School Of Mechanical & Manufacturing Engineering, NUST

Department of Mechanical Engineering

Fundamentals of Programming

Project Report

Lab Instructor: Muhammad Affan

Section: ME-15-A

Name	CMS ID
Aatika Kamran	464185
Ahmed Adil Hussain	477537
Abdullah bin Khorram	466612
Waleed Masud	470557

DATE:

20/05/2024

Problem 1

Code

```
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
```

Explanation

In this code a class “newstory” has been initialized with the following attributes :

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

The **self**-parameter is a reference to the current instance of the class. It is used to access variables that belong to the class.

5 getter functions are made to to access the attributes of the **NewsStory .** Each function returns the corresponding attribute.

Problem 2

Code

```
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
```

Explanation

The **PhraseTrigger** class is designed to determine if a specific phrase appears within a given text. It is the subclass of triggers. This subclass initiates self and phrase attributes.

- **phrase**: The phrase that the trigger will look for in the text.
- **self.phrase = phrase.lower()**: The phrase is converted to lowercase and stored as an instance variable. This ensures that the comparison is case-insensitive.

The phrase trigger has a function **is_phrase_in** that checks whether the phrase stored in the trigger appears in the given **text.**

The input text is converted to lowercase to ensure the comparison is case-insensitive. Then all punctuation characters in the text are replaced with spaces. This helps in splitting the text into words without punctuation interfering. The modified text is split into a list of words using spaces as delimiters. The phrase stored in the trigger is split into a list of words. The loop checks each possible position in **text_words** where **phrase_words** could start. If a segment of **text_words** matches **phrase_words**, the method returns True. If the loop completes without finding a match, the method returns **False**.

Problem 3

Code

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

Explanation

The **TitleTrigger** class is a subclass of the **PhraseTrigger** class, and it is specifically designed to evaluate whether a given phrase appears in the title of a news story. The **evaluate** method is used to determine if the phrase stored in the **TitleTrigger** instance appears in the title of a given news story. The function **evaluate** has parameters **self** and **story.** The story is expected to be an instance of a class (likely NewsStory) that has a method **get_title().** This return line calls the **get_title** method on the story object to retrieve the title of the news story. The title is then passed to the **is_phrase_in** method (inherited from PhraseTrigger). The

is_phrase_in method (defined in **PhraseTrigger**) checks whether the phrase stored in the **TitleTrigger** instance appears in the title of the story. It returns **True** if the phrase is found and **False** otherwise.

Problem 4

Code

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

Explanation

The `DescriptionTrigger` class inherits from the `PhraseTrigger` class and is designed to check if a specific phrase appears in the description of a news story.

evaluate(self, story): This method takes a `NewsStory` object as an argument and returns `True` if the phrase (stored in the `PhraseTrigger` instance) is found in the story's description. It uses the `is_phrase_in` method from the `PhraseTrigger` class to perform this check. If the phrase is not found, it returns `False`.

Problem 5

Code

```
# 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.
class TimeTrigger(Trigger):
    def __init__(self, time):
        self.time = datetime.strptime(time, "%d %b %Y %H:%M:%S")
```

Explanation

The TimeTrigger class, inheriting from Trigger, is crafted for managing triggers based on specific times.

This constructor initializes a trigger based on a specific time, converting a string representation of time into a datetime object. time is a string representing the time in the format "%d %b %Y %H:%M:%S". It converts the time string into a datetime object using datetime.strptime. For example, if time is "21 May 2023 14:30:00", it becomes a datetime object representing May 21, 2023, at 14:30:00.

Problem 6

Code

```
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
```

Explanation

These two classes, BeforeTrigger and AfterTrigger, both inherit from the TimeTrigger class. They are designed to evaluate whether a given news story occurred before or after a specified time, respectively.

BeforeTrigger:

This class determines if a news story occurred before a specified time.
Using function evaluate(story,we take a NewsStory object as input and returns True if the story's publication date is before the specified time (self.time), otherwise returns False.

If self.time is May 21, 2023, at 14:30:00, and story.get_pubdate() returns May 20, 2023, at 12:00:00, the method returns True.

AfterTrigger:

Checks if a news story occurred after a specified time.

Using the function `evaluate(story,` it takes a `NewsStory` object as input and returns `True` if the story's publication date is after the specified time (`self.time`), otherwise returns `False`. If `self.time` is May 21, 2023, at 14:30:00, and `story.get_pubdate()` returns May 22, 2023, at 10:00:00, the method returns `True`.

Problem 7

Code

```
# TODO: NotTrigger

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

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

Explanation

The `NotTrigger` class is a composite trigger that inverts the result of another trigger. The `__init__` method initializes the `NotTrigger` object by accepting another trigger object (`trigger`) and storing it in `self.trigger`. The `evaluate` method returns the negation (`not`) of the evaluation result of the stored trigger. This means if the stored trigger returns `True` for a given story, the `NotTrigger` will return `False`, and vice versa.

Problem 8

Code

```
# TODO: AndTrigger
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)
```

Explanation

The `AndTrigger` class is a composite trigger that combines two other triggers and fires only if both of them fire. The `__init__` method initializes the `AndTrigger` object by accepting two trigger objects (`trigger1` and `trigger2`) and storing them in `self.trigger1` and `self.trigger2`.The `evaluate` method returns `True` only if both `self.trigger1` and `self.trigger2` evaluate to `True` for a given story. If either trigger evaluates to `False`, the `AndTrigger` will return `False`.

Problem 9

Code

```
# TODO: OrTrigger

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)
```

Explanation

The `OrTrigger` class is a composite trigger that combines two other triggers and fires if at least one of them fires. The `__init__` method initializes the `OrTrigger` object by accepting two trigger objects (`trigger1` and `trigger2`) and storing them in `self.trigger1` and `self.trigger2`.The `evaluate` method returns `True` if either `self.trigger1` or `self.trigger2` evaluates to `True` for a given story. If both triggers evaluate to `False`, the `OrTrigger` will return `False`.

Problem 10

Code

```
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
    # This is a placeholder
    # (we're just returning all the stories, with no filtering)
    filtered_stories = []

    for story in stories:
        for trigger in triggerlist:
            if trigger.evaluate(story):
                filtered_stories.append(story)
                break

    return filtered_stories
```

Explanation

The filter_stories function filters a list of news stories based on a list of triggers. It initializes an empty list called filtered_stories to store stories that match any trigger. The function then loops through each story in the stories list and, for each story, it loops through each trigger in the triggerlist. If a trigger's evaluate method returns True for a story, that story is added to filtered_stories, and the function breaks out of the inner loop to stop checking other triggers for that story. Finally, the function returns the filtered_stories list containing stories that matched at least one trigger.

Problem 11

Code

```
def read_trigger_config(filename):
    """
    filename: the name of a trigger configuration file
    Returns: a list of trigger objects specified by the trigger configuration
file.
    """
      # We give you the code to read in the file and eliminate blank lines and
    # comments. You don't need to know how it works for now!
    trigger_file = open(filename, 'r')
    lines = [line.strip() for line in trigger_file.readlines() if line.strip()
and not line.strip().startswith('//')]
    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
```

Explanation

The `read_trigger_config` function reads a trigger configuration file and returns a list of trigger objects based on the file's content. It first reads the file, stripping out blank lines and comments. It then processes each line: if the line starts with 'ADD', it adds the specified

triggers to the `trigger_list`; otherwise, it creates a trigger object based on the type specified in the line and stores it in the `triggers` dictionary. The function supports various trigger types, including `TitleTrigger`, `DescriptionTrigger`, `AfterTrigger`, `BeforeTrigger`, `NotTrigger`, `AndTrigger`, and `OrTrigger`. Finally, the function returns the list of trigger objects specified by the 'ADD' commands in the configuration file.

Output

After changing t1 to iran , t2 to president and t3 to death in the txt file following news are shown :