Natural Language Processing Fundamentals

Table of Contents

- Read: Using Python and Jupyter Notebooks in This Course
- Tool: Writing and Navigating Jupyter Notebooks

Module Introduction: Find Simple Patterns in a Body of Text

- Watch: Defining Natural Language Processing
- Watch: NLP Terminology
- Watch: String Manipulation Basics in Python
- Code: Practice String Manipulation Basics in Python
- Watch: Preprocess Substrings With Operations
- Code: Practice Preprocessing Substrings With Operations
- Code: More Methods for Joining, Splitting, and Replacing Strings
- Watch: Count Substrings
- Code: Practice Counting Substrings
- Tool: String Manipulation Methods
- Watch: Overview of Regular Expressions
- Code: Practice Using Simple Regular Expressions
- Watch: Use Regular Expressions to Find Patterns
- Code: Practice Using Regular Expressions to Find Patterns
- Watch: Parsing Strings With Regular Expressions
- Tool: Regular Expressions Reference Guide
- Code: Practice Parsing Strings With Regular Expressions
- Quiz: Regular Expressions Basics
- Tool: Libraries and Methods for Finding Simple Patterns in Text
- Assignment: Course Project, Part One — Finding Simple Patterns in a Body of Text
- Module Wrap-up: Find Simple Patterns in a Body of Text

Module Introduction: Apply Preprocessing Techniques to Reduce a Text's Vocabulary

- Watch: The Goals of Preprocessing Text
- Watch: Text Tokenization
- Tool: NLTK Library
- Code: Parsing a Document into Tokens
- Quiz: Defining Tokenization Methods
- Watch: Working with Characters
- Code: Work with Characters to Standardize a Vocabulary
- Watch: Expanding Contractions
- Code: Modify and Apply a Contraction Map
- Watch: Text Correction
- Code: Perform Text Correction on a Document
- Watch: Spelling Correction with TextBlob
- Code: Generate Probabilities for Spelling Correction on a Document
- Watch: Stemming and Lemmatization
- Code: Stem and Lemmatize a Document to Measure Vocabulary Quality
- Watch: Removing Stopwords
- Code: Remove Stopwords From a Document
- Read: Suggested Reading: Removing HTML Tags in Preprocessing
- Tool: Choosing Preprocessing Techniques
- Quiz: Identifying the Appropriate Preprocessing Technique
- Tool: Libraries and Methods for Preprocessing Text to Reduce Vocabulary
- Assignment: Course Project, Part Two — Preprocess Text to Reduce Vocabulary
- Module Wrap-up: Preprocess Text to Reduce Vocabulary

Module Introduction: Tag and Parse a Document

- Watch: Parts-of-Speech Tagging
- Code: Tag a Sentence
- Read: Identifying Parts of Speech and Phrases
- Watch: Parsing Techniques: Shallow Parsing
- Code: Use Shallow Parsing on a Sentence
- Read: Using Tree Graphs to Understand Relationships

- Watch: Parsing Techniques: Dependency Parsing
- Code: Use Dependency Parsing on a Sentence
- Read: Parsing Techniques: Constituency Parsing
- Code: Use Constituency Parsing on a Sentence
- Quiz: Identifying Parsing Types
- Tool: Libraries and Methods for Tagging and Parsing a Document
- Assignment: Course Project, Part Three — Tagging and Parsing a Document
- Tool: Next Steps — Practice Exercises and Resources
- Module Wrap-up: Module Wrap-up
- Glossary
- Discussion: Project Forum

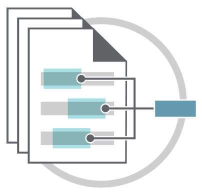
Homepage

Video Transcript

Picture yourself in front of the PC reviewing your emails. You can easily recognize a few spam messages, But as the volume increases to thousands or millions, we need help processing that amount of information. We need more sophisticated automation because we are in the age of an abundance of textual information with billions of emails, queries, web pages, chats, transcripts, tweets, millions of books, products, pages and medical records. Why is Python a popular platform for NLP? Well, it is easy to get started with and use because of its friendly syntax, easily accessible to anyone with a laptop or even just a web browser. It is also backed by a sea of NLP libraries and millions of contributors. It contains cutting-edge tools, which we will study in this course. There is a lot of opportunity here. Research and code base are being democratized via free online publishing services where anyone can try and contribute to new NLP tools. In this course, we will with with several smart algorithms that allow large scale automation of human decisions based on many kinds of inputs. So let's get started.

What you'll do

- **Divide a document into words and use regular expressions to find simple patterns**
- **Preprocess text in order to reduce a document's vocabulary and make your analysis more computationally efficient**
- **Tag and parse sentences in a document in order to relate words and phrases to one another**



Course Description

In recent years, using spellcheck on phones, grammar check in emails, and even autocomplete when sending messages has become normal. It's also possible to translate from one language to another very quickly using free, web-based tools, or even address small customer service problems by interacting not with a person, but a bot programmed to communicate like one. All of these now-common features rely on techniques from the field of natural language processing, or NLP for short.

But how can you get get those features? Where should you start if you want to end up with a chatbot, autocomplete, or autocorrect? Computers can't understand English, but they can work with patterns that you teach them, as long as you prepare (or preprocess) text properly. In this course, you will apply classical NLP techniques to text in order to identify patterns in a body of text, and then reduce the overall size of that text's vocabulary in order to make your task more computationally efficient; this efficiency is vitally important in NLP, as you will frequently be working with text at scale, meaning that there will be millions of documents being studied. Finally, you will tag and parse sentences, as this is another important preprocessing step that will allow various machine learning algorithms to be able to relate words and phrases to one another.

System requirements: This course contains a virtual programming environment that does not support the use of Safari, tablets, or mobile devices. Please use Chrome, Firefox, or Edge for this course. Refer to eCornell's **Technical Requirements** for up-to-date information on supported browsers.

Oleg Melnikov, Ph.D.
Visiting Lecturer
Cornell Bowers College of Computing and Information Science, Cornell University

Oleg Melnikov received his Ph.D. in Statistics from Rice University, advised by Dr. Katherine Ensor on the thesis topic of non-negative matrix factorization (NMF) applied to time series. He currently leads a Data Science team at ShareThis Inc. in Palo Alto, CA, and has decades of experience in mathematical and statistical research, teaching, databases and software development, finance (portfolio management and security analysis), AdTech, and other fields.

Professor Melnikov's academic path started with a Bachelor of Science in Computer Science (and some years in pre-med). Now, he holds Masters' degrees in computer science with a specialization in machine learning from the Georgia Institute of Technology; mathematics from the University of California, Irvine (where he was in the doctoral program); and statistics from Rice University. He also holds an MBA from the University of California, Los Angeles and an MFE-equivalent degree in quantitative finance. Passionate about education and the hard sciences, Professor Melnikov has taught statistics, machine learning, data science, quantitative finance, and programming courses at eCornell, Stanford University, UC Berkeley, Rice University, and UC Irvine.

Read: Using Python and Jupyter Notebooks in This Course

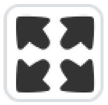
Throughout the course, there will be opportunities for you to implement natural language processing (NLP) concepts from Professor Melnikov's videos. Python is the programming language used in this course, and you will learn how to perform NLP-related tasks using several of its popular libraries and tools.

You will interact with Python throughout this course by writing and running code in workspaces hosted on Jupyter Notebooks. Our cloud-based implementation of Python and Jupyter Notebooks means that you can complete this course without installing Python on your machine.

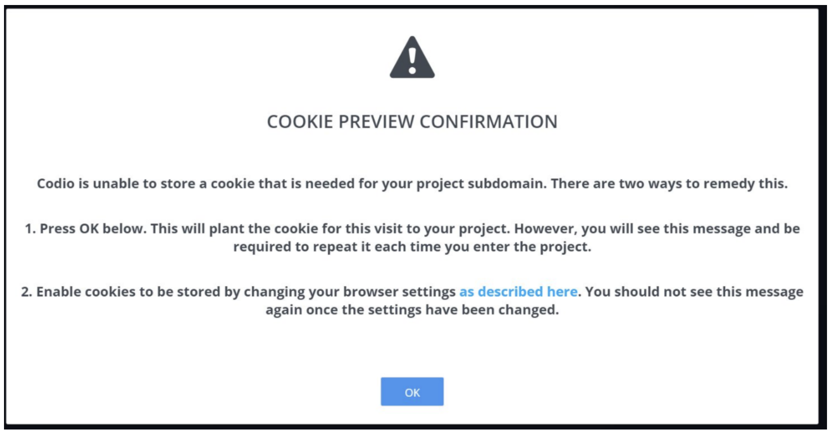
About Your Workspace

Each workspace in which you write Python code is distinct from every other workspace in the course; this means that the code you develop on one page of the course *will not carry over* to another page. Each workspace *is* persistent, however, so if you save some work and log out of the course, you can return to that page later and pick up where you left off in your previous session.

If you need more room in which to work, you can make your workspace fullscreen by clicking the fullscreen button in the bottom-right corner of the workspace. The image below is an example of what the button looks like.



Sometimes pop-up messages can occur when you work in this workspace. The below pop-up notifies you of a cookie requirement by the website that hosts Jupyter Notebooks, documents in which you will write code.



COOKIE PREVIEW CONFIRMATION

Codio is unable to store a cookie that is needed for your project subdomain. There are two ways to remedy this.

1. Press OK below. This will plant the cookie for this visit to your project. However, you will see this message and be required to repeat it each time you enter the project.

2. Enable cookies to be stored by changing your browser settings as described here. You should not see this message again once the settings have been changed.

Follow the instructions to resolve this issue and continue.

Jupyter Notebooks in This Course

The activities and projects use Jupyter Notebooks, which are web-based interactive documents that can weave code, data visualizations, outputs, equations, and text together in a single document. Each notebook within the course is hosted on a separate, cloud-based virtual machine (a.k.a. Codio Unit) that has Python and the necessary packages/dependencies already pre-installed.

The notebook activities throughout the course have been structured for you to:

1. **Setup** your workspace by importing necessary Python libraries and data.
2. **Review** coding snippets and concepts demonstrated in the videos.
3. **Practice** tackling related tasks, which are ungraded.

At the end of each module, you will demonstrate the skills that you have acquired in the module by completing one part of the graded **Course Project**.

Best Practices

- Do all of your work in the Jupyter Notebook workspace.
- Comment code to explain what is happening or demonstrate that you understand what is happening, and cite any sources other than documentation and class material used to assist in an exercise.
- Create test cells and perform tests outside of the graded function block, and leave this content in place to show your progression and process.

Back to Table of Contents

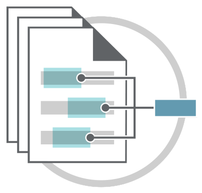
Tool: Writing and Navigating Jupyter Notebooks

Throughout this course and certificate, you will write and run Python scripts via Jupyter Notebooks. Use the following tool to familiarize yourself with Jupyter Notebooks or to refresh your memory for how they function. You can also save it for future reference.

Please view and download this tool in the course.

Back to Table of Contents

Module Introduction: Find Simple Patterns in a Body of Text



Computers might not be able to understand English (or any other natural language) but they can be taught how to recognize *patterns* in a natural language. Of course, for computers to learn these patterns, you have to teach them, and that involves starting with the basics: where does a word start or end? Where does a sentence start or end? How do you separate words with punctuation? What are the differences between *can*, *can not,* and *can't*?

In this module, you will become familiar with some basic vocabulary used in natural language processing (NLP), and perform several preprocessing and string manipulation techniques on single strings. You will also practice these techniques on larger documents to start to work with the power of NLP at scale. These foundational skills will prepare you to perform more complex string and document preprocessing techniques.

This module will require approximately *12 hours* to complete.

Connect with your peers — **Visit the Student Lounge**

Back to Table of Contents

Watch: Defining Natural Language Processing

Natural Language Processing (NLP) is broadly defined as the manipulation of a natural language. Typically, NLP is used *at scale*, in situations where you would be working with hundreds of books, thousands of emails, or millions of product descriptions at once. In this video, Professor Melnikov distinguishes natural languages from constructed languages and discusses how NLP can be used to work with natural languages. He also lists several NLP tools and outlines some reasons a business might use NLP.

Video Transcript

So what is natural language? It is a form of written and verbal communication evolved naturally by humans through use. Some examples include languages like English, Russian, Japanese, Hindi, Chinese, and many dialects of these languages, as well as emojis and memes.

On the other hand, we have constructed languages. These are purposefully developed by humans to have a particular grammar and vocabulary. Some examples of this include programming languages like Python and R; structured query languages, SQL. Web search queries are also in that category. Some of the spoken constructed languages are Esperanto and others. These are developed to communicate similar to natural languages, but they are specifically constructed.

We use natural language processing to standardize and analyze text at scale. This includes cleaning and preprocessing text to reduce noisy typos, removing unimportant words and phrases, parsing text into meaningful words and sentences. It also includes converting text to numeric representations so we can perform statistical modeling on text more easily. It also includes computing a degree of association among documents to measure their similarity or dissimilarity. Drawing structural and semantic meaning from text and organizing documents to search them more efficiently.

What about NLP in practice? What use cases do we know? There are several use cases of NLP and those include querying contents of trillions of web pages and documents, such as HTML, text, PDF files, PowerPoint presentations. Recommendation systems use NLP for recommending products and services; for example, "If you liked this particular product, you might also like this one." Language translation can be done at scale. We don't need to hire a single interpreter to help us translate from Russian to Chinese. We can translate using different translators on the web. Spelling and grammar correction can be done in the tool in the editor that we use to type documents. Question-and-answering systems, chatbots, and automated customer support can be used to answer questions for millions of customers at the same time. Text summarization and compression is used to shrink down the documents of large-sized publications, books, into just a few words or sentences, and we can visualize or read them more efficiently. There's also product categorization; that is a hierarchical structure of relationships or taxonomies between different products. This allows us to relate something like a keyboard to, say, a webcam, which are not in the same category, per se, but they both relate to a laptop.

Back to Table of Contents

Watch: NLP Terminology

Language is hierarchical, and its most basic components, such as letters and punctuation, can be grouped to create words and then form increasingly complex documents. NLP can be used on documents of any size, but it is most often used on very large documents. For this reason, as Professor Melnikov explains here, one of the first steps of classical NLP is to take advantage of the hierarchical nature of language by using a process called *tokenization*. Tokenization, which is also known as parsing, splits a large document into more manageable *tokens*, which are smaller units of contiguous characters such as letters, words, or sentences.

Video Transcript

What are some basic elements of a natural language? Typically, the language starts with symbols that are hierarchically grouped to represent the largest structures. In English, we have 26 letters and some punctuation. This can be formed together into words like "one," "two," "three." More complex words can also include non-characters or non-letters, such as "Los Angeles" has a space. "Up-to-date" has hyphens or dashes in between. "5-hour" can be a single word as well with a number in it. The words can have spaces, hyphens, numbers, accent marks, and other characters. They are not just defined by letters. The 26 letters that we have can create a tremendously large number of words. Oxford Dictionary has 170,000, and there are many more short-lasting words that people create every single day through use in chat communities, for instance, or tweets, or memes, emojis, abbreviations create millions of words that live for a day.

Words are combined into millions of phrases, which are assembled into trillions of sentences, which are united into paragraphs, which are clumped into chapters and articles, which are aggregated into documents which are collected into corpus, which are clustered into corpora. By the way, a corpus is just a large set of text documents. In fact, we loosely use these terms and often overload the term "documents." "Document" can be a paragraph or a sentence in NLP.

Such hierarchical organization of textual elements is easier to learn, use, and maintain. All this structure is amazingly rich and practically infinite dimensional. Essentially, a document is just a long sequence of characters. Each element starts and ends with some identifier. A symbol might be just an uninterrupted stroke of a pen. There are some exceptions; there's the letter i, j, semi-colon, colon. But the words are identified by spaces or punctuation or tab characters, `\t` symbol. A sentence typically ends with a quotation mark or a period, a question mark, exclamation, ellipsis. Paragraphs are separated by invisible characters, such as newline character, `\n`, or a carriage return,

`\r`, which is typical for Microsoft Windows systems. Documents have headers and titles and so on. By the way, the invisible characters, such as blank spaces or `\n`, `\r`, `\t` that I mentioned earlier, are called whitespace and are identified with a `\s` character.

While there are algorithms that operate on long strings of characters, most classical NLP starts with tokenization. In this process, we split or parse the string into tokens, which are contiguous groups of characters. Commonly these are words and sentences, but they can also be sub-words, characters, phrase tokens, and paragraph tokens. For example, we could have a sentence, "NLP is fun. I like it a ton" tokenized into characters, or words, or a couple of sentences.

Back to Table of Contents

Watch: String Manipulation Basics in Python

Strings, called `str` objects in Python, are sequences or arrays of characters that have properties and methods. Many of the NLP techniques that can be used on large documents can also be performed on simple strings.

Libraries/Methods

Standard Library

- `print()`
- `str()`
- `list()`

Consequently, string manipulation is a great place to delve into NLP and can be as simple as defining and printing a string. Watch Professor Melnikov perform several basic string manipulation techniques, such as displaying, splitting, and subsetting, on some example strings.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

A string is a character sequence. It is stored as an object in Python. As any other object, a string has properties and methods that are used to interact with a string or act upon it. For instance, we can get the length of a string by executing a length method or attribute. Oftentimes these are used interchangeably. The attributes, otherwise called properties, are mainly to retrieve a particular state, a descriptive state information about a string. The methods, on the other hand, are more commonly associated with changing that state.

Let's say we want to lowercase a string or capitalize its casing. Let's look at the following code. First, we can define a string with different quotation marks. Single quotes or double quotes on either side of the string, will place it into a single variable as a sequence of characters. We can execute it and display what's contained in that character without even the `print()` statement. If we add a `print()` statement, some of the characters are removed, like the quotation marks are suppressed, and the sentence is coming out nicer and more readable.

There is a way to put in newline characters in the string and that's if you use three single quotes or double quotes on either side of the string. A multi-line string in this way can be spawned over multiple lines. We can `print()` it and it comes out as such. If you just display the contents of that document string, then you will have all the `\n` or the newline characters displayed in a single line with the quotation marks on both sides.

There's another way to create a string and that's with a string function that basically converts any object that is passed to it to a string object, like the string itself can be converted to a string or a number, or at least would be represented as a list of characters. We can also convert to `str(None)` or string object in itself could be converted to a string or a function.

If we apply a list function to a string, what it does, it splits the string into a list of characters where every element is a single character of that original string. This is sometimes useful. It is useful for iterating over these characters if we need to and when we need to. This list is now accessible or gives us a way to extract different characters.

But there's another way to extract characters out of a list and that's called slicing or subsetting. If you have a document or a string of characters, you can subset or slice characters, retrieve a substring out of that document just by indicating from which index you want to start — Python is a zero-based index language — and where you want to stop. Now when you stop from 0-3, 3 is not the index where the slicing stops, it stops one character before that. So three characters, N-L-P, N at zero index, L at the first index, and P at the second index will be retrieved. There are other ways to retrieve or slice substrings and that's by starting from the end with negative indexing or not indicating where you want to stop, having a blank ending index or a blank beginning index.

Back to Table of Contents

Code: Practice String Manipulation Basics in Python

In the previous video, Professor Melnikov introduced a few basic ways to manipulate strings. In the "Review" section of this ungraded coding exercise, you will use these string manipulation techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will use some of these techniques on different strings.

All practice exercises are optional and ungraded.

Please complete this activity in the course.

Back to Table of Contents

Watch: Preprocess Substrings With Operations

Libraries/Methods

Standard Library

`s.join()`, `s.lower()`, `len()`, `s.find()`, `s.startswith()`, `s.endswith()`, `s.replace()`, `s.strip()`, `s.split()`

Your brain is able to detect patterns in text, even when that text contains non-standard sentences or misspelled words. Computers, however, are unable to determine that two different patterns are the same, even when the person who wrote them intended for them to be the same. For example, a computer cannot tell that a word with different patterns of capitalization, like "can" and "cAn," is the same word. Therefore, an important part of preprocessing text is to standardize that text by making words with the same meaning appear in the same way to the computer. In this video, Professor Melnikov demonstrates a variety of operations you can use to preprocess strings.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

Now we'll review a few string operations that allow us to join strings together, change letter casing, search and replace substrings, strip leading and trailing characters, split a string into smaller substrings, and check string attributes such as length, capitalization, character type, whether it's a digit or a letter, and so on.

Let's look at the code. We have two strings: `sDoc1` and `sDoc2` that we would like to glue together. We can do it in one of two ways, either the algebraic operation with a plus sign where `sDoc1` and a space in between is glued with `sDoc2`. `sDoc1` and `sDoc2` can be placed in a list and joined together as an alternative with the space as a separator. `join` is an operation or a method of a string.

Another operation on a string that is very useful is `lower`. It's a method that will lowercase all the characters in a string. There's an alternative, `upper`, `capitalize`, and `title`, which will do similar type of character letter case changes, but the `lower` seems to be the preferred method in NLP. It really doesn't matter which one you use as long as you are standardizing all the characters the same way.

One important method of a string is the size or length, and it's a method that we do not see, there is a length function in Python that will actually call the method, but what what is returned is the number of characters in the string, and that's useful in identifying whether there are any problems with a particular sentence or particular word during tokenization. If your words are returned with size 200, that might be problematic and higher tokenization quality might be needed.

The string can be searched for a particular substring. There is a `find` method that will search the location of a phone in my phone, and there is a number string. Then we can offset that location, the position plus another seven characters and extract or slice out the remaining characters, which will be just the phone number. In case we need to find a phone number starting from a particular position and that position needs to be identified by a substring. The phone `in` the particular string, `in` is an operation of Python that will return true or false, whether that element "phone" is in the list — sorry — is in the string. Now, it's the whole substring searched within a string that returns the Boolean true or false. You can also do `startswith` or `endswith`, and those are actually methods of a string that will return whether the string starts or ends with a particular substring.

There is a string, "NLP is great," which you can see is not very clean. It has other characters that may clutter the meaning of a particular string. When we want to replace some characters like the newline character or "gr8" can be replaced as "great," that could be done with the replace method. What's returned is much cleaner, more readable, legible string: "NLP is great." There are a few methods that will provide the status of characters in a string. One of them `isupper`, which will verify whether all characters are uppercase. There is also `islower`, which will do the same and verify whether all characters are lowercase. Both of those will return true or false.

There are many others that we'll check for. Is the character or all characters alpha, is the character or all characters alphanumeric, and so on. There's a strip operation that is useful in standardizing strings, words, or sentences, and so on, by stripping leading and trailing characters. It's very simple to apply the method. You just call `strip` at the end of the string.

Another very helpful operation is `split`, and that will split on any desired character or a set of characters. By default, it splits on something called whitespace. The whitespace includes a tab, a newline, character, a carriage return character, and a space. You will get back a list of words or tokens that were separated by any one of those characters in the original string. Here's another example where we're splitting specifically on newline character, or a space, or a newline character with a space, and what you get back are three different lists with the remaining characters. Notice that the character we searched for and split on is taken out from the results and list of elements.

Back to Table of Contents

Code: Practice Preprocessing Substrings With Operations

In the previous video, Professor Melnikov introduced different ways of preprocessing text to standardize substrings. In the "Review" section of this ungraded coding exercise, you will put these techniques to use. In the "Optional Practice" section of this exercise, you will try some of these techniques to preprocess different strings of text.

All practice exercises are optional and ungraded.

Please complete this activity in the course.

Back to Table of Contents

Code: More Methods for Joining, Splitting, and Replacing Strings

The `join()`, `replace()`, and `split()` methods contain many techniques you will find helpful as you preprocess strings. The following Jupyter Notebook contains brief descriptions and examples of other methods that Professor Melnikov did not cover in the video, but that you will use as you work through this course.

All practice exercises are optional and ungraded.

Please complete this activity in the course.

Back to Table of Contents

Watch: Count Substrings

In NLP, counting the number of times a substring, such as a particular word or letter, occurs within a document can give you insight into the contents of that document.

Libraries/Methods

Standard Library

- `count()` method
- `Counter()` object

One way this can be useful is in the context of a search engine because it means documents can be categorized without a human reading them. Counts of substrings can also be used to perform statistics to answer particular questions about a document, or to compare a document to other documents. Here, Professor Melnikov demonstrates how to count the substrings within a string of DNA nucleotides, which are represented by letters that indicate the four bases that comprise DNA molecules: adenine (A), cytosine (C), guanine (G), and thymine (T).

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

Counting substrings is exceptionally useful in NLP because the distribution of words in a document correlates with its meaning. Thus, the book on sailing is likely to have many counts of "sea" and "ocean," while the article on astronomy is likely to contain many appearances of "planet," "telescope," and "alien." Well, maybe not "alien." But in this video, we'll learn a string method `count`, which returns the frequency of substring occurrences. We then investigate a popular object `counter` which returns distribution, basically a histogram of elements of a particular list and it doesn't necessarily need to be a list of characters. We also manipulate the count results with some arithmetic and logic functions, list comprehension, and a useful `zip` function.

Let's look at the code. Here we have a document or a string which we want to count the substrings of "ice," space, and the period. The `method` of a string will return the counts of how many times those substrings appear. Keep in mind that this needs to be matching on the case and so if you're counting for the lowercase "ice," only lowercase "ice" will return. So we have 6, 9 and 2 returned as a tuple from this string.

There is a more sophisticated way of counting. The `counter` object from the collections package will scan the full string and give you the full distribution of letters that are found in this string. The string in this case is treated as a list, but you can also search and count elements of a list if you want to.

So we have this as DNA, which is a set of nucleotides as AGTC and these nucleotides may relate to different viral strands or to different viral DNAs simply by the count of how many times these nucleotides appear in each of the viral strings. We return back or the `counter` returns back a dictionary-like object where G represents the key and 16, or the count how many times G was met in the string, is the value. So T is the key, 15 is the value. Every unique key will be associated with some sort of value. Value can be a number or a more complicated structure. Here it's just a count.

Once this dictionary-like object is returned, if we have two of this, we can do some operations on these `counter` objects. In particular, we can sum them together, subtract, or end operation. With the summation, the same keys will be combined together and their values will be added together. With the minus operation, the values will be subtracted for the same key. You can look at the key B, which would be subtracted. If you're subtracting a larger value, than the key disappears from the results in `counter` object. The `union` will find the maximum of the two values, so if your B is one, has 1 value in the first `counter` object and a value of 2 in the other, the 2 will be returned for the key B and the intersection is the minimum of the two values.

We can also, because it's a dictionary-like object, extract value for a particular key. Here we're extracting the count of value A from the `counter` object and the return value is 13. The `most_common` will basically recast or return different representation of the keys and values. Now we have a list of tuples where the first element of a tuple is that search string, or the string that we counted, which is G, T, C, or A and in the second position of each tuple, we have the count that is associated with that element. These are placed in decreasing order of their counts.

The `most_common` takes an argument which is a number of elements you want to return, keep in mind these are most frequent elements and it is sometimes convenient to repackage the keys together as a separate list, so we can do that with the list comprehension by extracting only the first element from each tuple and packaging it as a list of all these different elements, G, T, and C. Likewise, we can use the `zip` function and what `zip` does is it takes the first element of every tuple of a list and packages it as a single tuple, and then takes every second element of a tuple of a list and packages it as a second tuple, and those tuples are returned as a list. So you have your keys separately and you have your associated values separately as a different tuple.

Back to Table of Contents

Code: Practice Counting Substrings

In the previous video, Professor Melnikov introduced some methods you can use to count substrings. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will use some of these techniques to count substrings in new phrases.

All practice exercises are optional and ungraded.

Please complete this activity in the course.

Back to Table of Contents

Tool: String Manipulation Methods

Professor Melnikov has demonstrated several methods from the Standard Python Library that you can use to manipulate strings or substrings. Loosely, these techniques can be categorized into methods for case conversions, replacement, tests, and `split`, `join`, and `strip` methods of a string object. This tool summarizes these methods, and you can use it as a quick reference guide.

If you are interested in more complete documentation of string manipulation methods, including syntax and examples, see **the official Python documentation**.

Please view and download this tool in the course.

Back to Table of Contents

Watch: Overview of Regular Expressions

Libraries/Methods

re Library

- `search()`
- `sub()`
- `split()`
- `findall()`

A regular expression, also known as a *regex (*plural: *regexes)*, is a powerful search pattern that allows you to find and replace multiple patterns at once. This is extremely useful because it is computationally efficient — your search only needs to make a single pass through a given document. This power comes with a tradeoff, however, in that it can be tricky to learn and implement regexes. Once you understand the syntax of a particular regex, though, you can use it across many different programming languages. Here, Professor Melnikov demonstrates four common regex search methods.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

Regular expressions or regex offer powerful find and replace methods for a string. In regex, we define search pattern, and this search pattern is lightning fast. It requires only a single pass for a document. Know that each pass is an expensive operation that we'll want to minimize. For example, when you search for A and B, you want to replace them with a C in a string, you can use a replace method to do that, but you would have to use it twice. You would first have to replace A with C, and second time with a replace method to replace B with C as well. With the regex operation substitute or `sub()`, you can do a pattern that matches either A or B and replace it with a C which is much faster in a single pass.

The common regex methods include `search`, which finds a particular pattern in a string, `sub()`, which replaces the matching pattern with a substring, `split()` which splits a string with a matched pattern, `findall()` which returns a list of substrings which match the given pattern.

Search behavior is controlled with regex flags. One very common regex flag is `ignoreCase`, which ignores letter case and makes a simpler search pattern. Instead of packaging different types of case in the pattern, you are only concerned with packaging one and `ignoreCase` will take care of ignoring the letter case of the search string.

Building a regex pattern is a mix of art and science so it comes with experience and knowledge. Now, let's look at the code. We have this string that we would like to anonymize the social security number and to replace every single digit with a string replace operation, we would have to apply replace ten different times to replace every one of these digits with an asterisk. This a little bit cumbersome and, again, we're passing over this string ten different times.

Instead, we could call a package `re` for regex. Regex has this very neat `sub()` or substitution operation or method where you can search for `\d`, which searches for any digit, and replace that with an asterisk. In first position you have the \d which is a search pattern, in the second position you have the replacement string. In third, you are given the string that you want to search. Alternatively, you can replace the digits with a 0-9 character class. The character class will look for any character in that class. Those are square brackets representing the character class, and there you can be a little bit more detailed, you can fine-tune the range of the characters from 0-5, from 4-7, or from 0-9 which would be equivalent to `\d` and replace all those with an asterisk.

There's a much larger list of matching rules, and this is not a comprehensive list by any means. You can go over this in detail and practice. It does take quite a bit of practice to tune a regex string and make sure it's not buggy. It fixes what you want to fix, but it doesn't create more problems somewhere else in the document.

Back to Table of Contents

Code: Practice Using Simple Regular Expressions

In the previous video, Professor Melnikov introduced regex patterns and demonstrated how to use them. In the "Review" section of this ungraded coding exercise, you will use regex techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will use these techniques on some new strings.

All practice exercises are optional and ungraded.

Please complete this activity in the course.

Back to Table of Contents

Watch: Use Regular Expressions to Find Patterns

Regex pattern searches are especially powerful for pattern identification within strings. In this video, Professor Melnikov demonstrates how combining regexes can create very specific or complex searches that can be easily executed. Then, he compares these methods to the alternative you would need to use if you did not use a regex search.

Libraries/Methods

re Library

- `search()`
- `findall()`
- `IGNORECASE()`

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

Now let us investigate a few basic regex examples in searching and retrieving parts of a string based on a specific pattern. We will also look at examples of replacing the strings.

Here is a string with lots of different -fix sub words or substrings in it and we want to look for a substring that starts or contains -fix, but it's followed by a word character, which includes letters of any case and digits and underscores, those are word characters. At least one word character must follow and as you can see, only fixture word is qualified for this type of search, which is exactly what this search method returns. But the search method returns more information. It also tells us where this fixture begins, where it ends, and if you have multiple searches, it'll be a list of all the different matches with all this detailed information.

We can extract individual components out of each of this search object, such as "fixture," could be extracted by indexing or slicing the list, or this result of the search object. You can also return the starting position by calling `start()` method on the result of a search object. Or you can return the ending position — 44 in this case — by calling the `end()` method of this result of the search object.

Here's another string where we have all these 1s and 0s, and it could be a very long string and you want to find every digit that is suffixed with two 0s or followed by two 0s. There are two such searches. Both of them are 100 and both of them are returned as a list if you are using the method called `findall()`. `findall()` doesn't return all this detailed information we've seen with the `search()` method, but it returns perhaps what you need for this particular search.

There's another application of `findall()`. Now we're using a character class. The square brackets represent the character class and inside the character class, we can place any characters that we want to search for. There's no ordering for those characters, so any cases of N or L or P are being looked for at least one time. The plus symbol indicates at least one instance of these letters. The only matching string is NLP, and that's what is returned as a list of findall(). If you do need only one word, the first word, you will be extracting that from the list. We can ignore the casing with the flag called `IGNORECASE`. In this case we don't have to provide all upper and lower cases of the word, which is cumbersome sometimes. We can just specify NLP lowercase, or any case, and the `IGNORECASE` will search for any casing of that pattern. In this `findall()` example we're searching for either "we" or "NLP," in which the "or" is indicated by the pipe symbol.

Here is another example where a string containing different morphological forms of the word "run" is searched for the word "runs," it is followed by 0 or more word characters regardless of the casing. "Runner," "running," "runs," all qualify for this pattern. All of this are returned as a result in the list.

Another example where substitution is made and we want to substitute "NLP," uppercase only, NLP uppercase with "NLP and Python" in the string, "we love NLP" and that's what you get as a result, with the NLP being replaced. Another example with the DNA sequence, where ACGT nucleotides are being used. What you're looking for here is A followed by either character A or T. They are put together in the character class, these square brackets. Or you're looking for another combination, or the A is preceded by the character G or T. It's also wrapped as a as a character class. In case either one of these patterns is matched, we are replacing this with two dashes, and that's what you get as a result, where the dashes are located is where the pattern was matched.

Back to Table of Contents

Code: Practice Using Regular Expressions to Find Patterns

In the previous video, Professor Melnikov introduced several regex methods from the `re` Library. In the "Review" section of this ungraded coding exercise, you will put those regex techniques to use. In the "Optional Practice" section of this exercise, you will apply these techniques on some new strings.

All practice exercises are optional and ungraded.

Please complete this activity in the course.

Back to Table of Contents

Watch: Parsing Strings With Regular Expressions

Libraries/Methods

re Library

- `findall()`
- `split()`

As you have discovered in this module, preprocessing strings by tokenizing them into substrings such as words or sentences is an important step in NLP. However, this task can be challenging when the text contains non-standard characters or words. For example, a book about the R programming language might include variable names that contain periods. These periods could be incorrectly identified as a sentence break when the text is tokenized. Here, Professor Melnikov demonstrates some of the challenges associated with parsing complex strings and uses regex to split some difficult-to-parse strings into more accurate tokens.

Later in these courses, you will evaluate and apply regex strings to fine-tune established models to parse text into the desired set of tokens. Even if the default regex string is left untouched, it is important to understand how it works and how it can be improved.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

Many highly sophisticated tokenizers use regex to parse a string into words and sentences. Some can even be trained on different languages to learn to identify tokens. Let's look at some examples where we use `findall()` and `split()` methods to break this string up into words and sentences. Here we have a string with words where some of the words are specifically combined with dashes and underscores. We'll see how this will be treated with different pattern searches of the regex.

We have find `findall()` and `split` next to each other. `findall()` will look for all the word characters. Again, these are letters of any case and digits and underscores. Plus indicates as many instances as we can find; at least one is what we're looking for. Each match will stop at whenever that pattern is no longer found. Any non-word character will stop the match, and the match will resume at any word character thereafter. We have these words returned as part of the list. The `split()` will look for any non-word characters appearing multiple times, at least once, as indicated with a plus sign. These results are almost the same with an exception of the last element were the `split()`

returns an empty string and that's because the ellipsis is also `split()` upon, was a `split()` operation.

Here's another example where we have two sentences separated by an exclamation sign. There's a lot more noise in the words. There are words that contain numbers and dashes. So "Python-3" would be challenging to split. With the word character, it will be a little bit more difficult. So we're using the character class that allows us to fine-tune the characters we want to split on. It's not ideal, but It's perhaps what the situation needs.

Again, we want to split into words. The `findall()` method will look for anything that is not, that's what is indicated by the caret symbol, anything that is not a space, not an exclamation, not a period, at least one time and that's indicated by a plus. The `split()` operation, will look for any separator such as space, period, or exclamation, again, at least one time. As a result, we get two lists of words where the "Python-3" is correctly matched and captured as a single word. Again, because of the period at the end of the sentence, `split()` returns an extra element.

Here we have "Python-3.x." Now the task is to split this string into sentences, but the period between "3.x" might be tricky. With the `findall()` and `split()` operations that are looking for words or looking for a character plus containing the period, we will be splitting the "3.x" as separate sentences, which is not what we want. This last `split()` operation will force a pattern search for a period, or an exclamation sign, or question mark followed by a space or as many spaces as we'd like, as indicated by the plus sign, which applies to the previous character, which is a space in this case. The "3.x" doesn't have any space followed by the period and it will be contained together. We are getting much better results with the two sentences, and the only drawback is that the period is still part of the second sentence.

Back to Table of Contents

Tool: Regular Expressions Reference Guide

In the previous videos, Professor Melnikov introduced Regular Expressions as a powerful way of manipulating strings, specifically to either find a pattern, or to find one pattern and replace it with another. The reference tool here serves as a summary of many basic regex pattern rules, as well as a summary of several of the methods that Professor Melnikov used in this module, with examples in Python. Refer to this tool as you work with regular expressions throughout this course and in your future work.

Please view and download this tool in the course.

Back to Table of Contents

Code: Practice Parsing Strings With Regular Expressions

In the previous video, Professor Melnikov used regexes to tokenize complex character strings. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will apply some of these techniques to parse different complex character strings.

All practice exercises are optional and ungraded.

Please complete this activity in the course.

Back to Table of Contents

Quiz: Regular Expressions Basics

In this module, Professor Melnikov introduced the basics of regexes. Now it is time to test your understanding of this useful method. Select your responses (keeping in mind that a question may have more than one correct answer) and submit your quiz when you have finished.

You may take the quiz as many times as you'd like. Your best score will be included in your course grade.

Please complete this quiz in the course.

Back to Table of Contents

Tool: Libraries and Methods for Finding Simple Patterns in Text

So far, you have used a variety of methods to manipulate strings, including methods from the Standard Python Library and from the `re` Library. Use this tool to help you recall the methods you have worked with in this module.

Please view and download this tool in the course.

Back to Table of Contents

Assignment: Course Project, Part One — Finding Simple Patterns in a Body of Text

In Part One of the course project, you will use the string manipulation techniques that you practiced in this module to tokenize a document into words. You will do this using methods from both the Standard Python package and the `re` package. The tasks you will perform are based on videos and coding activities in the current module, but may also rely on your preparation in Python and basic math.

Resources

Use these resources to help you as you complete the course project:

- Project Forum
- **Jupyter Notebook Guide**
- **Module 1 Libraries and Methods**

Use the tools from this module to help you complete this part of the course project. Additionally, you may consult your course facilitator and peers by posting in the project forum that is linked in the sidebar.

Instructions

Read the general project instructions within this accordion carefully, then follow the instructions specific to this part of the course project in the Jupyter Notebook below.

This exercise is **graded**, and may take ~3 hours to complete.

Please complete this activity in the course.

Back to Table of Contents

Module Wrap-up: Find Simple Patterns in a Body of Text

Understanding and being able to implement the basics of preprocessing is a key step you must master before working with more advanced preprocessing techniques or perform analysis with NLP.

In this module, you explored the definition of NLP and the hierarchical nature of language structure so that you have a solid understanding of how the strings, documents, and corpora you might work with in the future fit together. You also defined and practiced some basic vocabulary that can be used in NLP. Finally, you performed some preprocessing and string manipulation techniques, from both the Standard Python Library and regex, on single strings. You also practiced these techniques on some larger documents to start to work with the power of NLP at scale.

Back to Table of Contents

Module Introduction: Apply Preprocessing Techniques to Reduce a Text's Vocabulary

Consider all of English Wikipedia's content. That’s about 6 million articles, 4 billion words, 24 billion characters, and 600 words per article. When we search for “Mars lakes,” we want fast and relevant results. Searching every Wikipedia page for the query phrase is slow and unlikely to produce meaningful results. To conduct this search faster, you could build statistical distributions of words, phrases, and their semantic similarities to the query phrase, then retrieve representative articles and, finally, display these in order of relevancy. In other words, you need to reduce the vocabulary of the source text to make the search faster. So, it is crucial to understand what exactly you need to summarize in a document and how to clean and parse it into important elements, and then only work with these important elements.

In this module, you will build on your string manipulation skills and extend them to perform string parsing and document cleaning as part of preprocessing a text. These techniques are key to reducing the vocabulary within texts, which prepares texts for statistical analysis. Unfortunately, these techniques can be somewhat tedious, but most of them have been packaged into popular and efficient Python libraries by the NLP community. As a result, you can focus on learning how to apply these libraries for text preprocessing, including turning words into tokens, finding shared roots between words ("dances," "dancing," and "danced" all being stored as "dance," for example), and working with contractions.

This module will require approximately *13 hours* to complete.

Back to Table of Contents

Watch: The Goals of Preprocessing Text

When you use NLP, your goal is to better understand some component of a corpus, like the sentiment it expresses. You can do this by analyzing the vocabulary of the corpus, which is the set of tokens present in the corpus. Before you preprocess a corpus, however, every word, even misspellings or differently capitalized words, is included in the corpus' vocabulary. For example, if "can" and "Can" are both present in a corpus, the computer will interpret them as different tokens unless told otherwise. A critical preprocessing step is to reduce the vocabulary in the corpus you’re working with by standardizing the tokens present in that corpus. You have already seen some preprocessing techniques you can use to do this, such as removing capital letters, but there are many other preprocessing techniques you can use. In this video, Professor Melnikov discusses several text elements that might need to be removed or altered with different preprocessing techniques before you can begin to work with a corpus.

Note: In this video, Professor Melnikov uses the term ***lemma***. This term will be explored in more detail later in the course, but for now, understand that a lemma is a base (head) word. Words like "throws," "thrown," "threw," and "throwing" all would have "throw" as their lemma or a dictionary form of the word.

Video Transcript

Preprocessing is a set of techniques used to standardize textual content with minimal loss of information. Our goal here is to reduce the vocabulary size of our corpus. The original vocabulary might contain different capitalization of the same words, different morphological structures, different word tenses. It can contain intentional or accidental misspellings, acronyms, abbreviations, contractions, accented words, multilingual text, emojis, memes, and even structured identifiers like HTML tags. All of these contribute to a larger and larger vocabulary, which doesn't necessarily add additional meaning. We want to minimize and standardize the words and minimize the vocabulary so that we can process this text more efficiently.

Consider a search for a book in a corpus. That book that we're interested in needs to be related to code. Now, there are lots of books on code, and if you open them up, they will probably contain words like coding, code, codes, coded, coder, but not all of them are exactly in the word form of the word code. The search counts for the word code may differ and not necessarily be representative.

What you want to do is you want to standardize all those different words to a single word form that matches your search word. In that process, you're applying all these different preprocessing techniques to standardize those words and find the true lemma that you're looking for. Then you can retrieve a book that has the highest count of the word code and is most relevant to your research. This also helps the compression of the text. You compress the vocabulary, the library of words that the book contains. This allows you to speed up the search for its textual content as well as use minimal computational resources. In this module, we will study different techniques to address these different issues.

Back to Table of Contents

Watch: Text Tokenization

Tokenizing, or parsing, a document or corpus into smaller components is a key step in preprocessing because it will allow you to analyze documents in a more meaningful way than you would be able to based solely on a document's lexicon. For example, once you have tokenized a document into words, you can count how many times a particular token exists, which can help you understand what a document is about. In this video, Professor Melnikov discusses some of methods for tokenizing a corpus that are available in Python's **NLTK Language Toolkit**, such as `word_tokenize()` and `sent_tokenize()`.

Video Transcript

Word tokenization or segmentation is parsing a string document into words or tokens. These tokens are deduplicated to form a vocabulary, or we know it as a dictionary or a lexicon. Some of the popular ones are Oxford Dictionary of most common word forms.

Similar documents would typically have a greater overlap if you take their dictionaries and compare them to each other, but these dictionaries or these lexicons are typically biased towards infrequent words. So if you have two documents, one has a single word "cat" and the other document which is all about cats and has multiple words of "cat," maybe a hundred instances of the word "cat," if you're just looking at their lexicons, they will be very similar because cat will appear in both lexicons. However, if we take the count of the word "cat" into account, then we will have a much better, more precise similarity between these documents, measured in this way more properly. We can identify documents that have multiple words of "cats," as opposed to having this outlier with a single instance of the word "cat." We can tokenize a document into different phrases, sentences, paragraphs, chapters, any substring of different type, as long as the type as consistent.

A sentence in particular captures a statement or an idea, by grammatically arranging relevant words. Such grouping, or collocation, as we call it, allows us to build connections among words that frequently appear together in the same context. For these reasons, tokenization into words and sentences are typically the most common. NLTK or Natural Language Toolkit is one of those popular Python packages with different types of tokenizers. The default ones are `word_tokenize()`, `sentence_tokenize()`. Most of these tokenizers use very clever and computationally optimized regex patterns suited for different natural languages. In fact, these regex patterns are automatically trained in many cases on a large corpus to distinguish sentence bounders from abbreviations or URL strings or IP addresses, and many of them use `PunktSentenceTokenizer()` as this underlying engine for tokenization of the sentences.

Note that many of the popular tokenizers in `nltk` are default shortcuts for long-name parsers in `nltk`, such as `TreebankWordTokenizer` or `PunktSentenceTokenizer`, which you can access via `nltk.tokenize.treebank` and `nltk.tokenize.punkt` modules, respectively.

Back to Table of Contents

Tool: NLTK Library

Download and use the following tool as a reference when working with the NLTK Library.

Please view and download this tool in the course.

Back to Table of Contents

Code: Parsing a Document into Tokens

In the previous videos, Professor Melnikov discussed some preprocessing techniques, including using the NLTK Library to tokenize a document. In the "Review" section of this ungraded coding exercise, you will explore how to use these techniques by running code that carries out these functions. In the "Optional Practice" section of this exercise, you will apply some of these techniques to tokenize documents.

All practice exercises are optional and ungraded.

Please complete this activity in the course.

Back to Table of Contents

Quiz: Defining Tokenization Methods

In this module, Professor Melnikov has introduced different tokenization methods and you have practiced these techniques. Now it is time to test your understanding of them in this short quiz. Select your responses, keeping in mind that a question may have more than one correct answer. Submit your quiz when you have finished.

You may take the quiz as many times as you would like. Your best score will be included in your course grade.

Please complete this quiz in the course.

Back to Table of Contents

Watch: Working with Characters

Libraries/Methods

Standard Library

- `encode()` method
- `decode()` method
- `unicodedata.normalize()` method
- `sub()` method

So far in this course, you have worked with ASCII characters, which includes a relatively limited set of letters, numerals, and punctuation. Often, however, you will work with a much broader array of characters that are part of *Unicode*. Unicode includes basic characters but also encompasses a broader array of complex characters, such as accented characters or emojis. These complex characters, while ubiquitous, present a unique set of challenges that require extra planning as you preprocess text. In this video, Professor Melnikov explains the different ways that characters can be encoded, and walks through some examples of both removing and preserving accented characters by converting from ASCII to Unicode and back.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

Around the 1960s, ASCII code was developed to map about 128 numbers to lower and uppercase letters, some digits, zero to nine, some punctuation. This was sufficient at the time when computers had limited power. As digital borders opened up and technology advanced, the demand grew for encoding world alphabets or world characters, emojis, mathematical symbols, and much more. We know this Unicode character set was developed. It has about 1 million characters sufficient to represent many different languages, and only about 25% is currently in use.

Among many Unicode versions — there's more than one — UTF-8 is probably the most popular. It's the smaller one, and it uses a variable length code to represent different symbols. In Python-3, this is the default Unicode, and every string is encoded with this Unicode format. But you can explicitly indicate that your string is Unicode by using the letter u in front of the string. We can also explicitly convert between the Unicode and ASCII using the encode and decode methods of a string.

To avoid losing accented Unicode characters, we need to be careful. Some normalization of the string needs to come first. Other kinds of operations that we can do on characters are lowercase and uppercase in different strings. But we have discussed those earlier. We can also de-accent and remove special characters.

Let's look at the code, at how it's done in Python with different types of strings. Here we have a string which has "sugar-free," with a dash. It has "crème brûlée," the French version and "creme brulee," the English version and some numbers with a dollar sign, some prices. Very high-priced crème brûlée. Encoding will convert it to ASCII, but we will lose the accented letters, which is probably sub-optimal. To keep those letters, we can use this normalize method, with the NFKD form of normalization.

After that, when the equivalent letters have been found for the accented characters, not for the emojis that we had in the string, then we can encode it with the ASCII and then we can decode it if we want to back to UTF-8 format.

As a result, we get this string where the "crème brûlée" is correctly represented and emojis are lost, but everything else seems to be intact. We can also explicitly extract words out of a sentence, but we have to be careful when we switch from a word character `\w`, to letters A through Z. The letters A through Z do not understand the Unicode accented characters, and they will ignore those and not treat those as Latin alphabet. If you are looking for all the characters except A through Z in different casing or underscores and spaces, you will have "crème brûlée" missing some characters that were accented earlier. To have a correct search, use `\w`, which understands the Unicode characters as letters, and will keep those in your text, cleaning everything else out. Notice that if you do that, the emojis are gone, but also the dollar representation of a price will be misrepresented, because the periods and commas are now missing and they will require more a complicated pattern in regex.

Back to Table of Contents

Code: Work with Characters to Standardize a Vocabulary

In the previous video, Professor Melnikov introduced how to use regex to convert from ASCII to Unicode to preserve complex characters. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will apply some of these techniques on different strings.

All practice exercises are optional and ungraded.

Please complete this activity in the course.

Back to Table of Contents

Watch: Expanding Contractions

Libraries/Methods

re Library

- `unContract_generic()` algorithm
- `unContract_specific()` algorithm

contractions Library

- `fix()` method

Many modern languages use contractions, which combine or shorten words. Contractions might make a language easier to speak, but dealing with them can be challenging for computational linguists, data scientists, and NLP engineers because contractions are not used to consistently represent a group of words. For example, "Ed's car" can be just a possessive form (not a contraction) or can be a contraction that means "Ed has a car," "Ed was a car," or "Ed is a car," among other phrases.

Since there is no easy rule that would work in all situations, a Python algorithm applies whatever rule was coded in advance, unless some statistical reasoning takes place at the time of the contraction expansion decision. In general, we hope that the expansion rules are chosen to follow the more likely scenario and that they reduce the vocabulary of the corpus. Often, basic rules can expand contractions with reasonable accuracy, and Professor Melnikov demonstrates three of these rules in this video.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

In many languages, common phrases are constructed for brevity, but this adds noise and redundant words to our lexicons. In English, these are: I'm, we've, they've, he's. To reduce the vocabulary size, we often use rule-based expansions of these contractions. These matching rules require caution with respect to capitalization and types of quotation marks. In Python in particular, we have left and right single quotes, we have apostrophe, we have other marks that need to be accounted for. A generic rule, expand common suffixes regardless of the full phrase. So as soon as 's is found, it can be expanded to has. Unfortunately this expands possessive forms like "Mike's car" to "Mike is", or "Mike is a car" and "Mike has a car."

A specific rule, for that reason, can be affixed to this situation where a key value map is used to find the specific contraction and expanded to its expanded form. We can also use a mixed rule where a specific rule is used first and then a generic rule can be used in the second iteration to fix all the missed contractions. There are also more advanced algorithms that will account for the full context of a sentence or the contraction to produce higher quality expansions.

Let's look at the code. So we have this `unContract_generic` function which will take `sTxt`. Inside that text we're searching for the 't or 're, all these different endings of a word will be expanded into not, are, is, would, will, and so on, regardless of what the full phrase is. This works partially. Well, here's a sentence where it actually does work successfully because it expands "now's" and "NLP's" without specifically indicating that NLP is a contracted form.

Here's a sentence where it doesn't work successfully because "Ed's kitchen" is a possessive form and we do not want it to be actually expanded. For that reason, we will use this `ContractionMap` where we can explicitly indicate, hey, here's a list of the keys that are contracted phrases, and each key corresponds to expanded form, so "ain't" corresponds to "am not" and so on. This is also problematic in some situations because some contractions can expand to multiple forms. "It's" can be expanded to "it has" or "it is," and we can only choose one expansion here with this method.

Let's see how this is applied in a function. There's a function called `unContract_specific`, which will take a text and it will take this map or a dictionary. Inside this function, we are extracting all the keys from the `ContractionsMap` and compiling something called groups, the regex groups. Every group is separated by the "or" operator or a pipe and these groups allow us to reference the found patterns. So it will have a 100 different patterns, a 100 different contraction patterns, and the ones that are found can be referenced. Once we reference it, we can find in the map, we can find the corresponding expansion and replace it instead of a group. That's what this function does.

So let's see how it works with the same statement. It correctly expands "your" to "you are" because it was part of the list, but it doesn't expand "Ed's kitchen," "Ed's" remains as is because it wasn't on the list. We can, of course, modify this dictionary and add more contractions or remove the ones that are not needed, and so on. There is also a package called `contractions`. If we import this package, we can use it's fix method to expand on contractions in any sentence. All it is is just a wrapper for that dictionary, which we've seen that long list of contractions and expansions are packed inside this constructions package for convenience and it offers a few more methods of modifying this dictionary, but for the most part, it's a nice and clean way of applying contractions.

Back to Table of Contents

Code: Modify and Apply a Contraction Map

In the previous video, Professor Melnikov introduced generic rules, specific rules, and mixed rules that you can use to expand contractions. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them. In the "Optional Practice" section of this exercise, you will apply a contraction map to expand contractions in a corpus that is part of NLTK, Lewis Carroll's classic novel "Alice in Wonderland."

All practice exercises are optional and ungraded.

Please complete this activity in the course.

Back to Table of Contents

Watch: Text Correction

> *"That's **gr8**! I'm so happy **4u**! You can **fianally** follow **ur** dream! Sending you lots of **looooooooovvvveeee**!"*

Libraries/Methods

nltk Library

- `punkt` module
- `wordnet` database

Given the amount of text that humans generate online, NLP practitioners are bound to encounter text that contains intentional and unintentional typos, creative spellings, repeated characters, and phonetic substitutions ("gr8" for "great," to pick just one example). Text like this is common, such as in a product review written on a phone, and it can yield powerful insights as long as Python knows what to compare it against to find the "correct" or intended word. Watch as Professor Melnikov demonstrates how to use an algorithm and a lexicon to scan for and omit repeated characters.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

Much of the electronic text is generated by humans via web forms, or web entry forms, document editors, tweets, discussion forums, emails, optical character recognition scanners, and so on. The accidental spelling errors are introduced, which sometimes result in new words or new meanings. Note that some typos are even purposeful and malicious, where a bad actor is trying to bypass the algorithm and post a toxic message to Wikipedia or a social network. Some common types of misspellings are repeated characters and phonetically similar substitutions. Someone might use the number eight in the word "great," or the number four in the words "for you," or an expression "for you." We will investigate an algorithm that fixes repeated characters.

Of course, the algorithm doesn't know what a correct word is. It needs to check the word against a reputable lexicon such as WordNet. That's a very popular database of words. There are some other ones as well that we'll study in this course. The proper word is not deformed when we're looking for it. The algorithm iteratively reduces the duplicated character in the word until the word is found in that lexicon. Oxford Dictionary can be another one or any, any similar universal lexicon. If the word is not found, then the deduped form is returned.

Let's look at the code. Here we have a download of the punctuation which is required by this particular routine. But notice the WordNet database, and we'll talk more about WordNet later in this course sequence. For right now, just think of it as a lexicon or a dictionary of different words, we'll load it from NLTK corpus once it's downloaded locally.

There is this function that does all the scanning of repeated characters. The key scan and or pattern search is this line right here, which looks for three different groups, and the groups are identified and in parentheses, the first group looks for an arbitrary number of word characters, which is indicated by the `\w` followed by an asterisk. The second group looks for just one character, `\w`. Then after that second group, you see this `\2`, which means either we require a repeated character, the middle character to be scanned or searched for. Then finally, we wrap it up with another group of word characters of arbitrary size. Now this group is returned.

Once found, each one of these groups is returned in exactly the same sequence, except the repeated `\2`. It returned the first group, the second group, and the third one, but not the repeated or duplicated character. Every time we call this replace subfunction, we keep reducing the character, the repeated characters by one have more than one repeated characters, duplicate characters.

At every iteration, we would check if that word isn't synset in the WordNet database. If it is there, we exit. We just find the words that is in any reputable dictionary. If it is not there, we keep reducing the duplicated character by one until we either no longer can reduce it or, again, until we find the word in the dictionary. That's how it's done. This algorithm only works as it is with lowercase, or the same case, I should say, words, but you can make it adapt to characters that are of different casing and are still repeated next to each other. We will call this function `DedupTokens` on this, set of words. These words in their proper form have duplicated characters. "Bitter" is found correctly, "bassoon," "bookkeeper," but the "subbookkeeper" is not reduced to its correct from and that's because it's missing from that WordNet database.

If you do want a particular word, specific word, if you are aware of it, to be fixed correctly, you would place it into a database that you searched against. Here's a phrase where several different words have multiple duplicated characters, and they're not all necessarily incorrect. When we fix this by using `word_tokenize` were first parse this sentence into different words, then we run it through `DedupTokens`, and then we rejoin everything back together. Notice that the eCornell is fixed to its incorrect version; there's only a single L and that's because it's not found in the WordNet database or library, but the Cornell is fixed correctly. "Really amazing" are also fixed correctly.

Back to Table of Contents

Code: Perform Text Correction on a Document

In the previous video, Professor Melnikov introduced how to correct text within a document. In the "Review" section of this ungraded coding exercise, you will apply these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will add some code to the text correcting user defined function (UDF) Professor Melnikov defined in the video, and use it to correct new sentences.

All practice exercises are optional and ungraded.

Please complete this activity in the course.

Back to Table of Contents

Watch: Spelling Correction with TextBlob

Libraries/Methods

textblob Library

- `correct()` method
- `spellcheck()` method

Incorrect spelling is extremely common in written documents, especially in the largely unedited body of text people write when they communicate online. This doesn't always hinder communication between people, because most human brains can recognize that "sepllnig is hrad" actually means "spelling is hard." Computers, however, cannot recognize these patterns unless given instructions for how to do so. In this video, Professor Melnikov demonstrates how to deal with incorrect spelling that needs to be fixed before a document can be analyzed.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

Spelling correction is a notoriously difficult problem in every language because a candidate word needs to relate not only structural and syntactically, but also semantically. A trivial algorithm might use a map of common misspellings. It is fast but doesn't generalize well because you need to keep adding words, misspelled words into that map.

A more sophisticated algorithm might look for words with the minimal number of letter edits, such as deletion, insertion, substitution between the given word and the candidate word. This so-called edit distance or Levenshtein distance is very slow and needs to be computed for a wide range of candidate words. An even smarter algorithm might combine several different methods and use all probabilistic typos or account for them. This might arise from letters spaced out in a different way on a keyboard, or rising from human tendency to mix vowels and consonants or neighboring words in the phrase also could impact your misspelling in a sentence. Other human, physical, and grammatical factors are also accounted for.

Peter Norvig is a leader in this field and has created a very nice algorithm. He's the Director of Research at Google. This algorithm combines all of these different techniques is wrapped in a package called TextBlob. This is an amazing NLP library with rich functionality including tokenization, word tag, and many other preprocessing techniques.

Let's look at its spelling correction in the code. We'll load this textblob word class and wrap the word "fianlly" in the word object. This operates just like any string, but it has a few more attributes and methods, one of which is correct. That pulls the best candidate for the word "fianlly" which is "finally," which is exactly what we want.

If you want to see the different candidates and their confidence score, you can call the method called `spellcheck()`. The `spellcheck()` will return multiple candidates. In the case of "flaot" is returns "flat" and "float" with 85% and 15% confidence. That way you can gauge or tune your algorithms to pick the right word if you see it or if you don't see it at all, you might add it to the TextBlob or other library for your specific domain.

Here we have this sentence of all the misspelled or almost all misspelled words. We're applying the word correction to every element of this splitted, scrambled string. We split at whitespace, which are tabs, newline characters, carriage returns, and spaces. All the spaces are present in the string. We split on spaces and each word or token is wrapped in a word object. We'll call the correct method. We return this as a list of corrected words. The `join` operation will pull them back together as a space separator into a string or a sentence. Again, this is a synthetic sentence, but if you have your own domain where a particular word is consistently being correctly or incorrectly fixed, you might add these packages and add your word or common corrections into this library or some other library you were using.

Back to Table of Contents

Code: Generate Probabilities for Spelling Correction on a Document

In the previous videos, Professor Melnikov demonstrated how to perform spelling correction on a document with the `textblob` library. In the "Review" section of this ungraded coding exercise, you will run the code you saw Professor Melnikov demonstrate in the video. In the "Optional Practice" section of this exercise, you will apply some of these techniques to scramble words so that you can see their possible misspellings.

The three tasks in this exercise walk you through how to generate probabilities for suggested corrections of misspelled words. This would be a step in the process of actually correcting the misspelling. For example, if the difference between the probabilities of the first and second highest options is high enough, you might decide to simply replace the word with the most likely misspelling. On the other hand, in a situation where there are two or more words that are close to equally likely, you might have your program prompt the user to select the correct word from a list of those options.

All practice exercises are optional and ungraded.

Please complete this activity in the course.

Back to Table of Contents

Watch: Stemming and Lemmatization

In addition to what you have explored so far, there are two other preprocessing tools you can use to reduce the vocabulary of a corpus. These are *stemming* and *lemmatization*.

Libraries/Methods

`nltk` **Library**

- `stem.PorterStemmer()` method
- `stem.WordNetLemmatizer()` method
- **Brown** Corpus

A *stem* is a partial word with the suffix removed. A popular method (**Porter stemmer**) simply removes the "-ly," "-ing," or "-ed" from words, to name just a few suffixes, and what remains becomes the stem. This is a fast, but inaccurate, technique. Stemming uses a few clever rule-based ways of trimming suffixes, but there are many exceptions in word morphologies due to blends of languages, historical language transformations, and so on. For example, "plotly" is a Python package name, not an adverb form of the word "plot," although the two words are related.

Lemmatization retrieves the *lemma,* or dictionary form, of a word. This method takes time to reference against a lexicon, so that it knows what the correct lemma of a word is. There is a tradeoff of speed against accuracy with these two methods. In this video, Professor Melnikov walks through using both methods on a sample text.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

In the spirit of reducing the size of a lexicon underlying corpus of documents, we can use stemming to standardize varying word forms of a single underlying stem. A stem is a partial word with truncated suffix. A popular Porter Stemmer uses a set of rules to chop off common endings like -ed, -ly, or -ing and -s for plural wards. It is a simple and fast algorithm but tends to over-stem or under-stem non-Latin words. For example, "alumnus," which is a already singular form, is stemmed to "alum" or "universal" and "university," which have different meanings, are stemmed to "univers" without an e. Consequently, some unrelated words can yield the same stem or multiple morphs of the same word can still result in different stems.

If you want to improve that quality, you use lemmatization, you lemmatize words to their lemmas. A lemma is a root or a dictionary form of a word. Many of these are stored in the WordNet database, a common database that is included in natural language toolkit, the NLTK library. Lookups against the WordNet can be slow, but they are reasonably effective in word standardization, if parts of a speech label is provided for a word. This POS tag, such as verb, noun, or adjective, helps to identify the correct root and properly handle the word suffix. If a candidate lemma is not in some universal dictionary, the original word is returned.

In the following exercise, we draw vocabulary from a well-known Brown Corpus of 50,000 unique words. This collection was compiled in 1961 from 15 genres and 500 English sample, totaling about 1.2 million words.

Let's look at the code. Here, we'll load the necessary libraries and some identifiers of what nouns, verbs, and adjectives are. Then download the WordNet Corpus and Brown Corpus. This will be useful for our standardization and checking.

The Porter stemmer object is wrapped into pos and the Wordlet lemmatizer object is wrapped into `wlo`. The next four functions are lambda functions. They are basically very simple, anonymous functions that help us rep, for simplification purposes, rep the non-adjective forms of verb parameters into something shorter.

In the next cell, we're using selected words. For each one of those words, we create five different string. First string is the original string then the standard version of it, then the lemmatized version given that it's a noun, given that it's an adjective, and given that it's a verb.

The resulting list of tuples has these structures that we can wrap into a dataframe to present in a nicer format with column headers. You can see that in the first row, the word "running," the original word "running" is stemmed correctly and lemmatized correctly if the verb parameter is provided. With other parameters of our POS tags, noun and adjective, it remains as is. Same works for all the other words including "incubation." "Debug," for some reason, cannot be stemmed or lemmatized any further, doesn't matter which pos tag is supplied.

There is a Brown Corpus of words that first we need to create a set of. We'll create a set of unique words. There are 1.2 million words, 56,000 of them are unique. Here's an example of what these words are: "the," "Fulton," "County," and "grand," "jury," and so on. Let's first take a dataframe or values from the dataframe and flatten them. This is how the array of values looks like. It's still two-dimensional. If we flatten it, we have a single dimensional array that we can apply set operation to get rid of all the duplicates.

Now if we go over these elements of the array and check whether each one is in the SsBrownWords, we can leave as a result only those that are not found in this lexicon. Surprisingly, "corpora" and "debug" words are not in there, but this was a 1961 set of examples. Although "corpora" is a very old word, but "debug" maybe something that is recently used. The words "agre," "tri" and "incub" are properly flagged, and you can reapply a proper lemmatization technique or you can increase the set of SsBrownWords and combine it with the Oxford Dictionary, a more modern library of words, to properly account for "corpora" and "debug."

Back to Table of Contents

Code: Stem and Lemmatize a Document to Measure Vocabulary Quality

In the previous video, Professor Melnikov introduced the methods of stemming and lemmatization to help reduce the overall vocabulary. He also introduced several features of `nltk`, the WordNet lexicon, and the Brown Corpus. In the "Review" section of this ungraded coding exercise, you will run the code you saw Professor Melnikov demonstrate in the video. In the "Optional Practice" section of this exercise, you will lemmatize and stem a given paragraph, then check it against Brown Corpus.

All practice exercises are optional and ungraded.

Please complete this activity in the course.

Back to Table of Contents

Watch: Removing Stopwords

The final method of reducing vocabulary that you will examine in this module is the removal of stopwords. *Stopwords* are words that have no real significance in a document, such as "the," "a," "that," and so on. Stopwords are among the most frequently-occurring words in any document, but they rarely add value to your analysis because they do not define the document's meaning, topic, or theme. Neither do they help differentiate or relate documents to one another.

Libraries/Methods

`nltk` **Library**

- `stopwords.words('english')` list

Stopwords occur so frequently in most documents that removing them has a measurable impact on the size of a document and its vocabulary. Thus, removing them from a corpus greatly improves the efficiency of processing the corpus, comparing documents, and document storage. Watch as Professor Melnikov demonstrates how to remove stopwords using a built-in list of approximately 180 words from the `nltk` library.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Back to Table of Contents

Code: Remove Stopwords From a Document

In the previous video, Professor Melnikov explained what stopwords are, and demonstrated why they can be so troublesome for the overall size of the vocabulary. In the "Review" section of this ungraded coding exercise, you will examine the code Professor Melnikov demonstrated in the video. In the "Optional Practice" section of this exercise, you will practice removing stopwords from a sample text in order to reduce the vocabulary.

All practice exercises are optional and ungraded.

Please complete this activity in the course.

Back to Table of Contents

Read: Suggested Reading: Removing HTML Tags in Preprocessing

So far, you have examined several different techniques for preprocessing text that will help you standardize it by reducing its vocabulary. These techniques include tokenizing, stemming, lemmatizing, and removing stopwords. However, these techniques do not change any HTML tags that might appear in the text.

☆ **Key Points**

HTML tags can interfere with NLP analyses, but are frequently found in the types of documents you would want to analyze with NLP.

You can use functions from the `BeautifulSoup` and `re` libraries to strip HTML tags from a document.

The Problem With HTML Tags

Documents that you might want to analyze with NLP may contain HTML tags such as formatting tags, markdown symbols, JavaScript, or other semi-structured content. These might occur when text is scraped from the web, for example. While these tags are helpful to readers of the text, they create noise when you want to analyze that text with NLP because they both add unrelated words and are unlikely to create meaning. They also require unnecessary allocation of computing resources, such as memory, processing, and storage. Consider this example of raw HTML text:

```
<html>
  <head>
    Page Title
  </head>
  <body>
    <div>
      Sample text that may even include escape characters like this: /r/n.  Often it contains
<strong>inline formatting</strong>, <a href="http://www.samplelink.com">URL links</a> and oth
er tags.
    </div>
    <p id="34593">
      Here is a second paragraph that is formatted slightly differently than the first, this
time with a unique ID as a tag attribute, which would also contribute nothing to your analysi
s if you retained it.
    </p>
  </body>
</html>
```

Because of these problems, removing HTML tags from a text is part of typical NLP preprocessing pipelines for scraped web pages.

Two Ways to Remove HTML Tags From Text

In most situations, you can safely strip HTML tags (or structured tags of any kind, whether HTML or XML) with the `BeautifulSoup` library. There are many ways to use `BeautifulSoup` to define a corpus and strip HTML tags from it, and you can examine two examples below.

Example 1

The following code[1] begins by loading both the `re` Regex Library and the `BeautifulSoup` Library. This is a *very* powerful combination for web scraping.

```
import re
from bs4 import BeautifulSoup

def strip_html_tags(text)
     soup = BeautifulSoup(text, "html.parser")             # instantiate an html.parser objec
t
     [s.extract() for s in soup(['iframe', 'script'])]     # drop HTML tags
     return re.sub(r'\r|\n|\r\n]+', '\n', soup.get_text()) # readable text without line separ
ators
```

This example uses a user-defined function (UDF) called `strip_html_tags()`, but this is not strictly necessary. Either way, the first line of the code will be something like `soup = BeautifulSoup(text, "html.parser")`, that you see here. This "makes the soup."

The soup is the result of running the `BeautifulSoup()` constructor and passing it two parameters: the file to use (or the string), and the type of parser. `"html.parser"` is Python 3's standard HTML parser, though you can use other parsers (`xml`, for example, or `html5lib`). Next, you see the `extract()` method run on each string in the source text, which removes any embedded iframes or scripts from the text. The `get_text()` method then removes anything from the soup that is not semantically readable by humans, such as HTML tags, and apples regex pattern to it to substitute various forms of line separators.

Example 2

You could also write the following code:

```
html = urlopen('http://www.sample_url.com/page1.html')
soup = BeautifulSoup(html, "html.parser")
```

In this case, you're using the `urlopen()` function and sending the contents of the page to the `html` object, which you then use to make the soup.

Extend Your Knowledge

In this course, we're focusing on practicing the most common preprocessing techniques. Working with scraped content could be an entire course on its own. However, removing HTML tags is a worthwhile skill to have. If you would like to practice this technique, you can refer to the list of exercises within the **Next Steps** document at the end of this course to help you further your study.

Citation

[1]Sarkar, Dipanjan. *Text Analytics with Python: A Practitioner's Guide to Natural Language Processing*. Bangalore, Apress, 2019.

Back to Table of Contents

Tool: Choosing Preprocessing Techniques

A common thread across NLP projects is to simultaneously reduce the vocabulary of the corpus of interest, while minimizing the loss of information, however information is defined. To achieve this goal, you will clean, scrub, and otherwise standardize the text to collapse similar words into one word and remove "meaningless" tokens. Yet, not all preprocessing techniques are applicable in every case and you need to be particular in your choice of tools. As you decide how best to preprocess a corpus of interest, consider several basic rules summarized in this tool.

Please view and download this tool in the course.

Back to Table of Contents

Quiz: Identifying the Appropriate Preprocessing Technique

In this module, Professor Melnikov introduced several different preprocessing techniques. Now it is time to test your understanding of them. For each sentence, identify the first preprocessing technique you would use on the given sentence to reduce its vocabulary or remove noise.

You may take the quiz as many times as you'd like. Your best score will be included in your course grade.

Please complete this quiz in the course.

Back to Table of Contents

Tool: Libraries and Methods for Preprocessing Text to Reduce Vocabulary

As you've worked through this module, you have practiced many preprocessing techniques that you can use to standardize a corpus and reduce its vocabulary. Use this tool to reference the methods you have practiced throughout this module.

Please view and download this tool in the course.

Back to Table of Contents

Assignment: Course Project, Part Two — Preprocess Text to Reduce Vocabulary

In Part Two of the course project, you will complete the functions within the `Pipe()` class, which is a Python class for building custom text preprocessing pipelines. The tasks you will perform are based on videos and coding activities in the current module, but may also rely on your preparation from the previous module, as well as in Python and basic math.

Resources

Use these resources to help you as you complete the course project:

- Project Forum
- **Jupyter Notebook Guide**
- **Module 2 Libraries and Methods**

Use the tools from this module to help you complete this part of the course project. Additionally, you may consult your course facilitator and peers by posting in the project forum that is linked in the sidebar.

Instructions

Read the general project instructions within this accordion carefully, then follow the instructions specific to this part of the course project in the Jupyter Notebook below.

This exercise is **graded**, and may take ~3 hours to complete.

Please complete this activity in the course.

Back to Table of Contents

Module Wrap-up: Preprocess Text to Reduce Vocabulary

Standardizing text in large documents is critical when you are conducting analysis with NLP, both because it standardizes the tokens in your corpus and because it reduces the size of your document. This decreases the amount of storage space your corpus takes up, and reduces the length of time any further analysis on the corpus will need.

In this module, you examined string parsing and document cleaning, which are important steps in preparing vocabularies and statistical data for text analytics. You also practiced working with a few popular NLP Python libraries that handle much of the tedious preprocessing automatically. You also explored such preprocessing techniques as tokenization, mapping contractions to uncontracted words, stemming and lemmatization, and the removal of stopwords. Finally, you tied all of these techniques together by building a preprocessing pipeline and using it on a document. In the next module, you will apply these skills to tagging and parsing a document.

Back to Table of Contents

Module Introduction: Tag and Parse a Document

Recognizing the types of words, such as verbs, nouns, adjectives, that comprise a sentence is an important prerequisite in many NLP tasks. This type of grammatical tagging is called parts-of-speech POS tagging, and it is important because it allows a more meaningful compression of text to reduce the vocabulary of a corpus. For example, more abstract algorithms rely on POS tags to identify principal noun phrases in a sentence. Other algorithms use POS tags to convert verbs to their root form in the lemmatization process. Such normalization of verbs collapses *chose*, *chosen*, or *choosing* to just a single word, *choose*, thereby reducing vocabulary without significant loss in original information. Additionally, POS tags can help you identify the hierarchical relationships among words in a sentence, which can help you begin to understand what a text means.

In this module, you will explore and practice different techniques that can be used for POS tagging sentences. You will also examine several different parsing techniques (specifically: shallow parsing and dependency parsing), and practice them on a text corpus. Over the course of the module, you will gain further experience working with some of the more powerful features of the `nltk` library, and you will combine these techniques into a pipeline that can be used to tag, lemmatize, and uncover hierarchical relationships in a text.

This module will require approximately *10 hours* to complete.

Back to Table of Contents

Watch: Parts-of-Speech Tagging

Parts-of-speech (POS) tagging is an important preprocessing task in which grammatical labels that identify a word's function, such as verb, noun, or adjective, are assigned to each word in a sentence. These tags can be used to lemmatize word forms, thereby reducing or eliminating unnecessary tokens and morphological variants. They can also be used in analysis to extract meaningful phrases. In this video, Professor Melnikov walks through several examples of POS tagging, explains the differences between two sets of tags, and demonstrates how you can use tags to lemmatize a document.

Libraries/Methods

`nltk` Library

- `stem.wordLemmatizer()` definition
- `pos_tag()` method
- `word.tokenize()` method
- `Freq.Dist()` method

Standard Library

- `Counter` object
- `most.common()` method

We will often refer to POS tagging as "tagging," but it is important to note that there are many other non-POS tag sets in NLP, such as chunk tags.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

Parts-of-speech, POS, is a set of categories of words in a sentence. This includes noun, verb, and adjective. Words in the same POS category tend to express similar syntactic behavior. In particular, we can significantly improve lemmatization by specifying the POS tag of a word. A curious learner can investigate many of the sophisticated algorithms designed to recover POS tags, but we will learn to apply and interpret POS taggers.

Let's go to the code. Here we have several different definitions. One of them is WordNetLemmatizer, which we'll use to lemmatize words with application of POS tagging. We have a sentence from Walt Disney's quote, which uses some morphologically different verbs, different from their root form. We can easily tokenize it into a set of tokens with the word_tokenize method of nltk. It does a pretty good job of tokenize and everything to the point.

The `nltk` POS tagging or `pos_tag`, is a method that in the same seamless way will produce reasonably good quality tags. These tags that are by default and can be modified are UPenn or University of Pennsylvania tags that are very popular and common. So we can see that the word "the" has a tag of "DT" or determiner, "way" is a noun and so on. The NLTK Library also provides a `FreqDist` method which has a frequency distribution of whatever list we're providing, we can actually create a comprehension loop where we iterate over the tuples or the list elements.

For the tag, in particular, we would count this, and provide back the tag and its count. So "VB" or the verb appeared three times in a sentence, "to" appear twice, and so on. If you need more details on what these different tags are, because there are lots of them and not all of them can be remembered easily, you can look up the particular tags that start with a letter using the `upenn_tagset` method, and that'll provide the definition for that tag.

But the lemmatized method doesn't use the UPenn tag set, it uses something called `wordnet_tag`, which is a different set of tags that have lowercase a, v, n, instead of uppercase. So it's a smaller set of tags and we need to convert the UPenn tags to the WordNet tags. This is the function that does the conversion every time it needs tags, a tag that starts with "J," it will convert it to "a," or adjectives, tags that start with "V," it converts it to "v" for the verb, and so on.

So now if we apply the lemmatize method for each word in the sentence, the default application, we'll use the noun tag, and the ones that we tune will actually take the tag identified for that particular word, convert it from the UPenn to WordNet tag, and call the function lemmatized with the appropriate tag. Notice the difference you get in simplifying the word forms. In the second case, the words like "started" is converted correctly to the root "start." "Doing" is converted to "do." "Talking" is converted to the root "talk." This significantly simplifies and smallers your vocabulary, makes it easier to process text, and faster, and requires smaller storage.

Back to Table of Contents

Code: Tag a Sentence

In the previous video, Professor Melnikov introduced parts-of-speech tagging and demonstrated some methods you can use to tag the parts of speech in sentences. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will use some of these methods to preprocess new sentences.

All practice exercises are optional and ungraded.

Please complete this activity in the course.

Back to Table of Contents

Read: Identifying Parts of Speech and Phrases

A primary goal of NLP is to understand what the combination of words in a document actually means. Tagging the parts of speech (POS) in a sentence is a powerful preprocessing step because it allows you to more easily lemmatize a document. It can also help you distill the meaning from a document because you can use these labeled parts of speech, along with the structure of natural languages and computing tools, to determine the relationships between words and phrases within documents. You will examine these relationships later in this module. For now, we'll focus on the individual POS and how they can be combined into phrases that follow distinct patterns.

Parts of Speech

Language is complex and nuanced, but the most basic, general parts of speech are:

- *Noun* - A person, place, or thing.
- *Adjective* - A word that describes a noun.
- *Verb* - An action.
- *Adverb* - A word that modifies a verb.
- *Determiner* - A word that indicates the type of noun being used in a sentence.

You will also see several other groups as you move forward in the course. Each of the groups listed here contains many additional subgroups that further differentiate them.

Combining Parts of Speech into Phrases

Words can be combined to form phrases that can be classified based on their patterns. One common type of phrase is a verb phrase which is made up of a verb, followed by "to," followed by a verb. An example of this is the phrase "loved to work" or "dress to impress."

Another common type of phrase is a noun phrase, which is made up of an adjective followed by a noun, and may begin with a determiner. For example, "a lazy dog" and "many cats" are both examples of noun phrases.

Challenges of Automatic Tagging

In English, a word in a sentence can assume different functions, which often depends on the sentence's structure itself. For example, "building" is a verb in "building a house" and a noun in "a tall building." In the phrase "I drive to work," it's not even clear whether "work" is a noun or a verb. This ambiguity might disappear with a larger context in a phrase "I drive to work, where we share ideas." These examples highlight the challenges of automatic tagging. Understanding the different parts of speech you're working with will help you better identify inconsistencies in any analysis you do. It will also help you better understand and interpret your results.

Back to Table of Contents

Watch: Parsing Techniques: Shallow Parsing

Libraries/Methods

`nltk` **Library**

- `pos_tag()` method
- `word.tokenize()` method

`spaCy` **Library**

- `nlp()` object
- `displacy.render()` object

Words in a sentence follow standard grammatical patterns that make meaningful phrases. Once you have both tagged and tokenized text, you can recombine contiguous words into meaningful phrases, which is known as *shallow parsing* or *chunking*. Identifying these phrases can also help you identify entity names, such as locations, dates, or names of people or companies. Watch as Professor Melnikov demonstrates how to shallow parse a sentence, first with regex and then with a machine learning technique using the **spaCy Library**.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

Chunking or shallow parsing recombines the POS-tagged words into meaningful phrases called chunks, such as noun phrase or verb phrase. It is also used to recognize entity names, such as locations, dates, names of persons, companies. A simple and fast algorithm typically applies a regex pattern, but more advanced machine learning methods employ deeper contextual information from a sentence. Chinking, on the other hand, is a complementary technique of specifying token types that we want to be excluded from the output phrase.

Let's look at the code to see this in action. We have to download a few libraries or databases for this to work, but there is a sentence here. "An independent newspaper, the Cornell Daily Sun, was founded by William Ballard Hoyt in 1880." The two methods that make parsing and tagging easy are `word_tokenize` to parse this sentence, we've seen this before, and `pos_tag`, to tag it with the parts of speech tags or labels.

Here we have a list of tuples. Each tuple has the original word in the first position and the corresponding tag in the second position. So "an" is a determiner, and we have an adjective, noun, and so on.

We want to recombine these words into phrases, and we have to specify how this phrase would look like. So in this particular case, we're combining a determiner, an adjective, and noun in that particular order with this defined pattern. So "DT," question mark, "JJ" for adjective, asterisk for any number of characters in between, "NN" for any noun, word forms or tags that starts with "NN" are many different types of nouns, and the asterisk at the end indicates any number of nouns would be captured.

As a result, we get "an independent newspaper" as a phrase, which is correct, "the Cornell Daily Sun," which is a correct phrase. "William Ballard Hoyt," as a person's name and a correct phrase as well. We can build more sophisticated grammar by providing a definition for different types of phrases. "VP" stands for verb phrase, "NP" for noun phrase, and so on.

If we execute this, the result of the regex parsers that will look for these combinations or these structures of POS tags in a sequence of words will come out to be very similar to what we had before, but it will also capture any verb phrases such as was founded that we didn't see before. There's an easier way to do this. All this regex business could be incorporated or placed inside `spaCy`. It is already there packaged for us. If we install this `spaCy` package and download the English database or English model for that package, in just a couple of function calls or method calls, we can produce a very nice result, and we don't have to mess with the regex.

`space.load` will load English model and the NLP will create what's called the sequence of token objects. It's an object in itself and that's where most of the preprocessing is being done. For this doc variable, we can call a method `noun_chunks` and for each one of those chunks, we can identify the text of the chunk, the label of the chunk, and the root word of that chunk.

The result is packaged as a Pandas DataFrame. And we have "an independent newspaper," "the Cornell Daily Sun," and "William Ballard Hoyt" all as noun phrases with the root nouns displayed. Now if you want to take it a step further, you can use `spacy.displacy.render` methods to highlight different organizations, person's names, dates, and other entities in the sentence. All that, thanks to these shallow parsing techniques that are being used underneath.

Back to Table of Contents

Code: Use Shallow Parsing on a Sentence

In the previous video, Professor Melnikov demonstrated for you can use shallow parsing, or chunking, techniques to parse a sentence into smaller word groups that make represent meaningful phrases. In the "Review" section of this ungraded coding exercise, you will use these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will use shallow parsing methods to retrieve phrases from a larger text.

All practice exercises are optional and ungraded.

As you review and practice using these techniques, you'll hone your NLP skills and verify that you understand how to use each technique. Once you have completed both the "Review" and "Optional Practice" sections of this exercise, move on to the next page.

Please complete this activity in the course.

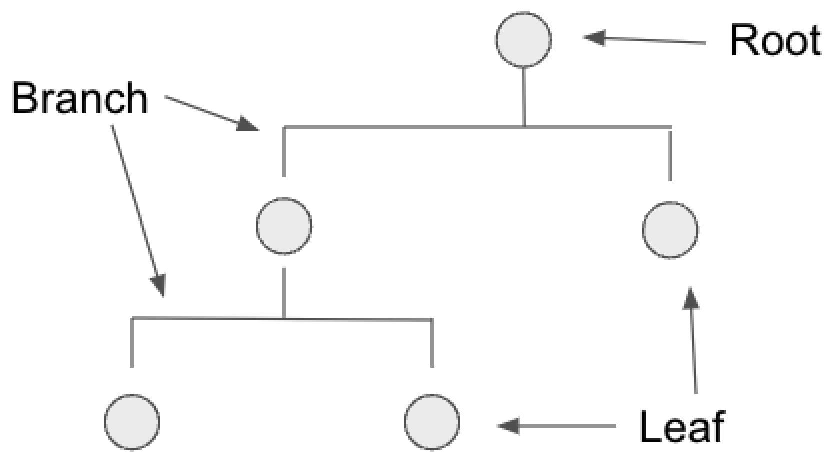
Back to Table of Contents

Read: Using Tree Graphs to Understand Relationships

As you discovered earlier in this course, language is hierarchical. One way to understand hierarchical relationships is to build a *tree graph*. You will discover how to build trees that display relationships of words within a sentence in the following video. Here, you will briefly learn about parts of a tree, define some of the associated terminology, and explore how trees display relationships.

Exploring a Tree

Trees display hierarchical relationships by showing how one category, object, or individual is related to another through the branching pattern of nodes. A tree begins with a *root*, or head, which branches into other nodes. Those nodes can then branch to other nodes. These branching patterns display the unique path between the root node and each other node in the tree. The terminal nodes are called *leaves*.



Tree Variations

Trees can be displayed in different ways. The output from the `nltk` library that you'll examine in the next part of the course looks slightly different from this tree, but describes the hierarchical relationships in the same manner.

Back to Table of Contents

Watch: Parsing Techniques: Dependency Parsing

Libraries/Methods

spaCy Library

`nlp()` object
`displacy()` object
`displacy.render()` method

Pandas Library

`Data.Frame()` method

Dependency parsing is another way that you can extract meaningful phrases from text. In contrast to shallow parsing, which only extracts short phrases of contiguous words, you can use dependency parsing to understand hierarchical relationships of words and phrases in a sentence. This parsing method works by tracking the associations between each word and all of the words that modify it. Then, these relationships can be displayed in a tree graph. Here, Professor Melnikov briefly discusses how you can interpret a tree that describes a sentence, and then demonstrates how you can perform dependency parsing to build a tree of token dependencies.

Note: Consider watching this video in full-screen mode so you can see the Python commands clearly.

Video Transcript

Dependency parsing attempts to recover syntactic relationships of words in a sentence. The output is a tree graph, which starts with a single root vertex, typically a verb. The directed edges are the arrows that connect the head node to the dependent node, which only has one incoming arrow, but might have multiple outgoing arrows. There is a unique path from the root to any other word or phrase.

A dependent is also called a child, a modifier, a subordinate, while the head maybe called a parent, a governor, or a regent in literature. For example, in the noun phrase "lazy dog," the dog is the head, and lazy is the dependent, which modifies the word dog. The entire phrase is a type of dog, not a type of lazy. Dependency trees are used in extracting named entities, such as persons' names, company names, dates, and so on, in relation extraction, in initialization of machine translation models, and in building knowledge graphs, such as the ones you see in Google search on the right-hand side with information drawn from Wikipedia and other sources.

Let's look at the code. Here we'll load the spaCy object and some of the other libraries that we will use. A sentence of concern is, "An independent newspaper. The Cornell Daily Sun, was founded by William Ballard Hoyt in 1880." There's a English model for `spaCy` that we'll need to load before we can parse it. Then we can parse it with an nlp object into tokens, which is essentially a string with many different attributes and properties that are useful to us. So it prints out as a string.

One way we can use this doc object is to iterate over tokens in it. For each token, it can print the token tag, the POS tag, or parts-of-speech tags, the dependency relationship, and the head, which would bring us to that token, iIts texts, and the tag. It prints us at least all tuples of strings. Again, the first element of that tuple or of that list is a tuple with "An" as the dependent, "newspaper" as the head, and the types or the POS tags are determiner and noun and the type of this phrase is determined.

There is the modifier phrase next, "independent newspaper" where newspaper is the noun, and independent is an adjective. All this could be nicely presented in a DataFrame format, where we transpose it with `.T` so that the columns become the rows and the sentences readable horizontally. So the tokens are "an independent newspape." and so on, sequentially presented as it was originally. Then we have a tag row with all these different POS tags: determiner, the adjective, noun, period, and so on. The type of relationship, the head is linked to the dependent word, and the type of the head. Everything that we've seen in the list of tuples above. This can be further visualized with this structure where "founded" is the root. We see all the arrows coming out of "founded," leading us to different phrases or different words. We can display this a little bit larger so that the relationships are legible. So the phrase, "The Cornell Daily Sun" is here where the "Sun" is the head linking us to all these different dependents underneath it in the tree structure.

Back to Table of Contents

Code: Use Dependency Parsing on a Sentence

In the previous video, Professor Melnikov explained dependency parsing and created a tree diagram from a sample sentence. In the "Review" section of this ungraded coding exercise, you will apply these techniques as Professor Melnikov presented them in the video. In the "Optional Practice" section of this exercise, you will practice the same technique on a different sample sentence, and you will interpret the resulting tree diagram.

All practice exercises are optional and ungraded.

Please complete this activity in the course.

Back to Table of Contents

Read: Parsing Techniques: Constituency Parsing

Constituency parsing is another syntactic parsing technique you can use to break a sentence down into its constituent parts. This is useful because often, sentences are more complex than strictly necessary to convey meaning. Complex sentences might make writing or communicating easier, but sometimes in NLP you are most interested in the basic meaning of large groups of sentences. For example, if you want to know what reviewers thought of a specific product, you might be most interested in whether they liked it, not in how descriptive their review was.

You will not be graded on this method in this course, but it is worth exploring it at a very high level so you are aware of it as a technique.

What is Constituency Parsing?

A constituency parser recursively splits a sentence into clauses until each word is in its own leaf of a recursive tree. Consider the following sentence, `S`:

> "The fox ate the berry in the meadow."

To determine the first split, we might break it into two large chunks: a complex noun phrase (`NP`) and a complex verb phrase (`VP`), hoping that this will produce a roughly balanced tree as a result. The regex rule `S:<NP><VP>` accomplishes this, if `NP` and `VP` allow recursive definitions. For example, we could define:

1. a noun phrase as `NP:<DT>?<JJ.>?<NN.*><PP>?`
   A. here the prepositional phrase `PP` is optional as indicated by the following `?`
   B. `DT` is the determiner, `JJ.` is any 3 letter adjective POS tag (starting with `JJ`), `NN.` is any 3 letter noun POS tag starting with `NN`
2. a verb phrase as `VP:(<VB.>|<MD>)<VP>?<NP>?<PP>?<ADJP>?<ADVP>?`
   A. itself contains an optional verb phrase, `VP`, noun phrase, `NP`, and a prepositional phrase, `PP`, adjective phrase, `ADJP`, and adverbial phrase, `ADVP`
3. a prepositional phrase as `PP:PREP[NP]` contains a preposition, `PREP`

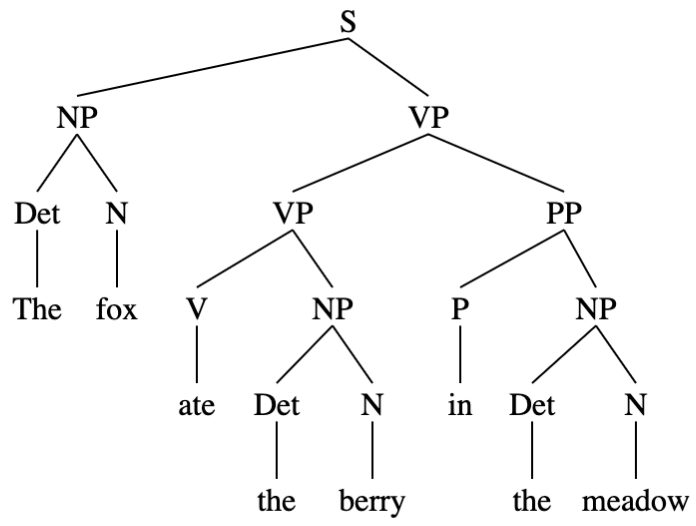
This recursive definition allows great flexibility in tree construction. In fact, there may be multiple trees satisfying the given grammar. Let's take a closer look at how this works in our example sentence.

Using Constituency Parsing on a Sentence

In our example, we might begin with "The fox" as our `NP` and "ate the berry in the meadow" as the `VP`. Check that each phrase satisfies the definition above. For example, "The fox" has a determiner and a noun, but no preposition. Since either of these is optional, the grammar for `NP` above is satisfied.

This split of the full sentence produces two tree branches, each of which can now be evaluated independently with the same grammar specification. We continue splitting each branch, as each of these phrases also have constituent parts. "The fox" is made up of an article ("the") and the noun ("fox").

We can conduct a similar breakdown on the `VP` of the initial sentence, as that first `VP` is composed of a second `VP` ("ate the berry") and a prepositional phrase ("in the meadow"). Each of these phrases can be further broken down until we have a tree that looks something like this:



Once we reach the word-level, there are no more constituent parts. The words are bottom (hanging) leaves of the top-bottom tree we built. Constituent structure matters for preprocessing because of what we will call *substitution*. Assuming the fox is male, we know that we can substitute "he" for "The fox" and the sentence will still be grammatically correct: "He ate the berry in the meadow." We can also substitute "it" for "the berry," or "there" for "in the meadow." Ultimately, the sentence "he ate it there" is not nearly as descriptive, but it is grammatically consistent with the initial sentence.

Constituency Parsing at Scale

This is a simple example, but constituency matters a great deal once we start dealing with more complex sentences. Remember, too, that this all needs to happen at scale, where you don't have simply a handful of complex sentences, but thousands or even millions. To see why this starts to matter, consider this beautifully illustrative example from Speech and Language Processing[1] by Daniel Jurafsky & James H. Martin: "Which flights to Denver depart before the Seattle flight?" In order to answer this question, they write, "We’ll need to know that the questioner wants a list of flights going to Denver, not flights going to Seattle," and we'll need to be able to "parse structure (knowing that to Denver modifies flights, and which flights to Denver is the subject of the depart)..."

Citation

[1]Jurafsky, Dan, and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Pearson, 2022.

Back to Table of Contents

Code: Use Constituency Parsing on a Sentence

On the previous read page, you discovered how to create constituency trees with constituency parsing techniques. In the "Review" section of this ungraded coding exercise, you will examine how to code these techniques. In the "Optional Practice" section of this exercise, you alter some of the sentences and create new trees.

All practice exercises are optional and ungraded.

Please complete this activity in the course.

Back to Table of Contents

Quiz: Identifying Parsing Types

In this module, Professor Melnikov has introduced different types of parsing. Now it is time to test your understanding of them. Select your responses (keeping in mind that a question may have more than one correct answer). Submit your quiz when you have finished.

You may take this quiz as many times as you would like. Your best score will be included in your course grade.

Please complete this quiz in the course.

Back to Table of Contents

Tool: Libraries and Methods for Tagging and Parsing a Document

As you've worked through this module, you practiced many preprocessing techniques that you can use for tagging and parsing a document. These steps can help you break up sentences within a document so that you can begin to understand how words and phrases within a sentence are related to each other. Use this tool to reference the methods you have practiced throughout this module.

Please view and download this tool in the course.

Back to Table of Contents

Assignment: Course Project, Part Three — Tagging and Parsing a Document

In Part Three of the course project, you will write code to complete a set of 12 functions that make up a preprocessing pipeline that you can use to tag, lemmatize, and parse a document, as well as uncover its hierarchical dependencies. The tasks you will perform are based on videos and coding activities in the current module, but also rely on techniques you used in the previous module and on your preparation in Python and basic math. Use the tools from this module to help you complete this part of the course project.

Resources

Use these resources to help you as you complete the course project:

- Project Forum
- **Jupyter Notebook Guide**
- **Module 3 Libraries and Methods**

Instructions

Read the general instructions within this accordion carefully, then follow the instructions specific to this part of the course project in the Jupyter Notebook below.

This exercise is **graded**, and may take ~3 hours to complete.

Please complete this activity in the course.

Back to Table of Contents

Tool: Next Steps — Practice Exercises and Resources

Throughout this course you have begun to apply fundamental NLP skills. Like any other type of computer programming, NLP is a learned concept that must be practiced and honed. At first, it can be hard, awkward and even unnatural, just like learning tennis or chess. The good news is that practice makes perfect: regular problem solving improves your skills.

If you would like further practice with the material covered in this course, you can find additional exercises in the tool on this page for you to complete on your own. These activities are not required for completion of this course, but have been prepared for you to help you verify and advance your understanding.

Please view and download this file in the course.

Back to Table of Contents

Module Wrap-up: Module Wrap-up

Understanding the relationships of words in a sentence can be critical for understanding a sentence's meaning, as well as knowing which words can be safely omitted.

In this module, you explored tagging and parsing sentences, and you practiced both on several text corpora. Performing grammatical analysis involves working with the meaning of words and phrases, which might start to make clear how chatbots can work, or how a search "knows" that you want results related to the noun "dance" and not the verb form. To understand this type of grammar, you need to understand how the words within a sentence are related, and you explored these hierarchical relationships with dependency parsing. Finally, you combined these powerful preprocessing tasks by building a pipeline that can be used on documents.

Back to Table of Contents

Thank You and Farewell

Congratulations on completing Natural Language Processing Fundamentals. I hope that you now have a better understanding of what NLP is, how it can be used in your workplace to address real problems, and the basic techniques you can use to preprocess text.

From all of us at Cornell University and eCornell, thank you for participating in this course.

Oleg Melnikov

Oleg Melnikov

Visiting Lecturer

Computing and Information Science
Cornell University

Back to Table of Contents

Glossary

Please view the glossary in the course.

Back to Table of Contents