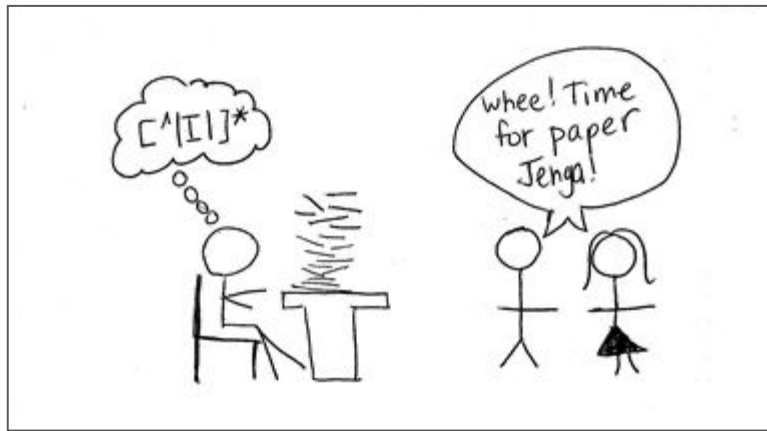




Data Science (COSE471) Spring 2021

# Regular Expressions, working with Text

Dept. of Computer Science and Engineering  
Korea University



\* This material is adapted from Berkeley CS 100 (ds100.org) and may be copyrighted by them.

# Goals For This Lecture

---

## Working With Text Data

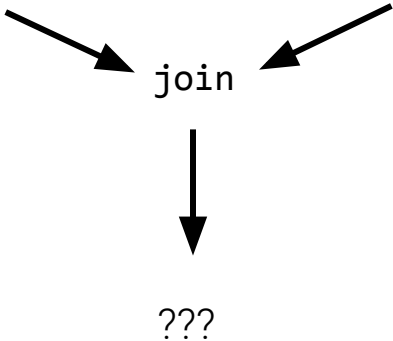
- Canonicalizing text data.
- Extracting data from text.
  - Using **split**.
  - Using **regular expressions**.

# String Canonicalization

# Goal 1: Joining Tables with Mismatched Labels

	County	State
0	De Witt County	IL
1	Lac qui Parle County	MN
2	Lewis and Clark County	MT
3	St John the Baptist Parish	LA

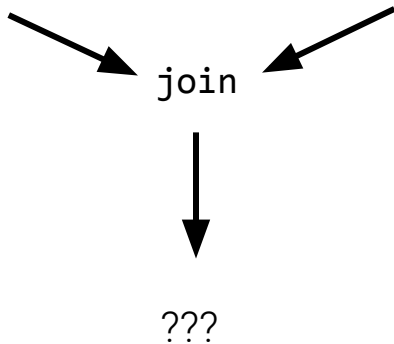
	County	Population
0	DeWitt	16798
1	Lac Qui Parle	8067
2	Lewis & Clark	55716
3	St. John the Baptist	43044



# A Joining Problem

	County	State
0	De Witt County	IL
1	Lac qui Parle County	MN
2	Lewis and Clark County	MT
3	St John the Baptist Parish	LA

	County	Population
0	DeWitt	16798
1	Lac Qui Parle	8067
2	Lewis & Clark	55716
3	St. John the Baptist	43044



To join our tables we'll need to **canonicalize** the county names.

- Canonicalize: Convert data that has more than one possible presentation into a standard form.

# Canonicalizing County Names

## County

De Witt County

Lac qui Parle County

Lewis and Clark County

St John the Baptist Parish

```
def canonicalize_county(county_name):  
    return (  
        county_name  
        .lower()           # lower case  
        .replace(' ', '')  # remove spaces  
        .replace('&', 'and') # replace &  
        .replace('.', '')  # remove dot  
        .replace('county', '') # remove county  
        .replace('parish', '') # remove parish  
    )
```

## County

dewitt

lacquiparle

lewisandclark

stjohnthebaptist

## County

DeWitt

Lac Qui Parle

Lewis & Clark

St. John the Baptist

```
def canonicalize_county(county_name):  
    return (  
        county_name  
        .lower()           # lower case  
        .replace(' ', '')  # remove spaces  
        .replace('&', 'and') # replace &  
        .replace('.', '')  # remove dot  
        .replace('county', '') # remove county  
        .replace('parish', '') # remove parish  
    )
```

# Canonicalization

---

## Canonicalization:

- Replace each string with a unique representation.
- Feels very “hacky”, but messy problems often have messy solutions.

Can be done slightly better but not by much →

- Code is very brittle! Requires maintenance.

Tools used:

```
def canonicalize_county(county_name):  
    return (  
        county_name  
        .lower()                # Lower case  
        .replace(' ', '')       # remove spaces  
        .replace('&', 'and')     # replace &  
        .replace('.', '')       # remove dot  
        .replace('county', '')  # remove county  
        .replace('parish', '')  # remove parish  
    )
```

Replacement	<code>str.replace('&amp;', 'and')</code>
Deletion	<code>str.replace(' ', '')</code>
Transformation	<code>str.lower()</code>

Extracting From Text Using Split



## Goal 2: Extracting Date Information

---

Suppose we want to extract times and dates from web server logs that look like the following:

```
169.237.46.168 - - [26/Jan/2014:10:47:58  
-0800] "GET /stat141/Winter04/ HTTP/1.1" 200  
2585 "http://anson.ucdavis.edu/courses/"
```

## Goal 2: Extracting Date Information

---

Suppose we want to extract times and dates from web server logs that look like the following:

```
169.237.46.168 - - [26/Jan/2014:10:47:58  
-0800] "GET /stat141/Winter04/ HTTP/1.1" 200  
2585 "http://anson.ucdavis.edu/courses/"
```

There are existing libraries that do most of the work for us, but let's try to do it from scratch.

- Will do together, just a little bit at a time.

## Extracting Date Information

---

```
169.237.46.168 - - [26/Jan/2014:10:47:58  
-0800] "GET /stat141/Winter04/ HTTP/1.1" 200  
2585 "http://anson.ucdavis.edu/courses/"
```

One possible solution:

```
day, month, rest = line.split(' ')[1].split(' ')[0].split('/')
```

```
year, hour, minute, seconds = rest.split(' ')[0].split(':')  
time_zone = rest.split(' ')[1]
```

## Extracting Date Information

---

```
169.237.46.168 - - [26/Jan/2014:10:47:58  
-0800] "GET /stat141/Winter04/ HTTP/1.1" 200  
2585 "http://anson.ucdavis.edu/courses/"
```

One possible solution:

```
day, month, rest = line.split(' ')[1].split(':')[0].split('/')  
year, hour, minute, seconds = rest.split(' ')[0].split(':')  
time_zone = rest.split(' ')[1]
```

What if webserver  
changes log formats,  $\Rightarrow$  *This solution breaks!! (brittle)*  
or has a bug?

# Regular Expression Basics

## Extracting Date Information

---

Earlier we saw that we can hack together code that uses `split` to extract info:

```
day, month, rest = line.split(' ')[1].split(' ')[0].split('/')

```

```
year, hour, minute, seconds = rest.split(' ')[0].split(':')
time_zone = rest.split(' ')[1]

```

An alternate approach is to use a so-called “regular expression”:

- Implementation provided in the `re` library built into Python.
- We’ll spend some time today working up to expressions like shown below.

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

```

# Regular Expressions

---

A *formal language* is a set of strings, typically described implicitly.

- Example: "The set of all strings of length < 10 that contain 'horse'"

A *regular language* is a formal language that can be described by a *regular expression* (which we will define soon).

Example: `[0-9]{3}-[0-9]{2}-[0-9]{4}` ← The language of SSNs is described by this regular expression.

3 of any digit, then a dash, then 2 of any digit, then a dash, then 4 of any digit.

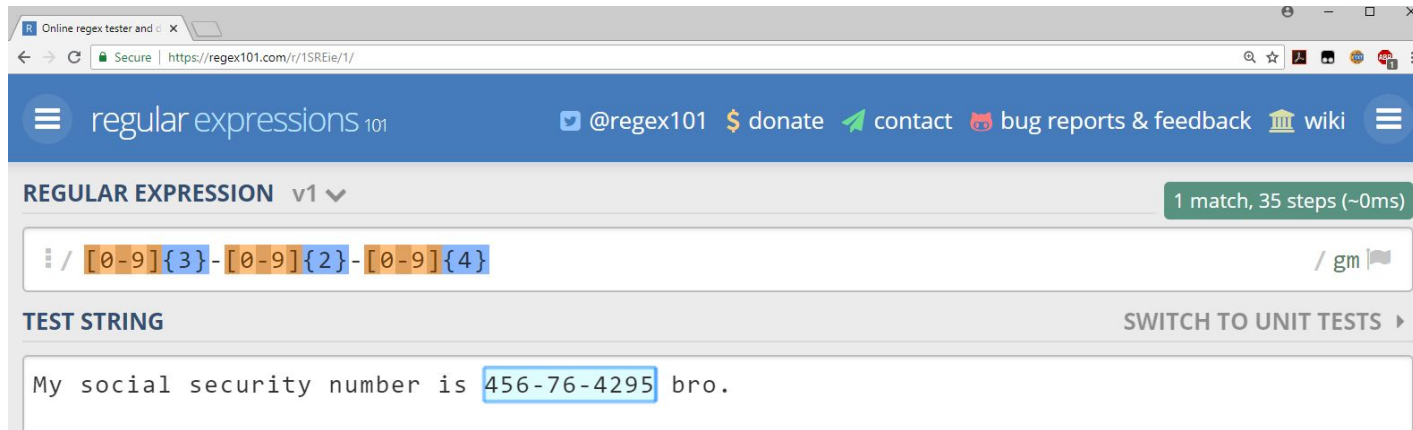
```
text = "My social security number is 123-45-6789.";
pattern = r"[0-9]{3}-[0-9]{2}-[0-9]{4}"
re.findall(pattern, text)
```

## [Regex101.com](https://regex101.com) (or the online tutorial [regexone.com](https://regexone.com))

There are a ton of nice resources out there to experiment with regular expressions (e.g. [regex101.com](https://regex101.com), [regexone.com](https://regexone.com), [sublime text](https://sublime-text.com), python, etc).

I recommend trying out [regex101.com](https://regex101.com), which provides a visually appealing and easy to use platform for experimenting with regular expressions.

- Example: <https://regex101.com/r/1SREie/1>





# 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	$AA \mid BAAB$	AA BAAB	every other string
closure (zero or more)	2	$AB^*A$	AA ABBBBBBA	AB ABABA
parenthesis	1	$A(A \mid B)AAB$	AAAAB ABAAB	every other string
		$(AB)^*A$	A ABABABABA	AA ABBA

## Puzzle: Use [regex101.com](https://regex101.com) to test! Or [tinyurl.com/reg913z](https://tinyurl.com/reg913z)

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

Give a regular expression that matches moon, moooon, etc. Your expression should match any **even** number of os except zero (i.e. don't match mn).

# Regular Expression moo(oo)\*n: <https://tinyurl.com/reg913m>

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

Give a regex that matches muun, muuuun, moon, moooon, etc. Your expression should match any even number of us or os except zero (i.e. don't match mn).

# 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`

`mooon`

# 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`  
`mooon`

`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`  
`oooon`

In regexes `|` comes last.

# Expanded Regular Expressions Syntax

# 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 com)</code>	horse@pizza.com horse@pizza.food.com	frank_99@yahoo.com hug@cs

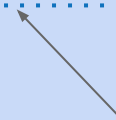
# Expanded Regex Puzzle: <https://tinyurl.com/reg913w>

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	jhn
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

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

# Expanded Regex Syntax Puzzle: <https://tinyurl.com/reg913v>

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	jhn
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



Select "Unit Tests" then  
click "Run Tests" to test  
your regex.

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

# 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.

"Some people, when confronted with a problem, think 'I know, I'll use regular expressions.' Now they have two problems." - Jamie Zawinski ([Source](#))

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)

The regular expression for email addresses (for the Perl programming language):

From: <http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html>

# 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”.

# Regular Expressions Puzzle: [tinyurl.com/reg913a](http://tinyurl.com/reg913a)

---

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

Create a regular expression that matches the red portion below.

```
169.237.46.168 - - [26/Jan/2014:10:47:58 -0800] "GET
/stat141/Winter04/ HTTP/1.1" 200 2585
"http://anson.ucdavis.edu/courses/"
```



# Quiz

Regex101 link: <https://tinyurl.com/reg913>

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

Create a regular expression that matches anything inside of angle brackets <>, but none of the string outside of angle brackets.

- Example: `<div><td valign="top">Moo</td></div>`
- Moo should not match because it is not between < and >.
- Note: This is equivalent to the problem of matching HTML tags.

# 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>	<code>5.*?5</code>	5005 55	5005005

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>

5.\*5 would match this!

# 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 social security number is 456-76-4295 bro, or  
actually maybe it's 456-67-4295.";  
pattern = r"[0-9]{3}-[0-9]{2}-[0-9]{4}"  
m = re.findall(pattern, text)  
print(m)
```

```
['456-76-4295', '456-67-4295']
```

## 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`, [regex101.com](https://www.regex101.com)) 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.)]
```

## Regex Puzzle

Fill in the regex below so that after code executes, day is "26", month is "Jan", and year is "2014".

```
pattern = "YOUR REGEX HERE"  
matches = re.findall(pattern, log[0])  
day, month, year = matches[0]
```

```
169.237.46.168 - - [26/Jan/2014:10:47:58 -0800] "GET
```

```
log[0]: /stat141/Winter04/ HTTP/1.1" 200 2585
```

```
"http://anson.ucdavis.edu/courses/"
```



## Extracting Date Information

---

With a little more work, we can do something similar and extract day, month, year, hour, minutes, seconds, and time zone all in one regular expression.

- Derivation is left as an exercise for you

```
import re
pattern = r'\[(\d+)/(\w+)/(\d+):(\d+):(\d+):(\d+) (.+)\]'
day, month, year, hour, minute, second, time_zone = re.findall(pattern, first)[0]
year, month, day, hour, minute, second, time_zone
```

You will also see code that uses `re.search` instead of `re.findall`.

- Beyond the scope of our lecture today.

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

# 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>