M01S01 Introduction to NLP

NLP - Lexical Processing

Introduction [1/15]

Natural language processing or text analytics plays a very vital role in today's era because of the sheer volume of text data that users generate around the world on digital channels such as social media apps, e-commerce websites, blog posts, etc. The first session of this module covers:

- Industry applications of text analytics
- Understanding textual data
- Regular expressions

Module-1 has lots of hands-on practice (especially while learning regular expressions), while Module-2 and Module-3 will focus more on concepts, algorithms and applications such as building POS taggers, NER systems, unsupervised analysis (twitter opinion mining), and finally semantics.

Terms 'natural language processing' and 'text analytics' will be used interchangeably.

Prerequisites

Knowledge of the previous courses on Statistics and ML.

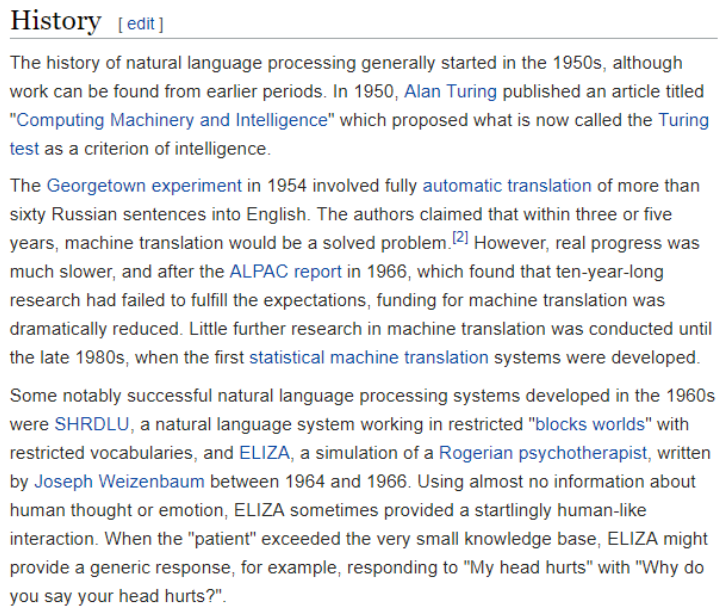
NLP: Areas of Application [2/15]

Before diving into what is textual data and how to handle it, let's take a look at the different industries that make use of text data to solve important problems.

Understanding Text [3/15]

NLP has a pretty wide array of applications - it finds use in many fields such as social media, banking, insurance and many more.

The data that we get while performing analytics on text usually is a sequence of words. Something like the text shown in the image below:



History [edit]

The history of natural language processing generally started in the 1950s, although work can be found from earlier periods. In 1950, Alan Turing published an article titled "Computing Machinery and Intelligence" which proposed what is now called the Turing test as a criterion of intelligence.

The Georgetown experiment in 1954 involved fully automatic translation of more than sixty Russian sentences into English. The authors claimed that within three or five years, machine translation would be a solved problem.[2] However, real progress was much slower, and after the ALPAC report in 1966, which found that ten-year-long research had failed to fulfill the expectations, funding for machine translation was dramatically reduced. Little further research in machine translation was conducted until the late 1980s, when the first statistical machine translation systems were developed.

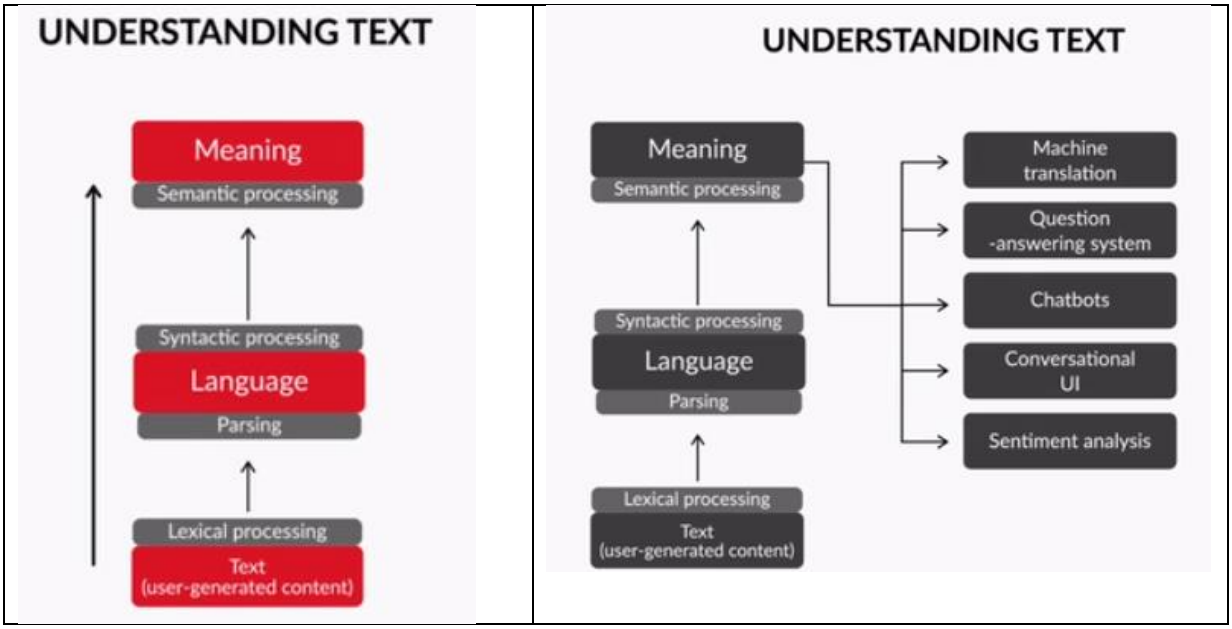
Some notably successful natural language processing systems developed in the 1960s were SHRDLU, a natural language system working in restricted "blocks worlds" with restricted vocabularies, and ELIZA, a simulation of a Rogerian psychotherapist, written by Joseph Weizenbaum between 1964 and 1966. Using almost no information about human thought or emotion, ELIZA sometimes provided a startlingly human-like interaction. When the "patient" exceeded the very small knowledge base, ELIZA might provide a generic response, for example, responding to "My head hurts" with "Why do you say your head hurts?".

A piece of text on NLP. How to translate this data to a different language using an algorithm?

System should be able to take the raw unprocessed data shown above, break the analysis down into smaller sequential problems (a pipeline), and solve each of those problems individually. The individual problems could be as simple as breaking the data into sentences, words etc. to something as complex as understanding what a word means, based on the words in its “neighbourhood”.

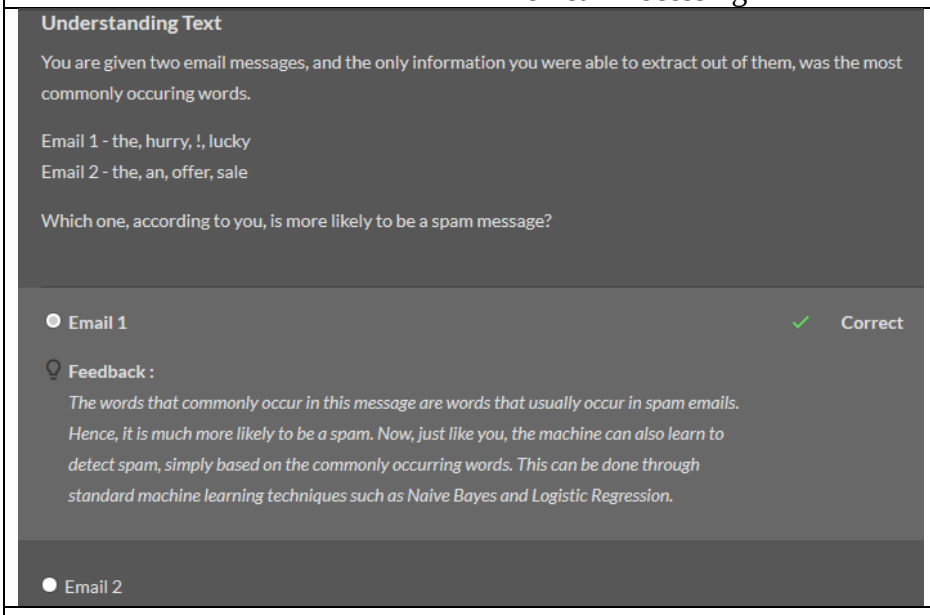
This course covers all the different “steps” generally undertaken on the journey from data to meaning. This journey can be divided roughly into three parts.

1. Lexical Processing
2. Syntactic Processing
3. Semantic Processing



After looked at the areas of text analytics, let's take a look at what does it mean to understand the text, i.e., how to approach a problem that deals with text.

Lexical Processing



Syntactic Processing

Parsing (Parts of speech)

Carry the water can to the well
I can play the next match

Carry the water can to the well
I can play the next match

Understanding Text

Siri, Apple's personal assistant, converted its user's command to text and performed a lexical analysis to find the most important words in the sentence. These were:

'Book', 'Good' and 'Dentist'

Feeling the need to analyse the command further, it performed a syntactic analysis on the command and noted that:

'Book' is a verb
'Good' is an adjective
'Dentist' is a noun

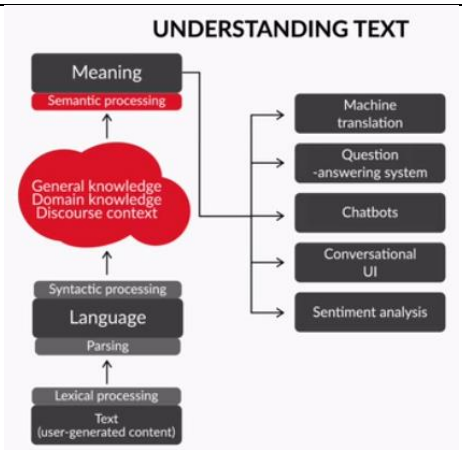
Based on these three words, and the information you get after syntactic analysis, what task does the user want Siri to perform?

- Book a good dentist ✓

  Feedback :
  'Book' is a verb here, signifying that the action of "booking" something is going on here. Hence, the other two interpretations, in which book is a noun, are proved incorrect.

- Look for a book on good dentists
- Look for a good book on dentists

Semantic processing



Understanding Text

The following image shows the results obtained after a few lines are run through google translate. However, in all of the cases, the translation is incorrect.

English	Hindi
I cracked the interview	मैंने साक्षात्कार को तोड़ दिया
The doctor examined my pupils	डॉक्टर ने मेरे विद्यार्थियों की जांच की
I watched a movie yesterday and it had an impact on me	मैंने कल एक फिल्म देखी और इसका मुझ पर असर पड़ा
I watched a movie yesterday and it was awful	मैंने कल एक फिल्म देखी और यह भयानक था

Can you explain why each one of these errors happened? Was it because of one of the following reasons, or because of some other reason? Why do you think that is the reason for the mistake?

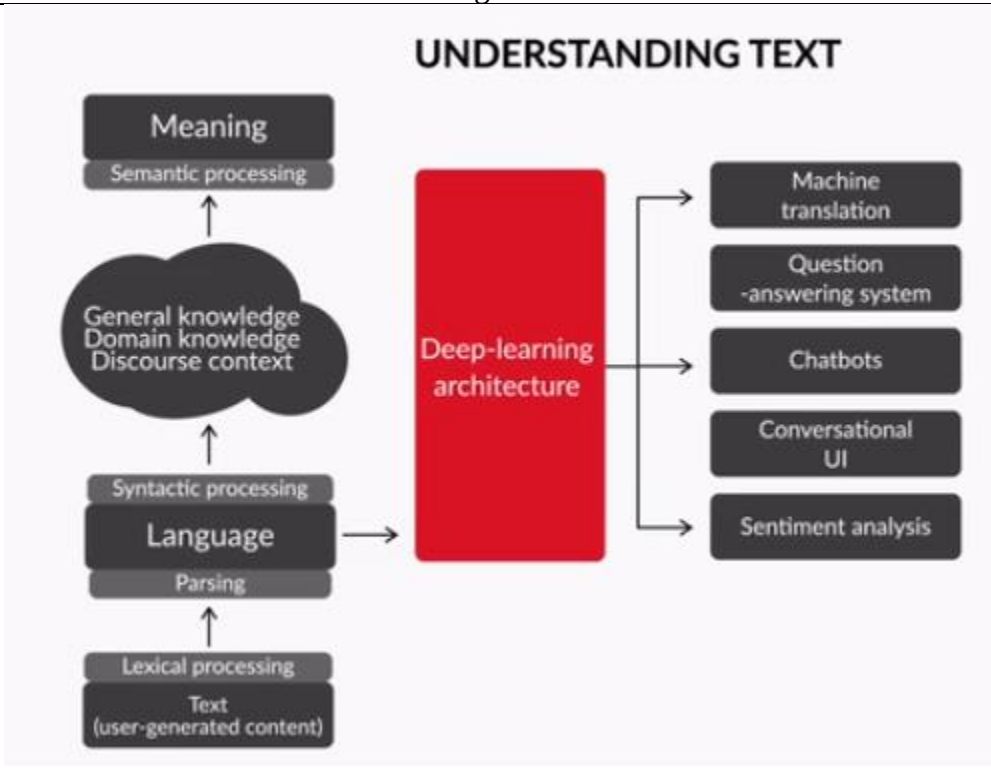
Reason 1 - Lack of general knowledge about Hindi/English.
Reason 2 - Lack of domain knowledge, i.e., knowledge of the subject matter.
Reason 3 - Lack of knowledge of the discourse context

These errors mainly happened for third reason, which is lack of knowledge of discourse context

<table>
<tr><td></td><td>Suggested Answer

We hope that this question has given you an idea of where we stand in language translation. This result also must have given a good idea about how promising are the state of the art results and what are the areas of improvement in language translation.

Here, we have used Google's English to Hindi translator to demonstrate the results shown to you. However, we're sure that other languages (there are so many!) have their own intricacies and peculiarities which come into play when we translate.</td></tr>
<tr><td colspan="2">Combined approach using Deep learning

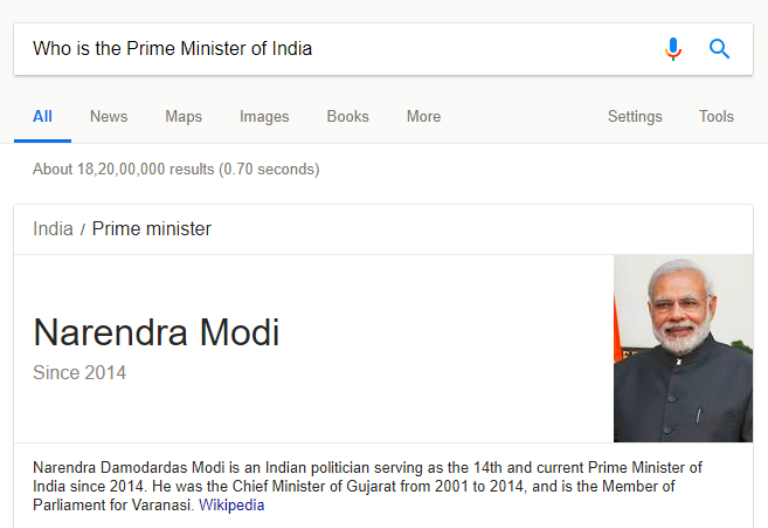
Emergent area</td></tr>
<tr><td colspan="2"> 
</td></tr>
</table>

In the wikipedia example, data (textual data) looked like a collection of characters, that machines can't make any sense of. Starting with this data, we move according to the following steps -

- **Lexical Processing:** Convert the raw text into words and, depending on application's needs, into sentences or paragraphs as well.
    1. An email containing and represented by words such as lottery, prize and luck is likely to be a spam email.
    2. In general, the group of words contained in a sentence gives a good idea of what it means. Many more processing steps are usually undertaken in order to make this group more representative of the sentence, for example, cat and cats are considered to be the same word. In general, we can consider all plural words to be equivalent to the singular form.
    3. For a simple application like spam detection, lexical processing works just fine, but it is usually not enough in more complex applications, like machine translation.

Sentences “My cat ate its third meal” and “My third cat ate its meal”, have very different meanings, but lexical processing will treat them as equal, as the “group of words” in is the same. Hence, we need a more advanced system of analysis.

- **Syntactic Processing:** Next step after lexical analysis is to extract more meaning from the sentence, by using its syntax. Instead of only looking at the words, look at the syntactic structures, i.e., the grammar of the language to understand what the meaning is.
    1. Differentiate between the subject and the object, i.e., identifying person performing the action and person affected by it. “Ram thanked Shyam” and “Shyam thanked Ram” are sentences with different meanings as the action of ‘thanking’ is done by Ram in one case and by Shyam in another, A syntactic analysis based on a sentence’s subjects and objects, will be able to make this distinction.
    2. In various other ways, syntactic analyses can enhance our understanding. A question answering system that is asked the question “Who is the Prime Minister of India?”, will perform much better, if it understands that the words “Prime Minister” are related to “India”. It can then look up in its database and provide the answer.



Syntactic Processing

- **Semantic Processing**: Lexical and syntactic processing don't suffice when it comes to building advanced NLP applications such as language translation, chatbots etc. The machine, after the two steps given above, will still be incapable of understanding the meaning of the text. Such an incapability can be a problem for, say, a question answering system, as it may be unable to understand that PM and Prime Minister mean the same thing. Hence, when somebody asks it the question, “Who is the PM of India?”, it may not be able to give an answer unless it has a separate database for PMs, as it won’t understand that the words PM and Prime Minister are the same. You could store the answer separately for both the variants of the meaning (PM and Prime Minister), but how many of these meanings can be stored manually? At some point, machine should be able to identify synonyms, antonyms, etc. on its own.
    1. This is typically done by inferring the word’s meaning to the collection of words that usually occur around it. So, if the words, PM and Prime Minister occur very

frequently around similar words, then you can assume that the meanings of the two words are similar as well.

2. In fact, this way, the machine should also be able to understand other semantic relations. For example, it should be able to understand that the words “King” and “Queen” are related to each other and that the word “Queen” is simply the female version of the word “King”. Also, both these words can be clubbed under the word “Monarch”. You can probably save these relations manually, but it will help you a lot more, if you can train your machine to look for the relations on its own and learn them. Exactly how that training can be done, is something we’ll explore in the third module.

Once you have the meaning of the words, obtained via semantic analysis, you can use it for a variety of applications. Machine translation, chatbots and many other applications require a complete understanding of the text, right from the lexical level to the understanding of syntax to that of meaning. Hence, in most of these applications, lexical and semantic processing simply form the “pre-processing” layer of the overall process. In some simpler applications, only lexical processing is also enough as the pre-processing part.

Understanding Text

Deep learning architectures short-circuit the conventional NLP path. Which stage is entirely eliminated from the three stage process?

○ Lexical processing

◉ **Syntactic processing** ✗ **Incorrect**

Feedback:
There is still need of some level of syntactic processing even if someone uses deep learning models.

○ **Semantic processing** ✓ **Correct**

Feedback:
The need of semantic processing is eradicated because deep learning models don't require need to feed had crafted features extracted by semantic processing.

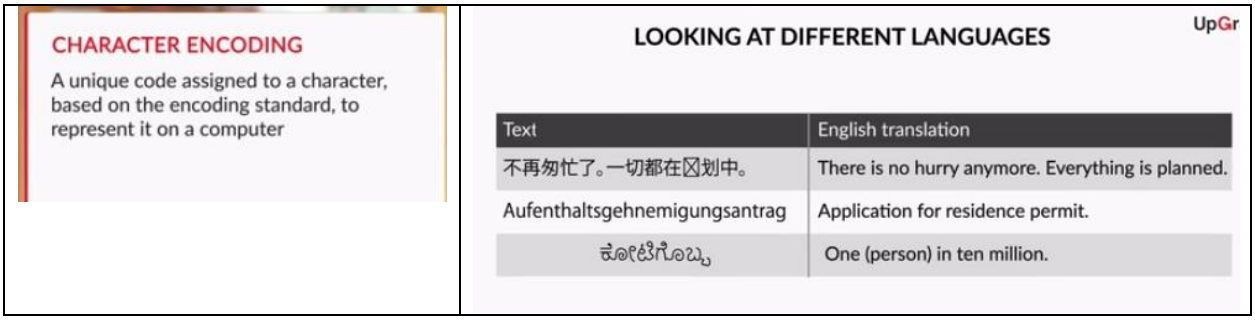
○ All of the above

This gives a basic idea of the process of analysing text and understanding the meaning behind it.

Text Encoding [4/15]

Data is being collected in many languages, but this course covers text analysis for the English language. These techniques might not work for other languages.

How characters of different languages are stored on computers.



It is not necessary that we always get to work with the English language. With so many languages in the world and internet being accessed by many countries, there is a lot of text in non-English languages. To work with non-English text, we need to understand how all the other characters are stored.

Computers could handle numbers directly and store them on registers (the smallest unit of memory on a computer). But they couldn't store the non-numeric characters as is. The alphabets and special characters were to be converted to a numeric value first before they could be stored.

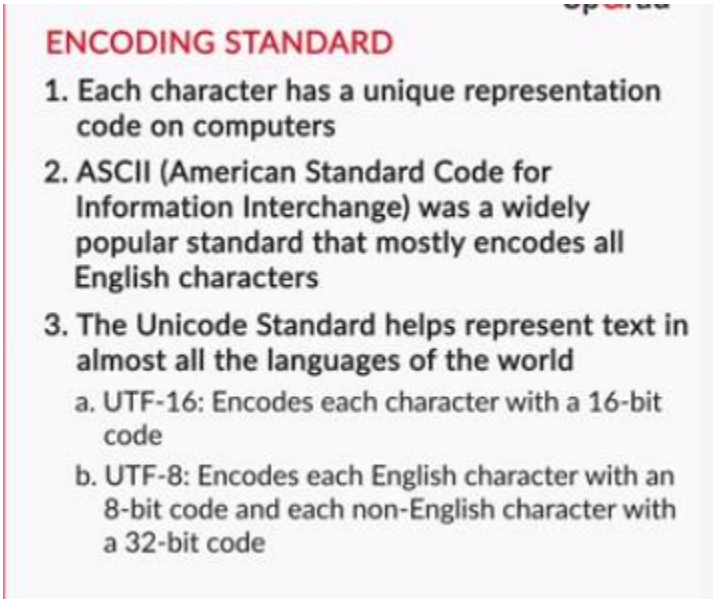
Hence, the concept of **encoding** came into existence. All the non-numeric characters were encoded to a number using a code. Also, the encoding techniques had to be standardised so that different computer manufacturers won't use different encoding techniques.

The first encoding standard that came into existence was the **ASCII (American Standard Code for Information Interchange) standard**, in 1960. ASCII standard assigned a unique code to each character of the keyboard which was known as **ASCII code**. ASCII code of the alphabet 'A' is 65 and that of the digit zero is 48. Since then, there have been several revisions made to the codes to incorporate new characters that came into existence after the initial encoding.

When ASCII was built, English alphabets were the only alphabets that were present on the keyboard. With time, new languages began to show up on keyboard sets which brought new characters. ASCII became outdated and couldn't incorporate so many languages. A new standard **Unicode standard** supports all languages - both modern and the older ones.

For someone working on text processing, knowing how to handle encodings becomes crucial. Before even beginning with any text processing, you need to know what kind of encoding the text has and if required, modify it to another encoding format.

In this segment, you'll understand how encoding works in Python and the different types of encodings that you can use in Python.



To get a more in-depth understanding of Unicode in Python, official Python guide at here.

There are two most popular encoding standards:

1. American Standard Code for Information Interchange (ASCII)
2. Unicode
    - UTF-8
    - UTF-16

Relation between ASCII, UTF-8 and UTF-16. Below table shows the ASCII, UTF-8 and UTF-16 codes for two symbols - the dollar sign and the Indian rupee symbol.

Symbol	ASCII code		UTF-8 code		UTF-16 (BE) code	
	Binary	Hex	Binary	Hex	Binary	Hex
$ (Dollar sign)	00100100	24	00100100	24	00000000 00100100	0024
₹ (Indian Rupee sign)	Doesn't exist	Doesn't exist	11100010 10000010 10111001	E282B9	00100000 10111001	20B9

Types of encoding

UTF-8 offers a big advantage in cases when the character is an English character or a character from the ASCII character set. Also, while UTF-8 uses only 8 bits to store the character, UTF-16 (BE) uses 16 bits to store it, which looks like a waste of memory.

However, in the second case, a symbol is used which doesn't appear in the ASCII character set. For this case, UTF-8 uses 24 bits, whereas UTF-16 (BE) only uses 16. Hence the storage advantages offered by UTF-8 is reversed and becomes a disadvantage here. Also, the advantage UTF-8 offered previously by being same as the ASCII code is also not of use here, as ASCII code doesn't even exist for this case.

The default encoding for strings in python is Unicode UTF-8. You can also look at this UTF-8 encoder-decoder to look how a string is stored. Note that, the online tool gives you the hexadecimal codes of a given string.

Try this code in your Jupyter notebook and look at its output. Feel free to tinker with the code.

```
# create a string

amount = u"₹50"
print'Default string: ', amount, '\n', 'Type of string', type(amount), '\n')
# encode to UTF-8 byte format
amount_encoded = amount.encode('utf-8')
print('Encoded to UTF-8: ', amount_encoded, '\n', 'Type of string', type(amount_encoded), '\n')
# sometime later in another computer...
# decode from UTF-8 byte format
amount_decoded = amount_encoded.decode('utf-8')
print('Decoded from UTF-8: ', amount_decoded, '\n', 'Type of string', type(amount_decoded), '\n')
```

In the next segment, you'll learn about **regular expressions** which are a must-know tool for anyone working in the field of natural language processing and text analytics.

Regular expressions: Quantifiers - I [5/15]

Regular expressions, also called **regex**, are very powerful programming tools that are used for a variety of purposes such as feature extraction from text, string replacement and other string manipulations. To become a master at text analytics, being proficient with regular expressions is a must-have skill.

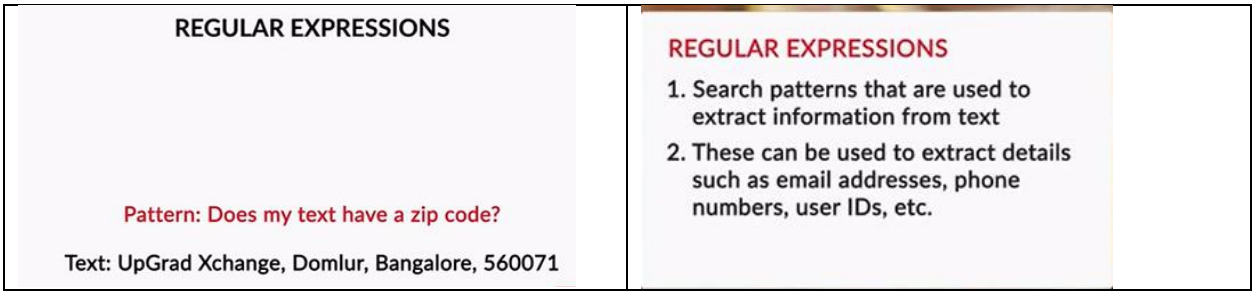
A regular expression is a set of characters, or a **pattern**, which is used to find substrings in a given string.

To extract all the hashtags from a tweet. It has a fixed pattern to it - a pound ('#') character followed by a string, like #mumbai, #bangalore, #upgrad. To achieve this task, provide this pattern and the tweet (pattern is - any string starting with #). Another example is to extract all the phone numbers from a large piece of textual data.

Any pattern in any string can be easily extracted, substituted and do all kinds of other string manipulation operations using regular expressions.

Learning regex means learning how to identify and define these patterns.

Regulars expressions are a language since they have their own compilers.



To use regex, Import module re and use 're.search()' function which detects whether the given regular expression pattern is present in the given input string. It returns a RegexObject if the pattern is found in the string, else it returns a None object.

match.start() and match.end() return index of starting and ending position of the match found.

Quantifiers allow to mention and have control over number of times character(s) in pattern should occur. Four types of quantifiers:

1. The '?' operator - Zero or one (tells whether a pattern is absent or present)
2. The '*' operator - Zero or more
3. The '+' operator - One or more
4. The '{m, n}' operator - Matches if a character is present from m to n number of times

The '?' can be used to make preceding character of pattern to be an **optional character in the string**. To write a regex that matches both 'car' and 'cars', the corresponding regex will be 'cars?'. 'S' followed by '?' means that 's' can be absent or present, i.e. it can be present zero or one time.

A '*' quantifier matches the preceding character **any number of times**.

Regular Expressions: Quantifiers - II [6/15]

- The '?' matches the preceding character zero or one time. It is generally used to mark the **optional presence** of a character.
- The '*' quantifier is used to mark the presence of the preceding character **zero or more times.**

Now, learn about a new quantifier - the '+'. It matches the preceding character **one or more times**. That means the preceding character must be present **at least once** for the pattern to match the string.

Only difference between '+' and '*' is that '+' needs a character to be present **at least once**, while the '*' does not.

Quantifiers

Let's say you have this pattern - 'abc+d'. Which of the following strings will not be matched by this pattern?

- abcd
- abccd
- **abd** Co

Feedback :

The '+' quantifier will match the string if the preceding character is present one or more times.
Here, the character 'c' is not present which doesn't meet the condition.

- abccccccccd

Following quantifiers until now:

- '?': Optional preceding character
- '*': Match preceding character zero or more times
- '+': Match preceding character one or more times (i.e. at least once)

But how to specify a regex when looking for a character that appears, say, exactly 5 times, or between 3-5 times? It's not doable using the quantifiers above.

The next quantifier will help specify occurrences of the preceding character **a fixed number of times**.

{m,n}: Matches if a character is present from m to n number of times

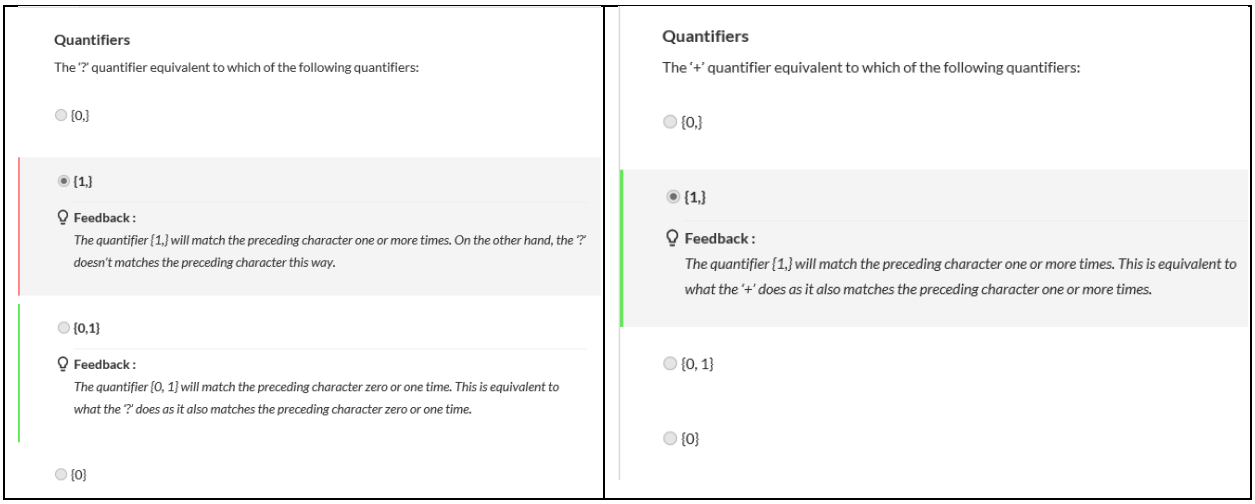
```
In [ ]: print(find_pattern("aabbbbbbc","ab{3,5}")) # return if a is followed by 'b' between 3-5 times

In [ ]: print(find_pattern("aabbbbbbc", "ab{7,10}"))  # return if a is followed by 'b' between 7-10 times

In [ ]: # {n}: Matches if a character is present exactly n number of times
        print(find_pattern("aabbbbbbc", "ab{4}"))   # return if a is followed by 'b' 4 times

In [ ]: print(find_pattern("aabbbbbbc", "ab{10}"))   # return if a is followed by 'b' 10 times
```

There are four variants of the quantifier:

1. **{m, n}**: Matches the preceding character ‘m’ times to ‘n’ times.
2. **{m, }**: Matches the preceding character ‘m’ times to infinite times, i.e. there is no upper limit to the occurrence of the preceding character.
3. **{, n}**: Matches the preceding character from zero to ‘n’ times, i.e. the upper limit is fixed regarding the occurrence of the preceding character.
4. **{n}**: Matches if the preceding character occurs exactly ‘n’ number of times.

While specifying {m,n} notation, avoid using a space after the comma, i.e. use {m,n} rather than {m, n}. This quantifier can replace the ‘?’, ‘*’ and the ‘+’ quantifier. That is because:

- '?' is equivalent to zero or once, or {0, 1}
- '*' is equivalent to zero or more times, or {0, }
- '+' is equivalent to one or more times, or {1, }



Python’s pattern matching is case sensitive by default.

Comprehension: Regular Expressions [7/15]

Use of **whitespace**. Till now, we didn't use a whitespace character which comprises of a single space, multiple spaces, tab space or a newline character (a vertical space). More about multiple spaces in a computer here. Spaces can be used in regular expression normally.

These whitespaces will match the corresponding spaces in the string. For example, the pattern ' +', i.e. a space followed by a plus sign will match one or more spaces. Similarly, spaces useable with other characters inside the pattern. The pattern, 'James Allen' allows to look for the name 'James Allen' in any given string.

In character classes section, more about the different types of spaces that one can use. Whitespaces are used extensively when used inside character sets.

The **parentheses**. If we put parentheses around some characters, the quantifier will look for repetition of the **group of characters** rather than just looking for repetitions of the preceding character. This is called **grouping**.

Pattern '(abc){1, 3}' will match the following strings:

- abc
- abcabc
- abcabcabc

Pattern (010)+ will match:

- 010
- 010010
- 010010010, and so on.

Pattern '(23)+(78)+' will match

- a string where '23' occurs one or more times followed by occurrence of '78' one or more times.

The **pipe operator** denoted by '|' and used as an OR operator. Always used inside the parentheses. Pattern '(d|g)one' will match both the strings - 'done' and 'gone'. Place inside the parentheses can be either 'd' or 'g'.

Pattern '(ICICI|HDFC) Bank' will match the strings 'ICICI Bank' and 'HDFC Bank'. Use quantifiers after the parentheses as usual even when there is a pipe operator inside. There can be an infinite number of pipe operators inside the parentheses. The pattern '(0|1|2){2} means 'exactly two occurrences of either of 0, 1 or 2', and it will match these strings - '00', '01', '02', '10', '11', '12', '20', '21' and '22'.

Lastly, to mention **special characters** such as '?', '*', '+', '(', ')', '{', etc. in regex. These have special meanings when they appear inside a regex pattern.

To extract all the questions from a document assuming all questions end with a question mark - '?'. Use **escape sequences**, denoted by a backslash '\', to escape the special meaning of the special characters. To match a question mark literally, use '\?' (escaping the character).

To match the addition symbol in a string, escape the '+' operator and the pattern to be used is '\+'.

To extract '(78)' from string - 'Dravid, who scored 56(78), was bowled by Brett Lee after lunchtime', use '\(78\)'. The escape character is preceded by the character that needs to escape.

To match the '\' character literally, use the pattern '\\' to escape the backslash.

Regex flags. A flag has a special meaning. To ignore the case of the text pass the *'re.I'* flag. Flag with the syntax *re.M* enables to search in multiple lines (in case the input text has multiple lines). Pass all these flags in the re.search() function.

```
re.search(pattern, string, flags=re.I | re.M)
```

The *re.compile()* function stores regex pattern in the cache memory for slightly faster searches. Pass the regex pattern to re.compile() function. Difference between searching with the compile function and without the compile function.

```
# without re.compile() function
result = re.search("a+", "abc")

# using the re.compile() function
pattern = re.compile("a+")
result = pattern.search("abc")
```

Regular Expressions: Anchors and Wildcard [8/15]

Anchors are used to specify the start and end of the string.

Anchors

```
In [19]: # '^': Indicates start of a string
         # '$': Indicates end of string

         print(find_pattern("James", "^J"))   # return true if string starts with 'J'
         print(find_pattern("Pramod", "^J"))  # return true if string starts with 'J'
         print(find_pattern("India", "a$"))   # return true if string ends with 'c'
         print(find_pattern("Japan", "a$"))   # return true if string ends with 'c'

         <_sre.SRE_Match object; span=(0, 1), match='J'>
         Not Found!
         <_sre.SRE_Match object; span=(4, 5), match='a'>
         Not Found!
```

Two anchors characters: '^' (Start of the string) and '$' (End of the string).

The character followed by the '^' in the pattern should be the first character of the string.

The character that precedes the '$' in the pattern should be the last character in the string.

Both the anchors can be specified in a single regular expression itself. RegEx pattern '^01*0$' will match any string that starts and end with zeroes with any number of 1s between them like '010', '0110', '01111111110' and even '00' ('*' matches zero or more 1s). But it will not match the string '0' because there is only one zero in this string and in the Pattern, we have specified that there needs to be two 0s, one at the start and one at the end.

One special character, the **wildcard character,** in regex, '.' (dot) character, acts as a placeholder and can match any character (literally!) in the given input string.

We don't always know the letter that we want to repeat. In such situations, use the wildcard.

The wildcard comes handy in many situations. It can be followed by a quantifier which specifies that any character is present a specified number of times.

To write a regex pattern that should match a string that starts with four characters, followed by three 0s and two 1s, followed by any two characters. The valid strings can be abcd00011ft, jkds00011hf, etc. The pattern would be '.{4}0{3}1{2}.{2}'. Or use '....00011..' where the dot acts as a placeholder; Anything can sit on the place of the dot. Both are correct regex patterns.

Regular Expressions: Characters Sets [9/15]

Until now, we used the actual letters (such as ab, 23, 78, etc.) or the wildcard character in regex patterns. There was no way of telling that the preceding character is a digit, or an alphabet, or a special character, or a combination of these.

To match phone numbers in a large document, we know that the numbers may contain hyphens, plus symbol etc. (e.g. +91-9930839123), but no alphabets. We need to somehow specify that only numerics and some other symbols needed but avoid alphabets.

To handle such situations, use **character sets**.

Character sets

Pattern	Matches
[abc]	Matches either an a, b or c character
[abcABC]	Matches either an a, A, b, B, c or C character
[a-z]	Matches any characters between a and z, including a and z
[A-Z]	Matches any characters between A and Z, including A and Z
[a-zA-Z]	Matches any characters between a and z, including a and z ignoring cases of the characters
[0-9]	Matches any character which is a number between 0 and 9

Character sets provide lot more flexibility than just typing a wildcard or the literal characters. Character sets can be specified with or without a quantifier. When no quantifier succeeds the character set, it matches only one character and the match is successful only if the character in the string is one of the characters present inside the character set. For example, the pattern '[a-z]ed' will match strings such as 'ted', 'bed', 'red' and so on because the first character of each string - 't', 'b' and 'r' - is present inside the range of the character set.

On the other hand, when we use a character set with a quantifier, such as in this case - '[a-z]+ed', it will match any word that ends with 'ed' such as 'watched', 'baked', 'jammed', 'educated' and so on. A character set is like a wildcard because it can also be used with or without a quantifier. It gives more power and flexibility!

Note that **a quantifier loses its special meaning** when it is present inside the character set. Inside square brackets, it is treated as any other character.

We can also mention a whitespace character inside a character set to specify one or more whitespaces inside the string. The pattern [A-z] can be used to match the full name of a person. It includes a space, so it can match the full name which includes the first name, a space, and the last name of the person.

To match every other character other than the one mentioned inside the character set, use the caret operator.

The '^' has two use cases. It can be used outside a character set to specify the start of a string as an **anchor**. Another use is inside a character set, where it acts as a **complement operator**, i.e. it specifies that it will match any character other than the ones mentioned inside the character set.

The pattern [0-9] matches any single digit number. On the other hand, the pattern '[^0-9]' matches any single digit character that is not a number.

Meta Sequences

When working with regex, we commonly use sets to match only digits, only alphabets, only alphanumeric characters, only whitespaces, etc. Meta-sequences are a shorthand way to write commonly used character sets in regex.

Meta sequences

Pattern	Equivalent to
\s	[\t\n\r\f\v]
\S	[^ \t\n\r\f\v]
\d	[0-9]
\D	[^0-9]
\w	[a-zA-Z0-9_]
\W	[^a-zA-Z0-9_]

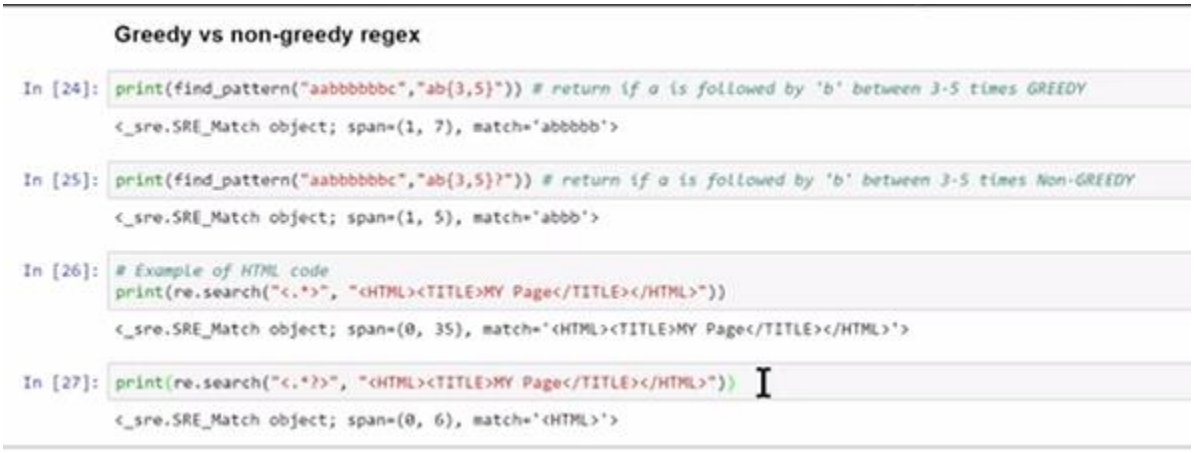
Those were the commonly used meta-sequences.

1. Use them **without the square brackets**. Pattern '\w+' will match any alphanumeric character.
2. Use **inside the square brackets**. Pattern '[\w]+' is same as '\w+'. But meta-sequences inside a square bracket are commonly used along with other meta-sequences. The '[\w\s]+' matches both alphanumeric characters and whitespaces.

Greedy versus Non-greedy Search [10/15]

The regex greedily tries to look for the longest pattern possible in the string. The pattern 'ab{2,5}' to match the string 'abbbbb' will look for the maximum number of occurrences of 'b' (in this case 5).

By default, a regular expression is greedy in nature. There is another approach called the non-greedy approach, also called the lazy approach, where the regex stops looking for the pattern once a particular condition is satisfied.



Suppose the string ‘3000’. Regex ‘30+’ will match the entire string, i.e. ‘3000’. This is the greedy way. In the non-greedy technique, it will only match ‘30’ because it still satisfies the pattern ‘30+’ but stops as soon as it matches the given pattern.

It is important to not confuse the greedy approach with matching multiple strings in a large piece of text - these are different use cases. Similarly, the lazy approach is different from matching only the first match.

Take the string ‘One batsman among many batsmen.’.

Run the patterns ‘bat*’ and ‘bat*?’ on this text, the pattern ‘bat*’ will match the substring ‘bat’ in ‘batsman’ and ‘batsmen’ while the pattern ‘bat*?’ will match the substring ‘ba’ in ‘batsman’ and ‘batsmen’. The pattern ‘bat*’ means look for the term ‘ba’ followed by zero or more ‘t’s so it greedily looks for as many ‘t’s as possible and the search ends at the substring ‘bat’. The pattern ‘bat*?’ will look for as few ‘t’s as possible. Since ‘*’ indicates zero or more, the lazy approach stops the search at ‘ba’.

To use a pattern in a non-greedy way, put a question mark at the end of any of the following quantifiers:	The lazy quantifiers of the above greedy quantifiers are:
• * • + • ? • {m, n} • {m,}	• *? • +? • ?? • {m, n}? • {m,}? • {, n}?

• {, n} • {n}	• {n}?

Commonly Used RE Functions [11/15]

While 're.search()' function or RE module is a very common function used while working with regular expressions in python, it is not the only function to work with regular expressions.

Four more functions in this section.

The five most important re functions required to be used most of the times are

1. match() Determine if the RE matches at the beginning of the string
2. search() Scan through a string, looking for any location where this RE matches
3. finall() Find all the substrings where the RE matches, and return them as a list
4. finditer() Find all substrings where RE matches and return them as asn iterator
5. sub() Find all substrings where the RE matches and substitute them with the given string

The match function will only match if the pattern is present at the very start of the string. On the other hand, the search function will look for the pattern starting from the left of the string and keeps searching until it sees the pattern and then returns the match.

The 're.match()' function

Given the pattern '\d+', on which of the following strings will the 're .match()' function return a non-empty match. (Multiple choices may be correct)

☐ dummy_user_001

☑ **100_crores** **Correct**

Feedback :
re.match() returns a non-empty match only if the match is present at the very beginning of the string. The pattern is present in the string right at the start. The substring '100' satisfies the condition.

☑ **78** **Correct**

Feedback :
re.match() returns a non-empty match only if the match is present at the very beginning of the string. The pattern is present in the string right at the start. '78' has digits present at the very start as it only has digits.

☐ UpGrad

The *re.sub()* function is used to substitute a substring with another substring of your choice.

Regular expression patterns can help find the substring in a given corpus of text to substitute with another string. For example, to replace the American spelling 'color' with the British spelling 'colour'. Similarly, the *re.sub()* function is very useful in text cleaning. It can be used to replace all the special characters in a given string with a flag string, say, SPCL_CHR, just to represent all the special characters in the text.

```
In [31]: ## Example usage of the sub() function. Replace Road with Rd.

         street = '21 Ramkrishna Road'
         print(re.sub('Road', 'Rd', street))

         21 Ramkrishna Rd

In [32]: print(re.sub('R\w+', 'Rd', street))

         21 Rd Rd
```

re.sub() function is used to substitute a part of string using a regex pattern. It is often the case when we want to replace a substring of our string where the substring has a particular pattern that can be matched by the regex engine and then it is replaced by the re.sub() command.

Note that, this command will replace all the occurrences of the pattern inside the string. For example, below code will change the string to: "My address is XXB, Baker Street":

pattern = "\d"
replacement = "X"
string = "My address is 13B, Baker Street"

re.sub(pattern, replacement, string)

The next set of functions lets us search the entire input string and return all the matches, in case there are more than one present.

```
text = 'Diwali is a festival of lights, Holi is a festival of colors!'
pattern = 'festival'
for match in re.finditer(pattern, text):
    print('START -', match.start(),end="")
    print('END -', match.end())
```

```
START - 12END - 20
START - 42END - 50
```

```
[34]: # - Example usage of Findall(). In the given URL find all dates
url= "http://www.telegraph.co.uk/formula-1/2017/10/28/mexican-grand-prix-2017-time-does-start-tv-channel-odds-lewis1/2017/05/12/"
date_regex = '/(\d{4})/(\d{1,2})/(\d{1,2})/'
print(re.findall(date_regex, url))
```

```
[('2017', '10', '28'), ('2017', '05', '12')]
```

The match and search command return only one match. But, often we need to extract all the matches rather than only the first match.

In a huge corpus of text, to extract all the dates, use the finditer() function or the findall() function to extract the results. The result of the findall() function is a list of all the matches and the finditer() function is used in a 'for' loop to iterate through each separate match one by one.

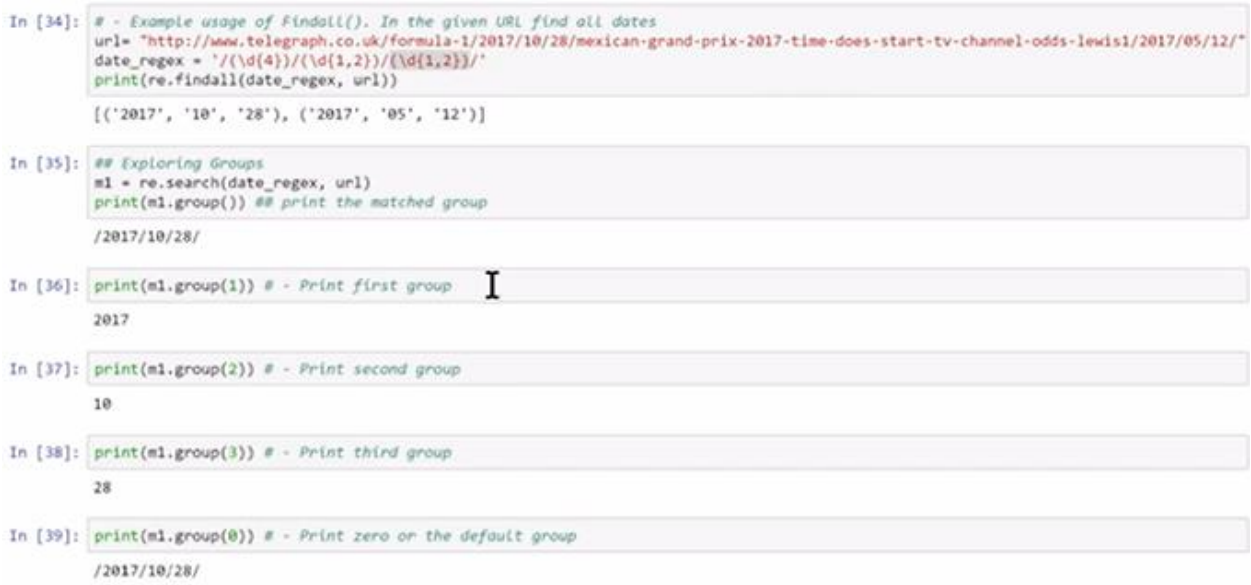
Regular Expressions: Grouping [12/15]

Grouping can be done to **extract sub-patterns out of a larger pattern**. To extract only the year from dates in textual data, use a regex pattern with **grouping** to match dates and then extract the component elements such as the day, month or the year from the date.

Grouping is achieved using the parenthesis operators.

Source string is: “Kartik’s birthday is on 15/03/1995”. To extract the date from this string, use the pattern - “^\d{1,2}\/\d{1,2}\/\d{4}$”.

To extract the year, put parentheses around the year part of the pattern. The pattern is: “^\d{1,2}/\d{1,2}/(\d{4})$”.



Grouping is a very useful technique to extract substrings from an entire match.

Regular Expressions: Use Cases [13/15]

These use cases will demonstrate practical applications of regex.

The following code demonstrates how regex can be used for a file search operation. Say you have a list of folders and filenames called 'items' and you want to extract (or read) only some specific files, say images.

```
# items contain all the files and folders of current directory
items = ['photos', 'documents', 'videos', 'image001.jpg','image002.jpg','image005.jpg', 'wallpaper.jpg',
'flower.jpg', 'earth.jpg', 'monkey.jpg', 'image002.png']
# create an empty list to store resultant files
images = []
# regex pattern to extract files that end with '.jpg'
pattern = ".*\.jpg$"
for item in items:
    if re.search(pattern, item):
        images.append(item)
# print result
print(images)
```

Running above code in Python will give the following result.

['image001.jpg', 'image002.jpg', 'image005.jpg', 'wallpaper.jpg', 'flower.jpg', 'earth.jpg', 'monkey.jpg']

The above code extracts only those documents which have ‘.jpg’ extension. Important thing here is the use of backslash. If not escape the dot operator with a backslash, results won’t come.

Another example; an extension to the previous example. In this case, we’re trying to extract documents that start with the prefix ‘image’ and end with the extension ‘.jpg’. Here’s the code:

```
# items contains all the files and folders of current directory
items = ['photos', 'documents', 'videos', 'image001.jpg','image002.jpg','image005.jpg', 'wallpaper.jpg',
'flower.jpg', 'earth.jpg', 'monkey.jpg', 'image002.png']
# create an empty list to store resultant files
images = []
# regex pattern to extract files that start with 'image' and end with '.jpg'
pattern = "image.*\.jpg$"
for item in items:
    if re.search(pattern, item):
        images.append(item)
# print result
print(images)
```

The output of the above code is as follows: ['image001.jpg', 'image002.jpg', 'image005.jpg']

To search for specific file names using regular expressions. Similarly, they can be used to extract features from text such as the ones listed below:

1. Extracting dates
2. Extracting emails
3. Extracting phone numbers, and other patterns.

Along with the applications in NLP, regexes are extensively used by software engineers in various applications such as checking if a new password meets the minimum criteria or not, checking if a new username meets the minimum criteria or not, and so on.

There are websites such as this one which helps to compile regex because sometimes it can get very hard to write regular expressions. There are various other online tools as well which provide intuitive interfaces to write and test regex instantly.

Regular Expressions - Practice Exercises

Regular expressions happen to be one of those concepts which you'll be able to retain in memory only with frequent practice. There are numerous online tools that teach and let you practice regex, online. Here is one such tool that you can use to quickly revise all the commonly used regex patterns and concepts.

Summary [14/15]

Different areas where text analytics is applied such as healthcare, e-commerce, retail, financial and various other industries. Stack that is generally followed to extract insights from the text and to build various applications of natural language processing. There are **three stages** in text analytics:

1. Lexical processing
2. Syntactic processing
3. Semantic processing

Then about **text encoding** and its various types such as **ASCII** and **Unicode**. How to change between different types of Unicode encodings in Python.

Then about **regular expressions**. How to manipulate and extract the information from a given text corpus using regular expressions. In regular expressions, **quantifiers**, their different types and how they are used to mention the number of times a character(s) is present. The **anchor characters** (^ and $) and the **wildcard** (.). The **character sets** and **meta-sequences** which are shorthand for common characters sets. The types of searches - **greedy and non-greedy** and how they differ. Use of **grouping characters** in a regular expression. Finally, different types of functions that are present in Python to facilitate the use of regular expressions in practical settings.

Refer to this link for a refresher in regular expressions in Python. There are some of the concepts that we've left untouched in regular expressions.

Graded Questions [15/15]