Individual Final Report

Youtube link : https://youtu.be/_QzUKvwiEqY

1. Naming & Style

Where: Everywhere in function and variable names (e.g., `avgWordLength` → `average_word_length`, `getScore` → `score_signature`)

What Changed: Renamed all functions and variables to follow PEP-8 guidelines (snake_case for function and variable names, lowercase with underscores).

Why:

- Improves readability and predictability.
- Aligns with Python community conventions
- Easier for teams or linters to maintain the code consistently.

Change	Where	Why
PEP-8 function & variable names	All function and variable definitions	Consistent snake_case makes it easier to read, navigate, and integrate with other Python code.

---

2. Command-Line Interface

Change:
Replaced all `input()` function calls with `argparse`-based command-line argument parsing.

Where:
In the `main()` function, where the program originally prompted users for input.

Why:

- **Non-interactive usage**: The program can now be run in batch mode or automated scripts.
- **Help messages**: Users get built-in help with `--help`.
- **Argument validation**: Ensures correct formats and required inputs are present, improving user experience and program robustness.

Change	Where	Why
Swapped `input()` for `argparse`	`main()` CLI entrypoint	Allows non-interactive usage, automatic help/usage messages, and argument validation.

---

3. Modularity & Reusability

Change 1:
Created a helper function `split_on_chars()` to handle text splitting based on punctuation.

Where:
Used in the logic for `get_sentences()` and `get_phrases()`.

Why:

- **Consolidates logic**: Instead of repeating similar loops, it centralizes the splitting behavior.
- **Eliminates duplication**: Less error-prone and easier to update.

Change 2:
Renamed metric functions like `type_token_ratio()` and `hapax_legomena_ratio()` using standard academic terminology.

Where:
In the signature metric calculation functions.

Why:

- **Self-documenting**: Anyone reading the code can recognize what these functions calculate.
- **Standardization makes** the project more professional and easier to explain.

Change 3:
Replaced `get_score()` and `lowest_score()` with `score_signature()` and `find_best_match()`.

Where:
In the scoring and comparison logic.

Why:

- **Clearer roles**: Each function does exactly one thing (single-responsibility principle).
- **Simplifies logic**: Easier to test, debug, and understand.

Change	Where	Why
Extracted splitting logic	New helper `split_on_chars()`	Consolidates all “split on punctuation” behavior in one place, eliminating duplicated loops in sentence, phrase, and signature functions.
Renamed metrics to standard terms	`type_token_ratio()`, `hapax_legomena_ratio()`	Uses established academic terminology, making the code self-documenting and clear to others.
Unified scoring/matching	Single functions `score_signature()` + `find_best_match()`	Replaces the two ad-hoc `get_score()` and `lowest_score()` routines with one clear, single-responsibility pair.

---

4. Robustness & Error Handling

Change 1:
Added a check to ensure the input mystery file exists.

Where:
In `process_data()`.

Why:

- **Fails fast**: Detects missing files early before proceeding with processing.
- **Prevents hidden bugs**: Makes error sources obvious.

Change 2:
Handled the edge case where the known texts directory is empty.

Where:
In `process_data()`, returns `None` if no known texts are found.

Why:

- **Avoids crashes**: Prevents the program from failing when it tries to compare to an empty list.
- **Allows user feedback**: The CLI can now print a clear error or help message.

Change 3:
Added `exit(1)` to indicate failure clearly in scripts.

Where:
In the `main()` function.

Why:

- **Supports scripting**: External tools can detect success/failure based on exit code.
- **Standard practice**: `exit(1)` signals an error occurred.

Change	Where	Why
File existence check	`process_data()` raises `FileNotFoundError`	Fails fast if the mystery file is missing, rather than hiding errors later.
Empty-directory handling	`process_data()` returns `None` when no known texts are found	Prevents downstream exceptions and allows the CLI to print a helpful message.
Exit codes on failure	`main()` prints error and `exit(1)`	Signals failure to calling scripts or shells, enabling automation and testing.

---

5. Documentation & Type Safety

Change 1:
Added docstrings to every function.

Where:
In all core utilities and helper functions.

Why:

- **Self-documentation**: Functions explain themselves.
- **Supports IDEs and tools**: Enables hover-over help and documentation generation.

Change 2:
Used Python's type hints (e.g., `List[str]`, `Optional[str]`).

Where:
In all function signatures.

Why:

- **Type safety**: Makes type mismatches easier to catch early.
- **Better tooling support**: IDEs can auto-complete, lint, and analyze the code more effectively.

Change	Where	Why
Docstrings on every function	All utilities and core functions	Immediate, in-code examples and explanations aid both users and IDEs.
Type hints (`List[str]`, etc.)	Function signatures	Makes it easier to catch type errors during linting or IDE inspections, and serves as lightweight documentation.

---

6. Project Structure & Invocation

Change:
Wrapped the execution logic inside a `main()` function and added a guard: `if __name__ == '__main__'`

Where:
At the bottom of the file.

Why:

- **Dual use**: The script can be run as a program or imported as a module.
- **Supports testing**: Prevents code from executing on import, which is important for unit tests.

Change	Where	Why
Single library + CLI guard	Entire file, `if __name__ == '__main__'`	Separates importable library code from script-only execution logic, supporting both direct use and import for testing.

Overall gains

This summary connects all improvements to practical benefits:

What We Gain	How It Was Achieved
Reusability	All logic in named functions, CLI logic separated via `if __name__ == '__main__'`
Testability	No `input()`, no side effects—just pure functions with parameters
Scriptability	Uses `argparse`, handles errors cleanly, can be automated
Maintainability	DRY code, consistent naming, centralized logic and error reporting