LECTURE 8

Text Wrangling and Regex Using string methods and regular expressions (regex) to work with textual data

Data Science@ Knowledge Stream

Sana Jabbar

1

Deal with a major challenge of EDA: cleaning text

• Operate on text data using str methods

Goals for this Lecture

Lecture 08

• Apply regex to identify patterns in strings

2

This Week ?



Question & Problem

Formulation

Prediction and Inference

**Data** 

**Acquisition**

**Exploratory** 

**Data Analysis**

**Reports, Decisions,**

**and Solutions**

**(Last weeks) (Today) (Next)**

Data Wrangling Intro to EDA

Working with Text Data Regular Expressions

Visualization

Code for plotting data

3

• Why work with text? • pandas str methods • Why regex?

• Regex basics

• Regex functions

AgendaLecture 08

4

• **Why work with text?** • pandas str methods • Why regex?

• Regex basics

Why Work With Text?

Lecture 08

• Regex functions

5

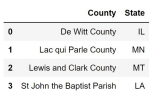
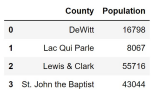
Why Work With Text? Two Common Goals

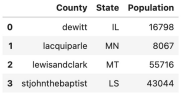
**1. Canonicalization**: Convert data that

has more than one possible

presentation into a standard form.

Ex Join tables with mismatched labels

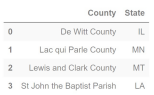
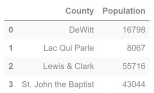


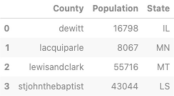
6

Why Work With Text? Two Common Goals **1. Canonicalization**: Convert data that

has more than one possible

presentation into a standard form.

Ex Join tables with mismatched labels **join?**

****

**2. Extract** information into a new feature.

Ex Extract dates and times from log files

169.237.46.168 - -

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

day, month, year = "26", "Jan", "2014" hour, minute, seconds = "10", "47", "58"

7

• Why work with text? • **Pandas str methods** • Why regex?

• Regex basics

Pandas str Methods

Lecture 08

• Regex functions

8

From String to str

In “base” Python, we have various string operations to work with text data. Recall:

transformation s.lower() s.upper()

replacement/ deletion

s.replace(…)

split s.split(…) substring s[1:4] membership 'ab' in s length len(s)

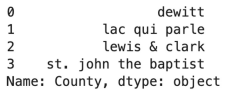
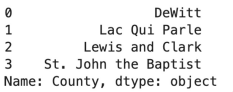
Problem: Python assumes we are working with one string at a time Need to loop over each entry – slow in large datasets!

9

str Methods

Fortunately, pandas offers a method of **vectorizing** text operations: the .str operator Series.str.string\_operation()

Apply the function string\_operation to *every* string contained in the Series

populations[“County”].str.lower() populations[“County”].str.replace('&', 'and') 

10

.str Methods

Most base Python string operations have a pandas str equivalent

**Operation Python (single string) pandas (Series of strings)**

transformation s.lower() s.upper()

ser.str.lower() ser.str.upper()

replacement/ deletion

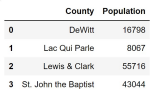
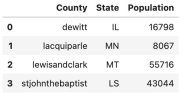
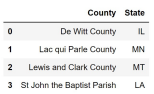
s.replace(…) ser.str.replace(…)

split s.split(…) ser.str.split(…) substring s[1:4] ser.str[1:4]

membership 'ab' in s ser.str.contains(…) length len(s) ser.str.len()

11

Demo 1: Canonicalization



**Example**

def canonicalize\_county(county\_series):

return (county\_series

.str.lower() # lowercase

.str.replace(' ', '') # remove space

.str.replace('&', 'and') # replace &

.str.replace('.', '') # remove dot

.str.replace('county', '')

.str.replace('parish', '') )

12

• Why work with text? • pandas str methods • **Why regex?**

• Regex basics

• Regex functions

Why regex?Lecture 08

13

Flexibility Matters!

169.237.46.168 - -

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

Formatting varies:

● Different # of characters before the date ● Different format for the day of the month ● Different # of characters after the date

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"

We don’t always know the exact format of our data in advance.

14

Flexibility Matters!

We made a big assumption in the previous example: knowing for certain what changes needed to be made to the text. “Eyeballing” the steps needed for canonicalization.

Consider our data extraction task from before – pulling out dates from log data:

169.237.46.168 - -

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

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"

15

Demo: Extracting Date Information

**Example**

169.237.46.168 - -

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

day, month, year = "26", "Jan", "2014" hour, minute, seconds = "10", "47", "58"

One possible solution:

pertinent = line.split("[")[1].split(']')[0] day, month, **rest** = pertinent.split('/') year, hour, minute, **rest** = **rest**.split(':') seconds, time\_zone = **rest**.split(' ')

16

String Extraction: An Alternate Approach

While we can hack together code that uses **replace/split**…

pertinent = line.split("[")[1].split(']')[0] day, month, rest = pertinent.split('/') year, hour, minute, rest = rest.split(':') seconds, time\_zone = rest.split(' ')

An alternate approach is to use a **regular expression**:

● Implementation provided in the Python **re** library and the pandas **str** accessor. ● Next, we’ll spend some time working up to expressions like this one:

import re

pattern = r'\[(\d+)\/(\w+)\/(\d+):(\d+):(\d+):(\d+) (.+)\]'

day, month, year, hour, minute, second, time\_zone = re.findall(pattern, line)[0]

Related: How would you extract all the "**moon** moo **moooooon** mon **moooon**" **moon**-like patterns in this string?

17

Seem impossible?

• Why work with text? • pandas str methods • Why regex?

• **Regex basics**

• Regex functions

Regex BasicsLecture 08

18

A Regular Expression Describes a Set of Strings Through *Patterns*

A regular expression (“regex”) is a sequence of characters that specifies a search *pattern*.

Example: **[0-9]{3}-[0-9]{2}-[0-9]{4}** 3 of any digit, then a dash,

then 2 of any digit, then a dash,

then 4 of any digit.

“Regex” pronunciation?

The language of Social Security Numbers is described by this regular expression.

Formal language, described implicitly

19

Goals for regex

The goal of today is NOT to memorize regex!

Instead:

1. Understand what regex is capable of.

2. Parse and create regex, **with a reference table**. high-level 3. Use vocabulary (metacharacter, escape character, groups, etc.)

to **describe regex** metacharacters.

4. **Differentiate** between (), [], {}

5. Design your own **character classes** with \d, \w, \s, […-…], ^, etc. 6. Use Python and pandas regex methods.

details;

hone with practice

20

Regex Basics

There are four basic operations in regex.

**Concatenation** – “look for consecutive characters” AABAAB matches AABAAB

**\*** – “zero or more”

AB\*A matches AA, ABA, ABBA, …

**|** – “or”

AA|BAAB matches AA *or* BAAB

**( )** – “consider a group”

(AB)\*A matches A, ABA, ABABA, … A(A|B)AAB matches AAAAB *or* ABAAB

\*, ( ), and | are called **metacharacters** – they represent an operation, rather than a literal text character

21

Resources for Practicing regex

There are many nice resources out there to experiment with regular expressions (e.g. regex101.com, regexone.com, Sublime Text).

I recommend trying out regex101.com, which provides a visually appealing and easy to use platform for experimenting with regular expressions.

● **Important:** choose the Python “flavor” in the left sidebar

22

In the Python docs: the RegEx HOWTO, *Not* The re Module Documentation The Python Regular Expression HOWTO: https://docs.python.org/3/howto/regex.html.

Make an empty notebook and play with the examples therein. Regex101 is phenomenal for learning basic regex *syntax,* but less so for learning the full functionality of regular expressions in programming (matching, splitting, search and replace, group management, …).

The examples can be pasted in directly:

23

In the Python docs: the RegEx HOWTO, *Not* The re Module Documentation The Python Regular Expression HOWTO: https://docs.python.org/3/howto/regex.html.

Make an empty notebook and play with the examples therein. Regex101 is phenomenal for learning basic regex *syntax,* but less so for learning the full functionality of regular expressions in programming (matching, splitting, search and replace, group management, …).

The examples can be pasted in directly:

24

Summary So Far

**Operation Order Example Matches Doesn’t match** (consecutive chars) 3 AABAAB AABAAB every other string

**concatenation**

**or,** | 4 AA|BAAB AA

BAAB every other string

\*

(zero or more) 2 AB\*A AA ABBBBBBA

A(A|B)AAB AAAAB

AB

ABABA

**group**

(parenthesis) 1

ABAAB every other string

(AB)\*A AABABABABAAA

ABBA

The regex order of operations. Grouping is evaluated first.

25

Regex Expanded

Six more regex operations.

**.** – “look for *any* character”

.U.U.U. matches CUMULUS, JUGULUM

**+** – “one or more”

AB+ matches AB, ABB, ABBB, …

**{x}** – “repeat exactly x times”

AB{2} matches ABB

**[ ]** – “define a character

class”

[A-Za-z] matches A, a, B, b…

**?** – “zero or one” ("optional")

AB? matches A, AB

**{x, y}** – “repeat between x and y times” AB{0,2} matches A, AB, ABB

(yes, it means these are the same: **\*** = **{0,}**, **+** = **{1,}**, and **?** = **{0,1}**)26

Character Classes

A character class describes a set of characters belonging to the class.

**[A-Z]** – any uppercase letter between A and Z **[0-9]** – any digit between 0 and 9

**[A-Za-z0-9]** – any letter, any digit

Regex built-in classes:

**\w** is equivalent to [A-Za-z0-9] **\d** is equivalent to [0-9] **\s** matches whitespace

Use **^** to negate a class = match any character *other* than what follows **[^A-Z]** – anything that is *not* an uppercase letter between A and Z

Equivalently, the capital versions of the regex built-in classes are negations: \W, \D, and \S

27

Summary So Far

**Operation Example Matches Doesn’t match**

(except newline) .U.U.U. CUMULUS

**any character**

JUGULUM

**character class** [A-Za-z][a-z]\* word Capitalized

SUCCUBUS TUMULTUOUS

camelCase 4illegal

**repeated exactly a times**: {a}

**repeated from a to**

j[aeiou]{3}hn jaoehn jooohn

jhn

jaeiouhn

**b times**: {a,b} j[ou]{1,2}hn john juohn

**at least one** jo+hn john joooooohn

jhn

jooohn

jhn

jjohn

28

Greediness

Regex is **greedy** – it will look for the *longest possible* match in a string

<div>.\*</div>

“This is a **<div>example</div>** of greediness <div>in</div> regular expressions.”

29

Greediness

Regex is **greedy** – it will look for the *longest possible* match in a string

<div>.\*</div>

In English:

● “Look for the exact string <div>"

● then, “look for any character 0 or more times”

● then, “look for the exact string </div>"

“This is a **<div>example</div>** of greediness <div>in</div> regular expressions.”

30

Greediness

Regex is **greedy** – it will look for the *longest possible* match in a string

<div>.\*</div>

In English:

● “Look for the exact string <div>"

● then, “look for any character 0 or more times”

● then, “look for the exact string </div>"

“This is a **<div>example</div>** of greediness <div>in</div> regular expressions.”

We can fix this by making the pattern **non-greedy:**

<div>.\*?</div>

31

Regex *Even More* Expanded The last set.

**\** – “read the next character literally” a\+b matches a+b

**^** – “match the beginning of a string” ^abc does not match “123 abc”

Be careful: ^ has different behavior

inside/outside of character classes!

**$** – “match the end of a string” abc$ does not match “abc 123”

32

Summary So Far

**Operation 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

**escape character** cow\.com cow.com cowscom

33

• Why work with text? • pandas str methods • Why regex?

• Regex basics

• **Regex functions**

Regex FunctionsLecture 08

34

Before We Begin: Raw Strings in Python

When specifying a pattern, we strongly suggest using **raw strings**. ● A raw string is created by prepending **r** to the string delimiters

pattern = r"[0-9]+"

(**r"...", r'...', r"""...""", r'''...'''**)

● The exact reason is a bit tedious. 

○ Rough idea: Regular expressions

and Python strings both use \ as a special character.

○ Using non-raw strings leads to uglier regular expressions.

For more information see “The Backslash Plague” under

https://docs.python.org/3/howto/regex.html#the-backslash-plague

35

Extraction

re.**findall**(**pattern**, text) Return a list of all matches to pattern.

text = "My social security number is 123-45- 6789 bro, or actually 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']

A **match** is a substring that

matches the provided regex.

36

Extraction

re.**findall**(pattern, text)

Return a list of all matches to pattern.

text = "My social security number is 123-45- 6789 bro, or actually 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']

ser.**str.findall**(**pattern**)

Returns a Series of lists

df["SSN"].str.findall(**pattern**)

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

37

Extraction with Capture Groups

Earlier we used parentheses to specify the **order of operations**.

Parenthesis can have another meaning:

● When using certain regex functions, ( ) specifies a **capture group**. ● Extract *only* the portion of the regex pattern inside the capture group

text = """I will meet you at 08:30:00 pm tomorrow""" **pattern =** "**(\d\d)**:**(\d\d)**:**(\d\d)**"

matches = re.findall(pattern, text)

matches

[('08', '30', '00')]

The capture groups each capture two digits.

38

Extraction with Capture Groups

ser.**str.extract**(**pattern**)

Returns a DataFrame of each capture group’s **first** match in the string

**pattern\_cg** = r"([0-9]{3})-([0-9]{2})-([0-9]{4})" df["SSN"].str.extract(**pattern\_cg**)

ser.**str.extractall**(**pattern**) Returns a multi-indexed DataFrame of **all** matches for each capture group

df["SSN"].str.extractall(**pattern\_cg**)

39

Substitution

re.sub(**pattern**, **repl**, text) Returns text with all instances of pattern replaced by repl.

text = '<div><tdvalign="top">**Moo**</td></div>' **pattern** = **r**"<[^>]+>"

re.sub(**pattern**, **''**, text) # returns Moo

**Moo**

How it works:

● **pattern** matches HTML tags ● Then, sub/replace HTML tags with **repl=''** (i.e., empty string)

40

Substitution

re.**sub**(pattern, repl, text)

Returns text with all instances of pattern replaced by repl.

text = '<div><td

valign="top">**Moo**</td></div>'

**pattern** = **r**"<[^>]+>"

re.sub(pattern, '', text) # returns Moo

Moo

How it works:

● **pattern** matches HTML tags ● Then, sub/replace HTML tags with **repl=''** (i.e., empty string)

ser.str.replace(**pattern**, **repl**,

regex=**True** )

Returns Series with all instances of pattern in Series ser replaced by **repl**.

df["Html"].str.replace(**pattern**, **''**) 

0 Moo

1 Link

2 Bold text

Name: Html, dtype: object

41

String Function Summary

**Base Python re pandas str**

s.lower() s.upper()

ser.str.lower() ser.str.upper()

s.replace(…) re.sub(…) ser.str.replace(…)

s.split(…) re.split(…) ser.str.split(…)

s[1:4] ser.str[1:4]

re.findall(…) ser.str.findall(…)

ser.str.extractall(…)

ser.str.extract(…)

'ab' in s re.search(…) ser.str.contains(…)

len(s) ser.str.len()

s.strip() ser.str.strip() �42 

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**”. A famous 1997 quote from Jamie Zawinski (co-creator of Firefox's predecessor)

*Some people, when confronted with a problem, think "I know, I'll use regular*

*expressions." Now they have two problems.*

Regular expressions are terrible at certain types of problems:

● For parsing a hierarchical structure, such as JSON, use the json.load() parser, not regex! ● Parsing real-world HTML/xml (lots of <div>...<tag>..</tag>..</div>): use html.parser. ● Complex features (e.g. valid email address).

● Counting (same number of instances of a and b). (impossible)

However, regular expressions are decent at **wrangling text data**.

43

Start Work on notebook 44