MATH40006: An Introduction To Computation
Module notes, Section 13

These notes, together with all the other resources you'll need, are available on Blackboard, at

`https://bb.imperial.ac.uk/`

---

13 Data analysis and pandas

A key use of computing in the modern age is **data analysis**: extracting information from data sets, creating models, etc. For example, this lies at the heart of the current excitement around **machine learning**.

You can do quite a lot of data analysis using just core Python plus NumPy, but Python does have specialised data analysis functionality. The most important data analysis module is called **pandas**.

The pandas module is built on two basic data structures: **Series** and **DataFrame**. (There's another, called **Panel**, but it's on the way out and we'll ignore it.) You can think of a DataFrame as being a bit like a spreadsheet: rows and columns of data. You can think of a Series as being a single column of a spreadsheet.

13.1 The Series structure

A pandas Series is in some respects a little like a dictionary: it consists of **index keys** and **values**. The first thing we notice, though, is that in Jupyter notebooks, a Series will automatically output, or print, in tabulated form. Here's an example:

```
import pandas as pd
platonic_s = pd.Series([4,6,8,12,20],
                       index=['tetrahedron',
                              'cube',
                              'octahedron',
                              'dodecahedron',
                              'icosahedron']
                      )
platonic_s
```

```
tetrahedron      4
cube             6
octahedron       8
dodecahedron    12
```

```
icosahedron     20
dtype: int64
```

An alternative way to set up the same series is using a dictionary:

```
platonic_dict = {'tetrahedron': 4,
                 'cube': 6,
                 'octahedron': 8,
                 'dodecahedron': 12,
                 'icosahedron': 20}

platonic_s2 = pd.Series(platonic_dict)

platonic_s2
```

```
cube             6
dodecahedron    12
icosahedron     20
octahedron       8
tetrahedron      4
dtype: int64
```

To sort our first series into index order, we can type

```
platonic_s = platonic_s.sort_index()

platonic_s
```

```
cube             6
dodecahedron    12
icosahedron     20
octahedron       8
tetrahedron      4
dtype: int64
```

To sort back into value order, we can type

```
platonic_s = platonic_s.sort_values()

platonic_s
```

```
tetrahedron      4
cube             6
octahedron       8
```

```
dodecahedron    12
icosahedron     20
dtype: int64
```

Something to notice straight away is that the `sort_index` and `sort_values` methods for Series aren't like Python's native `sort` method: they don't sort in place, as a side effect. Instead, they **return** the sorted Series, as a value. This is baked into the design of pandas; pretty much all pandas functions and methods work like that.

The equivalent of the `keys` and `values` methods for dictionaries aren't methods at all in the case of Series; they're implemented as what are called **attributes**. More about what precisely that means next week, but in the short term all you need to know is that there's no need for opening and closing parentheses:

```
print(platonic_s2.index)
print(platonic_s2.values)
```

```
Index(['tetrahedron', 'cube', 'octahedron', 'dodecahedron',
'icosahedron'], dtype='object')
[ 4  6  8 12 20]
```

Notice that the `index` attribute is returned as a special `Index` object, whereas the `values` attribute is returned as a NumPy array. Pandas is built on top of NumPy, and the two are intimately linked.

13.2 Mutability and homogeneity

So, pandas Series are a bit like dictionaries. But they're not similar in every respect. Dictionaries are extremely flexible things; Series are, by design, less so.

First, a similarity. I can, if I choose to be mathematically incorrect, change one of my values in `platonic_dict`:

```
platonic_dict['cube'] = 600

platonic_dict
```

```
{'tetrahedron': 4,
 'cube': 600,
 'octahedron': 8,
 'dodecahedron': 12,
 'icosahedron': 20}
```

I can do the same with a Series, and the syntax is exactly the same:

```
platonic_s2['cube'] = 600

platonic_s2
```

```
tetrahedron       4
cube            600
octahedron        8
dodecahedron     12
icosahedron      20
dtype: int64
```

Let's quickly change them back before anybody notices:

```
platonic_dict['cube'] = 6
platonic_s2['cube'] = 6
```

Now, a difference. Dictionaries support data of mixed type:

```
platonic_dict['cube'] = 6.0

platonic_dict
```

```
{'tetrahedron': 4,
 'cube': 6.0,
 'octahedron': 8,
 'dodecahedron': 12,
 'icosahedron': 20}
```

Series don't; the data must be **homogeneous**:

```
platonic_s2['cube'] = 6.0

platonic_s2
```

```
tetrahedron      4
cube             6
octahedron       8
dodecahedron    12
icosahedron     20
dtype: int64
```

Series can represent any kind of data (as long as it's all of the same type), but they were designed to represent **time series**: that is, data that represents the change of something over time. Here's some data representing closing prices of the New York stock exchange over a period of a week in 2019:

```
import numpy as np
dates = np.array(['2019-03-11', '2019-03-12', '2019-03-13',
       '2019-03-14',  '2019-03-15'])
closing_prices = np.array([20.889999, 20.370001, 20.1,
       19.950001, 19.690001])

prices_s = pd.Series(closing_prices, index=dates)
prices_s
```

```
2019-03-11    20.889999
2019-03-12    20.370001
2019-03-13    20.100000
2019-03-14    19.950001
2019-03-15    19.690001
dtype: float64
```

We can perform various types of analysis on this data–but this stuff really comes into its own when we talk about DataFrames, so let's defer that discussion till then. Instead, let's content ourselves with a line plot and a bar chart:

```
%matplotlib inline

prices_s.plot()
```

```
prices_s.plot.bar()
```

Shown in Figures 1 and 2 respectively.

One more thing to note: if we don't specify an index when setting up a series, the default integer indexing will be used:

```
closing_prices = np.array([20.889999, 20.370001, 20.1,
       19.950001, 19.690001])

prices_s2 = pd.Series(closing_prices)
prices_s2
```

```
0    20.889999
1    20.370001
2    20.100000
3    19.950001
4    19.690001
dtype: float64
```

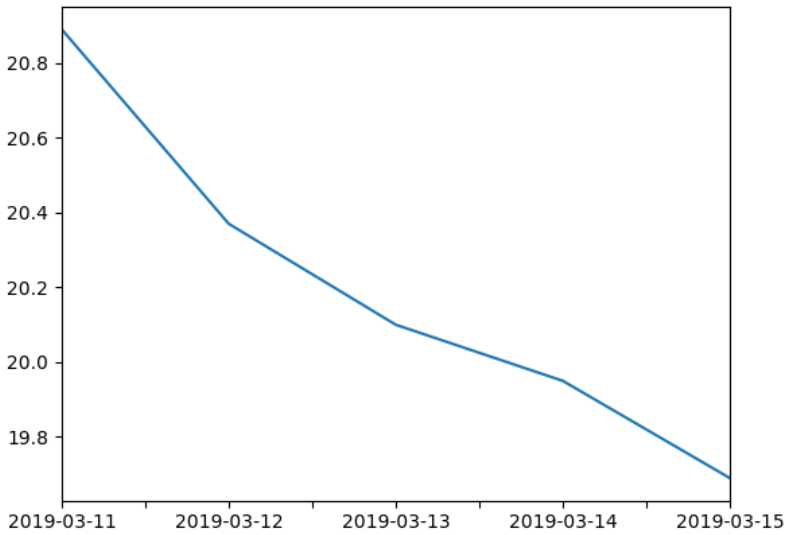


Figure 1: Line plot of closing stock prices

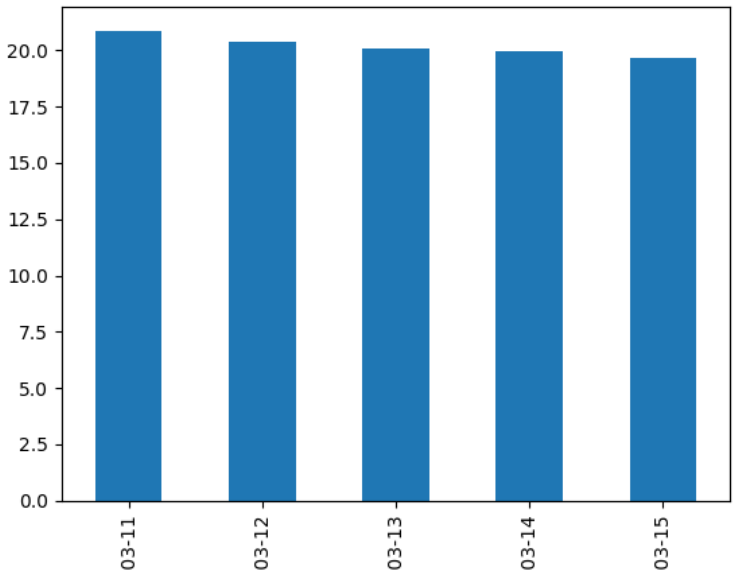


Figure 2: Bar chart of closing stock prices

13.3 DataFrames

A pandas DataFrame is a collection of pandas Series that share an index. You can think of a DataFrame as kind of like a spreadsheet, in which the index provides the row headings and the column headings specify the various series.

Notice that because each column of a DataFrame corresponds to a Series, each column must be homogeneous; however, the data type is perfectly free to vary from column to column.

Let's set one up. One way to do that is using a dictionary, in which the keys are the column headings, and the values come from lists, tuples, arrays or Series:

```
platonic_dict2 = {
    'name': ['tetrahedron',
             'cube',
             'octahedron',
             'dodecahedron',
             'icosahedron'],
    'faces': [4, 6, 8, 12, 20],
    'vertices': [4, 8, 6, 20, 12],
    'edges': [6, 12, 12, 30, 30],
    'face shape': ['triangle',
                   'square',
                   'triangle',
                   'pentagon',
                   'triangle']
}

platonic_df = pd.DataFrame(platonic_dict2)

platonic_df
```

	name	faces	vertices	edges	face shape
0	tetrahedron	4	4	6	triangle
1	cube	6	8	12	square
2	octahedron	8	6	12	triangle
3	dodecahedron	12	20	30	pentagon
4	icosahedron	20	12	30	triangle

Note that it's used the default indexing. Suppose we want to index by name instead:

```
platonic_df = platonic_df.set_index('name')

platonic_df
```

name	faces	vertices	edges	face shape
tetrahedron	4	4	6	triangle
cube	8	4	20	square
octahedron	8	6	12	triangle
dodecahedron	12	20	30	pentagon
icosahedron	20	12	30	triangle

We can get a particular column in the form of a Series:

```
platonic_df['faces']
```

```
name
tetrahedron      4
cube             6
octahedron       8
dodecahedron    12
icosahedron     20
Name: faces, dtype: int64
```

We can add a column to the DataFrame:

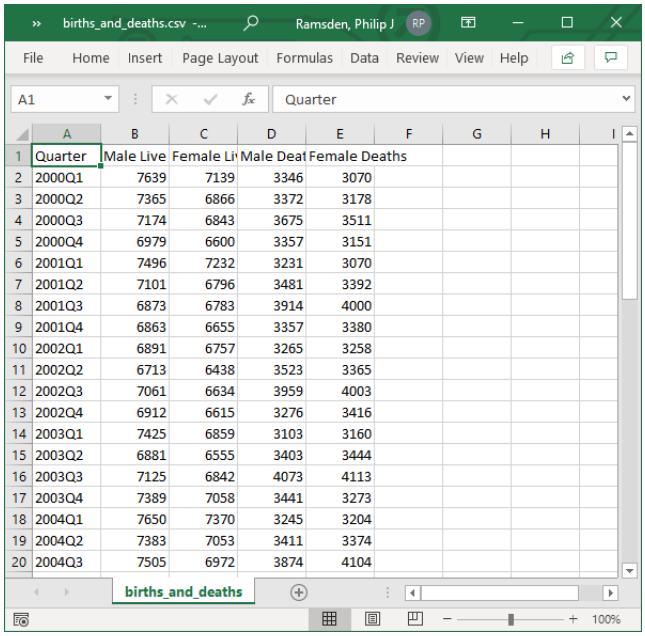
```
platonic_df['Euler check'] = platonic_df['faces'] + \
platonic_df['vertices'] - platonic_df['edges']

platonic_df
```

name	faces	vertices	edges	face shape	Euler check
tetrahedron	4	4	6	triangle	2
cube	8	4	20	square	2
octahedron	8	6	12	triangle	2
dodecahedron	12	20	30	pentagon	2
icosahedron	20	12	30	triangle	2

Notice that when we do calculations with them, Series objects behave just like NumPy arrays: we can add, subtract, multiply or divide them in a wholly vectorized way.

13.4 Reading from a file

It's actually fairly rare, however, that we'd want to set up a DataFrame ourselves from lists like this. More often, we'd want to read it in from an external file. Here's a partial printout



	A	B	C	D	E
1	Quarter	Male Live	Female Li	Male Deat	Female Deaths
2	2000Q1	7639	7139	3346	3070
3	2000Q2	7365	6866	3372	3178
4	2000Q3	7174	6843	3675	3511
5	2000Q4	6979	6600	3357	3151
6	2001Q1	7496	7232	3231	3070
7	2001Q2	7101	6796	3481	3392
8	2001Q3	6873	6783	3914	4000
9	2001Q4	6863	6655	3357	3380
10	2002Q1	6891	6757	3265	3258
11	2002Q2	6713	6438	3523	3365
12	2002Q3	7061	6634	3959	4003
13	2002Q4	6912	6615	3276	3416
14	2003Q1	7425	6859	3103	3160
15	2003Q2	6881	6555	3403	3444
16	2003Q3	7125	6842	4073	4113
17	2003Q4	7389	7058	3441	3273
18	2004Q1	7650	7370	3245	3204
19	2004Q2	7383	7053	3411	3374
20	2004Q3	7505	6972	3874	4104

Figure 3: A CSV file in Excel

of a file called `births_and_deaths.csv`, which is available on Blackboard; it shows births and deaths of males and females in a particular locality (one that seems a bit rigid about gender!)

```
Quarter,Male Live Births,Female Live Births,Male Deaths,Female Deaths
2000Q1,7639,7139,3346,3070
2000Q2,7365,6866,3372,3178
2000Q3,7174,6843,3675,3511
2000Q4,6979,6600,3357,3151
2001Q1,7496,7232,3231,3070
2001Q2,7101,6796,3481,3392
2001Q3,6873,6783,3914,4000
2001Q4,6863,6655,3357,3380
2002Q1,6891,6757,3265,3258
...
```

This is called **comma-separated values** format (or CSV), and it's a common convention for storing data in files.

Figure 3 shows what the same file looks like, displayed in Excel (which can read CSV).

And here's how we read it in as a DataFrame; this needs the file to be in the same folder as our notebook.

```
births_and_deaths_df = pd.read_csv('births_and_deaths.csv')
```

Let's display it; but not the whole file, which runs to 52 rows. Instead, we'll use a really handy pandas method called `head`, which just displays the first few:

```
births_and_deaths_df.head()
```

	Quarter	Male Live Births	Female Live Births	Male Deaths	Female Deaths
0	2000Q1	7639	7139	3346	3070
1	2000Q2	7365	6866	3372	3178
2	2000Q3	7174	6843	3675	3511
3	2000Q4	6979	6600	3357	3151
4	2001Q1	7496	7232	3231	3070

Again, we'd probably like to override the default indexing, and have the 'Quarter' values form the index. We can actually achieve that at the read-in stage:

```
births_and_deaths_df = pd.read_csv('births_and_deaths.csv',
                                   index_col = 'Quarter')

births_and_deaths_df.head()
```

Quarter	Male Live Births	Female Live Births	Male Deaths	Female Deaths
2000Q1	7639	7139	3346	3070
2000Q2	7365	6866	3372	3178
2000Q3	7174	6843	3675	3511
2000Q4	6979	6600	3357	3151
2001Q1	7496	7232	3231	3070

Finally, we can convert the index to the pandas DateTime format, which offers all sorts of benefits:

```
births_and_deaths_df.index = pd.to_datetime(births_and_deaths_df.index)

births_and_deaths_df.head()
```

Quarter	Male Live Births	Female Live Births	Male Deaths	Female Deaths
2000-01-01	7639	7139	3346	3070
2000-04-01	7365	6866	3372	3178
2000-07-01	7174	6843	3675	3511
2000-10-01	6979	6600	3357	3151
2001-01-01	7496	7232	3231	3070

13.5 Operations on DataFrames

There are now a whole host of things we can do with our data. Suppose we want to know the percentage changes quarter-on-quarter:

```
births_and_deaths_pc = births_and_deaths_df.pct_change()

births_and_deaths_pc.head()
```

Quarter	Male Live Births	Female Live Births	Male Deaths	Female Deaths
2000-01-01	NaN	NaN	NaN	NaN
2000-04-01	-0.035869	-0.038241	0.007770	0.035179
2000-07-01	-0.025933	-0.003350	0.089858	0.104783
2000-10-01	-0.027181	-0.035511	-0.086531	-0.102535
2001-01-01	0.074079	0.095758	-0.037534	-0.025706

Those NaNs are a bit annoying; let's "cleanse" the data by converting them to zeros:

```
births_and_deaths_pc = births_and_deaths_pc.fillna(0)

births_and_deaths_pc.head()
```

Quarter	Male Live Births	Female Live Births	Male Deaths	Female Deaths
2000-01-01	0.000000	0.000000	0.000000	0.000000
2000-04-01	-0.035869	-0.038241	0.007770	0.035179
2000-07-01	-0.025933	-0.003350	0.089858	0.104783
2000-10-01	-0.027181	-0.035511	-0.086531	-0.102535
2001-01-01	0.074079	0.095758	-0.037534	-0.025706

Let's create a DataFrame with running totals in all four columns:

```
births_and_deaths_tot = births_and_deaths_df.cumsum()

births_and_deaths_tot.head()
```

Quarter	Male Live Births	Female Live Births	Male Deaths	Female Deaths
2000-01-01	7639	7139	3346	3070
2000-04-01	15004	14005	6718	6248
2000-07-01	22178	20848	10393	9759
2000-10-01	29157	27448	13750	12910
2001-01-01	36653	34680	16981	15980

How about a table of four-quarter rolling averages?

```
births_and_deaths_rolling = births_and_deaths_df.rolling(4).mean()

births_and_deaths_rolling.head(8)
```

Quarter	Male Live Births	Female Live Births	Male Deaths	Female Deaths
2000-01-01	NaN	NaN	NaN	NaN
2000-04-01	NaN	NaN	NaN	NaN
2000-07-01	NaN	NaN	NaN	NaN
2000-10-01	7289.25	6862.00	3437.50	3227.50
2001-01-01	7253.50	6885.25	3408.75	3227.50
2001-04-01	7187.50	6867.75	3436.00	3281.00
2001-07-01	7112.25	6852.75	3495.75	3403.25
2001-10-01	7083.25	6866.50	3495.75	3460.50

Note the use of the optional argument to the `head` method, giving us some extra rows in the display.

And then a really useful feature, which we can only use because we converted the index to DateTime format. We can **resample** the DataFrame at a different frequency; let's say annually. We need to specify how to combine the data, and there are various ways to do this; in this case it makes sense to add the figures for all four quarters in each year. Here's how we do that:

```
births_and_deaths_resampled = births_and_deaths_df.resample('A').sum()

births_and_deaths_resampled.index = \
births_and_deaths_resampled.index.rename('Year')

births_and_deaths_resampled
```

Year	Male Live Births	Female Live Births	Male Deaths	Female Deaths
2000-12-31	29157	27448	13750	12910
2001-12-31	28333	27466	13983	13842
2002-12-31	27577	26444	14023	14042
2003-12-31	28820	27314	14020	13990
2004-12-31	29744	28329	14075	14344
2005-12-31	29546	28199	13431	13603
2006-12-31	30240	28953	13924	14321
2007-12-31	33013	31031	14275	14247
2008-12-31	33102	31241	14535	14653
2009-12-31	32112	30431	14480	14484
2010-12-31	32904	30993	14223	14215
2011-12-31	31476	29927	14823	15259
2012-12-31	31243	29935	15056	15043

The 'A' stands for 'annually'. If we'd wanted once every three years, we'd have used '3A'. Alternative **sampling rules** are things like '6M' (for six months), '2H' (for two hours), '20min', '10S' and so on.

Let's suppose we decide the term "Live" is unnecessary. How can we rename our columns? One way is this:

```
births_and_deaths_df = births_and_deaths_df.rename(
    columns = lambda str: str.replace('Live ', ''))

births_and_deaths_df.head()
```

Quarter	Male Births	Female Births	Male Deaths	Female Deaths
2000-01-01	7639	7139	3346	3070
2000-04-01	7365	6866	3372	3178
2000-07-01	7174	6843	3675	3511
2000-10-01	6979	6600	3357	3151
2001-01-01	7496	7232	3231	3070

Finally, let's produce an **aggregate table** showing totals and means:

```
births_and_deaths_agg = births_and_deaths_df.aggregate(['sum','mean'])

births_and_deaths_agg
```

	Male Births	Female Births	Male Deaths	Female Deaths
sum	397267.00	377711.000000	184598.000000	184953.000000
mean	7639.75	7263.673077	3549.961538	3556.788462

13.6 Plots

The pandas module supports a wide variety of specialist plotting tools. The general-purpose one is `plot`, which produces a line graph:

```
births_and_deaths_df.plot()
```

A bar chart for the first few rows of our DataFrame:

```
births_and_deaths_df.head().plot.bar()
```

A histogram for the 'Female Births' column:

```
births_and_deaths_df['Female Births'].plot.hist()
```

A histogram for the 'Female Births' and 'Male Births' columns:

```
births_and_deaths_df[['Female Births','Male Births']].plot.hist()
```

A scatter diagram for male and female births:

```
births_and_deaths_df.plot.scatter(x='Male Births',y='Female Births')
```

Box-and-whisker plots for male births and female births:

```
births_and_deaths_df[['Female Births','Male Births']].plot.box()
```

There are many others. The above are all shown in Figure 4.

13.7 Descriptive statistics

DataFrames support a wide variety of descriptive statistics, returned in the form of Series:

```
births_and_deaths_df.mean()
```

```
Male Births      7639.750000
Female Births    7263.673077
Male Deaths      3549.961538
Female Deaths    3556.788462
dtype: float64
```

```
births_and_deaths_df.median()
```

```
Male Births      7635.0
Female Births    7307.0
Male Deaths      3498.5
Female Deaths    3487.5
dtype: float64
```

```
births_and_deaths_df.std()
```

```
Male Births      506.576548
Female Births    445.757682
Male Deaths      272.253844
Female Deaths    329.419890
dtype: float64
```

```
births_and_deaths_df.quantile(0.75)
```

```
Male Births      8037.25
Female Births    7544.25
Male Deaths      3680.00
Female Deaths    3699.75
Name: 0.75, dtype: float64
```

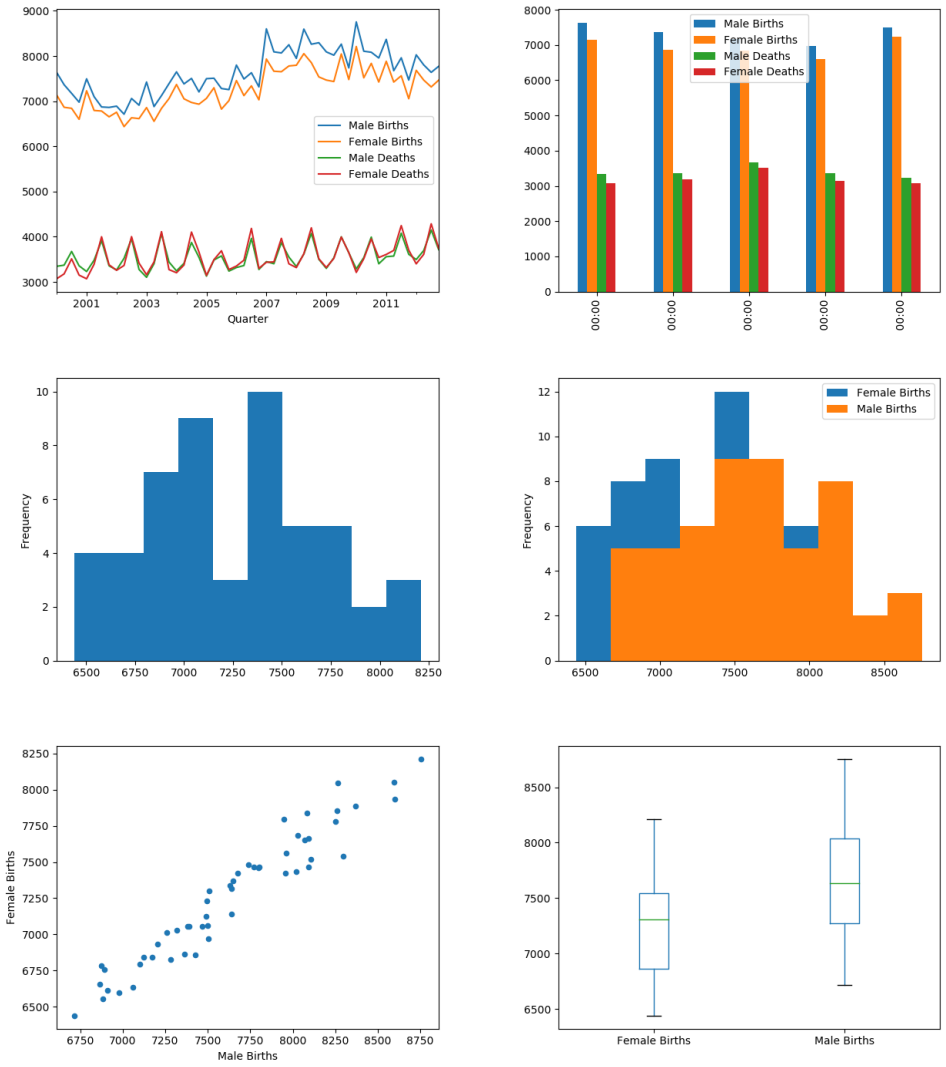


Figure 4: Plots of births and deaths data

```
births_and_deaths_df.quantile(0.75)-births_and_deaths_df.quantile(0.25)
```

```
Male Births      762.25
Female Births    680.00
Male Deaths      325.75
Female Deaths    380.50
dtype: float64
```

A DataFrame summarising a lot of this information is returned by the `describe` method:

```
births_and_deaths_df.describe()
```

	Male Births	Female Births	Male Deaths	Female Deaths
count	52.000000	52.000000	52.000000	52.000000
mean	7639.750000	7263.673077	3549.961538	3556.788462
std	506.576548	445.757682	272.253844	329.419890
min	6713.000000	6438.000000	3103.000000	3070.000000
25%	7275.000000	6864.250000	3354.250000	3319.250000
50%	7635.000000	7307.000000	3498.500000	3487.500000
75%	8037.250000	7544.250000	3680.000000	3699.750000
max	8756.000000	8212.000000	4149.000000	4287.000000

13.8 Queries

Let's load another DataFrame using `read_csv`; this one concerns sales of real estate in California. We'll look at several rows so you can get an idea of what the data shows; it's quite wide, so I've split it into two listings.

```
real_estate_df =
    pd.read_csv('real_estate.csv', index_col = 'sale_date')

real_estate_df.index = pd.to_datetime(real_estate_df.index)

real_estate_df.head(10)
```

sale_date	street	city	zip
2008-05-21	3526 HIGH ST	SACRAMENTO	95838
2008-05-21	51 OMAHA CT	SACRAMENTO	95823
2008-05-21	2796 BRANCH ST	SACRAMENTO	95815
2008-05-21	2805 JANETTE WAY	SACRAMENTO	95815
2008-05-21	6001 MCMAHON DR	SACRAMENTO	95824
2008-05-21	5828 PEPPERMILL CT	SACRAMENTO	95841
2008-05-21	6048 OGDEN NASH WAY	SACRAMENTO	95842
2008-05-21	2561 19TH AVE	SACRAMENTO	95820
2008-05-21	11150 TRINITY RIVER DR Unit 114	RANCHO CORDOVA	95670
2008-05-21	7325 10TH ST	RIO LINDA	95673

sale_date	beds	baths	sq__ft	type	price
2008-05-21	2	1	836	Residential	59222
2008-05-21	3	1	1167	Residential	68212
2008-05-21	2	1	796	Residential	68880
2008-05-21	2	1	852	Residential	69307
2008-05-21	2	1	797	Residential	81900
2008-05-21	3	1	1122	Condo	89921
2008-05-21	3	2	1104	Residential	90895
2008-05-21	3	1	1177	Residential	91002
2008-05-21	2	2	941	Condo	94905
2008-05-21	3	2	1146	Residential	98937

Let's start by finding all sales over $600000; we'll just show the city, square feet and price

```
real_estate_df.query('price > 600000')[['city', 'sq__ft', 'price']]
```

sale_date	city	sq__ft	price
2008-05-21	EL DORADO HILLS	0	606238
2008-05-21	SACRAMENTO	2325	660000
2008-05-21	EL DORADO HILLS	0	830000
2008-05-20	ROSEVILLE	3838	613401
2008-05-20	ROSEVILLE	0	614000
2008-05-20	FAIR OAKS	2846	680000
2008-05-20	SACRAMENTO	2484	699000
2008-05-20	LOOMIS	1624	839000
2008-05-19	FOLSOM	2660	636000
2008-05-19	CARMICHAEL	3357	668365
2008-05-19	GRANITE BAY	2896	676200
2008-05-19	PLACERVILLE	2025	677048
2008-05-19	WILTON	3788	691659
2008-05-19	GRANITE BAY	3670	760000
2008-05-16	ROSEVILLE	3579	610000
2008-05-16	EL DORADO HILLS	0	622500
2008-05-16	EL DORADO HILLS	0	680000
2008-05-16	EL DORADO HILLS	0	879000
2008-05-16	WILTON	4400	884790

This operation, **querying**, is one of the things we most often want to do with data.

13.9 Aggregating: pivot tables and `groupby`

The real estate data we have is arranged in what's called a **flat table**: there is a separate row for each individual case. There are a number of ways of creating DataFrames in which this data is **aggregated** (that is, combined) to give us tables that may be more informative to the reader. The easiest and most intuitive way to do that is probably the **pivot table**.

The basic idea of a pivot table is this. Let's say we've got a flat table with at least two columns, c_1 and c_2, and let's say c_1 contains a number of repeats: data items that occur more than once. In the case of our real estate data, c_1 could for example be the **city** column; we can see that SACRAMENTO occurs several times. Or perhaps it could be the **type** column, in which we can see multiple instances of Residential and Condo. The column c_2 would usually (though not always) contain numerical data, so **price** would be a good candidate.

A basic pivot table is a summary of the data in column c_2, broken down according to the categories in column c_1. At its simplest, this looks like this:

```
real_estate_pt0 = real_estate_df.pivot_table(
    index='city',
```

```
    values='price'
)

real_estate_pt0.head(10)
```

city	price
ANTELOPE	232496.393939
AUBURN	405890.800000
CAMERON PARK	267944.444444
CARMICHAEL	295684.750000
CITRUS HEIGHTS	187114.914286
COOL	300000.000000
DIAMOND SPRINGS	216033.000000
EL DORADO	247000.000000
EL DORADO HILLS	491698.956522
ELK GROVE	271157.692982

In this table, the data in the **price** column has been **aggregated**: the price figure for **ANTELOPE** is a combination of all the separate prices for every row in the original DataFrame that had **ANTELOPE** in the **city** column. What kind of combination? Well, the default behaviour is that what appears is the **mean**.

This behaviour can be overridden by setting a custom value for the argument `aggfunc` (short for "aggregation function"). If we wanted totals for all the prices for each city (a kind of "volume of sales" figure), we'd use the aggregation function `'sum'`, as follows:

```
real_estate_pt1 = real_estate_df.pivot_table(
    index='city',
    values='price',
    aggfunc='sum'
)

real_estate_pt1.head(10)
```

city	price
ANTELOPE	7672381
AUBURN	2029454
CAMERON PARK	2411500
CARMICHAEL	5913695
CITRUS HEIGHTS	6549022
COOL	300000
DIAMOND SPRINGS	216033
EL DORADO	494000
EL DORADO HILLS	11309076
ELK GROVE	30911977

Using `aggfunc=np.sum` would produce the same result.

For a simple count of the number of transactions, we can set `aggfunc` to `'count'` or (equivalently) to `len`:

```
real_estate_pt2 = real_estate_df.pivot_table(
    index='city',
    values='price',
    aggfunc='count'
)

real_estate_pt2.head(10)
```

city	price
ANTELOPE	33
AUBURN	5
CAMERON PARK	9
CARMICHAEL	20
CITRUS HEIGHTS	35
COOL	1
DIAMOND SPRINGS	1
EL DORADO	2
EL DORADO HILLS	23
ELK GROVE	114

Other values for `aggfunc` include `'first'`, `'last'`, `'min'`, `'max'` and `'std'` (standard deviation). You can also use a named function or a lambda-expression; anything that takes a sequence of arguments and returns a single object will do as an aggregation function.

Pivot tables become even more useful when we want to break the data in the c_2 column into further categories; in the case of our real estate data we might want to classify by **city** and then, within that, classify by **type**. The obvious way to do that would be to have more than one **price** column in our table, and that can be achieved by setting a value for the optional argument **columns**:

```
real_estate_pt3 = real_estate_df.pivot_table(
    index='city',
    values='price',
    aggfunc='sum',
    columns='type'
)

real_estate_pt3.head(10)
```

type city	Condo	Multi-Family	Residential	Unkown
ANTELOPE	115000.0	NaN	7557381.0	NaN
AUBURN	260000.0	285000.0	1484454.0	NaN
CAMERON PARK	119000.0	NaN	2292500.0	NaN
CARMICHAEL	571634.0	NaN	5342061.0	NaN
CITRUS HEIGHTS	185250.0	256054.0	6107718.0	NaN
COOL	NaN	NaN	300000.0	NaN
DIAMOND SPRINGS	NaN	NaN	216033.0	NaN
EL DORADO	NaN	NaN	494000.0	NaN
EL DORADO HILLS	NaN	NaN	11309076.0	NaN
ELK GROVE	688000.0	NaN	30223977.0	NaN

Let's tidy this up by turning those NaNs to zeroes and those floats into ints:

```
real_estate_pt3 = real_estate_df.pivot_table(
    index='city',
    values='price',
    aggfunc='sum',
    columns='type'
).fillna(0).astype(int)

real_estate_pt3.head(10)
```

type city	Condo	Multi-Family	Residential	Unkown
ANTELOPE	115000	0	7557381	0
AUBURN	260000	285000	1484454	0
CAMERON PARK	119000	0	2292500	0
CARMICHAEL	571634	0	5342061	0
CITRUS HEIGHTS	185250	256054	6107718	0
COOL	0	0	300000	0
DIAMOND SPRINGS	0	0	216033	0
EL DORADO	0	0	494000	0
EL DORADO HILLS	0	0	11309076	0
ELK GROVE	688000	0	30223977	0

There's another way of creating aggregated tables like this, which operates at a lower level, and is therefore more flexible but somewhat less easy to use; it's called `groupby`. All of the pivot table examples we've seen can be made using `groupby`. Here's an exact copy of the first one:

```
real_estate_gb0 = real_estate_df.groupby('city')[['price']].sum()

real_estate_gb0.head(10)
```

city	price
ANTELOPE	232496.393939
AUBURN	405890.800000
CAMERON PARK	267944.444444
CARMICHAEL	295684.750000
CITRUS HEIGHTS	187114.914286
COOL	300000.000000
DIAMOND SPRINGS	216033.000000
EL DORADO	247000.000000
EL DORADO HILLS	491698.956522
ELK GROVE	271157.692982

Note the way the aggregation works, via the use of `sum` as a **method**; `mean`, `first`, `std` and so on work in the same way. Note too the way we specify the **price** column by indexing using the list `['price']`; if we simply use `'price'` we get a Series rather than a DataFrame:

```
real_estate_gb0_ser = real_estate_df.groupby('city')['price'].sum()

real_estate_gb0.head(10)
```

```
city
ANTELOPE            7672381
AUBURN              2029454
CAMERON PARK        2411500
CARMICHAEL          5913695
CITRUS HEIGHTS      6549022
COOL                 300000
DIAMOND SPRINGS      216033
EL DORADO            494000
EL DORADO HILLS    11309076
ELK GROVE          30911977
Name: price, dtype: int64
```

And here's an equivalent of the last pivot table:

```
real_estate_gb3 =
    real_estate_df.groupby(['city','type'])[['price']].sum()

real_estate_gb3.head(10)
```

		price
city	type	
ANTELOPE	**Condo**	115000
	Residential	7557381
AUBURN	**Condo**	260000
	Multi-Family	285000
	Residential	1484454
CAMERON PARK	**Condo**	119000
	Residential	2292500
CARMICHAEL	**Condo**	571634
	Residential	5342061
CITRUS HEIGHTS	**Condo**	185250

Notice that this contains exactly the same data as `real_estate_pt3`, except that it's in what we call **stacked format**. To make it *exactly* the same, type

```
real_estate_gb3 =
    real_estate_df.groupby(['city','type'])[['price']].sum()

real_estate_gb3.unstack().head(10)
```

Similarly, to get a stacked version of the pivot table, type:

```
real_estate_pt3 = real_estate_df.pivot_table(
    index='city',
    values='price',
    aggfunc='sum',
    columns='type'
).fillna(0).astype(int)

real_estate_pt3,stack().head(10)
```

Both `pivot_table` and `groupby` offer ways of aggregating multiple columns in the original DataFrame; there's one minor difference, which is that `pivot_table` presents them in alphabetical order, whereas `groupby` presents them in the order specified by the user.

Compare and contrast:

```
real_estate_pt4=real_estate_df.pivot_table(
    index='city',
    values=['beds','baths','sq__ft','type','price'],
    aggfunc='last'
)

real_estate_pt4.head(10)
```

city	baths	beds	price	sq__ft	type
ANTELOPE	2	3	212000	1517	Residential
AUBURN	3	4	560000	0	Residential
CAMERON PARK	2	3	224500	0	Residential
CARMICHAEL	2	4	220000	1319	Residential
CITRUS HEIGHTS	2	3	235000	1216	Residential
COOL	2	3	300000	1457	Residential
DIAMOND SPRINGS	2	3	216033	1300	Residential
EL DORADO	1	2	205000	1040	Residential
EL DORADO HILLS	2	3	235738	1362	Residential
ELK GROVE	2	4	235301	1685	Residential

```
real_estate_gb4=real_estate_df.groupby('city')[
    ['beds','baths','sq__ft','type','price']
].last()

real_estate_gb4.head(10)
```

city	beds	baths	sq__ft	type	price
ANTELOPE	3	2	1517	Residential	212000
AUBURN	4	3	0	Residential	560000
CAMERON PARK	3	2	0	Residential	224500
CARMICHAEL	4	2	1319	Residential	220000
CITRUS HEIGHTS	3	2	1216	Residential	235000
COOL	3	2	1457	Residential	300000
DIAMOND SPRINGS	3	2	1300	Residential	216033
EL DORADO	2	1	1040	Residential	205000
EL DORADO HILLS	3	2	1362	Residential	235738
ELK GROVE	4	2	1685	Residential	235301