# Pandas Cheat Sheet

More information can be found in the docs: https://pandas.pydata.org/docs/reference/index.html
https://matplotlib.org/stable/api/index.html

## MOCK DATAFRAME AND SETUP

Note that pandas, matplotlib and seaborn are not standard packages and need to be installed and imported. They are imported using common shorthand as follows:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

The running example in the cheat sheet will be a dataframe containing information about students. Later we will introduce a 'exams' dataframe to perform joins etc.

Mock data: Our mock data contains information on five students, namely their 'name', 'DOB', 'sex', 'postcode' and whether they are a 'current_student' (1 if yes, 0 if no).

```
student_data = {
    "name": ["Alice", "Bob", "Charlie",
         "David", "Eva"],
    "DOB": ["2001-05-14", "2000-08-30",
         "2002-12-11", "1999-07-22", "
        2003-01-19"],
    "sex": ["F", "M", "M", "M", "F"],
    "postcode": ["AA345", "AB456", "
        AC567", "AB678", "AB789"],
    "current_student": [1, 0, 1, 0, 1]
}
```

Our mock dataframe (when we create it in the next section) will be called 'students'.

For many objects/functions only some of the passable arguments will be discussed, the non-discussed ones will be replaced by '...' - see the documentation for a full description of all arguments.

## DATAFRAMES

Dataframes are tables in pandas. They are a pandas object given as follows:

```
class pd.DataFrame(data=None, index=
    None, ...)
```

1. data: usually a dictionary, but can also be ndarrays, lists of dictionaries/Series etc.
2. index: a list with index labels

Dataframes may also be obtained from '.csv' files or otherwise. This is done using

```
pd.read_csv(file_path, sep, index_col,
    names, ...)
```

1. file_path: A string containing the path to the file to be imported.
2. sep: Specify the delimiter, this is ',' by default. Passing, for example, '|' allows .tsv files to be read in.
3. index_col: Specify column(s) to become the (multi)index
4. names: Pass a list of column labels. To replace the current column laebel in the .csv pair the 'columns' argument with the argument 'header = 0'.

Example:

```
students = pd.DataFrame(student_data)
print(students)
```

Output:

```
      name         DOB sex postcode  current_student
0    Alice  2001-05-14   F    AA345                1
1      Bob  2000-08-30   M    AB456                0
2  Charlie  2002-12-11   M    AC567                1
3    David  1999-07-22   M    AB678                0
4      Eva  2003-01-19   F    AB789                1
```

## INSPECTING DATAFRAMES

There are many methods and attributes on a DataFrame object wich we can use to obtain information about it. We use a DataFrame object called 'df'.

| Method/attribute | Short description |
|---|---|
| df.columns | Index/list of column names |
| len(df) | number of rows |
| df.shape | tuple (#rows, #columns) |
| df.head(n=5) | displays first '$n'$ rows of dataframe |
| df.tail(n=5) | displays last '$n'$ rows of dataframe |
| df.info() | information about dataframe |
| df.dtypes | column names and their dtypes |

For example, `students.info()` gives output

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 5 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   name             5 non-null      object
 1   DOB              5 non-null      object
 2   sex              5 non-null      object
 3   postcode         5 non-null      object
 4   current_student  5 non-null      int64
dtypes: int64(1), object(4)
memory usage: 328.0+ bytes
None
```

and `students.dtypes` gives output

```
name               object
DOB                object
sex                object
postcode           object
current_student     int64
dtype: object
```

## BASIC DATAFRAME ANALYSIS

We can do basic analysis on dataframes, such as finding the minimum value in each column or finding the most common value in each column. We use a DataFrame object called 'df'.

The methods in the following table return a Series object (see next section for a discussion on Series).

| Method | Short description |
|---|---|
| df.min() | finds the minimum value from each column |
| df.max() | finds the maximum value from each column |
| df.sum() | returns sums of the columns. For strings this is concatenation, for numeric columns is regular addition. |
| df.count() | counts non-NaN values in each column |
| df.mean( numeric_only=True) | finds mean of numeric columns |
| df.median( numeric_only=True) | finds median of numeric columns |

The following two methods return DataFrame objects:

| Method | Short description |
|---|---|
| df.mode() | returns columns and their most common values |
| df.describe() | returns stats for each column such as mean, mode, median, percentiles. By default this is only for numeric columns, passing "include='all' " produces descriptions for all columns. |

E.g., students.describe(include='all') gives...

```
         name         DOB  sex postcode  current_student
count       5           5    5        5         5.000000
unique      5           5    2        5              NaN
top     Alice  2001-05-14    M    AA345              NaN
freq        1           1    3        1              NaN
mean      NaN         NaN  NaN      NaN         0.600000
std       NaN         NaN  NaN      NaN         0.547723
min       NaN         NaN  NaN      NaN         0.000000
25%       NaN         NaN  NaN      NaN         0.000000
50%       NaN         NaN  NaN      NaN         1.000000
75%       NaN         NaN  NaN      NaN         1.000000
max       NaN         NaN  NaN      NaN         1.000000
```

## SERIES & SELECTING COLUMNS

A series is a one-dimensional array with labels (roughly, it is an $n \times 1$ dataframe, but with slightly different behaviour). They are initialised using

```
pd.Series(data,index,...)
```

- Obtaining a Series as a column of a dataframe:

  We can extract a column of a dataframe, df, as a Series as follows:

  ```
  df[column_name]
  ```

  The index will be the index of the dataframe, the values will be the values in the column.

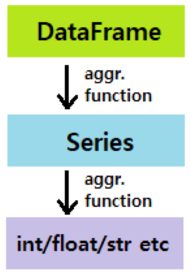
- Selecting multiple columns of a dataframe:

  To extract multiple columns of a dataframe we pass a list of column names, the resulting object will be a dataframe.

  ```
  df[[col_name_1, col_name_2,...]]
  ```

## BASIC SERIES ANALYSIS

Methods such as sum, min, max work on Series too. They return a int/float/string object. A flowchart to keep in mind is:



Some other methods and attributes of a Series, s, include:

- s.values - returns values of the Series as an array
- s.index - gives information about the index/labels
- Have head(), tail(), info(), describe() etc.
- s.unique() - array containing distinct values in the series
- s.nunique() - number of distinct values
- s.nlargest(n=5) - returns the 'n' largest values, by default 'n=5'
- s.nsmallest(n=5) - ...
- s.value_counts() - returns Series with labels given by values in the Series, s, and values given by number of occurrences.

For example, extracting the column 'sex' from our dataframe 'students' and printing value counts of the resulting Series can be done via:

```
s = students['sex']
print(s.value_counts())
```

which gives output:

```
sex
M    3
F    2
Name: count, dtype: int64
```

For dataframes we have

```
df.nlargest(n=5,columns)
```

where $n$ largest dataframe rows are returned based on values in the provided list of columns, 'columns'.

## INDEXING AND SORTING

To set the index (or labels) of a dataframe using a list of values or an existing column we use

```
df.set_index(keys,inplace = False)
```

where 'keys' is either an array (of the correct length) or a column name.

inplace - this optional argument replaces 'df' with the modified dataframe if set to True, and returns a new dataframe if set to False.

To sort rows by values in given column(s) we use

```
df.sort_values(by, ascending=True,
    inplace=False,key = None,...)
```

where 'by' is a column name or list of column names to sort by. Whether to sort in ascending or descending order is decided using the optional argument 'ascending'.

The argument 'key' allows us to pass a (vectorised) function which is applied to the sorting column before sorting.
For example, to sort our dataframe by length of the students name we use

```
students.sort_values('name', key =
    lambda x: x.str.len())
```

which returns the dataframe

```
      name         DOB sex postcode  current_student
1      Bob  2000-08-30   M    AB456                0
4      Eva  2003-01-19   F    AB789                1
0    Alice  2001-05-14   F    AA345                1
3    David  1999-07-22   M    AB678                0
2  Charlie  2002-12-11   M    AC567                1
```

To sort by index we can use

```
df.sort_index(ascending = True, inplace
     = False, ...)
```

Both of these sorts work for Series too, but of course there is no column name to pass to the 'sort_values' method.

## ACCESSING ROWS

To access rows of a dataframe or Series, we use 'loc'. To retrieve row(s) of a dataframe based on index name(s) we use

```
df.loc[index_names]
```

where index_names is either a single index name or a list of index names. This returns either a Series (if a single index name is passed) or a slice of the dataframe (if a list is passed).

We can also pass slices, boolean arrays (of the correct length), boolean Series to loc. There is lots of flexibility, see the docs for more.

We can also extract the $n^{th}$ row using '.iloc', e.g. to extract the $5^{th}$ row of a dataframe, df, we use

```
df.iloc[4] #indexing starts from 0.
```

Additionally, we can pass a second list consisting of column names, so we do not have to extract all columns for the specified rows.

## FILTERING DATAFRAMES

We can create a boolean series (a Series consisting of True/False values) by applying a comparison operator to a Series. For example

- equals (==):
  e.g. `df[column_name] == some_value`
- not equal (!=)
- less, greater, leq, geq (<, >, <=, >=)
- `.between(left,right)` (inclusive by default):
  e.g. `df[column_name].between(5,10)`
- `.isin(value: list-like)`:
  e.g. `df[column_name].isin(lst_of_values)`
- `.isna()`, `.notna()` (check if values are null, not null): e.g. `df[column_name].isna()`

This can also be referred to as a (boolean) mask. If boolean_series is our mask then we extract the rows evaluating to True using

```
df[boolean_series]
```

This has the same behavior as

```
df.loc[boolean_series].
```

To extract columns based on Trues within a boolean list (boolean_list) we use

```
df.loc[:,boolean_list].
```

The ':' means 'select all rows' and boolean_list is now targeting columns (since it is the second argument in loc).

We can also extract rows using a boolean list (ensuring the list is of the correct length i.e., the same length as the number of rows).

We combine conditions using AND (&) or OR (|). Boolean Series are negated using tilde (∼).

For example, to print rows containing females who are current students from our 'students' dataframe, we use:

```
mask = (students['sex'] == 'F') & (
    students['current_student'] == 1)
print(students[mask])
```

Output:

```
    name         DOB sex postcode  current_student
0  Alice  2001-05-14   F    AA345                1
4    Eva  2003-01-19   F    AB789                1
```

## ADDING AND REMOVING DATA

To add a column called 'new_column_name' to a dataframe we use

```
df['new_column_name'] = values
```

where values could be a constant, a list-like object, a dict, a Series etc.
In particular, we could create new columns using functions on old ones (including basic arithmetic/string functions).
e.g. `df[new_doubled_column] = df[old_column]*2`

To remove rows or columns we use

```
df.drop(index=None, columns=None,
    inplace=False,...)
```

where 'index' is a list of index names to be dropped, and 'columns' is a list of column names to be dropped.

## UPDATING VALUES

Firstly, we can update a value if we know its exact location:

```
df.loc[its_index,its_column] = new_
     value
```

To replace dynamically we can use the replace method:

```
df[column_name].replace(to_replace=None
     , value=None, inplace=False,...)
```

where ‘to_replace’ should be a value/list of values/regex expression etc, and ‘value’, gives the replacement(s).
Alternatively, ‘to_replace’ can be a dict object, then ‘value’ should be left blank.

To rename indices or column headers we use:

```
df.rename(index=None, columns=None,
     inplace=False,...)
```

where ‘index’ and/or ‘columns’ should be a mapper (such as a dict object) or a function.

## DATATYPES

The catchall dataype is ‘object’. It takes up lots of space. The (Series) method

```
.astype()
```

can be used to convert the type of a column of a dataframe. There is no ‘inplace’ here, so to update the datatype we need to do:

```
df[column_name] = df[column_name].
     astype(dtype)
```

Examples of dtypes include:

- int, float
- bool
- string
- datetime, timedelta
- category (this is good for columns with specific set of values it can take e.g. `[0,1]` for the current_student column for our ‘students’ dateframe.

There also exists a pandas method to convert values to numeric datatypes,

```
pd.to_numeric(arg,errors,...)
```

The argument can be a scalar, list-like, Series (e.g. a dataframe column). ‘errors’ tells pandas how to deal with values which cannot be converted, either ‘raise’ or ‘coerce’.

## WORKING WITH NULL VALUES

We briefly discuss three methods to deal with null values.

- `.isna()` - a method on Series and Dataframes. Returns a Series or Dataframe with True if null and False otherwise.
- `.dropna()` - for Series, creates a new Series with null values removed. For Dataframes it removes rows with at least one null value, but parameters can customise its behavior. See docs.
- `.fillna(value, inplace,...)` - fills null values with a given value, also a mapper can be passed. See docs.

## DATES AND TIMES

To convert columns to a date or time we can use the pandas method

```
pd.to_datetime(arg,dayfirst=False,
     yearfirst=False, format = None
     ,...)
```

The argument ‘arg’ can be str, list, Series, etc. The method automatically parses a lot of datatime formats. However, can specify dayfirst, yearfirst or pass a format e.g. `"%d/%b/%y"`. Search "Python date format codes" or similar for info on this.

To parse a column of a dataframe we use

```
df[column_name] = pd.to_datetime(df[
     column_name])
```

We access properties of datetime objects using .dt. Some examples are:

- `.dt.{year,month,hour,day,minute,second}` - extracts year,month,...,second from the datetime object.
- `.dt.dayofweek` - (Monday = 0,Sunday = 6).
- `.dt.month_name()`

We can do comparisons using datetimes and their properties.
We can add and subtract datetimes, which result in timedelta objects. Properties of timedelta objects are obtained using `.dt`..

## GROUPING

We can group rows in the table based on the values in given columns via

```
df.groupby(by = None,...)
```

where ‘by’ should be a column name or a list of column names.
Additionally, one can pass a function applied to a column. e.g. if grouping by the first letter of a name we can pass `df.['name'].str[0]` directly to the `.groupby()` method.

This returns a pd.Dataframe.groupby object (let ‘gbo’ be an arbitrary such object).

Some methods and attributes of groupby objects:

- `gbo.ngroups` - returns the number of groups.
- `gbo.groups` - returns dictionary with groups and indices within each group.
- `gbo.first()` - returns dataframe (with the grouped column as index) with the first row from each group.
- `gbo.get_group(name)` - returns the rows (as a dataframe) in the group with the given name.

For example, the code

```
gbo = students.groupby('sex')
print(gbo.ngroups)
print(gbo.groups)
```

gives output

```
2
{'F': [0, 4], 'M': [1, 2, 3]}
```

It is possible to iterate over a gbo:

```
for group_name, group_rows in
     gbo:
    #do something
```

where `group_rows` has value `gbo.get_group(group_name)`.

There are methods we can apply to groupby objects. We can apply them to grouped dataframe (e.g. `df.groupby(group_column)`) or select a specific column first and apply function to the grouped Series (i.e. `df.groupby(group_column)[column_name]`):

- `gbo.size()` - returns Series with the size of each group (this includes NaN values).
- `gbo.count()` - counts number of non-NaN values in each column from each group.
- `gbo.describe()` - gives some statistics of numerical columns for each group.
- `gbo.mean()`, `gbo.median()` - should pass ‘numeric_only=True’ to both the mean and median methods.

The method `.agg(func)` can be used to apply more complicated functions, or apply different functions to different columns.

- `gbo.agg('min')` - same as `gbo.min()`.
- `gbo.agg(['min','max'])` - applies min and max to all (numeric) columns.
- Alternatively, can pass dictionary to apply different functions to different columns e.g. `gbo.agg({column_1: "mean", column2: "min"})`
- can pass custom functions too, these functions should accept a Series/dataframe as the argument
- To name the resulting columns we use ‘`pd.NamedAgg`’ e.g. `gbo.agg(named_column = pd.NamedAgg(column_name,func))`

For example, the code

```
gbo = students.groupby('sex')
print(gbo.agg({'name': 'min','current_
     student': ['mean','median']}))
```

gives output

```
     name current_student
      min            mean median
sex
F    Alice       1.000000    1.0
M      Bob       0.333333    0.0
```

In the above example we have two layers of column labels, namely level 0 for the original column name, and level 1 for the name of the function being applied. This is known as hierarchical labelling.

## HIERARCHICAL INDEXING

Hierarchical (or multi-level) indexing can occur if dataframe is grouped by multiple columns. For example:

```
df.set_index([column_name,another_
     column_name], inplace=True)
```

Here, `df.index` returns `MultiIndex` (indices are tuples).
Sorting rows using multi-level indices is done as follows:

- `df.sort_index(level=None)` - passing `level=n` sorts by the index at level $n$.
- `df.sort_index(level,ascending)` - can pass list of levels and 'ascending' list (of same length) of True/Falses.

Can access rows using '.loc[]' as follows:

- `df.loc[level_0_index_value]` - access rows using the level 0 index.
- `df.loc[tuple_of_values]` - access rows using index tuple
- `df.loc[:,level_1_index_value,:]` - access by the level 1 index, etc.
- The cross section method, `.xs(..)`, exists for more cumbersome accessing (i.e. if there are many levels).

> Using group by and aggregate functions can produce hierarchical columns. These are accessed using tuples, e.g. `df[('level_0_heading', 'level_1_heading')]`.

We can strip and produce multi-level indices using stack/unstack:

- unstack - used to convert level of multi-index to a column e.g. `df.unstack(level=n)`
- `df.stack()` - convert $(m \times n)$ dataframe to a $(m \times n, 1)$ dataframe by converting columns into multi-index.

To group by indices of a certain level we use

```
df.groupby(level=n)
```

Can also pass a list of levels to 'level', or alternatively can pass index names.

## WORKING WITH STRINGS

Recall that we can convert textual columns from 'object' type to 'string' type via

```
df[column_name] = df[column_name].
     astype("string")
```

For this section let '$s$' be a Series object with textual values (either type object or string, it doesn't matter), for example, a column of a dataframe.
To apply string methods to a Series, s, apply them using `s.str`.

Some examples are:

- `s.str.upper()` - all caps
- `s.str.lower()` - all lower case
- `s.str[n]` - the $n^{th}$ character of each string
- `s.str.strip(to_strip = None)` - strips leading and trailing whitespace by default, if string of characters passed using 'to_strip' then these are striped instead. There also exists lstrip, rstrip.
- `s.str.split(pat=None, expand=False)` - splits on spaces (or passed characters 'pat'). This method produces Series where values are lists. If 'expand=True' then it creates a dataframe by expanding the splitting lists.
- `s.str.replace(to_replace, value)` - 'to_replace' and 'value' can be str, regex, list of strings, dict (if dict then 'value' shouldn't be provided) among others.

For example, suppose we wished to count the number of students (past and present) from each postcode area (i.e. 'AA', 'AB' or 'AC'). To do so we can apply a string function to the postcode, then group, then find the size.

```
gbo = students.groupby(students['
     postcode'].str[:2])
print(gbo.size())
```

with output

```
postcode
AA    1
AB    3
AC    1
dtype: int64
```

## APPLY, MAP, APPLYMAP

To apply a (non-vectorised) function to a Series we use the pandas.Series method

```
pd.Series.apply(func,arg=()).
```

If the function needs additio.nal arguments, these should be passed using 'arg'. This returns a new Series with values given by the function applied (elementwise/rowwise) to the original values.

We can also apply maps to Series using

```
pd.Series.map(arg)
```

For example, given Series, s, with values $1, 2, 3$ and wishing to change to $1^{st}$, $2^{nd}$, $3^{rd}$ can be done using

```
s.map({1:'1st',2:'2nd',3:'3rd'})
```

To apply a function elementwise to every value in a dataframe (and return a new dataframe) we use

```
df.applymap(func)
```

## JOINING SERIES & DATAFRAMES

To join series together we use the pandas' concat method:

```
pd.concat(objs,axis,join,ignore_index)
```

where 'objs' should be a list of Series, 'axis = 0' means stack vertically and 'axis =1' means stack horizionally (i.e. join by index). Passing 'ignore_index=True' creates a new index for the combined Series. When joining series by index (i.e. 'axis =1') pandas inserts NaN if a passed Series doesn't have a particular index. This is an outer join (the default). To include only rows where all series have the index pass 'join="inner"'.

As concat as a pandas method, concatenating dataframes works in much the same way. (Still have objs, axis, join that we can pass).

The last join to discuss is 'merge'. This is an SQL-like join method, where we join on specific columns:

```
df.merge(right,how,on,...)
```

where 'right' is another dataframe, 'how' is the join type, that is, 'how' is one of left, right, outer, inner, cross. 'on' is a column name or list of column names to join on.

Examples of merges.

Let 'exams' be another dataframe as follows:

```
    name  subject  score
0  Alice    maths     80
1  Alice  english     85
2    Bob    maths     75
3  David  science     60
```

An inner join of 'students' and 'exams' is done as follows

```
print(students.merge(exams,'inner','
     name'))
```

with output

```
     name        current_student  subject  score
0  Alice                       1    maths     80
1  Alice   ...                 1  english     85
2    Bob                       0    maths     75
3  David                       0  science     60
```

Note that both 'Charlie' and 'Eva' are missing, as they had no exam scores. If we want them to be included we can use an outer join:

```
print(students.merge(exams,'outer','
     name'))
```

with output

```
       name        current_student  subject  score
0     Alice                      1    maths   80.0
1     Alice                      1  english   85.0
2       Bob   ...                0    maths   75.0
3   Charlie                      1      NaN    NaN
4     David                      0  science   60.0
5       Eva                      1      NaN    NaN
```

Roughly, the different types of merges can be compared to different operations on two sets $A$ and $B$:

- left - corresponds to $A$. i.e. all of $A$ is included, and elements of $B$ which happen to be in the intersection are also included, elements in $B \setminus A$ are discarded.
- right - corresponds to $B$. Analogous to above with roles of $A$ and $B$ switched.
- inner - like the intersection $A \cap B$
- outer - like the union $A \cup B$
- cross - like the cartesian product $A \times B$.

In our example above, the exams.name column (the set $B$) is a proper subset of the students.name column ( the set $A$). Therefore, 'left' and 'outer' will behave the same, and 'right' and 'inner' will behave the same.

## MATPLOTLIB.PYPLOT

All plotting functionality in pandas uses matplotlib behind the scenes, so it's useful to know some matplotlib (specifically matplotlib.pyplot). It's imported as follows

```
import matplotlib.pyplot as plt
```

The first method to look at is

```
plt.plot()
```

This produces an empty plot as is. can pass lists, Series, two lists (first is x-axis, second is y-axis) and many kwargs such as:

- color: e.g. 'red', 'blue', '#[hex]', or any ccs recognised colour.
- linewidth: float
- linestyle: e.g. '-', '- -', '-.'. 'dashed' ,'dotted'
- marker - marks the plotted line with a mark, e.g. '<', '>', '∗'.
- label: a label for the plot

Using `plt.plot(...)` will not display anything, to do this we must use

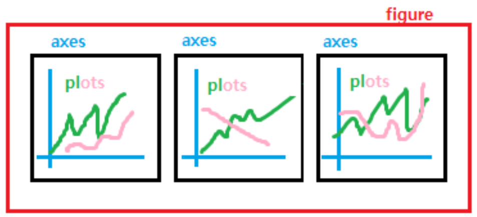
```
plt.show()
```

which displays all active figures.

Tip: Within numpy we have the arange() method which can produce evenly spaced values within an interval. This is good for producing x-values for a function, for example.

Figures, axes and plots.

In matplotlib, a 'plot' (the lines on a graph) lives on an 'axis'. These axes in turn are placed in a 'figure'. A diagram showing this is as follows:



So when we type `plt.plot(...)` matplotlib implicity creates a figure and an axes for us.

```
plt.plot(...)
plt.plot(...)
plt.show()
```

This code creates two plots on the same (implicitly created) axis contained in a (implicated created) figure. Whereas the code

```
plt.plot(...)
plt.figure(figsize=None, dpi=None,...)
plt.plot(...)
plt.show()
```

places the first plot on the first axis in the first figure (all implicit), with the second plot being placed on a new (implicitly created) axes within the newly created figure.
We may pass a 'figsize' and a 'dpi' to change the size of the figure. There are many other arguments to customise it, such as creating frames, colour, a label for referencing it.

We may change the styles of our plots using stylesheets:

```
plt.style.use(style:str)
```

We can choose from in-built styles (a list may be obtained via `plt.style.available`) or a custom one.

To give our axes titles we use

```
plt.title(label:str, loc: {'center', '
    left', 'right'})
```

where 'label' is the title, and 'loc' is the location. This method should be used within the scope of the target axis.

To set the x-label, y-label, x-ticks and/or y-ticks of an axis we use

```
plt.xlabel(xlabel: str)
plt.ylabel(ylabel: str)
plt.xticks(ticks, labels)
plt.yticks(ticks,labels)
```

where 'ticks' is a list of tick locations (e.g. a list of integers) and 'labels' is a list of labels to place at the tick locations.

If we have many plots on an axes, we may wish to use a legend to describe our plots, this is done via

```
plt.legend(loc = 'best',...)
```

This makes use of plot labels, the optional argument 'loc' is used to decide the location.

Some types of plots available are:

- Bar plots:
  `plt.bar(x,height)` - 'x' and 'height' are array-like (e.g. lists) giving x positions and heights of bars respectively.

  The title, x-labels, y-labels etc are all set as they were with `plt.plot(...)` This is the case of all plots in this list (and beyond).

  Two sets of data can be stacked by setting 'bottom' of second bar plot to be the heights of the bars of the first bar plot:

  ```
  plt.bar(height = height1)
  plt.bar(height=height2, bottom
      =height1)
  ```
- Horizontal bar plots:
  use .barh() instead of .bar().
- Scatter plots:
  `plt.scatter(x,y)`
- Pie plots:
  `plt.scatter(x,labels)` - 'x' are the wedge sizes, 'labels' are labels for the wedges.

To create subplots (multiple axes in the same figure) we use `plt.subplot(nrows, ncols, index, label)` which adds an axis to the current figure. The argument'label' can be used to give a labelto the axis to be used later (e.g. to get a plot to plot itself on the labelled axis).

For example, to produce something like the diagram displayed under the heading "Figures, axes, and plots" we could do:

```
plt.figure(figsize=(10,4)) #landscape
plt.subplot(1,3,1) #1 row, 3 columns,
    index 1
plt.plot() #the leftmost plot.
plt.subplot(1,3,2) #index 2
plt.plot() #the center plot.
plt.subplot(1,3,3)
plt.plot() #the rightmost plot.
plt.show()
```

We can give each axis its own title and labels by working in the scope to the relevant subplot. Additionally, `plt.suptitle(t: str)` gives "super title" 't' to the figure.

To ensure uniformity between x or y limits, ticks and scale there are the optional arguments for subplots called **sharex**, **sharey**, which are of type 'Axes'.

We can save figures using `plt.savefig(file_name)`.

## PANDAS PLOTTING

Pandas plotting uses matplotlib behind the scenes. Throughout, let 'df' be a dataframe, and 's' be a series.

Plotting a dataframe or series can be done as follows:

```
s.plot(kind)
df.plot(x,y,kind)
```

where 'kind' is, for example, 'line', 'bar', 'hist', 'box', 'scatter' etc.
For dataframes, optional arguments 'x' and 'y' can be used to plot columns against each other, otherwise, all columns are plotted against the index on the same axis.

Changing styles is done using matplotlib (see previous section).

Some other arguments of {s,df}.plot():

- ax: pass a matplotlib axes object to place plot on this axis.
- title, xlabel, ylabel, sticks, yticks.

We can also use plt methods instead, e.g. we can use `plt.title()` instead of passing the 'title' argument to the pandas plot. This allows for more customisation (changing font size, spacing, font colour etc). We can also mix and match, i..e use arguments for some customisations and plt methods for others.

Subplots

For dataframes, `df.plot()` accepts 'subplots=True' which creates a plot on a different axis for each column. 'layout:tuple' can also be passed to give the layout for the axes.

We can also "manually" procude subplots by first creating an array of axes using plt.subplots

```
fig, axs = plt.subplot(nrows, ncols)
```

and then plot on different axes by passing axis index (using 'ax') to the plots, e.g

```
s.plot(...,ax=axs[0]) #plots in first
    position
t.plot(...,ax=axs[1]) #plots in second
    position
```

This is useful as we can now customise each axes serpaertely using plt methods, e.g. `axs[0].set_title(...)` to change the title of the first axes only.