

Introduction to NLP

Welcome to the first module of natural language processing. Natural language processing, also referred to as 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 will take you through the following lectures:

1. Industry applications of text analytics
2. Understanding textual data
3. Regular expression

NLP: Areas of Application

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.

TEXT ANALYTICS: AREAS OF APPLICATION

1. Social media analytics
2. Banking and loan processing
3. Insurance claim processing
4. Customer relationship processing
5. Security and counter-terrorism
6. Computational social science
7. E-commerce
8. Psychology and cognitive science

So those were the different areas where text analytics is used extensively

Understanding Text

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

However, there is one question that still remains. The data you'll get while performing analytics on text, very often, will be just 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

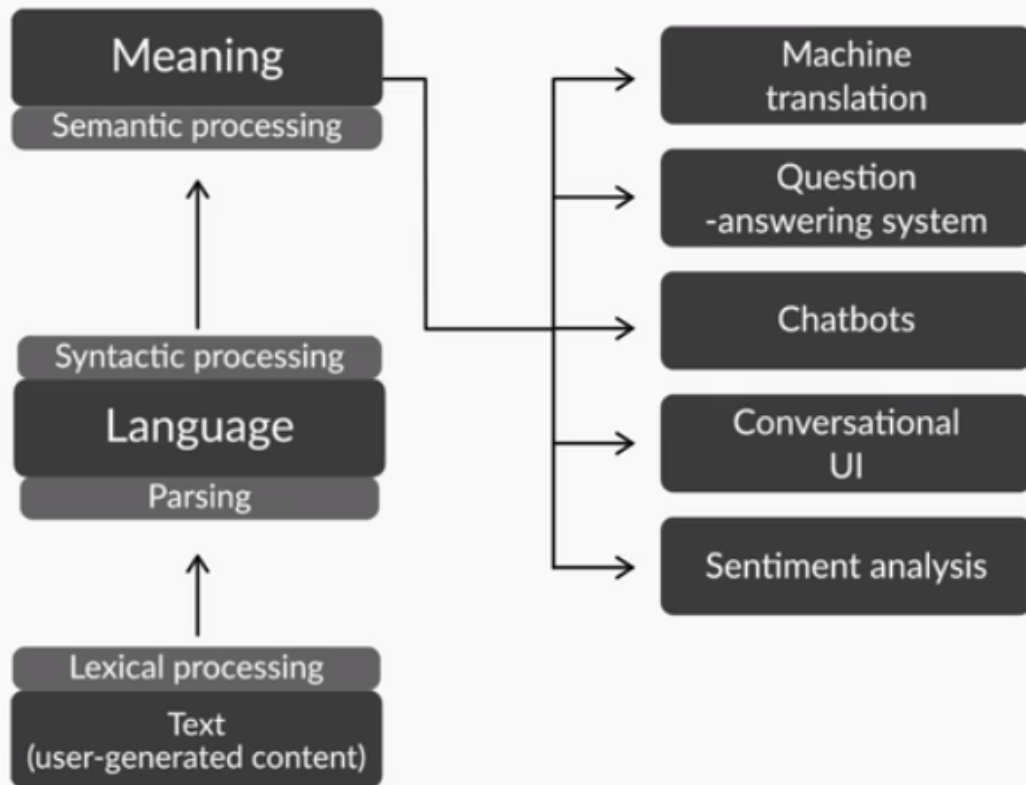
Now, think about it, if the data you get is of this form, and your task is to create an algorithm that translates this paragraph to a different language, say, Hindi, then how exactly will you do it?

To do so, your 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".

In this course on 'Text Analytics', you'll learn about all the different "steps" generally undertaken on the journey from data to meaning. This journey can be divided roughly into three parts, which correspond to the three modules that you'll study one-by-one in this course.

In Text Analytics we move up from lexical data that is human generated to

UNDERSTANDING TEXT



Lexical Processing :

User generated content is in the form of raw text, which we can break it up into words, letters, sentences, paragraphs, In Lexical Processing we can take the raw text and tokenize into different elements which make the linguistic constructs.

Once we finish the Lexical Processing we try to understand the language behind it, we do shallow parsing which is to identify what part of speech something is for eg : can (water can / able to do it)

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

Syntactic processing : Understanding the language itself, SP means if I take a sentence then can I understand how different parts of sentences are related with each other.

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?

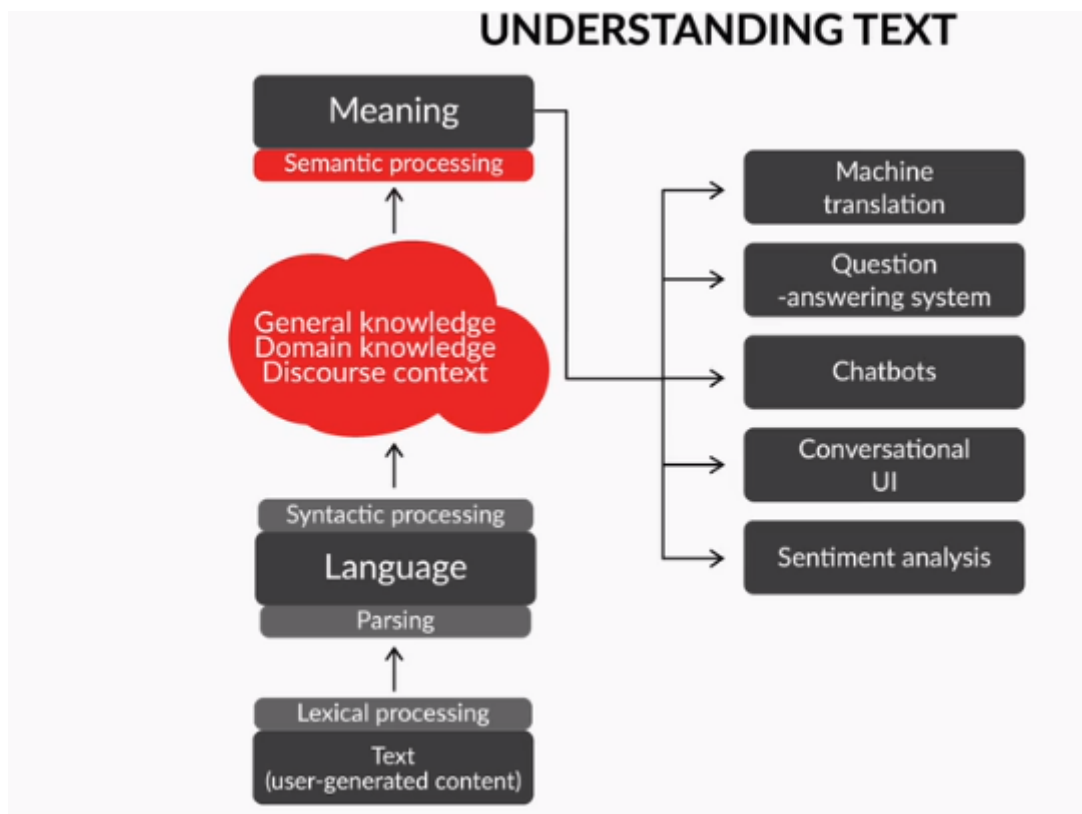
● **Book a good dentist**

✓ Correct!

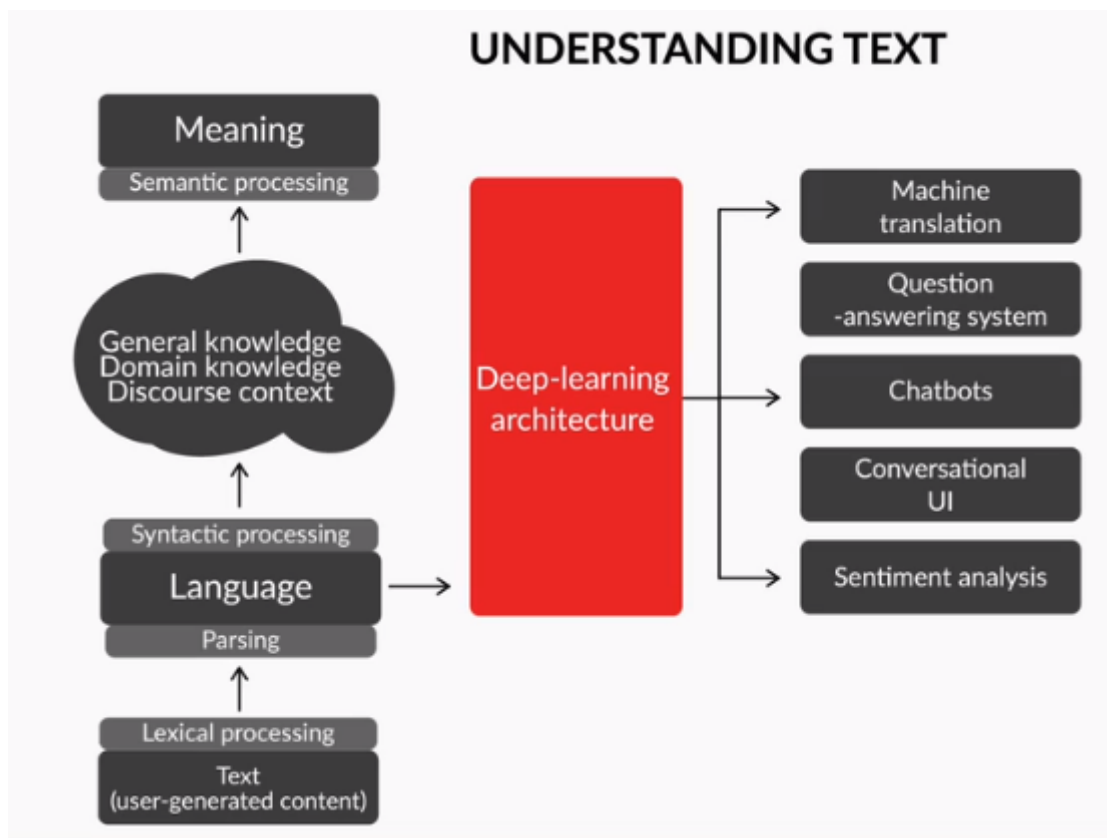
Q Feedback :

'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.

Semantic Processing(Meaning) : After understanding the text and language the next important thing is understanding the meaning, in SP we augment the language with some kind of general knowledge/Domain Knowledge/Discourse context. Understanding synonyms, antonyms etc. Once we have some notion of meaning we can start thinking of some kind of application



Deep learning architectures short-circuit the conventional NLP path. Semantic processing stage is entirely eliminated from the three stage process. The need of semantic processing is eradicated because deep learning models don't require need to feed hand crafted features extracted by semantic processing



Let's go back to the wikipedia example. Recall what the data (textual data) looked like - it was simply a collection of characters, that machines can't make any sense of. Starting with this data, you will move according to the following steps -

Lexical Processing:

First, you will just convert the raw text into words and, depending on your application's needs, into sentences or paragraphs as well.

For example, if an email contains words such as lottery, prize and luck, then the email is represented by these words, and it is likely to be a spam email.

Hence, in general, the group of words contained in a sentence gives us a pretty good idea of what that sentence 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.

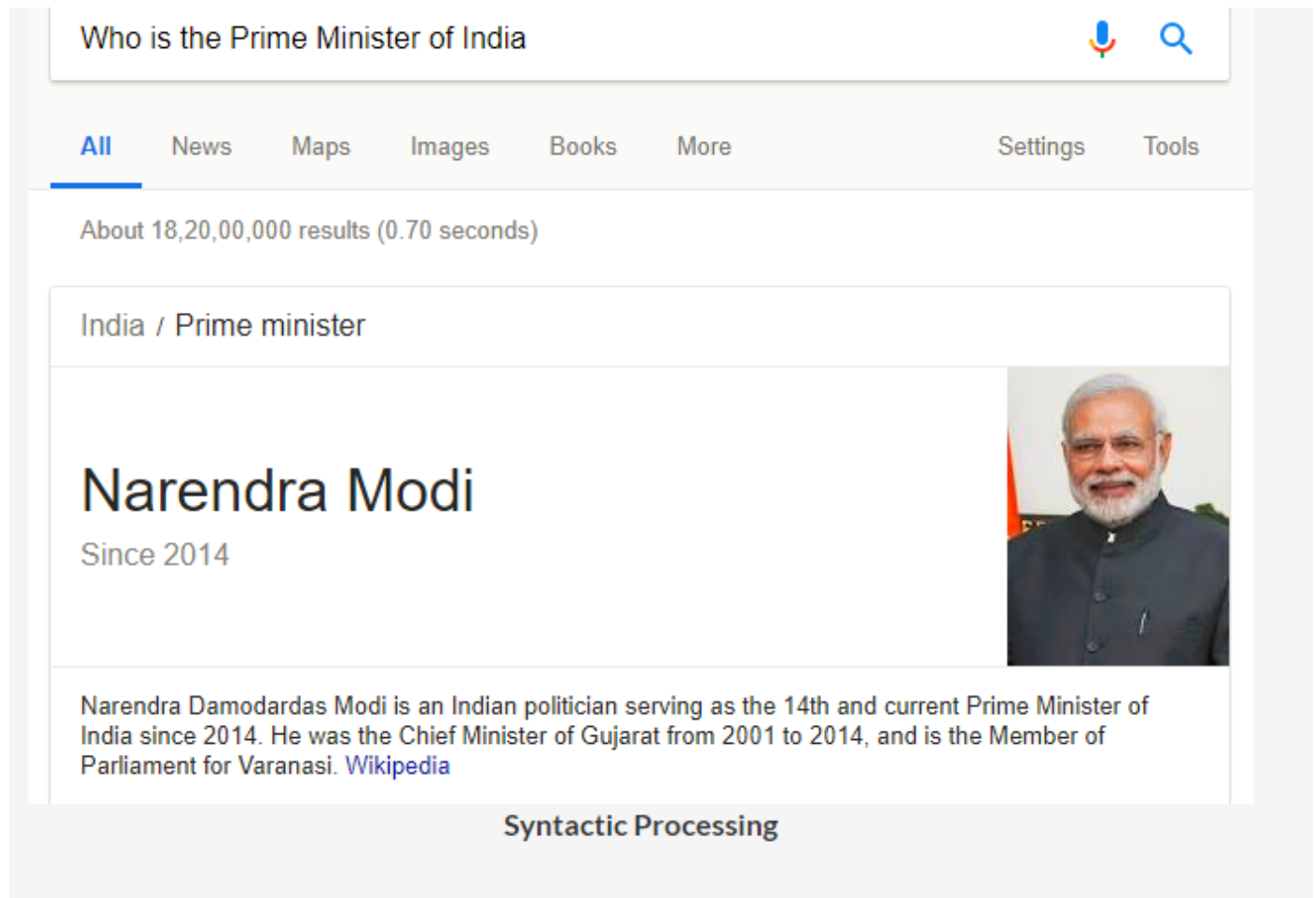
For a simple application like spam detection, lexical processing works just fine, but it is usually not enough in more complex applications, like, say, machine translation. For example, the sentences "My cat ate its third meal" and "My third cat ate its meal", have very different meanings. However, lexical processing will treat the two sentences as equal, as the "group of words" in both sentences is the same. Hence, we clearly need a more advanced system of analysis.

Syntactic Processing:

So, the next step after lexical analysis is where we try to extract more meaning from the sentence, by using its syntax this time. Instead of only looking at the words, we look at the syntactic structures, i.e., the grammar of the language to understand what the meaning is.

One example is differentiating between the subject and the object of the sentence, i.e., identifying who is performing the action and who is the person affected by it. For example, “Ram thanked Shyam” and “Shyam thanked Ram” are sentences with different meanings from each other because in the first instance, the action of ‘thanking’ is done by Ram and affects Shyam, whereas, in the other one, it is done by Shyam and affects Ram. Hence, a syntactic analysis that is based on a sentence’s subjects and objects, will be able to make this distinction.

There are various other ways in which these syntactic analyses can help us enhance our understanding. For example, a question answering system that is asked the question “Who is the Prime Minister of India?”, will perform much better, if it can understand that the words “Prime Minister” are related to “India”. It can then look up in its database, and provide the answer.



The screenshot shows a search engine interface. At the top, the search bar contains the text "Who is the Prime Minister of India". Below the search bar, there are tabs for "All", "News", "Maps", "Images", "Books", and "More". The "All" tab is selected. Below the tabs, it says "About 18,20,00,000 results (0.70 seconds)". The main result is for "India / Prime minister". It features a large heading "Narendra Modi" and a subheading "Since 2014". To the right of the text is a portrait of Narendra Modi. Below the portrait, there is a brief description: "Narendra Damodardas Modi is an Indian politician serving as the 14th and current Prime Minister of India since 2014. He was the Chief Minister of Gujarat from 2001 to 2014, and is the Member of Parliament for Varanasi. [Wikipedia](#)". At the bottom of the screenshot, the text "Syntactic Processing" is displayed.

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 actually 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 even 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 are you going to store manually? At some point, your machine should be able to identify synonyms, antonyms, etc. on its own.

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.

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 of 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 syntactic 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.

This gives you a basic idea of the process of analysing text and understanding the meaning behind it. Now, in the next segment, you’ll learn how text is stored on machines.

Text Encoding Data is being collected in many languages. However, in this course, you will be doing text analysis for the English language. The text analytics techniques that work for English might not work for other languages.

Text Encoding

Data is being collected in many languages. However, in this course, you will be doing text analysis for the English language. The text analytics techniques that work for English might not work for other languages. Lets see how characters of different languages are stored on computers.

LOOKING AT DIFFERENT LANGUAGES

Text	English translation
不再匆忙了。一切都在规划中。	There is no hurry anymore. Everything is planned.
Aufenthaltsgeheimungsantrag	Application for residence permit.
சொல்லீகோய்	One (person) in ten million.

Now, it is not necessary that when you work with text, you’ll 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. For you to work with non-English text, you 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.

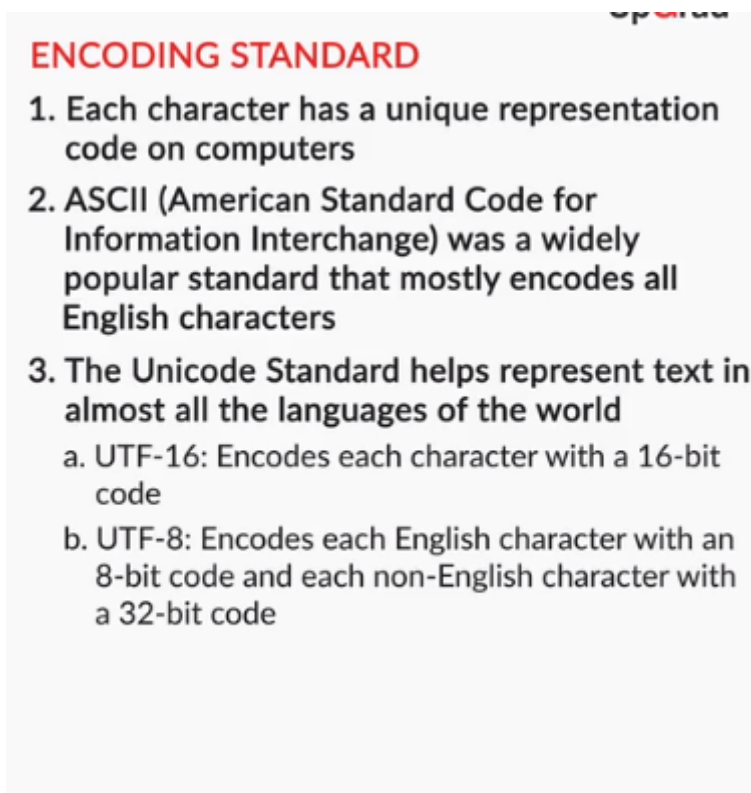
The first encoding standard that came into existence was the ASCII (American Standard Code for Information Interchange) standard, in 1960. ASCII standard assigned a unique code to each character of the keyboard which was known as ASCII code. For example, the ASCII code of the alphabet 'A' is 65 and that 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 has come into existence in recent years - the Unicode standard. It supports all the languages in the world - 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.

Lets see 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, there's a guide on official Python website. You can check it out here. (<https://docs.python.org/3/howto/unicode.html> (<https://docs.python.org/3/howto/unicode.html>))



ENCODING STANDARD

1. Each character has a unique representation code on computers
2. ASCII (American Standard Code for Information Interchange) was a widely popular standard that mostly encodes all English characters
3. The Unicode Standard helps represent text in almost all the languages of the world
 - a. UTF-16: Encodes each character with a 16-bit code
 - b. UTF-8: Encodes each English character with an 8-bit code and each non-English character with a 32-bit code

To summarise, there are two most popular encoding standards:

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

Let's look at the relation between ASCII, UTF-8 and UTF-16 through an example. The table below 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

As you can see, 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 actually 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.

In [1]:

```
# 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')
```

```
Default string: ₹50
Type of string <class 'str'>
```

```
Encoded to UTF-8: b'\xe2\x82\xb950'
Type of string <class 'bytes'>
```

```
Decoded from UTF-8: ₹50
Type of string <class 'str'>
```

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

This section onwards, you'll learn about regular expressions. 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. For someone 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.

Let's say you want to extract all the hashtags from a tweet. A hashtag has a fixed pattern to it, i.e. a pound ('#') character followed by a string. Some example hashtags are - #mumbai, #bangalore, #upgrad. You could easily achieve this task by providing this pattern and the tweet that you want to extract the pattern from (in this case, the pattern is - any string starting with #). Another example is to extract all the phone numbers from a large piece of textual data.

In short, if there's a pattern in any string, you can easily extract, substitute and do all kinds of other string manipulation operations using regular expressions.

Learning regular expressions basically means learning how to identify and define these patterns.

Regular expressions are a language in itself since they have their own compilers. Almost all popular programming languages support working with regexes and so does Python.

Let's take a look at how to work with regular expressions in Python. Download the Jupyter notebook provided below to follow along:

Regular Expressions

Regular expression is a set of characters, called as the pattern, which helps in finding substrings in a given string. The pattern is used to detect the substrings

For example, suppose you have a dataset of customer reviews about your restaurant. Say, you want to extract the emojis from the reviews because they are a good predictor of the sentiment of the review.

Take another example, the artificial assistants such as Siri, Google Now use information retrieval to give you better results. When you ask them for any query or ask them to search for something interesting on the screen, they look for common patterns such as emails, phone numbers, place names, date and time and so on. This is because then the assistant can automatically make a booking or ask you to call the restaurant to make a booking.

Regular expressions are a very powerful tool in text processing. It will help you to clean and handle your text in a much better way.

REGULAR EXPRESSIONS

1. Search patterns that are used to extract information from text
2. These can be used to extract details such as email addresses, phone numbers, user IDs, etc.

Example : Sentiment of customer reviews

USE CASES OF REGULAR EXPRESSIONS

Restaurant Reviews

I loved the pasta 🍷 and the fruit salad which came decorates with honey 🍷

Worst food ever 😞 . Money wasted!!!! 😡

The ambience of the place was nice but the food was average 😐

Let's import the regular expression library in python.

Let's define a function to match regular expression patterns

In [19]:

```
import re
```

In [21]:

```
#Let's do a quick search using a pattern.
re.search('Ravi', 'Ravi is an exceptional student!')
```

Out[21]:

```
<_sre.SRE_Match object; span=(0, 4), match='Ravi'>
```

In [22]:

```
# print output of re.search()
match = re.search('Ravi', 'Ravi is an exceptional student!')
print(match.group())
```

Ravi

In [23]:

```
# Let's define a function to match regular expression patterns

def find_pattern(text, patterns):
    if re.search(patterns, text):
        return re.search(patterns, text)
    else:
        return 'Not Found!'
```

Quantifiers

So that's how you import regular expressions library in python and use it. You saw how to use the `re.search()` function - it returns a regex object if the pattern is found in the string. Also, you saw two of its methods - `match.start()` and `match.end()` which return the index of the starting and ending position of the match found.

Apart from `re.search()`, there are other functions in the `re` library that are useful for other tasks. You'll look at the other functions later in this session.

Now, the first thing that you'll learn about regular expressions is the use of quantifiers. Quantifiers allow you to mention and have control over how many times you want the character(s) in your pattern to occur.

Let's take an example. Suppose you have some data which have the word 'awesome' in it. The list might look like - ['awesome', 'awesomeeee', 'awesomee']. You decide to extract only those elements which have more than one 'e' at the end of the word 'awesome'. This is where quantifiers come into picture. They let you handle these tasks.

You'll learn four types of quantifiers:

- 1.The '?' operator
- 2.The '*' operator
- 3.The '+' operator
- 4.The '{m, n}' operator

The first quantifier is '?'. Let's understand what the '?' quantifier does.

In [30]:

```
# '*': Zero or more
print(find_pattern("ac", "ab*"))
print(find_pattern("abc", "ab*"))
print(find_pattern("abbc", "ab*"))
```

```
<_sre.SRE_Match object; span=(0, 1), match='a'>
<_sre.SRE_Match object; span=(0, 2), match='ab'>
<_sre.SRE_Match object; span=(0, 3), match='abb'>
```

In [31]:

```
# '?': Zero or one (tells whether a pattern is absent or present)
print(find_pattern("ac", "ab?"))
print(find_pattern("abc", "ab?"))
print(find_pattern("abbc", "ab?"))
```

```
<_sre.SRE_Match object; span=(0, 1), match='a'>
<_sre.SRE_Match object; span=(0, 2), match='ab'>
<_sre.SRE_Match object; span=(0, 2), match='ab'>
```

In [32]:

```
# '+': One or more
print(find_pattern("ac", "ab+"))
print(find_pattern("abc", "ab+"))
print(find_pattern("abbc", "ab+"))
```

Not Found!

```
<_sre.SRE_Match object; span=(0, 2), match='ab'>
<_sre.SRE_Match object; span=(0, 3), match='abb'>
```

In [119]:

```
# {n}: Matches if a character is present exactly n number of times
print(find_pattern("abbc", "ab{2}"))
# {m,n}: Matches if a character is present from m to n number of times
print(find_pattern("aabbbbbbc", "ab{3,5}")) # return true if 'b' is present 3-5 times
print(find_pattern("aabbbbbbc", "ab{7,10}")) # return true if 'b' is present 7-10 times
print(find_pattern("aabbbbbbc", "ab{,10}")) # return true if 'b' is present atmost 10 times
print(find_pattern("aabbbbbbc", "ab{10,}")) # return true if 'b' is present from at least 10 times
```

```
<_sre.SRE_Match object; span=(0, 3), match='abb'>
<_sre.SRE_Match object; span=(1, 7), match='abbbbbb'>
Not Found!
<_sre.SRE_Match object; span=(0, 1), match='a'>
Not Found!
```

Description

Consider the following sentence: "The roots of education are bitter, but the fruit is sweet."

Write a regular expression pattern to check whether the word 'education' is present in the given string or not. Use the re.search() function.

In [83]:

```
#import the regular expression module
import re

# input string on which to test regex pattern
string = 'The roots of education are bitter, but the fruit is sweet.'

# regex pattern to check if 'education' is present in a input string or not.
pattern = 'education'# write regex to extract 'education'

# check whether pattern is present in string or not
result = re.search(pattern, string)

# evaluate result - don't change the following piece of code, it is used to evaluate your result
if result != None:
    print(True)
else:
    print(False)
```

True

In [84]:

```
# Consider the same problem as the previous question. Extract the word 'education'
# from the sentence 'The roots of education are bitter, but the fruit is sweet'.
# But this time, extract the starting position of the result using result.start().

# import the regular expression module
import re

# input string on which to test regex pattern
string = 'The roots of education are bitter, but the fruit is sweet.'

# regex pattern to check if 'education' is present in a input string or not.
pattern = 'education'# write regex to extract 'education'

# check whether pattern is present in string or not
result = re.search(pattern, string)

# evaluate result - don't change the following piece of code, it is used to evaluate your r
if result != None:
    print(True)
else:
    print(False)
```

True

In [86]:

```
# Consider the same problem as described in the first question.
# Extract the word 'education' from the sentence 'The roots of education are bitter,
# but the fruit is sweet'. But this time extract the ending position of the match using res
# import the regular expression module
import re

# input string
string = 'The roots of education are bitter, but the fruit is sweet.'

# write a regular expression pattern to check if 'education' is present in a given string o
pattern = 'education'

# store the match of regex
result = re.search(pattern, string)

# store the end of the match using result.end()
end_position = result.end()# write code here

# evaluate result - don't change the following piece of code, it is used to evaluate your r
print(end_position)
```

22

In [106]:

```
# Write a regular expression that matches the word 'tree' or 'trees' in a given piece of text

import ast, sys
string1 = '''Sample positive cases:
'The tree stands tall.'
'There are a lot of trees in the forest.' '''

string2 = '''Negative negative cases:
'The boy is heading for the school.'
'It's really hot outside!''' #sys.stdin.read() # input string

# import the regular expression module
import re

# regex pattern
pattern = 'tree?' # write regex here

# check whether pattern is present in string or not
result = re.search(pattern,string1) # pass the arguments to the re.search function

# evaluate result - don't change the following piece of code, it is used to evaluate your result
if result != None:
    print(True)
else:
    print(False)

# check whether pattern is present in string or not
result = re.search(pattern,string2) # pass the arguments to the re.search function

# evaluate result - don't change the following piece of code, it is used to evaluate your result
if result != None:
    print(True)
else:
    print(False)
```

True
False

In [108]:

```
'''Write a regular expression that matches the following words:

xyz

xy

xz

x

Make sure that the regular expression doesn't match the following words:

Xyyz

Xyzz

Xyy

Xzz

Yz
'''

import re
import ast, sys
string = sys.stdin.read()

# regex pattern
pattern = 'x?'# write regex pattern here

# check whether pattern is present in string or not
result = re.search(pattern, string)

# evaluate result - don't change the following piece of code, it is used to evaluate your r
if result != None:
    print(True)
else:
    print(False)
```

True

Match a binary number that starts with 101 and ends with zero or more number of zeroes.

Sample positive cases (pattern should match all of these):

1010

10100

101000

101

Sample negative cases (shouldn't match any of these):

10

100

1

In [109]:

```
import re
import ast, sys
string = sys.stdin.read()

# regex pattern
pattern = '1010*' # write your regex pattern here

# check whether pattern is present in string or not
result = re.search(pattern, string)

# evaluate result - don't change the following piece of code, it is used to evaluate your r
if result != None:
    print(True)
else:
    print(False)
```

False

Write a pattern that starts with 1 and ends with zero but has arbitrary number of 1s (zero or more) in between

Sample positive cases (should match all of these):

110

11111110

10

Sample negative cases (shouldn't match any of these):

11

00

1

0

In [110]:

```
import re
import ast, sys
string = sys.stdin.read()

# regex pattern
pattern = '11*0'# write regex pattern here

# check whether pattern is present in string or not
result = re.search(pattern, string)

# evaluate result - don't change the following piece of code, it is used to evaluate your r
if result != None:
    print(True)
else:
    print(False)
```

False

Regular Expressions: Quantifiers - II

In the last section, you learnt how to match strings with the '?' and the '*' quantifiers:

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, you'll learn about a new quantifier - the '+'

In [112]:

```
print(find_pattern("abbc", "b+"))
print(match_pattern("abbc", "b+"))
print(match_pattern("bbc", "b+"))
```

```
<_sre.SRE_Match object; span=(1, 3), match='bb'>
Not found!
<_sre.SRE_Match object; span=(0, 2), match='bb'>
```

The '+' quantifier matches the preceding character one or more times. That means the preceding character has to be present at least once for the pattern to match the string.

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

Write a pattern that matches numbers that are powers of 10.

Sample positive matches (should match all of the following):

10

100

1000

Sample negative matches (shouldn't match either of these):

0

1

15

In [118]:

```
import re
import ast, sys
string = '10' #change to 100,110

# regex pattern
pattern = '10+' # write your pattern here

# check whether pattern is present in string or not
result = re.search(pattern,string) # pass the parameters to check if pattern is present in

# evaluate result - don't change the following piece of code, it is used to evaluate your r
if result != None:
    print(True)
else:
    print(False)
```

True

To summarise, you have learnt the 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 do you specify a regex when you want to look for a character that appears, say, exactly 5 times, or between 3-5 times? You cannot do that using the quantifiers above.

Hence, the next quantifier that you'll learn will help you specify occurrences of the preceding character a fixed number of times.

In [154]:

```
# {n}: Matches if a character is present exactly n number of times
print(find_pattern("abbc", "ab{2}"))
# {m,n}: Matches if a character is present from m to n number of times
print(find_pattern("aabbbbbbc", "ab{3,5}")) # return true if 'b' is present 3-5 times
print(find_pattern("aabbbbbbc", "ab{7,10}")) # return true if 'b' is present 7-10 times
print(find_pattern("aabbbbbbc", "ab{,10}")) # return true if 'b' is present atmost 10 times
print(find_pattern("aabbbbbbc", "ab{10,}")) # return true if 'b' is present from at least 10 times
```

```
<_sre.SRE_Match object; span=(0, 3), match='abb'>
<_sre.SRE_Match object; span=(1, 7), match='abbbbbb'>
Not Found!
<_sre.SRE_Match object; span=(0, 1), match='a'>
Not Found!
```

There are four variants of the quantifier that you just saw:

{m, n}: Matches the preceding character 'm' times to 'n' times.

{m, }: Matches the preceding character 'm' times to infinite times, i.e. there is no upper limit to the occurrence of the preceding character.

{, n}: Matches the preceding character from zero to 'n' times, i.e. the upper limit is fixed regarding the occurrence of the preceding character.

{n}: Matches if the preceding character occurs exactly 'n' number of times.

Note that while specifying the {m,n} notation, avoid using a space after the comma, i.e. use {m,n} rather than {m, n}.

An interesting thing to note is that 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, }

Write a regular expression to match the word 'hurray'. But match only those variants of the word where there are a minimum of two 'r's and a maximum of five 'r's.

In [157]:

```
import re
import ast, sys
string = 'hurrerrrrray'

# regex pattern
pattern = 'hur{2,5}ay'# write your regex pattern here

# check whether pattern is present in string or not
result = re.search(pattern,string) # pass the pattern and string arguments to the function

# evaluate result - don't change the following piece of code, it is used to evaluate your r
if result != None:
    print(True)
else:
    print(False)
```

False

Write a regular expression which matches variants of the word 'awesome' where there are more than two 'e's at the end of the word.

The following strings should match:

Awesomeeee

awesomeeeee

The following strings shouldn't match:

Awesom

Awesome

Awesomee

In [170]:

```
# Write a regular expression which matches variants of the word 'awesome' where there are m
# two 'e's at the end of the word.
import re
import ast, sys
string = 'awosomeee'

# regex pattern
pattern = '[A-a]wesome{3,}'# write your regex pattern here

# check whether pattern is present in string or not
result = re.search(pattern, string)

# evaluate result - don't change the following piece of code, it is used to evaluate your r
if result != None:
    print(True)
else:
    print(False)
```

True

Write a regular expression that matches the a string where 'a' is followed by 'b' a maximum of three times.

Sample positive matches (should match all of these): a

ab

abb

abbb

Sample negative matches (shouldn't match either of these):

abbbb

abbbbbbbb

In [155]:

```
# code is wrong-- TBD
# Write a regular expression that matches the a string where
# 'a' is followed by 'b' a maximum of three times

import re
import ast, sys

string1 = 'abbbbb'

# regex pattern
pattern = 'ab{,3}'# write your regex here

# check whether pattern is present in string or not
result = re.search(pattern, string1)

print(result)
# evaluate result - don't change the following piece of code, it is used to evaluate your r
if result != None:
    print(True)
else:
    print(False)
```

```
<_sre.SRE_Match object; span=(0, 4), match='abbb'>
True
```

Write a regular expression to match a term that has three or more '0's followed by one or more '1's

Sample positive matches (should match all of these):

0001

000001111

000011

Negative positive matches (shouldn't match either of these):

00111

000

00

111

In [172]:

```
import re
import ast, sys
string = '000011'

# regex pattern
pattern = '0{3,}1{1,}'# write your regex here

# check whether pattern is present in string or not
result = re.search(pattern, string)

# evaluate result - don't change the following piece of code, it is used to evaluate your r
if result != None:
    print(True)
else:
    print(False)
```

True

Comprehension: Regular Expressions

In this section, you'll learn some new concepts of regular expressions.

The first is the use of whitespace. Till now, in the regular expression pattern, you didn't use a whitespace character. A whitespace comprises of a single space, multiple spaces, tab space or a newline character (also known as a vertical space). You can learn about multiple spaces in a computer here. Turns out, you can use these spaces in your 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, you could use spaces with other characters inside the pattern. The pattern, 'James Allen' will allow you to look for the name 'James Allen' in any given string.

When you learn about character classes later in this session, you'll see the different types of spaces that one can use. Whitespaces are used extensively when used inside character sets about which you'll study later in this session.

Moving onto the next notation - the parentheses. Till now, you have used quantifiers preceded by a single character which meant that the character preceded by the quantifier can repeat a specified number of times. If you put the 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 concept is called grouping in regular expression jargon. For example, the pattern '(abc){1, 3}' will match the following strings:

abc

abcbabc

abcbabcabc

Similarly, the pattern (010)+ will match:

010

010010

010010010, and so on.

You'll study about grouping later in this session.

Write a regular expression which matches a string where '23' occurs one or more times followed by occurrence of '78' one or more times

Sample positive matches (should match all of these):

2378

23237878

232323237878

Sample negative matches (shouldn't match either of these):

23

78

23378

223378

22337788

In [175]:

```
import re
import ast, sys
string = '2378'

# regex pattern
pattern = '(23){1,}(78){1,}' # write your regex here

# check whether pattern is present in string or not
result = re.search(pattern, string)

# evaluate result - don't change the following piece of code, it is used to evaluate your r
if result != None:
    print(True)
else:
    print(False)
```

True

Let's move to the next notation - the pipe operator. It's notated by '|'. The pipe operator is used as an OR operator. You need to use it inside the parentheses. For example, the pattern '(d|g)one' will match both the strings - 'done' and 'gone'. The pipe operator tells that the place inside the parentheses can be either 'd' or 'g'.

Similarly, the pattern '(ICICI|HDFC) Bank' will match the strings 'ICICI Bank' and 'HDFC Bank'. You can also use quantifiers after the parentheses as usual even when there is a pipe operator inside. Not only that, 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'.

Write a regular expression that matches the following strings: Basketball, Baseball, Volleyball, Softball, Football

In [176]:

```
import re
import ast, sys
string = 'Basketball'

# regex pattern
pattern = '(Basket|Base|Volley|Soft|Foot)ball'# write your regex here

# check whether pattern is present in string or not
result = re.search(pattern, string)

# evaluate result - don't change the following piece of code, it is used to evaluate your result
if result != None:
    print(True)
else:
    print(False)
```

True

Lastly, you will often find yourself in situations where you will need to mention characters such as '?', '*', '+', '(', ')', '{', etc. in your regular expressions. These are called special characters since they have special meanings when they appear inside a regex pattern (as you have already seen).

Suppose you want to extract all the questions from a document, and you assume that all questions end with a question mark - '?'. So you would need to use the '?' in the regular expression. Now, you already know that '?' has a special meaning in regular expressions. So, how do you tell regular expression engine that you want to match the question mark literally in the sentence, rather than as a special character (which it is by default)?

In situations such as these, you'll need to use escape sequences. The escape sequence, denoted by a backslash '\', is used to escape the special meaning of the special characters. To match a question mark literally, you need to use '?' (this is called escaping the character).

Now, let's say we have this string - 'David, who scored 56(78), was bowled by Brett Lee after lunchtime'. Suppose, we want to extract '(78)' from the given string. To do that, we can't use the pattern '(78)'. If we use it, we'll get '78' instead of '(78)'. What we really want is the substring '(78)'. Therefore, we need to escape the special meaning of the parentheses in this case. The pattern that we're going to use is '(78)'. The escape character is preceded by the character that you want to escape.

Note: The " itself is a special character, and to match the '\' character literally, you need to escape it too. You can use the pattern '\\' to escape the backslash.

Let's take another example - if you want to match the addition symbol in a string, you can't use the pattern '+'. You need to escape the '+' operator and the pattern that you're going to use in this case is '+'.

Write a regular expression that returns True when passed a multiplication equation. For any other equation, it should return False. In other words, it should return True if there an asterisk - '*' - present in the equation.

Sample positive cases (should match all of these):

3a*4b

3*2

456=120

Sample negative cases (shouldn't match either of these):

5+3=8

3%2=1

In [177]:

```
import re
import ast, sys
string = sys.stdin.read()

# regex pattern
pattern = # write your regex here

# check whether pattern is present in string or not
result = re.search(pattern, string)

# evaluate result - don't change the following piece of code, it is used to evaluate your r
if result != None:
    print(True)
else:
    print(False)
```

File "<ipython-input-177-9f0805219831>", line 6

pattern = # write your regex here

^

SyntaxError: invalid syntax

Now, there are something called as regex flags. A flag has a special meaning. For example, if you want your regex to ignore the case of the text then you can pass the 're.I' flag. Similarly, you have can have a flag with the syntax re.M that enables you to search in multiple lines (in case the input text has multiple lines). You can pass all these flags in the re.search() function. The syntax to pass multiple flags is:

In [178]:

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

Out[178]:

```
<_sre.SRE_Match object; span=(0, 10), match='Basketball'>
```

Last, you need to know about the re.compile() function. This function stores the regular expression pattern in

the cache memory and is said to result in a little faster searches. You need to pass the regex pattern to `re.compile()` function. The following piece of code shows the difference between searching with the compile function and without the compile function.

In [179]:

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

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

So that was all on quantifiers. In the next section, you'll learn about anchors.

Regular Expressions: Anchors and Wildcard

Next, you will learn about anchors in regular expressions. Anchors are used to specify the start and end of the string. Watch the following video where Krishna explains what are anchors and how to use them.

In [184]:

```
# '^': 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$")) # return true if string ends with 'a'
print(find_pattern("Japan", "a$")) # return true if string ends with 'a'
```

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

You learnt about the two anchors characters: '^' and '\$'.

The '^' specifies the start of the string. The character followed by the '^' in the pattern should be the first character of the string in order for a string to match the pattern.

Similarly, the '\$' specifies the end of the string. The character that precedes the '\$' in the pattern should be the last character in the string in order for the string to match the pattern.

Both the anchors can be specified in a single regular expression itself. For example, the regular expression pattern '^010\$' will match any string that starts and end with zeroes with any number of 1s between them. It will match '010', '0110', '0111111110' and even '00' (0 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.

Write a pattern that matches all the dictionary words that start with 'A'

Positive matches (should match all of these): Avenger Acute Altruism

Negative match (shouldn't match any of these): Bribe 10 Zenith

In [185]:

```
import re
import ast, sys
string = sys.stdin.read()

# regex pattern
pattern = '^A'# write your regex here

# check whether pattern is present in string or not
result = re.search(pattern, string, re.I) # re.I ignores the case of the string and the po

# evaluate result - don't change the following piece of code, it is used to evaluate your r
if result != None:
    print(True)
else:
    print(False)
```

False

Description

Write a pattern which matches a word that ends with 'ing'. Words such as 'playing', 'growing', 'raining', etc. should match while words that don't have 'ing' at the end shouldn't match.

In [186]:

```
import re
import ast, sys
string = sys.stdin.read()

# regex pattern
pattern = '(ing)$'# write your regex here

# check whether pattern is present in string or not
result = re.search(pattern,string) # pass parameters to the re.search() function

# evaluate result - don't change the following piece of code, it is used to evaluate your r
if result != None:
    print(True)
else:
    print(False)
```

False

Description

Write a regular expression that matches any string that starts with one or more '1's, followed by three or more '0's, followed by any number of ones (zero or more), followed by '0's (from one to seven), and then ends with either two or three '1's.

In [187]:

```
import re
import ast, sys
string = sys.stdin.read()

# regex pattern
pattern = '1{1,}0{3,}1*0{1,7}1{2,3}$' # write your regex here

# check whether pattern is present in string or not
result = re.search(pattern, string)

# evaluate result - don't change the following piece of code, it is used to evaluate your r
if result != None:
    print(True)
else:
    print(False)
```

False

Now, there is one special character in regular expressions that acts as a placeholder and can match any character (literally!) in the given input string. It's the '.' (dot) character is also called the wildcard character.

In [188]:

```
# '.': Matches any character
print(find_pattern("a", "."))
print(find_pattern("#", "."))
```

```
<_sre.SRE_Match object; span=(0, 1), match='a'>
<_sre.SRE_Match object; span=(0, 1), match='#'>
```

Till now, you were mentioning the exact character followed by a quantifier in your regular expression patterns. For example, the pattern 'hur{2,}ay' matches 'hurray', 'hurrray', 'hurrrray' and so on. Here, we had specified that the letter 'r' should be present two or more times. But you don't always know the letter that you want to repeat. In such situations, you'll need to use the wildcard instead.

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.

For example, if you're asked 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 that satisfies this kind of condition would be '{4}0{3}1{2}.{2}'. You can also use '...00011..' where the dot acts as a placeholder which means anything can sit on the place of the dot. Both are correct regex patterns.

Description

Write a regular expression to match first names (consider only first names, i.e. there are no spaces in a name) that have length between three and fifteen characters.

Sample positive match: Amandeep

Krishna

Sample negative match:

Balasubrahmanyam

In [189]:

```
import re
import ast, sys
string = 'Amandeep'

# regex pattern
pattern = '.{3,15}' # write your regex here

# check whether pattern is present in string or not
result = re.search(pattern, string)

# evaluate result - don't change the following piece of code, it is used to evaluate your r
if result != None:
    print(True)
else:
    print(False)
```

True

So, you learnt how the '.' character can act as a placeholder for any character and how to use it. In the next section, you'll learn about character sets.

Regular Expressions: Characters Sets

Until now, you were either using the actual letters (such as ab, 23, 78, etc.) or the wildcard character in your regular expression 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.

For example, say you want to match phone numbers in a large document. You know that the numbers may contain hyphens, plus symbol etc. (e.g. +91-9930839123) , but it will not have any alphabet. You need to somehow specify that you are looking only for numerics and some other symbols, but avoid alphabets.

To handle such situations, you can use what are called character sets in regular expression jargon.

The following lecture explains the various types of characters sets available in regular expressions, and how can you use them in different situations.

In [192]:

```
# Now we will look at '[' and ']'.
# They're used for specifying a character class, which is a set of characters that you wish
# Characters can be listed individually as follows
print(find_pattern("a", "[abc]"))

# Or a range of characters can be indicated by giving two characters and separating them by
print(find_pattern("c", "[a-c]")) # same as above
```

```
<_sre.SRE_Match object; span=(0, 1), match='a'>
<_sre.SRE_Match object; span=(0, 1), match='c'>
```

In [193]:

```
# '^' is used inside character set to indicate complementary set
print(find_pattern("a", "[^abc]")) # return true if neither of these is present - a,b or c
```

Not Found!

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. In this way, a character set is similar to a wildcard because it can also be used with or without a quantifier. It's just that a character set gives you more power and flexibility!

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

You 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.

But what if you want to match every other character other than the one mentioned inside the character set. You can use the caret operator to do this. Here, Krishna explains the use of caret operator inside a character set.

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

Pattern	Matches
<code>[a-zA-Z]</code>	Matches any characters between a and z, including a and z ignoring cases of the characters
<code>[0-9]</code>	Matches any character which is a number between 0 and 9

The '^' has two use cases. You already know that it can be used outside a character set to specify the start of a string. Here, it is known as an anchor.

It's another use is inside a character set. When used inside a character set, 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 you work with regular expressions, you'll find yourself using characters often. You'll commonly use sets to match only digits, only alphabets, only alphanumeric characters, only whitespaces, etc.

Therefore, there is a shorthand way to write commonly used character sets in regular expressions. These are called meta-sequences

Meta sequences

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

Those were the commonly used meta-sequences. You can use meta-sequences in two ways:

- 1.You can either use them without the square brackets. For example, the pattern `'\w+'` will match any alphanumeric character.
- 2.Or you can use them inside the square brackets. For example, the pattern `'[\w]+'` is same as `'\w+'`. But when you use meta-sequences inside a square bracket, they're commonly used along with other meta-sequences. For example, the `'[\w\s]+'` matches both alphanumeric characters and whitespaces. The square brackets are used to group these two meta-sequences into one.

Description

Write a regular expression with the help of meta-sequences that matches usernames of the users of a database. The username starts with alphabets of length one to ten characters long and then followed by a number of length 4.

Sample positive matches:

sam2340

irfann2590

Sample negative matches:

8730

bobby9073834

sameer728

radhagopalaswamy7890

In [194]:

```
import re
import ast, sys
string = 'sam2340'

# regex pattern
pattern = '[a-zA-Z]{1,10}[0-9]{4}' # write your regex here

# check whether pattern is present in string or not
result = re.search(pattern, string, re.I)

# evaluate result - don't change the following piece of code, it is used to evaluate your r
if result != None:
    print(True)
else:
    print(False)
```

False

Next up, you'll learn about greedy and non-greedy search.

Greedy versus Non-greedy Search

When you use a regular expression to match a string, the regex greedily tries to look for the longest pattern possible in the string. For example, when you specify the pattern 'ab{2,5}' to match the string 'abbbbb', it will look for the maximum number of occurrences of 'b' (in this case 5).

This is called a 'greedy approach'. 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.

Let's look in detail when and how to use the non-greedy technique.

Greedy vs non-greedy regex

In [195]:

```
print(find_pattern("aabbbbbbb", "ab{3,5}")) # return if a is followed by b 3-5 times GREEDY
```

```
<_sre.SRE_Match object; span=(1, 7), match='abbbbb'>
```

In [196]:

```
print(find_pattern("aabbbbbbb", "ab{3,5}?")) # return if a is followed by b 3-5 times GREEDY
```

```
<_sre.SRE_Match object; span=(1, 5), match='abbb'>
```

In [197]:

```
# Example of HTML code
print(re.search("<.*>", "<HTML><TITLE>My Page</TITLE></HTML>"))
```

```
<_sre.SRE_Match object; span=(0, 35), match='<HTML><TITLE>My Page</TITLE></H
TML>'>
```

In [198]:

```
# Example of HTML code
print(re.search("<.*?>", "<HTML><TITLE>My Page</TITLE></HTML>"))
```

```
<_sre.SRE_Match object; span=(0, 6), match='<HTML>'>
```

Let's understand the non-greedy or the lazy approach with another example. Suppose, you have the string '3000'. Now, if you use the regular expression '30+', it means that you want to look for a string which starts with '3' and then has one or more '0's followed by it. This pattern will match the entire string, i.e. '3000'. This is the greedy way. But if you use 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.

For example, take the string 'One batsman among many batsmen.'. If you run the patterns 'bat' and 'bat?' on this text, the pattern 'bat' will match the substring 'bat' in 'batsman' and 'bat' in 'batsmen' while the pattern 'bat?' will match the substring 'ba' in batsman and 'ba' in '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'. On the other hand, 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, you can just put a question mark at the end of any of the following quantifiers that you've studied till now

*

+

?

{m, n}

{m,}

{, n}

{n}

The lazy quantifiers of the above greedy quantifiers are:

*?

+?

??

{m, n}?

{m,}?

{, n}?

{n}?

In [201]:

```
### Description

'''
You're given the following html code:

<html>
<head>
<title> My amazing webpage </title>
</head>
<body> Welcome to my webpage! </body>
</html>
'''

# Write a greedy regular expression that matches the entire code.

import re
import ast, sys
string = sys.stdin.read()

# regex pattern
pattern = '.*'# write your regex here

# check whether pattern is present in string or not
result = re.search(pattern, string, re.M) # re.M enables the pattern to be searched in multiline

# evaluate result - don't change the following piece of code, it is used to evaluate your result
if (result != None) and (len(result.group()) > 6):
    print(True)
else:
    print(False)
```

False

Write a non-greedy regular expression to match the contents of only the first tag (the tag in this case) including the angular brackets.

In [202]:

```
import re
import ast, sys
string = sys.stdin.read()

# regex pattern
pattern = '<.*>'# write your regex here

# check whether pattern is present in string or not
result = re.search(pattern, string, re.M) # re.M enables the pattern to be searched in multiline

# evaluate result - don't change the following piece of code, it is used to evaluate your result
if (result != None) and (len(result.group()) <= 6):
    print(True)
else:
    print(False)
```

False

Commonly Used RE Functions

Till now you've seen only one function of the 're' module, that is, the 're.search()' function. While it is a very common function used while working with regular expressions in python, it is not the only function that you'd use while working with regular expressions.

You're going to learn about four more functions in this section where we explain them one-by-one. Let's look at the other functions.

The five most important re functions that you would be required to use most of the times are

match() Determine if the RE matches at the beginning of the string

search() Scan through a string, looking for any location where this RE matches

findall() Find all the substrings where the RE matches, and return them as a list

finditer() Find all substrings where RE matches and return them as an iterator

sub() Find all substrings where the RE matches and substitute them with the given string

In [203]:

```
# - this function uses the re.match() and let's see how it differs from re.search()
def match_pattern(text, patterns):
    if re.match(patterns, text):
        return re.match(patterns, text)
    else:
        return ('Not found!')
```

In [206]:

```
print(find_pattern("abbc", "b+"))
print(match_pattern("abbc", "b+"))
print(match_pattern("bbc", "b+"))
```

```
<_sre.SRE_Match object; span=(1, 3), match='bb'>
Not found!
<_sre.SRE_Match object; span=(0, 2), match='bb'>
```

You learnt about the match function and the search function. 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.

Description

Write a string such that when you run the `re.match()` function on the string using the given regex pattern `'a{2,}'`, the function returns a non-empty match.

In []:

```
import re
import ast, sys
pattern = sys.stdin.read()

# write a string such that the re.match() function returns a non-empty match while using the
string = # write your here

# check whether pattern is present in string or not
result = re.match(pattern, string, re.I)

# evaluate result - don't change the following piece of code, it is used to evaluate your result
if (result != None):
    print(True)
else:
    print(False)
```

The next function that you're going to study is the `re.sub()` function. It is used to substitute a substring with another substring of your choice.

Regular expression patterns can help you find the substring in a given corpus of text that you want to substitute with another string. For example, you might want 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 [207]:

```
## Example usage of the sub() function. Replace Road with rd.
```

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

21 Ramakrishna Rd

In [208]:

```
print(re.sub('R\w+', 'Rd', street))
```

21 Rd Rd

The `re.sub()` function is used to substitute a part of your string using a regex pattern. It is often the case when you want to replace a substring of your 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, take a look at the following command:

In [209]:

```
pattern = "\d"  
replacement = "X"  
string = "My address is 13B, Baker Street"  
  
re.sub(pattern, replacement, string)
```

Out[209]:

'My address is XXB, Baker Street'

Description

You are given the following string:

“You can reach us at 07400029954 or 02261562153 ”

Substitute all the 11-digit phone numbers present in the above string with “#####”.

In [210]:

```
import re
import ast, sys
string = '07400029954'

# regex pattern
pattern = '[0-9]{11}'# write a regex that detects 11-digit number

# replacement string
replacement = '####'# write the replacement string

# check whether pattern is present in string or not
result = re.sub(pattern, replacement, string)

# evaluate result - don't change the following piece of code, it is used to evaluate your r
if (re.search(replacement, result)) != None:
    print(True)
else:
    print(False)
```

True

Description

Write a regular expression such that it replaces the first letter of any given string with '\$'.

For example, the string 'Building careers of tomorrow' should be replaced by "\$uilding careers of tomorrow".

In [215]:

```
import re
import ast, sys
string = 'Building careers of tomorrow'

# regex pattern
pattern = string[0]# write a regex that detects the first character of a string

# replacement string
replacement = '$' # write the replacement string

# check whether pattern is present in string or not
result = re.sub(pattern, replacement, string) # pass the parameters to the sub function

# evaluate result - don't change the following piece of code, it is used to evaluate your r
print(result[0] == '$')
```

True

The next set of functions let you search the entire input string and return all the matches, in case there are more than one present. In the following video, me explains the remaining functions.

In [217]:

```
## Example usage of finditer(). Find all occurrences of word Festival in given sentence

text = 'Diwali is a festival of lights,Festival 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 - 12 END - 20
START - 50 END - 58
```

In [73]:

```
# 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-st
date_regex = '(/(\d{4})/(\d{1,2})/(\d{1,2})/)'
print(re.findall(date_regex, url))
```

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

To summarise, the match and search command return only one match. But you often need to extract all the matches rather than only the first match, and that's when you use the other methods.

Suppose, in a huge corpus of text, you want to extract all the dates, in that case you can 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.

Description

Write a regular expression to extract all the words from a given sentence. Then use the re.finditer() function and store all the matched words that are of length more than or equal to 5 letters in a separate list called result.

Sample input: "Do not compare apples with oranges. Compare apples with apples"

In [220]:

```
import re
import ast, sys
string = "Do not compare apples with oranges. Compare apples with apples"

# regex pattern
pattern = '\w+' # write regex to extract all the words from a given piece of text

# store results in the list 'result'
result = []

# iterate over the matches
for match in re.finditer(pattern, string): # replace the ___ with the 'finditer' function
    if len(match.group()) >= 5:
        result.append(match)
    else:
        continue

# evaluate result - don't change the following piece of code, it is used to evaluate your r
print(len(result))
```

6

Description

Write a regular expression to extract all the words that have the suffix 'ing' using the re.findall() function. Store the matches in the variable 'results' and print its length.

Sample input: "Playing outdoor games when its raining outside is always fun!"

In [223]:

```
import re
import ast, sys
string = "Playing outdoor games when its raining outside is always fun!"

# regex pattern
pattern = '(\w+ing)' # write regex to extract words ending with 'ing'

# store results in the list 'result'
result = re.findall(pattern,string)# extract words having the required pattern, using the f

# evaluate result - don't change the following piece of code, it is used to evaluate your r
print(len(result))
```

2

Regular Expressions: Grouping

Sometimes you need to extract sub-patterns out of a larger pattern. This can be done by using grouping. Suppose you have textual data with dates in it and you want to extract only the year. from the dates. You can use a regular expression pattern with grouping to match dates and then you can extract the component elements such as the day, month or the year from the date.

Grouping is achieved using the parenthesis operators. Let's understand grouping using an example.

Let's say the source string is: "Kartik's birthday is on 15/03/1995". To extract the date from this string you can use the pattern - "\d{1,2}/\d{1,2}/\d{4}".

Now to extract the year, you can put parentheses around the year part of the pattern. The pattern is: "\d{1,2}/\d{1,2}/(\d{4})\$".

In [224]:

```
# 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-st
date_regex = '(/(\d{4})/(\d{1,2})/(\d{1,2})/)'
print(re.findall(date_regex, url))
```

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

In [75]:

```
## Exploring Groups
m1 = re.search(date_regex, url)
print(m1.group()) ## print the matched group , may be only the first occurrence is printed
```

```
/2017/10/28/
```

In [76]:

```
print(m1.group(1)) # - Print first group
```

```
2017
```

In [77]:

```
print(m1.group(2)) # - Print second group
```

```
10
```

In [78]:

```
print(m1.group(3)) # - Print third group
```

28

In [79]:

```
print(m1.group(0)) # - Print zero or the default group
```

/2017/10/28/

In [80]:

m1

Out[80]:

<_sre.SRE_Match object; span=(36, 48), match='/2017/10/28/'>

Description

You have a string which contains a data in the format DD-MM-YYYY. Write a regular expression to extract the date from the string.

Sample input: "Today's date is 18-05-2018."

In [226]:

```
import re
import ast, sys
string = "Today's date is 18-05-2018."

# regex pattern
#date_regex = '/(\d{4})/(\d{1,2})/(\d{1,2})/'
pattern = '(\d{2})-(\d{2})-(\d{4})' # write regex to extract date in DD-MM-YYYY format

# store result
result = re.search(pattern=pattern,string=string) # pass the parameters to the re.search()

# evaluate result - don't change the following piece of code, it is used to evaluate your r
if result != None:
    print(result.group(0)) # result.group(0) will output the entire match
else:
    print(False)
```

18-05-2018

Description

In the last exercise, you had extracted the date from a given string. In this coding exercise, write the same regular expression. But this time, use grouping to extract the month from the date. The expected date format is DD-MM-YYYY only.

Sample input: Today's date is 18-05-2018

In [227]:

```
import re
import ast, sys
string = "Today's date is 18-05-2018"

# regex pattern
pattern = '(\d{2})-(\d{2})-(\d{4})' # write regex to extract date and use grouping to extract month

# store result
result = re.search(pattern, string)

# extract month using group command
if result != None:
    month = result.group(2) # extract month using group command
else:
    month = "NA"

# evaluate result - don't change the following piece of code, it is used to evaluate your result
print(month)
```

05

Description

Write a regular expression to extract the domain name from an email address. The format of the email is simple - the part before the '@' symbol contains alphabets, numbers and underscores. The part after the '@' symbol contains only alphabets followed by a dot followed by 'com'

Sample input: user_name_123@gmail.com (mailto:user_name_123@gmail.com)

In [243]:

```
import re
import ast, sys
string = "user_name_123@gmail.com"

# regex pattern
pattern = '(\S+)@(\S+.com)' # write regex to extract email and use groups to extract domain

# store result
result = re.search(pattern, string)

print(result.group())
# extract domain using group command
if result != None:
    domain = result.group(2) # use group to extract the domain from result
else:
    domain = "NA"

# evaluate result - don't change the following piece of code, it is used to evaluate your r
print(domain)
```

```
user_name_123@gmail.com
gmail.com
```

Regular Expressions: Use Cases

Let's see some examples where regular expressions can be used as a handy tool. These use cases will demonstrate how can you use regular expressions for practical applications.

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.

In [244]:

```
# 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 end with '.jpg'
pattern = ".*\\.jpg$"

for item in items:
    if re.search(pattern, item):
        images.append(item)

# print result
print(images)
```

```
['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. The pattern '.*.jpg\$' is pretty self-explanatory. The important thing here is the use of backslash. If you don't escape the dot operator with a backslash, you won't get the results you want. Try to run the code without the escape sequence.

Take a look at another example. This is just 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'

In [245]:

```
# 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)
```

```
['image001.jpg', 'image002.jpg', 'image005.jpg']
```

You saw how 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:

Extracting dates

Extracting emails

Extracting phone numbers, and other patterns.

Along with the applications in NLP, regular expressions 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.

Now, let's end the session with a practical tip. You've already gotten to know that there are websites such as this one which helps you to compile your regular expression 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 your regex instantly.

<https://regexone.com/> (<https://regexone.com/>)

Q1.

Write a regular expression to match all the files that have either .exe, .xml or .jar extensions. A valid file name can contain any alphabet, digit and underscore followed by the extension.

In [246]:

```
import re
files = ['employees.xml', 'calculator.jar', 'nfsmw.exe', 'bkgrnd001.jpg', 'sales_report.ppt']

pattern = "^.+\\. (xml|jar|exe)$"

result = []

for file in files:
    match = re.search(pattern, file)
    if match != None:
        result.append(file)

# print result - result should only contain the items that match the pattern
print(result)
```

```
['employees.xml', 'calculator.jar', 'nfsmw.exe']
```

Q2

Write a regular expression to match all the addresses that have Koramangala embedded in them.

Strings that should match:

466, 5th block, Koramangala, Bangalore 4th BLOCK, KORAMANGALA - 560034 Strings that shouldn't match:

999, St. Marks Road, Bangalore

In [247]:

```
addresses = ['466, 5th block, Koramangala, Bangalore', '4th BLOCK, KORAMANGALA - 560034', '4th BLOCK, KORAMANGALA - 560034']  
pattern = "^[\w\d\s,-]*koramangala[\w\d\s,-]*$"   
result = []  
  
for address in addresses:  
    match = re.search(pattern, address, re.I)  
    if match !=None:  
        result.append(address)  
  
# print result - result should only contain the items that match the pattern  
print(result)
```

```
['466, 5th block, Koramangala, Bangalore', '4th BLOCK, KORAMANGALA - 560034']
```

Q3.

Write a regular expression that matches either integer numbers or floats upto 2 decimal places.

Strings that should match:

2

2.3

4.56

.61

Strings that shouldn't match:

4.567

75.8792

ab

In [249]:

```
numbers = ['2', '2.3', '4.56', '.61', '4.567', '75.8792', 'abc']

pattern = "^[0-9]*(\.[0-9]{,2})?$"

result = []

for number in numbers:
    match = re.search(pattern, number)
    if match != None:
        result.append(number)

# print result - result should only contain the items that match the pattern
print(result)
```

```
['2', '2.3', '4.56', '.61']
```

Q4.¶

Write a regular expression to match the model names of smartphones which follow the following pattern:

mobile company name followed by underscore followed by model name followed by underscore followed by model number

Strings that should match:

apple_iphone_6

samsung_note_4

google_pixel_2

Strings that shouldn't match:

apple_6

iphone_6

google_pixel_

In [250]:

```
phones = ['apple_iphone_6', 'samsung_note_4', 'google_pixel_2', 'apple_6', 'iphone_6', 'goc
pattern = "^.*_.*_\d$"
result = []

for phone in phones:
    match = re.search(pattern, phone)
    if match !=None:
        result.append(phone)

# print result - result should only contain the items that match the pattern
print(result)
```

```
['apple_iphone_6', 'samsung_note_4', 'google_pixel_2']
```

Q5.

Write a regular expression that can be used to match the emails present in a database.

The pattern of a valid email address is defined as follows: The '@' character can be preceded either by alphanumeric characters, period characters or underscore characters. The length of the part that precedes the '@' character should be between 4 to 20 characters.

The '@' character should be followed by a domain name (e.g. gmail.com). The domain name has three parts - a prefix (e.g. 'gmail'), the period character and a suffix (e.g. 'com'). The prefix can have a length between 3 to 15 characters followed by a period character followed by either of these suffixes - 'com', 'in' or 'org'.

Emails that should match:

random.guy123@gmail.com (<mailto:random.guy123@gmail.com>)

mr_x_in_bombay@gov.in (mailto:mr_x_in_bombay@gov.in)

Emails that shouldn't match:

1@ued.org (<mailto:1@ued.org>)

[@gmail.com](#)

abc!@yahoo.in (<mailto:abc!@yahoo.in>)

sam_12@gov.us (mailto:sam_12@gov.us)

[neeraj@](#)

In [251]:

```
emails = ['random.guy123@gmail.com', 'mr_x_in_bombay@gov.in', '1@ued.org',
          '@gmail.com', 'abc!@yahoo.in', 'sam_12@gov.us', 'neeraj@']

pattern = "^[a-z_.0-9]{4,20}@[a-z]{3,15}\.(com|in|org)$"

result = []

for email in emails:
    match = re.search(pattern, email, re.I)
    if match != None:
        result.append(email)

# print result - result should only contain the items that match the pattern
print(result)
```

```
['random.guy123@gmail.com', 'mr_x_in_bombay@gov.in']
```

In this session, you learnt about the different areas where text analytics is applied such as healthcare, e-commerce, retail, financial and various other industries. Then you learnt about the stack that is generally followed to extract insights from the text and to build various applications of natural language processing. You learn there are three stages in text analytics:

Lexical processing

Syntactic processing

Semantic processing

Then you learnt about text encoding and its various types such as ASCII and Unicode. You learnt how to change between different types of Unicode encodings in Python.

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

Finally, you can refer to this link(<https://pycon2016.regex.training/cheat-sheet> (<https://pycon2016.regex.training/cheat-sheet>)) whenever you want a refresher in regular expressions in Python. There are some of the concepts that we've left untouched in regular expressions. But as someone who is working in the area of text analytics, you can achieve pretty much everything using the tools that you have learnt.

In []: