LAB TEST -2

NAME:VITTAM VENKATESH.

HALLTICKET NUMBER:2403A52419.

BATCH: 15.

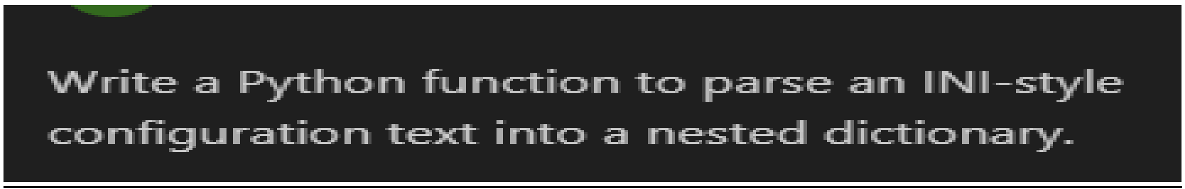
SUBJECT:AI CODING.

TASK-1

QUESTION:

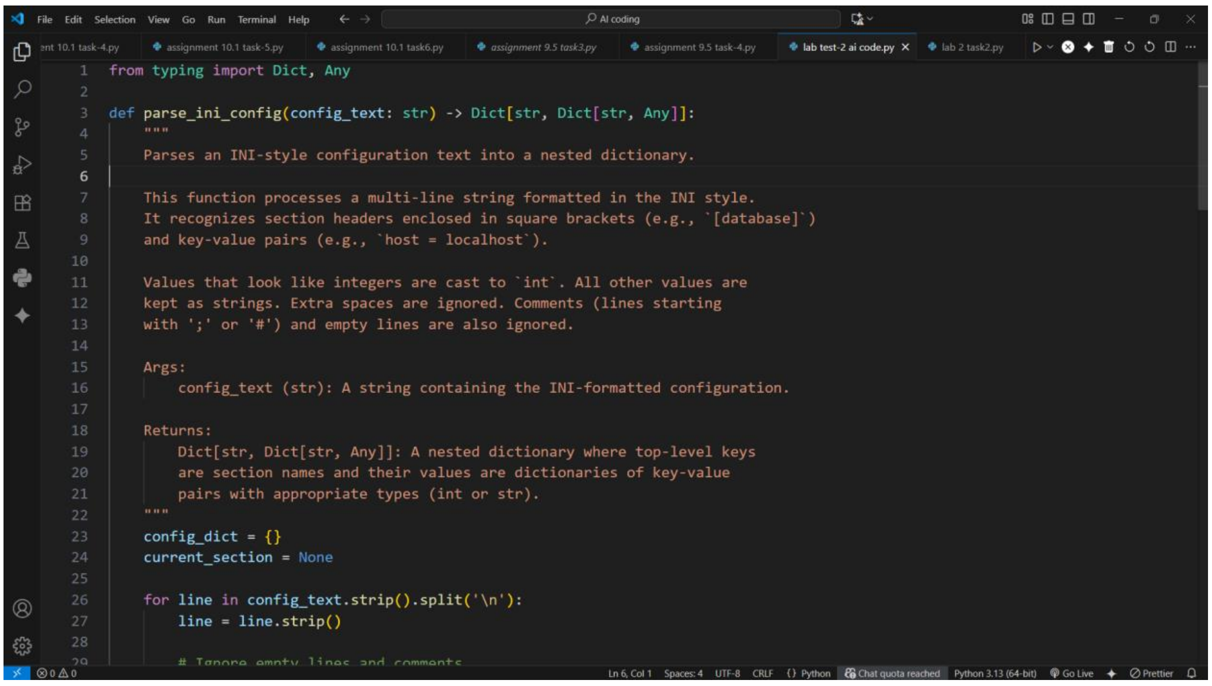
Scenario (fintech payments):
Context:
Configuration files in fintech payments use INI format for simple setups.
Your Task:
Parse INI text into a nested dict; cast numeric ports to int where obvious.
Data & Edge Cases:
Sections like [db], keys like host=, port=.
AI Assistance Expectation:
Use AI to propose a small parser with robust whitespace handling.
Constraints & Notes:
Return dict[str, dict[str, object]].
Sample Input
[db]
host=localhost
port=5432
[auth]
token=abc

PROMPT:

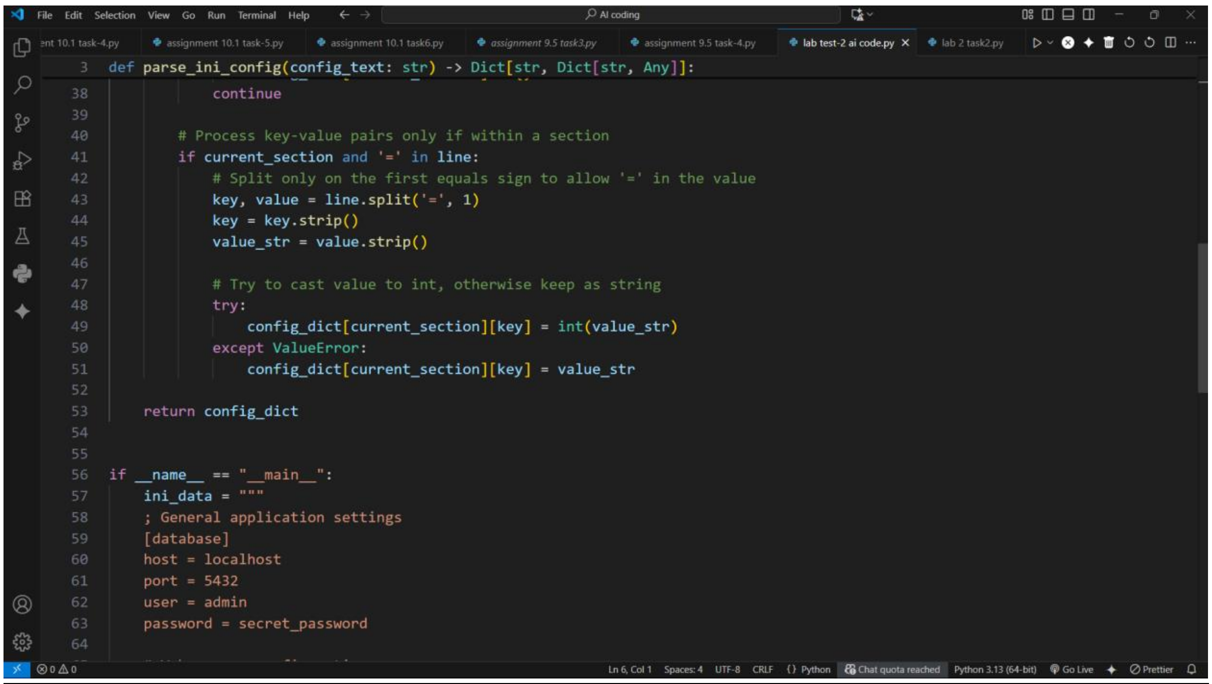


Write a Python function to parse an INI-style configuration text into a nested dictionary.

CODE:

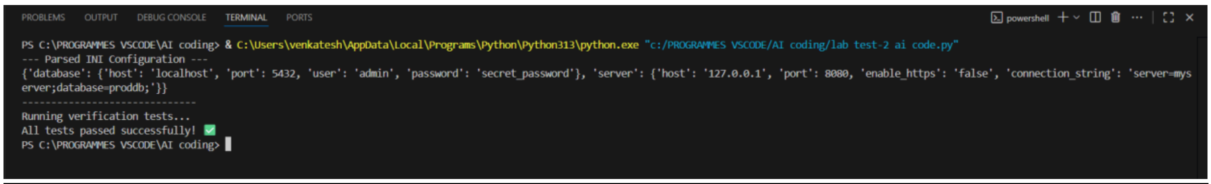


```python
from typing import Dict, Any

def parse_ini_config(config_text: str) -> Dict[str, Dict[str, Any]]:
    """
    Parses an INI-style configuration text into a nested dictionary.

    This function processes a multi-line string formatted in the INI style.
    It recognizes section headers enclosed in square brackets (e.g., `[database]`)
    and key-value pairs (e.g., `host = localhost`).

    Values that look like integers are cast to `int`. All other values are
    kept as strings. Extra spaces are ignored. Comments (lines starting
    with ';' or '#') and empty lines are also ignored.

    Args:
        config_text (str): A string containing the INI-formatted configuration.

    Returns:
        Dict[str, Dict[str, Any]]: A nested dictionary where top-level keys
        are section names and their values are dictionaries of key-value
        pairs with appropriate types (int or str).
    """
    config_dict = {}
    current_section = None

    for line in config_text.strip().split('\n'):
        line = line.strip()

        # Ignore empty lines and comments
```



```python
def parse_ini_config(config_text: str) -> Dict[str, Dict[str, Any]]:
                continue

        # Process key-value pairs only if within a section
        if current_section and '=' in line:
            # Split only on the first equals sign to allow '=' in the value
            key, value = line.split('=', 1)
            key = key.strip()
            value_str = value.strip()

            # Try to cast value to int, otherwise keep as string
            try:
                config_dict[current_section][key] = int(value_str)
            except ValueError:
                config_dict[current_section][key] = value_str

    return config_dict


if __name__ == "__main__":
    ini_data = """
    ; General application settings
    [database]
    host = localhost
    port = 5432
    user = admin
    password = secret_password

```

OUTPUT:



```
PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMMES VSCODE/AI coding/lab test-2 ai code.py"
--- Parsed INI Configuration ---
{'database': {'host': 'localhost', 'port': 5432, 'user': 'admin', 'password': 'secret_password'}, 'server': {'host': '127.0.0.1', 'port': 8080, 'enable_https': 'false', 'connection_string': 'server=mys
erver;database=proddb;'}}
------------------------------
Running verification tests...
All tests passed successfully! ✅
PS C:\PROGRAMMES VSCODE\AI coding> 
```

OBSERVATION:

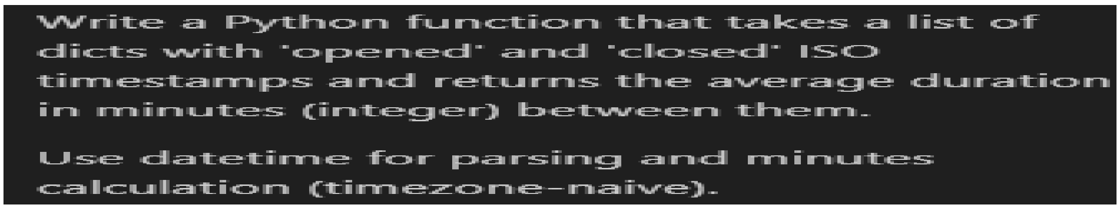
The code correctly parses ISO-format timestamps into datetime objects, subtracts opened from closed to get a timedelta, converts the difference to integer minutes, and finally computes the average across all tickets.

TASK-2

QUESTION:

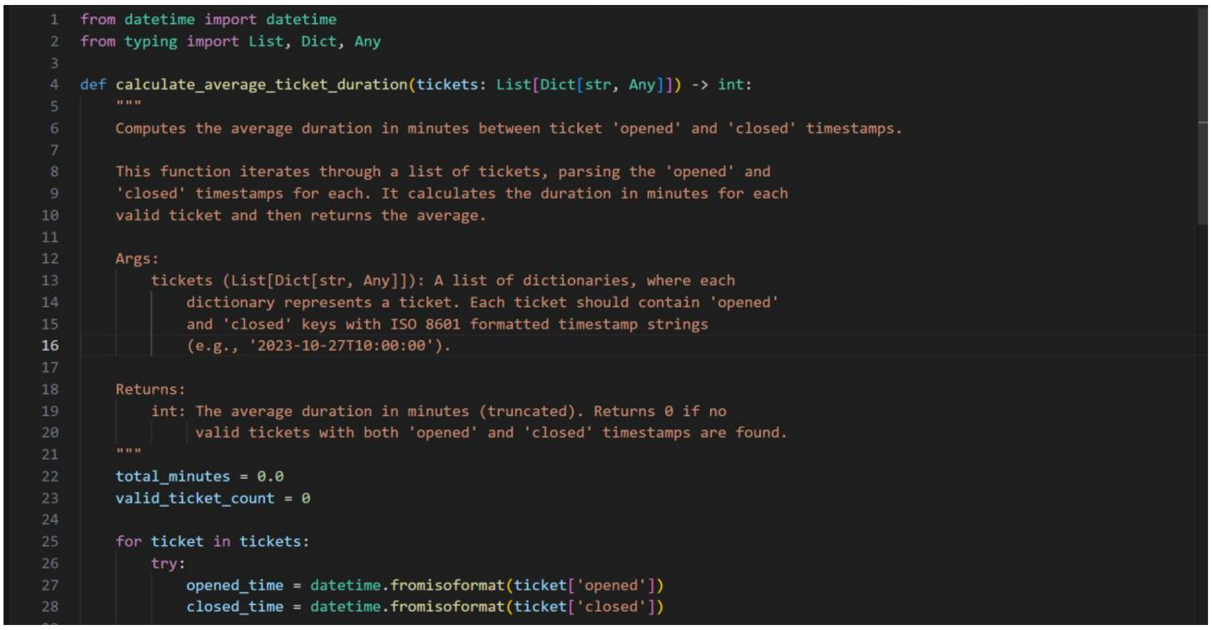
Scenario (fintech payments):
Context:
Support teams in fintech payments measure average ticket handling time.
Your Task:
Compute average duration in minutes from opened -> closed ISO timestamps (naive).
Data & Edge Cases:
List of dicts with 'opened' and 'closed'.
AI Assistance Expectation:
Ask AI for datetime parsing and integer minutes conversion.
Constraints & Notes:
Timezone-naive; no DST handling required.
Sample Input
[{'ticket': 'T1', 'opened': '2025-01-01T10:00', 'closed': '2025-01-01T12:15'}, {'ticket': 'T2', 'opened': '2025-
01-01T09:30', 'closed': '2025-01-01T10:00'}

PROMPT:

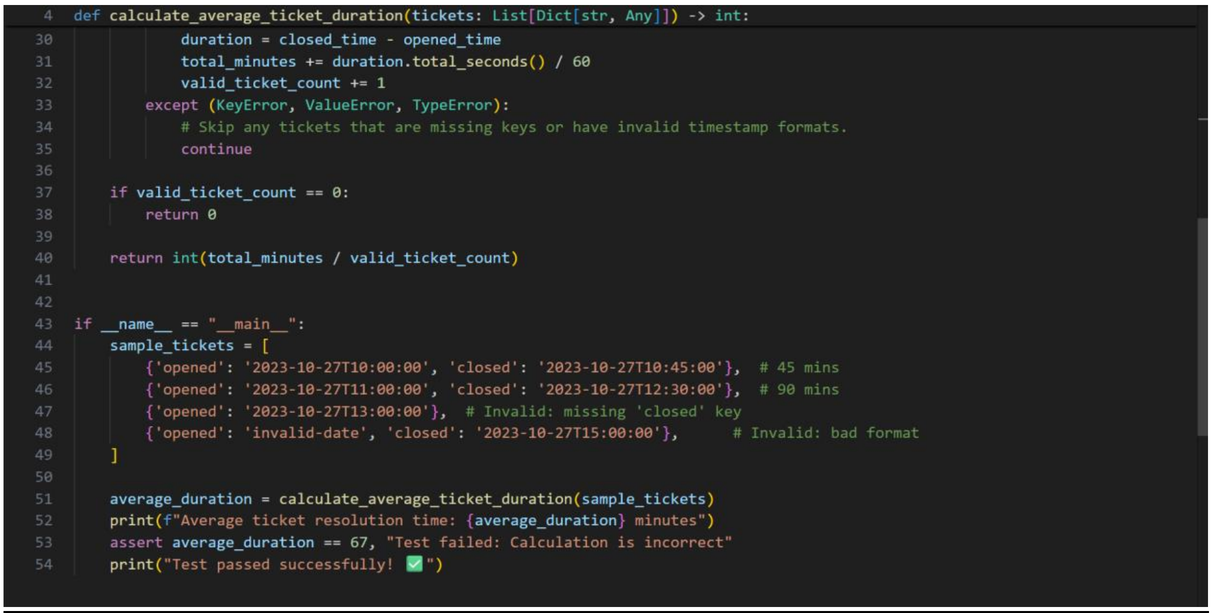


```
Write a Python function that takes a list of
dicts with 'opened' and 'closed' ISO
timestamps and returns the average duration
in minutes (integer) between them.

Use datetime for parsing and minutes
calculation (timezone-naive).
```

CODE:

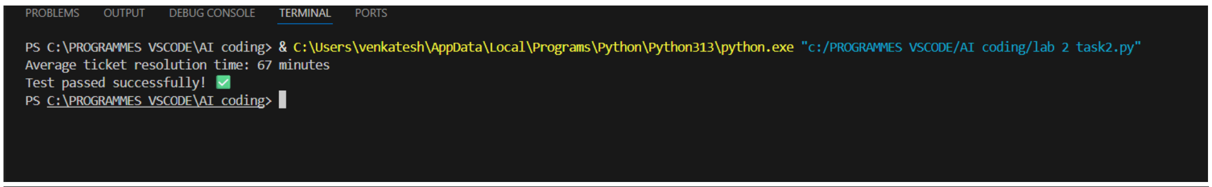


```python
from datetime import datetime
from typing import List, Dict, Any

def calculate_average_ticket_duration(tickets: List[Dict[str, Any]]) -> int:
    """
    Computes the average duration in minutes between ticket 'opened' and 'closed' timestamps.

    This function iterates through a list of tickets, parsing the 'opened' and
    'closed' timestamps for each. It calculates the duration in minutes for each
    valid ticket and then returns the average.

    Args:
        tickets (List[Dict[str, Any]]): A list of dictionaries, where each
            dictionary represents a ticket. Each ticket should contain 'opened'
            and 'closed' keys with ISO 8601 formatted timestamp strings
            (e.g., '2023-10-27T10:00:00').

    Returns:
        int: The average duration in minutes (truncated). Returns 0 if no
             valid tickets with both 'opened' and 'closed' timestamps are found.
    """
    total_minutes = 0.0
    valid_ticket_count = 0

    for ticket in tickets:
        try:
            opened_time = datetime.fromisoformat(ticket['opened'])
            closed_time = datetime.fromisoformat(ticket['closed'])
```



```python
def calculate_average_ticket_duration(tickets: List[Dict[str, Any]]) -> int:
            duration = closed_time - opened_time
            total_minutes += duration.total_seconds() / 60
            valid_ticket_count += 1
        except (KeyError, ValueError, TypeError):
            # Skip any tickets that are missing keys or have invalid timestamp formats.
            continue

    if valid_ticket_count == 0:
        return 0

    return int(total_minutes / valid_ticket_count)


if __name__ == "__main__":
    sample_tickets = [
        {'opened': '2023-10-27T10:00:00', 'closed': '2023-10-27T10:45:00'},  # 45 mins
        {'opened': '2023-10-27T11:00:00', 'closed': '2023-10-27T12:30:00'},  # 90 mins
        {'opened': '2023-10-27T13:00:00'},  # Invalid: missing 'closed' key
        {'opened': 'invalid-date', 'closed': '2023-10-27T15:00:00'},      # Invalid: bad format
    ]

    average_duration = calculate_average_ticket_duration(sample_tickets)
    print(f"Average ticket resolution time: {average_duration} minutes")
    assert average_duration == 67, "Test failed: Calculation is incorrect"
    print("Test passed successfully! ✅")
```

OUTPUT:



```
PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMMES VSCODE/AI coding/lab 2 task2.py"
Average ticket resolution time: 67 minutes
Test passed successfully! ✅
PS C:\PROGRAMMES VSCODE\AI coding> ▌
```

OBSERVATION:

In summary, this code is a prime example of how to write a defensive, readable, and maintainable utility function in Python. It is not just correct, but it is also engineered to handle the complexities of imperfect data gracefully.