Manipulating Text With Regular Expressions

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.

In [6]:

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

Main Functions in RegEx

In [2]:

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

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

In [3]:

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

# This is a bit of a fabricated example, but lets split this on all instances of
Amy
re.split("Amy", text)
#Note that the '   ' denotes the second Amy in the first line.
```

Out[3]:

```
['',
 ' ',
 ' 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.

In [4]:

```
#If we wanted to count how many times we have talked about Amy, we could use fin
dall()
print(re.findall("Amy", text))
len(re.findall("Amy", text))
```

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

Out[4]:

```
4
```

More Complex Patterns : Anchors

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.

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.

In [5]:

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

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

Out[5]:

```
<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

In [6]:

```
# 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 wo
uld just use B
re.findall("B",grades)
```

Out[6]:

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

Set Operator [ ]

In [7]:

```
# 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 in
side square brackets
re.findall("[AB]",grades)
```

Out[7]:

```
['A', 'A', 'A', 'A', 'A', 'B', 'B', 'B', 'A', 'A']
```

In [8]:

```
# This is called the set operator. You can also include a range of characters, w
hich 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 foll
owed by a B or a C
re.findall("[A][B-C]",grades)
```

Out[8]:

```
['AC', 'AB']
```

In [9]:

```
# 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 b
ack to back. You can write
# this pattern by using the pipe operator, which means OR
re.findall("AB|AC",grades)
```

Out[9]:

```
['AC', 'AB']
```

In [10]:

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

Out[10]:

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

In [11]:

```
# Note this carefully - the caret was previously matched to the beginning of a s
tring 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)
```

Out[11]:

```
[]
```

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.

Quantifiers

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.

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.

In [12]:

```
# 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[12]:

```
['AAAA', 'AA']
```

In [13]:

```
# 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[13]:

```
['AA', 'AA', 'AA']
```

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

In [14]:

```
re.findall("A{2, 2}",grades)
```

Out[14]:

```
[]
```

In [15]:

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

Out[15]:

```
['AA', 'AA', 'AA']
```

In [16]:

```
# 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[16]:

```
['AA', 'AA', 'AA']
```

In [17]:

```
# 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[17]:

```
['AAAABC']
```

Now, that's a bit of a hack, because we included a maximum that was just arbitrarily large.

More Quantifiers

There are three other quantifiers that are used as short hand,

- asterix * to match 0 or more times,
- question mark ? to match one or more times,
- + plus sign to match one or more times.

In [12]:

```
# 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
print(wiki[0:600])
print("...")
```

```
Overview[edit]
FERPA gives parents access to their child's education records, an op
portunity to seek to have the records amended, and some control over
the disclosure of information from the records. With several excepti
ons, schools must have a student's consent prior to the disclosure o
f education records after that student is 18 years old. The law appl
ies only to educational agencies and institutions that receive funds
under a program administered by the U.S. Department of Education.

Other regulations under this act, effective starting January 3, 201
2, allow for greater disclosures of perso
...
```

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.

In [13]:

```
#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[13]:

```
['Overview[edit]', 'records[edit]', 'records[edit]']
```

Meta-Character: \w \s

In [14]:

```
# 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[14]:

```
['Overview[edit]', 'records[edit]', 'records[edit]']
```

In [15]:

```
# 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[15]:

```
['Overview[edit]', 'records[edit]', 'records[edit]']
```

In [16]:

```
# Now that we have shortened the regex, let's improve it a little bit. We can ad
d in a spaces using the space
# character. Note that we added a space between /w and the closing square brace.
re.findall("[\w ]*\[edit\]",wiki)
```

Out[16]:

```
['Overview[edit]',
 'Access to public records[edit]',
 'Student medical records[edit]']
```

In [17]:

```
# Ok, so this gets us the list of section titles in the wikipedia page! You can
 now create a list of titles by
# iterating through this and applying another regex
for title in re.findall("[\w ]*\[edit\]",wiki):
    # Now we will take that intermediate result and split on the opening square
 bracket [ just taking the first result
    # Rmbr that split gives us a list of strings, so take the first string which
is the title.
    print(re.split("[\[]",title)[0])
```

```
Overview
Access to public records
Student medical records
```

Groups

Ok, this works, but it's a bit of a pain. To this point we have been talking about a regex as a single pattern which is matched. But, you can actually match different patterns, called groups, at the same time, and then refer to the groups you want.

To group patterns together you use parentheses, which is actually pretty natural.

In [18]:

```
#Lets rewrite our findall using groups
re.findall("([\w ]*)(\[edit\])",wiki)
```

Out[18]:

```
[('Overview', '[edit]'),
 ('Access to public records', '[edit]'),
 ('Student medical records', '[edit]')]
```

Nice - we see that the python re module breaks out the result by group. We can actually refer to groups by number as well with the match objects that are returned.

But, how do we get back a list of match objects?

In [11]:

```
# Thus far we've seen that findall() returns strings, and search() and match() r
eturn individual Match
# objects. But what do we do if we want a list of Match objects? In this case, w
e use the function finditer()
for item in re.finditer("([\w ]*)(\[edit\])",wiki):
    print(item.groups())
```

```
('References', '[edit]')
('External links', '[edit]')
```

In [26]:

```
# We see here that the groups() method returns a tuple of the group. We can get
 an individual group using
# group(number), where group(0) is the whole match, and each other number is the
portion of the match we are
# interested in. In this case, we want group(1)
for item in re.finditer("([\w ]*)(\[edit\])",wiki):
    print(item.group(1))
#In this case, item.group(2) would be the word [edit]
```

```
Overview
Access to public records
Student medical records
```

Labelling & Naming Groups

One more piece to regex groups that I rarely use but is a good idea is labeling or naming groups. In the previous example I showed you how you can use the position of the group. But giving them a label and looking at the results as a dictionary is pretty useful.

Syntax is: **(**?P[pattern] **)** because **each key-value pair is a group.**

In [27]:

```
#For that we use the syntax (?P<name>), where the parethesis starts the group, t
he ?P indicates that this is an extension to basic regexes, and <name> is the di
ctionary
# key we want to use wrapped in <>.
for item in re.finditer("(?P<title>[\w ]*)(?P<edit_link>\[edit\])",wiki):
    # We can get the dictionary returned for the item with .groupdict()
    print(item.groupdict()['title'])
```

```
Overview
Access to public records
Student medical records
```

In [28]:

```
# Of course, we can print out the whole dictionary for the item too, and see tha
t the [edit] string is still
# in there. Here's the dictionary kept for the last match
print(item.groupdict())
```

```
{'title': 'Student medical records', 'edit_link': '[edit]'}
```

Ok, we have seen how we can ...

- match individual character patterns with [],
- group matches together using (),
- use quantifiers such as *, ?, or m{n} to describe patterns.

Something I glossed over in the previous example was the \w, which standards for any word character. There are a number of short hands which are used with regexes for different kinds of characters, including: a . for any single character which is not a newline a \d for any digit and \s for any whitespace character, like spaces and tabs There are more, and a full list can be found in the python documentation for regexes

Look-ahead and Look-behind : ?=

One more concept to be familiar with is called "look ahead" and "look behind" matching. In this case, the pattern being given to the regex engine is for text either before or after the text we are trying to isolate.

For example, in our headers we want to isolate text which comes before the [edit] rendering, but we actually don't care about the [edit] text itself. Thus far we have been throwing the [edit] away, but if we want to use them to match but don't want to capture them **we could put them in a group and use look ahead instead with ?= syntax** .

In [29]:

```
# For example, in our headers we want to isolate text which  comes before the [e
dit] rendering, but
# we actually don't care about the [edit] text itself. Thus far we have been thr
owing the [edit] away, but if
# we want to use them to match but don't want to capture them we could put them
 in a group and use look ahead
# instead with ?= syntax
for item in re.finditer("(?P<title>[\w ]+)(?=\[edit\])",wiki):
    # What this regex says is match two groups, the first will be named and call
ed title, will have any amount
    # of whitespace or regular word characters, the second will be the character
s [edit] but we don't actually
    # want this edit put in our output match objects
    print(item)
```

```
<re.Match object; span=(0, 8), match='Overview'>
<re.Match object; span=(2715, 2739), match='Access to public record
s'>
<re.Match object; span=(3692, 3715), match='Student medical record
s'>
```

re.VERBOSE( ) & Multi-Line RegEx Patterns

In [9]:

```
# Let's look at some more wikipedia data. Here's some data on universities in th
e US which are buddhist-based
with open("datasets/buddhist.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
wiki
```

Out[9]:

```
'Buddhist universities and colleges in the United States\nFrom Wikip
edia, the free encyclopedia\nJump to navigationJump to search\n\nThi
s article needs additional citations for verification. Please help i
mprove this article by adding citations to reliable sources. Unsourc
ed material may be challenged and removed.\nFind sources: "Buddhist
universities and colleges in the United States" — news · newspapers
· books · scholar · JSTOR (December 2009) (Learn how and when to rem
ove this template message)\nThere are several Buddhist universities
in the United States. Some of these have existed for decades and are
accredited. Others are relatively new and are either in the process
of being accredited or else have no formal accreditation. The list i
ncludes:\n\nDhammakaya Open University — located in Azusa, Californi
a, part of the Thai Wat Phra Dhammakaya[1]\nDharmakirti College — lo
cated in Tucson, Arizona Now called Awam Tibetan Buddhist Institute
(http://awaminstitute.org/)\nDharma Realm Buddhist University — loca
ted in Ukiah, California\nEwam Buddhist Institute — located in Arle
e, Montana\nNaropa University is located in Boulder, Colorado (Accre
dited by the Higher Learning Commission)\nInstitute of Buddhist Stud
ies — located in Berkeley, California\nMaitripa College — located in
Portland, Oregon\nSoka University of America — located in Aliso Viej
o, California\nUniversity of the West — located in Rosemead, Califor
nia\nWon Institute of Graduate Studies — located in Glenside, Pennsy
lvania\nReferences[edit]\n^ Banchanon, Phongphiphat (3 February 201
5). รู้จัก "เครือข่ายธรรมกาย" [Getting to know the Dhammakaya network]. Fo
rbes Thailand (in Thai). Retrieved 11 November 2016.\nExternal links
[edit]\nList of Buddhist Universities and Colleges in the world\n'
```

In [8]:

```
# We can see that each university follows a fairly similar pattern, with the nam
e followed by an — then the
# words "located in" followed by the city and state

# I'll actually use this example to show you the verbose mode of python regexes.
The verbose mode allows you
# to write multi-line regexes and increases readability. For this mode, we have
 to explicitly indicate all
# whitespace characters, either by prepending them with a \ or by using the \s s
pecial value. However, this
# means we can write our regex a bit more like code, and can even include commen
ts with #
pattern="""#multiline string
(?P<title>.*)         #the university title
(—\ located\ in\ )    #an indicator of the location
(?P<city>\w*)         #city the university is in
(,\ )                 #separator for the state
(?P<state>\w*)        #the state the city is located in"""

# Now when we call finditer() we just pass the re.VERBOSE flag as the last param
eter, this makes it much
# easier to understand large regexes!
for item in re.finditer(pattern,wiki,re.VERBOSE):
    # We can get the dictionary returned for the item with .groupdict()
    print(item.groupdict())
```

```
{'title': 'Dhammakaya Open University ', 'city': 'Azusa', 'state':
'California'}
{'title': 'Dharmakirti College ', 'city': 'Tucson', 'state': 'Arizon
a'}
{'title': 'Dharma Realm Buddhist University ', 'city': 'Ukiah', 'sta
te': 'California'}
{'title': 'Ewam Buddhist Institute ', 'city': 'Arlee', 'state': 'Mon
tana'}
{'title': 'Institute of Buddhist Studies ', 'city': 'Berkeley', 'sta
te': 'California'}
{'title': 'Maitripa College ', 'city': 'Portland', 'state': 'Orego
n'}
{'title': 'University of the West ', 'city': 'Rosemead', 'state': 'C
alifornia'}
{'title': 'Won Institute of Graduate Studies ', 'city': 'Glenside',
'state': 'Pennsylvania'}
```

Example: New York Times and Hashtags

In [32]:

```
# Here's another example from the New York Times which covers health tweets on n
ews items. This data came from
# the UC Irvine Machine Learning Repository which is a great source of different
kinds of data
with open("datasets/nytimeshealth.txt","r") as file:
    # We'll read everything into a variable and take a look at it
    health=file.read()
health[:600]
```

Out[32]:

```
"548662191340421120|Sat Dec 27 02:10:34 +0000 2014|Risks in Using So
cial Media to Spot Signs of Mental Distress http://nyti.ms/1rqi9I1\n
548579831169163265|Fri Dec 26 20:43:18 +0000 2014|RT @paula_span: Th
e most effective nationwide diabetes prevention program you've proba
bly never heard of:  http://newoldage.blogs.nytimes.com/2014/12/26/d
iabetes-prevention-that-works/\n548579045269852161|Fri Dec 26 20:40:
11 +0000 2014|The New Old Age Blog: Diabetes Prevention That Works h
ttp://nyti.ms/1xm7fTi\n548444679529041920|Fri Dec 26 11:46:15 +0000
2014|Well: Comfort Casseroles for Winter Dinners http://nyti"
```

In [33]:

```
# So here we can see there are tweets with fields separated by pipes |. Lets try
and get a list of all of the
# hashtags that are included in this data. A hashtag begins with a pound sign (o
r hash mark) and continues
# until some whitespace is found

# So lets create a pattern. We want to include the hash sign first, then any num
ber of alphanumeric
# characters. And we end when we see some whitespace
pattern = '#[\w\d]*(?=\s)' # a hashmark, any no. of non-whitespace letters or ch
ars
# and we look ahead to see if we can find any whitespace

# Notice that the ending is a look ahead. We're not actually interested in match
ing whitespace in the return
# value. Also notice that I use an asterix * instead of the plus + for the match
ing of alphabetical characters
# or digits, because a + would require at least one of each

# Lets searchg and display all of the hashtags
re.findall(pattern, health)[:15]
```

Out[33]:

```
['#askwell',
 '#pregnancy',
 '#Colorado',
 '#VegetarianThanksgiving',
 '#FallPrevention',
 '#Ebola',
 '#Ebola',
 '#ebola',
 '#Ebola',
 '#Ebola',
 '#EbolaHysteria',
 '#AskNYT',
 '#Ebola',
 '#Ebola',
 '#Liberia']
```

We can see here that there were lots of ebola related tweeks in this particular dataset.

Summary

This lecture has been an overview of regular expressions, and really, we've just scratched the surface of what you can do. Now, I actually find regex's really frustrating - they're incredibly powerful, but if you don't use them for awhile you're left grasping for memory of some of the details, especially named groups and look ahead searches. But, there are lots of great examples and reference guides on the web, including the python documentation for regex, and with these in hand you should be able to write concise and readable code which performs well too. Having basic regex literacy is a core skill for applied data scientists.