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" on page 215.

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 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 curiosity, 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;amp; Brownie Layer Cake with Whipped Cream
&amp;amp; Cream Cheese Frosting and Marzipan Carrots'
```

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[33]: recipes.description.str.contains('[Bb]reakfast').sum()

Out[33]: 3524
```

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

```
In[34]: recipes.ingredients.str.contains('[Cc]innamon').sum()

Out[34]: 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.

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(spice, re.IGNORECASE))
                                                          for spice in spice_list))
spice_df.head()
```

```
Out[29]:
   cumin oregano paprika parsley pepper rosemary   sage   salt tarragon  thyme
0  False   False   False   False  False    False   True  False    False  False
1  False   False   False   False  False    False  False  False    False  False
2   True   False   False   False   True    False  False   True    False  False
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 `DataFrames`, discussed in "High-Performance Pandas: eval() and query()" on page 208:

```
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.

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.

Working with Time Series

Pandas was developed in the context of financial modeling, so as you might expect, it contains a fairly extensive set of tools for working with dates, times, and time-indexed data. Date and time data comes in a few flavors, which we will discuss here:

- *Time stamps* reference particular moments in time (e.g., July 4th, 2015, at 7:00 a.m.).
- *Time intervals* and *periods* reference a length of time between a particular beginning and end point—for example, the year 2015. Periods usually reference a special case of time intervals in which each interval is of uniform length and does not overlap (e.g., 24 hour-long periods constituting days).
- *Time deltas* or *durations* reference an exact length of time (e.g., a duration of 22.56 seconds).

In this section, we will introduce how to work with each of these types of date/time data in Pandas. This short section is by no means a complete guide to the time series tools available in Python or Pandas, but instead is intended as a broad overview of how you as a user should approach working with time series. We will start with a brief discussion of tools for dealing with dates and times in Python, before moving more specifically to a discussion of the tools provided by Pandas. After listing some resources that go into more depth, we will review some short examples of working with time series data in Pandas.

Dates and Times in Python

The Python world has a number of available representations of dates, times, deltas, and timespans. While the time series tools provided by Pandas tend to be the most useful for data science applications, it is helpful to see their relationship to other packages used in Python.