DS 100/200: Principles and Techniques of Data Science Date: Feb 14, 2020

Discussion #4 Solutions

Name:

Pandas: Grouping Multiple Columns

Throughout this section you'll be working with the babynames (left) and elections (right) datasets as shown below:

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134

	Year	Candidate	Party	Popular vote	Result	%
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122
1	1824	John Quincy Adams	Democratic-Republican	113142	win	42.789878
2	1828	Andrew Jackson	Democratic	642806	win	56.203927
3	1828	John Quincy Adams	National Republican	500897	loss	43.796073
4	1832	Andrew Jackson	Democratic	702735	win	54.574789

1. (a) Which of the following lines of code will output the following dataframe? Recall that the arguments to `pd.pivot_table` are as follows: `data` is the input dataframe, `index` includes the values we use as rows, `columns` are the columns of the pivot table, `values` are the values in the pivot table, and `aggfunc` is the aggregation function that we use to aggregate `values`.

Result	loss	win
Party		
Democratic	43.697060	51.441864
Democratic-Republican	57.210122	42.789878
National Union	NaN	54.951512
Republican	42.047791	52.366967
Whig	35.258650	50.180255

◯ A. `pd.pivot_table(data=elections, index='Party', columns='Result', values='%', aggfunc=np.mean)`

◯ B. `elections.groupby(['Party', 'Result'])['%'].mean()`

◯ C. `pd.pivot_table(data=elections, index='Result', columns='Party', values='%', aggfunc=np.mean)`

◯ D. `elections.groupby('%')[['Party', 'Result']].mean()`

(b) `name_counts_since_1940 = babynames[babynames["Year"] >= 1940].groupby(["Name", "Year"]).sum()` generates the multi-indexed DataFrame below.

Name	Year	
Aadan	**2008**	7
	2009	6
	2014	5
Aaden	**2007**	20
	2008	135
	2009	158
	2010	62
	2011	39
	2012	38

We can index into multi-indexed DataFrames using `loc` with slightly different syntax. For example `name_counts_since_1940.loc[("Aahna", 2008):("Aaiden", 2014)]` yields the DataFrame below.

		Count
Name	**Year**	
Aahna	**2014**	7
Aaiden	**2009**	11
	2010	11
	2011	8
	2013	13
	2014	12

Using `name_counts_since_1940`, set `imani_2013_count` equal to the number of babies born with the name `'Imani'` in the year 2013. You may use either '.loc'. Make sure you're returning a value and not a Series or DataFrame.

`imani_2013_count =`

Pandas: String Operations and Table Joining

2. (a) Create a new DataFrame called `elections_with_first_name` with a new column `'First Name'` that is equal to the Candidate's first name. Hint: Use `.str.split`.

`elections_with_first_name =`

(b) Now create `elections_and_names` by joining `elections_with_first_name` with `name_counts_since_1940_numerical_index` (the modified version of `name_counts_since_1940` with the index reset) on both the first names of each person along and the year.

```
elections_and_names =
```

Regular Expressions

Here's a complete list of metacharacters:

```
. ^ $ * + ? { } [ ] \ | ( )
```

Some reminders on what each can do (this is not exhaustive):

- `"^"` matches the position at the beginning of string (unless used for negation `"[^]"`)
- `"$"` matches the position at the end of string character.
- `"?"` match preceding literal or sub-expression 0 or 1 times.
- `"+"` match preceding literal or sub-expression *one* or more times.
- `"*"` match preceding literal or sub-expression *zero* or more times
- `"."` match any character except new line.
- `"[ ]"` match any one of the characters inside, accepts a range, e.g., `"[a-c]"`.
- `"( )"` used to create a sub-expression
- `"\d"` match any *digit* character. `"\D"` is the complement.
- `"\w"` match any *word* character (letters, digits, underscore). `"\W"` is the complement.
- `"\s"` match any *whitespace* character including tabs and newlines. `\S` is the complement.
- `"*?"` Non-greedy version of *. Not fully discussed in class.
- `"\b"` match boundary between words. Not discussed in class.
- `"+?"` Non-greedy version of +. Not discussed in class.

Some useful `re` package functions:

- `re.split(pattern, string)` split the `string` at substrings that match the `pattern`. Returns a list.
- `re.sub(pattern, replace, string)` apply the `pattern` to `string` replacing matching substrings with `replace`. Returns a string.
- `re.findall(pattern, string)` Returns a list of all matches for the given `pattern` in the `string`.

Regular Expressions

3. For each pattern specify the starting and ending position of the first match in the string. The index starts at zero and we are using closed intervals (both endpoints are included).

	`abcdefg`	`abcs!`	`ab␣abc`	`abc,␣123`
`abc*`	$[0, 2]$	$[0, 2]$	$[0, 1]$	$[0, 2]$
`[^\s]+`	$[0, 6]$	$[0, 4]$	$[0, 1]$	$[0, 3]$
`ab.*c`	$[0, 2]$	$[0, 2]$	$[0, 5]$	$[0, 2]$
`[a-z1,9]+`	$[0, 6]$	$[0, 3]$	$[0, 1]$	$[0, 3]$

4. Given the text:

```
"<record>␣Joey␣Gonzalez␣<jegonzal@cs.berkeley.edu>␣Faculty␣</record>"
"<record>␣Manana␣Hakobyan␣<manana.hakobyan@berkeley.edu>␣TA␣</record>"
```

Which of the following matches exactly to the email addresses (including angle brackets)?

◯ A. `<.*@.*>` ◯ B. `<[^>]*@[^>]*>` ◯ C. `<.*@\w+\..*>`

Solution: Greediness matches too much in the first and third choices.

5. Write a regular expression that matches strings that contain exactly 5 vowels.

Solution:

```
^([^aeiouAEIOU]*[aeiouAEIOU]){5}[^aeiouAEIOU]*$
```

6. Given that `sometext` is a string, use `re.sub` to replace all clusters of non-vowel characters with a single period. For example `"a␣big␣moon,␣between␣us..."` would be changed to `"a.i.oo.e.ee.u."`.

Solution:

```
re.sub( r"[^aeoiuAEIOU]+", "." , sometext)
```

7. Given the following text in a variable `log`:

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

Fill in the regular expression in the variable `pattern` below so that after it executes, day is 26, month is Jan, and year is 2014.

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

Solution:

```
pattern = "\[(.+)\/(.+)\/([^:]+).*\]"
matches = re.findall(pattern, log)
day, month, year = matches[0]
```

Optional Regex Practice

8. Which strings contain a match for the following regular expression, `"1+1$"`? The character `"␣"` represents a single space.

◯ A. `What␣is␣1+1` ◯ B. `Make␣a␣wish␣at␣11:11` ◯ C. `111␣Ways␣to␣Succeed`

Solution: Recall that `1+` matches on *at least one* occurrence of the character `1`, and `$` marks the end of the string.

9. Write a regular expression that matches strings (including the empty string) that only contain lowercase letters and numbers.

Solution:

```
^[a-z0-9]*$
```

10. Given that `address` is a string, use `re.sub` to replace all vowels with a lowercase letter "o". For example `"123␣Orange␣Street"` would be changed to `"123␣orongo␣Stroot"`.

Solution:

```
re.sub(r"[aeiuAEIOU]",  "o", address)
```

11. Given `sometext = "I've␣got␣10␣eggs,␣20␣gooses,␣and␣30␣giants."`, use `re.findall` to extract all the items and quantities from the string. The result should look like `['10 eggs', '20 gooses', '30 giants']`. You may assume that a space separates quantity and type, and that each item ends in s.

Solution:

```
re.findall(r"\d\d\s\w+s", sometext)
```