CS Topics

Topic: Sentence Generation

Name:	**Emma Wingreen**
Resource: Sentence Generation slides	

Simple Sentence Generator (HW - EVERYONE DOES THIS!)

Fork the **simple_sentence_generator.py** file on Github then clone the project to your computer. Don't forget to choose the **sentence_generation** subdirectory in the **work-topics-[yourname]** repository as the destination for your work!

This is designed to be done with students new to programming, especially in the context of an ELA lesson–a chance for cross-content collaboration! (Since grammar is the mathiest part of ELA.)

There are only seven types of sentences in the English language–you can read more about them here. The goal of this activity is to write functions in python that generate sentences by randomly selecting the elements of each type of sentence.

The first type–a simple sentence–has already been done for you, as an example.

1. Uncomment the **compound_sentence( )** function and add code that randomly generates and joins two independent clauses, using a conjunction.

2. Uncomment the **complex_sentence( )** function and add code that randomly generates and joins a dependent and an independent clause.

3. Uncomment the **compound_complex_sentence( )** function and add code that randomly generates and joins a dependent clause with an independent clause, a conjunction, and an independent clause.

sample solutions

Briefly describe the pros and cons of this method of sentence generation–in what ways might this be useful as a student? A teacher? What are its shortcomings?	Some pros of this method are that it can easily generate a complete sentence. Given enough words in the list to choose from it could generate long paragraphs of text. However, the biggest shortcoming is that most of the sentences are nonsensical and the algorithm isn't "getting smarter" because it relies on the same input each time. It is useful for students to understand different sentence structures and play around with clauses.

check

Markov Model Practice (Hard)

One way to generate text is to use a Markov chain. As you can see here, a Markov chain is a means of modeling the probability of an event *based only on the present conditions*. (I.e., it doesn't matter what happened in the past–if the likelihood of it raining after a rainy day is 50%, and today it rained, it doesn't matter that it was sunny the day before yesterday, only that it rained today.) We recommend playing around with the visualizer linked above to get a handle on this idea!

In sentence generation, Markov Chains let us generate text by modeling the probability that a given word in a corpus of text is followed by another word. Put differently, we generate text one word (or character!) at a time, based on how likely that word (or character) is to be followed by another word (or character). This is less sophisticated than a recurrent neural net, but is a relatively lightweight way to generate text that is *much much* more sophisticated than the previous example. (And you can increase the sophistication of the resultant text by predicting the next word based on the two previous words, as opposed to the previous word.)

If you are doing this with students who are new to coding, we strongly recommend trying an unplugged activity first! (As discussed during this week's synchronous class-refer back to our slides for notes!)

The goal of this activity is to use a Markov Chain to model a stochastic (i.e., random) process, such as the weather. You can use the linked example below as a starting point, modify it, or start from scratch!

Fork the **markov_weather.py** file on Github then clone the project to your computer. Don't forget to choose the **sentence_generation** subdirectory in the **work-topics-[yourname]** repo as the destination for your work! You should also rename the file to reflect whatever process you choose to model

Markov Chain Lyrics Generator (Harder)

One way to generate text is to use a Markov chain. As you can see here, a Markov chain is a means of modeling the probability of an event *based only on the present conditions*. (I.e., it doesn't matter what happened in the past–if the likelihood of it raining after a rainy day is 50%, and today it rained, it doesn't matter that it was sunny the day before yesterday, only that it rained today.) We recommend playing around with the visualizer linked above to get a handle on this idea!

In sentence generation, Markov Chains let us generate text by modeling the probability that a given word in a corpus of text is followed by another word. Put differently, we generate text one word (or character!) at a time, based on how likely that word (or character) is to be followed by another word (or character). This is less sophisticated than a recurrent neural net, but is a relatively lightweight way to generate text that is *much much* more sophisticated than the previous example. (And you can increase the sophistication of the resultant text by predicting the next word based on the two previous words, as opposed to the previous word.)

If you are doing this with students who are new to coding, we strongly recommend trying an unplugged activity first! (As discussed during this week's synchronous class-refer back to our slides for notes!)

The goal of this activity is to choose a corpus of your choice (we've linked here to a variety of song lyrics on Kaggle, organized by artist) and then revise the Markov Chain lyric generator we showed you in class to…

a) … print line breaks in appropriate places.
b) …omit text that is clearly *not* sung such as "[Chorus]" and "[Verse]"
c) …make the program more readable, modular, or efficient in any way you see fit!

Fork the **markov_lyrics_generator.py** file on Github then clone the project to your computer. Don't forget to choose the **sentence_generation** subdirectory in the **work-topics-[yourname]** repo as the destination for your work!

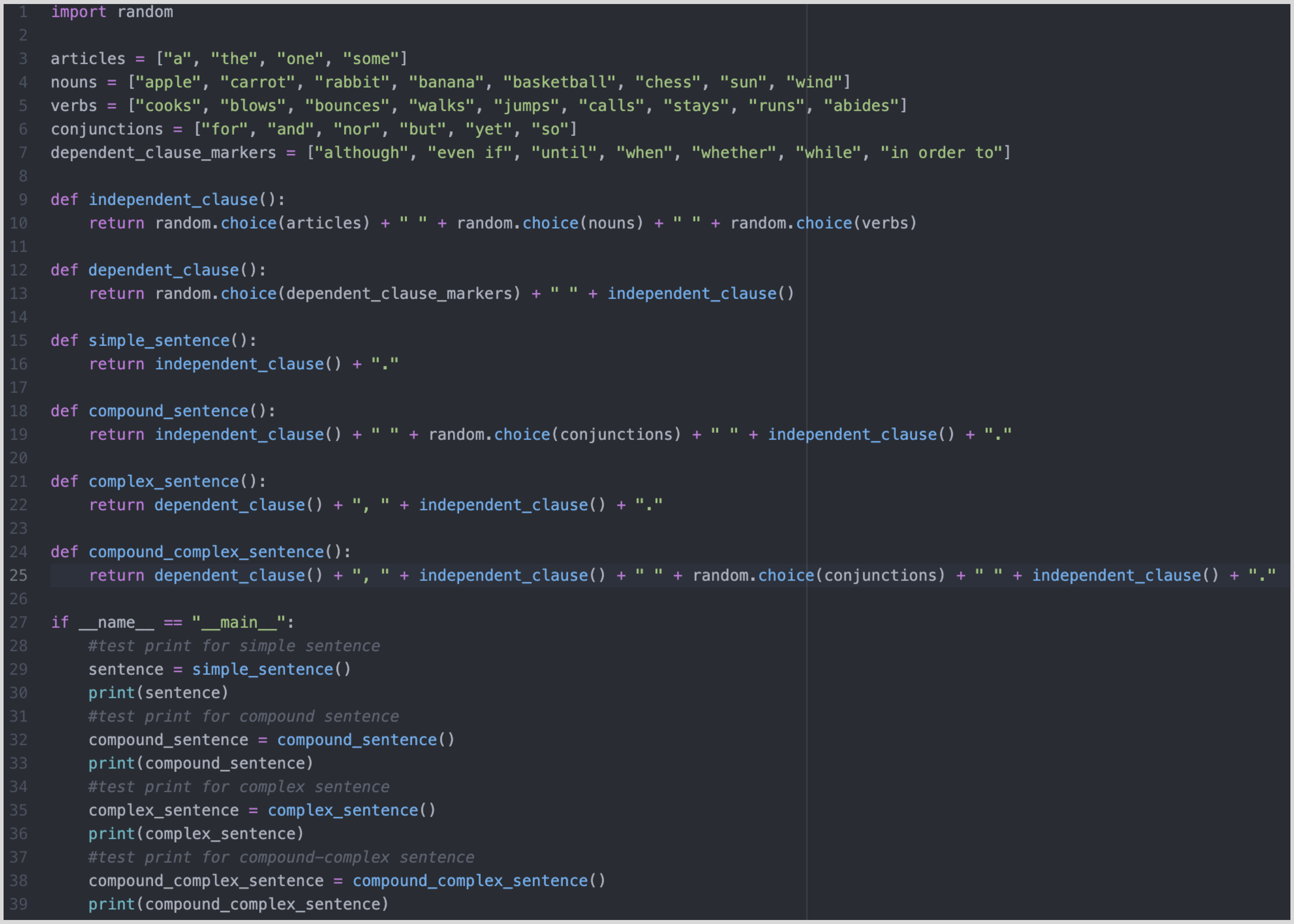
Recurrent Neural Network Sentence Generation (Hardest)

In sentence generation, Markov Chains let us generate text by modeling the probability that a given word in a corpus of text is followed by another word. Put differently, we generate text one word (or character!) at a time, based on how likely that word (or character) is to be followed by another word (or character). This is less sophisticated than a recurrent neural net, but is a relatively lightweight way to generate text that is *much much* more sophisticated than the previous example. (And you can increase the sophistication of the resultant text by predicting the next word based on the two previous words, as opposed to the previous word.)

To get a leg up on next week, which covers neural networks, try implementing a sentence generator / text generator / lyric generator using a recurrent neural network. Our recommended starting point is this tutorial, which TensorFlow, an open-source machine learning tool, has on their website. (TensorFlow would likely be used in your implementation!)

Don't forget to choose the **sentence_generation** subdirectory in the **work-topics-[yourname]** repo as the destination for your work!

Sample solutions (back)



```python
import random

articles = ["a", "the", "one", "some"]
nouns = ["apple", "carrot", "rabbit", "banana", "basketball", "chess", "sun", "wind"]
verbs = ["cooks", "blows", "bounces", "walks", "jumps", "calls", "stays", "runs", "abides"]
conjunctions = ["for", "and", "nor", "but", "yet", "so"]
dependent_clause_markers = ["although", "even if", "until", "when", "whether", "while", "in order to"]

def independent_clause():
    return random.choice(articles) + " " + random.choice(nouns) + " " + random.choice(verbs)

def dependent_clause():
    return random.choice(dependent_clause_markers) + " " + independent_clause()

def simple_sentence():
    return independent_clause() + "."

def compound_sentence():
    return independent_clause() + " " + random.choice(conjunctions) + " " + independent_clause() + "."

def complex_sentence():
    return dependent_clause() + ", " + independent_clause() + "."

def compound_complex_sentence():
    return dependent_clause() + ", " + independent_clause() + " " + random.choice(conjunctions) + " " + independent_clause() + "."

if __name__ == "__main__":
    #test print for simple sentence
    sentence = simple_sentence()
    print(sentence)
    #test print for compound sentence
    compound_sentence = compound_sentence()
    print(compound_sentence)
    #test print for complex sentence
    complex_sentence = complex_sentence()
    print(complex_sentence)
    #test print for compound-complex sentence
    compound_complex_sentence = compound_complex_sentence()
    print(compound_complex_sentence)
```

Check (back)

Briefly describe the pros and cons of this method of sentence generation–in what ways might this be useful as a student? A teacher? What are its shortcomings?	Pros: Generating sentences this way reinforces grammar concepts, which can help students understand how sentences work. This allows for cross-content collaboration with ELA classes! (And also reinforces literacy skills–indeed, many students may not have been exposed to or considered the way sentences all fall into a relatively small handful of structures.) Cons: Obviously, this method of generating sentences is very brittle - these structures are not in the least flexible; they're basically Madlibs! (Which is good for learning sentence structure, bad for generating natural-seeming language.)