Python libraries for working with data

In [1]:
```
# Copyright (c) Thalesians Ltd, 2019-2023. All rights reserved.
# Copyright (c) Paul Alexander Bilokon, 2019-2023. All rights reserved.
# Author: Paul Alexander Bilokon <paul@thalesians.com>
# This version: 2.0 (2023.11.17)
# Previous versions: 1.0 (2019.03.25)
# Email: info@thalesians.com
```

Motivation

Two third-party (but *de facto* standard) Python libraries play a particular role in Data Analysis: `pandas` and NumPy.

The data types defined in `pandas` — `DataFrame` and `Series` — enable the data scientist to work effectively with tables. `pandas` provides some of the functionality that we find in SQL databases and spreadsheets. Since much of the data that we deal with comes in the form of tables, `pandas` is extremely useful.

NumPy provides a lower-level type — the multidimensional array. This data type is used as an underlying implementation by `pandas` `DataFrame`s and `Series`. NumPy arrays are used wherever bulk operations on numbers are needed in Python, which is what much of Data Science is about. NumPy arrays can be seen as a Python implementation of the key linear algebra object — the matrix.

Objectives

1. To introduce `pandas` `DataFrames` and `Series`.
2. To show how... ...to quickly inspect a `pandas` `DataFrame`.
3. ... `pandas` `DataFrame`s can be indexed using `loc` and `iloc`.
4. ... `DataFrame`'s columns can be indexed.
5. ...to iterate through columns of a `DataFrame`.
6. ...a `DataFrame` can be effectively summarised.
7. ...to add a column to a `DataFrame`.
8. ...to overwrite a column in a `DataFrame`.
9. ...to rearrange columns in a `DataFrame`.
10. ...to delete a column from a `DataFrame`.
11. ...to apply a function to a column in a `DataFrame`.
12. ...to filter data in a `DataFrame` based on some boolean expression.
13. ...to deal with missing data (NaNs).

14. ...to use `groupby`.
15. ...to append to a `DataFrame`.
16. ...to join on a `DataFrame`.
17. ...to get at the NumPy arrays behind the `pandas` `DataFrame` and `Series`.
18. ...to define one-dimensional (flat) NumPy arrays.
19. ...to use `np.arange`, `np.linspace`, and `np.logspace`.
20. To explain the reasons to prefer NumPy arrays over standard Python data structures, such as lists.
21. To show how... ...to define two-dimensional NumPy arrays, thus implementing matrices.
22. ...to create NumPy arrays using `np.zeros`, `np.ones`, `np.full`, `np.empty`, `np.tile`.
23. ...to use 32-bit floating point numbers (`float32`) instead of 64-bit floating point numbers in NumPy arrays.
24. ...to generate random matrices.
25. ...to reshape matrices.
26. ...to multiply matrices by scalars.
27. ...to add matrices together.
28. ...to multiply matrices together.
29. ...to mix arrays and scalars in arithmetic operations; to explain broadcasting.
30. ...to transpose matrices.
31. ...to invert matrices.
32. ...to create identity matrices.
33. ...to stack NumPy arrays horizontally.
34. ...to stack NumPy arrays vertically.
35. ...to index NumPy arrays, including boolean indices.
36. ...to compare NumPy arrays.
37. ...to use some useful functions: `np.cumsum`, `np.cumprod`, `np.min`, `np.max`, `np.argmin`, `np.argmax`, `np.mean`, `np.var`, `np.std`.
38. ...to apply a function to each row/column in a NumPy array.
39. ...to use NumPy array flags.
40. ...to make NumPy arrays immutable (kind of).
41. ...to find out more about the configuration of the NumPy library, e.g. which BLAS it is using.

`pandas` `DataFrame`s and `Series`

`pandas` is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. We can import it with

In [2]:
```
import pandas as pd
```

The key data type provided by `pandas` is `DataFrame`. Let us create one:

In [3]:
```
df = pd.DataFrame(
    {
        'transaction date': [2012.917, 2012.917, 2013.583, 2013.500, 2012.833, 2012
```

```
        'distance to the nearest MRT station': [84.87882, 306.59470, 561.98450, 561
        'number of convenience stores': [10, 9, 5, 5, 5, 3, 7, 6, 1, 3, 1, 9, 5, 4,
        'latitude': [24.98298, 24.98034, 24.98746, 24.98746, 24.97937, 24.96305, 24
        'longitude': [121.54024, 121.53951, 121.54391, 121.54391, 121.54245, 121.51
        'house price per unit area': [37.9, 42.2, 47.3, 54.8, 43.1, 32.1, 40.3, 46.
    },
    columns=[
        'transaction date',
        'distance to the nearest MRT station',
        'number of convenience stores',
        'latitude',
        'longitude',
        'house price per unit area'
    ])
```

`pandas` `DataFrame`s in many ways resemble SQL database tables and worksheets in spreadsheet applications.

How to quickly inspect a `DataFrame`

Let's examine our newly created `DataFrame`:

In [4]:
```
df
```

Out[4]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
0	2012.917	84.87882	10	24.98298	121.54024	37.9
1	2012.917	306.59470	9	24.98034	121.53951	42.2
2	2013.583	561.98450	5	24.98746	121.54391	47.3
3	2013.500	561.98450	5	24.98746	121.54391	54.8
4	2012.833	390.56840	5	24.97937	121.54245	43.1
5	2012.667	2175.03000	3	24.96305	121.51254	32.1
6	2012.667	623.47310	7	24.97933	121.53642	40.3
7	2013.417	287.60250	6	24.98042	121.54228	46.7
8	2013.500	5512.03800	1	24.95095	121.48458	18.8
9	2013.417	1783.18000	3	24.96731	121.51486	22.1
10	2013.083	405.21340	1	24.97349	121.53372	41.4
11	2013.333	90.45606	9	24.97433	121.54310	58.1
12	2012.917	492.23130	5	24.96515	121.53737	39.3
13	2012.667	2469.64500	4	24.96108	121.51046	23.8
14	2013.500	1164.83800	4	24.99156	121.53406	34.3

In practice, we may be dealing with `DataFrame`s containing thousands of rows of data (not fifteen, as in our example). So instead of looking at the entire `DataFrame` we may look at its...

In [5]:
```
df.head()
```

Out[5]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
0	2012.917	84.87882	10	24.98298	121.54024	37.9
1	2012.917	306.59470	9	24.98034	121.53951	42.2
2	2013.583	561.98450	5	24.98746	121.54391	47.3
3	2013.500	561.98450	5	24.98746	121.54391	54.8
4	2012.833	390.56840	5	24.97937	121.54245	43.1

We may wish to look at more rows from the `DataFrame`'s head:

In [6]:
```
df.head(10)
```

Out[6]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
0	2012.917	84.87882	10	24.98298	121.54024	37.9
1	2012.917	306.59470	9	24.98034	121.53951	42.2
2	2013.583	561.98450	5	24.98746	121.54391	47.3
3	2013.500	561.98450	5	24.98746	121.54391	54.8
4	2012.833	390.56840	5	24.97937	121.54245	43.1
5	2012.667	2175.03000	3	24.96305	121.51254	32.1
6	2012.667	623.47310	7	24.97933	121.53642	40.3
7	2013.417	287.60250	6	24.98042	121.54228	46.7
8	2013.500	5512.03800	1	24.95095	121.48458	18.8
9	2013.417	1783.18000	3	24.96731	121.51486	22.1

Or examine its tail:

In [7]:
```
df.tail()
```

Out[7]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
10	2013.083	405.21340	1	24.97349	121.53372	41.4
11	2013.333	90.45606	9	24.97433	121.54310	58.1
12	2012.917	492.23130	5	24.96515	121.53737	39.3
13	2012.667	2469.64500	4	24.96108	121.51046	23.8
14	2013.500	1164.83800	4	24.99156	121.53406	34.3

The index of the `DataFrame`, `loc` and `iloc`

Notice the numbers in bold on the left. These constitute the **index** of the `DataFrame`:

In [8]:
```
list(df.index)
```

Out[8]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

We can access the list of **columns** of the data frame through

In [9]:
```
list(df.columns)
```

Out[9]:
```
['transaction date',
 'distance to the nearest MRT station',
 'number of convenience stores',
 'latitude',
 'longitude',
 'house price per unit area']
```

Thus the index indexes the rows, whereas the column names index the columns. We can access individual rows of the `DataFrame` through

In [10]:
```
df.loc[3]
```

Out[10]:
```
transaction date                        2013.50000
distance to the nearest MRT station      561.98450
number of convenience stores               5.00000
latitude                                  24.98746
longitude                                121.54391
house price per unit area                 54.80000
Name: 3, dtype: float64
```

Notice that the index of the data frame doesn't have to be formed from integers starting with zero, as in our case

In [11]:
```
list(df.index)
```

Out[11]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

We could have chosen to index the rows with strings, `'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O'`. Since the `DataFrame` is a **mutable** object, i.e. it can be changed after construction, we can replace its index accordingly:

In [12]:
```
df.index = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', '
```

In [13]:
```
df.head()
```

Out[13]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
A	2012.917	84.87882	10	24.98298	121.54024	37.9
B	2012.917	306.59470	9	24.98034	121.53951	42.2
C	2013.583	561.98450	5	24.98746	121.54391	47.3
D	2013.500	561.98450	5	24.98746	121.54391	54.8
E	2012.833	390.56840	5	24.97937	121.54245	43.1

Now we can index the rows with

In [14]:
```
df.loc['D']
```

Out[14]:
```
transaction date                        2013.50000
distance to the nearest MRT station      561.98450
number of convenience stores               5.00000
latitude                                  24.98746
longitude                                121.54391
house price per unit area                 54.80000
Name: D, dtype: float64
```

Whatever the index of the `DataFrame`, we can also refer to the rows through their integer offsets from the top, using `iloc` instead of `loc`:

In [15]:
```
df.iloc[3]
```

Out[15]:
```
transaction date                        2013.50000
distance to the nearest MRT station      561.98450
number of convenience stores               5.00000
latitude                                  24.98746
longitude                                121.54391
house price per unit area                 54.80000
Name: D, dtype: float64
```

Whereas the type of the entire object is

In [16]:
```
type(df)
```

Out[16]:
```
pandas.core.frame.DataFrame
```

The type of `df.loc['D']` (equivalently, of `df.iloc[3]`) is

In [17]:
```
type(df.loc['D'])
```

Out[17]:
```
pandas.core.series.Series
```

`Series` is the second most important type defined in the `pandas` package (after `DataFrame`).

Indexing columns

As we said above, the list of columns in the dataframe is accessible through

In [18]:
```
list(df.columns)
```

Out[18]:
```
['transaction date',
 'distance to the nearest MRT station',
 'number of convenience stores',
 'latitude',
 'longitude',
 'house price per unit area']
```

Individual columns can be indexed using

In [19]:
```
df['latitude']
```

Out[19]:
```
A    24.98298
B    24.98034
C    24.98746
D    24.98746
E    24.97937
F    24.96305
G    24.97933
H    24.98042
I    24.95095
J    24.96731
K    24.97349
L    24.97433
M    24.96515
N    24.96108
O    24.99156
Name: latitude, dtype: float64
```

The type of the object thus obtained is

In [20]:
```
type(df['latitude'])
```

Out[20]:
```
pandas.core.series.Series
```

You can also access multiple columns at once using

In [21]:
```
df[['latitude', 'longitude']]
```

Out[21]:

	latitude	longitude
A	24.98298	121.54024
B	24.98034	121.53951
C	24.98746	121.54391
D	24.98746	121.54391
E	24.97937	121.54245
F	24.96305	121.51254
G	24.97933	121.53642
H	24.98042	121.54228
I	24.95095	121.48458
J	24.96731	121.51486
K	24.97349	121.53372
L	24.97433	121.54310
M	24.96515	121.53737
N	24.96108	121.51046
O	24.99156	121.53406

The type of the resulting object is `DataFrame`, rather than `Series`:

In [22]:
```
type(df[['latitude', 'longitude']])
```

Out[22]:
```
pandas.core.frame.DataFrame
```

Iterating through columns

Let us demonstrate how we can iterate through the columns of a `DataFrame`. In this example we shall store the first (0th) value from each column in a dictionary:

In [23]:
```
firsts = {}
for c in df.columns:
    firsts[c] = df[c].iloc[0]
```

In [24]:
```
firsts
```

Out[24]:
```
{'transaction date': 2012.917,
 'distance to the nearest MRT station': 84.87882,
 'number of convenience stores': 10,
 'latitude': 24.98298,
 'longitude': 121.54024,
 'house price per unit area': 37.9}
```

Summarising the `DataFrame`

Another quick way to obtain a high-level view of a (potentially very large) `DataFrame` is with

In [25]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 15 entries, A to O
Data columns (total 6 columns):
 #   Column                                  Non-Null Count  Dtype  
---  ------                                  --------------  -----  
 0   transaction date                        15 non-null     float64
 1   distance to the nearest MRT station     15 non-null     float64
 2   number of convenience stores            15 non-null     int64  
 3   latitude                                15 non-null     float64
 4   longitude                               15 non-null     float64
 5   house price per unit area               15 non-null     float64
dtypes: float64(5), int64(1)
memory usage: 1.4+ KB
```

whereas the following will present summary statistics for each column:

In [26]:
```
df.describe()
```

Out[26]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
count	15.000000	15.000000	15.000000	15.000000	15.000000	15.000000
mean	2013.127867	1127.314552	5.133333	24.974952	121.530627	38.813333
std	0.347586	1425.732372	2.722044	0.011395	0.017235	11.251341
min	2012.667000	84.878820	1.000000	24.950950	121.484580	18.800000
25%	2012.875000	348.581550	3.500000	24.966230	121.524290	33.200000
50%	2013.083000	561.984500	5.000000	24.979330	121.537370	40.300000
75%	2013.458500	1474.009000	6.500000	24.981700	121.542365	44.900000
max	2013.583000	5512.038000	10.000000	24.991560	121.543910	58.100000

Adding a column

To add a column, assuming the column `'house age'` does not yet exist:

In [27]:
```
df['house age'] = [32.0, 19.5, 13.3, 13.3, 5.0, 7.1, 34.5, 20.3, 31.7, 17.9, 34.8,
```

In [28]:
```
df.head()
```

Out[28]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area	house age
A	2012.917	84.87882	10	24.98298	121.54024	37.9	32.0
B	2012.917	306.59470	9	24.98034	121.53951	42.2	19.5
C	2013.583	561.98450	5	24.98746	121.54391	47.3	13.3
D	2013.500	561.98450	5	24.98746	121.54391	54.8	13.3
E	2012.833	390.56840	5	24.97937	121.54245	43.1	5.0

Overwriting a column

Similarly, if the column `'house age'` already exists,

In [29]:
```
df['house age'] = [320., 195., 133., 133., 50., 71., 345., 203., 317., 179., 348.,
```

In [30]:
```
df.head()
```

Out[30]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area	house age
A	2012.917	84.87882	10	24.98298	121.54024	37.9	320.0
B	2012.917	306.59470	9	24.98034	121.53951	42.2	195.0
C	2013.583	561.98450	5	24.98746	121.54391	47.3	133.0
D	2013.500	561.98450	5	24.98746	121.54391	54.8	133.0
E	2012.833	390.56840	5	24.97937	121.54245	43.1	50.0

Rearranging the columns

To rearrange the columns:

In [31]:
```
df = df[[
    'transaction date', 'house age',
    'distance to the nearest MRT station', 'number of convenience stores',
    'latitude', 'longitude',
    'house price per unit area'
]]
```

In [32]:
```
df.head()
```

Out[32]:

	transaction date	house age	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
A	2012.917	320.0	84.87882	10	24.98298	121.54024	37.9
B	2012.917	195.0	306.59470	9	24.98034	121.53951	42.2
C	2013.583	133.0	561.98450	5	24.98746	121.54391	47.3
D	2013.500	133.0	561.98450	5	24.98746	121.54391	54.8
E	2012.833	50.0	390.56840	5	24.97937	121.54245	43.1

Deleting a column

In [33]:
```
del df['house age']
```

In [34]:
```
df.head()
```

Out[34]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
A	2012.917	84.87882	10	24.98298	121.54024	37.9
B	2012.917	306.59470	9	24.98034	121.53951	42.2
C	2013.583	561.98450	5	24.98746	121.54391	47.3
D	2013.500	561.98450	5	24.98746	121.54391	54.8
E	2012.833	390.56840	5	24.97937	121.54245	43.1

Applying a function to a column

In [35]:
```
import math
```

In [36]:
```
df['distance to the nearest MRT station'].apply(math.sqrt)
```

Out[36]:
```
A     9.212970
B    17.509846
C    23.706212
D    23.706212
E    19.762803
F    46.637217
G    24.969443
H    16.958847
I    74.243101
J    42.227716
K    20.129913
L     9.510839
M    22.186286
N    49.695523
O    34.129723
Name: distance to the nearest MRT station, dtype: float64
```

Notice that the original column hasn't been overwritten:

In [37]:
```
df.head()
```

Out[37]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
A	2012.917	84.87882	10	24.98298	121.54024	37.9
B	2012.917	306.59470	9	24.98034	121.53951	42.2
C	2013.583	561.98450	5	24.98746	121.54391	47.3
D	2013.500	561.98450	5	24.98746	121.54391	54.8
E	2012.833	390.56840	5	24.97937	121.54245	43.1

To overwrite it,

In [38]:
```
df['distance to the nearest MRT station'] = df['distance to the nearest MRT statior
```

```
C:\Users\paul\AppData\Local\Temp\ipykernel_37936\4176019659.py:1: SettingWithCopyW
arning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stabl
e/user_guide/indexing.html#returning-a-view-versus-a-copy
  df['distance to the nearest MRT station'] = df['distance to the nearest MRT stat
ion'].apply(math.sqrt)
```

In [39]:
```
df.head()
```

Out[39]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
A	2012.917	9.212970	10	24.98298	121.54024	37.9
B	2012.917	17.509846	9	24.98034	121.53951	42.2
C	2013.583	23.706212	5	24.98746	121.54391	47.3
D	2013.500	23.706212	5	24.98746	121.54391	54.8
E	2012.833	19.762803	5	24.97937	121.54245	43.1

To reverse what we have just done:

In [40]:
```
df['distance to the nearest MRT station'] = df['distance to the nearest MRT statior
```

```
C:\Users\paul\AppData\Local\Temp\ipykernel_37936\2618952524.py:1: SettingWithCopyW
arning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stabl
e/user_guide/indexing.html#returning-a-view-versus-a-copy
  df['distance to the nearest MRT station'] = df['distance to the nearest MRT stat
ion'].apply(lambda x: x * x)
```

In [41]:
```
df.head()
```

Out[41]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
A	2012.917	84.87882	10	24.98298	121.54024	37.9
B	2012.917	306.59470	9	24.98034	121.53951	42.2
C	2013.583	561.98450	5	24.98746	121.54391	47.3
D	2013.500	561.98450	5	24.98746	121.54391	54.8
E	2012.833	390.56840	5	24.97937	121.54245	43.1

Filtering data

Suppose that we want to consider only those rows where the value in the column `'number of convenience stores'` is greater than or equal to 7. We can then index `df` by

In [42]:
```
df['number of convenience stores'] >= 7
```

Out[42]:
```
A     True
B     True
C    False
D    False
E    False
F    False
G     True
H    False
I    False
J    False
K    False
L     True
M    False
N    False
O    False
Name: number of convenience stores, dtype: bool
```

obtaining, as a result,

In [43]:
```
df[df['number of convenience stores'] >= 7]
```

Out[43]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
A	2012.917	84.87882	10	24.98298	121.54024	37.9
B	2012.917	306.59470	9	24.98034	121.53951	42.2
G	2012.667	623.47310	7	24.97933	121.53642	40.3
L	2013.333	90.45606	9	24.97433	121.54310	58.1

If we want to consider those rows where the `'number of convenience stores'` is greater than or equal to seven *or* the distance to the nearest MRT station is less than 500, we index the dataframe by

In [44]:
```
(df['number of convenience stores'] >= 7) | (df['distance to the nearest MRT static
```

Out[44]:
```
A     True
B     True
C    False
D    False
E     True
F    False
G     True
H     True
I    False
J    False
K     True
L     True
M     True
N    False
O    False
dtype: bool
```

The result being

In [45]:
```
df[(df['number of convenience stores'] >= 7) | (df['distance to the nearest MRT sta
```

Out[45]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
A	2012.917	84.87882	10	24.98298	121.54024	37.9
B	2012.917	306.59470	9	24.98034	121.53951	42.2
E	2012.833	390.56840	5	24.97937	121.54245	43.1
G	2012.667	623.47310	7	24.97933	121.53642	40.3
H	2013.417	287.60250	6	24.98042	121.54228	46.7
K	2013.083	405.21340	1	24.97349	121.53372	41.4
L	2013.333	90.45606	9	24.97433	121.54310	58.1
M	2012.917	492.23130	5	24.96515	121.53737	39.3

If instead we want to consider those rows where the `'number of convenience stores'` is greater than or equal to seven *and* the distance to the nearest MRT station is less than 500, we index the dataframe by

In [46]:
```
(df['number of convenience stores'] >= 7) & (df['distance to the nearest MRT static
```

Out[46]:
```
A     True
B     True
C    False
D    False
E    False
F    False
G    False
H    False
I    False
J    False
K    False
L     True
M    False
N    False
O    False
dtype: bool
```

The result being

In [47]:
```
df[(df['number of convenience stores'] >= 7) & (df['distance to the nearest MRT sta
```

Out[47]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
A	2012.917	84.87882	10	24.98298	121.54024	37.9
B	2012.917	306.59470	9	24.98034	121.53951	42.2
L	2013.333	90.45606	9	24.97433	121.54310	58.1

Suppose that we set `df1` to the above...

In [48]:
```
df1 = df[(df['number of convenience stores'] >= 7) & (df['distance to the nearest M
```

In [49]:
```
df1.loc['L', 'number of convenience stores'] = 100
```

We get a `SettingWithCopyWarning: A value is trying to be set on a copy of a slice from a DataFrame`. The **slice** (the **view** of a subset of the original `DataFrame`) has changed:

In [50]:
```
df1
```

Out[50]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
A	2012.917	84.87882	10	24.98298	121.54024	37.9
B	2012.917	306.59470	9	24.98034	121.53951	42.2
L	2013.333	90.45606	100	24.97433	121.54310	58.1

Whereas the original has not changed:

In [51]:
```
df
```

Out[51]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
A	2012.917	84.87882	10	24.98298	121.54024	37.9
B	2012.917	306.59470	9	24.98034	121.53951	42.2
C	2013.583	561.98450	5	24.98746	121.54391	47.3
D	2013.500	561.98450	5	24.98746	121.54391	54.8
E	2012.833	390.56840	5	24.97937	121.54245	43.1
F	2012.667	2175.03000	3	24.96305	121.51254	32.1
G	2012.667	623.47310	7	24.97933	121.53642	40.3
H	2013.417	287.60250	6	24.98042	121.54228	46.7
I	2013.500	5512.03800	1	24.95095	121.48458	18.8
J	2013.417	1783.18000	3	24.96731	121.51486	22.1
K	2013.083	405.21340	1	24.97349	121.53372	41.4
L	2013.333	90.45606	9	24.97433	121.54310	58.1
M	2012.917	492.23130	5	24.96515	121.53737	39.3
N	2012.667	2469.64500	4	24.96108	121.51046	23.8
O	2013.500	1164.83800	4	24.99156	121.53406	34.3

For more information on this behaviour, read http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy

Dealing with missing data

Let us examine what happens if we have missing data in the `DataFrame`.

To this end, let us take a copy

In [52]:
```
df_copy = df.copy()
```

and modify it like so:

In [53]:
```
import numpy as np
df_copy.loc['C', 'distance to the nearest MRT station'] = np.nan
df_copy.loc['E', 'number of convenience stores'] = np.nan
```

In [54]:
```
df_copy
```

Out[54]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
A	2012.917	84.87882	10.0	24.98298	121.54024	37.9
B	2012.917	306.59470	9.0	24.98034	121.53951	42.2
C	2013.583	NaN	5.0	24.98746	121.54391	47.3
D	2013.500	561.98450	5.0	24.98746	121.54391	54.8
E	2012.833	390.56840	NaN	24.97937	121.54245	43.1
F	2012.667	2175.03000	3.0	24.96305	121.51254	32.1
G	2012.667	623.47310	7.0	24.97933	121.53642	40.3
H	2013.417	287.60250	6.0	24.98042	121.54228	46.7
I	2013.500	5512.03800	1.0	24.95095	121.48458	18.8
J	2013.417	1783.18000	3.0	24.96731	121.51486	22.1
K	2013.083	405.21340	1.0	24.97349	121.53372	41.4
L	2013.333	90.45606	9.0	24.97433	121.54310	58.1
M	2012.917	492.23130	5.0	24.96515	121.53737	39.3
N	2012.667	2469.64500	4.0	24.96108	121.51046	23.8
O	2013.500	1164.83800	4.0	24.99156	121.53406	34.3

The special `np.nan` value indicates that the data is missing or invalid (NaN stands for "not a number"). NaNs often result from numerical calculations. Their presence may interfere with further numerical work. One quick way to address the issue of NaNs is by dropping (removing) them:

In [55]:
```
df_copy.dropna()
```

Out[55]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
A	2012.917	84.87882	10.0	24.98298	121.54024	37.9
B	2012.917	306.59470	9.0	24.98034	121.53951	42.2
D	2013.500	561.98450	5.0	24.98746	121.54391	54.8
F	2012.667	2175.03000	3.0	24.96305	121.51254	32.1
G	2012.667	623.47310	7.0	24.97933	121.53642	40.3
H	2013.417	287.60250	6.0	24.98042	121.54228	46.7
I	2013.500	5512.03800	1.0	24.95095	121.48458	18.8
J	2013.417	1783.18000	3.0	24.96731	121.51486	22.1
K	2013.083	405.21340	1.0	24.97349	121.53372	41.4
L	2013.333	90.45606	9.0	24.97433	121.54310	58.1
M	2012.917	492.23130	5.0	24.96515	121.53737	39.3
N	2012.667	2469.64500	4.0	24.96108	121.51046	23.8
O	2013.500	1164.83800	4.0	24.99156	121.53406	34.3

This has returned a copy of the `DataFrame` with all rows containing NaNs removed. `df_copy` itself has not been changed:

In [56]:
```
df_copy
```

Out[56]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
A	2012.917	84.87882	10.0	24.98298	121.54024	37.9
B	2012.917	306.59470	9.0	24.98034	121.53951	42.2
C	2013.583	NaN	5.0	24.98746	121.54391	47.3
D	2013.500	561.98450	5.0	24.98746	121.54391	54.8
E	2012.833	390.56840	NaN	24.97937	121.54245	43.1
F	2012.667	2175.03000	3.0	24.96305	121.51254	32.1
G	2012.667	623.47310	7.0	24.97933	121.53642	40.3
H	2013.417	287.60250	6.0	24.98042	121.54228	46.7
I	2013.500	5512.03800	1.0	24.95095	121.48458	18.8
J	2013.417	1783.18000	3.0	24.96731	121.51486	22.1
K	2013.083	405.21340	1.0	24.97349	121.53372	41.4
L	2013.333	90.45606	9.0	24.97433	121.54310	58.1
M	2012.917	492.23130	5.0	24.96515	121.53737	39.3
N	2012.667	2469.64500	4.0	24.96108	121.51046	23.8
O	2013.500	1164.83800	4.0	24.99156	121.53406	34.3

But we could overwrite it with the result of `dropna()`:

In [57]:
```
df_copy = df_copy.dropna()
```

In [58]:
```
df_copy
```

Out[58]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
A	2012.917	84.87882	10.0	24.98298	121.54024	37.9
B	2012.917	306.59470	9.0	24.98034	121.53951	42.2
D	2013.500	561.98450	5.0	24.98746	121.54391	54.8
F	2012.667	2175.03000	3.0	24.96305	121.51254	32.1
G	2012.667	623.47310	7.0	24.97933	121.53642	40.3
H	2013.417	287.60250	6.0	24.98042	121.54228	46.7
I	2013.500	5512.03800	1.0	24.95095	121.48458	18.8
J	2013.417	1783.18000	3.0	24.96731	121.51486	22.1
K	2013.083	405.21340	1.0	24.97349	121.53372	41.4
L	2013.333	90.45606	9.0	24.97433	121.54310	58.1
M	2012.917	492.23130	5.0	24.96515	121.53737	39.3
N	2012.667	2469.64500	4.0	24.96108	121.51046	23.8
O	2013.500	1164.83800	4.0	24.99156	121.53406	34.3

groupby

It may be useful to summarise the `pandas` `DataFrame` for a given value of a particular column. This can be achieved using `groupby`. For example, we may wish to look at the mean of each column for each possible value of `'number of convenience stores'`:

In [59]:
```
df.groupby('number of convenience stores').mean()
```

Out[59]:

number of convenience stores	transaction date	distance to the nearest MRT station	latitude	longitude	house price per unit area
1	2013.29150	2958.625700	24.962220	121.509150	30.100
3	2013.04200	1979.105000	24.965180	121.513700	27.100
4	2013.08350	1817.241500	24.976320	121.522260	29.050
5	2013.20825	501.692175	24.979860	121.541910	46.125
6	2013.41700	287.602500	24.980420	121.542280	46.700
7	2012.66700	623.473100	24.979330	121.536420	40.300
9	2013.12500	198.525380	24.977335	121.541305	50.150
10	2012.91700	84.878820	24.982980	121.540240	37.900

It looks like the house price per unit area may increase with the number of convenience stores, although we aren't particularly certain about this conclusion, whereas the distance to the nearest MRT station does seem to decrease as the number of convenience stores increases.

Instead of the mean, we may consider the minimum:

In [60]:
```
df.groupby('number of convenience stores').min()
```

Out[60]:

number of convenience stores	transaction date	distance to the nearest MRT station	latitude	longitude	house price per unit area
1	2013.083	405.21340	24.95095	121.48458	18.8
3	2012.667	1783.18000	24.96305	121.51254	22.1
4	2012.667	1164.83800	24.96108	121.51046	23.8
5	2012.833	390.56840	24.96515	121.53737	39.3
6	2013.417	287.60250	24.98042	121.54228	46.7
7	2012.667	623.47310	24.97933	121.53642	40.3
9	2012.917	90.45606	24.97433	121.53951	42.2
10	2012.917	84.87882	24.98298	121.54024	37.9

Or maximum:

In [61]:
```
df.groupby('number of convenience stores').max()
```

Out[61]:

number of convenience stores	transaction date	distance to the nearest MRT station	latitude	longitude	house price per unit area
1	2013.500	5512.03800	24.97349	121.53372	41.4
3	2013.417	2175.03000	24.96731	121.51486	32.1
4	2013.500	2469.64500	24.99156	121.53406	34.3
5	2013.583	561.98450	24.98746	121.54391	54.8
6	2013.417	287.60250	24.98042	121.54228	46.7
7	2012.667	623.47310	24.97933	121.53642	40.3
9	2013.333	306.59470	24.98034	121.54310	58.1
10	2012.917	84.87882	24.98298	121.54024	37.9

Appending to a `DataFrame`

Let us again consider our `DataFrame`,

In [62]:
```
df
```

Out[62]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
A	2012.917	84.87882	10	24.98298	121.54024	37.9
B	2012.917	306.59470	9	24.98034	121.53951	42.2
C	2013.583	561.98450	5	24.98746	121.54391	47.3
D	2013.500	561.98450	5	24.98746	121.54391	54.8
E	2012.833	390.56840	5	24.97937	121.54245	43.1
F	2012.667	2175.03000	3	24.96305	121.51254	32.1
G	2012.667	623.47310	7	24.97933	121.53642	40.3
H	2013.417	287.60250	6	24.98042	121.54228	46.7
I	2013.500	5512.03800	1	24.95095	121.48458	18.8
J	2013.417	1783.18000	3	24.96731	121.51486	22.1
K	2013.083	405.21340	1	24.97349	121.53372	41.4
L	2013.333	90.45606	9	24.97433	121.54310	58.1
M	2012.917	492.23130	5	24.96515	121.53737	39.3
N	2012.667	2469.64500	4	24.96108	121.51046	23.8
O	2013.500	1164.83800	4	24.99156	121.53406	34.3

Suppose that we have another `DataFrame`,

In [63]:
```
df1 = pd.DataFrame({
        'transaction date': [2013.417, 2013.083, 2012.917],
        'distance to the nearest MRT station': [378.90278, 90.23891, 489.32891],
```

```
        'number of convenience stores': [5, 6, 7],
        'latitude': [24.97432, 24.97435, 24.97428],
        'longitude': [121.53290, 121.53290, 121.53390],
        'house price per unit area': [33.2, 82.0, 32.1]
    },
    index = ['P', 'Q', 'R'],
    columns = ['transaction date', 'distance to the nearest MRT station', 'number c
               'latitude', 'longitude', 'house price per unit area']
)
```

In [64]:
```
df1
```

Out[64]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
P	2013.417	378.90278	5	24.97432	121.5329	33.2
Q	2013.083	90.23891	6	24.97435	121.5329	82.0
R	2012.917	489.32891	7	24.97428	121.5339	32.1

We can append `df1` to the end of `df` using

In [65]:
```
pd.concat([df, df1])
```

Out[65]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
A	2012.917	84.87882	10	24.98298	121.54024	37.9
B	2012.917	306.59470	9	24.98034	121.53951	42.2
C	2013.583	561.98450	5	24.98746	121.54391	47.3
D	2013.500	561.98450	5	24.98746	121.54391	54.8
E	2012.833	390.56840	5	24.97937	121.54245	43.1
F	2012.667	2175.03000	3	24.96305	121.51254	32.1
G	2012.667	623.47310	7	24.97933	121.53642	40.3
H	2013.417	287.60250	6	24.98042	121.54228	46.7
I	2013.500	5512.03800	1	24.95095	121.48458	18.8
J	2013.417	1783.18000	3	24.96731	121.51486	22.1
K	2013.083	405.21340	1	24.97349	121.53372	41.4
L	2013.333	90.45606	9	24.97433	121.54310	58.1
M	2012.917	492.23130	5	24.96515	121.53737	39.3
N	2012.667	2469.64500	4	24.96108	121.51046	23.8
O	2013.500	1164.83800	4	24.99156	121.53406	34.3
P	2013.417	378.90278	5	24.97432	121.53290	33.2
Q	2013.083	90.23891	6	24.97435	121.53290	82.0
R	2012.917	489.32891	7	24.97428	121.53390	32.1

Joining on a `DataFrame`

Let us again consider our `DataFrame`,

In [66]:
```
df
```

Out[66]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area
A	2012.917	84.87882	10	24.98298	121.54024	37.9
B	2012.917	306.59470	9	24.98034	121.53951	42.2
C	2013.583	561.98450	5	24.98746	121.54391	47.3
D	2013.500	561.98450	5	24.98746	121.54391	54.8
E	2012.833	390.56840	5	24.97937	121.54245	43.1
F	2012.667	2175.03000	3	24.96305	121.51254	32.1
G	2012.667	623.47310	7	24.97933	121.53642	40.3
H	2013.417	287.60250	6	24.98042	121.54228	46.7
I	2013.500	5512.03800	1	24.95095	121.48458	18.8
J	2013.417	1783.18000	3	24.96731	121.51486	22.1
K	2013.083	405.21340	1	24.97349	121.53372	41.4
L	2013.333	90.45606	9	24.97433	121.54310	58.1
M	2012.917	492.23130	5	24.96515	121.53737	39.3
N	2012.667	2469.64500	4	24.96108	121.51046	23.8
O	2013.500	1164.83800	4	24.99156	121.53406	34.3

Suppose we have another `DataFrame`,

In [67]:
```
df_comments = pd.DataFrame({
        'comments': ['data to be validated', 'to be confirmed'],
    },
    index = ['H', 'J']
)
```

In [68]:
```
df_comments
```

Out[68]:

	comments
H	data to be validated
J	to be confirmed

We can join `df_comments` onto `df` based on the matching indices:

In [69]:
```
joined_df = df.join(df_comments)
joined_df
```

Out[69]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area	comments
A	2012.917	84.87882	10	24.98298	121.54024	37.9	NaN
B	2012.917	306.59470	9	24.98034	121.53951	42.2	NaN
C	2013.583	561.98450	5	24.98746	121.54391	47.3	NaN
D	2013.500	561.98450	5	24.98746	121.54391	54.8	NaN
E	2012.833	390.56840	5	24.97937	121.54245	43.1	NaN
F	2012.667	2175.03000	3	24.96305	121.51254	32.1	NaN
G	2012.667	623.47310	7	24.97933	121.53642	40.3	NaN
H	2013.417	287.60250	6	24.98042	121.54228	46.7	data to be validated
I	2013.500	5512.03800	1	24.95095	121.48458	18.8	NaN
J	2013.417	1783.18000	3	24.96731	121.51486	22.1	to be confirmed
K	2013.083	405.21340	1	24.97349	121.53372	41.4	NaN
L	2013.333	90.45606	9	24.97433	121.54310	58.1	NaN
M	2012.917	492.23130	5	24.96515	121.53737	39.3	NaN
N	2012.667	2469.64500	4	24.96108	121.51046	23.8	NaN
O	2013.500	1164.83800	4	24.99156	121.53406	34.3	NaN

We may wish to replace the NaNs that have resulted in the `'comments'` column with blank strings:

In [70]:
```
joined_df.loc[joined_df['comments'].isnull(), 'comments'] = ''
```

In [71]:
```
joined_df
```

Out[71]:

	transaction date	distance to the nearest MRT station	number of convenience stores	latitude	longitude	house price per unit area	comments
A	2012.917	84.87882	10	24.98298	121.54024	37.9	
B	2012.917	306.59470	9	24.98034	121.53951	42.2	
C	2013.583	561.98450	5	24.98746	121.54391	47.3	
D	2013.500	561.98450	5	24.98746	121.54391	54.8	
E	2012.833	390.56840	5	24.97937	121.54245	43.1	
F	2012.667	2175.03000	3	24.96305	121.51254	32.1	
G	2012.667	623.47310	7	24.97933	121.53642	40.3	
H	2013.417	287.60250	6	24.98042	121.54228	46.7	data to be validated
I	2013.500	5512.03800	1	24.95095	121.48458	18.8	
J	2013.417	1783.18000	3	24.96731	121.51486	22.1	to be confirmed
K	2013.083	405.21340	1	24.97349	121.53372	41.4	
L	2013.333	90.45606	9	24.97433	121.54310	58.1	
M	2012.917	492.23130	5	24.96515	121.53737	39.3	
N	2012.667	2469.64500	4	24.96108	121.51046	23.8	
O	2013.500	1164.83800	4	24.99156	121.53406	34.3	

Exercise

Consider the `DataFrame`

In [72]:
```
eg_df = pd.DataFrame({
        'date': ['2019-09-01', '2019-09-02', '2019-09-03', '2019-09-04', '2019-09-0
        'value': [3.78, 2.90, 3.29, 1.21, 3.20, 9.39, 8.90]
    },
    columns=['date', 'value'])
eg_df
```

Out[72]:

	date	value
0	2019-09-01	3.78
1	2019-09-02	2.90
2	2019-09-03	3.29
3	2019-09-04	1.21
4	2019-09-05	3.20
5	2019-09-06	9.39
6	2019-09-07	8.90

Replace the default index with the parsed dates from the `'date'` column. Once this is done, remove the `'date'` column.

Solution

In [73]:
```
import datetime as dt
eg_df.index = eg_df['date'].apply(lambda x: dt.datetime.strptime(x, '%Y-%m-%d').dat
del eg_df['date']
eg_df
```

Out[73]:

	value
date	
2019-09-01	3.78
2019-09-02	2.90
2019-09-03	3.29
2019-09-04	1.21
2019-09-05	3.20
2019-09-06	9.39
2019-09-07	8.90

We can now use dates in `loc` to index this `DataFrame`:

In [74]:
```
eg_df.loc[dt.date(2019, 9,  5)]
```

Out[74]:
```
value    3.2
Name: 2019-09-05, dtype: float64
```

In [75]:
```
eg_df.loc[dt.date(2019, 9,  5), 'value']
```

Out[75]:
```
3.2
```

We can still use integers with `iloc`:

In [76]:
```
eg_df.iloc[4]
```

Out[76]:
```
value    3.2
Name: 2019-09-05, dtype: float64
```

Exercise

Consider again the `DataFrame`

In [77]:
```
eg_df = pd.DataFrame({
        'date': ['2019-09-01', '2019-09-02', '2019-09-03', '2019-09-04', '2019-09-0
        'value': [3.78, 2.90, 3.29, 1.21, 3.20, 9.39, 8.90]
    },
    columns=['date', 'value'])
```

Find the mean, minimum, and maximum `'value'`. Use the library functions `numpy.mean`, `numpy.min`, and `numpy.max`.

Solution

In [78]:
```
import numpy as np
print('Mean:', np.mean(eg_df['value']))
```

```
print('Min:', np.min(eg_df['value']))
print('Max:', np.max(eg_df['value']))
```

```
Mean: 4.667142857142857
Min: 1.21
Max: 9.39
```

We didn't have to explicitly use NumPy for this; we could have used

In [79]:
```
eg_df['value'].mean()
```

Out[79]:
```
4.667142857142857
```

In [80]:
```
eg_df['value'].min()
```

Out[80]:
```
1.21
```

In [81]:
```
eg_df['value'].max()
```

Out[81]:
```
9.39
```

Exercise

Consider the `DataFrame`

In [82]:
```
eg_df = pd.DataFrame({
        'date': ['2019-09-01', '2019-09-02', '2019-09-03', '2019-09-04', '2019-09-0
        'value': [3.78, 2.90, 3.29, 1.21, 3.20, 9.39, 8.90]
    },
    columns=['date', 'value'])
eg_df
```

Out[82]:

	date	value
0	2019-09-01	3.78
1	2019-09-02	2.90
2	2019-09-03	3.29
3	2019-09-04	1.21
4	2019-09-05	3.20
5	2019-09-06	9.39
6	2019-09-07	8.90

Join it with another `DataFrame` so that, for dates `2019-09-03` and `2019-09-06` the comment `'missing data'` is added in the `'comments'` column.

Solution

In [83]:
```
eg_df1 = pd.DataFrame({
        'comments': ['missing data', 'missing data']
    },
    index = [2, 5]
)
eg_df1
```

Out[83]:

	comments
2	missing data
5	missing data

In [84]:
```
joined_df = eg_df.join(eg_df1)
joined_df.loc[joined_df['comments'].isnull(), 'comments'] = ''
joined_df
```

Out[84]:

	date	value	comments
0	2019-09-01	3.78	
1	2019-09-02	2.90	
2	2019-09-03	3.29	missing data
3	2019-09-04	1.21	
4	2019-09-05	3.20	
5	2019-09-06	9.39	missing data
6	2019-09-07	8.90	

Exercise

Consider the `DataFrame`

In [85]:
```
eg_df = pd.DataFrame({
        'date': ['2019-09-01', '2019-09-02', '2019-09-03', '2019-09-04', '2019-09-0
        'value': [3.78, 2.90, 3.29, 1.21, 3.20, 9.39, 8.90]
    },
    columns=['date', 'value'])
eg_df
```

Out[85]:

	date	value
0	2019-09-01	3.78
1	2019-09-02	2.90
2	2019-09-03	3.29
3	2019-09-04	1.21
4	2019-09-05	3.20
5	2019-09-06	9.39
6	2019-09-07	8.90

Two new data points become available. `9.89` for `2019-09-08` and `3.89` for `2019-09-09`. Append them to the `DataFrame`.

Solution

In [86]:
```
new_df = pd.DataFrame({
        'date': ['2019-09-08', '2019-09-09'],
        'value': [9.89, 3.89]
    },
```

```
    columns=['date', 'value'],
    index=[7, 8]
)
```

In [87]:
```
extended_df = pd.concat([eg_df, new_df])
extended_df
```

Out[87]:

	date	value
0	2019-09-01	3.78
1	2019-09-02	2.90
2	2019-09-03	3.29
3	2019-09-04	1.21
4	2019-09-05	3.20
5	2019-09-06	9.39
6	2019-09-07	8.90
7	2019-09-08	9.89
8	2019-09-09	3.89

Exercise

In the `DataFrame` resulting from appending the data in the previous exercise, find the mean value for each day of the weak (Monday, Tuesday, Wednesday, etc.). You can get a weekday from a date using

In [88]:
```
dt.date(2019, 9, 2).weekday()
```

Out[88]:
```
0
```

Solution

In [89]:
```
import datetime as dt
extended_df['weekday'] = extended_df['date'].apply(lambda x: dt.datetime.strptime(x
#extended_df.groupby('weekday').mean()
```

In [90]:
```
extended_df[['weekday', 'value']].groupby('weekday').mean()
```

Out[90]:

	value
weekday	
0	3.395
1	3.290
2	1.210
3	3.200
4	9.390
5	8.900
6	6.835

NumPy arrays

Many Python libraries (such as `pandas`) rely on NumPy under the hood. NumPy, imported with

In [91]:
```
import numpy as np
```

implements **multidimensional arrays**.

NumPy arrays behind the `pandas` `DataFrame` and `Series`

To access the underlying NumPy array of a `pandas` `DataFrame`, we can use `values`:

In [92]:
```
df.values
```

Out[92]:
```
array([[2.0129170e+03, 8.4878820e+01, 1.0000000e+01, 2.4982980e+01,
        1.2154024e+02, 3.7900000e+01],
       [2.0129170e+03, 3.0659470e+02, 9.0000000e+00, 2.4980340e+01,
        1.2153951e+02, 4.2200000e+01],
       [2.0135830e+03, 5.6198450e+02, 5.0000000e+00, 2.4987460e+01,
        1.2154391e+02, 4.7300000e+01],
       [2.0135000e+03, 5.6198450e+02, 5.0000000e+00, 2.4987460e+01,
        1.2154391e+02, 5.4800000e+01],
       [2.0128330e+03, 3.9056840e+02, 5.0000000e+00, 2.4979370e+01,
        1.2154245e+02, 4.3100000e+01],
       [2.0126670e+03, 2.1750300e+03, 3.0000000e+00, 2.4963050e+01,
        1.2151254e+02, 3.2100000e+01],
       [2.0126670e+03, 6.2347310e+02, 7.0000000e+00, 2.4979330e+01,
        1.2153642e+02, 4.0300000e+01],
       [2.0134170e+03, 2.8760250e+02, 6.0000000e+00, 2.4980420e+01,
        1.2154228e+02, 4.6700000e+01],
       [2.0135000e+03, 5.5120380e+03, 1.0000000e+00, 2.4950950e+01,
        1.2148458e+02, 1.8800000e+01],
       [2.0134170e+03, 1.7831800e+03, 3.0000000e+00, 2.4967310e+01,
        1.2151486e+02, 2.2100000e+01],
       [2.0130830e+03, 4.0521340e+02, 1.0000000e+00, 2.4973490e+01,
        1.2153372e+02, 4.1400000e+01],
       [2.0133330e+03, 9.0456060e+01, 9.0000000e+00, 2.4974330e+01,
        1.2154310e+02, 5.8100000e+01],
       [2.0129170e+03, 4.9223130e+02, 5.0000000e+00, 2.4965150e+01,
        1.2153737e+02, 3.9300000e+01],
       [2.0126670e+03, 2.4696450e+03, 4.0000000e+00, 2.4961080e+01,
        1.2151046e+02, 2.3800000e+01],
       [2.0135000e+03, 1.1648380e+03, 4.0000000e+00, 2.4991560e+01,
        1.2153406e+02, 3.4300000e+01]])
```

This gives us the raw numerical data. Similarly, we can access the NumPy array behind the `Series` representing a specific column:

In [93]:
```
df['transaction date'].values
```

Out[93]:
```
array([2012.917, 2012.917, 2013.583, 2013.5  , 2012.833, 2012.667,
       2012.667, 2013.417, 2013.5  , 2013.417, 2013.083, 2013.333,
       2012.917, 2012.667, 2013.5  ])
```

Or row:

In [94]:
```
df.loc['A'].values
```

Out[94]:
```
array([2012.917  ,   84.87882,   10.     ,   24.98298,  121.54024,
         37.9    ])
```

Defining one-dimensional (flat) NumPy arrays

Defining a **one-dimensional** (**flat**) array is easy: you can simply wrap a Python list:

In [95]:
```
a = np.array([3.57, 4.18, 25.7])
a
```

Out[95]:
```
array([ 3.57,  4.18, 25.7 ])
```

In [96]:
```
np.ndim(a)
```

Out[96]:
```
1
```

In [97]:
```
np.size(a)
```

Out[97]:
```
3
```

In [98]:
```
np.shape(a)
```

Out[98]:
```
(3,)
```

`np.arange`, `np.linspace`, `np.logspace`

Often we want to obtain a result similar to that of `range` but for floating point numbers (rather than integers); we want to obtain an array of floating point numbers at equally spaced intervals (e.g. for plotting). In NumPy this is attained using `linspace`. Thus to obtain an array of fifteen equally spaced floating point numbers starting at -5. (inclusive), finishing at 10. (inclusive), we can use

In [99]:
```
np.linspace(-5., 10., 15)
```

Out[99]:
```
array([-5.        , -3.92857143, -2.85714286, -1.78571429, -0.71428571,
        0.35714286,  1.42857143,  2.5       ,  3.57142857,  4.64285714,
        5.71428571,  6.78571429,  7.85714286,  8.92857143, 10.        ])
```

Notice that the arguments specifying the start and end of the interval are both inclusive: the first value is exactly `-5.`, the last value is exactly `10.`. This is unlike the arguments of Python's standard `range`: the start of the range is inclusive, but the end of the range is exclusive:

In [100…
```
list(range(-5, 10, 1))
```

Out[100]:
```
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Also note that the third argument of `np.linspace` is the total number of points to be produced, whereas the third argument of range is the step size. NumPy has a function `np.arange`, which is similar to `range`, but works with floating point arguments and produces NumPy arrays:

In [101…
```
np.arange(-5., 10., 1.25)
```

Out[101]:
```
array([-5.  , -3.75, -2.5 , -1.25,  0.  ,  1.25,  2.5 ,  3.75,  5.  ,
        6.25,  7.5 ,  8.75])
```

In [102…
```
np.arange(-5., 10., 1.)
```

Out[102]:
```
array([-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,
        8.,  9.])
```

`np.linspace` and `np.arange` are particularly useful for plotting:

In [103…
```
xs = np.linspace(0., 99. * math.pi / 8., 100)
ys = np.sin(xs)
```

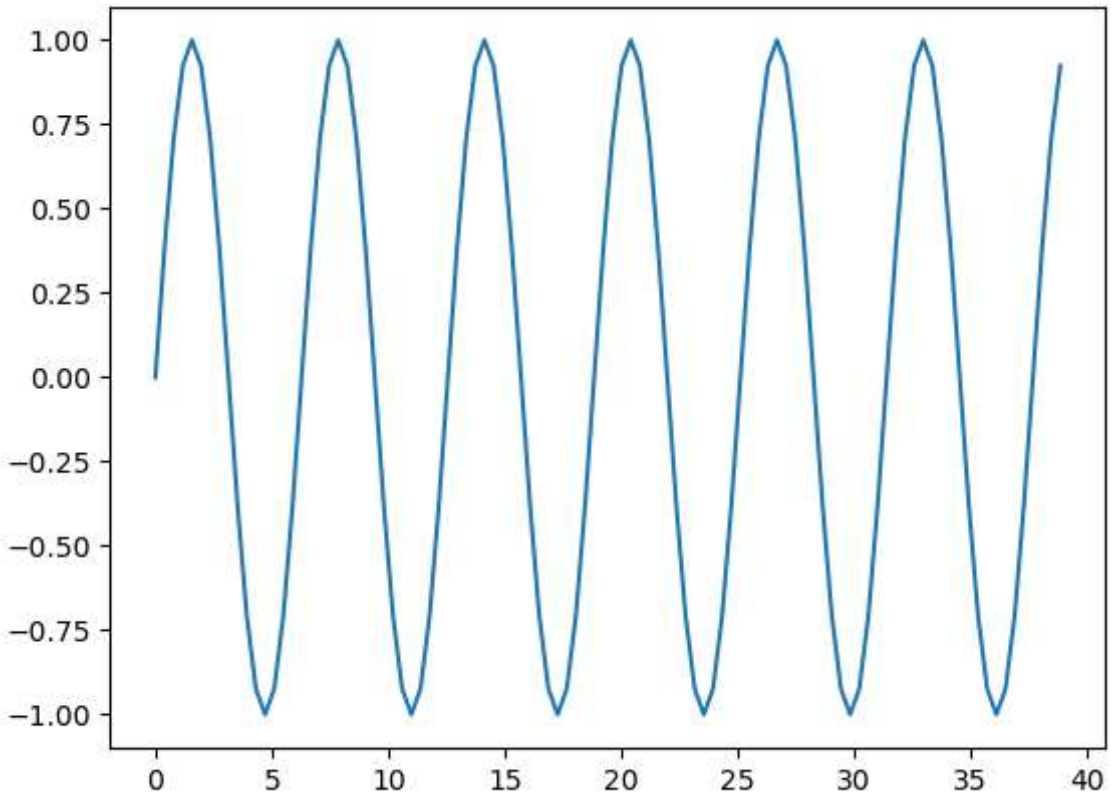
We need the following cell magic to enable plotting in a Jupyter notebook:

In [104…
```
%matplotlib inline
```

And then we plot using the library Matplotlib (also known as `pyplot`):

In [105…
```
import matplotlib.pyplot as plt
```

In [106…
```
plt.plot(xs, ys);
```



Instead of linearly (equally) spaced points, we can produce points spaced logarithmically using `np.logspace`:

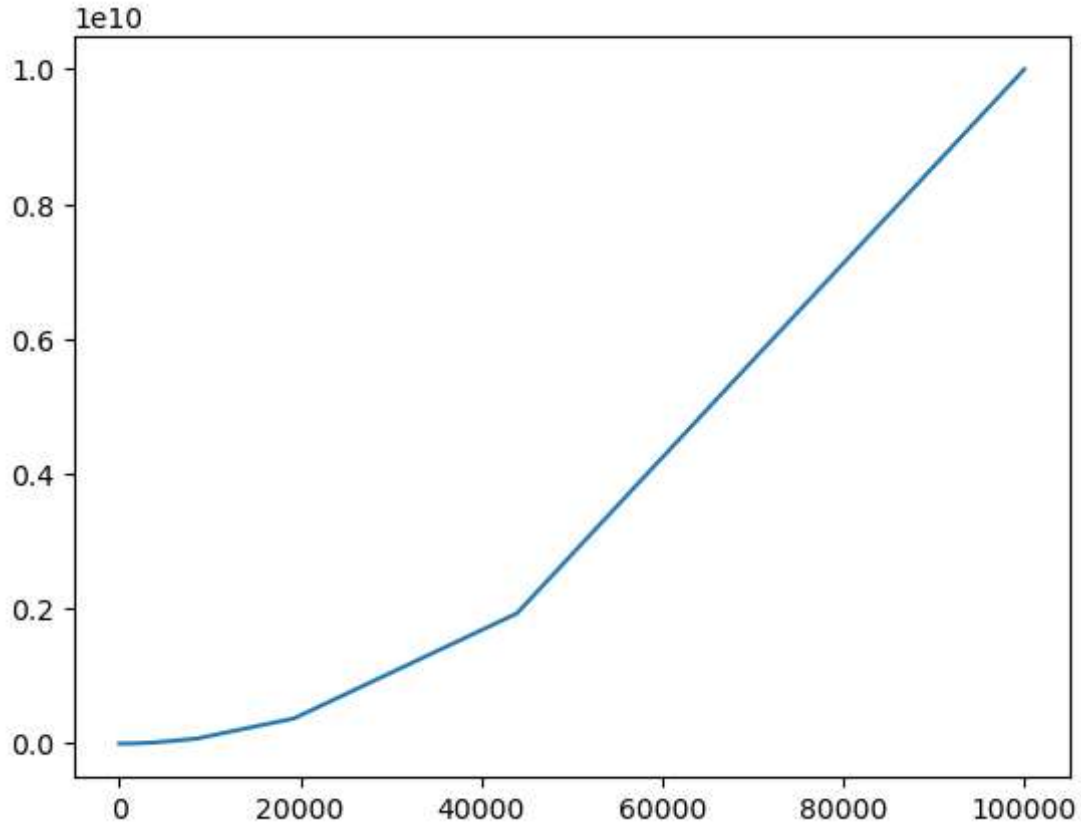
In [107…
```
xs = np.logspace(start=0., stop=5., num=15, base=10.)
xs
```

Out[107]:
```
array([1.00000000e+00, 2.27584593e+00, 5.17947468e+00, 1.17876863e+01,
       2.68269580e+01, 6.10540230e+01, 1.38949549e+02, 3.16227766e+02,
       7.19685673e+02, 1.63789371e+03, 3.72759372e+03, 8.48342898e+03,
       1.93069773e+04, 4.39397056e+04, 1.00000000e+05])
```

In [108…
```
ys = [x**2 for x in xs]
```

Here we start with the zeroth power of ten and end with the fifth power of 10.

In [109…
```
plt.plot(xs, ys);
```



Reasons to prefer NumPy arrays over standard Python data structures, such as lists

Suppose that we want to compute the sine of a number of regularly spaced angles (perhaps with a view of plotting the resulting graph):

In [110…
```
import math
```

In [111…
```
xs = [n * math.pi / 8. for n in range(10000)]
ys = [math.sin(x) for x in xs]
```

We can time the creation of each of the arrays, `xs` and `ys`, using the `timeit` cell magic:

In [112…
```
%timeit -o xs = [n * math.pi / 8. for n in range(10000)]
```

Out[112]:
```
1.83 ms ± 616 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
<TimeitResult : 1.83 ms ± 616 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops
each)>
```

The creation of `xs` took about 6 milliseconds.

In [113…
```
%timeit -o ys = [math.sin(x) for x in xs]
```

Out[113]:
```
2.25 ms ± 210 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
<TimeitResult : 2.25 ms ± 210 µs per loop (mean ± std. dev. of 7 runs, 100 loops e
ach)>
```

The creation of `ys` took about 7 milliseconds.

We can replace the above Python lists with NumPy arrays and initialise them using **vectorised** operations (vectorised operations work on many values in one go), `np.linspace` and `np.sin`:

In [114…
```
xs = np.linspace(0., 9999. * math.pi / 8., 10000)
ys = np.sin(xs)
```

Let us time these operations.

In [115…
```
%timeit -o xs = np.linspace(0., 9999. * math.pi / 8., 10000)
```

Out[115]:
```
41.1 µs ± 2.11 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
<TimeitResult : 41.1 µs ± 2.11 µs per loop (mean ± std. dev. of 7 runs, 10,000 loo
ps each)>
```

In [116…
```
%timeit -o ys = np.sin(xs)
```

Out[116]:
```
112 µs ± 6.67 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
<TimeitResult : 112 µs ± 6.67 µs per loop (mean ± std. dev. of 7 runs, 10,000 loop
s each)>
```

Vectorised operations on NumPy arrays take hundreds of *microseconds*, as opposed to *milliseconds* for Python lists, thus we have an order of magnitude improvement in speed.

Performance is one reason to prefer NumPy arrays over standard Python data structures, such as lists.

Defining two-dimensional NumPy arrays (implementing matrices)

The real power of NumPy is in supporting two-dimensional, arrays, which can implement matrices:

In [117…
```
my_matrix = np.array([[4.28, 3.23, 5.87], [1.23, 5.32, 3.33]])
my_matrix
```

Out[117]:
```
array([[4.28, 3.23, 5.87],
       [1.23, 5.32, 3.33]])
```

In [118…
```
np.ndim(my_matrix)
```

Out[118]:
```
2
```

In [119…
```
np.size(my_matrix)
```

Out[119]:
```
6
```

This particular two-dimensional NumPy array (matrix) is 2 by 3:

In [120…
```
np.shape(my_matrix)
```
Out[120]:
```
(2, 3)
```

Other ways of creating NumPy arrays

Instead of creating NumPy arrays from (either nested or flat) Python lists, we could use NumPy utility functions.

In [121…
```
np.zeros((3,))
```
Out[121]:
```
array([0., 0., 0.])
```

will create a one-dimensional array of zeros of size three, whereas

In [122…
```
np.zeros((3, 5))
```
Out[122]:
```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

will create a two-dimensional array of zeros with three rows and five columns. In each case, the argument (`(3,)` , `(3, 5)`) is a tuple specifying the shape of the NumPy array, giving the size of each of its dimensions.

`np.zeros` takes an optional `dtype` (data type) argument. Thus instead of 64-bit floating point numbers, we could use 32-bit floating point numbers:

In [123…
```
np.zeros((3, 5), dtype='float32')
```
Out[123]:
```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]], dtype=float32)
```

Many GPUs are optimised for 32-bit, rather than 64-bit, precision.

Instead of an array of zeros, we could obtain an array of ones:

In [124…
```
np.ones((3,))
```
Out[124]:
```
array([1., 1., 1.])
```

In [125…
```
np.ones((3, 5))
```
Out[125]:
```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

We could obtain an array with all values equal to a given number by multiplying the result of `np.ones` by that number:

In [126…
```
np.ones((3, 5)) * 7.5
```
Out[126]:
```
array([[7.5, 7.5, 7.5, 7.5, 7.5],
       [7.5, 7.5, 7.5, 7.5, 7.5],
       [7.5, 7.5, 7.5, 7.5, 7.5]])
```

However, it is more efficient (this only really matters for large arrays) to use `np.full`:

In [127…
```
np.full((3, 5), 7.5)
```

Out[127]:
```
array([[7.5, 7.5, 7.5, 7.5, 7.5],
       [7.5, 7.5, 7.5, 7.5, 7.5],
       [7.5, 7.5, 7.5, 7.5, 7.5]])
```

Unless you absolutely must initialise every value of an array to some number, it is most efficient to simply allocate the memory for the array (in which case you end up with whatever values that memory currently contains). This is done using `np.empty`:

In [128…
```
np.empty((3, 10))
```

Out[128]:
```
array([[8.56849635e-312, 1.02765654e-321, 0.00000000e+000,
        0.00000000e+000, 9.18824177e-312, 5.02034658e+175,
        3.73835794e-061, 1.10497312e+165, 8.85426453e+165,
        3.43897401e+175],
       [8.68666981e-043, 5.83258408e-144, 3.59751658e+252,
        3.96046095e+246, 1.04918621e-153, 7.69165785e+218,
        5.04621343e+180, 1.04917822e-153, 9.08366793e+223,
        3.73835794e-061],
       [1.10497312e+165, 8.85426453e+165, 3.43897401e+175,
        1.04915136e-153, 1.94918966e-153, 9.72158982e-072,
        2.26267370e-076, 9.06496566e-043, 2.59027896e-144,
        7.79952704e-143]])
```

You are then responsible for setting the elements of the array appropriately, perhaps using a numerical algorithm.

Because no initialisation takes place, `np.empty` is typically more efficient than `np.zeros`:

In [129…
```
%timeit -o np.zeros((3, 10))
```

Out[129]:
```
309 ns ± 12.4 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
<TimeitResult : 309 ns ± 12.4 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 l
oops each)>
```

In [130…
```
%timeit -o np.empty((3, 10))
```

Out[130]:
```
291 ns ± 24.4 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
<TimeitResult : 291 ns ± 24.4 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 l
oops each)>
```

In [131…
```
%timeit -o np.zeros((10000000,))
```

Out[131]:
```
18.9 µs ± 1.4 µs per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
<TimeitResult : 18.9 µs ± 1.4 µs per loop (mean ± std. dev. of 7 runs, 100,000 loo
ps each)>
```

In [132…
```
%timeit -o np.empty((10000000,))
```

Out[132]:
```
18.8 µs ± 1.76 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
<TimeitResult : 18.8 µs ± 1.76 µs per loop (mean ± std. dev. of 7 runs, 10,000 loo
ps each)>
```

However, the operating system may initialise large newly allocated chunks of memory to zero for security reasons, which may make the difference in performance immaterial.

Matrices can also be created out of blocks using `np.tile`:

In [133…
```
np.tile([[4.28, 3.23, 5.87], [1.23, 5.32, 3.33]], (2, 3))
```

Out[133]:
```
array([[4.28, 3.23, 5.87, 4.28, 3.23, 5.87, 4.28, 3.23, 5.87],
       [1.23, 5.32, 3.33, 1.23, 5.32, 3.33, 1.23, 5.32, 3.33],
       [4.28, 3.23, 5.87, 4.28, 3.23, 5.87, 4.28, 3.23, 5.87],
       [1.23, 5.32, 3.33, 1.23, 5.32, 3.33, 1.23, 5.32, 3.33]])
```

Generating random matrices

NumPy's `np.random.random` returns random floats in the half-open interval $[0, 1)$. Results are sampled from the continuous uniform distribution over this interval.

In [134…
```
np.random.random((2, 3))
```

Out[134]:
```
array([[0.34081837, 0.05191541, 0.50609225],
       [0.8878024 , 0.328979  , 0.90749874]])
```

To sample from $\text{Uniform}[a, b)$, $b > a$, multiply the output by $(b - a)$ and add a:

In [135…
```
np.random.random((2, 3)) * 10. + 100.
```

Out[135]:
```
array([[104.41376148, 104.04161426, 109.15663435],
       [108.6686644 , 103.38246868, 104.99669658]])
```

Reshaping matrices

Once a matrix has been created,

In [136…
```
my_matrix = np.array([[4.28, 3.23, 5.87], [1.23, 5.32, 3.33]])
my_matrix
```

Out[136]:
```
array([[4.28, 3.23, 5.87],
       [1.23, 5.32, 3.33]])
```

Its shape,

In [137…
```
my_matrix.shape
```

Out[137]:
```
(2, 3)
```

can be changed:

In [138…
```
my_matrix.shape = (3, 2)
my_matrix
```

Out[138]:
```
array([[4.28, 3.23],
       [5.87, 1.23],
       [5.32, 3.33]])
```

In [139…
```
my_matrix.shape = (6,)
my_matrix
```

Out[139]:
```
array([4.28, 3.23, 5.87, 1.23, 5.32, 3.33])
```

In [140…
```
my_matrix.shape = ((6, 1))
my_matrix
```

Out[140]:
```
array([[4.28],
       [3.23],
       [5.87],
       [1.23],
       [5.32],
       [3.33]])
```

In [141…
```
my_matrix.shape = ((1, 6))
my_matrix
```

Out[141]:
```
array([[4.28, 3.23, 5.87, 1.23, 5.32, 3.33]])
```

We could also use `np.reshape`, which works on Python lists as well as NumPy arrays:

In [142…
```
my_reshaped_matrix = np.reshape(my_matrix, (2, 3))
my_reshaped_matrix
```

Out[142]:
```
array([[4.28, 3.23, 5.87],
       [1.23, 5.32, 3.33]])
```

In [143…
```
my_matrix = my_reshaped_matrix
```

In [144…
```
np.reshape([[4.28, 3.23, 5.87], [1.23, 5.32, 3.33]], (6, 1))
```

Out[144]:
```
array([[4.28],
       [3.23],
       [5.87],
       [1.23],
       [5.32],
       [3.33]])
```

Multiplying matrices by scalars

Such matrices can be multiplied by scalars:

In [145…
```
3. * my_matrix
```

Out[145]:
```
array([[12.84,  9.69, 17.61],
       [ 3.69, 15.96,  9.99]])
```

Adding matrices

Added together:

In [146…
```
my_matrix + np.array([[3.00, 0., 0.], [5., 7., 0.]])
```

Out[146]:
```
array([[ 7.28,  3.23,  5.87],
       [ 6.23, 12.32,  3.33]])
```

Multiplying matrices

And multiplied by other compatible matrices:

In [147…
```
A = np.dot(my_matrix, np.array([[3., 1.], [1., -3.], [-3., -3.]]))
A
```

Out[147]:
```
array([[ -1.54, -23.02],
       [ -0.98, -24.72]])
```

We can also realise this as

In [148…
```
my_matrix @ np.array([[3., 1.], [1., -3.], [-3., -3.]])
```

Out[148]:
```
array([[ -1.54, -23.02],
       [ -0.98, -24.72]])
```

Mixing arrays and scalars in arithmetic operations

We can mix arrays and scalars in arithmetic expressions:

In [149…
```
10. * my_matrix - 2. * my_matrix * my_matrix + 1.
```

Out[149]:
```
array([[ 7.1632, 12.4342, -9.2138],
       [10.2742, -2.4048, 12.1222]])
```

The scalar is applied elementwise to each element of the array — we say that the scalar is **broadcast** across every element of the array:

In [150…
```
my_matrix
```

Out[150]:
```
array([[4.28, 3.23, 5.87],
       [1.23, 5.32, 3.33]])
```

In [151…
```
10. * my_matrix
```

Out[151]:
```
array([[42.8, 32.3, 58.7],
       [12.3, 53.2, 33.3]])
```

In [152…
```
10. * my_matrix + 1.
```

Out[152]:
```
array([[43.8, 33.3, 59.7],
       [13.3, 54.2, 34.3]])
```

Arrays can also be broadcast:

In [153…
```
a = np.array([[1., 3., 2.], [5., 8., 7.], [3., 2., 1]])
a
```

Out[153]:
```
array([[1., 3., 2.],
       [5., 8., 7.],
       [3., 2., 1.]])
```

In [154…
```
b = np.array([[101.], [102.], [103.]])
b
```

Out[154]:
```
array([[101.],
       [102.],
       [103.]])
```

In [155…
```
a + b
```

Out[155]:
```
array([[102., 104., 103.],
       [107., 110., 109.],
       [106., 105., 104.]])
```

Transposing matrices

Matrices can be transposed (flipped on their side):

In [156…
```
A.T
```

Out[156]:
```
array([[ -1.54,  -0.98],
       [-23.02, -24.72]])
```

In [157…
```
my_matrix.T
```

Out[157]:
```
array([[4.28, 1.23],
       [3.23, 5.32],
       [5.87, 3.33]])
```

Matrix inverses

NumPy can also be used to invert a matrix, i.e. to find such a matrix that, when multiplied by the given matrix, will give the identity matrix:

In [158…
```
Ainv = np.linalg.inv(A)
Ainv
```

Out[158]:
```
array([[-1.59389266,  1.4842803 ],
       [ 0.0631883 , -0.0992959 ]])
```

In mathematics it is customary to denote the inverse of A by A^{-1}. Let us check that the product AA^{-1} is indeed equal to the identity matrix:

In [159…
```
np.dot(A, Ainv)
```

Out[159]:
```
array([[1., 0.],
       [0., 1.]])
```

The product $A^{-1}A$ should also give the identity:

In [160…
```
np.dot(Ainv, A)
```

Out[160]:
```
array([[ 1.00000000e+00, -7.10542736e-15],
       [-1.38777878e-17,  1.00000000e+00]])
```

(Close enough.)

The identity matrix

To obtain the identity matrix of a given size, we can use `np.eye`:

In [161…
```
np.eye(1)
```

Out[161]:
```
array([[1.]])
```

In [162…
```
np.eye(2)
```

Out[162]:
```
array([[1., 0.],
       [0., 1.]])
```

In [163…
```
np.eye(3)
```

Out[163]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Stacking NumPy arrays horizontally

Let's say that we have a 5 by 4 NumPy array,

In [164…
```
array1 = np.array([
    [10., 20., 30., 40.],
    [50., 60., 70., 80.],
    [90., 100., 110., 120.],
    [130., 140., 150., 160.],
    [170., 180., 190., 200.]])
array1
```

Out[164]:
```
array([[ 10.,  20.,  30.,  40.],
       [ 50.,  60.,  70.,  80.],
       [ 90., 100., 110., 120.],
       [130., 140., 150., 160.],
       [170., 180., 190., 200.]])
```

and another 5 by 3 NumPy array,

In [165…
```
array2 = np.array([
    [1., 2., 3.],
    [4., 5., 6.],
    [7., 8., 9.],
    [10., 11., 12.],
    [13., 14., 15.]])
array2
```

Out[165]:
```
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [ 7.,  8.,  9.],
       [10., 11., 12.],
       [13., 14., 15.]])
```

We can stack these two arrays horizontally:

In [166…
```
np.hstack((array1, array2))
```

Out[166]:
```
array([[ 10.,  20.,  30.,  40.,   1.,   2.,   3.],
       [ 50.,  60.,  70.,  80.,   4.,   5.,   6.],
       [ 90., 100., 110., 120.,   7.,   8.,   9.],
       [130., 140., 150., 160.,  10.,  11.,  12.],
       [170., 180., 190., 200.,  13.,  14.,  15.]])
```

Stacking NumPy arrays vertically

It is also possible to stack arrays vertically:

In [167…
```
array1 = np.array([
    [1., 2.],
    [3., 4.],
    [5., 6.]])
```

In [168…
```
array2 = np.array([
    [10., 20.]])
```

In [169…
```
array3 = np.array([
    [100., 200.],
    [300., 400.],
    [500., 600.],
    [700., 800.]])
```

In [170…
```
np.vstack((array1, array2, array3))
```

Out[170]:
```
array([[  1.,   2.],
       [  3.,   4.],
       [  5.,   6.],
       [ 10.,  20.],
       [100., 200.],
       [300., 400.],
       [500., 600.],
       [700., 800.]])
```

Indexing NumPy arrays

Let us consider, for example, the NumPy array

In [171…
```
a = np.array([
    [10., 20., 30., 40.],
    [50., 60., 70., 80.],
    [90., 100., 110., 120.],
    [130., 140., 150., 160.],
    [170., 180., 190., 200.]])
```

To index a particular element of this array, we can use

In [172…
```
a[0, 0]
```

Out[172]:
```
10.0
```

In [173…
```
a[2, 3]
```

Out[173]:
```
120.0
```

NumPy arrays are mutable, so we can overwrite the values:

In [174…
```
a[2, 3] = a[2, 3] * 10.
```

In [175…
```
a
```

Out[175]:
```
array([[  10.,   20.,   30.,   40.],
       [  50.,   60.,   70.,   80.],
       [  90.,  100.,  110., 1200.],
       [ 130.,  140.,  150.,  160.],
       [ 170.,  180.,  190.,  200.]])
```

It is possible to index an entire row

In [176…
```
a[3, :]
```

Out[176]:
```
array([130., 140., 150., 160.])
```

or column

In [177…
```
a[:, 3]
```

Out[177]:
```
array([  40.,   80., 1200.,  160.,  200.])
```

We can thus set each element in a particular row to a specific value:

In [178…
```
a[3, :] = 100.
```

In [179…
```
a
```

Out[179]:
```
array([[  10.,   20.,   30.,   40.],
       [  50.,   60.,   70.,   80.],
       [  90.,  100.,  110., 1200.],
       [ 100.,  100.,  100.,  100.],
       [ 170.,  180.,  190.,  200.]])
```

More complex indexing enables us to access subblocks of a NumPy array

In [180…
```
a[0:4, 1:3]
```

Out[180]:
```
array([[ 20.,  30.],
       [ 60.,  70.],
       [100., 110.],
       [100., 100.]])
```

In addition to the start index (inclusive) and end index (exclusive), we may also provide the step size:

In [181…
```
a[0:4:2, 1:3]
```

Out[181]:
```
array([[ 20.,  30.],
       [100., 110.]])
```

It is important that the result of **slicing** an array in this manner is a view of the original array:

In [182…
```
a
```

Out[182]:
```
array([[  10.,   20.,   30.,   40.],
       [  50.,   60.,   70.,   80.],
       [  90.,  100.,  110., 1200.],
       [ 100.,  100.,  100.,  100.],
       [ 170.,  180.,  190.,  200.]])
```

In [183…
```
b = a[0:4:2, 1:3]
```

In [184…
```
b
```

Out[184]:
```
array([[ 20.,  30.],
       [100., 110.]])
```

In [185…
```
b[:] = 1000.
```

In [186…
```
b
```

Out[186]:
```
array([[1000., 1000.],
       [1000., 1000.]])
```

In [187…
```
a
```

Out[187]:
```
array([[  10., 1000., 1000.,   40.],
       [  50.,   60.,   70.,   80.],
       [  90., 1000., 1000., 1200.],
       [ 100.,  100.,  100.,  100.],
       [ 170.,  180.,  190.,  200.]])
```

Notice that the original array has changed. To avoid thid, take a copy of the slice:

In [188…
```
b = a[0:4:2, 1:3].copy()
```

Boolean indexing

Just as `pandas` `DataFrame`s support boolean indexing, so do NumPy arrays:

In [189…
```
a > 5
```

Out[189]:
```
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

In [190…
```
a[a > 5]
```

Out[190]:
```
array([  10., 1000., 1000.,   40.,   50.,   60.,   70.,   80.,   90.,
       1000., 1000., 1200.,  100.,  100.,  100.,  100.,  170.,  180.,
        190.,  200.])
```

In [191…
```
a[a > 5] = 100.
```

In [192…
```
a
```

Out[192]:
```
array([[100., 100., 100., 100.],
       [100., 100., 100., 100.],
       [100., 100., 100., 100.],
       [100., 100., 100., 100.],
       [100., 100., 100., 100.]])
```

Comparing arrays

One should be careful when comparing NumPy arrays, being mindful that one is dealing with floating point data.

In [193…
```
a = np.random.random(1000000)
```

In [194…
```
np.all(3 * (5 * a) * (7 * a) == 105 * a * a)
```

Out[194]:
```
False
```

However, the maximum absolute difference between the corresponding elements is small:

In [195…
```
np.max(np.abs(3 * (5 * a) * (7 * a) - 105 * a * a))
```

Out[195]:
```
2.842170943040401e-14
```

Thus for most practical purposes the arrays are equal.

In [196…
```
np.max(np.abs(3 * (5 * a) * (7 * a) - 105 * a * a)) < 1e-12
```

Out[196]:
```
True
```

Some useful functions

Consider the matrix

In [197…
```
a = np.array([
    [3.29, 3.12, 9.49, 4.28, 9.93],
    [3.21, 9.93, 0.90, 8.90, 2.33],
    [5.32, 9.90, 1.23, 9.89, 2.39]
])
```

To find the cumulative sums:

In [198…
```
np.cumsum(a)
```

Out[198]:
```
array([ 3.29,  6.41, 15.9 , 20.18, 30.11, 33.32, 43.25, 44.15, 53.05,
       55.38, 60.7 , 70.6 , 71.83, 81.72, 84.11])
```

The cumulative products:

In [199…
```
np.cumprod(a)
```

Out[199]:
```
array([3.29000000e+00, 1.02648000e+01, 9.74129520e+01, 4.16927435e+02,
       4.14008943e+03, 1.32896871e+04, 1.31966592e+05, 1.18769933e+05,
       1.05705241e+06, 2.46293210e+06, 1.31027988e+07, 1.29717708e+08,
       1.59552781e+08, 1.57797700e+09, 3.77136504e+09])
```

The minimum:

In [200…
```
np.min(a)
```

Out[200]:
```
0.9
```

The maximum:

In [201…
```
np.max(a)
```

Out[201]:
```
9.93
```

The index at which the minimum is attained:

In [202…
```
np.argmin(a)
```

Out[202]:
```
7
```

The index at which the maximum is attained:

In [203…
```
np.argmax(a)
```

Out[203]:
```
4
```

The mean:

In [204…
```
np.mean(a)
```

Out[204]:
```
5.607333333333333
```

The variance:

In [205…
```
np.var(a)
```

Out[205]:
```
12.10776622222222
```

The standard deviation:

In [206…
```
np.std(a)
```

Out[206]:
```
3.4796215630758214
```

Applying a function to each row or column

Consider the matrix

In [207…
```
a = np.array([
    [3.29, 3.12, 9.49, 4.28, 9.93],
    [3.21, 9.93, 0.90, 8.90, 2.33],
    [5.32, 9.90, 1.23, 9.89, 2.39]
])
```

Often we need to apply a function to each row or to each column of a matrix. In these cases `np.apply_along_axis` comes in useful.

For example, to find the mean of each row we could use

In [208…
```
np.apply_along_axis(np.mean, 1, a)
```

Out[208]:
```
array([6.022, 5.054, 5.746])
```

To find the mean of each column

In [209…
```
np.apply_along_axis(np.mean, 0, a)
```

Out[209]:
```
array([3.94      , 7.65      , 3.87333333, 7.69      , 4.88333333])
```

Flags; making NumPy arrays immutable

Let's again consider the NumPy array

In [210…
```
a = np.array([
    [3.29, 3.12, 9.49, 4.28, 9.93],
    [3.21, 9.93, 0.90, 8.90, 2.33],
    [5.32, 9.90, 1.23, 9.89, 2.39]
])
```

Let's have a look at its `flags`:

In [211…
```
a.flags
```

Out[211]:
```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
```

In particular, we see that the array is stored row-by-row (`C_CONTIGUOUS`) as opposed to column-by-column. Its transpose view will naturally be `F_CONTIGUOUS` :

In [212…
```
a.T.flags
```

Out[212]:
```
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
```

Another useful flag is `WRITEABLE` . We may set it to `False` in order to make a NumPy array immutable. This is useful if we want to make sure that we don't accidentally change the values in the array, e.g. when we pass it round to functions:

In [213…
```
a[0, 0] = 9.23
```

In [214…
```
a.flags.writeable = False
```

Something like `a[0, 0] = 1.23` will now throw `ValueError: assignment destination is read-only` .

Of course, if someone *really* wants to modify the array, they can set

In [215…
```
a.flags.writeable = True
```

In [216…
```
a[0, 0] = 1.23
```

In [217…
```
a
```

Out[217]:
```
array([[1.23, 3.12, 9.49, 4.28, 9.93],
       [3.21, 9.93, 0.9 , 8.9 , 2.33],
       [5.32, 9.9 , 1.23, 9.89, 2.39]])
```

Finding out more about the configuration of the NumPy library

We can find out the version of the NumPy library using

In [218…
```
np.version.version
```

Out[218]:
```
'1.24.3'
```

To check which BLAS (Basic Linear Algebra Subprograms, a lower level library) is being used by NumPy, use

In [219…
```
np.show_config()
```

```
blas_armpl_info:
  NOT AVAILABLE
blas_mkl_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/Programs/Win64/Anaconda/V2023.09-0_3.11\\Library\\lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['C:/Programs/Win64/Anaconda/V2023.09-0_3.11\\Library\\includ
e']
blas_opt_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/Programs/Win64/Anaconda/V2023.09-0_3.11\\Library\\lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['C:/Programs/Win64/Anaconda/V2023.09-0_3.11\\Library\\includ
e']
lapack_armpl_info:
  NOT AVAILABLE
lapack_mkl_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/Programs/Win64/Anaconda/V2023.09-0_3.11\\Library\\lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['C:/Programs/Win64/Anaconda/V2023.09-0_3.11\\Library\\includ
e']
lapack_opt_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/Programs/Win64/Anaconda/V2023.09-0_3.11\\Library\\lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['C:/Programs/Win64/Anaconda/V2023.09-0_3.11\\Library\\includ
e']
Supported SIMD extensions in this NumPy install:
    baseline = SSE,SSE2,SSE3
    found = SSSE3,SSE41,POPCNT,SSE42,AVX,F16C,FMA3,AVX2
    not found = AVX512F,AVX512CD,AVX512_SKX,AVX512_CLX,AVX512_CNL,AVX512_ICL
```

Thus we see that Intel MKL BLAS routines are not being used, it's the OpenBLAS that we are using.

Exercise

Consider the matrix

In [220…
```
a = np.array([[3.89, 3.90, 1.39], [2.90, 4.32, 8.32], [5.32, 9.90, 8.98]])
a
```

Out[220]:
```
array([[3.89, 3.9 , 1.39],
       [2.9 , 4.32, 8.32],
       [5.32, 9.9 , 8.98]])
```

The elements 3.89, 4.32, and 8.98 constitute the **diagonal** of this matrix. "Bump" the diagonal (increase each entry in the diagonal) by 10.0.

Solution

In [221…
```
for i in range(np.shape(a)[0]):
    a[i, i] += 10.
```

In [222…
```
a
```

Out[222]:
```
array([[13.89,  3.9 ,  1.39],
       [ 2.9 , 14.32,  8.32],
       [ 5.32,  9.9 , 18.98]])
```

Exercise

Consider the matrix

In [223…
```
a = np.array([[3.89, 3.90, 1.39], [2.90, 4.32, 8.32], [5.32, 9.90, 8.98]])
a
```

Out[223]:
```
array([[3.89, 3.9 , 1.39],
       [2.9 , 4.32, 8.32],
       [5.32, 9.9 , 8.98]])
```

Save the diagonal of this matrix in a flat (one-dimensional) numpy array.

Solution

In [224…
```
d = np.empty((np.shape(a)[0],))
for i in range(np.shape(a)[0]):
    d[i] = a[i, i]
```

In [225…
```
d
```

Out[225]:
```
array([3.89, 4.32, 8.98])
```

Exercise

Stack the NumPy arrays to obtain a single matrix, where the top left block is the 3 by 3 identity matrix, the top right block the 3 by 4 zero matrix, the bottom left block the 4 by 3 matrix of ones, and the bottom right block the 4 by 4 matrix of 3s.

Solution

In [226…
```
tl = np.eye(3)
tr = np.zeros((3, 4))
bl = np.ones((4, 3))
br = np.ones((4, 4)) * 3
t = np.hstack((tl, tr))
b = np.hstack((bl, br))
a = np.vstack((t, b))
a
```

Out[226]:
```
array([[1., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0.],
       [1., 1., 1., 3., 3., 3., 3.],
       [1., 1., 1., 3., 3., 3., 3.],
       [1., 1., 1., 3., 3., 3., 3.],
       [1., 1., 1., 3., 3., 3., 3.]])
```

Exercise

What is the result of multiplying the matrix from the previous exercise by the vector (column matrix)

In [227…
```
v = np.array([[1.], [2.], [3.], [4.], [5.], [6.], [7.]])
```

Solution

In [228…
```
np.dot(a, v)
```

Out[228]:
```
array([[ 1.],
       [ 2.],
       [ 3.],
       [72.],
       [72.],
       [72.],
       [72.]])
```

Exercise

Consider the matrix

In [229…
```
a = np.array([
    [3.29, 3.12, 9.49, 4.28, 9.93],
    [3.21, 9.93, 0.90, 8.90, 2.33],
    [5.32, 9.90, 1.23, 9.89, 2.39]
])
```

Find the minimum in each even-indexed (0th, 2nd, etc.) column and the maximum in each odd-indexed (1st, 3rd, etc.) column.

Solution

In [230…
```
np.apply_along_axis(np.min, 0, a[:, 0::2])
```

Out[230]:
```
array([3.21, 0.9 , 2.33])
```

In [231…
```
np.apply_along_axis(np.max, 0, a[:, 1::2])
```

Out[231]:
```
array([9.93, 9.89])
```

Exercise

Use `np.zeros` and indexing to create the NumPy array

array([[0., 5., 0., 0., 0., 0., 0.], [8., 0., 5., 0., 0., 0., 0.], [0., 8., 0., 5., 0., 0., 0.], [0., 0., 8., 0., 5., 0., 0.], [0., 0., 0., 8., 0., 5., 0.], [0., 0., 0., 0., 8., 0., 5.], [0., 0., 0., 0., 0., 8., 0.]])

Solution

In [232…
```
a = np.zeros(49)
a[1::8] = 5.
a[7::8] = 8.
a.shape = (7, 7)
a
```

Out[232]:
```
array([[0., 5., 0., 0., 0., 0., 0.],
       [8., 0., 5., 0., 0., 0., 0.],
       [0., 8., 0., 5., 0., 0., 0.],
       [0., 0., 8., 0., 5., 0., 0.],
       [0., 0., 0., 8., 0., 5., 0.],
       [0., 0., 0., 0., 8., 0., 5.],
       [0., 0., 0., 0., 0., 8., 0.]])
```