Working with Pandas DataFrames

13/07/2024

Pandas Data Structures

Python has several data structures already, such as tuples, lists, and dictionaries. Pandas provides two main structures to facilitate working with data: Series and DataFrame.

Note: To understand these data structures, we need to first look at NumPy which provides the n-dimensional arrays that pandas builds upon.

These data structures are implemented as Python classes; when created, they are referred to as objects or instances.

This distinction is important because some actions can be performed using the object itself (a method), while others require passing the object as an argument to a function.

Pandas Data Structures

We use a pandas' function to read a CSV file into a DataFrame object but use methods on DataFrame objects to perform actions like dropping columns or calculating summary statistics.

With pandas, we often access the object's attributes to get information such as dimensions, column names, data types, and whether it is empty.

For this section, we will read five rows from Earthquake data comes from the USGS API for earthquakes (source: https://earthquake.usgs.gov/fdsnws/event/1/).

```
>>> import numpy as np

>>> data = np.genfromtxt(
...     'data/example_data.csv', delimiter=';',
...     names=True, dtype=None, encoding='UTF'
... )
```

Pandas Data Structures


```
array([('2018-10-13 11:10:23.560',
        '262km NW of Ozernovskiy, Russia',
        'mww', 6.7, 'green', 1),
       ('2018-10-13 04:34:15.580',
        '25km E of Bitung, Indonesia', 'mww', 5.2, 'green', 0),
       ('2018-10-13 00:13:46.220', '42km WNW of Sola, Vanuatu',
        'mww', 5.7, 'green', 0),
       ('2018-10-12 21:09:49.240',
        '13km E of Nueva Concepcion, Guatemala',
        'mww', 5.7, 'green', 0),
       ('2018-10-12 02:52:03.620',
        '128km SE of Kimbe, Papua New Guinea',
        'mww', 5.6, 'green', 1)],
      dtype=[('time', '<U23'), ('place', '<U37'),
             ('magType', '<U3'), ('mag', '<f8'),
             ('alert', '<U5'), ('tsunami', '<i8')])
```

```
>>> data.shape
(5,)
>>> data.dtype
dtype([('time', '<U23'), ('place', '<U37'), ('magType', '<U3'),
       ('mag', '<f8'), ('alert', '<U5'), ('tsunami', '<i8')])
```

Pandas Data Structures

Each of the entries in the array is a row from the CSV file. NumPy arrays contain a single data type (unlike lists, which allow mixed types); this allows for fast, vectorized operations.

When we read in the data, we got an array of *numpy.void* objects, which are used to store flexible types. This is because NumPy had to store several different data types per row: four strings, a float, and an integer.

Unfortunately, this means that we can't take advantage of the performance improvements NumPy provides for single data type objects.

```
>>> %%timeit
>>> max([row[3] for row in data])
9.74 µs ± 177 ns per loop
(mean ± std. dev. of 7 runs, 100000 loops each)
```

Pandas Data Structures

If we create a NumPy array for each column instead, this operation is much easier (and more efficient) to perform.

```
>>> array_dict = {
...     col: np.array([row[i] for row in data])
...     for i, col in enumerate(data.dtype.names)
... }
>>> array_dict
{'time': array(['2018-10-13 11:10:23.560',
       '2018-10-13 04:34:15.580', '2018-10-13 00:13:46.220',
       '2018-10-12 21:09:49.240', '2018-10-12 02:52:03.620'],
       dtype='<U23'),
 'place': array(['262km NW of Ozernovskiy, Russia',
       '25km E of Bitung, Indonesia',
       '42km WNW of Sola, Vanuatu',
       '13km E of Nueva Concepcion, Guatemala',
       '128km SE of Kimbe, Papua New Guinea'], dtype='<U37'),
 'magType': array(['mww', 'mww', 'mww', 'mww', 'mww'],
       dtype='<U3'),
 'mag': array([6.7, 5.2, 5.7, 5.7, 5.6]),
 'alert': array(['green', 'green', 'green', 'green', 'green'],
       dtype='<U5'),
 'tsunami': array([1, 0, 0, 0, 1])}
```

Pandas Data Structures

Grabbing the maximum magnitude is now simply a matter of selecting the mag key and calling the max() method on the NumPy array.

```
>>> %%timeit
>>> array_dict['mag'].max()
5.22 µs ± 100 ns per loop
(mean ± std. dev. of 7 runs, 100000 loops each)
```

However, this representation has other issues. Say we wanted to grab all the information for the earthquake with the maximum magnitude; how would we go about that?

```
>>> np.array([
...     value[array_dict['mag'].argmax()]
...     for key, value in array_dict.items()
... ])
array(['2018-10-13 11:10:23.560',
       '262km NW of Ozernovskiy, Russia',
       'mww', '6.7', 'green', '1'], dtype='<U31')
```

Pandas Data Structures

Consider how we would go about sorting the data by magnitude from smallest to largest.

In the first representation, we would have to sort the rows by examining the third index.

With the second representation, we would have to determine the order of the indices from the mag column, and then sort all the other arrays with those same indices.

Clearly, working with several NumPy arrays containing different data types at once is a bit cumbersome;

However, pandas builds on top of NumPy arrays to make this easier. Let's start our exploration of pandas with an overview of the **Series** data structure.

Pandas Data Structures - Series

The Series class provides a data structure for arrays of a single type, just like the NumPy array. However, it comes with some additional functionality.

```
>>> import pandas as pd

>>> place = pd.Series(array_dict['place'], name='place')
>>> place
0           262km NW of Ozernovskiy, Russia
1               25km E of Bitung, Indonesia
2                 42km WNW of Sola, Vanuatu
3    13km E of Nueva Concepcion, Guatemala
4      128km SE of Kimbe, Papua New Guinea
Name: place, dtype: object
```

To access attributes of the Series object, we use attribute notation of the form <object>.<attribute_name>.

Pandas Data Structures - Series

The following are some common attributes we will access. Notice that dtype and shape are available, just as we saw with the NumPy array:

Attribute	Returns
`name`	The name of the `Series` object
`dtype`	The data type of the `Series` object
`shape`	Dimensions of the `Series` object in a tuple of the form `(number of rows,)`
`index`	The `Index` object that is part of the `Series` object
`values`	The data in the `Series` object

Be sure to bookmark the pandas.Series documentation (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html) for reference later.

Pandas Data Structures - Index

The addition of the Index class makes the Series class significantly more powerful than a NumPy array.

The Index class gives us row labels, which enable selection by row.

```
>>> place_index = place.index
>>> place_index
RangeIndex(start=0, stop=5, step=1)
```

The default index class is RangeIndex; however, we can change the index .Often, we work with an Index object of row numbers or date(time)s.

Pandas Data Structures - Index

As with Series objects, we can access the underlying data via the values attribute. Note that this Index object is built on top of a NumPy array:

```
>>> place_index.values
array([0, 1, 2, 3, 4], dtype=int64)
```

Some of the useful attributes of Index objects include the following:

Attribute	Returns
`name`	The name of the `Index` object
`dtype`	The data type of the `Index` object
`shape`	Dimensions of the `Index` object
`values`	The data in the `Index` object
`is_unique`	Check if the `Index` object has all unique values

Pandas Data Structures - Index

Both NumPy and pandas support arithmetic operations, which will be performed element-wise. NumPy will use the position in the array for this:

```
>>> np.array([1, 1, 1]) + np.array([-1, 0, 1])
array([0, 1, 2])
```

With pandas, this element-wise arithmetic is performed on matching index values.

```
>>> numbers = np.linspace(0, 10, num=5) # [0, 2.5, 5, 7.5, 10]
>>> x = pd.Series(numbers) # index is [0, 1, 2, 3, 4]
>>> y = pd.Series(numbers, index=pd.Index([1, 2, 3, 4, 5]))
>>> x + y
0     NaN
1     2.5
2     7.5
3    12.5
4    17.5
5     NaN
dtype: float64
```

Pandas Data Structures - DataFrame

With the Series class, we essentially had columns of a spreadsheet, with the data all being of the same type. The **DataFrame** class builds upon the Series class and can have many columns, each with its own data type; we can think of it as representing the spreadsheet as a whole.

We can turn either of the NumPy representations we built from the example data into a DataFrame object:

```
>>> df = pd.DataFrame(array_dict)
>>> df
```

	time	place	magType	mag	alert	tsunami
0	2018-10-13 11:10:23.560	262km NW of Ozernovskiy, Russia	mww	6.7	green	1
1	2018-10-13 04:34:15.580	25km E of Bitung, Indonesia	mww	5.2	green	0
2	2018-10-13 00:13:46.220	42km WNW of Sola, Vanuatu	mww	5.7	green	0
3	2018-10-12 21:09:49.240	13km E of Nueva Concepcion, Guatemala	mww	5.7	green	0
4	2018-10-12 02:52:03.620	128km SE of Kimbe, Papua New Guinea	mww	5.6	green	1

Pandas Data Structures - DataFrame

Our columns each have a single data type, but they don't all share the same data type:

```
>>> df.dtypes
time         object
place        object
magType      object
mag         float64
alert        object
tsunami       int64
dtype: object
```

Pandas Data Structures - DataFrame

The values of the dataframe look very similar to the initial NumPy representation we had:

```
>>> df.values
array([['2018-10-13 11:10:23.560',
        '262km NW of Ozernovskiy, Russia',
        'mww', 6.7, 'green', 1],
       ['2018-10-13 04:34:15.580',
        '25km E of Bitung, Indonesia', 'mww', 5.2, 'green', 0],
       ['2018-10-13 00:13:46.220', '42km WNW of Sola, Vanuatu',
        'mww', 5.7, 'green', 0],
       ['2018-10-12 21:09:49.240',
        '13km E of Nueva Concepcion, Guatemala',
        'mww', 5.7, 'green', 0],
       ['2018-10-12 02:52:03.620','128 km SE of Kimbe,
         Papua New Guinea', 'mww', 5.6, 'green', 1]],
      dtype=object)
```

Pandas Data Structures - DataFrame

We can access the column names via the columns attribute. Note that they are actually stored in an Index object as well:

```
>>> df.columns
Index(['time', 'place', 'magType', 'mag', 'alert', 'tsunami'],
      dtype='object')
```

The following are some commonly used dataframe attributes:

Attribute	Returns
`dtypes`	The data types of each column
`shape`	Dimensions of the `DataFrame` object in a tuple of the form `(number of rows, number of columns)`
`index`	The `Index` object along the rows of the `DataFrame` object
`columns`	The name of the columns (as an `Index` object)
`values`	The data in the `DataFrame` object
`empty`	Check if the `DataFrame` object is empty

Pandas Data Structures - DataFrame

Note that we can also perform arithmetic on dataframes. For example, we can add df to itself, which will sum the numeric columns and concatenate the string columns:

```
>>> df + df
```

Pandas will only perform the operation when both the index and column match. Here, pandas concatenated the string columns (time, place, magType, and alert) across dataframes. The numeric columns (mag and tsunami) were summed:

	time	place	magType	mag	alert	tsunami
0	2018-10-13 11:10:23.5602018-10-13 11:10:23.560	262km NW of Ozernovskiy, Russia262km NW of Oze...	mwwmww	13.4	greengreen	2
1	2018-10-13 04:34:15.5802018-10-13 04:34:15.580	25km E of Bitung, Indonesia25km E of Bitung, I...	mwwmww	10.4	greengreen	0
2	2018-10-13 00:13:46.2202018-10-13 00:13:46.220	42km WNW of Sola, Vanuatu42km WNW of Sola, Van...	mwwmww	11.4	greengreen	0
3	2018-10-12 21:09:49.2402018-10-12 21:09:49.240	13km E of Nueva Concepcion, Guatemala13km E of...	mwwmww	11.4	greengreen	0
4	2018-10-12 02:52:03.6202018-10-12 02:52:03.620	128km SE of Kimbe, Papua New Guinea128km SE of...	mwwmww	11.2	greengreen	2

Creating a Pandas DataFrame

Let's import the packages we will need for the upcoming examples:

```
>>> import datetime as dt
>>> import numpy as np
>>> import pandas as pd
```

We are now ready to begin using pandas. First, we will learn how to create pandas objects from other Python objects. Then, we will learn how to do so with flat files, tables in a database, and responses from API requests.

Creating a Pandas DataFrame – From Python Object

Before we cover how to create a DataFrame from a Python object, we should learn how to make a Series. Since a Series is essentially a column in a DataFrame, understanding this will make creating a DataFrame easier.

```
>>> np.random.seed(0) # set a seed for reproducibility
>>> pd.Series(np.random.rand(5), name='random')
0    0.548814
1    0.715189
2    0.602763
3    0.544883
4    0.423655
Name: random, dtype: float64
```

Making a DataFrame object is an extension of making a Series object; it will be composed of one or more series, and each will be distinctly named. This should remind us of dictionary-like structures in Python.

Note that if we want to turn a single Series object into a DataFrame object, we can use its to_frame() method.

Creating a Pandas DataFrame – From Python Object

We will create a DatetimeIndex object using pd.date_range(), containing five dates (periods=5) one day apart (freq='1D'), ending on some date (end), and named date.

We will Package the columns in a dictionary using the desired column names as keys and pass this to the pd.DataFrame() constructor, with the index passed as the index argument:

```
>>> np.random.seed(0) # set seed so result is reproducible
>>> pd.DataFrame(
...     {
...         'random': np.random.rand(5),
...         'text': ['hot', 'warm', 'cool', 'cold', None],
...         'truth': [np.random.choice([True, False])
...                   for _ in range(5)]
...     },
...     index=pd.date_range(
...         end=dt.date(2019, 4, 21),
...         freq='1D', periods=5, name='date'
...     )
... )
```

Creating a Pandas DataFrame – From Python Object

date	random	text	truth
2019-04-17	0.548814	hot	False
2019-04-18	0.715189	warm	True
2019-04-19	0.602763	cool	True
2019-04-20	0.544883	cold	False
2019-04-21	0.423655	None	True

Having dates in the index makes it easy to select entries by date (or even in a date range), as we will see later.

Creating a Pandas DataFrame – From Python Object

In cases where the data isn't a dictionary, but rather a list of dictionaries, we can still use pd.DataFrame()

```
>>> pd.DataFrame([
...     {'mag': 5.2, 'place': 'California'},
...     {'mag': 1.2, 'place': 'Alaska'},
...     {'mag': 0.2, 'place': 'California'},
... ])
```

Data in this format is what we would expect when consuming from an API. Each entry in the list will be a dictionary, where the keys of the dictionary are the column names, and the values of the dictionary are the values for that column at that index.

	mag	place
0	5.2	California
1	1.2	Alaska
2	0.2	California

Creating a Pandas DataFrame – From Python Object

In fact, pd.DataFrame() also works for lists of tuples. Note that we can also pass in the column names as a list through the columns argument:

```
>>> list_of_tuples = [(n, n**2, n**3) for n in range(5)]
>>> list_of_tuples
[(0, 0, 0), (1, 1, 1), (2, 4, 8), (3, 9, 27), (4, 16, 64)]

>>> pd.DataFrame(
...     list_of_tuples,
...     columns=['n', 'n_squared', 'n_cubed']
... )
```

Each tuple is treated like a record and becomes a row in the dataframe:

	n	n_squared	n_cubed
0	0	0	0
1	1	1	1
2	2	4	8
3	3	9	27
4	4	16	64

Creating a Pandas DataFrame – From Python Object

We also have the option of using pd.DataFrame() with NumPy arrays:

```
>>> pd.DataFrame(
...     np.array([
...         [0, 0, 0],
...         [1, 1, 1],
...         [2, 4, 8],
...         [3, 9, 27],
...         [4, 16, 64]
...     ]), columns=['n', 'n_squared', 'n_cubed']
... )
```

This will have the effect of stacking each entry in the array as rows in a dataframe, giving us the same result:

	n	n_squared	n_cubed
0	0	0	0
1	1	1	1
2	2	4	8
3	3	9	27
4	4	16	64

Creating a Pandas DataFrame – From a File

The data we want to analyze will most often come from outside Python. Often, these data dumps will come in the form of a text file (.txt) or a CSV file (.csv).

Our earthquake data is a CSV file; therefore, we use the pd.read_csv() function to read it in.

However, we should always do an initial inspection of the file before attempting to read it in; this will inform us of whether we need to pass additional arguments, such as sep to specify the delimiter or names to provide the column names ourselves in the absence of a header row in the file.

Creating a Pandas DataFrame – From a File

First, we should check how big the file is, both in terms of lines and in terms of bytes.

```
>>> !wc -l data/earthquakes.csv
9333 data/earthquakes.csv
```

Now, let's check the file's size.

```
>>> !ls -lh data | grep earthquakes.csv
-rw-r--r-- 1 stefanie stefanie 3.4M ... earthquakes.csv
```

Note that IPython also lets us capture the result of the command in a Python variable, so if we aren't comfortable with pipes (|) or grep, we can do the following:

```
>>> files = !ls -lh data
>>> [file for file in files if 'earthquake' in file]
['-rw-r--r-- 1 stefanie stefanie 3.4M ... earthquakes.csv']
```

Creating a Pandas DataFrame – From a File

Now, let's take a look at the top few rows to see if the file comes with headers.

```
>>> !head -n 2 data/earthquakes.csv
alert,cdi,code,detail,dmin,felt,gap,ids,mag,magType,mmi,
net,nst,place,rms,sig,sources,status,time,title,tsunami,
type,types,tz,updated,url
,,37389218,https://earthquake.usgs.gov/[...],0.008693,,85.0,",
ci37389218,",1.35,ml,,ci,26.0,"9km NE of Aguanga,
CA",0.19,28,",ci,",automatic,1539475168010,"M 1.4 -
9km NE of Aguanga, CA",0,earthquake,",geoserve,nearby-
cities,origin,phase-data,",-480.0,1539475395144,
https://earthquake.usgs.gov/earthquakes/eventpage/ci37389218
```

Note that we should also check the bottom rows to make sure there is no extraneous data that we will need to ignore by using the tail utility.

Creating a Pandas DataFrame – From a File

Lastly, we may be interested in seeing the column count in our data. While we could just count the fields in the first row of the result of head, we have the option of using the awk utility (for pattern scanning and processing) to count our columns.

```
>>> !awk -F',' '{print NF; exit}' data/earthquakes.csv
26
```

Since we know that the first line of the file contains headers and that the file is comma-separated, we can also count the columns by using head to get the headers and Python to parse them:

```
>>> headers = !head -n 1 data/earthquakes.csv
>>> len(headers[0].split(','))
26
```

Creating a Pandas DataFrame – From a File

To summarize, we now know that the file is 3.4 MB and is comma-delimited with 26 columns and 9,333 rows, with the first one being the header. This means that we can use the pd.read_csv() function with the defaults:

```
>>> df = pd.read_csv('earthquakes.csv')
```

Note that we aren't limited to reading in data from files on our local machines; file paths can be URLs as well. As an example, let's read in the same CSV file from GitHub:

```
>>> df = pd.read_csv(
...     'https://github.com/stefmolin/'
...     'Hands-On-Data-Analysis-with-Pandas-2nd-edition'
...     '/blob/master/ch_02/data/earthquakes.csv?raw=True'
... )
```

Creating a Pandas DataFrame – From a File

Pandas usually figures out the best options based on the input data, so we often don't need to add arguments; however, many options are available if needed:

Parameter	Purpose
`sep`	Specifies the delimiter
`header`	Row number where the column names are located; the default option has `pandas` infer whether they are present
`names`	List of column names to use as the header
`index_col`	Column to use as the index
`usecols`	Specifies which columns to read in
`dtype`	Specifies data types for the columns
`converters`	Specifies functions for converting data in certain columns
`skiprows`	Rows to skip
`nrows`	Number of rows to read at a time (combine with `skiprows` to read a file bit by bit)
`parse_dates`	Automatically parse columns containing dates into datetime objects
`chunksize`	For reading the file in chunks
`compression`	For reading in compressed files without extracting beforehand
`encoding`	Specifies the file encoding

Creating a Pandas DataFrame – From a File

Note that we can use the read_excel() function to read in Excel files, the read_json() function for JSON (JavaScript Object Notation) files, and for other delimited files, such as tab (\t), we can use the read_csv() function with the sep argument equal to the delimiter.

To write our dataframe to a CSV file, we call its to_csv() method.

```
>>> df.to_csv('output.csv', index=False)
```

As with reading from files, Series and DataFrame objects have methods to write data to Excel (to_excel()) and JSON files (to_json()).

Creating a Pandas DataFrame – From a Database

Pandas can interact with SQLite databases without additional packages, but SQLAlchemy is required to interact with other database types.

Before we read from a database, let's write to one. We simply call to_sql() on our dataframe, telling it which table to write to, which database connection to use, and how to handle if the table already exists.

```
>>> import sqlite3

>>> with sqlite3.connect('data/quakes.db') as connection:
...     pd.read_csv('data/tsunamis.csv').to_sql(
...         'tsunamis', connection, index=False,
...         if_exists='replace'
...     )
```

Creating a Pandas DataFrame – From a Database

To actually query the database, we use pd.read_sql(), passing in our query and the database connection:

```
>>> import sqlite3

>>> with sqlite3.connect('data/quakes.db') as connection:
...     tsunamis = \
...         pd.read_sql('SELECT * FROM tsunamis', connection)

>>> tsunamis.head()
```

Creating a Pandas DataFrame – From a Database

We now have the tsunamis data in a dataframe:

	alert	type	title	place	magType	mag	time
0	None	earthquake	M 5.0 - 165km NNW of Flying Fish Cove, Christm...	165km NNW of Flying Fish Cove, Christmas Island	mww	5.0	1539459504090
1	green	earthquake	M 6.7 - 262km NW of Ozernovskiy, Russia	262km NW of Ozernovskiy, Russia	mww	6.7	1539429023560
2	green	earthquake	M 5.6 - 128km SE of Kimbe, Papua New Guinea	128km SE of Kimbe, Papua New Guinea	mww	5.6	1539312723620
3	green	earthquake	M 6.5 - 148km S of Severo-Kuril'sk, Russia	148km S of Severo-Kuril'sk, Russia	mww	6.5	1539213362130
4	green	earthquake	M 6.2 - 94km SW of Kokopo, Papua New Guinea	94km SW of Kokopo, Papua New Guinea	mww	6.2	1539208835130

Important note - The connection object we created in both code blocks is an example of a context manager, which, when used with the with statement, automatically handles cleanup after the code in the block executes (closing the connection, in this case).

Creating a Pandas DataFrame – From an API

We can create Series and DataFrame objects from Python data or files, but obtaining data from online sources like APIs requires flexibility.

Now, we'll request earthquake data from the USGS API and convert it into a dataframe

So now we have to import the packages we need once again:

```
>>> import datetime as dt
>>> import pandas as pd
>>> import requests
```

Creating a Pandas DataFrame – From an API

So now we have Next, we will make a GET request to the USGS API for a JSON payload (a dictionarylike response containing the data that's sent with a request or response).

```
>>> yesterday = dt.date.today() - dt.timedelta(days=1)
>>> api = 'https://earthquake.usgs.gov/fdsnws/event/1/query'
>>> payload = {
...     'format': 'geojson',
...     'starttime': yesterday - dt.timedelta(days=30),
...     'endtime': yesterday
... }
>>> response = requests.get(api, params=payload)
```

Before we try to create a dataframe out of this, we should make sure that our request was successful.

```
>>> response.status_code
200
```

Creating a Pandas DataFrame – From an API

We asked the API for a JSON payload, which is essentially a dictionary, so we can use dictionary methods on it to get more information about its structure.

We need to isolate the JSON payload from the HTTP response (stored in the response variable), and then look at the keys to view the main sections of the resulting data:

```
>>> earthquake_json = response.json()
>>> earthquake_json.keys()
dict_keys(['type', 'metadata', 'features', 'bbox'])
```

We can inspect what kind of data we have as values for each of these keys; one of them will be the data we are after.

Creating a Pandas DataFrame – From an API

The metadata portion tells us some information about our request.

```
>>> earthquake_json['metadata']
{'generated': 1604267813000,
 'url': 'https://earthquake.usgs.gov/fdsnws/event/1/query?
format=geojson&starttime=2020-10-01&endtime=2020-10-31',
 'title': 'USGS Earthquakes',
 'status': 200,
 'api': '1.10.3',
 'count': 13706}
```

Creating a Pandas DataFrame – From an API

The features key looks promising; if this does indeed contain all our data, we should check what type it is so that we don't end up trying to print everything to the screen:

```
>>> type(earthquake_json['features'])
list
```

This key contains a list, so let's look at the first entry to see if this is the data we want.

```
>>> earthquake_json['features'][0]
{'type': 'Feature',
 'properties': {'mag': 1,
  'place': '50 km ENE of Susitna North, Alaska',
  'time': 1604102395919, 'updated': 1604103325550, 'tz': None,
  'url': 'https://earthquake.usgs.gov/earthquakes/eventpage/
ak020dz5f85a',
  'detail': 'https://earthquake.usgs.gov/fdsnws/event/1/
query?eventid=ak020dz5f85a&format=geojson',
  'felt': None, 'cdi': None, 'mmi': None, 'alert': None,
  'status': 'reviewed', 'tsunami': 0, 'sig': 15, 'net': 'ak',
  'code': '020dz5f85a', 'ids': ',ak020dz5f85a,',
  'sources': ',ak,', 'types': ',origin,phase-data,',
  'nst': None, 'dmin': None, 'rms': 1.36, 'gap': None,
  'magType': 'ml', 'type': 'earthquake',
  'title': 'M 1.0 - 50 km ENE of Susitna North, Alaska'},
 'geometry': {'type': 'Point', 'coordinates': [-148.9807,
62.3533, 5]},
 'id': 'ak020dz5f85a'}
```

Creating a Pandas DataFrame – From an API

This is the data we are after, but do we need all of it? Upon closer inspection, we only really care about what is inside the properties dictionary.

```
>>> earthquake_properties_data = [
...     quake['properties']
...     for quake in earthquake_json['features']
... ]
```

Finally, we are ready to create our dataframe. Pandas knows how to handle data in this format already (a list of dictionaries), so all we have to do is pass in the data when we call pd.DataFrame():

```
>>> df = pd.DataFrame(earthquake_properties_data)
```

Inspecting a DataFrame Object - Examining the Data

1. Examining the data

First, we want to make sure that we have data in our dataframe. We can check the empty attribute to find out:

```
>>> df.empty
False
```

Next, we should check how much data we read in; we want to know the number of observations (rows) and the number of variables (columns) we have.

```
>>> df.shape
(9332, 26)
```

Now, let's use the columns attribute to see the names of the columns in our dataset:

```
>>> df.columns
Index(['alert', 'cdi', 'code', 'detail', 'dmin', 'felt', 'gap',
       'ids', 'mag', 'magType', 'mmi', 'net', 'nst', 'place',
       'rms', 'sig', 'sources', 'status', 'time', 'title',
       'tsunami', 'type', 'types', 'tz', 'updated', 'url'],
      dtype='object')
```

Inspecting a DataFrame Object - Examining the Data

We can use the head() and tail() methods to look at the top and bottom rows, respectively.

```
>>> df.head()
```

The following are the first five rows we get using head():

	alert	...	dmin	felt	...	mag	magType	...	place	...	time	title	tsunami	...	updated	url
0	NaN	...	0.008693	NaN	...	1.35	ml	...	9km NE of Aguanga, CA	...	1539475168010	M 1.4 - 9km NE of Aguanga, CA	0	...	1539475395144	https...
1	NaN	...	0.020030	NaN	...	1.29	ml	...	9km NE of Aguanga, CA	...	1539475129610	M 1.3 - 9km NE of Aguanga, CA	0	...	1539475253925	https...
2	NaN	...	0.021370	28.0	...	3.42	ml	...	8km NE of Aguanga, CA	...	1539475062610	M 3.4 - 8km NE of Aguanga, CA	0	...	1539536756176	https...
3	NaN	...	0.026180	NaN	...	0.44	ml	...	9km NE of Aguanga, CA	...	1539474978070	M 0.4 - 9km NE of Aguanga, CA	0	...	1539475196167	https...
4	NaN	...	0.077990	NaN	...	2.16	md	...	10km NW of Avenal, CA	...	1539474716050	M 2.2 - 10km NW of Avenal, CA	0	...	1539477547926	https...

Inspecting a DataFrame Object - Examining the Data

To get the last two rows, we use the tail() method and pass 2 as the number of rows:

```
>>> df.tail(2)
```

The following is the result:

	alert	...	dmin	felt	...	mag	magType	...	place	...	time	title	tsunami	...	updated	url
9330	NaN	...	0.01865	NaN	...	1.10	ml	...	9km NE of Aguanga, CA	...	1537229545350	M 1.1 - 9km NE of Aguanga, CA	0	...	1537230211640	https...
9331	NaN	...	0.01698	NaN	...	0.66	ml	...	9km NE of Aguanga, CA	...	1537228864470	M 0.7 - 9km NE of Aguanga, CA	0	...	1537305830770	https...

Tip

By default, when we print dataframes with many columns in a Jupyter Notebook, only a subset of them will be displayed. This is because `pandas` has a limit on the number of columns it will show. We can modify this behavior using `pd.set_option('display.max_columns', <new_value>)`. Consult the documentation at `https://pandas.pydata.org/pandas-docs/stable/user_guide/options.html` for additional information. The notebook also contains a few example commands.

Inspecting a DataFrame Object - Examining the Data

We can use the dtypes attribute to see the data types of the columns, which makes it easy to see when columns are being stored as the wrong type.

```
>>> df.dtypes
alert          object
...
mag           float64
magType        object
...
time             int64
title          object
tsunami          int64
...
tz            float64
updated          int64
url            object
dtype: object
```

Inspecting a DataFrame Object - Examining the Data

Lastly, we can use the info() method to see how many non-null entries of each column we have and get information on our index.

```
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9332 entries, 0 to 9331
Data columns (total 26 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   alert    59 non-null     object
 ...
 8   mag      9331 non-null   float64
 9   magType  9331 non-null   object
 ...
 18  time     9332 non-null   int64
 19  title    9332 non-null   object
 20  tsunami  9332 non-null   int64
 ...
 23  tz       9331 non-null   float64
 24  updated  9332 non-null   int64
 25  url      9332 non-null   object
dtypes: float64(9), int64(4), object(13)
memory usage: 1.9+ MB
```

Null values are missing values, which will typically be represented as:

- None for objects
- NaN for non-numeric values in a float or integer column

Inspecting a DataFrame Object - Describing and Summarizing

2. Describing and summarizing the data

The next step is to calculate summary statistics, which will help us get to know our data better.

Pandas provides several methods for easily doing so; one such method is describe(), This gives us the 5-number summary, along with the count, mean, and standard deviation of the numeric columns:

	cdi	dmin	felt	gap	mag	...	sig	time	tsunami	tz	updated
count	329.000000	6139.000000	329.000000	6164.000000	9331.000000	...	9332.000000	9.332000e+03	9332.000000	9331.000000	9.332000e+03
mean	2.754711	0.544925	12.310030	121.506588	1.497345	...	56.899914	1.538284e+12	0.006537	-451.990140	1.538537e+12
std	1.010637	2.214305	48.954944	72.962363	1.203347	...	91.872163	6.080306e+08	0.080589	231.752571	6.564135e+08
min	0.000000	0.000648	0.000000	12.000000	-1.260000	...	0.000000	1.537229e+12	0.000000	-720.000000	1.537230e+12
25%	2.000000	0.020425	1.000000	66.142500	0.720000	...	8.000000	1.537793e+12	0.000000	-540.000000	1.537996e+12
50%	2.700000	0.059050	2.000000	105.000000	1.300000	...	26.000000	1.538245e+12	0.000000	-480.000000	1.538621e+12
75%	3.300000	0.177250	5.000000	159.000000	1.900000	...	56.000000	1.538766e+12	0.000000	-480.000000	1.539110e+12
max	8.400000	53.737000	580.000000	355.910000	7.500000	...	2015.000000	1.539475e+12	1.000000	720.000000	1.539537e+12

Inspecting a DataFrame Object - Describing and Summarizing

Tip - If we want different percentiles, we can pass them in with the percentiles argument. For example, if we wanted only the 5th and 95th percentiles, we would run df.describe(percentiles=[0.05, 0.95]). Note we will still get the 50th percentile back because that is the median.

By default, describe() won't give us any information about the columns of type object, but we can either provide include='all' as an argument or run it separately for the data of type np.object:

```
>>> df.describe(include=np.object)
```

Inspecting a DataFrame Object - Describing and Summarizing

When describing non-numeric data, we still get the count of non-null occurrences (count); however, instead of the other summary statistics, we get the number of unique values (unique), the mode (top), and the number of times the mode was observed (freq)

	alert	code	detail	ids	magType	net	place	sources	status	title	type	types	url
count	59	9332	9332	9332	9331	9332	9332	9332	9332	9332	9332	9332	9332
unique	2	9332	9332	9332	10	14	5433	52	2	7807	5	42	9332
top	green	70628507	https://ear...	,pr201827...	ml	ak	10km NE of Aguanga, CA	,ak,	reviewed	M 0.4 - 10km NE of Aguanga, CA	earthquake	,geoserve, origin, phase-data,	https://ear...
freq	58	1	1	1	6803	3166	306	2981	7797	55	9081	5301	1

Important note

The `describe()` method only gives us summary statistics for non-null values. This means that, if we had 100 rows and half of our data was null, then the average would be calculated as the sum of the 50 non-null rows divided by 50.

Inspecting a DataFrame Object - Describing and Summarizing

What is we just wanting a particular statistic?, either for a specific column or for all the columns. The following table includes methods that will work for both Series and DataFrame objects:

Method	Description	Data types
`count()`	The number of non-null observations	Any
`nunique()`	The number of unique values	Any
`sum()`	The total of the values	Numerical or Boolean
`mean()`	The average of the values	Numerical or Boolean
`median()`	The median of the values	Numerical
`min()`	The minimum of the values	Numerical
`idxmin()`	The index where the minimum values occurs	Numerical
`max()`	The maximum of the values	Numerical
`idxmax()`	The index where the maximum value occurs	Numerical

Inspecting a DataFrame Object - Describing and Summarizing

What is we just wanting a particular statistic?, either for a specific column or for all the columns. The following table includes methods that will work for both Series and DataFrame objects:

`abs()`	The absolute values of the data	Numerical
`std()`	The standard deviation	Numerical
`var()`	The variance	Numerical
`cov()`	The covariance between two `Series`, or a covariance matrix for all column combinations in a `DataFrame`	Numerical
`corr()`	The correlation between two `Series`, or a correlation matrix for all column combinations in a `DataFrame`	Numerical
`quantile()`	Calculates a specific quantile	Numerical
`cumsum()`	The cumulative sum	Numerical or Boolean
`cummin()`	The cumulative minimum	Numerical
`cummax()`	The cumulative maximum	Numerical

Inspecting a DataFrame Object - Describing and Summarizing

With Series objects, we have some additional methods for describing our data:

- unique(): Returns the distinct values of the column.
- value_counts(): Returns a frequency table of the number of times each unique value in each column appears, or, alternatively, the percentage of times each unique value appears when passed normalize=True.
- mode(): Returns the most common value of the column.

```
>>> df.alert.unique()
array([nan, 'green', 'red'], dtype=object)
```

```
>>> df.alert.value_counts()
green    58
red       1
Name: alert, dtype: int64
```

Inspecting a DataFrame Object - Describing and Summarizing

Note that Index objects also have several methods that can help us describe and summarize our data:

Method	Description
`argmax()` / `argmin()`	Find the location of the maximum/minimum value in the index
`equals()`	Compare the index to another `Index` object for equality
`isin()`	Check if the index values are in a list of values and return an array of Booleans
`max()` / `min()`	Find the maximum/minimum value in the index
`nunique()`	Get the number of unique values in the index
`to_series()`	Create a `Series` object from the index
`unique()`	Find the unique values of the index
`value_counts()`	Create a frequency table for the unique values in the index

Grabbing Subsets of the Data

We will often be interested in performing operations and/or analyses on subsets of our data. There are many types of subsets we may look to isolate from our data, such as selecting only specific columns or rows as a whole or when a specific criterion is met.

Grabbing Subsets of the Data - Selecting Columns

1. Selecting columns

Remember that a column is a Series object, so, for example, selecting the mag column in the earthquake data gives us the magnitudes of the earthquakes as a Series object:

```
>>> df.mag                          >>> df['mag']
0       1.35                        0       1.35
1       1.29                        1       1.29
2       3.42                        2       3.42
3       0.44                        3       0.44
4       2.16                        4       2.16
        ...                                 ...
9327    0.62                        9327    0.62
9328    1.00                        9328    1.00
9329    2.40                        9329    2.40
9330    1.10                        9330    1.10
9331    0.66                        9331    0.66
Name: mag, Length: 9332,            Name: mag, Length: 9332, dtype: float64
```

Grabbing Subsets of the Data - Selecting Columns

Note that we aren't limited to selecting one column at a time. By passing a list to the dictionary lookup, we can select many columns, giving us a DataFrame object that is a subset of our original dataframe:

```
>>> df[['mag', 'title']]
```

This gives us the full mag and title columns from the original dataframe:

	mag	title
0	1.35	M 1.4 - 9km NE of Aguanga, CA
1	1.29	M 1.3 - 9km NE of Aguanga, CA
2	3.42	M 3.4 - 8km NE of Aguanga, CA
3	0.44	M 0.4 - 9km NE of Aguanga, CA
4	2.16	M 2.2 - 10km NW of Avenal, CA
...	...	...
9327	0.62	M 0.6 - 9km ENE of Mammoth Lakes, CA
9328	1.00	M 1.0 - 3km W of Julian, CA
9329	2.40	M 2.4 - 35km NNE of Hatillo, Puerto Rico
9330	1.10	M 1.1 - 9km NE of Aguanga, CA
9331	0.66	M 0.7 - 9km NE of Aguanga, CA

Grabbing Subsets of the Data - Selecting Columns

String methods are a very powerful way to select columns. For example, if we wanted to select all the columns that start with mag, along with the title and time columns, we would do the following:

```
>>> df[
...     ['title', 'time']
...     + [col for col in df.columns if col.startswith('mag')]
... ]
```

	title	time	mag	magType
0	M 1.4 - 9km NE of Aguanga, CA	1539475168010	1.35	ml
1	M 1.3 - 9km NE of Aguanga, CA	1539475129610	1.29	ml
2	M 3.4 - 8km NE of Aguanga, CA	1539475062610	3.42	ml
3	M 0.4 - 9km NE of Aguanga, CA	1539474978070	0.44	ml
4	M 2.2 - 10km NW of Avenal, CA	1539474716050	2.16	md
...	...	...	...	...
9327	M 0.6 - 9km ENE of Mammoth Lakes, CA	1537230228060	0.62	md
9328	M 1.0 - 3km W of Julian, CA	1537230135130	1.00	ml
9329	M 2.4 - 35km NNE of Hatillo, Puerto Rico	1537229908180	2.40	md
9330	M 1.1 - 9km NE of Aguanga, CA	1537229545350	1.10	ml
9331	M 0.7 - 9km NE of Aguanga, CA	1537228864470	0.66	ml

Grabbing Subsets of the Data - Selecting Columns

> **Tip**
>
> We can also select columns using the `get()` method. This has the benefits of not raising an error if the column doesn't exist and allowing us to provide a backup value—the default is `None`. For example, if we call `df.get('event', False)`, it will return `False` since we don't have an `event` column.

> **Tip**
>
> A complete list of string methods can be found in the Python 3 documentation at `https://docs.python.org/3/library/stdtypes.html#string-methods`.

Grabbing Subsets of the Data - Slicing

2. Slicing

When we want to extract certain rows (slices) from our dataframe, we use slicing. DataFrame slicing works similarly to slicing with other Python objects, such as lists and tuples, with the first index being inclusive and the last index being exclusive:

```
>>> df[100:103]
```

When specifying a slice of 100:103, we get back rows 100, 101, and 102:

	alert	...	dmin	felt	...	mag	magType	...	place	...	time	title	tsunami	...	updated	url
100	NaN	...	NaN	NaN	...	1.20	ml	...	25km NW of Ester, Alaska	...	1539435449480	M 1.2 - 25km NW of Ester, Alaska	0	...	1539443551010	https...
101	NaN	...	0.01355	NaN	...	0.59	md	...	8km ESE of Mammoth Lakes, CA	...	1539435391320	M 0.6 - 8km ESE of Mammoth Lakes, CA	0	...	1539439802162	https...
102	NaN	...	0.02987	NaN	...	1.33	ml	...	8km ENE of Aguanga, CA	...	1539435293090	M 1.3 - 8km ENE of Aguanga, CA	0	...	1539435940470	https...

Grabbing Subsets of the Data - Slicing

We can combine our row and column selections by using what is known as chaining:

```
>>> df[['title', 'time']][100:103]
```

First, we selected the title and time columns for all the rows, and then we pulled out rows with indices 100, 101, and 102:

	title	time
100	M 1.2 - 25km NW of Ester, Alaska	1539435449480
101	M 0.6 - 8km ESE of Mammoth Lakes, CA	1539435391320
102	M 1.3 - 8km ENE of Aguanga, CA	1539435293090

Grabbing Subsets of the Data - Slicing

If we decide to use chaining to update the values in our data, we will find pandas complaining that we aren't doing so correctly (even if it works).

```
>>> df[110:113]['title'] = df[110:113]['title'].str.lower()
/.../book_env/lib/python3.7/[...]:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a
DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
```

It's not enough to know selection and slicing—we must also master indexing.

Since this is just a warning, our values have been updated, but this may not always be the case:

```
>>> df[110:113]['title']
110                m 1.1 - 35km s of ester, alaska
111    m 1.9 - 93km wnw of arctic village, alaska
112      m 0.9 - 20km wsw of smith valley, nevada
Name: title, dtype: object
```

Grabbing Subsets of the Data - Indexing

3. Indexing

Pandas indexing operations provide us with a one-method way to select both the rows and the columns we want. We can use loc[] and iloc[] to subset our dataframe using label-based or integer-based lookups, respectively.

```
df.loc[row_indexer, column_indexer]
```

Note that by using loc[], as indicated in the warning message, we no longer trigger any warnings from pandas for this operation. We also changed the end index from 113 to 112 because loc[] is inclusive of endpoints:

```
>>> df.loc[110:112, 'title'] = \
...     df.loc[110:112, 'title'].str.lower()
>>> df.loc[110:112, 'title']
110                m 1.1 - 35km s of ester, alaska
111    m 1.9 - 93km wnw of arctic village, alaska
112      m 0.9 - 20km wsw of smith valley, nevada
Name: title, dtype: object
```

Grabbing Subsets of the Data - Indexing

We can select multiple rows and columns at the same time with loc[]:

```
>>> df.loc[10:15, ['title', 'mag']]
```

This leaves us with rows 10 through 15 for the title and mag columns only:

	title	mag
10	M 0.5 - 10km NE of Aguanga, CA	0.50
11	M 2.8 - 53km SE of Punta Cana, Dominican Republic	2.77
12	M 0.5 - 9km NE of Aguanga, CA	0.50
13	M 4.5 - 120km SSW of Banda Aceh, Indonesia	4.50
14	M 2.1 - 14km NW of Parkfield, CA	2.13
15	M 2.0 - 156km WNW of Haines Junction, Canada	2.00

Grabbing Subsets of the Data - Indexing

As we have seen, when using loc[], our end index is inclusive. This isn't the case with iloc[]:

```
>>> df.iloc[10:15, [19, 8]]
```

Using iloc[], we lost the row at index 15:

	title	mag
10	M 0.5 - 10km NE of Aguanga, CA	0.50
11	M 2.8 - 53km SE of Punta Cana, Dominican Republic	2.77
12	M 0.5 - 9km NE of Aguanga, CA	0.50
13	M 4.5 - 120km SSW of Banda Aceh, Indonesia	4.50
14	M 2.1 - 14km NW of Parkfield, CA	2.13

Grabbing Subsets of the Data - Indexing

We aren't limited to using the slicing syntax for the rows, though; columns work as well:

```
>>> df.iloc[10:15, 6:10]
```

By using slicing, we can easily grab adjacent rows and columns:

	gap	ids	mag	magType
10	57.0	,ci37389162,	0.50	ml
11	186.0	,pr2018286010,	2.77	md
12	76.0	,ci37389146,	0.50	ml
13	157.0	,us1000hbti,	4.50	mb
14	71.0	,nc73096921,	2.13	md

Grabbing Subsets of the Data - Indexing

To look up scalar values, we use at[] and iat[], which are faster:

```
>>> df.at[10, 'mag']
0.5
```

The magnitude column has a column index of 8; therefore, we can also look up the magnitude with iat[]:

```
>>> df.iat[10, 8]
0.5
```

Grabbing Subsets of the Data - Filtering

4. Filtering

Pandas gives us a few options for filtering our data, including Boolean masks and some special methods.

```
>>> df.mag > 2
0       False
1       False
2        True
3       False
        ...
9328    False
9329     True
9330    False
9331    False
Name: mag, Length: 9332, dtype: bool
```

Grabbing Subsets of the Data - Filtering

Running this on the entire dataframe isn't useful for our earthquake data with various data types. Instead, we can use this strategy to get the subset where the earthquake magnitude is 7.0 or greater:

```
>>> df[df.mag >= 7.0]
```

Our resulting dataframe has just two rows:

	alert	...	dmin	felt	...	mag	magType	...	place	...	time	title	tsunami	...	updated	url
837	green	...	1.763	3.0	...	7.0	mww	...	117km E of Kimbe, Papua New Guinea	...	1539204500290	M 7.0 - 117km E of Kimbe, Papua New Guinea	1	...	1539378744253	https...
5263	red	...	1.589	18.0	...	7.5	mww	...	78km N of Palu, Indonesia	...	1538128963480	M 7.5 - 78km N of Palu, Indonesia	1	...	1539123134531	https...

Grabbing Subsets of the Data - Filtering

We got back a lot of columns we didn't need, though. We could have chained a column selection to the end of the last code snippet; however, loc[] can handle Boolean masks as well:

```
>>> df.loc[
...     df.mag >= 7.0,
...     ['alert', 'mag', 'magType', 'title', 'tsunami', 'type']
... ]
```

The following dataframe has been filtered so that it only contains relevant columns:

	alert	mag	magType	title	tsunami	type
837	green	7.0	mww	M 7.0 - 117km E of Kimbe, Papua New Guinea	1	earthquake
5263	red	7.5	mww	M 7.5 - 78km N of Palu, Indonesia	1	earthquake

Grabbing Subsets of the Data - Filtering

We aren't limited to just one criterion, either.

```
>>> df.loc[
...     (df.tsunami == 1) & (df.alert == 'red'),
...     ['alert', 'mag', 'magType', 'title', 'tsunami', 'type']
... ]
```

There was only a single earthquake in the data that met our criteria:

	alert	mag	magType	title	tsunami	type
5263	red	7.5	mww	M 7.5 - 78km N of Palu, Indonesia	1	earthquake

Grabbing Subsets of the Data - Filtering

If, instead, we want at least one of our conditions to be true, we can use the bitwise OR operator (|):

```
>>> df.loc[
...     (df.tsunami == 1) | (df.alert == 'red'),
...     ['alert', 'mag', 'magType', 'title', 'tsunami', 'type']
... ]
```

Notice that this filter is much less restrictive since, while both conditions can be true, we only require that one of them is:

	alert	mag	magType	title	tsunami	type
36	NaN	5.0	mww	M 5.0 - 165km NNW of Flying Fish Cove, Christm...	1	earthquake
118	green	6.7	mww	M 6.7 - 262km NW of Ozernovskiy, Russia	1	earthquake
501	green	5.6	mww	M 5.6 - 128km SE of Kimbe, Papua New Guinea	1	earthquake
799	green	6.5	mww	M 6.5 - 148km S of Severo-Kuril'sk, Russia	1	earthquake
816	green	6.2	mww	M 6.2 - 94km SW of Kokopo, Papua New Guinea	1	earthquake
...	...	...	...	...	...	...
8561	NaN	5.4	mb	M 5.4 - 228km S of Taron, Papua New Guinea	1	earthquake
8624	NaN	5.1	mb	M 5.1 - 278km SE of Pondaguitan, Philippines	1	earthquake
9133	green	5.1	ml	M 5.1 - 64km SSW of Kaktovik, Alaska	1	earthquake
9175	NaN	5.2	mb	M 5.2 - 126km N of Dili, East Timor	1	earthquake
9304	NaN	5.1	mb	M 5.1 - 34km NW of Finschhafen, Papua New Guinea	1	earthquake

Grabbing Subsets of the Data - Filtering

In the previous two examples, our conditions involved equality; however, we are by no means limited to this. Let's select all the earthquakes in Alaska where we have a non-null value for the alert column:

```
>>> df.loc[
...     (df.place.str.contains('Alaska'))
...     & (df.alert.notnull()),
...     ['alert', 'mag', 'magType', 'title', 'tsunami', 'type']
... ]
```

This gives the result:

	alert	mag	magType	title	tsunami	type
1015	green	5.0	ml	M 5.0 - 61km SSW of Chignik Lake, Alaska	1	earthquake
1273	green	4.0	ml	M 4.0 - 71km SW of Kaktovik, Alaska	1	earthquake
1795	green	4.0	ml	M 4.0 - 60km WNW of Valdez, Alaska	1	earthquake
2752	green	4.0	ml	M 4.0 - 67km SSW of Kaktovik, Alaska	1	earthquake
3260	green	3.9	ml	M 3.9 - 44km N of North Nenana, Alaska	0	earthquake
4101	green	4.2	ml	M 4.2 - 131km NNW of Arctic Village, Alaska	0	earthquake
6897	green	3.8	ml	M 3.8 - 80km SSW of Kaktovik, Alaska	0	earthquake
8524	green	3.8	ml	M 3.8 - 69km SSW of Kaktovik, Alaska	0	earthquake
9133	green	5.1	ml	M 5.1 - 64km SSW of Kaktovik, Alaska	1	earthquake

Grabbing Subsets of the Data - Filtering

Note that we aren't limited to checking if each row contains text; we can use regular expressions as well.

```
>>> df.loc[
...     (df.place.str.contains(r'CA|California$'))
...     & (df.mag > 3.8),
...     ['alert', 'mag', 'magType', 'title', 'tsunami', 'type']
... ]
```

This gives the result:

	alert	mag	magType	title	tsunami	type
1465	green	3.83	mw	M 3.8 - 109km WNW of Trinidad, CA	0	earthquake
2414	green	3.83	mw	M 3.8 - 5km SW of Tres Pinos, CA	1	earthquake

Grabbing Subsets of the Data - Filtering

What if we want to get all earthquakes with magnitudes between 6.5 and 7.5?

We can use two Boolean masks—one for magnitudes ≥ 6.5 and another for magnitudes ≤ 7.5—and combine them with the & operator.

Thankfully, pandas makes this type of mask much easier to create by providing us with the between() method:

```
>>> df.loc[
...     df.mag.between(6.5, 7.5),
...     ['alert', 'mag', 'magType', 'title', 'tsunami', 'type']
... ]
```

	alert	mag	magType	title	tsunami	type
118	green	6.7	mww	M 6.7 - 262km NW of Ozernovskiy, Russia	1	earthquake
799	green	6.5	mww	M 6.5 - 148km S of Severo-Kuril'sk, Russia	1	earthquake
837	green	7.0	mww	M 7.0 - 117km E of Kimbe, Papua New Guinea	1	earthquake
4363	green	6.7	mww	M 6.7 - 263km NNE of Ndoi Island, Fiji	1	earthquake
5263	red	7.5	mww	M 7.5 - 78km N of Palu, Indonesia	1	earthquake

Grabbing Subsets of the Data - Filtering

We can use the isin() method to create a Boolean mask for values that match one of a list of values.

```
>>> df.loc[
...     df.magType.isin(['mw', 'mwb']),
...     ['alert', 'mag', 'magType', 'title', 'tsunami', 'type']
... ]
```

	alert	mag	magType	title	tsunami	type
995	NaN	3.35	mw	M 3.4 - 9km WNW of Cobb, CA	0	earthquake
1465	green	3.83	mw	M 3.8 - 109km WNW of Trinidad, CA	0	earthquake
2414	green	3.83	mw	M 3.8 - 5km SW of Tres Pinos, CA	1	earthquake
4988	green	4.41	mw	M 4.4 - 1km SE of Delta, B.C., MX	1	earthquake
6307	green	5.80	mwb	M 5.8 - 297km NNE of Ndoi Island, Fiji	0	earthquake
8257	green	5.70	mwb	M 5.7 - 175km SSE of Lambasa, Fiji	0	earthquake

Grabbing Subsets of the Data - Filtering

So far, we have been filtering on specific values, but suppose we wanted to see all the data for the lowest-magnitude and highest-magnitude earthquakes.

```
>>> df.loc[
...     [df.mag.idxmin(), df.mag.idxmax()],
...     ['alert', 'mag', 'magType', 'title', 'tsunami', 'type']
... ]
```

	alert	mag	magType	title	tsunami	type
2409	NaN	-1.26	ml	M -1.3 - 41km ENE of Adak, Alaska	0	earthquake
5263	red	7.50	mww	M 7.5 - 78km N of Palu, Indonesia	1	earthquake

Exercises

Using the data/parsed.csv file and the material from this chapter, complete the following exercises to practice your pandas skills:

1. Find the 95th percentile of earthquake magnitude in Japan using the mb magnitude type.

2. Find the percentage of earthquakes in Indonesia that were coupled with tsunamis.

3. Calculate summary statistics for earthquakes in Nevada.

4. Calculate the number of earthquakes in the Ring of Fire locations and the number outside of them.

5. Find the tsunami count along the Ring of Fire.

Adding and Removing Data

Before adding or removing data, note that most methods return a new DataFrame, but some change the data in place. If we want to avoid changing the original data, we should copy the dataframe before making modifications:

```
df_to_modify = df.copy()
```

We will once again be working with the earthquake data, but this time, we will only read in a subset of the columns:

```
>>> import pandas as pd

>>> df = pd.read_csv(
...     'data/earthquakes.csv',
...     usecols=[
...         'time', 'title', 'place', 'magType',
...         'mag', 'alert', 'tsunami'
...     ]
... )
```

Creating New Data

Creating new columns can be achieved in the same fashion as variable assignment.

```
>>> df['source'] = 'USGS API'
>>> df.head()
```

The new column is created to the right of the original columns, with a value of USGS API for every row:

	alert	mag	magType	place	time	title	tsunami	source
0	NaN	1.35	ml	9km NE of Aguanga, CA	1539475168010	M 1.4 - 9km NE of Aguanga, CA	0	USGS API
1	NaN	1.29	ml	9km NE of Aguanga, CA	1539475129610	M 1.3 - 9km NE of Aguanga, CA	0	USGS API
2	NaN	3.42	ml	8km NE of Aguanga, CA	1539475062610	M 3.4 - 8km NE of Aguanga, CA	0	USGS API
3	NaN	0.44	ml	9km NE of Aguanga, CA	1539474978070	M 0.4 - 9km NE of Aguanga, CA	0	USGS API
4	NaN	2.16	md	10km NW of Avenal, CA	1539474716050	M 2.2 - 10km NW of Avenal, CA	0	USGS API

Important note

We cannot create the column with attribute notation (`df.source`) because the dataframe doesn't have that attribute yet, so we must use dictionary notation (`df['source']`).

Creating New Data

We aren't limited to broadcasting one value to the entire column; we can have the column hold the result of Boolean logic or a mathematical equation.

```
>>> df['mag_negative'] = df.mag < 0
>>> df.head()
```

Note that the new column has been added to the right:

	alert	mag	magType	place	time	title	tsunami	source	mag_negative
0	NaN	1.35	ml	9km NE of Aguanga, CA	1539475168010	M 1.4 - 9km NE of Aguanga, CA	0	USGS API	False
1	NaN	1.29	ml	9km NE of Aguanga, CA	1539475129610	M 1.3 - 9km NE of Aguanga, CA	0	USGS API	False
2	NaN	3.42	ml	8km NE of Aguanga, CA	1539475062610	M 3.4 - 8km NE of Aguanga, CA	0	USGS API	False
3	NaN	0.44	ml	9km NE of Aguanga, CA	1539474978070	M 0.4 - 9km NE of Aguanga, CA	0	USGS API	False
4	NaN	2.16	md	10km NW of Avenal, CA	1539474716050	M 2.2 - 10km NW of Avenal, CA	0	USGS API	False

Creating New Data

...

Q&A

Questions and answers

Thanks!