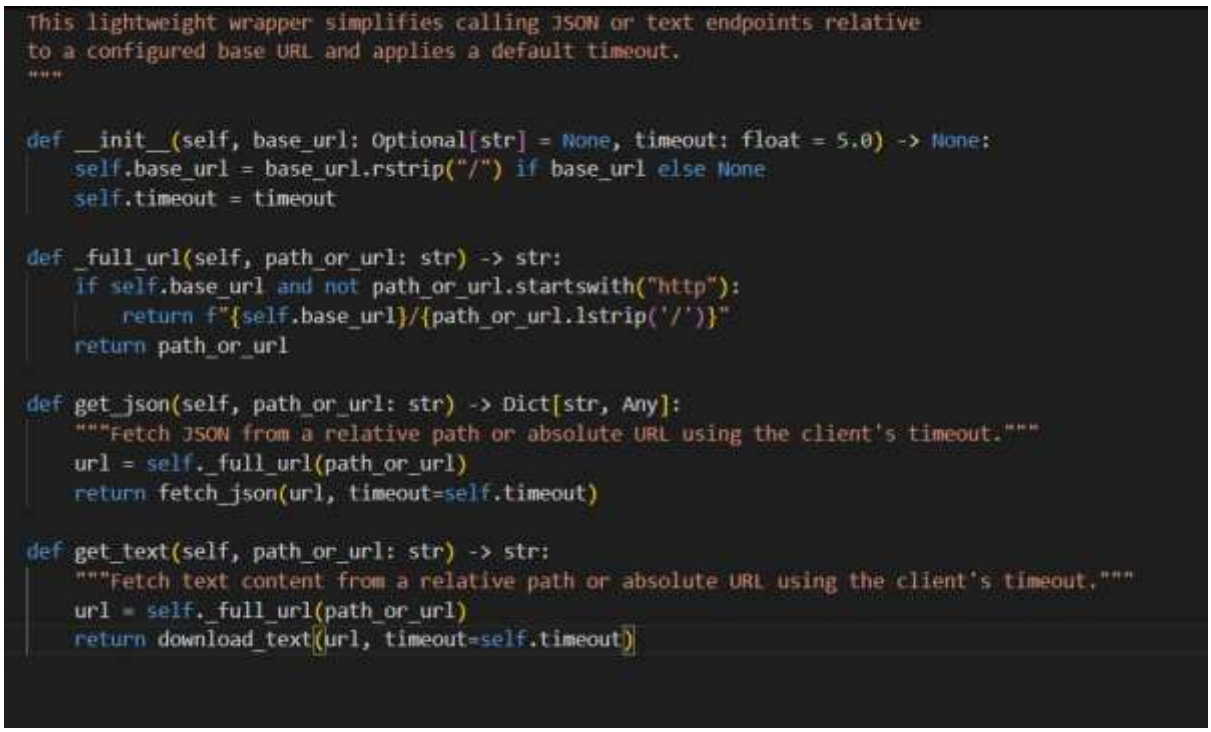
Prompt: Add a module-level docstring to the top of this Python file. Describe the module's purpose, required dependencies, brief descriptions of key functions/classes, and include a short usage example. Keep tone professional and the docstring multi-line.

Code:

```
"""netutils – small HTTP helper utilities.

Purpose:
    Convenience helpers for small HTTP interactions (pinging URLs, downloading
    JSON or text content, and a simple client wrapper). Use this module to
    centralize lightweight network helpers used across scripts or services.

Dependencies:
    - requests (pip install requests)

Key functions/classes:
    - is_url_reachable(url, timeout=3.0) -> bool
        Send a HEAD request to check whether a URL is accessible (status < 400).
    - fetch_json(url, timeout=5.0) -> dict
        GET a URL and return the parsed JSON body; raises on non-2xx responses.
    - download_text(url, timeout=10.0) -> str
        GET a URL and return the response body as text.
    - SimpleHTTPClient(base_url=None, timeout=5.0)
        Minimal wrapper to perform relative GET requests against a base URL
        and return JSON or text.

Example:
    >>> from netutils import fetch_json, SimpleHTTPClient
    >>> data = fetch_json("https://api.github.com")
    >>> client = SimpleHTTPClient(base_url="https://api.example.com")
    >>> client.get_json("/status")
    {'status': 'ok'}
"""
from typing import Any, Dict, Optional
import requests


def is_url_reachable(url: str, timeout: float = 3.0) -> bool:
    """Return True if the URL responds (via HEAD) with a successful status.

    Uses a HEAD request to minimize transferred payload. Treats any exception
```



```
    or HTTP status >= 400 as unreachable.
    """
    try:
        resp = requests.head(url, timeout=timeout, allow_redirects=True)
        return resp.status_code < 400
    except requests.RequestException:
        return False


def fetch_json(url: str, timeout: float = 5.0) -> Dict[str, Any]:
    """GET the URL and return parsed JSON.

    Raises requests.HTTPError for non-successful responses. Caller should
    handle requests.RequestException for network-related errors.
    """
    resp = requests.get(url, timeout=timeout)
    resp.raise_for_status()
    return resp.json()


def download_text(url: str, timeout: float = 10.0) -> str:
    """GET the URL and return the response body as text.

    Useful for small text resources; for large downloads prefer a streaming
    approach to avoid high memory usage.
    """
    resp = requests.get(url, timeout=timeout)
    resp.raise_for_status()
    return resp.text


class SimpleHTTPClient:
    """A minimal HTTP client for performing relative requests against a base URL.

    This lightweight wrapper simplifies calling JSON or text endpoints relative
```



```
    This lightweight wrapper simplifies calling JSON or text endpoints relative
    to a configured base URL and applies a default timeout.
    """

    def __init__(self, base_url: Optional[str] = None, timeout: float = 5.0) -> None:
        self.base_url = base_url.rstrip("/") if base_url else None
        self.timeout = timeout

    def _full_url(self, path_or_url: str) -> str:
        if self.base_url and not path_or_url.startswith("http"):
            return f"{self.base_url}/{path_or_url.lstrip('/')}"
        return path_or_url

    def get_json(self, path_or_url: str) -> Dict[str, Any]:
        """Fetch JSON from a relative path or absolute URL using the client's timeout."""
        url = self._full_url(path_or_url)
        return fetch_json(url, timeout=self.timeout)

    def get_text(self, path_or_url: str) -> str:
        """Fetch text content from a relative path or absolute URL using the client's timeout."""
        url = self._full_url(path_or_url)
        return download_text(url, timeout=self.timeout)
```

JUSTIFICATION:

- A module-level docstring gives an immediate, high-level overview (purpose, deps, key API) to anyone opening the file, speeding onboarding and review.

- Listing dependencies and short examples reduces friction for reuse and testing.
- Including brief function/class summaries helps maintainers and tooling (IDE tooltips, Sphinx) without reading the entire implementation.

TASK-4: Converting Developer Comments into Structured Docstrings PROMPT:

Convert long inline comments inside functions into Google-style docstrings placed immediately after each def. Move explanatory comments (purpose, algorithm, invariants, edge cases) into the docstring and remove redundant inline comments from the body. Preserve behavior, types, and wording where possible. Return the full updated file.

CODE:

```python
import heapq
from typing import List, Iterable, Tuple

def merge_intervals(intervals: List[Tuple[int, int]]) -> List[Tuple[int, int]]:
    # If there are no intervals, return empty list
    # Sort intervals by start time to make merging easy
    # Iterate through sorted intervals and merge overlapping ones:
    # - if current interval start <= last merged end, extend the end
    # - else, append a new interval
    if not intervals:
        return []
    intervals.sort(key=lambda x: x[0])
    merged = [intervals[0]]
    for start, end in intervals[1:]:
        last_start, last_end = merged[-1]
        if start <= last_end:
            merged[-1] = (last_start, max(last_end, end))
        else:
            merged.append((start, end))
    return merged

def running_median(stream: Iterable[float]) -> List[float]:
    # Maintain two heaps: maxheap for lower half, minheap for upper half
    # Balancing property: sizes differ by at most one
    # For each new value:
    # - if value <= -maxheap[0] push to maxheap else to minheap
    # - rebalance heaps if size difference > 1
    # - median is top of the heap(s) depending on sizes
    lower = []  # maxheap (store negatives)
    upper = []  # minheap
    medians = []
    for value in stream:
        if not lower or value <= -lower[0]:
            heapq.heappush(lower, -value)
        else:
            heapq.heappush(upper, value)
```

```python
lower = []  # maxheap (store negatives)
upper = []  # minheap
medians = []
for value in stream:
    if not lower or value <= -lower[0]:
        heapq.heappush(lower, -value)
    else:
        heapq.heappush(upper, value)
    # rebalance
    if len(lower) > len(upper) + 1:
        heapq.heappush(upper, -heapq.heappop(lower))
    elif len(upper) > len(lower) + 1:
        heapq.heappush(lower, -heapq.heappop(upper))
    # compute median
    if len(lower) == len(upper):
        med = (-lower[0] + upper[0]) / 2.0
    elif len(lower) > len(upper):
        med = -lower[0]
    else:
        med = upper[0]
    medians.append(med)
return medians
```

JUSTIFICATION:

- Moving explanatory comments into docstrings standardizes documentation and surfaces intent to tools/IDEs.
- Removing redundant inline comments declutters function bodies and focuses comments on non-obvious points captured in docstrings.
- Google-style docstrings provide a consistent, machine-parseable format for API docs, type hints, and onboarding.

TASK-5: Building a Mini Automatic Documentation Generator PROMPT:

"Generate a Python utility that scans a .py file, finds functions and classes, and inserts Google-style placeholder docstrings automatically."

CODE:

```python
import ast

def add_docstrings(file_path):
    with open(file_path, "r") as f:
        source = f.read()

    tree = ast.parse(source)
    lines = source.split("\n")

    for node in ast.walk(tree):
        if isinstance(node, (ast.FunctionDef, ast.ClassDef)):
            if not ast.get_docstring(node):
                docstring = [
                    '"""',
                    f"{'Class' if isinstance(node, ast.ClassDef) else 'Function'} {node.name} summary.",
                    "",
                    "Args:",
                    "    TODO: add arguments.",
                    "",
                    "Returns:",
                    "    TODO: add return description.",
                    '"""'
                ]
                insert_line = node.body[0].lineno - 1
                lines.insert(insert_line, "\n".join(docstring))

    with open(file_path.replace(".py", "_doc.py"), "w") as f:
        f.write("\n".join(lines))

# Example usage:
# add_docstrings("example.py")
```

JUSTIFICATION:

This utility leverages Python's ast module to parse the file structure safely, ensuring functions and classes are detected without executing code. It checks for missing docstrings and inserts Google-style placeholders, giving developers a quick starting point for documentation while maintaining consistency.