Applied Data Science

L3. Data Science toolkit. Part 1

Irina Mohorianu

Head of Bioinformatics/ Scientific Computing @CSCI

Quick overview of your first steps in Python

- Basic types
- Control Flow
  - Conditional statements
  - Iterating through elements
- Variables and Functions
- Inputs and outputs
- Data visualisation

This is intended as a quick recap for info already presented in other modules.
We are making sure we are all on the same page.

Suggested resources:
http://scipy-lectures.org/
https://scikit-learn.org/
https://pandas.pydata.org/

Why Python?

Python is an imperative **programming language**
[other examples C/ C++, Fortran, Java, Scala, C#, Perl]
[a] is *interpreted* (as opposed to *compiled*) language.
Python can be used **interactively** - which commands and scripts can be executed.
Python is based on OOP (object-oriented paradigm),
with dynamic typing (the same variable may refer to objects of different types during execution)

[b] is released under an **open-source** licence i.e. code can be used and distributed free of charge

[c] **multi-platform** (portable across platforms)
Python is available for all OSs, Windows, Linux/Unix, MacOS X

[d] Python is a **readable language** with clear non-verbose syntax
and easy to integrate/ interface with other languages, in particular C and C++.

[e] benefits from a large variety of **high-quality (maintained) packages** (from web frameworks to scientific computing).

Python. Basic types

Python supports the following numerical, scalar types:

Integer:

```
>>> 1 + 1
2
>>> a = 4
>>> type(a)
<type 'int'>
```

Floats:

```
>>> c = 2.1
>>> type(c)
<type 'float'>
```

Complex:

```
>>> a = 1.5 + 0.5j
>>> a.real
1.5
>>> a.imag
0.5
>>> type(1. + 0j)
<type 'complex'>
```

Booleans:

```
>>> 3 > 4
False
>>> test = (3 > 4)
>>> test
False
>>> type(test)
<type 'bool'>
```

Basic arithmetic operators:
+ - * / %

Beware of differences between Python2 and Python 3

⚠ Integer division

In Python 2:

```
>>> 3 / 2
1
```

In Python 3:

```
>>> 3 / 2
1.5
```

To be safe: use floats:

```
>>> 3 / 2.
1.5
>>> a = 3
>>> b = 2
>>> a / b # In Python 2
1
>>> a / float(b)
1.5
```

If you explicitly want integer division use `//`:

```
>>> 3.0 // 2
1.0
```

Python. Basic types. Containers

Lists

A list is an ordered collection of objects, that may have different types. For example:

```
>>> colors = ['red', 'blue', 'green', 'black', 'white']
>>> type(colors)
<type 'list'>
```

Indexing: accessing individual objects contained in the list:

```
>>> colors[2]
'green'
```

Counting from the end with negative indices:

```
>>> colors[-1]
'white'
>>> colors[-2]
'black'
```

Slicing: obtaining sublists of regularly-spaced elements:

```
>>> colors
['red', 'blue', 'green', 'black', 'white']
>>> colors[2:4]
['green', 'black']
```

⚠ **Indexing starts at 0** (as in C), not at 1 (as in Fortran or Matlab)!

Python. Basic types. Containers

⚠ Note that `colors[start:stop]` contains the elements with indices `i` such as `start<= i < stop` (`i` ranging from `start` to `stop-1`). Therefore, `colors[start:stop]` has `(stop - start)` elements.

Slicing syntax: `colors[start:stop:stride]`

All slicing parameters are optional:

```
>>> colors
['red', 'blue', 'green', 'black', 'white']
>>> colors[3:]
['black', 'white']
>>> colors[:3]
['red', 'blue', 'green']
>>> colors[::2]
['red', 'green', 'white']
```

Lists are *mutable* objects and can be modified:

```
>>> colors[0] = 'yellow'
>>> colors
['yellow', 'blue', 'green', 'black', 'white']
>>> colors[2:4] = ['gray', 'purple']
>>> colors
['yellow', 'blue', 'gray', 'purple', 'white']
```

Note: The elements of a list may have different types:

```
>>> colors = [3, -200, 'hello']
>>> colors
[3, -200, 'hello']
>>> colors[1], colors[2]
(-200, 'hello')
```

For collections of numerical data that all have the same type, it is often **more efficient** to use the `array` type provided by the `numpy` module. A NumPy array is a chunk of memory containing fixed-sized items. With NumPy arrays, operations on elements can be faster because elements are regularly spaced in memory and more operations are performed through specialized C functions instead of Python loops.

Python. Basic types. Containers

Add and remove elements:

```
>>> colors = ['red', 'blue', 'green', 'black', 'white']
>>> colors.append('pink')
>>> colors
['red', 'blue', 'green', 'black', 'white', 'pink']
>>> colors.pop() # removes and returns the last item
'pink'
>>> colors
['red', 'blue', 'green', 'black', 'white']
>>> colors.extend(['pink', 'purple']) # extend colors, in-place
>>> colors
['red', 'blue', 'green', 'black', 'white', 'pink', 'purple']
>>> colors = colors[:-2]
>>> colors
['red', 'blue', 'green', 'black', 'white']
```

Concatenate and repeat lists:

```
>>> rcolors + colors
['white', 'black', 'green', 'blue', 'red', 'red', 'blue', 'green', 'black',
     'white']
>>> rcolors * 2
['white', 'black', 'green', 'blue', 'red', 'white', 'black', 'green', 'blue',
     'red']
```

Sort:

```
>>> sorted(rcolors) # new object
['black', 'blue', 'green', 'red', 'white']
>>> rcolors
['white', 'black', 'green', 'blue', 'red']
>>> rcolors.sort()  # in-place
>>> rcolors
['black', 'blue', 'green', 'red', 'white']
```

Reverse:

```
>>> rcolors = colors[::-1]
>>> rcolors
['white', 'black', 'green', 'blue', 'red']
>>> rcolors2 = list(colors) # new object that is a copy of colors in a dif-
      ferent memory area
>>> rcolors2
['red', 'blue', 'green', 'black', 'white']
>>> rcolors2.reverse() # in-place; reversing rcolors2 does not affect colors
>>> rcolors2
['white', 'black', 'green', 'blue', 'red']
```

Python. Basic types. Containers

Different string syntaxes (simple, double or triple quotes):

```
s = 'Hello, how are you?'
s = "Hi, what's up"
s = '''Hello,
       how are you'''          #
                               #

s = """Hi,
what's up?"""
```

Indexing:

```
>>> a = "hello"
>>> a[0]
'h'
>>> a[1]
'e'
>>> a[-1]
'o'
```

(Remember that negative indices correspond to counting from the right end.)

Slicing:

```
>>> a = "hello, world!"
>>> a[3:6] # 3rd to 6th (excluded) elements: elements 3, 4, 5
'lo,'
>>> a[2:10:2] # Syntax: a[start:stop:step]
'lo o'
>>> a[::3] # every three characters, from beginning to end
'hl r!'
```

Accents and special characters can also be handled as in Python 3 strings consist of Unicode characters.

A string is an **immutable object** and it is not possible to modify its contents. One may however create new strings from the original one.

Python. Basic types. Containers

Slicing:

```
>>> a = "hello, world!"
>>> a[3:6] # 3rd to 6th (excluded) elements: elements 3, 4, 5
'lo,'
>>> a[2:10:2] # Syntax: a[start:stop:step]
'lo o'
>>> a[::3] # every three characters, from beginning to end
'hl r!'
```

Accents and special characters can also be handled as in Python 3 strings consist of Unicode characters.

A string is an **immutable object** and it is not possible to modify its contents. One may however create new strings from the original one.

```
In [53]: a = "hello, world!"
In [54]: a[2] = 'z'
---------------------------------------------------------------------------
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
In [55]: a.replace('l', 'z', 1)
Out[55]: 'hezlo, world!'
In [56]: a.replace('l', 'z')
Out[56]: 'hezzo, worzd!'
```

Python. Basic types. Containers

A dictionary is basically an efficient table that **maps keys to values**. It is an **unordered** container

```
>>> tel = {'emmanuelle': 5752, 'sebastian': 5578}
>>> tel['francis'] = 5915
>>> tel
{'sebastian': 5578, 'francis': 5915, 'emmanuelle': 5752}
>>> tel['sebastian']
5578
>>> tel.keys()
['sebastian', 'francis', 'emmanuelle']
>>> tel.values()
[5578, 5915, 5752]
>>> 'francis' in tel
True
```

It can be used to conveniently store and retrieve values associated with a name (a string for a date, a name, etc.). See https://docs.python.org/tutorial/datastructures.html#dictionaries for more information.
A dictionary can have keys (resp. values) with different types:

```
>>> d = {'a':1, 'b':2, 3:'hello'}
>>> d
{'a': 1, 3: 'hello', 'b': 2}
```

Python. Basic types. Containers

Tuples

Tuples are basically immutable lists. The elements of a tuple are written between parentheses, or just separated by commas:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> u = (0, 2)
```

Sets: unordered, unique items:

```
>>> s = set(('a', 'b', 'c', 'a'))
>>> s
set(['a', 'c', 'b'])
>>> s.difference(('a', 'b'))
set(['c'])
```

Python. Basic types. NumPy

NumPy (short for Numerical Python) was created in 2005 by merging Numarray into Numeric.

NumPy library has evolved into an essential library for scientific computing in Python
i.e. a building block of many other scientific libraries, such as SciPy, Scikit-learn, Pandas, and others.

What makes NumPy so very attractive to the scientific community is the convenient Python interface for working with multi-dimensional array data structures efficiently; t

The NumPy array data structure is also called ndarray, which is short for *n*-dimensional array.

NumPy is built around **ndarrays** objects - high-performance multi-dimensional array data structures.

a one-dimensional NumPy array is a data structure representing a vector of elements
i.e. a fixed-size Python list where all elements share the same type.

A two-dimensional array as a data structure to represent a matrix or a Python list of lists.

NumPy arrays can have up to 32 dimensions if it was compiled without alterations to the source code

Python. Basic types. NumPy

In:

```
a = [1., 2., 3.]
np.array(a)
```

Out:

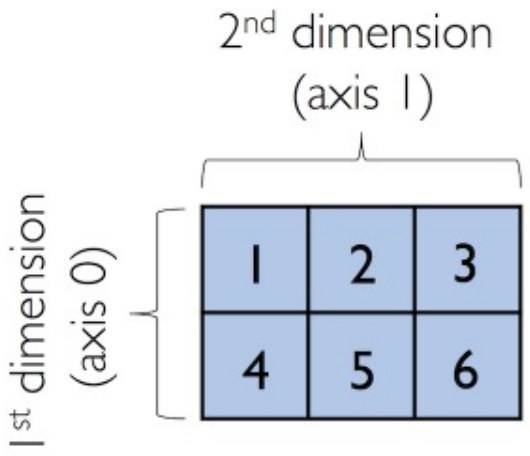
```
array([1., 2., 3.])
```

In:

```
lst = [[1, 2, 3],
       [4, 5, 6]]
ary2d = np.array(lst)
ary2d

# rows x columns
```

Out:

```
array([[1, 2, 3],
       [4, 5, 6]])
```



In:

```
np.ones((3, 4), dtype=np.int)
```

Out:

```
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]])
```

In:

```
np.zeros((3, 3))
```

Out:

```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

In:

```
np.zeros((3, 3)) + 99
```

Out:

```
array([[99., 99., 99.],
       [99., 99., 99.],
       [99., 99., 99.]])
```

Python. Basic types. NumPy

In:

```
np.eye(3)
```

Out:

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

In:

```
np.diag((1, 2, 3))
```

Out:

```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

In:

```
np.arange(5)
```

Out:

```
array([0, 1, 2, 3, 4])
```

In:

```
np.arange(1., 11., 0.1)
```

Out:

```
array([ 1. ,  1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9,  2. ,
        2.1,  2.2,  2.3,  2.4,  2.5,  2.6,  2.7,  2.8,  2.9,  3. ,  3.1,
        3.2,  3.3,  3.4,  3.5,  3.6,  3.7,  3.8,  3.9,  4. ,  4.1,  4.2,
        4.3,  4.4,  4.5,  4.6,  4.7,  4.8,  4.9,  5. ,  5.1,  5.2,  5.3,
        5.4,  5.5,  5.6,  5.7,  5.8,  5.9,  6. ,  6.1,  6.2,  6.3,  6.4,
        6.5,  6.6,  6.7,  6.8,  6.9,  7. ,  7.1,  7.2,  7.3,  7.4,  7.5,
        7.6,  7.7,  7.8,  7.9,  8. ,  8.1,  8.2,  8.3,  8.4,  8.5,  8.6,
        8.7,  8.8,  8.9,  9. ,  9.1,  9.2,  9.3,  9.4,  9.5,  9.6,  9.7,
        9.8,  9.9, 10. , 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8,
       10.9])
```

Python. Basic types. NumPy
Access elements

In:

```
ary = np.array([1, 2, 3])
ary[0]
```

Out:

```
1
```

In:

```
ary[0:3] # equivalent to ary[0:2]
```

Out:

```
array([1, 2, 3])
```

In:

```
ary = np.array([[1, 2, 3],
                [4, 5, 6]])

ary[0, -2] # first row, second from last element
```

Out:

```
2
```

In:

```
ary[-1, -1] # lower right
```

Out:

```
6
```

In:

```
ary[1, 1] # first row, second column
```

Out:

```
5
```

Python. Basic types. NumPy
Keep it simple.

In:

```
lst = [[1, 2, 3],
       [4, 5, 6]] # 2d array

for row_idx, row_val in enumerate(lst):
    for col_idx, col_val in enumerate(row_val):
        lst[row_idx][col_idx] += 1
lst
```

Out:

```
[[2, 3, 4], [5, 6, 7]]
```

In:

```
lst = [[1, 2, 3], [4, 5, 6]]
[[cell + 1 for cell in row] for row in lst]
```

Out:

```
[[2, 3, 4], [5, 6, 7]]
```

In:

```
ary = np.array([[1, 2, 3], [4, 5, 6]])
ary = np.add(ary, 1) # binary ufunc
ary
```

Out:

```
array([[2, 3, 4],
       [5, 6, 7]])
```

The ufuncs for basic arithmetic operations are add, subtract, divide, multiply, power, and exp (exponential). However, NumPy uses operator overloading so that we can use mathematical operators (+, -, /, *, and **) directly

In:

```
np.add(ary, 1)
```

Out:

```
array([[3, 4, 5],
       [6, 7, 8]])
```

In:

```
ary + 1
```

Out:

```
array([[3, 4, 5],
       [6, 7, 8]])
```

Python. Basic types. NumPy

Other useful unary ufuncs are:

np.mean (computes arithmetic mean or average)
np.std (computes the standard deviation)
np.var (computes variance)
np.sort (sorts an array)
np.argsort (returns indices that would sort an array)
np.min (returns the minimum value of an array)
np.max (returns the maximum value of an array)
np.argmin (returns the index of the minimum value)
np.argmax (returns the index of the maximum value)
np.array_equal (checks if two arrays have the same shape and elements)

Python. Basic types. NumPy broadcasting

`np.array([1, 2, 3]) + 1:`

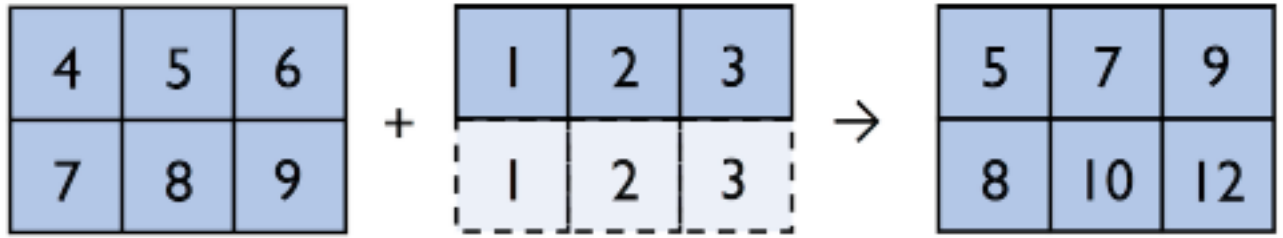
In:

```
ary = np.array([1, 2, 3])
ary + 1
```

Out:

```
array([2, 3, 4])
```

`np.array([[4, 5, 6], [7, 8, 9]]) + np.array([1, 2, 3]):`



In:

```
ary2 = np.array([[4, 5, 6],
                 [7, 8, 9]])


ary2 + ary
```

Out:

```
array([[ 5,  7,  9],
       [ 8, 10, 12]])
```

Python. Control Flow. Conditional statements

```
>>> if 2**2 == 4:
...     print('Obvious!')
...
Obvious!
```

Blocks are delimited by indentation

Type the following lines in your Python interpreter, and be careful to **respect the indentation depth**. The Ipython shell automatically increases the indentation depth after a colon `:` sign; to decrease the indentation depth, go four spaces to the left with the Backspace key. Press the Enter key twice to leave the logical block.

```
>>> a = 10
>>> if a == 1:
...     print(1)
... elif a == 2:
...     print(2)
... else:
...     print('A lot')
A lot
```

Indentation is compulsory in scripts as well. As an exercise, re-type the previous lines with the same indentation in a script `condition.py`, and execute the script with `run condition.py` in Ipython.

Python. Control Flow.
Iterating with an index (for)

Iterating with an index:

```
>>> for i in range(4):
...     print(i)
0
1
2
3
```

But most often, it is more readable to iterate over values:

```
>>> for word in ('cool', 'powerful', 'readable'):
...     print('Python is %s' % word)
Python is cool
Python is powerful
Python is readable
```

Python. Control Flow
Iterating on conditions

Typical C-style while loop (Mandelbrot problem):

```
>>> z = 1 + 1j
>>> while abs(z) < 100:
...     z = z**2 + 1
>>> z
(-134+352j)
```

More advanced features

`break` out of enclosing for/while loop:

```
>>> z = 1 + 1j
>>> while abs(z) < 100:
...     if z.imag == 0:
...         break
...     z = z**2 + 1
```

`continue` the next iteration of a loop.:

```
>>> a = [1, 0, 2, 4]
>>> for element in a:
...     if element == 0:
...         continue
...     print(1. / element)
1.0
0.5
0.25
```

Python. Control Flow

`if <OBJECT>:`	Evaluates to False: • any number equal to zero (0, 0.0, 0+0j) • an empty container (list, tuple, set, dictionary, …) • `False`, `None` Evaluates to True: • everything else

`a == b:`	Tests equality, with logics:
	`>>> 1 == 1.`
	`True`

`a is b:`	Tests identity: both sides are the same object:
	`>>> 1 is 1.`
	`False`
	`>>> a = 1`
	`>>> b = 1`
	`>>> a is b`
	`True`

`a in b:`	For any collection `b`: `b` contains `a`
	`>>> b = [1, 2, 3]`
	`>>> 2 in b`
	`True`
	`>>> 5 in b`
	`False`
	
	If `b` is a dictionary, this tests that `a` is a key of `b`.

Python. Variables and functions

```
In [56]: def test():
   ....:     print('in test function')
   ....:
   ....:
In [57]: test()
in test function
```

⚠ Function blocks must be indented as other control-flow blocks.

Functions can *optionally* return values.

```
In [6]: def disk_area(radius):
   ...:     return 3.14 * radius * radius
   ...:
In [8]: disk_area(1.5)
Out[8]: 7.0649999999999995
```

Note: By default, functions return `None`.

Note: Note the syntax to define a function:

- the `def` keyword;
- is followed by the function's **name**, then
- the arguments of the function are given between parentheses followed by a colon.
- the function body;
- and `return object` for optionally returning values.

Mandatory parameters (positional arguments)

```
In [81]: def double_it(x):
   ....:     return x * 2
   ....:
In [82]: double_it(3)
Out[82]: 6
In [83]: double_it()
---------------------------------------------------------------
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: double_it() takes exactly 1 argument (0 given)
```

Optional parameters (keyword or named arguments)

```
In [84]: def double_it(x=2):
   ....:     return x * 2
   ....:
In [85]: double_it()
Out[85]: 4
In [86]: double_it(3)
Out[86]: 6
```

Keyword arguments allow you to specify *default values*.

Python. Variables and functions
Parameters transferred by value/ reference

```
>>> def try_to_modify(x, y, z):
...     x = 23
...     y.append(42)
...     z = [99] # new reference
...     print(x)
...     print(y)
...     print(z)
...
>>> a = 77    # immutable variable
>>> b = [99]  # mutable variable
>>> c = [28]
>>> try_to_modify(a, b, c)
23
[99, 42]
[99]
>>> print(a)
77
>>> print(b)
[99, 42]
>>> print(c)
[28]
```

```
In [117]: def setx(y):
   .....:     x = y
   .....:     print('x is %d' % x)
   .....:
   .....:
In [118]: setx(10)
x is 10
In [120]: x
Out[120]: 5
```

```
In [121]: def setx(y):
   .....:     global x
   .....:     x = y
   .....:     print('x is %d' % x)
   .....:
   .....:
In [122]: setx(10)
x is 10
In [123]: x
Out[123]: 10
```

```
In [114]: x = 5
In [115]: def addx(y):
   .....:     return x + y
   .....:
In [116]: addx(10)
Out[116]: 15
```

Python. Variables and functions

Special forms of parameters:

- `*args`: any number of positional arguments packed into a tuple
- `**kwargs`: any number of keyword arguments packed into a dictionary

```
In [35]: def variable_args(*args, **kwargs):
   ....:     print('args is', args)
   ....:     print('kwargs is', kwargs)
   ....:
In [36]: variable_args('one', 'two', x=1, y=2, z=3)
args is ('one', 'two')
kwargs is {'x': 1, 'y': 2, 'z': 3}
```

Documentation about what the function does and its parameters. General convention:

```
In [67]: def funcname(params):
   ....:     """Concise one-line sentence describing the function.
   ....:
   ....:     Extended summary which can contain multiple paragraphs.
   ....:     """
   ....:     # function body
   ....:     pass
   ....:
In [68]: funcname?
Type:           function
Base Class:     type 'function'>
String Form:    <function funcname at 0xeaa0f0>
Namespace:      Interactive
File:           <ipython console>
Definition:     funcname(params)
Docstring:
    Concise one-line sentence describing the function.
    Extended summary which can contain multiple paragraphs.
```

Python. Inputs and Outputs

```
>>> f = open('workfile', 'w') # opens the workfile file
>>> type(f)
<type 'file'>
>>> f.write('This is a test \nand another test')
>>> f.close()
```

To read from a file

```
In [1]: f = open('workfile', 'r')
In [2]: s = f.read()
In [3]: print(s)
This is a test
and another test
In [4]: f.close()
```

```
In [6]: f = open('workfile', 'r')
In [7]: for line in f:
   ...:     print(line)
   ...:
This is a test
and another test
In [8]: f.close()
```

File modes

- Read-only: `r`
- Write-only: `w`
  - Note: Create a new file or *overwrite* existing file.
- Append a file: `a`
- Read and Write: `r+`
- Binary mode: `b`
  - Note: Use for binary files, especially on Windows.

Python. Exception handling
easier to ask for forgiveness than permission

Exceptions are raised by errors in Python:

```
In [1]: 1/0
---------------------------------------------------------------------------
ZeroDivisionError: integer division or modulo by zero
In [2]: 1 + 'e'
---------------------------------------------------------------------------
TypeError: unsupported operand type(s) for +: 'int' and 'str'
In [3]: d = {1:1, 2:2}
In [4]: d[3]
---------------------------------------------------------------------------
KeyError: 3
In [5]: l = [1, 2, 3]
In [6]: l[4]
---------------------------------------------------------------------------
IndexError: list index out of range
In [7]: l.foobar
---------------------------------------------------------------------------
AttributeError: 'list' object has no attribute 'foobar'
```

```
In [10]: while True:
    ....:     try:
    ....:         x = int(raw_input('Please enter a number: '))
    ....:         break
    ....:     except ValueError:
    ....:         print('That was no valid number.  Try again...')
    ....:
Please enter a number: a
That was no valid number.  Try again...
Please enter a number: 1
In [9]: x
Out[9]: 1
```

```
In [10]: try:
    ....:     x = int(raw_input('Please enter a number: '))
    ....: finally:
    ....:     print('Thank you for your input')
    ....:
    ....:
Please enter a number: a
Thank you for your input
---------------------------------------------------------------------------
ValueError: invalid literal for int() with base 10: 'a'
```

Important for resource management (e.g. closing a file)

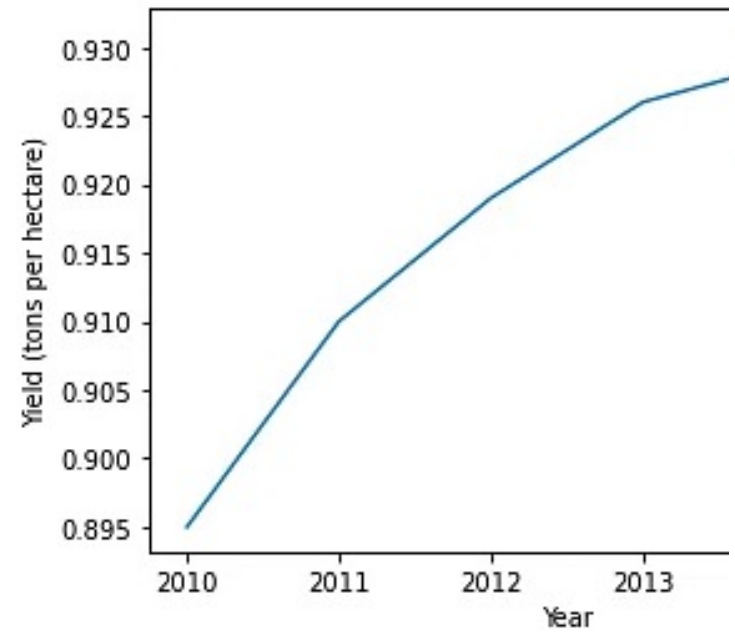
Python. Data visualisation. Matplotlib and Seaborn

Matplotlib	Seaborn
It is used for basic graph plotting like line charts, bar graphs and others	It is mainly used for statistics visualization and can perform complex visualizations with fewer commands.
It mainly works with datasets and arrays.	It works with entire datasets.
Matplotlib acts productively with data arrays and frames. It regards the aces and figures as objects.	Seaborn is considerably more organized and functional than Matplotlib and treats the entire dataset as a solitary unit.
Matplotlib is more customizable and pairs well with Pandas and Numpy for Exploratory Data Analysis.	Seaborn has more inbuilt themes and is mainly used for statistical analysis.

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

Python. Data visualisation
Line plots. Matplotlib

```
plt.plot(years, yield_apples)
plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')
```




```
years = range(2000, 2012)
apples = [0.895, 0.91, 0.919, 0.926, 0.929, 0.931, 0.934, 0.936, 0.937, 0.9375, 0.9372, 0.939]
oranges = [0.962, 0.941, 0.930, 0.923, 0.918, 0.908, 0.907, 0.904, 0.901, 0.898, 0.9, 0.896, ]
```

```
plt.plot(years, apples)
plt.plot(years, oranges)
plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)');
```

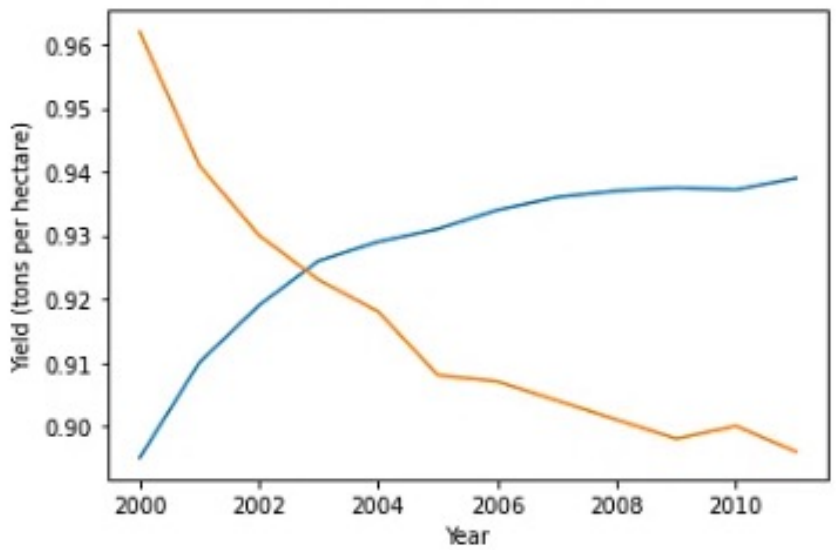
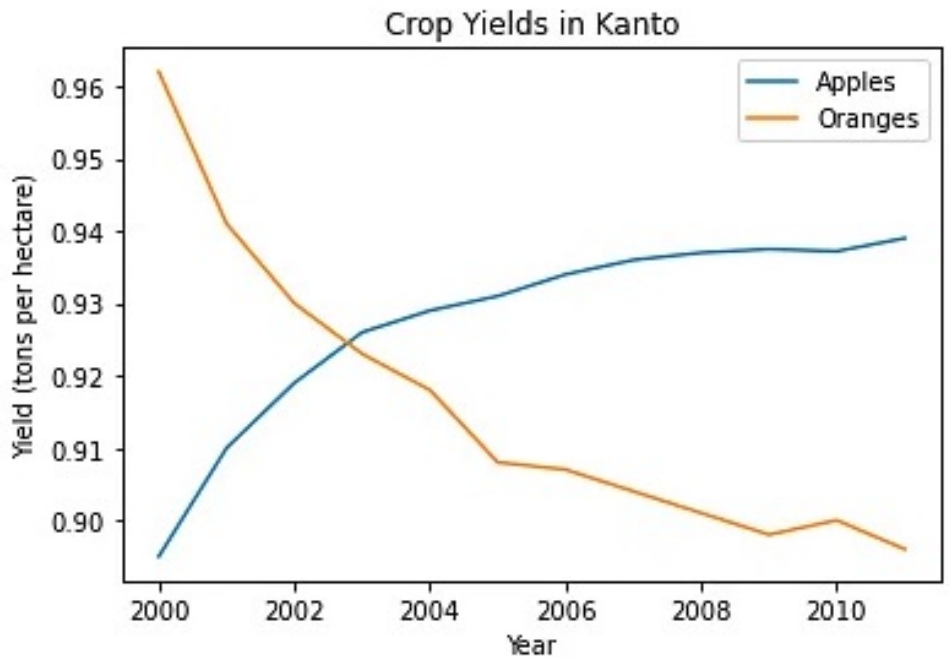


Figure 5: Axis with labels

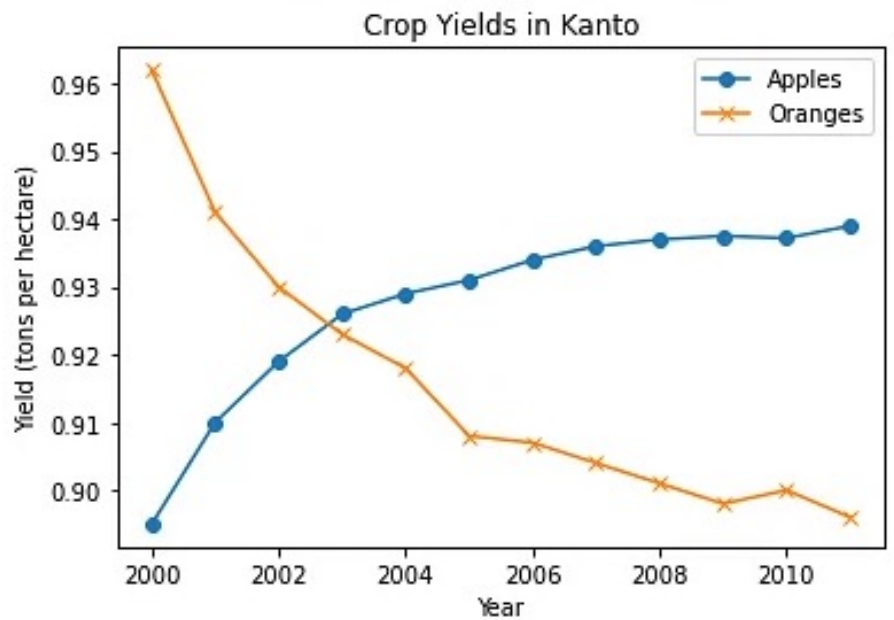
Python. Data visualisation.
Line plots. Matplotlib

```
plt.plot(years, apples)
plt.plot(years, oranges)

plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')

plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges']);
```




```
plt.plot(years, apples, marker='o')
plt.plot(years, oranges, marker='x')

plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')

plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges'])
```

```
<matplotlib.legend.Legend at 0x2054a8940d0>
```

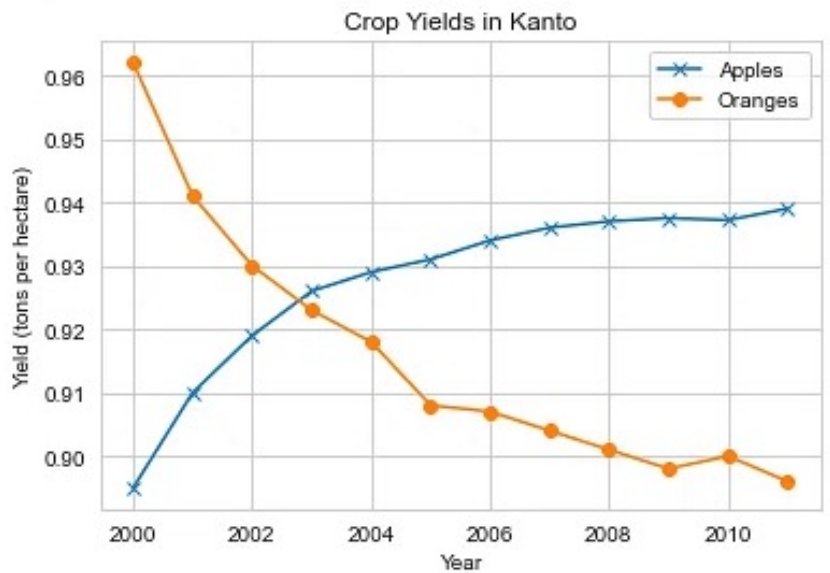


Python. Data visualisation
Line plots. Seaborn

```
sns.set_style("whitegrid")
```

```
plt.plot(years, apples, marker = 'x')
plt.plot(years, oranges, marker = 'o')

plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')

plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges'])
```

```
<matplotlib.legend.Legend at 0x2054aa1d1f0>
```

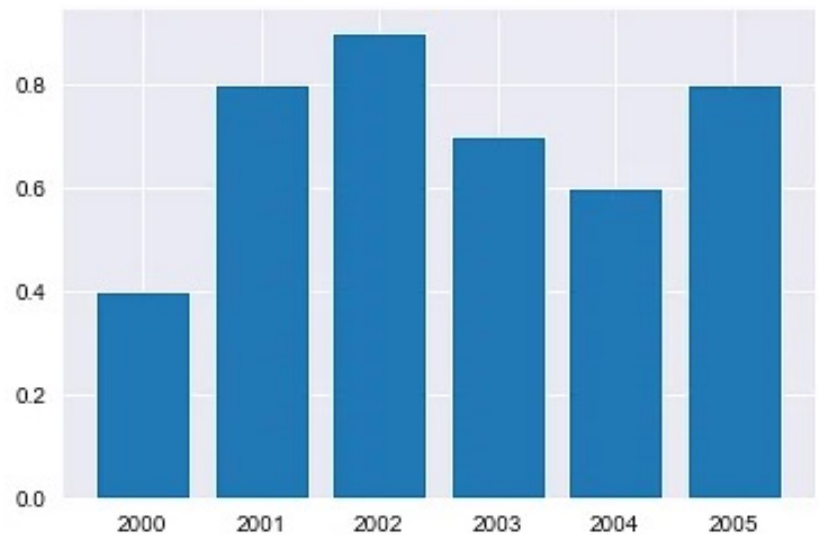


Python. Data visualisation
Bar plots

```
years = range(2000, 2006)
apples = [0.35, 0.6, 0.9, 0.8, 0.65, 0.8]
oranges = [0.4, 0.8, 0.9, 0.7, 0.6, 0.8]
```

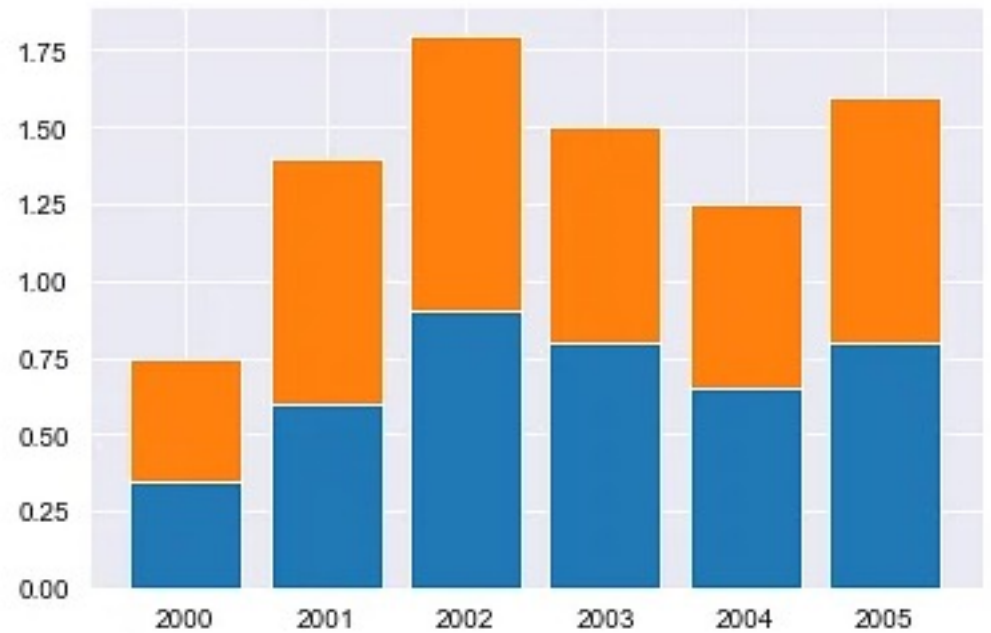
```
plt.bar(years, oranges)

plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')

plt.title("Crop Yields in Kanto")
```

```
<BarContainer object of 6 artists>
```




```
plt.bar(years, apples)
plt.bar(years, oranges, bottom=apples)
```

```
<BarContainer object of 6 artists>
```



Python. Data visualisation
Bar plots

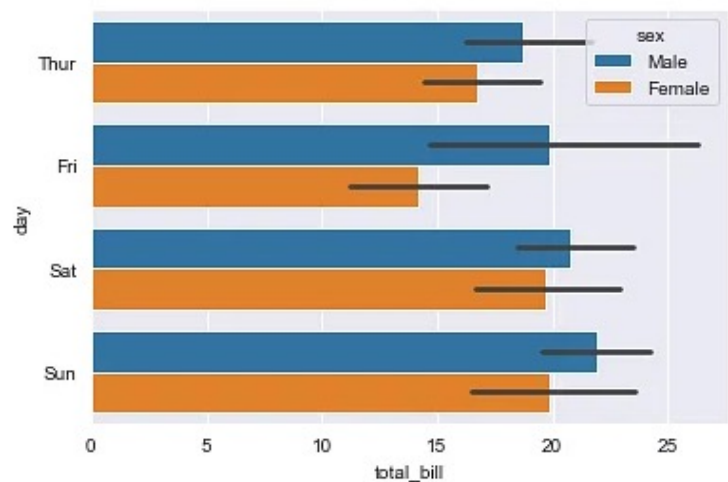
```
tips_df = sns.load_dataset("tips")
tips_df
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	2						
4	2						
...							
239	2						
240	2						
241	2						
242	1						
243	1						

244 rows

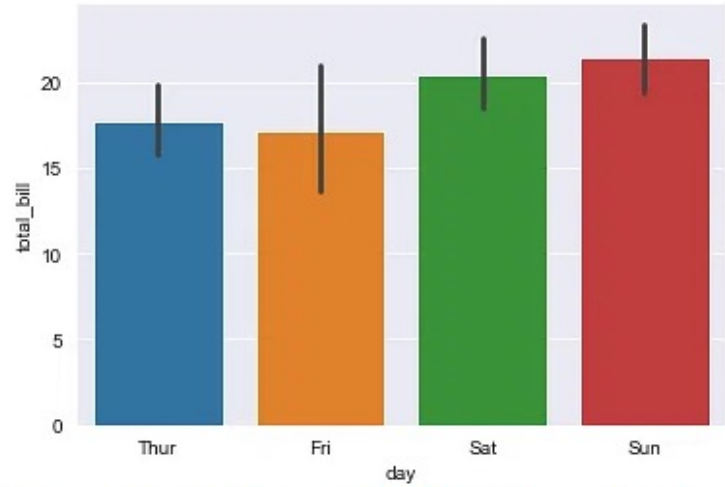
```
sns.barplot(x='total_bill', y='day', hue='sex', data=tips_df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2054af1b250>
```



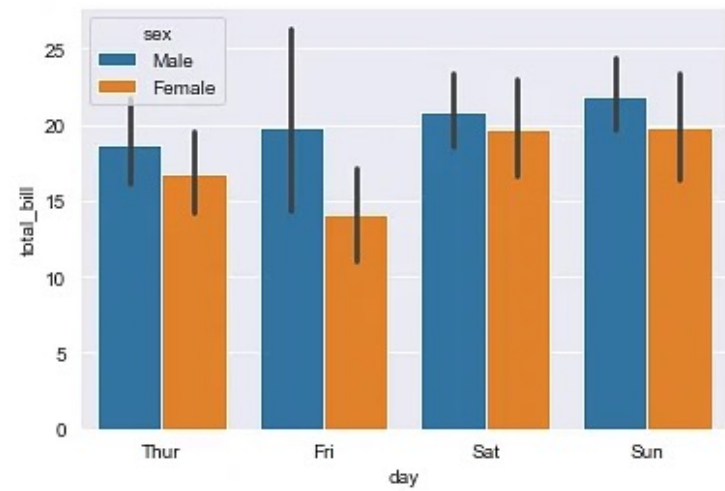

```
sns.barplot(x='day', y='total_bill', data=tips_df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2054afa7370>
```




```
sns.barplot(x='day', y='total_bill', hue='sex', data=tips_df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2054b046700>
```

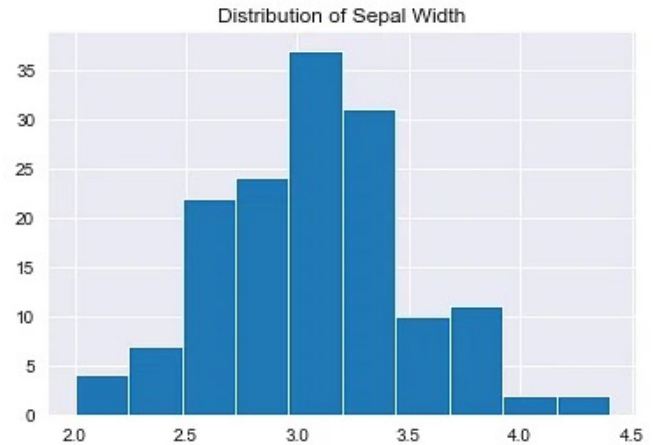


Python. Data visualisation Histograms

```
flowers_df = sns.load_dataset("iris")
flowers_df.sepal_width
```

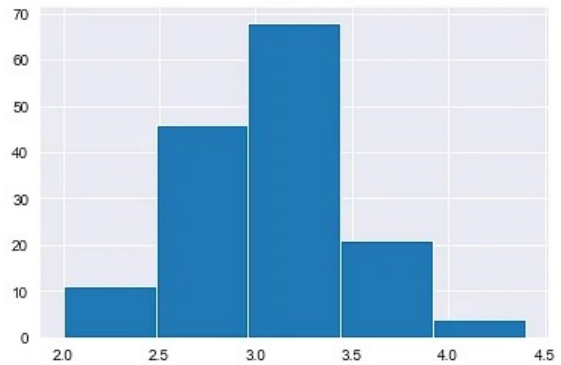
```
0      3.5
1      3.0
2      3.2
3      3.1
4      3.6
      ...
145    3.0
146    2.5
147    3.0
148    3.4
149    3.0
Name: sepal_width, Length: 150, dtype: float64
```

```
plt.title("Distribution of Sepal Width")
plt.hist(flowers_df.sepal_width)
```




```
# Specifying the number of bins
plt.hist(flowers_df.sepal_width, bins=5)
```

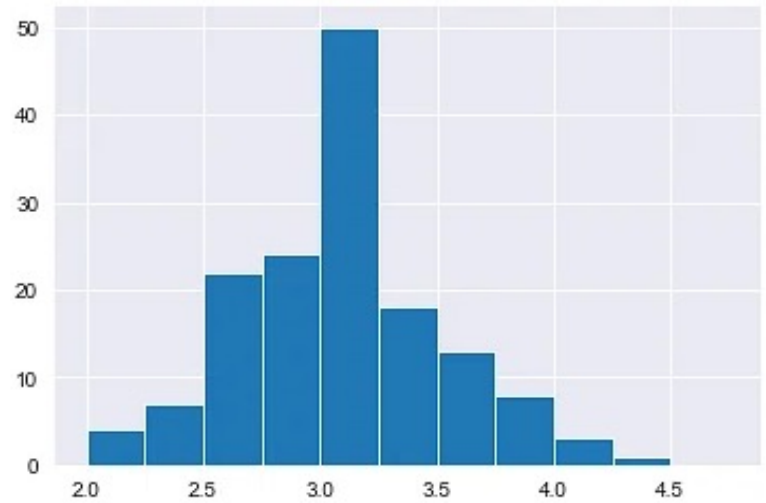
```
(array([11., 46., 68., 21.,  4.]),
 array([2.  , 2.48, 2.96, 3.44, 3.92, 4.4 ]),
 <a list of 5 Patch objects>)
```



Python. Data visualisation
Histograms

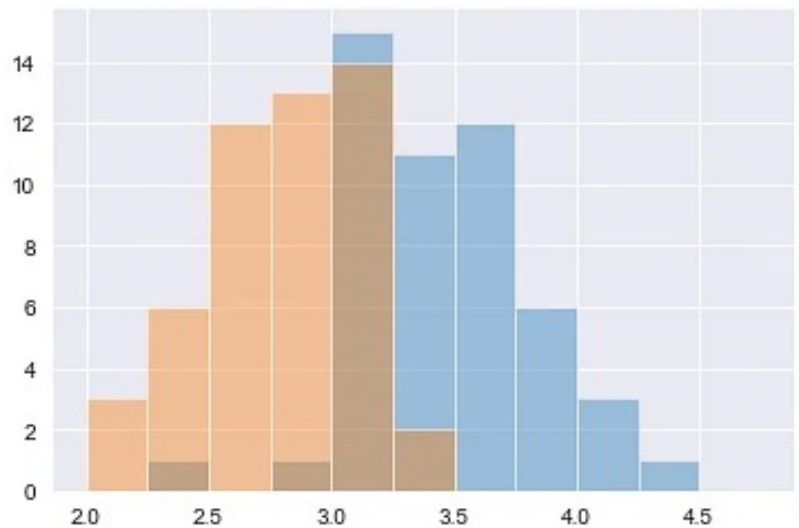
```
import numpy as np

# Specifying the boundaries of each bin
plt.hist(flowers_df.sepal_width, bins=np.arange(2, 5, 0.25))
```

```
(array([ 4.,  7., 22., 24., 50., 18., 13.,  8.,  3.,  1.,  0.]),
 array([2.  , 2.25, 2.5 , 2.75, 3.  , 3.25, 3.5 , 3.75, 4.  , 4.25, 4.5 ,
        4.75]),
 <a list of 11 Patch objects>)
```



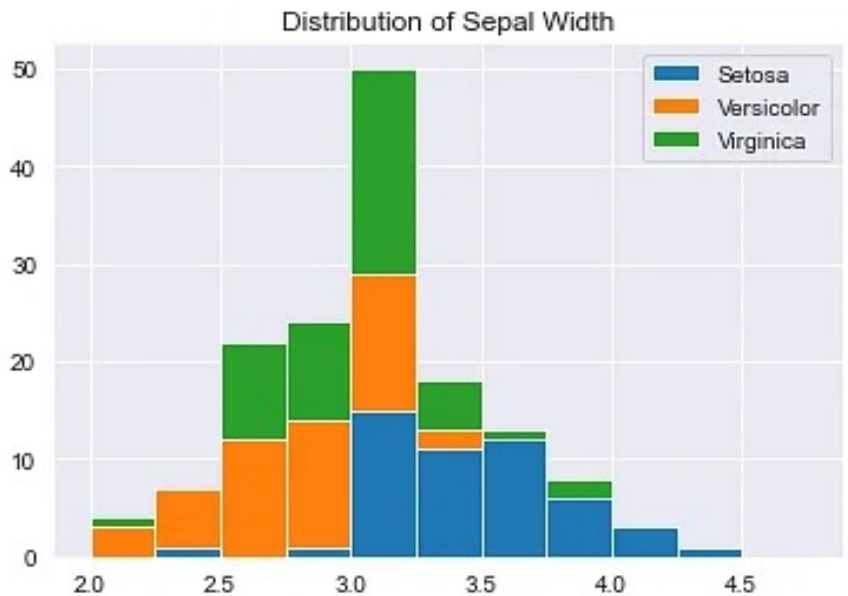

```
setosa_df = flowers_df[flowers_df.species == 'setosa']
versicolor_df = flowers_df[flowers_df.species == 'versicolor']
virginica_df = flowers_df[flowers_df.species == 'virginica']
```

```
plt.hist(setosa_df.sepal_width, alpha=0.4, bins=np.arange(2, 5, 0.25));
plt.hist(versicolor_df.sepal_width, alpha=0.4, bins=np.arange(2, 5, 0.25));
```



Python. Data visualisation Histograms

```
plt.title('Distribution of Sepal Width')

plt.hist([setosa_df.sepal_width, versicolor_df.sepal_width, virginica_df.sepal_width],
         bins=np.arange(2, 5, 0.25),
         stacked=True);

plt.legend(['Setosa', 'Versicolor', 'Virginica']);
```



Python. Data visualisation Heatmaps

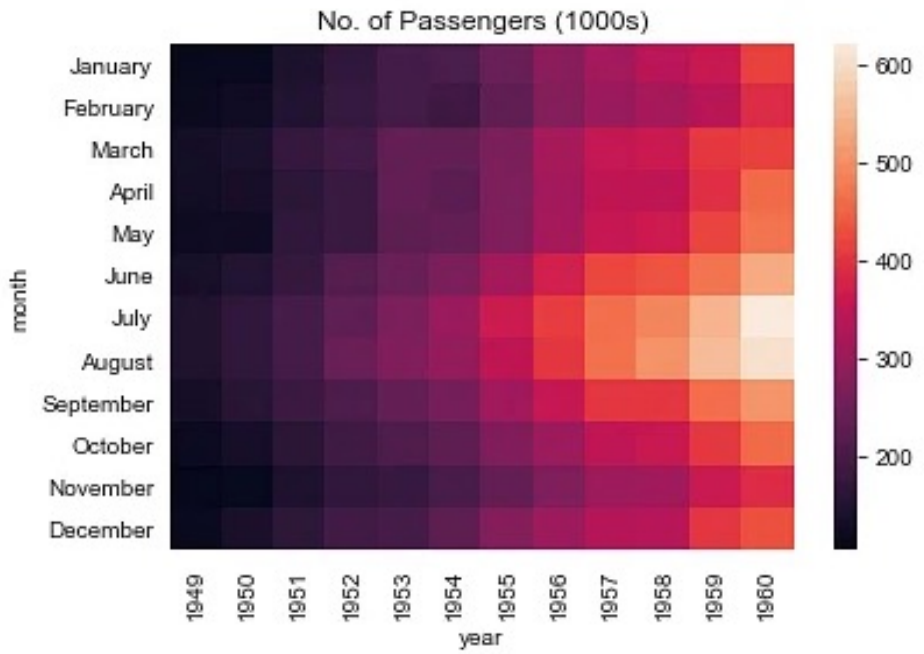
```
flights_df = sns.load_dataset("flights").pivot("month", "year", "passengers")
flights_df
```

year	1949	1950	1951	1952	1953	1954	1955	1956	1957	1958	1959	1960
month												
January	112	115	145	171	196	204	242	284	315	340	360	417
February	118	126	150	180	196	188	233	277	301	318	342	391
March	132	141	178	193	236	235	267	317	356	362	406	419
April	129	135	163	181	235	227	269	313	348	348	396	461
May	121	125	172	183	229	234	270	318	355	363	420	472
June	135	149	178	218	243	264	315	374	422	435	472	535
July	148	170	199	230	264	302	364	413	465	491	548	622
August	148	170	199	242	272	293	347	405	467	505	559	606
September	136	158	184	209	237	259	312	355	404	404	463	508
October	119	133	162	191	211	229	274	306	347	359	407	461
November	104	114	146	172	180	203	237	271	305	310	362	390
December	118	140	166	194	201	229	278	306	336	337	405	432

Python. Data visualisation
Heatmaps

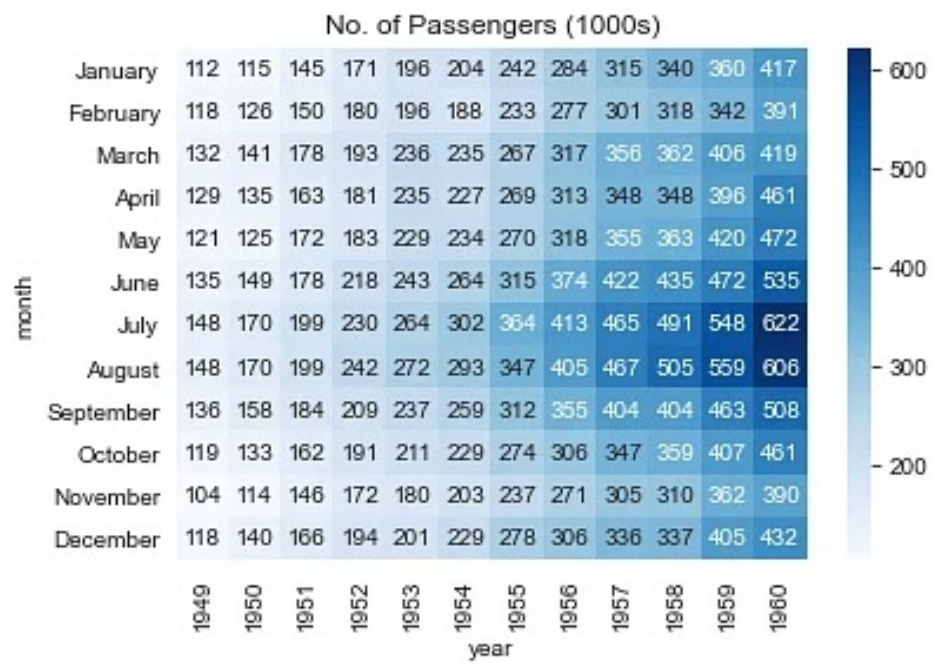
```
plt.title("No. of Passengers (1000s)")
sns.heatmap(flights_df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2054c1feaf0>
```




```
plt.title("No. of Passengers (1000s)")
sns.heatmap(flights_df, fmt="d", annot=True, cmap='Blues')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2054c2dc850>
```



Python. Data visualisation
Iris datasets



1. sepal length in cm
2. sepal width in cm
3. petal length in cm
4. petal width in cm
5. class:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
%matplotlib inline
names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'species']
df = pd.read_csv(r'E:\ML - Data\iris.data.txt', header = None, names = names)
```

Python. Data visualisation
Iris datasets

```
df.head(5)
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
sepal_length    150 non-null float64
sepal_width     150 non-null float64
petal_length    150 non-null float64
petal_width     150 non-null float64
species         150 non-null object
dtypes: float64(4), object(1)
memory usage: 5.9+ KB
```

Here we've total 150 rows and 5 columns.

```
df.describe()
```

	sepal_length	sepal_width	petal_length	petal_width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

Python. Data visualisation
Iris datasets

```
df.keys()
```

```
Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
       'species'],
      dtype='object')
```

```
df.shape
```

```
(150, 5)
```

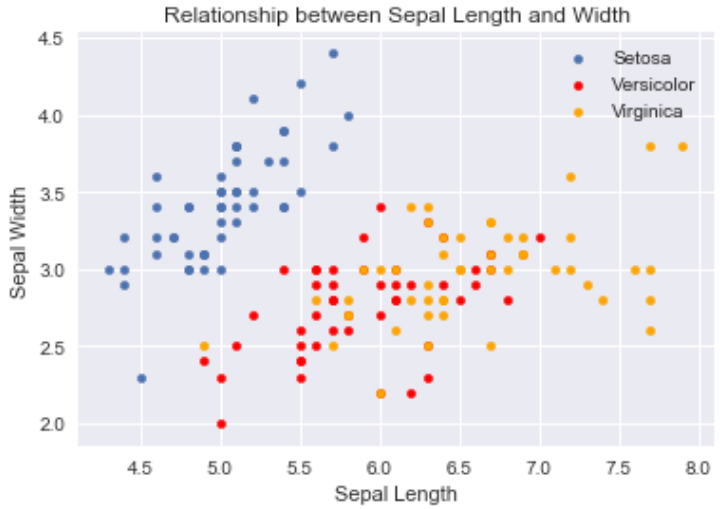
```
df['species'].value_counts()
```

```
Iris-setosa        50
Iris-versicolor    50
Iris-virginica     50
Name: species, dtype: int64
```

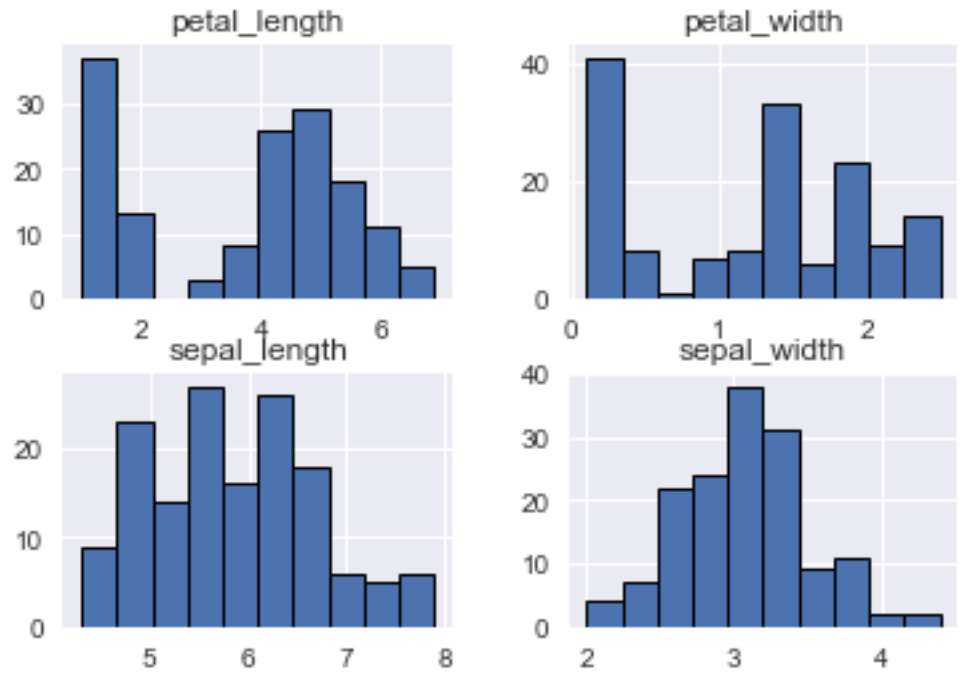
Python. Data visualisation
Iris datasets. Matplotlib

```
plt.figure(figsize=(14,8))
ax = df[df.species=='Iris-setosa'].plot.scatter(x='sepal_length', y='sepal_width', label='Setosa')
df[df.species=='Iris-versicolor'].plot.scatter(x='sepal_length', y='sepal_width', color='red', label='Versi
df[df.species=='Iris-virginica'].plot.scatter(x='sepal_length', y='sepal_width', color='orange', label='Vir
ax.set_xlabel("Sepal Length")
ax.set_ylabel("Sepal Width")
ax.set_title("Relationship between Sepal Length and Width")
plt.show()
```

```
<matplotlib.figure.Figure at 0xa8be6d8>
```



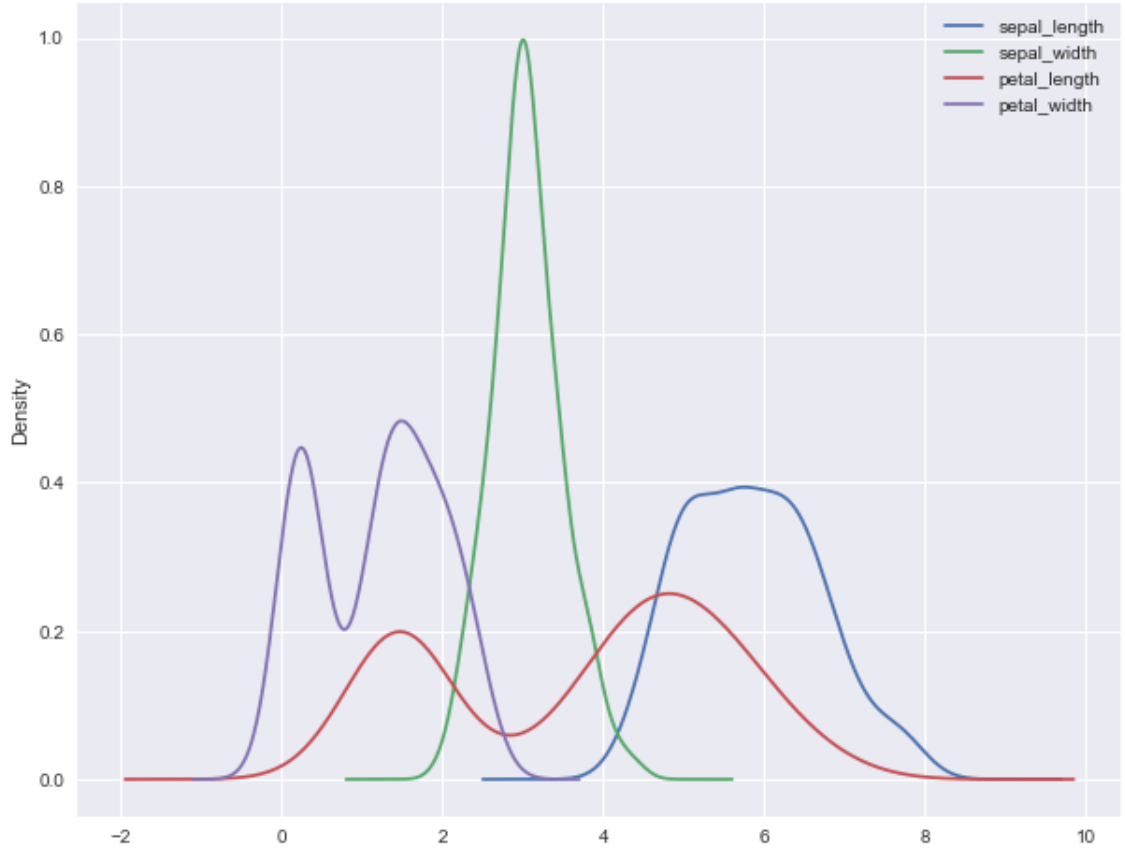

```
df.hist(edgecolor='black', linewidth=1)
plt.show()
```



Python. Data visualisation Iris datasets.Matplotlib

```
df.plot(kind = "density", figsize=(10,8))
```

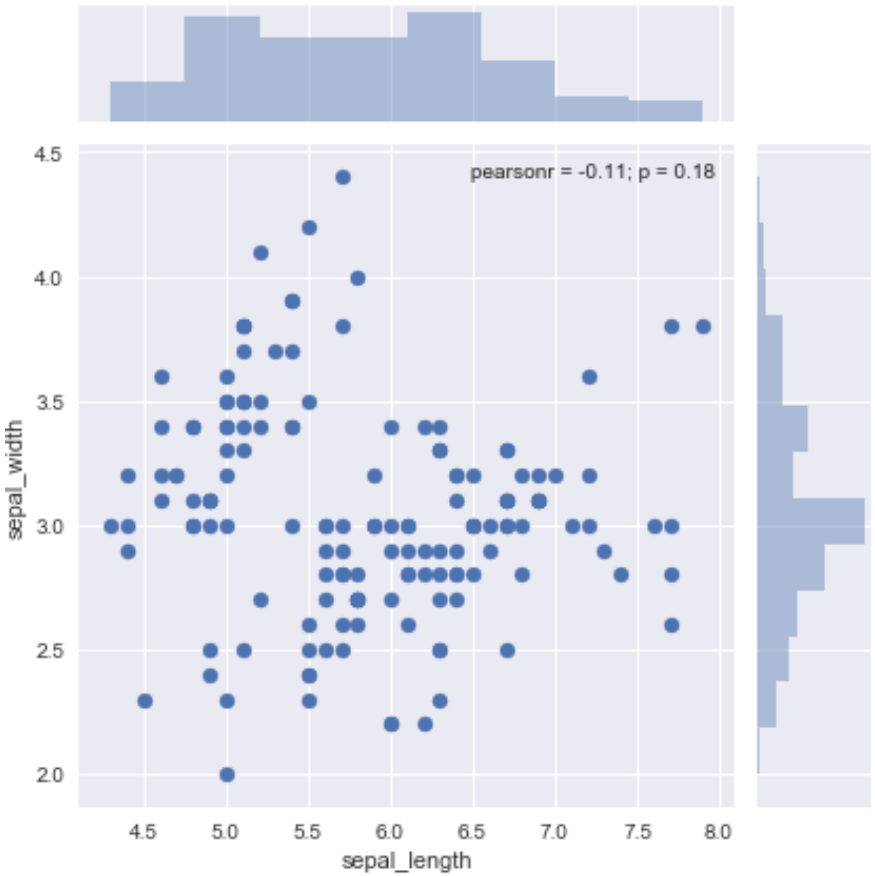
```
<matplotlib.axes._subplots.AxesSubplot at 0xbcb2e80>
```



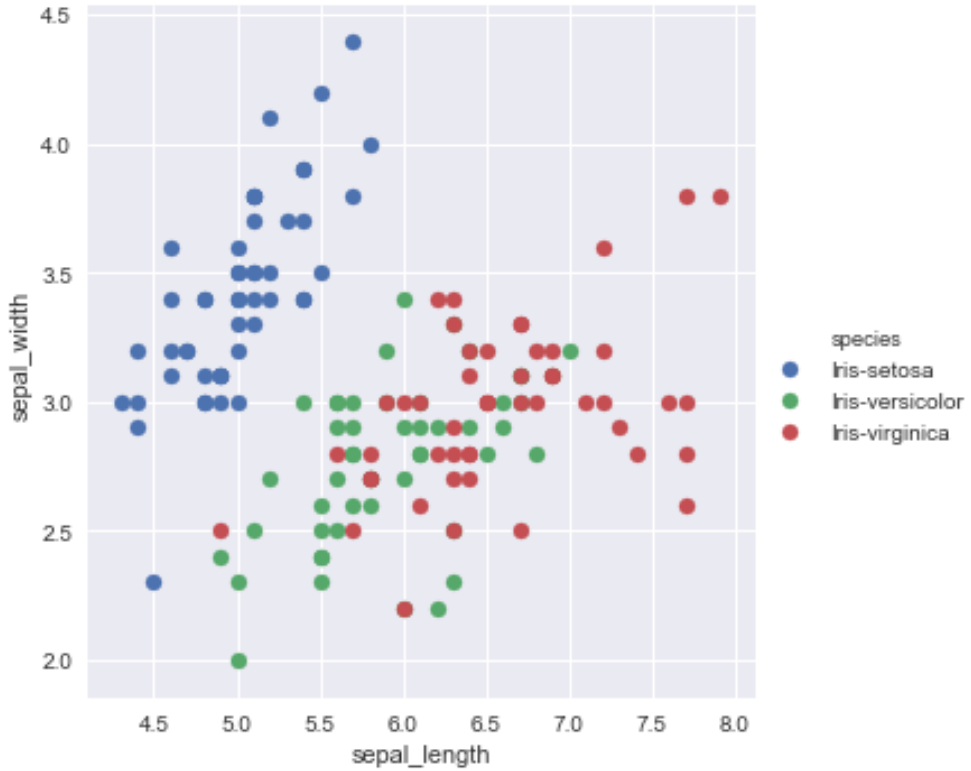
Python. Data visualisation
Iris dataset. Seaborn

```
sns.jointplot(x = 'sepal_length', y = 'sepal_width', data = df)
```

```
<seaborn.axisgrid.JointGrid at 0x4d74940>
```

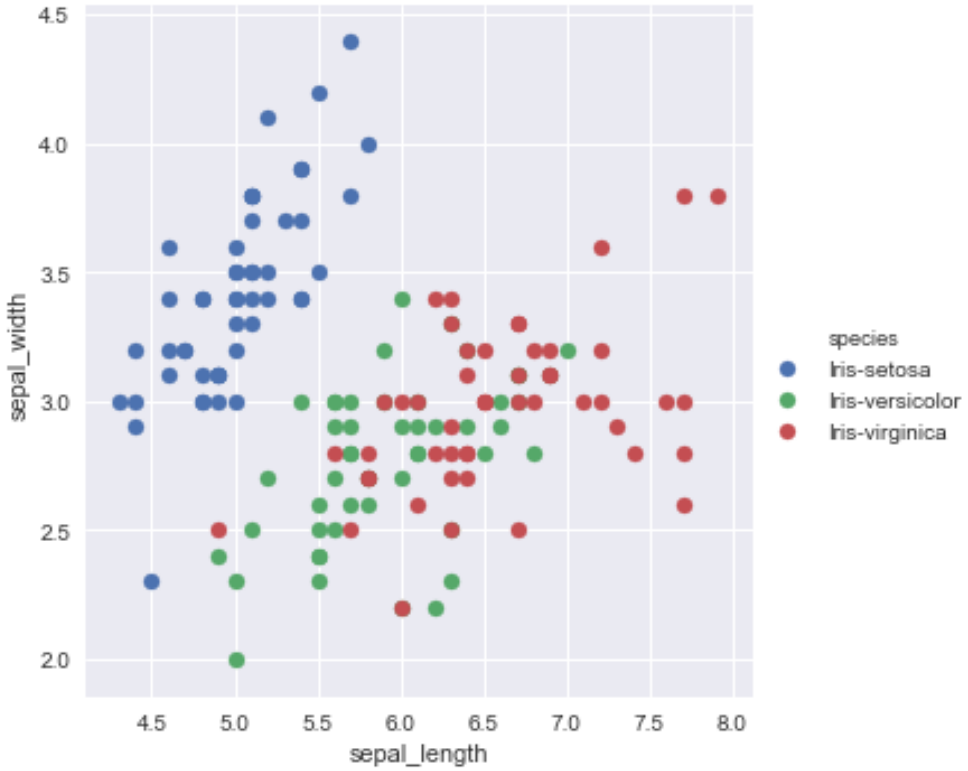



```
#seaborn scatterplot by species on sepal_length vs sepal_width
g = sns.FacetGrid(df, hue='species', size=5)
g = g.map(plt.scatter, 'sepal_length', 'sepal_width').add_legend()
```

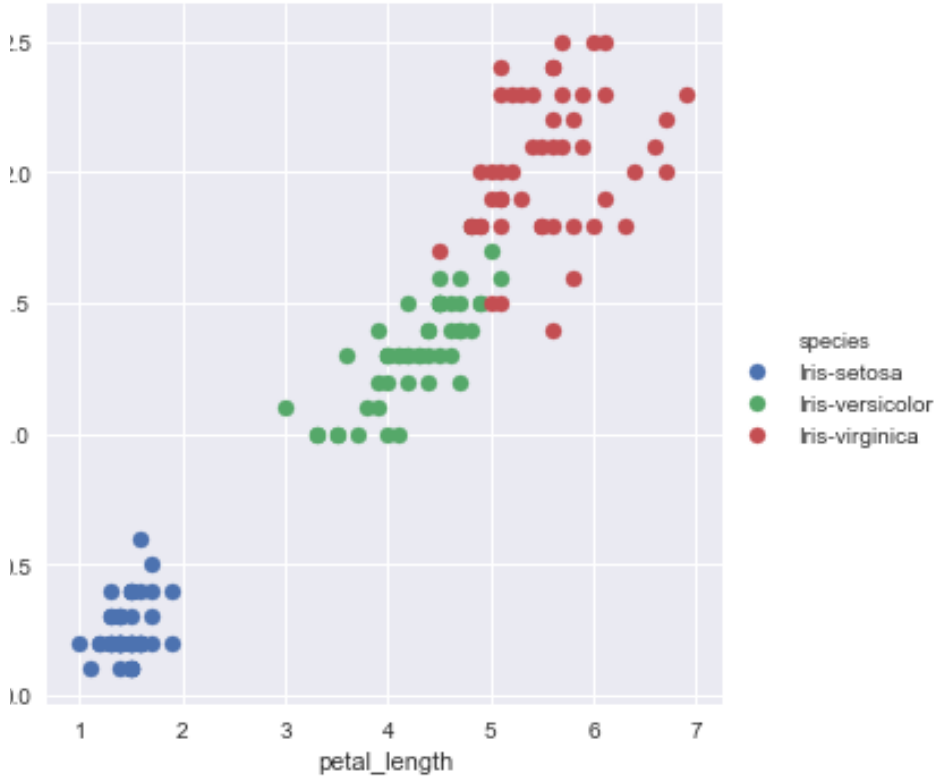


Python. Data visualisation
Iris dataset

```
#seaborn scatterplot by species on sepal_length vs sepal_width
g = sns.FacetGrid(df, hue='species', size=5)
g = g.map(plt.scatter, 'sepal_length', 'sepal_width').add_legend()
```



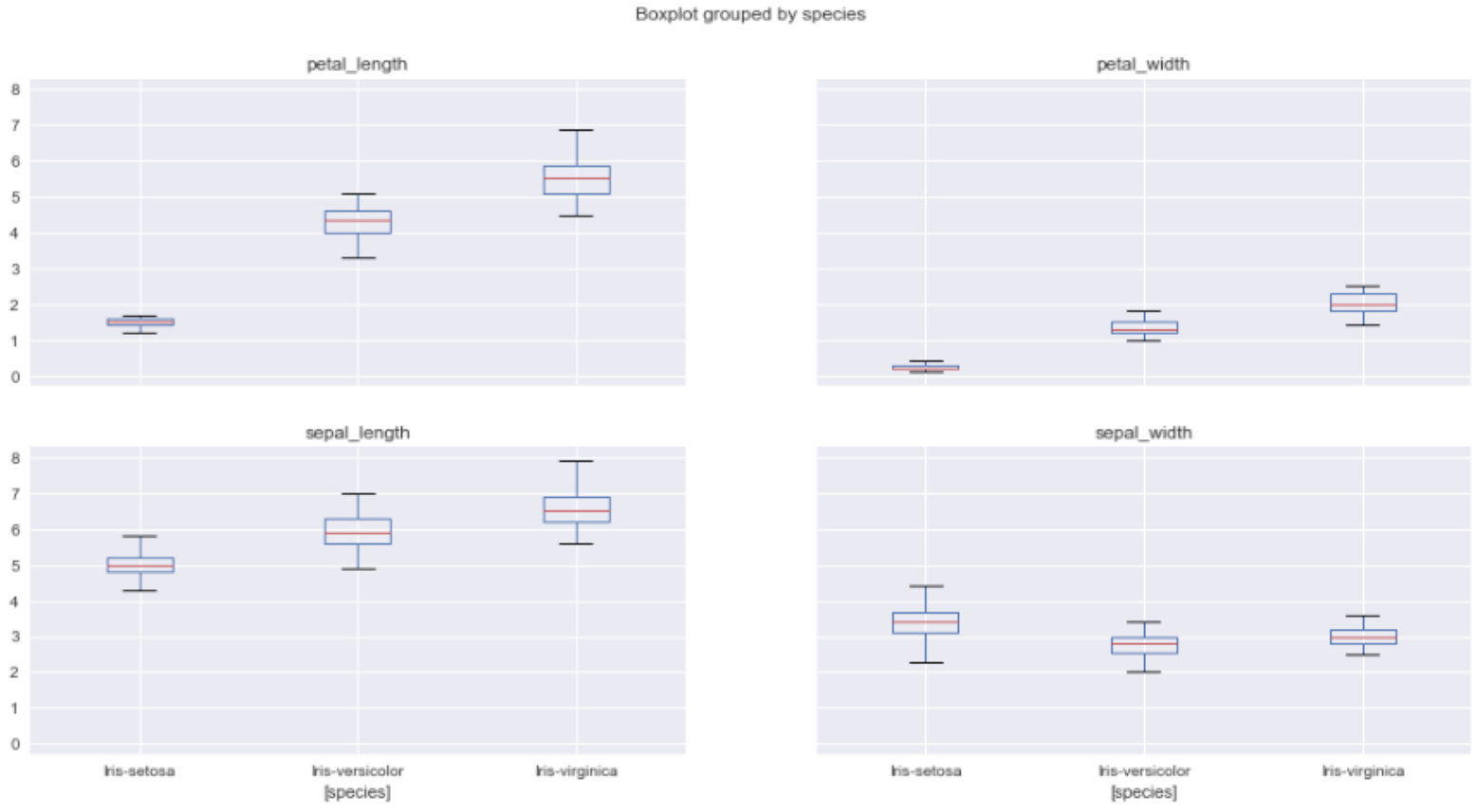

```
g = sns.FacetGrid(df, hue='species', size=5)
g = g.map(plt.scatter, 'petal_length', 'petal_width').add_legend()
```



Python. Data visualisation
Iris dataset

```
# Seaborn boxplot for on each features split out by specis. Try this plot again just changing figsize.
df.boxplot(by = 'species', figsize = (16,8))
```

```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x000000000BCC71D0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000000000A8BE588>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x000000000BC47390>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000000000BB68390>]], dtype=object)
```

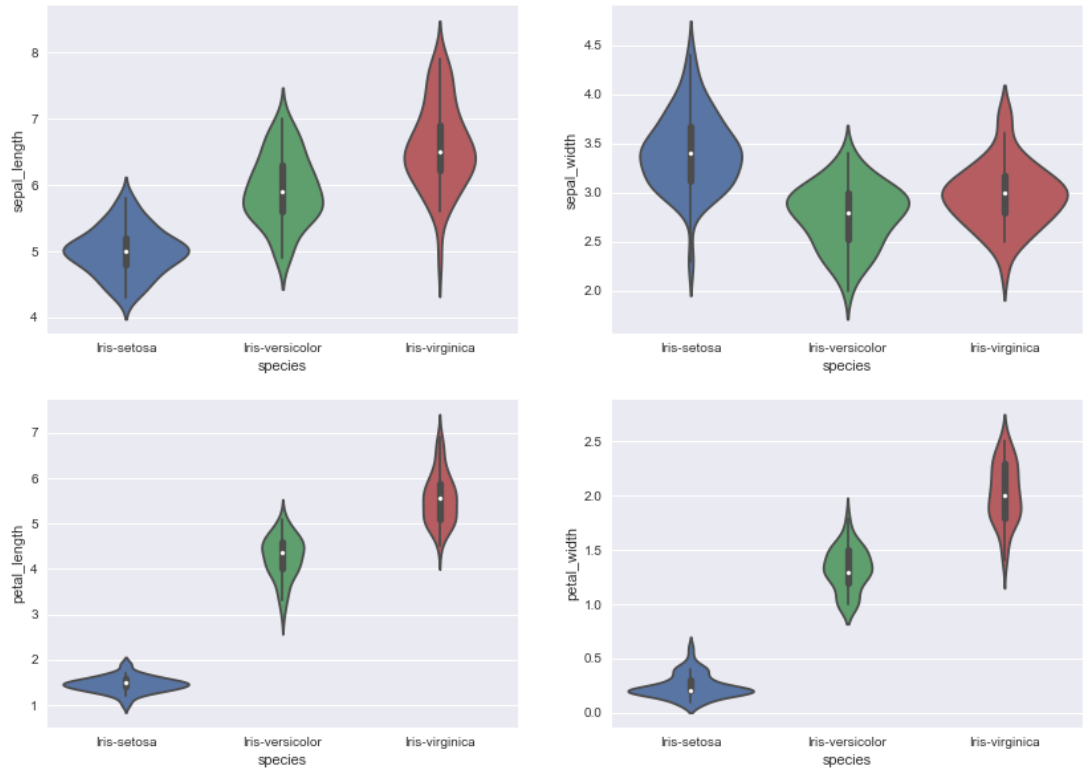


Identify some positive/ negative characteristics for this representation

Python. Data visualisation
Iris dataset

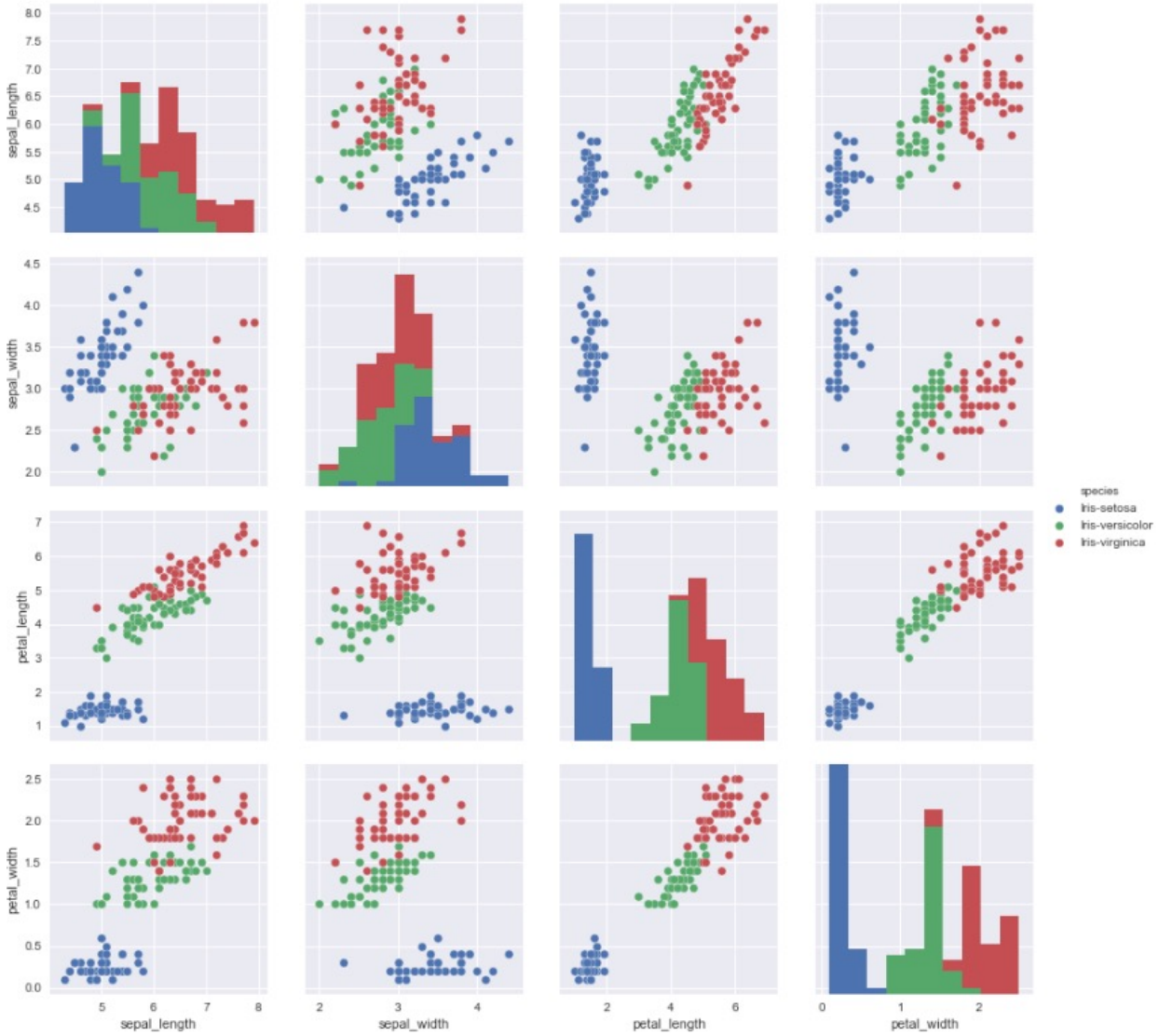
```
# The violinplot shows density of the length and width in the species
# Denser regions of the data are fatter, and sparser thiner in a violin plot
plt.figure(figsize=(14,10))
plt.subplot(2,2,1)
sns.violinplot(x='species', y='sepal_length', data=df, size=5)
plt.subplot(2,2,2)
sns.violinplot(x='species', y='sepal_width', data=df, size=5)
plt.subplot(2,2,3)
sns.violinplot(x='species', y='petal_length', data=df, size=5)
plt.subplot(2,2,4)
sns.violinplot(x='species', y='petal_width', data=df, size=5)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0xc4a4da0>
```



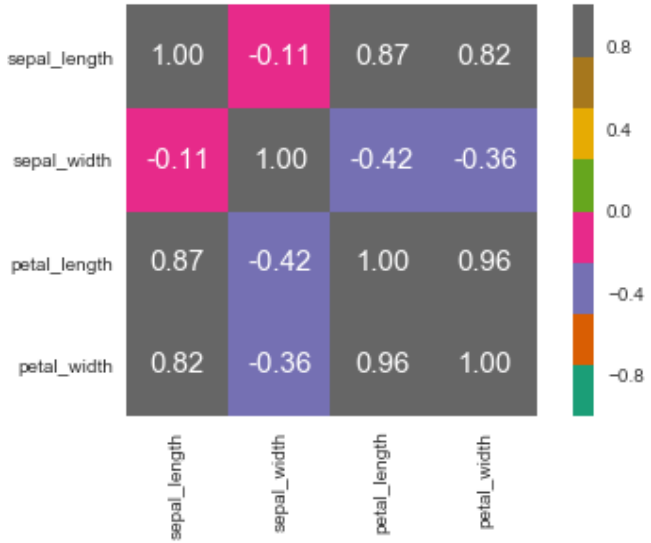

```
sns.pairplot(data = df, hue = 'species', size = 3)
```

```
<seaborn.axisgrid.PairGrid at 0xc74a390>
```



Python. Data visualisation
Iris dataset

```
plt.figure(figsize=(7,4))
#draws  heatmap with input as the correlation matrix calculted by(df.corr())
sns.heatmap(df.corr(),cbar = True, square = True, annot=True, fmt='.2f',annot_kws={'size': 15},cmap='Dark2'
plt.show()
```



The dataset is ready for some ML pre-processing!

Python.
SciPy. High level scientific computing

The scipy package contains various toolboxes dedicated to common issues in scientific computing. Its different submodules correspond to different applications, such as interpolation, integration, optimization, image processing, statistics, special functions, etc.

File input/output: scipy.io
Special functions: scipy.special
Linear algebra operations: scipy.linalg
Interpolation: scipy.interpolate
Optimization and fit: scipy.optimize
Statistics and random numbers: scipy.stats
Numerical integration: scipy.integrate
Image manipulation: scipy.ndimage

`scipy.cluster`	Vector quantization / Kmeans
`scipy.constants`	Physical and mathematical constants
`scipy.fftpack`	Fourier transform
`scipy.integrate`	Integration routines
`scipy.interpolate`	Interpolation
`scipy.io`	Data input and output
`scipy.linalg`	Linear algebra routines
`scipy.ndimage`	n-dimensional image package
`scipy.odr`	Orthogonal distance regression
`scipy.optimize`	Optimization
`scipy.signal`	Signal processing
`scipy.sparse`	Sparse matrices
`scipy.spatial`	Spatial data structures and algorithms
`scipy.special`	Any special mathematical functions
`scipy.stats`	Statistics

Python. SciPy. High level scientific computing scipy.io

scipy.io contains functions for loading and saving data in several common formats including Matlab, IDL, Matrix Market, and Harwell-Boeing.

Matlab files: Loading and saving:

```
>>> import scipy as sp
>>> a = np.ones((3, 3))
>>> sp.io.savemat('file.mat', {'a': a})  # savemat expects a dictionary
>>> data = sp.io.loadmat('file.mat')
>>> data['a']
array([[1.,  1.,  1.],
       [1.,  1.,  1.],
       [1.,  1.,  1.]])
```

numpy.loadtxt

```
numpy.loadtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None,
converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0,
encoding='bytes', max_rows=None, *, quotechar=None, like=None)
```
[source]

Load data from a text file.

Parameters: fname : *file, str, pathlib.Path, list of str, generator*

File, filename, list, or generator to read. If the filename extension is `.gz` or `.bz2`, the file is first decompressed. Note that generators must return bytes or strings. The strings in a list or produced by a generator are treated as lines.

numpy.savetxt

```
numpy.savetxt(fname, X, fmt='%.18e', delimiter=' ', newline='\n', header='',
footer='', comments='# ', encoding=None)
```
[source]

Save an array to a text file.

Parameters: fname : *filename or file handle*

If the filename ends in `.gz`, the file is automatically saved in compressed gzip format. `loadtxt` understands gzipped files transparently.

X : *1D or 2D array_like*

Data to be saved to a text file.

Python.
SciPy. High level scientific computing

numpy.save

`numpy.save(file, arr, allow_pickle=True, fix_imports=True)` [source]

Save an array to a binary file in NumPy `.npy` format.

Parameters: **file** : *file, str, or pathlib.Path*

File or filename to which the data is saved. If file is a file-object, then the filename is unchanged. If file is a string or Path, a `.npy` extension will be appended to the filename if it does not already have one.

arr : *array_like*

Array data to be saved.

allow_pickle : *bool, optional*

Allow saving object arrays using Python pickles. Reasons for disallowing pickles include security (loading pickled data can execute arbitrary code) and portability (pickled objects may not be loadable on different Python installations, for example if the stored objects require libraries that are not available, and not all pickled data is compatible between Python 2 and Python 3). Default: True

numpy.load

`numpy.load(file, mmap_mode=None, allow_pickle=False, fix_imports=True, encoding='ASCII', *, max_header_size=10000)` [source]

Load arrays or pickled objects from `.npy`, `.npz` or pickled files.

> **Warning**
>
> Loading files that contain object arrays uses the `pickle` module, which is not secure against erroneous or maliciously constructed data. Consider passing `allow_pickle=False` to load data that is known not to contain object arrays for the safer handling of untrusted sources.

Parameters: **file** : *file-like object, string, or pathlib.Path*

The file to read. File-like objects must support the `seek()` and `read()` methods and must always be opened in binary mode. Pickled files require that the file-like object support the `readline()` method as well.

mmap_mode : *{None, 'r+', 'r', 'w+', 'c'}, optional*

If not None, then memory-map the file, using the given mode (see `numpy.memmap` for a detailed description of the modes). A memory-mapped array is kept on disk. However, it can be accessed and sliced like any ndarray. Memory mapping is especially useful for accessing small fragments of large files without reading the entire file into memory.

Python.SciPy. High level scientific computing scipy.special

“Special” functions are functions commonly used in science and mathematics that are not considered to be “elementary” functions. Examples include

- the gamma function, **`scipy.special.gamma()`**,
- the error function, **`scipy.special.erf()`**,
- Bessel functions, such as **`scipy.special.jv()`** (Bessel functions of the first kind), and
- elliptic functions, such as **`scipy.special.ellipj()`** (Jacobian elliptic functions).

Other special functions are combinations of familiar elementary functions, but they offer better accuracy or robustness than their naive implementations would.

Python. SciPy. High level scientific computing scipy.linalg

The `scipy.linalg` module provides standard linear algebra operations, relying on an underlying efficient implementation (BLAS, LAPACK).

- The `scipy.linalg.det()` function computes the determinant of a square matrix:

```
>>> from scipy import linalg
>>> arr = np.array([[1, 2],
...                 [3, 4]])
>>> linalg.det(arr)
-2.0
>>> arr = np.array([[3, 2],
...                 [6, 4]])
>>> linalg.det(arr)
0.0
>>> linalg.det(np.ones((3, 4)))
Traceback (most recent call last):
...
ValueError: expected square matrix
Traceback (most recent call last):
...
ValueError: expected square matrix
```

Python. SciPy. High level scientific computing scipy.linalg

- The `scipy.linalg.inv()` function computes the inverse of a square matrix:

```
>>> arr = np.array([[1, 2],
...                 [3, 4]])
>>> iarr = linalg.inv(arr)
>>> iarr
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
>>> np.allclose(np.dot(arr, iarr), np.eye(2))
True
```

Finally computing the inverse of a singular matrix (its determinant is zero) will raise `LinAlgError`:

```
>>> arr = np.array([[3, 2],
...                 [6, 4]])
>>> linalg.inv(arr)
Traceback (most recent call last):
...
...LinAlgError: singular matrix
Traceback (most recent call last):
...
...LinAlgError: singular matrix
```

Python. SciPy. High level scientific computing scipy.interpolate

scipy.interpolate is useful for fitting a function from experimental data and thus evaluating points where no reference value exists. The module includes, but not limited to FITPACK Fortran subroutines.

By imagining experimental data close to a sine function:

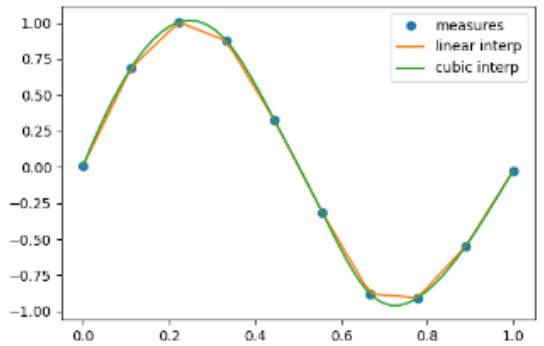
```
>>> measured_time = np.linspace(0, 1, 10)
>>> rng = np.random.default_rng()
>>> noise = (rng.random(10)*2 - 1) * 1e-1
>>> measures = np.sin(2 * np.pi * measured_time) + noise
```

scipy.interpolate has many interpolation methods which need to be chosen based on the data. See the tutorial for some guidelines:

```
>>> spline = sp.interpolate.CubicSpline(measured_time, measures)
```

Then the result can be evaluated at the time of interest:

```
>>> interpolation_time = np.lin-
      space(0, 1, 50)
>>> linear_results = spline(inter-
      polation_time)
```



scipy.interpolate.CloughTocher2DInterpolator is similar to **scipy.interpolate.CubicSpline**, but for 2-D arrays. See the summary exercise on Maximum wind speed prediction at the Sprogø station for a more advanced spline interpolation example.

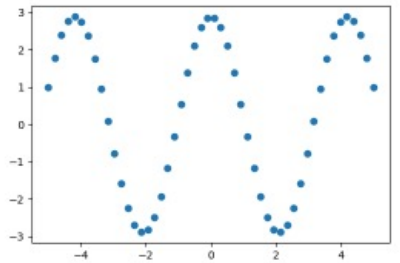
Python. SciPy. High level scientific computing scipy.optimize

scipy.optimize.root_scalar() attempts to find a root of a specified scalar-valued function (i.e., an argument at which the function value is zero). Like many **scipy.optimize** functions, the function needs an initial guess of the solution, which the algorithm will refine until it converges or recognizes failure. We also provide the derivative to improve the rate of convergence.

```
>>> def f(x):
...     return (x-1)*(x-2)
>>> def df(x):
...     return 2*x - 3
>>> x0 = 0  # guess
>>> res = sp.optimize.root_scalar(f, x0=x0, fprime=df)
>>> res
      converged: True
           flag: converged
 function_calls: 12
     iterations: 6
           root: 1.0
```

Suppose we have data that is sinusoidal but noisy:

```
>>> x = np.linspace(-5, 5, num=50)  # 50
      values between -5 and 5
>>> noise = 0.01 * np.cos(100 * x)
>>> a, b = 2.9, 1.5
>>> y = a * np.cos(b * x) + noise
```

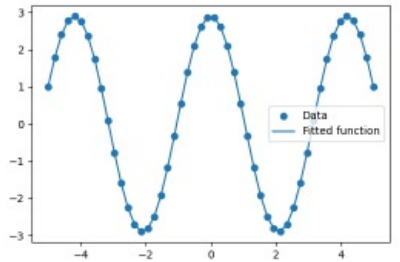


We can approximate the underlying amplitude, frequency, and phase from the data by least squares curve fitting. To begin, we write a function that accepts the independent variable as the first argument and all parameters to fit as separate arguments:

```
>>> def f(x, a, b, c):
...     return a * np.sin(b * x + c)
```

We then use **scipy.optimize.curve_fit()** to find a and b:

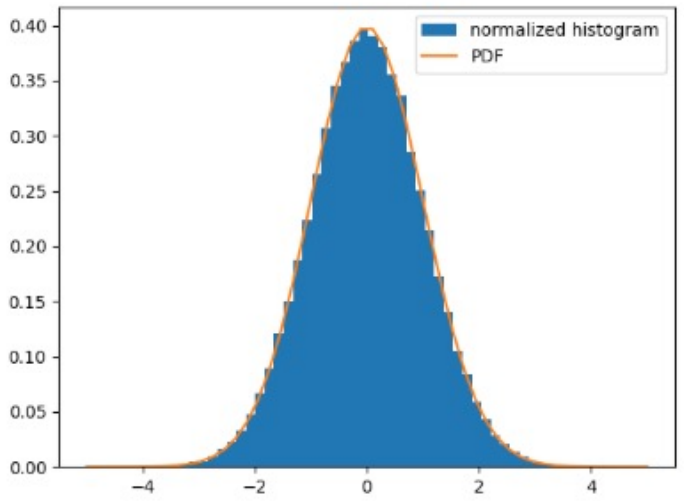
```
>>> params, _ = sp.optimize.curve_fit(f,
      x, y, p0=[2, 1, 3])
>>> params
array([2.900026  , 1.50012043,
       1.57079633])
>>> ref = [a, b, np.pi/2]  # what we'd
      expect
>>> np.allclose(params, ref, rtol=1e-3)
True
```



Python. SciPy. High level scientific computing scipy. stats

Consider a random variable distributed according to the standard normal. We draw a sample consisting of 100000 observations from the random variable. The normalized histogram of the sample is an estimator of the random variable's probability density function (PDF):

```
>>> dist = sp.stats.norm(loc=0, scale=1)  # standard normal distribution
>>> sample = dist.rvs(size=100000)  # "random variate sample"
>>> plt.hist(sample, bins=50, density=True, label='normalized histogram')
>>> x = np.linspace(-5, 5)
>>> plt.plot(x, dist.pdf(x), label='PDF')
[<matplotlib.lines.Line2D object at ...>]
>>> plt.legend()
<matplotlib.legend.Legend object at ...>
```



The sample mean is an estimator of the mean of the distribution from which the sample was drawn:

```
>>> np.mean(sample)
0.001576700508...
```

NumPy includes some of the most fundamental sample statistics (e.g. **`numpy.mean()`**, **`numpy.var()`**, **`numpy.percentile()`**); **`scipy.stats`** includes many more. For instance, the geometric mean is a common measure of central tendency for data that tends to be distributed over many orders of magnitude.

```
>>> sp.stats.gmean(2**sample)
1.0010934829...
```

SciPy also includes a variety of hypothesis tests that produce a sample statistic and a p-value. For instance, suppose we wish to test the null hypothesis that `sample` was drawn from a normal distribution:

```
>>> res = sp.stats.normaltest(sample)
>>> res.statistic
5.20841759...
>>> res.pvalue
0.07396163283...
```

Python.SciPy. High level scientific computing scipy.ndimage

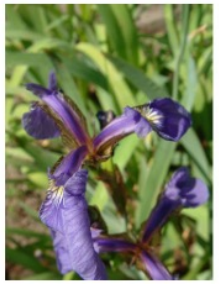
Changing orientation, resolution, ..

```
>>> from scipy import misc
>>> lena = misc.lena()
>>> shifted_lena = ndimage.shift(lena, (50, 50))
>>> shifted_lena2 = ndimage.shift(lena, (50, 50), mode='nearest')
>>> rotated_lena = ndimage.rotate(lena, 30)
>>> cropped_lena = lena[50:-50, 50:-50]
>>> zoomed_lena = ndimage.zoom(lena, 2)
>>> zoomed_lena.shape
(1024, 1024)
```

```
>>> from scipy import misc
>>> lena = misc.lena()
>>> import numpy as np
>>> noisy_lena = np.copy(lena).astype(np.float)
>>> noisy_lena += lena.std() * 0.5 * np.random.standard_normal(lena.shape)
>>> blurred_lena = ndimage.gaussian_filter(noisy_lena, sigma=3)
>>> median_lena = ndimage.median_filter(blurred_lena, size=5)
>>> from scipy import signal
>>> wiener_lena = signal.wiener(blurred_lena, (5, 5))
```

Many other filters in `scipy.ndimage.filters` and `scipy.signal` can be applied to images.

Python.
Scikit-learn. Machine Learning



We have 150 observations of the iris flower specifying some measurements: sepal length, sepal width, petal length and petal width together with its subtype: *Iris setosa*, *Iris versicolor*, *Iris virginica*.

To load the dataset into a Python object:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
```

This data is stored in the `.data` member, which is a `(n_samples, n_features)` array.

```
>>> iris.data.shape
(150, 4)
```

The class of each observation is stored in the `.target` attribute of the dataset. This is an integer 1D array of length `n_samples`:

```
>>> iris.target.shape
(150,)
>>> import numpy as np
>>> np.unique(iris.target)
array([0, 1, 2])
```

Python.
Scikit-learn. Machine Learning

Now that we've got some data, we would like to learn from it and predict on new one. In `scikit-learn`, we learn from existing data by creating an `estimator` and calling its `fit(X, Y)` method.

```
>>> from sklearn import svm
>>> clf = svm.LinearSVC()
>>> clf.fit(iris.data, iris.target) # learn from the data
LinearSVC(...)
```

Once we have learned from the data, we can use our model to predict the most likely outcome on unseen data:

```
>>> clf.predict([[ 5.0,  3.6,  1.3,  0.25]])
array([0])
```

We can access the parameters of the model via its attributes ending with an underscore:

```
>>> clf.coef_
array([[ 0...]])
```

Python.
Scikit-learn. Machine Learning

Grid-search

The scikit-learn provides an object that, given data, computes the score during the fit of an estimator on a parameter grid and chooses the parameters to maximize the cross-validation score. This object takes an estimator during the construction and exposes an estimator API:

```
>>> from sklearn import svm, grid_search
>>> gammas = np.logspace(-6, -1, 10)
>>> svc = svm.SVC()
>>> clf = grid_search.GridSearchCV(estimator=svc, param_grid=dict(gamma=gammas),
...                                n_jobs=-1)
>>> clf.fit(digits.data[:1000], digits.target[:1000])
GridSearchCV(cv=None,...)
>>> clf.best_score_
0.9...
>>> clf.best_estimator_.gamma
0.00059948425031894088
```

By default the `GridSearchCV` uses a 3-fold cross-validation. However, if it detects that a classifier is passed, rather than a regressor, it uses a stratified 3-fold.

More details on cross-validation will be provided when we discuss each approach in detail.

Next Lecture …

Applied Data Science
L6-7. Supervised learning. Regression