03.10-Working-With-Strings

February 27, 2017

< Pivot Tables | Contents | Working with Time Series >

1 Vectorized String Operations

One strength of Python is its relative ease in handling and manipulating string data. Pandas builds on this and provides a comprehensive set of *vectorized string operations* that become an essential piece of the type of munging required when working with (read: cleaning up) real-world data. In this section, we'll walk through some of the Pandas string operations, and then take a look at using them to partially clean up a very messy dataset of recipes collected from the Internet.

1.1 Introducing Pandas String Operations

We saw in previous sections how tools like NumPy and Pandas generalize arithmetic operations so that we can easily and quickly perform the same operation on many array elements. For example:

```
In [1]: import numpy as np
        x = np.array([2, 3, 5, 7, 11, 13])
        x * 2

Out[1]: array([ 4,  6, 10, 14, 22, 26])
```

This *vectorization* of operations simplifies the syntax of operating on arrays of data: we no longer have to worry about the size or shape of the array, but just about what operation we want done. For arrays of strings, NumPy does not provide such simple access, and thus you're stuck using a more verbose loop syntax:

```
In [2]: data = ['peter', 'Paul', 'MARY', 'gUIDO']
        [s.capitalize() for s in data]

Out[2]: ['Peter', 'Paul', 'Mary', 'Guido']
```

This is perhaps sufficient to work with some data, but it will break if there are any missing values. For example:

```
In [3]: data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
        [s.capitalize() for s in data]


        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)

        <ipython-input-3-fc1d891ab539> in <module>()
          1 data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
    ----> 2 [s.capitalize() for s in data]


        <ipython-input-3-fc1d891ab539> in <listcomp>(.0)
          1 data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
    ----> 2 [s.capitalize() for s in data]


        AttributeError: 'NoneType' object has no attribute 'capitalize'
```

Pandas includes features to address both this need for vectorized string operations and for correctly handling missing data via the `str` attribute of Pandas Series and Index objects containing strings. So, for example, suppose we create a Pandas Series with this data:

```
In [4]: import pandas as pd
        names = pd.Series(data)
        names

Out[4]: 0    peter
        1     Paul
        2     None
        3     MARY
        4    gUIDO
        dtype: object
```

We can now call a single method that will capitalize all the entries, while skipping over any missing values:

```
In [5]: names.str.capitalize()

Out[5]: 0    Peter
        1     Paul
        2     None
        3     Mary
        4    Guido
        dtype: object
```

Using tab completion on this `str` attribute will list all the vectorized string methods available to Pandas.

1.2 Tables of Pandas String Methods

If you have a good understanding of string manipulation in Python, most of Pandas string syntax is intuitive enough that it's probably sufficient to just list a table of available methods; we will start with that here, before diving deeper into a few of the subtleties. The examples in this section use the following series of names:

```
In [6]: monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',
                           'Eric Idle', 'Terry Jones', 'Michael Palin'])
```

1.2.1 Methods similar to Python string methods

Nearly all Python's built-in string methods are mirrored by a Pandas vectorized string method. Here is a list of Pandas `str` methods that mirror Python string methods:

`len()`	`lower()`	`translate()`	`islower()`
`ljust()`	`upper()`	`startswith()`	`isupper()`
`rjust()`	`find()`	`endswith()`	`isnumeric()`
`center()`	`rfind()`	`isalnum()`	`isdecimal()`
`zfill()`	`index()`	`isalpha()`	`split()`
`strip()`	`rindex()`	`isdigit()`	`rsplit()`
`rstrip()`	`capitalize()`	`isspace()`	`partition()`
`lstrip()`	`swapcase()`	`istitle()`	`rpartition()`

Notice that these have various return values. Some, like `lower()`, return a series of strings:

```
In [7]: monte.str.lower()

Out[7]: 0    graham chapman
        1       john cleese
        2     terry gilliam
        3         eric idle
        4       terry jones
        5     michael palin
        dtype: object
```

But some others return numbers:

```
In [8]: monte.str.len()

Out[8]: 0    14
        1    11
        2    13
        3     9
        4    11
        5    13
        dtype: int64
```

Or Boolean values:

```
In [9]: monte.str.startswith('T')

Out[9]: 0    False
        1    False
        2     True
        3    False
        4     True
        5    False
        dtype: bool
```

Still others return lists or other compound values for each element:

```
In [10]: monte.str.split()

Out[10]: 0    [Graham, Chapman]
         1       [John, Cleese]
         2     [Terry, Gilliam]
         3         [Eric, Idle]
         4       [Terry, Jones]
         5     [Michael, Palin]
         dtype: object
```

We'll see further manipulations of this kind of series-of-lists object as we continue our discussion.

1.2.2 Methods using regular expressions

In addition, there are several methods that accept regular expressions to examine the content of each string element, and follow some of the API conventions of Python's built-in `re` module:

Method	Description
`match()`	Call `re.match()` on each element, returning a boolean.
`extract()`	Call `re.match()` on each element, returning matched groups as strings.
`findall()`	Call `re.findall()` on each element
`replace()`	Replace occurrences of pattern with some other string
`contains()`	Call `re.search()` on each element, returning a boolean
`count()`	Count occurrences of pattern
`split()`	Equivalent to `str.split()`, but accepts regexps
`rsplit()`	Equivalent to `str.rsplit()`, but accepts regexps

With these, you can do a wide range of interesting operations. For example, we can extract the first name from each by asking for a contiguous group of characters at the beginning of each element:

```
In [11]: monte.str.extract('([A-Za-z]+)', expand=False)

Out[11]: 0     Graham
         1       John
```

```
2       Terry
3        Eric
4       Terry
5     Michael
dtype: object
```

Or we can do something more complicated, like finding all names that start and end with a consonant, making use of the start-of-string (^) and end-of-string ($) regular expression characters:

```
In [12]: monte.str.findall(r'^[^AEIOU].*[^aeiou]$')

Out[12]: 0    [Graham Chapman]
         1                  []
         2     [Terry Gilliam]
         3                  []
         4       [Terry Jones]
         5     [Michael Palin]
         dtype: object
```

The ability to concisely apply regular expressions across `Series` or `Dataframe` entries opens up many possibilities for analysis and cleaning of data.

1.2.3 Miscellaneous methods

Finally, there are some miscellaneous methods that enable other convenient operations:

Method	Description
`get()`	Index each element
`slice()`	Slice each element
`slice_replace()`	Replace slice in each element with passed value
`cat()`	Concatenate strings
`repeat()`	Repeat values
`normalize()`	Return Unicode form of string
`pad()`	Add whitespace to left, right, or both sides of strings
`wrap()`	Split long strings into lines with length less than a given width
`join()`	Join strings in each element of the Series with passed separator
`get_dummies()`	extract dummy variables as a dataframe

Vectorized item access and slicing The `get()` and `slice()` operations, in particular, enable vectorized element access from each array. For example, we can get a slice of the first three characters of each array using `str.slice(0, 3)`. Note that this behavior is also available through Python's normal indexing syntax–for example, `df.str.slice(0, 3)` is equivalent to `df.str[0:3]`:

```
In [13]: monte.str[0:3]

Out[13]: 0    Gra
```

```
1    Joh
2    Ter
3    Eri
4    Ter
5    Mic
dtype: object
```

Indexing via `df.str.get(i)` and `df.str[i]` is likewise similar.

These `get()` and `slice()` methods also let you access elements of arrays returned by `split()`. For example, to extract the last name of each entry, we can combine `split()` and `get()`:

```
In [14]: monte.str.split().str.get(-1)

Out[14]: 0    Chapman
         1     Cleese
         2    Gilliam
         3       Idle
         4      Jones
         5      Palin
         dtype: object
```

Indicator variables Another method that requires a bit of extra explanation is the `get_dummies()` method. This is useful when your data has a column containing some sort of coded indicator. For example, we might have a dataset that contains information in the form of codes, such as A="born in America," B="born in the United Kingdom," C="likes cheese," D="likes spam":

```
In [15]: full_monte = pd.DataFrame({'name': monte,
                                    'info': ['B|C|D', 'B|D', 'A|C',
                                             'B|D', 'B|C', 'B|C|D']})
         full_monte

Out[15]:     info            name
         0  B|C|D  Graham Chapman
         1    B|D     John Cleese
         2    A|C   Terry Gilliam
         3    B|D       Eric Idle
         4    B|C     Terry Jones
         5  B|C|D   Michael Palin
```

The `get_dummies()` routine lets you quickly split-out these indicator variables into a `DataFrame`:

```
In [16]: full_monte['info'].str.get_dummies('|')

Out[16]:    A  B  C  D
         0  0  1  1  1
         1  0  1  0  1
```

```
2  1  0  1  0
3  0  1  0  1
4  0  1  1  0
5  0  1  1  1
```

With these operations as building blocks, you can construct an endless range of string processing procedures when cleaning your data.

We won't dive further into these methods here, but I encourage you to read through "Working with Text Data" in the Pandas online documentation, or to refer to the resources listed in Further Resources.

1.3 Example: Recipe Database

These vectorized string operations become most useful in the process of cleaning up messy, real-world data. Here I'll walk through an example of that, using an open recipe database compiled from various sources on the Web. Our goal will be to parse the recipe data into ingredient lists, so we can quickly find a recipe based on some ingredients we have on hand.

The scripts used to compile this can be found at https://github.com/fictivekin/openrecipes, and the link to the current version of the database is found there as well.

As of Spring 2016, this database is about 30 MB, and can be downloaded and unzipped with these commands:

```
In [17]: # !curl -O http://openrecipes.s3.amazonaws.com/recipeitems-latest.json.gz
         # !gunzip recipeitems-latest.json.gz
```

The database is in JSON format, so we will try `pd.read_json` to read it:

```
In [18]: try:
             recipes = pd.read_json('recipeitems-latest.json')
         except ValueError as e:
             print("ValueError:", e)

ValueError: Trailing data
```

Oops! We get a `ValueError` mentioning that there is "trailing data." Searching for the text of this error on the Internet, it seems that it's due to using a file in which *each line* is itself a valid JSON, but the full file is not. Let's check if this interpretation is true:

```
In [19]: with open('recipeitems-latest.json') as f:
             line = f.readline()
         pd.read_json(line).shape

Out[19]: (2, 12)
```

Yes, apparently each line is a valid JSON, so we'll need to string them together. One way we can do this is to actually construct a string representation containing all these JSON entries, and then load the whole thing with `pd.read_json`:

```
In [20]: # read the entire file into a Python array
         with open('recipeitems-latest.json', 'r') as f:
             # Extract each line
             data = (line.strip() for line in f)
             # Reformat so each line is the element of a list
             data_json = "[{0}]".format(','.join(data))
         # read the result as a JSON
         recipes = pd.read_json(data_json)
```

```
In [21]: recipes.shape
```

```
Out[21]: (173278, 17)
```

We see there are nearly 200,000 recipes, and 17 columns. Let's take a look at one row to see what we have:

```
In [22]: recipes.iloc[0]
```

```
Out[22]: _id                                {'$oid': '5160756b96cc62079cc2db15'}
         cookTime                                                           PT30M
         creator                                                              NaN
         dateModified                                                         NaN
         datePublished                                                 2013-03-11
         description           Late Saturday afternoon, after Marlboro Man ha...
         image                 http://static.thepioneerwoman.com/cooking/file...
         ingredients           Biscuits\n3 cups All-purpose Flour\n2 Tablespo...
         name                                      Drop Biscuits and Sausage Gravy
         prepTime                                                           PT10M
         recipeCategory                                                       NaN
         recipeInstructions                                                   NaN
         recipeYield                                                           12
         source                                                   thepioneerwoman
         totalTime                                                            NaN
         ts                                             {'$date': 1365276011104}
         url                   http://thepioneerwoman.com/cooking/2013/03/dro...
         Name: 0, dtype: object
```

There is a lot of information there, but much of it is in a very messy form, as is typical of data scraped from the Web. In particular, the ingredient list is in string format; we're going to have to carefully extract the information we're interested in. Let's start by taking a closer look at the ingredients:

```
In [23]: recipes.ingredients.str.len().describe()
```

```
Out[23]: count    173278.000000
         mean        244.617926
         std         146.705285
         min           0.000000
         25%         147.000000
```

```
50%           221.000000
75%           314.000000
max          9067.000000
Name: ingredients, dtype: float64
```

The ingredient lists average 250 characters long, with a minimum of 0 and a maximum of nearly 10,000 characters!

Just out of curiousity, let's see which recipe has the longest ingredient list:

```
In [24]: recipes.name[np.argmax(recipes.ingredients.str.len())]

Out[24]: 'Carrot Pineapple Spice &amp; Brownie Layer Cake with Whipped Cream &amp;
```

That certainly looks like an involved recipe.

We can do other aggregate explorations; for example, let's see how many of the recipes are for breakfast food:

```
In [25]: recipes.description.str.contains('[Bb]reakfast').sum()

Out[25]: 3524
```

Or how many of the recipes list cinnamon as an ingredient:

```
In [26]: recipes.ingredients.str.contains('[Cc]innamon').sum()

Out[26]: 10526
```

We could even look to see whether any recipes misspell the ingredient as "cinamon":

```
In [27]: recipes.ingredients.str.contains('[Cc]inamon').sum()

Out[27]: 11
```

This is the type of essential data exploration that is possible with Pandas string tools. It is data munging like this that Python really excels at.

1.3.1 A simple recipe recommender

Let's go a bit further, and start working on a simple recipe recommendation system: given a list of ingredients, find a recipe that uses all those ingredients. While conceptually straightforward, the task is complicated by the heterogeneity of the data: there is no easy operation, for example, to extract a clean list of ingredients from each row. So we will cheat a bit: we'll start with a list of common ingredients, and simply search to see whether they are in each recipe's ingredient list. For simplicity, let's just stick with herbs and spices for the time being:

```
In [28]: spice_list = ['salt', 'pepper', 'oregano', 'sage', 'parsley',
                       'rosemary', 'tarragon', 'thyme', 'paprika', 'cumin']
```

We can then build a Boolean `DataFrame` consisting of True and False values, indicating whether this ingredient appears in the list:

```
In [29]: import re
         spice_df = pd.DataFrame(dict((spice, recipes.ingredients.str.contains(spic
                                       for spice in spice_list))
         spice_df.head()

Out[29]:    cumin oregano paprika parsley pepper rosemary   sage   salt tarragon  t
         0  False   False   False   False  False    False   True  False    False  F
         1  False   False   False   False  False    False  False  False    False  F
         2   True   False   False   False   True    False  False   True    False  F
         3  False   False   False   False  False    False  False  False    False  F
         4  False   False   False   False  False    False  False  False    False  F
```

Now, as an example, let's say we'd like to find a recipe that uses parsley, paprika, and tarragon. We can compute this very quickly using the `query()` method of `DataFrame`s, discussed in High-Performance Pandas: `eval()` and `query()`:

```
In [30]: selection = spice_df.query('parsley & paprika & tarragon')
         len(selection)

Out[30]: 10
```

We find only 10 recipes with this combination; let's use the index returned by this selection to discover the names of the recipes that have this combination:

```
In [31]: recipes.name[selection.index]

Out[31]: 2069      All cremat with a Little Gem, dandelion and wa...
         74964                         Lobster with Thermidor butter
         93768      Burton's Southern Fried Chicken with White Gravy
         113926                     Mijo's Slow Cooker Shredded Beef
         137686                     Asparagus Soup with Poached Eggs
         140530                                 Fried Oyster Po'boys
         158475                Lamb shank tagine with herb tabbouleh
         158486                 Southern fried chicken in buttermilk
         163175            Fried Chicken Sliders with Pickles + Slaw
         165243                        Bar Tartine Cauliflower Salad
         Name: name, dtype: object
```

Now that we have narrowed down our recipe selection by a factor of almost 20,000, we are in a position to make a more informed decision about what we'd like to cook for dinner.

1.3.2 Going further with recipes

Hopefully this example has given you a bit of a flavor (ba-dum!) for the types of data cleaning operations that are efficiently enabled by Pandas string methods. Of course, building a very robust recipe recommendation system would require a *lot* more work! Extracting full ingredient lists from each recipe would be an important piece of the task; unfortunately, the wide variety of formats used makes this a relatively time-consuming process. This points to the truism that in data science, cleaning and munging of real-world data often comprises the majority of the work, and Pandas provides the tools that can help you do this efficiently.

< Pivot Tables | Contents | Working with Time Series >