Girls' Programming Network

Markov Chains
Extensions

Extend your program to get more text from files and to incorporate many different texts to make a mash up!

This project was created by GPN Australia for GPN sites all around Australia!

This workbook and related materials were created by tutors at:

Sydney, Canberra and Perth

Girls' Programming Network

If you see any of the following tutors don't forget to thank them!!

Writers

Maddy Reid
Renee Noble
Pauline Kelly
Olivia Ross
Manou Rosenberg

Testers

Anton Black
Olivia Gorton

A massive thanks to our sponsors for supporting us!

ATLASSIAN

Extension 1: Reading Texts!

We will work on some really nice extensions. Before you start with this workbook, make sure you have gone through the second workbook. We will extend our code from there.

Task 1.1: Download a text

Instead of copying and pasting the text, let's make python do the reading for us! We'll get our text from a file.

1. Go to the link below and **download** a file that looks interesting.
   girlsprogramming.network/markov-files

2. Save it as something like harry_potter.txt in the **same location** as your python file.

Task 1.2: Reading a file

Now that you have selected your favourite text file, we want python to read it.

1. At the top of your file (*after the imports*) create a new variable and **store the name of your chosen file** as a string. *Don't forget the file extension on the end .txt.*

2. **Go to** where your program reads its text from currently (where you store the text in a huge string). **Comment** out this line. *(Put a **#** at the start of the line)*

3. On the following line, **open** the file and **read** the content.
   Don't forget to **store** the text inside in a variable.

   *Use the **same variable** name to store the text you read from the file as the one you used to store your big string. Your program already expects text to be in there!*

Hint Reading/writing files

Remember that reading from a file looks like this:

```
with open(my_filename_variable, "r") as input_file:
    data = input_file.read()

# Continue writing your program here
```

Task 1.3: Run your code!

Run your code!

- What do the sentences you generate look like now?
- Are there any differences in the readability or meaningfulness in the sentences?
- Why do you think that is?

Error Help: Handling the UnicodeDecodeError

You might encounter a **UnicodeDecodeError** when trying to read in a file:

```
UnicodeDecodeError: 'charmap' codec can't decode byte 0x9d in
position 103302: character maps to <undefined>
```

This happens when Python doesn't know how to interpret the way the file is written and stored! It can be resolved by specifying the **encoding** used by the file, usually it's **UTF-8**.

You can specify the encoding like this:

```
text_file = open("somefile.txt", "rU", encoding="utf8")
```

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Part 2:

- ☐ Your program generates random sentences
- ☐ The program reads the text in from a file
- ☐ You've tried some different source text files

★ Challenge - Store your output in a file ★

If you wanted to create a very long generated text you wouldn't want to print it all out! See if you can work out how to save your generated text to a file!

Hint Reading/writing files

Similarly, we can write data! Just make sure you specify the name of the file you want to store the data in.

To write to a file one word at a time we need to **append** each word to the file. We can **open** a file in append mode by using the **"a"** option like this:

```
data = "Here is some data"
with open("written_data.txt", "a") as output_file:
    output_file.write(data)
```

Extension 2: Longer Texts!

Task 2.1: Find your own files!

You can find texts you like on the internet to use in your markov chain making!

1. File some text you like on the internet. Copy and paste it to a file (using a text editor on your computer like Notepad), save it as SOMETHING.txt

2. Update your code to read text from your new file. Run it and see what it looks like!

Task 2.2: Find your own files!

Did you notice anything when you ran your code? Did the text you generated have crazy punctuation and capitalisation?

The text you chose might be a little messier than the ones we've given you.
You might need to remove:

- Capital letters
- Punctuation
- New lines/Enters

We can replace all these things we don't want in our text!

Use replace() to try removing different kinds of punctuation from your text. Look at the results and decide what you like best

Hint

You can replace text in a string using .replace() like in these examples:
This would replace a question mark with nothing (the empty string).

```
my_string = "Hi, how are you doing?"
my_string = my_string.replace("?", "")
```

This would replace a new line/ENTER with a space.

```
my_string = """Hi, how are you doing?
                I really like cats"""
my_string = my_string.replace("\n", " ")
```

Enter is represented by \n, the n stands for New line.

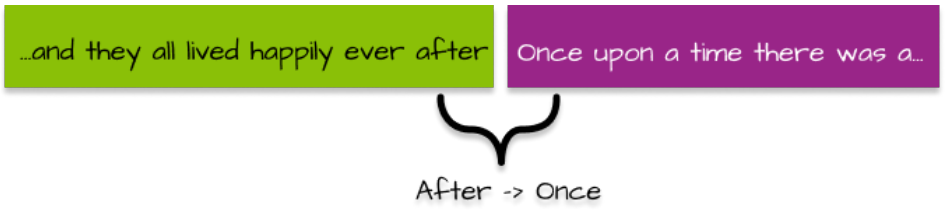
You can also make text all lowercase like this:

```
my_string = my_string.lower()
```

Extension 3: One file is not enough!

Having just one single file to train our text generator on might not be enough. Maybe we want to read multiple files and then mash them together!

But we can't just stick all of our texts in one file, one after another!
If we stick to texts next to each other we'll end up with some bigrams that **aren't correct**



"after" and ***"once"*** don't **actually** go next to each other, and sound a bit silly together. We don't want incorrect things like that in our dictionary! So, we need to read the files separately.

We'll do this by **repeating** our dictionary building code to add new bigrams **for each text**.

Task 3.1: Multiple files!

To train our code on different source-text files we first select our files!

1. Go back to the website to get more texts! Put at **least two** you want to mash up in the folder with your python file.

2. Go to the top of the file to where you stored your file name. We need to have multiple files now, so **delete** this line. (we'll replace it in the next step_

3. In that space, create a list. Fill it with all the names of the files you want to mash up, in string form. Remember to assign your list to a variable, **eg:** `texts.`

Task 3.2: Put it in a loop!

We need to **repeat** reading the file and adding its bigram to the dictionary for each file, so we'll use a loop to do this:

1. Go to where you open and read the file, **create a new line above**. On the new line **create** a **for loop** that loops through **each file name** in the list you created.

2. **Indent,** the code that builds the dictionary to be inside the loop.

3. Make sure you update where you **open** the file to use the **loop variable**.

4. Make sure that the place you create the empty **cups dictionary** is **above** the loop.

Task 3.3: Try it out!

Run your code!

- Is your text mashed up?
- Try out your code a couple of times with different start words.
- Try it with some different files!

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Part 3:

- ☐ The program reads the text in from several text files
- ☐ The program uses the texts from all of the files to train the Markov model
- ☐ You have tried out your code with different start words

Extension 4: A Smarter Generator

Sometimes our generated text does make a lot of sense!

It might generate something like "**I'm going to Friday**", it doesn't know that Friday isn't somewhere you go. All it knows is that "**to Friday**" is a pattern that can occur, eg: "**Monday to Friday**".

How do we make our generator smarter? By giving it more information with Tri-grams!

What are Tri-grams?

What we were writing before were bi-grams. This is where we had:
"Cat" -> "in"
"in" -> "the"

Now let's do tri-grams! They are better than bi-grams, because when we choose a random word, they will have more context of what we have already said.
"The Cat" -> "in"
"Cat in" -> "the"

As you can see, we still have a key and a value pair, however they are better and contain more knowledge than before.

Task 1: Building a tri-grams dictionary

If you have a look at your previous code, you can see that we were creating bi-grams (where 1 word leads to another word, a pair of words).

Try to modify your code so that you are creating a dictionary of tri-grams (where 2 words lead to a new word, a trio of words)! You will need to update your cup making code and your text generating code.

Check out the helper dictionary below to see what your `cups` dictionary will look like now!

Hint

You can find the helper dictionary on the website as indicated by your tutor.

It will look something like this...

```
{'said Loonquawl': ['with', 'nervously,'],
'came to': ['life'],
'even so': ['they'],
'Loonquawl nervously,': ['do']... }
```

Watch out when you create your dictionary not to look past the end of your list of words!

Task 2: Generating with trigrams

Update the code that generates your new text to remember the last two words that were generated. Join these together to get a key so you can randomly select the next word.

Run your code, but provide two words as your start word, eg: “even so”.

Task 3: Bigrams Plus Trigrams!

Now you’ve got your trigrams working, it would be good if you could just enter a single starting word. But now our cups dictionary dones’ know any singular keys!

Work out how to add both the bigrams you used to have, as well as your trigrams to the dictionary. Then you can start with a singular word!