

LECTURE 7

# Regular Expressions

Using string methods and regular expressions to work with textual data

# Goals For This Lecture

---

## Working With Text Data

- Canonicalizing text data.
- Extracting data from text.
  - Using **split**.
  - Using **regular expressions**.

# Regular Expression Basics

# Regular Expression Syntax

---

The four basic operations for regular expressions.

- Can technically do anything with just these basic four (albeit tediously).

operation	order	example	matches	does not match
concatenation	3	AABAAB	AABAAB	every other string
or	4	AA BAAB	AA BAAB	every other string
closure (zero or more)	2	AB*A	AA ABBBBBBA	AB ABABA
parenthesis	1	A(A B)AAB	AAAAB ABAAB	every other string
		(AB)*A	A ABABABABA	AA ABBA

# Regular Expression Syntax

$AB^*$ : A then zero or more copies of B: A, AB, ABB, ABBB

$(AB)^*$ : Zero or more copies of AB: ABABABAB, ABAB, AB,

Matches the  
empty string!



operation	order	example	matches	does not match
concatenation	3	AABAAB	AABAAB	every other string
or	4	AA   BAAB	AA BAAB	every other string
closure (zero or more)	2	$AB^*A$	AA ABBBBBBA	AB ABABA
parenthesis	1	$A(A B)AAB$	AAAAB ABAAB	every other string
		$(AB)^*A$	A ABABABABA	AA ABBA

# Order of Operations in Regexes

---

`m(uu(uu)* | oo(oo)* )n`

- Matches starting with m and ending with n, with either of the following in the middle:
  - uu(uu)\*
  - oo(oo)\*

Match examples:

muun

muuuun

moon

mooooon

# Order of Operations in Regexes

---

`m(uu(uu)* | oo(oo)* )n`

- Matches starting with m and ending with n, with either of the following in the middle:
  - uu(uu)\*
  - oo(oo)\*

Match examples:

muun  
muuuun  
moon  
mooooon

`m(uu(uu)* ) | (oo(oo)* )n`

- Matches either of the following
  - m followed by uu(uu)\*
  - oo(oo)\* followed by n

Match examples:

muu  
muuuu  
oon  
ooooon

In regexes | comes last.

# Expanded Regex Syntax

---

operation	example	matches	does not match
any character (except newline)	.U.U.U.	CUMULUS JUGULUM	SUCCUBUS TUMULTUOUS
character class	[A-Za-z][a-z]*	word Capitalized	camelCase 4illegal
at least one	jo+hn	john joooooooohn	jhn jjohn
zero or one	joh?n	jon john	any other string
repeated exactly {a} times	j[aeiou]{3}hn	jaoehn jooohn	jhn jaeiouhn
repeated from a to b times: {a,b}	j[ou]{1,2}hn	john juohn	jhn jooohn



# More Regular Expression Examples

regex	matches	does not match
<code>.*SPB.*</code>	RASPBERRY CRISPBREAD	SUBSPACE SUBSPECIES
<code>[0-9]{3}-[0-9]{2}-[0-9]{4}</code>	231-41-5121 573-57-1821	231415121 57-3571821
<code>[a-z]+@([a-z]+\.)+((edu\.cn) edu com)</code>	horse@pizza.com horse@pizza.food.com	frank_99@yahoo.com hug@cs

Example: `1[0-9]{2}-[0-9]{4}-[0-9]{4}`

3 of digits starting from 1, then a dash, then 4 of any digit, then a dash, then 4 of any digit.

## Exercises

---

- Give a regular expression for any lowercase string that has a repeated vowel (i.e. noon, peel, festoon, loop, etc).

## Exercises

---

- Give a regular expression for any lowercase string that has a repeated vowel (i.e. noon, peel, festoon, loop, etc).

- `[a-z]*(aa|ee|ii|oo|uu)[a-z]*`

## Exercises

---

- Give a regular expression for any lowercase string that has a repeated vowel (i.e. noon, peel, festoon, loop, etc).

- `[a-z]*(aa|ee|ii|oo|uu)[a-z]*`

- Give a regular expression for any string that contains both a lowercase letter and a number.

## Exercises

---

- Give a regular expression for any lowercase string that has a repeated vowel (i.e. noon, peel, festoon, loop, etc).

- `[a-z]*(aa|ee|ii|oo|uu)[a-z]*`

- Give a regular expression for any string that contains both a lowercase letter and a number.

- `(.*[0-9].*[a-z].*)|(. *[a-z].*[0-9].*)`

- `.*[a-z0-9]{2}.*`

# More Advanced Regular Expressions Syntax

# Limitations of Regular Expressions

---

Writing regular expressions is like writing a program.

- Need to know the syntax well.
- Can be easier to write than to read.
- Can be difficult to debug.

Regular expressions sometimes jokingly referred to as a “write only language”.

Regular expressions are terrible at certain types of problems. Examples:

- For parsing a hierarchical structure, such as JSON, use a parser, not a regex!
- Complex features (e.g. valid email address).
- Counting (same number of instances of a and b). (impossible)
- Complex properties (palindromes, balanced parentheses). (impossible)

## Email Address Regular Expression (a probably bad idea)

## The regular expression for email addresses

[illegible]



## Even More Regular Expression Syntax

operation	example	matches	does not match
built-in character classes	<code>\w+</code> <code>\d+</code>	fawef 231231	this person 423 people
character class negation	<code>[^a-z]+</code>	PEPPERS3982 17211!↑å	porch CLAmS
escape character	<code>cow\.com</code>	cow.com	cowscom

Suppose you want to match one of our special characters like `.` or `[` or `]`

- In these cases, you must “escape” the character using the backslash.
- You can think of the backslash as meaning “take this next character literally”.

[26/Jan/2014:10:47:58 -0800]

```
import re
pattern = r'\[(\d+)/(\d+)/(\d+):(\d+):(\d+):(\d+) (.+)\]'
day, month, year, hour, minute, second, time_zone = re.search(pattern, line).groups()
```

## Even More Regular Expression Features

operation	example	matches	does not match
beginning of line	<code>^ark</code>	ark two ark o ark	dark
end of line	<code>ark\$</code>	dark ark o ark	ark two
<b>lazy</b> version of zero or more <code>*?</code> (lazy means as few as possible)	<code>5.*?5</code>	5005 55	5005005

5.\*5 would match this!

A few additional common regex features are listed above.

- Won't discuss these in class, but might come up in discussion or hw.
- There are even more out there!

The official guide is good! <https://docs.python.org/3/howto/regex.html>

# Regular Expressions in Python (and Regex Groups)

## re.findall in Python

---

In Python, `re.findall(pattern, text)` will return a list of all matches.

```
text = "My Phone number is 145-6576-4295";  
pattern = r"1[0-9]{2}-[0-9]{4}-[0-9]{4}"  
m = re.findall(pattern, text)  
print(m)
```

```
[ '145-6576-4295' ]
```

## re.sub in Python

---

In Python, `re.sub(pattern, repl, text)` will return text with all instances of pattern replaced by repl.

```
text = '<div><td valign="top">Moo</td></div>'
pattern = r"<[^>]+>"
cleaned = re.sub(pattern, '', text)
print(cleaned)
```

```
'Moo'
```

# Raw Strings in Python

---

Note: When specifying a pattern, we strongly suggest using “raw strings”.

- A raw string is created using `r"""` or `r"` instead of just `"""` or `"`.
- The exact reason is a bit tedious.
  - Rough idea: Regular expressions and Python strings both use `\` as an escape character.
  - Using non-raw strings leads to uglier regular expressions.

Regular String	Raw string
<code>"ab*"</code>	<code>r"ab*"</code>
<code>"\\\\section"</code>	<code>r"\\section"</code>
<code>"\\w+\\s+\\1"</code>	<code>r"\\w+\\s+\\1"</code>

For more information see “The Backslash Plague” under <https://docs.python.org/3/howto/regex.html>.

# Regular Expression Groups

---

Earlier we used parentheses to specify the order of operations.

Parentheses have another meaning:

- Every set of parentheses specifies a so-called “group”.
- Regular expression matchers (e.g. `re.findall`) will return matches organized by groups. In Python, returned as tuples.

```
s = """Observations: 03:04:53 - Horse awakens.  
03:05:14 - Horse goes back to sleep."""  
pattern = "(\d\d):(\d\d):(\d\d) - (.*)"   
matches = re.findall(pattern, s)
```

```
[('03', '04', '53', 'Horse awakens.'),  
 ('03', '05', '14', 'Horse goes back to sleep.)]
```

# Summary

---

Today we saw many different string manipulation tools.

- There are many many more!
- With just this basic set of tools, you can do most of what you'll need.

basic python	re	pandas
	<code>re.findall</code>	<code>df.str.findall</code>
<code>str.replace</code>	<code>re.sub</code>	<code>df.str.replace</code>
<code>str.split</code>	<code>re.split</code>	<code>df.str.split</code>
<code>'ab' in str</code>	<code>re.search</code>	<code>df.str.contains</code>
<code>len(str)</code>		<code>df.str.len</code>
<code>str[1:4]</code>		<code>df.str[1:4]</code>



## Optional (but Handy) Regex Concepts

---

These regex features aren't going to be on an exam, but they are useful:

- **Lookaround:** match “good” if it’s not preceded by “not”: `(?<!not)good`
- **Backreferences:** match HTML tags of the same name: `<(\w+)>.*<\1>`
  - e.g. `<hhhh><div><td valign="top">Moo</td></div>`
- **Named groups:** match a vowel as a named group: `(?<vowel>[aeiou])`
- **Free Space:** Allow free space and comments in a pattern:

*# Match a 20th or 21st century date in yyyy-mm-dd format*

`(19|20)\d\d[- /.](0[1-9]|1[012])[- /.](0[1-9]|[12][0-9]|3[01])`

`(19|20)\d\d`                      *# year (group 1)*

`[- /.]`                              *# separator: -, /, .*

`(0[1-9]|1[012])`                *# month (group 2)*

`[- /.]`                              *# separator*

`(0[1-9]|[12][0-9]|3[01])`    *# day (group 3)*

## Real World Example

---

- Go to lec7 notebook