| Name | Zhou King Tao | Class | 5B | No | 28 | Date |  | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

### **Word Analysis and Plagiarism Detection System**

### **Task 1 (Design & Implementation)**

#### 1. Menu Design

Design a menu for the program (e.g., options for word counting, sorting, etc.).

* List the menu options.
* Describe the purpose of each option.

| Menu options | Description |
| --- | --- |
| (for example)  1. Word Counting | (for example)  Count the number of words in the input text. |
| 1. Analyze a single file | Count the number of total words and unique words, and frequency of each word |
| 2. Compare two files for plagiarism | Analyze the two input files using the first option, and then compare intersections of common words to get plagiarism percentage and level |
| 3. Search for a word in a file | Show how many times a word had appeared in a file, and their position |
| 4. Replace a word in a file | Replace a word in a file with another word |
| 5. Configure settings | Configure some settings/variables related to how the program behaves |
| 6. Exit | Exit the program |

#### 2. Sub-Program Design

Describe the design of each sub-program (e.g., word counting, sorting, etc.).

* List the sub-programs.
* Explain the purpose and logic of each sub-program.

| Sub-Program | Description |
| --- | --- |
| (for example)  split\_vocab() | (for example)  Splits the text into individual words. |
| ## nltk\_plagiarism.py module | (include functions for finding similarity between texts with Natural Language ToolKit (nltk)) |
| update | download/update the wordbanks used by nltk |
| preprocess\_text | Tokenize, remove stop words, and Lemmatize text |
| calculate\_similarity | calculate the cosine similarity of query text’s and reference texts’s features |
| bow\_features | vectorize a collection of texts to a matrix of Bag of Words (BoW) features (token counts) |
| tfidf\_features | vectorize a collection of texts to a matrix of Term Frequency - Inverse Document Frequency (TF-IDF) features |
| get\_similarity\_score | calculate similarity score of query text and reference texts |
| ## helpers.py module | (include functions that aren't directly related to operation of main.py) |
| animated\_print | Replaces normal print function with an animated typewriter-effect one |
| animated\_input | Replaces normal input function with an animated typewriter-effect one |
| quick\_sort | Sorts an input list using quick sort (because it is fast) |
| linear\_search | Searches for an item inside a list |
| split\_exclude\_ANSI | splits a string and returns the splitted string, but preserves ANSI codes (at least 3 characters long) by considering them as 1 character while splitting |
| max | Return the largest item in the input sequence |
| min | Return the smallest item in the input sequence |
| all | Return True if all item in the input sequence are True (or contains non-zero value), otherwise returns False |
| any | Return True if any item in the input sequence are True (or contains non-zero value), otherwise returns False |
| ## main.py | (common functions used in both CLI and GUI) |
| config | Read, write, and store application settings |
| config.reset\_to\_defaults | Reset all settings to their default values |
| config.save | Save current configuration settings to config file (WAPDS.config) |
| read\_file | Read a file, suppresses all errors related to retrieving a file’s content |
| clean\_text | Clean a text by truncating multiple consecutive spaces into one singular space, and remove punctuations and special characters |
| count\_words | Count the frequency of each word in the a text |
| search\_word\_position | Return all positions of word matching in a text, supports regex |
| sort\_alphabetically | Sort a processed word count tuple alphabetically and return the sorted tuple |
| sort\_by\_frequency | Sort a processed word count tuple by frequency and return the sorted tuple |
| calculate\_simularity | Get two processed word count tuples and return the similarity by dividing common word count by unique word count |
| ## main.py - WordAnalysisApp | (GUI) |
| \_\_init\_\_ | Initiate the GUI window and everything related |
| create\_analyze\_tab | Create the GUI for analyze tab |
| create\_compare\_tab | Create the GUI for compare tab |
| create\_search\_tab | Create the GUI for search tab |
| create\_replace\_tab | Create the GUI for replace tab |
| create\_config\_tab | Create the GUI for configuration tab |
| browse\_file1 | Triggers when browse button in analyze tab is pressed  Allow user to choose a file for analyze |
| browse\_compare\_file1 | Triggers when browse button of file1 in compare tab is pressed  Allow user to choose the first file for comparison |
| browse\_compare\_file2 | Triggers when browse button of file2 in compare tab is pressed (nltk off)  Allow user to choose the second file for comparison |
| browse\_compare\_file2\_nltk | Triggers when browse button of file2 in compare tab is pressed (nltk on)  Allow user to choose the files for reference |
| browse\_search\_file | Triggers when browse button in search tab is pressed  Allow user to choose a file to search |
| browse\_replace\_file | Triggers when browse button in replace tab is pressed  Allow user to choose a file to perform word replacement |
| analyze\_file | Triggers when analyze button in analyze tab is pressed  Updates the analyze tab to show analyze result, including total words, unique words, words sorted both alphabetically and by frequency, and a graph figure |
| create\_frequency\_graph | Creates a bar graph of word frequencies used in analyze tab |
| compare\_files | Triggers when compare files button in analyze tab is pressed  Updates the compare tab to show compare result, including total words and unique words for both files, plagiarism percentage between the two files, and a graph figure |
| create\_comparison\_graph | Create a comparison graph of word frequencies for compare tab with nltk mode off |
| create\_nltk\_comparison\_graph | Create a comparison graph of word frequencies for compare tab with nltk mode on |
| search\_word | Triggers when search button in search tab is pressed  Updates the search tab to show search result, including frequency of the input word/pattern, and their respective position |
| replace\_word | Triggers when replace button in replace tab is pressed  Updates the replace tab to show original text in the file, and the modified text after replacing, and allow user to further modify the text before saving |
| save\_modified\_text | Triggers when save to file button in replace tab is pressed  Save the modified text to a file |
| save\_config | Triggers when save button in configuration tab is pressed  Saves the configuration set in configuration tab |
| reset\_last\_save\_config | Triggers when cancel button in configuration tab is pressed  Cancel all unsaved edit in configuration tab |
| reset\_default\_config | Triggers when reset button in configuration tab is pressed  Reset all values in configuration tab to default and saves |
| ##main.py | (CLI) |
| configure\_test\_input | Prompt for input in CLI and validate the input |
| configure | Configure the settings for the program in CLI |
| search\_word | Search for a word in a file and display its positions in CLI |
| replace\_word | Replace occurrences of a target word with a replacement word in a file in CLI |
| display\_result | Display analysis results for single text file in CLI |
| compare\_files | Compare two text files and calculate their similarity percentage in CLI |
| analyze\_file | Analyze a single text file in CLI |
| ##main.py | (main entry point) |
| save\_window\_size | Record the current GUI window size upon resizing |
| GUI\_exit | Triggers when GUI window is closing  Confirm if user want to exit the application, if so, quit everything |
| mainGUI | Starts the GUI |
| mainCLI | Menu for the CLI, basically the main body |

#### 3. Function Comparison

Compare the functions used in the program (e.g., built-in vs. custom functions).

* List the functions used.
* Compare their efficiency, readability, and use.

| (for example) Sorting | (for example)  Selection Sort | (for example)  Bubble Sort | (for example)  Decision |
| --- | --- | --- | --- |
| Sorts words alphabetically |  | less efficient for large datasets. |  |

| Searching | Linear search | Binary search | Decision |
| --- | --- | --- | --- |
| Return the index of a specific item in an iterable  Used for searching index at the end of the animation cycle in animated input | Use for loop, can be used on any iterables  O(n) | Use double pointers, faster, requires iterable to be sorted  O(log n) | Linear Search  Time efficiency from binary search will be deducted from the sorting before it, and there are very few sorted iterables used in the program. Therefore linear search is in general faster than sorting+binary search |

| Sorting | Quick sort | Bubble sort | Decision |
| --- | --- | --- | --- |
| Sorts an iterable in ascending or descending order  Used for sorting words alphabetically and by frequency | Use recursion, faster, larger memory usage  O(n log n) | Use nested for loop, slower, less memory usage  O(n^2) | Quick sort  It’s faster  Memory usage is less prioritised than Time usage here |

#### 4. Final Source Code

Provide the final source code for the program.

* Paste the complete Python code.
* Include comments for clarity.

\* WAPDS.config will be generated upon first run, as no file is considered corrupted and a new one will be created.

Therefore WAPDS.config is not provided here

\* No pip installs (including the use of nltk\_plagiarism.py) are required for main.py to run, but some features will be unavailable.

\* All pip installs are for adding functionalities for GUI ONLY

\* In case the code is updated after the submission of this documentation, The code is also hosted on

[github](https://github.com/TaokyleYT/23-26_ICT_SBA) - private, accessible through token and git clone shown below, most up to date

and

[replit](https://replit.com/@s3D27ZHOU/5B28Blk1SBA) - public, because I don’t have replit core subscription. Less up to date due to replit’s limited development time (restricts duration of using replit to code)

And the descriptions of every subprogram is in their docstrings

Access and clone the github repository using

git clone https://TaokyleYT:github\_pat\_11AYPYTYQ0o4sOnNzaBZg7\_1GEYLn3vteaQUw05Rnh59qyQA93iNz0lb90v20zKg4zOEGP32FHOLFsXxRm@github.com/TaokyleYT/23-26\_ICT\_SBA

Git can be installed through

@REM windows

echo "y" | winget install Git.Git

#Unix/linux

sudo apt-get install git



nltk\_plagarism.py



import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

from nltk.stem import WordNetLemmatizer

from sklearn.feature\_extraction.text import CountVectorizer, TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

import os

from os.path import exists

# Set data directory for NLTK data

if os.name == 'nt':

data\_dir = \_\_file\_\_.rsplit("\\", 1)[0] + "\\nltk\_data"

else:

data\_dir = \_\_file\_\_.rsplit("/", 1)[0] + "/nltk\_data"

nltk.data.path.append(data\_dir)

def update():

"""

Update NLTK data if necessary.

This function checks if the NLTK data directory exists and downloads the required

data if it doesn't.

"""

if not exists(data\_dir + "/tokenizers"):

nltk.download('punkt', data\_dir)

nltk.download('punkt\_tab', data\_dir)

if not exists(data\_dir + "/corpora"):

nltk.download('stopwords', data\_dir)

nltk.download('wordnet', data\_dir)

def preprocess\_text(text:str) -> str:

"""

Preprocess a text by tokenizing, removing punctuation and stop words, and lemmatizing.

Args:

text (str): The text to preprocess.

Returns:

str: The preprocessed text.

"""

stop\_words = stopwords.words('english')

lemmatizer = WordNetLemmatizer()

# Tokenization

tokens = word\_tokenize(text)

# Removing punctuation and stop words

tokens = [word.lower() for word in tokens if word.isalnum() and word.lower() not in stop\_words]

# Lemmatization

tokens = [lemmatizer.lemmatize(word) for word in tokens]

# Join the tokens back into a single string

processed\_text = ' '.join(tokens)

return processed\_text

def calculate\_similarity(query\_features, reference\_features):

"""

Calculate the cosine similarity between two sets of features.

Args:

query\_features (scipy.sparse.csr\_matrix): The features of the query text.

reference\_features (scipy.sparse.csr\_matrix): The features of the reference texts.

Returns:

list: A list of similarity scores, one for each reference text.

"""

# Check dimensions and transpose if necessary

if query\_features.shape[1] != reference\_features.shape[1]:

reference\_features = reference\_features.T

# Check dimensions again after potential transposition

if query\_features.shape[1] != reference\_features.shape[1]:

raise ValueError("Incompatible dimensions for query and reference features")

similarity = cosine\_similarity(query\_features, reference\_features)

return similarity

def bow\_features(texts:str) -> tuple:

"""

Extract bag-of-words features from a list of texts.

Args:

texts (list): A list of texts.

Returns:

tuple: A tuple containing the features and the vectorizer.

"""

vectorizer = CountVectorizer()

features = vectorizer.fit\_transform(texts)

return features, vectorizer

def tfidf\_features(texts:str) -> tuple:

"""

Extract TF-IDF features from a list of texts.

Args:

texts (list): A list of texts.

Returns:

tuple: A tuple containing the features and the vectorizer.

"""

vectorizer = TfidfVectorizer()

features = vectorizer.fit\_transform(texts)

return features, vectorizer

def get\_similarity\_score(query\_text, reference\_texts):

"""

Calculate the cosine similarity between a query text and one or more reference texts.

Args:

query\_text (str): The query text.

reference\_texts (list): A list of reference texts.

Returns:

list: A list of tuples, where each tuple contains the reference text and its corresponding similarity score.

"""

preprocessed\_query = preprocess\_text(query\_text)

preprocessed\_references = [preprocess\_text(text) for text in reference\_texts]

# Extract TF-IDF features for query and reference texts

features\_query, vectorizer = tfidf\_features([preprocessed\_query] + preprocessed\_references)

features\_references = features\_query[1:]

features\_query = features\_query[:1] # Extract query feature separately

# Calculate similarity

similarity\_scores = calculate\_similarity(features\_query, features\_references)

# Identify plagiarized content

plagiarism\_results = []

for i, score in enumerate(similarity\_scores[0]):

plagiarism\_results.append([reference\_texts[i], score])

return plagiarism\_results

if \_\_name\_\_ == "\_\_main\_\_":

update()

# Read example document from file

with open("test2\_1.txt", "r") as file:

example\_document = file.read()

# Define the reference texts

reference\_texts = []

# Read reference texts from files

files = ["test2\_2.txt", "test2\_3.txt", "test2\_4.txt"]

for file in files:

with open(file, "r") as f:

reference\_texts.append(f.read())

try:

# Test plagiarism detection

results = get\_similarity\_score(example\_document, reference\_texts)

# Print results

if results:

print("Plagiarized content detected:")

for result in results:

print(f"Similarity Score: {(result[1]\*100):.2f}%")

print()

else:

print("No plagiarism detected.")

except ValueError as e:

print("Error:", e)

helpers.py

from typing import Iterable # Import Iterable types for type hinting

import time # For implementing delays in output

import os # For accessing and manipulating the file system

import sys # For system-specific parameters and functions

import string # For string constants and character classifications

import re # For regular expression operations for string matching

# Platform-specific imports for terminal control to change input mode and handle character printing

if sys.platform == "win32":

import ctypes # Required for Windows console control

import ctypes.wintypes # Required for Windows data types

else:

import termios # For controlling terminal settings on Unix/Linux

import tty # For setting terminal to character-by-character input mode

def animated\_print(txt: str | Iterable[str] | Iterable[Iterable[str]] = "",

end: str = "\n",

delay: float = 0.01,

line\_offset: int = 1,

\_override: bool = False,

wrap\_override: bool = False) -> None:

"""

Prints text with an animation effect, simulating a typewriter-style output.

This function displays text character by character with a delay between each character,

creating a typewriter-like animation effect. It handles multi-line text, text wrapping,

and preserves ANSI escape sequences for colored or formatted text.

printing steps:

repeat:

↟ to the top

repeat:

→ prints ← ↲

until reach line that is not yet at their turn to print or after last line

until everything is printed out

per iteration:

```

↑→→→

↑ ↲

↑ ↲

↑↲

```

Warning: Multi-line text including ANSI codes may not output as expected, as

full support would require significantly more complex implementation.

More Warning: line wrapping, despite implemented, seldom fail for whatever reason

and I can't seem to fix it without breaking something else.

Please use a bigger terminal or a smaller font size to avoid breaking everything that is being outputted

Args:

txt (str | Iterable[str] | Iterable[Iterable[str]]):

The text to print. Can be a single string, a list of strings, or a nested list of strings.

end (str):

A string to print after the entire text has been animated (default: newline).

delay (float):

The time in seconds between each character print (default: 0.01s).

line\_offset (int):

The number of lines to offset the text vertically for animation (default: 1).

\_override (bool):

An internal parameter for optimization during recursion to override certain type checks.

Not intended for direct use.

wrap\_override (bool):

When True, enables line wrapping even when \_override is True (default: False).

Intended for direct use when you touched the "Not intended for direct use" \_override parameter

Raises:

TypeError: If `end` is not a string, `delay` is not a number, or `txt` is not

a string or an iterable of strings.

Returns:

None

"""

# Validate input parameters to ensure correct types

if not isinstance(end, str):

raise TypeError("end must be a string")

if not isinstance(delay, (int, float)):

raise TypeError("delay must be a number")

# Handle case where no text is provided

if len(txt) == 0:

print(end=end) # Print the "end" string (newline or other)

return # Exit function to avoid additional processing

# Process and validate the input text

if all(

(all(isinstance(subchar, str) for subchar in char) # Ensure nested strings in iterables

if (isinstance(char, (list, tuple)) and \_override) # Handling nested iterables

else isinstance(char, str))

for char in txt): # Ensure all elements are strings

if all(len(char) == 1 for char in txt): # If all items are single characters

txt = (''.join(txt)).split('\n') # Merge the txt and split it into lines

else:

txt = list(txt) # Convert txt to a list for processing

else:

raise TypeError("txt must be a str or iterable of str") # Raise an error for incorrect types

# Get terminal size for proper text wrapping

term\_size = os.get\_terminal\_size()

txt\_lst = [] # Initialize a list to hold processed text lines

# Process each line for wrapping and animation

for line in txt:

if \_override and not wrap\_override: # Skip processing if override is active and wrap\_override is False

txt\_lst = txt # Skip additional processing

break

if not line:

txt\_lst.append("") # Handle empty line

continue # Continue to next line

if \_override and wrap\_override:

# Handle case where line is already a list of characters

txt\_lst\_temp = [] # Temporary list for wrapped characters

current\_line = [] # Current line being built

for char in line:

current\_line.append(char)

if len(current\_line) >= term\_size.columns - 1:

txt\_lst\_temp.append(current\_line)

current\_line = []

if current\_line: # Add any remaining characters

txt\_lst\_temp.append(current\_line)

txt\_lst.extend(txt\_lst\_temp) # Add wrapped lines to main list

else:

# Split line by spaces while excluding ANSI escape sequences

temp = split\_exclude\_ANSI(line, " ")

words: list[str] = [(item if i == len(temp)-1 else item + " ")

for i, item in enumerate(temp)] # Append space to all but last word

index: int = 0 # Initialize index for processing words

# Wrap text according to terminal width

while index < len(words):

if len(words[index]) > term\_size.columns - 1:

# Break long words that exceed terminal width into chunks

for wrapped\_line in (

line[i:i + term\_size.columns - 1]

for i in range(0, len(line), term\_size.columns - 1)):

txt\_lst.append(wrapped\_line) # Append wrapped line to list

index += 1

continue

# Add words to line until it exceeds terminal width

txt\_lst.append("") # Start new line

for i, word in enumerate(words[index:]):

if len(txt\_lst[-1]) + len(word) > term\_size.columns - 1:

index += i # Update index for next processing

break # Break word addition

txt\_lst[-1] += word

else:

break # Exit while loop if all words processed

if not \_override:

# Split each processed line into characters while preserving ANSI codes

txt\_lst = [split\_exclude\_ANSI(line) for line in txt\_lst]

# Truncate text if it exceeds terminal height

txt\_lst, truncated = txt\_lst[:term\_size.lines - 1], txt\_lst[term\_size.lines - 1:]

# Find the maximum line length for animation timing

max\_wordlen = max(len(line) for line in txt\_lst)

# Print spaces for the animation based on the number of lines

print('\n' \* (len(txt\_lst)), end='\x1b[A') # Move cursor up to start animation

# Animate the text character by character

for i in range(max\_wordlen + line\_offset \* len(txt\_lst)):

# Move cursor up to the top of the animation output

print("\x1b[A" \* (len(txt\_lst) - 1), end='')

time.sleep(delay) # Pause between characters

for j, line in enumerate(txt\_lst):

current\_char = j \* line\_offset + 1 # Track the current character position

# Update lines that need to be redrawn

if (i - current\_char >= 0 and i - current\_char < len(line)):

# Print current character with front effect

print(

"\x1b[" + str(current\_char) + "D" + # Move cursor left to overwrite character

line[i - current\_char] + # Print the current character

"\x1b[" + str(current\_char) + "C" + # Move cursor right

"\b\x1b[B", # Move cursor to the next line

end='',

flush=True) # Ensure immediate printing

else:

print("\x1b[B", end='', flush=True) # Move line without printing

print("\x1b[A\x1b[C", end='', flush=True) # Move cursor back up for the next iteration

time.sleep(delay) # Additional delay to complete character output

# Reset cursor position for any following outputs

print("\x1b[" + str(line\_offset \* len(txt\_lst)) + "D",

end='\n' if truncated else '',

flush=True)

# Recursively handle any truncated text for complete output

animated\_print(truncated, end, delay, line\_offset, \_override=True, wrap\_override=wrap\_override)

return

def animated\_input(prompt: str = "",

delay: float = 0.01,

line\_offset: int = 1,

single\_letter: bool = False,

\_log: bool = False) -> str:

"""

Animated version of input() that displays a prompt with animation effects.

This function displays an animated prompt and accepts user input with

character-by-character animation. Special keys like backspace are handled,

and ANSI escape sequences in the input are supported.

Warning: Multi-line prompts including ANSI codes may not output as expected

due to implementation complexity.

Note that this input does not support multi-line input, just like the original one

Args:

prompt (str):

Text to display before the input field.

delay (float):

Time in seconds between character animations (default: 0.01s).

line\_offset (int):

Number of lines to offset the text vertically (default: 1).

single\_letter (bool):

Whether to accept a single letter input and return immediately (default: False).

If True, skips animation and returns after a single keypress.

\_log (bool):

Whether to log input for debugging purposes (internal use, default: False).

When enabled, writes input events to "input\_log.txt".

Returns:

str: The user's input string (without the trailing newline).

"""

# Display the prompt with animated typing effect

animated\_print(prompt, "", delay, line\_offset)

# Get terminal width for text wrapping

columns = os.get\_terminal\_size().columns - 2 # Adjust for prompt/other text

# Set terminal to character-by-character input mode (raw mode)

if sys.platform == "win32": # Windows implementation

dword = ctypes.wintypes.DWORD()

kernel = ctypes.windll.kernel32

kernel.GetConsoleMode(kernel.GetStdHandle(-10), ctypes.byref(dword))

kernel.SetConsoleMode(kernel.GetStdHandle(-10), 0) # Set to raw mode

kernel.GetConsoleMode(kernel.GetStdHandle(-11), ctypes.byref(dword))

kernel.SetConsoleMode(kernel.GetStdHandle(-11), 7) # Enable ANSI support

else: # Unix/Linux implementation

stdin = sys.stdin.fileno() # Get standard input file descriptor

original\_term = termios.tcgetattr(stdin) # Get current terminal attributes

tty.setcbreak(stdin, termios.TCSANOW) # Set terminal to raw mode for character input

# Check if single letter input mode is enabled

if single\_letter:

if \_log: # If logging is enabled, write to log file

with open("input\_log.txt", "a") as f:

f.write("\n-----\nInput started (single letter)\n-----\n")

result = "\n" # Initialize result with newline

while result == "\n": # Await input until a character is provided

# This loop is to handle Windows stdin buffering issues

# On Windows, sometimes need to press enter twice due to buffer behavior

result = sys.stdin.read(1) # Read one character from stdin

# Handle keyboard Interrupt (Ctrl+C and Ctrl+D)

if result in "\x03\x04":

if \_log: # If logging is enabled, note the event

with open("input\_log.txt", "a") as f:

f.write(f"KeyboardInterrupt with {result}\n")

raise KeyboardInterrupt # Raise KeyboardInterrupt if Ctrl+C is pressed

print(result) # Print the character immediately

# Restore terminal state

if sys.platform == "win32": # Windows restoration

kernel.SetConsoleMode(kernel.GetStdHandle(-10), dword)

kernel.SetConsoleMode(kernel.GetStdHandle(-11), dword)

else: # Unix/Linux restoration

termios.tcsetattr(stdin, termios.TCSANOW, original\_term)

if \_log: # Log the submitted input

with open("input\_log.txt", "a") as f:

f.write(f"submitted {repr(result)} \n")

return result # Return the single character

# Get cursor position for multi-character input

sys.stdout.write("\x1b[?25l\x1b[6n") # Hide cursor and request cursor position

sys.stdout.flush() # Ensure stdout is updated

result = sys.stdin.read(1) # Read initial character

while not result.endswith("R"): # Await complete cursor position response

result += sys.stdin.read(1) # Append additional read characters

# Parse cursor position response from terminal

reg = re.match(r"^\x1b\[(\d\*);(\d\*)R", result) # Extract row and column from ANSI response

ptr = int(reg.groups()[1]) if reg else 1 # Get current cursor position

# Prepare character sets for animation

printables = [" "] + list(string.printable)[:-6] # Include printable ASCII characters

output = sys.stdout.write # Alias for writing output

# Read input character by character until the Enter key

char = sys.stdin.read(1) # Read first character

result = "" # Input string to gather characters

ansi = "" # Holder for ANSI escape sequences

if \_log: # Log input start

with open("input\_log.txt", "a") as f:

f.write("\n-----\nInput started\n-----\n")

# Main input loop

while char != "\n": # Continue until Enter key

# Handle Keyboard Interrupt (Ctrl+C and Ctrl+D)

if char in "\x03\x04": # Ctrl+C or Ctrl+D

if \_log: # Log the keyboard interrupt

with open("input\_log.txt", "a") as f:

f.write(f"KeyboardInterrupt with {char}\n")

raise KeyboardInterrupt

# Handle backspace character for deleting input

if char in "\x7f\x08":

if len(result) > 0: # If there is a character to backspace

if \_log: # Log the deletion action

with open("input\_log.txt", "a") as f:

f.write("del\n")

result = result[:-1] # Remove last character from result

if ptr % columns != 0: # If not at start of the line

output("\b \b") # Erase character on same line

else: # If at the start of the line

output(f"\x1b[F\x1b[{columns-1}G\b \b") # Move up one line and back to the last character

sys.stdout.flush() # Show updated result immediately

ptr -= 1 # Move cursor back

else: # If a normal input character

if \_log: # Log addition of character

with open("input\_log.txt", "a") as f:

f.write(f"add {repr(char)} \n")

result += char # Add character to result

if ptr % columns == 0: # If at end of terminal width

output(" \n") # Create a new line to continue input

# Handle ANSI escape sequences explicitly here

if char == "\x1b": # Start of ANSI escape sequence

ansi = char # Store starting ANSI sequence

elif ansi: # Continue building ANSI sequence

ansi += char

# Determine animation length based on character type

if char in printables: # If character is printable

end\_cyc = linear\_search(printables, char) + 1 # Index for animation limit

elif char in "\0 \b\n" or ansi: # Handle special characters and ANSI

end\_cyc = 0

else: # If not special

end\_cyc = len(printables)

# Animation effect: cycle through characters

for c in printables[:end\_cyc]: # Show each character in printables

output(ansi) # Print any ANSI information

output(c + "\b") # Show character then backspace to show just the animated char

sys.stdout.flush() # Prompt for immediate output

time.sleep(delay / 10) # Shorter sleep for quicker animation

# Handle final character display

if ansi == "":

output(char) # Print character if no ANSI involved

elif ansi == "\x1b[" or ansi[-1] < "@" or ansi[-1] > "~":

pass # Incomplete ANSI sequence, do nothing

else:

output(result + " \b") # Display result string followed by backspace

ansi = "" # Reset ANSI storage for next sequence

sys.stdout.flush() # Prompt for immediate output

ptr += 1 # Increment cursor position

char = sys.stdin.read(1) # Read next character

output("\n") # Final newline to complete input

# Restore terminal state to original settings based on platform

if sys.platform == "win32": # Windows restoration

kernel.SetConsoleMode(kernel.GetStdHandle(-10), dword)

kernel.SetConsoleMode(kernel.GetStdHandle(-11), dword)

else: # Unix/Linux restoration

termios.tcsetattr(stdin, termios.TCSANOW, original\_term)

# Log final result if logging is enabled

if \_log:

with open("input\_log.txt", "a") as f:

f.write(f"submitted {repr(result)} \n")

return result # Return the string input by the user

def quick\_sort(iterable: Iterable, /, \*, key = None, reverse: bool = True) -> list:

"""

Sorts a list using the quick sort algorithm.

This is a recursive implementation of the quick sort algorithm that uses

the first element as the pivot. It creates new lists rather than sorting in-place.

Args:

Iterable (Iterable):

The list or sequence to sort.

key (callable):

A key function to extract a comparison key from each element (default: None).

reverse (bool):

Sort in ascending order if True, otherwise in descending order (default: True).

Returns:

list: A new sorted list.

Time Complexity:

- Average case: O(n log n)

- Worst case: O(n^2) when the list is already sorted

Space Complexity:

O(n) due to the creation of new lists during recursion.

"""

# Base case: If list has 1 or 0 items, it's already sorted

if len(iterable) < 2:

return iterable

pivot = iterable[0] # Choose the first element as the pivot

comp\_pivot = pivot if key == None else key(pivot)

less = [] # List to hold elements less than the pivot

more = [] # List to hold elements greater than or equal to the pivot

# Divide the input list into smaller partitions based on the pivot

for item in iterable[1:]:

comp\_item = item if key is None else key(item)

if not reverse:

if comp\_item < comp\_pivot:

less.append(item) # Add to "less" if it's smaller than the pivot

else:

more.append(item) # Add to "more" otherwise

else:

if comp\_item > comp\_pivot:

less.append(item) # Add to "less" if it's larger than the pivot (cuz reverse)

else:

more.append(item) # Add to "more" otherwise

# Recursively sort both partitions and combine them

# Reverse the result if descending order is requested

return (list(quick\_sort(less)) + [pivot] +

list(quick\_sort(more)))

def linear\_search(iterable: list,

value,

start=0,

stop=9223372036854775807,

/): # The '/' indicates that the parameters to the left cannot be passed as keyword arguments.

"""

Performs a linear search for a value in a list.

Sequentially checks each element in the list until it finds a match or reaches the end.

Allows specifying a range to search within.

Args:

iterable (list):

The list to search in.

value:

The value to search for.

start (int):

Starting index for the search (default: 0).

stop (int):

Ending index (exclusive) for the search

(default: 9223372036854775807, which is 2^63-1,

aka largest signed integer in 64 bit system, I copied it straight out from list.index docs,

to search the entire list).

Returns:

int: Index of the first occurrence of `value` if found, -1 otherwise.

Time Complexity:

O(n) where n is the number of elements in the specified search range.

"""

# Ensure `stop` does not exceed the length of the list

stop = min(stop, len(iterable))

# Check each element from start to stop

for n in range(start, stop):

if iterable[n] == value: # If element matches the searched value

return n # Return the index

return -1 # Return -1 if the value is not found

def split\_exclude\_ANSI(text: str, sep: str | list[str] | tuple[str] = ""):

"""

Splits a string by separator(s) while preserving ANSI escape sequences.

This functions similar to `str.split()` but retains ANSI escape codes within each

resulting substring. If the separator is empty, it splits into individual characters

while keeping ANSI sequences together.

Args:

text (str):

The string to split.

sep (str | list[str] | tuple[str]):

Separator(s) to split by. Can be a string or an iterable (list or tuple) of strings.

If empty string, it splits into individual characters.

Returns:

list[str]: A list of strings after splitting by separator(s).

Raises:

TypeError: If `sep` is not a string, a list of strings, or a tuple of strings.

Example:

>>> split\_exclude\_ANSI("\x1b[31mHello\x1b[0m World", "")

['\x1b[31m', 'Hello, '\x1b[0m', 'World']

>>> split\_exclude\_ANSI("\x1b[31mHello\x1b[0m World", " ")

['\x1b[31mHello\x1b[0m', 'World']

"""

# Check the type of `sep` to ensure it is valid

if not (isinstance(sep, str) or all(isinstance(s, str) for s in sep)):

raise TypeError(

"sep should be either string or an iterable that contains only strings"

)

# Handle case where text is empty

if not text:

return [] # Return an empty list if there is no text

# Convert `sep` to a list for uniform handling

separators = [sep] if isinstance(sep, str) else list(sep)

# Initialize variables for splitting logic

result = [] # Resultant list of split strings

i = 0 # Index to traverse the text

start = 0 # Start index for current segment

in\_ansi = False # Flag indicating if we are within an ANSI sequence

ansi\_start = -1 # Start index of the ANSI sequence

while i < len(text):

# Check if entering an ANSI escape sequence

if text[i] == '\x1b' and i + 1 < len(text) and text[i + 1] == '[':

in\_ansi = True # Set flag indicating we are inside an ANSI sequence

ansi\_start = i # Mark the start index of the ANSI sequence

# Check if at the end of an ANSI sequence

if in\_ansi and i > ansi\_start + 1 and text[i] > "@" and text[i] < "~":

in\_ansi = False # Reset flag when we exit the ANSI sequence

# Split logic outside of ANSI sequences

if not in\_ansi:

# Special case: if separator is empty, split into individual characters

if separators == [""]:

i += 1 # Increment index to read next character

# Check for any specified separator matches

for separator in separators:

if i + len(separator) <= len(text) and text[i:i + len(separator)] == separator:

result.append(text[start:i]) # Append current segment

i += len(separator) - 1 # Move index past the separator

start = i + 1 # Update start for the next segment

break # Break to re-check for more separators as needed

i += 1 # Advance the iterator

# Add the last segment post-loop

result.append(text[start:])

# Remove any trailing empty strings in the result

while result and result[-1] == "":

del result[-1] # Delete empty strings at the end of the list

return result # Return the list of split sections

def max(\*args):

"""

Returns the largest item in an iterable or the largest of multiple arguments.

This function reimplements the built-in max() to allow use without directly importing

built-in functions or creating excessively large loops.

Args:

\*args: Accepts either a single iterable or multiple arguments to compare.

Returns:

The largest item among those provided.

Raises:

TypeError: If no arguments are provided.

Example:

>>> max(1, 2, 3)

3

>>> max([1, 2, 3])

3

"""

if not args: # Check if no arguments were received

raise TypeError("max expected at least 1 argument, 0 received")

elif len(args) == 1: # Handle single iterable input

args = args[0]

if isinstance(args, Iterable): # Convert to list if it's an iterable

args = list(args)

else:

return args # Directly return the value if it's a single non-iterable argument

maximum = args[0] # Initialize maximum with the first argument

for n in args: # Loop to find maximum

if n > maximum: # If current value is greater than maximum

maximum = n # Update maximum

return maximum # Return the largest value found

def min(\*args):

"""

Returns the smallest item in an iterable or the smallest of multiple arguments.

This function reimplements the built-in min() to allow use without directly importing

built-in functions or creating excessively large loops.

Args:

\*args: Accepts either a single iterable or multiple arguments to compare.

Returns:

The smallest item among those provided.

Raises:

TypeError: If no arguments are provided.

Example:

>>> min(1, 2, 3)

1

>>> min([1, 2, 3])

1

"""

if not args: # Check if no arguments were received

raise TypeError("min expected at least 1 argument, 0 received")

elif len(args) == 1: # Handle single iterable input

args = args[0]

if isinstance(args, Iterable): # Convert to list if it's an iterable

args = list(args)

else:

return args # Directly return the value if it's a single non-iterable argument

minimum = args[0] # Initialize minimum with the first argument

for n in args: # Loop to find minimum

if n < minimum: # If current value is smaller than minimum

minimum = n # Update minimum

return minimum # Return the smallest value found

def all(\*args):

"""

Returns True if all elements of the iterable are true (or if iterable is empty).

This function reimplements the built-in all() to allow use without importing

built-in functions or creating excessively large loops.

Args:

\*args: Accepts either a single iterable or multiple arguments to check.

Returns:

bool: True if all elements are true, False otherwise.

Example:

>>> all([True, True, True])

True

>>> all([True, False, True])

False

"""

if not args: # Check if no arguments were received

return True # Return True since empty iterables are considered "all true"

elif len(args) == 1: # Handle single iterable input

args = args[0]

if isinstance(args, Iterable): # Convert iterable to list

args = list(args)

else:

return args # Directly return value if it's a single non-iterable argument

for n in args: # Iterate through each argument

if not n: # If any argument evaluates to False

return False # return False

return True # All elements are true

def any(\*args):

"""

Returns True if any element of the iterable is true.

This function reimplements the built-in any() to allow use without directly importing

built-in functions or creating excessively large loops. If the iterable is empty, returns False.

Args:

\*args: Accepts either a single iterable or multiple arguments to check.

Returns:

bool: True if any element is true, False otherwise.

Example:

>>> any([False, False, True])

True

>>> any([False, False, False])

False

"""

if not args: # Check if no arguments were received

return False # Return False since empty iterables are considered "no true values"

elif len(args) == 1: # Handle single iterable input

args = args[0]

if isinstance(args, Iterable): # Convert iterable to list

args = list(args)

else:

return args # Directly return value if it's a single non-iterable argument

for n in args: # Iterate through each argument

if n: # If any argument evaluates to True

return True # return True

return False # Neither element is true

main.py



