

Subject with Code	Exploratory Data Analysis BDS613B
Sem	VI

Module 3

Data Manipulation with Pandas II

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.

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.

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'])
```

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:

<code>len()</code>	<code>lower()</code>	<code>translate()</code>	<code>islower()</code>
<code>ljust()</code>	<code>upper()</code>	<code>startswith()</code>	<code>isupper()</code>
<code>rjust()</code>	<code>find()</code>	<code>endswith()</code>	<code>isnumeric()</code>
<code>center()</code>	<code>rfind()</code>	<code>isalnum()</code>	<code>isdecimal()</code>
<code>zfill()</code>	<code>index()</code>	<code>isalpha()</code>	<code>split()</code>
<code>strip()</code>	<code>rindex()</code>	<code>isdigit()</code>	<code>rsplit()</code>

```
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.

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

<code>match()</code>	Call <code>re.match()</code> on each element, returning a boolean.
----------------------	--

<code>extract()</code>	Call <code>re.match()</code> on each element, returning matched groups as strings.
------------------------	--

<code>findall()</code>	Call <code>re.findall()</code> on each element
------------------------	--

<code>replace()</code>	Replace occurrences of pattern with some other string
------------------------	---

<code>contains()</code>	Call <code>re.search()</code> on each element, returning a boolean
-------------------------	--

<code>count()</code>	Count occurrences of pattern
----------------------	------------------------------

<code>split()</code>	Equivalent to <code>str.split()</code> , but accepts regexps
----------------------	--

<code>rsplit()</code>	Equivalent to <code>str.rsplit()</code> , 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.

Miscellaneous methods

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

Method	Description
<code>get()</code>	Index each element

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

	info	name
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.

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 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 & Brownie Layer Cake with Whipped Cream & Cream Chees  
e 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 [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.

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]:

[illegible]

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()`:

```
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 7:00am).
- *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 comprising 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.

Native Python dates and times: `datetime` and `dateutil`

Python's basic objects for working with dates and times reside in the built-in `datetime` module. Along with the third-party `dateutil` module, you can use it to quickly perform a host of useful functionalities on dates and times. For example, you can manually build a date using the `datetime` type:

```
In [1]:
```

```
from datetime import datetime
datetime(year=2015, month=7, day=4)
Out[1]:
```

```
datetime.datetime(2015, 7, 4, 0, 0)
```

Or, using the `dateutil` module, you can parse dates from a variety of string formats:

```
In [2]:
```

```
from dateutil import parser
date = parser.parse("4th of July, 2015")
date
Out[2]:
```

```
datetime.datetime(2015, 7, 4, 0, 0)
```

Once you have a `datetime` object, you can do things like printing the day of the week:

```
In [3]:
```

```
date.strftime('%A')
Out[3]:
```

```
'Saturday'
```

In the final line, we've used one of the standard string format codes for printing dates ("`%A`"), which you can read about in the `strftime` section of Python's `datetime` documentation. Documentation of other useful date utilities can be found in `dateutil`'s online documentation. A related package to be aware of is `pytz`, which contains tools for working with the most migraine-inducing piece of time series data: time zones.

The power of `datetime` and `dateutil` lie in their flexibility and easy syntax: you can use these objects and their built-in methods to easily perform nearly any operation you might be interested in. Where they break down is when you wish to work with large arrays of dates and times: just as lists of Python numerical variables are suboptimal compared to NumPy-style typed numerical arrays, lists of Python `datetime` objects are suboptimal compared to typed arrays of encoded dates.

Typed arrays of times: NumPy's `datetime64`

The weaknesses of Python's `datetime` format inspired the NumPy team to add a set of native time series data type to NumPy. The `datetime64` dtype encodes dates as 64-bit

integers, and thus allows arrays of dates to be represented very compactly. The `datetime64` requires a very specific input format:

```
In [4]:
```

```
import numpy as np
date = np.array('2015-07-04', dtype=np.datetime64)
date
```

```
Out[4]:
```

```
array(datetime.date(2015, 7, 4), dtype='datetime64[D]')
```

Once we have this date formatted, however, we can quickly do vectorized operations on it:

```
In [5]:
```

```
date + np.arange(12)
```

```
Out[5]:
```

```
array(['2015-07-04', '2015-07-05', '2015-07-06', '2015-07-07',
      '2015-07-08', '2015-07-09', '2015-07-10', '2015-07-11',
      '2015-07-12', '2015-07-13', '2015-07-14', '2015-07-15'], dtype='datetime64[D]')
)
```

Because of the uniform type in NumPy `datetime64` arrays, this type of operation can be accomplished much more quickly than if we were working directly with Python's `datetime` objects, especially as arrays get large (we introduced this type of vectorization in [Computation on NumPy Arrays: Universal Functions](#)).

One detail of the `datetime64` and `timedelta64` objects is that they are built on a *fundamental time unit*. Because the `datetime64` object is limited to 64-bit precision, the range of encodable times is 2^{64} times this fundamental unit. In other words, `datetime64` imposes a trade-off between *time resolution* and *maximum time span*.

For example, if you want a time resolution of one nanosecond, you only have enough information to encode a range of 2^{64} nanoseconds, or just under 600 years. NumPy will infer the desired unit from the input; for example, here is a day-based datetime:

```
In [6]:
```

```
np.datetime64('2015-07-04')
```

```
Out[6]:
```

```
numpy.datetime64('2015-07-04')
```

Here is a minute-based datetime:

```
In [7]:
```

```
np.datetime64('2015-07-04 12:00')
```

```
Out[7]:
```

```
numpy.datetime64('2015-07-04T12:00')
```

Notice that the time zone is automatically set to the local time on the computer executing the code. You can force any desired fundamental unit using one of many format codes; for example, here we'll force a nanosecond-based time:

```
In [8]:
```

```
np.datetime64('2015-07-04 12:59:59.50', 'ns')
```

```
Out[8]:
```

```
numpy.datetime64('2015-07-04T12:59:59.500000000')
```

The following table, drawn from the NumPy datetime64 documentation, lists the available format codes along with the relative and absolute timespans that they can encode:

Code	Meaning	Time span (relative)	Time span (absolute)
Y	Year	$\pm 9.2\text{e}18$ years	[9.2e18 BC, 9.2e18 AD]
M	Month	$\pm 7.6\text{e}17$ years	[7.6e17 BC, 7.6e17 AD]
W	Week	$\pm 1.7\text{e}17$ years	[1.7e17 BC, 1.7e17 AD]
D	Day	$\pm 2.5\text{e}16$ years	[2.5e16 BC, 2.5e16 AD]
h	Hour	$\pm 1.0\text{e}15$ years	[1.0e15 BC, 1.0e15 AD]
m	Minute	$\pm 1.7\text{e}13$ years	[1.7e13 BC, 1.7e13 AD]
s	Second	$\pm 2.9\text{e}12$ years	[2.9e9 BC, 2.9e9 AD]
ms	Millisecond	$\pm 2.9\text{e}9$ years	[2.9e6 BC, 2.9e6 AD]
us	Microsecond	$\pm 2.9\text{e}6$ years	[290301 BC, 294241 AD]
ns	Nanosecond	± 292 years	[1678 AD, 2262 AD]

Code	Meaning	Time span (relative)	Time span (absolute)
ps	Picosecond	± 106 days	[1969 AD, 1970 AD]
fs	Femtosecond	± 2.6 hours	[1969 AD, 1970 AD]
as	Attosecond	± 9.2 seconds	[1969 AD, 1970 AD]

For the types of data we see in the real world, a useful default is `datetime64[ns]`, as it can encode a useful range of modern dates with a suitably fine precision.

Finally, we will note that while the `datetime64` data type addresses some of the deficiencies of the built-in Python `datetime` type, it lacks many of the convenient methods and functions provided by `datetime` and especially `dateutil`. More information can be found in NumPy's `datetime64` documentation.

Dates and times in pandas: best of both worlds

Pandas builds upon all the tools just discussed to provide a `Timestamp` object, which combines the ease-of-use of `datetime` and `dateutil` with the efficient storage and vectorized interface of `numpy.datetime64`. From a group of these `Timestamp` objects, Pandas can construct a `DatetimeIndex` that can be used to index data in a `Series` or `DataFrame`; we'll see many examples of this below.

For example, we can use Pandas tools to repeat the demonstration from above. We can parse a flexibly formatted string date, and use format codes to output the day of the week:

```
In [9]:
```

```
import pandas as pd
date = pd.to_datetime("4th of July, 2015")
date
```

```
Out[9]:
```

```
Timestamp('2015-07-04 00:00:00')
```

```
In [10]:
```

```
date.strftime('%A')
```

```
Out[10]:
```

```
'Saturday'
```

Additionally, we can do NumPy-style vectorized operations directly on this same object:

```
In [11]:
date + pd.to_timedelta(np.arange(12), 'D')
Out[11]:
DatetimeIndex(['2015-07-04', '2015-07-05', '2015-07-06', '2015-07-07',
               '2015-07-08', '2015-07-09', '2015-07-10', '2015-07-11',
               '2015-07-12', '2015-07-13', '2015-07-14', '2015-07-15'],
              dtype='datetime64[ns]', freq=None)
```

In the next section, we will take a closer look at manipulating time series data with the tools provided by Pandas.

Pandas Time Series: Indexing by Time

Where the Pandas time series tools really become useful is when you begin to *index data by timestamps*. For example, we can construct a `Series` object that has time indexed data:

```
In [12]:
index = pd.DatetimeIndex(['2014-07-04', '2014-08-04',
                          '2015-07-04', '2015-08-04'])
data = pd.Series([0, 1, 2, 3], index=index)
data
Out[12]:
2014-07-04    0
2014-08-04    1
2015-07-04    2
2015-08-04    3
dtype: int64
```

Now that we have this data in a `Series`, we can make use of any of the `Series` indexing patterns we discussed in previous sections, passing values that can be coerced into dates:

```
In [13]:
data['2014-07-04':'2015-07-04']
Out[13]:
2014-07-04    0
2014-08-04    1
2015-07-04    2
dtype: int64
```

There are additional special date-only indexing operations, such as passing a year to obtain a slice of all data from that year:

```
In [14]:
data['2015']
Out[14]:
```

```
2015-07-04    2
2015-08-04    3
dtype: int64
```

Later, we will see additional examples of the convenience of dates-as-indices. But first, a closer look at the available time series data structures.

Pandas Time Series Data Structures

This section will introduce the fundamental Pandas data structures for working with time series data:

- For *time stamps*, Pandas provides the `Timestamp` type. As mentioned before, it is essentially a replacement for Python's native `datetime`, but is based on the more efficient `numpy.datetime64` data type. The associated Index structure is `DatetimeIndex`.
- For *time Periods*, Pandas provides the `Period` type. This encodes a fixed-frequency interval based on `numpy.datetime64`. The associated index structure is `PeriodIndex`.
- For *time deltas* or *durations*, Pandas provides the `Timedelta` type. `Timedelta` is a more efficient replacement for Python's native `datetime.timedelta` type, and is based on `numpy.timedelta64`. The associated index structure is `TimedeltaIndex`.

The most fundamental of these date/time objects are the `Timestamp` and `DatetimeIndex` objects. While these class objects can be invoked directly, it is more common to use the `pd.to_datetime()` function, which can parse a wide variety of formats. Passing a single date to `pd.to_datetime()` yields a `Timestamp`; passing a series of dates by default yields a `DatetimeIndex`:

```
In [15]:
dates = pd.to_datetime([datetime(2015, 7, 3), '4th of July, 2015',
                        '2015-Jul-6', '07-07-2015', '20150708'])
dates
Out[15]:
DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
               '2015-07-08'],
              dtype='datetime64[ns]', freq=None)
```

Any `DatetimeIndex` can be converted to a `PeriodIndex` with the `to_period()` function with the addition of a frequency code; here we'll use `'D'` to indicate daily frequency:

```
In [16]:
```

```
dates.to_period('D')
```

```
Out[16]:
```

```
PeriodIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',  
            '2015-07-08'],  
            dtype='int64', freq='D')
```

A `TimedeltaIndex` is created, for example, when a date is subtracted from another:

```
In [17]:
```

```
dates - dates[0]
```

```
Out[17]:
```

```
TimedeltaIndex(['0 days', '1 days', '3 days', '4 days', '5 days'], dtype='timedelta64  
[ns]', freq=None)
```

Regular sequences: `pd.date_range()`

To make the creation of regular date sequences more convenient, Pandas offers a few functions for this purpose: `pd.date_range()` for timestamps, `pd.period_range()` for periods, and `pd.timedelta_range()` for time deltas. We've seen that Python's `range()` and NumPy's `np.arange()` turn a startpoint, endpoint, and optional stepsize into a sequence. Similarly, `pd.date_range()` accepts a start date, an end date, and an optional frequency code to create a regular sequence of dates. By default, the frequency is one day:

```
In [18]:
```

```
pd.date_range('2015-07-03', '2015-07-10')
```

```
Out[18]:
```

```
DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-05', '2015-07-06',  
            '2015-07-07', '2015-07-08', '2015-07-09', '2015-07-10'],  
            dtype='datetime64[ns]', freq='D')
```

Alternatively, the date range can be specified not with a start and endpoint, but with a startpoint and a number of periods:

```
In [19]:
```

```
pd.date_range('2015-07-03', periods=8)
```

```
Out[19]:
```

```
DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-05', '2015-07-06',
```

```
'2015-07-07', '2015-07-08', '2015-07-09', '2015-07-10'],  
dtype='datetime64[ns]', freq='D')
```

The spacing can be modified by altering the `freq` argument, which defaults to `D`. For example, here we will construct a range of hourly timestamps:

```
In [20]:
```

```
pd.date_range('2015-07-03', periods=8, freq='H')
```

```
Out[20]:
```

```
DatetimeIndex(['2015-07-03 00:00:00', '2015-07-03 01:00:00',  
               '2015-07-03 02:00:00', '2015-07-03 03:00:00',  
               '2015-07-03 04:00:00', '2015-07-03 05:00:00',  
               '2015-07-03 06:00:00', '2015-07-03 07:00:00'],  
              dtype='datetime64[ns]', freq='H')
```

To create regular sequences of `Period` or `Timedelta` values, the very similar `pd.period_range()` and `pd.timedelta_range()` functions are useful. Here are some monthly periods:

```
In [21]:
```

```
pd.period_range('2015-07', periods=8, freq='M')
```

```
Out[21]:
```

```
PeriodIndex(['2015-07', '2015-08', '2015-09', '2015-10', '2015-11', '2015-12',  
            '2016-01', '2016-02'],  
           dtype='int64', freq='M')
```

And a sequence of durations increasing by an hour:

```
In [22]:
```

```
pd.timedelta_range(0, periods=10, freq='H')
```

```
Out[22]:
```

```
TimedeltaIndex(['00:00:00', '01:00:00', '02:00:00', '03:00:00', '04:00:00',  
               '05:00:00', '06:00:00', '07:00:00', '08:00:00', '09:00:00'],  
              dtype='timedelta64[ns]', freq='H')
```

All of these require an understanding of Pandas frequency codes, which we'll summarize in the next section.

Frequencies and Offsets

Fundamental to these Pandas time series tools is the concept of a frequency or date offset. Just as we saw the `D` (day) and `H` (hour) codes above, we can use such codes to

specify any desired frequency spacing. The following table summarizes the main codes available:

CodeDescription CodeDescription

D	Calendar day	B	Business day
W	Weekly		
M	Month end	BM	Business month end
Q	Quarter end	BQ	Business quarter end
A	Year end	BA	Business year end
H	Hours	BH	Business hours
T	Minutes		
S	Seconds		
L	Milliseonds		
U	Microseconds		
N	nanoseconds		

The monthly, quarterly, and annual frequencies are all marked at the end of the specified period. By adding an S suffix to any of these, they instead will be marked at the beginning:

CodeDescription CodeDescription

MS	Month start	BMS	Business month start
QS	Quarter start	BQS	Business quarter start

CodeDescription CodeDescription

AS Year start BAS Business year start

Additionally, you can change the month used to mark any quarterly or annual code by adding a three-letter month code as a suffix:

- Q-JAN, BQ-FEB, QS-MAR, BQS-APR, etc.
- A-JAN, BA-FEB, AS-MAR, BAS-APR, etc.

In the same way, the split-point of the weekly frequency can be modified by adding a three-letter weekday code:

- W-SUN, W-MON, W-TUE, W-WED, etc.

On top of this, codes can be combined with numbers to specify other frequencies. For example, for a frequency of 2 hours 30 minutes, we can combine the hour (H) and minute (T) codes as follows:

```
In [23]:
```

```
pd.timedelta_range(0, periods=9, freq="2H30T")
```

```
Out[23]:
```

```
TimedeltaIndex(['00:00:00', '02:30:00', '05:00:00', '07:30:00', '10:00:00',  
                '12:30:00', '15:00:00', '17:30:00', '20:00:00'],  
               dtype='timedelta64[ns]', freq='150T')
```

All of these short codes refer to specific instances of Pandas time series offsets, which can be found in the `pd.tseries.offsets` module. For example, we can create a business day offset directly as follows:

```
In [24]:
```

```
from pandas.tseries.offsets import BDay  
pd.date_range('2015-07-01', periods=5, freq=BDay())
```

```
Out[24]:
```

```
DatetimeIndex(['2015-07-01', '2015-07-02', '2015-07-03', '2015-07-06',  
              '2015-07-07'],  
             dtype='datetime64[ns]', freq='B')
```

For more discussion of the use of frequencies and offsets, see the ["DateOffset"](#) section of the Pandas documentation.

Resampling, Shifting, and Windowing

The ability to use dates and times as indices to intuitively organize and access data is an important piece of the Pandas time series tools. The benefits of indexed data in general (automatic alignment during operations, intuitive data slicing and access, etc.) still apply, and Pandas provides several additional time series-specific operations.

We will take a look at a few of those here, using some stock price data as an example. Because Pandas was developed largely in a finance context, it includes some very specific tools for financial data. For example, the accompanying `pandas-datareader` package (installable via `conda install pandas-datareader`), knows how to import financial data from a number of available sources, including Yahoo finance, Google Finance, and others. Here we will load Google's closing price history:

In [25]:

```
from pandas_datareader import data

goog = data.DataReader('GOOG', start='2004', end='2016',
                        data_source='google')
goog.head()
```

Out[25]:

	Open	High	Low	Close	Volume
Date					
2004-08-19	49.96	51.98	47.93	50.12	NaN
2004-08-20	50.69	54.49	50.20	54.10	NaN
2004-08-23	55.32	56.68	54.47	54.65	NaN
2004-08-24	55.56	55.74	51.73	52.38	NaN
2004-08-25	52.43	53.95	51.89	52.95	NaN

For simplicity, we'll use just the closing price:

```
In [26]:
```

```
goog = goog['Close']
```

We can visualize this using the `plot()` method, after the normal Matplotlib setup boilerplate (see [Chapter 4](#)):

```
In [27]:
```

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()
```

```
In [28]:
```

```
goog.plot();
```

Resampling and converting frequencies

One common need for time series data is resampling at a higher or lower frequency. This can be done using the `resample()` method, or the much simpler `asfreq()` method. The primary difference between the two is that `resample()` is fundamentally a *data aggregation*, while `asfreq()` is fundamentally a *data selection*.

Taking a look at the Google closing price, let's compare what the two return when we down-sample the data. Here we will resample the data at the end of business year:

```
In [29]:
```

```
goog.plot(alpha=0.5, style='-')
goog.resample('BA').mean().plot(style=':')
goog.asfreq('BA').plot(style='--');
plt.legend(['input', 'resample', 'asfreq'],
           loc='upper left');
```

Notice the difference: at each point, `resample` reports the *average of the previous year*, while `asfreq` reports the *value at the end of the year*.

For up-sampling, `resample()` and `asfreq()` are largely equivalent, though `resample` has many more options available. In this case, the default for both methods is to leave the up-sampled points empty, that is, filled with NA values. Just as with the `pd.fillna()` function discussed previously, `asfreq()` accepts a `method` argument to specify how values are

imputed. Here, we will resample the business day data at a daily frequency (i.e., including weekends):

```
In [30]:
fig, ax = plt.subplots(2, sharex=True)
data = goog.iloc[:10]

data.asfreq('D').plot(ax=ax[0], marker='o')

data.asfreq('D', method='bfill').plot(ax=ax[1], style='--o')
data.asfreq('D', method='ffill').plot(ax=ax[1], style='--o')
ax[1].legend(["back-fill", "forward-fill"]);
```

The top panel is the default: non-business days are left as NA values and do not appear on the plot. The bottom panel shows the differences between two strategies for filling the gaps: forward-filling and backward-filling.

Time-shifts

Another common time series-specific operation is shifting of data in time. Pandas has two closely related methods for computing this: `shift()` and `tshift()`. In short, the difference between them is that `shift()` *shifts the data*, while `tshift()` *shifts the index*. In both cases, the shift is specified in multiples of the frequency.

Here we will both `shift()` and `tshift()` by 900 days;

```
In [31]:
fig, ax = plt.subplots(3, sharex=True)

# apply a frequency to the data
goog = goog.asfreq('D', method='pad')

goog.plot(ax=ax[0])
goog.shift(900).plot(ax=ax[1])
goog.tshift(900).plot(ax=ax[2])

# legends and annotations
local_max = pd.to_datetime('2007-11-05')
offset = pd.Timedelta(900, 'D')

ax[0].legend(['input'], loc=2)
ax[0].get_xticklabels()[2].set(weight='heavy', color='red')
ax[0].axvline(local_max, alpha=0.3, color='red')
```

```

ax[1].legend(['shift(900)'], loc=2)
ax[1].get_xticklabels()[2].set(weight='heavy', color='red')
ax[1].axvline(local_max + offset, alpha=0.3, color='red')

ax[2].legend(['tshift(900)'], loc=2)
ax[2].get_xticklabels()[1].set(weight='heavy', color='red')
ax[2].axvline(local_max + offset, alpha=0.3, color='red');

```

We see here that `shift(900)` shifts the *data* by 900 days, pushing some of it off the end of the graph (and leaving NA values at the other end), while `tshift(900)` shifts the *index values* by 900 days.

A common context for this type of shift is in computing differences over time. For example, we use shifted values to compute the one-year return on investment for Google stock over the course of the dataset:

```

In [32]:
ROI = 100 * (goog.tshift(-365) / goog - 1)
ROI.plot()
plt.ylabel('% Return on Investment');

```

This helps us to see the overall trend in Google stock: thus far, the most profitable times to invest in Google have been (unsurprisingly, in retrospect) shortly after its IPO, and in the middle of the 2009 recession.

Rolling windows

Rolling statistics are a third type of time series-specific operation implemented by Pandas. These can be accomplished via the `rolling()` attribute of `Series` and `DataFrame` objects, which returns a view similar to what we saw with the `groupby` operation (see Aggregation and Grouping). This rolling view makes available a number of aggregation operations by default.

For example, here is the one-year centered rolling mean and standard deviation of the Google stock prices:

```

In [33]:
rolling = goog.rolling(365, center=True)

```

```
data = pd.DataFrame({'input': goog,  
                    'one-year rolling_mean': rolling.mean(),  
                    'one-year rolling_std': rolling.std()})  
ax = data.plot(style=['-', '--', ':'])  
ax.lines[0].set_alpha(0.3)
```

As with group-by operations, the `aggregate()` and `apply()` methods can be used for custom rolling computations.

Where to Learn More

This section has provided only a brief summary of some of the most essential features of time series tools provided by Pandas; for a more complete discussion, you can refer to the "Time Series/Date" section of the Pandas online documentation.

Another excellent resource is the textbook Python for Data Analysis by Wes McKinney (O'Reilly, 2012). Although it is now a few years old, it is an invaluable resource on the use of Pandas. In particular, this book emphasizes time series tools in the context of business and finance, and focuses much more on particular details of business calendars, time zones, and related topics.

As always, you can also use the IPython help functionality to explore and try further options available to the functions and methods discussed here. I find this often is the best way to learn a new Python tool.

Example: Visualizing Seattle Bicycle Counts

As a more involved example of working with some time series data, let's take a look at bicycle counts on Seattle's Fremont Bridge. This data comes from an automated bicycle counter, installed in late 2012, which has inductive sensors on the east and west sidewalks of the bridge. The hourly bicycle counts can be downloaded from <http://data.seattle.gov/>; here is the direct link to the dataset.

As of summer 2016, the CSV can be downloaded as follows:

```
In [34]:
```

```
# !curl -o FremontBridge.csv https://data.seattle.gov/api/views/65db-xm6k/rows.csv?accessType=DOWNLOAD
```

Once this dataset is downloaded, we can use Pandas to read the CSV output into a `DataFrame`. We will specify that we want the Date as an index, and we want these dates to be automatically parsed:

```
In [35]:
```

```
data = pd.read_csv('FremontBridge.csv', index_col='Date', parse_dates=True)
data.head()
```

```
Out[35]:
```

	Fremont Bridge West Sidewalk	Fremont Bridge East Sidewalk
Date		
2012-10-03 00:00:00	4.0	9.0
2012-10-03 01:00:00	4.0	6.0
2012-10-03 02:00:00	1.0	1.0
2012-10-03 03:00:00	2.0	3.0
2012-10-03 04:00:00	6.0	1.0

For convenience, we'll further process this dataset by shortening the column names and adding a "Total" column:

```
In [36]:
```

```
data.columns = ['West', 'East']
data['Total'] = data.eval('West + East')
```

Now let's take a look at the summary statistics for this data:

```
In [37]:
```

```
data.dropna().describe()
```

```
Out[37]:
```

	West	East	Total
count	35752.000000	35752.000000	35752.000000
mean	61.470267	54.410774	115.881042
std	82.588484	77.659796	145.392385
min	0.000000	0.000000	0.000000
25%	8.000000	7.000000	16.000000
50%	33.000000	28.000000	65.000000
75%	79.000000	67.000000	151.000000
max	825.000000	717.000000	1186.000000

Visualizing the data

We can gain some insight into the dataset by visualizing it. Let's start by plotting the raw data:

```
In [38]:
%matplotlib inline
import seaborn; seaborn.set()
In [39]:
data.plot()
plt.ylabel('Hourly Bicycle Count');
```

The ~25,000 hourly samples are far too dense for us to make much sense of. We can gain more insight by resampling the data to a coarser grid. Let's resample by week:

```
In [40]:
weekly = data.resample('W').sum()
weekly.plot(style=[':', '--', '-'])
plt.ylabel('Weekly bicycle count');
```


This shows us some interesting seasonal trends: as you might expect, people bicycle more in the summer than in the winter, and even within a particular season the bicycle use varies from week to week (likely dependent on weather; see In Depth: Linear Regression where we explore this further).

Another way that comes in handy for aggregating the data is to use a rolling mean, utilizing the `pd.rolling_mean()` function. Here we'll do a 30 day rolling mean of our data, making sure to center the window:

```
In [41]:
```

```
daily = data.resample('D').sum()
daily.rolling(30, center=True).sum().plot(style=[':', '--', '-'])
plt.ylabel('mean hourly count');
```

The jaggedness of the result is due to the hard cutoff of the window. We can get a smoother version of a rolling mean using a window function—for example, a Gaussian window. The following code specifies both the width of the window (we chose 50 days) and the width of the Gaussian within the window (we chose 10 days):

```
In [42]:
```

```
daily.rolling(50, center=True,
              win_type='gaussian').sum(std=10).plot(style=[':', '--', '-']);
```

Digging into the data

While these smoothed data views are useful to get an idea of the general trend in the data, they hide much of the interesting structure. For example, we might want to look at the average traffic as a function of the time of day. We can do this using the GroupBy functionality discussed in Aggregation and Grouping:

```
In [43]:
```

```
by_time = data.groupby(data.index.time).mean()
hourly_ticks = 4 * 60 * 60 * np.arange(6)
by_time.plot(xticks=hourly_ticks, style=[':', '--', '-']);
```

The hourly traffic is a strongly bimodal distribution, with peaks around 8:00 in the morning and 5:00 in the evening. This is likely evidence of a strong component of commuter traffic crossing the bridge. This is further evidenced by the differences between the western sidewalk (generally used going toward downtown Seattle), which peaks more strongly in the morning, and the eastern sidewalk (generally used going away from downtown Seattle), which peaks more strongly in the evening.

We also might be curious about how things change based on the day of the week. Again, we can do this with a simple groupby:

```
In [44]:
```

```
by_weekday = data.groupby(data.index.dayofweek).mean()
by_weekday.index = ['Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat', 'Sun']
by_weekday.plot(style=[':', '--', '-']);
```

This shows a strong distinction between weekday and weekend totals, with around twice as many average riders crossing the bridge on Monday through Friday than on Saturday and Sunday.

With this in mind, let's do a compound GroupBy and look at the hourly trend on weekdays versus weekends. We'll start by grouping by both a flag marking the weekend, and the time of day:

```
In [45]:
```

```
weekend = np.where(data.index.weekday < 5, 'Weekday', 'Weekend')
by_time = data.groupby([weekend, data.index.time]).mean()
```

Now we'll use some of the Matplotlib tools described in [Multiple Subplots](#) to plot two panels side by side:

```
In [46]:
```

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(1, 2, figsize=(14, 5))
by_time.ix['Weekday'].plot(ax=ax[0], title='Weekdays',
                           xticks=hourly_ticks, style=[':', '--', '-'])
by_time.ix['Weekend'].plot(ax=ax[1], title='Weekends',
                           xticks=hourly_ticks, style=[':', '--', '-']);
```

The result is very interesting: we see a bimodal commute pattern during the work week, and a unimodal recreational pattern during the weekends. It would be interesting to dig through this data in more detail, and examine the effect of weather, temperature, time of year, and other factors on people's commuting patterns; for further discussion, see my blog post "Is Seattle Really Seeing an Uptick In Cycling?", which uses a subset of this data. We will also revisit this dataset in the context of modeling in In Depth: Linear Regression.

High-Performance Pandas: `eval()` and `query()`

As we've already seen in previous sections, the power of the PyData stack is built upon the ability of NumPy and Pandas to push basic operations into C via an intuitive syntax: examples are vectorized/broadcasted operations in NumPy, and grouping-type operations in Pandas. While these abstractions are efficient and effective for many common use cases, they often rely on the creation of temporary intermediate objects, which can cause undue overhead in computational time and memory use.

As of version 0.13 (released January 2014), Pandas includes some experimental tools that allow you to directly access C-speed operations without costly allocation of intermediate arrays. These are the `eval()` and `query()` functions, which rely on the Numexpr package. In this notebook we will walk through their use and give some rules-of-thumb about when you might think about using them.

Motivating `query()` and `eval()`: Compound Expressions

We've seen previously that NumPy and Pandas support fast vectorized operations; for example, when adding the elements of two arrays:

```
In [1]:  
  
import numpy as np  
rng = np.random.RandomState(42)  
x = rng.rand(1000000)  
y = rng.rand(1000000)  
%timeit x + y  
100 loops, best of 3: 3.39 ms per loop
```

As discussed in Computation on NumPy Arrays: Universal Functions, this is much faster than doing the addition via a Python loop or comprehension:

In [2]:

```
%timeit np.fromiter((xi + yi for xi, yi in zip(x, y)), dtype=x.dtype, count=len(x))
1 loop, best of 3: 266 ms per loop
```

But this abstraction can become less efficient when computing compound expressions. For example, consider the following expression:

In [3]:

```
mask = (x > 0.5) & (y < 0.5)
```

Because NumPy evaluates each subexpression, this is roughly equivalent to the following:

In [4]:

```
tmp1 = (x > 0.5)
tmp2 = (y < 0.5)
mask = tmp1 & tmp2
```

In other words, *every intermediate step is explicitly allocated in memory*. If the `x` and `y` arrays are very large, this can lead to significant memory and computational overhead. The Numexpr library gives you the ability to compute this type of compound expression element by element, without the need to allocate full intermediate arrays. The Numexpr documentation has more details, but for the time being it is sufficient to say that the library accepts a *string* giving the NumPy-style expression you'd like to compute:

In [5]:

```
import numexpr
mask_numexpr = numexpr.evaluate('(x > 0.5) & (y < 0.5)')
np.allclose(mask, mask_numexpr)
```

Out[5]:

True

The benefit here is that Numexpr evaluates the expression in a way that does not use full-sized temporary arrays, and thus can be much more efficient than NumPy, especially for large arrays. The Pandas `eval()` and `query()` tools that we will discuss here are conceptually similar, and depend on the Numexpr package.

`pandas.eval()` for Efficient Operations

The `eval()` function in Pandas uses string expressions to efficiently compute operations using `DataFrames`. For example, consider the following `DataFrames`:

```
In [6]:  
  
import pandas as pd  
nrows, ncols = 100000, 100  
rng = np.random.RandomState(42)  
df1, df2, df3, df4 = (pd.DataFrame(rng.rand(nrows, ncols))  
                        for i in range(4))
```

To compute the sum of all four `DataFrames` using the typical Pandas approach, we can just write the sum:

```
In [7]:  
  
%timeit df1 + df2 + df3 + df4  
10 loops, best of 3: 87.1 ms per loop
```

The same result can be computed via `pd.eval` by constructing the expression as a string:

```
In [8]:  
  
%timeit pd.eval('df1 + df2 + df3 + df4')  
10 loops, best of 3: 42.2 ms per loop
```

The `eval()` version of this expression is about 50% faster (and uses much less memory), while giving the same result:

```
In [9]:  
  
np.allclose(df1 + df2 + df3 + df4,  
            pd.eval('df1 + df2 + df3 + df4'))  
Out[9]:  
  
True
```

Operations supported by `pd.eval()`

As of Pandas v0.16, `pd.eval()` supports a wide range of operations. To demonstrate these, we'll use the following integer `DataFrames`:

```
In [10]:  
  
df1, df2, df3, df4, df5 = (pd.DataFrame(rng.randint(0, 1000, (100, 3)))  
                            for i in range(5))
```

Arithmetic operators

`pd.eval()` supports all arithmetic operators. For example:

```
In [11]:  
result1 = -df1 * df2 / (df3 + df4) - df5  
result2 = pd.eval('-df1 * df2 / (df3 + df4) - df5')  
np.allclose(result1, result2)  
Out[11]:  
True
```

Comparison operators

`pd.eval()` supports all comparison operators, including chained expressions:

```
In [12]:  
result1 = (df1 < df2) & (df2 <= df3) & (df3 != df4)  
result2 = pd.eval('df1 < df2 <= df3 != df4')  
np.allclose(result1, result2)  
Out[12]:  
True
```

Bitwise operators

`pd.eval()` supports the `&` and `|` bitwise operators:

```
In [13]:  
result1 = (df1 < 0.5) & (df2 < 0.5) | (df3 < df4)  
result2 = pd.eval('(df1 < 0.5) & (df2 < 0.5) | (df3 < df4)')  
np.allclose(result1, result2)  
Out[13]:  
True
```

In addition, it supports the use of the literal `and` and `or` in Boolean expressions:

```
In [14]:  
result3 = pd.eval('(df1 < 0.5) and (df2 < 0.5) or (df3 < df4)')  
np.allclose(result1, result3)  
Out[14]:  
True
```

Object attributes and indices

`pd.eval()` supports access to object attributes via the `obj.attr` syntax, and indexes via the `obj[index]` syntax:

```
In [15]:
result1 = df2.T[0] + df3.iloc[1]
result2 = pd.eval('df2.T[0] + df3.iloc[1]')
np.allclose(result1, result2)
Out[15]:
True
```

Other operations

Other operations such as function calls, conditional statements, loops, and other more involved constructs are currently *not* implemented in `pd.eval()`. If you'd like to execute these more complicated types of expressions, you can use the Numexpr library itself.

`DataFrame.eval()` for Column-Wise Operations

Just as Pandas has a top-level `pd.eval()` function, `DataFrame`s have an `eval()` method that works in similar ways. The benefit of the `eval()` method is that columns can be referred to *by name*. We'll use this labeled array as an example:

```
In [16]:
df = pd.DataFrame(rng.rand(1000, 3), columns=['A', 'B', 'C'])
df.head()
Out[16]:
```

	A	B	C
0	0.375506	0.406939	0.069938
1	0.069087	0.235615	0.154374
2	0.677945	0.433839	0.652324
3	0.264038	0.808055	0.347197

	A	B	C
4	0.589161	0.252418	0.557789

Using `pd.eval()` as above, we can compute expressions with the three columns like this:

```
In [17]:
result1 = (df['A'] + df['B']) / (df['C'] - 1)
result2 = pd.eval("(df.A + df.B) / (df.C - 1)")
np.allclose(result1, result2)
Out[17]:
```

True

The `DataFrame.eval()` method allows much more succinct evaluation of expressions with the columns:

```
In [18]:
result3 = df.eval('(A + B) / (C - 1)')
np.allclose(result1, result3)
Out[18]:
```

True

Notice here that we treat *column names as variables* within the evaluated expression, and the result is what we would wish.

Assignment in DataFrame.eval()

In addition to the options just discussed, `DataFrame.eval()` also allows assignment to any column. Let's use the `DataFrame` from before, which has columns 'A', 'B', and 'C':

```
In [19]:
df.head()
Out[19]:
```

	A	B	C
0	0.375506	0.406939	0.069938
1	0.069087	0.235615	0.154374

	A	B	C
2	0.677945	0.433839	0.652324
3	0.264038	0.808055	0.347197
4	0.589161	0.252418	0.557789

We can use `df.eval()` to create a new column 'D' and assign to it a value computed from the other columns:

In [20]:

```
df.eval('D = (A + B) / C', inplace=True)
df.head()
```

Out[20]:

	A	B	C	D
0	0.375506	0.406939	0.069938	11.187620
1	0.069087	0.235615	0.154374	1.973796
2	0.677945	0.433839	0.652324	1.704344
3	0.264038	0.808055	0.347197	3.087857
4	0.589161	0.252418	0.557789	1.508776

In the same way, any existing column can be modified:

In [21]:

```
df.eval('D = (A - B) / C', inplace=True)
df.head()
```

Out[21]:

	A	B	C	D
0	0.375506	0.406939	0.069938	-0.449425
1	0.069087	0.235615	0.154374	-1.078728
2	0.677945	0.433839	0.652324	0.374209
3	0.264038	0.808055	0.347197	-1.566886
4	0.589161	0.252418	0.557789	0.603708

Local variables in DataFrame.eval()

The `DataFrame.eval()` method supports an additional syntax that lets it work with local Python variables. Consider the following:

In [22]:

```
column_mean = df.mean(1)
result1 = df['A'] + column_mean
result2 = df.eval('A + @column_mean')
np.allclose(result1, result2)
```

Out[22]:

True

The `@` character here marks a *variable name* rather than a *column name*, and lets you efficiently evaluate expressions involving the two "namespaces": the namespace of columns, and the namespace of Python objects. Notice that this `@` character is only supported by the `DataFrame.eval()` *method*, not by the `pandas.eval()` *function*, because the `pandas.eval()` function only has access to the one (Python) namespace.

DataFrame.query() Method

The `DataFrame` has another method based on evaluated strings, called the `query()` method. Consider the following:

In [23]:

```
result1 = df[(df.A < 0.5) & (df.B < 0.5)]
result2 = pd.eval('df[(df.A < 0.5) & (df.B < 0.5)]')
np.allclose(result1, result2)
Out[23]:
```

True

As with the example used in our discussion of `DataFrame.eval()`, this is an expression involving columns of the `DataFrame`. It cannot be expressed using the `DataFrame.eval()` syntax, however! Instead, for this type of filtering operation, you can use the `query()` method:

```
In [24]:
result2 = df.query('A < 0.5 and B < 0.5')
np.allclose(result1, result2)
Out[24]:
```

True

In addition to being a more efficient computation, compared to the masking expression this is much easier to read and understand. Note that the `query()` method also accepts the `@` flag to mark local variables:

```
In [25]:
Cmean = df['C'].mean()
result1 = df[(df.A < Cmean) & (df.B < Cmean)]
result2 = df.query('A < @Cmean and B < @Cmean')
np.allclose(result1, result2)
Out[25]:
```

True

Performance: When to Use These Functions

When considering whether to use these functions, there are two considerations: *computation time* and *memory use*. Memory use is the most predictable aspect. As already mentioned, every compound expression involving NumPy arrays or Pandas `DataFrames` will result in implicit creation of temporary arrays: For example, this:

```
In [26]:
x = df[(df.A < 0.5) & (df.B < 0.5)]
```

Is roughly equivalent to this:

```
In [27]:
```

```
tmp1 = df.A < 0.5
tmp2 = df.B < 0.5
tmp3 = tmp1 & tmp2
x = df[tmp3]
```

If the size of the temporary `DataFrames` is significant compared to your available system memory (typically several gigabytes) then it's a good idea to use an `eval()` or `query()` expression. You can check the approximate size of your array in bytes using this:

```
In [28]:
```

```
df.values.nbytes
```

```
Out[28]:
```

```
32000
```

On the performance side, `eval()` can be faster even when you are not maxing-out your system memory. The issue is how your temporary `DataFrames` compare to the size of the L1 or L2 CPU cache on your system (typically a few megabytes in 2016); if they are much bigger, then `eval()` can avoid some potentially slow movement of values between the different memory caches. In practice, I find that the difference in computation time between the traditional methods and the `eval/query` method is usually not significant—if anything, the traditional method is faster for smaller arrays! The benefit of `eval/query` is mainly in the saved memory, and the sometimes cleaner syntax they offer.