Text Generation with Markov Chains
Advanced Workbook

Girls' Programming Network

Updating our Text Generator

Now that we've written our text generator, we want to make it more robust and useful! These activities will build upon code that we've already written.

Exercise 1: Reading Files

In order to train our Markov Chain better, we want to give it lots of sample text! However, this isn't particularly nice to do by asking the user for input. Instead, we'll write all the sample text to a file, and then get python to read the file.

Learning activities:
Step 1: Ask the user for a file name
Hint: You'll need to make sure you put the file in the same directory as your python file.

Step 2: Read from a file!
Hint: You'll need to open the file in read mode, using `"r"`!

Step 3: Write to a file!
Hint: You'll need to open the file in write mode, using `"w"`!

Activity:
We need to ask our user for a filename that we can read our sample text from! Open up the file, and store the text in a variable.

At the end of this exercise, your program should:

- ❏ Ask the user for a file name
- ❏ Open the file and read the text
- ❏ Store the text in a variable
- ❏ Print out a message thanking the user

Exercise 2: Case Insensitive

When checking to see if a sentence contains a certain word, we need to make sure it finds it even if we've written the word in uppercase characters, lowercase characters, or a mix of both!

Learning activities:

Step 1: Convert a word to all uppercase characters!

Hint: You can use `myText.upper()`*!*

Step 2: Convert a word to all lowercase characters!

Hint: You can use `myText.lower()`*!*

Activity:

Update your if statement that checks if "sam" is here to be case insensitive, so it can find "sam", "Sam" and "SAM".

At the end of this exercise, your program should:

- ❏ Convert each word being checked to either lowercase or uppercase.
- ❏ Compare each word to see if matches "sam" if you are using lowercase or "SAM" if you are using uppercase.

Exercise 3: Using a counter!

Instead of looping through a sentence and counting each word, we can use a counter to count the words for us! A counter will count how many times something occurs in a list.

Learning Activities:

Step 1: Create a counter

Hint: Try using `help(collections.Counter)`

Step 2: Update the counter with the string “aabbccdeeaaddee”. How many times does “a” occur?

Step 3: Count how many times the word `"I"` appears in the string `"I went to the shops and I had a good time!"` using the counter.

Step 4: Convert your counter to a dictionary!

Hint: You will need to use `dict()`.

Activity:

Update your dictionary that stores the count of each word by using a counter instead of the for loop.

At the end of this exercise, your program should:

- ❑ Import counter from the collections library
- ❑ Split your sample text into a list
- ❑ Create a counter variable
- ❑ Update your counter variable with the list that you created from your sample text
- ❑ Convert your counter to a dictionary
- ❑ Print your dictionary out.

Exercise 4: Store the next word!

Now that we've created bigrams, let's create trigrams as well! This will help improve the sentences we generate by increasing what the Markov Chain knows.

Learning Activities:

Step 1: Create a tuple that contains the following items: "blue", "red" and 5

Hint: You can use `myTuple = ("item1", "item2", item3)`

Step 2: Create a list of tuples! Do one of your favourite books and their authors.

Hint: You can use `myList = [(apples, bananas), (tomatoes, lettuce)]`

Step 3: Create a dictionary with a tuple as the key!

Hint: You can use `myDict = {(apples, bananas): "are my favourite fruits!"}`

Step 4: Create a dictionary with a tuple as the key and a list as the value

Hint: You can use `myDict[("apple", "banana"): ["Yummy", "healthy"]}`

Activity:

Instead of just storing the current word as the key in the dictionary, let's use a tuple containing the current and next word. The value will then be a list of the third word in the trigrams. You will need to create a new for loop to go over the sample text to generate your dictionary. Then, update your while loop used to generate your sentence to use the previous word and the current word as the key to randomly select the next word.

At the end of this exercise, your program should:

- ❏ Create a new dictionary
- ❏ Create a new for loop that goes over the sample text
- ❏ Store the current word and the next word as a tuple
- ❏ Check to see if the tuple is already a key in the dictionary.
    - ❏ If it isn’t, add it, and store the next word that comes after those two as an item in a list of for the value
    - ❏ If it is, add the next word that comes after those two as an item in the list.
- ❏ Print out your dictionary
- ❏ Update your while loop so that if the previous word added to your Markov Chain and the current word form a key in the trigram dictionary, select a word from that key’s list instead of from your bigram dictionary.

Exercise 5: Select a starting word and ending word!

Rather than asking a user for a starting word, let's select one randomly by using the `random()` library! We've already used the random library before, so let's extend it and make sure we're not selecting a word that has a full stop.

Learning Activities

Step 1: Check to see if a word starts with an uppercase character

Hint: You can use `if myText.isupper():` to do this.

Step 2: Loop over a sentence and count all the words that end with the letter H

Hint: You can use `myText.endswith("o")`

Activities

Select a word from random and check if it starts with an uppercase character. If it does, start your Markov Chain with it. If it doesn't, select a different starting word from random. During the creation of your Markov Chain, check to see if the selected word ends with a full stop. If it does, print your completed sentence!

At the end of this exercise, your program should:

- ❏ Import the random library
- ❏ Select a word from random as the starting word from your bigram dictionary
- ❏ Check to see if the starting word begins with an uppercase character
    - ❏ If it does, check to see if it ends with a full stop
        - ❏ If it ends with a full stop, select another starting word
        - ❏ If it doesn't, begin your Markov Chain
    - ❏ If it doesn't start with a capital letter, select another starting word
- ❏ When adding the next word to the generated sentence, check to see if the word ends with a full stop
    - ❏ If it does, end your Markov Chain and print your sentence
    - ❏ If it doesn't, keep going!

Exercise 6: Generate a paragraph!

We can already generate a sentence, but let’s go further! Let’s generate a whole paragraph!

Learning Activities:

Step 1: Ask the user for a number, and then convert it to an integer!

Hint: You can use `int(myvariable)` *to do this!*

Step 2: Convert the integer you just created back into a string!

Hint: You can use `str(myvariable)` *to do this!*

Activity:

Ask the user how many sentences that they would like to have written. Using a for loop, create as many sentences as the user asked for!

At the end of this exercise, your program should:

- ❑ Ask the user for how many sentences they would like written.
- ❑ Convert the user input to an integer.
- ❑ Create a for loop that runs as many times as the user asks
- ❑ Within the for loop, create a sentence each time the loop runs

Exercise 7: Creating functions!

Functions are really useful for recycling code. They can be used to hide code away so that our main block of code is really easy to read. We've already seen several examples of functions, including `print()`, `input()`, `random()` and `counter()`! Now it's time to write our own.

Learning Activities:

Step 1: Create a function that adds two numbers together and returns the sum of the numbers

Hint: You can use

```
def Sum(a, b):
      # Insert code here that adds the numbers together
      return myResult
```

Step 2: Create a function that takes an integer and a string and creates a sentence!

Hint: You can use

```
def Sentence(a, b):
      # Insert code here that creates a sentence
      return mySentence
```

Activity:

Move your for loop that creates your Markov Chain paragraphs to a function which accepts the number of sentences to generate as input.

At the end of this exercise, your program should:

- ❑ Do exactly what it did before, but now it will call functions!