

Khối mã này cung cấp cách bạn có thể tải các hình ảnh có trong chương sách này.

```
from os.path import basename, exists
```

```
def download(url):
    filename = basename(url)
    if not exists(filename):
        from urllib.request import urlretrieve

        local, _ = urlretrieve(url, filename)
        print("Downloaded " + str(local))
    return filename
```

```
download('https://github.com/AllenDowney/ThinkPython/raw/v3/thinkpython.py');
download('https://github.com/AllenDowney/ThinkPython/raw/v3/diagram.py');
```

```
import thinkpython
```

⇒ Downloaded thinkpython.py
Downloaded diagram.py

✓ Phân tích và Tạo văn bản

Tại thời điểm này, chúng ta đã đề cập đến các cấu trúc dữ liệu cốt lõi của Python - danh sách, từ điển và tuple - và một số thuật toán sử dụng chúng. Trong chương này, chúng ta sẽ sử dụng chúng để khám phá, phân tích, và tạo văn bản Markov:

- Phân tích văn bản là một cách để mô tả các mối quan hệ thống kê giữa các từ trong một tài liệu, chẳng hạn như xác suất một từ được theo sau bởi một từ khác.
- Tạo văn bản Markov là một cách để tạo văn bản mới với các từ và cụm từ tương tự như văn bản gốc.

Các thuật toán này tương tự với một phần của Mô hình Ngôn ngữ Lớn (LLM), thành phần chính của một chatbot.

Chúng ta sẽ bắt đầu bằng cách đếm số lần xuất hiện của mỗi từ trong một cuốn sách. Sau đó, chúng ta sẽ xem xét các cặp từ và lập danh sách các từ có thể theo sau mỗi từ. Chúng ta sẽ tạo một phiên bản đơn giản của một trình tạo văn bản Markov, và như một bài tập, bạn sẽ có cơ hội tạo một phiên bản tổng quát hơn.

✓ Từ duy nhất

Là bước đầu tiên hướng tới phân tích văn bản, hãy đọc cuốn sách *"The Strange Case Of Dr. Jekyll và Mr. Hyde"* của Robert Louis Stevenson và đếm số lượng các từ duy nhất.

Hướng dẫn tải xuống sách có trong sổ ghi chép (notebook) của chương này.

Đoạn mã sau đây tải xuống cuốn sách từ Project Gutenberg.

```
download('https://www.gutenberg.org/cache/epub/43/pg43.txt');
```

 Downloaded pg43.txt

Phiên bản có sẵn từ Project Gutenberg bao gồm thông tin về cuốn sách ở đầu và thông tin về giấy phép ở cuối.

Chúng ta sẽ sử dụng `clean_file` từ Chương 8 để loại bỏ tài liệu này và viết một tệp "sạch" chỉ chứa văn bản là nội dung của cuốn sách.

```
def is_special_line(line):
    return line.strip().startswith('*** ')

def clean_file(input_file, output_file):
    reader = open(input_file, encoding='utf-8')
    writer = open(output_file, 'w')

    for line in reader:
        if is_special_line(line):
            break

    for line in reader:
        if is_special_line(line):
            break
        writer.write(line)

    reader.close()
    writer.close()

filename = 'dr_jekyll.txt'
```

```
clean_file('pg43.txt', filename)
```

Chúng ta sẽ sử dụng vòng lặp `for` để đọc các dòng từ tệp và sử dụng `split` để chia các dòng thành các từ. Sau đó, để theo dõi các từ duy nhất, chúng ta sẽ lưu trữ mỗi từ dưới dạng một khóa trong một từ điển.

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

```
len(unique_words)
```

```
⇒ 6040
```

Độ dài của từ điển là số lượng các từ duy nhất - khoảng 6000 theo cách đếm này. Nhưng nếu chúng ta kiểm tra chúng, chúng ta sẽ thấy rằng một số không phải là từ hợp lệ.

Ví dụ: hãy xem các từ dài nhất trong `unique_words`. Chúng ta có thể sử dụng `sorted` để sắp xếp các từ, truyền hàm `len` làm đối số từ khóa để các từ được sắp xếp theo độ dài.

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

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

Chỉ số cắt, `[-5:]`, chọn 5 phần tử cuối cùng của danh sách đã sắp xếp, đó là các từ dài nhất.

Danh sách bao gồm một số từ dài hợp lệ, chẳng hạn như "circumscription", và một số từ được nối bằng dấu gạch ngang, chẳng hạn như "chocolate-coloured". Nhưng một số "từ" dài nhất thực sự là hai từ được phân tách bằng dấu gạch ngang. Và các từ khác bao gồm dấu câu như dấu chấm, dấu chấm than và dấu ngoặc kép.

Vì vậy, trước khi tiếp tục, chúng ta hãy xử lý dấu gạch ngang và các dấu câu khác.

✓ Dấu câu

Để xác định các từ trong văn bản, chúng ta cần giải quyết hai vấn đề:

Khi một dấu gạch ngang xuất hiện trong một dòng, chúng ta nên thay thế nó bằng một khoảng trắng - sau đó khi chúng ta sử dụng `split`, các từ sẽ được phân tách.

Sau khi tách các từ, chúng ta có thể sử dụng `strip` để loại bỏ dấu câu.

Để xử lý vấn đề đầu tiên, chúng ta có thể sử dụng hàm sau, hàm này nhận một chuỗi, thay thế dấu gạch ngang bằng khoảng trắng, tách chuỗi và trả về danh sách kết quả.

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

Lưu ý rằng `split_line` chỉ thay thế dấu gạch ngang, không phải dấu gạch nối. Dưới đây là một ví dụ.

```
split_line('coolness-frightened')
```

```
➞ ['coolness', 'frightened']
```

Bây giờ, để loại bỏ dấu câu khỏi đầu và cuối của mỗi từ, chúng ta có thể sử dụng `strip`, nhưng chúng ta cần một danh sách các ký tự được coi là dấu câu.

Các ký tự trong chuỗi Python nằm trong Unicode, là một tiêu chuẩn quốc tế được sử dụng để biểu diễn các chữ cái trong hầu hết mọi bảng chữ cái, số, ký hiệu, dấu câu và hơn thế nữa. Mô-đun `unicodedata` cung cấp một hàm `category` mà chúng ta có thể sử dụng để xác định loại ký tự là gì. Đối với một chữ cái nhất định, nó trả về một chuỗi với thông tin về loại của chữ cái đó.

```
import unicodedata
```

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

```
➞ 'Lu'
```

Chuỗi phân loại của 'A' là 'Lu' - 'L' nghĩa là nó là một chữ cái và 'u' nghĩa là nó là chữ hoa.

Chuỗi phân loại của '.' là 'Po' - 'P' nghĩa là nó là dấu câu và 'o' nghĩa là phân loại phụ của nó là "khác".

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

↔ 'Po'

Chúng ta có thể tìm các dấu câu trong sách bằng cách kiểm tra các ký tự có danh mục bắt đầu bằng 'P'. Vòng lặp sau đây lưu trữ các dấu câu duy nhất trong một từ điển.

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

Để tạo một danh sách các dấu câu, chúng ta có thể nối các khóa của từ điển thành một chuỗi.

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

↔ .',;,-"":?-'!()_

Bây giờ chúng ta đã biết ký tự nào trong cuốn sách là dấu câu, chúng ta có thể viết một hàm nhận một từ, loại bỏ dấu câu khỏi đầu và cuối, và chuyển đổi nó thành chữ thường.

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

Đây là ví dụ

```
clean_word('"Behold!"')
```

Bởi vì `strip` loại bỏ các ký tự từ đầu và cuối, nó để lại các từ nối bằng dấu gạch ngang.

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

```
⇒ 'pocket-handkerchief'
```

Bây giờ, đây là một vòng lặp sử dụng `split_line` và `clean_word` để xác định các từ duy nhất trong sách.

```
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
```

Với định nghĩa chặt chẽ hơn về từ, có khoảng 4000 từ duy nhất. Và chúng ta có thể xác nhận rằng danh sách các từ dài nhất đã được làm sạch.

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

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

Bây giờ, hãy xem mỗi từ được sử dụng bao nhiêu lần.

✓ Tần suất từ

Vòng lặp sau tính toán tần suất của mỗi từ duy nhất.

```
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
```

Lần đầu tiên chúng ta nhìn thấy một từ, chúng ta khởi tạo tần suất của nó thành 1. Nếu chúng ta nhìn thấy cùng một từ sau đó, chúng ta tăng tần suất của nó lên.

Để xem từ nào xuất hiện nhiều nhất, chúng ta có thể sử dụng `items` để lấy các cặp khóa - giá trị từ `word_counter` và sắp xếp chúng theo phần tử thứ hai của cặp, đó là tần suất. Đầu tiên, chúng ta sẽ định nghĩa một hàm chọn phần tử thứ hai.

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

Bây giờ chúng ta có thể sử dụng hàm `sorted` với hai đối số từ khóa:

- `key=second_element` nghĩa là các phần tử sẽ được sắp xếp theo tần suất của các từ.
- `reverse=True` nghĩa là các phần tử sẽ được sắp xếp theo thứ tự ngược, với các từ xuất hiện nhiều nhất ở đầu.

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

Đây là năm từ xuất hiện nhiều nhất.

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

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

Trong phần tiếp theo, chúng ta sẽ đóng gói vòng lặp này trong một hàm. Và chúng ta sẽ sử dụng nó để chứng minh một tính năng mới - các tham số tùy chọn.

✓ Tham số tùy chọn

Chúng ta đã sử dụng các hàm tích hợp sẵn nhận các tham số tùy chọn. Ví dụ: `round` nhận một tham số tùy chọn gọi là `ndigits` chỉ định các chữ số thập phân cần giữ lại.

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

```
➞ 3.142
```

Nhưng không chỉ các hàm tích hợp sẵn - chúng ta có thể viết các hàm có tham số tùy chọn. Ví dụ, hàm sau nhận hai tham số, `word_counter` và `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')
```

Tham số thứ hai trông giống như một câu lệnh gán, nhưng nó không phải - nó là một tham số tùy chọn.

Nếu bạn gọi hàm này với một đối số, `num` nhận giá trị mặc định là 5.

```
print_most_common(word_counter)
```

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

Nếu bạn gọi hàm này với hai đối số, đối số thứ hai sẽ được gán cho `num` thay vì giá trị mặc định.

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

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


Trong trường hợp đó, chúng ta sẽ nói rằng đối số tùy chọn ghi đè lên giá trị mặc định.

Nếu một hàm có cả tham số bắt buộc và tùy chọn, tất cả các tham số bắt buộc phải đến trước, tiếp theo là các tham số tùy chọn.

```
%%expect SyntaxError
```

```
def bad_function(n=5, word_counter):  
    return None
```

```
File "<ipython-input-27-e9ac4bb13aea>", line 1  
    def bad_function(n=5, word_counter):  
        ^  
SyntaxError: non-default argument follows default argument
```

✓ Trừ từ điển

Giả sử chúng ta muốn kiểm tra chính tả của một cuốn sách - tức là tìm danh sách các từ có thể bị viết sai chính tả. Một cách để làm điều đó là tìm các từ trong sách không xuất hiện trong danh sách các từ hợp lệ. Trong các chương trước, chúng ta đã sử dụng một danh sách các từ được coi là hợp lệ trong các trò chơi chữ như Scrabble. Bây giờ, chúng ta sẽ sử dụng danh sách này để kiểm tra chính tả của Robert Louis Stevenson.

Chúng ta có thể nghĩ về vấn đề này như là phép trừ tập hợp - nghĩa là chúng ta muốn tìm tất cả các từ từ một tập hợp (các từ trong cuốn sách) không nằm trong tập hợp kia (các từ trong danh sách).

Đoạn mã sau đây tải xuống danh sách từ.

```
download('https://raw.githubusercontent.com/AllenDowney/ThinkPython/v3/words.txt')
```

```
Downloaded words.txt
```

Như chúng ta đã làm trước đây, chúng ta có thể đọc nội dung của `words.txt` và chia nó thành một danh sách các chuỗi.

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

Sau đó, chúng ta sẽ lưu trữ các từ dưới dạng khóa trong một từ điển để chúng ta có thể sử dụng toán tử `in` để kiểm tra nhanh xem một từ có hợp lệ hay

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

Bây giờ, để xác định các từ xuất hiện trong sách nhưng không xuất hiện trong danh sách từ, chúng ta sẽ sử dụng `subtract`, hàm này nhận hai từ điển làm tham số và trả về một từ điển mới chứa tất cả các khóa từ một từ điển mà không có trong từ điển kia.

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

Đây là cách chúng ta sử dụng nó.

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

Để lấy mẫu các từ có thể bị viết sai chính tả, chúng ta có thể in các từ phổ biến nhất trong `diff`.

```
print_most_common(diff)
```

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

Các từ "sai chính tả" phổ biến nhất chủ yếu là tên riêng và một vài từ đơn âm (Mr. Utterson is Dr. Jekyll's friend and lawyer).

Nếu chúng ta chọn những từ chỉ xuất hiện một lần, chúng có nhiều khả năng là lỗi chính tả thực sự. Chúng ta có thể làm điều đó bằng cách lặp qua các mục và tạo một danh sách các từ có tần suất là 1.

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

Đây là một vài phần tử cuối cùng của danh sách.

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

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

Hầu hết trong số chúng là những từ hợp lệ không có trong danh sách từ. Nhưng 'reindue' dường như là một lỗi chính tả của 'reinduce', vì vậy ít nhất chúng ta đã tìm thấy một lỗi hợp lệ.

✓ Số ngẫu nhiên

Là một bước tiến tới việc tạo văn bản Markov, tiếp theo chúng ta sẽ chọn một chuỗi các từ ngẫu nhiên từ `word_counter`. Nhưng trước tiên, hãy nói về ngẫu nhiên.

Cho cùng một đầu vào, hầu hết các chương trình máy tính là **xác định**, nghĩa là chúng tạo ra cùng một đầu ra mỗi lần. Tính xác định thường là một điều tốt, vì chúng ta mong đợi cùng một phép tính sẽ cho cùng một kết quả. Tuy nhiên, đối với một số ứng dụng, chúng ta muốn máy tính không thể đoán trước được. Trò chơi là một ví dụ, nhưng còn nhiều hơn thế.

Việc tạo ra một chương trình thực sự không xác định hóa ra là khó khăn, nhưng có những cách để giả mạo nó. Một cách là sử dụng các thuật toán tạo ra các số **giả ngẫu nhiên**. Các số giả ngẫu nhiên không thực sự ngẫu nhiên vì chúng được tạo ra bằng một phép tính xác định, nhưng chỉ bằng cách nhìn vào các số, gần như không thể phân biệt chúng với các số ngẫu nhiên.

Mô-đun `random` cung cấp các hàm tạo ra các số giả ngẫu nhiên - mà tôi sẽ đơn giản gọi là "ngẫu nhiên" từ đây trở đi. Chúng ta có thể nhập nó như thế này:

```
import random
```

```
# đoạn mã này khởi tạo bộ tạo số ngẫu nhiên để nó  
# tạo ra cùng một chuỗi mỗi khi số ghi chép chạy.
```

```
random.seed(4)
```

Mô-đun `random` cung cấp một hàm gọi là `choice` chọn một phần tử từ một danh sách một cách ngẫu nhiên, với mỗi phần tử có cùng xác suất được chọn.

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

```
➡ 1
```

Nếu bạn gọi lại hàm, bạn có thể nhận được cùng một phần tử hoặc một phần tử khác.

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

```
➡ 2
```

Về lâu dài, chúng ta mong đợi nhận được mọi phần tử cùng một số lần.

Nếu bạn sử dụng `choice` với một từ điển, bạn sẽ gặp lỗi `KeyError`.

```
%%expect KeyError
```

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

```
➡ KeyError: 422
```

Để chọn một khóa ngẫu nhiên, bạn phải đặt các khóa vào một danh sách và sau đó gọi `choice`.

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

```
➡ 'posture'
```

Nếu chúng ta tạo một chuỗi các từ ngẫu nhiên, nó không có nhiều ý nghĩa.

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

➡ listen relieved apocryphal nor busy spoke

Phần lớn vấn đề là chúng ta không tính đến việc một số từ phổ biến hơn những từ khác. Kết quả sẽ tốt hơn nếu chúng ta chọn các từ với "trọng số" khác nhau, để một số từ được chọn thường xuyên hơn những từ khác.

Nếu chúng ta sử dụng các giá trị từ `word_counter` làm trọng số, mỗi từ được chọn với xác suất phụ thuộc vào tần suất của nó.

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

Mô-đun `random` cung cấp một hàm khác gọi là `choices` nhận trọng số làm đối số tùy chọn.

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

➡ ['than']

Và nó nhận một đối số tùy chọn khác, `k`, xác định số lượng từ cần chọn.

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

➡ ['reach', 'streets', 'disappearance', 'a', 'said', 'to']

Kết quả là một danh sách các chuỗi mà chúng ta có thể nối lại thành một thứ gì đó giống với một câu hơn.

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

➡ 'reach streets disappearance a said to'

Nếu bạn chọn các từ từ cuốn sách một cách ngẫu nhiên, bạn sẽ có cảm giác về vốn từ vựng, nhưng một loạt các từ ngẫu nhiên hiếm khi có ý nghĩa vì không có mối quan hệ nào giữa các từ liên tiếp. Ví dụ: trong một câu thực, bạn mong đợi một mạo từ như "the" được theo sau bởi một tính từ hoặc một danh từ, và có lẽ không phải là một động từ hoặc trạng từ. Vì vậy, bước tiếp theo là xem xét các mối quan hệ giữa các từ.

✓ Chuỗi hai từ

Thay vì xem xét từng từ một, bây giờ chúng ta sẽ xem xét các chuỗi hai từ, được gọi là **chuỗi hai từ (bigram)**. Một chuỗi ba từ được gọi là **chuỗi ba từ (trigram)** và một chuỗi với một số lượng từ không xác định được gọi là **chuỗi n từ (n-gram)**.

Hãy viết một chương trình tìm tất cả các bigram trong sách và số lần xuất hiện của mỗi bigram. Để lưu trữ kết quả, chúng ta sẽ sử dụng một từ điển trong đó:

- **Khóa** là các tuple của các chuỗi đại diện cho bigram.
- **Giá trị** là các số nguyên đại diện cho tần suất.

Hãy gọi nó là `bigram_counter`.

```
bigram_counter = {}
```

Hàm sau nhận một danh sách hai chuỗi làm tham số. Đầu tiên, nó tạo một tuple của hai chuỗi, có thể được sử dụng làm khóa trong một từ điển.

Sau đó, nó thêm khóa vào `bigram_counter` nếu nó không tồn tại, hoặc tăng tần suất nếu nó tồn tại.

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

Khi chúng ta lướt qua cuốn sách, chúng ta phải theo dõi từng cặp từ liên tiếp. Vì vậy, nếu chúng ta thấy chuỗi "man is not truly one", chúng ta sẽ thêm các bigram "man is", "is not", "not truly", v.v. Để theo dõi các bigram này, chúng ta sẽ sử dụng một danh sách gọi là `window`, vì nó giống như một cửa sổ trượt trên các trang sách, chỉ hiển thị hai từ tại một thời điểm. Ban đầu, `window` trống.

```
window = []
```

Chúng ta sẽ sử dụng hàm sau để xử lý các từ từng cái một.

```
def process_word(word):  
    window.append(word)  
  
    if len(window) == 2:  
        count_bigram(window)  
        window.pop(0)
```

Lần đầu tiên hàm này được gọi, nó nối từ đã cho vào `window`. Vì chỉ có một từ trong cửa sổ nên chúng ta chưa có bigram nào, vì vậy hàm kết thúc.

Lần thứ hai nó được gọi - và mọi lần sau đó - nó nối thêm một từ thứ hai vào `window`. Vì có hai từ trong cửa sổ nên nó gọi `count_bigram` để theo dõi số lần xuất hiện của mỗi bigram. Sau đó, nó sử dụng `pop` để xóa từ đầu tiên khỏi cửa sổ.

Chương trình sau lặp qua các từ trong sách và xử lý chúng từng cái một.

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

Kết quả là một từ điển ánh xạ từ mỗi bigram đến số lần xuất hiện của nó. Chúng ta có thể sử dụng `print_most_common` để xem các bigram phổ biến nhất.

```
print_most_common(bigram_counter)
```

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

Nhìn vào những kết quả này, chúng ta có thể hiểu được cặp từ nào có khả năng xuất hiện cùng nhau nhất. Chúng ta cũng có thể sử dụng kết quả để tạo văn bản ngẫu nhiên, như thế này.

```
random.seed(0)
```

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

`bigrams` là một danh sách các bigram xuất hiện trong sách. `weights` là một danh sách tần suất của chúng, vì vậy `random_bigrams` là một mẫu mà xác suất một bigram được chọn tỷ lệ thuận với tần suất của nó.

Đây là kết quả.

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

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

Cách tạo văn bản này tốt hơn so với việc chọn các từ ngẫu nhiên, nhưng vẫn không có nhiều ý nghĩa.

✓ Phân tích Markov

Chúng ta có thể làm tốt hơn với phân tích chuỗi Markov, tính toán, đối với mỗi từ trong một văn bản, danh sách các từ xuất hiện tiếp theo. Ví dụ, chúng ta sẽ phân tích lời bài hát của bài hát "Eric, the Half a Bee" của Monty Python:


```
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?
""
```

Để lưu trữ kết quả, chúng ta sẽ sử dụng một từ điển ánh xạ từ mỗi từ đến danh sách các từ theo sau nó.

```
successor_map = {}
```

Ví dụ, hãy bắt đầu với hai từ đầu tiên của bài hát.

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

Nếu từ đầu tiên không có trong `successor_map`, chúng ta phải thêm một mục mới ánh xạ từ từ đầu tiên đến một danh sách chứa từ thứ hai.

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

```
↔ {'half': ['a']}
```

Nếu từ đầu tiên đã có trong từ điển, chúng ta có thể tra cứu nó để lấy danh sách các từ kế tiếp mà chúng ta đã thấy cho đến nay và nối thêm từ mới.

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

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

```
↔ {'half': ['a', 'not']}
```

Hàm sau đóng gói các bước này.

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

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

Nếu cùng một bigram xuất hiện nhiều hơn một lần, từ thứ hai được thêm vào danh sách nhiều hơn một lần. Bằng cách này, `successor_map` theo dõi số lần xuất hiện của mỗi từ kế tiếp.

Giống như trước đây, chúng ta sẽ sử dụng một danh sách gọi là `window` để lưu trữ các cặp từ liên tiếp. Và chúng ta sẽ sử dụng hàm sau để xử lý các từ từng cái một.

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

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

Đây là cách chúng ta sử dụng nó để xử lý các từ trong bài hát.

```
successor_map = {}
window = []

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

Và đây là kết quả.

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']}
```

Từ "half" có thể được theo sau bởi "a", "not" hoặc "the". Từ "a" có thể được theo sau bởi "bee" hoặc "vis". Hầu hết các từ khác chỉ xuất hiện một lần, vì vậy chúng chỉ được theo sau bởi một từ duy nhất.

Bây giờ hãy phân tích cuốn sách.

```
successor_map = {}  
window = []  
  
for line in open(filename):  
    for word in split_line(line):  
        word = clean_word(word)  
        process_word_bigram(word)
```

Chúng ta có thể tra cứu bất kỳ từ nào và tìm các từ có thể theo sau nó.

```
# Tôi đã sử dụng ô này để tìm một tiên tố có số lượng hậu tố khả thi tốt  
# và ít nhất một từ lặp lại.
```

```
def has_duplicates(t):  
    return len(set(t)) < len(t)
```

```
for key, value in successor_map.items():  
    if len(value) == 7 and has_duplicates(value):  
        print(key, value)
```

```
⇒ story ['of', 'of', 'indeed', 'but', 'for', 'of', 'that']  
incident ['of', 'of', 'at', 'of', 'of', 'at', 'this']  
lanyon's ['narrative', 'there', 'face', 'manner', 'narrative', 'the', 'condem']  
common ['it', 'interest', 'friends', 'friends', 'observers', 'but', 'quarry']  
relief ['the', 'to', 'when', 'that', 'that', 'of', 'it']  
appearance ['of', 'well', 'something', 'he', 'amply', 'of', 'which']  
going ['east', 'in', 'to', 'to', 'up', 'to', 'of']  
till ['at', 'the', 'i', 'yesterday', 'the', 'that', 'weariness']  
walk ['and', 'into', 'with', 'all', 'steadfastly', 'attired', 'with']  
sounds ['nothing', 'carried', 'out', 'the', 'of', 'of', 'with']  
really ['like', 'damnable', 'can', 'a', 'a', 'not', 'be']  
does ['not', 'not', 'indeed', 'not', 'the', 'not', 'not']  
reply ['i', 'but', 'whose', 'i', 'some', 'that's', 'i']  
continued ['mr', 'the', 'the', 'the', 'the', 'poole', 'utterson']  
seems ['scarcely', 'hardly', 'to', 'she', 'much', 'he', 'to']  
walked ['on', 'over', 'some', 'was', 'on', 'with', 'fast']  
that's ['a', 'it', 'talking', 'very', 'not', 'such', 'not']  
although ['i', 'a', 'it', 'the', 'we', 'they', 'i']  
until ['the', 'the', 'they', 'they', 'the', 'to-morrow', 'i']  
disappearance ['or', 'the', 'of', 'of', 'here', 'and', 'but']  
step ['into', 'or', 'back', 'natural', 'into', 'of', 'leaping']  
wish ['the', 'you', 'to', 'you', 'to', 'i', 'to']  
aware ['of', 'of', 'of', 'that', 'jekyll's', 'that', 'of']  
thank ['you', 'you', 'you', 'you', 'you', 'god', 'you']  
maid ['servant', 'described', 'fainted', 'had', 'calls', 'had', 'lifted']  
besides ['were', 'was', 'for', 'a', 'with', 'with', 'which']  
observed ['the', 'utterson', 'with', 'the', 'that', 'that', 'that']  
among ['other', 'the', 'the', 'the', 'my', 'my', 'temptations']
```

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

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

Trong danh sách các từ kế tiếp này, hãy chú ý rằng từ "to" xuất hiện ba lần - các từ kế tiếp khác chỉ xuất hiện một lần.

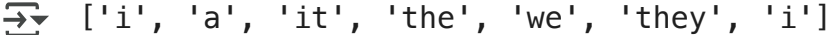
✓ Tạo văn bản

Chúng ta có thể sử dụng kết quả từ phần trước để tạo văn bản mới với các mối quan hệ giữa các từ liên tiếp giống như trong bản gốc. Đây là cách nó hoạt động:

- Bắt đầu với bất kỳ từ nào xuất hiện trong văn bản, chúng ta tra cứu các từ kế tiếp có thể có và chọn một từ ngẫu nhiên.
- Sau đó, sử dụng từ được chọn, chúng ta tra cứu các từ kế tiếp có thể có và chọn một từ ngẫu nhiên.

Chúng ta có thể lặp lại quá trình này để tạo ra nhiều từ tùy thích. Ví dụ: hãy bắt đầu với từ "although". Đây là những từ có thể theo sau nó:

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



```
# Đoạn mã này khởi tạo bộ tạo số ngẫu nhiên để nó
# bắt đầu từ cùng một điểm trong chuỗi mỗi khi
# số ghi chép chạy.
```

```
random.seed(2)
```

Chúng ta có thể sử dụng `choice` để chọn từ danh sách với xác suất bằng nhau.

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



Nếu cùng một từ xuất hiện nhiều hơn một lần trong danh sách, nó có nhiều khả năng được chọn.

Lặp lại các bước này, chúng ta có thể sử dụng vòng lặp sau để tạo một chuỗi dài hơn.

```
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

Kết quả nghe giống một câu thực hơn, nhưng nó vẫn không có nhiều ý nghĩa.

Chúng ta có thể làm tốt hơn bằng cách sử dụng nhiều hơn một từ làm khóa trong `successor_map`. Ví dụ, chúng ta có thể tạo một từ điển ánh xạ từ mỗi bigram - hoặc trigram - đến danh sách các từ xuất hiện tiếp theo. Như một bài tập, bạn sẽ có cơ hội thực hiện phân tích này và xem kết quả trông như thế nào.

✓ Gỡ lỗi

Đến thời điểm này, chúng ta đang viết các chương trình quan trọng hơn và bạn có thể thấy mình dành nhiều thời gian hơn cho việc gỡ rối.

Nếu bạn gặp phải lỗi khó, dưới đây là một vài điều bạn có thể thử:

- **Đọc:** Kiểm tra kỹ lưỡng mã của bạn, đọc lại nó và kiểm tra xem nó có khớp với những gì bạn định nói không.
- **Chạy:** Thực nghiệm bằng cách thực hiện các thay đổi nhỏ và chạy các phiên bản khác nhau. Thông thường, nếu bạn hiển thị đúng thứ gì đó ở đúng vị trí trong chương trình, vấn đề sẽ trở nên rõ ràng, nhưng đôi khi bạn phải xây dựng giàn giáo.
- **Suy ngẫm:** Dành thời gian để suy nghĩ! Đó là lỗi gì: cú pháp, thời gian chạy hay ngữ nghĩa? Bạn có thể lấy được thông tin gì từ thông báo lỗi hoặc từ đầu ra của chương trình? Loại lỗi nào có thể gây ra sự cố bạn đang gặp phải? Bạn đã thay đổi gì gần đây, trước khi sự cố xuất hiện?
- **Gỡ rối vịt cao su:** Nếu bạn giải thích vấn đề cho người khác, đôi khi bạn có thể tìm thấy câu trả lời trước khi bạn hỏi xong. Thông thường bạn không cần người khác; bạn chỉ có thể nói chuyện với một con vịt cao su. Và đó chính là nguồn gốc của chiến lược nổi tiếng gọi là gỡ rối vịt cao su. Tôi không bịa chuyện đâu - hãy xem Wikipedia về **Gỡ rối vịt cao su** https://en.wikipedia.org/wiki/Rubber_duck_debugging.
- **Quay lại:** Tại một thời điểm nào đó, điều tốt nhất bạn nên làm là sao lưu - hoàn tác các thay đổi gần đây - cho đến khi bạn có được một chương trình hoạt động. Sau đó, bạn có thể bắt đầu xây dựng lại.
- **Nghỉ ngơi:** Nếu bạn cho não nghỉ ngơi, đôi khi nó sẽ tìm ra vấn đề cho bạn.

Lập trình viên mới bắt đầu đôi khi bị mắc kẹt trong một trong những hoạt động này và quên những hoạt động khác. Mỗi hoạt động đều có chế độ thất bại riêng.

Ví dụ: Đọc mã của bạn hoạt động nếu vấn đề là lỗi đánh máy, nhưng không hoạt động nếu vấn đề là hiểu sai về khái niệm. Nếu bạn không hiểu chương trình của mình hoạt động như thế nào, bạn có thể đọc nó 100 lần và không bao giờ thấy lỗi, vì lỗi nằm trong đầu bạn.

Thực hiện các thí nghiệm có thể hoạt động, đặc biệt nếu bạn chạy các thử nghiệm nhỏ và đơn giản. Nhưng nếu bạn thực hiện các thí nghiệm mà không suy nghĩ hoặc đọc mã của mình, bạn có thể mất nhiều thời gian để tìm ra những gì đang xảy ra.

Bạn phải dành thời gian để suy nghĩ. Gỡ lỗi giống như một khoa học thực nghiệm. Bạn nên có ít nhất một giả thuyết về vấn đề là gì. Nếu có hai hoặc nhiều khả năng, hãy thử nghĩ ra một bài kiểm tra để loại bỏ một trong số chúng.

Nhưng ngay cả những kỹ thuật gỡ rối tốt nhất cũng sẽ thất bại nếu có quá nhiều lỗi hoặc nếu mã bạn đang cố sửa quá lớn và phức tạp. Đôi khi lựa chọn tốt nhất là lùi lại, đơn giản hóa chương trình cho đến khi bạn quay lại được với một chương trình hoạt động.

Lập trình viên mới bắt đầu thường miễn cưỡng lùi bước vì họ không thể chịu xóa một dòng mã (ngay cả khi nó sai). Nếu điều đó khiến bạn cảm thấy tốt hơn, hãy sao chép chương trình của bạn vào một tệp khác trước khi bạn bắt đầu loại bỏ nó. Sau đó, bạn có thể sao chép các phần từng cái một.

Tìm một lỗi khó đòi hỏi phải đọc, chạy, suy ngẫm, rút lui và đôi khi nghỉ ngơi. Nếu bạn bị mắc kẹt ở một trong những hoạt động này, hãy thử những hoạt động khác.

Thuật ngữ

default value - giá trị mặc định: Giá trị mặc định được gán cho một tham số nếu không có đối số nào được cung cấp.

override - ghi đè: Thay thế một giá trị mặc định bằng một đối số.

deterministic - xác định: Một chương trình xác định làm điều tương tự mỗi khi nó chạy, với cùng một đầu vào.

pseudorandom - giả ngẫu nhiên: Một dãy số giả ngẫu nhiên xuất hiện ngẫu nhiên nhưng được tạo ra bởi một chương trình xác định.

bigram - chuỗi hai từ: Một chuỗi gồm hai phần tử, thường là từ.

trigram - chuỗi ba từ: Một chuỗi gồm ba phần tử.

n-gram - chuỗi n từ: Một chuỗi gồm một số lượng phần tử không xác định.

rubber duck debugging - gỡ rối vịt cao su: Một cách gỡ lỗi bằng cách giải thích vấn đề một cách rõ ràng cho một vật vô tri vô giác.

✓ Bài tập

```
# Ô này yêu cầu Jupyter cung cấp thông tin gỡ lỗi chi tiết
# khi xảy ra lỗi thời gian chạy. Chạy nó trước khi làm các bài tập.
```

```
%xmode Verbose
```

✓ Hỏi một trợ lý ảo

Trong hàm `add_bigram`, câu lệnh `if` tạo ra một danh sách mới hoặc nối thêm một phần tử vào một danh sách hiện có, tùy thuộc vào việc khóa đã có trong từ điển hay chưa.

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

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

Từ điển cung cấp một phương thức gọi là `setdefault` giúp chúng ta thực hiện cùng một việc một cách ngắn gọn hơn. Bạn có thể hỏi một trợ lý ảo về cách hoạt động của nó, hoặc sao chép hàm `add_word` vào một trợ lý ảo và hỏi "Bạn có thể viết lại đoạn này bằng `setdefault` không?"

Trong chương này, chúng ta đã thực hiện phân tích và tạo văn bản dựa trên chuỗi Markov. Nếu bạn tò mò, bạn có thể hỏi một trợ lý ảo để biết thêm thông tin về chủ đề này. Một điều thú vị là các trợ lý ảo sử dụng các thuật toán tương tự nhau ở nhiều khía cạnh, nhưng cũng khác nhau ở những điểm quan trọng. Bạn có thể hỏi một trợ lý ảo, "Sự khác biệt giữa các mô hình ngôn ngữ lớn như GPT và phân tích chuỗi Markov là gì?"

✓ Bài tập 1

Viết một hàm đếm số lần xuất hiện của mỗi trigram (chuỗi ba từ). Nếu bạn kiểm tra hàm của mình với văn bản của "Dr. Jekyll and Mr. Hyde", bạn sẽ thấy rằng trigram phổ biến nhất là "said the lawyer".

Gợi ý: Viết một hàm gọi là `count_trigram` tương tự như `count_bigram`. Sau đó, viết một hàm gọi là `process_word_trigram` tương tự như `process_word_bigram`.

Bạn có thể sử dụng vòng lặp sau để đọc sách và xử lý các từ.

```
trigram_counter = {}
window = []

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

Sau đó, sử dụng `print_most_common` để tìm các trigram phổ biến nhất trong sách.

```
print_most_common(trigram_counter)
```

✓ Bài tập 2

Bây giờ, hãy triển khai phân tích văn bản chuỗi Markov với một ánh xạ từ mỗi bigram đến một danh sách các hậu tố có thể.

Bắt đầu với `add_bigram`, hãy viết một hàm gọi là `add_trigram` nhận một danh sách ba từ và thêm hoặc cập nhật một mục trong `successor_map`, sử dụng hai từ đầu làm khóa và từ thứ ba làm hậu tố có thể.

Đây là một phiên bản của `process_word_trigram` gọi `add_trigram`.

```
def process_word_trigram(word):  
    window.append(word)  
  
    if len(window) == 3:  
        add_trigram(window)  
        window.pop(0)
```

Bạn có thể sử dụng vòng lặp sau để kiểm tra hàm của bạn với lời bài hát của "Eric, the Half a Bee".

```
successor_map = {}  
window = []  
  
for string in song.split():  
    word = string.strip(punctuation).lower()  
    process_word_trigram(word)
```

Nếu hàm của bạn hoạt động đúng, tiền tố ('half', 'a') sẽ ánh xạ đến một danh sách với phần tử duy nhất 'bee'. Trên thực tế, như đã xảy ra, mỗi bigram trong bài hát này chỉ xuất hiện một lần, vì vậy tất cả các giá trị trong `successor_map` đều có một phần tử duy nhất.

```
successor_map
```

Bạn có thể sử dụng vòng lặp sau để kiểm tra hàm của bạn với các từ trong sách.

```
successor_map = {}
window = []

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

Trong bài tập tiếp theo, bạn sẽ sử dụng kết quả để tạo văn bản ngẫu nhiên mới.

✓ Bài tập 3

Đối với bài tập này, chúng ta sẽ giả định rằng `successor_map` là một từ điển ánh xạ từ mỗi bigram đến danh sách các từ theo sau nó.

```
# Đoạn mã này khởi tạo bộ tạo số ngẫu nhiên để nó
# bắt đầu từ cùng một điểm trong chuỗi mỗi khi
# số ghi chép chạy.
```

```
random.seed(3)
```

Để tạo văn bản ngẫu nhiên, chúng ta sẽ bắt đầu bằng cách chọn một khóa ngẫu nhiên từ `successor_map`.

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

Bây giờ, hãy viết một vòng lặp tạo ra 50 từ tiếp theo theo các bước sau:

1. Trong `successor_map`, tìm danh sách các từ có thể theo sau `bigram`.
2. Chọn ngẫu nhiên một trong số các từ đó và in ra.
3. Đối với lần lặp tiếp theo, tạo một `bigram` mới chứa từ thứ hai của `bigram` cũ và từ vừa chọn.

Ví dụ, nếu chúng ta bắt đầu với `bigram` là `('doubted', 'if')` và chọn `'from'` làm từ kế tiếp, thì `bigram` tiếp theo sẽ là `('if', 'from')`.

Nếu mọi thứ hoạt động đúng, bạn sẽ thấy rằng văn bản được tạo ra có phong cách tương tự với bản gốc và một số cụm từ có ý nghĩa, nhưng văn bản có thể đi lang thang từ chủ đề này sang chủ đề khác.

Như một bài tập bổ sung, hãy sửa đổi giải pháp của bạn cho hai bài tập cuối để sử dụng trigram làm khóa trong `successor_map` và xem ảnh hưởng của nó đến kết quả.