Problem Set 5: RSS Feed Filter

Handed out: Tuesday, March 19, 2013.
Due: 11:59pm, Tuesday, April 2, 2013.

Objectives

The goal of this problem set is to help you become familiar and comfortable with the following topics:

- Many facets of object oriented programming, specifically:
    - Implementing new classes and their attributes.
    - Understanding class methods.
    - Understanding inheritance.
    - Telling the difference between a class and an instance of that class – recall that a *class* is a blueprint of an object, whilst an *instance* is a single, unique unit of a class.
- Utilizing libraries as black boxes.

Introduction

In problem set 5, you will 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 (for example, the user may be interested in a notification whenever a story related to the Red Sox is posted).

This problem set has a lot of words, but don't get intimidated! The staff solution has about 80 lines of code; we recommend that the solutions you write for each problem should stay under about 15-20 lines of code (the solutions for some problems will be *much* shorter than that). If you find yourself writing way more code than that, you should come visit us at office hours to see how you can simplify things.

We recommend starting early because there is a lot of reading here, but you ought to be able to do this problem set sequentially in the order that we've laid out. There are a lot of references on Python classes available (look for classes in the readings listed in the Reference Links section of the webpage); here is the official Python tutorial on classes, sections 9.1-9.7 (excepting 9.5.1) will be useful for this problem set.

Getting Started

Download and save

pset5.zip: A zip file of all the files you need, including:

- `ps5.py`, a skeleton of the solution
- `ps5_test.py`, a test suite that will help you check your answers
- `triggers.txt`, a sample trigger configuration file. You may modify this file to try other trigger configurations
- `feedparser.py`, a module that will retrieve and parse feeds for you
- `project_util.py`, a module that includes a function to convert simple HTML fragments to plain text

The two modules (`feedparser.py, project_util.py`) are necessary for this lab to work, but you will *not* need to modify them. Feel free to read through them if you'd like to understand what's going on.

RSS Overview

Many websites have content that is updated on an unpredictable schedule. News sites, such as Google News, are a good example of this. One tedious way to keep track of this changing content is to load the website up in your browser, and periodically hit the refresh button.

Fortunately, this process can be streamlined and automated by connecting to the website's RSS feed, using an *RSS feed reader* instead of a web browser (e.g. Sage). An RSS reader will periodically collect and draw your attention to updated content.

RSS stands for "*Really Simple Syndication*." An RSS feed consists of (periodically changing) data stored in an XML-format file residing on a web-server. For this project the details are unimportant. You don't need to know what XML is, nor do you need to know how to access these files over the network.

We will use a special Python module to deal with these low-level details. The higher-level details of the structure of the Google News RSS feed as described in the notes below should be enough for our purposes.

Part I: Data structure design

RSS Feed Structure: Google News

First, let's talk about one specific RSS feed: Google News. The URL for the Google News feed is: `http://news.google.com/?output=rss`

If you try to load this URL in your browser, you'll probably see your browser's interpretation of the XML code generated by the feed. You can view the XML source with your browser's "View Page Source" function, though it probably will not make much sense to you. Abstractly, whenever you connect to the Google News RSS feed, you receive a **list of items**. Each **entry** in this list represents a single news item. In a Google News feed, every entry has the following fields:

- **`guid`** : A globally unique identifier for this news story.
- **`title`** : The news story's headline.
- **`subject`** : A subject tag for this story (e.g. 'Top Stories', or 'Sports').
- **`summary`** : A paragraph or so summarizing the news story.
- **`link`** : A link to a web-site with the entire story.

Generalizing the Problem

This is a little trickier than we'd like it to be, because each of these RSS feeds is structured a little bit differently than the others. So, our goal in Part I is to come up with a unified, standard representation that we'll use to store a news story.

Why do we want this? When all is said and done, we want an application that aggregates several RSS feeds from various sources and can act on all of them in the exact same way: we should be able to read news stories from various RSS feeds all in one place. If you've ever used an RSS feed reader, be assured that it has had to solve the exact problem we're going to tackle in this pset!

Problem 1.

Parsing (see below this box for a definition) all of this information from the feeds that Google/Yahoo/the New York Times/etc. gives us is no small feat. So, let's tackle an easy part of the problem first: Pretend that someone has already done the specific parsing, and has left you with variables that contain the following information for a news story:

- globally unique identifier (GUID) – a string that serves as a unique name for this entry
- title – a string
- subject – a string
- summary – a string
- link to more content – a string

We want to store this information in an *object* that we can then pass around in the rest of our program. Your task, in this problem, is to write a class, `NewsStory`, **starting with a constructor** that takes `(guid, title, subject, summary, link)` as arguments and stores them appropriately. NewsStory also needs to contain the following methods:

- `get_guid(self)`
- `get_title(self)`
- `get_subject(self)`
- `get_summary(self)`
- `get_link(self)`

The solution to this problem should be relatively short and very straightforward (please review what `get` methods should do if you find yourself writing multiple lines of code for each). Once you have implemented `NewsStory` all the `NewsStory` test cases should work.

Parsing the Feed

Parsing is the process of turning a data stream into a structured format that is more convenient to work with. We have provided you with code that will retrieve and parse the Google and Yahoo news feeds.

Part II: Triggers

Given a set of news stories, your program will generate **alerts** for a subset of those stories. Stories with alerts will be displayed to the user, and the other stories will be silently discarded. We will represent alerting rules as **triggers**. A trigger is a rule that is evaluated over a single news story and may fire to generate an alert. For example, a simple trigger could fire for every news story whose title contained the word "Microsoft". Another trigger may be set up to fire for all news stories where the summary contained the word "Boston". Finally, a more specific trigger could be set up to fire only when a news story contained both the words "Microsoft" and "Boston" in the summary.

In order to simplify our code, we will use object polymorphism. We will define a trigger interface and then implement a number of different classes that implement that trigger interface in different ways.

Trigger interface

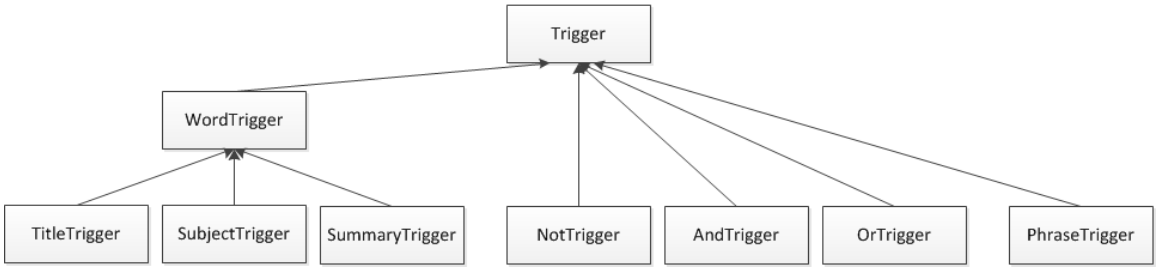
Each trigger class you define should implement the following interface, either directly or transitively. It must implement the **`evaluate`** method that takes a news item (`NewsStory` object) as an input and returns `True` if an alert should be generated for that item. We will not directly use the implementation of the Trigger class, which is why it throws an exception should anyone attempt to use it

The class below implements the Trigger interface (you will not modify this). Any subclass that inherits from it will have an evaluate method. By default, they will use the evaluate method in Trigger, the superclass, unless they define their own evaluate function, which would then be used instead. If some subclass neglects to define its own evaluate() method, calls to it will go to Trigger.evaluate(), which fails (albeit cleanly) with the NotImplementedError exception:

```
class Trigger(object):
    def evaluate(self, story):
        """
        Returns True if an alert should be generated
        """
        raise NotImplementedError
```

We will define a number of classes that inherit from Trigger. In the figure below, Trigger is a superclass, which all other classes inherit from. The arrow from WordTrigger to Trigger means that WordTrigger inherits from Trigger — a WordTrigger ***is a Trigger***. Note that other classes inherit from WordTrigger.

Trigger Class Inheritance



Whole Word Triggers

Having a trigger that always fires isn't interesting; let's write some that are. A user may want to be alerted about news items that contain specific words. For instance, a simple trigger could fire for every news item whose *title* contains the word "Microsoft". In the following problems, you will create a word trigger *abstract class* and implement three classes that implement this word trigger. The trigger should fire when the whole word is present. For example, a trigger for "soft" should fire on:

- Koala bears are soft and cuddly.

- I prefer pillows that are soft.
- Soft drinks are great.
- Soft's the new pink!
- "Soft!" he exclaimed as he threw the football.

But should not fire on

- Microsoft recently released the Windows 8 Consumer Preview.

This is a little tricky, especially the case with the apostrophe. For the purpose of your parsing, pretend that a space or any character in `string.punctuation` is a word separator. If you've never seen `string.punctuation` before, go to your interpreter and type:

```
>>> import string
>>> print string.punctuation
```

Play around with this a bit to get comfortable with what it is. The `split` and `replace` method of strings will almost certainly be helpful as you tackle this part.
You may also find the string methods `lower` and/or `upper` useful for this problem.

Problem 2.

Implement a word trigger abstract class, `WordTrigger`. It should take in a string `word` as an argument to the class's constructor.

`WordTrigger` should be a subclass of `Trigger`. It has one new method, `is_word_in`, which takes in one string argument `text`. It returns `True` if the whole word `word` is present in text, `False` otherwise, as described in the above examples. This method should not be case-sensitive. Implement this method.

Because this is an abstract class, we will not be directly instantiating any `WordTriggers`. `WordTrigger` should inherit its `evaluate` method from `Trigger`. We do this because now we can create subclasses of WordTrigger that use its `is_word_in`function. In this way, it is much like the Trigger interface, except now actual code from this WordTrigger class is used in its subclasses.

You are now ready to implement `WordTrigger`'s three subclasses: `TitleTrigger`, `SubjectTrigger`, and `SummaryTrigger`.

Problem 3.

Implement a word trigger class, `TitleTrigger` that fires when a news item's **title** contains a given word. The word should be an argument to the class's constructor. This trigger should not be case-sensitive (it should treat "Intel" and "intel" as being equal).

For example, an instance of this type of trigger could be used to generate an alert whenever the word "Intel" occurred in the title of a news item. Another instance could generate an alert whenever the word "Microsoft" occurred in the title of an item.

Think carefully about what methods should be defined in `TitleTrigger` and what methods should be inherited from the superclass.

Once you've implemented `TitleTrigger`, the `TitleTrigger` unit tests in our test suite should pass.

Problem 4.

Implement a word trigger class, `SubjectTrigger`, that fires when a news item's **subject** contains a given word. The word should be an argument to the class's constructor. This trigger should not be case-sensitive.

Once you've implemented `SubjectTrigger`, the `SubjectTrigger` unit tests in our test suite should pass.

Problem 5.

Implement a word trigger class, `SummaryTrigger`, that fires when a news item's **summary** contains a given word. The word should be an argument to the class's constructor. This trigger should not be case-sensitive.

Once you've implemented `SummaryTrigger`, the `SummaryTrigger` unit tests in our test suite should pass.

Composite Triggers

So the triggers above are mildly interesting, but we want to do better: we want to 'compose' the earlier triggers, to set up more powerful alert rules. For instance, we may want to raise an alert only when both "google" and "stock" were present in the news item (an idea we can't express right now).

Note that these triggers are *not* word triggers and should not be subclasses of `WordTrigger`.

Problem 6.

Implement a NOT trigger (`NotTrigger`).

This trigger should produce its output by inverting the output of another trigger. The NOT trigger should take this other trigger as an argument to its constructor (why its constructor? Because we can't change what parameters `evaluate` takes in...that'd break our polymorphism). So, given a trigger `T` and a news item `x`, the output of the NOT trigger's `evaluate`method should be equivalent to `not T.evaluate(x)`.

When this is done, the `NotTrigger` unit tests should pass.

Problem 7.

Implement an AND trigger (`AndTrigger`).

This trigger should take two triggers as arguments to its constructor, and should fire on a news story only if *both* of the inputted triggers would fire on that item.

When this is done, the `AndTrigger` unit tests should pass.

Problem 8.

Implement an OR trigger (`OrTrigger`).

This trigger should take two triggers as arguments to its constructor, and should fire if either one (or both) of its inputted triggers would fire on that item.

When this is done, the `OrTrigger` unit tests should pass.

Phrase Triggers

At this point, you have no way of writing a trigger that matches on "New York City" — the only triggers you know how to write would be a trigger that would fire on "New" AND "York" AND "City" – which also fires on the phrase "New students at York University love the city". It's time to fix this. Since here you're asking for an exact match, we will require that the cases match, but we'll be a little more flexible on word matching. So, "New York City" will match:

- New York City sees movie premiere
- In the heart of New York City's famous cafe
- New York Cityrandomtexttoproveapointhere

but will not match:

- I love new york city
- I love New York City!!!!!!!!!!!!!!

Problem 9.

Implement a phrase trigger (`PhraseTrigger`) that fires when a given phrase is in **any** of the story's subject, title, or summary. The phrase should be an argument to the class's constructor. You may find the Python operator `in` helpful, as in:

```
>>> print "New York City" in "In the heart of New York City's famous
cafe"
True
>>> print "New York City" in "I love new york city"
False
```

When this is done, the `PhraseTrigger` unit tests should pass.

Part III: Filtering

At this point, you can run `ps5.py`, and it will fetch and display Google and Yahoo news items for you in little pop-up windows. How many news items? *All of them*.

Right now, the code we've given you in `ps5.py` gets all of the feeds every minute, and displays the result. This is nice, but, remember, the goal here was to filter out only the stories we wanted.

Problem 10.

Write a function, `filter_stories(stories, triggerlist)` that takes in a list of news stories and a list of triggers, and returns a list of only the stories for which a trigger fires.

After completing Problem 10, you can try running `ps5.py`, and various RSS news items should pop up, filtered by some hard-coded triggers defined for you in some code near the bottom. The code runs an infinite loop, checking the RSS feed for new stories every 60 seconds.

Part IV: User-Specified Triggers

Right now, your triggers are specified in your Python code, and to change them, you have to edit your program. This is very user-unfriendly. (Imagine if you had to edit the source code of your web browser every time you wanted to add a bookmark!)

Instead, we want you to read your trigger configuration from a `triggers.txt` file, every time your application starts, and use the triggers specified there.

Consider the following example configuration file:

```
# subject trigger named t1
t1 SUBJECT world

# title trigger named t2
t2 TITLE Intel

# phrase trigger named t3
t3 PHRASE New York City
```

```
# composite trigger named t4
t4 AND t2 t3

# the trigger set contains t1 and t4
ADD t1 t4
```

The example file specifies that four triggers should be created, and that two of those triggers should be added to the trigger list:

- A trigger that fires when a subject contains the word 'world' (`t1`).
- A trigger that fires when the title contains the word 'intel' and the news item contains the phrase 'New York City' somewhere (`t4`).

The two other triggers (`t2` and `t3`) are created but not added to the trigger set directly. They are used as arguments for the composite `AND` trigger's definition (`t4`).
Each line in this file does one of the following:

- is blank
- is a comment (begins with a #)
- defines a named trigger
- adds triggers to the trigger list.

Each type of line is described below.

Blank: blank lines are ignored. A line that consists only of whitespace is a blank line.

Comments: Any line that begins with a # character is ignored.

Trigger definitions: Lines that do not begin with the keyword `ADD` define named triggers. The first element in a trigger definition is the name of the trigger. The name can be any combination of letters without spaces, except for "ADD". The second element of a trigger definition is a keyword (e.g., `TITLE`, `PHRASE`, etc.) that specifies the kind of trigger being defined. The remaining elements of the definition are the trigger arguments. What arguments are required depends on the trigger type:

- **`TITLE`** : a single word.
- **`SUBJECT`** : a single word.
- **`SUMMARY`** : a single word.
- **`NOT`** : the name of the trigger that will be NOT'd.
- **`AND`** : the names of the two other triggers that will be AND'd.
- **`OR`** : the names of the two other triggers that will be OR'd.
- **`PHRASE`** : a phrase.

Trigger addition: A trigger definition should create a trigger and associate it with a name but should not automatically add that trigger to the running trigger list. One or more `ADD` lines in the .txt file will specify which triggers should be in the trigger list. An addition line begins with the `ADD` keyword. Following `ADD` are the names of one or more previously defined triggers. These triggers will be added to the trigger list.

Problem 11.

Updating your `triggers.txt (filename)`. In `readTriggerConfig`, we've written code to open the file and throw away all the lines that don't begin with instructions (e.g. comments, blank spaces) and return the list of triggers specified in the configuration file. Your job is to *play* with the configuration file and customize a RSS reader that tracks news of the US budget sequestration. The triggers you design should meet the following conditions:

1. The title should have both "sequestration" and "Congress" but not "Senate"

2. The newstory should have the phrase "President Obama"

Once that's done, modify the code within the function `main_thread` to use the trigger list specified in your configuration file, instead of the one we hard-coded for you:

```
# TODO: Problem 11
# After updating triggers.txt, uncomment the line below:
# triggerlist = readTriggerConfig("triggers.txt")
```

After completing Problem 11, you can try running ps5.py, and depending on your triggers.txt file, only specific RSS news items should pop up for easy reading. The code runs an infinite loop, checking the RSS feed for new stories every 60 seconds.

This completes the problem set!

Handin Procedure

1. Save
Save your solution files with the names they were provided: All your code should be in a single file called `ps5.py`. You *should also* turn in triggers.txt.
Do not ignore this step or save your file(s) with a different name!

2. Time and collaboration info
At the start of the file, in a comment, write down the number of hours (roughly) you spent on this problem set, and the names of whomever you collaborated with. For example:

```
# Problem Set 5
# Name: Jane Lee
# Collaborators (Discussion): John Doe
# Collaborators (Identical Solution): Jane Smith
# Time: 3:20
#
.... your code goes here ...
```

3. Submit
To submit a file, upload it to your Stellar workspace. You may upload new versions of each file until the 11:59pm deadline, but anything uploaded after that time will be counted towards your late days, if you have any remaining. If you have no remaining late days, you will receive no credit for a late submission.

To submit a pset with one file, submit it through the Stellar submission page. Be sure that the top of your file contains a comment with the title of the file you are submitting, e.g.

```
# Problem Set 5
```

Also, after you submit, please be sure to open up your submitted file and double-check you submitted the right file. Please do not have more that one submission per file. If you wish to resubmit a file you've previously submitted, delete the old file (using the Stellar "delete" link) and then submit the revised copy.