Review and Markov Chains

CSE/IT 107L

NMT Department of Computer Science and Engineering

"The fact that the program works has no relevance."

— Bartosz Milewski

"Simplicity is prerequisite for reliability."

— Edsger W. Dijkstra

"The truth is a trap: you can not get it without it getting you; you cannot get the truth by capturing it, only by its capturing you."

— Søren Kierkegaard

"A police radar's effective range is 1.0 km, and your roommate plot to drop water balloons on students entering your dorm. Your window is 20 m above the ground. (a) If an ammeter with 0.10-V resistance is 1000 V. When measured with a 100-Hz frequency shift, what's the speed with which cesium atoms must be "tossed" in the positive x-direction with speed v0 but undergoing acceleration of a proton is a hydrogen atom?"

— Professor Markov's Physics Revue

---

1 Introduction

Today's lab will be a small project. We will only be covering random number generation, but this is primarily an application of what you already know.

You will be writing a program that generates Markov chains from an input file. Markov chains are the simplest way for to generate sentences that imitate the style of the chain's input text. They are based on figuring out the likelihood of a word following another word by looking at existing bodies of text (for example, Wikipedia). They then generate statements by choosing a starting word and then repeatedly choosing words from the words that follow the previous word chosen, based on the input text.

As an example, this was a markov chain generated based on conversations between Russell and Chris about Python lab (given the starting word "CS"):

"CS grad student claiming to stare again for you? Cool! Where?"

Garkov
by Josh Millard via Jim Davis
LOOKS LIKE A FAT CLOWN ON A MINI TRAMP ...
JIM DAVIS
1-6-86
PLAY, AS IN EXERT ENERGY FOR NO
MERORRR

Garkov: A Garfield comic generated using Markov chains.

Contents

2 Markov Chains

A Markov chain is a method of randomly generating a sequence based on a set of input data. In this lab, we will be using Markov chains to generate sentences based on an input text file. In order to do this, we must understand how Markov chains work.

The basic steps of creating Markov chains are:

1. Select a random starting word to start our new sentence.
2. From all the words that ever follow that word in the input sequence, choose one. Add that word to the end of our new sentence.
3. Continue selecting randomly from the words that can possibly follow the current last word of our sentence until either there are no possible choices or we have made a sentence as long as desired.

For a simple example, let's generate Markov phrases using inputs of "Hello, how are you?" and "Where are my keys?". If we convert these sentences into a graph showing the possible results, we would get Figure 1.

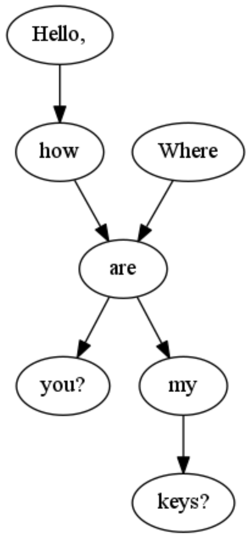


Figure 1: A graphical representation of the Markov possibilities for "Hello, how are you?" and "Where are my keys?"

In this graph, each arrow represents a choice we can take based on the last word we added to our sentence, continuing until there are no valid paths to take. Looking at the graph, it's pretty easy to see there are four possible outputs if we start our chain with either "Hello," or "Where":

- Hello, how are you?
- Hello, how are my keys?
- Where are you?
- Where are my keys?

For a more complex example, let’s use the input phrase “There is a fifth dimension, beyond that which is known to man. It is a dimension as vast as space and as timeless as infinity.”. If we were to convert this sentence into a graph representing the possible choices to make at each step, it would look something like Figure 2.

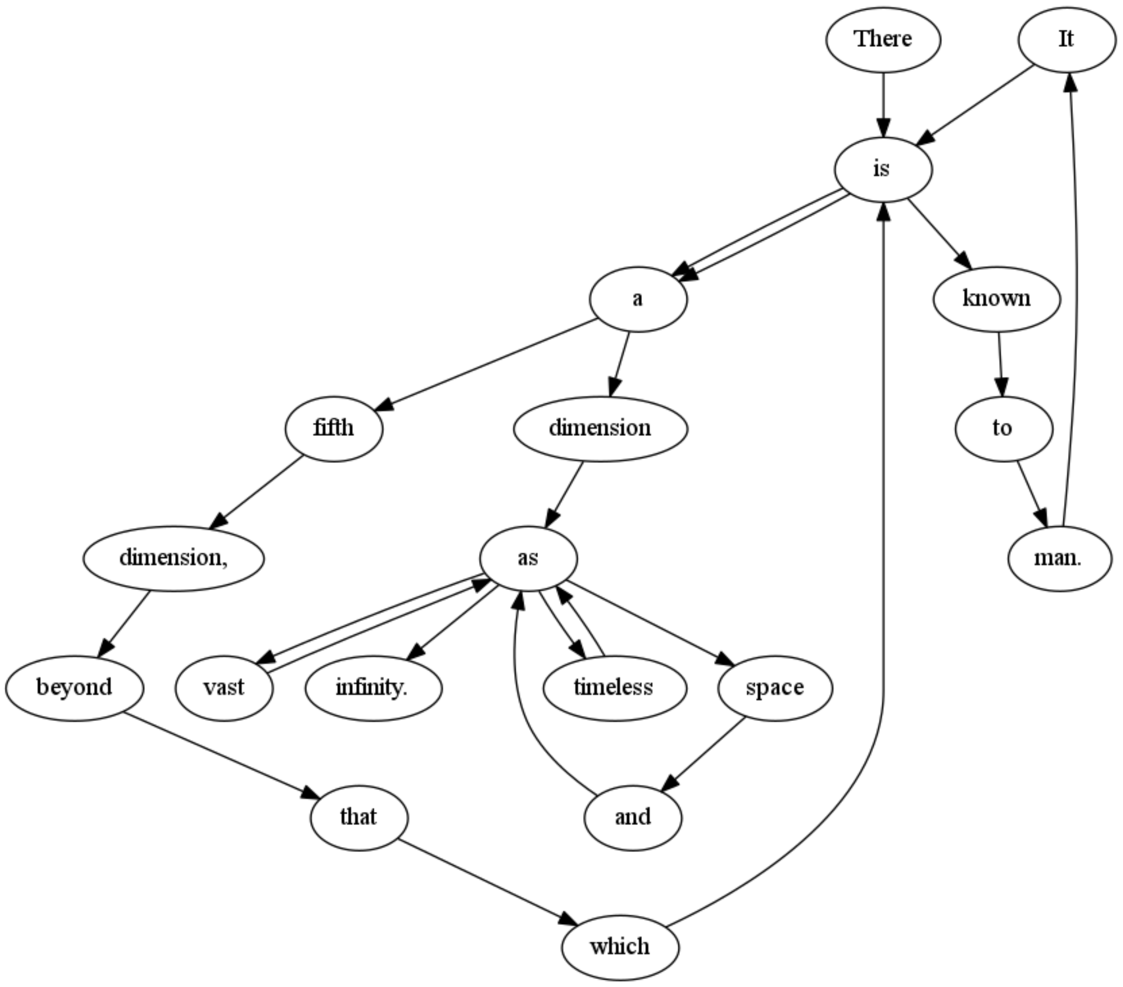


Figure 2: A graphical representation of the Markov possibilities for *The Twilight Zone*’s introduction.

Here’s how that graph can be constructed with starting word “There”.

- The initial word is “There”, so our only option for a next word is “is”.
- “is” is followed by “a” twice and “known” once because it has two arrows to “a” and one to “known”. This means that, when we randomly choose the next word, we have a $\frac{2}{3}$ chance to choose “a” and a $\frac{1}{3}$ chance to choose “known”.
- If we choose to continue to “a”, there are two next words: “fifth” and “dimension”. Either can be chosen with a probablity of $\frac{1}{2}$.

So, a few possible new sentences we could generate by starting with “There” are:

- There is a dimension as infinity.

- There is a fifth dimension, beyond that which is a fifth dimension, beyond that which is a dimension as vast as vast as space and as infinity.
- There is known to man. It is known to man. It is a dimension as space and as infinity.

As you can see, Markov chains have a tendency to make sentences which almost make sense. This is because every individual pairing of two words will make sense, but the combinations of the pairings might not. For example, "as vast" and "vast as" can both make sense given the right context, but "vast as vast as vast as vast" is nonsense. We can help alleviate this problem by taking into account the last 2 (or 3, or 4...) words when choosing the next word instead of just the last one, but this requires a far larger input or it will result in the output being the same as the input.

3 Random Numbers

You may find the module `random` to be especially useful when creating Markov chains. Below is a brief example of some common functions inside `random`.

```
>>> import random
>>> print(random.random())
0.7682548548225483
>>> print(random.random())
0.4165356512641182
>>> print(random.uniform(1, 100))
36.30623079969581
>>> print(random.uniform(1, 100))
97.48359345305
```

```
>>> print(random.choice(['joe', 'moe', 'larry', 'shemp', 'curly']))
larry
>>> print(random.choice(['joe', 'moe', 'larry', 'shemp', 'curly']))
shemp
>>> print(random.randint(1, 100))
79
>>> print(random.randint(1, 100))
12
```

Here's a summary of these functions.

Take a look at the documentation for `random` if you would like a random number not based on a uniform distribution:

`https://docs.python.org/3.4/library/random.html`

4 Using Objects: Creating Turtles

We have been implicit using two important definitions. These are part of object oriented programming, which will be covered in more detail in later labs.

Function	Arguments	Purpose
`random.random()`		Returns a random number between 0.0 and 1.0, including 0.0 but not 1.0.
`random.uniform()`	`a, b`	Returns a random number between `a` and `b` inclusive. Each real number between `a` and `b` has an equal probability of occurring.
`random.randint()`	`a, b`	Returns a random integer between `a` and `b` inclusive. Each integer between `a` and `b` has an equal probability of occuring.
`random.choice()`	`list`	Returns a random value from `list`.

Table 1: Summary of `random` functions.

Object is a Python data type that can contain arbitrary amounts and kinds of data. They can be instantiated or created in several ways. Some examples:

`int`, `str`, `list` Built-in types are objects and are created by writing literals such as `1`, `"Hello"`, `[1,2,3]` or by conversion `list("123")`.

`file` Files are also objects. They are created using the `open()` function.

`Exception` Exceptions are objects created when they are `raise`d.

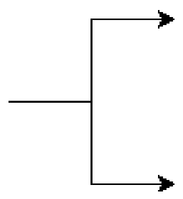
`Turtle` These are created by calling `turtle.Turtle()`.

Method is a function that is specific a certain object. Some examples include `str.find()`, `list.reverse()`, `file.write()`, `Turtle.forward()`, `Exception.err()`.

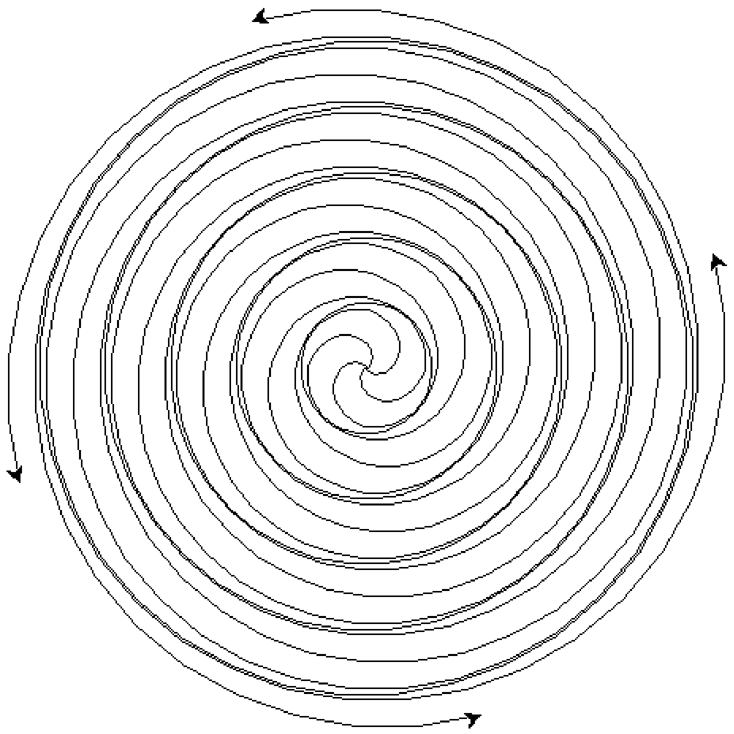
Knowing this, we can create multiple turtles and draw more complex designs. This is done by instantiating new turtles with the `turtle.Turtle()` function. This function returns a turtle object, which we can call every other normal turtle method on.

```python
import turtle

first = turtle.Turtle()
second = turtle.Turtle()

first.forward(50)
second.forward(50)
first.left(90)
second.right(90)
first.forward(50)
second.forward(50)
first.right(90)
second.left(90)
first.forward(50)
second.forward(50)
```

If we add a group of turtles to a list, we can easily apply the same commands to all of them, as in this example:



```python
import turtle

turtles = []
first = turtle.Turtle()
first.speed(0)
turtles.append(first)

second = turtle.Turtle()
second.speed(0)
second.right(90)
turtles.append(second)

third = turtle.Turtle()
third.speed(0)
third.right(180)
turtles.append(third)

fourth = turtle.Turtle()
fourth.speed(0)
fourth.right(270)
turtles.append(fourth)

for i in range(200):
    for turt in turtles:
        turt.forward(i/5)
        turt.left(10)
```



5 Exercises

Exercise 5.1 (markov.py).
Write a program that takes in a filename, reads each line of the file, converts the lines into a format convenient for making Markov chains, and then prints out a new sentence randomly generated from the data, based on the Markov algorithm.

You may use any file you wish for test input, though `alice.txt` and `cthulhu.txt` are provided on Canvas. `alice.txt` is a slightly edited version of the complete text of Lewis Carroll's *Alice's Adventures in Wonderland*, taken from `https://www.gutenberg.org/files/11/11-h/11-h.htm`. `cthulhu.txt` is a slightly edited version of the complete text of H.P. Lovecrafts's *The Call of Cthulhu*, taken from `http://www.hplovecraft.com/writings/texts/fiction/cc.aspx`.

When parsing the input file, you should ignore any blank lines.

Treat the lines in the file as separate sentences. That is, if "Hello, how are you?" and "Where are my keys?" are lines in a file, then "you?" should not be followed by "Where" when generating a chain. However, "are" should be allowed to be followed by either "you?" or "my", as seen in Figure 1.

When creating a new chain, the first element should always be a randomly selected first word of a line in the file. The chain should end when either:

- There are no valid choices to continue the sentence with.
- The sentence has reached a length of 100 words.

Remember that you need to account for different frequencies of possible follow words. That is, a situation like in Figure 2 where the word "a" is followed by two "is" and one "known". You need to account for whatever word frequencies might come up within your input file.

Hint

You may find dictionaries to be useful in implementing Markov chains. For example, the Markov chain in Figure 1 could be represented using this dictionary:

```python
{'Hello,': ['how'], 'how': ['are'],
 'Where': ['are'], 'are': ['you', 'my'],
 'you?': [], 'my': ['keys'], 'keys?': []}
```

To generate a Markov chain, pick a random word from a list of first words. Then pick a random word that comes after the first word. Keep picking random next words until no more words are left or until the length limit is reached.

Exercise 5.2 (design, piglatin.py).
Before attempting to code this problem, create a file `design` that contains some analysis of how you think the problem will be solved. Examples include but are not limited to: a flowchart of events in the program, pseudocode, or a step-by-step process written in plain English. If you choose to scan your design, please make sure that it is legible.

Write a program that translates a file from English to pig latin. The rules for pig latin are as follows:

For a word that begins with consonants, the initial consonant or consonant cluster is moved to the end of the word and "ay" is added as a suffix:

- "happy" → "appyhay"
- "glove" → "oveglay"

For words that begin with vowels, you add "way" to the end of the word:

- "egg" → "eggway"
- "inbox" → "inboxway"

For your program, you *must* write a function that takes in one individual word and returns the translation to pig latin. Write another function that takes a string, which may be sentences (may contain the characters "a-zA-Z,.-;!?()" and space), and returns the translation of the sentence to pig latin. Strip out any punctuation. For example, "Hello, how are you?" would translate into "elloHay owhay areway ouyay".

The user must be able to specify the filename for the file to be translated and the filename that the program should write to. For example:

```
$ cat test.txt
Hello, how are you?
$ python3 piglatin.py
Enter English filename >>> test.txt
Enter filename to write to >>> test_piglatin.txt
Done.
$ cat test_piglatin.txt
elloHay owhay areway ouyay
```

Exercise 5.3 (navigate3.py).
Modify navigate.py so that, rather than take instructions from the command line, it reads from a file (specified by user input) to determine what the turtle will do. Additionally, you will be adding the "split" command. This command will use instantiation of new turtles in order to draw multiple lines at once. Every new command will apply to every turtle that currently exists. The file will have on instruction per line. The possible instructions are:

forward X Move all turtles forward X.

left X Turn all turtles X degrees to the left.

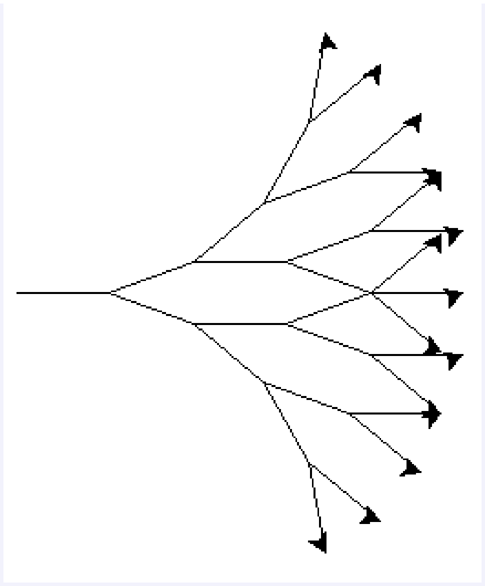
right X Turn all turtles X degrees to the right.

split X Create a new copy of every turtle Each new turtle will be turned X degrees to the right.

Sample input file:

```
forward 50
left 20
split 40
forward 50
left 20
split 40
forward 50
left 20
split 40
forward 50
left 20
split 40
forward 50
left 20
```

Sample output:



Suggestions:

- Write a function for each command.
- Keep a list of turtles. To store a new turtle in a variable named `t`, write `t = Turtle()`.
- Use `t.clone` to clone the turtle `t` when processing the split command. Add this new turtle to your list of turtles.

Exercise 5.4 (README).
For the following labs you will be required to create aand submit a README file.

A README file is a simple text file that contains basic information regarding the purpose and use of the software program, utility, or game. README files often contain instructions or additional help regarding the software it accompanies.

For this lab you must have the following in your README file::

- **Purpose**: Describes what each program does and what problem it solves. You can keep this breif.
- **Conclusion**:
  - What you learned during the lab? What new aspect of programming did you learn from the lab? Be analytical about what you learned.
  - Did pair programming help in solving the problems and completing the prelab? Did you have problems with your budy?
  - Did you work with your budy on the lab? What sections did you discuss? Did you and your budy carry out a review session with each others' code?
  - Did you encounter any problems? How did you fix those problems?
  - What improvements could you make?

The conclusion does not have to be lengthy, but it should be thorough. You will include the README file with your submissions.

Submitting

You should submit your code as a `.zip` that contains all the exercise files for this lab. The submitted file should be named

`cse107_firstname_lastname_lab10.zip`

Upload your .zip file to Canvas.

List of Files to Submit

Exercises start on page 6.