School Of Mechanical & Manufacturing Engineering, NUST

Department of Mechanical Engineering

CS-116 - Fundamentals of Programming II

FINAL PROJECT

Lab Instructor: **Dr. Saqib Nazir**

Submitted by:

Names	CMS IDs
Ayesha Khan	478212
Faizan Ahmed	476602
Juveriah Waqqas	460510
Muhammad Abdullah Tahseen	462573

DATE:

22-04-24

INTRODUCTION

The purpose of this project is to develop a program that automates the process of monitoring and filtering news feeds based on user-defined criteria. Using object-oriented programming techniques, particularly classes and inheritance, the program will parse RSS feeds from various news sources, evaluate them against user-defined triggers, and alert the user when relevant stories are found.

BREAKDOWN OF CODE

DATA STRUCTURE DESIGN

Objective: Create a class to store and manage news story information.
Task: Implement a NewsStory class that includes attributes for the unique identifier (GUID), title, description, link, and publication date. The class should also provide getter methods for these attributes.
Reason: This structure is necessary to handle news stories consistently throughout the program.

PARSING THE FEED

Objective: Extract and organize data from RSS feeds.
Task: Utilize provided modules to fetch and parse XML data from news feeds.
Reason: Converting the raw feed data into structured NewsStory objects enables further processing and filtering.

PHRASE TRIGGERS

Objective: Create triggers that activate based on specific phrases in the news stories.
Tasks: Implement an abstract PhraseTrigger class.
Create subclasses like TitleTrigger and DescriptionTrigger that fire when the title or description of a news story contains a specified phrase.
Reason: These triggers allow users to receive alerts for news stories that mention particular topics of interest.

TIME TRIGGERS

Objective: Develop triggers based on the publication time of news stories.
Tasks: Implement a TimeTrigger abstract class.
Create BeforeTrigger and AfterTrigger subclasses to fire based on whether a story was published before or after a specific time.
Reason: Time-based triggers help filter news stories relevant to specific timeframes, such as recent events.

COMPOSITE TRIGGERS

Objective: Combine multiple triggers to form more complex alert conditions.
Tasks: Implement NotTrigger, AndTrigger, and OrTrigger classes.

Reason: Composite triggers enhance the flexibility and power of the alerting system, allowing for more nuanced filtering criteria.

FILTERING STORIES

Objective: Apply a list of triggers to filter the news stories.
Task: Write a function filter_stories that returns a list of news stories for which any trigger fires.
Reason: This function ensures that only relevant news stories, as defined by the user's triggers, are presented.

USER-SPECIFIED TRIGGERS

Objective: Allow users to configure their own triggers through an external file.
Task: Implement read_trigger_config to parse a configuration file (triggers.txt) and set up triggers accordingly.
Reason: This feature makes the program user-friendly by enabling dynamic configuration of triggers without modifying the source code.

PROBLEM-WISE CODE EXPLAINATION

Problem 1:

NewsStory Class

This class is designed to represent a news article. Each news article has several attributes: a unique identifier, a title, a description, a link to the full story, and the publication date.

Code:

```
# Problem 1

# TODO: NewsStory
54 usages
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

    7 usages (6 dynamic)
    def get_guid(self):
        return self.guid

    7 usages (6 dynamic)
    def get_title(self):
        return self.title

    7 usages (6 dynamic)
    def get_description(self):
        return self.description
```

```
    1 usage
    def get_link(self):
        return self.link

    7 usages (6 dynamic)
    def get_pubdate(self):
        return self.pubdate
```

Explanation:

- **__init__ Method**: This is the constructor method. It is called when an instance of **NewsStory** is created. It initializes the object's attributes with the provided values:

- **guid**: A unique identifier for the news story.
- **title**: The title of the news story.
- **description**: A brief summary or description of the news story.
- **link**: A URL link to the full news story.
- **pubdate**: The publication date of the news story.

These attributes are stored as instance variables so they can be accessed later.

Getter Methods:

These methods allow access to the attributes of a **NewsStory** object.

- **get_guid**: Returns the GUID of the news story.
- **get_title**: Returns the title of the news story.
- **get_description**: Returns the description of the news story.
- **get_link**: Returns the link to the full news story.
- **get_pubdate**: Returns the publication date of the news story.

These methods provide a way to access the private data stored in the instance variables of a **NewsStory** object.

Triggers

The **Trigger** class is an abstract base class for different types of triggers. A trigger is a condition that, when met, will cause an alert to be generated for a given news item.

Explanation:

- **Trigger Class**: This is a base class that defines the interface for all triggers. It is meant to be subclassed by other classes that implement specific types of triggers.
- **evaluate Method**: This method takes a **story** (an instance of **NewsStory**) as a parameter and returns **True** if the trigger condition is met, indicating that an alert should be generated for the given news item. Otherwise, it returns **False**.
    - The method currently raises a **NotImplementedError**, which means that any subclass of **Trigger** must provide its own implementation of the **evaluate** method.

This is a common technique in object-oriented programming to define a required method that must be implemented by subclasses.

The **Trigger** class serves as a template for creating different types of triggers. Subclasses will override the **evaluate** method to provide specific conditions for generating alerts.

Problem 2:

Phrase Triggers

The **PhraseTrigger** class is a subclass of **Trigger**. It is designed to check if a given phrase appears in a text. The text can be the title, description, or any other part of a news story.

Code:

```
# PHRASE TRIGGERS
# Problem 2
# TODO: PhraseTrigger
2 usages
class PhraseTrigger(Trigger):
    def __init__(self, phrase):
        self.phrase = phrase.lower()

    2 usages
    def is_phrase_in(self, text):
        text = text.lower()
        for punc in string.punctuation:
            text = text.replace(punc, ' ')
        words = text.split()
        normalized_text = ' '.join(words)
        return f' {self.phrase} ' in f' {normalized_text} '

    def evaluate(self, story):
        raise NotImplementedError  # This class is abstract, so it shouldn't implement evaluate directly
```

is_phrase_in Method:

This method checks if the phrase is present in the given text.

Explanation:

- **Lowercase Conversion**: The input **text** is converted to lowercase to ensure case-insensitive matching.
- **Punctuation Removal**: All punctuation marks are replaced with spaces to normalize the text.

- **Splitting and Normalizing Text**: The text is split into words, and then rejoined into a single string with spaces between words. This ensures that multiple spaces or punctuation don't affect the phrase matching.
- **Phrase Checking**: It checks if the phrase (surrounded by spaces) is present in the normalized text (also surrounded by spaces). This ensures that partial matches within words are not counted (e.g., "cat" won't match "caterpillar").

evaluate Method:

The **evaluate** method is meant to be implemented by subclasses that inherit from **PhraseTrigger**.

python

Copy code

def evaluate(self, story): raise NotImplementedError # This class is abstract, so it shouldn't implement evaluate directly

Explanation:

- **evaluate Method**: This method is defined but not implemented. It raises a **NotImplementedError**, indicating that **PhraseTrigger** is an abstract class and should not be instantiated directly. Subclasses of **PhraseTrigger** will implement this method to specify how the phrase trigger should be used (e.g., checking the title or description of a **NewsStory**).

Problem 3:

Title Trigger

The **TitleTrigger** class is a subclass of **PhraseTrigger**. It is designed to check if a given phrase appears in the title of a news story.

Code:

```
# Problem 3
# TODO: TitleTrigger
4 usages
class TitleTrigger(PhraseTrigger):
    def evaluate(self, story):
        return self.is_phrase_in(story.get_title())
```

Explanation:

1. **Class Definition**:

class TitleTrigger(PhraseTrigger):

- The **TitleTrigger** class inherits from the **PhraseTrigger** class. This means it has access to all the methods and attributes of **PhraseTrigger**.

2. **evaluate Method**:
   - This method overrides the **evaluate** method from the **Trigger** base class.
   - **Parameters**:
     - **story**: An instance of the **NewsStory** class.
   - **Functionality**:
     - It calls the **get_title** method on the **story** object to retrieve the title of the news story.
     - It then calls the **is_phrase_in** method (inherited from **PhraseTrigger**) with the title as the argument to check if the phrase is present in the title.
     - The method returns **True** if the phrase is found in the title, and **False** otherwise.

Problem 4:

DescriptionTrigger Class Explanation

```
# Problem 4
# TODO: DescriptionTrigger
4 usages
class DescriptionTrigger(PhraseTrigger):
    def evaluate(self, story):
        return self.is_phrase_in(story.get_description())
```

1. **Class Definition**:
   - **DescriptionTrigger** inherits from **PhraseTrigger**.
2. **evaluate Method**:
   - Overrides the **evaluate** method from **Trigger**.
   - Uses **is_phrase_in** method (inherited from **PhraseTrigger**) to check if the phrase is in the news story's description.

Key Points

- **Inheritance**: **DescriptionTrigger** reuses **is_phrase_in** method from **PhraseTrigger**.
- **Purpose**: Triggers alerts based on phrases found in the description of a news story.

Usage

- Create an instance of **DescriptionTrigger** with a specific phrase.
- Call **evaluate** on a **NewsStory** instance to check if the phrase is in the description.

Problem 5:

TimeTrigger Class Explanation

```
# Problem 5
# TODO: TimeTrigger
# Constructor:
#        Input: Time has to be in EST and in the format of "%d %b %Y %H:%M:%S".
#        Convert time from string to a datetime before saving it as an attribute.
2 usages
class TimeTrigger(Trigger):
    def __init__(self, time_string):
        # Convert the time string to a datetime object in UTC
        self.time = datetime.strptime(time_string, _format: "%d %b %Y %H:%M:%S")
        # Set the timezone to UTC
        self.time = self.time.replace(tzinfo=pytz.utc)
        # Convert to EST timezone
        self.time = self.time.astimezone(pytz.timezone("US/Eastern"))
```

1. **Class Definition:**
    - **TimeTrigger inherits from Trigger.**
2. **__init__ Method:**
    - **Input: time_string is expected in the format "%d %b %Y %H:%M:%S" and in EST.**
    - **Step-by-Step Conversion:**
        1. **String to datetime:**
            - **datetime.strptime(time_string, "%d %b %Y %H:%M:%S") converts the input string to a datetime object.**
        2. **Set Timezone to UTC:**
            - **self.time.replace(tzinfo=pytz.utc) sets the timezone of the datetime object to UTC.**
        3. **Convert to EST:**
            - **self.time.astimezone(pytz.timezone("US/Eastern")) converts the UTC datetime object to Eastern Standard Time (EST).**

Key Points

- **Inheritance: TimeTrigger extends the base Trigger class.**
- **Purpose: To handle time-based triggers using a specific time format and timezone conversion.**

Usage

- **Create an instance of TimeTrigger with a specific time string.**
- **The instance will hold the converted time in EST.**

Problem 6:

BeforeTrigger and AfterTrigger

```
# TODO: BeforeTrigger and AfterTrigger
3 usages
class BeforeTrigger(TimeTrigger):
    def evaluate(self, story):
        return story.get_pubdate() < self.time

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

BeforeTrigger:

- Inherits from **TimeTrigger**.
- **evaluate** method returns **True** if the story's publication date is before the trigger's time.

AfterTrigger:

- Inherits from **TimeTrigger**.
- **evaluate** method returns **True** if the story's publication date is after the trigger's time.

Problem 7:

NotTrigger

```python
# Problem 7
# TODO: NotTrigger
3 usages
class NotTrigger(Trigger):
    def __init__(self, trigger):
        self.trigger = trigger

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

NotTrigger:

- Inverts the result of another trigger.
- **__init__** method takes another trigger as an argument.
- **evaluate** method returns the opposite of the given trigger's evaluation.

Problem 8:

AndTrigger

```python
# Problem 8
# TODO: AndTrigger
5 usages
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)
```

AndTrigger:

- Combines two triggers.
- **__init__** method takes two triggers as arguments.
- **evaluate** method returns **True** only if both triggers evaluate to **True**.

Problem 9:

OrTrigger

```python
# Problem 9
# TODO: OrTrigger
5 usages
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)
```

OrTrigger:

- Combines two triggers.
- **__init__** method takes two triggers as arguments.
- **evaluate** method returns **True** if at least one of the triggers evaluates to **True**.

Problem 10:

filter_stories

```python
# Problem 10
4 usages
def filter_stories(stories, triggerlist):
    """
    Takes in a list of NewsStory instances.

    Returns: a list of only the stories for which a trigger in triggerlist fires.
    """
    # TODO: Problem 10
    filtered_stories = []

    for story in stories:
        for trigger in triggerlist:
            if trigger.evaluate(story):
                filtered_stories.append(story)
                break  # No need to check other triggers if one has fired

    return filtered_stories
```

filter_stories Function:

- Takes a list of **NewsStory** instances and a list of triggers.
- Returns a list of stories for which any trigger in **triggerlist** fires.
- **Logic**:
    - Iterates through each story.

- For each story, iterates through each trigger in the **triggerlist**.
- If a trigger fires (evaluates to **True**), the story is added to **filtered_stories**.
- Stops checking further triggers for the current story once a trigger fires.

Problem 11:

```
# Problem 11
1 usage
def read_trigger_config(triggers):
    """
    filename: the name of a trigger configuration file

    Returns: a list of trigger objects specified by the trigger configuration file.
    """
    try:
        trigger_file = open(triggers, 'r')
    except FileNotFoundError:
        print(f"Error: The file {triggers} was not found.")
        return []

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

    def parse_trigger_line(line):
```

```
def parse_trigger_line(line):
    parts = line.split(',')
    trigger_name = parts[0].strip()
    trigger_type = parts[1].strip()
    trigger_args = [arg.strip() for arg in parts[2:]]
    return trigger_name, trigger_type, trigger_args

triggers = {}
trigger_list = []
for line in lines:
    if line.startswith('ADD'):
        trigger_names = line.split(',')[1:]
        trigger_list.extend(triggers[name] for name in trigger_names if name in triggers)
    else:
        trigger_name, trigger_type, trigger_args = parse_trigger_line(line)
        if trigger_type == 'TITLE':
            trigger = TitleTrigger(*trigger_args)
        elif trigger_type == 'DESCRIPTION':
            trigger = DescriptionTrigger(*trigger_args)
        elif trigger_type == 'AFTER':
            trigger = AfterTrigger(*trigger_args)
        elif trigger_type == 'BEFORE':
            trigger = BeforeTrigger(*trigger_args)
        elif trigger_type == 'NOT':
            trigger = NotTrigger(triggers[trigger_args[0]])
        elif trigger_type == 'AND':
            trigger = AndTrigger(triggers[trigger_args[0]], triggers[trigger_args[1]])
        elif trigger_type == 'OR':
            trigger = OrTrigger(triggers[trigger_args[0]], triggers[trigger_args[1]])
        triggers[trigger_name] = trigger

    return trigger_list
```

Function: read_trigger_config

This function reads a trigger configuration file and returns a list of trigger objects specified by the file.

Function Definition and Docstring

- **Parameters**:
    - **triggers**: The name of the trigger configuration file.
- **Returns**:
    - A list of trigger objects specified in the configuration file.

File Handling

- Attempts to open the specified configuration file.
- If the file is not found, prints an error message and returns an empty list.

Reading and Cleaning Lines

- Reads each line from the file, removes any trailing whitespace.
- Ignores empty lines and lines starting with // (comments).
- Adds the cleaned lines to the **lines** list.

Parsing a Trigger Line

- **Helper Function**:
    - Splits a line into its components: trigger name, type, and arguments.
    - Strips whitespace from each component.
    - Returns the parsed components.

Creating Trigger Objects

- Initializes an empty dictionary **triggers** and an empty list **trigger_list**.
- Iterates over each line:
    - If a line starts with **ADD**:
        - Extracts trigger names and adds the corresponding trigger objects from the **triggers** dictionary to the **trigger_list**.
    - Otherwise:
        - Parses the line into **trigger_name**, **trigger_type**, and **trigger_args**.
        - Creates a trigger object based on the **trigger_type** and **trigger_args**.
        - Stores the trigger object in the **triggers** dictionary with **trigger_name** as the key.

Returning the Trigger List

- Returns the list of trigger objects specified in the configuration file.

School Of Mechanical & Manufacturing Engineering, NUST

Department of Mechanical Engineering

ERRORS ENCOUNTERED IN CODE IMPLEMENTATION AND THEIR SOLUTIONS

Errors in function 'feedparser'

Following are the errors encountered by us while running these codes along with their respective fixes:

1. **Python version incompatibility**

   Old python error:

   The error was encountered due to the use of the deprecated ***'base64.decodestring'*** function in your feedparser.py file. This function was removed in Python 3.9. The python that was being used here was Python 3.115.

   Solution

   In this version, the correct method to use was base64.decodebytes.Hence the base64.decodestring function was replaced with the new and compatible base64.decodebytes function.

2. **Naming Conflict Error**

   The local file that housed the feedparser function which was named feedparser.py was conflicting with the importation of the actual feedparser library.

   Solution

   Renaming the Local feedparser.py file:

   The local file named feedparser.py that was overshadowing the actual feedparser library was renamed to my_feedparser.py.

3. **Instation of feedparser library Error**

   Another one of the errors encountered was the 'ModuleNotFoundError' indicating that the feedparser module was not found.

   Solution:

   This was solved by using 'pip' to install the library, following the given steps:

   - *Open a Command Prompt or Terminal:*
     Press the Windows key, type 'cmd', and press Enter.
   - *Type the Install Command:*
     In the command prompt or terminal window, type the following command and then press Enter:
     'pip install feedparser'
   - *Wait for the Installation to Complete:*
     You will see some messages indicating that pip is downloading and installing the feedparser library. Once it's done, you'll get a message that the installation was successful.

Errors in installation of ‘pytz’ library

Another error encountered was the 'ModuleNotFoundError', indicating that the ‘pytz’ module was not found.

Solution:

This was resolved by installing the library using pip, using the same steps as above but typing the following command in the command prompt or terminal window

‘pip install pytz’

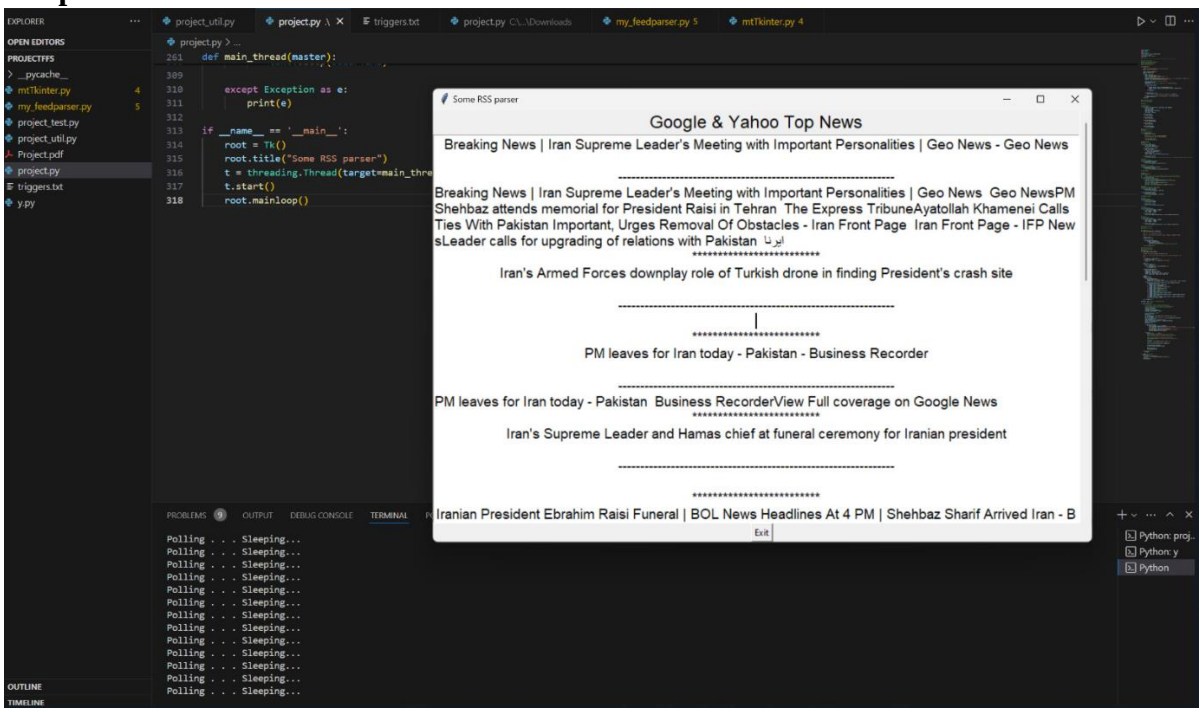
OUTPUT SCREENSHOTS

1. Triggers

triggers.txt

```
// trigger file - if you've done problem 9 but no stories are popping up, you
// should edit this file to contain triggers that will fire on current news
// stories!

// title trigger named t1
t1,TITLE,Iran

// description trigger named t2
t2,DESCRIPTION,death

// description trigger named t3
t3,DESCRIPTION,Crash

// after trigger named t4
t4,AFTER,18 May 2024 17:00:10

// composite trigger named t4
t5,AND,t2,t3

// composite trigger named t4
t6,AND,t1,t4

// the trigger list contains t1 and t4
ADD,t5,t6
```

Output



2. Trigger

triggers.txt

```
// trigger file - if you've done problem 9 but no stories are popping up, you
// should edit this file to contain triggers that will fire on current news
// stories!

// title trigger named t1
t1,TITLE,Krygyzstan

// description trigger named t2
t2,DESCRIPTION,Pakistani

// description trigger named t3
t3,DESCRIPTION,Students

// after trigger named t4
t4,AFTER,18 May 2024 17:00:10

// composite trigger named t4
t5,AND,t2,t3

// composite trigger named t4
t6,AND,t1,t4

// the trigger list contains t1 and t4
ADD,t5,t6
```

Output