Regex(usnig python) is best to get idea how will we text manipulation in the data Science

In this lecture we're going to talk about pattern matching in strings using regular expressions. Regular expressions, or regexes, are written in a condensed formatting language. In general, you can think of a regular expression as a pattern which you give to a regex processor with some source data. The processor then parses that source data using that pattern, and returns chunks of text back to the a data scientist or programmer for further manipulation. There's really three main reasons you would want to do this - to check whether a pattern exists within some source data, to get all instances of a complex pattern from some source data, or to clean your source data using a pattern generally through string splitting. Regexes are not trivial, but they are a foundational technique for data cleaning in data science applications, and a solid understanding of regexs will help you quickly and efficiently manipulate text data for further data science application.

Now, you could teach a whole course on regular expressions alone, especially if you wanted to demystify how the regex parsing engine works and efficient mechanisms for parsing text. In this lecture I want to give you basic understanding of how regex works - enough knowledge that, with a little directed sleuthing, you'll be able to make sense of the regex patterns you see others use, and you can build up your practical knowledge of how to use regexes to improve your data cleaning. By the end of this lecture, you will understand the basics of regular expressions, how to define patterns for matching, how to apply these patterns to strings, and how to use the results of those patterns in data processing.

Finally, a note that in order to best learn regexes you need to write regexes. I encourage you to stop the video at any time and try out new patterns or syntax you learn at any time.

Search Method

```
# First we'll import the re module, which is where python stores regular expression libraries.
import re
```

```
# There are several main processing functions in re that you might use. The first, match() checks for a match
# that is at the beginning of the string and returns a boolean. Similarly, search(), checks for a match
# anywhere in the string, and returns a boolean.

# Lets create some text for an example
text = "This is a good day."

# Now, lets see if it's a good day or not:
if re.search("good", text): # the first parameter here is the pattern
    print("Wonderful!")
else:
    print("Alas :(")
```

```
Wonderful!
```

findAll() and splitall() method functioon. split method split the string into the chunk and findall collect the deserving data from the split output

```
# In addition to checking for conditionals, we can segment a string. The work that regex does here is called
# tokenizing, where the string is separated into substrings based on patterns. Tokenizing is a core activity
# in natural language processing, which we won't talk much about here but that you will study in the future

# The findall() and split() functions will parse the string for us and return chunks. Lets try and example
text = "Amy works diligently. Amy gets good grades. Our student Amy is succesful."

# This is a bit of a fabricated example, but lets split this on all instances of Amy
re.split("Amy", text)
```

```
['',
 ' works diligently. ',
 ' gets good grades. Our student ',
 ' is succesful.']
```

```
# You'll notice that split has returned an empty string, followed by a number of statements about Amy, all as
# elements of a list. If we wanted to count how many times we have talked about Amy, we could use findall()
re.findall("Amy", text)
```

```
['Amy', 'Amy', 'Amy']
```

```
# Ok, so we've seen that .search() looks for some pattern and returns a boolean, that .split() will use a
# pattern for creating a list of substrings, and that .findall() will look for a pattern and pull out all
# occurences.
```

used of character ^ start and $ mean at the end and How work API of regex

```
# Now that we know how the python regex API works, lets talk about more complex patterns. The regex
# specification standard defines a markup language to describe patterns in text. Lets start with anchors.
# Anchors specify the start and/or the end of the string that you are trying to match. The caret character ^
# means start and the dollar sign character $ means end. If you put ^ before a string, it means that the text
# the regex processor retrieves must start with the string you specify. For ending, you have to put the $
# character after the string, it means that the text Regex retrieves must end with the string you specify.

# Here's an example
text = "Amy works diligently. Amy gets good grades. Our student Amy is succesful."

# Lets see if this begins with Amy
re.search("^Amy",text)
```

```
<re.Match object; span=(0, 3), match='Amy'>
```

```
# Notice that re.search() actually returned to us a new object, called re.Match object. An re.Match object
# always has a boolean value of True, as something was found, so you can always evaluate it in an if statement
# as we did earlier. The rendering of the match object also tells you what pattern was matched, in this case
# the word Amy, and the location the match was in, as the span.
```

Patterns and Character Classes

[8]:
```
# Let's talk more about patterns and start with character classes. Let's create a string of a single learners'
# grades over a semester in one course across all of their assignments
grades="ACAAAABCBCBAA"

# If we want to answer the question "How many B's were in the grade list?" we would just use B
re.findall("B",grades)
```

[8]:
```
['B', 'B', 'B']
```

[9]:
```
# If we wanted to count the number of A's or B's in the list, we can't use "AB" since this is used to match
# all A's followed immediately by a B. Instead, we put the characters A and B inside square brackets
re.findall("[AB]",grades)
```

[9]:
```
['A', 'A', 'A', 'A', 'A', 'B', 'B', 'B', 'A', 'A']
```

[10]:
```
# This is called the set operator. You can also include a range of characters, which are ordered
# alphanumerically. For instance, if we want to refer to all lower case letters we could use [a-z] Lets build
# a simple regex to parse out all instances where this student receive an A followed by a B or a C
re.findall("[A][B-C]",grades)
```

[10]:
```
['AC', 'AB']
```

In [10]:
```
# This is called the set operator. You can also include a range of characters, which are ordered
# alphanumerically. For instance, if we want to refer to all lower case letters we could use [a-z] Lets build
# a simple regex to parse out all instances where this student receive an A followed by a B or a C
re.findall("[A][B-C]",grades)
```

Out[10]:
```
['AC', 'AB']
```

In [11]:
```
# Notice how the [AB] pattern describes a set of possible characters which could be either (A OR B), while the
# [A][B-C] pattern denoted two sets of characters which must have been matched back to back. You can write
# this pattern by using the pipe operator, which means OR
re.findall("AB|AC",grades)
```

Out[11]:
```
['AC', 'AB']
```

In [12]:
```
# We can use the caret with the set operator to negate our results. For instance, if we want to parse out only
# the grades which were not A's
re.findall("[^A]",grades)
```

Out[12]:
```
['C', 'B', 'C', 'B', 'C', 'B']
```

In [ ]:
```
# Note this carefully - the caret was previously matched to the beginning of a string as an anchor point, but
# inside of the set operator the caret, and the other special characters we will be talking about, lose their
# meaning. This can be a bit confusing. What do you think the result would be of this?
re.findall("^[^A]",grades)
```

Use of ^ opeator

```
# We can use the caret with the set operator to negate our results. For instance, if we want to parse out only
# the grades which were not A's
re.findall("[^A]",grades)
```

```
['C', 'B', 'C', 'B', 'C', 'B']
```

```
# Note this carefully - the caret was previously matched to the beginning of a string as an anchor point, but
# inside of the set operator the caret, and the other special characters we will be talking about, lose their
# meaning. This can be a bit confusing. What do you think the result would be of this?
re.findall("^[^A]",grades)
```

```
[]
```

```
# It's an empty list, because the regex says that we want to match any value at the beginning of the string
# which is not an A. Our string though starts with an A, so there is no match found. And remember when you are
# using the set operator you are doing character-based matching. So you are matching individual characters in
# an OR method.
```

Qunatifiers and Used list{min , max} operator

Quantifiers

In [15]:
```
# Ok, so we've talked about anchors and matching to the beginning and end of patterns. And we've talked about
# characters and using sets with the [] notation. We've also talked about character negation, and how the pipe
# | character allows us to or operations. Lets move on to quantifiers.
```

In [16]:
```
# Quantifiers are the number of times you want a pattern to be matched in order to match. The most basic
# quantifier is expressed as e{m,n}, where e is the expression or character we are matching, m is the minimum
# number of times you want it to matched, and n is the maximum number of times the item could be matched.

# Let's use these grades as an example. How many times has this student been on a back-to-back A's streak?
re.findall("A{2,10}",grades) # we'll use 2 as our min, but ten as our max
```

Out[16]:
```
['AAAA', 'AA']
```

In [17]:
```
# So we see that there were two streaks, one where the student had four A's, and one where they had only two
# A's

# We might try and do this using single values and just repeating the pattern
re.findall("A{1,1}A{1,1}",grades)
```

Out[17]:
```
['AA', 'AA', 'AA']
```

In [ ]:
```
# As you can see, this is different than the first example. The first pattern is looking for any combination
# of two A'
```

Qunatifiers and Used list{min , max} operator

In [18]:
```
# As you can see, this is different than the first example. The first pattern is looking for any combination
# of two A's up to ten A's in a row. So it sees four A's as a single streak. The second pattern is looking for
# two A's back to back, so it sees two A's followed immediately by two more A's. We say that the regex
# processor begins at the start of the string and consumes variables which match patterns as it does.

# It's important to note that the regex quantifier syntax does not allow you to deviate from the {m,n}
# pattern. In particular, if you have an extra space in between the braces you'll get an empty result
re.findall("A{2, 2}",grades)
```

Out[18]:
```
[]
```

In [19]:
```
# And as we have already seen, if we don't include a quantifier then the default is {1,1}
re.findall("AA",grades)
```

Out[19]:
```
['AA', 'AA', 'AA']
```

In [20]:
```
# Oh, and if you just have one number in the braces, it's considered to be both m and n
re.findall("A{2}",grades)
```

Out[20]:
```
['AA', 'AA', 'AA']
```

In [21]:
```
# Using this, we could find a decreasing trend in a student's grades
re.findall("A{1,10}B{1,10}C{1,10}",grades)
```

Out[21]:
```
['AAAABC']
```

Scrapping from the websites

```
]: # Now, that's a bit of a hack, because we included a maximum that was just arbitrarily large. There are three
   # other quantifiers that are used as short hand, an asterix * to match 0 or more times, a question mark ? to
   # match one or more times, or a + plus sign to match one or more times. Lets look at a more complex example,
   # and load some data scraped from wikipedia
   with open("datasets/ferpa.txt","r") as file:
       # we'll read that into a variable called wiki
       wiki=file.read()
   # and lets print that variable out to the screen
```

```
ds will remain under the protection of FERPA, not the Health Insurance Portability and Accountability Act (HIPAA).
This is due to the "FERPA Exception" written within HIPAA.[9]'
```

```
In [23]: # Scanning through this document one of the things we notice is that the headers all have the words [edit]
         # behind them, followed by a newline character. So if we wanted to get a list of all of the headers in this
         # article we could do so using re.findall
         re.findall("[a-zA-Z]{1,100}\[edit\]",wiki)
```

```
Out[23]: ['Overview[edit]', 'records[edit]', 'records[edit]']
```

```
In [24]: # Ok, that didn't quite work. It got all of the headers, but only the last word of the header, and it really
         # was quite clunky. Lets iteratively improve this. First, we can use \w to match any letter, including digits
         # and numbers.
         re.findall("[\w]{1,100}\[edit\]",wiki)
```

```
Out[24]: ['Overview[edit]', 'records[edit]', 'records[edit]']
```

```
In [25]: # This is something new. \w is a metacharacter, and indicates a special pattern of any letter or digit. There
         # are actually a number of different metacharacters listed in the documentation. For instance, \s matches any
         # whitespace character.

         # Next, there are three other quantifiers we can use which shorten up the curly brace syntax. We can use an
         # asterix * to match 0 or more times, so let's try that.
         re.findall("[\w]*\[edit\]",wiki)
```

```
Out[25]: ['Overview[edit]', 'records[edit]', 'records[edit]']
```