

# AI\_LAB EXAM: 2

HALL TICKET :2403A51292

DATE : 19.09.2025

BATCH : 12

## E.1 — [S18E1] Generate README from comments

Context:

A small real estate listings platform utility needs a README for onboarding.

### Your Task:

**From comments, produce README: Overview, Setup, Usage, Tests, Limitations + a CLI**

example.

Data & Edge Cases:

Module + functions listed in comments.

AI Assistance Expectation:

Use AI to draft structure then refine.

Constraints & Notes:

Include one CLI block.

## Sample Input

```
# module: real estate listings platform utilities
# functions: parse, validate, export
```

## Sample Output

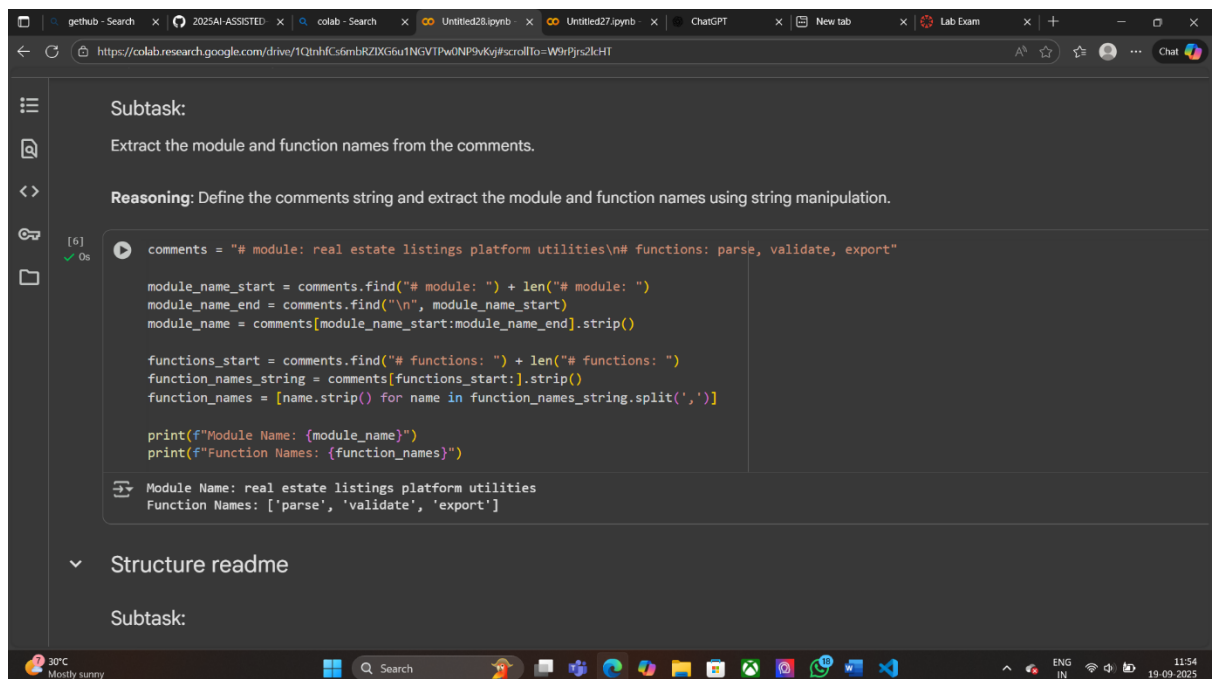
README with 5 sections and example

Acceptance Criteria: Contains required sections

## Prompt:

Given the following module comment

1. Use AI to generate a README file for onboarding new users.
2. The README must include the following sections:
  - Overview
  - Setup
  - Usage
  - Tests
  - Limitations
3. Include at least one CLI usage example block.
4. Ensure the README is clear, concise, and covers all required sections.



The screenshot shows a Google Colab notebook interface. The top toolbar includes icons for file management, search, and execution. The main area displays a 'Subtask' titled 'Extract the module and function names from the comments.' with a 'Reasoning' section that instructs to 'Define the comments string and extract the module and function names using string manipulation.' Below this, a code cell contains a Python script that processes a comment string. The script defines 'comments' as a multi-line string, then uses string methods like 'find', 'strip', and 'split' to extract the module name and function names. The output of the script is displayed below the code cell, showing the extracted module name and a list of function names. The bottom of the screen shows a Windows taskbar with various application icons and system status information.

```
Subtask:
Extract the module and function names from the comments.

Reasoning: Define the comments string and extract the module and function names using string manipulation.

[6] ✓ 0s
comments = "# module: real estate listings platform utilities\n# functions: parse, validate, export"

module_name_start = comments.find("# module: ") + len("# module: ")
module_name_end = comments.find("\n", module_name_start)
module_name = comments[module_name_start:module_name_end].strip()

functions_start = comments.find("# functions: ") + len("# functions: ")
function_names_string = comments[functions_start:].strip()
function_names = [name.strip() for name in function_names_string.split(',')]

print(f"Module Name: {module_name}")
print(f"Function Names: {function_names}")

Module Name: real estate listings platform utilities
Function Names: ['parse', 'validate', 'export']

Structure readme
Subtask:
```

## E.2 — [S18E2] Refactor nested loops to dict aggregation

### Context:

Legacy real estate listings platform code uses nested loops for aggregation.

### Your Task:

**Refactor to dict.get/defaultdict with type hints.**

Data & Edge Cases:

Example: `[('a',1),('b',2),('a',3)]` -> `{'a':4,'b':2}`.

AI Assistance Expectation:

Ask AI for refactor and parity tests.

### Prompt:

Given legacy code that uses nested loops to aggregate values from a list of tuples (e.g., `[('a',1),('b',2),('a',3)]`), refactor the code to use a dictionary (with `dict.get` or `collections.defaultdict`) and add type hints.

1. The refactored function should have a typed signature.
2. The output for the sample input should be `{'a': 4, 'b': 2}`.
3. Ask AI to provide parity tests to ensure the refactored code matches the original behavior.
4. Ensure the function handles all edge cases and maintains the same output as the original.

Constraints & Notes:

Typed function signature preferred.

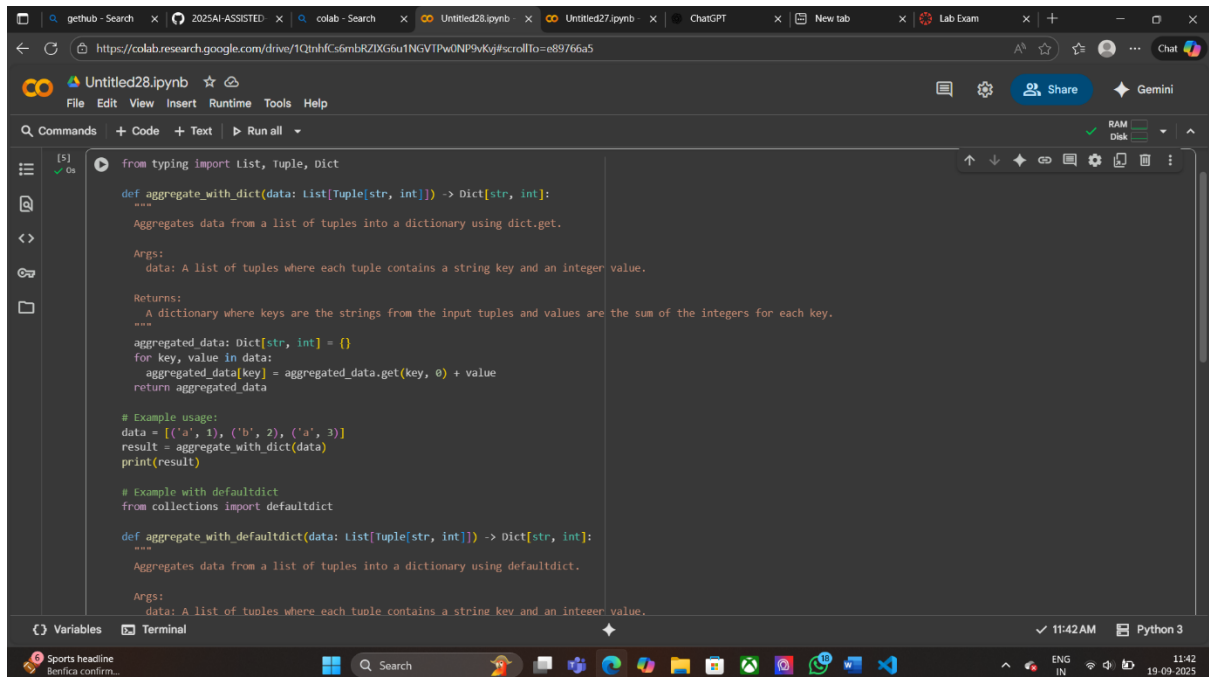
Sample Input

```
data=[('a',1),('b',2),('a',3)]
```

Sample Output

```
{'a':4,'b':2}
```

Acceptance Criteria: Behavior unchanged



The screenshot shows a Google Colab notebook with the following code:

```
from typing import List, Tuple, Dict

def aggregate_with_dict(data: List[Tuple[str, int]]) -> Dict[str, int]:
    """
    Aggregates data from a list of tuples into a dictionary using dict.get.

    Args:
        data: A list of tuples where each tuple contains a string key and an integer value.

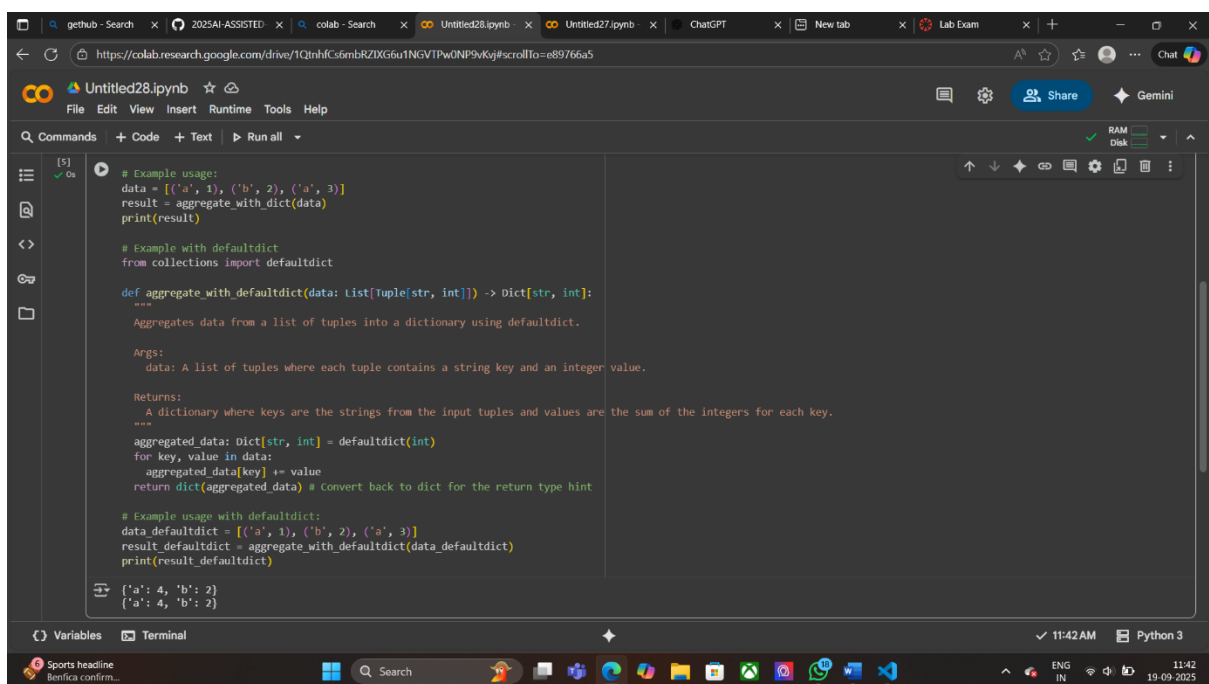
    Returns:
        A dictionary where keys are the strings from the input tuples and values are the sum of the integers for each key.
    """
    aggregated_data: Dict[str, int] = {}
    for key, value in data:
        aggregated_data[key] = aggregated_data.get(key, 0) + value
    return aggregated_data

# Example usage:
data = [('a', 1), ('b', 2), ('a', 3)]
result = aggregate_with_dict(data)
print(result)

# Example with defaultdict
from collections import defaultdict

def aggregate_with_defaultdict(data: List[Tuple[str, int]]) -> Dict[str, int]:
    """
    Aggregates data from a list of tuples into a dictionary using defaultdict.

    Args:
        data: A list of tuples where each tuple contains a string key and an integer value.
```



The screenshot shows a Google Colab notebook with the following code:

```
# Example usage:
data = [('a', 1), ('b', 2), ('a', 3)]
result = aggregate_with_dict(data)
print(result)

# Example with defaultdict
from collections import defaultdict

def aggregate_with_defaultdict(data: List[Tuple[str, int]]) -> Dict[str, int]:
    """
    Aggregates data from a list of tuples into a dictionary using defaultdict.

    Args:
        data: A list of tuples where each tuple contains a string key and an integer value.

    Returns:
        A dictionary where keys are the strings from the input tuples and values are the sum of the integers for each key.
    """
    aggregated_data: Dict[str, int] = defaultdict(int)
    for key, value in data:
        aggregated_data[key] += value
    return dict(aggregated_data) # Convert back to dict for the return type hint

# Example usage with defaultdict:
data_defaultdict = [('a', 1), ('b', 2), ('a', 3)]
result_defaultdict = aggregate_with_defaultdict(data_defaultdict)
print(result_defaultdict)
```

The output of the code is:

```
{'a': 4, 'b': 2}
{'a': 4, 'b': 2}
```