Text Analysis and Generation

Contents

- 12.1. Unique words
- 12.2. Punctuation
- 12.3. Word frequencies
- 12.4. Optional parameters
- 12.5. Dictionary subtraction
- 12.6. Random numbers
- 12.7. Bigrams
- 12.8. Markov analysis
- 12.9. Generating text
- 12.10. Debugging
- 12.11. Glossary
- 12.12. Exercises

You can order print and ebook versions of *Think Python 3e* from Bookshop.org and Amazon.

At this point we have covered Python's core data structures – lists, dictionaries, and tuples – and some algorithms that use them. In this chapter, we'll use them to explore text analysis and Markov generation:

- Text analysis is a way to describe the statistical relationships between the words in a document, like the probability that one word is followed by another, and
- Markov generation is a way to generate new text with words and phrases similar to the original text.

These algorithms are similar to parts of a Large Language Model (LLM), which is the key component of a chatbot.

We'll start by counting the number of times each word appears in a book. Then we'll look at pairs of words, and make a list of the words that can follow each word. We'll make a simple version of a Markov generator, and as an exercise, you'll have a chance to make a more general version.

12.1. Unique words

As a first step toward text analysis, let's read a book – *The Strange Case Of Dr. Jekyll And Mr. Hyde* by Robert Louis Stevenson – and count the number of unique words. Instructions for downloading the book are in the notebook for this chapter.

```
filename = 'dr_jekyll.txt'
```

We'll use a `for` loop to read lines from the file and `split` to divide the lines into words. Then, to keep track of unique words, we'll store each word as a key in a dictionary.

```
unique_words = {}
for line in open(filename):
    seq = line.split()
    for word in seq:
        unique_words[word] = 1

len(unique_words)
```

```
6040
```

The length of the dictionary is the number of unique words – about `6000` by this way of counting. But if we inspect them, we'll see that some are not valid words.

For example, let's look at the longest words in `unique_words`. We can use `sorted` to sort the words, passing the `len` function as a keyword argument so the words are sorted by length.

```
sorted(unique_words, key=len)[-5:]
```

```
['chocolate-coloured',
 'superiors—behold!"',
 'coolness—frightened',
 'gentleman—something',
 'pocket-handkerchief.']
```

The slice index, `[-5:]`, selects the last `5` elements of the sorted list, which are the longest words.

The list includes some legitimately long words, like "circumscription", and some hyphenated words, like "chocolate-coloured". But some of the longest "words" are actually two words separated by a dash. And other words include punctuation like periods, exclamation points, and quotation marks.

So, before we move on, let's deal with dashes and other punctuation.

12.2. Punctuation

To identify the words in the text, we need to deal with two issues:

- When a dash appears in a line, we should replace it with a space – then when we use `split`, the words will be separated.
- After splitting the words, we can use `strip` to remove punctuation.

To handle the first issue, we can use the following function, which takes a string, replaces dashes with spaces, splits the string, and returns the resulting list.

```
def split_line(line):
    return line.replace('—', ' ').split()
```

Notice that `split_line` only replaces dashes, not hyphens. Here's an example.

```
split_line('coolness—frightened')
```

```
['coolness', 'frightened']
```

Now, to remove punctuation from the beginning and end of each word, we can use `strip`, but we need a list of characters that are considered punctuation.

Characters in Python strings are in Unicode, which is an international standard used to represent letters in nearly every alphabet, numbers, symbols, punctuation marks, and more. The `unicodedata` module provides a `category` function we can use to tell which characters are punctuation. Given a letter, it returns a string with information about what category the letter is in.

```
import unicodedata

unicodedata.category('A')
```

```
'Lu'
```

The category string of `'A'` is `'Lu'` – the `'L'` means it is a letter and the `'u'` means it is uppercase.

The category string of `'.'` is `'Po'` – the `'P'` means it is punctuation and the `'o'` means its subcategory is "other".

```
unicodedata.category('.')
```

```
'Po'
```

We can find the punctuation marks in the book by checking for characters with categories that begin with `'P'`. The following loop stores the unique punctuation marks in a dictionary.

```
punc_marks = {}
for line in open(filename):
    for char in line:
        category = unicodedata.category(char)
        if category.startswith('P'):
            punc_marks[char] = 1
```

To make a list of punctuation marks, we can join the keys of the dictionary into a string.

```
punctuation = ''.join(punc_marks)
print(punctuation)
```

```
.';,-“”:?—‘!()_
```

Now that we know which characters in the book are punctuation, we can write a function that takes a word, strips punctuation from the beginning and end, and converts it to lower case.

```
def clean_word(word):
    return word.strip(punctuation).lower()
```

Here’s an example.

```
clean_word('“Behold!”')
```

```
'behold'
```

Because `strip` removes characters from the beginning and end, it leaves hyphenated words alone.

```
clean_word('pocket-handkerchief')
```

```
'pocket-handkerchief'
```

Now here’s a loop that uses `split_line` and `clean_word` to identify the unique words in the book.

```
unique_words2 = {}
for line in open(filename):
    for word in split_line(line):
        word = clean_word(word)
        unique_words2[word] = 1

len(unique_words2)
```

```
4005
```

With this stricter definition of what a word is, there are about 4000 unique words. And we can confirm that the list of longest words has been cleaned up.

```
sorted(unique_words2, key=len)[-5:]
```

```
['circumscription',
 'unimpressionable',
 'fellow-creatures',
 'chocolate-coloured',
 'pocket-handkerchief']
```

Now let's see how many times each word is used.

12.3. Word frequencies

The following loop computes the frequency of each unique word.

```
word_counter = {}
for line in open(filename):
    for word in split_line(line):
        word = clean_word(word)
        if word not in word_counter:
            word_counter[word] = 1
        else:
            word_counter[word] += 1
```

The first time we see a word, we initialize its frequency to `1`. If we see the same word again later, we increment its frequency.

To see which words appear most often, we can use `items` to get the key-value pairs from `word_counter`, and sort them by the second element of the pair, which is the frequency. First we'll define a function that selects the second element.

```
def second_element(t):
    return t[1]
```

Now we can use `sorted` with two keyword arguments:

- `key=second_element` means the items will be sorted according to the frequencies of the words.
- `reverse=True` means the items will be sorted in reverse order, with the most frequent words first.

```
items = sorted(word_counter.items(), key=second_element, reverse=True)
```

Here are the five most frequent words.

```
for word, freq in items[:5]:
    print(freq, word, sep='\t')
```

```
1614	the
972	and
941	of
640	to
640	i
```

In the next section, we'll encapsulate this loop in a function. And we'll use it to demonstrate a new feature – optional parameters.

12.4. Optional parameters

We've used built-in functions that take optional parameters. For example, `round` takes an optional parameters called `ndigits` that indicates how many decimal places to keep.

```
round(3.141592653589793, ndigits=3)
```

```
3.142
```

But it's not just built-in functions – we can write functions with optional parameters, too. For example, the following function takes two parameters, `word_counter` and `num`.

```
def print_most_common(word_counter, num=5):
    items = sorted(word_counter.items(), key=second_element, reverse=True)

    for word, freq in items[:num]:
        print(freq, word, sep='\t')
```

The second parameter looks like an assignment statement, but it's not – it's an optional parameter.

If you call this function with one argument, `num` gets the **default value**, which is `5`.

```
print_most_common(word_counter)
```

```
1614	the
972	and
941	of
640	to
640	i
```

If you call this function with two arguments, the second argument gets assigned to `num` instead of the default value.

```
print_most_common(word_counter, 3)
```

```
1614    the
972     and
941     of
```

In that case, we would say the optional argument **overrides** the default value.

If a function has both required and optional parameters, all of the required parameters have to come first, followed by the optional ones.

12.5. Dictionary subtraction

Suppose we want to spell-check a book – that is, find a list of words that might be misspelled. One way to do that is to find words in the book that don't appear in a list of valid words. In previous chapters, we've used a list of words that are considered valid in word games like Scrabble. Now we'll use this list to spell-check Robert Louis Stevenson.

We can think of this problem as set subtraction – that is, we want to find all the words from one set (the words in the book) that are not in the other (the words in the list).

As we've done before, we can read the contents of `words.txt` and split it into a list of strings.

```
word_list = open('words.txt').read().split()
```

Then we'll store the words as keys in a dictionary so we can use the `in` operator to check quickly whether a word is valid.

```
valid_words = {}
for word in word_list:
    valid_words[word] = 1
```

Now, to identify words that appear in the book but not in the word list, we'll use `subtract`, which takes two dictionaries as parameters and returns a new dictionary that contains all the keys from one that are not in the other.

```
def subtract(d1, d2):
    res = {}
    for key in d1:
        if key not in d2:
            res[key] = d1[key]
    return res
```

Here's how we use it.

```
diff = subtract(word_counter, valid_words)
```

To get a sample of words that might be misspelled, we can print the most common words in `diff`.

```
print_most_common(diff)
```

```
640	i
628	a
128	utterson
124	mr
98	hyde
```

The most common "misspelled" words are mostly names and a few single-letter words (Mr. Utterson is Dr. Jekyll's friend and lawyer).

If we select words that only appear once, they are more likely to be actual misspellings. We can do that by looping through the items and making a list of words with frequency `1`.

```
singletons = []
for word, freq in diff.items():
    if freq == 1:
        singletons.append(word)
```

Here are the last few elements of the list.

```
singletons[-5:]
```

```
['gesticulated', 'abjection', 'circumscription', 'reindue', 'fearstruck']
```

Most of them are valid words that are not in the word list. But `'reindue'` appears to be a misspelling of `'reinduce'`, so at least we found one legitimate error.

12.6. Random numbers

As a step toward Markov text generation, next we'll choose a random sequence of words from `word_counter`. But first let's talk about randomness.

Given the same inputs, most computer programs are **deterministic**, which means they generate the same outputs every time. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are one example, but there are more.

Making a program truly nondeterministic turns out to be difficult, but there are ways to fake it. One is to use algorithms that generate **pseudorandom** numbers. Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.

The `random` module provides functions that generate pseudorandom numbers – which I will simply call "random" from here on. We can import it like this.

```
import random
```

The `random` module provides a function called `choice` that chooses an element from a list at random, with every element having the same probability of being chosen.

```
t = [1, 2, 3]
random.choice(t)
```

```
1
```

If you call the function again, you might get the same element again, or a different one.

```
random.choice(t)
```

```
2
```

In the long run, we expect to get every element about the same number of times.

If you use `choice` with a dictionary, you get a `KeyError`.

```
random.choice(word_counter)
```

```
KeyError: 422
```

To choose a random key, you have to put the keys in a list and then call `choice`.

```
words = list(word_counter)
random.choice(words)
```

```
'posture'
```

If we generate a random sequence of words, it doesn't make much sense.

```
for i in range(6):
    word = random.choice(words)
    print(word, end=' ')
```

```
ill-contained written apocryphal nor busy spoke
```

Part of the problem is that we are not taking into account that some words are more common than others. The results will be better if we choose words with different "weights", so that some are chosen more often than others.

If we use the values from `word_counter` as weights, each word is chosen with a probability that depends on its frequency.

```
weights = word_counter.values()
```

The `random` module provides another function called `choices` that takes weights as an optional argument.

```
random.choices(words, weights=weights)
```

```
['than']
```

And it takes another optional argument, `k`, that specifies the number of words to select.

```
random_words = random.choices(words, weights=weights, k=6)
random_words
```

```
['reach', 'streets', 'edward', 'a', 'said', 'to']
```

The result is a list of strings that we can join into something that's looks more like a sentence.

```
' '.join(random_words)
```

```
'reach streets edward a said to'
```

If you choose words from the book at random, you get a sense of the vocabulary, but a series of random words seldom makes sense because there is no relationship between successive words. For example, in a real sentence you expect an article like "the" to be followed by an adjective or a noun, and probably not a verb or adverb. So the next step is to look at these relationships between words.

12.7. Bigrams

Instead of looking at one word at a time, now we'll look at sequences of two words, which are called **bigrams**. A sequence of three words is called a **trigram**, and a sequence with some unspecified number of words is called an **n-gram**.

Let's write a program that finds all of the bigrams in the book and the number of times each one appears. To store the results, we'll use a dictionary where

- The keys are tuples of strings that represent bigrams, and
- The values are integers that represent frequencies.

Let's call it `bigram_counter`.

```
bigram_counter = {}
```

The following function takes a list of two strings as a parameter. First it makes a tuple of the two strings, which can be used as a key in a dictionary. Then it adds the key to `bigram_counter`, if it doesn't exist, or increments the frequency if it does.

```
def count_bigram(bigram):
    key = tuple(bigram)
    if key not in bigram_counter:
        bigram_counter[key] = 1
    else:
        bigram_counter[key] += 1
```

As we go through the book, we have to keep track of each pair of consecutive words. So if we see the sequence "man is not truly one", we would add the bigrams "man is", "is not", "not truly", and so on.

To keep track of these bigrams, we'll use a list called `window`, because it is like a window that slides over the pages of the book, showing only two words at a time. Initially, `window` is empty.

```
window = []
```

We'll use the following function to process the words one at a time.

```
def process_word(word):
    window.append(word)

    if len(window) == 2:
        count_bigram(window)
        window.pop(0)
```

The first time this function is called, it appends the given word to `window`. Since there is only one word in the window, we don't have a bigram yet, so the function ends.

The second time it's called – and every time thereafter – it appends a second word to `window`. Since there are two words in the window, it calls `count_bigram` to keep track of how many times each bigram appears. Then it uses `pop` to remove the first word from the window.

The following program loops through the words in the book and processes them one at a time.

```
for line in open(filename):
    for word in split_line(line):
        word = clean_word(word)
        process_word(word)
```

The result is a dictionary that maps from each bigram to the number of times it appears. We can use `print_most_common` to see the most common bigrams.

```
print_most_common(bigram_counter)
```

```
178     ('of', 'the')
139     ('in', 'the')
94      ('it', 'was')
80      ('and', 'the')
73      ('to', 'the')
```

Looking at these results, we can get a sense of which pairs of words are most likely to appear together. We can also use the results to generate random text, like this.

```
bigrams = list(bigram_counter)
weights = bigram_counter.values()
random_bigrams = random.choices(bigrams, weights=weights, k=6)
```

`bigrams` is a list of the bigrams that appear in the books. `weights` is a list of their frequencies, so `random_bigrams` is a sample where the probability a bigram is selected is proportional to its frequency.

Here are the results.

```
for pair in random_bigrams:
    print(' '.join(pair), end=' ')
```

```
to suggest this preface to detain fact is above all the laboratory
```

This way of generating text is better than choosing random words, but still doesn't make a lot of sense.

12.8. Markov analysis

We can do better with Markov chain text analysis, which computes, for each word in a text, the list of words that come next. As an example, we'll analyze these lyrics from the Monty Python song *Eric, the Half a Bee*:

```
song = """
Half a bee, philosophically,
Must, ipso facto, half not be.
But half the bee has got to be
Vis a vis, its entity. D'you see?
"""
```

To store the results, we'll use a dictionary that maps from each word to the list of words that follow it.

```
successor_map = {}
```

As an example, let's start with the first two words of the song.

```
first = 'half'
second = 'a'
```

If the first word is not in `successor_map`, we have to add a new item that maps from the first word to a list containing the second word.

```
successor_map[first] = [second]
successor_map
```

```
{'half': ['a']}
```

If the first word is already in the dictionary, we can look it up to get the list of successors we've seen so far, and append the new one.

```
first = 'half'
second = 'not'

successor_map[first].append(second)
successor_map
```

```
{'half': ['a', 'not']}
```

The following function encapsulates these steps.

```
def add_bigram(bigram):
    first, second = bigram

    if first not in successor_map:
        successor_map[first] = [second]
    else:
        successor_map[first].append(second)
```

If the same bigram appears more that once, the second word is added to the list more than once. In this way, `successor_map` keeps track of how many times each successor appears.

As we did in the previous section, we'll use a list called `window` to store pairs of consecutive words. And we'll use the following function to process the words one at a time.

```
def process_word_bigram(word):
    window.append(word)

    if len(window) == 2:
        add_bigram(window)
        window.pop(0)
```

Here's how we use it to process the words in the song.

```
successor_map = {}
window = []

for word in song.split():
    word = clean_word(word)
    process_word_bigram(word)
```

And here are the results.

```
successor_map
```

```
{'half': ['a', 'not', 'the'],
 'a': ['bee', 'vis'],
 'bee': ['philosophically', 'has'],
 'philosophically': ['must'],
 'must': ['ipso'],
 'ipso': ['facto'],
 'facto': ['half'],
 'not': ['be'],
 'be': ['but', 'vis'],
 'but': ['half'],
 'the': ['bee'],
 'has': ['got'],
 'got': ['to'],
 'to': ['be'],
 'vis': ['a', 'its'],
 'its': ['entity'],
 'entity': ["d'you"],
 "d'you": ['see']}
```

The word `'half'` can be followed by `'a'`, `'not'`, or `'the'`. The word `'a'` can be followed by `'bee'` or `'vis'`. Most of the other words appear only once, so they are followed by only a single word.

Now let’s analyze the book.

```
successor_map = {}
window = []

for line in open(filename):
    for word in split_line(line):
        word = clean_word(word)
        process_word_bigram(word)
```

We can look up any word and find the words that can follow it.

```
successor_map['going']
```

```
['east', 'in', 'to', 'to', 'up', 'to', 'of']
```

In this list of successors, notice that the word `'to'` appears three times – the other successors only appear once.

12.9. Generating text

We can use the results from the previous section to generate new text with the same relationships between consecutive words as in the original. Here's how it works:

- Starting with any word that appears in the text, we look up its possible successors and choose one at random.
- Then, using the chosen word, we look up its possible successors, and choose one at random.

We can repeat this process to generate as many words as we want. As an example, let's start with the word `'although'`. Here are the words that can follow it.

```
word = 'although'
successors = successor_map[word]
successors
```

```
['i', 'a', 'it', 'the', 'we', 'they', 'i']
```

We can use `choice` to choose from the list with equal probability.

```
word = random.choice(successors)
word
```

```
'i'
```

If the same word appears more than once in the list, it is more likely to be selected.

Repeating these steps, we can use the following loop to generate a longer series.

```
for i in range(10):
    successors = successor_map[word]
    word = random.choice(successors)
    print(word, end=' ')
```

```
continue to hesitate and swallowed the smile withered from that
```

The result sounds more like a real sentence, but it still doesn't make much sense.

We can do better using more than one word as a key in `successor_map`. For example, we can make a dictionary that maps from each bigram – or trigram – to the list of words that come next. As an exercise, you'll have a chance to implement this analysis and see what the results look like.

12.10. Debugging

At this point we are writing more substantial programs, and you might find that you are spending more time debugging. If you are stuck on a difficult bug, here are a few things to try:

- Reading: Examine your code, read it back to yourself, and check that it says what you meant to say.
- Running: Experiment by making changes and running different versions. Often if you display the right thing at the right place in the program, the problem becomes obvious, but sometimes you have to build scaffolding.
- Ruminating: Take some time to think! What kind of error is it: syntax, runtime, or semantic? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you're seeing? What did you change last, before the problem appeared?
- Rubberducking: If you explain the problem to someone else, you sometimes find the answer before you finish asking the question. Often you don't need the other person; you could just talk to a rubber duck. And that's the origin of the well-known strategy called **rubber duck debugging**. I am not making this up – see https://en.wikipedia.org/wiki/Rubber_duck_debugging.
- Retreating: At some point, the best thing to do is back up – undoing recent changes – until you get to a program that works. Then you can start rebuilding.
- Resting: If you give your brain a break, sometime it will find the problem for you.

Beginning programmers sometimes get stuck on one of these activities and forget the others. Each activity comes with its own failure mode.

For example, reading your code works if the problem is a typographical error, but not if the problem is a conceptual misunderstanding. If you don't understand what your program does, you can read it 100 times and never see the error, because the error is in your head.

Running experiments can work, especially if you run small, simple tests. But if you run experiments without thinking or reading your code, it can take a long time to figure out what's happening.

You have to take time to think. Debugging is like an experimental science. You should have at least one hypothesis about what the problem is. If there are two or more possibilities, try to think of a test that would eliminate one of them.

But even the best debugging techniques will fail if there are too many errors, or if the code you are trying to fix is too big and complicated. Sometimes the best option is to retreat, simplifying the program until you get back to something that works.

Beginning programmers are often reluctant to retreat because they can't stand to delete a line of code (even if it's wrong). If it makes you feel better, copy your program into another file before you start stripping it down. Then you can copy the pieces back one at a time.

Finding a hard bug requires reading, running, ruminating, retreating, and sometimes resting. If you get stuck on one of these activities, try the others.

12.11. Glossary

default value: The value assigned to a parameter if no argument is provided.

override: To replace a default value with an argument.

deterministic: A deterministic program does the same thing each time it runs, given the same inputs.

pseudorandom: A pseudorandom sequence of numbers appears to be random, but is generated by a deterministic program.

bigram: A sequence of two elements, often words.

trigram: A sequence of three elements.

n-gram: A sequence of an unspecified number of elements.

rubber duck debugging: A way of debugging by explaining a problem aloud to an inanimate object.

12.12. Exercises

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

12.12.1. Ask a virtual assistant

In `add_bigram`, the `if` statement creates a new list or appends an element to an existing list, depending on whether the key is already in the dictionary.

```
def add_bigram(bigram):
    first, second = bigram

    if first not in successor_map:
        successor_map[first] = [second]
    else:
        successor_map[first].append(second)
```

Dictionaries provide a method called `setdefault` that we can use to do the same thing more concisely. Ask a virtual assistant how it works, or copy `add_bigram` into a virtual assistant and ask "Can you rewrite this using `setdefault`?"

In this chapter we implemented Markov chain text analysis and generation. If you are curious, you can ask a virtual assistant for more information on the topic. One of the things you might learn is that virtual assistants use algorithms that are similar in many ways – but also different in important ways. Ask a VA, "What are the differences between large language models like GPT and Markov chain text analysis?"

12.12.2. Exercise

Write a function that counts the number of times each trigram (sequence of three words) appears. If you test your function with the text of *Dr. Jekyll and Mr. Hyde*, you should find that the most common trigram is "said the lawyer".

Hint: Write a function called `count_trigram` that is similar to `count_bigram`. Then write a function called `process_word_trigram` that is similar to `process_word_bigram`.

12.12.3. Exercise

Now let's implement Markov chain text analysis with a mapping from each bigram to a list of possible successors.

Starting with `add_bigram`, write a function called `add_trigram` that takes a list of three words and either adds or updates an item in `successor_map`, using the first two words as the key and the third word as a possible successor.

Here's a version of `process_word_trigram` that calls `add_trigram`.

```
def process_word_trigram(word):
    window.append(word)

    if len(window) == 3:
        add_trigram(window)
        window.pop(0)
```

You can use the following loop to test your function with the words from the book.

```
successor_map = {}
window = []

for line in open(filename):
    for word in split_line(line):
        word = clean_word(word)
        process_word_trigram(word)
```

In the next exercise, you'll use the results to generate new random text.

12.12.4. Exercise

For this exercise, we'll assume that `successor_map` is a dictionary that maps from each bigram to the list of words that follow it.

To generate random text, we'll start by choosing a random key from `successor_map`.

```
successors = list(successor_map)
bigram = random.choice(successors)
bigram
```

```
('doubted', 'if')
```

Now write a loop that generates 50 more words following these steps:

1. In `successor_map`, look up the list of words that can follow `bigram`.
2. Choose one of them at random and print it.
3. For the next iteration, make a new bigram that contains the second word from `bigram` and the chosen successor.

For example, if we start with the bigram `('doubted', 'if')` and choose `'from'` as its successor, the next bigram is `('if', 'from')`.

If everything is working, you should find that the generated text is recognizably similar in style to the original, and some phrases make sense, but the text might wander from one topic to another.

As a bonus exercise, modify your solution to the last two exercises to use trigrams as keys in `successor_map`, and see what effect it has on the results.