2015 Computational Social Science Workshop

Day 1 - Introduction to `python` - Part 3 / 3

with **Jongbin Jung** (jongbin at stanford.edu)

- PhD student. Decision analysis, MS&E

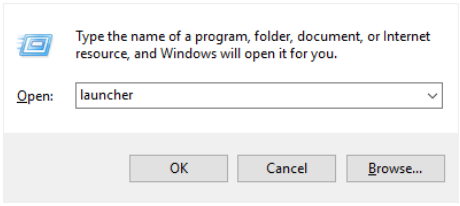
All material for days 1 (intro to `python`) and 2 (web scraping with `python`) publicly available at https://github.com/jongbinjung/css-python-workshop (https://github.com/jongbinjung/css-python-workshop)

3. Jupyter (aka iPython Notebook) and more ... - 1 of 2

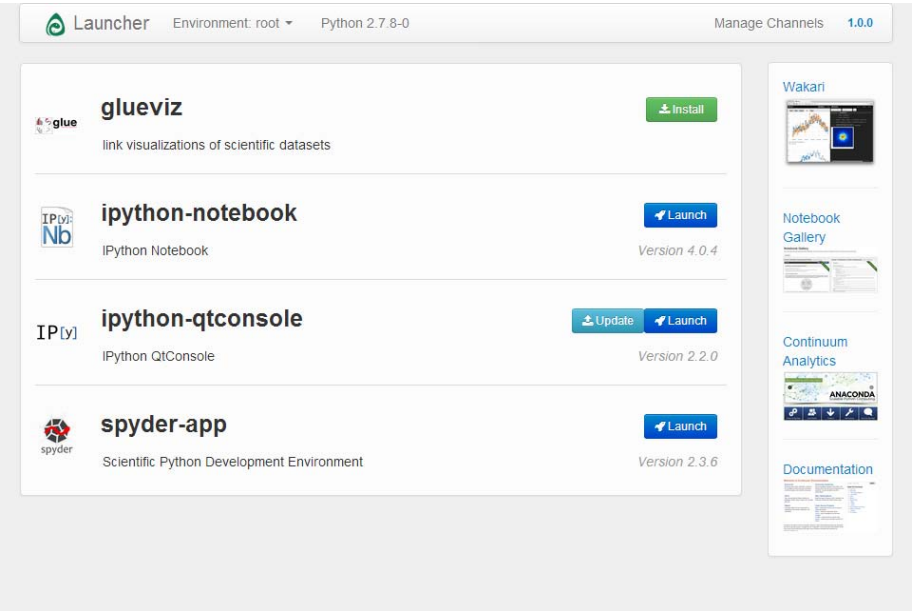
- web-based interface to `iPython` (an 'interactive' version of `python`)
- fully functional (anything you can do in `python`, you can do in an Jupyter)
- your entire project in the form of a blog post

Launching Jupyter

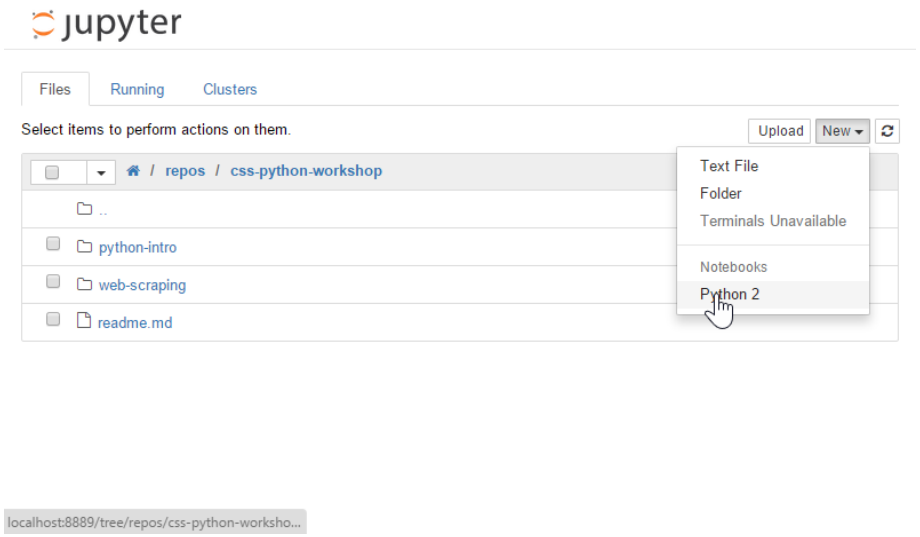
- launch the `launcher`
  - **Windows**
    - `Win + r` , type `launcher` and hit `OK` (or `Enter`)



  - **OS X / *nix**
    - Open a terminal
    - type `launcher` and hit `Enter`

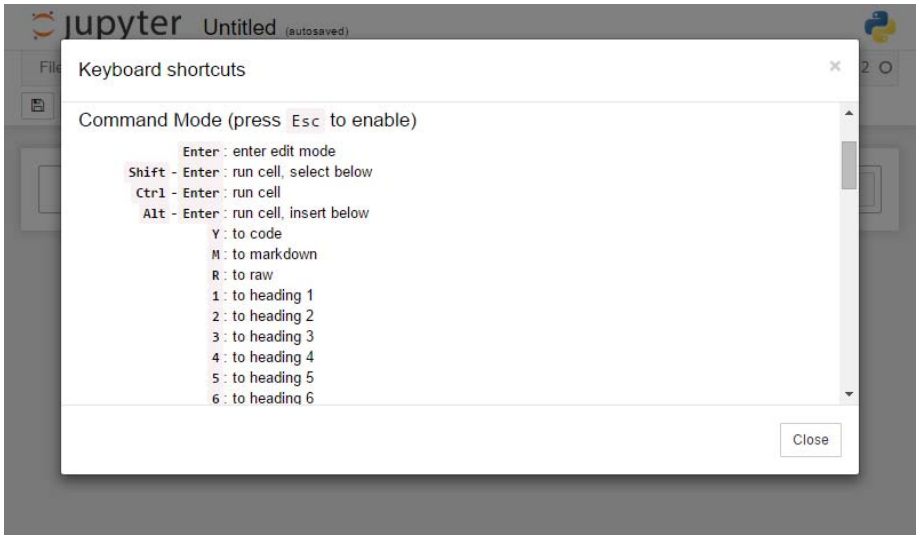


- hit `Launch` for `ipython-notebook`
- Jupyter will launch in your default browser, with the `$HOME` directory open
- you can browse the file system, create folders, and create notebooks
- create a new `python` notebook in the location of your choice, and open!



Basic stuff

- an iPython Notebook has two type of "cells"
  - `Code` cell: write and execute `python` code
  - `Markdown` cell: write notes (in `Markdown` with $\LaTeX$ support)
- the notes/slides for today's workshop are all iPython Notebooks!
- you can be in one of two *modes*
  - `command` mode: browse through the notebook (using `UP` and `DOWN` arrows), indicated by grey cell borders
  - `edit` mode: actually write stuff to the cell, indicated by green cell borders
- hit `Enter` to enable `edit` mode, and `ESC` to enable `command` mode
- in any mode, hit `Ctrl` + `Enter` to run/compile the selected cell
- checkout **Help > Keyboard Shortcuts** for more useful shortcuts



Let's make a `code` cell and write some `python` code!

```
In [1]: print 'This is python in a web browser!'.upper()
        THIS IS PYTHON IN A WEB BROWSER!
```

Even though Jupyter is running in your web browser, the `python` functionality is backed up by a full-blown `python` interpreter running in the background (aka the 'kernel'). This mean the `python` code you run on Jupyter is *the real thing*, not a limited look-alike!

You can write your notes, documentation, etc. in a Markdown cell. This cell uses the popular Markdown formatting and simple $\LaTeX$, backed by MathJax (https://www.mathjax.org/).

Some simple markdown syntax:

```
markdown
# this is a header (level 1)
## this is another header (level 2)
### keep adding hashes to make deeper headers ...

1. this is a numbered list
1. this will be number two (the '1.' just indicates it's part of a numbered list)

[links](some.link.address) are created like this! (the link address in parentheses wil
l be hidden)

- this is a bullet list
 - this is a nested bullet list
  - you can keep nesting lists with indents
   1. you can also nest numbered lists in bullet lists
     - and vice versa
```

The above `Markdown` code will render in a Markdown cell as:

this is a header (level 1)

this is another header (level 2)

keep adding hashes to make deeper headers ...

1. this is a numbered list
2. this will be number two (the '1.' just indicates it's part of a numbered list)

links (some.link.address) are created like this! (the link address in parentheses will be hidden)

- this is a bullet list
  - this is a nested bullet list
  - you can keep nesting lists with indents
    1. you can also nest numbered lists in bullet lists
       - and vice versa

$\LaTeX$ expressions can be inserted inline between single dollar signs ($...$), and in a separate block with double-dollar signs ($$...$$)

This should be enough to get you going with Jupyter, but I highly recommend that you spend some time in the `Help` menu to learn more about what Jupyter can do for you!

pandas

- `pandas` is a `python` library that helps you deal with data
- normally, you'd have to install `pandas`, but Anaconda (https://store.continuum.io/cshop/anaconda/) comes with `pandas` pre-installed, so all you have to do is `import` it in `python`

In [2]:
```
import pandas
```

Bring data into python with pandas

Let's go ahead and load some data. There are a series of `read_*` methods (e.g., `read_excel`) in `pandas`. Usually, you can start reading data by giving the `read_*` method an appropriate file path. For example, if there is a `csv` file named `data.csv` in the same directory as your notebook, you can read it like

```
panda.read_csv('./data.csv')
```

Today, we'll play with the wine quality dataset (http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv) from the UCI machine learning database. You can either download it to your machine from here (http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv), or use the URL (http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv (http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv)) directly in the `read_csv()` method to have `pandas` load it from the source.

In [3]:
```
data_src = 'http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv'
wine_data = pandas.read_csv(data_src)
wine_data.head()
```

Out[3]:

	fixed acidity;"volatile acidity";"citric acid";"residual sugar";"chlorides";"free sulfur dioxide";"total sulfur dioxide";"density";"pH";"sulphates";"alcohol";"quality"
0	7.4;0.7;0;1.9;0.076;11;34;0.9978;3.51;0.56;9.4;5
1	7.8;0.88;0;2.6;0.098;25;67;0.9968;3.2;0.68;9.8;5
2	7.8;0.76;0.04;2.3;0.092;15;54;0.997;3.26;0.65;...
3	11.2;0.28;0.56;1.9;0.075;17;60;0.998;3.16;0.58...
4	7.4;0.7;0;1.9;0.076;11;34;0.9978;3.51;0.56;9.4;5

The first line above saves the address to the data in a variable we name `data_src`. We can also write the link directly in the `read_csv` call, i.e.,

```
pandas.read_csv('http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv')
```

but saving the address in a separate string is generally good practice (and will make your life easier if you decide to change that address some day).

The second line loads the data into a variable we named `wine_data`. The third line tells `pandas` to print the first couple lines (`head`) of `wine_data`.

Notice from the output above that the data seems to be chunked into a single column. This is probably because we used `read_csv`, which expects the data to be separated by commas(,) - hence the name **c**omma **s**eparated **v**alues- but the data (if you look carefully) is actually separated by semi-colons(;). Instead of bloating the library with a `read_semi_colon_separated_value` method, the `read_csv` method in `pandas` let's you specify what the separating charater is, if not the expected comma. The syntax for that looks like

```
pandas.read_csv(path_to_data, sep=';')
```

(`read_csv` actually has a whole lot more options. Read the docs with `Shift` + `Tab` - a nice feature of Jupyter)

In [4]:
```
wine_data = pandas.read_csv(data_src, sep=';')
wine_data.head()
```

Out[4]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quali
0	7.4	0.70	0.00	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25	67	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15	54	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17	60	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5

That looks much better. Now that we've loaded a proper `pandas` `DataFrame` (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html#pandas.DataFrame) we can use methods provided with `pandas` to take a closer look at things. We've already seen `head()`, but we can also see `tail()`.

In [5]:
```
wine_data.tail()
```

Out[5]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	q
1594	6.2	0.600	0.08	2.0	0.090	32	44	0.99490	3.45	0.58	10.5	5
1595	5.9	0.550	0.10	2.2	0.062	39	51	0.99512	3.52	0.76	11.2	6
1596	6.3	0.510	0.13	2.3	0.076	29	40	0.99574	3.42	0.75	11.0	6
1597	5.9	0.645	0.12	2.0	0.075	32	44	0.99547	3.57	0.71	10.2	5
1598	6.0	0.310	0.47	3.6	0.067	18	42	0.99549	3.39	0.66	11.0	6

Get a quick summary with `describe()`

In [6]:
```
wine_data.describe()
```

Out[6]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide
count	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000
mean	8.319637	0.527821	0.270976	2.538806	0.087467	15.874922	46.467792
std	1.741096	0.179060	0.194801	1.409928	0.047065	10.460157	32.895324
min	4.600000	0.120000	0.000000	0.900000	0.012000	1.000000	6.000000
25%	7.100000	0.390000	0.090000	1.900000	0.070000	7.000000	22.000000
50%	7.900000	0.520000	0.260000	2.200000	0.079000	14.000000	38.000000
75%	9.200000	0.640000	0.420000	2.600000	0.090000	21.000000	62.000000
max	15.900000	1.580000	1.000000	15.500000	0.611000	72.000000	289.000000

Or take a look at the column names (more useful if you have a whole lot of columns):

In [7]:
```
cols = wine_data.columns
print cols
```

```
Index([u'fixed acidity', u'volatile acidity', u'citric acid', u'residual sugar',
u'chlorides', u'free sulfur dioxide', u'total sulfur dioxide', u'density', u'pH',
 u'sulphates', u'alcohol', u'quality'], dtype='object')
```

You probably notice that `columns` returned a list-like sequence. Hence, we can use the convenient `len()` function to count the number of columns in our data:

In [8]:
```
print len(cols)  # the length of column names, i.e., number of columns
```

```
12
```

Manipulating data

Most of your data manipulating needs can be categorized into the following five categories:

1. **select**: select a subset of *rows* or *columns* according to certain conditions
2. **transform**: add *new columns* to the data (usually as a function of existing columns)
3. **sort**: reorder the rows of the data in a specific order
4. **summarize**: aggregate values to representative *statistics* (sum, mean, median, etc.)

1. select

Use the `loc[row_index, col_index]` attribute to select subsets of data. The `row/column_index` can be

- a single label (e.g., 'quality')
- a `list` or `array` of labels (e.g., ['density', 'pH', 'quality']
- slice with **labels** (e.g., 'density':'quality', in this case the slice is *inclusive* on each end)
- array of boolean conditions

Put a single `:` as an index if you want all rows/columns. For example,

```
wine_data.loc[:, 'quality']
```

will return the entire 'quality' column.

In [9]:
```
wine_data.loc[0:5, 'quality']  # get rows 0:5 and column 'quality'
```

Out[9]:
```
0    5
1    5
2    5
3    6
4    5
5    5
Name: quality, dtype: int64
```

In [10]:
```
wine_data.loc[ 495:500 , ['density', 'quality', 'pH'] ]  # get rows 495:500 and the
 three specified columns
```

Out[10]:

	density	quality	pH
495	0.9972	8	3.15
496	0.9984	6	3.43
497	0.9966	5	3.32
498	0.9972	8	3.15
499	1.0002	6	3.48
500	0.9984	6	3.43

In [11]:
```
wine_data.loc[ 495:500, 'density':'quality' ]  # get rows 495:500 and all columns b
etween 'density' and 'quality' (inclusive)
```

Out[11]:

	density	pH	sulphates	alcohol	quality
495	0.9972	3.15	0.65	11.0	8
496	0.9984	3.43	0.65	9.0	6
497	0.9966	3.32	0.79	11.1	5
498	0.9972	3.15	0.65	11.0	8
499	1.0002	3.48	0.74	11.6	6
500	0.9984	3.43	0.65	9.0	6

In [12]:
```
wine_data.loc[ wine_data.loc[:, 'quality'] == 8, : ].head()  # rows with quality 8
```

Out[12]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	qu
267	7.9	0.35	0.46	3.6	0.078	15	37	0.9973	3.35	0.86	12.8	8
278	10.3	0.32	0.45	6.4	0.073	5	13	0.9976	3.23	0.82	12.6	8
390	5.6	0.85	0.05	1.4	0.045	12	88	0.9924	3.56	0.82	12.9	8
440	12.6	0.31	0.72	2.2	0.072	6	29	0.9987	2.88	0.82	9.8	8
455	11.3	0.62	0.67	5.2	0.086	6	19	0.9988	3.22	0.69	13.4	8

That last one might look tricky, but it's actually quite simple.

```
wine_data.loc[:, 'quality'] == 8
```

will give a boolean array that is `True` where the `quality` is 8, and `False` otherwise. Using this boolean array as the `row_index` of the outer `loc[]` we effectively select all rows where quality is 8.

Similar conditions can be built using other comparison operators such as `!=`, >=, <=.

Use `iloc[row_index, col_index]` if you want to specify the subset by integer position. In this case, slices are treated as regular `python` slices (start inclusive and end exclusive).

In [13]:
```
wine_data.iloc[ 0:5, 0:3 ]
```

Out[13]:

	fixed acidity	volatile acidity	citric acid
0	7.4	0.70	0.00
1	7.8	0.88	0.00
2	7.8	0.76	0.04
3	11.2	0.28	0.56
4	7.4	0.70	0.00

Using `iloc[]` with labels will result in a `ValueError`

In [14]:
```
wine_data.iloc[:, 'quality']
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-14-a393f445c44b> in <module>()
----> 1 wine_data.iloc[:, 'quality']

C:\Users\Jongbin\Anaconda\lib\site-packages\pandas\core\indexing.pyc in __getitem
__(self, key)
   1140     def __getitem__(self, key):
   1141         if type(key) is tuple:
-> 1142             return self._getitem_tuple(key)
   1143         else:
   1144             return self._getitem_axis(key, axis=0)

C:\Users\Jongbin\Anaconda\lib\site-packages\pandas\core\indexing.pyc in _getitem_
tuple(self, tup)
   1345     def _getitem_tuple(self, tup):
   1346
-> 1347         self._has_valid_tuple(tup)
   1348         try:
   1349             return self._getitem_lowerdim(tup)

C:\Users\Jongbin\Anaconda\lib\site-packages\pandas\core\indexing.pyc in _has_vali
d_tuple(self, key)
    130             if not self._has_valid_type(k, i):
    131                 raise ValueError("Location based indexing can only have [
%s] "
--> 132                                  "types" % self._valid_types)
    133
    134     def _is_nested_tuple_indexer(self, tup):

ValueError: Location based indexing can only have [integer, integer slice (START
point is INCLUDED, END point is EXCLUDED), listlike of integers, boolean array] t
ypes
```

2. transform

Setting a new column will automatically align the data by the indices.

In [15]:
```
import numpy as np
wine_data['new_column'] = np.nan
wine_data.head()
```

Out[15]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	qual
0	7.4	0.70	0.00	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25	67	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15	54	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17	60	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5

The nan object from the library numpy is used by pandas to specify missing values. Note that we've imported numpy using a slightly different syntax, namely

Note that by assigning a single np.nan value to a new column (named new_column), we've created a column full of NaNs. This illustrates the point, but isn't too useful. Let's remove this new column with the drop(label, axis) method. The axis argument in drop() specifies which axis (0 = row, 1 = column) to drop. In this case, we're dropping the entire column, so we specify axis=1.

In [16]:
```
wine_data = wine_data.drop('new_column', axis=1)
wine_data.head()
```

Out[16]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	qual
0	7.4	0.70	0.00	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25	67	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15	54	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17	60	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5

Let's try something more useful. We want to change the numeric 'quality' column into a string where values greater than 7 are labeled 'good', greater than 4 are labeled 'not bad', and less than or equal to 4 are labeled 'bad'.

To do that, we first define a function that will take an integer and return either 'good', 'not bad', or 'bad' depending on its value. Then, we apply this function to each element of the column 'quaity' using the apply() (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.apply.html) method, which is directly assigned to a new column in wine_data labelled 'quality_str'.

In [17]:
```
def quality_to_str(qual):
    if qual > 7:
        return 'good'
    elif qual > 4:
        return 'not bad'
    else:
        return 'bad'

wine_data['quality_str'] = wine_data.loc[:, 'quality'].apply(quality_to_str)
wine_data.head()
```

Out[17]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	qual
0	7.4	0.70	0.00	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25	67	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15	54	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17	60	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5

With this workflow, you can write functions to transform data in any way you need. This isn't necessarily convenient, but keep in mind that pandas is still under (relatively active) development, so we might get more functionality in this aspect, soon!

3. sort

This is easily achieved using the `sort()` method.

In [18]:
```
wine_data.sort(columns='quality').head()  # the default is to sort in ascending ord
er
```

Out[18]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	q
1478	7.1	0.875	0.05	5.7	0.082	3	14	0.99808	3.40	0.52	10.2	3
832	10.4	0.440	0.42	1.5	0.145	34	48	0.99832	3.38	0.86	9.9	3
899	8.3	1.020	0.02	3.4	0.084	6	11	0.99892	3.48	0.49	11.0	3
1374	6.8	0.815	0.00	1.2	0.267	16	29	0.99471	3.32	0.51	9.8	3
459	11.6	0.580	0.66	2.2	0.074	10	47	1.00080	3.25	0.57	9.0	3

In [19]:
```
wine_data.sort(columns='quality', ascending=False).head()  # specify ascending=Fals
e to sort in descending order
```

Out[19]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	q
495	10.7	0.35	0.53	2.6	0.070	5	16	0.99720	3.15	0.65	11.0	8
1403	7.2	0.33	0.33	1.7	0.061	3	13	0.99600	3.23	1.10	10.0	8
390	5.6	0.85	0.05	1.4	0.045	12	88	0.99240	3.56	0.82	12.9	8
1061	9.1	0.40	0.50	1.8	0.071	7	16	0.99462	3.21	0.69	12.5	8
1202	8.6	0.42	0.39	1.8	0.068	6	12	0.99516	3.35	0.69	11.7	8

In [20]:
```
wine_data.sort(columns='alcohol', ascending=False).iloc[0:2, :]  # chain with iloc
to get top-n entries
```

Out[20]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	qu
652	15.9	0.36	0.65	7.5	0.096	22	71	0.9976	2.98	0.84	14.9	5
588	5.0	0.42	0.24	2.0	0.060	19	50	0.9917	3.72	0.74	14.0	8

4. summarize

`pandas` has a few summary functions that will meet most of your needs:

```
In [21]: wine_data.sum()  # return the sum of each column
```

```
Out[21]: fixed acidity                                                 13303.1
         volatile acidity                                              843.985
         citric acid                                                    433.29
         residual sugar                                                4059.55
         chlorides                                                     139.859
         free sulfur dioxide                                             25384
         total sulfur dioxide                                            74302
         density                                                      1593.798
         pH                                                            5294.47
         sulphates                                                     1052.38
         alcohol                                                      16666.35
         quality                                                          9012
         quality_str            not badnot badnot badnot badnot badnot badnot ...
         dtype: object
```

```
In [22]: wine_data.mean()  # the mean of each column (where applicable)
```

```
Out[22]: fixed acidity           8.319637
         volatile acidity        0.527821
         citric acid             0.270976
         residual sugar          2.538806
         chlorides               0.087467
         free sulfur dioxide    15.874922
         total sulfur dioxide   46.467792
         density                 0.996747
         pH                      3.311113
         sulphates               0.658149
         alcohol                10.422983
         quality                 5.636023
         dtype: float64
```

```
In [23]: wine_data.median()  # median
```

```
Out[23]: fixed acidity           7.90000
         volatile acidity        0.52000
         citric acid             0.26000
         residual sugar          2.20000
         chlorides               0.07900
         free sulfur dioxide    14.00000
         total sulfur dioxide   38.00000
         density                 0.99675
         pH                      3.31000
         sulphates               0.62000
         alcohol                10.20000
         quality                 6.00000
         dtype: float64
```

Or as shown earlier, the `describe()` method gives you a quick summary of each column:

In [24]:
```
wine_data.describe()
```

Out[24]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide
count	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000
mean	8.319637	0.527821	0.270976	2.538806	0.087467	15.874922	46.467792
std	1.741096	0.179060	0.194801	1.409928	0.047065	10.460157	32.895324
min	4.600000	0.120000	0.000000	0.900000	0.012000	1.000000	6.000000
25%	7.100000	0.390000	0.090000	1.900000	0.070000	7.000000	22.000000
50%	7.900000	0.520000	0.260000	2.200000	0.079000	14.000000	38.000000
75%	9.200000	0.640000	0.420000	2.600000	0.090000	21.000000	62.000000
max	15.900000	1.580000	1.000000	15.500000	0.611000	72.000000	289.000000

Usually, summaries are more useful when combined with `groupby` operations. A `groupby` operation lets you group the data be its value in one or more column, then you can generate summaries per group:

In [25]:
```
wine_data.groupby('quality_str').mean()
```

Out[25]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH
quality_str									
bad	7.871429	0.724206	0.173651	2.684921	0.095730	12.063492	34.444444	0.996689	3.384
good	8.566667	0.423333	0.391111	2.577778	0.068444	13.277778	33.444444	0.995212	3.267
not bad	8.335310	0.520909	0.273590	2.532279	0.087349	16.063900	47.121212	0.996767	3.308

Recap - Data Manipulation

To reiterate, most of your data manipulation needs can be fulfilled with the following four functions

1. **select**: select a subset of *rows* or *columns* according to certain conditions
2. **transform**: add *new columns* to the data (usually as a function of existing columns)
3. **sort**: reorder the rows of the data in a specific order
4. **summarize**: aggregate values to representative *statistics* (sum, mean, median, etc.)

Exercise 5.

1. Add a new column to the `wine_data` labelled '`sulfur difference`', with values defined as `"total sulfur dioxide" - "free sulfur dioxide"`.
2. Find the top 10 entries with highest `sulfur difference`.
3. Find the mean `sulfur difference` for each (numeric) quality level