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 are an important part of the type of munging required when working with (read: cleaning up) real-world data. In this chapter, 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.

Introducing Pandas String Operations

We saw in previous chapters 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, so this approach requires putting in extra checks:

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

Out[3]:
```
['Peter', 'Paul', None, 'Mary', 'Guido']
```

This kind of manual approach is not only verbose and inconvenient, it can be error-prone.

Pandas includes features to address both this need for vectorized string operations and the need for correctly handling missing data via the `str` attribute of Pandas `Series` and `Index` objects containing strings. So, for example, if we create a Pandas `Series` with this data we can directly call the `str.capitalize` method, which has missing value handling built in:

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

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

Tables of Pandas String Methods

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

In [5]:
```
monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',
                   'Eric Idle', 'Terry Jones', 'Michael Palin'])
```

Methods Similar to Python String Methods

Nearly all of 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 [6]:
```
monte.str.lower()
```

Out[6]:
```
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 [7]:
```
monte.str.len()
```

Out[7]:
```
0    14
1    11
2    13
3     9
4    11
5    13
dtype: int64
```

Or Boolean values:

In [8]:
```
monte.str.startswith('T')
```

Out[8]:
```
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 [9]:
```
monte.str.split()
```

Out[9]:
```
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.

Methods Using Regular Expressions

In addition, there are several methods that accept regular expressions (regexps) 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`	Calls `re.match` on each element, returning a Boolean.
`extract`	Calls `re.match` on each element, returning matched groups as strings.
`findall`	Calls `re.findall` on each element
`replace`	Replaces occurrences of pattern with some other string
`contains`	Calls `re.search` on each element, returning a boolean
`count`	Counts occurrences of pattern
`split`	Equivalent to `str.split`, but accepts regexps
`rsplit`	Equivalent to `str.rsplit`, but accepts regexps

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

In [10]:
```
monte.str.extract('([A-Za-z]+)', expand=False)
```

Out[10]:
```
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 [11]:
```
monte.str.findall(r'^[^AEIOU].*[^aeiou]$')
```

Out[11]:
```
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.

Miscellaneous Methods

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

Method	Description
`get`	Indexes each element
`slice`	Slices each element
`slice_replace`	Replaces slice in each element with the passed value
`cat`	Concatenates strings
`repeat`	Repeats values
`normalize`	Returns Unicode form of strings
`pad`	Adds whitespace to left, right, or both sides of strings
`wrap`	Splits long strings into lines with length less than a given width
`join`	Joins strings in each element of the `Series` with the passed separator
`get_dummies`	Extracts 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 [12]:
```
monte.str[0:3]
```

Out[12]:
```
0    Gra
1    Joh
2    Ter
3    Eri
4    Ter
5    Mic
dtype: object
```

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

These indexing 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` with `str` indexing:

In [13]:
```
monte.str.split().str[-1]
```

Out[13]:
```
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 [14]:
```
full_monte = pd.DataFrame({'name': monte,
                           'info': ['B|C|D', 'B|D', 'A|C',
                                    'B|D', 'B|C', 'B|C|D']})
full_monte
```

Out[14]:

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

The `get_dummies` routine lets us split out these indicator variables into a `DataFrame`:

In [15]:
```
full_monte['info'].str.get_dummies('|')
```

Out[15]:

	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" (https://pandas.pydata.org/pandas-docs/stable/user_guide/text.html) in the Pandas online documentation, or to refer to the resources listed in Further Resources (03.13-Further-Resources.ipynb).

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 (https://github.com/fictivekin/openrecipes), and the link to the most recent version of the database is found there as well.

This database is about 30 MB, and can be downloaded and unzipped with these commands:

In [16]:
```
# repo = "https://raw.githubusercontent.com/jakevdp/open-recipe-data/master"
# !cd data && curl -O {repo}/recipeitems.json.gz
# !gunzip data/recipeitems.json.gz
```

The database is in JSON format, so we will use `pd.read_json` to read it (`lines=True` is required for this dataset because each line of the file is a JSON entry):

In [17]:
```
recipes = pd.read_json('data/recipeitems.json', lines=True)
recipes.shape
```

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

We see there are nearly 175,000 recipes, and 17 columns. Let's take a look at one row to see what we have:

In [18]:
```
recipes.iloc[0]
```

Out[18]:
```
_id                                   {'$oid': '5160756b96cc62079cc2db15'}
name                                       Drop Biscuits and Sausage Gravy
ingredients           Biscuits\n3 cups All-purpose Flour\n2 Tablespo...
url                   http://thepioneerwoman.com/cooking/2013/03/dro... (htt
p://thepioneerwoman.com/cooking/2013/03/dro...)
image                 http://static.thepioneerwoman.com/cooking/file... (htt
p://static.thepioneerwoman.com/cooking/file...)
ts                                            {'$date': 1365276011104}
cookTime                                                         PT30M
source                                                 thepioneerwoman
recipeYield                                                         12
datePublished                                               2013-03-11
prepTime                                                         PT10M
description           Late Saturday afternoon, after Marlboro Man ha...
totalTime                                                          NaN
creator                                                            NaN
recipeCategory                                                     NaN
dateModified                                                       NaN
recipeInstructions                                                 NaN
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 [19]:
```
recipes.ingredients.str.len().describe()
```

Out[19]:
```
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 curiosity, let's see which recipe has the longest ingredient list:

In [20]:
```
recipes.name[np.argmax(recipes.ingredients.str.len())]
```

Out[20]:
```
'Carrot Pineapple Spice &amp; Brownie Layer Cake with Whipped Cream &amp; Cre
am Cheese Frosting and Marzipan Carrots'
```

We can do other aggregate explorations; for example, we can see how many of the recipes are for breakfast foods (using regular expression syntax to match both lowercase and capital letters):

In [21]:
```
recipes.description.str.contains('[Bb]reakfast').sum()
```

Out[21]: 3524

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

In [22]:
```
recipes.ingredients.str.contains('[Cc]innamon').sum()
```

Out[22]: 10526

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

In [23]:
```
recipes.ingredients.str.contains('[Cc]inamon').sum()
```

Out[23]: 11

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

A Simple Recipe Recommender

Let's go a bit further, and start working on a simple recipe recommendation system: given a list of ingredients, we want to find any recipes that use 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 [24]:
```
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 each ingredient appears in the list:

In [25]:
```
import re
spice_df = pd.DataFrame({
    spice: recipes.ingredients.str.contains(spice, re.IGNORECASE)
    for spice in spice_list})
spice_df.head()
```

Out[25]:

	salt	pepper	oregano	sage	parsley	rosemary	tarragon	thyme	paprika	cumin
0	False	False	False	True	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False	False	False
2	True	True	False	False	False	False	False	False	False	True
3	False	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False	False

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 further in [High-Performance Pandas: `eval()` and `query()` (03.12-Performance-Eval-and-Query.ipynb)](03.12-Performance-Eval-and-Query.ipynb):

In [26]:
```
selection = spice_df.query('parsley & paprika & tarragon')
len(selection)
```

Out[26]:
```
10
```

We find only 10 recipes with this combination. Let's use the index returned by this selection to discover the names of those recipes:

In [27]:
```
recipes.name[selection.index]
```

Out[27]:
```
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 from 175,000 to 10, we are in a position to make a more informed decision about what we'd like to cook for dinner.

Going Further with Recipes

Hopefully this example has given you a bit of a flavor (heh) of the types of data cleaning operations that are efficiently enabled by Pandas string methods. Of course, building a 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.