

This document include;

Summary

Structured Conversation Logs

All conversations can be seen via this link;

<https://chatgpt.com/share/689f4148-4e1c-8006-a872-2a9079efa5e4>

Before/After Implementation Examples


Summary

Among the two project options, I chose the Text Analysis Tool (TAT).

This is because we can complete this project by preparing only one table, whereas the Personal Finance Tracker may require three or more.

At first, I thought there would be major challenges in the TAT project, such as detecting sarcasm or determining whether a comment is negative or positive. However, after conversation with AI, I realized that counting words, identifying frequent words, and measuring word length would be sufficient, so I decided to select TAT.

In regard to Modular Design, We, I and AI, broke programs into 4 steps, Input, Tokenize, Count and Plot. The following is functions of these 4 steps.

Step	Function name example	What it does	Possible file/module 
Input	<code>get_user_text()</code>	Asks the user to paste text (using <code>pyinputplus</code>)	<code>input_utils.py</code>
Tokenize	<code>tokenize_text(text)</code>	Lowercases, removes punctuation, splits into words	<code>preprocess.py</code>
Count	<code>analyze_words(words)</code>	Counts total words, finds top N, average lengths	<code>analysis.py</code>
Plot	<code>plot_top_words(word _counts)</code>	Draws bar chart of top words	<code>viz.py</code>

(Step 1) This function asks the user;
to paste or type text,
then returns what they typed as a string.

(Step 2) This function;

takes the text,
makes it lowercase,
removes punctuation,
and then splits it into a list of words.

(Step 3) This function;
counts total words,
calculates the average word length,
finds how often each word appears,
and returns the top 10 most frequent words.

(Step 4) This function runs all the other steps in order:
it gets the text,
tokenizes it,
analyses it,
and then displays the results in the console.


Implementation has been done like the following. Firstly, I asked an AI to write code, not to use it, but to share it with other AIs, get their opinions and alternatives and compare suggested codes.

Then, I found challenges, such as;

The method (`isalnum()`), which were initially used, removes apostrophes.
(e.g., "can't" → "cant").

This was in Step 2 (Tokenize).
The code was initially like the following;

python

 Copy  Edit

```
def tokenize_text(text):  
    text = text.lower()  
    clean_text = ""  
    for char in text:  
        if char.isalnum() or char.isspace():  
            clean_text += char  
    words = clean_text.split()  
    return words
```

Here, we could pick up a character if it is a letter or number (a-z, A-Z, 0-9) or a space, but could not if it is an apostrophe, "'". Then, "can't" was converted into "cant".

In order to catch an apostrophe "'", I changed the code
if char.isalnum() or char.isspace():
into
if char.isalnum() or char.isspace() or char == "'":

Structured Conversation Logs

Conversation 1

(Me)

What assumptions are you making about the improvement after we use pandas, matplotlib, and other relevant libraries. We are required to use Effectively those libraries.

(AI)

Great question. Here's exactly what I'm assuming will improve—and what “use effectively” should look like for your marker—when you add pandas, matplotlib, and relevant libs (incl. pyinputplus) to your TAT.

- 1) Assumptions about improvements
 - a) pandas (tables & counts)

Cleaner, fewer lines: replace manual loops/dicts with Series.value_counts() and simple slicing.

Fewer bugs / clearer intent: frequency tables and “top-N” become one-liners you can format and display.

Easy extensions: n-grams, stopword toggles, merging comparisons (two texts) via DataFrames.

- b) matplotlib (visualisation)

Readable evidence: bar chart of top words, histogram of word lengths—clear, labeled, saved to file.

Control over layout: tick rotation, titles, tight_layout() so nothing overlaps.

Assessment fit: directly satisfies “Include appropriate visualisations of data.”

- c) pyinputplus (user interface)

Robust menu: numbered choices, simple validation, fewer crashes.

User friendly: consistent prompts for “enter text / analyze / plot / compare”.

d) (Optionally) numpy

Smoother histograms: quick numeric ops for bins (not mandatory, but nice).

2) What “use effectively” should demonstrate (concrete, checkable)

For pandas

Use `pd.Series(words).value_counts()` for frequency.

Convert to a neat table: `top10 = counts.head(10).reset_index().rename(columns={'index':'word',0:'count'})`.

Show comparisons (two texts) by building two Series and joining them (`.to_frame()` + `.join()`), then fill missing with 0.

For matplotlib

Two plots minimum:

Bar chart of top-N words with rotated x-ticks and axis labels

Histogram of word lengths with integer bins

Call `plt.tight_layout()` and save figures (e.g., `plt.savefig('docs/figures/top_words.png')`).

For pyinputplus

`inputMenu([...], numbered=True)` for the main menu.

Guard rails: if text is empty, print a friendly message and return to menu.

If you do these three things, you’re “using effectively” in a way the rubric can verify.

Conversation 2

(Me)

I selected "y" in option 3, about save image, and found the error like the picture.

What assumptions are you making about the cause of the error?

```
resubmission-assessment1-csc101.ipynb  Unsaved changes since 8:25 PM
File Edit View Insert Runtime Tools Help

hands + Code + Text ▶ Run all ▼

6. Clear current text
7. Quit
3
Save image to file? (y/n): y
-----
FileNotFoundError                                Traceback (most recent call last)
/tmp/ipython-input-3490198786.py in <cell line: 0>()
    184 # ----- Entry point -----
    185 if __name__ == "__main__":
--> 186     main_menu()
    187

----- 9 frames -----
/usr/local/lib/python3.11/dist-packages/PIL/Image.py in save(self, fp, format, **params)
    2581     fp = builtins.open(filename, "r+b")
    2582     else:
-> 2583     fp = builtins.open(filename, "w+b")
    2584     else:
    2585     fp = cast(IO[bytes], fp)

FileNotFoundError: [Errno 2] No such file or directory: 'docs/figures/top_words.png'
```

(AI)

If you got an error right after selecting `***y***` for saving the image in option 3, my top assumptions (based on how the code is set up) would be:

`**1. The `docs/figures/` folder doesn't exist**`

* In my latest code, the save path is something like:

```
```python
plt.savefig("docs/figures/top_words.png")
```
```

* If your project folder does not already have a ``docs/figures/`` directory, ``matplotlib`` will throw:

```
```
FileNotFoundError: [Errno 2] No such file or directory: 'docs/figures/top_words.png'
```
```

* This is the `**most common cause**` in console programs where we specify subfolders.

2. Permission issues

- * If you're running in a location where your Python process cannot create or write files (e.g., certain restricted directories, Google Colab read-only system folders).
- * Usually gives a `PermissionError`.

3. Relative path confusion

- * If your current working directory isn't where you think it is, `docs/figures/` might be interpreted relative to a completely different location.
- * You can confirm this by adding:

```
```python
import os
print(os.getcwd())
```
```

4. Running in a restricted environment

- * Some hosted environments (certain sandboxes) allow plotting but block file creation unless it's in a temporary folder they allow.
- * In that case, you'd need to save to something like `/content/` in Colab.

💡 ****Likely fix:****

Modify the code so it creates the directory automatically if it doesn't exist:

```
```python
import os

save_path = "docs/figures/top_words.png"
os.makedirs(os.path.dirname(save_path), exist_ok=True)
plt.savefig(save_path)
```
```

If you want, I can **patch the latest code** so it never errors when saving the plot — it will auto-create folders and give you the exact saved location. This will make the assignment run smoother.

Conversation 3

(Me)

Can you show both a "non-saving" and a "saving" approach to this problem?

In a non-saving approach, just showing image is enough and saving image is not needed.

(AI)

Got it — you want both approaches:

1. Non-Saving Approach

Just display the bar chart in the console (or Colab) without saving anything.

```
```python
import matplotlib.pyplot as plt

def plot_top_words(df):
 df.plot(kind="bar", x="Word", y="Frequency", legend=False)
 plt.title("Top 10 Words")
 plt.xlabel("Word")
 plt.ylabel("Frequency")
 plt.show() # Only display, don't save
```
```

2. Saving Approach (with folder auto-creation)

Show the chart **and** save it to a file, making sure the folder exists.

```
```python
import os
import matplotlib.pyplot as plt

def plot_top_words_and_save(df, save_path="docs/figures/top_words.png"):
 # Make sure the folder exists
 os.makedirs(os.path.dirname(save_path), exist_ok=True)
```

```

df.plot(kind="bar", x="Word", y="Frequency", legend=False)
plt.title("Top 10 Words")
plt.xlabel("Word")
plt.ylabel("Frequency")
plt.savefig(save_path) # Save chart
plt.show() # Also display chart
print(f"Chart saved to {save_path}")
...

```

---

### \*\*Key Difference\*\*

Approach	Shows Chart?	Saves Chart?
Non-Saving	✓	✗
Saving	✓	✓

---

If you want, I can **update your latest TAT project code** so that in the console menu,

- \* If the user chooses "no", it runs **non-saving**
- \* If the user chooses "yes", it runs **saving** without any error.

That way it fully meets your **pyinputplus** and **matplotlib** requirements.  
Do you want me to do that?

## Conversation 4

(Me)

Can you show a more Pythonic version of this function?

(AI)

Sure — here's a **more Pythonic** version of your `tokenize_text()` function.

The main changes are:

- \* Use a **generator expression** instead of appending to a list
- \* Handle both straight ``` and curly ```` apostrophes in a **set**
- \* Join in one line



\* Strip and split directly

**\*\*Pythonic version:\*\***

```
```python
def tokenize_text(text):
    """
    Lowercase; keep letters/digits/spaces/apostrophes; replace others with spaces;
    split to words.
    Preserves both straight (') and curly (').
    """
    text = text.lower()
    keep_apostrophes = {'"', "'"}
    allowed = lambda ch: ch.isalnum() or ch.isspace() or ch in keep_apostrophes
    cleaned_text = "".join(ch if allowed(ch) else " " for ch in text)
    return cleaned_text.split()
```
```

**\*\*Why more Pythonic?\*\***

- \* `***lambda***` for quick inline character check
- \* `***''.join(...)**` with a generator avoids intermediate lists (memory efficient)
- \* `***Set lookup (ch in keep_apostrophes)**` is faster and cleaner than ``or`` chains
- \* Compact but still **\*\*readable\*\***

If you want, I can **\*\*swap the old tokenizer in your one-shot program with this Pythonic version\*\*** so your whole script is cleaner. Would you like me to do that?

## Conversation 5

(Me)

Would a generator be more memory-efficient here than a list?

(AI)

Short answer: **\*\*yes, a generator is more memory-efficient for the cleaning step\*\***, but the overall memory win is **\*\*modest\*\*** because ``split()`` still creates a list of all words at the end.

Here's what's happening:

\* In the Pythonic tokenizer we used:

```
```python
```

```
cleaned_text = "".join(ch if allowed(ch) else " " for ch in text)
...`
```

That ``(... for ch in text)`` is a **generator expression**. It feeds characters to ``join()`` on the fly, so **no intermediate list of characters** is created.

Memory: $O(1)$ extra (besides the final string) vs $O(n)$ if you did ``"".join(list_comprehension)``.

* If you wrote the non-Pythonic version:

```
```python
cleaned_text = "".join([ch if allowed(ch) else " " for ch in text])
...`
```

The ``[...]`` builds a full **list** of all characters first ( $O(n)$  extra memory) and then joins them.

\* After that, we call:

```
```python
cleaned_text.split()
...`
```

which **must** produce a **list of all words**. That list dominates memory once created, regardless of generator vs list in the previous step.

Practical takeaway

* Using a generator in ``join()`` is the right, more memory-efficient style and scales better for very large texts.

* The **big memory consumer** is the final ``words`` list (and later the pandas ``Series``)—so the generator helps, but won't change that final requirement.

Rule of thumb

* Prefer generator expressions inside ``join()`` (and in pipelines) for streaming behavior and lower peak memory.

* If you truly need lower memory overall, you'd avoid holding all words at once (e.g., stream processing), but that's beyond your current assignment's scope.

Before/After Implementation Examples

(The 1st code)

Basic text tokenizer (apostrophes removed because of isalnum())

```
def tokenize_text(text):
```

```
    """
```

```
    Tokenize text by removing all punctuation, splitting into words.
```

```
    """
```

```
    text = text.lower()
```

```
    clean_text = ""
```

```
    for char in text:
```

```
        if char.isalnum() or char.isspace():
```

```
            clean_text += char
```

```
        else:
```

```
            clean_text += " "
```

```
    words = clean_text.split()
```

```
    return words
```

Example usage:

```
sample = "Hello! Can't you see?"
```

```
print(tokenize_text(sample))
```

Output: ['hello', 'cant', 'you', 'see'] <-- apostrophe removed

(The 2nd code)

Tokenizer keeps apostrophes inside words (e.g., "can't")

```
def tokenize_text(text):
```

```
    """
```

```
    Tokenize text, preserving apostrophes inside words.
```

```
    """
```

```
    text = text.lower() # Step 1: make lowercase
```

```
    clean_text = "" # Step 2: empty container for cleaned characters
```

```
    for char in text: # Step 3: go through each character
```

```
        if char.isalnum() or char.isspace() or char == "'":
```

```
            clean_text += char
```

```
        else:
```

```
            clean_text += " " # Replace other punctuation with space
```

```
    words = clean_text.split() # Step 4: split into list of words
```

```
    return words
```

Example usage:

```
sample = "Hello! Can't you see?"
```

```
print(tokenize_text(sample))
# Output: ['hello', "can't", 'you', 'see'] <-- apostrophe preserved
```

(The 3rd code)

```
# ----- Imports -----
import pyinputplus as pyp

# ----- Step 1: Input -----
def get_text_input():
    """
    Prompt the user for text input.
    """
    return pyp.inputStr("Enter your text: ")

# ----- Step 2: Tokenizer -----
def tokenize_text(text):
    """
    Tokenize text, preserving apostrophes inside words.
    """
    text = text.lower()
    clean_text = ""
    for char in text:
        if char.isalnum() or char.isspace() or char == "'":
            clean_text += char
        else:
            clean_text += " "
    words = clean_text.split()
    return words

# ----- Step 3: Analysis -----
def word_count(words):
    return len(words)

def unique_word_count(words):
    return len(set(words))

def top_n_words(words, n=5):
    from collections import Counter
    return Counter(words).most_common(n)

# ----- Step 4: Menu -----
def main_menu():
    text = get_text_input()
    words = tokenize_text(text)
```

```
while True:
    choice = pyp.inputMenu(
        ["Word count", "Unique words", "Top words", "Exit"],
        numbered=True
    )

    if choice == "Word count":
        print("Total words:", word_count(words))
    elif choice == "Unique words":
        print("Unique words:", unique_word_count(words))
    elif choice == "Top words":
        print(top_n_words(words))
    elif choice == "Exit":
        break

# ----- Step 5: Run -----
if __name__ == "__main__":
    main_menu()
```