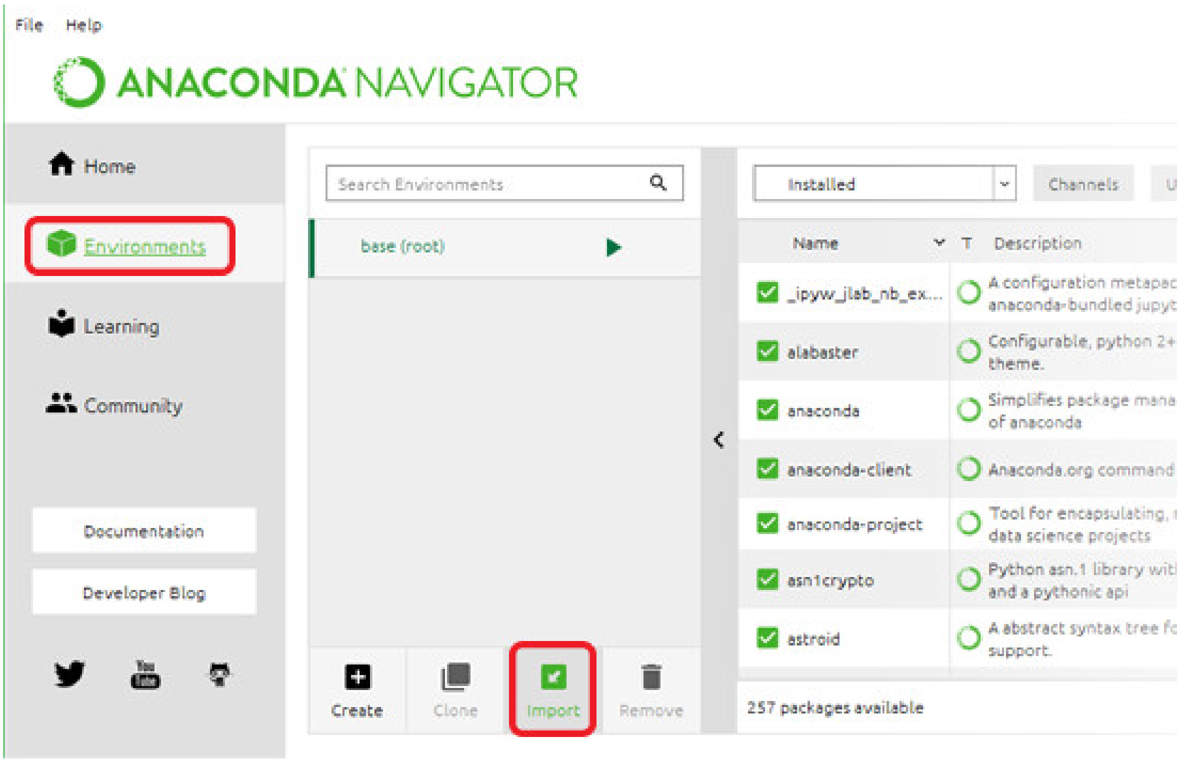
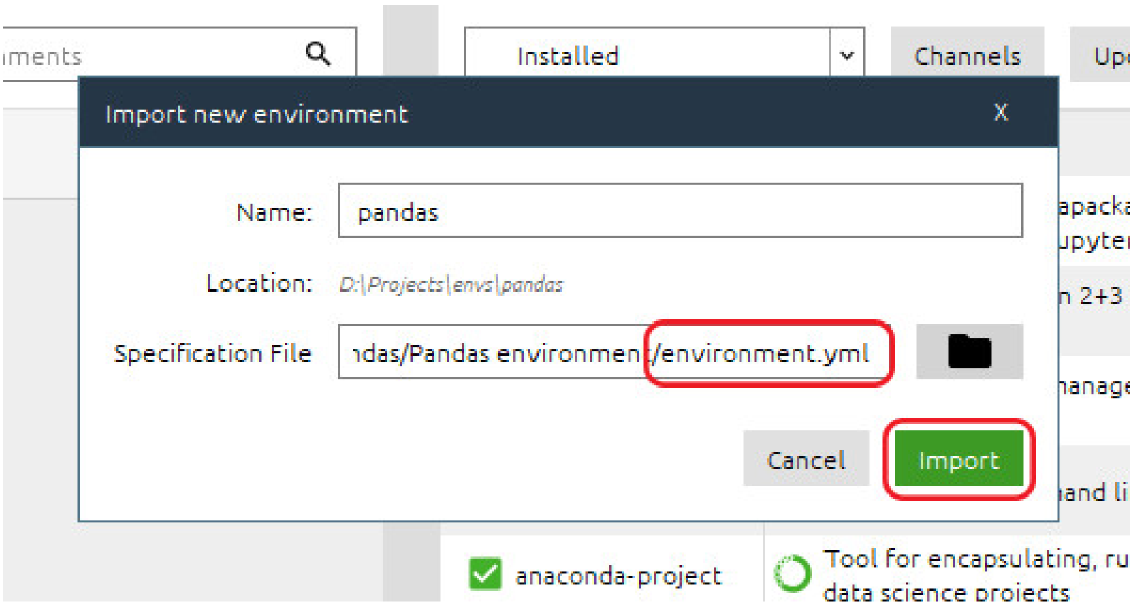
In this lesson we'll learn about DataFrames (the data structure Pandas works with) and learning how to access data in the DataFrame using Pandas.

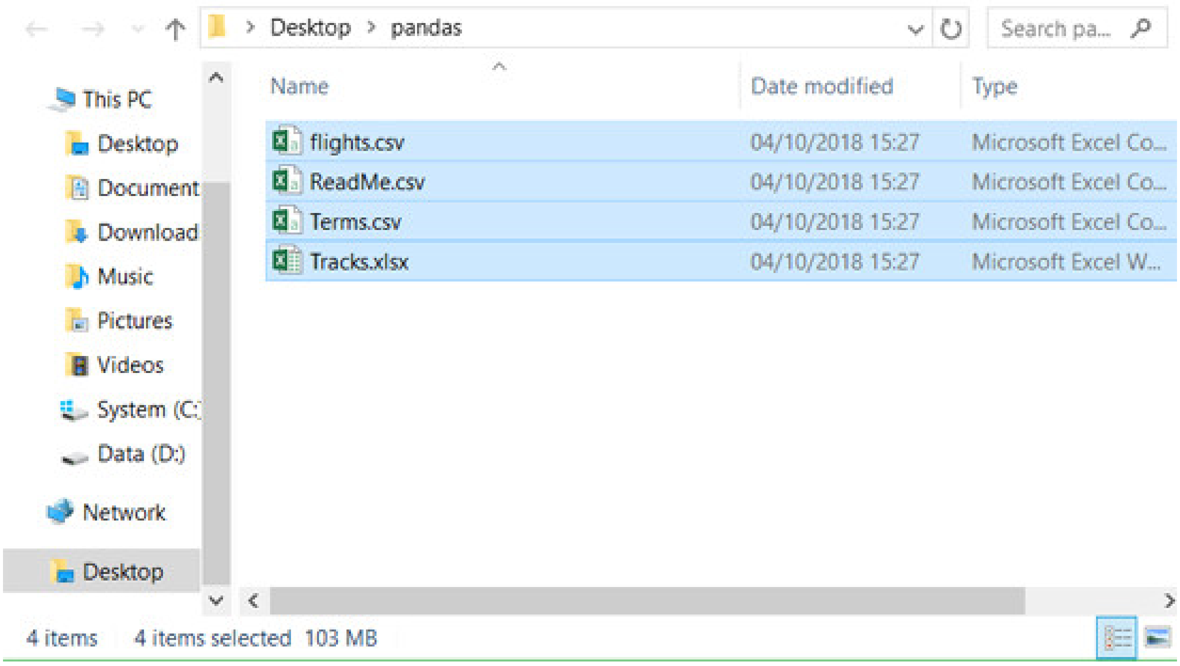
Download the files for this lesson and unzip them both.

Open Anaconda Navigator and load the **environment** file (**note that it uses Pandas version 0.23.4**):



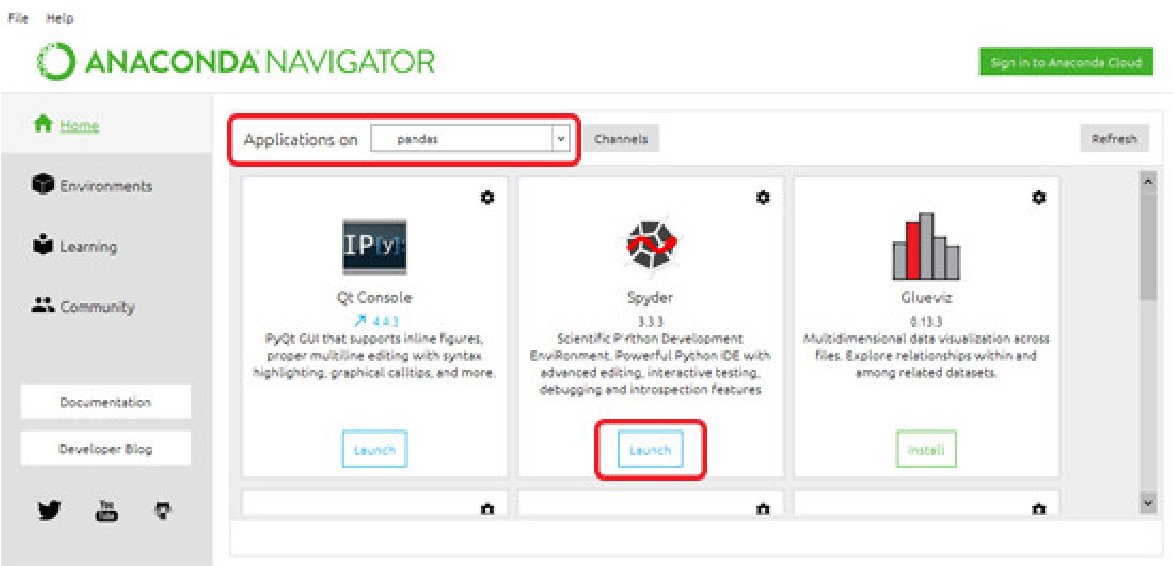


Now we need to copy the files we'll be working with into your working directory – maybe create a folder specifically for this course. The files we need are **flights.csv**, **ReadMe.csv**, **Terms.csv** and **Tracks.xlsx**.

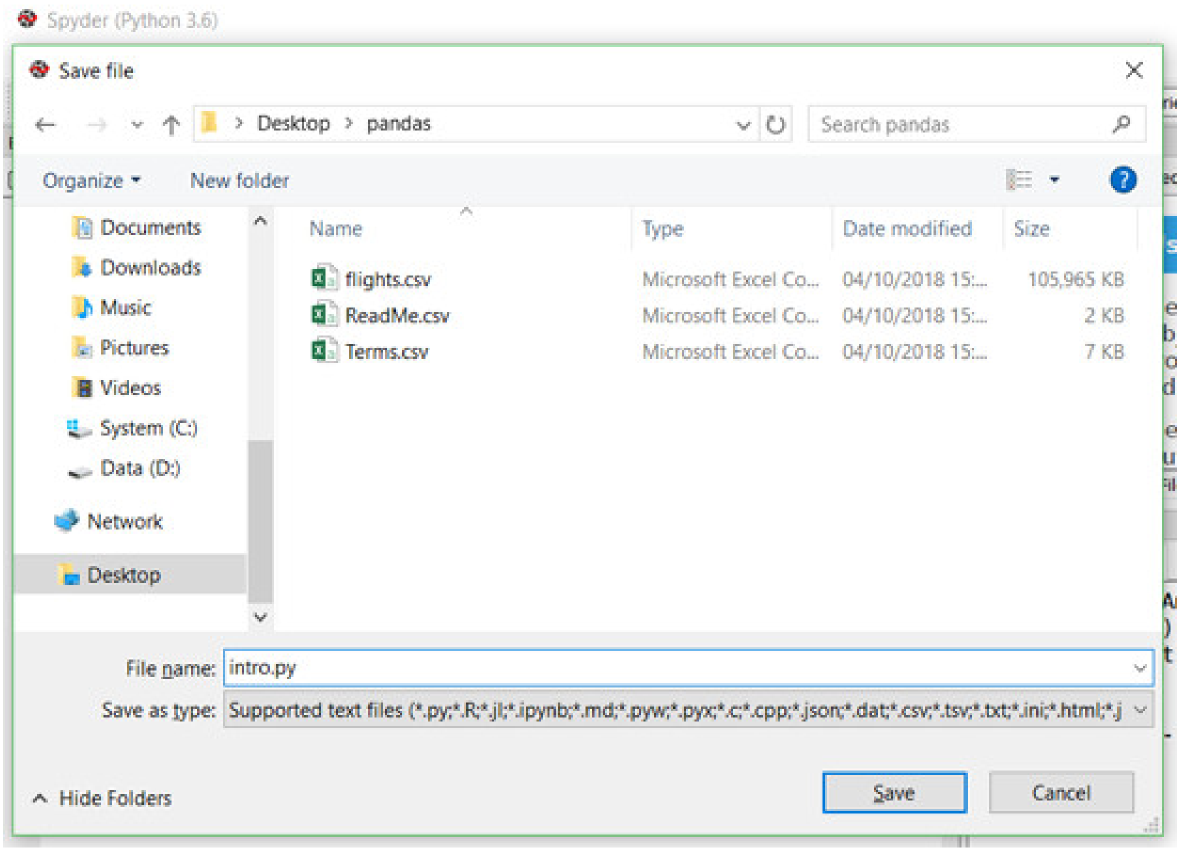


Now we can get started with Pandas!

In **Anaconda Navigator** make sure you have the **pandas environment** selected before launching **Spyder**.



Save your new file in the same directory as the spreadsheets so they'll be easy to access.



Now we'll start by importing Pandas, and then Numpy, which we'll use to populate data.

```
import pandas as pd
import numpy as np
```

Creating a DataFrame:

A DataFrame is a like a spreadsheet – a 2D table with rows and columns, very similar to a spreadsheet you'd open in Excel. Using Pandas gives us a lot more power behind how we can work with them.

Let's start by creating a DataFrame from data we have.

Remembering it's a 2D table, a way we can define a DataFrame is to make a **dictionary** first, then give that to Pandas to convert into a DataFrame. So let's make a Python dictionary. Usually we use **df** to mean **DataFrame**:

```
df_data = {

}
```

Within this dictionary, each **key** will be a **column**, with their **values** being the **row data** in that column.

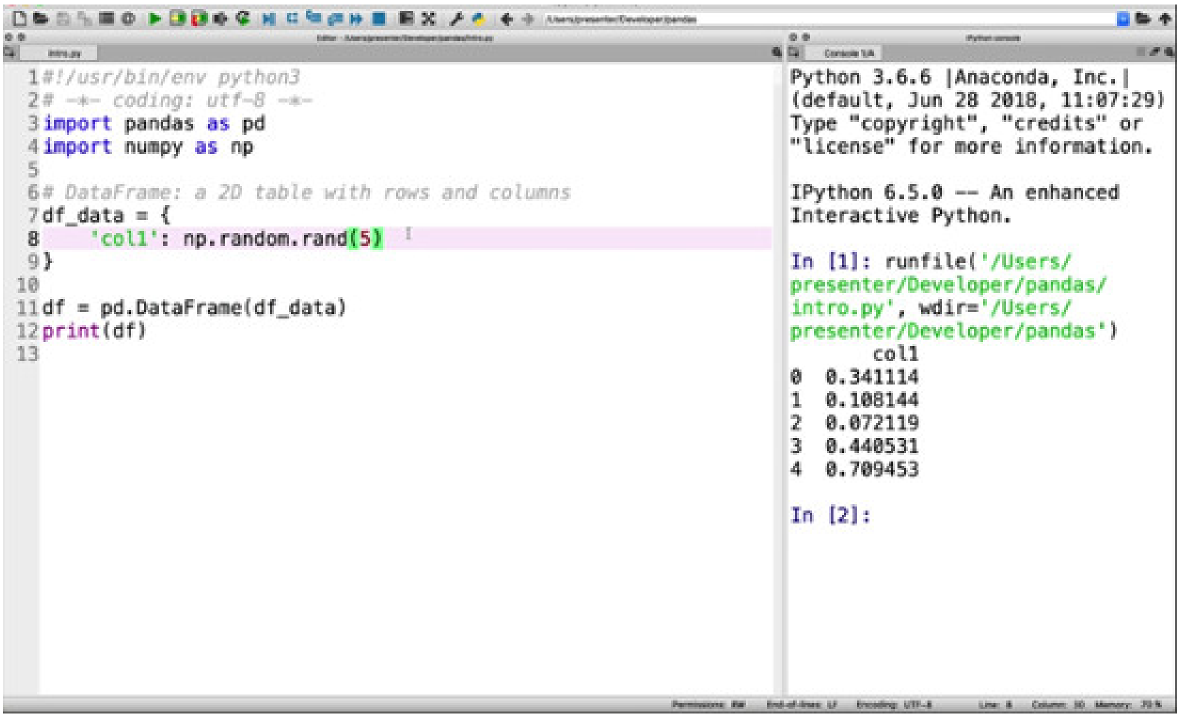
To see how this looks, let's populate a column with some random values using Numpy.

```
df_data = {
'col1': np.random.rand(5)
}
```

This will generate a single column with 5 rows. Let’s convert it to a DataFrame to see what it looks like before we add more columns.

```
df = pd.DataFrame(df_data)
print(df)
```

If you **run** this, you’ll see our DataFrame printed to the console populated with 5 random values thanks to Numpy.



Let’s go ahead and make another couple of columns and **run** it again:

```
df_data = {
'col1': np.random.rand(5),
'col2': np.random.rand(5),
'col3': np.random.rand(5)
}
```



```
In [2]: runfile('/Users/presenter/Developer/pandas/intro.py', wdir='/Users/presenter/Developer/pandas')
       col1      col2      col3
0  0.286746  0.588534  0.358781
1  0.217800  0.435274  0.370113
2  0.236691  0.840714  0.467573
3  0.889170  0.100153  0.899585
4  0.920694  0.733788  0.935381

In [3]:
```

Now you can see how we can give data to Pandas in the form of **dictionaries** and how they'll look when converted into **DataFrames**, including how Pandas automatically **indexes** our rows for us.

```
In [2]: runfile('/Users/presenter/Developer/pandas/intro.py', wdir='/Users/presenter/Developer/pandas')
       col1      col2      col3
0  0.286746  0.588534  0.358781
1  0.217800  0.435274  0.370113
2  0.236691  0.840714  0.467573
3  0.889170  0.100153  0.899585
4  0.920694  0.733788  0.935381

In [3]:
```

Fetching Rows:

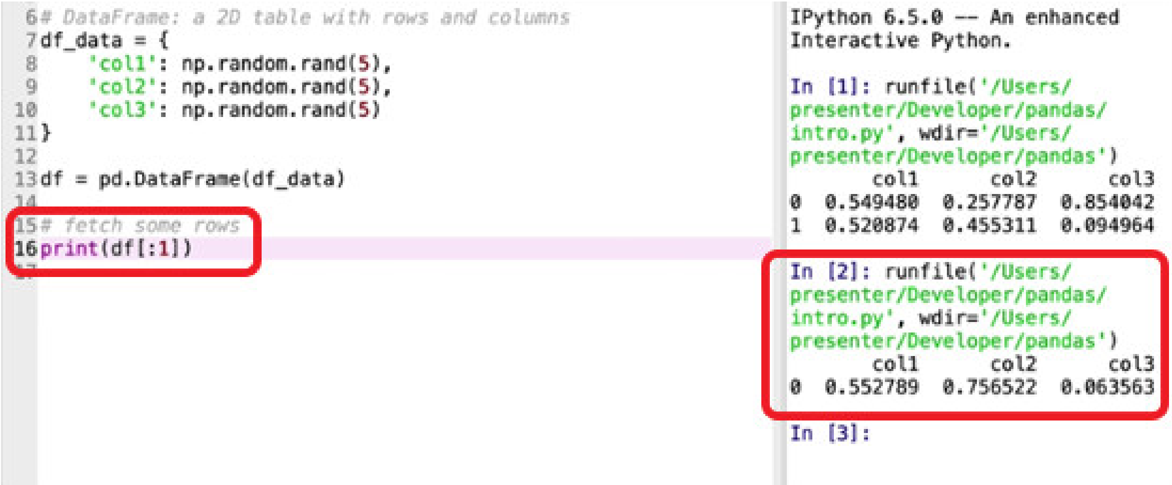
We can deal with it much like you would a list, using **indices** to return the values we want. Try printing only the contents of the first 2 rows.

```
print(df[:2])
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import pandas as pd
import numpy as np

# DataFrame: a 2D table with rows and columns
df_data = {
    'col1': np.random.rand(5),
    'col2': np.random.rand(5),
    'col3': np.random.rand(5)
}

df = pd.DataFrame(df_data)

# fetch some rows
print(df[:2])
```

```
Python 3.6.6 |Anaconda, Inc.| (default, Jun 28 2018, 11:07:29)
Type "copyright", "credits" or "license" for more information.

IPython 6.5.0 -- An enhanced Interactive Python.

In [1]: runfile('/Users/presenter/Developer/pandas/intro.py', wdir='/Users/presenter/Developer/pandas')
       col1      col2      col3
0  0.549480  0.257787  0.854042
1  0.520874  0.455311  0.094964

In [2]:
```

Remember: indices start at 0 and go up to but don't include the final stated index. So if we only wanted the first row, we would use **[:1]**.



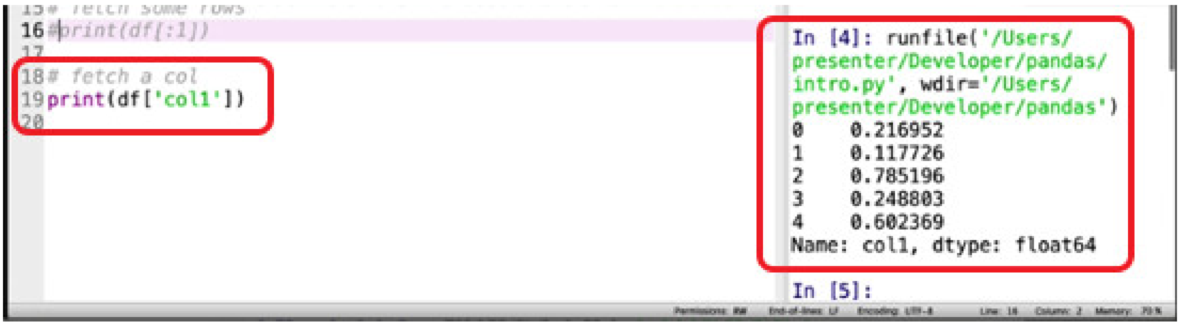
```
# DataFrame: a 2D table with rows and columns
df_data = {
    'col1': np.random.rand(5),
    'col2': np.random.rand(5),
    'col3': np.random.rand(5)
}

df = pd.DataFrame(df_data)

# fetch some rows
print(df[:1])
```

```
IPython 6.5.0 -- An enhanced
Interactive Python.

In [1]: runfile('/Users/
presenter/Developer/pandas/
intro.py', wdir='/Users/
presenter/Developer/pandas')
       col1      col2      col3
0  0.549480  0.257787  0.854042
1  0.520874  0.455311  0.094964

In [2]: runfile('/Users/
presenter/Developer/pandas/
intro.py', wdir='/Users/
presenter/Developer/pandas')
       col1      col2      col3
0  0.552789  0.756522  0.063563

In [3]:
```

Fetching Columns:

This is only slightly different – instead of numbered indices, we refer to the **column's name**. Change your print command to fetch all the data from the first column only.

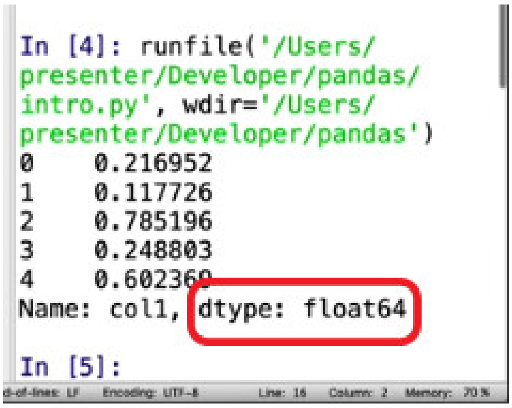
```
print(df['col1'])
```



```
# fetch some rows
#print(df[:1])

# fetch a col
print(df['col1'])
```

```
In [4]: runfile('/Users/
presenter/Developer/pandas/
intro.py', wdir='/Users/
presenter/Developer/pandas')
0    0.216952
1    0.117726
2    0.785196
3    0.248803
4    0.602369
Name: col1, dtype: float64

In [5]:
```

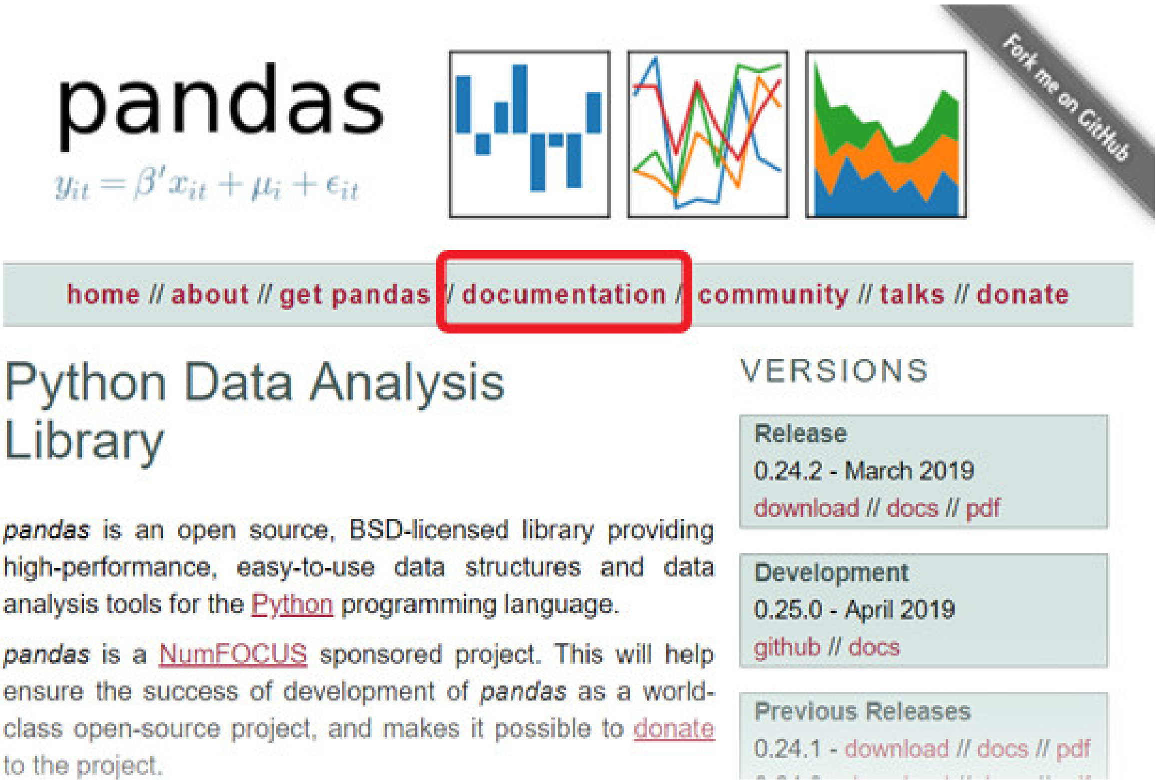
You'll see it also helpfully returns the type of data contained in that column's rows.

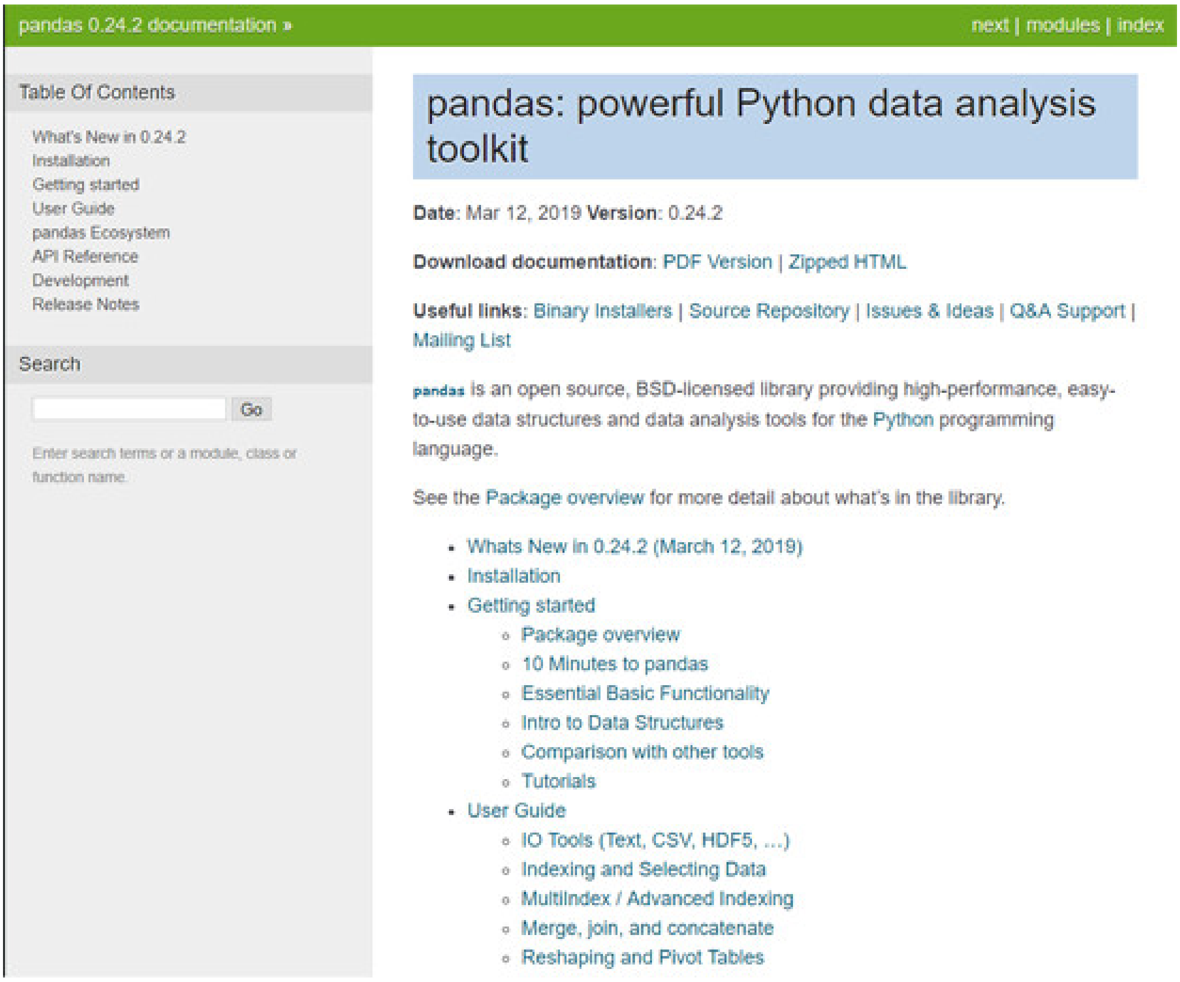


```
In [4]: runfile('/Users/
presenter/Developer/pandas/
intro.py', wdir='/Users/
presenter/Developer/pandas')
0    0.216952
1    0.117726
2    0.785196
3    0.248803
4    0.602369
Name: col1, dtype: float64

In [5]:
```

Documentation:

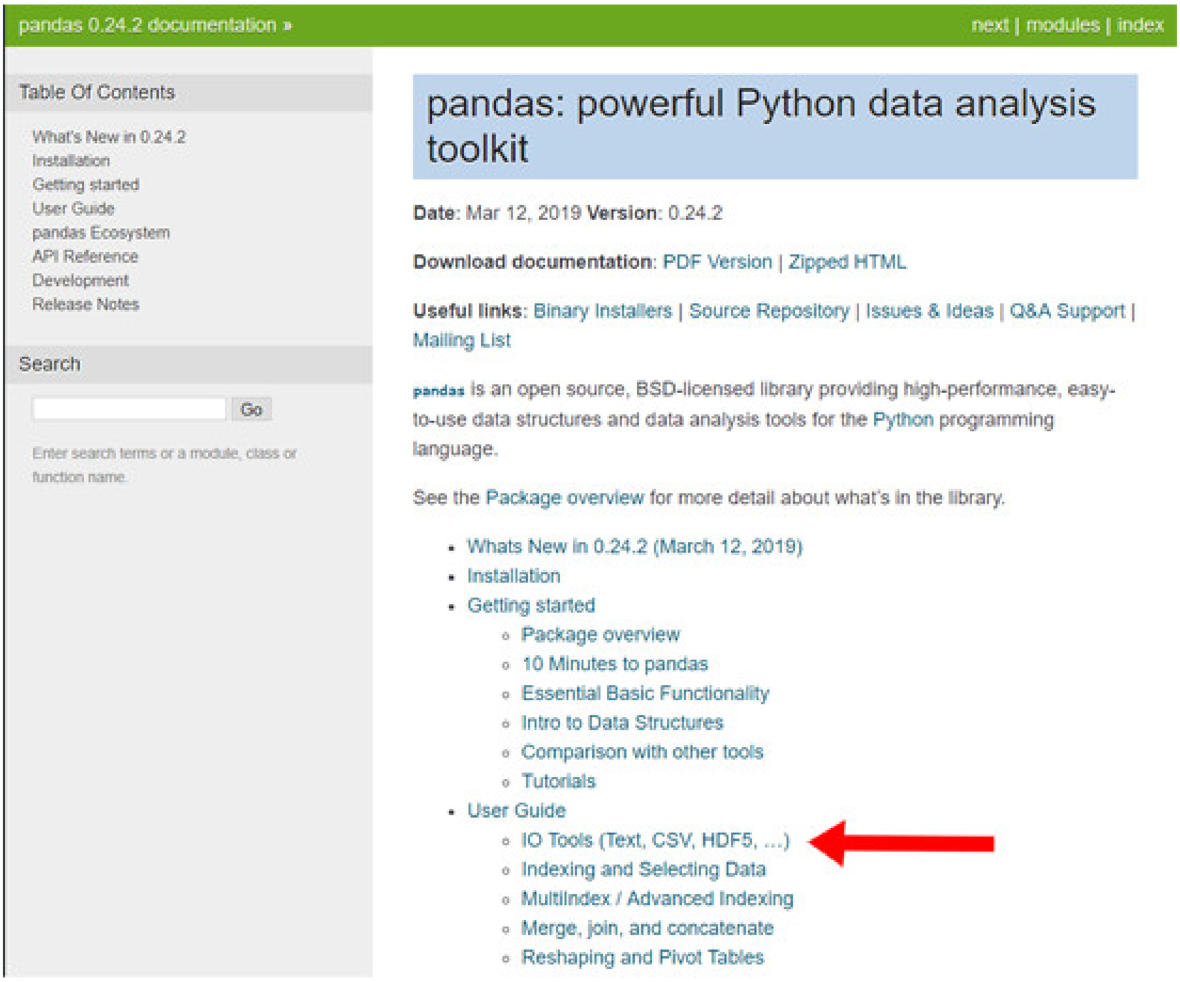
Go to https://pandas.pydata.org/ and click on the **documentation** link.

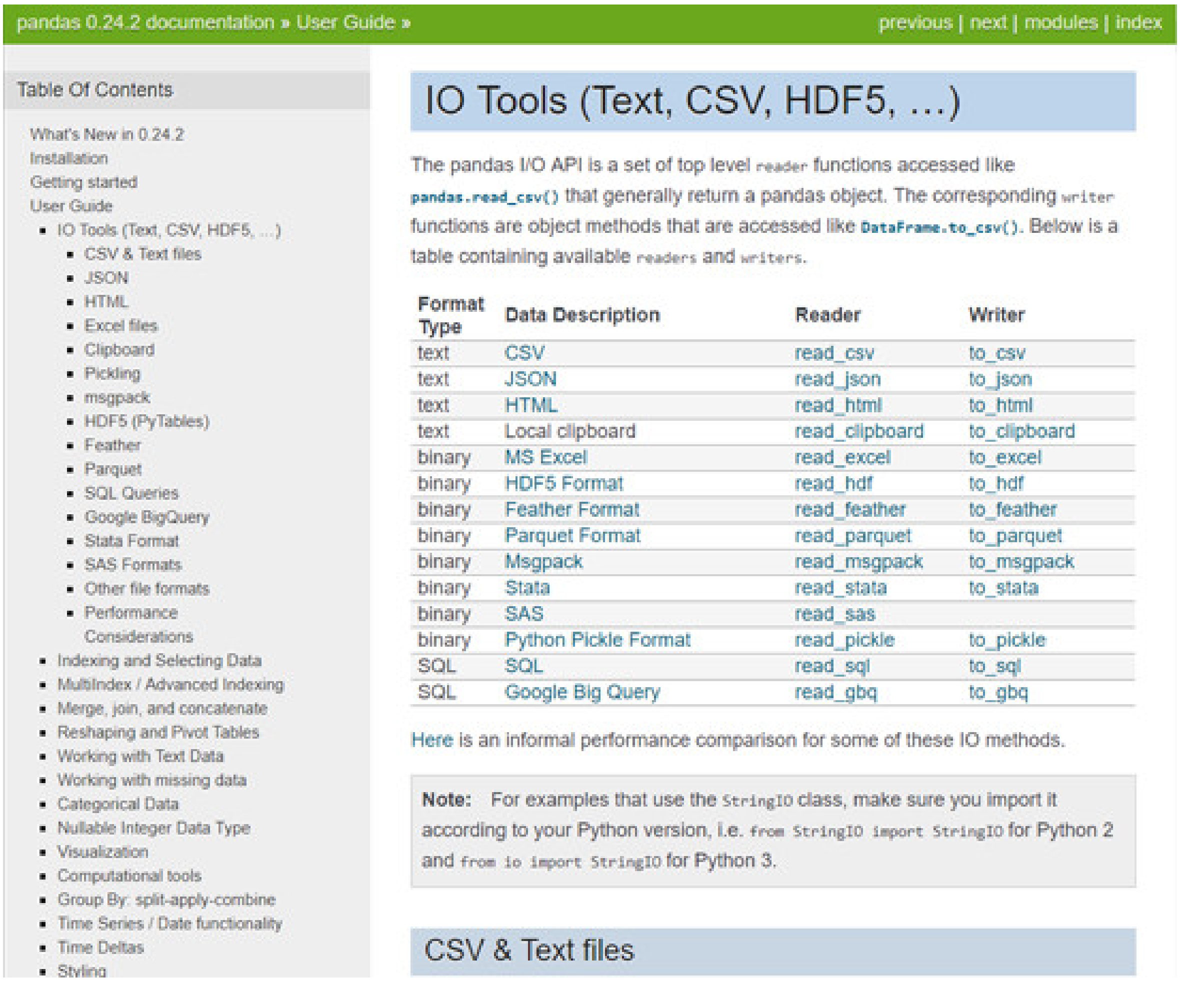




Here we have all the documentation we need to understand and use Pandas. If you’re ever stuck, try here!

Under **User Guide**, click on **IO Tools (Text, CSV, HDF5, …)**, and you'll find all the functions that we can use to read CSV files.





pandas 0.24.2 documentation » User Guide » previous | next | modules | index

Table Of Contents

What's New in 0.24.2
Installation
Getting started
User Guide
- IO Tools (Text, CSV, HDF5, ...)
  - CSV & Text files
  - JSON
  - HTML
  - Excel files
  - Clipboard
  - Pickling
  - msgpack
  - HDF5 (PyTables)
  - Feather
  - Parquet
  - SQL Queries
  - Google BigQuery
  - Stata Format
  - SAS Formats
  - Other file formats
  - Performance Considerations
- Indexing and Selecting Data
- MultiIndex / Advanced Indexing
- Merge, join, and concatenate
- Reshaping and Pivot Tables
- Working with Text Data
- Working with missing data
- Categorical Data
- Nullable Integer Data Type
- Visualization
- Computational tools
- Group By: split-apply-combine
- Time Series / Date functionality
- Time Deltas
- Styling

IO Tools (Text, CSV, HDF5, …)

The pandas I/O API is a set of top level `reader` functions accessed like `pandas.read_csv()` that generally return a pandas object. The corresponding `writer` functions are object methods that are accessed like `DataFrame.to_csv()`. Below is a table containing available `readers` and `writers`.

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	Msgpack	read_msgpack	to_msgpack
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google Big Query	read_gbq	to_gbq

Here is an informal performance comparison for some of these IO methods.

Note: For examples that use the `StringIO` class, make sure you import it according to your Python version, i.e. `from StringIO import StringIO` for Python 2 and `from io import StringIO` for Python 3.

CSV & Text files

Now if you click on any function, say, **pandas.read_csv()**…

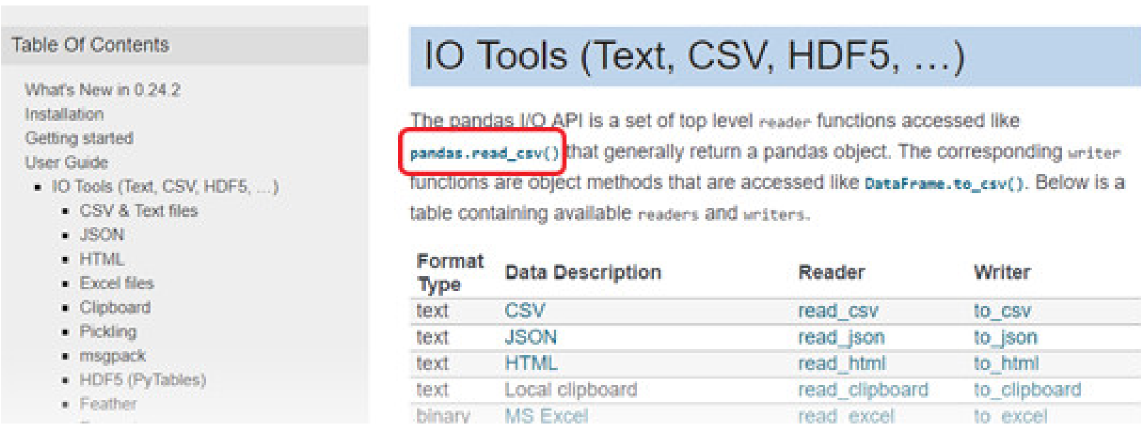


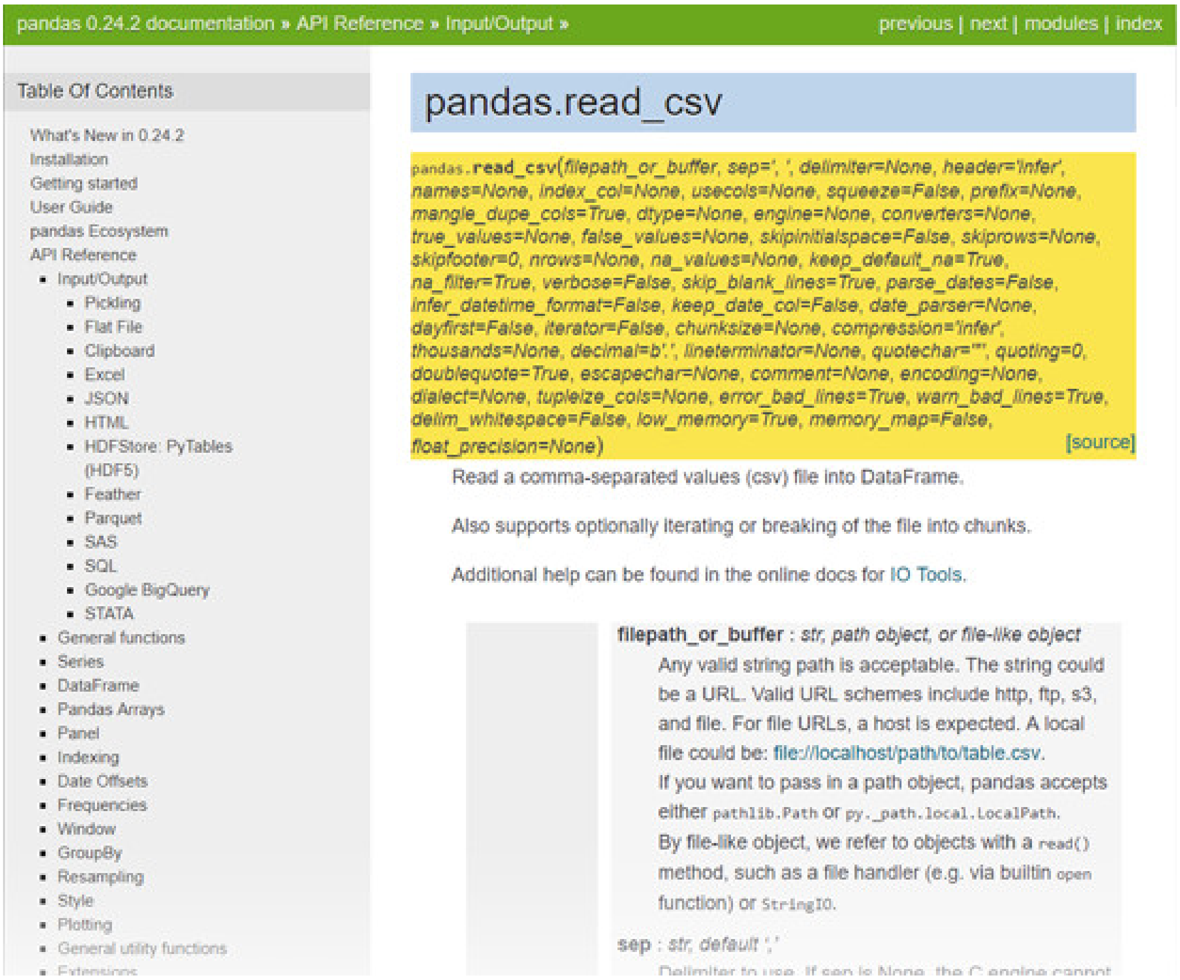
Table Of Contents

What's New in 0.24.2
Installation
Getting started
User Guide
- IO Tools (Text, CSV, HDF5, ...)
  - CSV & Text files
  - JSON
  - HTML
  - Excel files
  - Clipboard
  - Pickling
  - msgpack
  - HDF5 (PyTables)
  - Feather

IO Tools (Text, CSV, HDF5, …)

The pandas I/O API is a set of top level `reader` functions accessed like `pandas.read_csv()` that generally return a pandas object. The corresponding `writer` functions are object methods that are accessed like `DataFrame.to_csv()`. Below is a table containing available `readers` and `writers`.

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel



pandas 0.24.2 documentation » API Reference » Input/Output » previous | next | modules | index

Table Of Contents

What's New in 0.24.2
Installation
Getting started
User Guide
pandas Ecosystem
API Reference
- Input/Output
  - Pickling
  - Flat File
  - Clipboard
  - Excel
  - JSON
  - HTML
  - HDFStore: PyTables (HDF5)
  - Feather
  - Parquet
  - SAS
  - SQL
  - Google BigQuery
  - STATA
- General functions
- Series
- DataFrame
- Pandas Arrays
- Panel
- Indexing
- Date Offsets
- Frequencies
- Window
- GroupBy
- Resampling
- Style
- Plotting
- General utility functions

pandas.read_csv

pandas.**read_csv**(*filepath_or_buffer, sep=',', delimiter=None, header='infer', names=None, index_col=None, usecols=None, squeeze=False, prefix=None, mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None, skipfooter=0, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True, parse_dates=False, infer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False, iterator=False, chunksize=None, compression='infer', thousands=None, decimal=b'.', lineterminator=None, quotechar='"', quoting=0, doublequote=True, escapechar=None, comment=None, encoding=None, dialect=None, tupleize_cols=None, error_bad_lines=True, warn_bad_lines=True, delim_whitespace=False, low_memory=True, memory_map=False, float_precision=None*) [source]

Read a comma-separated values (csv) file into DataFrame.

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the online docs for IO Tools.

filepath_or_buffer : *str, path object, or file-like object*
Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv.
If you want to pass in a path object, pandas accepts either `pathlib.Path` or `py._path.local.LocalPath`.
By file-like object, we refer to objects with a `read()` method, such as a file handler (e.g. via builtin `open` function) or `StringIO`.

sep : *str, default ','*

You can see all of its specific documentation, explaining what you can do and how. In this case, **pandas.read.csv()** starts with a **filepath_or_buffer**, and then has all kinds of other information you could use to customize that function.

Elsewhere in the **User Guide** there are sections that can also help you with using Pandas to read from JSOM, Excel or HDF5, and more! There's also a **search bar** under the Table of Contents so you can search for specific function names, so if you ever get stuck or want to improve your Pandas skills, this is a great place to start!

Challenge:

This challenge will help you get used to using the Pandas documentation to solve problems.

Your aim is to select **multiple columns** from the DataFrame we made – **columns 1 and 2**.

Clue: When looking in the documentation, the answer will be somewhere **near the top** of the page called **Indexing and Selecting Data**. Try to figure out how to do this on your own before reading on.

Solution:

In case you didn't find it, the part of the documentation you need is under the subtitle **Basics** and begins "You can pass a list of columns to [] to select columns in that order."

pandas 0.24.2 documentation » API Reference » Input/Output » previous | next | modules | index

Table Of Contents

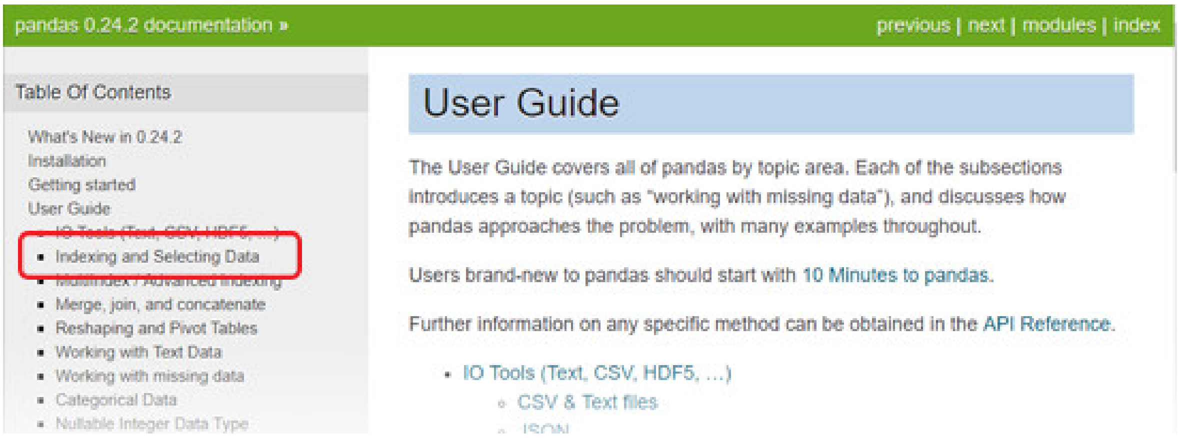
- What's New in 0.24.2
- Installation
- Getting started
- User Guide
- pandas Ecosystem
- API Reference
  - Input/Output
    - Pickling
    - Flat File
    - Clipboard
    - Excel
    - JSON
    - HTML
    - HDFStore: PyTables (HDF5)
    - Feather
    - Parquet
    - SAS

pandas.read_csv

pandas.**read_csv**(*filepath_or_buffer, sep=', ', delimiter=None, header='infer', names=None, index_col=None, usecols=None, squeeze=False, prefix=None, mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None, skipfooter=0, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True, parse_dates=False, infer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False, iterator=False, chunksize=None, compression='infer', thousands=None, decimal=b'.', lineterminator=None, quotechar='"', quoting=0, doublequote=True, escapechar=None, comment=None, encoding=None, dialect=None, tupleize_cols=None, error_bad_lines=True, warn_bad_lines=True, delim_whitespace=False, low_memory=True, memory_map=False, float_precision=None*) [source]

Read a comma-separated values (csv) file into DataFrame.

Also supports optionally iterating or breaking of the file into chunks.



pandas 0.24.2 documentation » previous | next | modules | index

Table Of Contents

- What's New in 0.24.2
- Installation
- Getting started
- User Guide
  - IO Tools (Text, CSV, HDF5, …)
  - Indexing and Selecting Data
  - MultiIndex / Advanced Indexing
  - Merge, join, and concatenate
  - Reshaping and Pivot Tables
  - Working with Text Data
  - Working with missing data
  - Categorical Data
  - Nullable Integer Data Type

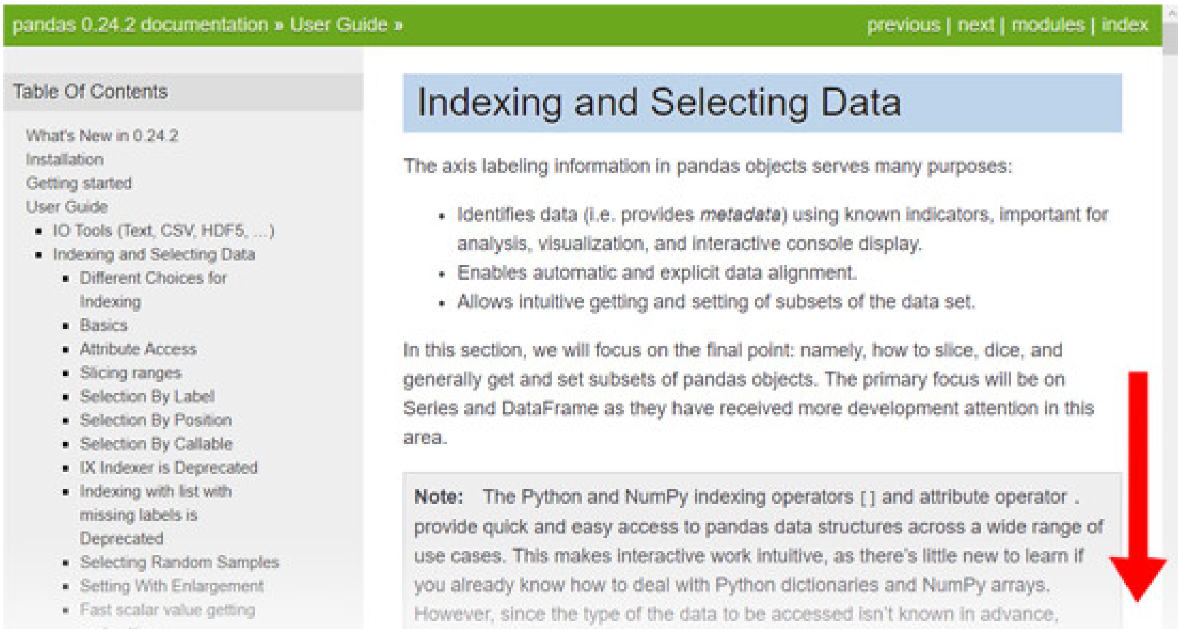
User Guide

The User Guide covers all of pandas by topic area. Each of the subsections introduces a topic (such as "working with missing data"), and discusses how pandas approaches the problem, with many examples throughout.

Users brand-new to pandas should start with 10 Minutes to pandas.

Further information on any specific method can be obtained in the API Reference.

- IO Tools (Text, CSV, HDF5, …)
  - CSV & Text files
  - JSON



pandas 0.24.2 documentation » User Guide » previous | next | modules | index

Table Of Contents

- What's New in 0.24.2
- Installation
- Getting started
- User Guide
  - IO Tools (Text, CSV, HDF5, …)
  - Indexing and Selecting Data
    - Different Choices for Indexing
    - Basics
    - Attribute Access
    - Slicing ranges
    - Selection By Label
    - Selection By Position
    - Selection By Callable
    - IX Indexer is Deprecated
    - Indexing with list with missing labels is Deprecated
    - Selecting Random Samples
    - Setting With Enlargement
    - Fast scalar value getting and setting

Indexing and Selecting Data

The axis labeling information in pandas objects serves many purposes:

- Identifies data (i.e. provides *metadata*) using known indicators, important for analysis, visualization, and interactive console display.
- Enables automatic and explicit data alignment.
- Allows intuitive getting and setting of subsets of the data set.

In this section, we will focus on the final point: namely, how to slice, dice, and generally get and set subsets of pandas objects. The primary focus will be on Series and DataFrame as they have received more development attention in this area.

Note: The Python and NumPy indexing operators [] and attribute operator . provide quick and easy access to pandas data structures across a wide range of use cases. This makes interactive work intuitive, as there's little new to learn if you already know how to deal with Python dictionaries and NumPy arrays. However, since the type of the data to be accessed isn't known in advance,


```
2000-01-05 -0.484513  0.962970  1.174465 -0.888276
2000-01-06 -0.733231  0.509598 -0.580194  0.724113
2000-01-07  0.345164  0.972995 -0.816769 -0.840143
2000-01-08 -0.430188 -0.761943 -0.446079  1.044010
```

You can pass a list of columns to `[]` to select columns in that order. If a column is not contained in the DataFrame, an exception will be raised. Multiple columns can also be set in this manner:

```
In [9]: df
Out[9]:
                   A         B         C         D
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632
2000-01-02  1.212112 -0.173215  0.119209 -1.044236
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
2000-01-04  0.721555 -0.706771 -1.039575  0.271860
2000-01-05 -0.424972  0.567020  0.276232 -1.087401
2000-01-06 -0.673690  0.113648 -1.478427  0.524988
2000-01-07  0.404705  0.577046 -1.715002 -1.039268
2000-01-08 -0.370647 -1.157892 -1.344312  0.844885

In [10]: df[['B', 'A']] = df[['A', 'B']]

In [11]: df
Out[11]:
                   A         B         C         D
2000-01-01 -0.282863  0.469112 -1.509059 -1.135632
2000-01-02 -0.173215  1.212112  0.119209 -1.044236
2000-01-03 -2.104569 -0.861849 -0.494929  1.071804
2000-01-04 -0.706771  0.721555 -1.039575  0.271860
2000-01-05  0.567020 -0.424972  0.276232 -1.087401
2000-01-06  0.113648 -0.673690 -1.478427  0.524988
2000-01-07  0.577046  0.404705 -1.715002 -1.039268
2000-01-08 -1.157892 -0.370647 -1.344312  0.844885
```

You may find this useful for applying a transform (in-place) to a subset of the columns.

Scroll To Top

Warning: pandas aligns all AXES when setting `Series` and `DataFrame` from `.loc`

In the above example, columns B and A and selected and the data in their rows are swapped.

So, to get the data from multiple columns, instead of a string, we can use a **list**:

```
print(df[['col1', 'col2']])
```

```python
# DataFrame: a 2D table with rows and columns
df_data = {
    'col1': np.random.rand(5),
    'col2': np.random.rand(5),
    'col3': np.random.rand(5)
}

df = pd.DataFrame(df_data)

# fetch some rows
#print(df[:1])

# fetch a col
#print(df['col1'])

# fetch multiple cols
print(df[['col1', 'col2']])
```

```
IPython 6.5.0 -- An enhanced
Interactive Python.

In [1]: runfile('/Users/
presenter/Developer/pandas/
intro.py', wdir='/Users/
presenter/Developer/pandas')
       col1      col2
0  0.682355  0.872070
1  0.743886  0.409340
2  0.276640  0.879390
3  0.687909  0.857781
4  0.326488  0.850433

In [2]:
```


The code in the video for this particular lesson has been updated, please see the lesson notes below for the corrected code.

In this lesson, we’ll learn how to read data from CSV and Excel into Pandas and save as a Pandas data file.

There’s a function in Pandas that we can use for this that’s easy to use and handles all of the formatting needed – all we have to do is give it a file name!

Make a new file in **Spyder** in the same directory as the project files.

Start by loading Pandas:

```
import pandas as pd
```

The following code has been updated, and differs from the video:

Next, we’ll have Pandas read an **Excel** spreadsheet. Open **Tracks.xls** in Excel and let’s have a look at what we’ll be working with first.

TrackId	Name	AlbumId	MediaTypeId	GenreId	Composer	Milliseconds	Bytes	UnitPrice
1	For Those About To Rock (W	1	1	1	Angus Young, Malcolm Y	343719	11170334	0.99
2	Balls to the Wall	2	2	1		342562	5510424	0.99
3	Fast As a Shark	3	2	1	F. Baltes, S. Kaufman, U.	230619	3990994	0.99
4	Restless and Wild	3	2	1	F. Baltes, R.A. Smith-Die	252051	4331779	0.99
5	Princess of the Dawn	3	2	1	Deaffy & R.A. Smith-Die	375418	6290521	0.99
6	Put The Finger On You	1	1	1	Angus Young, Malcolm Y	205662	6713451	0.99
7	Let's Get It Up	1	1	1	Angus Young, Malcolm Y	233926	7636561	0.99
8	Inject The Venom	1	1	1	Angus Young, Malcolm Y	210834	6852860	0.99
9	Snowballed	1	1	1	Angus Young, Malcolm Y	203102	6599424	0.99
10	Evil Walks	1	1	1	Angus Young, Malcolm Y	263497	8611245	0.99
11	C.O.D.	1	1	1	Angus Young, Malcolm Y	199836	6566314	0.99
12	Breaking The Rules	1	1	1	Angus Young, Malcolm Y	263288	8596840	0.99
13	Night Of The Long Knives	1	1	1	Angus Young, Malcolm Y	205688	6706347	0.99
14	Spellbound	1	1	1	Angus Young, Malcolm Y	270863	8817038	0.99
15	Go Down	4	1	1	AC/DC	331180	10847611	0.99
16	Dog Eat Dog	4	1	1	AC/DC	215196	7032162	0.99
17	Let There Be Rock	4	1	1	AC/DC	366654	12021261	0.99
18	Bad Boy Boogie	4	1	1	AC/DC	267728	8776140	0.99
19	Problem Child	4	1	1	AC/DC	325041	10617116	0.99
20	Overdose	4	1	1	AC/DC	369319	12066294	0.99
21	Hell Ain't A Bad Place To Be	4	1	1	AC/DC	254380	8331286	0.99
22	Whole Lotta Rosie	4	1	1	AC/DC	323761	10547154	0.99

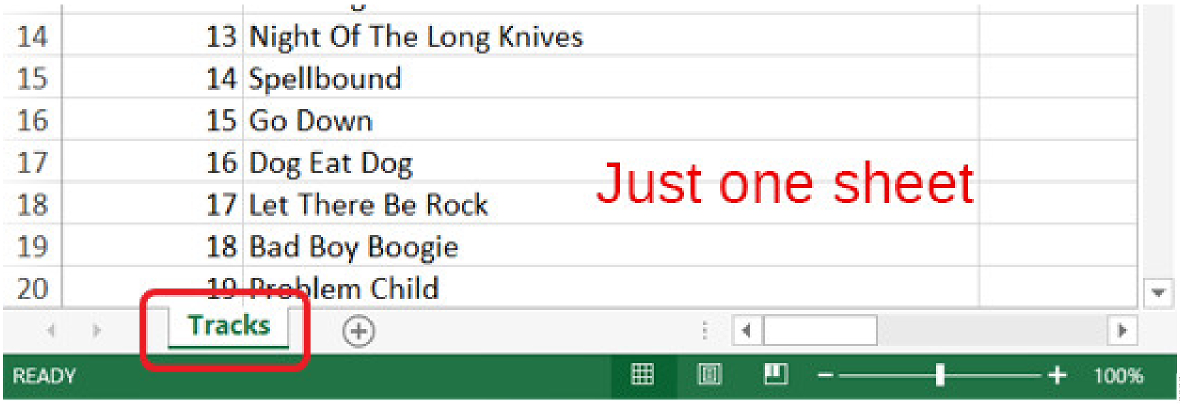
We can see that it’s a list of songs with various related data, such as album ID, Artist and price.

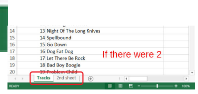
The Pandas function we’re using is **read_excel**, and we want it to load the file **Tracks.xls**.

Note that we’re using a .xls file instead of a .xlsx one:

```
tracks = pd.read_excel('Tracks.xls')
```

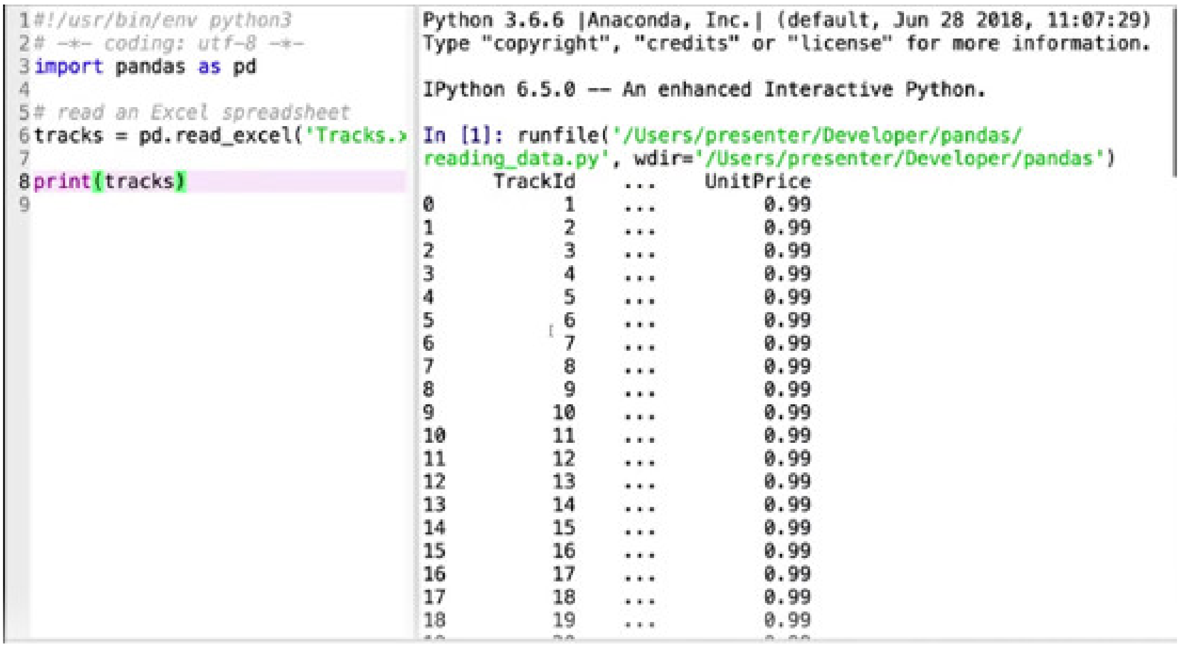
We may also need to choose which **sheet** within the Excel Workbook we want to load, as we can only load one sheet per DataFrame. If you have multiple sheets, you’ll need to create multiple DataFrames, each utilizing this function. If we don’t specify which we want, it’ll select the first sheet by default.



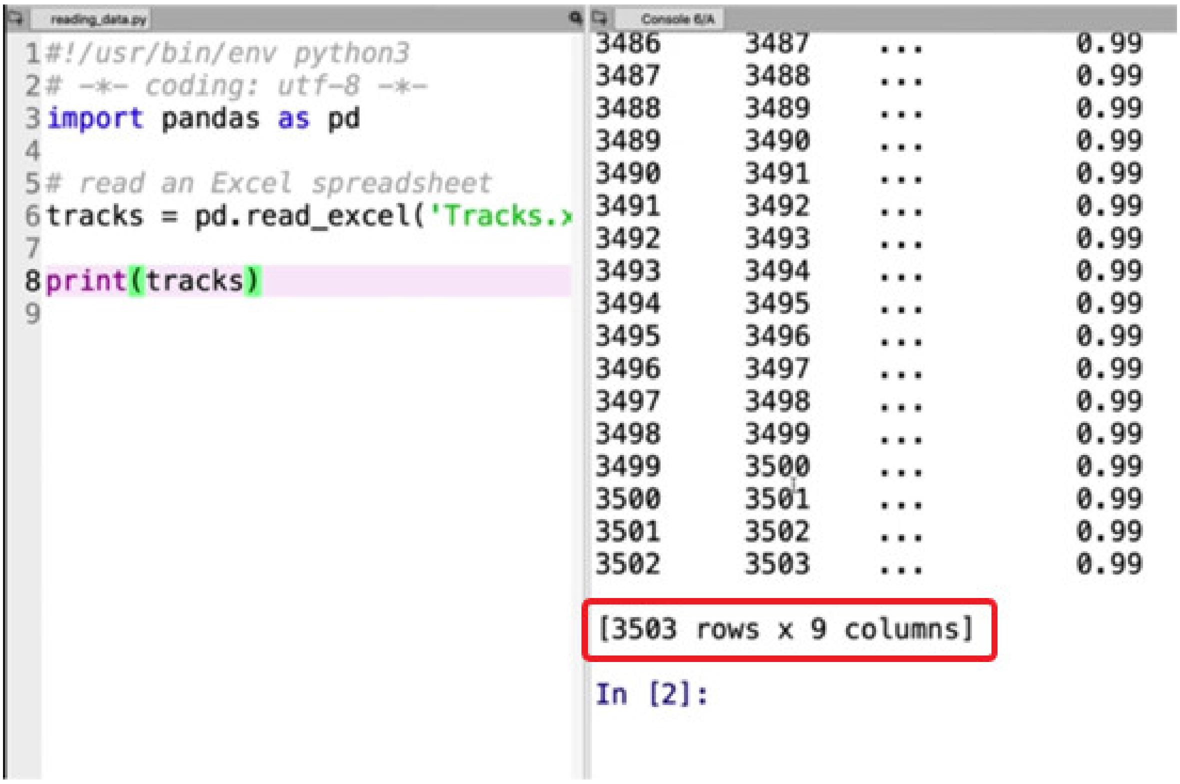


We can specify which sheet either by using the actual **name** of the sheet or using a **zero-indexed integer**. In this case there is only one sheet in the workbook so it's not necessary, but it's worth practicing!

```
tracks = pd.read_excel('Tracks.xls', sheet_name=0)
print(tracks)
```



There's a lot of data here! In addition to printing our sheet, this tells us how many rows and columns the sheet has.

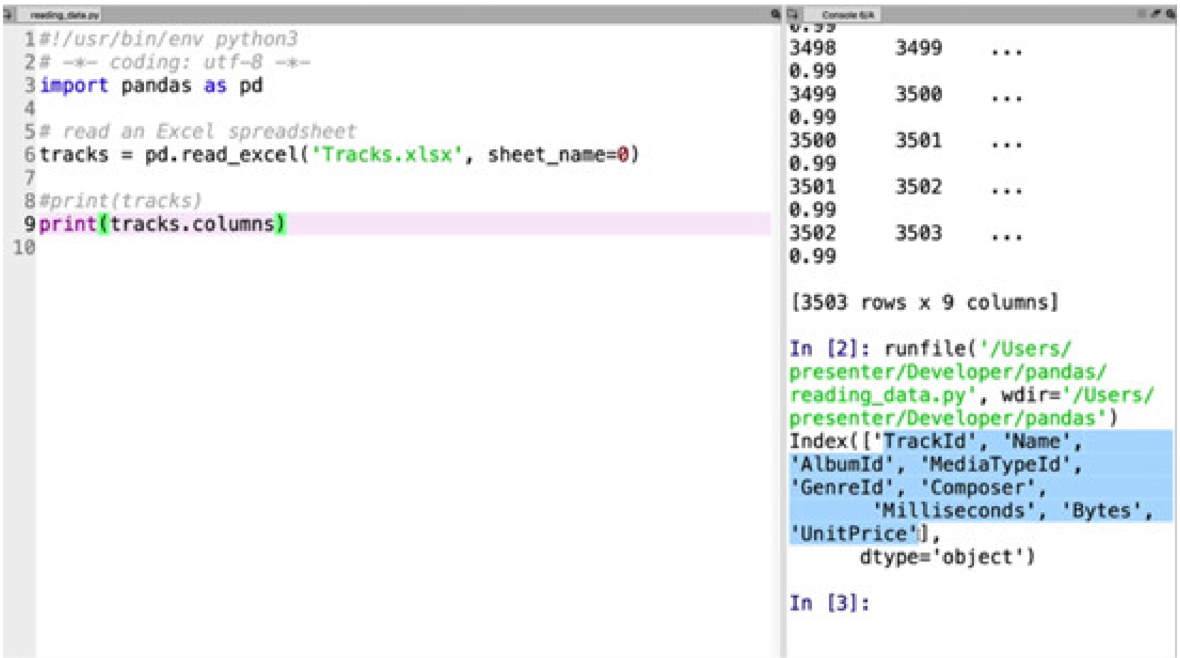


Pandas **seems** to have removed the middle columns and rows, but it hasn't actually. This is purely for display purposes to make it easier for us given the quantity of data we're working with – the data is still there, don't worry! This is obviously very convenient given that our sheet has over 3,500 rows!

```
23      24    ...    0.99
24      25    ...    0.99
25      26    ...    0.99
26      27    ...    0.99
27      28    ...    0.99
28      29    ...    0.99
29      30    ...    0.99
...    ...    ...     ...
3473  3474    ...    0.99
3474  3475    ...    0.99
3475  3476    ...    0.99
3476  3477    ...    0.99
3477  3478    ...    0.99
3478  3479    ...    0.99
3479  3480    ...    0.99
```

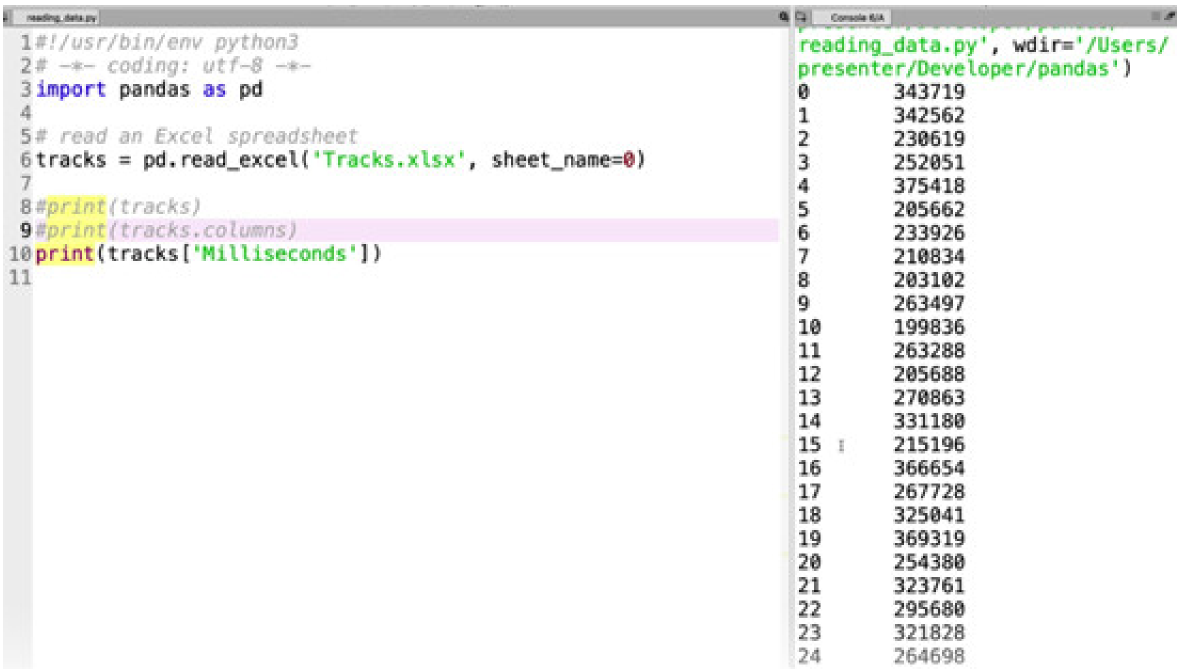
If you’re ever concerned, you can ask for those missing columns and rows. Let’s do that now by printing the columns.

```
print(tracks.columns)
```

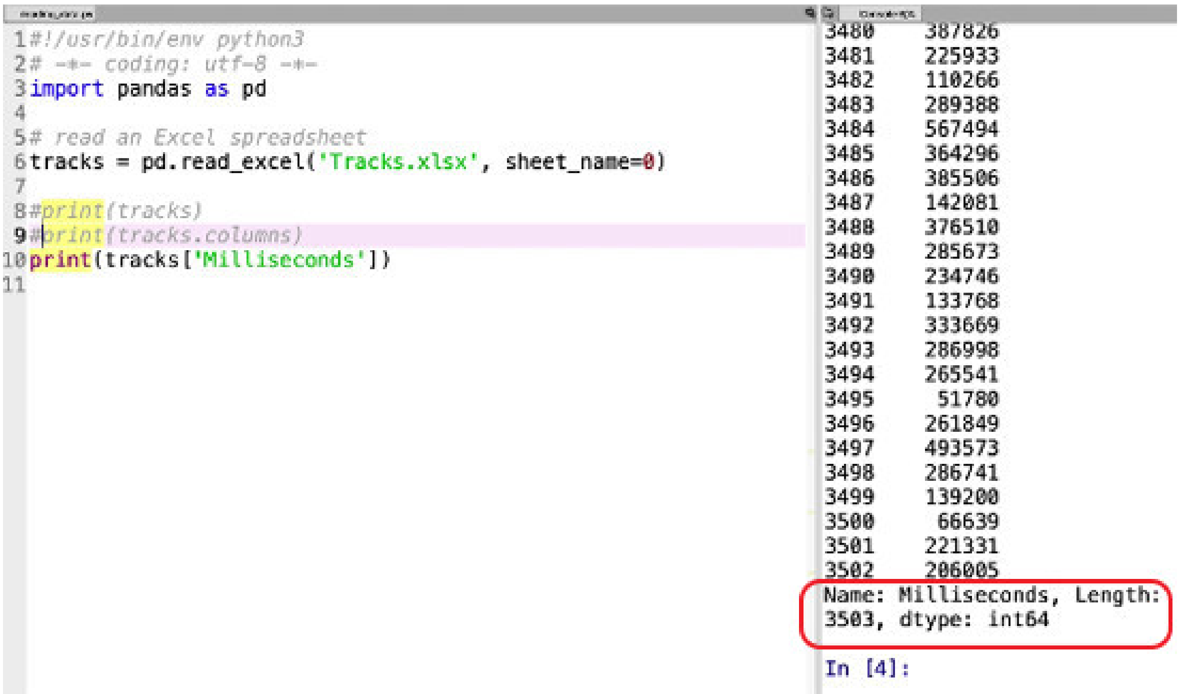


We can now see that it has all the columns we wanted. We can make extra sure by asking for the contents of an individual column. Let's print out the milliseconds column:

```
print(tracks['Milliseconds'])
```



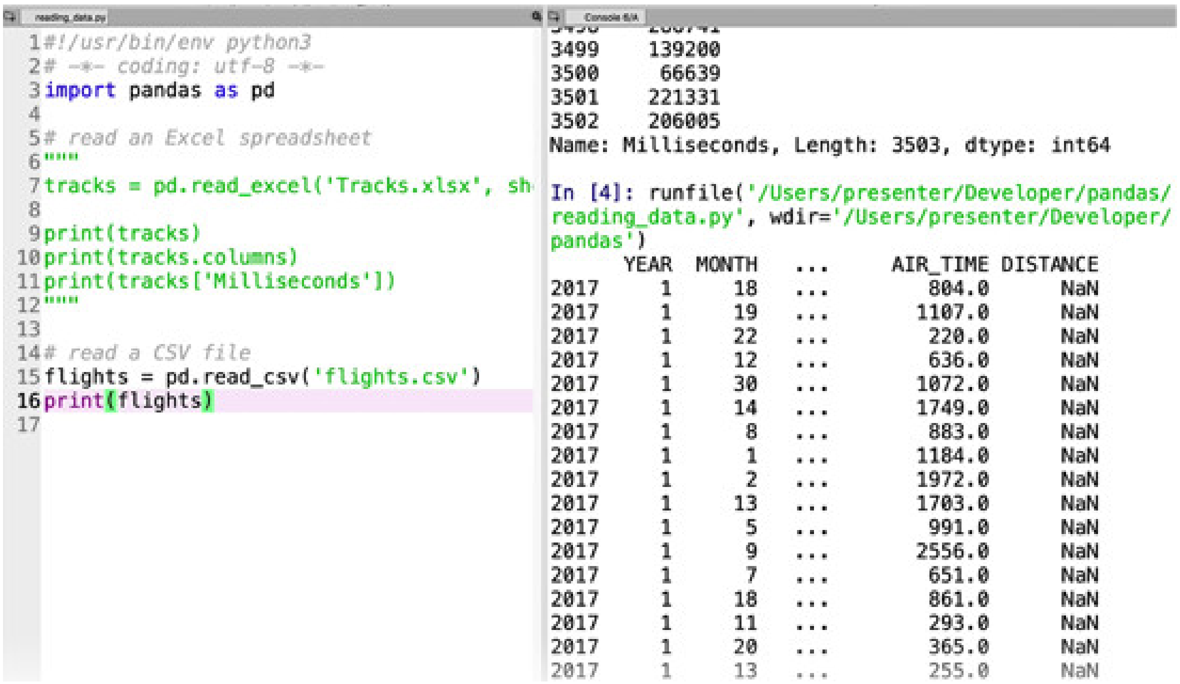
And there it is, abbreviated as before with dots in the middle, and with some helpful information at the bottom: the name of the column, the number of rows, and the data type.



Now let’s do the same with a CSV spreadsheet – **flights.csv**. Have a look at it before if you like. It contains lots of information about flights during 2017. Instead of **read_excel**, we’re using **read_csv** – other than that it’s exactly the same:

```
flights = pd.read_csv('flights.csv')
print(flights)
```

Run this. You may have to wait a little – it’s a big file!



You’ll notice that we have a similar issue. With **600,000 rows** and **25 columns**, the data we actually have printed out is reduced nicely. That's good, but there seems to be a problem – the data isn't matching up with the column names.

```
In [4]: runfile('/Users/presenter/Developer/panc
reading_data.py', wdir='/Users/presenter/Develo
pandas')
      YEAR  MONTH    ...     AIR_TIME  DISTANCE
2017     1     18    ...        804.0       NaN
2017     1     19    ...       1107.0       NaN
2017     1     22    ...        220.0       NaN
2017     1     12    ...        636.0       NaN
2017     1     30    ...       1072.0       NaN
2017     1     14    ...       1749.0       NaN
2017     1      8    ...        883.0       NaN
2017     1      1    ...       1184.0       NaN
2017     1      2    ...       1972.0       NaN
2017     1     13    ...       1703.0       NaN
2017     1      5    ...        991.0       NaN
2017     1      9    ...       2556.0       NaN
2017     1      7    ...        651.0       NaN
2017     1     18    ...        861.0       NaN
2017     1     11    ...        293.0       NaN
2017     1     20    ...        365.0       NaN
2017     1     13    ...        255.0       NaN
```

This is because when we load a CSV like this in Pandas, unlike with Excel files, it will try to use the first column as the **index**, which is why everything's been offset by one. To resolve this, we need to add a parameter:

```
flights = pd.read_csv('flights.csv', index_col=False)
print(flights)
```

```
In [5]: runfile('/Users/presenter/Developer/
pandas/reading_data.py', wdir='/Users/presenter/
Developer/pandas')
        YEAR  MONTH    ...     AIR_TIME  DISTANCE
0       2017      1    ...        107.0     804.0
1       2017      1    ...        153.0    1107.0
2       2017      1    ...         40.0     220.0
3       2017      1    ...         97.0     636.0
4       2017      1    ...        137.0    1072.0
5       2017      1    ...        266.0    1749.0
6       2017      1    ...        131.0     883.0
7       2017      1    ...        130.0    1184.0
8       2017      1    ...        290.0    1972.0
9       2017      1    ...        222.0    1703.0
10      2017      1    ...        145.0     991.0
11      2017      1    ...        269.0    2556.0
12      2017      1    ...         74.0     651.0
13      2017      1    ...        111.0     861.0
14      2017      1    ...         55.0     293.0
```

Fixed! Once again, if we want to check on those apparently missing rows and columns we can.

```
print(flights.columns)
```

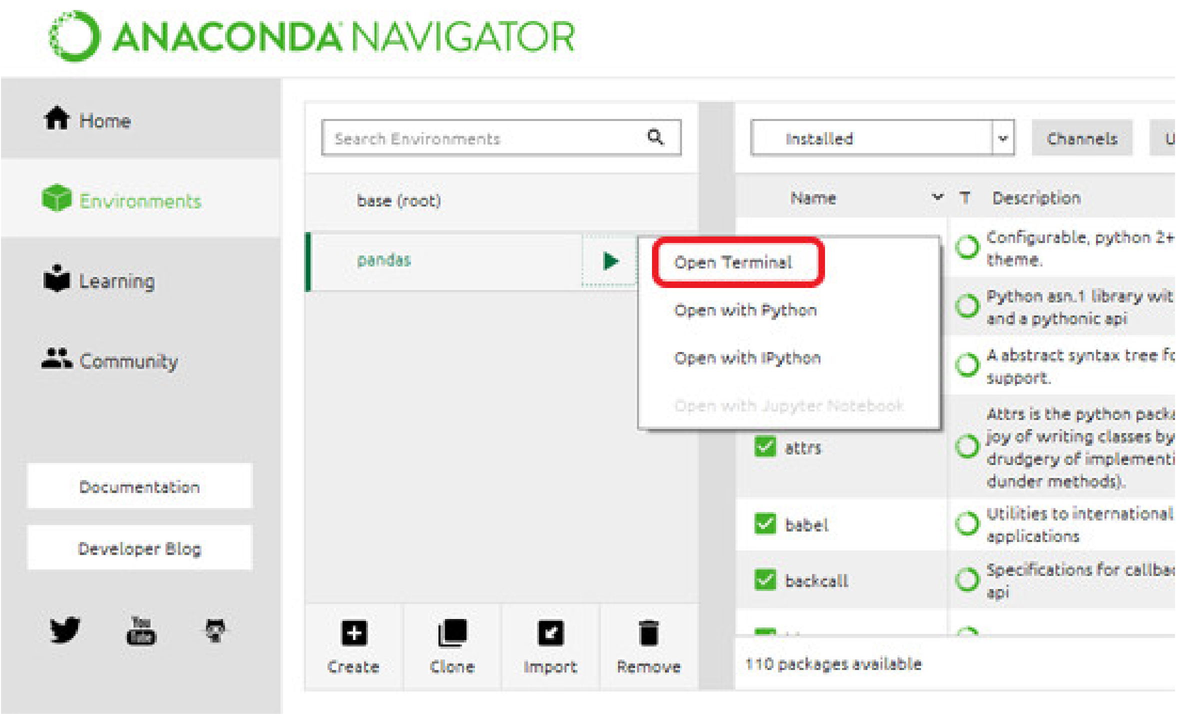
```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import pandas as pd

# read an Excel spreadsheet
"""
tracks = pd.read_excel('Tracks.xlsx', sheet_

print(tracks)
print(tracks.columns)
print(tracks['Milliseconds'])
"""

# read a CSV file
flights = pd.read_csv('flights.csv', index_c
print(flights)
```

```
599993  2017     12    ...        136.0    1024.0
599994  2017     12    ...         91.0     689.0
599995  2017     12    ...         25.0     121.0
599996  2017     12    ...         78.0     588.0
599997  2017     12    ...        160.0    1371.0
599998  2017     12    ...         48.0     268.0
599999  2017     12    ...        213.0    1546.0

[600000 rows x 25 columns]

In [6]: flights.columns
Out[6]:
Index(['YEAR', 'MONTH', 'DAY_OF_MONTH',
'DAY_OF_WEEK', 'FL_DATE', 'AIRLINE_ID',
       'TAIL_NUM', 'ORIGIN', 'ORIGIN_CITY_NAME',
'ORIGIN_STATE_NM', 'DEST',
       'DEST_CITY_NAME', 'DEST_STATE_NM',
'CRS_DEP_TIME', 'DEP_TIME',
       'TAXI_OUT', 'TAXI_IN', 'CRS_ARR_TIME',
'ARR_TIME', 'CANCELLED',
       'DIVERTED', 'CRS_ELAPSED_TIME',
'ACTUAL_ELAPSED_TIME', 'AIR_TIME',
       'DISTANCE'],
      dtype='object')

In [7]:
```

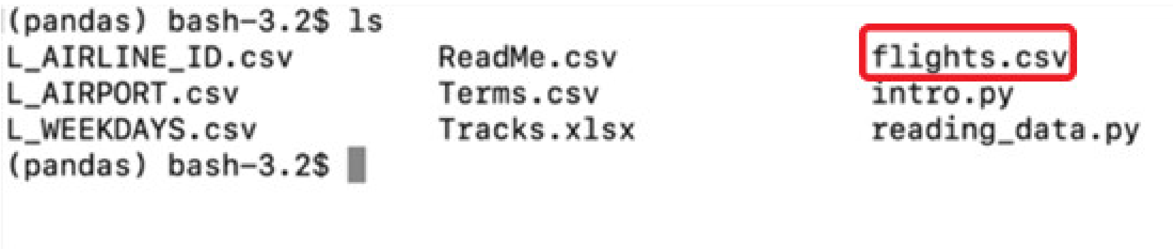
It's important to know your data well before starting any kind of data analysis – the kind of data you're working with and what that data means will determine how you work with it later.

This lesson we'll be working with the data set **flights.csv**.

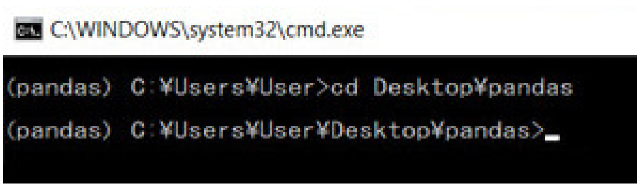
Open **Anaconda**, go to **Environments**, select the **pandas** environment, click the green arrow and select **Open Terminal**.



First we need make sure we're in the correct directory – the same one as where **flights.csv** is. To see the content of the directory you're currently in, enter **dir** (Windows) or **ls** (Mac, that's 'LS').



If needed, use **cd** to change the directory, using **cd ..** to go up a level. For example, **cd \Desktop\pandas**



Now start a **Python interpreter**. Enter **ipython**.

```
(pandas) bash-3.2$ ls
L_AIRLINE_ID.csv         ReadMe.csv               flights.csv
L_AIRPORT.csv            Terms.csv                intro.py
L_WEEKDAYS.csv           Tracks.xlsx              reading_data.py
(pandas) bash-3.2$ ipython
Python 3.6.6 |Anaconda, Inc.| (default, Jun 28 2018, 11:07:29)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

From now on we can input any Python code here and press enter to have it run immediately.

Let’s load our flight data into Pandas. **Import pandas** then read the data from the file using the function we learned in the last lesson, remembering to set **index_column** equal to **false** to avoid any indexing problems.

```
import pandas as pd
flights = pd.read_csv('flights.csv', index_col=False)
```

```
(pandas) bash-3.2$ ls
L_AIRLINE_ID.csv         ReadMe.csv               flights.csv
L_AIRPORT.csv            Terms.csv                intro.py
L_WEEKDAYS.csv           Tracks.xlsx              reading_data.py
(pandas) bash-3.2$ ipython
Python 3.6.6 |Anaconda, Inc.| (default, Jun 28 2018, 11:07:29)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import pandas as pd

In [2]: flights = pd.read_csv('flights.csv', index_col=False)

In [3]:
```

To get more information on our data set, we can now enter **flights** and hit enter.

```
In [3]: flights
Out[3]:
       YEAR  MONTH  DAY_OF_MONTH    ...     ACTUAL_ELAPSED_TIME AIR_TIME  DISTANCE
0      2017      1            18    ...                   130.0    107.0     804.0
1      2017      1            19    ...                   189.0    153.0    1107.0
2      2017      1            22    ...                    53.0     40.0     220.0
3      2017      1            12    ...                   131.0     97.0     636.0
4      2017      1            30    ...                   154.0    137.0    1072.0
5      2017      1            14    ...                   283.0    266.0    1749.0
6      2017      1             8    ...                   152.0    131.0     883.0
7      2017      1             1    ...                   164.0    130.0    1184.0
8      2017      1             2    ...                   374.0    290.0    1972.0
9      2017      1            13    ...                   237.0    222.0    1703.0
10     2017      1             5    ...                   164.0    145.0     991.0
```


The result is much the same as we had in the last session with rows and columns being excluded for ease of viewing. At the bottom we can see that the data set has **600,000 rows** and **25 columns**. At the top we can see some of the column names and check that the **indices** are working properly, which they are.

```
In [3]: flights
Out[3]:
       YEAR  MONTH  DAY_OF_MONTH    ...
0      2017      1            18    ...
1      2017      1            19    ...
2      2017      1            22    ...
3      2017      1            12    ...
4      2017      1            30    ...
5      2017      1            14    ...
6      2017      1             8    ...
7      2017      1             1    ...
8      2017      1             2    ...
9      2017      1            13    ...
10     2017      1             5    ...
```

The data in this file is a sample of domestic flight data in the US during 2017. We know then that the **Year** value will always be 2017, and the numbers under **Month** will be in the range **1-12**, corresponding to January, February, etc.

Let's print out the other columns to get a better overview. Enter **flights.columns**.

```
In [4]: flights.columns
Out[4]:
Index(['YEAR', 'MONTH', 'DAY_OF_MONTH', 'DAY_OF_WEEK', 'FL_DATE', 'AIRLINE_ID',
       'TAIL_NUM', 'ORIGIN', 'ORIGIN_CITY_NAME', 'ORIGIN_STATE_NM', 'DEST',
       'DEST_CITY_NAME', 'DEST_STATE_NM', 'CRS_DEP_TIME', 'DEP_TIME',
       'TAXI_OUT', 'TAXI_IN', 'CRS_ARR_TIME', 'ARR_TIME', 'CANCELLED',
       'DIVERTED', 'CRS_ELAPSED_TIME', 'ACTUAL_ELAPSED_TIME', 'AIR_TIME',
       'DISTANCE'],
      dtype='object')

In [5]:
```

We now have all the column names. Now we can pick any one of these to print specifically. Let's go for **DAY_OF_WEEK**:

```
flights['DAY_OF_WEEK']
```



```
In [5]: flights['DAY_OF_WEEK']
Out[5]:
0          3
1          4
2          7
3          4
4          1
5          6
6          7
7          7
8          1
9          5
10         4
11         1
12         6
13         3
14         3
```

We can see that the day of the week is stored as a number between **1** and **7**. You can probably guess what each number means, but it's good to be certain and data in other columns aren't as clear. Luckily we've been supplied with additional files to act as important reference points.

Open **ReadMe.csv**.

	A	B	C	D	E	F	G	H
1	SYS_FIELD_NAME	FIELD_DESC						
2	YEAR	Year						
3	MONTH	Month						
4	DAY_OF_MONTH	Day of Month						
5	DAY_OF_WEEK	Day of Week						
6	FL_DATE	Flight Date (yyyymmdd)						
7	AIRLINE_ID	An identification number assigned by US DOT to identify a unique airline (carrier).						
8	TAIL_NUM	Tail Number						
9	ORIGIN	Origin Airport						
10	ORIGIN_CITY_NAME	Origin Airport, City Name						
11	ORIGIN_STATE_NM	Origin Airport, State Name						
12	DEST	Destination Airport						
13	DEST_CITY_NAME	Destination Airport, City Name						
14	DEST_STATE_NM	Destination Airport, State Name						
15	CRS_DEP_TIME	CRS Departure Time (local time: hhmm)						
16	DEP_TIME	Actual Departure Time (local time: hhmm)						
17	TAXI_OUT	Taxi Out Time, in Minutes						

This contains all the column names in our data set and what they actually mean. We also have **Terms.csv** which explains a lot of the terminology used.

	A	B	C	D	E	F	G	H
1	TERM	DEFINITION						
2	Actual Arrival Times	Gate arrival time is the instance when the pilot sets the aircraft parking brake a						
3	Actual Departure Times	Gate departure time is the instance when the pilot releases the aircraft parkin						
4	Airline ID	An identification number assigned by US DOT to identify a unique airline (carri						
5	Airport Code	A three character alpha-numeric code issued by the U.S. Department of Trans						
6	Airport ID	An identification number assigned by US DOT to identify a unique airport. Use						
7	Arrival Delay	Arrival delay equals the difference of the actual arrival time minus the schedul						
8	CRS	Computer Reservation System. CRS provide information on airline schedules,						
9	Cancelled Flight	A flight that was listed in a carrier's computer reservation system during the se						
10	Carrier Code	Code assigned by IATA and commonly used to identify a carrier. As the same c						
11	Certificate Of Public Conveni	A certificate issued to an air carrier under 49 U.S.C. 41102, by the Department						
12	Certificated Air Carrier	An air carrier holding a Certificate of Public Convenience and Necessity issued						
13	Certified Air Carrier	An air carrier holding a Certificate of Public Convenience and Necessity issued						
14	City Market ID	An identification number assigned by US DOT to identify a city market. Use th						
15	Departure Delay	The difference between the scheduled departure time and the actual departur						
16	Diverted Flight	A flight that is required to land at a destination other than the original schedul						
17	Domestic Operations	All air carrier operations having destinations within the 50 United States, the D						

For example, one of our columns is called **CRS_DEP_TIME**. In **ReadMe** we see that this means **CRS Departure Time**, but we still don't know what **CRS** is.

10	ORIGIN_CITY_NAME	Origin Airport, City Name
11	ORIGIN_STATE_NM	Origin Airport, State Name
12	DEST	Destination Airport
13	DEST_CITY_NAME	Destination Airport, City Name
14	DEST_STATE_NM	Destination Airport, State Name
15	CRS_DEP_TIME	CRS Departure Time (local time: hhmm)
16	DEP_TIME	Actual Departure Time (local time: hhmm)
17	TAXI_OUT	Taxi Out Time, in Minutes
18	TAXI_IN	Taxi In Time, in Minutes
19	CRS_ARR_TIME	CRS Arrival Time (local time: hhmm)
20	ARR_TIME	Actual Arrival Time (local time: hhmm)

If we look for it in **Terms**, we'll discover that it stands for **Computer Reservation System**, and we can read about what that specifically means.

3	Actual Departure Times	Gate departure time is the instance when the pilot releases the aircraft parkin
4	Airline ID	An identification number assigned by US DOT to identify a unique airline (carri
5	Airport Code	A three character alpha-numeric code issued by the U.S. Department of Trans
6	Airport ID	An identification number assigned by US DOT to identify a unique airport. Use
7	Arrival Delay	Arrival delay equals the difference of the actual arrival time minus the schedul
8	CRS	Computer Reservation System. CRS provide information on airline schedules,
9	Cancelled Flight	A flight that was listed in a carrier's computer reservation system during the se
10	Carrier Code	Code assigned by IATA and commonly used to identify a carrier. As the same c
11	Certificate Of Public Conveni	A certificate issued to an air carrier under 49 U.S.C. 41102, by the Department
12	Certificated Air Carrier	An air carrier holding a Certificate of Public Convenience and Necessity issued
13	Certified Air Carrier	An air carrier holding a Certificate of Public Convenience and Necessity issued

Besides the column **names**, we also need **ReadMe** to understand the **data**. For example, the entry for **FL_DATE** tells us both that it means **Flight Date** and that the data comes in the format **yyyymmdd**.

	A	B	C	D
1	SYS_FIELD_NAME	FIELD_DESC		
2	YEAR	Year		
3	MONTH	Month		
4	DAY_OF_MONTH	Day of Month		
5	DAY_OF_WEEK	Day of Week		
6	FL_DATE	Flight Date (yyyymmdd)		
7	AIRLINE_ID	An identification number assigned l		
8	TAIL_NUM	Tail Number		
9	ORIGIN	Origin Airport		
10	ORIGIN_CITY_NAME	Origin Airport, City Name		

We also have a few other spreadsheets to explain certain specific columns: For **DAY_OF_WEEK** we can look at **L_WEEKDAYS.csv** which confirms what day each numerical value corresponds to.

	A	B	C
1	Code	Description	
2	1	Monday	
3	2	Tuesday	
4	3	Wednesday	
5	4	Thursday	
6	5	Friday	
7	6	Saturday	
8	7	Sunday	
9	9	Unknown	
10			

Next there’s **AIRLINE_ID** which, according to **ReadMe**, provides the unique ID number of the airline. Using our **L_AIRLINE_ID.csv** spreadsheet, we can use the values in **AIRLINE_ID** to see what airline the ID number corresponds to.

Code	Description
19031	Mackey International Inc.: MAC
19032	Munz Northern Airlines Inc.: XY
19033	Cochise Airlines Inc.: COC
19034	Golden Gate Airlines Inc.: GSA
19035	Aeromech Inc.: RZZ
19036	Golden West Airlines Co.: GLW
19037	Puerto Rico Intl Airlines: PRN
19038	Air America Inc.: STZ
19039	Swift Aire Lines Inc.: SWT
19040	American Central Airlines: TSF
19041	Valdez Airlines: VEZ
19042	Southeast Alaska Airlines: WEB
19043	Altair Airlines Inc.: AAR
19044	Chitina Air Service: CHI
19045	Marco Island Airways Inc.: MRC
19046	Caribbean Air Services Inc.: OHZ

Finally we have **ORIGIN** and its corresponding file **L_AIRPORT.csv**. The ID data in **ORIGIN** refers to the **origin airport**, where the flight left from, and by looking up that ID in **L_AIRPORT**, we can see where specifically that is.

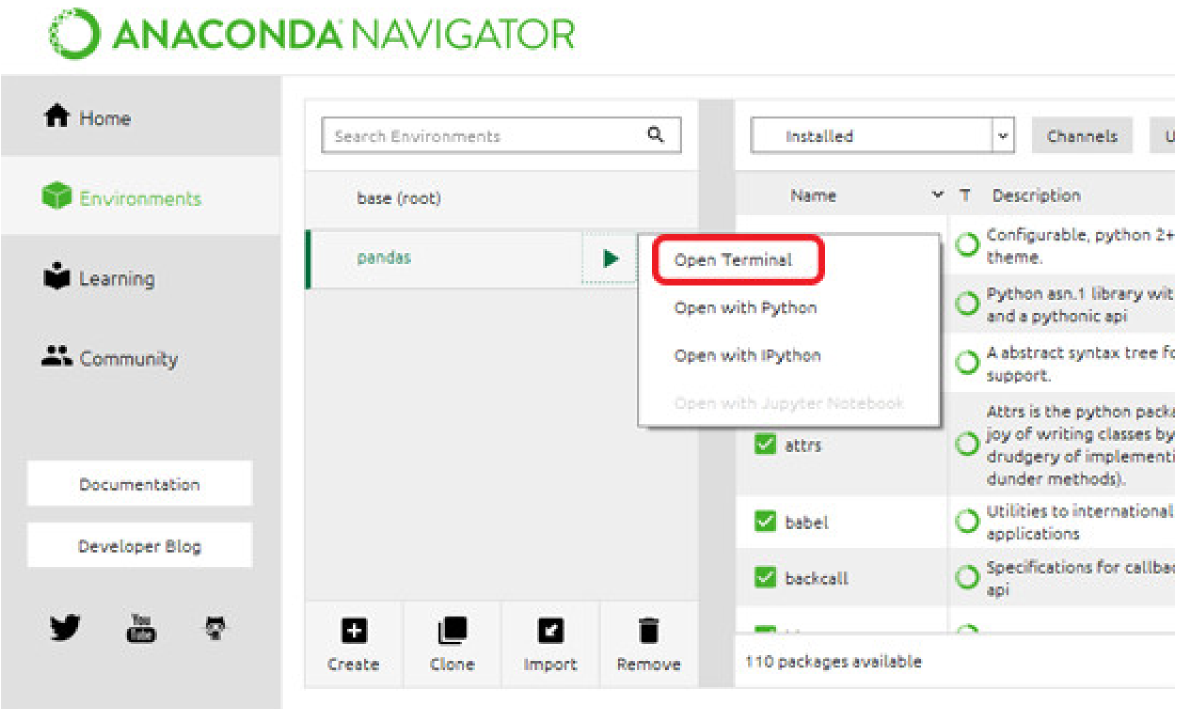
	A	B	C	D	E	F
1	Code	Description				
2	01A	Afognak Lake, AK: Afognak Lake Airport				
3	03A	Granite Mountain, AK: Bear Creek Mining Strip				
4	04A	Lik, AK: Lik Mining Camp				
5	05A	Little Squaw, AK: Little Squaw Airport				
6	06A	Kizhuyak, AK: Kizhuyak Bay				
7	07A	Klawock, AK: Klawock Seaplane Base				
8	08A	Elizabeth Island, AK: Elizabeth Island Airport				
9	09A	Homer, AK: Augustin Island				
10	1B1	Hudson, NY: Columbia County				
11	1G4	Peach Springs, AZ: Grand Canyon West				
12	1N7	Blairstown, NJ: Blairstown Airport				
13	1NY	Penn Yan, NY: Penn Yan Airport				
14	6B0	Middlebury, VT: Middlebury State				
15	7AK	Akun, AK: Akun Airport				
16	8F3	Crosbyton, TX: Crosbyton Municipal				
17	A01	Fairbanks/Ft. Wainwright, AK: Blair Lake				
18	A02	Deadmans Bay, AK: Deadmans Bay Airport				
19	A03	Hallo Bay, AK: Hallo Bay Airport				

Examine the remaining columns and look up any that you're at all unsure about until you have a good grasp of it all.

In this lesson we'll learn how to select data in Pandas, in particular looking at selecting rows and columns.

Instead of working in Spyder, we're going to load an **IPython** session. Because **flights** is such a large data set, working with it in Spyder would mean we'd have to keep running the file over and over again waiting a while each time, but in an IPython session we can simply load it once and then work directly from that.

As in the last lesson, load up a **terminal**, make sure you're in the **right directory** and enter **ipython**.




```
(pandas) bash-3.2$ ls
L_AIRLINE_ID.csv        ReadMe.csv          flights.csv
L_AIRPORT.csv           Terms.csv           intro.py
L_WEEKDAYS.csv          Tracks.xlsx         reading_data.py
(pandas) bash-3.2$ ipython
Python 3.6.6 |Anaconda, Inc.| (default, Jun 28 2018, 11:07:29)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

Now import **pandas** and load in **flights**.

```
import pandas as pd
flights = pd.read_csv('flights.csv', index_col=False)
```

As we know, if we enter **flights**, we'll see data from the first and last few rows and columns. This is great for getting a quick overview, but we'll want to get more specific. Let's review how we select columns and rows.

Selecting a Column:

Let’s say we want to get all of the **origin** airports for each row. We know we can get the data from a column by using its name rather than an index. Try this now.

```
flights['ORIGIN']
```

```
In [6]: flights['ORIGIN']
Out[6]:
0           RNO
1           LGA
2           BWI
3           JFK
4           MCO
5           DTW
6           BWI
7           DAL
8           MIA
9           FLL
10          DEN
11          HNL
```

```
599990     MIA
599991     ATL
599992     RIC
599993     DEN
599994     SEA
599995     ITO
599996     LGB
599997     LAS
599998     PHL
599999     MCO
Name: ORIGIN, Length: 600000, dtype: object
```

So this has returned all of the origin airports from all the flights, as well some additional information at the bottom – the column **name**, the **length** (number of rows) and the **data type**. In this case the data type is an **object**, which in pandas is essentially a **python string**, not to be confused with python objects!

Now let's say we want the **origin** and the **destination** airports. We do this by putting our column names inside a **list**.

```
flights[['ORIGIN', 'DEST']]
```

```
In [8]: flights[['ORIGIN', 'DEST']]
Out[8]:
       ORIGIN DEST
0         RNO  DEN
1         LGA  MCI
2         BWI  ISP
3         JFK  CHS
4         MCO  PVD
5         DTW  LAS
6         BWI  PBI
7         DAL  DCA
8         MIA  PHX
9         FLL  DEN
10        DEN  PDX
11        HNL  LAX
12        MSY  CLT
```



```
599992     RIC   BOS
599993     DEN   SEA
599994     SEA   SLC
599995     ITO   OGG
599996     LGB   SLC
599997     LAS   STL
599998     PHL   PIT
599999     MCO   DEN

[600000 rows x 2 columns]

In [9]:
```

This returns both of the desired columns and a little less information – only that there are **600,000 rows** and **2 columns**.

Selecting Rows:

Let’s say we want just **the first 3 rows**. Remember that for rows we use **indexes** and index **slicing** which starts at 0 and goes up to but not including the final stated index. Have a go at that now.

```
flights[:3]
```

```
In [10]: flights[:3]
Out[10]:
   YEAR  MONTH  DAY_OF_MONTH    ...     ACTUAL_ELAPSED_TIME AIR_TIME  DISTANCE
0  2017      1            18    ...                   130.0    107.0     804.0
1  2017      1            19    ...                   189.0    153.0    1107.0
2  2017      1            22    ...                    53.0     40.0     220.0

[3 rows x 25 columns]
```

This returns the first 3 rows with the middle columns visually excluded for convenience, and includes the additional data, **3 rows x 25 columns**.

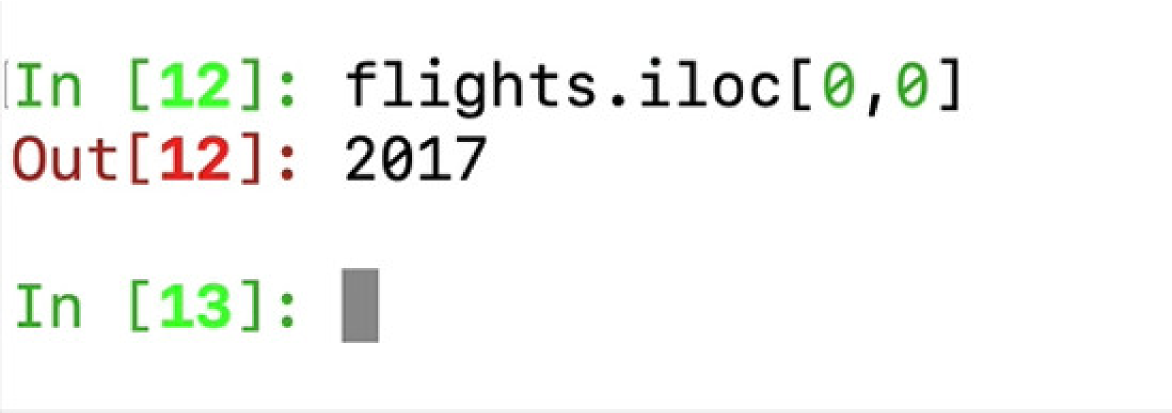
Selecting Rows & Columns:

We’re in new territory now! This time we want to get data from rows and columns at the same time.

We’ll use an object called **iloc**, which stands for **integer location**. We give it a row and a column and it returns the single value at that intersection. The iloc object only takes **integer** values, so this time we specify the column as an integer, not a string. When entering our indices, we give it the **row** location first and the **column** location second (**data.iloc[row, column]**).

Let’s try and get the value from the **1st row and 1st column**.

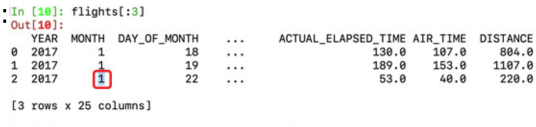
```
flights.iloc[0,0]
```



```
In [12]: flights.iloc[0,0]
Out[12]: 2017

In [13]:
```

Exercise:

Use pandas to fetch the value highlighted in the image below:



```
In [10]: flights[:3]
Out[10]:
   YEAR  MONTH  DAY_OF_MONTH    ...     ACTUAL_ELAPSED_TIME AIR_TIME  DISTANCE
0  2017      1            18    ...                   130.0    107.0     804.0
1  2017      1            19    ...                   189.0    153.0    1107.0
2  2017      1            22    ...                    53.0     40.0     220.0

[3 rows x 25 columns]
```

Answer:

To get this value, you have to ask for the **3rd row** and **2nd column**.

```
flights.iloc[2,1]
```

```
In [14]: flights.iloc[2, 1]
Out[14]: 1

In [15]:
```

An alternative to working out and entering the desired column's index is the function **get_loc**. This takes a **column name string** and returns its index.

Let's use **get_loc**, attached to **flights.columns**, to get the value at the **3rd row** of the **DAY_OF_MONTH** column:

```
flights.iloc[2, flights.columns.get_loc('DAY_OF_MONTH')]
```



```
In [15]: # mixing row indices with string column names

In [16]: flights.iloc[2, flights.columns.get_loc('DAY_OF_MONTH')]
Out[16]: 22

In [17]:
```

We can now build on this knowledge and use this to get multiple data points at once. Let's get the **DAY_OF_MONTH** data for **the first 3 rows**:

```
flights.iloc[:3, flights.columns.get_loc('DAY_OF_MONTH')]
```

```
In [17]: flights.iloc[:3, flights.columns.get_loc('DAY_OF_MONTH')]
Out[17]:
0    18
1    19
2    22
Name: DAY_OF_MONTH, dtype: int64

In [18]:
```

We can also do something similar to get specific data from multiple columns. Let's say we want the **origin** and **destination** data from the **first** flight. We'll do the same thing, but just like when we wanted multiple columns before, we're going to make a list.

```
flights.iloc[0, [flights.columns.get_loc('ORIGIN'), flights.columns.get_loc('DEST')]]
```

```
In [18]: flights.iloc[0, [flights.columns.get_loc('ORIGIN'), flights.columns.get_loc('DEST')]]
Out[18]:
ORIGIN    RNO
DEST      DEN
Name: 0, dtype: object

In [19]:
```

And of course we can go even further! Let's get the **origin** and **destination** data for the **first 3 rows**.

```
flights.iloc[:3, [flights.columns.get_loc('ORIGIN'), flights.columns.get_loc('DEST')]
]
```



```
In [19]: flights.iloc[:3, [flights.columns.get_loc('ORIGIN'), flights.columns.get_loc('DEST')]]
Out[19]:
  ORIGIN DEST
0    RNO  DEN
1    LGA  MCI
2    BWI  ISP

In [20]:
```

In this video we'll learn how to sort our data by a particular column, sort it in ascending or descending order, and how to sort by multiple columns.

Start a new **iPython** session, ensure you're in the correct directory, load **ipython**, import **pandas** and load in our flight data.

```
(pandas) bash-3.2$ ls
L_AIRLINE_ID.csv        ReadMe.csv              flights.csv
L_AIRPORT.csv           Terms.csv               intro.py
L_WEEKDAYS.csv          Tracks.xlsx             reading_data.py
(pandas) bash-3.2$ ipython
Python 3.6.6 |Anaconda, Inc.| (default, Jun 28 2018, 11:07:29)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import pandas as pd

In [2]: flights = pd.read_csv('flights.csv', index_col=False)

In [3]:
```

You can enter **flights** to make sure.

```
In [3]: flights
Out[3]:
      YEAR  MONTH  DAY_OF_MONTH    ...     ACTUAL_ELAPSED_TIME  AIR_TIME  DISTANCE
0     2017      1            18    ...                   130.0     107.0     804.0
1     2017      1            19    ...                   189.0     153.0    1107.0
2     2017      1            22    ...                    53.0      40.0     220.0
3     2017      1            12    ...                   131.0      97.0     636.0
4     2017      1            30    ...                   154.0     137.0    1072.0
5     2017      1            14    ...                   283.0     266.0    1749.0
6     2017      1             8    ...                   152.0     131.0     883.0
7     2017      1             1    ...                   164.0     130.0    1184.0
8     2017      1             2    ...                   374.0     290.0    1972.0
9     2017      1            13    ...                   237.0     222.0    1703.0
10    2017      1             5    ...                   164.0     145.0     991.0
11    2017      1             9    ...                   304.0     269.0    2556.0
```

Sorting Values by Column:

Imagine we want to predict how late a flight might be based on how long its journey is. In this case, it would be helpful to be able to sort our data by the **DISTANCE** column, showing us what the shortest and longest flights are.

We can do this using the **sort_values** function. This takes an argument which is a **list** of **columns**.

```
flights.sort_values(by=['DISTANCE'])
```

```
In [4]: # sort values by a column

In [5]: flights.sort_values(by=['DISTANCE'])
Out[5]:
        YEAR  MONTH  DAY_OF_MONTH    ...     ACTUAL_ELAPSED_TIME AIR_TIME  DISTANCE
578968  2017     12            23    ...                    20.0      9.0      31.0
36133   2017      1             3    ...                    16.0     10.0      31.0
14122   2017      1            23    ...                    17.0      9.0      31.0
432689  2017      9            14    ...                    21.0      9.0      31.0
280640  2017      6            14    ...                    27.0     18.0      31.0
293689  2017      6            26    ...                    31.0     10.0      31.0
195067  2017      4            25    ...                    16.0     10.0      31.0
375805  2017      8            11    ...                    22.0     10.0      31.0
333057  2017      7            25    ...                    22.0     12.0      31.0
186529  2017      4            23    ...                    19.0      8.0      31.0
181527  2017      4            12    ...                    16.0     11.0      31.0
31316   2017      1            26    ...                    21.0      9.0      31.0

62398   2017      2            23    ...                   544.0    517.0    4983.0
199063  2017      4            26    ...                   598.0    576.0    4983.0
270081  2017      6            13    ...                   576.0    551.0    4983.0
236540  2017      5            15    ...                   609.0    590.0    4983.0
122517  2017      3             3    ...                   547.0    522.0    4983.0
249934  2017      5            17    ...                   578.0    558.0    4983.0
128977  2017      3            25    ...                   627.0    605.0    4983.0
598116  2017     12             7    ...                   661.0    622.0    4983.0
4566    2017      1             6    ...                   543.0    516.0    4983.0
303454  2017      7            29    ...                   560.0    532.0    4983.0
568779  2017     12            27    ...                   549.0    530.0    4983.0
310308  2017      7            20    ...                   568.0    543.0    4983.0
94529   2017      2            25    ...                   528.0    501.0    4983.0
245357  2017      5            29    ...                   587.0    563.0    4983.0

[600000 rows x 25 columns]
```

The data is now sorted from lowest to highest distance (**ascending** values), from just 31 miles to 4,983. In the first flight returned they spent only 9 minutes in the air (**AIR_TIME**), while some of the flights near the bottom took more than 600 minutes. We can also see by looking at the other data that there aren't any obvious patterns – people seem just as likely to travel any distance regardless of the month or day. This doesn't mean there isn't a pattern to be found, but it's an important first impression.

We can also see that our **indices** have remained in the original order.

```
In [4]: # sort values by a column

In [5]: flights.sort_values(by=['DISTANCE'])
Out[5]:
        YEAR  MONTH  DAY_OF_MONTH    ...     ACTUAL_ELAPSED_TIME AIR_TIME  DISTANCE
578968  2017     12            23    ...                    20.0      9.0      31.0
36133   2017      1             3    ...                    16.0     10.0      31.0
14122   2017      1            23    ...                    17.0      9.0      31.0
432689  2017      9            14    ...                    21.0      9.0      31.0
280640  2017      6            14    ...                    27.0     18.0      31.0
293689  2017      6            26    ...                    31.0     10.0      31.0
195067  2017      4            25    ...                    16.0     10.0      31.0
375805  2017      8            11    ...                    22.0     10.0      31.0
333057  2017      7            25    ...                    22.0     12.0      31.0
186529  2017      4            23    ...                    19.0      8.0      31.0
```

For the same column in **descending** order, we use an extra parameter setting **ascending** to **False**.

```
flights.sort_values(by=['DISTANCE'], ascending=False)
```



```
In [6]: flights.sort_values(by=['DISTANCE'], ascending=False)
Out[6]:
        YEAR  MONTH  DAY_OF_MONTH    ...     ACTUAL_ELAPSED_TIME AIR_TIME  DISTANCE
560359  2017     12            29    ...                   570.0    520.0    4983.0
144801  2017      3             8    ...                   544.0    515.0    4983.0
173307  2017      4            25    ...                   607.0    581.0    4983.0
270081  2017      6            13    ...                   576.0    551.0    4983.0
236540  2017      5            15    ...                   609.0    590.0    4983.0
47417   2017      1            29    ...                   602.0    569.0    4983.0
97147   2017      2            26    ...                   517.0    498.0    4983.0
581849  2017     12             6    ...                   548.0    526.0    4983.0
138737  2017      3             6    ...                   694.0    663.0    4983.0
568393  2017     12            21    ...                   581.0    531.0    4983.0

382346  2017      8            17    ...                    29.0     13.0      31.0
197331  2017      4            20    ...                    17.0     10.0      31.0
519828  2017     11            11    ...                    23.0     12.0      31.0
279370  2017      6             5    ...                     NaN      NaN      31.0
280640  2017      6            14    ...                    27.0     18.0      31.0
293689  2017      6            26    ...                    31.0     10.0      31.0
75969   2017      2             6    ...                    17.0      9.0      31.0
265678  2017      6            21    ...                    16.0      9.0      31.0
208715  2017      5            19    ...                    22.0     10.0      31.0
268879  2017      6             2    ...                    31.0     11.0      31.0
271022  2017      6            19    ...                    17.0      9.0      31.0
```

Exercise:

Sort the data by **AIR_TIME** values, **descending**.

We've seen that the **distance** of a flight seems strongly but not totally reliably related to **air time**, so looking at **AIR_TIME** could help clarify their relationship.

Answer:

```
flights.sort_values(by=['AIR_TIME'], ascending=False)
```



```
In [7]: # sort by AIR_TIME descending

In [8]: flights.sort_values(by=['AIR_TIME'], ascending=False)
Out[8]:
        YEAR  MONTH  DAY_OF_MONTH    ...    ACTUAL_ELAPSED_TIME  AIR_TIME  DISTANCE
86348   2017      2             5    ...                  737.0     711.0    4983.0
523244  2017     11            19    ...                  745.0     700.0    4983.0
63556   2017      2             8    ...                  745.0     696.0    4962.0
133803  2017      3            10    ...                  751.0     688.0    4983.0
517417  2017     11            25    ...                  704.0     675.0    4983.0
517460  2017     11            10    ...                  698.0     671.0    4983.0
97156   2017      2            28    ...                  692.0     670.0    4983.0
527312  2017     11            16    ...                  699.0     668.0    4983.0
67870   2017      2             1    ...                  688.0     667.0    4962.0
138737  2017      3             6    ...                  694.0     663.0    4983.0
493779  2017     10             6    ...                  680.0     661.0    4983.0
78332   2017      2            10    ...                  697.0     656.0    4962.0
34320   2017      1             7    ...                  716.0     655.0    4983.0
28361   2017      1            21    ...                  682.0     653.0    4962.0
102284  2017      3            22    ...                  689.0     653.0    4983.0
```

Most of the top flights have the longest possible distance, but the 3rd flight is slightly shorter. This result shows us that while it’s broadly true that longer distances mean longer air time, it’s not 100% reliable – delays can happen.

Sorting by Multiple Columns:

This allows us to sort by one column first and then within that result, sort by a second column (and then perhaps by a third…).

To sort by multiple columns, we simply add them to the **list**. Let’s sort by **DISTANCE** and then **AIR_TIME** in **descending** order.

```
flights.sort_values(by=['DISTANCE', 'AIR_TIME'], ascending=False)
```

```
In [9]: # sort values by multiple columns

In [10]: flights.sort_values(by=['DISTANCE', 'AIR_TIME'], ascending=False)
Out[10]:
        YEAR  MONTH  DAY_OF_MONTH    ...    ACTUAL_ELAPSED_TIME  AIR_TIME  DISTANCE
86348   2017      2             5    ...                  737.0     711.0    4983.0
523244  2017     11            19    ...                  745.0     700.0    4983.0
133803  2017      3            10    ...                  751.0     688.0    4983.0
517417  2017     11            25    ...                  704.0     675.0    4983.0
517460  2017     11            10    ...                  698.0     671.0    4983.0
97156   2017      2            28    ...                  692.0     670.0    4983.0
527312  2017     11            16    ...                  699.0     668.0    4983.0
138737  2017      3             6    ...                  694.0     663.0    4983.0
493779  2017     10             6    ...                  680.0     661.0    4983.0
34320   2017      1             7    ...                  716.0     655.0    4983.0
102284  2017      3            22    ...                  689.0     653.0    4983.0
172     2017      1            14    ...                  679.0     650.0    4983.0
18000   2017      1            30    ...                  670.0     646.0    4983.0
105861  2017      3            27    ...                  676.0     646.0    4983.0
```

We can see then that the longest air time for the longest distance (4,983 miles) is 711 minutes.

Looking through the other air time values for this distance we can see that the values have quite a large range, from 711 to 604 (and possibly less, as the results are hidden), confirming that the **DISTANCE** value doesn't precisely indicate the **AIR_TIME**.

304832	2017	7	25	...	641.0	612.0	4983.0
225473	2017	5	8	...	647.0	611.0	4983.0
128977	2017	3	25	...	627.0	605.0	4983.0
304747	2017	7	12	...	631.0	604.0	4983.0
...	...	...	...	...	...	...	...
31316	2017	1	26	...	21.0	9.0	31.0
75969	2017	2	6	...	17.0	9.0	31.0
129017	2017	3	18	...	18.0	9.0	31.0

Start a **terminal**, navigate to the correct **directory**, import **pandas** and load the **flight** data.

```
(pandas) bash-3.2$ ls
L_AIRLINE_ID.csv        ReadMe.csv          flights.csv
L_AIRPORT.csv           Terms.csv           intro.py
L_WEEKDAYS.csv          Tracks.xlsx         reading_data.py
(pandas) bash-3.2$ ipython
Python 3.6.6 |Anaconda, Inc.| (default, Jun 28 2018, 11:07:29)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import pandas as pd

In [2]: flights = pd.read_csv('flights.csv', index_col=False)

In [3]:
```

All filters use a **boolean** (true/false expression) to check values against, and will return all rows that satisfy that expression.

Let’s say we only want data for flights in **January**. If we input **flights[‘MONTH’] == 1**, this will return a list of boolean values with all the flights from January being marked **True**.

```
In [4]: # fetch all January flights

In [5]: flights['MONTH'] == 1
Out[5]:
0         True
1         True
2         True
3         True
4         True
5         True
6         True
7         True
8         True
9         True
10        True
11        True
```

Now we can say what to select, to filter the rows we index this statement into flights:

flights[flights['MONTH'] == 1]

```
In [6]: flights[flights['MONTH'] == 1]
Out[6]:
       YEAR  MONTH  DAY_OF_MONTH    ...     ACTUAL_ELAPSED_TIME  AIR_TIME  DISTANCE
0      2017      1            18    ...                   130.0     107.0     804.0
1      2017      1            19    ...                   189.0     153.0    1107.0
2      2017      1            22    ...                    53.0      40.0     220.0
3      2017      1            12    ...                   131.0      97.0     636.0
4      2017      1            30    ...                   154.0     137.0    1072.0
5      2017      1            14    ...                   283.0     266.0    1749.0
6      2017      1             8    ...                   152.0     131.0     883.0
7      2017      1             1    ...                   164.0     130.0    1184.0
8      2017      1             2    ...                   374.0     290.0    1972.0
9      2017      1            13    ...                   237.0     222.0    1703.0
10     2017      1             5    ...                   164.0     145.0     991.0
11     2017      1             9    ...                   304.0     269.0    2556.0
```

You can see the filtering has worked because we've now got **50,000 rows** rather than the **600,000** we started with, and they all have a **MONTH** value of **1**.

```
49992  2017      1            16    ...                    96.0      81.0     569.0
49993  2017      1             6    ...                   204.0     167.0    1107.0
49994  2017      1             6    ...                   174.0     127.0    1121.0
49995  2017      1             1    ...                   185.0     166.0    1171.0
49996  2017      1            12    ...                   168.0     135.0     937.0
49997  2017      1             9    ...                   147.0     123.0    1023.0
49998  2017      1             7    ...                   168.0     157.0    1447.0
49999  2017      1            19    ...                    53.0      31.0     122.0

[50000 rows x 25 columns]
```

We can also do this with **string** values. Let's fetch all **flights leaving the state of New York**:

```
flights[flights['ORIGIN_STATE_NM'] == 'New York']
```

```
In [9]: # fetch all flights leaving New York

In [10]: flights[flights['ORIGIN_STATE_NM'] == 'New York']
Out[10]:
       YEAR  MONTH  DAY_OF_MONTH    ...     ACTUAL_ELAPSED_TIME  AIR_TIME  DISTANCE
1      2017      1            19    ...                   189.0     153.0    1107.0
3      2017      1            12    ...                   131.0      97.0     636.0
28     2017      1            10    ...                    58.0      36.0      96.0
41     2017      1            31    ...                   135.0     103.0     636.0
42     2017      1             8    ...                     NaN       NaN    2586.0
46     2017      1            23    ...                   154.0     140.0     972.0
48     2017      1            10    ...                    64.0      51.0     241.0
95     2017      1             8    ...                   122.0      78.0     474.0
138    2017      1            13    ...                   170.0     117.0     762.0
161    2017      1            27    ...                   104.0      83.0     546.0
165    2017      1            17                          197.0     156.0    1204.0
```

We can also use other expressions like comparison operators.

```
long_flights = flights[flights['DISTANCE'] > 4000]
long_flights
```

```
In [11]: long_flights = flights[flights['DISTANCE'] > 4000]

In [12]: long_flights
Out[12]:
        YEAR  MONTH  DAY_OF_MONTH    ...     ACTUAL_ELAPSED_TIME AIR_TIME  DISTANCE
172     2017      1            14    ...                   679.0    650.0    4983.0
911     2017      1             3    ...                   534.0    502.0    4983.0
1506    2017      1             3    ...                   627.0    607.0    4817.0
2966    2017      1            30    ...                   527.0    502.0    4502.0
3650    2017      1             9    ...                   492.0    464.0    4502.0
4566    2017      1             6    ...                   543.0    516.0    4983.0
5305    2017      1            16    ...                   592.0    562.0    4962.0
5571    2017      1            18    ...                   584.0    561.0    4983.0
6034    2017      1             3    ...                     NaN      NaN    4243.0
8618    2017      1            23    ...                   578.0    530.0    4983.0
```

Because the result is also a DataFrame we can add more filters to refine the data further. Let's say we want all the **long flights which start in the state of Hawaii**.

```
long_flights[long_flights['ORIGIN_STATE_NM'] == "Hawaii"]
```

```
In [13]: long_flights[long_flights['ORIGIN_STATE_NM'] == "Hawaii"]
Out[13]:
        YEAR  MONTH  DAY_OF_MONTH    ...     ACTUAL_ELAPSED_TIME AIR_TIME  DISTANCE
911     2017      1             3    ...                   534.0    502.0    4983.0
2966    2017      1            30    ...                   527.0    502.0    4502.0
3650    2017      1             9    ...                   492.0    464.0    4502.0
4566    2017      1             6    ...                   543.0    516.0    4983.0
5305    2017      1            16    ...                   592.0    562.0    4962.0
5571    2017      1            18    ...                   584.0    561.0    4983.0
8618    2017      1            23    ...                   578.0    530.0    4983.0
8895    2017      1            18    ...                   558.0    528.0    4962.0
10013   2017      1            24    ...                   489.0    460.0    4502.0
11348   2017      1            21    ...                   426.0    402.0    4243.0
11655   2017      1             1    ...                   482.0    459.0    4502.0
12197   2017      1            12    ...                   556.0    521.0    4962.0
```

Combining Operators:

Unlike normal Python we can't use **and**, **or** and **not**. This is because we're working with DataFrames, so there's more than one true/false value to check.

Instead we use bitwise operators: **&**, **|**, and **~**. We also need to wrap all the conditions we're checking in **parentheses** to account for how Python calculates the order of operations.

OR (|):

Get all long flights which start or end at Hawaii.

```
long_flights[(long_flights['ORIGIN_STATE_NM'] == "Hawaii") | (long_flights['DEST_STAT
E_NM'] == "Hawaii")]
```



```
In [15]: long_flights[(long_flights['ORIGIN_STATE_NM'] == "Hawaii") | (long_flights['DEST_STATE_NM
    ...: '] == "Hawaii")]
Out[15]:
      YEAR  MONTH  DAY_OF_MONTH   ...     ACTUAL_ELAPSED_TIME AIR_TIME  DISTANCE
172   2017      1            14   ...                   679.0    650.0    4983.0
911   2017      1             3   ...                   534.0    502.0    4983.0
1506  2017      1             3   ...                   627.0    607.0    4817.0
2966  2017      1            30   ...                   527.0    502.0    4502.0
3650  2017      1             9   ...                   492.0    464.0    4502.0
4566  2017      1             6   ...                   543.0    516.0    4983.0
5305  2017      1            16   ...                   592.0    562.0    4962.0
5571  2017      1            18   ...                   584.0    561.0    4983.0
6034  2017      1             3   ...                     NaN      NaN    4243.0
8618  2017      1            23   ...                   578.0    530.0    4983.0
8895  2017      1            18   ...                   558.0    528.0    4962.0
```

Interestingly, this is the same sample as **long_flights**, so we have the insight that all the long flights recorded started or ended in Hawaii.

AND (&):

Get all flights which are more than 4,000 miles and which happened in January.

```
flights[(flights['DISTANCE'] > 4000) & (flights['MONTH'] == 1)]
```

```
In [16]: # long flights in January

In [17]: flights[(flights['DISTANCE'] > 4000) & (flights['MONTH'] == 1)]
Out[17]:
      YEAR  MONTH  DAY_OF_MONTH   ...     ACTUAL_ELAPSED_TIME AIR_TIME  DISTANCE
172   2017      1            14   ...                   679.0    650.0    4983.0
911   2017      1             3   ...                   534.0    502.0    4983.0
1506  2017      1             3   ...                   627.0    607.0    4817.0
2966  2017      1            30   ...                   527.0    502.0    4502.0
3650  2017      1             9   ...                   492.0    464.0    4502.0
4566  2017      1             6   ...                   543.0    516.0    4983.0
5305  2017      1            16   ...                   592.0    562.0    4962.0
5571  2017      1            18   ...                   584.0    561.0    4983.0
6034  2017      1             3   ...                     NaN      NaN    4243.0
8618  2017      1            23   ...                   578.0    530.0    4983.0
```

NOT (~):

Get all flights which are more than 4,000 miles and weren't in January.

```
flights[(flights['DISTANCE'] > 4000) & ~(flights['MONTH'] == 1)]
```



```
583209  2017    12          10   ...            502.0   475.0   4243.0
583841  2017    12           7   ...            505.0   475.0   4243.0
590067  2017    12           2   ...            569.0   549.0   4983.0
592050  2017    12          25   ...            534.0   496.0   4962.0
593103  2017    12          12   ...            575.0   547.0   4962.0
594109  2017    12          14   ...            493.0   465.0   4243.0
597534  2017    12          17   ...            598.0   564.0   4502.0
598116  2017    12           7   ...            661.0   622.0   4983.0

[313 rows x 25 columns]

In [20]:
```


This lesson we'll learn how to **group data** and apply **aggregate** functions to them.

Start a **terminal**, navigate to the correct **directory**, import **pandas**, import **numpy** and load the **flights** data.

```
import pandas as pd
import numpy as np
flights = pd.read_csv('flights.csv', index_col=False)
```

```
(pandas) bash-3.2$ ls
L_AIRLINE_ID.csv        ReadMe.csv              filtering_data.py       reading_data.py
L_AIRPORT.csv           Terms.csv               flights.csv             selecting_data.py
L_WEEKDAYS.csv          Tracks.xlsx             intro.py                sorting_data.py
(pandas) bash-3.2$ ipython
Python 3.6.6 |Anaconda, Inc.| (default, Jun 28 2018, 11:07:29)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import pandas as pd

In [2]: import numpy as np

In [3]: flights = pd.read_csv('flights.csv', index_col=False)

In [4]:
```

Grouping data:

Let's group our data by month using the **groupby** function, storing the grouped data in a variable.

```
flights_by_month = flights.groupby('MONTH')
flights_by_month
```

```
In [5]: # group flights by month

In [6]: flights_by_month = flights.groupby('MONTH')

In [7]: flights_by_month
Out[7]: <pandas.core.groupby.groupby.DataFrameGroupBy object at 0x1254acb38>

In [8]:
```

This returns a Python representation because this isn't how we work with groups.

Working with Grouped Data:

Once grouped, we have to ask for which group of data we want. Let's ask for the data from the **December** group.

```
flights_by_month.get_group(12)
```



```
599990  2017  12    6  ...    159.0  136.0  1096.0
599991  2017  12    4  ...    188.0  169.0  1199.0
599992  2017  12   22  ...     78.0   65.0   474.0
599993  2017  12   10  ...    163.0  136.0  1024.0
599994  2017  12    9  ...    123.0   91.0   689.0
599995  2017  12    6  ...     38.0   25.0   121.0
599996  2017  12    4  ...     93.0   78.0   588.0
599997  2017  12   22  ...    182.0  160.0  1371.0
599998  2017  12   22  ...     65.0   48.0   268.0
599999  2017  12   10  ...    229.0  213.0  1546.0

[50000 rows x 25 columns]

In [10]:
```

Now say we want to know the **total distance** traveled by planes **in each month**. We start with our data grouped by month, then specify the **DISTANCE** values, then call the **.aggregate** function. For its parameter we'll enter a NumPy function which will calculate the **sum** total of all distances **for each group**.

```
flights_by_month['DISTANCE'].aggregate(np.sum)
```

```
In [11]: flights_by_month['DISTANCE'].aggregate(np.sum)
Out[11]:
MONTH
1     42428340.0
2     42392773.0
3     42718411.0
4     42531603.0
5     42479974.0
6     43279241.0
7     44057833.0
8     43229468.0
9     42529698.0
10    41978931.0
11    42563667.0
12    43403428.0
Name: DISTANCE, dtype: float64
```

We now have the total number of miles traveled in each month. We can use this same statement with other NumPy functions to get a variety of interesting information back, such as the **mean** average distance of a flight per month…

```
flights_by_month['DISTANCE'].aggregate(np.mean)
```

```
In [12]: flights_by_month['DISTANCE'].aggregate(np.mean)
Out[12]:
MONTH
1     848.56680
2     847.85546
3     854.36822
4     850.63206
5     849.59948
6     865.58482
7     881.15666
8     864.58936
9     850.59396
10    839.57862
11    851.27334
12    868.06856
Name: DISTANCE, dtype: float64
```

...the largest distance covered by a flight per month...

```
flights_by_month['DISTANCE'].aggregate(np.max)
```

```
In [13]: flights_by_month['DISTANCE'].aggregate(np.max)
Out[13]:
MONTH
1     4983.0
2     4983.0
3     4983.0
4     4983.0
5     4983.0
6     4983.0
7     4983.0
8     4983.0
9     4983.0
10    4983.0
11    4983.0
12    4983.0
Name: DISTANCE, dtype: float64
```

...and the smallest distance traveled per month.

```
flights_by_month['DISTANCE'].aggregate(np.min)
```

```
In [14]: flights_by_month['DISTANCE'].aggregate(np.min)
Out[14]:
MONTH
1     31.0
2     31.0
3     31.0
4     31.0
5     31.0
6     31.0
7     31.0
8     31.0
9     31.0
10    31.0
11    31.0
12    31.0
Name: DISTANCE, dtype: float64
```

We can also use **groupby** to quickly get important information, for example, which month was the biggest for travel (having the largest total distance). Applying **.max()** to our **sum distance** statement will return the single largest value from our 12 groups.

```
flights_by_month['DISTANCE'].aggregate(np.sum).max()
```

```
In [15]: # get the max total distance travelled and the month

In [16]: flights_by_month['DISTANCE'].aggregate(np.sum).max()
Out[16]: 44057833.0

In [17]:
```

But this is only the value – we need the **index** to know which month that was. To get that, we use **idxmax()**, meaning **index max**, to return the index of the largest value.

```
flights_by_month['DISTANCE'].aggregate(np.sum).idxmax()
```

```
In [16]: flights_by_month['DISTANCE'].aggregate(np.sum).max()
Out[16]: 44057833.0

In [17]: flights_by_month['DISTANCE'].aggregate(np.sum).idxmax()
Out[17]: 7

In [18]:
```

44,057,833 miles in July

Now try getting the opposite result – which month had the smallest total distance covered and how many miles was that?

```
flights_by_month['DISTANCE'].aggregate(np.sum).min()
flights_by_month['DISTANCE'].aggregate(np.sum).idxmin()
```

```
In [18]: # get the min total distance travelled and the month

In [19]: flights_by_month['DISTANCE'].aggregate(np.sum).min()
Out[19]: 41978931.0

In [20]: flights_by_month['DISTANCE'].aggregate(np.sum).idxmin()
Out[20]: 10
```

As a final exercise, try and fetch the total number of **cancelled** flights per month.

```
flights_by_month['CANCELLED'].aggregate(np.sum)
```

```
In [21]: # number of cancelled flights per month

In [22]: flights_by_month['CANCELLED'].aggregate(np.sum)
Out[22]:
MONTH
1       966.0
2       777.0
3       851.0
4       792.0
5       376.0
6       522.0
7       500.0
8      1048.0
9      1671.0
10      339.0
11      155.0
12      528.0
Name: CANCELLED, dtype: float64
```

And that's it for this course! Feel free to have a go at applying different aggregate functions to different columns to test your skills and see what data insights you can gather.

