

## Introduction

This is a worksheet for the seventh COMP3071 lab.

A `blackboard` system is a learning system where there is a collection of information (the blackboard) and a set of agents that can read information to and from the blackboard. One important feature of such a system is that it can be readily expanded by adding new agents. Because the agents don't interact directly, there is no need for a complex coordination mechanism—they just each read from and write to the board, in an arbitrary order, and, if the system is well designed, the information on the board becomes more relevant to the task.

This idea is illustrated in today's exercise by a creative language task. The information on the board consists of lines of text, and the overall aim is to make the text more poetic. To this end, we will create agents that will make lines rhyme, make them have the same length/rhythm, add metaphorical language.

The ideas for this worksheet were inspired by the following paper:

Misztal-Radecka, Joanna., & Indurkha, Bipin. (2016). A Blackboard System for Generating Poetry. *Computer Science*, 17(2), 265–294.

You can find this paper here: <https://journals.agh.edu.pl/csci/article/view/1576/1478>

There is no need to read this before you start the exercise, but if you want to take this kind of system further for your coursework project, then it would be a good idea to read it.

## Getting Started

To get started, download `BlackboardPoetry_viz.py`, `agent_ls.py`, and `unittest_lab7.py`.

a. `BlackboardPoetry_viz.py`

The `BlackboardPoetry_viz` file contains a Python class called `PoemBlackboard`, which has a variable named `self.lines` that contain a list of 4-list. These sub-lists contain the x and y positions of the line on the display, the text, and a unique ID for that line. The `__init__` method initializes this variable, sets up a window for display, and adds the lines to the board.

Next, the file has a `run` method that continuously loops, selecting an agent from `self.agentList` to execute, replacing the current lines with the new ones generated by the agent, and redrawing the board.

Under the Python main check '`__main__`', there are several variables. The `txt` variable holds the poem but structured as a list of strings, and the timer update feeds into the `self.canvas.after` method. Note that the first parameter in this method is the number of `milliseconds` delay between repeats of the loop, so you can increase this value to slow down the program for debugging purposes.

Additionally, the `agent_cons` variable is used to store the agents. All agent is available for import from the `agent_ls.py`. To add new agent, simply add the agent method name from the `agent_ls` Python file.

For example, to an agent name `remove_adjective` and `replace_with_synonym`, amend the following to the `agent_cons`

```
agent_cons = [remove_adjective, replace_with_synonym]
```

Execute the `BlackboardPoetry_viz` and observe the changes in the sentence as the windows are being updated.

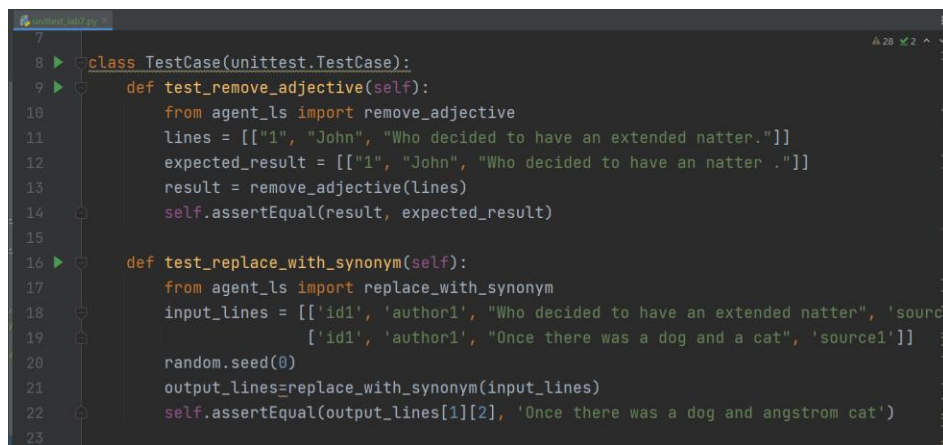
## b. agent\_ls.py

Currently, there are three agents available in the program, but only two of them are active. The first agent is called `remove_adjective`. This agent selects a random line, tags it with parts-of-speech, and removes one adjective that is tagged as "JJ". The second agent, `make_two_lines_rhyme`, takes two lines and replaces the word at the end of one line with a rhyming word from the other line. The third agent, `replace_with_synonym`, is currently inactive. It replaces a random word in a random line with a synonym.

Apart from that, there are several methods in the code with only commented skeletons that you will use to add a new agent to the system. To do so, you need to add a function to the code that takes a single parameter consisting of the lines and returns the revised set of lines.

## c. unittest\_lab7.py

To ensure that your code produces the expected output, I have created a unit test for each task. You can download the `unittest_lab7.py` file from the Moodle page to run these tests. If you are using JetBrains products like PyCharm or IDEA, you will find a **green gutter icon** (▶) on the left next to the line numbers for each method as shown in the image below. In the image, the gutter is available in lines 8, 9 and 16. Just click on the one for the test you want. Alternatively, once you have completed all the tasks, you can click the gutter next to line 8 to run the complete test unit.



For those using other IDEs, you can refer to the respective product documentation for more detailed information on running tests. For example, if you are using VS Code, kindly direct to the following link <https://code.visualstudio.com/docs/python/testing>

Worse case, you can comment out all the methods except the one you want to test and execute the `unittest_lab7.py` file.

Please note that the unit-test file is in its **infancy**. If you are confident that your code is correct and the unit test produces different results, please inform me. You are also welcome to make any necessary improvements to the unit test.

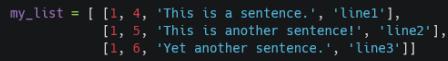
## Tasks

Today's task involving adding a new agent to the system under the `agent_ls` file. You will work with several agents namely. For each of the agent, you need to add a function to the code which has a single parameter consisting of the lines, and which returns the revised set of lines. Then, you need to add the name of your new agent to `self.agentList`.

1. `update_exclamation_mark_end_sentence`:

Add another agent to the system that randomly selects a line and adds an exclamation mark at the sentence's end, or remove one if it is already there.

For instance, given a list of lines,



```
my_list = [ [1, 4, 'This is a sentence.', 'line1'],
            [1, 5, 'This is another sentence!', 'line2'],
            [1, 6, 'Yet another sentence.', 'line3']]
```

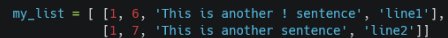
The agent should add the exclamation mark at end of the line "This is a sentence." and output "This is a sentence!". On the other hand, given the input, the agent should remove the exclamation mark from the line "This is another sentence!" and output "This is another sentence " (note the trailing space). At this stage, we did not care to improve the trailing empty space issue.

To implement this, you can refer to the `remove_adjective` or `replace_with_synonym` methods to randomly select the list of lines, and use a word tokenizer to create a list of words.

You may also consider using the `rstrip` method to remove any trailing characters. You can find more information about `rstrip` in the following [documentation](#):

2. `update_exclamation_mark_random_pos`:

Add another agent to the system that adds an exclamation mark next to a random word to a random line, or remove one if there is already one. [easy]



```
my_list = [ [1, 6, 'This is another ! sentence', 'line1'],
            [1, 7, 'This is another sentence', 'line2']]
```

The agent should remove the exclamation mark from the input 'This is another! sentence' and output 'This is another sentence'. Alternatively, it should remove the exclamation mark from the phrase 'This is another sentence' and output 'This is ! another sentence'

3. `swap_and_words`

- a. Add another agent that looks for a pair of words that are linked by the word "and", and swaps them round [fairly easy].

- i. For instance, in the sentence "once there was a dog and a cat", the program would swap the words "a dog" and "a cat".
- ii. To accomplish this, we need to perform the following steps for each string in the list
  - Tokenize the sentence.
  - Use the `pos_tag` function to tag each token and store the results in the `tagged_tokens` variable.
  - Extract the index of the "and" word in the `tagged_tokens` variable using a combination of for-loops and if-statements and store it in the `and_index` variable. In this example, the indices of the four words in the `tagged_tokens` list are 3, 4, 5, 6, and 7, which correspond to the words "a", "dog", "and", "a", and "cat".

Using the index numbers, extract the tagged tokens for the five words and store the result in the `word_pair_btw_and` variable.

```

idx_link_and = range(and_index - 2, and_index + 3)
word_pair_bt看_and = [tagged_tokens[i] for i in idx_link_and]

```

You should get something like

```

[ ('a', 'DT'),
  ('cat', 'NN'),
  ('and', 'CC'),
  ('a', 'DT'),
  ('dog', 'NN')
]

```

- Swap the pair of words in the `word_pair_bt看_and` variable and store the results in the `swap_words_bt看_and` variable, which should yield as below

```

[ ('a', 'DT'),
  ('dog', 'NN'),
  ('and', 'CC'),
  ('a', 'DT'),
  ('cat', 'NN')
]

```

- The next step is to insert the swapped words back into the original sentence. To keep the code organized and ease the next task, this operation is performed under the method `update_swap_words_existing_line`.

This method takes the following inputs: `tagged_tokens`, `idx_link_and`, `swap_words_bt看_and`, and `check_both_noun`.

For now, you can ignore the `check_both_noun` input, as it will be used in the next task.

Within the `update_swap_words_existing_line` method, the following for-loop is responsible for replacing the original words in `tagged_tokens` with the swapped words in `swap_words_bt看_and`.

```

for i, val in zip(idx_link_and, swap_words_list):
    tagged_tokens[i] = val

```

Review the code snippet and let me know if you have any difficulties understanding it.

- iii. To make the task more challenging [medium], you can modify the program to check that both words separated by "and" are nouns before performing the swap. This would mean that a sentence like "once there was a funny and a cat" would not be processed. However, if there are any descriptive adjectives with the nouns, the method will swap them

as well. For example, the program would swap "once there was funny cat and silly dog" for "once there was silly dog and funny cat".

- To this, ensure to set the `check_both_noun` to `True` when you want to apply this approach.
- Under the `update_swap_words_existing_line` method, particularly within the `if check_both_noun` block, check if the words adjacent to "and" are adjectives by evaluating whether the tag is "JJ". If such tags exist, you can swap the words. If not, you should not make any changes.

#### 4. `repair_a_an`

Note that when the rhyme agent changes a word, it doesn't change the "a/an" before the word. So, you can get something like "a associate". It would be possible to change the code in the rhyme agent to deal with this, but a more common approach in this sort of system is to make a new agent that does the corrections.

[fairly easy]

- To accomplish this, perform the following steps for each string in the list:
- Tokenize the sentence.
- Loop through the tokens and get the index and the word, which you can store in the variable `preceding_word`.

```
for idx, preceding_word in enumerate(tokens[:-1]):
```

- Check if the `preceding_word` is either a or an using [regular expressions](#).

```
r'^(a|an)$', preceding_word
```

- If the preceding word is either `a` or `an`, extract the succeeding word. In this example, the succeeding word is `associate`, which you can store in the variable `first_letter_of_succ_word`.
- Then, check if the first letter of the succeeding word is a vowel. If it is a vowel, the preceding word must be `an`; otherwise, it should be `a`. For example,

```
if first_letter_of_succ_word in vowels and preceding_word == 'a':
    tokens[idx] = 'an'
```

- Finally, join all the tokens and update the respective lines with the modified sentence.

#### 5. `generate_rhyming_lines`

*"Rhyme is the repetition of syllables, typically at the end of a verse line. Rhymed words conventionally share all sounds following the word's last stressed syllable"<sup>1</sup>.*

<sup>1</sup> Extracted from <https://clpe.org.uk/poetry/poetic-devices/rhyme>

The CMU Pronouncing Dictionary<sup>2</sup> is a computer-readable pronunciation dictionary developed by the Speech Group at Carnegie Mellon University (CMU). It contains over 100,000 words and their phonetic transcriptions using the ARPAbet symbol set. For example, the word "blossom" can be pronounced as `B L AA1 S AH0 M`, while "time" can be pronounced as `T AY1 M`. Similarly, the words "collar" and "dollar" can be pronounced as `K AA1 L ER0` and `D AA1 L ER0`, respectively.

Have a more careful look at the rhyme agent. Experiment with changing the “level” parameter—this controls how many syllables at the end of the line need to be matched to create a rhyme.

When the rhyming level is set to 1, the rhymes are not very strong and only involve the last syllable of words. For example, the words "blossom" and "time" share the same ending sound, represented as **M** in the pronunciations `B L AA1 S AH0 **M**` and `T AY1 **M**`, respectively, and are therefore considered to rhyme.

At a rhyming level of 2, the rhymes are stronger and involve the last two sounds of the words. For instance, the words "collar" and "dollar" share the same last two sounds, represented as **L ER0**, in their pronunciations `K AA1 **L ER0**` and `D AA1 **L ER0**`, respectively. Therefore, they are considered to rhyme at level 2.

The main problem with the rhyme agent is that it generates words that rhyme but which don't match the meaning. Instead of replacing the word with an arbitrary rhyming word, we want to replace it with an (approximate) synonym.

Take a look at the `replace_with_synonym` agent, which selects a random line and substitutes one word with a synonym.

Use this idea to create a new agent under the method `rhyme_and_make_sense` [fairly challenging]. This can be achieved

- Take for example, there are two lines from the poem

```
line 1: "Once there was a dog and a cat"
line 2: "Who decided to have an extended natter"
```

- The first step was already accomplished, which involves creating a list of words that rhyme with the word at the end of the first line.

For instance, in this example,

```
line 1: "Once there was a dog and a cat"
```

we will find the rhymes for the pronunciation of the word "cat" at level 1, and store the information under the variable name "rhymes," using the following code snippet:

<sup>2</sup> <http://www.speech.cs.cmu.edu/cgi-bin/cmudict>

```
def make_two_lines_rhyme(lines, level=1):
    last_word1, last_word2, lines, ll1, ll2 = pop_lines_extract_last_word(lines)
    entries = nltk.corpus.cmudict.entries()

    # Get the pronunciation of the last word of the first line
    syllables = [(word, syl) for word, syl in entries if word == last_word1]
    rhymes = match_by_level(syllables, entries, level=level)
```

The variable `rhymes` containing the list of rhyming words of the last word of the first line. Both the variable `rhymes` as well as the last word from the second line (e.g., `last_word2`) will be provided as input arguments to the `rhyme_and_make_sense` method. This will allow the method to determine which words from the list of rhyming words also have a synonym that matches the last word of the second line.

- Next, implement the following steps under the `rhyme_and_make_sense` method, made the following
  - o Create a list of words that are synonyms of the word stored in the variable `last_word_2` and store the result in a new variable called `synonyms_unique`
  - o Find the intersection between `synonyms_unique` and `rhymes` and store the result in a new variable called `synonyms_that_rhyme`.
  - o Return the `synonyms_that_rhyme`.
- To use this agent, uncomment the following code under the method `make_two_lines_rhyme` and call the `rhyme_and_make_sense` method from there.

From

```
# Find rhyming synonyms of the last word of the second line
# rhymes = rhyme_and_make_sense(last_word2, rhymes)
```

to

```
# Find rhyming synonyms of the last word of the second line
rhymes = rhyme_and_make_sense(last_word2, rhymes)
```

- You should now get pairs such as:
 

“Once there was a dog and a cat; who decided to have an extended chat”

which correspond to the CMU pronunciation of cat: `K AE1 T` and chat: `CH AE1 T` for a 1-level rhyme

## 6. `add_sensible_adjective`

The agent `add_sensible_adjective` is incomplete. This uses a corpus of text to decide which adjectives are meaningful for a particular noun, by seeing which adjectives have previously been found before that noun before.

There is a gap in the code (fairly challenging), and you need to build up the program to achieve this goal under the method `noun_selector`. Make the following

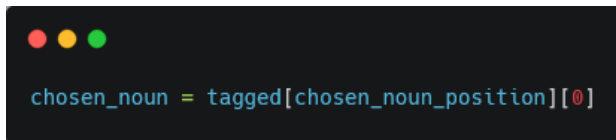
- Tokenise the sentence
- tag it with parts-of-speech.
- Search the resulting list for nouns and set the variable `noun_positions` to the positions of those words,

For example, if the sentence was "Once there was a dog and a cat,"

the value of `noun_positions` might be `[4, 7]`, where index 4 and 7 correspond to the positions of 'dog' and 'cat', respectively.

Randomly select an index position from the `noun_positions` list and save it to a variable called `chosen_noun_position`.

To slice the noun from the tokenized word, you may consider something as below



```
chosen_noun = tagged[chosen_noun_position][0]
```

which will set the `chosen_noun` to "cat" if the `chosen_noun_position` is randomly selected to 7.

- If your code works perfectly, you should get lines such as "Once there was a dog and a **scared** cat".

### Extensions

There are many possible extensions, each of which you implement by implementing a new agent and adding it to the list. Implementing a few of these and examining how well they interact would be a good coursework project for this module.

- Create an agent that takes two lines and tries to make them the same length, by adding and removing adjectives and/or replacing words with synonyms. The easier way to do this is to measure length in terms of number of letters. A more appropriate way to do it would be to convert the words into syllables (see `make_two_lines_rhyme` for details of this) and make them the same number of syllables. (fairly challenging)
- Experiment with alliteration, i.e. sequences of words that begin with the same letter or sound. Make a copy of `replace_with_synonym` that tries to use synonyms that have the same letter or sound as the word before or after. (fairly challenging)
- Improve `replace_with_synonym` by only replacing words with the same part of speech (nouns with nouns, adjectives with adjectives, etc.) (fairly challenging)
- Experiment with "synonyms of synonyms" in `replace_with_synonym` (fairly easy)
- Extend `replace_with_synonym` by finding *hypernyms* (words that represent specific types of that word e.g. "dalmatian" rather than "dog") and replacing a word with one of its hypernyms. (not too hard once you have found the relevant functions in wordnet)
- Find which lines contain multiple adjectives before a word, and remove all but one of those adjectives (fairly easy)
- Find out about antonyms (words that mean the opposite to a word) and how they are handled in nltk and WordNet, then write an agent similar to `replace_with_synonym` that replaces words with their antonym instead (fairly easy)
- An agent that copies a line and places it in a different position on the blackboard (easy)
- Restrict the vocabulary to the most common 1000 words in the language (fairly challenging)



- Incorporate the metaphor agent from the previous class into this system (fairly challenging)
- Write one or more agents that *evaluate* existing lines, e.g. against a target number of lines, for grammaticality, etc. If the lines are good, duplicate them, move them to the top of the screen, etc.; if they are too bad, remove them. (variable difficulty depending on the task)
- Add in an agent that looks for *existing* pairs of lines that rhyme and moves them closer together. Make this do this for more complex patterns, e.g. assembling a list of four lines where the first and third lines rhyme and the second and fourth do. (fairly challenging)
- Move text that is overlapping to elsewhere on the board (fairly easy)
- Have a separate category on the blackboard for “ideas” or “theme” and only replace words with words that are “close enough” to the theme—e.g. “synonyms of synonyms of”, hyponyms of, mentioned close to a theme word in a corpus, etc. (difficult)
- Have a look at the “creative language retrieval” paper <https://www.aclweb.org/anthology/P11-1029.pdf> and incorporate some ideas from that into this system (difficult)
- Get the starting text from, e.g. a news story on the web (fairly challenging)