6 Regular Expressions

Learning Outcomes

- Understand Python string manipulation, `pandas Series` methods
- Parse and create regex, with a reference table
- Use vocabulary (closure, metacharacters, groups, etc.) to describe regex metacharacters

6.1 Why Work with Text?

Last lecture, we learned of the difference between quantitative and qualitative variable types. The latter includes string data — the primary focus of lecture 6. In this note, we'll discuss the necessary tools to manipulate text: Python string manipulation and regular expressions.

There are two main reasons for working with text.

1. Canonicalization: Convert data that has multiple formats into a standard form.
    - By manipulating text, we can join tables with mismatched string labels.
2. Extract information into a new feature.
    - For example, we can extract date and time features from text.

6.2 Python String Methods

First, we'll introduce a few methods useful for string manipulation. The following table includes a number of string operations supported by Python and `pandas`. The Python functions operate on a single string, while their equivalent in `pandas` are **vectorized** — they operate on a `Series` of string data.

Operation	**Python**	`Pandas` (`Series`)
Transformation	• `s.lower()` • `s.upper()`	• `ser.str.lower()` • `ser.str.upper()`
Replacement + Deletion	• `s.replace(_)`	• `ser.str.replace(_)`
Split	• `s.split(_)`	• `ser.str.split(_)`
Substring	• `s[1:4]`	• `ser.str[1:4]`
Membership	• `'_' in s`	• `ser.str.contains(_)`

Operation	Python	`Pandas` (`Series`)
Length	• `len(s)`	• `ser.str.len()`

We'll discuss the differences between Python string functions and `pandas Series` methods in the following section on canonicalization.

6.2.1 Canonicalization

Assume we want to merge the given tables.

▶ Code

```
display(county_and_state), display(county_and_pop);
```

	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	LS

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

Can we convert these columns into one standard, canonical form to merge the two tables?

6.2.1.1 Canonicalization with Python String Manipulation

The following function uses Python string manipulation to convert a single county name into canonical form. It does so by eliminating whitespace, punctuation, and unnecessary text.

```
def canonicalize_county(county_name):
    return (
        county_name
            .lower()
            .replace(' ', '')
            .replace('&', 'and')
            .replace('.', '')
            .replace('county', '')
            .replace('parish', '')
    )
```

```
canonicalize_county("St. John the Baptist")
```

```
'stjohnthebaptist'
```

6.2.1.2 Canonicalization with Pandas Series Methods

Alternatively, we can use `pandas Series` methods to create this standardized column. To do so, we must call the `.str` attribute of our `Series` object prior to calling any methods, like `.lower` and `.replace`. Notice how these method names match their equivalent built-in Python string functions.

Chaining multiple `Series` methods in this manner eliminates the need to use the `map` function (as this code is vectorized).

```
def canonicalize_county_series(county_series):
    return (
        county_series
            .str.lower()
            .str.replace(' ', '')
            .str.replace('&', 'and')
            .str.replace('.', '')
            .str.replace('county', '')
            .str.replace('parish', '')
    )

county_and_pop['clean_county_pandas'] = canonicalize_county_series(county_and_pop['County
county_and_state['clean_county_pandas'] = canonicalize_county_series(county_and_state['Co
display(county_and_pop), display(county_and_state);
```

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

	County	State	clean_county_pandas
0	De Witt County	IL	dewitt
1	Lac qui Parle County	MN	lacquiparle
2	Lewis and Clark County	MT	lewisandclark
3	St John the Baptist Parish	LS	stjohnthebaptist

6.2.2 Extraction

Extraction explores the idea of obtaining useful information from text data. This will be particularly important in model building, which we'll study in a few weeks.

Say we want to read some data from a `.txt` file.

```
with open('data/log.txt', 'r') as f:
    log_lines = f.readlines()

log_lines
```

```
['169.237.46.168 - - [26/Jan/2014:10:47:58 -0800] "GET /stat141/Winter04/ HTTP/1.1" 200
2585 "http://anson.ucdavis.edu/courses/"\n',
 '193.205.203.3 - - [2/Feb/2005:17:23:6 -0800] "GET /stat141/Notes/dim.html HTTP/1.0" 404
302 "http://eeyore.ucdavis.edu/stat141/Notes/session.html"\n',
 '169.237.46.240 - "" [3/Feb/2006:10:18:37 -0800] "GET
/stat141/homework/Solutions/hw1Sol.pdf HTTP/1.1"\n']
```

Suppose we want to extract the day, month, year, hour, minutes, seconds, and time zone. Unfortunately, these items are not in a fixed position from the beginning of the string, so slicing by some fixed offset won't work.

Instead, we can use some clever thinking. Notice how the relevant information is contained within a set of brackets, further separated by `/` and `:`. We can hone in on this region of text, and split the data on these characters. Python's built-in `.split` function makes this easy.

```
first = log_lines[0] # Only considering the first row of data

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

```
('26', 'Jan', '2014', '10', '47', '58', '-0800')
```

There are two problems with this code:

1. Python's built-in functions limit us to extract data one record at a time,
2. The code is quite verbose.
    - This is a larger issue that is trickier to solve

In the next section, we'll introduce regular expressions - a tool that solves problem 2.

6.3 RegEx Basics

A **regular expression ("RegEx")** is a sequence of characters that specifies a search pattern. They are written to extract specific information from text. Regular expressions are essentially part of a smaller programming language embedded in Python, made available through the `re` module. As such, they have a stand-alone syntax and methods for various capabilities.

Regular expressions are useful in many applications beyond data science. For example, Social Security Numbers (SSNs) are often validated with regular expressions.

```
r"[0-9]{3}-[0-9]{2}-[0-9]{4}" # Regular Expression Syntax

# 3 of any digit, then a dash,
# then 2 of any digit, then a dash,
# then 4 of any digit
```

```
'[0-9]{3}-[0-9]{2}-[0-9]{4}'
```

There are a ton of resources to learn and experiment with regular expressions. A few are provided below:

- Official Regex Guide
- Data 100 Reference Sheet
- Regex101.com
    - Be sure to check Python under the category on the left.

6.3.1 Basics RegEx Syntax

There are four basic operations with regular expressions.

Operation	Order	Syntax Example	Matches	Doesn't Match
`Or`: `|`	4	AA|BAAB	AA BAAB	every other string
`Concatenation`	3	AABAAB	AABAAB	every other string
`Closure`: `*` (zero or more)	2	AB*A	AA ABBBBBBA	AB ABABA
`Group`: `()` (parenthesis)	1	A(A|B)AAB	AAAAB ABAAB	every other string
		(AB)*A	A ABABABABA	AA ABBA

Notice how these metacharacter operations are ordered. Rather than being literal characters, these **metacharacters** manipulate adjacent characters. `()` takes precedence, followed by `*`, and finally `|`. This allows us to differentiate between very different regex commands like `AB*` and `(AB)*`. The former reads "`A` then zero or more copies of `B`", while the latter specifies "zero or more copies of `AB`".

6.3.1.1 Examples

Question 1: Give a regular expression that matches `moon`, `moooon`, etc. Your expression should match any even number of `o`s except zero (i.e. don't match `mn`).

▶ **Answer1**

Question 2: Using only basic operations, formulate a regex that matches `muun`, `muuuun`, `moon`, `moooon`, etc. Your expression should match any even number of `u`s or `o`s except zero (i.e. don't match `mn`).

▶ **Answer2**

6.4 RegEx Expanded

Provided below are more complex regular expression functions.

Operation	Syntax Example	Matches	Doesn't Match
`Any Character: .` (except newline)	.U.U.U.	CUMULUS JUGULUM	SUCCUBUS TUMULTUOUS
`Character Class: []` (match one character in `[]`)	[A-Za-z][a-z]*	word Capitalized	camelCase 4illegal
`Repeated "a" Times: {a}`	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
`At Least One: +`	jo+hn	john jooooooohn	jhn jjohn
`Zero or One: ?`	joh?n	jon john	any other string

A character class matches a single character in its class. These characters can be hardcoded —— in the case of `[aeiou]` —— or shorthand can be specified to mean a range of characters. Examples include:

1. `[A-Z]` : Any capitalized letter
2. `[a-z]` : Any lowercase letter
3. `[0-9]` : Any single digit
4. `[A-Za-z]` : Any capitalized or lowercase letter
5. `[A-Za-z0-9]` : Any capitalized or lowercase letter or single digit

6.4.0.1 Examples

Let's analyze a few examples of complex regular expressions.

Matches	Does Not Match
1. `.*SPB.*`	
RASPBERRY SPBOO	SUBSPACE SUBSPECIES
2. `[0-9]{3}-[0-9]{2}-[0-9]{4}`	
231-41-5121 573-57-1821	231415121 57-3571821
3. `[a-z]+@([a-z]+\.)+(edu|com)`	
horse@pizza.com horse@pizza.food.com	frank_99@yahoo.com hug@cs

Explanations

1. `.*SPB.*` only matches strings that contain the substring `SPB`.
    - The `.*` metacharacter matches any amount of non-negative characters. Newlines do not count.
2. This regular expression matches 3 of any digit, then a dash, then 2 of any digit, then a dash, then 4 of any digit.
    - You'll recognize this as the familiar Social Security Number regular expression.
3. Matches any email with a `com` or `edu` domain, where all characters of the email are letters.
    - At least one `.` must precede the domain name. Including a backslash `\` before any metacharacter (in this case, the `.`) tells RegEx to match that character exactly.

6.5 Convenient RegEx

Here are a few more convenient regular expressions.

Operation	Syntax Example	Matches	Doesn't Match

`built in character class`	`\w+`
	`\d+`
	`\s+`
		Fawef_03
		231123
		`whitespace`
			this person
			423 people
			`non-whitespace`

`character class negation: [^]` (everything except the given characters)	[^a-z]+.	PEPPERS3982 17211!↑å	porch CLAmS
`escape character: \` (match the literal next character)	cow\.com	cow.com	cowscom

Operation	Syntax Example	Matches	Doesn't Match
`beginning of line: ^`	^ark	ark two ark o ark	dark
`end of line: $`	ark$	dark ark o ark	ark two
`lazy version of zero or more : *?`	5.*?5	5005 55	5005005

6.5.1 Greediness

In order to fully understand the last operation in the table, we have to discuss greediness. RegEx is greedy – it will look for the longest possible match in a string. To motivate this with an example, consider the pattern `<div>.*</div>`. In the sentence below, we would hope that the bolded portions would be matched:

"This is a ** example ** of greediness ** in ** regular expressions."

However, in reality, RegEx captures far more of the sentence. The way RegEx processes the text given that pattern is as follows:

1. "Look for the exact string <>"
2. Then, "look for any character 0 or more times"
3. Then, "look for the exact string "

The result would be all the characters starting from the leftmost and the rightmost (inclusive):

"This is a ** example of greediness in ** regular expressions."

We can fix this by making our pattern non-greedy, `<div>.*?</div>`. You can read up more in the documentation here.

6.5.2 Examples

Let's revisit our earlier problem of extracting date/time data from the given `.txt` files. Here is how the data looked.

```
log_lines[0]
```

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

Question: Give a regular expression that matches everything contained within and including the brackets - the day, month, year, hour, minutes, seconds, and time zone.

▶ **Answer**

6.6 Regex in Python and Pandas (RegEx Groups)

6.6.1 Canonicalization

6.6.1.1 Canonicalization with RegEx

Earlier in this note, we examined the process of canonicalization using `python` string manipulation and `pandas Series` methods. However, we mentioned this approach had a major flaw: our code was unnecessarily verbose. Equipped with our knowledge of regular expressions, let's fix this.

To do so, we need to understand a few functions in the `re` module. The first of these is the substitute function: `re.sub(pattern, rep1, text)`. It behaves similarly to `python`'s built-in `.replace` function, and returns text with all instances of `pattern` replaced by `rep1`.

The regular expression here removes text surrounded by `<>` (also known as HTML tags).

In order, the pattern matches … 1. a single `<` 2. any character that is not a `>` : div, td valign…, /td, /div 3. a single `>`

Any substring in `text` that fulfills all three conditions will be replaced by `''`.

```
import re

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

```
'Moo'
```

Notice the `r` preceding the regular expression pattern; this specifies the regular expression is a raw string. Raw strings do not recognize escape sequences (i.e., the Python newline metacharacter `\n`). This makes them useful for regular expressions, which often contain literal `\` characters.

In other words, don't forget to tag your RegEx with an `r`.

6.6.1.2 Canonicalization with `pandas`

We can also use regular expressions with `pandas Series` methods. This gives us the benefit of operating on an entire column of data as opposed to a single value. The code is simple: `ser.str.replace(pattern, repl, regex=True)`.

Consider the following `DataFrame html_data` with a single column.

▶ Code

```
html_data
```

	HTML
0	<td valign='top'>Moo</td>
1	Link
2	Bold text

```
pattern = r"<[^>]+>"
html_data['HTML'].str.replace(pattern, '', regex=True)
```

```
0          Moo
1         Link
2    Bold text
Name: HTML, dtype: object
```

6.6.2 Extraction

6.6.2.1 Extraction with RegEx

Just like with canonicalization, the `re` module provides capability to extract relevant text from a string: `re.findall(pattern, text)`. This function returns a list of all matches to `pattern`.

Using the familiar regular expression for Social Security Numbers:

```
text = "My social security number is 123-45-6789 bro, or maybe it's 321-45-6789."
pattern = r"[0-9]{3}-[0-9]{2}-[0-9]{4}"
re.findall(pattern, text)
```

```
['123-45-6789', '321-45-6789']
```

6.6.2.2 Extraction with `pandas`

`pandas` similarily provides extraction functionality on a `Series` of data: `ser.str.findall(pattern)`

Consider the following `DataFrame` `ssn_data`.

▶ Code

```
ssn_data
```

	SSN
0	987-65-4321
1	forty
2	123-45-6789 bro or 321-45-6789
3	999-99-9999

```
ssn_data["SSN"].str.findall(pattern)
```

```
0                  [987-65-4321]
1                             []
2    [123-45-6789, 321-45-6789]
3                  [999-99-9999]
Name: SSN, dtype: object
```

This function returns a list for every row containing the pattern matches in a given string.

As you may expect, there are similar `pandas` equivalents for other `re` functions as well. `Series.str.extract` takes in a pattern and returns a `DataFrame` of each capture group's first match in the string. In contrast, `Series.str.extractall` returns a multi-indexed `DataFrame` of all matches for each capture group. You can see the difference in the outputs below:

```
pattern_cg = r"([0-9]{3})-([0-9]{2})-([0-9]{4})"
ssn_data["SSN"].str.extract(pattern_cg)
```

	0	1	2
0	987	65	4321
1	NaN	NaN	NaN
2	123	45	6789
3	999	99	9999

```
ssn_data["SSN"].str.extractall(pattern_cg)
```

		0	1	2
	match			
0	**0**	987	65	4321
2	**0**	123	45	6789
	1	321	45	6789
3	**0**	999	99	9999

6.6.3 Regular Expression Capture Groups

Earlier we used parentheses `( )` to specify the highest order of operation in regular expressions. However, they have another meaning; parentheses are often used to represent **capture groups**. Capture groups are essentially, a set of smaller regular expressions that match multiple substrings in text data.

Let's take a look at an example.

6.6.3.1 Example 1

```
text = "Observations: 03:04:53 - Horse awakens. \
        03:05:14 - Horse goes back to sleep."
```

Say we want to capture all occurences of time data (hour, minute, and second) as *separate entities*.

```
pattern_1 = r"(\d\d):(\d\d):(\d\d)"
re.findall(pattern_1, text)
```

```
[('03', '04', '53'), ('03', '05', '14')]
```

Notice how the given pattern has 3 capture groups, each specified by the regular expression `(\d\d)`. We then use `re.findall` to return these capture groups, each as tuples containing 3 matches.

These regular expression capture groups can be different. We can use the `(\d{2})` shorthand to extract the same data.

```
pattern_2 = r"(\d\d):(\d\d):(\d{2})"
re.findall(pattern_2, text)
```

```
[('03', '04', '53'), ('03', '05', '14')]
```

6.6.3.2 Example 2

With the notion of capture groups, convince yourself how the following regular expression works.

```
first = log_lines[0]
first
```

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

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

```
26 Jan 2014 10 47 58 -0800
```

6.7 Limitations of Regular Expressions

Today, we explored the capabilities of regular expressions in data wrangling with text data. However, there are a few things to be wary of.

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 are terrible at certain types of problems:

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

Ultimately, the goal is not to memorize all regular expressions. Rather, the aim is to:

- Understand what RegEx is capable of.
- Parse and create RegEx, with a reference table
- Use vocabulary (metacharacter, escape character, groups, etc.) to describe regex metacharacters.
- Differentiate between (), [], {}
- Design your own character classes with `\d`, `\w`, `\s`, `[…–…]`, `^`, etc.
- Use `python` and `pandas` RegEx methods.