Master Data Analysis with Python - Intro to Pandas

by
Ted Petrou

Contents

Part I

Environment Setup and Jupyter Notebooks

Chapter 1

Installing Python and Setting up an Environment for Data Science

This document provides a guide for installing Python 3 onto your system and creating an environment suitable for data science work. It is important that you follow this guide precisely so that your system is setup to run all the code in this book. This document covers this setup for Windows, macOS, and Linux operating systems. A video tutorial is available.

1.1 Conda

There are a number of ways to set up your system to do data science in Python. This tutorial relies upon a tool called conda which is both a **package manager** and an **environment manager**.

Package Manager

A package manager is a tool that installs, updates, and removes computer programs. For us, these computer programs will be third-party Python packages. A third-party Python package is any package that is not part of the Python standard library, which comes with every Python installation. There are many thousands of third-party packages available to be installed onto your system.

Another popular package manager is **pip**, which existed long before conda and is the default package manager for new Python installations. Although pip is a good tool, we will not use it, as conda contains more features and resolves dependencies better.

Environment Manager

An environment manager is a tool that creates an environment (sometimes referred to as a virtual environment), a completely separate and isolated area of your computer with its own installation of Python and own third-party packages that are independent from any other Python installation on your machine.

1.2 Miniconda

Miniconda is a distribution of Python that includes conda and a small number of other packages that conda depends on.

Python Installation

Python comes preinstalled on macOS and most Linux operating systems, but it is not a good idea to use this preinstalled version for development as some critical system software might rely on it. It's also usually not the latest version of the software. Instead, we will install a completely new version of Python in a different location. There are many ways to get Python such as directly from Python.org, but for this book we will obtain it through the Miniconda distribution, which will also install conda.

Miniconda Download

DOWNLOAD MINICONDA HERE - Choose the installation for your operating system. Both Windows and macOS have graphical installers (.pkg file for macOS). macOS and Linux both have command line installers (.sh file). There will be instructions for both the graphical and command line installers, so you can choose either one.

Many people will be aware of Anaconda, a different distribution of Python that contains hundreds of packages and other software. Both Anaconda and Miniconda are maintained by a company called Anaconda (formerly known as Continuum Analytics). They both install the same version of Python and conda. The reason I recommend installing Miniconda is that Anaconda installs many unnecessary packages and software that you will likely never use. This extra 'bloat' will amount to around 2GB of extra hard disk space.

Any extra software provided by Anaconda may be installed from conda, so you needn't worry that you might be missing some special piece of software. You can get any software provided in the Anaconda distribution at a later date.

Keeping Anaconda

If you already have Anaconda you can either uninstall it or keep it. Even if you are happy with the current status, you might consider uninstalling it as there is quite a lot of excess software and it does not take too much effort to get a minimal clean installation. Use the next section to uninstall Anaconda.

If you do not want to uninstall Anaconda, then do not install Miniconda. Instead, run the following from the command line:

```
conda update conda
```

This will update conda to the latest version which is necessary and important to ensure that your system is set up properly. You will still need to create the environment for this book, so please skip to the section **Creating a new environment just for data analysis**.

Uninstalling Anaconda

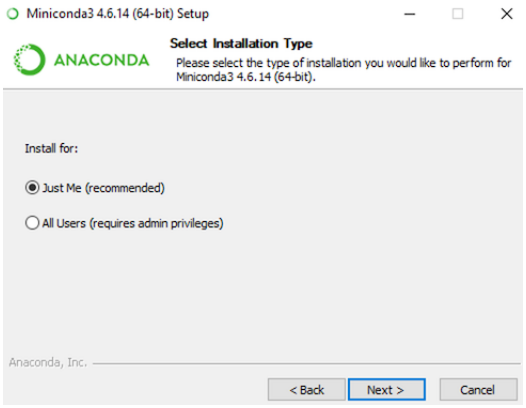
If you wish to uninstall Anaconda, navigate to the official Uninstalling Anaconda page. For macOS and Linux, use option B first and then complete the simple remove with option A.

Miniconda Installation

We will now continue with the Miniconda installation assuming you have downloaded the correct file for your operating system.

Windows

The name of the Windows file will begin with **Miniconda3-latest-Windows**. Start the setup and after agreeing to the license you will be given the choice to install for 'Just Me' or 'All Users'. It's best to select 'Just Me' as this will give you full control of the installation without needing to have administrator rights to install new packages.

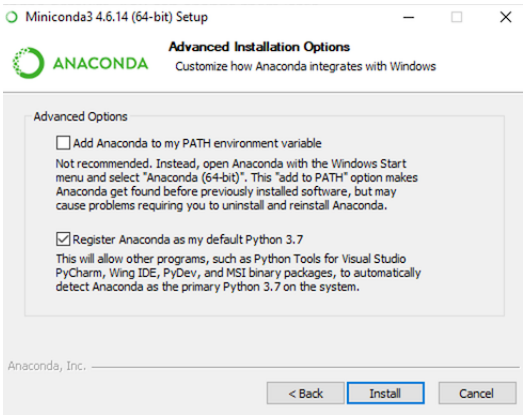


The next step will ask you for a file location for the installation. You should select the default location which will be `C:\Users\<UserName>\Miniconda3` where `<UserName>` is the name of your user folder.

Add to PATH?

The next screen asks whether you'd like to add Anaconda (should say Miniconda) to the PATH and recommends that you do not do so. The main advantage of adding Miniconda to the path is to have access to it directly from the Command Prompt program. This is unnecessary as Miniconda provides a small program called **Anaconda Prompt** that does add the necessary file location to the PATH.

Keep the defaults as they are and complete the installation.



macOS

Graphical Installer

The graphical installer file will begin with **Miniconda3-latest-MacOSX** and end in **.pkg**. After agreeing to the license, you will be asked to choose to either 'Install for me only' or 'Install on a specific disk'. Choose to 'Install for me only' and select the default location which is `/Users/<UserName>/Miniconda3` where `<UserName>` is your specific user name. This will complete the installation.

Command Line Installer

The command line installer will end in **.sh**. Open up your terminal and navigate to the location of where you downloaded the installer and then run the following command. Make sure to use `bash` regardless of the shell you are using.

```
bash Miniconda3-latest-MacOSX-x86_64.sh
```

This will start a stream of text that you'll need to press **enter** to move through. You'll be prompted to agree to the license and whether to accept the default location for the installation which will be `/Users/<UserName>/Miniconda3`. Press ENTER exactly **once**. The installation will appear to have paused and tempt you to press enter again. Do not do this. Instead, just wait patiently.

You will then be prompted to initialize Miniconda3. Enter 'yes' to complete the installation. Exit out of the shell.

Linux

If you aren't able to use a web browser should be able to download the installer with the following command:

```
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
```

To begin the installation run the following command making sure to use `bash` regardless of the shell you are using.

```
bash Miniconda3-latest-Linux-x86_64.sh
```

This will start a stream of text that you'll need to press **enter** to move through. You'll be prompted to agree to the license and whether to accept the default location for the installation which will be `/home/<UserName>/miniconda3`. Press ENTER exactly **once**. The installation will appear to have paused and tempt you to press enter again. Do not do this. Instead, just wait patiently.

You will then be prompted to initialize Miniconda3. Enter 'yes' to complete the installation. Exit out of the shell.

1.3 Python and Conda installation complete

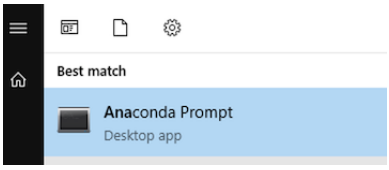
After completing the steps above, you will have finished installing both Python and conda along with a small number of other Python packages.

Test Installation

Let's test that we have a successful installation.

Windows Users

Windows users must always begin by starting the program **Anaconda Prompt**. This program is easily found by tapping the windows key and then typing in the exact name 'Anaconda Prompt'.



macOS/Linux Users

Start your terminal program.

All users

Run the command `conda list` which will return the name, version, build number, and channel for each package currently installed.

```
# Name                    Version                   Build  Channel
asn1crypto                0.24.0                    py37_0
ca-certificates           2019.1.23                      0
certifi                   2019.3.9                  py37_0
cffi                      1.12.2           py37h2e261b9_1
chardet                   3.0.4                     py37_1
conda                     4.6.14                    py37_0
cryptography              2.6.1            py37h1ba5d50_0
idna                      2.8                       py37_0
libedit                   3.1.20181209         hc058e9b_0
libffi                    3.2.1                hd88cf55_4
libgcc-ng                 8.2.0                hdf63c60_1
libstdcxx-ng              8.2.0                hdf63c60_1
ncurses                   6.1                  he6710b0_1
openssl                   1.1.1b               h7b6447c_1
pip                       19.0.3                    py37_0
pycosat                   0.6.3            py37h14c3975_0
pycparser                 2.19                      py37_0
pyopenssl                 19.0.0                    py37_0
pysocks                   1.6.8                     py37_0
python                    3.7.3                h0371630_0
```

This is a list of all the packages that come installed with the default Miniconda distribution. Notice that Python is considered a package. This might appear to be a large number of packages, but it is a fraction of what is installed with the full Anaconda distribution.

The PATH

All operating systems have something called a **PATH** which is a list of directories that the operating system looks through in order to find executable programs. The directories in Windows paths are separated by a semi-colon, while macOS and Linux directories are separated by a colon. Let's take a look at the path now.

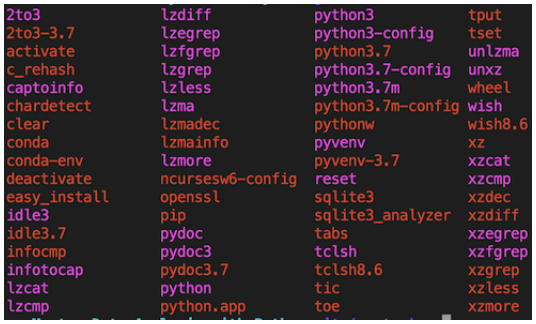
- Windows users - `echo %PATH%`
- macOS/Linux - echo $PATH

All operating systems should have the following two directories in their path that end with the following.

- `miniconda3/bin`
- `miniconda3/condabin`

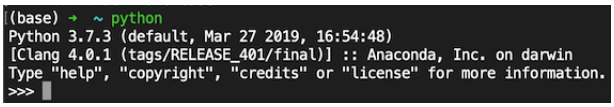
It is the executable programs within these folders that are available for us to use on the command line. Listing out these programs on my macOS machine yields the following:

```
ls /Users/Ted/miniconda3/bin
```



Verifying the Installation

By default, Miniconda creates an environment with the name 'base' that has all the packages displayed from `conda list` installed. Notice that '(base)' has been prepended to the prompt to indicate that this environment is active. When an environment is active, it means that your Python code will be interpreted by the Python executable in that environment. Let's verify this by starting a Python interpreter by running the command `python`.



We can verify the location of the Python executable by importing the `sys` library and fetching the `executable` attribute.

While it is possible to get started doing data analysis within this environment, I recommend creating a completely separate environment.

A note on directory path

For the rest of this tutorial, I will use `miniconda3/bin` and `miniconda3/condabin` as shortcuts referring to the full path to these locations. If you followed the installation instructions from above, the full path will look similar on all operating systems. The full path for `miniconda3/bin` will be one of the following:

- Windows - `C:\Users\<UserName>\miniconda3\bin`
- macOS - `/Users/<UserName>/miniconda3/bin`
- Linux - `/home/<UserName>/miniconda3/bin`

Deactivating the base environment

By default, the base environment will always be active upon opening the terminal. Specifically, the `miniconda3/bin` directory will be added to your path and allow you to start Python and all the other programs listed above. In my opinion, this isn't good practice and its better to explicitly activate the environment.

We can change conda's configuration settings so that it does not automatically activate the base environment upon opening of the terminal. On the command line run the following:

```
conda config --set auto_activate_base false
```

Exit the shell, re-enter it, and output the path again. You should notice that only the `miniconda3/condabin` directory remains. The `miniconda3/bin` is no longer in the path. This command seems to have no effect on Windows Anaconda Prompt. Windows users will have to manually deactivate their base environment with `conda deactivate`. Also, notice that '(base)' is no longer prepended to the prompt.

Activating the base environment

You can reactivate the base environment with the following command:

```
conda activate base
```

This will prepend the `miniconda3/bin` directory back to your path and add '(base)' to the prompt. You can deactivate it again with the command:

```
conda deactivate
```

1.4 Creating a new environment just for data analysis

While it's possible to use the base environment to do all of our data science work, we will instead create a new environment where all of the packages are installed from the conda-forge channel. But, before we do that, it's important to understand what a conda channel is.

Conda Channels

A conda **channel** is simply a repository of Python packages. There are dozens (if not hundreds) of channels available each with their own collection of Python packages. Whenever you install a new package using conda, its contents will come from exactly one channel. By default, conda will only install from the **defaults** channel. You can verify that the defaults channel is the only one available by running the following command:

```
conda config --show channels
```

```
→  ~ conda config --show channels
channels:
  - defaults
```

All channels have at least one URL available where the repository is located. You can find these URLs with the following command:

```
conda info
```

```
channel URLs : https://repo.anaconda.com/pkgs/main/osx-64
               https://repo.anaconda.com/pkgs/main/noarch
               https://repo.anaconda.com/pkgs/free/osx-64
               https://repo.anaconda.com/pkgs/free/noarch
               https://repo.anaconda.com/pkgs/r/osx-64
               https://repo.anaconda.com/pkgs/r/noarch
```

The above results are from my macOS. Linux and Windows channel URLS will look very similar. Notice that there are multiple URLs for this one channel. There's even a URL for R packages, which seems bizarre, but conda is not a tool just for managing Python packages. It is a general purpose package manager that can work with any other programming language.

Navigate to one of the URLS in a browser and you will see a list of the packages available to download.

main/osx-64

Filename	Size	Last Modified	MD5
repodata.json	14.1 MB	2019-06-04 15:21:34 +0000	1b81c33488a72877b1b19f2a9c4f9603
repodata.json.bz2	1.9 MB	2019-06-04 15:21:41 +0000	bb149be4e0087561d6037ca427618403
repodata_from_packages.json	14.0 MB	2019-06-04 14:38:05 +0000	de2f67e683562ca631100d6e5e3f9aa6
repodata_from_packages.json.bz2	1.9 MB	2019-06-04 14:38:09 +0000	0314a9d8491d8d78c2e5442d2b171d99
repodata2.json	17.8 MB	2019-06-04 15:21:58 +0000	307ae25f46a43bf6c42e8b9c0401b582
patch_instructions.json	513 KB	2019-06-04 15:21:32 +0000	b270aaf5b5105326e6fa5bbbacb5848d
_mutex_mxnet-0.0.40-mkl.tar.bz2	2 KB	2018-08-04 10:28:43 +0000	6f8fedc693d635a50cc7145a2ff58d83
_py-xgboost-mutex-2.0-cpu_0.tar.bz2	8 KB	2018-06-28 20:40:43 +0000	94ca8f3a619747af1299df17a8eab661
_tflow_1100_select-0.0.2-eigen.tar.bz2	3 KB	2018-08-24 18:53:19 +0000	27601f8af8dd845b2baccd1690f0164d
_tflow_190_select-0.0.2-eigen.tar.bz2	3 KB	2018-08-10 15:00:09 +0000	4d517003d20e272d47b0c533a50fcaa1
_tflow_190_select-0.0.3-mkl.tar.bz2	3 KB	2018-08-10 15:00:05 +0000	265b9000cc25e3db93978539f25722d7
_tflow_select-2.2.0-eigen.tar.bz2	3 KB	2018-10-05 04:04:31 +0000	34203ba34d2a42af01b219940dfc4f4a
_tflow_select-2.3.0-mkl.tar.bz2	3 KB	2018-10-05 04:04:28 +0000	7d90066b8c47991ae068a27073d8c623
anaconda-docs-2.0.18-hb3a96d1_0.tar.bz2	54.9 MB	2017-11-16 22:07:52 +0000	2907791bc0c93962e7c618667506b623
anaconda-docs-2.0.19-hb3a96d1_0.tar.bz2	57.8 MB	2017-11-30 21:36:45 +0000	933d590ca33f1f672aa0c878f644d199
anaconda-docs-2.0.20-0.tar.bz2	46.0 MB	2017-12-18 20:19:04 +0000	44d45b06e9cf393d1d811e6546b0982b
anaconda-docs-2.0.21-0.tar.bz2	48.7 MB	2018-01-25 20:19:24 +0000	12140f2ca95ca384607a64f43a4f8250
anaconda-docs-2.0.22-0.tar.bz2	48.2 MB	2018-02-08 18:44:07 +0000	c988fda31cb8c4a6d782e39e8787708c
anaconda-docs-2.0.23-0.tar.bz2	53.8 MB	2018-03-09 22:59:21 +0000	bc24546a055bb0e763adfdccc03a67e8
anaconda-oss-docs-1.0.1-0.tar.bz2	246.2 MB	2018-01-25 20:52:46 +0000	e1ef8b3f69c3bc24a77d3e9cebc45961
anaconda-oss-docs-1.0.2-0.tar.bz2	286.0 MB	2018-03-09 21:51:34 +0000	2fc19cdf202611606e0d5de5107120ec
apr-1.6.3-he795440_0.tar.bz2	1.4 MB	2017-12-13 17:10:56 +0000	50d668c7859ba9babfb42c85e643a446
basemap-data-hires-1.2.0-0.tar.bz2	105.3 MB	2018-10-10 15:21:03 +0000	b5d2c8fbf9b464d362f18fc3d6f46b6a
binsort-0.4_1-h1de35cc_0.tar.bz2	151 KB	2018-10-11 20:24:58 +0000	ecb787779b7a7d6cb916da9dd15ceef8

The defaults channel contains a hand-picked list from the team at Anaconda of popular and powerful packages to do scientific computing. However, there are many thousands of packages that exist that are not available in the defaults channel. This is where the conda-forge channel becomes important.

The conda-forge channel

Anaconda the company allows anyone to create a channel and will host these packages in the Anaconda Cloud. You can create an account right now and start your own channel with your specific collection of packages.

conda-forge is the most popular channel outside of the defaults and contains many more packages. My recommendation, at the time of this writing, is to install packages only from conda-forge if possible and not from the defaults. The reasons for this are described in the conda-forge documentation reprinted below:

- all packages are shared in a single channel named conda-forge
- care is taken that all packages are up-to-date
- common standards ensure that all packages have compatible versions
- by default, we build packages for macOS, linux amd64 and windows amd64
- many packages are updated by multiple maintainers with an easy option to become a maintainer
- an active core developer team is trying to also maintain abandoned packages

One of the main reasons to use a single channel such as conda-forge is the consistency it provides with package compatibility. For packages that have components written in a compiled language like C, compatibility improves when they are all compiled from the same base C library.

1.5 Create a new environment

It's finally time that we create our new environment that we will use for data science. There are a few different ways to successfully accomplish this. One way will be shown now, with other alternative ways shown later on.

Create an empty environment

Let's create an environment with the name 'minimal_ds' that has no packages in it, not even Python.

```
conda create -n minimal_ds
```

Confirm the creation and notice that its location in your file system will be `miniconda3/envs/minimal_ds`. Any downloads for the environment will be located here. Activate the environment with:

```
conda activate minimal_ds
```

By default, this environment will install packages from the defaults channel.

Add the conda-forge channel

Let's add the conda-forge channel as an option for just this environment. The `--env` option ensures that conda-forge is added only to our currently active environment.

```
conda config --env --add channels conda-forge
```

Running this command will create a configuration file named `.condarc` in the environment's home directory (`miniconda3/envs/minimal_ds`). You can verify this by outputting its contents to the screen.

```
cat miniconda3/envs/minimal_ds/.condarc
```

Instead of outputting the configuration file's contents, we can use the following command which will show the same thing.

```
conda config --show channels
```

```
channels:
  - conda-forge
  - defaults
```

Adding a channel will not remove any previous channels. Instead, it will become the first channel that conda looks to find packages. Currently, if it cannot find a package in conda-forge it will then look in defaults for it. But, if the same package exists in both, then it will choose to install it from the channel with the newest version. For instance, if conda-forge has pandas version 0.23 and the defaults has version 0.24 then conda will install pandas 0.24 from the defaults channel.

This behavior is unintuitive to me and it makes more sense to always use the channel that appears first in the channels list regardless of the version. Conda gives us a way to change this with the following command:

```
conda config --env --set channel_priority strict
```

Let's verify the configuration change.

```
conda config --show channel_priority
```

The `.condarc` file has also been updated with the same information. Changing this setting will cause conda to always install packages from conda-forge unless they don't exist in it at all and then look to the defaults channel. There are some packages that only exist on the defaults channel.

This behavior is changing when conda 4.7 is released. At the time of this writing, conda is on version 4.6. The channel_priority variable will be defaulted to 'strict' so this step can be skipped if you are on version 4.7 or later. Check your version of conda by running `conda -V`.

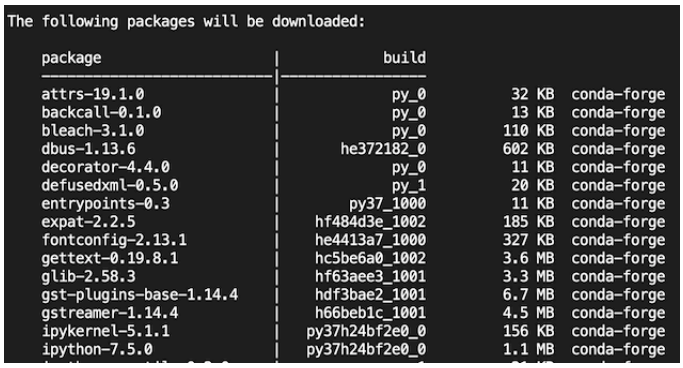
Installing the packages

We are finally ready to install packages into our new environment. Conda will only look in the conda-forge channel unless a package is missing and then turn to defaults.

Personally, a minimal data science environment has numpy, scipy, pandas, scikit-learn, and matplotlib along with the newest stable version of Python (which is 3.7 at the time of this writing). It will also have jupyter notebooks available. Let's install these packages now.

```
conda install pandas scikit-learn matplotlib notebook
```

Notice that Python, numpy, and scipy weren't explicitly included in the list of packages to install. These packages are dependencies of at least one of the included packages and will be installed along with many other dependencies. Let's take a look at the packages to be installed before confirming.



```
The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    attrs-19.1.0               |             py_0          32 KB  conda-forge
    backcall-0.1.0             |             py_0          13 KB  conda-forge
    bleach-3.1.0               |             py_0         110 KB  conda-forge
    dbus-1.13.6                |       he372182_0         602 KB  conda-forge
    decorator-4.4.0            |             py_0          11 KB  conda-forge
    defusedxml-0.5.0           |             py_1          20 KB  conda-forge
    entrypoints-0.3            |       py37_1000          11 KB  conda-forge
    expat-2.2.5                |    hf484d3e_1002         185 KB  conda-forge
    fontconfig-2.13.1          |    he4413a7_1000         327 KB  conda-forge
    gettext-0.19.8.1           |    hc5be6a0_1002         3.6 MB  conda-forge
    glib-2.58.3                |    hf63aee3_1001         3.3 MB  conda-forge
    gst-plugins-base-1.14.4    |    hdf3bae2_1001         6.7 MB  conda-forge
    gstreamer-1.14.4           |    h66beb1c_1001         4.5 MB  conda-forge
    ipykernel-5.1.1            |   py37h24bf2e0_0         156 KB  conda-forge
    ipython-7.5.0              |   py37h24bf2e0_0         1.1 MB  conda-forge
```

Conveniently, this list shows the name of the package, the version number, the size, and the channel. If the last column has a blank value, it indicates that it is being sourced from the defaults channel.

Other packages to install

There are a few other packages used during the book that you will want to install. Specifically, we will install seaborn, a visualization library, sqlalchemy (a library to connect to SQL databases), and jupyter_contrib_nbextensions, a library for adding more functionality to the notebook. They are available on conda-forge. Make sure the minimal_ds is activated before running the following install command.

```
conda install seaborn sqlalchemy jupyter_contrib_nbextensions
```

Only installing from conda-forge

Our current setup allows for packages not found on conda-forge to be searched for on defaults. It may improve compatibility issues to only use the conda-forge channel. To ensure that packages only come from conda-forge, you'll have to specify the channel name (with the `c` option) together with the `--override-channels` option.

```
conda install -c conda-forge --override-channels <package_name>
```

Conda always asks for confirmation of the installation after showing you the plan, which allows you to verify the channel and package versions before proceeding.

Installing packages not in conda-forge or defaults

As discussed, if a package does not exist in conda-forge, it will be searched for in the defaults channel. If it does not exist in the defaults channel, then the installation will fail with an error message. You can specify a different channel to use, as long as you know its name. The easiest way to find the channel name of a package is to visit anaconda.org and search for it at the top of the page.

For instance, plotly_express is not available in either conda-forge or defaults. Searching for it on anaconda.org reveals its channel as plotly. Let's install it by specifying the channel.

```
conda install -c plotly plotly_express
```

Note that the package plotly is a dependency and will also be installed from the plotly channel. If you search for the package plotly, you will see that it is available on conda-forge, but in this instance will be installed from the plotly channel and not with conda-forge. Any channels provided to the `c` option will take precedence over the channels in the `.condarc` file.

Installing Quandl for financial data with pip

A few times within the book, we will read financial data using Quandl package. Although this package is available on both conda-forge, one of it's dependencies has not been updated at the time of this writing. Installing it from the defaults channel results in a host of other compatibility issues.

Instead, we will install it with **pip**, the original package manager for Python. Make sure minimal_ds is activated and run the following command:

```
pip install quandl
```

Quandl registration and API key

Once it's installed, you'll need to register for an account at the Quandl home page. Once, you've registered for an account, visit your account profile and copy your API key. Paste the key in the file `api_key.txt` which is in the top-level directory for this book. It should be the only text in the file. Now, you'll be able to run any code in the book that uses the quandl package.

Environment Setup Complete

Your environment should now be set up correctly to do data science using Python for this book. It contains a minimal number of the most common and useful Python packages and will use conda-forge as its primary channel.

1.6 Other Considerations

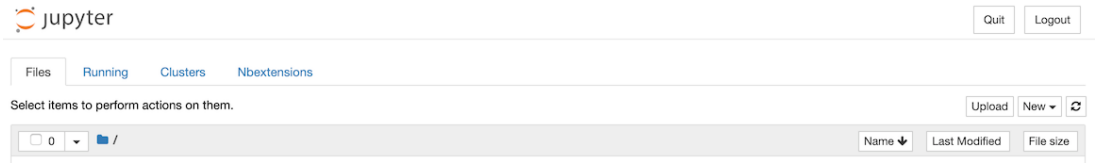
There are a few other items that are worthy of discussion not mentioned above.

Verify that Jupyter Notebooks execute in correct environment

Although we have created our own environment with the ability to create Jupyter Notebooks, we are not guaranteed that they will execute Python in the same environment that they were launched. For instance, if we launch a Jupyter Notebook from the minimal_ds environment, it is possible that we are executing Python from the base environment. This is quite surprising behavior as you would expect environments to be isolated from one another, but this isn't quite the case.

We need to verify that executing notebooks launched from the minimal_ds environment execute Python from the minimal_ds environment and not from anywhere else.

With the minimal_ds environment activated, run the command `jupyter notebook`. This will start a Jupyter Notebook server running from your localhost at the default port 8888. Your default web browser should open up with the top of that page looking like the following image.



Click the **New** button on the right-hand-side of the page and start a new 'Python 3' notebook. In the first cell of the notebook, write the the following two lines of code:

```
import sys
sys.executable
```

Execute this code by pressing **shift + enter**. The result should return a path to the environment Python (`miniconda3/envs/minimal_ds/bin`). If it returns the location for the base environment (`miniconda3/bin`) then you aren't executing Python from your environment.

The cause of this is a 'User' kernel that is masking the environment kernel. Exit out of the Jupyter Notebook by pressing `ctrl + c + c` in your terminal. Then run the following command to see the list of kernels:

```
jupyter kernelspec list
```

Check to see if your python3 kernel is indeed a User kernel. Find the default locations for User kernels for your OS. The User kernel has the highest precedence over the Environment and System kernels. Yes, that's right, even when you have an active environment, the User kernel takes precedence and allows you to execute Python from other environments.

If you indeed have a User kernel, then I recommend removing it with the following command:

```
jupyter kernelspec remove python3
```

Rerun the command `jupyter kernelspec list` and you will see the Environment kernel with the same name (python3). You should now launch another Jupyter Notebook and verify that the Python executable is located in the active environment.

Alternative Environment Creation

Above, we created an empty environment first and then installed the packages with a separate command. We did this so that we could add the conda-forge channel and set its priority. It is possible to do this in a single step.

```
conda create -n minimal_ds -c conda-forge --strict-channel-priority pandas scikit-learn matplotlib notebook
```

An issue with this method is that no configuration file for the environment is created. You would have to specify the option to use the conda-forge channel and strict channel priority for each new package installed. Of course, you could set the configuration file after the package installation.

Another method for creating an environment is with a text file usually given the name `environment.yml`. The contents of the file contain the name, channel(s) and packages. The contents of the file that would have created our environment would look like this:

```
name: minimal_ds
channels:
  - conda-forge
  - plotly
dependencies:
- pandas
- scikit-learn
- matplotlib
- notebook
- seaborn
- sqlalchemy
- jupyter_contrib_nbextensions
- plotly_express
- pip:
  - quandl
```

We would then run the command:

```
conda env create -f environment.yml
```

A major issue with this method is that there is no (current) way to set the channel_priority to strict and like the previous one no configuration file for the environment will be created.

Updating packages

All the popular data science packages are under constant development and make new releases from time to time. You can update all of the packages at once with the following command:

```
conda update --all
```

Before the update happens, conda will show you a list of all the packages that it will update. This allows you to view all the latest versions of each package and make a decision on whether to update or not. To update an individual package run `conda update packagename`.

Updating conda

It's also important to update the tool conda itself from time to time. Before updating conda, you'll need to know if you installed conda in the current environment. If you followed the instructions on this page, then you won't have conda installed in the minimal_ds environment. It'll only be installed in the `Miniconda/condabin` directory.

Therefore, to update conda, deactivate the minimal_ds environment with `conda deactivate`. Then run:

```
conda update conda
```

If you do have conda installed in your environment then you can update both installations with the same command. Just activate the environment first and rerun the above command.

1.7 Summary of Steps

There were a lot of text that separated the steps in this tutorial, so below is a summary of just the commands.

1. Install Miniconda for your OS with the default settings
2. Prevent the base environment from automatically activating `conda config --set auto_activate_base false`
3. Create an empty environment `conda create -n minimal_ds`
4. Activate the environment `conda activate minimal_ds`
5. Add conda-forge as first channel `conda config --env --add channels conda-forge`
6. Ensure that conda-forge is used if the package is available `conda config --env --set channel_priority strict` (Not necessary for conda version 4.7+)
7. Install packages `conda install pandas scikit-learn matplotlib notebook seaborn jupyter_contrib_nbextensions`
8. Install packages not in conda-forge/defaults `conda install -c plotly plotly_express`
9. Install Quandl with `pip install quandl`

1.8 Using the book with Jupyter Notebooks

In my opinion, the best way to work through the contents of this book is within Jupyter Notebooks. They allow you to read the material, run the code, and write solutions to the exercises and projects all in one place.

If you'd like to use the Jupyter Notebooks for this book, make sure that you have activated the minimal_ds environment and have navigated to the directory where the contents of this file are located. By default, the folder is titled, "Master Data Analysis with Python by Ted Petrou''. Launch the Jupyter Notebook with the following command:

```
jupyter notebook
```

This will start a Jupyter Notebook server and open up a new tab in your web browser. The contents of your file system of the directory where you ran the `jupyter notebook` command will be shown on the page in your browser.

All the notebooks for the book are contained in a folder titled **Jupyter Notebooks**. This document is available as the first notebook in the **Environment Setup and Jupyter Notebooks** folder titled **01. Installing Python and Setting up an Environment for Data Science**. To launch a specific notebook, simply click its title. All notebooks have file names that end in **.ipynb** which references their origins as IPython Notebooks. New notebooks will open in their own browser tab. You may now move on to the second chapter, **02. Introduction to Jupyter Notebooks**.

Chapter 2

Introduction to Jupyter Notebooks

2.1 Jupyter Notebook Basics

Where am I?

If you have never seen this application, then you might be wondering what strange place you have landed on the internet. You are now running a **Jupyter Notebook** server from your localhost at default port 8888. This specific notebook is connected to an IPython kernel capable of executing Python commands.

What are Jupyter Notebooks and why are we using them?

Jupyter Notebooks allow you to interactively write code, visualize results, and enhance the output via markdown, HTML, LaTeX, and other rich content streams like this Monte Python image below.

Monte Python!

We are using them because they provide an excellent means to interactively and iteratively explore and visualize data in one place. They are also great for providing space for exercises.

What's the difference between IPython notebooks and Jupyter Notebooks?

IPython Notebook was the original name for these notebooks. IPython itself, is an interactive Python shell that was created by Fernando Perez in 2001. You can run the IPython shell by simply opening a terminal (activating your minimal_ds environment) and running the command `ipython`. IPython Notebooks, created around 2011, gave IPython the web interface we use today. The idea of a web-based notebook was not new. They were already a popular way to do interactive computing in other languages like Maple or Mathematica.

At first, IPython Notebooks only supported the execution of Python. As IPython Notebooks started to rise in popularity, developers started creating kernels capable of executing other languages. You can see a list of all the backend execution kernels currently available. In 2015, the project was renamed Jupyter, a name derived from the languages Julia, Python, and R. This 'big split' resulted in the language agnostic bits separated from the actual execution of the code. As it turns out, lots of people enjoy working in this type of an environment and hopefully you will too.

2.2 Getting Started with Jupyter Notebooks

It is very important that you understand the fundamentals of how to use Jupyter Notebooks. They are one of the most popular environments to explore data with and are the environment where all the material for this book was written. The remaining content in this chapter detail the very basics of how to use the Jupyter Notebook.

All notebooks are composed of cells

Jupyter Notebooks are composed of **cells**. Every bit of content within a Jupyter Notebook, including this text, resides within a **cell**.

Markdown and Code cells

There are two primary types of cells - **Markdown** and **Code** cells. The cell containing this text is a markdown cell. Markdown is a simple plain-text language that allows you to add some basic styles and links with little effort. Markdown is easy to learn with only about a dozen common commands. A brief tutorial is provided in a later chapter.

For nearly all of the book, you will be working inside **Code** cells. Code cells are where you write and execute your Python code. Code cells only understand Python code.

Identifying Markdown and Code Cells

It is easy to identify whether you are in a markdown or code cell. Click anywhere on this text exactly **once**. The outside border of the cell will turn **blue**. The word **Markdown** will also appear in the dropdown box in the center of the menu at the top of this page.

Code cells will have the word **Code** in that same menu. Also, all code cells will have the word `In` displayed directly to the left of the cell and a number inside of brackets denoting its order of execution. If there is no number inside the brackets, then the code cell has yet to be executed. If the cell has been executed, the result will be displayed directly below with the word `Out` directly to the left. Some code cells do not output anything upon execution and will not have an `Out` section.

```
In [1]:

Out[1]:
```

Edit Mode vs Command Mode

Jupyter Notebook cells are always in one of two modes: **edit** or **command** mode. When in edit mode:

- Cells are outlined in **green**
- Cells have a blinking cursor in them
- Typing results in plain text inside the cell
- A small pencil icon is visible in the top right-hand-side of the menu bar

When in command mode:

- Cells are outlined in **blue**
- Cells do not have a blinking cursor and there is nowhere to type
- The keys have special meanings
- No small pencil icon visible in the top right-hand-side of the menu bar

Selected Cells

There is always at least one cell in a notebook that is **selected**. A selected cell will be outlined in either green or blue depending on what mode it is in. Clicking on a cell is one way to select it.

Changing between edit and command Mode

The very first task we will learn is how to change between edit and command mode. This is a common, valuable, and simple task. Let's practice changing from edit mode to command mode within a code cell.

Click anywhere in the code cell below. It has a Python comment in green text. Clicking in the cell places you in edit mode. You should see a blinking cursor, a green outline around the cell, and will be able to type text directly into it.

Press **esc** once to enter **command** mode. The cell outline will turn blue and the blinking cursor will have disappeared. From command mode, press **enter** to go back into edit mode. Practice changing from edit to command mode several times by alternating between pressing **esc** and **enter**.

[1]:
```
# I am a code cell. Click here.
```

Changing modes in a markdown cell

We will now practice changing between edit and command mode in a markdown cell. Click **this text** in this markdown cell once to select it. It should be outlined in blue, meaning it is in command mode. Change to **edit** mode by pressing **enter** (or double-clicking it).

By changing to edit mode, you have revealed the underlying markdown text used to create the nicely formatted page. Press **shift + enter** to execute the markdown and return the cell contents back to the formatted display.

More on Command Mode

While in command mode, the keys are mapped to some specific command, such as inserting a new cell, deleting a cell, etc… You are quite literally *commanding* the notebook to do something. More details on command mode will be shown further below.

2.3 Executing Cells

To execute all the lines of Python code within a cell, select it, and then press **shift + enter**. If the last line of code in your cell produces any output, then this value will be displayed in the `Out` section of the cell directly below it.

Execute your first cell

Let's practice this by executing the cell directly below. It is a code cell that adds two numbers together. Select the cell in either edit or command mode and press **shift + enter** to execute it.

[2]:
```
5 + 7
```

[2]:
```
12
```

Executing with ctrl + enter

In addition to the output of the operation, the focus is moved to the next cell (this one), which has been selected in command mode. It is possible to execute a cell without moving the focus to the next cell by pressing **ctrl + enter**. Place your cursor in the next cell and execute it with **ctrl + enter**. You will remain in the current cell but in command mode.

[3]:
```
3 - 10
```

[3]:
```
-7
```

Executing a cell with no output

Not all code cells produce an `Out` section. Only those where the last line in the cell contains output. Run the following cell, which assigns the value 7 to the variable name `a`.

[4]:
```
a = 7
```

Only last line of cell is output

You can write any number of Python statements in a single cell, but only the last line of the cell will be outputted to the screen. Execute the following cell to verify this.

[5]:
```
5 + 7
3 + 6
4 - 2
```

[5]:
```
2
```

Another example with multiple statements but no output

It doesn't matter if statements before the last line produce output. What determines output is the last line of the cell. Execute the following cell which produces no output.

[6]:
```
5 + 7
3 + 6
a = 4
```

An exception with the `print` function

It is possible to produce output from lines that are not the last with the `print` function. Notice how the results of the first two statements are printed below the cell with the output of the last line still in its own `Out` section.

[7]:
```
print(5 + 7)
print(3 + 6)
4 - 2
```

```
12
9
```

[7]:
```
2
```

2.4 Keyboard Shortcuts

There are a few keyboard shortcuts that will make your life working in a Jupyter Notebook much easier. Keyboard shortcuts exist both in command and edit mode and in fact, we have already issued some of them. Pressing **enter** in command mode is the keyboard shortcut to switch to edit mode. Pressing **esc** while in edit mode is the shortcut for returning to command mode. Cell execution with **shift/ctrl + enter** is a keyboard shortcut that works in both command and edit mode.

Inserting cells

Inserting cells above or below the currently selected cell is a common task that you can perform in command mode. To begin, select a cell and activate command mode by pressing **esc** once (you don't need to hold it). Press **A** to insert a cell above and B to insert a cell below the selected cell. Practice this now by inserting cells above and below the cell directly below.

Select this cell. Activate command mode (press **esc**) and then insert cells above by pressing **A** and below by pressing B.

[ ]:

Deleting Cells

It's often that you'll need to delete a cell. This is done in command mode by pressing **D** twice. Practice deleting some of the cells you just inserted above.

Using the menu to insert and delete cells

Cell insertion and deletion can also be done through the menu at the top of the screen. The **Insert** menu handles insertion and the **Edit** menu has an option for deletion.

Keep your hands on the keyboard - Use the shortcuts

I strongly recommend using the keyboard shortcuts instead of the menu commands. Keeping your hands on the keyboard allows you to focus on completing tasks instead of reaching for the mouse. They shortcuts will save you lots of time. There are only about a half-dozen commands that make up nearly all of my keyboard shortcut usage, so it should not be hard to memorize.

Changing between Code and Markdown cells

By default, all cells are code cells. To change the type of cell, select the cell in command mode and press M to change it to markdown. When in a markdown cell, press **Y** to change it back to a code cell. You can also change cell types from the dropdown menu at the top of the screen or from the Cell menu.

Raw Cells

You'll notice in both the dropdown menu above and from the **Cell** menu that you can change the cell type to a **raw** cell. This is an additional type of cell that has no special functionality. It is simply raw text. These cells can be used to contain code during notebook conversion to PDF or HTML through a library called NBConvert.

Other keyboard shortcuts

There are many other keyboard shortcuts available, but the ones covered above should represent the vast majority of your usage. To find all the other shortcuts, activate command mode and press **H** or find them in the Help menu.

Stopping Execution

Occasionally, you will write code that creates an infinite loop or just takes too long to complete. You will see a star within the brackets like this (`In [*]`) when a cell is being executed.

To stop execution, press the stop button ( ) above or from the **Kernel** menu above select **interrupt**. If cell execution does not stop, select **restart** from the **Kernel** menu. All variable and function definitions will be lost when the notebook is restarted. If you are still unable to stop the execution, you need to go back to the terminal where you launched the notebook from and kill it pressing **ctrl + c** twice.

Exiting the browser tab

Exiting the browser tab of where a notebook was open does not shut it down. The notebook will still be running in the same state as it was with all of the variables still in memory. You just no longer can visually see it in the browser. You will be able to reopen it from the Jupyter home page and continue using it just as you were.

If you completely exit out of your browser, you can still get back to your Jupyter home page by navigating to URL `localhost:8888`.

Shutting down Jupyter

To exit completely out of Jupyter, you'll need to exit the program where you launched it from - either a terminal shell or the Anaconda Prompt or by pressing **ctrl + c** twice within that program.

2.5 Completing exercises in this book

This book contains several hundred exercises all written in markdown cells. All exercises have a single code cell following them where you will write the solution. Unfortunately, there is one issue that will appear during every exercise. Whenever you execute the code for your solution, the next cell (the next exercise) will be selected in command mode. While this is fine if you have completed the exercise and want to move on to the next one, it is not ideal if you want to continue writing more code in a new cell.

When you find yourself in this situation (which frequently happens), simply press A to insert a new cell above and then enter to go into edit mode in that new cell to continue writing more code. Practice resolving this situation by executing the code cell below exercise 1 by pressing **shift + enter** and then adding a new cell once you land on top of exercise 2.

Exercise 1

What is 489 times 143?

[ ]:

Exercise 2

Some exercise here that is in the way!

[ ]:

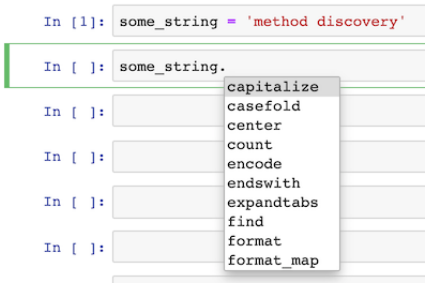
2.6 Getting help in the notebook

In this section, we will cover a couple tools provided within the notebook that help with writing code.

Attribute and method discovery

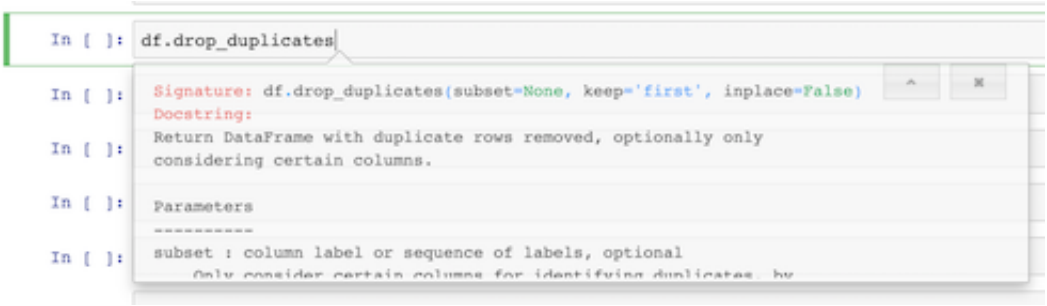
A common difficultly when writing code (in any language) is to remember the syntax available to use. In Python, every object has attributes and methods that provide the functionality. Some objects have dozens or even hundreds of available attributes and methods. Even those who use these objects on a daily basis will have trouble remembering their names.

The notebooks provide a way to discover all these possibilities with a simple process. With your variable name already assigned to an object, place a **dot** after it and then press **tab**. A dropdown, scrollable list of all the possible attributes and methods will appear. The following image depicts this process with a string object.



Reveal the documentation

After you have selected a method to execute from your object, you'll need to know how to use it. With your cursor at the end of the method press **shift + tab + tab** to reveal the documentation. This is a trick I use frequently when working in a Jupyter Notebook. The data science libraries are vast and difficult to retain all the information in working memory. Revealing the documentation in the notebook as I am coding is very helpful. The following image depicts the documentation from a pandas DataFrame method.



I find that being able to discover attributes and methods paired with the ability to reveal the documentation a powerful and useful combination when using Jupyter Notebooks.

Chapter 3

Markdown Tutorial

I believe learning markdown is very valuable as it provides a means to improve the quality of the notebooks you create and even turn them into reports that can be published. In fact, the entirety of this book is written in markdown within Jupyter Notebooks. This tutorial will not cover all of the markdown specification, but instead focus on a few of the most important and useful commands. To learn more markdown, visit the following two resources:

- Markdown Cheatsheet by Adam Pritchard
- Markdown Tutorial by Garen Torikian

3.1 Headers

Headers are larger-sized text that precede a paragraph of text. Headers are created by beginning a line with hash symbols followed by a space and then by the text of the header. Let's see an example:

```
# Some header
```

Writing the above in a markdown cell creates the largest possible header. If you are familiar with HTML, it corresponds with an **H1** header. Adding more hash symbols will decrease the size of the header. For instance, `#### Some header` creates an **H4** header. The smallest header, **H6**, uses six hash symbols.

3.2 Italics and bold

To italicize words, surround them with a single asterisk. For instance, `*some phrase*` would create *some phrase* in markdown.

To make words bold, surround them with a two consecutive asterisks. `**some phrase**` becomes **some phrase**.

3.3 Code formatting

When writing code within text, it's important that is formatted differently in order to read it as code more easily. To format code, wrap the contents in backticks, '. For instance, 'a = 99' renders as `a = 99`. You can write blocks of code on separate lines with three backticks to start and end the block. The following shows an example of how you would write a few lines of code in a block.

```
a = 3
b = 4
c = a + b
```

The above markdown is rendered as:

```
a = 3
b = 4
c = a + b
```

3.4 Lists

It is possible to create both unordered and ordered lists. Unordered lists are created by beginning a line with a single asterisk followed by a space. You can create a sublist by indenting an asterisk. The following markdown creates an unordered list with the second item having its own sublist.

```
* some iterm
* another item
    * sublist first item
    * sublist second item
* yet another item
```

The rendered markdown will look like the following:

- some iterm
- another item
  - sublist first item
  - sublist second item
- yet another item

Ordered lists are created by begging a line with a number followed by a period and a space. The first number determines the starting number of the list. The other numbers do not affect the actual number that is rendered. For instance, the following creates an ordered list from 1 to 3.

```
1. first item
1. second item
1. third item
```

And here is the rendered markdown:

1. first item
2. second item
3. third item

The following also creates an ordered list beginning at 4. Only the first item in the list controls which number is actually rendered.

```
4. first item
10. second item
3. third item
```

The rendered markdown:

4. first item
5. second item
6. third item

3.5 Hyperlinks

It is possible to create a hyperlink to any valid URL. There are two types of links, **reference** and **inline**. For both types, you begin by placing the words you would like to link within square brackets. For reference links, append an additional set of square brackets immediately following these brackets and place the reference text inside. The reference text can be anything, but I typically use numbers. Finally, place the reference text within square brackets on its own line followed by a colon and the URL. I can link to my business's home page like this:

```
[Visit Dunder Data for expert data science training][1]

[1]: https://dunderdata.com
```

The rendered markdown:

Visit Dunder Data for expert data science training

With inline links, you place the link in parentheses immediately following the square brackets like this.

```
[Visit Dunder Data for expert data science training](https://dunderdata.com)
```

Both reference and inline links render the sam.e Although inline links offer slightly less typing, I prefer reference links as the markdown is easier to read. When I use reference links, I organize them by placing them all at the bottom of the cell.

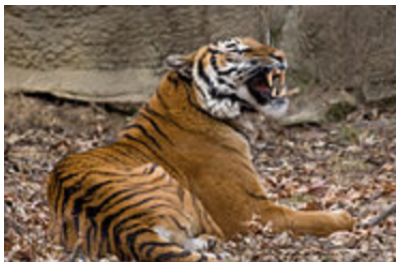
3.6 Images

Adding images is similar to adding hyperlinks. The only difference is that a single exclamation mark, !, precedes the square brackets. The first set of square brackets contains text for the visually impaired, usually referred to as **alt text**. The link to the image can be a URL or relative link to a location in your file system that contains an image. The following is markdown using a reference link to a relative file location.

```
![A tiger lying down][2]

[2]: images/tiger.png
```

Here is the rendered markdown:



A tiger lying down

3.7 New lines

One surprising feature of markdown is how it enforces the creation of new lines. Simply pressing enter at the end of a line will not render a new line. You must end a line with two blank spaces in order for a new line to be rendered. For instance, the following markdown is rendered as one with its output below.

```
This
is
one
line
of
markdown
```

This is one line of markdown

Chapter 4

Jupyter Notebooks More

The Jupyter Notebook has limitations just like any tool and is not the best for every situation. There are certain situations that Jupyter Notebooks excel at and there are others where it fails. Depending on what you are doing, Jupyter may or may not be the appropriate tool.

4.1 Where Jupyter Notebooks Excel

Jupyter Notebooks are excellent for constructing data explorations that incorporate code, visualizations, text, images, videos, and math text. Essentially, they are great at building a report that someone can read and get information from. They are great for blog posts, teaching, formal reports, and interview assignments.

Iterative and interactive workflow

One of the best features of the notebook is that you can run one line of code, get output, and then use this output to run the next line of code. For many scientists, it will be absolutely necessary to process their thinking one line at a time. Writing multiple lines of code in a row just doesn't make sense because there needs to be some analysis or decision on the output of the previous line before continuing.

This type of workflow is **iterative**, because code is constantly being executed, output examined, which leads to code being modified and executed again. It is **interactive** because we are getting results immediately upon running the code.

Using Jupyter as a light-weight Interactive Dashboard

There are a number of interactive widgets that you can embed into the Notebook to give a user control. There are also interactive visualization libraries such as bokeh and plotly that both provide powerful tools for dashboards.

4.2 Where Jupyter Notebooks Fail

You might not like notebooks. Don't fear, you will not be alone. There is no mandatory requirement to use Jupyter Notebooks to do data science. Traditional software development, such as building an application, would not be a time where Jupyter Notebooks would be a good choice. Other programming environments, such as PyCharm or Visual Studio Code provide much better tools to organize, test, write, refactor, debug and version control code.

Notebooks suffer from messiness

It is incredibly easy to make a complete mess of your notebooks. They can become very difficult to follow especially if you do not document what is happening. It is easy to get carried away doing lots of scratch work and creating dozens of code cells that may or may not work.

Out of order code execution

One built-in feature/problem with notebooks is that you can run any cell at any place at any time. In a normal computer program, control flows from one line to the next. With notebooks, you control which cells get executed, so it is easy to execute cells in a different order than their natural ordering. Furthermore, you can lose track of variable values.

Know where Jupyter Notebook development ends and traditional software development begins

I only use Jupyter Notebooks whenever I am exploring data, building a formal report, or creating content. When exploring data, I typically use two notebooks, one for scratch work and another for my formal report. I ensure the formal notebook has the same behavior as a normal program by executing all the cells in order from the top. You can do this by using the option **Restart & Run All** from the **Kernel** menu or with the **Run All** option in the **Cell menu**. I also document my thoughts carefully in markdown cells.

4.3 On-demand Jupyter Notebooks

There are a number of websites that host Jupyter Notebooks that you can use at any time with just a connection to the internet. All of the following sites allow for quick access to a notebook from anywhere in the world.

- Google Colaboratory
- Microsoft Azure
- Project Jupyter Home

Chapter 5

Jupyter Notebook Extensions

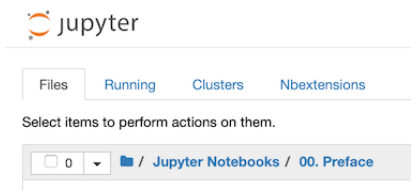
Jupyter Notebooks were designed such that other developers could extend their functionality. A set of Jupyter Notebooks extensions is found in the jupyter_contrib_nbextensions package. Instructions for installing the package are found in the **Installing Python and Setting up an Environment for Data Science** chapter. If you have not yet installed this package, you can do so with a single command:

```
conda install -c conda-forge jupyter_contrib_nbextensions
```

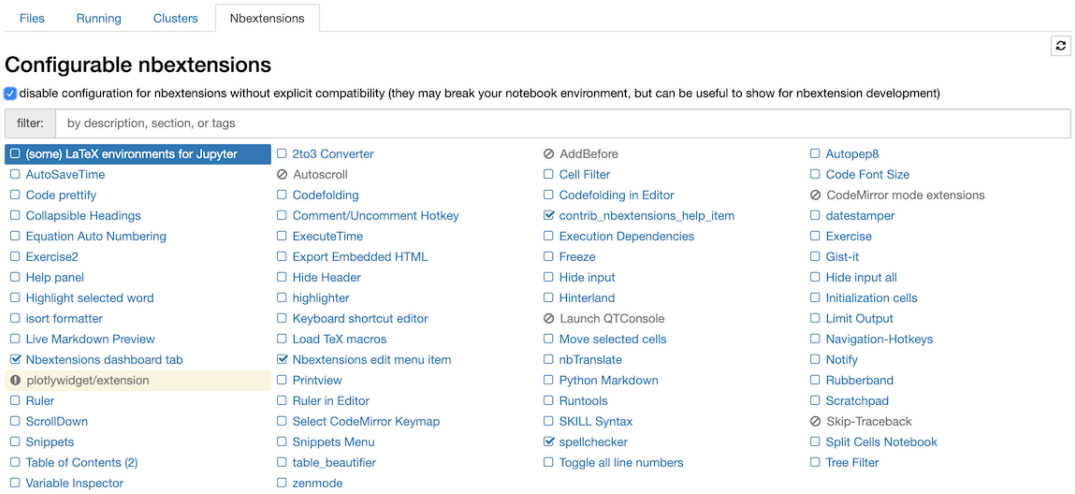
These extensions are contributed by a community not officially related to Jupyter so they may not all work as well as the other components in the notebook. There also might be issues when using multiple different extensions simultaneously, so I advise to use as few as possible at one time.

5.1 The NBextensions tab

If you installed the jupyter_contrib_nbextensions package correctly, you will see a new tab titled **NBExtensions** when you first run Jupyter.



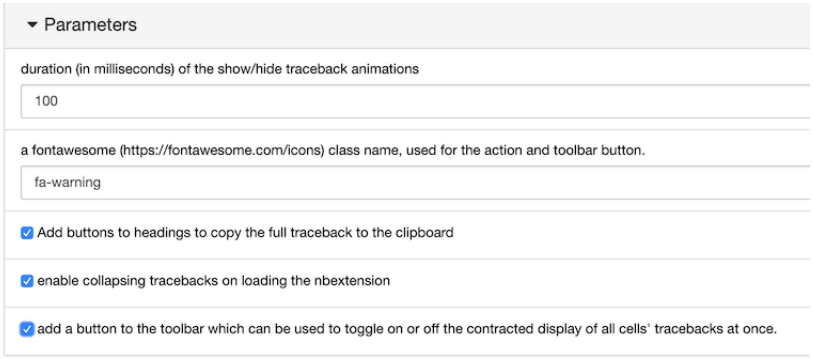
Click on this tab to reveal the several dozen extensions available.



5.2 The Skip-Traceback Extension

I only recommend activating the **Skip-Traceback** extension for this book. You'll notice that it is unable to be selected at the current state. Look at the top-left corner of the page under the **Configurable nbextensions** header. There is a checkbox that warns about possible compatibility issues with the notebook breaking for certain extensions. This is a warning and means that not all of the extensions have been fully tested. I have yet to encounter any person having issue with the Skip-Traceback extension.

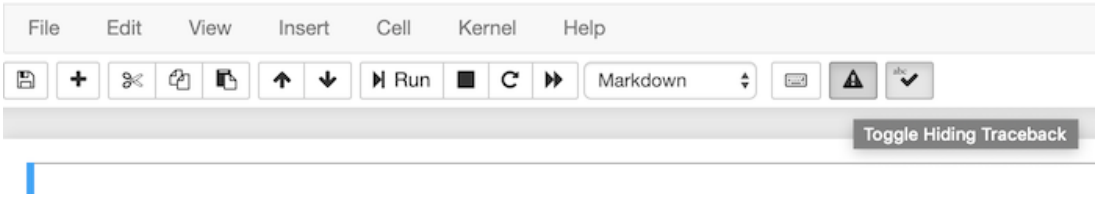
Uncheck the box and then activate the Skip-Traceback extension. Most extensions provide a few options to control their functionality. Scroll down the page and you'll see a checkbox next to an option for 'add a button to the toolbar...'. Go ahead and check the box.



Because I do a lot of writing in the notebooks, I also activate the **spellchecker** extension which highlights misspelled words when editing a markdown cell.

5.3 Skip-Traceback in the Notebook

Open up a new Jupyter Notebook or reload the page for this current notebook and you will see a little icon with an exclamation mark within a triangle in the menu bar above. By default, the Skip-Traceback feature will be turned on.

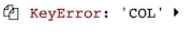


How it works

As the name implies, the extension allows you to skip the traceback. So, what is a traceback? A **traceback** is the set of programming statements that were executed whenever an error occurs in the program. Tracebacks allow you to literally trace back the steps of execution to help you find bugs. While this feature is helpful for developers, it isn't helpful for users of the library who are only interested in the error message.

When Skip-Traceback is activated, only the error message are visible and not the complete traceback. The reason this can be extremely useful is that tracebacks for some libraries like pandas can be very long and take up the entire screen. The following image shows the complete traceback for the case when a column name that does not exist is selected from a DataFrame. This is a very common error and it's unnecessary to see this wall of text.

Skip-Traceback will only show the most important part of the traceback for the users, which is the type of error and the error message, which typically is just a single line of output. Skip-Traceback provides a toggle arrow to reveal the traceback if desired.



Skip-Traceback is optional

I find the Skip-Traceback extension to be extremely useful, especially for those new to programming. It allows you to focus on the error message which is the most important part of the traceback. Even so, it is not a necessary component of the book and so you may not enable it if desired.

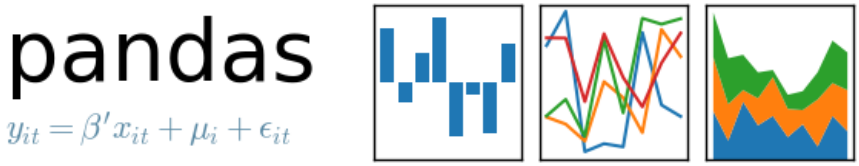
Part II

Intro to pandas

Chapter 6

What is pandas?

In this chapter, you will get introduced to the pandas library and see examples of many of the data analysis tasks it is able to complete.



What is pandas?

pandas is one of the most popular open source data exploration libraries currently available. It gives its users the power to explore, manipulate, query, aggregate, and visualize **tabular** data. Tabular meaning data that is two-dimensional with rows and columns; i.e. a table.

Why pandas and not xyz?

In this current age of data explosion, there are now dozens of other tools that have many of the same capabilities as the pandas library. However, there are many aspects of pandas that make it an attractive choice for data analysis and it continues to have one of the fastest growing user bases.

- It's a Python library and integrates well with the other popular data science libraries such as numpy, scikit-learn, statsmodels, matplotlib and seaborn.
- It is nearly self-contained in that lots of functionality is built into one package. This contrasts with R, where many packages are needed to obtain the same functionality.
- The community is excellent. Looking at Stack Overflow, for example, there are many ten's of thousands of pandas questions. If you need help, you are nearly guaranteed to find it quickly.

Why is it named after an East Asian bear?

The pandas library was begun by Wes McKinney beginning in 2008 at a hedge fund named AQR. In the financial world, it is common to refer to tabular data as 'panel data' which smashed together becomes pandas. If you are really interested in the history, you can hear it from the creator himself.

Python already has data structures to handle data, why do we need another one?

Even though Python is a high-level language, its primary built-in data structure to contain a sequence of values, lists, are not built for scientific computing. Lists are a general purpose data structure that can store any object of any type and are not optimized for tabular data analysis. What lacks, is a data structure that contains homogeneous data types for fast access and numerical computation. This data structure, usually referred to as an 'array' in most languages is provided by the numpy third-part library.

pandas is built directly on numpy

numpy ('numerical Python') is the most popular third-party Python library for scientific computing and forms the foundation for dozens of others, including pandas. numpy's primary data structure is an n-dimensional array which is much more powerful than a Python list and with much better performance.

All of the data in pandas is stored in numpy arrays. That said, it isn't necessary to know much about numpy when learning pandas. You can think of pandas as a higher-level, easier to use interface for doing data analysis than numpy. It is a good idea to eventually learn numpy, but for most tasks, pandas will be the right tool.

6.1 pandas operates on tabular (table) data

There are numerous formats for data such as XML, JSON, raw bytes, and many others. But, for our purposes, we will only be examining what most people think of when they think of data - a table. pandas is built just for analyzing this tabular, rectangular, very deceptively normal concept of data. pandas has the capability to read in many different formats of data, but they all will be converted to tabular data.

The DataFrame and Series

The DataFrame and Series are the two primary pandas objects that we will be using throughout this book.

- **DataFrame** - A two-dimensional data structure that looks like any other rectangular table of data you have seen with rows and columns.
- **Series** - A single dimension of data. It is analogous to a single column of data or a one dimensional array.

6.2 pandas examples

The rest of this chapter is dedicated to showing examples of what pandas is capable of doing. There will be one or two examples from each of the following major areas of the library.

- Reading data
- Filtering data
- Aggregating methods
- Non-Aggregating methods
- Aggregating within groups
- Tidying data
- Joining data
- Time series analysis
- Visualization

The goal is to give you a broad overview of what pandas is capable of doing. You are not expected to understand the syntax but rather get a few ideas of what you can expect to accomplish when using pandas. Explanations will be brief, but hopefully will provide just enough information so that you can logically follow what the end result is.

The `head` method

You will notice that many of the last lines of code end with the `head` method. By default, this method returns the first five rows of the DataFrame or Series that call it. The purpose of this method is to limit the output so that it easily fits on a screen or page in a book. If the `head` method is not used, then pandas will display the first 60 rows of data by default. To reduce output even further, an integer (usually 3) will be passed to the `head` method. This integer controls the number of rows returned.

6.3 Reading data

There will be multiple datasets used during the rest of this chapter. pandas can read in a variety of different data formats. The `read_csv` function is able to read in text data that is separated by a delimiter. By default, the delimiter is a comma. Below, we read in public bike usage data from the city of Chicago into a pandas DataFrame.

[1]:
```
import pandas as pd
bikes = pd.read_csv('../data/bikes.csv')
bikes.head(3)
```

[1]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...

6.4 Filtering data

pandas can filter the rows of a DataFrame based on whether the values in that row meet a condition. For instance, we can select only the rides that had a `tripduration` greater than 5000 (seconds). This example is a single condition that gets tested for each row. Only the rows that meet this condition are returned.

Single Condition

[2]:
```
filt = bikes['tripduration'] > 5000
bikes[filt].head(3)
```

[2]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
18	40924	Subscriber	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	...
40	61401	Subscriber	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	...
77	87005	Subscriber	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	...

Multiple Conditions

We can test for multiple conditions in a single row. The following example only returns riders that are female **and** have a `tripduration` greater than 5000.

[3]:
```
filt1 = bikes['tripduration'] > 5000
filt2 = bikes['gender'] == 'Female'
filt = filt1 & filt2
bikes[filt].head(3)
```

[3]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
40	61401	Subscriber	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	...
77	87005	Subscriber	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	...
1954	1103416	Subscriber	Female	2013-12-28 11:37:00	2013-12-28 13:34:00	7050	...

The next example has multiple conditions but only requires that one of the conditions is true. It returns all the rows where either the rider is female **or** the `tripduration` is greater than 5000.

[4]:
```
filt = filt1 | filt2
bikes[filt].head(3)
```

[4]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
9	23558	Subscriber	Female	2013-07-04 15:00:00	2013-07-04 15:16:00	922	...
14	31121	Subscriber	Female	2013-07-06 12:39:00	2013-07-06 12:49:00	610	...
18	40924	Subscriber	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	...

6.5 Aggregating methods

The technical definition of an **aggregation** is when a sequence of values is summarized by a **single** number. For example, sum, mean, median, min, and mix are all examples of aggregation functions. By default, calling these

methods on a pandas DataFrame will apply the aggregation to each column. Below, we use a dataset containing the percentage of undergraduate races for all US colleges.

[5]:
```
college = pd.read_csv('../data/college.csv', index_col='instnm')
cr = college.loc[:, 'ugds_white':'ugds_unkn']
cr.head(3)
```

[5]:

instnm	ugds_white	ugds_black	ugds_hisp	ugds_asian	ugds_aian	ugds_nhpi	...
Alabama A & M University	0.0333	0.9353	0.0055	0.0019	0.0024	0.0019	...
University of Alabama at Bi...	0.5922	0.2600	0.0283	0.0518	0.0022	0.0007	...
Amridge University	0.2990	0.4192	0.0069	0.0034	0.0000	0.0000	...

The `mean` method returns the mean of each column.

[6]:
```
cr.mean()
```

[6]:

ugds_white	0.510207
ugds_black	0.189997
ugds_hisp	0.161635
ugds_asian	0.033544
ugds_aian	0.013813
ugds_nhpi	0.004569
ugds_2mor	0.023950
ugds_nra	0.016086
ugds_unkn	0.045181

pandas allows you to aggregate rows as well. You must use the `axis` parameter to change the direction of the aggregation.

[7]:
```
cr.sum(axis=1).head(3)
```

[7]:

instnm	1.0000
Alabama A & M University	0.9999
University of Alabama at Bi...	1.0000

6.6 Non-aggregating methods

There are methods that perform some calculation on the DataFrame that do not aggregate the data and usually preserve the shape of the DataFrame. For example, the `round` method will round each number to a given decimal place.

[8]:
```
cr.round(2).head(3)
```

[8]:

instnm	ugds_white	ugds_black	ugds_hisp	ugds_asian	ugds_aian	ugds_nhpi	...
Alabama A & M University	0.03	0.94	0.01	0.00	0.0	0.0	...
University of Alabama at Bi...	0.59	0.26	0.03	0.05	0.0	0.0	...
Amridge University	0.30	0.42	0.01	0.00	0.0	0.0	...

6.7 Aggregating within groups

Above, we performed aggregations on the entire DataFrame. We can instead perform aggregations within groups of the data. Below we use an insurance dataset.

[9]:
```
ins = pd.read_csv('../data/insurance.csv')
ins.head(3)
```

[9]:

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.90	0	yes	southwest	16884.9240
1	18	male	33.77	1	no	southeast	1725.5523
2	28	male	33.00	3	no	southeast	4449.4620

One of the simplest aggregations is the frequency of occurrence of all the unique values within a single column. This is performed below with the `value_counts` method.

Frequency of unique values in a single column

[10]:
```
ins['region'].value_counts()
```

[10]:

southeast	364
southwest	325
northwest	325
northeast	324

Single aggregation function

Let's say we wish to find the mean charges for each of the unique values in the `sex` column. The `groupby` method gives us this functionality.

[11]:
```
ins.groupby('sex').agg({'charges': 'mean'}).round(-3)
```

[11]:

	charges
sex	
female	13000.0
male	14000.0

Multiple aggregation functions

pandas allows us to perform multiple aggregations at the same time. Below, we calculate the mean and max of the `charges` column as well as count the number of non-missing values.

[12]:
```
ins.groupby('sex').agg({'charges': ['mean', 'max', 'count']}).round(0)
```

[12]:

	charges		
	mean	max	count
sex			
female	12570.0	63770.0	662
male	13957.0	62593.0	676

Multiple Grouping columns

pandas allows us to form groups based on multiple columns. In the below example, each unique combination of `sex` and `region` form a group. For each of these groups, the same aggregations as above are performed on the `charges` column.

[13]:
```
ins.groupby(['sex', 'region']).agg({'charges': ['mean', 'max', 'count']}).round(0)
```

[13]:

		charges		
		mean	max	count
sex	region			
female	northeast	12953.0	58571.0	161
	northwest	12480.0	55135.0	164
	southeast	13500.0	63770.0	175
	southwest	11274.0	48824.0	162
male	northeast	13854.0	48549.0	163
	northwest	12354.0	60021.0	161
	southeast	15880.0	62593.0	189
	southwest	13413.0	52591.0	163

Pivot Tables

We can reproduce the exact same output as above in a different shape with the `pivot_table` method. It groups and aggregates the same way as `groupby` but places the unique values of one of the grouping columns as the new columns in the resulting DataFrame. Notice that pivot tables make for easier comparisons across groups.

[14]:
```
pt = ins.pivot_table(index='sex', columns='region',
                     values='charges', aggfunc='mean').round(0)
pt
```

[14]:

region	northeast	northwest	southeast	southwest
sex				
female	12953.0	12480.0	13500.0	11274.0
male	13854.0	12354.0	15880.0	13413.0

Styling DataFrames

To help make your data really pop-out, pandas enables you to style DataFrames in various ways. Below, the maximum value of each column is highlighted.

[15]:
```
pt.style.highlight_max()
```

[15]:

region	northeast	northwest	southeast	southwest
sex				
female	12953	12480	13500	11274
male	13854	12354	15880	13413

6.8 Tidying

Many datasets need to be cleaned and tidied before we can perform analysis on them. pandas provides many tools to prepare our data for further analysis.

Options in the `read_csv` function

Below, we read in a new dataset on plane crashes. Notice all the question marks. They represent missing values, but pandas will read them in as strings.

[16]:
```
pc = pd.read_csv('../data/tidy/planecrashinfo.csv')
pc.head(3)
```

[16]:

	date	time	location	operator	flight_no	...
0	September 17, 1908	17:18	Fort Myer, Virginia	Military - U.S. Army	?	...
1	September 07, 1909	?	Juvisy-sur-Orge, France	?	?	...
2	July 12, 1912	06:30	Atlantic City, New Jersey	Military - U.S. Navy	?	...

The `read_csv` has dozens of options to help read in messy data. One of the options allows you to convert a particular string to missing values. Notice that all of the question marks are now labeled as `NaN` (not a number).

[17]:
```
pc = pd.read_csv('../data/tidy/planecrashinfo.csv', na_values='?')
pc.head(3)
```

[17]:

	date	time	location	operator	flight_no	...
0	September 17, 1908	17:18	Fort Myer, Virginia	Military - U.S. Army	NaN	...
1	September 07, 1909	NaN	Juvisy-sur-Orge, France	NaN	NaN	...
2	July 12, 1912	06:30	Atlantic City, New Jersey	Military - U.S. Navy	NaN	...

String manipulation

Often times there is data stuck within a string column that you will need to extract. The `aboard` column appears to have three distinct pieces of information; the total number of people on board, the number of passengers, and the number of crew.

[18]:
```
aboard = pc['aboard']
aboard.head()
```

[18]:

0	2 (passengers:1 crew:1)
1	1 (passengers:0 crew:1)
2	5 (passengers:0 crew:5)
3	1 (passengers:0 crew:1)
4	20 (passengers:? crew:?)

pandas has special functionality for manipulating strings. Below, we use a regular expression to extract the pertinent numbers from the `aboard` column.

[19]:
```
aboard.str.extract(r'(\d+)?\D*(\d+)?\D*(\d+)?').head()
```

[19]:

	0	1	2
0	2	1	1
1	1	0	1
2	5	0	5
3	1	0	1
4	20	NaN	NaN

Reshaping into tidy form

Occasionally, you will have several columns of data that all belong in a single column. Take a look at the DataFrame below on average arrival delay of airlines at different airports. All the columns with three-letter airport codes could be placed in the same column as they all contain the arrival delay which has the same units.

[20]:
```
aad = pd.read_csv('../data/tidy/average_arrival_delay.csv').head()
aad
```

[20]:

	airline	ATL	DEN	DFW	IAH	LAS	LAX	MSP	ORD	PHX	SFO
0	AA	4.0	9.0	5.0	11.0	8.0	3.0	1.0	8.0	5.0	3.0
1	AS	6.0	-3.0	-5.0	1.0	2.0	-3.0	6.0	2.0	-9.0	4.0
2	B6	NaN	12.0	4.0	NaN	11.0	2.0	NaN	23.0	20.0	5.0
3	DL	0.0	-3.0	10.0	3.0	-3.0	3.0	-1.0	7.0	-4.0	0.0
4	EV	7.0	14.0	10.0	3.0	NaN	NaN	10.0	8.0	-14.0	NaN

The `melt` method stacks columns one on top of the other. Here, it places all of the three-letter airport code columns into a single column. The first two airports (ATL and DEN) are shown below in the new tidy DataFrame.

[21]:
```
aad.melt(id_vars='airline', var_name='airport', value_name='delay').head(10)
```

[21]:

	airline	airport	delay
0	AA	ATL	4.0
1	AS	ATL	6.0
2	B6	ATL	NaN
3	DL	ATL	0.0
4	EV	ATL	7.0
5	AA	DEN	9.0
6	AS	DEN	-3.0
7	B6	DEN	12.0
8	DL	DEN	-3.0
9	EV	DEN	14.0

6.9 Joining Data

pandas can join multiple DataFrames together by matching values in one or more columns. If you are familiar with SQL, then pandas performs joins in a similar fashion. Below, we make a connection to a database and read in two of its tables.

[22]:
```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///../data/neurIPS.db')

authors = pd.read_sql('Authors', engine)
pa = pd.read_sql('PaperAuthors', engine)
```

Output the first 5 rows of each DataFrame.

[23]:
```
authors.head(3)
```

[23]:

	Id	Name
0	178	Yoshua Bengio
1	200	Yann LeCun
2	205	Avrim Blum

[24]:
```
pa.head(3)
```

[24]:

	Id	PaperId	AuthorId
0	1	5677	7956
1	2	5677	2649
2	3	5941	8299

We can now join these tables together using the `merge` method. The `AuthorID` column from the `pa` table is aligned with the `Id` column of the `authors` table.

[25]:
```
pa.merge(authors, how='left', left_on='AuthorId', right_on='Id').head(3)
```

[25]:

	Id_x	PaperId	AuthorId	Id_y	Name
0	1	5677	7956	7956	Nihar Bhadresh Shah
1	2	5677	2649	2649	Denny Zhou
2	3	5941	8299	8299	Brendan van Rooyen

6.10 Time Series Analysis

One of the original purposes of pandas was to do time series analysis. Below, we read in 4 years of Apple's closing stock price data with help from Quandl API. Instructions for setting up this API are found in the **Case Study: Calculating Normality of Stock Market Returns** chapter of the **Essential Commands** part.

[26]:
```
import quandl

with open('../api_key.txt') as f:
    api_key = f.read()

aapl = quandl.get(dataset='WIKI/AAPL', start_date='2014-01-01',
                  end_date='2017-12-31', api_key=api_key)
aapl = aapl[['Adj. Close', 'Adj. Volume']]
aapl.columns = ['close', 'volume']
aapl.head(3)
```

[26]:

Date	close	volume
2014-01-02	73.523423	58671200.0
2014-01-03	71.908415	98116900.0
2014-01-06	72.300536	103152700.0

Select a period of time

pandas allows us to easily select a period of time. Below, we select all of the trading data from February 27, 2018 through March 2, 2018.

[27]:
```
aapl['2018-02-27':'2018-03-02']
```

[27]:

Empty DataFrame Columns: Index(['close', 'volume'], dtype='object') Index: DatetimeIndex([], dtype='datetime64[ns]', name=

Group by time

We can group by some length of time. Here, we group together every month of trading data and return the average closing price of that month.

[28]:
```
aapl_mc = aapl.resample('M').agg({'close':'mean'})
aapl_mc.head(3)
```

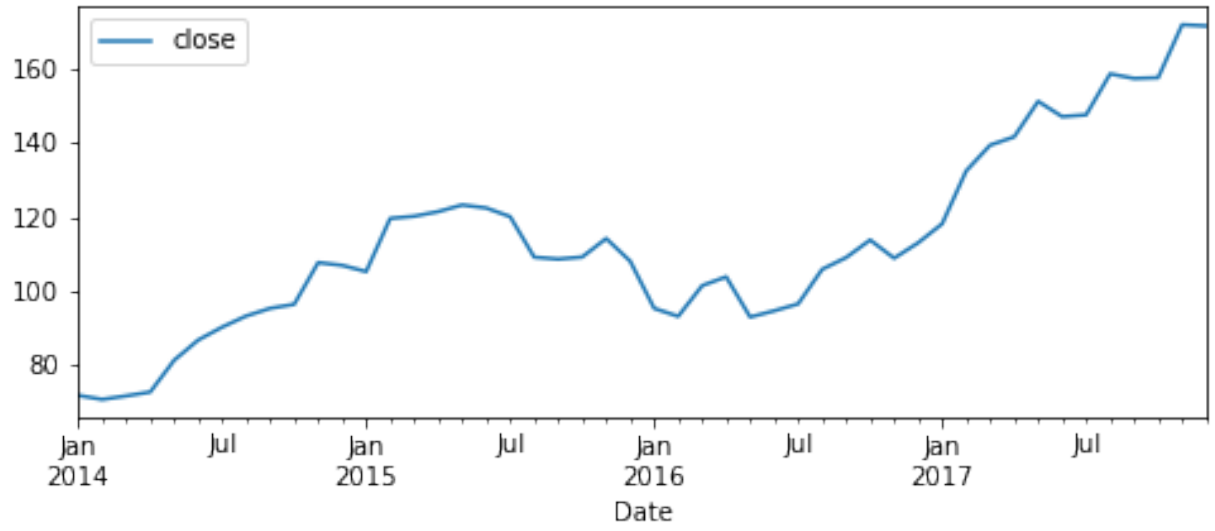
[28]:

Date	close
2014-01-31	71.438619
2014-02-28	70.347470
2014-03-31	71.297968

6.11 Visualization

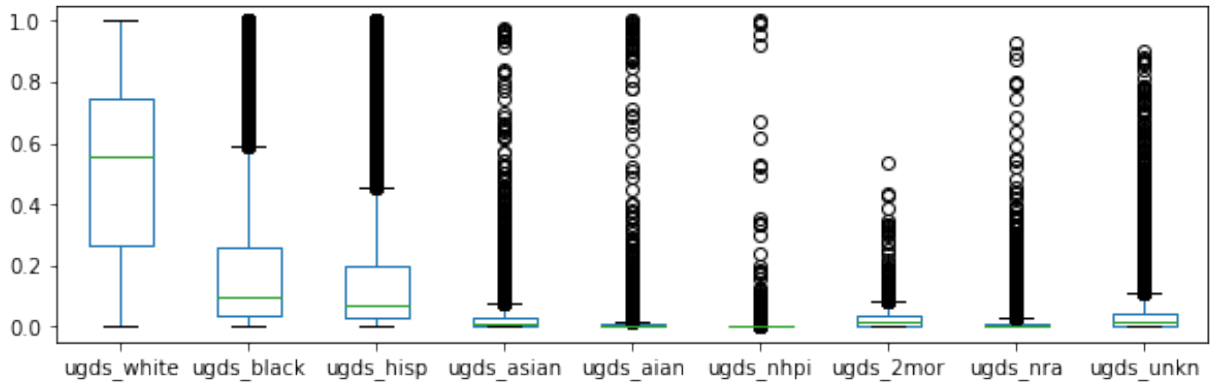
pandas provides basic visualization abilities by giving its users a few nice default plots. Below, we plot the average monthly closing price of Apple for the last 5 years.

[29]:
```
%matplotlib inline
aapl_mc.plot(kind='line', figsize=(8, 3));
```



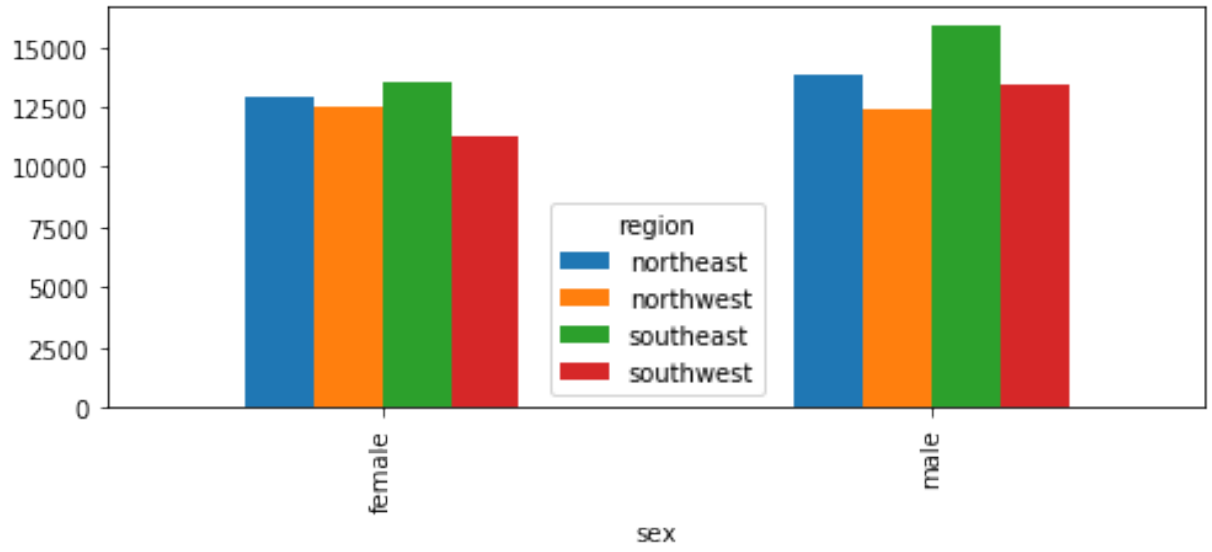
Here, we use the college race data to create a box plot of each of the race percentage columns.

[30]:
```
cr.plot(kind='box', figsize=(10, 3));
```



We turn our pivot table from above into a bar graph.

[31]:
```
pt.plot(kind='bar', figsize=(8, 3));
```



6.12 Much More

The above was just a small sampling that pandas has to offer, but does show many basic examples from many of the major sections of the library.

Chapter 7

The DataFrame and Series

The DataFrame and Series are the two primary pandas objects that we will be using throughout this book.

- **DataFrame** - A two-dimensional data structure that looks like any other rectangular table of data you have seen with rows and columns.
- **Series** - A single dimension of data. It is analogous to a single column of data or a one dimensional array.

7.1 Import pandas and read in data with `read_csv`

By convention, pandas is imported and aliased as `pd`. We will read in the `bikes` dataset with the `read_csv` function. Its first parameter is the location of the file relative to the current directory as a string. All of the data for this book is stored in the `data` directory one level above where this notebook is located. The two dots in the path passed to `read_csv` are interpreted as the directory immediately above the current one.

[1]:
```
import pandas as pd
bikes = pd.read_csv('../data/bikes.csv')
```

Display DataFrame in Jupyter Notebook

We assigned the output from the `read_csv` function to the `bikes` variable which now refers to our DataFrame object. To visually display the DataFrame, place the variable name as the last line in a code cell. By default, pandas outputs 60 rows and 20 columns.

`head` and `tail` methods

A very useful and simple method is `head`, which returns the first 5 rows of the DataFrame by default. This avoids long default output and is something I highly recommend when doing data analysis within a notebook. The `tail` method returns the last 5 rows by default. There will be very few instances in the book where the `head` method is not used as displaying 60 rows is far too many and will take up a lot of space on a screen or page.

[2]:
```
bikes.head()
```

[2]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...
3	12907	Subscriber	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667	...
4	13168	Subscriber	Male	2013-07-01 11:16:00	2013-07-01 11:18:00	130	...

The last five rows of the DataFrame are displayed with the `tail` method.

[3]:
```
bikes.tail()
```

[3]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
50084	17534938	Subscriber	Male	2017-12-30 13:07:00	2017-12-30 13:34:00	1625	...
50085	17534969	Subscriber	Male	2017-12-30 13:34:00	2017-12-30 13:44:00	585	...
50086	17534972	Subscriber	Male	2017-12-30 13:34:00	2017-12-30 13:48:00	824	...
50087	17535645	Subscriber	Female	2017-12-31 09:30:00	2017-12-31 09:33:00	178	...
50088	17536246	Subscriber	Male	2017-12-31 15:22:00	2017-12-31 15:26:00	214	...

First and last `n` rows

Both the `head` and `tail` methods take a single integer parameter `n` controlling the number of rows returned. Here, we output the first three rows.

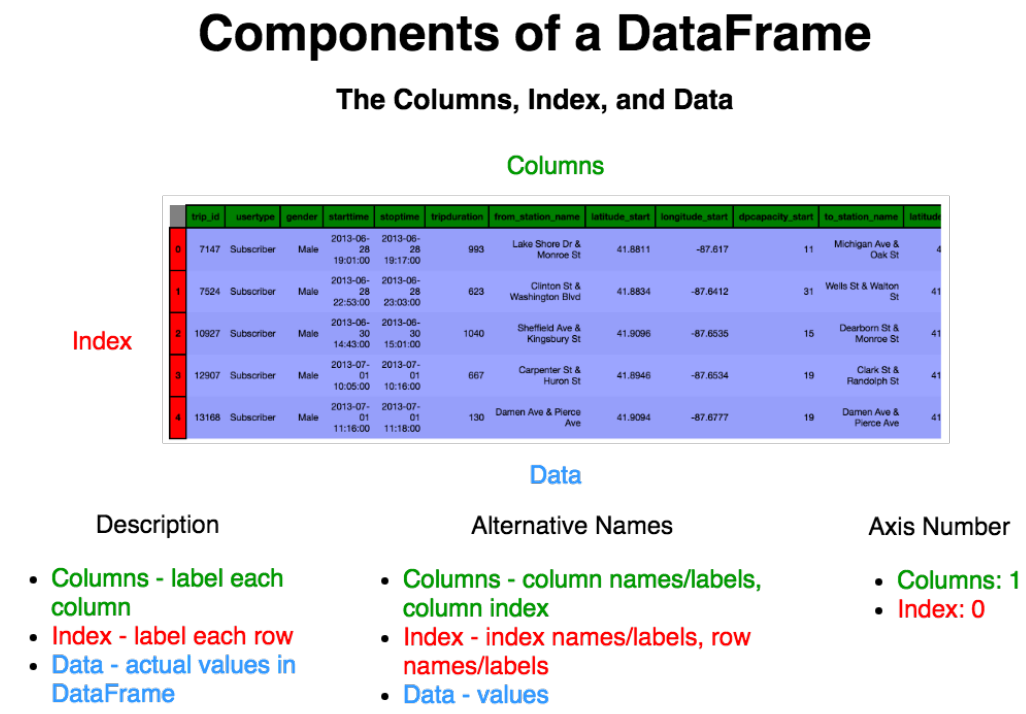
[4]:
```
bikes.head(3)
```

[4]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...

7.2 Components of a DataFrame

The DataFrame is composed of three separate components - the **columns**, the **index**, and the **data**. These terms will be used throughout the book and understanding them is vital to your ability to use pandas. Take a look at the following graphic of our `bikes` DataFrame stylized to put emphasis on each component.



The index

The index provides a **label** for each row and will always be displayed in **bold** font to the left of the data. For instance, the index label **3** references all the values in its row.

The index is also referred to as the **row labels** and individual values in the index are referred to as an **index label** or **row label**. A major purpose of the index is to identify each row in the DataFrame. Similarly to how we humans have names to identify ourselves, the index has labels to identify each row.

In the above DataFrame, the index is simply a sequence of integers beginning at 0. The values in the index are not limited to integers. Strings are a common type that are used in the index and make for more descriptive labels.

Surprisingly, values in the index are not required to be unique. In fact, all the index values can be the same. This is important to understand, as knowing a row label does not guarantee a one-to-one mapping to one specific row.

The columns

The columns provide a **label** for each column and will always be displayed in **bold** font above the data. In the above DataFrame, the column name `from_station_name` references all the values in that column.

The columns are also referred to as the **column names** or the **column labels** with individual values referred to as a **column name** or **column label**.

The column names in the above DataFrame are strings, but can be other types such as integers though string names are more descriptive. Like the index, pandas will default column names to an integer sequence beginning at 0, if the columns are not provided upon DataFrame creation.

As with the index, the column names are not required to be unique, though having duplicate columns would be bad practice as its vital to be able to uniquely identify each column.

The data

The actual data is to the right of the index and below the columns and is always in normal font. The data is also referred to as the **values**. The data represents all the values for all the columns. It is important to note that the index and the columns are NOT part of the data. They are separate objects and act as **labels**.

The Axes

The index and columns are known collectively as the **axes** and each represent a single **axis** of the two-dimensional DataFrame. pandas uses the integer **0** for the index and 1 for the columns. pandas allows you to use these integers when referencing the index and columns within methods. This terminology is borrowed from numpy's ndarray objects which can have any number of dimensions.

7.3 What type of object is `bikes`?

Let's verify that `bikes` is indeed a DataFrame with the `type` function.

[5]:
```
type(bikes)
```

[5]:
```
pandas.core.frame.DataFrame
```

Fully-qualified name

The above output returns something called the **fully-qualified name**. Only the word after the last dot is the name of the type and in this case we have verified that the `bikes` variable has type `DataFrame`. The fully-qualified name always returns the package and module name of where the type was defined.

The package name is the first part of the fully-qualified name and in this case is `pandas`. The module name is the word immediately preceding the name of the type and in this case is `frame`.

Package vs Module

A Python **package** is a directory containing other directories or modules. A Python **module** is a file (typically a text file ending in .py) that contains Python code of where the type is defined (typically with a `class` statement).

Sub-packages

Any directory containing other directories or modules within a Python package is considered a **sub-package**. In this case, `core` is the sub-package.

Where are the packages located on my machine?

Third-party packages are installed in the `site-packages` directory which itself is set up during Python installation. We can get the actual location with the help from the standard library's `site` module's `getsitepackages` function.

[6]:
```
import site
sp = site.getsitepackages()
sp
```

[6]:
```
['/Users/Ted/miniconda3/envs/minimal_ds/lib/python3.7/site-packages']
```

You can navigate to this directory with your file explorer. The will be a 'pandas' directory found there and within that will be a 'core' directory which will contain the 'frame.py' file. It is this text file which contains Python code where the DataFrame class is defined.

7.4 Select a single column from a DataFrame - a Series

To select a single column from a DataFrame, pass the name of one of the columns to the brackets operator, `[]`. The returned object will be a pandas **Series**. Let's select the column name `tripduration`, assign it to a variable, and output the first few values to the screen. The `head` and `tail` methods work the same as they do with DataFrames.

[7]:
```
trip_duration = bikes['tripduration']
trip_duration.head()
```

[7]:

0	993
1	623
2	1040
3	667
4	130

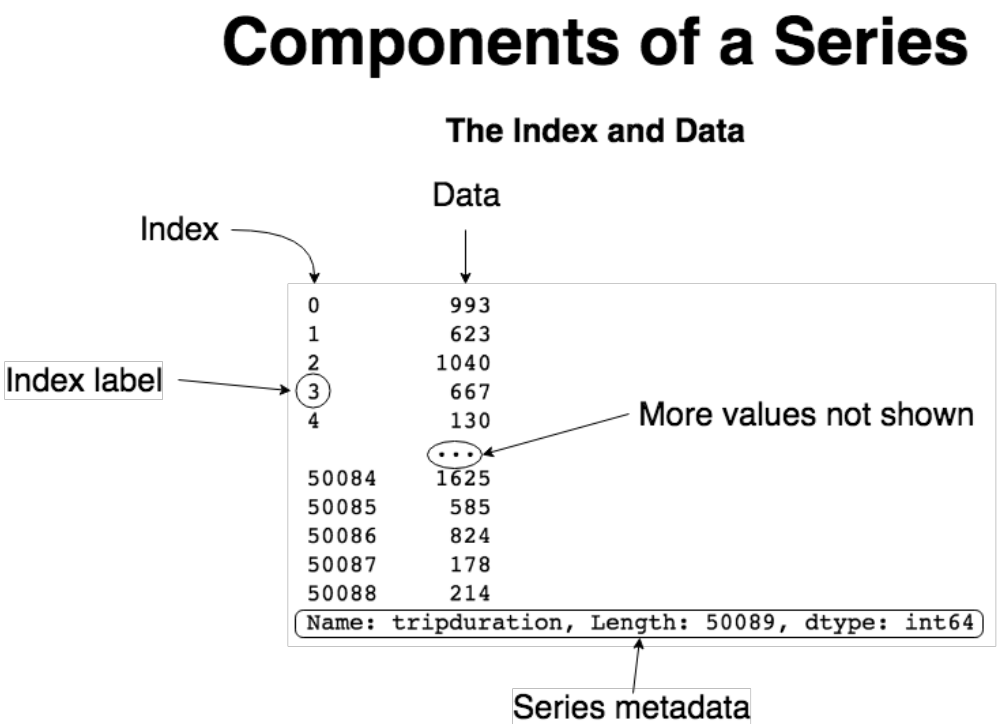
[8]:
```
trip_duration.tail(3)
```

[8]:

50086	824
50087	178
50088	214

7.5 Components of a Series - the index and the data

A Series is simpler than a DataFrame with just a single dimension of data. It has two components - the **index** and the **data**. Let's take a look at a stylized Series graphic.



It's important to note that a Series has no rows and no columns. In appearance, it resembles a one-column DataFrame, but it technically has no columns. It just has a sequence of values that are labeled by an index.

The index

A Series index serves as labels for the values and a single index **label** always references a single value. In the above image, the index label **3** corresponds with the value 667. The Series index is virtually identical to the DataFrame index, so the same rules apply to it. Index values can be duplicated and can be types other than integers, such as strings.

Output of Series vs DataFrame

Notice that there is no nice HTML styling for the Series. It's just plain text. Also, below each Series will be some metadata on it - the **name**, **length**, and **dtype**.

- The **name** is not important right now. If the Series is formed from a column of a DataFrame it will be set to that column name.
- The **length** is the number of values in the Series
- The **dtype** is the data type of the Series, which will be discussed in upcoming chapters.

It's important to note that this metadata is NOT part of the Series itself and is just some extra info pandas outputs for your information.

7.6 Exercises

Use the `bikes` DataFrame for the following exercises.

Exercise 1

Select the column `events`, the type of weather that was recorded, and assign it to a variable with the same name. Output the first 10 values of it.

[ ]:

Exercise 2

What type of object is `events`?

[ ]:

Exercise 3

Select the last 2 rows of the `bikes` DataFrame and assign it to the variable `bikes_last_2`. What type of object is `bikes_last_2`?

[ ]:

Chapter 8

Data Types and Missing Values

Each column of data in a pandas DataFrame has a data type. This is a very similar concept to types in Python. Just like every Python object has a type, every column has a data type. pandas has a large number of data types available for each column. This chapter focuses only on the most common data types and provides a brief summary of each one. For extensive coverage of each and every data type, see chapter **Changing Data Types** in the **Essential Commands** part.

Most Common Data Types

The following are the most common data types that appear frequently in DataFrames.

- **boolean** - only two possible values, `True` and `False`
- **integer** - whole numbers without decimals
- **float** - numbers with decimals
- **object** - typically strings, but may contain any object
- **datetime** - a specific date and time with nanosecond precision

More on the object data type

The object data type is the most confusing and deserves a longer discussion. Each value in an object column can be **any** Python object. This means object columns can contain integers, floats, or even complex types such as lists or dictionaries. Anything can be contained in object columns. But, nearly all of the time, object data type columns contain **strings**. When you see that a column is an object data type, you should expect the values to be strings. pandas, unfortunately, does not provide its users with a specific data type for strings so if you do have strings in your columns, the data type will be object.

The importance of knowing the data type

Knowing the data type of each column of your pandas DataFrame is very important. The main reason for this is that every value in each column will be of the same type. For instance, if you select a single value from a column that has an integer data type, then you are guaranteed that this value is also an integer. Knowing the data type of a column is one of the most fundamental pieces of knowledge of your DataFrame.

A major exception with the object data type

The object data type, is unfortunately, an exception to the information in the previous section. Although columns that have object data type are typically strings, there is no guarantee that each value will be a string. You could very well have an integer, list, or even another DataFrame as a value in the same object column.

8.1 Missing Value Representation

`NaN`, `None`, and `NaT`

pandas represents missing values differently based on the data type of the column.

- `NaN` stands for not a number and is technically a float data type
- `None` is the literal Python object `None` and will only be found in object columns
- `NaT` stands for not a time and is used for missing values in datetime columns

Missing values for each data type

- **boolean and integer** - No representation for missing values exist for boolean and integer columns. This is an unfortunate limitation, though recently pandas has created an entire new integer data type that does have `NaN`s available.
- **floats** - Uses only `NaN` as the missing value.
- **object** - Columns of object data type can contain any Python object so all three of the missing value representation may appear in these columns but typically they will be either `NaN` or `None`.
- **datetime** - Uses only `NaT` as the missing value.

Missing values in boolean and integer columns

Knowing that a column is either a boolean or integer column guarantees that there are no missing values in that column as pandas does not allow for it. If, for instance, you would like to place a missing value in a boolean or integer column, then pandas will convert the entire column to float as it is a numerical data type with a missing value representation. When booleans are converted to floats, `False` becomes 0 and `True` becomes 1.

Integer NaN update for pandas 0.24

With the release of pandas version 0.24 (February 2019), missing value representation was made possible for a special kind of integer data type called **Int64Dtype**. There is still no missing value representation for the default integer data type.

8.2 Finding the data types of each column

The `dtypes` DataFrame attribute (NOT a method) returns the data type of each column. Let's get the data types of our `bikes` DataFrame. Note that the returned data is a Series with the column names now in the index and the data type as the values.

[1]:
```
import pandas as pd
bikes = pd.read_csv('../data/bikes.csv')
bikes.head(3)
```

[1]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...

[2]:
```
bikes.dtypes
```

[2]:

trip_id	int64
usertype	object
gender	object
starttime	object
stoptime	object
tripduration	int64
from_station_name	object
latitude_start	float64
longitude_start	float64
dpcapacity_start	float64
to_station_name	object
latitude_end	float64
longitude_end	float64
dpcapacity_end	float64
temperature	float64
visibility	float64
wind_speed	float64
precipitation	float64
events	object

Why are `starttime` and `stoptime` object data types?

If you look at the output of the `bikes` DataFrame, it's apparent that both the `starttime` and `stoptime` columns are datetimes, but the output from `dtypes` states that they are objects. The `read_csv` function requires that you provide a list of columns that are datetimes to the `parse_dates` parameter, otherwise it will read them in as strings. Let's reread the data using the `parse_dates` parameter.

[3]:
```
bikes = pd.read_csv('../data/bikes.csv', parse_dates=['starttime', 'stoptime'])
bikes.dtypes.head()
```

[3]:

trip_id	int64
usertype	object
gender	object
starttime	datetime64[ns]
stoptime	datetime64[ns]

What are all those 64's at the end of the data types?

Booleans, integers, floats, and datetimes all use a particular amount of memory for each of their values. The memory is measured in **bits**. The number of bits used for each value is the number appended to the end of the data type name. For instance, integers can be either 8, 16, 32, or 64 bits while floats can be 16, 32, 64, or 128. A 128-bit float column will show up as `float128`.

Technically a `float128` is a different data type than a `float64` but generally you will not have to worry about such a distinction as the operations between different float columns will be the same.

Booleans are always stored as 8-bits. There is no set bit size for values in an **object** column as each value can be of any size.

8.3 Getting more metadata

Metadata can be defined as data on the data. The data type of each column is an example of **metadata**. The number of rows and columns is another piece of metadata. We find this with the `shape` attribute, which returns a tuple of integers.

[4]:
```
bikes.shape
```

[4]:
```
(50089, 19)
```

Total number of values with the `size` attribute

The `size` attribute returns the total number of values (the number of columns multiplied by the number of rows) in the DataFrame.

[5]:
```
bikes.size
```

[5]:
```
951691
```

Get data types plus more with the `info` method

The `info` DataFrame method provides output similar to `dtypes`, but also shows the number of non-missing values in each column along with more info such as:

- Type of object (always a DataFrame)
- The type of index and number of rows
- The number of columns
- The data types of each column and the number of non-missing (a.k.a non-null)
- The frequency count of all data types
- The total memory usage

[6]:
```
bikes.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50089 entries, 0 to 50088
Data columns (total 19 columns):
trip_id              50089 non-null int64
usertype             50089 non-null object
gender               50089 non-null object
starttime            50089 non-null datetime64[ns]
stoptime             50089 non-null datetime64[ns]
tripduration         50089 non-null int64
from_station_name    50089 non-null object
latitude_start       50083 non-null float64
longitude_start      50083 non-null float64
dpcapacity_start     50083 non-null float64
to_station_name      50089 non-null object
latitude_end         50077 non-null float64
longitude_end        50077 non-null float64
dpcapacity_end       50077 non-null float64
temperature          50089 non-null float64
visibility           50089 non-null float64
wind_speed           50089 non-null float64
precipitation        50089 non-null float64
events               50089 non-null object
dtypes: datetime64[ns](2), float64(10), int64(2), object(5)
memory usage: 7.3+ MB
```

8.4 More data types

There are several more data types available in pandas. An extensive and formal discussion on all data types is available in the chapter **Changing Data Types** from the **Essential Commands** part.

8.5 Exercises

Use the `bikes` DataFrame for the following:

Exercise 1

What type of object is returned from the `dtypes` attribute?

[ ]:

Exercise 2

What type of object is returned from the `shape` attribute?

[ ]:

Exercise 3

What type of object is returned from the `info` method?

[ ]:

Exercise 4

The memory usage from the `info` method isn't correct when you have objects in your DataFrame. Read the docstrings from it and get the true memory usage.

[ ]:

Chapter 9

Setting a Meaningful Index

The index of a DataFrame provides a label for each of the rows. If not explicitly provided, pandas will use the sequence of consecutive integers beginning at 0 as the index. In this chapter, we learn how to set one of the columns of the DataFrame as the new index so that it provides a more meaningful label to each row.

9.1 Extracting the components of a DataFrame

The DataFrame consists of three components - the index, columns, and data. It is possible to extract each component and assign them to their own variable. Let's read in a small dataset to show how this is done. Notice that when we read in the data, we choose the first column to be the index by setting the `index_col` parameter to 0.

[1]:
```
import pandas as pd
df = pd.read_csv('../data/sample_data.csv', index_col=0)
df
```

[1]:

	state	color	food	age	height	score
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

The attributes `index`, `columns`, and `values`

The index, columns, and data are each separate objects and can be extracted from the DataFrame. Notice that each of these objects are extracted as attributes and NOT methods. Let's assign them to their own variables.

[2]:
```
index = df.index
columns = df.columns
data = df.values
```

View these objects

Let's get a visual representation of these objects by outputting them to the screen.

[3]:
```
index
```

[3]:
```
Index(['Jane', 'Niko', 'Aaron', 'Penelope', 'Dean', 'Christina', 'Cornelia'],
dtype='object')
```

[4]:
```
columns
```

[4]:
```
Index(['state', 'color', 'food', 'age', 'height', 'score'], dtype='object')
```

[5]:
```
data
```

[5]:
```
array([['NY', 'blue', 'Steak', 30, 165, 4.6],
       ['TX', 'green', 'Lamb', 2, 70, 8.3],
       ['FL', 'red', 'Mango', 12, 120, 9.0],
       ['AL', 'white', 'Apple', 4, 80, 3.3],
       ['AK', 'gray', 'Cheese', 32, 180, 1.8],
       ['TX', 'black', 'Melon', 33, 172, 9.5],
       ['TX', 'red', 'Beans', 69, 150, 2.2]], dtype=object)
```

Find the type of these objects

The output of these objects looks correct, but we don't know the exact type of each one. Let's find out the types of each object.

[6]:
```
type(index)
```

[6]:
```
pandas.core.indexes.base.Index
```

[7]:
```
type(columns)
```

[7]:
```
pandas.core.indexes.base.Index
```

[8]:
```
type(data)
```

[8]:
```
numpy.ndarray
```

pandas `Index` type

pandas has a special type of object called an `Index`. This object is similar to a list or a one dimensional array. You can think of it as a sequence of labels for either the rows or the columns. You will not deal with this object directly much, so we will not go into further details about it here. Notice that both the index and columns are of the same type.

Two-dimensional numpy array

The values are returned as a single two-dimensional numpy array.

Operating with the DataFrame as a whole

You will rarely need to operate with these components directly and instead be working with the entire DataFrame. But, it is important to understand that they are separate components and you can access them directly if needed.

9.2 Extracting the components of a Series

Similarly, we can extract the two Series components - the index and the data. Let's first select a single column from our DataFrame so that we have a Series.

[9]:
```
color = df['color']
color
```

[9]:

Jane	blue
Niko	green
Aaron	red
Penelope	white
Dean	gray
Christina	black
Cornelia	red

Verify that `color` is a Series and then extract its components.

[10]:
```
type(color)
```

[10]:
```
pandas.core.series.Series
```

[11]:
```
color.index
```

[11]:
```
Index(['Jane', 'Niko', 'Aaron', 'Penelope', 'Dean', 'Christina', 'Cornelia'],
dtype='object')
```

[12]:
```
color.values
```

[12]:
```
array(['blue', 'green', 'red', 'white', 'gray', 'black', 'red'],
      dtype=object)
```

9.3 The default index

If you don't specify an index when first reading in a DataFrame, then pandas will create one for you as integers beginning at 0. An index always exists, even if it just appears to be the row number. Let's read in the movie dataset without explicitly setting an index.

[13]:
```
movie = pd.read_csv('../data/movie.csv')
movie.head(3)
```

[13]:

	title	year	color	content_rating	duration	director_name	director_fb	...
0	Avatar	2009.0	Color	PG-13	178.0	James Cameron	0.0	...
1	Pirates of the Caribbean: ...	2007.0	Color	PG-13	169.0	Gore Verbinski	563.0	...
2	Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	0.0	...

Notice the integers in the index

These integers are the default index labels for each of the rows. Let's examine the underlying index object.

[14]:
```
idx = movie.index
idx
```

[14]:
```
RangeIndex(start=0, stop=4916, step=1)
```

[15]:
```
type(idx)
```

[15]:
```
pandas.core.indexes.range.RangeIndex
```

The RangeIndex

pandas has various types of index objects. A `RangeIndex` is the simplest index and represent the sequence of consecutive integers beginning at 0. It is similar to a Python `range` object in that the values are not actually stored in memory.

Select a value from the index

The index is a complex object on its own and has many attributes and methods. The minimum we should know about an index is how to select values from it. We can select single values from an index just like we do with a Python list, by placing the integer location of the item we want within the square brackets. Here, we select the 6th item (integer location 5) from the index.

[16]:
```
idx[5]
```

[16]:
```
5
```

A numpy array underlies the index

To get the underlying values into a numpy array, use the `values` attribute. This is similar to how we get the underlying data from a pandas DataFrame.

[17]:
```
idx.values
```

[17]:
```
array([   0,    1,    2, ..., 4913, 4914, 4915])
```

If you don't assign the index to a variable, you can retrieve the array from the DataFrame by chaining the attributes together like this:

[18]:
```
movie.index.values
```

[18]:
```
array([   0,    1,    2, ..., 4913, 4914, 4915])
```

9.4 Setting an index on read

The `read_csv` function provides dozens of parameters that allow us to read in a wide variety of text files. The `index_col` parameter may be used to select a particular column as the index. We can either use the column name or its integer location.

Reread the movie dataset with the movie title as the index

There's a column in the movie dataset named `title`. Let's reread in the data with it as the index.

[19]:
```
movie = pd.read_csv('../data/movie.csv', index_col='title')
movie.head(3)
```

[19]:

title	year	color	content_rating	duration	director_name	director_fb	...
Avatar	2009.0	Color	PG-13	178.0	James Cameron	0.0	...
Pirates of the Caribbean: A...	2007.0	Color	PG-13	169.0	Gore Verbinski	563.0	...
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	0.0	...

Notice that now the titles of each movie serve as the label for each row. Also notice that the word **title** appears directly above the index. This is a bit confusing - **title** is NOT a column name, but rather the **name** of the index.

Extract the new index and output its type

Let's extract this new index and find its exact type.

[20]:
```
idx2 = movie.index
idx2
```

[20]:
```
Index(['Avatar', 'Pirates of the Caribbean: At World's End', 'Spectre',
       'The Dark Knight Rises', 'Star Wars: Episode VII - The Force Awakens',
```

```
       'John Carter', 'Spider-Man 3', 'Tangled', 'Avengers: Age of Ultron',
       'Harry Potter and the Half-Blood Prince',
       ...
       'Primer', 'Cavite', 'El Mariachi', 'The Mongol King', 'Newlyweds',
       'Signed Sealed Delivered', 'The Following', 'A Plague So Pleasant',
       'Shanghai Calling', 'My Date with Drew'],
      dtype='object', name='title', length=4916)
```

[21]:
```
type(idx2)
```

[21]:
```
pandas.core.indexes.base.Index
```

Selecting values from this index

Just like we did with our `RangeIndex`, we use the brackets operator to select a single index value.

[22]:
```
idx2[105]
```

[22]:
```
'Poseidon'
```

Selection with slice notation

As with Python lists, you can select a range of values using slice notation. Provide the start, stop, and step components of slice notation separated by a colon like this - `start:stop:step`

[23]:
```
idx2[100:120:4]
```

[23]:
```
Index(['The Fast and the Furious', 'The Sorcerer's Apprentice', 'Warcraft',
       'Transformers', 'Hancock'],
      dtype='object', name='title')
```

Selection with a list of integers

You can select multiple individual values with a list of integers. This type of selection does not exist for Python lists.

[24]:
```
nums = [1000, 453, 713, 2999]
idx2[nums]
```

[24]:
```
Index(['The Life Aquatic with Steve Zissou', 'Daredevil', 'Daddy Day Care',
       'The Ladies Man'],
      dtype='object', name='title')
```

9.5 Choosing a good index

Before even considering using one of the columns as an index, know that it's not a necessity. You can complete all of your analysis with just the default `RangeIndex`.

Setting a column to be an index can help make certain analysis easier in some situations, so it is something that can be considered. If you do choose to set an index for your DataFrame, I suggest using columns that are both **unique** and **descriptive**. Pandas does not enforce uniqueness for its index allowing the same value to repeat multiple times. That said, a good index will have unique values to identify each row.

9.6 Setting the index with `set_index`

It is possible to set the index after reading the data with the `set_index` method. Pass it the name of the column you would like to use as the index. Below, we read in our data without setting an index.

[25]:
```
movie = pd.read_csv('../data/movie.csv')
movie = movie.set_index('title')
movie.head(3)
```

[25]:

title	year	color	content_rating	duration	director_name	director_fb	...
Avatar	2009.0	Color	PG-13	178.0	James Cameron	0.0	...
Pirates of the Caribbean: A...	2007.0	Color	PG-13	169.0	Gore Verbinski	563.0	...
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	0.0	...

Reassigned `movie` variable

In the last code block, we reassigned the variable name `movie` to the result of the `set_index` command. This is necessary because `set_index` makes an entire new copy of the data and does not change the calling DataFrame. If we run the same commands, but do not assign the result of the `set_index` method, the DataFrame will not be changed. Let's verify this by changing the second line of code from above while using a new variable name, `movie2`. Notice that `title` is still a column and has not been set as the index.

[26]:
```
movie2 = pd.read_csv('../data/movie.csv')
movie2.set_index('title')
movie2.head(3)
```

[26]:

	title	year	color	content_rating	duration	director_name	director_fb	...
0	Avatar	2009.0	Color	PG-13	178.0	James Cameron	0.0	...
1	Pirates of the Caribbean: ...	2007.0	Color	PG-13	169.0	Gore Verbinski	563.0	...
2	Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	0.0	...

9.7 Changing Display Options

pandas gives you the ability to change how the output on your screen is displayed. For instance, the default number of columns displayed for a DataFrame is 20, meaning that if your DataFrame has more than 20 columns then only the first and last 10 columns will be shown on the screen.

Get current option value with `get_option`

You can retrieve any option with the `get_option` function. Notice that this is not a DataFrame method, but instead a function that is accessed directly from `pd`. It is not necessary to remember the option names. They are all available in the docstrings of the `get_option`. The official documentation also provides descriptions for all available options. Below are three of the most common options to change.

[27]:
```
pd.get_option('display.max_columns')
```

[27]:
```
20
```

[28]:
```
pd.get_option('display.max_rows')
```

[28]:
```
60
```

[29]:
```
pd.get_option('display.max_colwidth')
```

[29]:
```
30
```

Use the `set_option` function to change an option value

To set a new option value, use the `set_option` function. You can set as many options as you would like at one time. It's usage is a bit strange. Pass it the option name as a string and follow it immediately with the value you want to

set it to. Continue this pattern of option name followed by new value to set as many options as you desire. Below, we set the maximum number of columns to 40 and the maximum number of rows to 4. We will now be able to view all the columns in the movie DataFrame.

[30]:
```
pd.set_option('display.max_columns', 40, 'display.max_rows', 4)
```

[31]:
```
movie
```

[31]:

title	year	color	content_rating	duration	director_name	director_fb	...
Avatar	2009.0	Color	PG-13	178.0	James Cameron	0.0	...
Pirates of the Caribbean: A...	2007.0	Color	PG-13	169.0	Gore Verbinski	563.0	...
Shanghai Calling	2012.0	Color	PG-13	100.0	Daniel Hsia	0.0	...
My Date with Drew	2004.0	Color	PG	90.0	Jon Gunn	16.0	...

9.8 Exercises

You may wish to change the display options before completing the exercises.

Exercise 1

Read in the movie dataset and set the index to be something other than movie title. Are there any other good columns to use as an index?

[ ]:
```

```

Exercise 2

Use `set_index` to set the index and keep the column as part of the data. Read the docstrings to find the parameter that controls this functionality.

[ ]:
```

```

Exercise 3

Read in the movie DataFrame and set the index as the title column. Assign the index to its own variable and output the last 10 movies titles.

[ ]:
```

```

Exercise 4

Use an integer instead of the column name for `index_col` when reading in the data using `read_csv`. What does it do?

[ ]:
```

```

Exercise 5

Use `pd.reset_option('all')` to reset the options to their default values. Test that this worked.

[ ]:
```

```

Chapter 10

Five-Step Process for Data Exploration

Major issues arise for beginners when too many lines of code are written in a single cell of a notebook. It's important to get feedback on every single line of code that you write and verify that it is in fact correct. Only once you have verified the result should you move on to the next line of code. To help increase your ability to do data exploration in Jupyter Notebooks, I recommend the following five-step process:

1. Write and execute a single line of code to explore your data
2. Verify that this line of code works by inspecting the output
3. Assign the result to a variable
4. Within the same cell, in a second line, output the head of the DataFrame or Series
5. Continue to the next cell. Do not add more lines of code to the cell

Apply to every part of the analysis

You can apply this five-step process to every part of your data analysis. Let's begin by reading in the bikes dataset and applying the five-step process for setting the index of our DataFrame as the `trip_id` column.

[1]:
```
import pandas as pd
bikes = pd.read_csv('../data/bikes.csv')
bikes.head(3)
```

[1]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...

Step 1: Write and execute a single line of code to explore your data

In this step, we call the `set_index` method to be the `trip_id` column.

[2]:
```
bikes.set_index('trip_id').head(3)
```

[2]:

trip_id	usertype	gender	starttime	stoptime	tripduration	...
7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...
7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...
10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...

Step 2: Verify that this line of code works by inspecting the output

Looking above, the output appears to be correct. The `trip_id` column has been set as the index and is no longer a column.

Step 3: Assign the result to a variable

You would normally do this step in the same cell, but for this demonstration, we will place it in the cell below.

[3]:
```
bikes2 = bikes.set_index('trip_id')
```

Step 4: Within the same cell, in a second line, output the head of the DataFrame or Series

Again, all these steps would be combined in the same cell.

[4]:
```
bikes2.head(3)
```

[4]:

trip_id	usertype	gender	starttime	stoptime	tripduration	...
7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...
7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...
10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...

Step 5: Continue to the next cell. Do not add more lines of code to the cell

It is tempting to do more analysis in a single cell. I advise against doing so when you are a beginner. By limiting your analysis to a single main line of code per cell, and outputting that result, you can easily trace your work from one step to the next. Most lines of code in a notebook will apply some operation to the data. It is vital that you can see exactly what this operation is doing. If you put multiple lines of code in a single cell, you lose track of what is happening and can't easily determine the veracity of each operation.

All steps in one cell

The five-step process was shown above one step at a time in different cells. When you actually explore data with this process, you would complete it in a single cell and up with the following result.

[5]:
```
bikes2 = bikes.set_index('trip_id')
bikes2.head(3)
```

[5]:

trip_id	usertype	gender	starttime	stoptime	tripduration	...
7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...
7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...
10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...

More examples

Let's see a more complex example of the five-step process. Let's find the `from_station_name` that has the longest average trip duration. This example will be completed with two rounds of the five-step process. First we will find the average trip duration for each station and then we will sort it. This example uses the `groupby` method which is covered in the **Grouping Data** part of the book.

[6]:
```
avg_td = bikes.groupby('from_station_name').agg({'tripduration':'mean'})
avg_td.head(3)
```

[6]:

from_station_name	tripduration
2112 W Peterson Ave	911.625000
63rd St Beach	1027.666667
900 W Harrison	495.500000

After grouping, we can sort from greatest to least.

[7]:
```
top_stations = avg_td.sort_values('tripduration', ascending=False)
top_stations.head(3)
```

[7]:

from_station_name	tripduration
Western Blvd & 48th Pl	7902.000000
Kedzie Ave & Lake St	5474.823529
Ridge Blvd & Howard St	4839.666667

While it is possible to complete this exercise in a single cell, I recommend executing only a single main line of code that explores the data.

No strict requirement for one line of code

The above examples each had a single main line of code followed by outputting the head of the DataFrame. Often times there will be a few more simple lines of code that can be written in the same cell. You should not strictly adhere to writing a single line of code, but instead, think about keeping the amount of code written in a single cell to a minimum.

For instance, the following block is used to select a subset of the data with three lines of code. The first is simple and creates a list of column names as strings. This is an instance where multiple lines of code are easily interpreted.

[8]:
```
cols = ['gender', 'tripduration']
bikes_gt = bikes[cols]
bikes_gt.head(3)
```

[8]:

	gender	tripduration
0	Male	993
1	Male	623
2	Male	1040

When to assign the result to a variable

Not all operations on our data will need to be assigned to a variable. We might just be interested in seeing the results. But, for many operations, you will want to continue with the new transformed data. By assigning the result to a variable, you will have immediate access to the result.

When to create a new variable name

During step 3 of the first example, the result of our new dataset was assigned to `bikes2`. We could have reassigned the result back to `bikes` and continued on with our analysis. When first exploring data, I recommend creating a new variable for each major result. By doing so, you will have preserved each step of your work and be able to inspect it later on. Creating new variables makes it much easier to find errors at different places in your analysis.

When to reuse variable names

The downside to using new variable names is that each variable can hold a copy of the data and if your dataset is large, you might run out of memory. By reassigning a result to the same variable name, you'll reduce memory used.

Another time to reuse variable names is when you are confident that the analysis you have produced is correct and no longer need to preserve all the previous results.

Continuously verifying results

Regardless of how adept you become at doing data explorations, it is good practice to verify each line of code. Data science is difficult and it is easy to make mistakes. Data is also messy and it is good to be skeptical while proceeding through an analysis. Getting visual verification that each line of code is producing the desired result is important. Doing this also provides feedback to help you think about what avenues to explore next.

Chapter 11

Solutions

11.1 2. The DataFrame and Series

[1]:
```
import pandas as pd
import numpy as np

pd.options.display.max_columns = 40
bikes = pd.read_csv('../data/bikes.csv')
```

Exercise 1

Select the column `events`, the type of weather that was recorded, and assign it to a variable with the same name. Output the first 10 values of it.

[2]:
```
events = bikes['events']
events.head(10)
```

[2]:

0	mostlycloudy
1	partlycloudy
2	mostlycloudy
3	mostlycloudy
4	partlycloudy
5	mostlycloudy
6	cloudy
7	cloudy
8	cloudy
9	mostlycloudy

Exercise 2

What type of object is `events`?

[3]:
```
# it's a Series
type(events)
```

[3]:
```
pandas.core.series.Series
```

Exercise 3

Select the last 2 rows of the `bikes` DataFrame and assign it to the variable `bikes_last_2`. What type of object is `bikes_last_2`?

[4]:
```
# it's a DataFrame
bikes_last_2 = bikes.tail(2)
type(bikes_last_2)
```

[4]:
```
pandas.core.frame.DataFrame
```

11.2 3. Data Types and Missing Values

Exercise 1

What type of object is returned from the `dtypes` attribute?

[5]:
```
# a Series
type(bikes.dtypes)
```

[5]:
```
pandas.core.series.Series
```

Exercise 2

What type of object is returned from the `shape` attribute?

[6]:
```
# a tuple of rows, columns
type(bikes.shape)
```

[6]:
```
tuple
```

Exercise 3

What type of object is returned from the `info` method?

The object `None` is returned. What you see is just output printed to the screen.

[7]:
```
info_return = bikes.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50089 entries, 0 to 50088
Data columns (total 19 columns):
trip_id               50089 non-null int64
usertype              50089 non-null object
gender                50089 non-null object
starttime             50089 non-null object
stoptime              50089 non-null object
tripduration          50089 non-null int64
from_station_name     50089 non-null object
latitude_start        50083 non-null float64
longitude_start       50083 non-null float64
dpcapacity_start      50083 non-null float64
to_station_name       50089 non-null object
latitude_end          50077 non-null float64
longitude_end         50077 non-null float64
dpcapacity_end        50077 non-null float64
temperature           50089 non-null float64
visibility            50089 non-null float64
wind_speed            50089 non-null float64
precipitation         50089 non-null float64
events                50089 non-null object
dtypes: float64(10), int64(2), object(7)
memory usage: 7.3+ MB
```

[8]:
```
type(info_return)
```

[8]:
```
NoneType
```

Exercise 4

The memory usage from the `info` method isn't correct when you have objects in your DataFrame. Read the docstrings from it and get the true memory usage.

[9]:
```
bikes.info(memory_usage='deep')
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50089 entries, 0 to 50088
Data columns (total 19 columns):
trip_id              50089 non-null int64
usertype             50089 non-null object
gender               50089 non-null object
starttime            50089 non-null object
stoptime             50089 non-null object
tripduration         50089 non-null int64
from_station_name    50089 non-null object
latitude_start       50083 non-null float64
longitude_start      50083 non-null float64
dpcapacity_start     50083 non-null float64
to_station_name      50089 non-null object
latitude_end         50077 non-null float64
longitude_end        50077 non-null float64
dpcapacity_end       50077 non-null float64
temperature          50089 non-null float64
visibility           50089 non-null float64
wind_speed           50089 non-null float64
precipitation        50089 non-null float64
events               50089 non-null object
dtypes: float64(10), int64(2), object(7)
memory usage: 28.9 MB
```

11.3 4. Setting a meaningful index

Exercise 1

Read in the movie dataset and set the index to be something other than movie title. Are there any other good columns to use as an index?

[10]:
```
movies = pd.read_csv('../data/movie.csv', index_col='director_name')
movies.head(3)
```

[10]:

director_name	title	year	color	content_rating	duration	director_fb	...
James Cameron	Avatar	2009.0	Color	PG-13	178.0	0.0	...
Gore Verbinski	Pirates of the Caribbean: ...	2007.0	Color	PG-13	169.0	563.0	...
Sam Mendes	Spectre	2015.0	Color	PG-13	148.0	0.0	...

Director name isn't unique. There aren't any other good column names to use as an index.

Exercise 2

Use `set_index` to set the index and keep the column as part of the data. Read the docstrings to find the parameter that controls this functionality.

[11]:
```
movies = pd.read_csv('../data/movie.csv')
movies = movies.set_index('title', drop=False)
movies.head(3)
```

[11]:

title	title	year	color	content_rating	duration	...
Avatar	Avatar	2009.0	Color	PG-13	178.0	...
Pirates of the Caribbean: A...	Pirates of the Caribbean: ...	2007.0	Color	PG-13	169.0	...
Spectre	Spectre	2015.0	Color	PG-13	148.0	...

Exercise 3

Read in the movie DataFrame and set the index as the title column. Assign the index to its own variable and output the last 10 movies titles.

[12]:
```
movies = pd.read_csv('../data/movie.csv', index_col='title')
index = movies.index
index[-10:]
```

[12]:
```
Index(['Primer', 'Cavite', 'El Mariachi', 'The Mongol King', 'Newlyweds',
       'Signed Sealed Delivered', 'The Following', 'A Plague So Pleasant',
       'Shanghai Calling', 'My Date with Drew'],
      dtype='object', name='title')
```

Exercise 4

Use an integer instead of the column name for `index_col` when reading in the data using `read_csv`. What does it do?

[13]:
```
movies = pd.read_csv('../data/movie.csv', index_col=-5)
movies.head(3)
```

[13]:

plot_keywords	title	year	color	content_rating	duration	...
avatar | future | marine | native...	Avatar	2009.0	Color	PG-13	178.0	...
goddess | marriage ceremony | m...	Pirates of the Caribbean: ...	2007.0	Color	PG-13	169.0	...
bomb | espionage | sequel | spy | t...	Spectre	2015.0	Color	PG-13	148.0	...

It chooses the column name with that integer location.

Exercise 5

Use `pd.reset_option('all')` to reset the options to their default values. Test that this worked.

[14]:
```
pd.reset_option('all')
```

Part III

Selecting Subsets of Data

Chapter 12

Selecting Subsets of Data from DataFrames with just the brackets

12.1 Selecting Subsets of Data

One of the most common tasks during a data analysis is to select a subset of the dataset. In pandas, this means selecting particular rows and/or columns from a DataFrame or Series. Although subset selection sounds like an easy task, and is an easy task for many other data manipulation tools, it's actually quite complex with pandas.

Examples of Selections of Subsets of Data

The following images show different types of subset selection that are possible. We will first highlight the values we want to select and then show the corresponding DataFrame after the completed selection.

Selection of columns

The most common subset selection, involves selecting one or more of the columns of a DataFrame. In this example, we select the `color`, `age`, and `height` columns.

	state	color	food	age	height	score
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

Resulting DataFrame:

	color	age	height
Jane	blue	30	165
Niko	green	2	70
Aaron	red	12	120
Penelope	white	4	80
Dean	gray	32	180
Christina	black	33	172
Cornelia	red	69	150

Selection of rows

Subsets of rows are a less frequent selection. In this example, we select the rows labeled `Aaron` and `Dean`.

	state	color	food	age	height	score
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

Resulting DataFrame:

	state	color	food	age	height	score
Aaron	FL	red	Mango	12	120	9.0
Dean	AK	gray	Cheese	32	180	1.8

Simultaneous selection of rows and columns

The last type of subset selection involves selecting rows and columns simultaneously. In this example, we select the `color`, `age`, and `height` columns along with the rows labeled `Aaron` and `Dean`.

	state	color	food	age	height	score
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

Resulting DataFrame:

	color	age	height
Aaron	red	12	120
Dean	gray	32	180

12.2 pandas dual references: by label and by integer location

As previously mentioned, the index of each DataFrame provides a label to reference each individual row. Similarly, the column names provide a label to reference each column. What hasn't been mentioned, is that each row and column may be referenced by an integer as well. I call this **integer location**. The integer location begins at 0 for the first row and continues sequentially one integer at a time until the last row. The last row will have integer location `n - 1`, where `n` is the total number of rows in the DataFrame.

Take a look above at our sample DataFrame one more time. The rows with labels `Aaron` and `Dean` can also be referenced by their respective integer locations 2 and 4. Similarly, the columns `color`, `age`, and `height` can be referenced by their integer locations 1, 3, and 4.

The official pandas documentation refers to integer location as **position**. I don't particularly like this terminology as it's not as explicit as integer location. The key term here is **integers**.

What's the difference between indexing and selecting subsets of data?

The documentation uses the term **indexing** frequently. This term means is a shorter, more technical term that implies **subset selection**. I prefer the term subset selection as, again, it is more descriptive of what is actually happening. Indexing is also the term used in the official Python documentation (for selecting subsets of lists or strings for example).

12.3 The three indexers [ ], loc, iloc

pandas provides three **indexers** to select subsets of data. An indexer is a term for one of `[ ]`, `loc`, or `iloc` and what makes the subset selection. All the details on how to make selections with each of these indexers will be covered. Each indexer has different rules for how it works. All of our selections will look similar to the following, except they will have something placed within the brackets.

```
>>> df[]
>>> df.loc[]
>>> df.iloc[]
```

Terminology

When the brackets are placed directly after the DataFrame, the term **just the brackets** will be used to differentiate them from the brackets after `loc` and `iloc`.

Square brackets instead of parentheses

One of the most common mistakes when using `loc` and `iloc` is to append parentheses to them, instead of square brackets. One of the main reasons this mistake is done is because `loc` and `iloc` appear to be methods and all methods are called with parentheses. Both `loc` and `iloc` are not methods, but are accessed in the same manner as methods through the dot notation, which leads to the mistake.

Few objects accessed through the dot notation will use brackets instead of parentheses. It helps to know that the brackets are a universal operator for selecting subsets of data regardless of the type of object. The brackets select subsets of lists, strings, and select a single value in a dictionary. numpy arrays use the brackets operator for subset selection. So, if you are doing subset selection, it's likely that the object uses brackets and not parentheses.

12.4 Begin with *just the brackets*

As we saw in a previous chapter, just the brackets are used to select a single column as a Series. We place the column name inside the brackets to return the Series. Let's read in a simple, small DataFrame and select a single column.

[1]:
```
import pandas as pd
df = pd.read_csv('../data/sample_data.csv', index_col=0)
df
```

[1]:

	state	color	food	age	height	score
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

Append square brackets directly to the name of the DataFrame and then place the name of the column within those brackets. A single column of data is selected as a Series.

[2]:
```
df['color']
```

[2]:

Jane	blue
Niko	green
Aaron	red
Penelope	white
Dean	gray
Christina	black
Cornelia	red

12.5 Select Multiple Columns with a List

You can select multiple columns by placing them in a list inside of just the brackets. Notice that a DataFrame and NOT a Series is returned.

[3]:
```
df[['color', 'age', 'score']]
```

[3]:

	color	age	score
Jane	blue	30	4.6
Niko	green	2	8.3
Aaron	red	12	9.0
Penelope	white	4	3.3
Dean	gray	32	1.8
Christina	black	33	9.5
Cornelia	red	69	2.2

You must use an inner set of brackets

You might be tempted to do the following which will NOT work. When selecting multiple columns, you must use a **list** to contain the names. Remember that a list is defined by a set of square brackets, so the following will raise an error.

[4]:
```
df['color', 'age', 'score']
```

```
KeyError: ('color', 'age', 'score')
```

Use two lines of code to select multiple columns

To help clarify the process of making subset selection, I recommend using intermediate variables. In this instance, we can assign the columns we would like to select to a list and then pass this list to the brackets.

[5]:
```
cols = ['color', 'age', 'score']
df[cols]
```

[5]:

	color	age	score
Jane	blue	30	4.6
Niko	green	2	8.3
Aaron	red	12	9.0
Penelope	white	4	3.3
Dean	gray	32	1.8
Christina	black	33	9.5
Cornelia	red	69	2.2

Column order does not matter

You can create new DataFrames in any column order you wish. They do need not match the original column order.

[6]:
```
cols = ['height', 'age']
df[cols]
```

[6]:

	height	age
Jane	165	30
Niko	70	2
Aaron	120	12
Penelope	80	4
Dean	180	32
Christina	172	33
Cornelia	150	69

12.6 Exercises

For the following exercises, make sure to use the movie dataset with `title` set as the index. It's good practice to shorten your output with the `head` method.

Exercise 1

Select the column with the director's name as a Series

[ ]:

Exercise 2

Select the column with the director's name and number of Facebook likes.

[ ]:

Exercise 3

Select a single column as a DataFrame and not a Series

[ ]:

Exercise 4

Look in the data folder and read in another dataset. Select some columns from it.

[ ]:

Chapter 13

Selecting Subsets of Data from DataFrames with `loc`

13.1 Subset selection with `loc`

The `loc` indexer selects data in a different manner than *just the brackets* and has its own set of rules that we must learn.

Simultaneous row and column subset selection with `loc`

The `loc` indexer can select rows and columns simultaneously. This is not possible with *just the brackets*. This is done by separating the row and column selections with a **comma**. The selection will look something like this:

```
df.loc[rows, cols]
```

`loc` primarily selects data by label

Very importantly, `loc` primarily selects data by the **label** of the rows and columns. Provide `loc` with the label of the rows and/or columns you would like to select. It also makes selections via boolean selection, a topic covered in a later chapter.

Read in data

Let's get started by reading in a sample DataFrame.

[1]:
```
import pandas as pd
df = pd.read_csv('../data/sample_data.csv', index_col=0)
df
```

[1]:

	state	color	food	age	height	score
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

Select two rows and three columns with `loc`

Let's make our first selection with `loc` by simultaneously selecting some rows and some columns. Let's select the rows `Dean` and `Cornelia` along with the columns `age`, `state`, and `score`. A list is used to contain both the row and column selection before being placed within the brackets. Row and column selection must be separated by a comma.

[2]:
```
rows = ['Dean', 'Cornelia']
cols = ['age', 'state', 'score']
df.loc[rows, cols]
```

[2]:

	age	state	score
Dean	32	AK	1.8
Cornelia	69	TX	2.2

The possible types of row and column selections

In the above example, we used a list of labels for both the row and column selection. You are not limited to just lists. All of the following are valid objects available for both row and column selections with `loc`.

- A single label
- A list of labels
- A slice with labels
- A boolean Series or array (covered in a later chapter)

Select two rows and a single column

Let's select the rows `Aaron` and `Dean` along with the `food` column. We can use a list for the row selection and a single string for the column selection.

[3]:
```
rows = ['Dean', 'Aaron']
cols = 'food'
df.loc[rows, cols]
```

[3]:

Dean	Cheese
Aaron	Mango

Series Returned

In the above example, a Series and not a DataFrame was returned. Whenever you select a single row or a single column using a string label, pandas will return a Series

13.2 `loc` with slice notation

Lists, tuples, and strings are the core Python objects that allow subset selection with slice notation. This same notation is allowed with DataFrames. Let's select all of the rows from `Jane` to `Penelope` with slice notation along with the columns `state` and `color`.

[4]:
```
cols = ['state', 'color']
df.loc['Jane':'Penelope', cols]
```

[4]:

	state	color
Jane	NY	blue
Niko	TX	green
Aaron	FL	red
Penelope	AL	white

Slice notation is inclusive of the stop label

Slice notation with the `loc` indexer is inclusive of the stop label. This functions differently that slicing done on Python lists, which is exclusive of the stop integer.

Slice notation only works within the brackets attached to the object

Python only allows us to use slice notation within the brackets that are attached to an object. If we try and assign slice notation outside of this, we will get a syntax error like we do below.

[5]:
```
rows = 'Jane':'Penelope'
```

```
SyntaxError: invalid syntax
```

Slice both the rows and columns

Both row and column selections support slice notation. In the following example, we slice all the rows up to and including label `Dean` along with columns from `height` until the end.

[6]:
```
df.loc[:'Dean', 'height':]
```

[6]:

	height	score
Jane	165	4.6
Niko	70	8.3
Aaron	120	9.0
Penelope	80	3.3
Dean	180	1.8

Selecting all of the rows and some of the columns

It is possible to use slice notation to select all of the rows and done with a single colon. In this example, we select all of the rows and two of the columns.

[7]:
```
cols = ['food', 'color']
df.loc[:, cols]
```

[7]:

	food	color
Jane	Steak	blue
Niko	Lamb	green
Aaron	Mango	red
Penelope	Apple	white
Dean	Cheese	gray
Christina	Melon	black
Cornelia	Beans	red

Could have used *just the brackets*

It is not necessary to use `loc` to for this selection as ultimately, it is just a selection of two columns. This could have been accomplished with *just the brackets*.

[8]:
```
cols = ['food', 'color']
df[cols]
```

[8]:

	food	color
Jane	Steak	blue
Niko	Lamb	green
Aaron	Mango	red
Penelope	Apple	white
Dean	Cheese	gray
Christina	Melon	black
Cornelia	Beans	red

A single colon is slice notation for select all

That single colon might be intimidating but it is technically slice notation that selects all items. In the following example, all of the elements of a Python list are selected using a single colon.

[9]:
```
a_list = [1, 2, 3, 4, 5, 6]
a_list[:]
```

[9]:
```
[1, 2, 3, 4, 5, 6]
```

Use a single colon to select all the columns

It is possible to use a single colon to represent a slice of all the rows or all of the columns. Below, a colon is used as slice notation for all of the columns.

[10]:
```
rows = ['Penelope','Cornelia']
df.loc[rows, :]
```

[10]:

	state	color	food	age	height	score
Penelope	AL	white	Apple	4	80	3.3
Cornelia	TX	red	Beans	69	150	2.2

The above can be shortened

By default, pandas will select all of the columns if you only provide a row selection. Providing the colon is not necessary and the following will do the same.

[11]:
```
rows = ['Penelope', 'Cornelia']
df.loc[rows]
```

[11]:

	state	color	food	age	height	score
Penelope	AL	white	Apple	4	80	3.3
Cornelia	TX	red	Beans	69	150	2.2

Though it is not syntactically necessary, one reason to use the colon is to reinforce the idea that `loc` may be used for simultaneous column selection. The first object passed to `loc` always selects rows and the second always selects columns.

Use slice notation to select a range of rows with all of the columns

Similarly, we can slice from `Niko` through `Dean` while selecting all of the columns. Again, we do not provide a specific column selection to return all of the columns.

[12]:
```
df.loc['Niko':'Dean']
```

[12]:

	state	color	food	age	height	score
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8

Again, you could have written the above as `df.loc['Niko':'Dean', :]` to reinforce the fact that `loc` first selects rows and then columns.

Changing the step size

The step size must be an integer when using slice notation with `loc`. In this example, we select every other row beginning at `Niko` and ending at `Christina`.

[13]:
```
df.loc['Niko':'Christina':2]
```

[13]:

	state	color	food	age	height	score
Niko	TX	green	Lamb	2	70	8.3
Penelope	AL	white	Apple	4	80	3.3
Christina	TX	black	Melon	33	172	9.5

13.3 Select a single row and a single column

If the row and column selections are both a single label, then a scalar value and NOT a DataFrame or Series is returned.

[14]:
```
rows = 'Jane'
cols = 'state'
df.loc[rows, cols]
```

[14]:
```
'NY'
```

Select a single row as a Series with `loc`

The `loc` indexer will return a single row as a Series when given a single row label. Let's select the row for `Niko`. Notice that the column names have now become index labels.

[15]:
```
df.loc['Niko']
```

[15]:

state	TX
color	green
food	Lamb
age	2
height	70
score	8.3

Confusing output

This output is potentially confusing. The once horizontal `Niko` row has now been visually represented as a vertical Series. It has an appearance of a column, but if you look at the values, you will see they align with their old column names.

13.4 Summary of `loc`

- Primarily uses labels
- Selects rows and columns simultaneously
- Selection can be a single label, a list of labels, a slice of labels, or a boolean Series/array
- A comma separates row and column selections

13.5 Exercises

Exercise 1

Read in the movie dataset and set the title column as the index. Select all columns for the movie 'The Dark Knight Rises'.

[ ]:
```

```

Exercise 2

Select all columns for the movies 'Tangled' and 'Avatar'.

[ ]:

Exercise 3

What year was 'Tangled' and 'Avatar' made and what was their IMBD scores?

[ ]:

Exercise 4

Can you tell what the data type of the `year` column is by just looking at its values?

[16]:
```
# Turn this into a markdown cell and write your answer here
```

Exercise 5

Use a single method to output the data type and number of non-missing values of `year`. Is it missing any?

[ ]:

Exercise 6

Select every 300th movie between 'Tangled' and 'Forrest Gump'. Why doesn't 'Forrest Gump' appear in the results?

[ ]:

Chapter 14

Selecting Subsets of Data from DataFrames with `iloc`

14.1 Getting started with `iloc`

The `iloc` indexer is very similar to `loc` but only uses **integer location** to make its selections. The word `iloc` itself stands for integer location so that should help remind you what it does.

Simultaneous row and column subset selection with `iloc`

Selection with iloc will look like the following with a comma separating the row and column selections.

```
df.iloc[rows, cols]
```

Let's read in some sample data and then begin making selections with integer location.

[1]:
```
import pandas as pd
df = pd.read_csv('../data/sample_data.csv', index_col=0)
df
```

[1]:

	state	color	food	age	height	score
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

Use a list for both rows and columns

Let's select rows with integer location 2 and 4 along with the first and last columns. It is possible to use negative integers in the same manner as Python lists.

[2]:
```
rows = [2, 4]
cols = [0, -1]
df.iloc[rows, cols]
```

[2]:

	state	score
Aaron	FL	9.0
Dean	AK	1.8

The possible types of selections for `iloc`

In the above example, we used a list of integers for both the row and column selection. You are not limited to just lists. All of the following are valid objects available for both row and column selections with `iloc`. The `iloc` indexer, unlike `loc`, is unable to do boolean selection.

- A single integer
- A list of integers
- A slice with integers

Slice the rows and use a list for the columns

Let's use slice notation to select rows with integer location 2 and 3 and a list to select columns with integer location 4 and 2. Notice that the stop integer location is **excluded** with `iloc`, which is exactly how slicing works with Python lists, tuples, and strings. Slicing with `loc` is **inclusive** of the stop label.

[3]:
```
cols = [4, 2]
df.iloc[2:4, cols]
```

[3]:

	height	food
Aaron	120	Mango
Penelope	80	Apple

Use a list for the rows and a slice for the columns

In this example, we use a list for the row selection and slice notation for the columns.

[4]:
```
rows = [5, 2, 4]
df.iloc[rows, 3:]
```

[4]:

	age	height	score
Christina	33	172	9.5
Aaron	12	120	9.0
Dean	32	180	1.8

Selecting some rows and all of the columns

If you leave the column selection empty, then all of the columns will be selected.

[5]:
```
rows = [3, 2]
df.iloc[rows]
```

[5]:

	state	color	food	age	height	score
Penelope	AL	white	Apple	4	80	3.3
Aaron	FL	red	Mango	12	120	9.0

It is possible to rewrite the above with both row and column selections by using a colon to represent a slice of all of the columns. Just as with `loc`, this can be instructive and reinforce the concept that the `iloc` does simultaneous row and column selection with the row selection first.

[6]:
```
df.iloc[rows, :]
```

[6]:

	state	color	food	age	height	score
Penelope	AL	white	Apple	4	80	3.3
Aaron	FL	red	Mango	12	120	9.0

Select all of the rows and some of the columns

Let's use a single colon to create slice notation to select all of the rows and a list to select two columns.

[7]:
```
cols = [1, 5]
df.iloc[:, cols]
```

[7]:

	color	score
Jane	blue	4.6
Niko	green	8.3
Aaron	red	9.0
Penelope	white	3.3
Dean	gray	1.8
Christina	black	9.5
Cornelia	red	2.2

Cannot do this with *just the brackets*

Just the brackets does select columns but it only understands **labels** and not **integer location**. The following produces an error as pandas is looking for column names that are the integers `1` and `5`.

[8]:
```
cols = [1, 5]
df[cols]
```

```
KeyError: "None of [Int64Index([1, 5], dtype='int64')] are in the [columns]"
```

Integer column names

pandas allows integers as column names and in fact you can have a mix of strings and integers (along with other types). So, if a column name was the integer 1, you would select it by writing `df[1]`. I would avoid using integer column names if possible as they do not provide descriptive names.

Select some rows and a single column

In this example, a list of integers is used for the rows along with a single integer for the columns. pandas returns a Series when a single integer is used to select either a row or column.

[9]:
```
rows = [2, 3, 5]
cols = 4
df.iloc[rows, cols]
```

[9]:

Aaron	120
Penelope	80
Christina	172

Select a single row or column as a DataFrame and NOT a Series

You can select a single row (or column) and return a DataFrame and not a Series if you use a list to make the selection. Let's replicate the selection from the previous example, but use a one-item list for the column selection.

[10]:
```
rows = [2, 3, 5]
cols = [4]
df.iloc[rows, cols]
```

[10]:

	height
Aaron	120
Penelope	80
Christina	172

Select a single row as a Series with `iloc`

By passing a single integer to `iloc`, it will select one row as a Series.

[11]:
```
df.iloc[2]
```

[11]:

state	FL
color	red
food	Mango
age	12
height	120
score	9

14.2 Summary of `iloc`

The `iloc` indexer is analogous to `loc` but only uses **integer location** for selection. The official pandas documentation refers to this as selection by **position**.

- Uses only integer location
- Selects rows and columns simultaneously
- Selection can be a single integer, a list of integers, or a slice of integers
- A comma separates row and column selections

14.3 Exericses

- Use the movie dataset for the following exercises

Exercise 1

Select the rows with integer location 10, 5, and 1

[ ]:
```

```

Exericse 2

Select the columns with integer location 10, 5, and 1

[ ]:
```

```

Exercise 3

Select rows with integer location 100 to 104 along with the column integer location 5.

[ ]:
```

```

Chapter 15

Selecting Subsets of Data from a Series

15.1 Using Dot Notation to Select a Column as a Series

Previously we learned how to use *just the brackets* to select a single column as a Series. Another common way to do this uses dot notation. Place the column name following a dot after the name of your DataFrame. Let's begin by reading in the movie dataset and setting the index as the title.

[1]:
```
import pandas as pd
movie = pd.read_csv('../data/movie.csv', index_col='title')
movie.head(3)
```

[1]:

title	year	color	content_rating	duration	director_name	director_fb	...
Avatar	2009.0	Color	PG-13	178.0	James Cameron	0.0	...
Pirates of the Caribbean: A...	2007.0	Color	PG-13	169.0	Gore Verbinski	563.0	...
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	0.0	...

Instead of using *just the brackets* to select a single column, you can use dot notation. Let's select the `year` column in such a manner.

[2]:
```
movie.year.head(3)
```

[2]:

title	2009.0
Avatar	2007.0
Pirates of the Caribbean: A...	2015.0

I don't recommend doing this

Although this is valid pandas syntax I don't recommend using this notation for the following reasons:

- You cannot select columns with spaces in them
- You cannot select columns that have the same name as a pandas method such as `count`
- You cannot use a variable name that is assigned to the name of a column

Using *just the brackets* **always** works so I recommend doing the following instead:

[3]:
```
movie['year'].head(3)
```

[3]:

title	2009.0
Avatar	2007.0
Pirates of the Caribbean: A...	2015.0

Why even know about this?

pandas is written differently by different people and you will definitely see this syntax around, so it's important to be aware of it.

15.2 Selecting Subsets of Data From a Series

Selecting subsets of data from a Series is very similar to that as a DataFrame. Since there are no columns in a Series, there isn't a need to use *just the brackets*. Instead, you can do all of your subset selection with `loc` and `iloc`. Let's select the `imdb_score` column as a Series and output the head.

[4]:
```
imdb = movie['imdb_score']
imdb.head(3)
```

[4]:

title	7.9
Avatar	7.1
Pirates of the Caribbean: A...	6.8

Selection with a scalar, a list, and a slice

Just like with a DataFrame, both `loc` and `iloc` accept either a single scalar, a list, or a slice. The `loc` indexer also accepts a boolean Series/array which will be covered in a later chapter. Let's select the IMDB score for 'Forrest Gump'. Since we are selecting a single label, only the value is returned.

[5]:
```
imdb.loc['Forrest Gump']
```

[5]:
```
8.8
```

Select the scores for both 'Forrest Gump' and 'Avatar' with a list. Notice that a Series is returned.

[6]:
```
locs = ['Forrest Gump', 'Avatar']
imdb.loc[locs]
```

[6]:

title	8.8
Forrest Gump	7.9

Select every 100th movie from 'Avatar' to 'Forrest Gump' with slice notation:

[7]:
```
imdb.loc['Avatar':'Forrest Gump':100]
```

[7]:

title	7.9
Avatar	6.7
The Fast and the Furious	7.5
Harry Potter and the Sorcer...	6.7
Epic	4.8
102 Dalmatians	5.6
Pompeii	6.3
Wall Street: Money Never Sl...	5.5
Hop	6.5

Repeat with `iloc`

The `iloc` indexer works analogously as `loc` on Series but only uses integer location. Let's make selections with a single integer, a list of integers, and a slice of integers. We'll begin by selecting the score of the 21st movie (integer location 20).

[8]:
```
imdb.iloc[20]
```

[8]:
```
7.5
```

In this example, we select three scores with a list.

[9]:
```
ilocs = [10, 20, 30]
imdb.iloc[ilocs]
```

[9]:

title	6.9
Batman v Superman: Dawn of ...	7.5
The Hobbit: The Battle of t...	7.8

Here is an example that uses slice notation.

[10]:
```
imdb.iloc[3000:3050:10]
```

[10]:

title	6.8
Quartet	5.4
The Guru	6.2
Machine Gun McCain	6.3
The Blue Butterfly	6.9

Trouble with *just the brackets*

It is possible to use just the brackets to make the same selections as above. See the following examples:

[11]:
```
imdb['Forrest Gump']
```

[11]:
```
8.8
```

[12]:
```
imdb['Avatar':'Forrest Gump':100]
```

[12]:

title	7.9
Avatar	6.7
The Fast and the Furious	7.5
Harry Potter and the Sorcer...	6.7
Epic	4.8
102 Dalmatians	5.6
Pompeii	6.3
Wall Street: Money Never Sl...	5.5
Hop	6.5

[13]:
```
ilocs = [10, 20, 30]
imdb[ilocs]
```

[13]:

title	6.9
Batman v Superman: Dawn of ...	7.5
The Hobbit: The Battle of t...	7.8

[14]:
```
imdb[3000:3050:10]
```

[14]:

title	6.8
Quartet	5.4
The Guru	6.2
Machine Gun McCain	6.3
The Blue Butterfly	6.9

Can you spot the problem?

The major issue is that using *just the brackets* is **ambiguous** and **not explicit**. We don't know if we are selecting by label or by integer location. With `loc` and `iloc`, it is clear what our intentions are. I suggest using `loc` and `iloc` for clarity.

15.3 Comparison to Python Lists and Dictionaries

It may be helpful to compare pandas ability to make selections by label and integer location to that of Python lists and dictionaries. Python lists allow for selection of data only through **integer location**. You can use a single integer or slice notation to make the selection but NOT a list of integers. Let's see examples of subset selection of lists using integers:

[15]:
```
a_list = [10, 5, 3, 89, 20, 44, 37]
```

[16]:
```
a_list[4]
```

[16]:
```
20
```

[17]:
```
a_list[-3:]
```

[17]:
```
[20, 44, 37]
```

Selection by label with Python dictionaries

All values in each dictionary are labeled by a key. We use this key to make single selections. Dictionaries only allow selection with a single label. Slices and lists of labels are not allowed.

[18]:
```
d = {'a':1, 'b':2, 't':20, 'z':26, 'A':27}
d['a']
```

[18]:
```
1
```

[19]:
```
d['A']
```

[19]:
```
27
```

pandas has the power of lists and dictionaries

DataFrames and Series are able to make selections with integers like a list and with labels like a dictionary.

15.4 Exercises

Exercise 1

Read in the bikes dataset. We will be using it for the next few questions. Select the wind speed column as a Series and assign it to a variable and output the head. What kind of index does this Series have?

[ ]:
```

```

Exercise 2

From the wind speed Series, select the integer locations 4 through, but not including 10

[ ]:
```

```

Exercise 3

Copy and paste your answer to problem 2 below but use `loc` instead. Do you get the same result? Why not?

[ ]:

Exercise 4

Read in the movie dataset and set the index to be the title. Select `actor1` as a Series. Who is the `actor1` for 'My Big Fat Greek Wedding'?

[ ]:

Exercise 5

Find `actor1` for your favorite two movies?

[ ]:

Exercise 6

Select the last 3 values from `actor1` using two different ways?

[ ]:

Chapter 16

Boolean Selection Single Conditions

16.1 Boolean Selection

Boolean Selection, also referred to as **Boolean Indexing**, is the process of selecting subsets of rows from DataFrames (or Series) based on the actual data values and NOT by their labels or integer locations. All of the previous subset selections where done using either labels or integer location and nothing to do with the actual values.

Examples of Boolean Selection

Let's see some examples of actual questions (in plain English) that boolean selection can help us answer from the bikes dataset. The term **query** is used to refer to these sorts of questions.

- Find all rides by males
- Find all rides with duration longer than 2 hours
- Find all rides that took place between March and June of 2015.
- Find all the rides with a duration longer than 2 hours by females with temperature higher than 90 degrees

All queries have a logical condition

Each of the above queries have a strict logical condition that must be checked one row at a time.

Keep or discard an entire row of data

If you were to manually answer the above queries, you would need to scan each row and determine whether the row, as a whole, meets the condition. If so, then it is kept in the result, otherwise it is discarded.

Each row will have a True or False value associated with it

When you perform boolean selection, each row of the DataFrame (or value of a Series) will have a `True` or `False` value associated with it that corresponds to the outcome of the logical condition.

Begin with a small DataFrame

We will perform our first boolean selection on our sample DataFrame. Let's read it in now.

[1]:
```
import pandas as pd
df = pd.read_csv('../data/sample_data.csv', index_col=0)
df
```

[1]:

	state	color	food	age	height	score
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

16.2 Manual filtering the data

Let's find all the people who are younger than 30 years old. We will do this manually by inspecting the data.

Create a list of booleans

By inspecting the data, we see that `Niko`, `Aaron`, and `Penelope` are all under 30 years old. To signify which people are under 30, we create a list of 7 boolean values corresponding to the 7 rows in the DataFrame. The values in the list that correspond with the positions of `Niko`, `Aaron`, and `Penelope` are `True`. All other values are `False`. `Niko`, `Aaron`, and `Penelope` are the 2nd, 3rd, and 4th rows, so these are the locations in the list that are `True`.

[2]:
```
filt = [False, True, True, True, False, False, False]
```

Variable name `filt`

The variable name `filt`, which is just an abbreviation for filter, will be used throughout the book to refer to the sequence of booleans. You are free to use any variable name you like for the sequence of booleans, but being consistent makes your code easier to understand.

Pass this list into just the brackets

The above list has `True` in the 2nd, 3rd, and 4th positions. These will be the rows that are kept in the result during boolean selection. Place the list inside the brackets to complete the selection.

[3]:
```
df[filt]
```

[3]:

	state	color	food	age	height	score
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3

Wait a second... Isn't `[ ]` just for column selection?

The primary purpose of *just the brackets* for a DataFrame is to select one or more columns by using either a string or a list of strings. All of a sudden, this example shows entire rows being selected with boolean values. This is what makes pandas, unfortunately, a confusing library to learn and use.

16.3 Operator Overloading

Just the brackets is **overloaded**. Depending on the inputs, pandas will do something completely different. Here are the rules for the different objects passed to *just the brackets*.

- **string**—return a column as a Series
- **list of strings**—return all those columns as a DataFrame
- **sequence of booleans**—select all rows where `True`
- **slice**—select rows (can do both label and integer location—confusing!) I never do this as it is ambiguous. This will be covered in the **Miscellaneous Subset Selection** chapter.

In summary, *just the brackets* primarily selects columns, but if you pass it a sequence of booleans it will select all rows that are `True`.

Using booleans in a Series and not a list

Instead of a list, we can use a Series of booleans, which will make the same selection. Below, we use the Series constructor to create a Series object. The Series must have the same index as the DataFrame it is selecting from in order to work properly, so we create it with the same index as the original DataFrame. This automatic alignment of the index is important and covered in the **Joining Data** part.

[4]:
```
filt = pd.Series([False, True, True, True, False, False, False],
                 index=df.index)
filt
```

[4]:

Jane	False
Niko	True
Aaron	True
Penelope	True
Dean	False
Christina	False
Cornelia	False

Use the boolean Series to do the boolean selection

Placing the Series directly in the brackets will again select only the rows that correspond with the `True` values in the Series.

[5]:
```
df[filt]
```

[5]:

	state	color	food	age	height	score
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3

16.4 Practical Boolean Selection

We will almost never create boolean lists/Series manually like we did above but instead use the actual data to create them.

Creating boolean Series from column data

By far the most common way to create a boolean Series will be from the values of one particular column. We will test a condition using one of the six comparison operators:

- `<`
- `<=`
- `>`
- `>=`
- `==`
- `!=`

Let's begin by reading in the bikes dataset.

[6]:
```
bikes = pd.read_csv('../data/bikes.csv')
bikes.head(3)
```

[6]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...

Create a boolean Series

Let's create a boolean Series by determining which rows have a trip duration greater than 1000 seconds. To make the comparison, we select the `tripduration` column as a Series and compare it against the integer 1000.

[7]:
```
filt = bikes['tripduration'] > 1000
filt.head(3)
```

[7]:

0	False
1	False
2	True

When we write `bikes['tripduration'] > 1000`, pandas compares each value in the `tripduration` column against 1000. It returns a new Series the same length as `tripduartion` with boolean values corresponding to the outcome of the comparison. Let's verify that the `filt` Series is the same length as the DataFrame.

[8]:
```
len(filt)
```

[8]:
```
50089
```

[9]:
```
len(bikes)
```

[9]:
```
50089
```

Manually verify correctness

Take a look at the `tripduration` column to manually verify that only the third row satisfied the condition.

Complete the boolean selection

We can now pass the `filt` boolean Series into *just the brackets* to filter the entire DataFrame to return all the rows in the bikes dataset that have a trip duration greater than 1000. Verify that all `tripduration` values are greater than 1000.

[10]:
```
bikes[filt].head(3)
```

[10]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...
8	21028	Subscriber	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	...
10	24383	Subscriber	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523	...

How many rows have a trip duration greater than 1000?

To answer this question, let's assign the result of the boolean selection to a variable and then compare the number of rows between it and the original DataFrame.

[11]:
```
bikes_duration_1000 = bikes[filt]
```

Let's find the number of rows in each DataFrame.

[12]:
```
len(bikes)
```

[12]:
```
50089
```

[13]:
```
len(bikes_duration_1000)
```

[13]:
```
10178
```

We compute that 20% of the rides are longer than 1,000 seconds.

[14]:
```
len(bikes_duration_1000) / len(bikes)
```

[14]:
```
0.20319830701351593
```

16.5 Boolean selection in one line

Often, you will see boolean selection completed in a single line of code instead of the two lines we used above. The expression for the filter is placed directly inside the brackets. Although this method will save a line of code, I recommend assigning the filter as a separate variable to help with readability.

[15]:
```
bikes[bikes['tripduration'] > 1000].head(3)
```

[15]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...
8	21028	Subscriber	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	...
10	24383	Subscriber	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523	...

16.6 Single condition expression

Our first example tested a single condition (whether the trip duration was 1,000 or more). Let's test a different single condition and find all the rides that happened when the weather was cloudy. We use the == operator to test for equality and again pass this variable to the brackets which completes our selection.

[16]:
```
filt = bikes['events'] == 'cloudy'
bikes[filt].head(3)
```

[16]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
6	18880	Subscriber	Male	2013-07-02 17:47:00	2013-07-02 17:56:00	565	...
7	19689	Subscriber	Male	2013-07-03 09:07:00	2013-07-03 09:16:00	505	...
8	21028	Subscriber	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	...

16.7 Exercises

Exercise 1

Read in the movie dataset and set the index to be the title. Select all movies that have Tom Hanks as `actor1`. How many of these movies has he starred in?

[ ]:
```

```

Exercise 2

Select movies with an IMDB score greater than 9.

[ ]:
```

```

Exercise 3

Write a function that accepts a single parameter to find the number of movies for a given content rating. Use the function to find the number of movies for ratings 'R', 'PG-13', and 'PG'.

[ ]:

Chapter 17

Boolean Selection Multiple Conditions

17.1 Multiple condition expression

So far, our boolean selections have involved a single condition. You can have as many conditions as you would like. To do so, you will need to combine your boolean expressions using the three logical operators, and, or, and not.

Use `&`, `|` , `~`

Although Python uses the keywords `and`, `or`, and `not`, these will not work with boolean Series. You must use the following operators instead.

- `&` for and (ampersand character)
- `|` for or (pipe character)
- `~` for not (tilde character)

Our first multiple condition expression

Let's find all the rides longer than 1,000 seconds when it was cloudy. This query has two conditions - trip durations greater than 1,000 and cloudy weather. The way we approach the problem is to assign each condition to a separate variable. Since we desire both of the conditions to be true, we must use the and (`&`) operator.

[1]:
```
import pandas as pd
bikes = pd.read_csv('../data/bikes.csv')
bikes.head(3)
```

[1]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...

Each single condition is placed on its own line before using the `&` operator to create the final filter that completes the boolean selection.

[2]:
```
filt1 = bikes['tripduration'] > 1000
filt2 = bikes['events'] == 'cloudy'
filt = filt1 & filt2
bikes[filt].head(3)
```

[2]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
8	21028	Subscriber	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	...
18	40924	Subscriber	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	...
80	90932	Subscriber	Female	2013-07-22 07:59:00	2013-07-22 08:19:00	1224	...

17.2 Multiple conditions in one line

It is possible to combine the entire expression into a single line. Many pandas users like doing this. Regardless, it is a good idea to know how it's done as you will definitely encounter it.

Use parentheses to separate conditions

You must encapsulate each condition in a set of parentheses in order to make this work. Each condition will be separated like this:

```
(bikes['tripduration'] > 1000) & (bikes['events'] == 'cloudy')
```

Same results

The above expression is placed inside of *just the brackets* to get the same results. Again, I prefer assigning each condition to its own variable for better readability.

[3]:
```
bikes[(bikes['tripduration'] > 1000) & (bikes['events'] == 'cloudy')].head(3)
```

[3]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
8	21028	Subscriber	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	...
18	40924	Subscriber	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	...
80	90932	Subscriber	Female	2013-07-22 07:59:00	2013-07-22 08:19:00	1224	...

17.3 Using an or condition

Let's find all the rides that were done by females **or** had trip durations longer than 1,000 seconds. In this example, we need at least one of the conditions to be true, which necessitates the use of the or (|) operator.

[4]:
```
filt1 = bikes['tripduration'] > 1000
filt2 = bikes['gender'] == 'Female'
filt = filt1 | filt2
bikes[filt].head(3)
```

[4]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...
8	21028	Subscriber	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	...
9	23558	Subscriber	Female	2013-07-04 15:00:00	2013-07-04 15:16:00	922	...

17.4 Inverting a condition with the not operator

The tilde character, ~, represents the not operator and inverts a condition. For instance, if we wanted all the rides with trip duration less than or equal to 1000, we could do it like this:

[5]:
```
filt = bikes['tripduration'] > 1000
bikes[~filt].head(3)
```

[5]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...
3	12907	Subscriber	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667	...

Of course, inverting single conditions is basically pointless as we can use the less than or equal to operator instead.

[6]:
```
filt = bikes['tripduration'] <= 1000
bikes[filt].head(3)
```

[6]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...
3	12907	Subscriber	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667	...

Invert a more complex condition

Typically, we will save the not operator for reversing more complex conditions. Let's reverse the condition for selecting rides by females or those with duration over 1,000 seconds. Logically, this should return only male riders with duration 1,000 or less.

[7]:
```
filt1 = bikes['tripduration'] > 1000
filt2 = bikes['gender'] == 'Female'
filt = filt1 | filt2
bikes[~filt].head(3)
```

[7]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...
3	12907	Subscriber	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667	...

Even more complex conditions

It is possible to build extremely complex conditions to select rows of your DataFrame that meet a very specific query. For instance, we can select males riders with trip duration between 1,000 and 2,000 seconds along with female riders with trip duration between 5,000 and 10,000 seconds. With multiple conditions, it's probably best to break out the logic into multiple steps:

[8]:
```
filt1 = ((bikes['gender'] == 'Male') &
         (bikes['tripduration'] >= 1000) &
         (bikes['tripduration'] <= 2000))

filt2 = ((bikes['gender'] == 'Female') &
         (bikes['tripduration'] >= 5000) &
         (bikes['tripduration'] <= 10000))
filt = filt1 | filt2
bikes[filt].head(10)
```

[8]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...
8	21028	Subscriber	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	...
10	24383	Subscriber	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523	...
11	24673	Subscriber	Male	2013-07-04 18:13:00	2013-07-04 18:42:00	1697	...
13	30404	Subscriber	Male	2013-07-06 09:43:00	2013-07-06 10:06:00	1365	...
26	51130	Subscriber	Male	2013-07-12 01:07:00	2013-07-12 01:24:00	1043	...
34	54257	Subscriber	Male	2013-07-12 18:13:00	2013-07-12 18:40:00	1616	...
40	61401	Subscriber	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	...
41	64257	Subscriber	Male	2013-07-15 06:26:00	2013-07-15 06:44:00	1125	...
47	67013	Subscriber	Male	2013-07-15 19:10:00	2013-07-15 19:34:00	1463	...

17.5 Lots of equality conditions in a single column - use `isin`

Occasionally, we will want to test equality in a single column with multiple values. This is most common in string columns. For instance, let's say we wanted to find all the rides where the events were either rain, snow, tstorms or sleet. One way to do this would be with four or conditions.

[9]:
```
filt = ((bikes['events'] == 'rain') |
        (bikes['events'] == 'snow') |
        (bikes['events'] == 'tstorms') |
        (bikes['events'] == 'sleet'))

bikes[filt].head(3)
```

[9]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
45	66336	Subscriber	Male	2013-07-15 16:43:00	2013-07-15 16:55:00	727	...
78	89180	Subscriber	Male	2013-07-21 16:35:00	2013-07-21 17:06:00	1809	...
79	89228	Subscriber	Male	2013-07-21 16:47:00	2013-07-21 17:03:00	999	...

Instead of using an operator, we will use the `isin` method. Pass it a list (or a set) of all the values you want to test equality with. The `isin` method will return a boolean Series and in this example, the same exact boolean Series as the previous one.

[10]:
```
filt = bikes['events'].isin(['rain', 'snow', 'tstorms', 'sleet'])
bikes[filt].head(3)
```

[10]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
45	66336	Subscriber	Male	2013-07-15 16:43:00	2013-07-15 16:55:00	727	...
78	89180	Subscriber	Male	2013-07-21 16:35:00	2013-07-21 17:06:00	1809	...
79	89228	Subscriber	Male	2013-07-21 16:47:00	2013-07-21 17:03:00	999	...

Combining `isin` with other filters

You can use the resulting boolean Series from the `isin` method in the same way as you would from the logical operators. For instance, If we wanted to find all the rides that had the same events as above and had a duration greater than 2,000 we would do the following:

[11]:
```
filt1 = bikes['events'].isin(['rain', 'snow', 'tstorms', 'sleet'])
filt2 = bikes['tripduration'] > 2000
filt = filt1 & filt2
bikes[filt].head(3)
```

[11]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
2344	1266453	Subscriber	Female	2014-03-19 07:23:00	2014-03-19 08:00:00	2181	...
7697	3557596	Subscriber	Male	2014-09-12 14:20:00	2014-09-12 14:57:00	2213	...
8357	3801419	Subscriber	Male	2014-09-30 08:21:00	2014-09-30 08:58:00	2246	...

17.6 Exercises

Exercise 1

Select all movies from the 1970s.

[ ]:
```

```

Exercise 2

Select all movies from the 1970s that had IMDB scores greater than 8.

[ ]:
```

```

Exercise 3

Select movies that were rated either R, PG-13, or PG.

[ ]:

Exercise 4

Select movies that are either rated PG-13 or were made after 2010.

[ ]:

Exercise 5

Find all the movies that have at least one of the three actors with more than 10,000 Facebook likes.

[ ]:

Exercise 6

Reverse the condition from problem 4. In words, what have you selected?

[ ]:

Chapter 18

Boolean Selection More

18.1 Boolean selection on a Series

All of the examples thus far have taken place on DataFrames. Boolean selection on a Series happens almost identically. Since there is only one dimension of data, the queries you ask are usually going to be simpler. First, let's select a single column of data as a Series such as the temperature column.

[1]:
```
import pandas as pd
bikes = pd.read_csv('../data/bikes.csv', parse_dates=['starttime', 'stoptime'])
temp = bikes['temperature']
temp.head(3)
```

[1]:
```
0    73.9
1    69.1
2    73.0
```

Let's select temperatures greater than 90. The procedure is the same as with DataFrames. Create a boolean Series and pass that Series to *just the bracketes*.

[2]:
```
filt = temp > 90
temp[filt].head(3)
```

[2]:
```
54    91.0
55    91.0
56    91.0
```

Select temperature less than 0 or greater than 95. Multiple condition boolean Series also work the same.

[3]:
```
filt1 = temp < 0
filt2 = temp > 95
filt = filt1 | filt2
temp[filt].head()
```

[3]:
```
395     96.1
396     96.1
397     96.1
1871    -2.0
2049    -2.0
```

Re-read data with `starttime` in the index

The default index is not very helpful. Let's reread the data with `starttime` in the index. While, this column may not be unique it does provide us with useful labels for each row.

[4]:
```
bikes = pd.read_csv('../data/bikes.csv',
                    parse_dates=['starttime', 'stoptime'],
                    index_col='starttime')
temp2 = bikes['temperature']
temp2.head()
```

[4]:

starttime	73.9
2013-06-28 19:01:00	69.1
2013-06-28 22:53:00	73.0
2013-06-30 14:43:00	72.0
2013-07-01 10:05:00	73.0

Let's select temperatures greater than 90. We expect to get a summer month and we do.

[5]:
```
filt = temp2 > 90
temp2[filt].head(5)
```

[5]:

starttime	91.0
2013-07-16 15:13:00	91.0
2013-07-16 15:31:00	91.0
2013-07-16 16:35:00	93.0
2013-07-17 17:08:00	93.0

Select temperature less than 0 or greater than 95. We expect to get some winter months in the result and we do.

[6]:
```
filt1 = temp2 < 0
filt2 = temp2 > 95
filt = filt1 | filt2
temp2[filt].head()
```

[6]:

starttime	96.1
2013-08-30 15:33:00	96.1
2013-08-30 15:37:00	96.1
2013-08-30 15:49:00	-2.0
2013-12-12 05:13:00	-2.0

18.2 The `between` method

The `between` method returns a boolean Series by testing whether the current value is between two given values. For instance, if want to select the temperatures between 50 and 60 degrees we do the following:

[7]:
```
filt = temp2.between(50, 60)
filt.head(3)
```

[7]:

starttime	False
2013-06-28 19:01:00	False
2013-06-28 22:53:00	False

By default, the `between` method is inclusive of the given values, so temperatures of exactly 50 or 60 would be found in the result. We pass this boolean Series to *just the brackets* to complete the selection.

[8]:
```
temp2[filt].head(3)
```

[8]:

starttime	54.0
2013-09-13 07:55:00	57.9
2013-09-13 08:04:00	57.9

18.3 Simultaneous boolean selection of rows and column labels with `loc`

The `loc` indexer was thoroughly covered in an earlier chapter and will now be brought up again to show how it can simultaneously selects rows with boolean selection and columns by labels.

Remember that `loc` takes both a row selection and a column selection separated by a comma. Since the row selection comes first, you can pass it the same exact inputs that you do for *just the brackets* and get the same results. Let's run some of the previous examples of boolean selection with `loc`.

[9]:
```
filt = bikes['tripduration'] > 1000
bikes.loc[filt].head(3)
```

[9]:

starttime	trip_id	usertype	gender	stoptime	tripduration	...
2013-06-30 14:43:00	10927	Subscriber	Male	2013-06-30 15:01:00	1040	...
2013-07-03 15:21:00	21028	Subscriber	Male	2013-07-03 15:42:00	1300	...
2013-07-04 17:17:00	24383	Subscriber	Male	2013-07-04 17:42:00	1523	...

[10]:
```
filt = bikes['events'].isin(['rain', 'snow', 'tstorms', 'sleet'])
bikes.loc[filt].head(3)
```

[10]:

starttime	trip_id	usertype	gender	stoptime	tripduration	...
2013-07-15 16:43:00	66336	Subscriber	Male	2013-07-15 16:55:00	727	...
2013-07-21 16:35:00	89180	Subscriber	Male	2013-07-21 17:06:00	1809	...
2013-07-21 16:47:00	89228	Subscriber	Male	2013-07-21 17:03:00	999	...

Separate row and column selection with a comma for `loc`

The nice benefit of `loc` is that it allows us to simultaneously do boolean selection for the rows and select columns by label. Let's select rides during rain or snow and the columns `events` and `tripduration`.

[11]:
```
filt = bikes['events'].isin(['rain', 'snow'])
cols = ['events', 'tripduration']
bikes.loc[filt, cols].head()
```

[11]:

starttime	events	tripduration
2013-07-15 16:43:00	rain	727
2013-07-26 19:10:00	rain	1395
2013-07-30 18:53:00	rain	442
2013-08-05 17:09:00	rain	890
2013-09-07 16:09:00	rain	978

Now let's find all female riders with trip duration greater than 5000 when it was cloudy. We'll only return the columns used during the boolean selection.

[12]:
```
filt1 = bikes['gender'] == 'Female'
filt2 = bikes['tripduration'] > 5000
filt3 = bikes['events'] == 'cloudy'
filt = filt1 & filt2 & filt3
cols = ['gender', 'tripduration', 'events']
bikes.loc[filt, cols]
```

[12]:

starttime	gender	tripduration	events
2014-04-15 15:56:00	Female	79988	cloudy
2015-07-08 14:52:00	Female	7197	cloudy
2016-03-15 12:44:00	Female	13205	cloudy
2017-02-28 15:07:00	Female	19922	cloudy

18.4 Column to column comparisons

So far, we have created filters by comparing each of our column values to a single scalar value. It is possible to do element-by-element comparisons by comparing two columns to one another. For instance, the total bike capacity at the each station at the start and end of the ride is stored in the `dpcapacity_start` and `dpcapacity_end` columns. If we wanted to test whether there were more capacity at the start of the ride vs the end, we would do the following:

[13]:
```
filt = bikes['dpcapacity_start'] > bikes['dpcapacity_end']
```

Let's use this filter with `loc` to return all the rows where the start capacity is greater than the end.

[14]:
```
cols = ['dpcapacity_start', 'dpcapacity_end']
bikes.loc[filt, cols].head(3)
```

[14]:

starttime	dpcapacity_start	dpcapacity_end
2013-06-28 22:53:00	31.0	19.0
2013-07-02 17:47:00	31.0	19.0
2013-07-03 15:21:00	31.0	15.0

Boolean selection with `iloc` does not work

The pandas developers decided not to allow boolean selection with `iloc`.

[15]:
```
bikes.iloc[filt]
```

```
ValueError: iLocation based boolean indexing cannot use an indexable as a mask
```

18.5 Finding Missing Values with `isna`

The `isna` method called from either a DataFrame or a Series returns `True` for every value that is missing and `False` for any other value. Let's see this in action by calling `isna` on the start capacity column.

[16]:
```
bikes['dpcapacity_start'].isna().head(3)
```

[16]:

starttime	False
2013-06-28 19:01:00	False
2013-06-28 22:53:00	False

Filtering for missing values

We can now use this boolean Series to select all the rows where the capacity start column is missing. Verify that those values are indeed missing.

[17]:
```
filt = bikes['dpcapacity_start'].isna()
bikes[filt].head(3)
```

[17]:

starttime	trip_id	usertype	gender	stoptime	tripduration	...
2015-09-06 07:52:00	7319012	Subscriber	Male	2015-09-06 07:55:00	207	...
2015-09-07 09:52:00	7341764	Subscriber	Female	2015-09-07 09:57:00	293	...
2015-09-15 08:25:00	7468970	Subscriber	Male	2015-09-15 08:33:00	473	...

`isnull` is an alias for `isna`

There is an identical method named `isnull` that you will see in other tutorials. It is an **alias** of `isna` meaning it does the exact same thing but has a different name. Either one is suitable to use, but I prefer `isna` because of the similarity to **NaN**, the representation of missing values. There are also other methods such as `dropna` and `fillna` that use the 'na' in their method names.

18.6 Exercises

Exercise 1

Select the wind speed column a a Series and assign it to a variable. Are there any negative wind speeds?

[ ]:

Exercise 2

Select all wind speed values between 12 and 16.

[ ]:

Exercise 3

Select the `events` and `gender` columns for all trip durations longer than 1,000 seconds.

[ ]:

Exercise 4

Read in the movie dataset with the title as the index. We will use this DataFrame for the rest of the problems. Select all the movies such that the Facebook likes for actor 2 are greater than those for actor 1.

[ ]:

Exercise 5

Select the year, content rating, and IMDB score columns for movies from the year 2016 with IMDB score less than 4.

[ ]:

Exercise 6

Select all the movies that are missing values for content rating.

[ ]:

Exercise 7

Select all the movies that are missing values for both the gross and budget columns. Return just those columns to verify that those values are indeed missing.

[ ]:

Exercise 8

Write a function `find_missing` that has three parameters, `df`, `col1` and `col2` where `df` is a DataFrame and `col1` and `col2` are column names. This function should return all the rows of the DataFrame where `col1` and `col2` are missing. Only return the two columns as well. Answer problem 7 with this function.

[ ]:

Chapter 19

Miscellaneous Subset Selection

In this chapter, a few more methods for subset selection are described. The methods used in this chapter do not add any additional functionality to pandas, but are covered for completeness.

Believe it or not, there are still a few more ways to select subsets of data. I personally do not use the methods described in this chapter as each one of them provides no more functionality over the previously covered methods. These methods are presented for completeness. They are all valid syntax and many pandas users do actually use them so you may find them valuable.

[1]:
```
import pandas as pd
bikes = pd.read_csv('../data/bikes.csv')
bikes.head(3)
```

[1]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...

19.1 The `query` method

The `query` method allows you to make boolean selections by writing the filter as a string. For instance, you would pass the string `'tripduration > 1000'` to select all rows of the `bikes` dataset that have a `tripduration` less than 1000. Let's see this command now.

[2]:
```
bikes.query('tripduration > 1000').head(3)
```

[2]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...
8	21028	Subscriber	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	...
10	24383	Subscriber	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523	...

Less syntax and more readable

The `query` method generally uses less syntax than boolean selection and is usually more readable. For instance, to reproduce the above with boolean selection in a single line would look like the following:

```
bikes[bikes['tripduration'] > 1000]
```

This looks a bit clumsy with the name `bikes` written twice right next to one another.

Use strings `and`, `or`, `not`

Unlike boolean selection, you can use the strings `and`, `or`, and `not` instead of the operators which further aides readability with `query`. Let's select `tripduration` greater than 1000 and `temperature` greater than 85.

[3]:
```
bikes.query('tripduration > 1000 and temperature > 85').head(3)
```

[3]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
40	61401	Subscriber	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	...
53	68864	Subscriber	Male	2013-07-16 13:04:00	2013-07-16 13:28:00	1435	...
60	71812	Subscriber	Male	2013-07-17 10:23:00	2013-07-17 10:40:00	1024	...

Use the @ symbol to reference a variable name

By default, all words within the query string will attempt to reference the column name. You can, however, reference a variable name by preceding it with the `@` symbol. Let's assign the variable `x` to 5000 and reference it in a query.

[4]:
```
x = 5000
bikes.query('tripduration > @x').head(3)
```

[4]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
18	40924	Subscriber	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	...
40	61401	Subscriber	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	...
77	87005	Subscriber	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	...

Reference strings with quotation marks

If you would like to reference a literal string within a query, you need to wrap it in quotes, or else pandas will attempt to use it as a column name. Let's select all 'Female' riders.

[5]:
```
bikes.query('gender == "Female"').head(3)
```

[5]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
9	23558	Subscriber	Female	2013-07-04 15:00:00	2013-07-04 15:16:00	922	...
14	31121	Subscriber	Female	2013-07-06 12:39:00	2013-07-06 12:49:00	610	...
20	42488	Subscriber	Female	2013-07-09 17:39:00	2013-07-09 17:55:00	943	...

Use 'in' for multiple equalities

You can query for multiple equalities with the word 'in' within your query like this:

[6]:
```
bikes.query('events in ["snow", "rain"]').head(3)
```

[6]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
45	66336	Subscriber	Male	2013-07-15 16:43:00	2013-07-15 16:55:00	727	...
112	111568	Subscriber	Male	2013-07-26 19:10:00	2013-07-26 19:33:00	1395	...
124	130156	Subscriber	Male	2013-07-30 18:53:00	2013-07-30 19:00:00	442	...

There are multiple syntaxes for the above that all work the same.

- `bikes.query('["snow", "rain"] in events')`
- `bikes.query('["snow", "rain"] == events')`
- `bikes.query('events == ["snow", "rain"]')`

Use 'not in' to invert the condition

You can invert the result of an 'in' clause by placing the word 'not' before it.

[7]:
```
bikes.query('events not in ["cloudy", "partlycloudy", "mostlycloudy"]').head(3)
```

[7]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
25	47798	Subscriber	Female	2013-07-11 08:17:00	2013-07-11 08:31:00	830	...
26	51130	Subscriber	Male	2013-07-12 01:07:00	2013-07-12 01:24:00	1043	...
33	53963	Subscriber	Male	2013-07-12 17:22:00	2013-07-12 17:34:00	730	...

Using the index with `query`

You can even use the word `index` to make comparisons against the index as if it were a normal column. Here, we select only the `events` that were 'cloudy' for an index value greater than 4000.

[8]:
```
bikes.query('index > 4000 and events =="cloudy" ').head(3)
```

[8]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
4007	2003400	Subscriber	Male	2014-06-07 14:07:00	2014-06-07 14:31:00	1434	...
4008	2004978	Subscriber	Male	2014-06-07 14:58:00	2014-06-07 15:19:00	1258	...
4009	2005778	Subscriber	Male	2014-06-07 15:23:00	2014-06-07 15:28:00	297	...

Use multiple comparison operators in a row

You can test that a column is contained between two values without using 'and'. Place the column name between the two less than (or greater than) signs as is done below.

[9]:
```
bikes.query('5000 < tripduration < 6000').head(3)
```

[9]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
18	40924	Subscriber	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	...
3406	1715861	Subscriber	Male	2014-05-19 09:26:00	2014-05-19 10:50:00	5059	...
4046	2024090	Subscriber	Male	2014-06-08 17:31:00	2014-06-08 18:54:00	5020	...

Why I avoid query

The `query` method appears to provide a more readable approach to filtering our data based on the values, but it currently lacks the ability to reference column names with spaces. For instance, if we had a column name of 'trip duration', then we would have no way to reference it with `query`.

Using boolean selection as shown in previous chapters works for every situation, so I only use it. There has been some discussion amongst the pandas developers to add this feature of selecting column names with spaces in the library, but it has yet to be built.

19.2 Slicing with just the brackets

So far, we have covered three ways to select subsets of data with just the brackets. With a single string, a list of strings, and a boolean Series. Let's quickly review those ways right now.

A single string

[10]:
```
bikes['tripduration'].head(3)
```

[10]:

0	993
1	623
2	1040

A list of strings

[11]:
```
cols = ['trip_id', 'tripduration']
bikes[cols].head(3)
```

[11]:

	trip_id	tripduration
0	7147	993
1	7524	623
2	10927	1040

A boolean Series

The previous two examples selected columns. Boolean Series select rows.

[12]:
```
filt = bikes['tripduration'] > 5000
bikes[filt].head(3)
```

[12]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
18	40924	Subscriber	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	...
40	61401	Subscriber	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	...
77	87005	Subscriber	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	...

Using a slice

It is possible to use slice notation within just the brackets. For example, the following selects the rows beginning at location 2 up to location 10 with a step size of 4. You can even use slice notation when the index is strings.

[13]:
```
bikes[2:10:4]
```

[13]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...
6	18880	Subscriber	Male	2013-07-02 17:47:00	2013-07-02 17:56:00	565	...

I do not recommend using slicing with *just the brackets*

Although slicing with *just the brackets* seems simple, I do not recommend using it. This is because it is ambiguous and can make selections either by integer location or by label. I always prefer explicit, unambiguous methods. Both `loc` and `iloc` are unambiguous and explicit. Meaning that even without knowing anything about the DataFrame, you would be able to explain exactly how the selection will take place.

If you do want to slice the rows, then use `loc` if you are using labels and `iloc` if you are using integer location, but do not use *just the brackets*.

19.3 Selecting a single cell with `at` and `iat`

pandas provides two more rarely seen indexers, `at`, and `iat`. These indexers are analogous to `loc` and `iloc` respectively, but only select a single cell of a DataFrame. Since they only select a single cell, you must pass both a row and column selection as either a label (`loc`) or an integer location (`iloc`). Let's see an example of each.

[14]:
```
bikes.at[40, 'temperature']
```

[14]:
```
87.1
```

[15]:
```
bikes.iat[-30, 5]
```

[15]:
```
389
```

The current index labels for `bikes` is integers which is why the number 40 was used above. It is the label for a row, but also happens to be an integer.

What's the purpose of these indexers?

All usages of `at` and `iat` may be replaced with `loc` and `iloc` in your data analysis and the code would produce the exact same results. Let's verify this below.

[16]:
```
bikes.loc[40, 'temperature']
```

[16]:
```
87.1
```

[17]:
```
bikes.iloc[-30, 5]
```

[17]:
```
389
```

These `at` and `iat` indexers are optimized to select a single cell of data and therefore provide slightly better performance than `loc` or `iloc`.

I never use these indexers

Personally, I never use these specialty indexers as the performance advantage for a single selection is minor. It would require a case where single element selections were happening in great numbers to see any significant improvement and doing so is rare in data analysis.

Much bigger performance improvement using numpy directly

If you truly wanted a large performance improvement for single-cell selection, you would select directly from numpy arrays and not a pandas DataFrame. Below, the data is extracted into the underlying numpy array with the `values` attribute. We then time the performance for selecting with numpy and also with `iat` and `iloc` on a DataFrame. On my machine, `iat` shows a 30-40% improvement over `iloc`, but selecting with numpy is about 50x as fast. There is no comparison here, if you care about performance for selecting a single cell of data, use numpy.

[18]:
```
values = bikes.values
```

[19]:
```
%timeit -n 5 values[-30, 5]
```

```
183 ns ś 127 ns per loop (mean ś std. dev. of 7 runs, 5 loops each)
```

[20]:
```
%timeit -n 5 bikes.iat[-30, 5]
```

```
7.39 ţs ś 3.64 ţs per loop (mean ś std. dev. of 7 runs, 5 loops each)
```

[21]:
```
%timeit -n 5 bikes.iloc[-30, 5]
```

```
10.6 ţs ś 3.22 ţs per loop (mean ś std. dev. of 7 runs, 5 loops each)
```

19.4 Exercises

Exercise 1

Use the `query` method to select trip durations between 5000 and 10000 when it was partlycloudly or mostlycloudly. Create a set to contain the possible events and assign it to a variable. Reference this variable within the query string. Then, redo the operation again using boolean selection.

[ ]:
```

```

Chapter 20

Solutions

20.1 1. Selecting Subsets of Data from DataFrames with just the brackets

[1]:
```
import pandas as pd
movie = pd.read_csv('../data/movie.csv', index_col='title')
```

Exercise 1

Select the column with the director's name as a Series

[2]:
```
movie['director_name'].head(3)
```

[2]:

title	
Avatar	James Cameron
Pirates of the Caribbean: A...	Gore Verbinski
Spectre	Sam Mendes

Exercise 2

Select the column with the director's name and number of Facebook likes.

[3]:
```
movie[['director_name', 'director_fb']].head(3)
```

[3]:

title	director_name	director_fb
Avatar	James Cameron	0.0
Pirates of the Caribbean: A...	Gore Verbinski	563.0
Spectre	Sam Mendes	0.0

Exercise 3

Select a single column as a DataFrame and not a Series

[4]:
```
# make a one item list
col = ['director_name']
movie[col].head(3)
```

[4]:

title	director_name
Avatar	James Cameron
Pirates of the Caribbean: A...	Gore Verbinski
Spectre	Sam Mendes

20.2 2. Selecting Subsets of Data from DataFrames with `loc`

Exercise 1

Read in the movie dataset and set the title column as the index. Select all columns for the movie 'The Dark Knight Rises'.

[5]:
```
movie = pd.read_csv('../data/movie.csv', index_col='title')
movie.head(3)
```

[5]:

title	year	color	content_rating	duration	director_name	director_fb	...
Avatar	2009.0	Color	PG-13	178.0	James Cameron	0.0	...
Pirates of the Caribbean: A...	2007.0	Color	PG-13	169.0	Gore Verbinski	563.0	...
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	0.0	...

[6]:
```
movie.loc['The Dark Knight Rises']
```

[6]:

year	2012
color	Color
content_rating	PG-13
duration	164
director_name	Christopher Nolan
director_fb	22000
actor1	Tom Hardy
actor1_fb	27000
actor2	Christian Bale
actor2_fb	23000
actor3	Joseph Gordon-Levitt
actor3_fb	23000
gross	4.48131e+08
genres	Action | Thriller
num_reviews	813
num_voted_users	1144337
plot_keywords	deception | imprisonment | law...
language	English
country	USA
budget	2.5e+08
imdb_score	8.5

Exercise 2

Select all columns for the movies 'Tangled' and 'Avatar'.

[7]:
```
rows = ['Tangled', 'Avatar']
movie.loc[rows]
```

[7]:

title	year	color	content_rating	duration	director_name	director_fb	actor1	actor1_fb	...
Tangled	2010.0	Color	PG	100.0	Nathan Greno	15.0	Brad Garrett	799.0	...
Avatar	2009.0	Color	PG-13	178.0	James Cameron	0.0	CCH Pounder	1000.0	...

Alternatively, use a colon.

[8]:
```
# movie.loc[rows, :]
```

Exercise 3

What year was 'Tangled' and 'Avatar' made and what was their IMBD scores?

[9]:
```
movie.loc[['Tangled', 'Avatar'], ['year', 'imdb_score']]
```

[9]:

title	year	imdb_score
Tangled	2010.0	7.8
Avatar	2009.0	7.9

Exercise 4

Can you tell what the data type of the year column is by just looking at its values?

Yes, because it has a decimal value it must be a float. Integers do not have decimals

Exercise 5

Use a single method to output the data type and number of non-missing values of year. Is it missing any?

[10]:
```
# yes, its missing many values. 4432 non-missing vs 4916 total
movie.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 4916 entries, Avatar to My Date with Drew
Data columns (total 21 columns):
year               4810 non-null float64
color              4897 non-null object
content_rating     4616 non-null object
duration           4901 non-null float64
director_name      4814 non-null object
director_fb        4814 non-null float64
actor1             4909 non-null object
actor1_fb          4909 non-null float64
actor2             4903 non-null object
actor2_fb          4903 non-null float64
actor3             4893 non-null object
actor3_fb          4893 non-null float64
gross              4054 non-null float64
genres             4916 non-null object
num_reviews        4867 non-null float64
num_voted_users    4916 non-null int64
plot_keywords      4764 non-null object
language           4904 non-null object
country            4911 non-null object
budget             4432 non-null float64
imdb_score         4916 non-null float64
dtypes: float64(10), int64(1), object(10)
memory usage: 1004.9+ KB
```

Exercise 6

Select every 300th movie between 'Tangled' and 'Forrest Gump'. Why doesn't 'Forrest Gump' appear in the results?

[11]:
```
# Forrest Gump is not a multiple of 100 away from Tangled
movie.loc['Tangled':'Forrest Gump':300]
```

[11]:

title	year	color	content_rating	duration	director_name	director_fb	actor1	...
Tangled	2010.0	Color	PG	100.0	Nathan Greno	15.0	Brad Garrett	...
Cloud Atlas	2012.0	Color	R	172.0	Tom Tykwer	670.0	Tom Hanks	...
Doom	2005.0	Color	R	113.0	Andrzej Bartkowiak	43.0	Dwayne Johnson	...

20.3 3. Selecting Subsets of Data from DataFrames with `iloc`

Exercise 1

Select the rows with integer location 10, 5, and 1

[12]:
```
movie.iloc[[10, 5, 1]]
```

[12]:

title	year	color	content_rating	duration	director_name	director_fb	...
Batman v Superman: Dawn of ...	2016.0	Color	PG-13	183.0	Zack Snyder	0.0	...
John Carter	2012.0	Color	PG-13	132.0	Andrew Stanton	475.0	...
Pirates of the Caribbean: A...	2007.0	Color	PG-13	169.0	Gore Verbinski	563.0	...

Exercise 2

Select the columns with integer location 10, 5, and 1

[13]:
```
movie.iloc[:, [10, 5, 1]].head(3)
```

[13]:

title	actor3	director_fb	color
Avatar	Wes Studi	0.0	Color
Pirates of the Caribbean: A...	Jack Davenport	563.0	Color
Spectre	Stephanie Sigman	0.0	Color

Exercise 3

Select rows with integer location 100 to 104 along with the column integer location 5.

[14]:
```
movie.iloc[100:105, 5]
```

[14]:

title	357.0
The Fast and the Furious	21000.0
The Curious Case of Benjami...	905.0
X-Men: First Class	508.0
The Hunger Games: Mockingja...	226.0

20.4 4. Selecting Subsets of Data from a Series

Exercise 1

Read in the bikes dataset. We will be using it for the next few questions. Select the wind speed column as a Series and assign it to a variable and output the head. What kind of index does this Series have?

[15]:
```
bikes = pd.read_csv('../data/bikes.csv')
bikes.head(3)
```

[15]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...

[16]:
```
wind = bikes['wind_speed']
wind.head(3)
```

[16]:

0	12.7
1	6.9
2	16.1

This index is a **RangeIndex**

[17]:
```
wind.index
```

[17]:
```
RangeIndex(start=0, stop=50089, step=1)
```

[18]:
```
wind.loc[4:10]
```

[18]:

4	17.3
5	17.3
6	15.0
7	5.8
8	0.0
9	12.7
10	9.2

Exercise 2

From the wind speed Series, select the integer locations 4 through, but not including 10

[19]:
```
wind.iloc[4:10]
```

[19]:

4	17.3
5	17.3
6	15.0
7	5.8
8	0.0
9	12.7

Exercise 3

Copy and paste your answer to Exercise 2 below but use `loc` instead. Do you get the same result? Why not?

[20]:
```
wind.loc[4:10]
```

[20]:

4	17.3
5	17.3
6	15.0
7	5.8
8	0.0
9	12.7
10	9.2

This is tricky - the index in this case contains integers and not strings. So the labels themselves are also integers and happen to be the same integers corresponding to integer location. The reason `.iloc` and `.loc` produce different results is that `.loc` always includes the last value when slicing.

Exercise 4

Read in the movie dataset and set the index to be the title. Select `actor1` as a Series. Who is the `actor1` for 'My Big Fat Greek Wedding'?

[21]:
```
movie = pd.read_csv('../data/movie.csv', index_col='title')
actor1 = movie['actor1']
```

[22]:
```
actor1.loc['My Big Fat Greek Wedding']
```

[22]:
```
'Nia Vardalos'
```

Exercise 5

Find `actor1` for your favorite two movies?

[23]:
```
actor1.loc[['Titanic', 'Blood Diamond']]
```

[23]:

title	Leonardo DiCaprio
Titanic	Leonardo DiCaprio

Exercise 6

Select the last 3 values from `actor1` using two different ways?

[24]:
```
actor1.iloc[-3:]
```

[24]:

title	Eva Boehnke
A Plague So Pleasant	Alan Ruck
Shanghai Calling	John August

[25]:
```
actor1.tail(3)
```

[25]:

title	Eva Boehnke
A Plague So Pleasant	Alan Ruck
Shanghai Calling	John August

20.5 5. Boolean Selection Single Conditions

[26]:
```
import pandas as pd
movie = pd.read_csv('../data/movie.csv', index_col='title')
movie.head(3)
```

[26]:

title	year	color	content_rating	duration	director_name	director_fb	...
Avatar	2009.0	Color	PG-13	178.0	James Cameron	0.0	...
Pirates of the Caribbean: A...	2007.0	Color	PG-13	169.0	Gore Verbinski	563.0	...
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	0.0	...

Exercise 1

Read in the movie dataset and set the index to be the title. Select all movies that have Tom Hanks as `actor1`. How many of these movies has he starred in?

[27]:
```
filt = movie['actor1'] == 'Tom Hanks'
hanks_movies = movie[filt]
```

```
hanks_movies.head(3)
```

[27]:

title	year	color	content_rating	duration	director_name	director_fb	actor1	...
Toy Story 3	2010.0	Color	G	103.0	Lee Unkrich	125.0	Tom Hanks	...
The Polar Express	2004.0	Color	G	100.0	Robert Zemeckis	0.0	Tom Hanks	...
Angels & Demons	2009.0	Color	PG-13	146.0	Ron Howard	2000.0	Tom Hanks	...

He's starred in 24 movies

[28]:
```
hanks_movies.shape
```

[28]:
```
(24, 21)
```

Exercise 2

Select movies with an IMDB score greater than 9.

[29]:
```
filt = movie['imdb_score'] > 9
movie[filt]
```

[29]:

title	year	color	content_rating	duration	director_name	director_fb	...
The Shawshank Redemption	1994.0	Color	R	142.0	Frank Darabont	0.0	...
Towering Inferno	NaN	Color	NaN	65.0	John Blanchard	0.0	...
Dekalog	NaN	Color	TV-MA	55.0	NaN	NaN	...
The Godfather	1972.0	Color	R	175.0	Francis Ford Coppola	0.0	...
Kickboxer: Vengeance	2016.0	NaN	NaN	90.0	John Stockwell	134.0	...

Exercise 3

Write a function that accepts a single parameter to find the number of movies for a given content rating. Use the function to find the number of movies for ratings 'R', 'PG-13', and 'PG'.

[30]:
```
def count_rating(rating):
    filt = movie['content_rating'] == rating
    count = len(movie[filt])
    return f'There are {count} movies rated {rating}'
```

[31]:
```
count_rating('R')
```

[31]:
```
'There are 2067 movies rated R'
```

[32]:
```
count_rating('PG-13')
```

[32]:
```
'There are 1411 movies rated PG-13'
```

[33]:
```
count_rating('PG')
```

[33]:
```
'There are 686 movies rated PG'
```

20.6 6. Boolean Selection Multiple Conditions

Exercise 1

Select all movies from the 1970s.

[34]:
```
filt1 = movie['year'] >= 1970
filt2 = movie['year'] <= 1979
filt = filt1 & filt2
movie[filt].head(3)
```

[34]:

title	year	color	content_rating	duration	director_name	director_fb	...
All That Jazz	1979.0	Color	R	123.0	Bob Fosse	189.0	...
Superman	1978.0	Color	PG	188.0	Richard Donner	503.0	...
Solaris	1972.0	Black and White	PG	115.0	Andrei Tarkovsky	0.0	...

Exercise 2

Select all movies from the 1970s that had IMDB scores greater than 8.

[35]:
```
filt1 = movie['year'] >= 1970
filt2 = movie['year'] <= 1979
filt3 = movie['imdb_score'] > 8

filt = filt1 & filt2 & filt3
movie[filt].head(3)
```

[35]:

title	year	color	content_rating	duration	director_name	director_fb	...
Solaris	1972.0	Black and White	PG	115.0	Andrei Tarkovsky	0.0	...
Apocalypse Now	1979.0	Color	R	289.0	Francis Ford Coppola	0.0	...
The Deer Hunter	1978.0	Color	R	183.0	Michael Cimino	517.0	...

Exercise 3

Select movies that were rated either R, PG-13, or PG.

[36]:
```
filt = movie['content_rating'].isin(['R', 'PG-13', 'PG'])
movie[filt].head(3)
```

[36]:

title	year	color	content_rating	duration	director_name	director_fb	...
Avatar	2009.0	Color	PG-13	178.0	James Cameron	0.0	...
Pirates of the Caribbean: A...	2007.0	Color	PG-13	169.0	Gore Verbinski	563.0	...
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	0.0	...

Exercise 4

Select movies that are either rated PG-13 or were made after 2010.

[37]:
```
filt1 = movie['content_rating'] == 'PG-13'
filt2 = movie['year'] > 2010
filt = filt1 | filt2

movie[filt].head(3)
```

[37]:

title	year	color	content_rating	duration	director_name	director_fb	...
Avatar	2009.0	Color	PG-13	178.0	James Cameron	0.0	...
Pirates of the Caribbean: A...	2007.0	Color	PG-13	169.0	Gore Verbinski	563.0	...
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	0.0	...

Exercise 5

Find all the movies that have at least one of the three actors with more than 10,000 Facebook likes.

[38]:
```
filt1 = movie['actor1_fb'] > 10000
filt2 = movie['actor2_fb'] > 10000
filt3 = movie['actor3_fb'] > 10000
filt = filt1 | filt2 | filt3

movie[filt].head(3)
```

[38]:

title	year	color	content_rating	duration	director_name	director_fb	...
Pirates of the Caribbean: A...	2007.0	Color	PG-13	169.0	Gore Verbinski	563.0	...
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	0.0	...
The Dark Knight Rises	2012.0	Color	PG-13	164.0	Christopher Nolan	22000.0	...

Exercise 6

Reverse the condition from Exercise 4. In words, what have you selected?

The following selects non-PG-13 movies made in the year 2010 or before.

[39]:
```
filt1 = movie['content_rating'] == 'PG-13'
filt2 = movie['year'] > 2010
filt = filt1 | filt2

movie[~filt].head(3)
```

[39]:

title	year	color	content_rating	duration	director_name	director_fb	...
Star Wars: Episode VII - Th...	NaN	NaN	NaN	NaN	Doug Walker	131.0	...
Tangled	2010.0	Color	PG	100.0	Nathan Greno	15.0	...
Harry Potter and the Half-B...	2009.0	Color	PG	153.0	David Yates	282.0	...

20.7 7. Boolean Selection More

[40]:
```
import pandas as pd
bikes = pd.read_csv('../data/bikes.csv')
```

Exercise 1

Select the wind speed column a a Series and assign it to a variable. Are there any negative wind speeds?

[41]:
```
wind = bikes['wind_speed']
wind.head(3)
```

[41]:

0	12.7
1	6.9
2	16.1

Yes, there is really strong negative wind! Or maybe its just bad data...

[42]:
```
filt = wind < 0
wind[filt].head(3)
```

[42]:

22990	-9999.0
27168	-9999.0
28368	-9999.0

Exercise 2

Select all wind speed values between 12 and 16.

[43]:
```
filt = wind.between(12, 16)
wind[filt].head(3)
```

[43]:

0	12.7
6	15.0
9	12.7

Exercise 3

Select the events and gender columns for all trip durations longer than 1,000 seconds.

[44]:
```
filt = bikes['tripduration'] > 1000
cols = ['events', 'gender']
bikes.loc[filt, cols].head(3)
```

[44]:

	events	gender
2	mostlycloudy	Male
8	cloudy	Male
10	mostlycloudy	Male

Exercise 4

Read in the movie dataset with the title as the index. We will use this DataFrame for the rest of the Exercises. Select all the movies such that the Facebook likes for actor 2 are greater than those for actor 1.

[45]:
```
movie = pd.read_csv('../data/movie.csv', index_col='title')
movie.head(3)
```

[45]:

title	year	color	content_rating	duration	director_name	director_fb	...
Avatar	2009.0	Color	PG-13	178.0	James Cameron	0.0	...
Pirates of the Caribbean: A...	2007.0	Color	PG-13	169.0	Gore Verbinski	563.0	...
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	0.0	...

There are none!

[46]:
```
filt = movie['actor2_fb'] > movie['actor2_fb']
movie[filt]
```

[46]:

Empty DataFrame Columns: Index(['year', 'color', 'content_rating', 'duration', 'director_name', 'director_fb', 'actor1', 'actor1_f

Exercise 5

Select the year, content rating, and IMDB score columns for movies from the year 2016 with IMDB score less than 4.

[47]:
```
filt1 = movie['year'] == 2016
filt2 = movie['imdb_score'] < 4
filt = filt1 & filt2
cols = ['year', 'content_rating', 'imdb_score']
```

```
movie.loc[filt, cols]
```

[47]:

title	year	content_rating	imdb_score
Fifty Shades of Black	2016.0	R	3.5
Cabin Fever	2016.0	Not Rated	3.7
God's Not Dead 2	2016.0	PG	3.4

Exercise 6

Select all the movies that are missing values for content rating.

[48]:
```
filt = movie['content_rating'].isna()
movie[filt].head(3)
```

[48]:

title	year	color	content_rating	duration	director_name	director_fb	...
Star Wars: Episode VII - Th...	NaN	NaN	NaN	NaN	Doug Walker	131.0	...
Godzilla Resurgence	2016.0	Color	NaN	120.0	Hideaki Anno	28.0	...
Harry Potter and the Deathl...	2011.0	Color	NaN	NaN	Matt Birch	0.0	...

Exercise 7

Select all the movies that are missing values for both the gross and budget columns. Return just those columns to verify that those values are indeed missing.

[49]:
```
filt = movie['gross'].isna() & movie['budget'].isna()
cols = ['gross', 'budget']
movie.loc[filt, cols].head(3)
```

[49]:

title	gross	budget
Star Wars: Episode VII - Th...	NaN	NaN
The Lovers	NaN	NaN
Godzilla Resurgence	NaN	NaN

Exercise 8

Write a function `find_missing` that has three parameters, `df`, `col1` and `col2` where `df` is a DataFrame and `col1` and `col2` are column names. This function should return all the rows of the DataFrame where `col1` and `col2` are missing. Only return the two columns as well. Answer Exercise 7 with this function.

[50]:
```
def find_missing(df, col1, col2):
    filt = df[col1].isna() & df[col2].isna()
    cols = [col1, col2]

    return df.loc[filt, cols]
```

[51]:
```
movie_missing = find_missing(movie, 'gross', 'budget')
movie_missing.head(3)
```

[51]:

title	gross	budget
Star Wars: Episode VII - Th...	NaN	NaN
The Lovers	NaN	NaN
Godzilla Resurgence	NaN	NaN

20.8 8. Miscellaneous Subset Selection

[52]:
```
import pandas as pd
bikes = pd.read_csv('../data/bikes.csv')
```

Exercise 1

Use the query method to select trip durations between 5000 and 10000 when it was partlycloudly or mostlycloudly. Create a set to contain the possible events and assign it to a variable. Reference this variable within the query string. Then redo the operation again using boolean selection.

[53]:
```
event_types = {"partlycloudy", "mostlycloudy"}
bikes.query('5000 <= tripduration <= 10000 and events in @event_types').head(3)
```

[53]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
40	61401	Subscriber	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	...
77	87005	Subscriber	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	...
3406	1715861	Subscriber	Male	2014-05-19 09:26:00	2014-05-19 10:50:00	5059	...

[54]:
```
filt1 = bikes['tripduration'].between(5000, 10000)
filt2 = bikes['events'].isin(event_types)
filt = filt1 & filt2
bikes[filt].head(3)
```

[54]:

	trip_id	usertype	gender	starttime	stoptime	tripduration	...
40	61401	Subscriber	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	...
77	87005	Subscriber	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	...
3406	1715861	Subscriber	Male	2014-05-19 09:26:00	2014-05-19 10:50:00	5059	...