Text and Words:

- Text is something we are familiar with, as we read and write everyday.
- In NLP, text is treated as raw data for the programs written.

Getting started with python:

- Python allows you to type directly into the interactive **interpreter.**
- You can access the Python interpreter using a simple graphical interface called Interactive Development Environment (IDLE).
- Under Unix you can run python from shell by typing **idle**.
    - If idle is not installed, then install via,
    - **sudo apt install idle**
- You can also run python from shell by typing **python.**

Getting Started with NLTK:

- **Installation:**
    - Before beginning, we need to install NLTK.
    - For installation visit: install_NLTK.
    - Download the required version for your platform via the given link.
- **Install required NLTK data:**
    - Once NLTK is installed, install the required NLTK data via the following instruction.
        - ***import nltk***- import the nltk.
        - ***nltk.download()***- browse the available packages.
        - Finally, select the line based on the requirements. Example: If you want to install a ***book*** module then select ***book*** and click ***download*.**
        - **Import** via- ***from nltk.book import ****
- **Loading BOOK:**
    - Load some texts to explore:
        - **from nltk.book import ***

```
>>> from nltk.book import *
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
>>>
```

- For knowing about these text, simply type for example **text1** at the python prompt

Searching Text:

- **concordance**
  - There are many ways to examine the context of a text apart from simply reading it.
  - Concordance view shows us every occurrence of a given word, together with some context.
  - **Syntax**: *text.**concordance**("desired_word")*

```
>>> text1.concordance("monstrous")
Displaying 11 of 11 matches:
ong the former , one was of a most monstrous size . ... This came towards us ,
ON OF THE PSALMS . " Touching that monstrous bulk of the whale or ork we have r
ll over with a heathenish array of monstrous clubs and spears . Some were thick
d as you gazed , and wondered what monstrous cannibal and savage could ever hav
that has survived the flood ; most monstrous and most mountainous ! That Himmal
they might scout at Moby Dick as a monstrous fable , or still worse and more de
th of Radney .'" CHAPTER 55 Of the monstrous Pictures of Whales . I shall ere l
ing Scenes . In connexion with the monstrous pictures of whales , I am strongly
ere to enter upon those still more monstrous stories of them which are to be fo
ght have been rummaged out of this monstrous cabinet there is no telling . But
of Whale - Bones ; for Whales of a monstrous size are oftentimes cast up dead u
>>>
```

  - **Thus,** concordance permits us to see words in context.
- **similar**
  - Concordance helps to see words in context.
  - What other words appear in a similar range of contexts?
  - We can find out by appending the term similar to the name of the text in question, then inserting the relevant word in parentheses:
    - **Syntax**: *Text.**similar**(word, num=20)*
      - *word(str)- The word used to seed the similarity search*
      - *num(int)- The number of words to generate (default=20)*

```
>>> text1.similar("monstrous")
mean part maddens doleful gamesome subtly uncommon careful untoward
exasperate loving passing mouldy christian few true mystifying
imperial modifies contemptible
>>> text2.similar("monstrous")
very heartily so exceedingly remarkably as vast a great amazingly
extremely good sweet
>>>
```

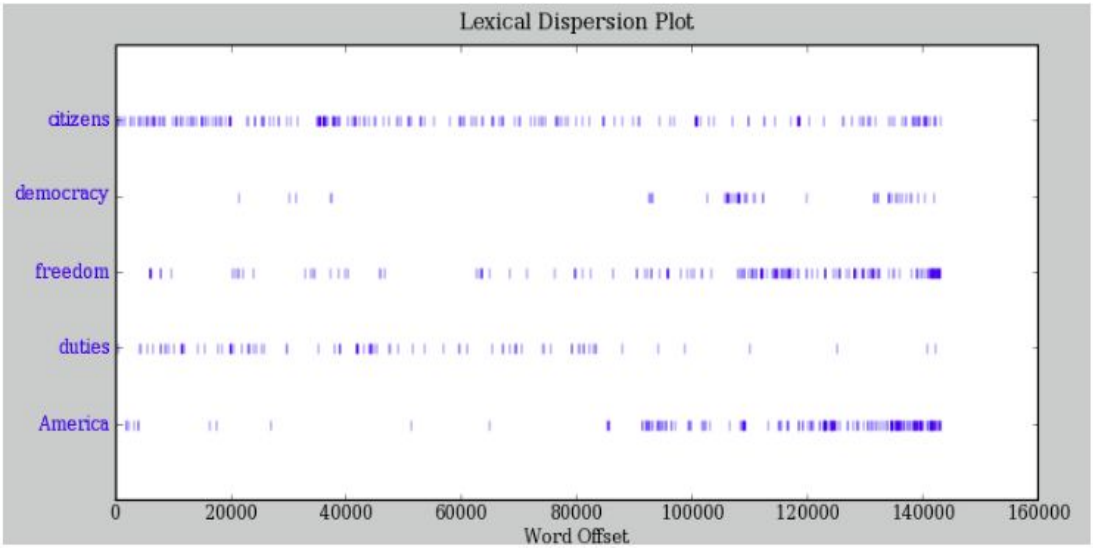
- **Common_contexts**

- Allows us to examine just the contexts that are shared by two or more words.
- list most frequent contexts first.

```
>>> text2.common_contexts(["monstrous", "very"])
a_pretty is_pretty am_glad be_glad a_lucky
>>>
```

- **Syntax:** *Text.**common_contexts**([word1, word2], num=20)*
  - *[word1, word2](str)- list of word to find common contexts*
  - *num(int)- number of common_contexts to display (default=20)*

- **dispersion_plot**
  - We can determine the location of words in the text i.e. how many words from the beginning it appears. It looks like a barcode.
  - It shows the spread of any particular word across the whole text.
  - In Dispersion plot x-axis represents the “narrative time”- measured by the number of words in Text.
  - When the desired word appears in the text a black vertical line is plotted, otherwise it remains blank (white line).
  - **Syntax:***Text.**dispersion_plot**([“word1”, “word2”, “word3”])*



Counting Vocabulary:

- Motivation is to use a computer to count the number of words in a text in a variety of useful ways.
- **Find length of text:**
  - Calculating length of text from start to finish.

- **len()** function is used to calculate length of given text.
- While calculating length of given text, it takes account into words and punctuation symbols that appear.
- Syntax: **len(***Text***)**

- **Token:**
  - A **token** is the technical name for a sequence of characters --such as hairy, his, or, :)-- that is treated as a group.
- **Distinct words:**
  - How many distinct words does the given text contain?
  - Distinct words in text are just the set of tokens, as in set there are no duplicate items.
  - **Syntax: sorted**(**set**(*Text*))
  - Gives a sorted list of vocabulary items, beginning with various punctuation symbols and continuing with words starting with A.
- **Lexical richness of Text:**
  - Lexical richness is about the quality of vocabulary in a language sample.
  - **Lexical richness** = length_of_distinct_words/length_of_text.
  - **Syntax: len**(**set**(*text*))/**len**(*text*).
  - **Note:** Higher the number of distinct words in given text, higher the lexical richness will be.
  - Lexical richness and Lexical diversity is quite similar.
  - **Example:** Lexical diversity of various Genres in the *Brown Corpus*

Genre	Tokens	Types	Lexical diversity
skill and hobbies	82345	11935	0.145
humor	21695	5017	0.231
fiction: science	14470	3233	0.223
press: reportage	100554	14394	0.143
fiction: romance	70022	8452	0.121
religion	39399	6373	0.162

```
def lexical_diversity(text):
    '''
    calculates lexical richness of text

    Arguments:
    text: text whose lexical richness to be calculated

    Returns:
    lexical_richness: calculates lexical_richness of  text
    '''
    lexical_richness = len(set(text))/len(text)
    return lexical_richness
```

- **Word count in Text:**
  - Counting the number of times the given word occur in the given text
  - **Syntax**: *Text*.**count***(word)*

```
#count word "smote" in text3
smote_count = text3.count("smote")
print(f"The word 'smote' occured {smote_count} times in 'text3'")

#percentage of "text5" taken by the word "lol"
percent_lol_in_text5 = text5.count("lol")/len(text5)
print(f"\npercentage of word 'lol' in 'text5' is {percent_lol_in_text5}")
```

```
The word 'smote' occured 5 times in 'text3'

percentage of word 'lol' in 'text5' is 0.015640968673628082
```

Text as Lists of Words

- In context of NLP Text is nothing more than a sequence of words and punctuation.
- **1. Lists and basic operations with list:**

```
#define list
sentence_1  = ['call', 'me', 'Ishmael', '.']
sentence_1
```

```
['call', 'me', 'Ishmael', '.']
```

```
#calculate lexical richness
print(f"lexical richness is: {lexical_diversity(sentence_1)}")
```

```
lexical richness is: 1.0
```

```
#some basic operations
ex1 = ['Monty', 'Python', 'and', 'the', 'Holy', 'Grail']

#sort list, ascending
print(sorted(ex1))

#find length of distinct items of list
print(len(set(ex1)))

#count the word **the** in list
print(ex1.count('the'))

#adding two lists
print(sent1+sent2)

#appendind item to list
print(f"\nlist before appending items:{sent1}")
sent1.append("Added")
print(f"\nlist after appending items:{sent1}")
```

```
['Grail', 'Holy', 'Monty', 'Python', 'and', 'the']
6
1
['Call', 'me', 'Ishmael', '.', 'The', 'family', 'of', 'Dashwood', 'had', 'long', 'been', 'settled', 'in', 'Susse
x', '.']

list before appending items:['Call', 'me', 'Ishmael', '.']

list after appending items:['Call', 'me', 'Ishmael', '.', 'Added']
```

- **Indexing lists:**
  - Indexes are a common way to access the words of text, generally, the elements of any text.
  - Extracting list items with the help of indexes associated with those items in lists.
  - **Code Snippet:**

```
#extract item with index 173 from text4
print(f"item with index 173 from text4 is: {text4[173]}")

#find index of word "awaken" from text4
print(f"index of word 'awaken' from text4 is: {text4.index('awaken')}")
```

```
item with index 173 from text4 is: awaken
index of word 'awaken' from text4 is: 173
```

- **Slicing lists:**
  - Python permits us to access sublists as well.
  - Extracting manageable pieces of language from large texts, a technique known as slicing.
  - Visit for more: https://www.pythoncentral.io/how-to-slice-listsarrays-and-tuples-in-python/

```
#slicing text5

#slicing with start and end index
print(text5[100:110])

#slicing with only end index
print(text5[:3])

#slicing with only start index
# print(text5[10000:])

#slicing with negative index
print(text5[-3:]) #starting from 3rd index from last will print further
```

```
['my', 'cousin', 'drew', 'a', 'messed', 'up', 'pic', 'on', 'my', 'cast']
['now', 'im', 'left']
['U98', 'Uh', '.']
```

- **2. String and Basic Operation**
  - Some methods used with lists also work with individual words, or strings.

- index string, slice string are possible as in list.
- **visit for more**: https://www.w3schools.com/python/python_strings.asp

```
#basic operation
name = 'Monty'

#access first character
print(name[0])

#slice upto 4 character
print(name[:4])
```

```
M
Mont
```

```
#multiplication with strings
print(f"Multiplication with strings: {name*2}")

#addition with strings
string_add = name+'!'
print(f"Addition with strings: {string_add}")
```

```
Multiplication with strings: MontyMonty
Addition with strings: Monty!
```

```
#join words of list to make single string
sample_list = ['Monty', 'Python']
print("join list to string:", ' '.join(sample_list)) #join by space

#Split string to list
sample_string = "Monty Python"
print("split string to list:", sample_string.split()) #split string by white space
```

```
join list to string: Monty Python
split string to list: ['Monty', 'Python']
```

Frequency Distribution From Text

- **How can we automatically identify the words of a text that are most informative about the topic and genre?**
- Frequency Distribution tells us the frequency of each vocabulary item in the text.
- It is a "distribution" because it tells us how the total number of word tokens in the text are distributed across the vocabulary items.
- Frequency Distribution is often needed in language processing, NLTK provides built-in support for them.
- Let's see **FreqDist,**

```
#Extracting frequency distribution from text
fdist = FreqDist(text1)
```

```
<FreqDist with 19317 samples and 260819 outcomes>
```

```
#extract frequency distribution of word "that"
fdist["that"]
```

```
2982
```

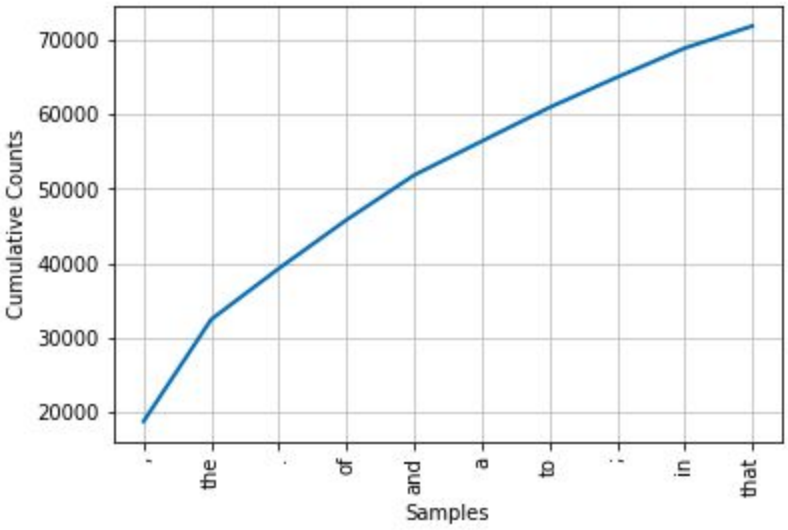
```
#find number of unique word in given text or corpus
len(fdist)
```

```
19317
```

```
#find most common words in given text
fdist.most_common(50) #50 most common words in text1
```

```
[(',', 18713),
 ('the', 13721),
 ('.', 6862),
 ('of', 6536),
 ('and', 6024),
 ('a', 4569),
 ('to', 4542),
 (';', 4072),
 ('in', 3916),
 ('that', 2982),
 ("'", 2684),
 ('-', 2552),
 ('his', 2459),
 ('it', 2209),
 ('I', 2124),
 ('s', 1739),
 ('is', 1695),
 ('he', 1661),
```

- **Cumulative frequency plot for most common 50 words**



- **Hapaxes**
  - Gives us the rare words in the given text.
  - As we see from above, frequent words do not give genre, or what the text or book is about.
  - Maybe, hapaxes i.e. infrequent words gives more sense on that.

```
In [73]: #list infrequent words in given text
         fdist.hapaxes()
```

```
Out[73]: ['Herman',
          'Melville',
          ']',
          'ETYMOLOGY',
          'Late',
          'Consumptive',
          'School',
          'threadbare',
          'lexicons',
          'mockingly',
          'flags',
          'mortality',
          'signification',
          'HACKLUYT',
```

Frequency Distribution From Text

- previously, frequent words and infrequent words failed to give proper meaning to text.
- lets, take a look at the long words of a text; perhaps these will be more characteristic and informative.
- For this, let's find the vocabulary of the text that is more than 15 characters long.

```
#extracting unique words of length greater than 15, from text1
unique_words = set(text1)
long_words = [word for word in unique_words if len(word)>15]
sorted(long_words)
```

```
['CIRCUMNAVIGATION',
 'Physiognomically',
 'apprehensiveness',
 'cannibalistically',
 'characteristically',
 'circumnavigating',
 'circumnavigation',
 'circumnavigations',
 'comprehensiveness',
 'hermaphroditical',
 'indiscriminately',
```

Collocations and Bigrams

- A **collocation** is a sequence of words that occur together unusually often, Thus red wine is a collocation, whereas the wine is not.
- Also, collocations are phrases or expressions that are highly likely to co-occur.
- We frequently extract words from the text.
- Example: good film, las vegas, new york and so-on

How collocations different from Bigrams

- Bigrams means the set of two words(e.g. this is, red wine, is said and so-on) that co-occur, while trigrams means a set of three words that co-occur.
- Bigrams or Trigrams may not give us meaning phrases.
- Example:
    - consider sentence: "He applied machine learning" contains biagrams i.e.
    - Biagrams: 'He applied', 'applied machine', 'machine learning'.
    - 'He applied' and 'applied machine' is not meaningful, while 'machine learning' is a meaningful biagrams.

- Thus, just considering co-occurring words may not be a good idea, since phrases such as 'of the' may occur frequently, but are actually not meaningful.
- Thus the need for collocations from NLTK library. It only gives us the meaningful bigrams and trigrams.

How is one collocation better than the other?

- Pointwise Mutual Information or PMI score is used.
- PMI score is used to rank the bigrams and trigrams churned out by collocations library.
    - PMI(a, b) = log(p(a, b)/(p(a) * p(b)))

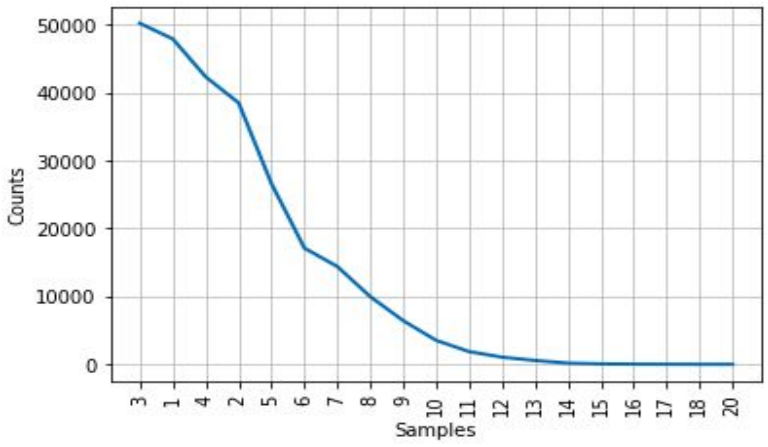
Counting Other Things Than Words

- Counting words is useful, but we can count other things too.
- Example: We can look at the distribution of word lengths in text, shown below,

```
#Distribution of word lengths in text1

#make list of length of all words
word_length_list = [len(word) for word in text1]

#generate frequency distribution using FreqDist
fdist = FreqDist(word_length_list)
print(fdist)

#show frequency  distribution
fdist
```

```
<FreqDist with 19 samples and 260819 outcomes>

FreqDist({3: 50223, 1: 47933, 4: 42345, 2: 38513, 5: 26597, 6: 17111,
```

- Also, plot frequency distribution of length of words from text1 of nltk.book



Frequency Distribution Summary:

Example	Description
`fdist = FreqDist(samples)`	create a frequency distribution containing the given samples
`fdist[sample] += 1`	increment the count for this sample
`fdist['monstrous']`	count of the number of times a given sample occurred
`fdist.freq('monstrous')`	frequency of a given sample
`fdist.N()`	total number of samples
`fdist.most_common(n)`	the `n` most common samples and their frequencies
`for sample in fdist:`	iterate over the samples
`fdist.max()`	sample with the greatest count
`fdist.tabulate()`	tabulate the frequency distribution
`fdist.plot()`	graphical plot of the frequency distribution
`fdist.plot(cumulative=True)`	cumulative plot of the frequency distribution
`fdist1 |= fdist2`	update `fdist1` with counts from `fdist2`
`fdist1 < fdist2`	test if samples in `fdist1` occur less frequently than in `fdist2`

Making Decisions and Taking Control:

- A key feature of programming is the ability of machines to make decisions on our behalf such as:
    - Executing instruction when certain conditions are met,
    - Repeatedly looping through text data until some condition is satisfied.
- This feature is called control
- **1.Conditionals:**
    - **Numerical Comparison Operators**
        - Python supports a wide range of operators, such as < and >=, for testing the relationship between values
        - The full set of these relational operators is shown:
            - < less than
            - <= less than or equal to
            - == equal to (note this is two "=" signs, not one)
            - != not equal to
            - greater than
            - greater than or equal to
    - Demonstration is as shown below,

```
#less than
print([w for w in sent7 if len(w) < 4])

#less than or equal to
print([w for w in sent7 if len(w) <= 4])

#equal to
print([w for w in sent7 if len(w) == 4])

#not equal to
print([w for w in sent7 if len(w) != 4])
```

- **Word Comparison Operators:**
  - Helps to select words from our texts.
  - Some word comparisons operators are listed below
    - **s.startswith(t):** test if s starts with t
    - **s.endswith(t):** test if s ends with t
    - **t in s:** test if t is a substring of s
    - **s.islower():** test if s contains cased characters and all are lowercase
    - **s.isupper():** test if s contains cased characters and all are uppercase
    - **s.isalpha():** test if s is non-empty and all characters in s are alphabetic
    - **s.isalnum():** test if s is non-empty and all characters in s are alphanumeric
    - **s.isdigit():** test if s is non-empty and all characters in s are digits
    - **s.istitle():** test if s contains cased characters and is titlecased (i.e. all words in s have initial capitals)