Project Assignment 4: Markov Chain

Overview

This assignment is designed to further your experience and understanding of functional programming in Python. In addition, it will teach you a new algorithm that is entertaining in what it produces. We are often faced with writing programs that must consume text, transform that text, and generate textual output. The problem we are asking you to solve in this project assignment is exactly this type of program, albeit a bit unusual. In particular, you will complete a program that generates random English text from books that are stored in files.

In the wonderful book *The Practice of Programming* (POP), by Brian Kernighan and Rob Pike, the authors describe an algorithm known as the *Markov chain algorithm*. Before you begin this assignment, read chapter three (link on Moodle) of that book to gain an understanding of the algorithm. The algorithm is also explained below and in other sources you may discover. The excerpt from the book describes two implementations in C and in Java. This treatment of the Markov chain problem is useful, even though we will be implementing a slightly different version in Python. Note that this reading is mainly to provide background information.

Learning Goals

- To read and understand existing problem documentation.
- Be able to use search engines and discover external resources to help solve problems.
- To write code that conforms to a specification.
- Gain experience developing code incrementally, testing the individual functions and then testing them in integration.
- Gain experience writing tests that help develop code but are not explicitly provided.
- Understand what the goal of this software is, and that testing it on data that it is ultimately designed to process is a required step in the development process.
- To understand the Markov Chain algorithm.
- To understand and implement functional programming to solve a problem.
- To work with stream objects as iterators in higher order functions.

General Information

Read this entire document. If, after a careful reading, something seems ambiguous or unclear to you, then communicate to the course staff immediately. Start this assignment as soon as possible. Do not wait until the night the assignment is due to tell us you don't understand something, as our ability to help you will be minimal.

Reminder: Copying partial or whole solutions, obtained from other students or elsewhere, is academic dishonesty. Do not share your code with your classmates, and do not use your classmates' code. If you are confused about what constitutes academic dishonesty you should re-read the course syllabus and policies. We assume you have read the course information in detail

and by submitting this assignment you have provided your virtual signature in agreement with these policies.

Grading: You are responsible for submitting project assignments that compile and are configured correctly. If your project submission does not follow these policies exactly you may receive a grade of zero for this assignment. Code that does not run will receive a score of zero points.

Policies:

- For all assignments, you must use the script provided to zip up your code. Your code file, class and method names must match exactly to the specification we provide. Refer to our coding style guide for naming and style conventions.
- The course staff are here to help you figure out errors (not solve them for you), BEFORE you submit your solution. When you submit your solution, be sure to remove all compilation errors from your project. Any compilation errors in your project will cause the auto-grader to fail your assignment, and you will receive a zero for your submission. No Exceptions!

Part 1: Getting Oriented

The Markov Chain Building algorithm.

The Markov chain algorithm has two phases: build and generate. In the build phase, you create a transition table for an order-2 Markov Chain. The table, or chain, consists of mappings from prefixes to suffixes. Prefixes are a tuple of words that represent the current 'state'. A suffix is a tuple of words which represent possible states that one could transition to. An order-2 chain means that we will use two words for our prefix. We will not be concerned with expanding this to accommodate other orders, but you can imagine how that might be done.

Consider the following table of prefix and suffix pairs generated by the following text (Note that the use of [] and {} does not indicate Python data structures- it is purely for notation).

This is a nice sentence. This is another sentence. This is another sentence.

['\n','\n'] = {'This' : 1}

['\n', 'This'] = {'is' : 1}

['This', 'is'] = {'a' : 1, 'another' : 2}

['is', 'a'] = {'nice' : 1}

['a', 'nice'] = {'sentence.' : 1}

['nice', 'sentence.'] = {'This' : 1}

['sentence.', 'This'] = {'is' : 2}

['is', 'another'] = {'sentence.' : 2}

['another', 'sentence.'] = {'This' : 1, '\n' : 1}

The process starts with a prefix of two "sentinel" characters. In this case, we are using newline characters as "blank" characters for the first two prefixes to start the building process. We also use the newline character as a sentinel in the sense that this character will be used to stop the generation process. Although it is not technically a "word", it will be used in prefixes and suffixes, so we'll define it as "NONWORD". We will be processing text files on a line by line basis. The newline character at the end of each line will be ignored. NONWORD will be added to the chain as the suffix to the last prefix; after the last line has been read and processed.

After the start prefix has been processed, for each following prefix, the word following the last prefix word is entered into the suffix associated with that prefix. A new prefix is created by "shifting" the suffix word into the new prefix, so that the first word in the new prefix is the second word of the old prefix, etc. Notice that a suffix is represented by a bag of words, where each word is a key mapped to a frequency count. That way, each unique word appears once in any suffix. We do not care about punctuation, so that 'sentence' is considered a different word that 'sentence.'.

Generating Random Text using the Transition Table

Once your transition table is complete, you can use it to generate random text. You generate text (words) by choosing a prefix, then uniformly randomly selecting a word in its suffix. This word goes on the list of words you are generating. It is also 'shifted' into the prefix for the next lookup. The number of words to generate, num_words, is given to the program.

Here is an example using the order-2 Transition Table described above.

> Start by selecting the prefix ['\n','\n'] in the map. That maps to the suffix containing ['This']. The only choice here is to select the word 'This' for the output string, then shift it into the prefix to get: ['\n', 'This'].
> Next, look up the prefix ['\n', 'This'] in the map. This maps to the suffix ['is'].
> You have only one choice, to take the word 'is',
> Shifting into the prefix we get ['This', 'is'], which maps to ['a', 'another', 'another'].
> This is where the randomization comes in. Since there are 3 words in the suffix, 1/3 of the time you will select the word 'a' and 2/3 of the time you will select 'another'. Let's imagine that in this example you happened to choose 'another'.
> The next prefix would be: [is, another].

Looking up [is, another] we get [‘sentence.’, ‘sentence.’].
No choice here, choose ‘sentence’ and then consider the prefix [another, sentence.].
Looking up [another, sentence.] obtains [‘This’] 50% of the time and [‘\n’] 50% of the time.
The process stops if we happened to choose ‘\n’ (which in our program is NONWORD).
If ‘This’ was chosen, then we will next consider [‘sentence.’,’This’].
The process continues until the number of words generated equals num_words, or a suffix word ‘\n’, or NONWORD is chosen.

Part 2: Project Requirements and Structure

After you download and unzip the starter kit you should take some time reviewing the provided code and comments.

The files that comprise this project are:

- markov_main.py - (provided) an example of how to run the code.
- markov.py - (provided) an example of how to run the code from the cmd line.
- prefix.py –the prefix interface using 2 words as a tuple.
- suffix.py – the suffix interface- uses an ImmDict to handle word-frequency pairs.
- immdict.py – a class that provides an immutable dictionary to store the chain. Wraps a Python dictionary.
- builder.py – given a text file, builds the Markov Chain
- generator.py – given a Markov Chain, a randomizer function, the number of words to generate, and NONWORD, generates a string of words from the chain.

In addition to the two source files, markov.py and markov_main.py, we provide a set of texts for you to analyze in the ‘books’ directory.

Your submission files and all functions must be named exactly as above to receive credit for this assignment. The following section provides a description of the modules and functions that you must define and implement in each file. Remember that you must define these signatures exactly to receive any credit for this assignment.

The only looping constructs allowed, `for` or `while` loop, are in the *`line_gen`* and *`pairs_gen`* functions.

Module Interfaces

module prefix: Supports an abstraction of a prefix as a tuple of two strings.

new_prefix
Takes two parameters- the first and second word for the prefix.
Returns a tuple: (first, second)

shift_in
Takes a tuple and a word as parameters, returns a new tuple with word in second and second in first. For example, if the prefix passed in was: ('the' 'big') and the word passed in was 'dog', this function would return this tuple: ('big' 'dog').

---

module suffix: Functions that provide an interface to an Immutable dictionary (class ImmDict)
imports *ImmDict*

empty_suffix
Returns an instance of the ImmDict class- an empty immutable dictionary.

add_word
Takes a suffix and a word as parameters, returns a new ImmDict instance with word added to its list. The function adds the word as a key and increments that key's frequency count if it exists in the suffix. If not, it enters the word with a frequency of 1.

choose_word
Takes three parameters: a chain (an ImmDict), a prefix, and a randomizer. It finds the suffix (also an ImmDict) associated with the prefix in the chain. It then randomly chooses a word in the suffix and returns it. Note that your function must choose with uniform probability. For example, if there are three words in the suffix with these frequencies:

'also':3, 'been':5, 'there':1

Your algorithm must choose 'also' approximately 33% of the time, 'been' about 56% of the time, and 'there' about 11% of the time. You will want to write helper functions to implement this feature. Remember that no *for* or *while* loop code is allowed, so write recursive functions and use HOFs (listcomps OK) if you need them.

---

module immdict

class ImmDict: A class representing an immutable dictionary (ImmDict).
This class wraps a Python dictionary to provide an immutable version. This class is used to store the markov chain as prefix-suffix pairs. The keys are the prefix tuples. The value mapped to a key is a suffix, which is also represented by an immutable dictionary (ImmDict).

A constructor that initializes the class to the empty dictionary: a Python dictionary {}.

put
Takes a key, and a value, as parameters. Returns a new ImmDict instance with the parameters added to its dictionary. Note: you want to copy the existing dictionary, add the k,v args, and then create the new instance.

get
Takes a key parameter and returns the value mapped to that key. Returns None if the key does not exist.

keys
No parameters. Returns a (copy) of the list of keys in the dictionary.

values
No parameters. Returns a (copy) of the list of keys in the dictionary.

---

module builder: The functions in this module build the Markov chain. A text file is passed in as well as a randomizer (see the markov and markov_main modules). A Markov chain is returned as an ImmDict instance.
Imports *prefix* and *suffix*.

NONWORD
A variable defined as '\n'.

build
Takes a file name as a parameter. Returns an ImmDict instance- the chain. Calls *pairs_gen* and passes that to *build_chain*.

build_chain

Takes three parameters: a function to add a prefix, word pair (a tuple), to an ImmDict object (see spec for add_to_chain), a generator object (an iterator) that produces prefix, word pairs as tuples (supplied by *pairs_gen*), and an empty ImmDict object. The function returns an ImmDict object containing the complete Markov chain. Hint: use reduce here.

add_to_chain

Takes two parameters: the chain (an ImmDict object), and a prefix, which is a word pair (a tuple). It returns a new ImmDict with the pair added to it. Calls the *suffix* function *add_to_chain*.

line_gen

Takes a file name as a parameter. Returns a generator object that produces lines from the text file upon demand. A loop is allowed here as it is fully encapsulated.

pairs_gen

Takes a file name and a line generating function (such as *line_gen*) as parameters. It calls the line generating function to get the lines of text. It returns a generator that produces pairs of (prefix, word) as tuples on demand. At the start, it creates the starting prefix (contains NONWORD , NONWORD) and pairs it with the first word in the first line. It then shifts that word in to the next prefix, and so on. In processing the lines of text, it must keep track of the last two words in a line as that will be the 'sliding' prefix for the next line. Note that the last tuple returned will include the last prefix and NONWORD. A loop is allowed here as it is fully encapsulated.

---

module generator: The functions in this module generate a string of words chosen from the Markov chain built by the builder module. A Markov chain (an ImmDict) and the number of words to generate are the inputs to this module. It outputs the words as a string. Imports *prefix* and *suffix*.

get_word_list

Takes a Markov chain, a prefix, a randomizer function, an int- the number of words to generate, and NONWORD as parameters. Calls the *suffix* function *choose_word*. It returns a tuple of words with length equal to the number of words passed in- unless the NONWORD sentinel was encountered first. Note that this function may not be written with a `for` or `while` loop.

generate

Takes a Markov chain, a randomizer function, an int- the number of words to generate, and the NONWORD as parameters. It returns a string containing the words returned from *get_word_list*.

Diagrams

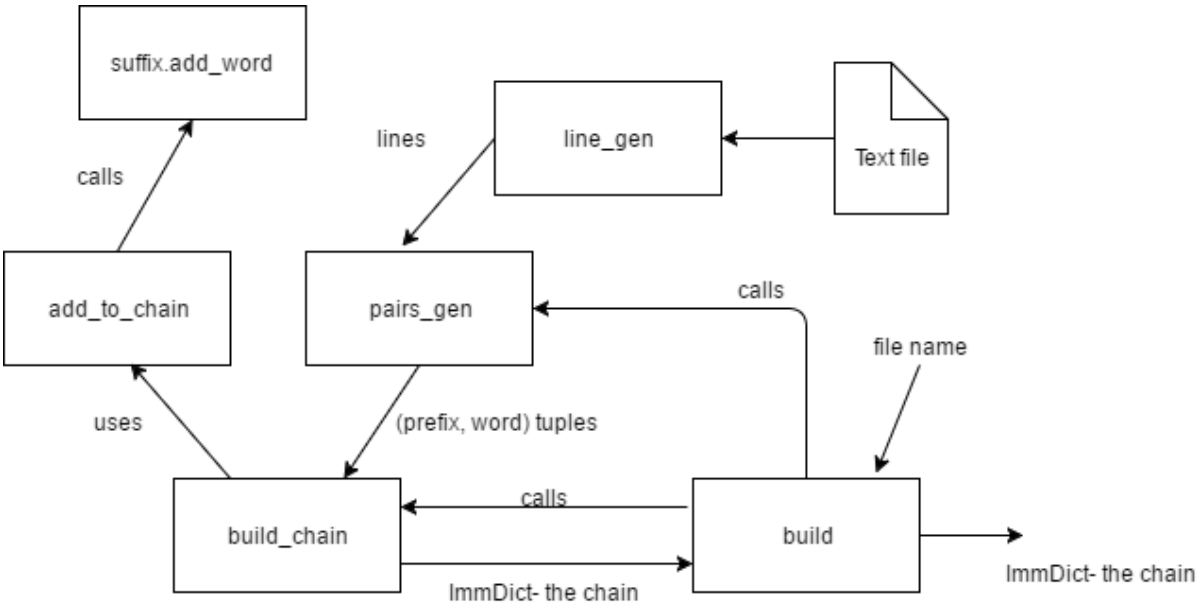

Figure 1: The Builder Module

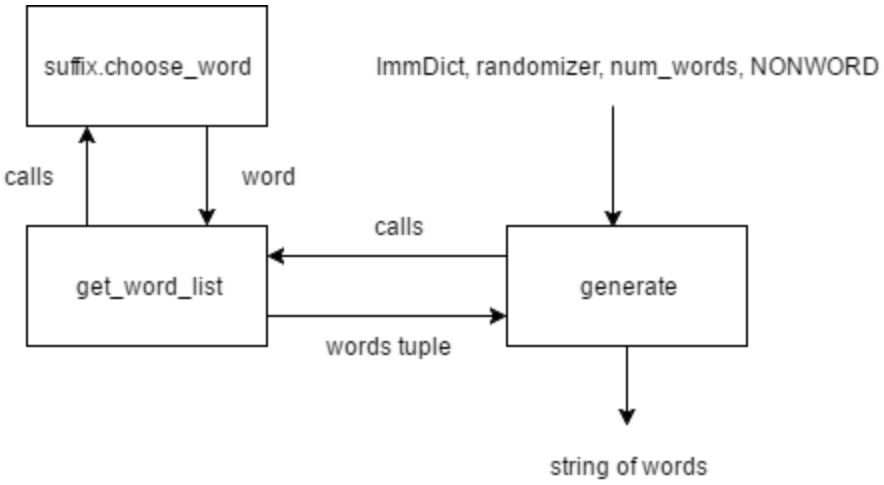

Figure 2: The Generator Module

In Figure 1, the *pairs_gen* takes a line of text, supplied by *line_gen*, as input and outputs a generator that produces a (prefix, word) tuples at each iteration. This function must maintain a 'sliding' window during the reading of all of the lines of the text file, so it needs to save the current prefix from one line to another. The current prefix begins as the empty prefix, (NONWORD, NONWORD). At the end of one line the prefix will be the last two words in that line. It will be applied to the first word of the next line. When the last line has been processed, the remaining prefix will be paired with the sentinel NONWORD.

Generation involves moving through the chain in the way previously described. The word in a suffix is chosen by calling the *suffix* method *choose_word*. The *suffix* module defines all of the functions required to carry out the uniform random selection as described above. Words are chosen until the specified word count has been reached, or NONWORD is chosen. This effectively ends the generation.

Part 3: Testing

Develop your code incrementally, testing as you make progress. Don't write a lot of code and then test it. Remember that we will not assess code that does not run at this point.

Functions that are self-contained are easier to test that code with side effects. Write a simple version of a function and then write a test for it. Simulate the inputs with something really simple. For example, if one of the inputs is a stream (generator), simulate that with a list of a few of the types of elements that the real stream would generate. Remember that the higher order built in functions are looking for an iterator, so you can test with a list.

Write helper functions if you need to. Test them in isolation. No matter how simple, test it before you rely on it! Note that this takes time, but if you want full credit and a piece of working software, this is the best way to achieve that end.

When you are confident a function works, test it in integration with other functions it is called by or that it calls.

When you are ready to read text from files, try text files with these sequences:

‘a’, ‘a’, ‘b’ and ‘a’, ‘a’, ‘a’, ‘b’

For a second order Markov chain trained on the sequence ‘a’, ‘a’, ‘a’, ‘b’ you would generate the map:

Prefix Suffix

[‘\n‘, ‘\n‘] = {‘a’:1}

[‘\n‘, ‘a’] = {‘a’:1}

[‘a’, ‘a’] = {‘a’:1, ‘b’:1}

[‘a’, ‘b’] = {‘\n‘}

The ‘\n’ character is defined as NONWORD in the `builder` module. Showing ‘\n’ here for brevity.

This ensures you are checking the start and stop conditions and are making the chain properly. Once you are building the correct structure, try a text file with the three lines from the example used in part 2. Then, make up a larger text file and try that. Finally, try the files in the ‘books’ directory. (Note that the use of [] and {} does not indicate Python data structures- it is purely for notation). This code may take several minutes to complete on the larger files. Let the code run until it completes. Make sure it does complete.

Submission

Run the script to ‘zip’ up your files and submit them to the Moodle assignment before the due date/time.