# Unit 8: Handling data with pandas

Richard Foltyn

November 27, 2020

## Contents



# 1 Handling data with pandas

## 1.1 Motivation

So far, we have encountered NumPy arrays as the only way to store numerical data (we mostly ignored the built-in contains provided directly in Python).

However, while NumPy arrays are great for storing homogenous data which does not have any particular structure, they are somewhat limited when we want to use them for high-level data analysis.

For example, we usually want to process data sets with

1. several variables;
2. multiple observations, which need not be identical across variables (some values may be missing);
3. non-homogenous data types: for examples, names need to be stored as strings, birthdays as dates and income as a floating-point number.

While NumPy can in principle handle such situations, it puts all the burden on the user. Most users would prefer to not have to deal with such low-level details.

Imagine we want to store names, birth dates and annual income for two people:

| Name | Date of birth | Income |
|---|---|---|
| Alice | 1985-01-01 | 30,000 |
| Bob | 1997-05-12 | - |

No income was reported for Bob, so it's missing. With NumPy, we could do this as follows:

[1]:
```
import numpy as np
from datetime import date

date1 = date(1985, 1, 1)          # birth date for Alice
date2 = date(1997, 5, 12)         # birth date for Bob
```

```
data = np.array([['Alice', date1, 30000.0],
                 ['Bob', date2, None]])

data
```

[1]:
```
array([['Alice', datetime.date(1985, 1, 1), 30000.0],
       ['Bob', datetime.date(1997, 5, 12), None]], dtype=object)
```

[2]:
```
data.dtype          # print array data type
```

[2]:
```
dtype('O')
```

While we can create such arrays, they are almost useless for data analysis, in particular since everything is stored as a generic `object`.

- To be fair, NumPy offers an alternative array type called "record" or "structured" array which can handle fields of different data types.

  However, the pandas library offers much more beyond that, so there is little reason to use structured arrays.

*Pandas* was created to offer more versatile data structures that are straightforward to use for storing, manipulating and analysing heterogeneous data:

1. Data is clearly organised in *variables* and *observations*, similar to econometrics programs such as Stata.
2. Each variable is permitted to have a different data type.
3. We can use *labels* to select observations, instead of having to use a linear numerical index as with NumPy.

   We could, for example, index a data set using National Insurance Numbers.
4. Pandas offers many convenient data aggregation and reduction routines that can be applied to subsets of data.

   For example, we can easily group observations by city and compute average incomes.
5. Pandas also offers many convenient data import / export functions that go beyond what's in NumPy.

Should we be using pandas at all times, then? No!

- For low-level tasks where performance is essential, use NumPy.
- For homogenous data without any particular data structure, use NumPy.
- On the other hand, if data is heterogeneous, needs to be imported from an external data source and cleaned or transformed before performing computations, use pandas.

There are numerous tutorials on pandas on the internet, so we will keep this unit short and illustrate only the main concepts. Useful references to additional material include:

- The official user guide.
- The official pandas cheat sheet which nicely illustrates the most frequently used operations.
- The official API reference with details on every pandas object and function.
- There are numerous tutorials (including videos) available on the internet. See here for a list.

## 1.2 Creating pandas data structures

Pandas has two main data structures:

1. `Series` represents observations of a single variable.
2. `DataFrame` is a container for several variables. You can think of each individual column of a `DataFrame` as a `Series`, and each row represents one observation.

The easiest way to create a `Series` or `DataFrame` is to create them from pre-existing data.

To access pandas data structures and routines, we need to import them first. The near-universal convention is to make pandas available using the name `pd`:

[3]:
```
import pandas as pd
```

*Examples:*

We can create a `DataFrame` from a NumPy array:

[4]:
```
import numpy as np
import pandas as pd             # universal convention: import using pd
from numpy.random import default_rng

# Draw normally distributed data
rng = default_rng(123)
data = rng.normal(size=(10,3))

# Define variable (or column) names
varnames = ['A', 'B', 'C']

# Create pandas DataFrame
pd.DataFrame(data, columns=varnames)
```

[4]:
```
          A         B         C
0 -0.989121 -0.367787  1.287925
1  0.193974  0.920231  0.577104
2 -0.636464  0.541952 -0.316595
3 -0.322389  0.097167 -1.525930
4  1.192166 -0.671090  1.000269
5  0.136321  1.532033 -0.659969
6 -0.311795  0.337769 -2.207471
7  0.827921  1.541630  1.126807
8  0.754770 -0.145978  1.281902
9  1.074031  0.392621  0.005114
```

This code creates a `DataFrame` of three variables called `A`, `B` and `C` with 10 observations each.

Alternatively, we can create a `DataFrame` from non-homogenous data as follows:

[5]:
```
# Names (strings)
names = ['Alice', 'Bob']

# Birth dates (datetime objects)
bdates = pd.to_datetime(['1985-01-01', '1997-05-12'])

# Incomes (floats)
incomes = np.array([35000, np.nan])         # code missing income as NaN

# create DataFrame from dictionary
pd.DataFrame({'Name': names, 'Birthdate': bdates, 'Income': incomes})
```

[5]:
```
    Name  Birthdate   Income
0  Alice 1985-01-01  35000.0
1    Bob 1997-05-12      NaN
```

If data types differ across columns, as in the above example, it is often convenient to create the `DataFrame` by passing a dictionary as an argument. Each key represents a column name and each corresponding value contains the data for that variable.

### 1.3 Viewing data

With large data sets, you hardly ever want to print the entire `DataFrame`. Pandas by default limits the amount of data shown. You can use the `head()` and `tail()` methods to explicitly display a specific number of rows from the top or the end of a `DataFrame`.

To illustrate, we use a data set of a few UK universities that contains their name, number of students and budget in million pounds (both from Wikipedia), and their Times Higher Education (THE) ranking.

We read in the data stored in the file `universities.csv` (from the `data/` folder) like this:

[6]:
```
import pandas as pd

# relative path to CSV file
file = '../data/universities.csv'

# Load sample data set of UK universities
df = pd.read_csv(file, sep=';')
```

We can now display the first and last three rows:

[7]:
```
df.head(3)        # show first three rows
```

[7]:
```
                    Institution   Country  Founded  Students  Budget  Rank
0      University of Glasgow   Scotland     1451     30805   626.5    92
1    University of Edinburgh   Scotland     1583     34275  1102.0    30
2  University of St Andrews   Scotland     1413      8984   251.2   201
```

[8]:
```
df.tail(3)        # show last three rows
```

[8]:
```
                      Institution            Country  Founded  Students  Budget  \
20        University of Stirling           Scotland     1967      9548   113.3
21  Queen’s University Belfast   Northern Ireland     1810     18438   369.2
22             Swansea University              Wales     1920     20620     NaN

    Rank
20   301
21   200
22   251
```

To quickly compute some descriptive statistics for the *numerical* variables in the `DataFrame`, we use `describe()`:

[9]:
```
df.describe()
```

[9]:
```
           Founded      Students       Budget        Rank
count    23.000000     23.000000    22.000000   23.000000
mean   1745.652174  24106.782609   768.609091  124.739130
std     256.992149   9093.000735   608.234948  104.498463
min    1096.000000   8984.000000   113.300000    1.000000
25%    1589.000000  18776.500000   340.850000   32.500000
50%    1826.000000  23247.000000   643.750000  107.000000
75%    1941.500000  30801.500000  1023.500000  195.500000
max    2004.000000  41180.000000  2450.000000  401.000000
```

Note that this automatically ignores the columns `Institution` and `Country` as they contain strings and computing the mean, etc. of a string variable does not make sense.

To see low-level information about the data type used in each column, we call `info()`:

[10]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 23 entries, 0 to 22
```

```
Data columns (total 6 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   Institution  23 non-null     object
 1   Country      23 non-null     object
 2   Founded      23 non-null     int64
 3   Students     23 non-null     int64
 4   Budget       22 non-null     float64
 5   Rank         23 non-null     int64
dtypes: float64(1), int64(3), object(2)
memory usage: 1.2+ KB
```

Note that pandas automatically discards missing information in computations. For example, the budget for Swansea University is missing, and this observation is therefore dropped when computing descriptive statistics. Consequently, you see in the above table that the column `Budget` contains only 22 *non-null* entries.

## 1.4 Indexing

Pandas supports two types of indexing:

1. Indexing by position. This is basically identical to the indexing of other Python and NumPy containers.
2. Indexing by label, ie. by the values assigned to the row or column index. These labels need not be integers in increasing order, as is the case for NumPy.

   We will see how to assign labels below.

Pandas indexing is performed either by using brackets `[]`, or by using `.loc[]` for label indexing, or `.iloc[]` for positional indexing.

Indexing via `[]` can be somewhat confusing:

- specifying `df['name']` returns the column `name` as a `Series` object.
- On the other hand, specifying a range such as `df[5:10]` returns the *rows* associated with the *positions* 5,. . . ,9.

*Examples:*

[11]:
```
import pandas as pd

# Load sample data set of UK universities
df = pd.read_csv('../data/universities.csv', sep=';')
df['Institution']                # select a single column
```

[11]:
```
0                University of Glasgow
1              University of Edinburgh
2             University of St Andrews
3               University of Aberdeen
4            University of Strathclyde
5                                  LSE
6                                  UCL
7              University of Cambridge
8                 University of Oxford
9                University of Warwick
10             Imperial College London
11               King's College London
12            University of Manchester
13               University of Bristol
14            University of Birmingham
15     Queen Mary University of London
16                  University of York
```

```
17           University of Nottingham
18               University of Dundee
19                 Cardiff University
20             University of Stirling
21         Queen's University Belfast
22                 Swansea University
Name: Institution, dtype: object
```

[12]:
```
df[['Institution', 'Rank']]      # select multiple columns using a list
```

[12]:
```
                       Institution  Rank
0            University of Glasgow    92
1          University of Edinburgh    30
2         University of St Andrews   201
3           University of Aberdeen   178
4        University of Strathclyde   401
5                              LSE    27
6                              UCL    16
7          University of Cambridge     6
8             University of Oxford     1
9            University of Warwick    77
10         Imperial College London    11
11           King's College London    35
12        University of Manchester    51
13           University of Bristol    91
14        University of Birmingham   107
15  Queen Mary University of London   110
16              University of York   133
17        University of Nottingham   158
18            University of Dundee   201
19              Cardiff University   191
20          University of Stirling   301
21      Queen's University Belfast   200
22              Swansea University   251
```

To return the rows at positions 1, 2 and 3 we use

[13]:
```
df[1:4]
```

[13]:
```
                Institution   Country  Founded  Students  Budget  Rank
1   University of Edinburgh  Scotland     1583     34275  1102.0    30
2  University of St Andrews  Scotland     1413      8984   251.2   201
3    University of Aberdeen  Scotland     1495     14775   219.5   178
```

Pandas follows the Python convention that indices are 0-based, and the endpoint of a slice is not included.

### 1.4.1 Selection by label

Pandas uses *labels* to index and align data. These can be integer values starting at 0 with increments of 1 for each additional element, which is the default, but they need not be.

For example, we can replace the row index and use the Roman lower-case characters `a`, `b`, `c`, ... as labels instead of integers:

[14]:
```
import pandas as pd
df = pd.read_csv('../data/universities.csv', sep=';')

# Create list of lower-case letters which has same
# length as the number of observations.
index = [chr(97+i) for i in range(len(df))]      # len(df) returns number of obs.
index
```

```
[14]: ['a',
 'b',
 'c',
 'd',
 'e',
 'f',
 'g',
 'h',
 'i',
 'j',
 'k',
 'l',
 'm',
 'n',
 'o',
 'p',
 'q',
 'r',
 's',
 't',
 'u',
 'v',
 'w']
```

```
[15]: df['index'] = index                              # create new column 'index'
      df.set_index(keys=['index'], inplace=True)       # set letters as index!

      # print first 3 rows using labels
      df['a':'c']              # This is the same as df[:3]
```

```
[15]:                   Institution   Country  Founded  Students  Budget  Rank
      index
      a        University of Glasgow  Scotland     1451     30805   626.5    92
      b      University of Edinburgh  Scotland     1583     34275  1102.0    30
      c     University of St Andrews  Scotland     1413      8984   251.2   201
```

To add to the confusion, note that when specifying a range in terms of labels, the last element *is* included! Hence the row with index `c` in the above example is shown.

To more clearly distinguish between selection by label and by position, pandas provides the `.loc[]` and `.iloc[]` methods of indexing. To make your intention obvious, you should therefore adhere to the following rules:

1. Use `df['name']` only to select *columns* and nothing else.
2. Use `.loc[]` to select by label.
3. Use `.iloc[]` to select by position.

To illustrate, using `.loc[]` unambiguously indexes by label:

```
[16]: df.loc['d':'f', ['Institution', 'Students']]
```

```
[16]:                    Institution  Students
      index
      d        University of Aberdeen     14775
      e     University of Strathclyde     22640
      f                           LSE     11850
```

With `.loc[]` we can even perform slicing on column names, which is not possible with the simpler `df[]` syntax:

```
[17]: df.loc['d':'f', 'Institution':'Founded']
```

```
[17]:                      Institution   Country  Founded
index
d         University of Aberdeen  Scotland     1495
e      University of Strathclyde  Scotland     1964
f                            LSE   England     1895
```

This includes all the columns between `Institution` and `Founded`, where the latter is again included since we are slicing by label.

Trying to pass in positional arguments will return an error if the index does not happen to be of type integer and the given values are actual labels:

```
[18]: df.loc[0:4]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-11cc54301474> in <module>
----> 1 df.loc[0:4]

~/.conda/envs/py3-default/lib/python3.7/site-packages/pandas/core/indexing.py in␣
↪__getitem__(self, key)
    877
    878             maybe_callable = com.apply_if_callable(key, self.obj)
--> 879             return self._getitem_axis(maybe_callable, axis=axis)
    880
    881     def _is_scalar_access(self, key: Tuple):

~/.conda/envs/py3-default/lib/python3.7/site-packages/pandas/core/indexing.py in␣
↪_getitem_axis(self, key, axis)
   1086         if isinstance(key, slice):
   1087             self._validate_key(key, axis)
-> 1088             return self._get_slice_axis(key, axis=axis)
   1089         elif com.is_bool_indexer(key):
   1090             return self._getbool_axis(key, axis=axis)

~/.conda/envs/py3-default/lib/python3.7/site-packages/pandas/core/indexing.py in␣
↪_get_slice_axis(self, slice_obj, axis)
   1121         labels = obj._get_axis(axis)
   1122         indexer = labels.slice_indexer(
-> 1123             slice_obj.start, slice_obj.stop, slice_obj.step, kind="loc"
   1124         )
   1125

~/.conda/envs/py3-default/lib/python3.7/site-packages/pandas/core/indexes/base.py
↪in slice_indexer(self, start, end, step, kind)
   4964         slice(1, 3, None)
   4965         """
-> 4966         start_slice, end_slice = self.slice_locs(start, end, step=step,␣
↪kind=kind)
   4967
   4968         # return a slice

~/.conda/envs/py3-default/lib/python3.7/site-packages/pandas/core/indexes/base.py
↪in slice_locs(self, start, end, step, kind)
   5165         start_slice = None
   5166         if start is not None:
-> 5167             start_slice = self.get_slice_bound(start, "left", kind)
   5168         if start_slice is None:
   5169             start_slice = 0

~/.conda/envs/py3-default/lib/python3.7/site-packages/pandas/core/indexes/base.py
↪in get_slice_bound(self, label, side, kind)
```

```
   5077         # For datetime indices label may be a string that has to be␣
 ↪converted
   5078         # to datetime boundary according to its resolution.
-> 5079         label = self._maybe_cast_slice_bound(label, side, kind)
   5080
   5081         # we need to look up the label

~/.conda/envs/py3-default/lib/python3.7/site-packages/pandas/core/indexes/base.py␣
 ↪in _maybe_cast_slice_bound(self, label, side, kind)
   5029         # this is rejected (generally .loc gets you here)
   5030         elif is_integer(label):
-> 5031             self._invalid_indexer("slice", label)
   5032
   5033         return label

~/.conda/envs/py3-default/lib/python3.7/site-packages/pandas/core/indexes/base.py␣
 ↪in _invalid_indexer(self, form, key)
   3266         """
   3267         raise TypeError(
-> 3268             f"cannot do {form} indexing on {type(self).__name__} with␣
 ↪these "
   3269             f"indexers [{key}] of type {type(key).__name__}"
   3270         )

TypeError: cannot do slice indexing on Index with these indexers [0] of type int
```

Somewhat surprisingly, we can also pass boolean arrays to `.loc[]` even though these are clearly not labels:

[19]:
```
df.loc[df['Country'] == 'Scotland']
```

[19]:
```
                     Institution   Country  Founded  Students  Budget  Rank
index
a          University of Glasgow  Scotland     1451     30805   626.5    92
b        University of Edinburgh  Scotland     1583     34275  1102.0    30
c       University of St Andrews  Scotland     1413      8984   251.2   201
d         University of Aberdeen  Scotland     1495     14775   219.5   178
e      University of Strathclyde  Scotland     1964     22640   304.4   401
s           University of Dundee  Scotland     1967     15915   256.4   201
u         University of Stirling  Scotland     1967      9548   113.3   301
```

Indexing via `.loc[]` supports a few more types of arguments, see the official documentation for details.

### 1.4.2 Selection by position

Conversely, if we want to select items exclusively by their position and ignore their labels, we use `.iloc[]`:

[20]:
```
df.iloc[0:4, 0:2]            # select first 4 rows, first 2 columns
```

[20]:
```
                    Institution   Country
index
a         University of Glasgow  Scotland
b       University of Edinburgh  Scotland
c      University of St Andrews  Scotland
d        University of Aberdeen  Scotland
```

Again, `.iloc[]` supports a multitude of other arguments, including boolean arrays. See the official documentation for details.

## 1.5 Aggregation and reduction

### 1.5.1 Working with entire DataFrames

The simplest way to perform data reduction is to invoke the desired routine on the entire `DataFrame`:

[21]:
```
import pandas as pd

df = pd.read_csv('../data/universities.csv', sep=';')
df.mean()
```

[21]:
```
Founded       1745.652174
Students     24106.782609
Budget         768.609091
Rank           124.739130
dtype: float64
```

Methods such as `mean()` are by default applied column-wise to each numerical column.

One big advantage over NumPy is that missing values (represented by `np.nan`) are automatically ignored:

[22]:
```
# Set Aberdeen's THE ranking to missing
df.loc[df['Institution'] == 'University of Aberdeen', 'Rank'] = np.nan
df.loc[df['Institution'] == 'University of Aberdeen']
```

[22]:
```
              Institution   Country  Founded  Students  Budget  Rank
3  University of Aberdeen  Scotland     1495     14775   219.5   NaN
```

[23]:
```
# mean() automatically drops missing observations
df['Rank'].mean()
```

[23]:
```
122.31818181818181
```

### 1.5.2 Splitting and grouping

Applying aggregation functions to the entire `DataFrame` is similar to what we can do with NumPy. The added flexibility of pandas becomes obvious once we want to apply these functions to subsets of data, ie. groups, which we can define based on values or index labels.

For example, we can easily group our universities by country:

[24]:
```
import pandas as pd

df = pd.read_csv('../data/universities.csv', sep=';')

groups = df.groupby(['Country'])
```

Here `groups` is a special pandas objects which can subsequently be used to process group-specific data. To compute the group-wise averages, we can simply run

[25]:
```
groups.mean()
```

[25]:
```
                      Founded      Students       Budget        Rank
Country
England           1745.923077  27119.846154  1001.700000   63.307692
Northern Ireland  1810.000000  18438.000000   369.200000  200.000000
Scotland          1691.428571  19563.142857   410.471429  200.571429
Wales             1901.500000  23259.000000   644.800000  221.000000
```

Groups support column indexing: if we want to only compute the total number of students for each country in our sample, we can do this as follows:

[26]:
```
groups['Students'].sum()
```

[26]:
```
Country
England             352558
Northern Ireland     18438
Scotland            136942
Wales                46518
Name: Students, dtype: int64
```

There are numerous routines to aggregate grouped data, for example:

- `mean()`, `sum()`: averages and sums over numerical items within groups.
- `std()`, `var()`: within-group std. dev. and variances
- `size()`: group sizes
- `first()`, `last()`: first and last elements in each group
- `min()`, `max()`: minimum and maximum elements within a group

*Examples:*

[27]:
```
groups.size()          # return number of elements in each group
```

[27]:
```
Country
England             13
Northern Ireland     1
Scotland             7
Wales                2
dtype: int64
```

[28]:
```
groups.first()         # return first element in each group
```

[28]:
```
                                Institution  Founded  Students  Budget  Rank
Country
England                                 LSE     1895     11850   415.1    27
Northern Ireland  Queen's University Belfast     1810     18438   369.2   200
Scotland              University of Glasgow     1451     30805   626.5    92
Wales                    Cardiff University     1883     25898   644.8   191
```

We can create custom aggregation routines by calling `agg()` or `aggregate()` on the grouped object. To illustrate, we count the number of universities in each country that have more than 20,000 students:

[29]:
```
groups['Students'].agg(lambda x: np.sum(x >= 20000))
```

[29]:
```
Country
England             10
Northern Ireland     0
Scotland             3
Wales                2
Name: Students, dtype: int64
```

Note that we called `agg()` only on the column `Students`, otherwise the function would be applied to every column separately, which is not what we want.

The most flexible aggregation method is `apply()` which calls a given function, passing the entire group-specific subset of data (including all columns) as an argument, and glues together the results.

For example, if we want to compute the average budget per student (in pounds), we can do this as follows:

[30]:
```
# Budget is in millions of pounds, rescale by 1.0e6
groups.apply(lambda x: x['Budget'].sum() / x['Students'].sum() * 1.0e6)
```

[30]:
```
Country
England             36936.050239
```

```
Northern Ireland     20023.863760
Scotland             20981.875539
Wales                13861.301002
dtype: float64
```

We couldn't have done this with `agg()`, since `agg()` never gets to see the entire chunk of data but only one column at a time.

This section provided only a first look at pandas's "split-apply-combine" functionality implemented via `groupby`. See the official documentation for more details.

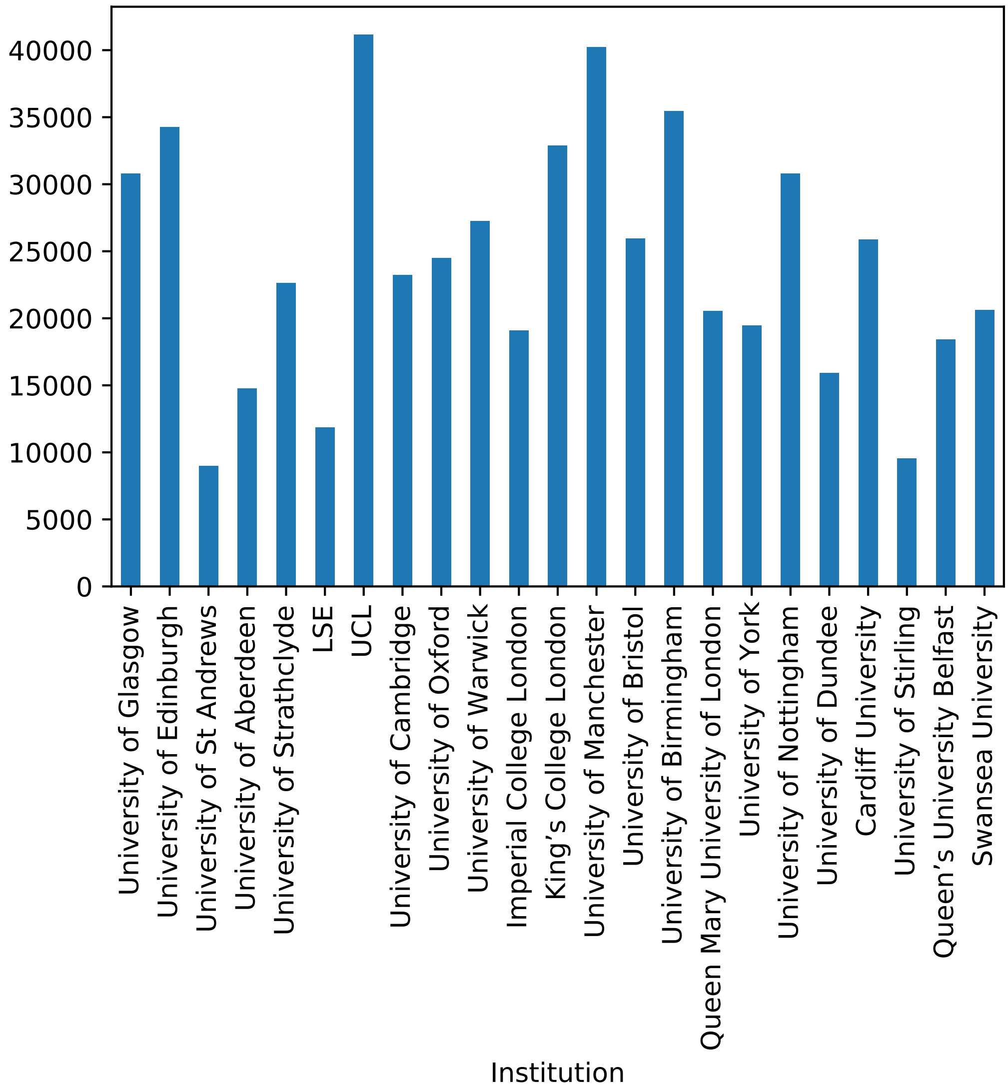
## 1.6 Visualisation

We covered plotting with Matplotlib in earlier units. Pandas itself implements some convenience wrappers around Matplotlib plotting routines which allow us to quickly inspect data stored in `DataFrames`. Alternatively, we can extract the numerical data and pass it to Matplotlib's routines manually.

For example, to plot student numbers as a bar chart, we can directly use pandas:

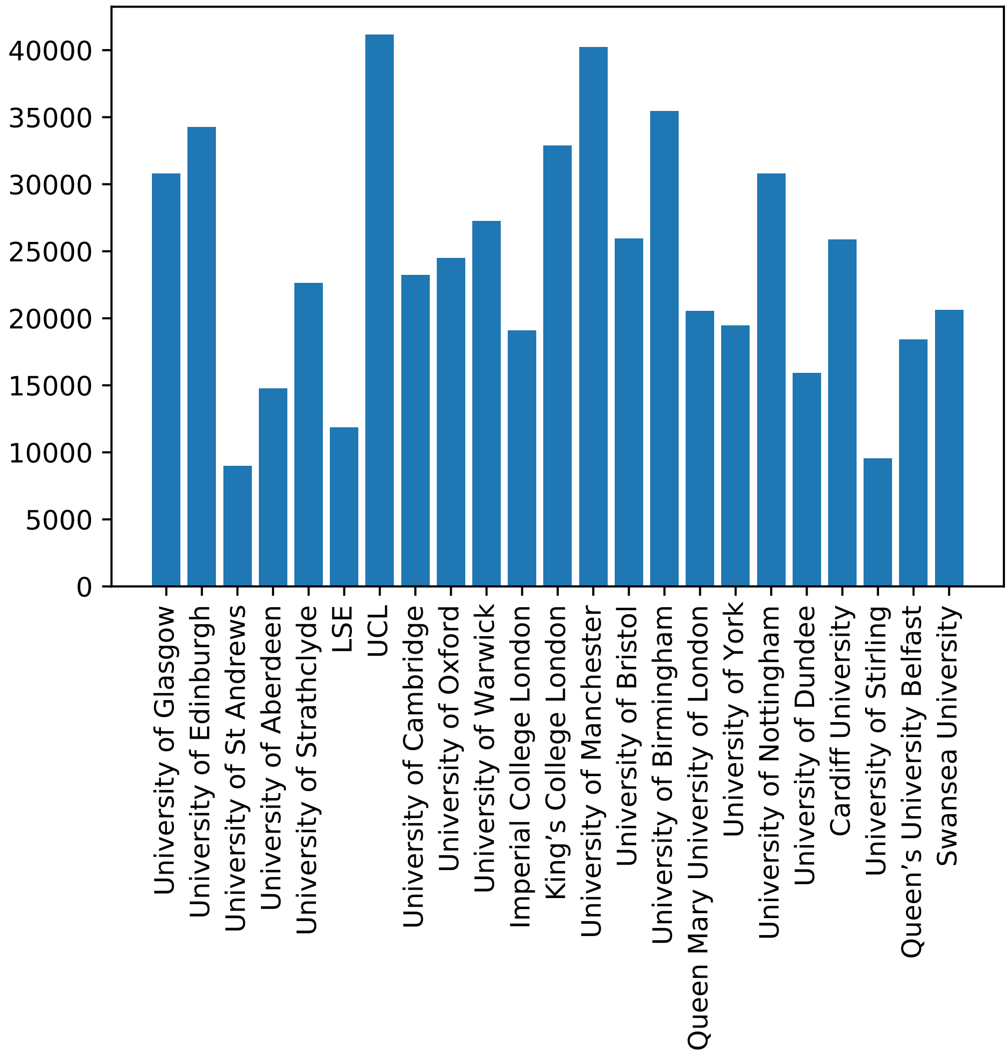
[31]:
```
import pandas as pd

df = pd.read_csv('../data/universities.csv', sep=';')

# set institution as label so they automatically show up in plot
df2 = df.set_index(keys=['Institution'])
df2['Students'].plot(kind='bar')
```

[31]:
```
<AxesSubplot:xlabel='Institution'>
```




Alternatively, we can construct the graph using Matplotlib ourselves:

[32]:
```
import matplotlib.pyplot as plt

labels = df['Institution'].to_list()        # labels as list
values = df['Students'].to_numpy()          # data as NumPy array
plt.bar(labels, values)
plt.tick_params(axis='x', labelrotation=90)
```

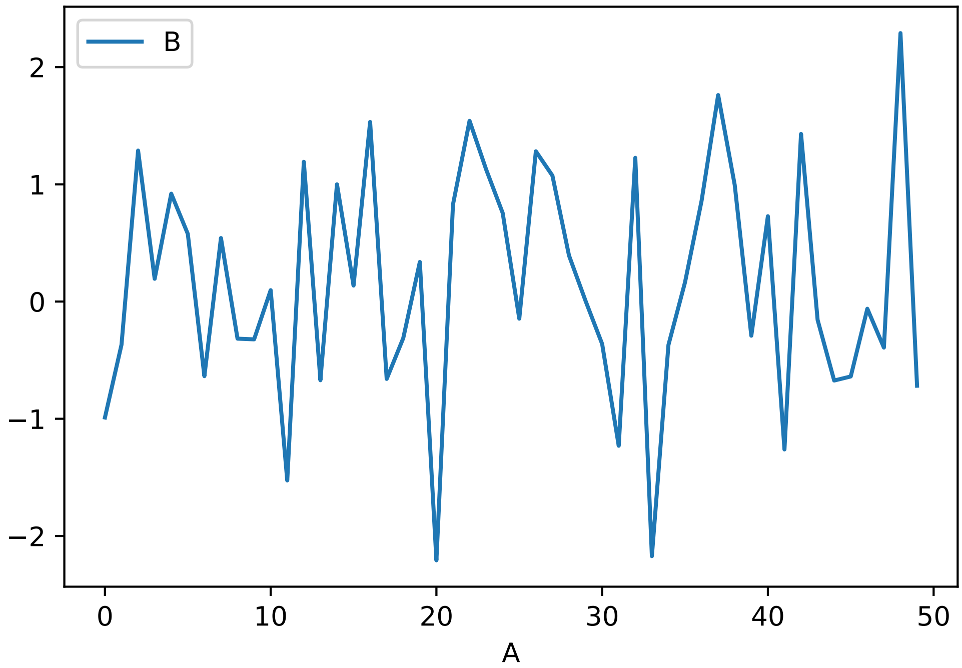



Sometimes Matplotlib's routines directly work with pandas's data structures, sometimes they don't. In cases where they don't, we can convert a `DataFrame` or `Series` object to a NumPy array using the `to_numpy()` method, and convert a `Series` to a Python list using `to_list()`, as illustrated in the example above.

To plot timeseries-like data, we can use the `plot()` method, which optionally accepts arguments to specify which columns should be used for the x-axis and which for the y-axis:

[33]:
```
import numpy as np
import pandas as pd

# Instantiate RNG
rng = np.random.default_rng(123)

# Create pandas DataFrame
nobs = 50
df = pd.DataFrame({'A': np.arange(nobs), 'B': rng.normal(size=nobs)})

df.plot(x='A', y='B')        # plot A on x-axis, B on y-axis
```

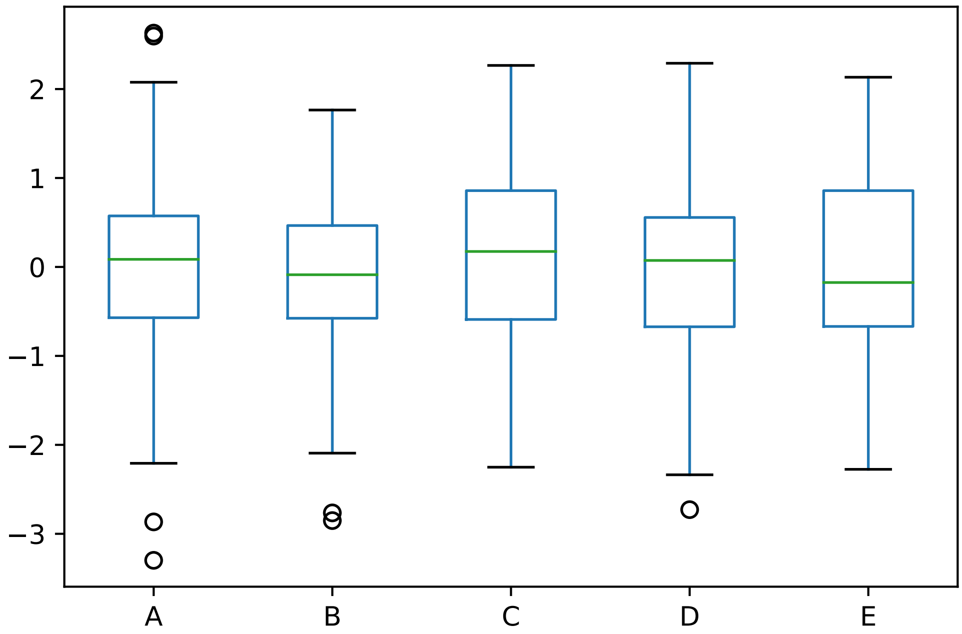
[33]:
```
<AxesSubplot:xlabel='A'>
```




To quickly generate some descriptive statistics, we can use the built-in box plot:

[34]:
```
import numpy as np
import pandas as pd

# Instantiate RNG
rng = np.random.default_rng(123)

# Create pandas DataFrame
df = pd.DataFrame(rng.normal(size=(100, 5)),
                  columns=['A', 'B', 'C', 'D', 'E'])

df.plot.box()            # same as df.plot(kind='box')
```

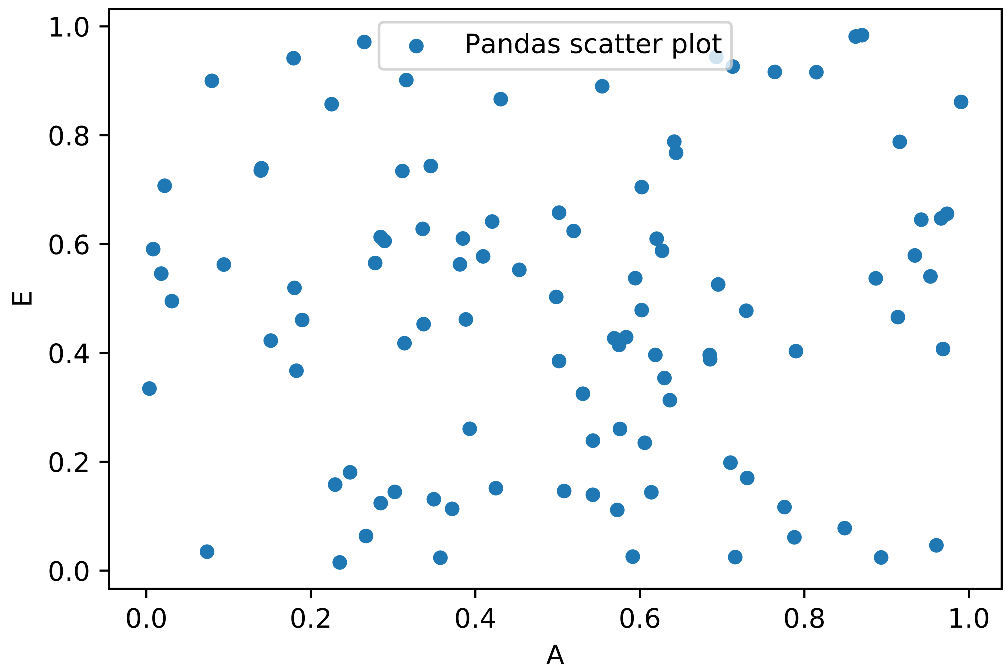
[34]: <AxesSubplot:>




Similarly, we can generate scatter plots, plotting one column against another:

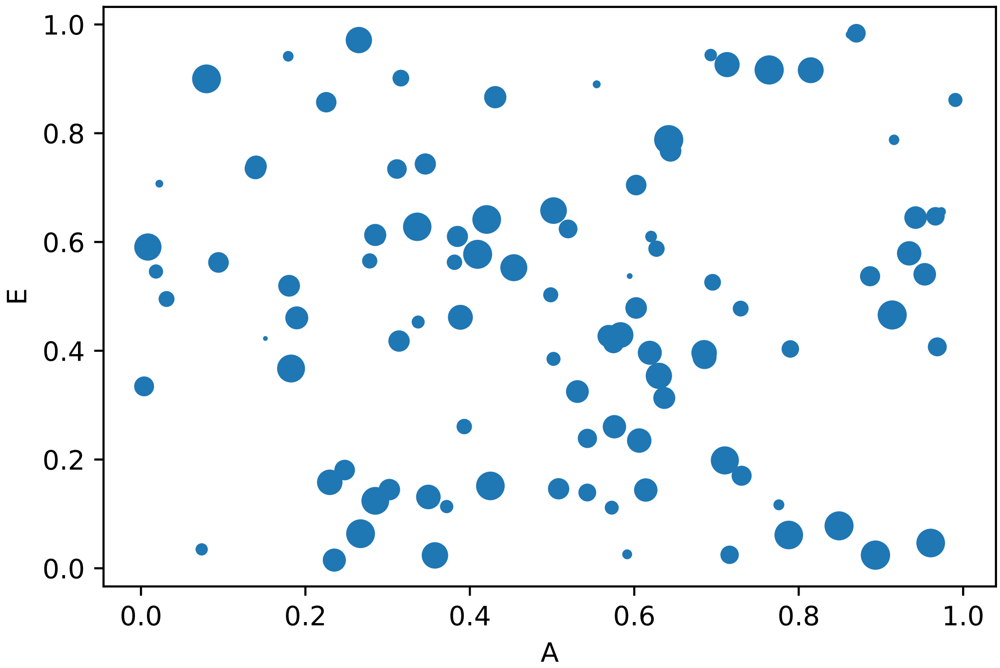
[35]:
```
# Create pandas DataFrame
df = pd.DataFrame(rng.uniform(size=(100, 5)),
                  columns=['A', 'B', 'C', 'D', 'E'])

df.plot.scatter(x='A', y='E', label='Pandas scatter plot')
```

[35]: <AxesSubplot:xlabel='A', ylabel='E'>



```
[36]: # We can even use a column to specify the dot size!
      df.plot.scatter(x='A', y='E', s=df['B']*100.0)
```

```
[36]: <AxesSubplot:xlabel='A', ylabel='E'>
```




In general, the wrappers implemented in pandas are useful to get an idea how the data looks like. For reusable code or more complex graphs, we'll usually want to directly use Matplotlib and pass the data converted to NumPy arrays.

## 2 Exercises

TBA

## 3 Solutions

TBA