School of Mechanical & Manufacturing Engineering

FOP Project:

Group Members:

#	Name	CMS
1	Muhammad Muzammil Riaz	467817
2	Ramzan Sameer	464899
3	Romaisa Yaqoob	469297
4	Sameen Waseem	465064

Course: CS-116 Fundamentals of Programming II

Batch: ME-15

Section: A

Submitted to: Sir. Saqib Nazir

Date:19/05/2024

Task:

You will use object-oriented programming (classes and inheritance) to build a program to monitor news feeds over the Internet. Your program will filter the news, alerting the user when it notices a news story that matches that user's interests.

Code:

```
import feedparser

import string
import time
import threading
from project_util import translate_html
from tkinter import *
from datetime import datetime

#-----------------------------------------------------------------------

#======================
# Code for retrieving and parsing
# Google and Yahoo News feeds
#======================

def process(url):
    """
    Fetches news items from the rss url and parses them.
    Returns a list of NewsStory instances.
    """
    feed = feedparser.parse(url)
    entries = feed.entries
    ret = []
    for entry in entries:
        guid = entry.guid
        title = translate_html(entry.title)
        link = entry.link

        # Check if description field exists
        if 'description' in entry:
            description = translate_html(entry.description)
        else:
            description = ""

        # Handling different date formats
        if 'published' in entry:
            pubdate_str = entry.published
        elif 'published_parsed' in entry:
            pubdate_str = time.strftime('%a, %d %b %Y %H:%M:%S %Z',
entry.published_parsed)
        else:
            continue

        try:
            pubdate = datetime.strptime(pubdate_str, "%a, %d %b %Y %H:%M:%S %Z")
        except ValueError:
            pubdate = datetime.strptime(pubdate_str, "%Y-%m-%dT%H:%M:%SZ")

        newsStory = NewsStory(guid, title, description, link, pubdate)
        ret.append(newsStory)
    return ret
```

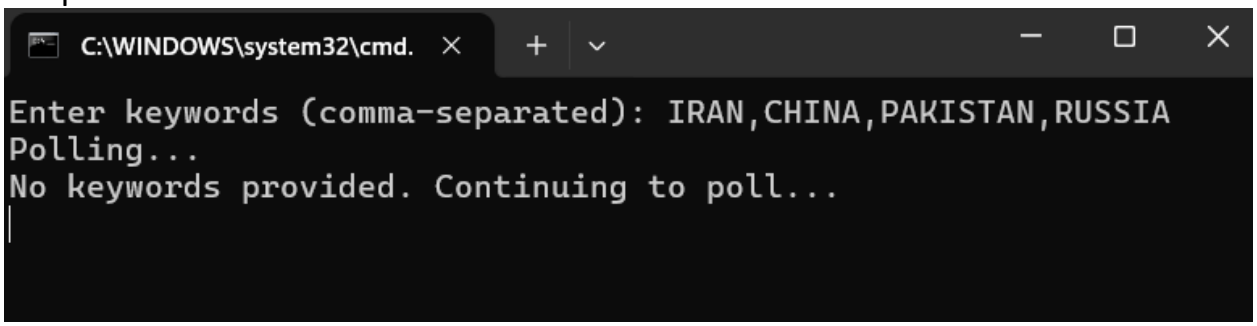
```
#======================
# Data structure design
#======================

class NewsStory:
    def __init__(self, guid, title, description, link, pubdate):
        self.guid = guid
        self.title = title
        self.description = description
        self.link = link
        self.pubdate = pubdate

    def get_guid(self):
        return self.guid

    def get_title(self):
        return self.title

    def get_description(self):
        return self.description

    def get_link(self):
        return self.link

    def get_pubdate(self):
        return self.pubdate

#======================
# Triggers
#======================

class Trigger(object):
    def evaluate(self, story):
        raise NotImplementedError

class PhraseTrigger(Trigger):
    def __init__(self, phrase):
        self.phrase = phrase.lower()

    def is_phrase_in(self, text):
        text = text.lower()
        for char in string.punctuation:
            text = text.replace(char, ' ')
        text_words = text.split()
        phrase_words = self.phrase.split()
        for i in range(len(text_words) - len(phrase_words) + 1):
            if text_words[i:i + len(phrase_words)] == phrase_words:
                return True
        return False

class TitleTrigger(PhraseTrigger):
    def evaluate(self, story):
        return self.is_phrase_in(story.get_title())

class DescriptionTrigger(PhraseTrigger):
    def evaluate(self, story):
        return self.is_phrase_in(story.get_description())

class TimeTrigger(Trigger):
    def __init__(self, time):
        self.time = datetime.strptime(time, "%Y-%m-%dT%H:%M:%SZ")

class BeforeTrigger(TimeTrigger):
    def evaluate(self, story):
        return story.get_pubdate() < self.time
```

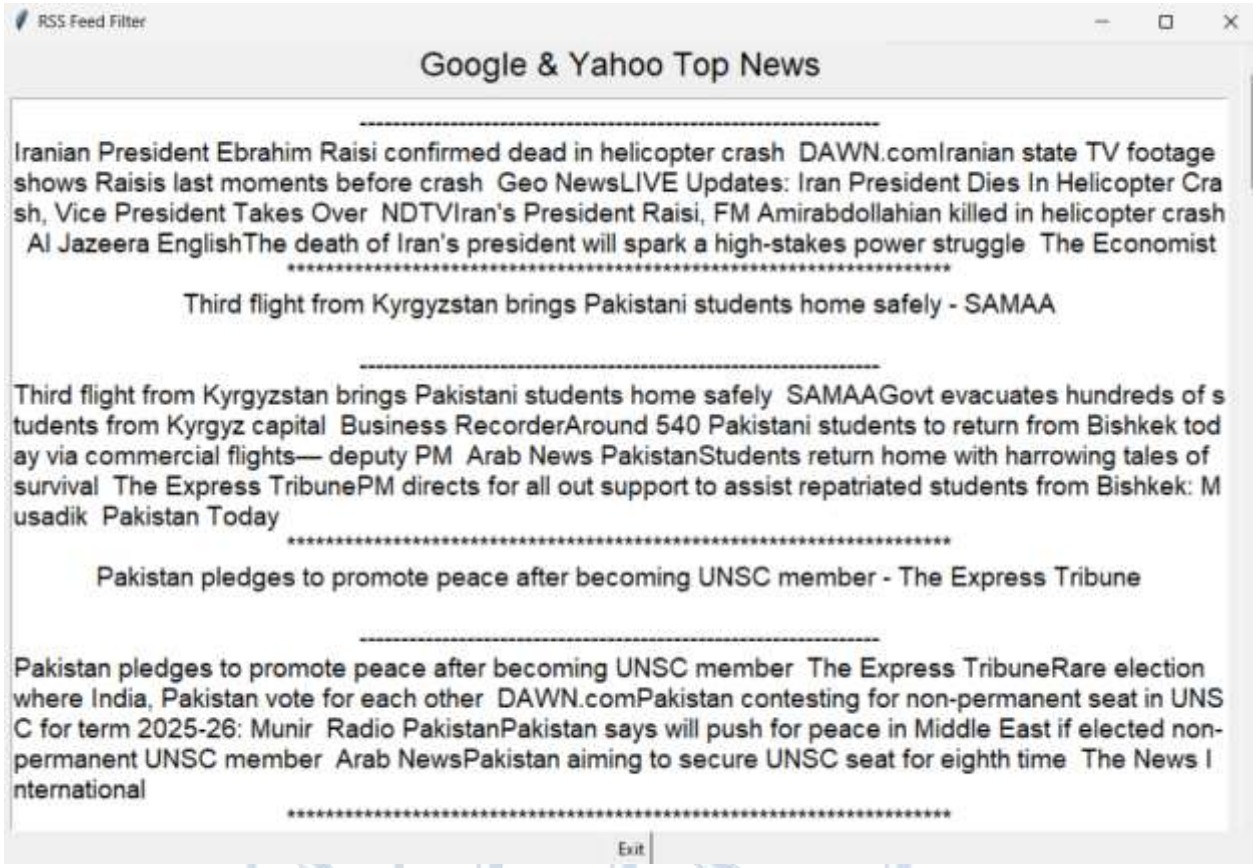
```
class AfterTrigger(TimeTrigger):
    def evaluate(self, story):
        return story.get_pubdate() > self.time

class NotTrigger(Trigger):
    def __init__(self, trigger):
        self.trigger = trigger

    def evaluate(self, story):
        return not self.trigger.evaluate(story)

class AndTrigger(Trigger):
    def __init__(self, trigger1, trigger2):
        self.trigger1 = trigger1
        self.trigger2 = trigger2

    def evaluate(self, story):
        return self.trigger1.evaluate(story) and self.trigger2.evaluate(story)

class OrTrigger(Trigger):
    def __init__(self, trigger1, trigger2):
        self.trigger1 = trigger1
        self.trigger2 = trigger2

    def evaluate(self, story):
        return self.trigger1.evaluate(story) or self.trigger2.evaluate(story)

#======================
# Filtering
#======================

def filter_stories(stories, triggerlist):
    filtered_stories = []
    for story in stories:
        for trigger in triggerlist:
            if trigger.evaluate(story):
                filtered_stories.append(story)
                break
    return filtered_stories

#======================
# User-Specified Triggers
#======================

def read_trigger_config(filename):
    trigger_file = open(filename, 'r')
    lines = []
    for line in trigger_file:
        line = line.rstrip()
        if not (len(line) == 0 or line.startswith('//')):
            lines.append(line)
    trigger_file.close()

    triggers = {}
    trigger_list = []

    for line in lines:
        parts = line.split(',')
        if parts[0] == 'ADD':
            for name in parts[1:]:
                if name in triggers:
                    trigger_list.append(triggers[name])
        else:
            trigger_name = parts[0]
            trigger_type = parts[1]
            if trigger_type == 'TITLE':
                triggers[trigger_name] = TitleTrigger(parts[2])
```

```
            elif trigger_type == 'DESCRIPTION':
                triggers[trigger_name] = DescriptionTrigger(parts[2])
            elif trigger_type == 'AFTER':
                triggers[trigger_name] = AfterTrigger(parts[2])
            elif trigger_type == 'BEFORE':
                triggers[trigger_name] = BeforeTrigger(parts[2])
            elif trigger_type == 'NOT':
                if parts[2] in triggers:
                    triggers[trigger_name] = NotTrigger(triggers[parts[2]])
            elif trigger_type == 'AND':
                if parts[2] in triggers and parts[3] in triggers:
                    triggers[trigger_name] = AndTrigger(triggers[parts[2]],
triggers[parts[3]])
            elif trigger_type == 'OR':
                if parts[2] in triggers and parts[3] in triggers:
                    triggers[trigger_name] = OrTrigger(triggers[parts[2]],
triggers[parts[3]])

    return trigger_list

#======================
# Main Thread
#======================

SLEEPTIME = 120  # seconds

def main_thread(master, keywords):
    try:
        triggerlist = []
        if keywords:
            for keyword in keywords:
                triggerlist.append(OrTrigger(TitleTrigger(keyword),
DescriptionTrigger(keyword)))

        frame = Frame(master)
        frame.pack(side=BOTTOM)
        scrollbar = Scrollbar(master)
        scrollbar.pack(side=RIGHT, fill=Y)

        t = "Google & Yahoo Top News"
        title = StringVar()
        title.set(t)
        ttl = Label(master, textvariable=title, font=("Helvetica", 18))
        ttl.pack(side=TOP)
        cont = Text(master, font=("Helvetica", 14), yscrollcommand=scrollbar.set)
        cont.pack(side=BOTTOM)
        cont.tag_config("title", justify='center')
        button = Button(frame, text="Exit", command=master.destroy)
        button.pack(side=BOTTOM)
        guidShown = []

        def get_cont(newstory):
            if newstory.get_guid() not in guidShown:
                cont.insert(END, newstory.get_title() + "\n", "title")
                cont.insert(END, "\n----------------------------------------
------------------\n", "title")
                cont.insert(END, newstory.get_description())
                cont.insert(END,
"\n**********************************************************************\n",
"title")
                guidShown.append(newstory.get_guid())

        while True:
            print("Polling...")
            stories = process("http://news.google.com/news?output=rss")
            stories.extend(process("http://news.yahoo.com/rss/topstories"))
```

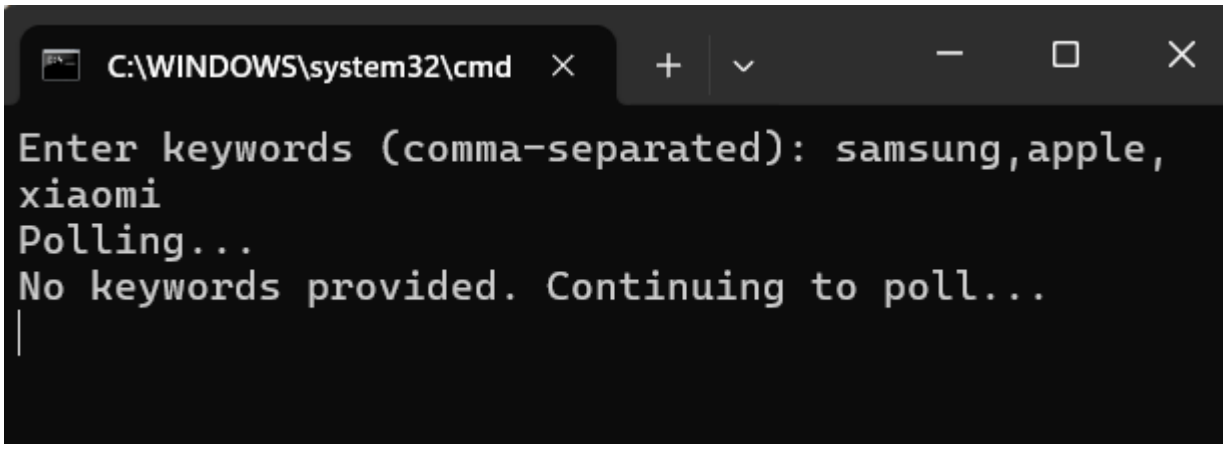
```
            stories = filter_stories(stories, triggerlist)

            list(map(get_cont, stories))
            scrollbar.config(command=cont.yview)

            print(f"No keywords provided. Continuing to poll...")
            time.sleep(SLEEPTIME)

    except Exception as e:
        print(f"Error occurred: {e}")


        def get_cont(newstory):
            if newstory.get_guid() not in guidShown:
                cont.insert(END, newstory.get_title() + "\n", "title")
                cont.insert(END, "\n----------------------------------------
-----------------\n", "title")
                cont.insert(END, newstory.get_description())
                cont.insert(END,
"\n*********************************************************************\n",
"title")
                guidShown.append(newstory.get_guid())

        while True:
            print("Polling...")
            stories = process("http://news.google.com/news?output=rss")
            stories.extend(process("http://news.yahoo.com/rss/topstories"))

            stories = filter_stories(stories, triggerlist)

            list(map(get_cont, stories))
            scrollbar.config(command=cont.yview)

            print(f"No keywords provided. Continuing to poll...")
            time.sleep(SLEEPTIME)

    except Exception as e:
        print(f"Error occurred: {e}")

if __name__ == '__main__':
    root = Tk()
    root.title("RSS Feed Filter")

    keywords = input("Enter keywords (comma-separated): ").strip().split(',')
    keywords = [keyword.strip() for keyword in keywords if keyword.strip()]

    t = threading.Thread(target=main_thread, args=(root, keywords))
    t.start()

    root.mainloop()
```

Output 1:





Output 2:



Explanation of the code:

A) Code for Retrieving and Parsing Google and Yahoo News Feeds

PROBLEM#1: Parsing

This section of the code is responsible for fetching and parsing news items from specified RSS feed URLs. It defines the process function which takes a URL, retrieves the feed, and parses the news items into instances of the NewsStory class. Here's a breakdown:

- Fetching and Parsing: The feedparser.parse(url) function is used to fetch and parse the RSS feed from the given URL.
- Extracting Entries: It iterates through the feed entries to extract details such as guid, title, link, description, and pubdate.
- Handling Descriptions and Dates: It checks if the description field exists and handles different date formats.
- Creating NewsStory Instances: Each entry is converted into a NewsStory instance, which is then added to the list of news stories.

B) Data and Structural Design

This part of the code defines the NewsStory class which encapsulates the data structure for storing news items. It includes:

- Constructor and Attributes: The __init__ method initializes a NewsStory object with attributes like guid, title, description, link, and pubdate.
- Getter Methods: It provides getter methods (get_guid, get_title, get_description, get_link, get_pubdate) to access these attributes.

C) Triggers

Triggers are used to filter news stories based on specific criteria. The Trigger class and its subclasses define various types of triggers. Here's a detailed explanation:

PROBLEM#2: PhraseTrigger

The PhraseTrigger class is designed to detect if a specific phrase is present in a given text. It operates by normalizing the text (converting it to lowercase and removing punctuation) and then checking if the phrase appears as a sequence of words in the text.

- Initialization: The __init__ method initializes the trigger with a specific phrase.
- is_phrase_in Method: This method takes a text string, converts it to lowercase, removes punctuation, splits it into words, and checks if the phrase (also split into words) appears consecutively in the text.

PROBLEM#3: TitleTrigger

The TitleTrigger class is a subclass of PhraseTrigger and is used to check if a given phrase is present in the title of a news story.

- Initialization: Inherits initialization from PhraseTrigger.
- evaluate Method: This method takes a NewsStory instance and uses the is_phrase_in method to check if the phrase is in the story's title (story.get_title()).

PROBLEM#4: DescriptionTrigger

The DescriptionTrigger class is another subclass of PhraseTrigger, designed to check if a given phrase is present in the description of a news story.

- Initialization: Inherits initialization from PhraseTrigger.
- evaluate Method: This method takes a NewsStory instance and uses the is_phrase_in method to check if the phrase is in the story's description (story.get_description()).

PROBLEM#5: TimeTrigger

The TimeTrigger class handles triggers based on time. It serves as a base class for triggers that need to compare a news story's publication date to a specific time.

- Initialization: The __init__ method takes a time string in the format %Y-%m-%dT%H:%M:%SZ and converts it to a datetime object.

- evaluate Method: This method is not implemented in TimeTrigger itself but in its subclasses.

PROBLEM#6: BeforeTrigger and AfterTrigger

These classes inherit from TimeTrigger and evaluate whether a news story was published before or after a specified time.

- BeforeTrigger:
    - Initialization: Inherits from TimeTrigger.
    - evaluate Method: Checks if the publication date of the story (story.get_pubdate()) is before the specified time.
- AfterTrigger:
    - Initialization: Inherits from TimeTrigger.
    - evaluate Method: Checks if the publication date of the story (story.get_pubdate()) is after the specified time.

PROBLEM#7: NOTTRIGGER

The "NotTrigger" class in the given code is a type of trigger that negates another trigger's evaluation result. Triggers are used to filter news stories based on specific conditions, and NotTrigger inverts the condition it wraps around.

Explanation:

The **NotTrigger** class, which inherits from the **Trigger** base class, is designed to invert the condition of another trigger. The class's **__init__** method takes a single parameter trigger**,** which is an instance of another trigger, and stores this instance as an instance variable **self.trigger**. The key functionality of the NotTrigger class lies in its `evaluate` method. This method is tasked with determining whether a given news story meets the specified condition of the wrapped trigger, but with an inverted logic. It accepts a **story** parameter, which is an instance of the **NewsStory** class, and calls the **evaluate** method of **self.trigger** with this story as an argument. The result of this evaluation is then negated using the `not` operator. Consequently, if **self.trigger.evaluate(story)** returns `True`, the **NotTrigger's evaluate** method will return `False`, and if **self.trigger.evaluate(story)** returns `False` else the **NotTrigger's evaluate** method will return `True`. This inversion allows for more flexible and complex filtering conditions in the news story filtering system.

PROBLEM#8: ANDTRIGGER

The "AndTrigger" class in the RSS (Really Simple Syndication or Rich Site Summary) feed filter script is a specific type of trigger designed to combine two other triggers and only activate when both of the combined triggers are satisfied.

Explanation:

The class inherits from the **Trigger** base class and implements the **evaluate** method, which is used to determine if a given news story meets the criteria defined by the triggers. When an **AndTrigger** object is created, it takes two triggers as arguments, referred to as **trigger1 and trigger2**. The **evaluate** method checks the conditions of both these triggers: it calls the evaluate

method on trigger1 and trigger2 with the same news story as input. If both triggers return `True`, indicating that both conditions are met, the AndTrigger also returns `True`. Otherwise, it returns `False`. This logical combination allows for more complex filtering of news stories, ensuring that a story must satisfy multiple criteria to be considered relevant. For instance, an AndTrigger could be used to filter stories that contain a specific keyword in the title and are published after a certain date, thus enabling precise and targeted news retrieval based on multiple conditions.

PROBLEM#9: ORTRIGGER:

The basic purpose of the "OrTrigger" function is to evaluate whether a news story meets at least one of two specified conditions, ensuring that the story is included if it satisfies either condition. This allows for more flexible and inclusive filtering of news content.

Explanation:

This class inherits from the base **Trigger** class and implements the **evaluate** method, which is essential for determining whether a news story meets the specified conditions. When an **OrTrigger** object is instantiated, it takes two triggers as parameters, designated as **trigger1 and trigger2**. The evaluate method of the OrTrigger calls the evaluate method on both trigger1 and trigger2 with the same news story as input. If either trigger1 or trigger2 (or both) return 'True', indicating that the news story satisfies at least one of the conditions, the OrTrigger also returns 'True'. If both triggers return 'False', then the OrTrigger returns 'False'.

In short, The OrTrigger allows for flexible filtering by selecting news stories that meet at least one of two conditions, such as containing a specific keyword in the title or being published before a certain date. This ensures any story meeting either criterion is included, providing a broader and more inclusive content aggregation. This flexibility is useful for monitoring diverse news categories or conditions.

D) Filtering:

PROBLEM#10: IMPLEMENT FILTER STORIES:

Goal: Write a function filter_stories(stories, triggerlist) that filters news stories based on a list of triggers.

Code for the respective filtering function:

```
#======================
# Filtering
#======================

def filter_stories(stories, triggerlist):
    filtered_stories = []
    for story in stories:
        for trigger in triggerlist:
            if trigger.evaluate(story):
                filtered_stories.append(story)
                break
    return filtered_stories
```

Explanation of the code:

1. First line

```
def filter_stories(stories, triggerlist):
```

It defines the filter_stories function that takes two parameters: stories (a list of NewsStory instances) and triggerlist (a list of Trigger instances).

2. Second line

```
filtered_stories = []
```

This keyword is used to initializes an empty list filtered_stories to store the stories that match any of the triggers.

3. Third line

```
for story in stories:
```

This line is written in the code to start a loop to iterate over each story in the stories list.

4. Fourth line

```
for trigger in triggerlist:
```

For each story, starts another loop to iterate over each trigger in the triggerlist.

5. Fifth line

```
if trigger.evaluate(story):
```

Checks if the current trigger evaluates to True for the current story by calling trigger.evaluate(story).

6. Sixth line:

```
filtered_stories.append(story)
break
```

If the trigger evaluates to True, appends the story to the filtered_stories list.
Uses break to exit the inner loop, ensuring the story is not added multiple times if it matches more than one trigger.

7. Seventh line:

```
return filtered_stories
```

Returns the list of filtered_stories that matched any of the triggers.

E) User Specified Triggers:

PROBLEM#11: Read Trigger Configuration:

```
def read_trigger_config(filename):
    trigger_file = open(filename, 'r')
    lines = []
    for line in trigger_file:
        line = line.rstrip()
        if not (len(line) == 0 or line.startswith('//')):
            lines.append(line)
    trigger_file.close()
```

1. Opening the Configuration File:

```
trigger_file = open(filename, 'r')
```

Opens the file named filename for reading.

2. Initializing Lines List:

```
lines = []
```

Initializes an empty list lines to store the lines from the file that are not blank or comments.

3. Reading and Filtering Lines:

```
for line in trigger_file:
    line = line.rstrip()
    if not (len(line) == 0 or line.startswith('//')):
        lines.append(line)
```

Iterates over each line in the file.
Strips trailing whitespace from the line.
Adds the line to lines if it's not empty and doesn't start with // (indicating a comment).

4. Closing the File:

```
trigger_file.close()
```

Closes the file after reading all relevant lines.
triggers = {} trigger_list = []

5. Initialize Triggers Dictionary and Trigger List:

```
triggers = {}
trigger_list = []
```

6. Processing Each Line:

```
for line in lines:
    parts = line.split(',')
    if parts[0] == 'ADD':
        for name in parts[1:]:
            if name in triggers:
                trigger_list.append(triggers[name])
```

Splits each line by commas into parts.
If the first part is 'ADD', it adds the triggers with names specified in the subsequent parts to trigger_list.

7. Creating Triggers:

```
else:
    trigger_name = parts[0]
    trigger_type = parts[1]
    if trigger_type == 'TITLE':
        triggers[trigger_name] = TitleTrigger(parts[2])
    elif trigger_type == 'DESCRIPTION':
        triggers[trigger_name] = DescriptionTrigger(parts[2])
    elif trigger_type == 'AFTER':
        triggers[trigger_name] = AfterTrigger(parts[2])
    elif trigger_type == 'BEFORE':
        triggers[trigger_name] = BeforeTrigger(parts[2])
    elif trigger_type == 'NOT':
        if parts[2] in triggers:
            triggers[trigger_name] = NotTrigger(triggers[parts[2]])
    elif trigger_type == 'AND':
        if parts[2] in trigger                    gers:
            triggers[trigger_name] = AndTrigger(triggers[parts[2]], triggers[parts[3]])
    elif trigger_type == 'OR':
        if parts[2] in triggers and parts[3] in triggers:
            triggers[trigger_name] = OrTrigger(triggers[parts[2]], triggers[parts[3]])
```

(variable) trigger_name: Any

For lines not starting with 'ADD', extracts the trigger_name and trigger_type.
Depending on the trigger_type, creates the appropriate trigger object using the subsequent parts and stores it in the triggers dictionary.

8. Return Trigger List:

```
return trigger_list
```

Returns the final list of triggers to be used.

F) Main Thread

```
#======================
# Main Thread
#======================

SLEEPTIME = 120  # seconds
```

1.Define Sleep Time:

SLEEPTIME = 120 # seconds
Sets the sleep time to 120 seconds between each polling of news stories.

```
def main_thread(master, keywords):
    try:
        triggerlist = []
        if keywords:
            for keyword in keywords:
                triggerlist.append(OrTrigger(TitleTrigger(keyword), DescriptionTrigger(keyword)))
```

2. Define main_thread Function:

```
def main_thread(master, keywords):
```

Defines the main_thread function that takes master (the main window for the GUI) and keywords (a list of keywords) as parameters.

3. Initialize Trigger List:

```
triggerlist = []
if keywords:
    for keyword in keywords:
        triggerlist.append(OrTrigger(TitleTrigger(keyword), DescriptionTrigger(keyword)))
```

Initializes an empty triggerlist.
If keywords are provided, creates an OrTrigger for each keyword using both TitleTrigger and DescriptionTrigger and adds it to triggerlist.

4. Set Up GUI Frame and Scrollbar:

```
frame = Frame(master)
frame.pack(side=BOTTOM)
scrollbar = Scrollbar(master)
scrollbar.pack(side=RIGHT, fill=Y)
```

Creates a Frame in the master window and packs it at the bottom.
Creates a Scrollbar and packs it on the right side, filling the vertical space.

5. Set Up GUI Elements:

```
get_cont
ttl.pack(side=TOP)
cont = Text(master, font=("Helvetica", 14), yscrollcommand=sc
cont.pack(side=BOTTOM)
cont.tag_config("title", justify='center')
button = Button(frame, text="Exit", command=master.destroy)
button.pack(side=BOTTOM)
guidShown = []
```

Creates and sets up the title label with text "Google & Yahoo Top News".
Creates a Text widget for displaying news stories with the scrollbar linked.
Configures the Text widget to center the title.
Creates an Exit button to close the application.
Initializes an empty list guidShown to keep track of displayed stories.

6. Define get_cont Function:

```
def get_cont(newstory):
    if newstory.get_guid() not in guidShown:
        cont.insert(END, newstory.get_title() + "\n", "title")
        cont.insert(END, "\n----------\n", "title")
        cont.insert(END, newstory.get_description())
        cont.insert(END, "\n**********\n", "title")
        guidShown.append(newstory.get_guid())
```

Defines get_cont function to display a new story in the Text widget if it hasn't been shown yet.

Inserts the story title, a separator, the story description, and another separator into the Text widget.

Adds the story's GUID to guidShown to avoid duplicates.

7. Polling Loop:

```
while True:
    print("Polling...")
    stories = process("http://news.google.com/news?output=rss")
    stories.extend(process("http://news.yahoo.com/rss/topstories"))

    stories = filter_stories(stories, triggerlist)

    list(map(get_cont, stories))
    scrollbar.config(command=cont.yview)

    print(f"No keywords provided. Continuing to poll...")
    time.sleep(SLEEPTIME)
```

Enters an infinite loop to poll news stories continuously.

Prints "Polling..." to indicate polling has started.

Fetches stories from Google and Yahoo news RSS feeds using the process function.

Filters the stories using filter_stories with the triggerlist.

Uses map to apply get_cont to each story, displaying them in the Text widget.

Configures the scrollbar to work with the Text widget.

Sleeps for SLEEPTIME seconds before polling again.

8. Exception Handling:

```
except Exception as e:
    print(f"Error occurred: {e}")
```

Catches and prints any exceptions that occur during the execution of the main_thread.