First and Foremost

- Numpy and Pandas are two of the key modules used in data science
- Numpy, or Numerical Python, is a package for performing mathematical and statistical operations **quickly**
- Pandas is the go-to module for handling datasets
- *Note that we use aliases np and pd. This simply makes referencing them in the code easier, and is best practice.*

In [1]:

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

DataSeries

A DataSeries is a generalised list with advanced indexing.

Suppose I have taken a set of readings of the final speed of a ball in meters per second rolling down a slope.

In [2]:

```
my_readings = [3.12, 3.54, 3.24, 3.67, 3.56, 3.87]
```

We can create a DataSeries using any list or np.array.

In [3]:

```
ds_readings = pd.Series(my_readings)
ds_readings
```

Out[3]:

```
0    3.12
1    3.54
2    3.24
3    3.67
4    3.56
5    3.87
dtype: float64
```

The data series consists of two parts: **index**, and **value.** The index is the ID of the item inside the series

We also get information about the **datatype** of the series, displayed underneath

In [4]:

```python
## Output Index
ds_readings.index
```

Out[4]:

```
RangeIndex(start=0, stop=6, step=1)
```

In [5]:

```python
## Output Value
ds_readings.values
```

Out[5]:

```
array([3.12, 3.54, 3.24, 3.67, 3.56, 3.87])
```

Values inside the series can be referred to by index. Indices start at 0, by default.

In [6]:

```python
ds_readings[0]
```

Out[6]:

```
3.12
```

In [7]:

```python
ds_readings[4]
```

Out[7]:

```
3.56
```

We can also supply a range, if index supports it, by using **colon (:)**

In [8]:

```python
ds_readings[1:4]
```

Out[8]:

```
1    3.54
2    3.24
3    3.67
dtype: float64
```

Alternatively, we can also filter a series using a list of True and Falses. An element at position True will be kept in the output, otherwise it is discarded.

This is not called masking. Masking uses `.mask()`, which is the opposite f `.where()`

In [9]:

```
ds_readings
```

Out[9]:

```
0    3.12
1    3.54
2    3.24
3    3.67
4    3.56
5    3.87
dtype: float64
```

In [10]:

```
## Note list input
ds_readings[[True, False, True, False, False, False]]
```

Out[10]:

```
0    3.12
2    3.24
dtype: float64
```

Operations are done on series element-wise.

In [11]:

```
ds_readings + 10
```

Out[11]:

```
0    13.12
1    13.54
2    13.24
3    13.67
4    13.56
5    13.87
dtype: float64
```

This also applies to comparison operators. Name comparison operators?

In [12]:

```
ds_readings > 3.5
```

Out[12]:

```
0    False
1     True
2    False
3     True
4     True
5     True
dtype: bool
```

Q: Why might the above be useful?

Output would be very useful when used as a mask for another series

In [13]:

```
ds_readings[ds_readings > 3.5]
```

Out[13]:

```
1    3.54
3    3.67
4    3.56
5    3.87
dtype: float64
```

We can always combine filtering and operators

In [14]:

```
ds_readings[ds_readings > 3.5] * 2
```

Out[14]:

```
1    7.08
3    7.34
4    7.12
5    7.74
dtype: float64
```

And to change the values, either all, or partial is easy,

In [15]:

```
ds_readings[ds_readings > 3.5] = 0
```

In [16]:

```
ds_readings
```

Out[16]:

```
0    3.12
1    0.00
2    3.24
3    0.00
4    0.00
5    0.00
dtype: float64
```

There are two methods we can use to perform similar operations to the above. One of these is called `.where()` , and the other `.mask()` . Mask and where can be thought of as opposites of one another, however, they both preserve the initial shape and structure of a dataseries/frame.

`where()`

In [83]:

```
ds_readings.where(ds_readings>0, 20)
```

Out[83]:

```
00:03:00     3.12
00:03:01    20.00
00:03:02     3.24
00:03:03    20.00
00:03:04    20.00
00:03:05    20.00
00:03:06    20.00
00:03:07     3.48
dtype: float64
```

`mask()`

In [85]:

```
ds_readings.mask(ds_readings>0)
```

Out[85]:

```
00:03:00    NaN
00:03:01    0.0
00:03:02    NaN
00:03:03    0.0
00:03:04    0.0
00:03:05    0.0
00:03:06    NaN
00:03:07    NaN
dtype: float64
```

Exercise

- ### Create a new DataSeries on a topic of your choosing(numeric, length = 8)
- ### Output the 2nd, last, and last two elements
- ### Subtract a number from all elements
- ### Generate and apply a mask
- ### Use the mask to set values to 0 ### Hint `pd.Series(<list>)`
    - If finished - find the difference between doing this and using `.where()`

A useful feature of series is that we may choose the way they are indexed. **An index does not have to be sequential** numbers. They can be any python object

In [18]:

```
timings = [datetime.time(0, 3, increment) for increment in range(6)]
```

In [19]:

```
timings
```

Out[19]:

```
[datetime.time(0, 3),
 datetime.time(0, 3, 1),
 datetime.time(0, 3, 2),
 datetime.time(0, 3, 3),
 datetime.time(0, 3, 4),
 datetime.time(0, 3, 5)]
```

In [20]:

```
ds_readings.index = timings
```

In [21]:

```
ds_readings
```

Out[21]:

```
00:03:00    3.12
00:03:01    0.00
00:03:02    3.24
00:03:03    0.00
00:03:04    0.00
00:03:05    0.00
dtype: float64
```

The `.value_counts` function finds all the unique values in the series and gives the number of ocurrences of the same number in the series,

In [22]:

```
ds_readings.value_counts()
```

Out[22]:

```
0.00    4
3.12    1
3.24    1
dtype: int64
```

we can also sort, ascending and descending

In [23]:

```
ds_readings.sort_values()
```

Out[23]:

```
00:03:01    0.00
00:03:03    0.00
00:03:04    0.00
00:03:05    0.00
00:03:00    3.12
00:03:02    3.24
dtype: float64
```

In [24]:

```
ds_readings.sort_values(ascending = False)
```

Out[24]:

```
00:03:02    3.24
00:03:00    3.12
00:03:05    0.00
00:03:04    0.00
00:03:03    0.00
00:03:01    0.00
dtype: float64
```

np.nan

`np.nan` refers to a value that should be but do not exist. And pandas provides an easiy function to check emptiness

We first add np.nan into the seies

In [25]:

```
ds_readings[datetime.time(0,3,6)] =  np.nan
```

In [26]:

```
ds_readings
```

Out[26]:

```
00:03:00    3.12
00:03:01    0.00
00:03:02    3.24
00:03:03    0.00
00:03:04    0.00
00:03:05    0.00
00:03:06     NaN
dtype: float64
```

In [27]:

```
ds_readings[datetime.time(0,3,7)] =  3.48
```

In [28]:

```
ds_readings
```

Out[28]:

```
00:03:00    3.12
00:03:01    0.00
00:03:02    3.24
00:03:03    0.00
00:03:04    0.00
00:03:05    0.00
00:03:06     NaN
00:03:07    3.48
dtype: float64
```

And pandas provides method for checking emptiness

In [29]:

```
ds_readings.isna()
```

Out[29]:

```
00:03:00    False
00:03:01    False
00:03:02    False
00:03:03    False
00:03:04    False
00:03:05    False
00:03:06     True
00:03:07    False
dtype: bool
```

In [30]:

```
ds_readings[~ds_readings.isna()]
```

Out[30]:

```
00:03:00    3.12
00:03:01    0.00
00:03:02    3.24
00:03:03    0.00
00:03:04    0.00
00:03:05    0.00
00:03:07    3.48
dtype: float64
```

In [31]:

```
result = ds_readings[ds_readings.isna()]
```

In [32]:

```
result.index
```

Out[32]:

```
Index([00:03:06], dtype='object')
```

To remove items, use drop. You need to refer to the index value.

In [33]:

```
ds_readings.drop(result.index)
```

Out[33]:

```
00:03:00    3.12
00:03:01    0.00
00:03:02    3.24
00:03:03    0.00
00:03:04    0.00
00:03:05    0.00
00:03:07    3.48
dtype: float64
```

In [34]:

```
ds_readings.isna()
```

Out[34]:

```
00:03:00    False
00:03:01    False
00:03:02    False
00:03:03    False
00:03:04    False
00:03:05    False
00:03:06     True
00:03:07    False
dtype: bool
```

There are reduction methods

In [35]:

```
ds_readings.isna().any()
```

Out[35]:

```
True
```

In [36]:

```
ds_readings.isna().all()
```

Out[36]:

```
False
```

In [37]:

```
ds_readings.isna().sum()
```

Out[37]:

```
1
```

To displace a set of unique values in the series, use `unique()`

In [38]:

```
ds_readings.unique()
```

Out[38]:

```
array([3.12, 0.  , 3.24,  nan, 3.48])
```

Mappings

The `.replace()` function applies a map or function on every element of the `pd.Series()` object that matches the map keys.

In [39]:

```
print(ds_readings)
mapping = {0: 10.0, np.nan: 0.0}
ds_readings.replace(mapping)
```

```
00:03:00    3.12
00:03:01    0.00
00:03:02    3.24
00:03:03    0.00
00:03:04    0.00
00:03:05    0.00
00:03:06     NaN
00:03:07    3.48
dtype: float64
```

Out[39]:

```
00:03:00     3.12
00:03:01    10.00
00:03:02     3.24
00:03:03    10.00
00:03:04    10.00
00:03:05    10.00
00:03:06     0.00
00:03:07     3.48
dtype: float64
```

.map() on the other hand, expects to replace all values in the series and if one isn't in the mapping list it will still replace the values by np.NaN.

We can define a function we wish to apply to the series

In [40]:

```
def myround(x):
    return round(x, 1)
```

In [41]:

```
ds_readings.map(myround)
```

Out[41]:

```
00:03:00    3.1
00:03:01    0.0
00:03:02    3.2
00:03:03    0.0
00:03:04    0.0
00:03:05    0.0
00:03:06    NaN
00:03:07    3.5
dtype: float64
```

Or we can use a lambda, defined within the call of `map()`

In [42]:

```
ds_readings.map(lambda x: round(x,1))
```

Out[42]:

```
00:03:00    3.1
00:03:01    0.0
00:03:02    3.2
00:03:03    0.0
00:03:04    0.0
00:03:05    0.0
00:03:06    NaN
00:03:07    3.5
dtype: float64
```

Exercise

- ### Change the index of the DataSeries you created in the previous exercise so that it is indexed by time
- ### Insert several np.nan values
- ### remove these values using `.isna()` and `.drop()` or ~
- ### Do not do(Define a function which squares numbers given as input and apply it accross the list using `.map()`)

DataFrames

DataFrames are tables, and the main workhorse in pandas. They may also be thought of as collections of series, each representing a column, and every series uses the same index.

Here, we create a dictionary object and pass it to pd.DataFrame().

In [43]:

```
students = ['Alice', 'Bob', 'Emily', 'Charlie']
```

In [44]:

```
scores = [90, 80, 65, 50]
```

In [45]:

```
exam_results = pd.DataFrame(
    {
        "name": students,
        "scores": scores
    }
)
```

In [46]:

```
exam_results
```

Out[46]:

	name	scores
0	Alice	90
1	Bob	80
2	Emily	65
3	Charlie	50

One of the most useful methods we have at our diposal when we are working with DataFrames is `describe()` . The describe method gives us a summary of our DataFrame, generating a profile of all the features. It will default to only display the numeric attributes, but we can get all by passing the keyword `all` .

In [47]:

```
import seaborn as sns
titanic = sns.load_dataset('titanic')

print(titanic.isnull().sum())

#sns.violinplot(t['sex'], t['class'].)

print(titanic.head())
```

```
survived         0
pclass           0
sex              0
age            177
sibsp            0
parch            0
fare             0
embarked         2
class            0
who              0
adult_male       0
deck           688
embark_town      2
alive            0
alone            0
dtype: int64
   survived  pclass     sex   age  sibsp  parch     fare embarked  c
lass  \
0         0       3    male  22.0      1      0   7.2500        S  T
hird
1         1       1  female  38.0      1      0  71.2833        C  F
irst
2         1       3  female  26.0      0      0   7.9250        S  T
hird
3         1       1  female  35.0      1      0  53.1000        S  F
irst
4         0       3    male  35.0      0      0   8.0500        S  T
hird

     who  adult_male deck  embark_town alive  alone
0    man        True  NaN  Southampton    no  False
1  woman       False    C    Cherbourg   yes  False
2  woman       False  NaN  Southampton   yes   True
3  woman       False    C  Southampton   yes  False
4    man        True  NaN  Southampton    no   True
```

One of the most useful methods we have at our diposal when we are working with DataFrames is `describe()` . The describe method gives us a summary of our DataFrame, generating a profile of all the features. It will default to only display the numeric attributes, but we can get all by passing the keyword `all` .

In [48]:

```
titanic.describe(include='all')
```

Out[48]:

	survived	pclass	sex	age	sibsp	parch	fare	embai
count	891.000000	891.000000	891	714.000000	891.000000	891.000000	891.000000	
unique	NaN	NaN	2	NaN	NaN	NaN	NaN	
top	NaN	NaN	male	NaN	NaN	NaN	NaN	
freq	NaN	NaN	577	NaN	NaN	NaN	NaN	
mean	0.383838	2.308642	NaN	29.699118	0.523008	0.381594	32.204208	
std	0.486592	0.836071	NaN	14.526497	1.102743	0.806057	49.693429	
min	0.000000	1.000000	NaN	0.420000	0.000000	0.000000	0.000000	
25%	0.000000	2.000000	NaN	20.125000	0.000000	0.000000	7.910400	
50%	0.000000	3.000000	NaN	28.000000	0.000000	0.000000	14.454200	
75%	1.000000	3.000000	NaN	38.000000	1.000000	0.000000	31.000000	
max	1.000000	3.000000	NaN	80.000000	8.000000	6.000000	512.329200	

It would make more sense to display these seperately, so we specify our seperate inclusin types as `np.number` and `np.object` .

In [49]:

```
titanic.describe(include=np.number)
```

Out[49]:

	survived	pclass	age	sibsp	parch	fare
count	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

In [50]:

```
titanic.describe(include=np.object)
```

Out[50]:

	sex	embarked	who	embark_town	alive
count	891	889	891	889	891
unique	2	3	3	3	2
top	male	S	man	Southampton	no
freq	577	644	537	644	549

We can also import data from various file formats, as we can see from the various read x methods.

In [51]:

```
methods = pd.Series(dir(pd))
methods[methods.str.contains('read')]
```

Out[51]:

```
104     read_clipboard
105           read_csv
106         read_excel
107       read_feather
108           read_fwf
109           read_gbq
110           read_hdf
111          read_html
112          read_json
113       read_msgpack
114       read_parquet
115        read_pickle
116           read_sas
117           read_sql
118     read_sql_query
119     read_sql_table
120         read_stata
121         read_table
dtype: object
```

Searching (SQL SELECT)

To search the table, we can use both the `.loc()` and the `.iloc()` methods. The former is used to access labels, while the latter is used to access indices.

In [91]:

```
titanic.loc[0,'age']
```

Out[91]:

```
22.0
```

In [92]:

```
titanic.loc[:,'age'].head()
```

Out[92]:

```
0    22.0
1    38.0
2    26.0
3    35.0
4    35.0
Name: age, dtype: float64
```

In [94]:

```
titanic.loc[2,:]
```

Out[94]:

```
survived                 1
pclass                   3
sex                 female
age                     26
sibsp                    0
parch                    0
fare                 7.925
embarked                 S
class                Third
who                  woman
adult_male           False
deck                   NaN
embark_town    Southampton
alive                  yes
alone                 True
Name: 2, dtype: object
```

`.iloc[]`

In [96]:

```
titanic.iloc[2]
```

Out[96]:

	sex	age
0	male	22.0
1	female	38.0
2	female	26.0
3	female	35.0
4	male	35.0
5	male	NaN
6	male	54.0
7	male	2.0
8	female	27.0
9	female	14.0
10	female	4.0
11	female	58.0
12	male	20.0
13	male	39.0
14	female	14.0
15	female	55.0
16	male	2.0
17	male	NaN
18	female	31.0
19	female	NaN
20	male	35.0
21	male	34.0
22	female	15.0
23	male	28.0
24	female	8.0
25	female	38.0
26	male	NaN
27	male	19.0
28	female	NaN
29	male	NaN
...	...	...
861	male	21.0
862	female	48.0
863	female	NaN
864	male	24.0
865	female	42.0

	sex	age
866	female	27.0
867	male	31.0
868	male	NaN
869	male	4.0
870	male	26.0
871	female	47.0
872	male	33.0
873	male	47.0
874	female	28.0
875	female	15.0
876	male	20.0
877	male	19.0
878	male	NaN
879	female	56.0
880	female	25.0
881	male	33.0
882	female	22.0
883	male	28.0
884	male	25.0
885	female	39.0
886	male	27.0
887	female	19.0
888	female	NaN
889	male	26.0
890	male	32.0

891 rows × 2 columns

In [ ]:

```
titanic.iloc[2:4]
```

In [ ]:

```
titanic.iloc[,2:4]
```

columns can either be referred by using `loc` , or using the indexer `[]` directly on the dataframe

Columns are returned as a pandas series, with the same index used in the dataframe

We can request multiple columns

In [103]:

```
titanic.loc[:,('age','survived')].head()
```

Out[103]:

	age	survived
0	22.0	0
1	38.0	1
2	26.0	1
3	35.0	1
4	35.0	0

In [104]:

```
titanic.describe()
```

Out[104]:

	survived	pclass	age	sibsp	parch	fare
count	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

In [106]:

```
titanic['cost per year alive'] = titanic.loc[:,'fare']/titanic.loc[:,'age']
titanic.describe()
```

Out[106]:

	survived	pclass	age	sibsp	parch	fare	pounds per year alive
count	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000	714.000000
mean	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208	2.391841
std	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429	8.115102
min	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400	0.342403
50%	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200	0.565217
75%	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000	1.673857
max	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200	164.728261

In [111]:

```
titanic.loc[:,('cost per year alive','age','fare')].sort_values('cost per year a
live',ascending=False).head()
```

Out[111]:

	cost per year alive	age	fare
305	164.728261	0.92	151.5500
297	75.775000	2.00	151.5500
386	46.900000	1.00	46.9000
164	39.687500	1.00	39.6875
183	39.000000	1.00	39.0000

Remove Columns

In [112]:

```
titanic.drop('cost per year alive', axis=1).head()
```

Out[112]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_mal
0	0	3	male	22.0	1	0	7.2500	S	Third	man	Tru
1	1	1	female	38.0	1	0	71.2833	C	First	woman	Fals
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	Fals
3	1	1	female	35.0	1	0	53.1000	S	First	woman	Fals
4	0	3	male	35.0	0	0	8.0500	S	Third	man	Tru

Column names and Row Index

In [61]:

```
titanic.columns
```

Out[61]:

```
Index(['name', 'scores', 'gender'], dtype='object')
```

In [62]:

```
titanic.index
```

Out[62]:

```
RangeIndex(start=0, stop=4, step=1)
```

Renaming Columns

In [63]:

```
titanic.rename({'alive': 'not dead'}, axis=1)
```

Out[63]:

	first_name	scores	gender
0	Alice	90	female
1	Bob	80	male
2	Emily	65	female
3	Charlie	50	male

Dealing with missing values

Dataframes have specific commands which help with the handling of missing data

In [113]:

```
titanic.isna()
```

Out[113]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male
0	False	False	False	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False	False	False
5	False	False	False	True	False	False	False	False	False	False	False
6	False	False	False	False	False	False	False	False	False	False	False
7	False	False	False	False	False	False	False	False	False	False	False
8	False	False	False	False	False	False	False	False	False	False	False
9	False	False	False	False	False	False	False	False	False	False	False
10	False	False	False	False	False	False	False	False	False	False	False
11	False	False	False	False	False	False	False	False	False	False	False
12	False	False	False	False	False	False	False	False	False	False	False
13	False	False	False	False	False	False	False	False	False	False	False
14	False	False	False	False	False	False	False	False	False	False	False
15	False	False	False	False	False	False	False	False	False	False	False
16	False	False	False	False	False	False	False	False	False	False	False
17	False	False	False	True	False	False	False	False	False	False	False
18	False	False	False	False	False	False	False	False	False	False	False
19	False	False	False	True	False	False	False	False	False	False	False
20	False	False	False	False	False	False	False	False	False	False	False
21	False	False	False	False	False	False	False	False	False	False	False
22	False	False	False	False	False	False	False	False	False	False	False
23	False	False	False	False	False	False	False	False	False	False	False
24	False	False	False	False	False	False	False	False	False	False	False
25	False	False	False	False	False	False	False	False	False	False	False
26	False	False	False	True	False	False	False	False	False	False	False
27	False	False	False	False	False	False	False	False	False	False	False
28	False	False	False	True	False	False	False	False	False	False	False
29	False	False	False	True	False	False	False	False	False	False	False
...	...	...	...	...	...	...	...	...	...	...	...
861	False	False	False	False	False	False	False	False	False	False	False
862	False	False	False	False	False	False	False	False	False	False	False
863	False	False	False	True	False	False	False	False	False	False	False

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male
864	False	False	False	False	False	False	False	False	False	False	False
865	False	False	False	False	False	False	False	False	False	False	False
866	False	False	False	False	False	False	False	False	False	False	False
867	False	False	False	False	False	False	False	False	False	False	False
868	False	False	False	True	False	False	False	False	False	False	False
869	False	False	False	False	False	False	False	False	False	False	False
870	False	False	False	False	False	False	False	False	False	False	False
871	False	False	False	False	False	False	False	False	False	False	False
872	False	False	False	False	False	False	False	False	False	False	False
873	False	False	False	False	False	False	False	False	False	False	False
874	False	False	False	False	False	False	False	False	False	False	False
875	False	False	False	False	False	False	False	False	False	False	False
876	False	False	False	False	False	False	False	False	False	False	False
877	False	False	False	False	False	False	False	False	False	False	False
878	False	False	False	True	False	False	False	False	False	False	False
879	False	False	False	False	False	False	False	False	False	False	False
880	False	False	False	False	False	False	False	False	False	False	False
881	False	False	False	False	False	False	False	False	False	False	False
882	False	False	False	False	False	False	False	False	False	False	False
883	False	False	False	False	False	False	False	False	False	False	False
884	False	False	False	False	False	False	False	False	False	False	False
885	False	False	False	False	False	False	False	False	False	False	False
886	False	False	False	False	False	False	False	False	False	False	False
887	False	False	False	False	False	False	False	False	False	False	False
888	False	False	False	True	False	False	False	False	False	False	False
889	False	False	False	False	False	False	False	False	False	False	False
890	False	False	False	False	False	False	False	False	False	False	False

891 rows × 17 columns

In [114]:

```
titanic.isna().sum()
```

Out[114]:

```
survived                 0
pclass                   0
sex                      0
age                    177
sibsp                    0
parch                    0
fare                     0
embarked                 2
class                    0
who                      0
adult_male               0
deck                   688
embark_town              2
alive                    0
alone                    0
pounds per year alive  177
cost per year alive    177
dtype: int64
```

The axis tells us whether we drop rows, columns, other directional dimensions etc.

In [119]:

```
titanic.dropna(axis=1).isna().sum()
```

Out[119]:

```
survived      0
pclass        0
sex           0
sibsp         0
parch         0
fare          0
class         0
who           0
adult_male    0
alive         0
alone         0
dtype: int64
```

Group by

The `groupby()` method of `DataFrame` behaves just like the SQL `GROUP BY` clause.

It groups rows into several groups based on the criteria we put in the argument. Any aggregations or mappings appied afterwards will be done on each individual groups.

For example, suppose we wish to compute the mean scores of different genders in the `exam_results` table, then we can do a groupby.

In [121]:

```
titanic.groupby('pclass').mean()
```

Out[121]:

pclass	survived	age	sibsp	parch	fare	adult_male	alone	pounds per year alive
1	0.629630	38.233441	0.416667	0.356481	84.154687	0.550926	0.504630	4.189522
2	0.472826	29.877630	0.402174	0.380435	20.662183	0.538043	0.565217	2.088590
3	0.242363	25.140620	0.615071	0.393075	13.675550	0.649695	0.659878	1.597739

The criteria do not have to be column names, they can be any array-like object, or function (acting on the index).

If the criterion is an array-like object, e.g. a list, np.array, or pd.Series, then this array is mapped directly to the row index, and the row indices are grouped in such way that all the same values in the array are grouped together.

In [70]:

```
mylist = [1,2,1,2]

exam_results.groupby(mylist).get_group(1)
```

Out[70]:

	name	scores	gender
0	Alice	90	female
2	Emily	65	female

In [71]:

```
exam_results.groupby(mylist).get_group(2)
```

Out[71]:

	name	scores	gender
1	Bob	80	male
3	Charlie	50	male

Thus, suppose we want to group people by fares above or below 70. We would write

In [123]:

```
fare_over_70 = titanic['fare'] > 70
fare_over_70.head()
```

Out[123]:

```
0    False
1     True
2    False
3    False
4    False
Name: fare, dtype: bool
```

This will compute the mean scores of students who scored above or below 70

In [124]:

```
titanic.groupby(fare_over_70).mean()
```

Out[124]:

	survived	pclass	age	sibsp	parch	fare	adult_male	alone
fare								
False	0.338422	2.477099	28.919368	0.505089	0.353690	18.537876	0.634860	0.642494
True	0.723810	1.047619	34.658969	0.657143	0.590476	134.506467	0.361905	0.304762

Quick Plotting

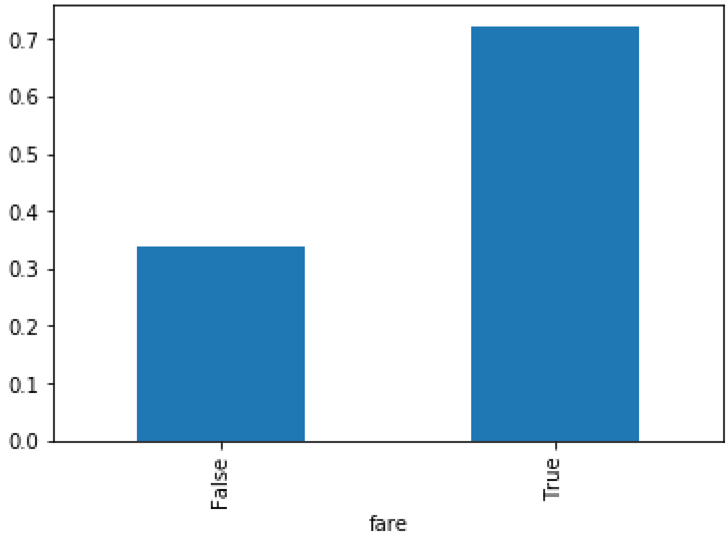
Dataframes and DataSeries come with inbuilt plotting methods. We go into more depth about how we use these in the next section, but we will demomnstrate how easy it is to quicly generate a plot below using `.plot()`

In [134]:

```
titanic.groupby(fare_over_70).mean().loc[:,'survived'].plot(kind='bar', x='Fare
 over £70')
```

Out[134]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1a18a4e208>
```



Merge and Joining

In [74]:

```
contacts = pd.DataFrame(
    {
        'name' : ['Alice', 'Charlie'],
        'tel'  : ['+44 1234 5678', '+44 3245 5564']
    }
)
```

In [75]:

```
exam_results.merge(contacts, left_on='name', right_on='name')    # defaults to in
ner join
```

Out[75]:

	name	scores	gender	tel
0	Alice	90	female	+44 1234 5678
1	Charlie	50	male	+44 3245 5564

In [76]:

```
exam_results.merge(contacts, left_on='name', right_on='name', how='outer')
```

Out[76]:

	name	scores	gender	tel
0	Alice	90	female	+44 1234 5678
1	Bob	80	male	NaN
2	Emily	65	female	NaN
3	Charlie	50	male	+44 3245 5564

In [77]:

```
tel = contacts['tel']
tel.index = contacts['name']

exam_results.join(tel, on='name')        # defaults to left outer join
```

Out[77]:

	name	scores	gender	tel
0	Alice	90	female	+44 1234 5678
1	Bob	80	male	NaN
2	Emily	65	female	NaN
3	Charlie	50	male	+44 3245 5564

In [ ]:

```

```