INTRODUCTION TO PANDAS

WHAT IS PANDAS?

- A Python package for data manipulation and analysis
- Simplifies common tasks carried out with data
- Fast and efficient handling of large amounts of data

OVERVIEW OF FUNCTIONALITY

- Load data from numerous type of files and online sources
- Filtering, sorting, editing and processing of data
- Joining and aggregation of datasets
- Tools for time series and statistical analysis
- Display of data in tables and charts

GETTING STARTED

Importing pandas

```
import pandas as pd
```

The pandas package is imported and given an alias of `pd`

Reading a CSV file

```
df = pd.read_csv('data/titanic.csv')
```

- The pandas `.read_csv()` function is called
- It has only one required parameter, which is the location of a CSV (comma-separated values) file
- `'data/titanic.csv'` is provided as the argument for this parameter
- The result (which is a pandas DataFrame object) is assigned to a variable called `df`

GETTING AN OVERVIEW OF THE DATASET

```
df.head()
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	ali
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	n
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	ye
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	ye
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	ye
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	n

pandas DataFrame objects (such as `df` which was just created) have numerous associated methods:

- `.head()` is an example of one of these; by default it will return the top five rows of the DataFrame
- This method takes one optional parameter, which is the number of rows to display
- `.tail()` works in the same way, except that is shows the last rows in the DataFrame
- `.sample(5)` allows you to sample rows at random

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   survived     891 non-null    int64
 1   pclass       891 non-null    int64
 2   sex          891 non-null    object
 3   age          714 non-null    float64
 4   sibsp        891 non-null    int64
 5   parch        891 non-null    int64
 6   fare         891 non-null    float64
 7   embarked     889 non-null    object
 8   class        891 non-null    object
 9   who          891 non-null    object
 10  adult_male   891 non-null    bool
 11  deck         203 non-null    object
 12  embark_town  889 non-null    object
 13  alive        891 non-null    object
 14  alone        891 non-null    bool
dtypes: bool(2), float64(2), int64(4), object(7)
memory usage: 92.4+ KB
```

- The `.info()` method allows us to understand more about the dataset; how many values there are and what might be missing
- Notice it tells us about `dtypes`; these are the pandas data types for each column as interpreted by pandas when reading the CSV file

```
df.describe()
```

	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

- The `.describe()` method provides statistics about the columns with numerical data, from which we can get a feel of the range, distribution and skew of the values
    - `count`, `mean`, `min` and `max` are self-explanatory
    - `std` gives us the standard deviation of the column values
    - `25%`, `50%` and `75%` give us the quartiles

```
df.shape
```

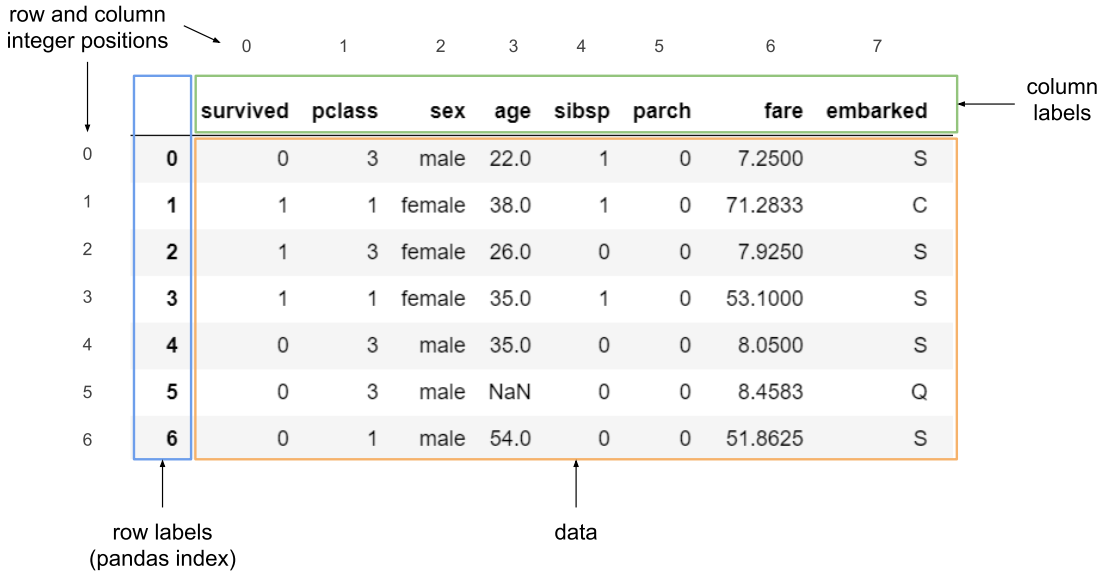
```
(891, 15)
```

The `.shape` attribute of a DataFrame gives us the number of rows and columns it contains

```
rows = df.shape[0]
rows
```

```
891
```

WHAT'S A DATAFRAME?



	survived	pclass	sex	age	sibsp	parch	fare	embarked
0	0	3	male	22.0	1	0	7.2500	S
1	1	1	female	38.0	1	0	71.2833	C
2	1	3	female	26.0	0	0	7.9250	S
3	1	1	female	35.0	1	0	53.1000	S
4	0	3	male	35.0	0	0	8.0500	S
5	0	3	male	NaN	0	0	8.4583	Q
6	0	1	male	54.0	0	0	51.8625	S

- A two-dimensional tabular data structure made up of rows and columns of data
- Rows and columns have associated labels to enable data selection and filtering

- Each row represents an entry (or 'case' or 'record') of values associated with one another
- Values in a given column are of the same pandas datatype (although note that the pandas `object` datatype can accomodate numerous Python datatypes, such as lists, dictionaries etc)

PANDAS DATA TYPES

- object - used for strings, or if the column contains a mix of data types
- int64 - used for integers ('64' relates to memory usage)
- float64 - used for floats, or where the column has both integers and NaN values
- datetime64 / timedelta - time-based values
- bool - booleans, i.e. True or False

PANDAS MISSING VALUES

- NaN is used to indicate missing values in most instances, and is supported by the `float` datatype
- NaT is used to indicate missing values where a `datetime` object may have been expected

```
df.dtypes
```

```
survived         int64
pclass           int64
sex             object
age            float64
sibsp            int64
parch            int64
fare           float64
embarked        object
class           object
who             object
adult_male        bool
deck            object
embark_town     object
alive           object
alone             bool
dtype: object
```

RETRIEVING SPECIFIC DATA FROM A DATAFRAME

This is known as indexing in pandas, and can be done using:

- position: zero-based integers for row and column location
- labels: row labels (numeric by default) and column names

THE DATAFRAME INDEX

- The row labels (which can be numeric or text-based) are known as the index of the DataFrame
- When a DataFrame is created, pandas by default creates an integer-based index, called a RangeIndex
- The `.index` attribute has information about the index of the DataFrame

```
df.index
```

```
RangeIndex(start=0, stop=891, step=1)
```

USE .iloc[] FOR POSITION-BASED INDEXING

```
df.iloc[0:2, 0:4]
```

	survived	pclass	sex	age
0	0	3	male	22.0
1	1	1	female	38.0

- Note the [brackets] rather than (parentheses)
- The syntax is `[rows,columns]`, where `rows` and `columns` can be either single values or ranges
- Ranges work in the same manner as Python lists (0:3 returns three elements, from positions 0, 1 and 2)

```
df.iloc[0:2]
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	ali
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	n
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	y

- If only a single value or single range is provided, all columns for the given rows will be returned (unless preceeded by `:,`, in which case all rows for the given column(s) would be returned)

```
df.iloc[1000:1002]
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alo

- If the index is not found or out of range, an empty DataFrame will be returned (rather than an error message)

```
df.iloc[0].head()
```

```
survived       0
pclass         3
sex         male
age           22
sibsp          1
Name: 0, dtype: object
```

- Explicitly requesting a specific single row or column will return a pandas Series rather than a DataFrame

USE .loc[] FOR LABEL-BASED INDEXING

```
df.loc[0:3, ['alone','age', 'sex']]
```

	alone	age	sex
0	False	22.0	male
1	False	38.0	female
2	True	26.0	female
3	False	35.0	female

- In this instance we could have used .iloc instead of .loc, because the position-based and label-based row indexes are equivalent
- If the row index was text-based (for example, passenger names), we would need to use .loc

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

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

- As with `.iloc()`, we can provide a pair of values (or a pair of groups of values), in the format `[rows,columns]`
- Providing only a single value or value group will result in indexing only by row, with all columns being returned
- Groups of values can be provided as a [list], (tuple), or {set}; if a set is provided, the original column order will be retained in the output

- Note how `0:3` is interpreted differently here, in that this refers to the labels of the row index rather than the position, and that row `3` is included, whereas in the positional syntax (and usual Python list syntax) this would not be the case.

- label-based indexing (`.loc`) is more robust for looking up data, in that changes to the order or presence of rows or columns will not result in the wrong data being returned

UPDATING VALUES FOUND USING INDEXING

- We can use these indexing methods to both get (i.e. retrieve values) and set (i.e. add or modify values)

AN EXAMPLE

```
deck_df = df.copy()
deck_df.loc[0, 'deck'] = 'G'
deck_df.head(2)
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	ali
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	G	Southampton	n
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	ye

```
deck_df = df.copy()
```

- We created a copy of the original DataFrame, so that our modifications do not affect it

```
deck_df.loc[0, 'deck'] = 'G'
```

- We used label-based indexing to locate a specific cell and assign the value `G` to it

- Using `deck_df = df` would have meant that subsequent modifications made to the data in `deck_df` also affected `df`
- This is due to the way pandas works, sometimes creating a view of the original DataFrame rather than a copy of it
- The rules behind when and why pandas creates views or copies are quite complex; if in doubt, consider using `.copy()`

Part 1: Importing, examining, and updating values in a dataset

intro-pandas-workbook.ipynb

FILTERING DATAFRAMES

COLUMN SELECTION

```
df['age'].head(2) # bracket notation
```

```
0    22.0
1    38.0
Name: age, dtype: float64
```

```
df.age.head(2) # dot notation
```

```
0    22.0
1    38.0
Name: age, dtype: float64
```

- Providing a single column name within [brackets] after the DataFrame name will return the given Series
- Dot notation can also be used to return single columns, although requires column_names with no spaces in the name, and cannot be used to create new columns
- Bracket notation always works but some people prefer dot notation when possible for readability.

```
df[['survived','alone']].head(2)
```

	survived	alone
0	0	False
1	1	False

- Providing several column names in a list will return a DataFrame with the given columns

```
df_less_cols = df.drop(['class', 'embarked', 'alive'], axis=1)
df_less_cols.head()
```

	survived	pclass	sex	age	sibsp	parch	fare	who	adult_male	deck	embark_town	alone
0	0	3	male	22.0	1	0	7.2500	man	True	NaN	Southampton	False
1	1	1	female	38.0	1	0	71.2833	woman	False	C	Cherbourg	False
2	1	3	female	26.0	0	0	7.9250	woman	False	NaN	Southampton	True
3	1	1	female	35.0	1	0	53.1000	woman	False	C	Southampton	False
4	0	3	male	35.0	0	0	8.0500	man	True	NaN	Southampton	True

- The `.drop()` method allows us to specify labels of rows or columns to drop
- The `axis` parameter determines whether the labels refer to rows (`0`, the default) or columns (`1`)
- We have assigned the result to a new variable; if we wanted to remove these columns from the original DataFrame, we need to add `inplace=True` when calling the method

ROW SELECTION AND MASKS

```
is_male_mask = df['sex'] == 'male'

males = df[is_male_mask]
males.head()
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no
5	0	3	male	NaN	0	0	8.4583	Q	Third	man	True	NaN	Queenstown	no
6	0	1	male	54.0	0	0	51.8625	S	First	man	True	E	Southampton	no
7	0	3	male	2.0	3	1	21.0750	S	Third	child	False	NaN	Southampton	no

```
is_male_mask = df['sex'] == 'male'
```

- Will create a "mask" that allows to select only rows that match this condition.
- The mask is actually a series of `True` and `False`, that corresponds to whether or not the row should be kept

```
is_male_mask.head()
```

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

```
males = df[is_male_mask]
```

- Will apply the mask on our dataframe and filter the rows

```
msk_female = df['sex'] == 'female'
msk_child = df['who'] == 'child'

girls = df[msk_female & msk_child]
girls.head()
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	al
9	1	2	female	14.0	1	0	30.0708	C	Second	child	False	NaN	Cherbourg	y
10	1	3	female	4.0	1	1	16.7000	S	Third	child	False	G	Southampton	y
14	0	3	female	14.0	0	0	7.8542	S	Third	child	False	NaN	Southampton	r
22	1	3	female	15.0	0	0	8.0292	Q	Third	child	False	NaN	Queenstown	y
24	0	3	female	8.0	3	1	21.0750	S	Third	child	False	NaN	Southampton	r

We can combine multiple masks:

```
girls = df[msk_female & msk_child]
```

retrieve from `df` where `df['sex']` equals `'female'` and `df['who']` equals `'child'`

AN EXAMPLE COMBINING FILTERING AND INDEXING

Finding the age of the first survivor in our DataFrame

```
msk_survived = df['survived'] == 1
survived = df[msk_survived]
survived.head(3)
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	ali
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	ye
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	ye
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	ye

```
survived.iloc[0].loc['age']
```

```
38.0
```

- Here we have used `.iloc` (position-based) to access the first entry in the `survived` DataFrame, and then `.loc` (label-based) to access the `age` of this entry
- using `.loc` in the first part of the statement would have thrown an error since there is no entry in the filtered DataFrame with a row index label of `0`

```
details = survived.iloc[4].loc[['class', 'age']]
details
```

```
class    Second
age          14
Name: 9, dtype: object
```

```
details.name
```

```
9
```

- `details` here is itself a Series, with a `.name` attribute which is the row label (`9`) in addition to the requested values

AN EXAMPLE USING FILTERING AND STRING MATCHING

Does `embarked` give us any different information to `embark_town`? (if not, we might decide to remove it)

```
embark_mismatch = df['embarked'] != df['embark_town'].str[0]
df[embark_mismatch]
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	ali
61	1	1	female	38.0	0	0	80.0	NaN	First	woman	False	B	NaN	ye
829	1	1	female	62.0	0	0	80.0	NaN	First	woman	False	B	NaN	ye

- Here we are querying `df` for any entries where `embarked` is not equal (`-!`) to the first character (`[0]`) in the string (`.str`) of the value in the `embark_town` column
- Two rows are returned, both of which have missing values in both columns, meaning that pandas cannot perform the comparison operation on them

```
embark_info_avail = df['embarked'].notnull() | df['embark_town'].notnull()
```

- Here we have used the `.notnull()` method and the pipe (`|`) character, which is an OR operator
- Values in the resulting Series will be `True` if there is a value in either of the two columns

```
df[embark_mismatch & embark_info_avail]
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alo

```
df[embark_mismatch & ~embark_info_avail]
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	ali
61	1	1	female	38.0	0	0	80.0	NaN	First	woman	False	B	NaN	ye
829	1	1	female	62.0	0	0	80.0	NaN	First	woman	False	B	NaN	ye

- We we have used the masks to filter our DataFrame, combining them with the `&` operator; both masks must be `True` for the given row to be returned
- Masks can be inverted (turning `True` to `False` and `False` to `True`) using a preceding `~`
- This approach can be useful if we are likely to want to re-use or combine different filters

SORTING DATAFRAMES

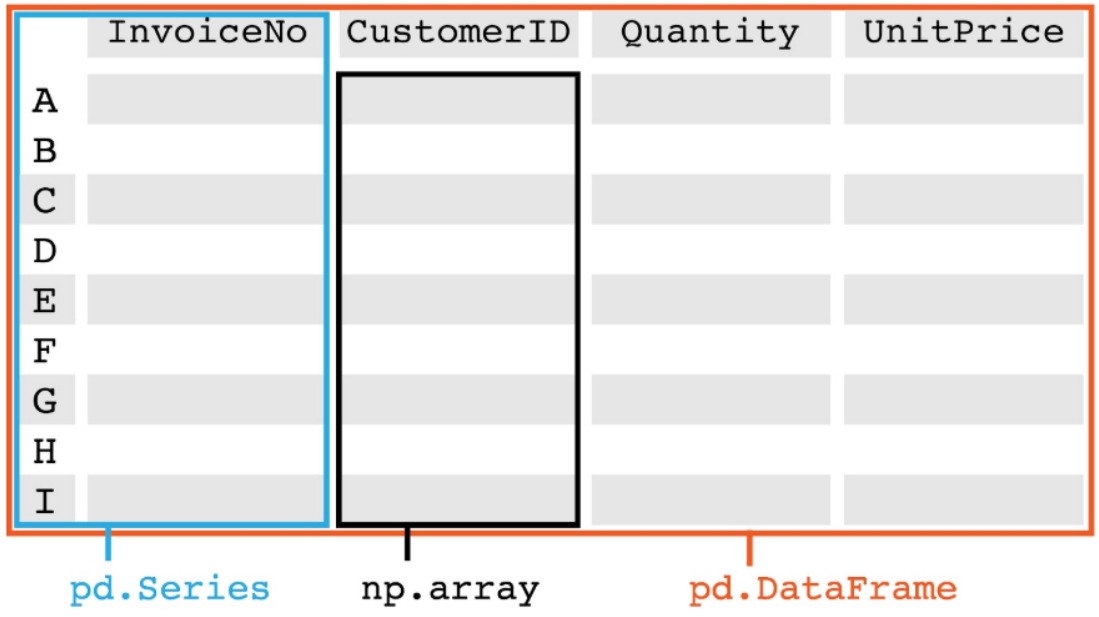
```
df.sort_values(by='fare', ascending=False)
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town
258	1	1	female	35.0	0	0	512.3292	C	First	woman	False	NaN	Cherbourg
737	1	1	male	35.0	0	0	512.3292	C	First	man	True	B	Cherbourg
679	1	1	male	36.0	0	1	512.3292	C	First	man	True	B	Cherbourg
88	1	1	female	23.0	3	2	263.0000	S	First	woman	False	C	Southampton
27	0	1	male	19.0	3	2	263.0000	S	First	man	True	C	Southampton
...	...	...	...	...	...	...	...	...	...	...	...	...	...
633	0	1	male	NaN	0	0	0.0000	S	First	man	True	NaN	Southampton
413	0	2	male	NaN	0	0	0.0000	S	Second	man	True	NaN	Southampton
822	0	1	male	38.0	0	0	0.0000	S	First	man	True	NaN	Southampton

891 rows × 15 columns

- The `sort_values()` method allows us to choose a column in the DataFrame by which to sort it
- The default `ascending` parameter is `True`

PANDAS SERIES



`np.array`
A grid of values, all of the same data type, with an integer-based index (np is for NumPy, a package used by pandas)
`pd.Series`
A one-dimensional NumPy array with a label-based index; index labels can be integers or text
`pd.DataFrame`
A two-dimensional labelled data structure where columns can be of different data types, i.e. a collection of Series of equal length and the same index

SERIES METHODS

Some examples for numerical columns:

```
df['age'].mean()
```

```
29.69911764705882
```

```
df['age'].max()
```

```
80.0
```

```
df['age'].std() # standard deviation
```

```
14.526497332334042
```

And for text columns:

```
df['embark_town'].unique()
```

```
array(['Southampton', 'Cherbourg', 'Queenstown', nan], dtype=object)
```

```
df['embark_town'].unique().tolist()
```

```
['Southampton', 'Cherbourg', 'Queenstown', nan]
```

- The `.unique()` pandas Series method returns a NumPy array
- `.tolist()` is a NumPy array method which returns a list of the values
- Notice how we can chain one method after another in a single line of code

```
df['sex'].value_counts().to_list()
```

```
[577, 314]
```

```
df['sex'].value_counts().to_dict()
```

```
{'male': 577, 'female': 314}
```

- Notice the slightly different method name here (`.to_list()` vs `.tolist()` from the previous example, which was called on a NumPy array)
- `to_list()` is the equivalent method in pandas for Series objects, which is what is returned by the `value_counts()` method
- In this example, a dictionary may be more useful, so we can be sure which number is related to each sex

SERIES ATTRIBUTES

```
df['deck'].shape # the dimensions of the series
```

```
(891,)
```

```
df['deck'].index # if the index labels were strings, those would be returned
```

```
RangeIndex(start=0, stop=891, step=1)
```

```
df['deck'].hasnans # are there any nan values in the series?
```

```
True
```

- pandas Series objects have numerous methods and attributes available
- We can extract data from them into NumPy and Python data types
- See the documentation for further examples

AN EXAMPLE COMBINING SERIES METHODS WITH DATAFRAME FILTERING

```
top_fare_origins = df[df['fare']==df['fare'].max()]['embark_town'].value_counts()
```

```
top_fare_origins
```

```
Cherbourg    3
Name: embark_town, dtype: int64
```

```
df[df['fare'] == df['fare'].max()]
```

- We selected only the rows from the DataFrame where the `fare` was equal to the `.max()` value in the `fare` column

```
['embark_town'].value_counts()
```

- From the resulting DataFrame, we retrieved the `value_counts()` for the `embark_town` column

MODIFYING DATAFRAMES

SETTING THE INDEX USING VALUES FROM A COLUMN

```
df2 = df.set_index('embark_town')
df2.head()
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	alive
embark_town													
Southampton	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	no
Cherbourg	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	yes
Southampton	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	yes
Southampton	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	yes
Southampton	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	no

```
df2.index
```

```
Index(['Southampton',   'Cherbourg', 'Southampton', 'Southampton',
       'Southampton',  'Queenstown', 'Southampton', 'Southampton',
       'Southampton',   'Cherbourg',
       ...
       'Southampton', 'Southampton', 'Southampton', 'Southampton',
        'Queenstown', 'Southampton', 'Southampton', 'Southampton',
         'Cherbourg',  'Queenstown'],
      dtype='object', name='embark_town', length=891)
```

```
df3 = df.set_index('survived', drop=False)
df3.head()
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_to
survived													
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampt
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbour
1	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampt
1	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampt
0	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampt

```
df3.index
```

```
Int64Index([0, 1, 1, 1, 0, 0, 0, 0, 1, 1,
            ...
            0, 0, 0, 0, 0, 0, 1, 0, 1, 0],
           dtype='int64', name='survived', length=891)
```

- The `set_index` method can be used to set a given column as the index
- By default, the chosen column is dropped from the DataFrame
    - Here we have overwritten the default parameter using `drop=False`; we see the `survived` values both as the index and as a column within the DataFrame itself

- Note how on each occasion we assigned the returned DataFrame to a new variable (`df2` or `df3`)
    - When using the default parameters for the `.set_index` method, the original DataFrame is not itself modified
- We can assign the result to a new variable, or we can use the `inplace=True` parameter when calling the `set_index` method, which will modify the original DataFrame

- If we have an index which we would prefer to have as a normal data column, the `reset_index` method can be used
- Again use `inplace=True` if we want to modify the existing DataFrame

```
df2.reset_index(inplace=True)
df2.head()
```

	embark_town	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	ali
0	Southampton	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	n
1	Cherbourg	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	y
2	Southampton	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	y
3	Southampton	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	y
4	Southampton	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	n

- Index values do not have to be unique and can be numbers or text
- The number of rows and row order remain unchanged
- Data columns can be used as the index and vice-versa

ADDING COLUMNS

`sibsp` is the number of siblings and partners the passenger had on board `parch` is the number of parents and children the passenger had on board

```
df['relatives'] = df['sibsp'] + df['parch']
df['relatives'].mean()
```

```
0.9046015712682379
```

`sibsp` is the number of siblings and partners the passenger had on board
`parch` is the number of parents and children the passenger had on board

- We can specify the name of the column and the calculation for the values in a single line
- If a column with the given name exists, its values will be overwritten. If not, a new column will be created

MODIFYING COLUMN DATA

```
df['survived'] = df['survived'].astype(bool)
df['survived'].value_counts()
```

```
False    549
True     342
Name: survived, dtype: int64
```

- We used the `.astype()` method to change a column containing `1` and `0` values to `True` and `False` values
- Note that pandas will use the `float64` data type for a column containing both integers and missing values; for this reason we cannot convert the `age` column to `int`

`.map()`

To do a simple replacement `old value` -> `new value`, you can also use `.map()`:

```
df['survived'].map({1: True, 0: False})
```

- You need to provide a dictionary that maps the old values to new ones

REMOVING COLUMNS WITH .drop()

```
df_new = df.drop(['embarked', 'class', 'alive'], axis=1)
df_new.head(3)
```

	survived	pclass	sex	age	sibsp	parch	fare	who	adult_male	deck	embark_town	alone	relatives
0	False	3	male	22.0	1	0	7.2500	man	True	NaN	Southampton	False	1
1	True	1	female	38.0	1	0	71.2833	woman	False	C	Cherbourg	False	1
2	True	3	female	26.0	0	0	7.9250	woman	False	NaN	Southampton	True	0

- We used the `.drop()` method with a list of labels and the `axis` on which to operate (without `axis=1` the method would search the row index labels for matches and throw an error if each label is not found)
- We could have used `inplace=True` on the original DataFrame, although this would be destructive in that the columns would be lost, and all of our code would need to be re-run to retrieve them

RENAMING COLUMNS WITH .rename()

```
df_new.rename(columns={'pclass':'passenger_class',
                       'sibsp':'siblings_partners',
                       'parch':'parents_children'}
              ).head(3)
```

	survived	passenger_class	sex	age	siblings_partners	parents_children	fare	who	adult_male	deck	emb
0	False	3	male	22.0	1	0	7.2500	man	True	NaN	Sou
1	True	1	female	38.0	1	0	71.2833	woman	False	C	Ch
2	True	3	female	26.0	0	0	7.9250	woman	False	NaN	Sou

- The `.rename()` method allows us to modify column and row index labels
- The `columns` parameter can take a dictonary of key:value pairs where the key is the original label and the value is the new label

USING FUNCTIONS TO ADD OR MODIFY DATA

Here's a function to help us convert the strings in the `alive` column to boolean values:

```
def alive_bool(text):

    return text == 'yes'
```

Notice that when we want a Boolean result from a statement, we do not need to create an `if: ... else: ...` code block; the statement itself (here being `text == 'yes'`) is sufficient, and the result will be returned.

.apply()

```
df['alive'] = df['alive'].apply(alive_bool)
df['alive'].value_counts()
```

```
False    891
Name: alive, dtype: int64
```

- We used the `.apply()` method to call the `alive_bool` function with the value in the `alive` column as the argument

APPLYING FUNCTIONS WITH MULTIPLE ARGUMENTS

Here's a more advanced example where we call `.apply()` on multiple columns at the same time. We will compare `pclass` and `class` to see if they are giving the same information:

```
def check_class_cols(row):

    class_val = row["pclass"]
    class_text = row["class"]

    class_dict = {1: 'First', 2: 'Second', 3: 'Third'}

    return class_dict[class_val] == class_text
```

```
df['class_check'] = df[['pclass', 'class']].apply(check_class_cols, axis=1)
```

- Here the function receives a row with multiple columns instead of just one value
- You can access the columns in your function and return values accordingly
- `axis=1` is used to apply the function to each row of the DataFrame

```
df['class_check'].sum() == len(df)
```

```
True
```

- We can use the `.sum()` method on a boolean column to count the number of `True` values
- We can find the number of rows in a DataFrame using the Python `len()` function
- The comparison operator `==` will return `True` if the compared values are equal

`lambda` FUNCTIONS

Our previous function `alive_bool` was quite simple, involving a single comparison of values. For cases like this, you could use a `lambda` function instead of defining a separate function for use with `.apply()`:

```
df['alive'].apply(lambda text: text == "yes")
```

- lambda (or anonymous) functions are small, temporary functions which do not require a name

If you prefer to have a single process for applying functions, it's fine to stick with defining your function sepatately and then calling it within `.apply()` as shown earlier.

Part 2: Filtering, sorting, and modifying DataFrames

intro-pandas-workbook.ipynb