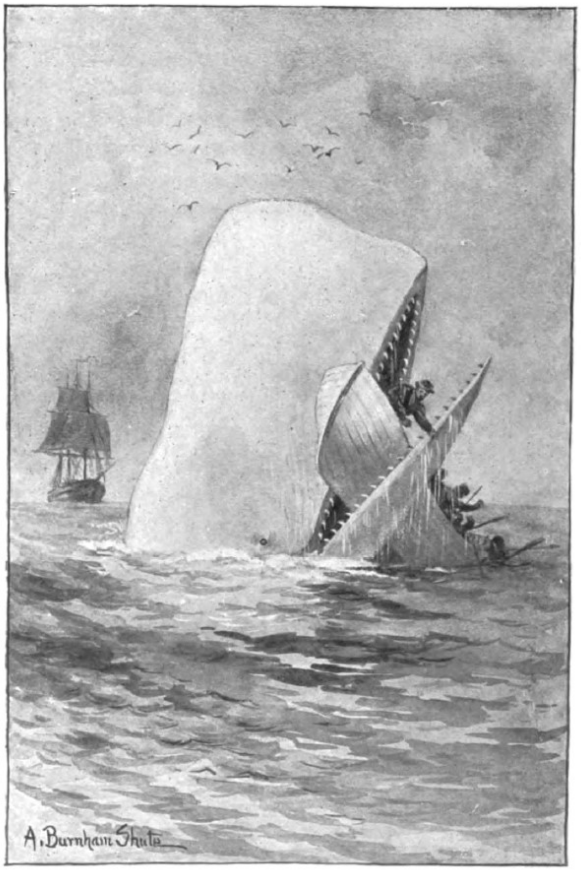
1. Tools for text processing

What are the most frequent words in Herman Melville's novel Moby Dick and how often do they occur?

In this notebook, we'll scrape the novel *Moby Dick* from the website Project Gutenberg (https://www.gutenberg.org/) (which contains a large corpus of books) using the Python package `requests`. Then we'll extract words from this web data using `BeautifulSoup`. Finally, we'll dive into analyzing the distribution of words using the Natural Language ToolKit (`nltk`).

The *Data Science pipeline* we'll build in this notebook can be used to visualize the word frequency distributions of any novel that you can find on Project Gutenberg. The natural language processing tools used here apply to much of the data that data scientists encounter as a vast proportion of the world's data is unstructured data and includes a great deal of text.

Let's start by loading in the three main python packages we are going to use.



"Both jaws, like enormous shears, bit the craft completely in twain."

—*Page 510.*

In [90]:

```
# Importing requests, BeautifulSoup and nltk
import requests
from bs4 import BeautifulSoup
import nltk
```

In [91]:

```
%%nose

import sys

def test_example():
    assert ('requests' in sys.modules and
            'bs4' in sys.modules and
            'nltk' in sys.modules ), \
    'The modules requests, BeautifulSoup, and nltk should be imported.'
```

Out[91]:

```
1/1 tests passed
```

2. Request Moby Dick

To analyze Moby Dick, we need to get the contents of Moby Dick from *somewhere*. Luckily, the text is freely available online at Project Gutenberg as an HTML file: https://www.gutenberg.org/files/2701/2701-h/2701-h.htm .

Note that HTML stands for Hypertext Markup Language and is the standard markup language for the web.

To fetch the HTML file with Moby Dick we're going to use the `request` package to make a `GET` request for the website, which means we're *getting* data from it. This is what you're doing through a browser when visiting a webpage, but now we're getting the requested page directly into python instead.

In [92]:

```
# Getting the Moby Dick HTML
r = requests.get("https://s3.amazonaws.com/assets.datacamp.com/production/project_147/d
atasets/2701-h.htm")

# Setting the correct text encoding of the HTML page
r.encoding = 'utf-8'

# Extracting the HTML from the request object
html = r.text

# Printing the first 2000 characters in html
print(html[:2000])
```

```
<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE html
   PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd" >

<html xmlns="http://www.w3.org/1999/xhtml" lang="en">
  <head>
    <title>
      Moby Dick; Or the Whale, by Herman Melville
    </title>
    <style type="text/css" xml:space="preserve">

    body { background:#faebd0; color:black; margin-left:15%; margin-right:
15%; text-align:justify }
    P { text-indent: 1em; margin-top: .25em; margin-bottom: .25em; }
    H1,H2,H3,H4,H5,H6 { text-align: center; margin-left: 15%; margin-righ
t: 15%; }
    hr  { width: 50%; text-align: center;}
    .foot { margin-left: 20%; margin-right: 20%; text-align: justify; text
-indent: -3em; font-size: 90%; }
    blockquote {font-size: 100%; margin-left: 0%; margin-right: 0%;}
    .mynote    {background-color: #DDE; color: #000; padding: .5em; margin
-left: 10%; margin-right: 10%; font-family: sans-serif; font-size: 95%;}
    .toc       { margin-left: 10%; margin-bottom: .75em;}
    .toc2      { margin-left: 20%;}
    div.fig    { display:block; margin:0 auto; text-align:center; }
    div.middle { margin-left: 20%; margin-right: 20%; text-align: justify;
}
    .figleft   {float: left; margin-left: 0%; margin-right: 1%;}
    .figright  {float: right; margin-right: 0%; margin-left: 1%;}
    .pagenum   {display:inline; font-size: 70%; font-style:normal;
                margin: 0; padding: 0; position: absolute; right: 1%;
                text-align: right;}
    pre        { font-family: times new roman; font-size: 100%; margin-lef
t: 10%;}

    table      {margin-left: 10%;}

a:link {color:blue;
                text-decoration:none}
link {color:blue;
                text-decoration:none}
a:visited {color:blue;
                text-decoration:none}
a:hover {color:red}

</style>
  </head>
  <body>
<pre xml:space="preserve">

The Project Gutenberg EBook of Moby Dick; or The Whale, by Herman Melville

This eBook is for the use of anyone anywh
```

In [93]:

```
%%nose

def test_r_correct():
    assert r.request.path_url == '/assets.datacamp.com/production/project_147/datasets/
2701-h.htm', \
    "r should be a get request for 'https://s3.amazonaws.com/assets.datacamp.com/produc
tion/project_147/datasets/2701-h.htm'"

def test_text_read_in_correctly():
    assert len(html) == 1500996, \
    'html should contain the text of the request r.'
```

Out[93]:

```
2/2 tests passed
```

3. Get the text from the HTML

This HTML is not quite what we want. However, it does *contain* what we want: the text of *Moby Dick*. What we need to do now is *wrangle* this HTML to extract the text of the novel. For this we'll use the package `BeautifulSoup`.

Firstly, a word on the name of the package: Beautiful Soup? In web development, the term "tag soup" refers to structurally or syntactically incorrect HTML code written for a web page. What Beautiful Soup does best is to make tag soup beautiful again and to extract information from it with ease! In fact, the main object created and queried when using this package is called `BeautifulSoup`. After creating the soup, we can use its `.get_text()` method to extract the text.

In [94]:

```
# Creating a BeautifulSoup object from the HTML
soup = BeautifulSoup(html)

# Getting the text out of the soup
text = soup.get_text()

# Printing out text between characters 32000 and 34000
text[3200:3400]
```

Out[94]:

```
'15. Chowder. \n      \n      \n         CHAPTER 16. The Ship. \n      \n
     \n         CHAPTER 17. The Ramadan. \n      \n      \n         CHAPTE
R 18. His Mark. \n      \n      \n         CHAPTER 19. The Prophe'
```

In [95]:

```
%%nose

import bs4

def test_text_correct_type():
    assert isinstance(text, str), \
    'text should be a string'

def test_soup_correct_type():
    assert isinstance(soup, bs4.BeautifulSoup), \
    'soup should be a BeautifulSoup object'
```

Out[95]:

```
2/2 tests passed
```

4. Extract the words

We now have the text of the novel! There is some unwanted stuff at the start and some unwanted stuff at the end. We could remove it, but this content is so much smaller in amount than the text of Moby Dick that, to a first approximation, it is okay to leave it in.

Now that we have the text of interest, it's time to count how many times each word appears, and for this we'll use `nltk` – the Natural Language Toolkit. We'll start by tokenizing the text, that is, remove everything that isn't a word (whitespace, punctuation, etc.) and then split the text into a list of words.

In [96]:

```
# Creating a tokenizer
tokenizer = nltk.tokenize.RegexpTokenizer('\w+')


# Tokenizing the text
tokens = tokenizer.tokenize(text)

# Printing out the first 8 words / tokens
print(tokens[:8])
```

```
['Moby', 'Dick', 'Or', 'the', 'Whale', 'by', 'Herman', 'Melville']
```

In [97]:

```
%%nose

import nltk

def test_correct_tokenizer():
    correct_tokenizer = nltk.tokenize.RegexpTokenizer('\w+')
    assert isinstance(tokenizer, nltk.tokenize.regexp.RegexpTokenizer), \
    'tokenizer should be created using the function nltk.tokenize.RegexpTokenizer .'

def test_correct_tokens():
    correct_tokenizer = nltk.tokenize.RegexpTokenizer('\w+')
    correct_tokens = correct_tokenizer.tokenize(text)
    assert isinstance(tokens, list) and len(tokens) > 150000 , \
    'tokens should be a list with the words in text'
```

Out[97]:

2/2 tests passed

5. Make the words lowercase

OK! We're nearly there. Note that in the above 'Or' has a capital 'O' and that in other places it may not, but both 'Or' and 'or' should be counted as the same word. For this reason, we should build a list of all words in *Moby Dick* in which all capital letters have been made lower case.

In [98]:

```
# A new list to hold the lowercased words
words = []

# Looping through the tokens and make them lower case
for word in tokens:
    words.append(word.lower())

# Printing out the first 8 words / tokens
print(words[:8])
```

['moby', 'dick', 'or', 'the', 'whale', 'by', 'herman', 'melville']

In [99]:

```
%%nose

correct_words = []
for word in tokens:
    correct_words.append(word.lower())

def test_correct_words():
    assert correct_words == words, \
    'words should contain every element in tokens, but lowercased.'
```

Out[99]:

1/1 tests passed

6. Load in stop words

It is common practice to remove words that appear a lot in the English language such as 'the', 'of' and 'a' because they're not so interesting. Such words are known as *stop words*. The package `nltk` includes a good list of stop words in English that we can use.

In [100]:

```
# Getting the English stop words from nltk
#nltk.download('stopwords')
sw = nltk.corpus.stopwords.words('english')

# Printing out the first eight stop words
print(sw[:8])
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves']
```

In [101]:

```
%%nose

def test_correct_sw():
    correct_sw = nltk.corpus.stopwords.words('english')
    assert correct_sw == sw, \
    'sw should contain the stop words from nltk.'
```

Out[101]:

```
1/1 tests passed
```

7. Remove stop words in Moby Dick

We now want to create a new list with all `words` in Moby Dick, except those that are stop words (that is, those words listed in `sw`). One way to get this list is to loop over all elements of `words` and add each word to a new list if they are *not* in `sw`.

In [102]:

```
# A new list to hold Moby Dick with No Stop words
words_ns = []

# Appending to words_ns all words that are in words but not in sw
for word in words:
    if word not in sw:
        words_ns.append(word)

# Printing the first 5 words_ns to check that stop words are gone
print(words_ns[:5])
```

```
['moby', 'dick', 'whale', 'herman', 'melville']
```

In [103]:

```
%%nose

def test_correct_words_ns():
    correct_words_ns = []
    for word in words:
        if word not in sw:
            correct_words_ns.append(word)
    assert correct_words_ns == words_ns, \
    'words_ns should contain all words of Moby Dick but with the stop words removed.'
```

Out[103]:

```
1/1 tests passed
```

8. We have the answer

Our original question was:

> What are the most frequent words in Herman Melville's novel Moby Dick and how often do they occur?

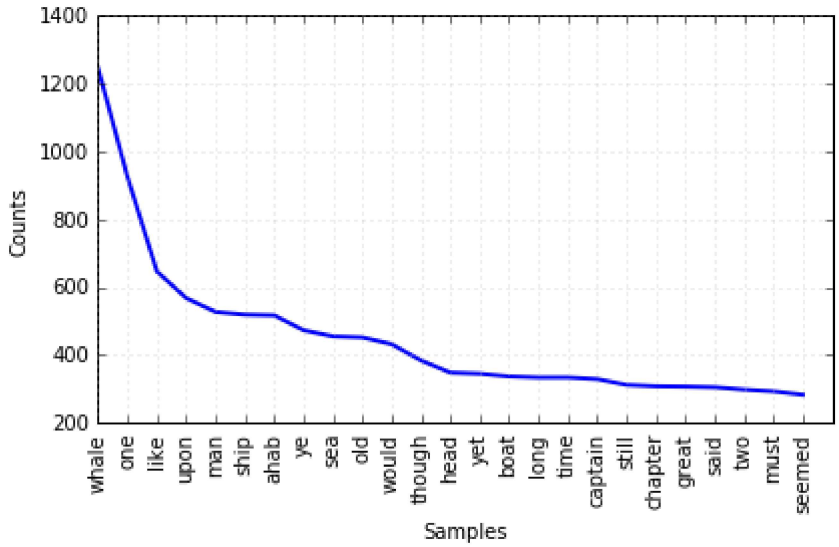
We are now ready to answer that! Let's create a word frequency distribution plot using `nltk`.

In [104]:

```
# This command display figures inline
%matplotlib inline

# Creating the word frequency distribution
freqdist = nltk.FreqDist(words_ns)

# Plotting the word frequency distribution
freqdist.plot(25)
```



In [105]:

```
%%nose

def test_correct_freqdist():
    correct_freqdist = nltk.FreqDist(words_ns)
    assert correct_freqdist == freqdist, \
    'freqdist should contain the word frequencies of words_ns.'
```

Out[105]:

```
1/1 tests passed
```

9. The most common word

Nice! The frequency distribution plot above is the answer to our question.

The natural language processing skills we used in this notebook are also applicable to much of the data that Data Scientists encounter as the vast proportion of the world's data is unstructured data and includes a great deal of text.

So, what word turned out to (*not surprisingly*) be the most common word in Moby Dick?

In [106]:

```
# What's the most common word in Moby Dick?
most_common_word = 'whale'
```

In [107]:

```
%%nose

def test_most_common_word():
    assert most_common_word.lower() == 'whale', \
    "That's not the most common word in moby dick."
```

Out[107]:

```
1/1 tests passed
```