Regex_ed

November 7, 2020

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.

[1]:
```
# First we'll import the re module, which is where python stores regular␣
 ↪expression libraries.
import re
```

[2]:
```
# 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!
```

[3]:
```
# 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)
```

[3]:
```
['',
 ' works diligently. ',
 ' gets good grades. Our student ',
 ' is succesful.']
```

[4]:
```
# 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)
```

[4]:
```
['Amy', 'Amy', 'Amy']
```

[5]:
```
# 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.
```

[6]:
```
# 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)
```

[6]:
```
<re.Match object; span=(0, 3), match='Amy'>
```

[7]:
```
# 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.
```

1 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']
```

[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)
```

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

[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)
```

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

[13]:
```
# 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)
```

[13]:
```
[]
```

[14]:
```
# 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.
```

2 Quantifiers

[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.
```

[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
```

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

[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)
```

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

[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)
```

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

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

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

[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)
```

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

[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)
```

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

[22]:
```
# 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
wiki
```

[22]:
```
'Overview[edit]\nFERPA gives parents access to their child\'s education records,
an opportunity to seek to have the records amended, and some control over the
disclosure of information from the records. With several exceptions, schools
must have a student\'s consent prior to the disclosure of education records
after that student is 18 years old. The law applies only to educational agencies
and institutions that receive funds under a program administered by the U.S.
Department of Education.\n\nOther regulations under this act, effective starting
January 3, 2012, allow for greater disclosures of personal and directory student
identifying information and regulate student IDs and e-mail addresses.[2] For
example, schools may provide external companies with a student\'s personally
identifiable information without the student\'s consent.[2]\n\nExamples of
situations affected by FERPA include school employees divulging information to
anyone other than the student about the student\'s grades or behavior, and
school work posted on a bulletin board with a grade. Generally, schools must
have written permission from the parent or eligible student in order to release
any information from a student\'s education record.\n\nThis privacy policy also
governs how state agencies transmit testing data to federal agencies, such as
the Education Data Exchange Network.\n\nThis U.S. federal law also gave students
18 years of age or older, or students of any age if enrolled in any post-
secondary educational institution, the right of privacy regarding grades,
enrollment, and even billing information unless the school has specific
permission from the student to share that specific type of information.\n\nFERPA
also permits a school to disclose personally identifiable information from
education records of an "eligible student" (a student age 18 or older or
enrolled in a postsecondary institution at any age) to his or her parents if the
student is a "dependent student" as that term is defined in Section 152 of the
Internal Revenue Code. Generally, if either parent has claimed the student as a
dependent on the parent\'s most recent income tax statement, the school may non-
consensually disclose the student\'s education records to both
parents.[3]\n\nThe law allowed students who apply to an educational institution
such as graduate school permission to view recommendations submitted by others
as part of the application. However, on standard application forms, students are
given the option to waive this right.\n\nFERPA specifically excludes employees
of an educational institution if they are not students.\n\nThe act is also
```

```
referred to as the Buckley Amendment, for one of its proponents, Senator James
L. Buckley of New York.\n\nAccess to public records[edit]\nThe citing of FERPA
to conceal public records that are not "educational" in nature has been widely
criticized, including by the act\'s primary Senate sponsor.[4] For example, in
the Owasso Independent School District v. Falvo case, an important part of the
debate was determining the relationship between peer-grading and "education
records" as defined in FERPA. In the Court of Appeals, it was ruled that
students placing grades on the work of other students made such work into an
"education record." Thus, peer-grading was determined as a violation of FERPA
privacy policies because students had access to other students\' academic
performance without full consent.[5] However, when the case went to the Supreme
Court, it was officially ruled that peer-grading was not a violation of FERPA.
This is because a grade written on a student\'s work does not become an
"education record" until the teacher writes the final grade into a grade
book.[6]\n\nStudent medical records[edit]\nLegal experts have debated the issue
of whether student medical records (for example records of therapy sessions with
a therapist at an on-campus counseling center) might be released to the school
administration under certain triggering events, such as when a student sued his
college or university.[7][8]\n\nUsually, student medical treatment records 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]'
```

[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)
```

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

[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)
```

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

[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)
```

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

[26]:
```
# Now that we have shortened the regex, let's improve it a little bit. We can␣
 ↪add in a spaces using the space
# character
re.findall("[\w ]*\[edit\]",wiki)
```

[26]:
```
['Overview[edit]',
 'Access to public records[edit]',
 'Student medical records[edit]']
```

[27]:
```
# 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 square bracket␣
 ↪[ just taking the first result
    print(re.split("[\[]",title)[0])
```

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

3 Groups

[28]:
```
# 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. Lets rewrite our findall using groups
re.findall("([\w ]*)(\[edit\])",wiki)
```

[28]:
```
[('Overview', '[edit]'),
 ('Access to public records', '[edit]'),
 ('Student medical records', '[edit]')]
```

[29]:
```
# 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?
# Thus far we've seen that findall() returns strings, and search() and match()␣
 ↪return individual Match
# objects. But what do we do if we want a list of Match objects? In this case,␣
 ↪we use the function finditer()
```

```
for item in re.finditer("([\w ]*)(\[edit\])",wiki):
    print(item.groups())
```

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

[30]:
```
# 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))
```

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

[31]:
```
# 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. For that we use the syntax (?
 ↪P<name>), where the parethesis
# starts the group, the ?P indicates that this is an extension to basic␣
 ↪regexes, and <name> is the dictionary
# 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
```

[32]:
```
# Of course, we can print out the whole dictionary for the item too, and see␣
 ↪that 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]'}
```

[33]:
```
# Ok, we have seen how we can match individual character patterns with [], how␣
 ↪we can group matches together
```

```
# using (), and how we can 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
```

4 Look-ahead and Look-behind

[34]:
```
# 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
for item in re.finditer("(?P<title>[\w ]+)(?=\[edit\])",wiki):
    # What this regex says is match two groups, the first will be named and␣
 ↪called title, will have any amount
    # of whitespace or regular word characters, the second will be the␣
 ↪characters [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 records'>
<re.Match object; span=(3692, 3715), match='Student medical records'>
```

5 Example: Wikipedia Data

[35]:
```
# Let's look at some more wikipedia data. Here's some data on universities in␣
 ↪the 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
```

[35]:
```
'Buddhist universities and colleges in the United States\nFrom Wikipedia, the
free encyclopedia\nJump to navigationJump to search\n\nThis article needs
additional citations for verification. Please help improve this article by
adding citations to reliable sources. Unsourced material may be challenged and
removed.\nFind sources: "Buddhist universities and colleges in the United
States"  news ů newspapers ů books ů scholar ů JSTOR (December 2009) (Learn how
and when to remove 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
includes:\n\nDhammakaya Open University  located in Azusa, California, part of
the Thai Wat Phra Dhammakaya[1]\nDharmakirti College  located in Tucson,
Arizona Now called Awam Tibetan Buddhist Institute
(http://awaminstitute.org/)\nDharma Realm Buddhist University  located in
Ukiah, California\nEwam Buddhist Institute  located in Arlee, Montana\nNaropa
University is located in Boulder, Colorado (Accredited by the Higher Learning
Commission)\nInstitute of Buddhist Studies  located in Berkeley,
California\nMaitripa College  located in Portland, Oregon\nSoka University of
America  located in Aliso Viejo, California\nUniversity of the West  located
in Rosemead, California\nWon Institute of Graduate Studies  located in
Glenside, Pennsylvania\nReferences[edit]\n^ Banchanon, Phongphiphat (3 February
2015).  "" [Getting to know the Dhammakaya network].
Forbes Thailand (in Thai). Retrieved 11 November 2016.\nExternal
links[edit]\nList of Buddhist Universities and Colleges in the world\n'
```

[36]:
```
# We can see that each university follows a fairly similar pattern, with the␣
 ↪name 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␣
 ↪special value. However, this
# means we can write our regex a bit more like code, and can even include␣
 ↪comments with #
pattern="""
(?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"""
```