Introduction to Computational Physics (PHYS105)

Lecture 2: Data Structures

Carl Gwilliam

(C.Gwilliam@liverpool.ac.uk)

UNIVERSITY OF LIVERPOOL

Attendance Code: 652195

Lecture 1: Recap

- Last week we learnt the basic building blocks of python programs
  - As well as learning how to work with jupyter notebooks in CoCalc

- **Variables**: symbolic name referring to an object or value stored in memory, which are assigned to it using the = operator
  - The type of the variable defines what can be done with it
  - E.g. integer, float, string

- **Functions**: blocks of code that do something when called using `()` with the input variables as arguments between the brackets
  - If called from modules (libraries of code) you need to proceed the function name with the module name followed by a dot

- **Comments**: documentary pieces of code, starting with a `#`, to explain what is being done
  - Should be concise and will be taken into account in marking

- We combined these into simple programs.

Calculate the sine of an angle

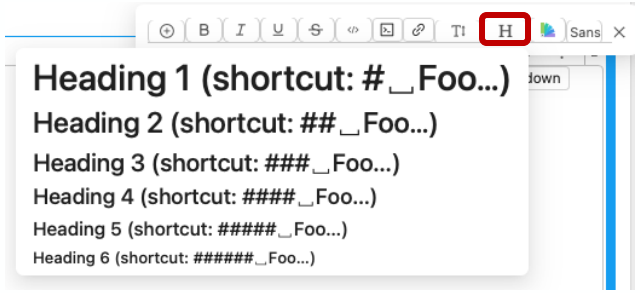
```
# Import the necessary math module
import math

# Assign the variable
angle = 90*math.pi/180

# Call the math function on it
sine = math.sin(angle)

# Print the result
print(sine)
```

```
1.0
```

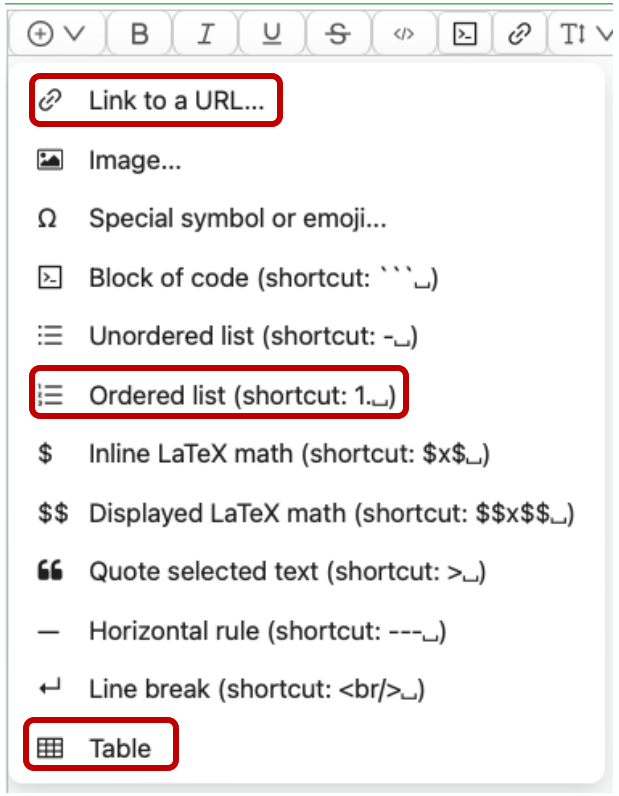
Lecture 1: Formative problem solutions

- Exercise 1: Making and deleting a cell
  - Click on "+" in the menu (or type `Esc+b`),
  - ***Edit menu → Delete cells (***or type `Esc+d+d`)

- Exercise 2: Write a simple text document
  - Make sure cell is in text mode!
  - Enter the heading using the H option from the cell's menu



  - Then from the "+" menu enter
    - A link and an ordered list
    - A table: need to add/delete rows and columns via Markdown



This is the heading

Here are a few words of text.

I need to buy:

1. Cheese
2. Butter
3. Bread
4. Apples
5. Oranges

Here is a table of the days of the week:

Number	Day
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday
7	Sunday

The liverpool university website is at http://www.liverpool.ac.uk

Lecture 1: Formative problem solutions

- Exercise 3: “Hello, World!” Reply
  - Change the variable to your name e.g. John
  - Change the text to reply to PHYS105 or me

```
[2]: # This program prints "Hello Carl, my name is John!"
     name = "John"
     print ("Hello Carl, my name is", name, "!")

     Hello Carl, my name is John !
```

  - Technically it is not a reply if you just change the name to yours (which many people did) ☺

```
[3]: name = "John"
     print ("Hello,", name, "!")

     Hello, John !
```

- Exercise 4: Mathematical operators
  - +: addition
  - - : subtraction
  - *: multiplication
  - /: division
  - **: raise to the power
  - //: integer or floor division
    - i.e. throw away decimal
  - %: remainder after division
    - i.e. after integer division

```
[3]: 1 + 4
[3]: 5

[4]:  3 - 8
[4]: -5

[5]: 4*3
[5]: 12

[6]: 3/4
[6]: 0.75

[7]: 2**10
[7]: 1024

[8]: 17//5
[8]: 3

[9]: 17%5
[9]: 2
```

Lecture 1: Formative problem solutions

- Exercise 5: sine and cosine
  - Import the relevant items from the `math` module
  - Call `cos` or `sin` function
    - With the input angle passed in brackets
    - Not forgetting to convert to radians!
  - Assign the results to a variable
    - And output them using the `print` function
  - Print the sum of the squares similarly
    - Here, since I am only using the value once, I chose not to assign the result to a variable first.

```
from math import cos, sin, pi
c60 = cos(60 * pi/180)
print(c60)

s60 = sin(60 * pi/180)
print(s60)

print(c60**2 + s60**2)
```

```
0.5000000000000001
0.8660254037844386
1.0
```

- Marks for the summative exercise and feedback will be released in Canvas by next Monday

Data

- Often, we want to perform calculations not just on a single value but on sets of data
  - Such as measurements taken in a lab or the results of many calculations

Volume (v)	V Error	Pressure (P)	P Error
57	1	1023.6	0.1
55	1	1058	2
53	1	1090.1	0.8
51	1	1127.6	0.4
49	1	1166.7	0.9
47	1	1206	1
45	1	1250	2
43	1	1296	1
41	1	1346	3
39	1	1408.6	0.9

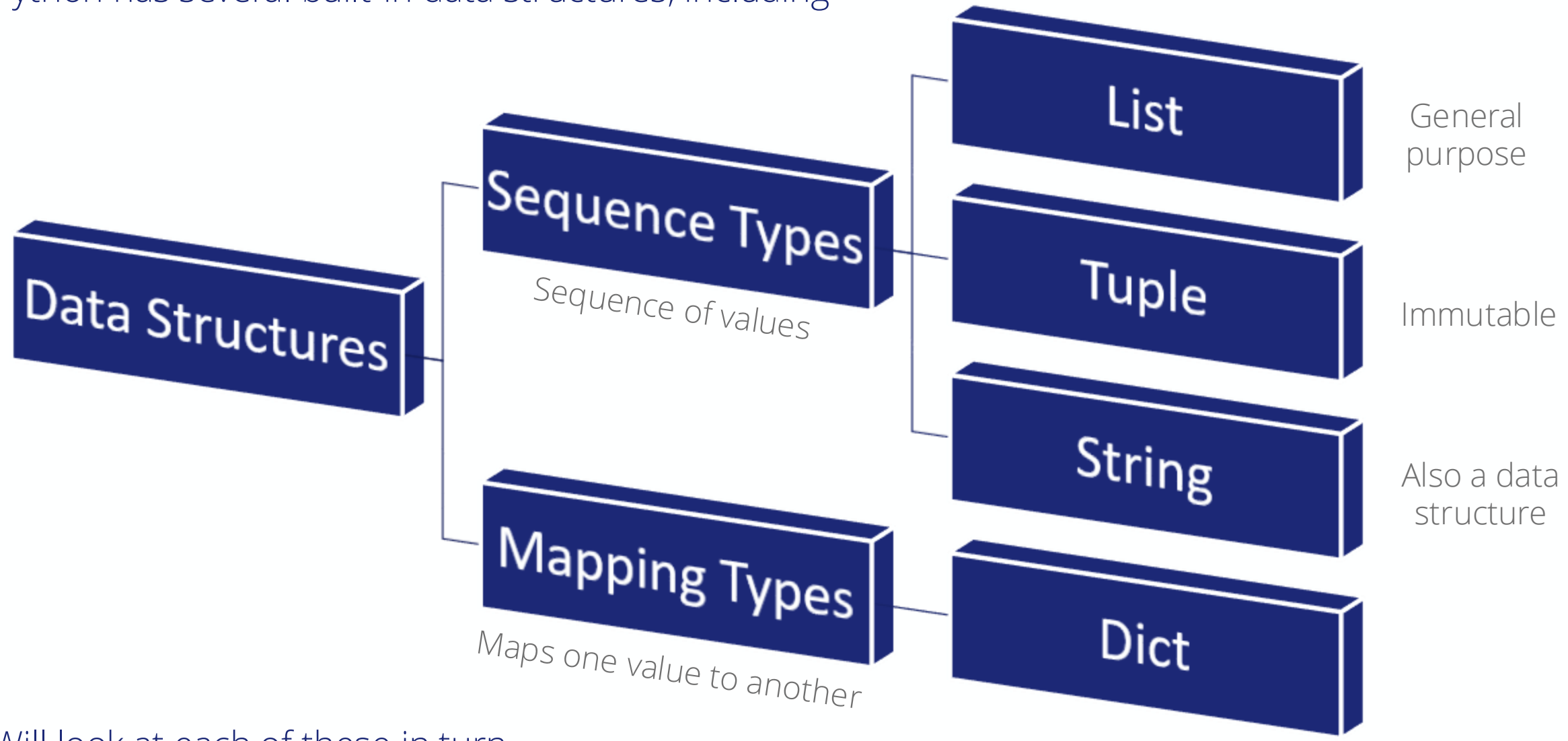
Table: Data from a measurement of Boyle's law (PV = constant)

- This week, we will look at how to store and manipulate series of data in python
  - Including reading in / writing out data to / from files

- You will, of course, practice this in this week's notebook

Python data structures

- Python has several built-in data structures, including



- Will look at each of these in turn ...

Lists

- Lists are the most versatile of python's data structures, simply consisting of a sequence of values
  - They are assigned by a comma-separated list of values in square brackets

```
[3]: mylist = [1, 2.278, -8, "cheese", 'blue ', math.pi]
```

  - Note: the items in a list can have a mixture of types (this is true of most built-in data structures)

- We access a specific element via the index
  - Passed in square brackets `[]`

```
[5]: print("Second element (1) =", mylist[1])
     Second element (1) = 2.278
```

  - Note: indices start from 0 in python (not 1)!

- We can change the elements of a list
  - Assign a new value to an index with =

```
[6]: mylist[1] = "car"
     print("Second element [1] =", mylist[1])
     Second element (1) = car
```

- Once you have the element of a list you can use it exactly as you would the element itself
  - Operator results depend on type of element

```
[9]: print("Elements [0] + [2] =", mylist[0] + mylist[2])
     print("Elements [4] + [3] =", mylist[4] + mylist[3])
     Elements (0) + (2) = -7
     Elements (4) + (3) = blue cheese
```

Tuples and strings

- Tuples are assigned with round brackets and are similar to lists, except they are immutable
  - I.e. unlike a list, the elements of a tuple cannot be changed after the tuple is constructed

Create a tuple

```
[15]: mytuple = (3.14, -9, "orange")
```

Access the second element

```
[16]: print("Second element (1) =", mytuple[1])
```

```
Second element (1) = -9
```

Attempt to change a value

```
[17]: mytuple[1] = 3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-17-c1b4ff1ad89f> in <module>
----> 1 mytuple[1] = 3

TypeError: 'tuple' object does not support item assignment
```

- Strings are also data structures, which simply contain a sequence of characters
  - Individual characters can be accessed using indices but like tuples they are immutable

```
[20]: mystring = "Blue car"
      print("First letter = ", mystring[0])
```

```
First letter =  B
```

```
[22]: mystring[7] = "t"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-22-db68dbfd1d09> in <module>
----> 1 mystring[7] = "t"

TypeError: 'str' object does not support item assignment
```

More on indexing

- For all the sequence types (lists, tuples, strings) you can also use negative indices or "slices"

- Negative indices refer to the position of an element from the end of the data structure
  - Rather than the beginning
  - The last element is -1, the second to last -2, etc.

```
[42]: print(mylist[-1], "is the same as", mylist[5])
      print(mylist[-2], "is the same as", mylist[4])
```

```
3.1415926535897932 is the same as 3.1415926535897932
blue  is the same as blue
```

- Slices allow us to access a range of elements, rather than just one
  - Take the form of the beginning and end index of the range, separated by a colon (`:`)
  - Note, the first index is included (inclusive) while the last is not (exclusive)

```
[50]: print("list =", mylist)
      print("2nd to 3rd element =", mylist[1:3])
```

```
list = [1, 'car', -8, 'cheese', 'blue ', 3.1415926535897932]
2nd to 3rd element = ['car', -8]
```

Dictionaries

- Dictionaries (or `dicts`) are a mapping data structure: they associate a key to a given value
  - They are assigned by a comma-separated list of key:value pairs in curly brackets `{}`

```
[23]: mydict = {"electron" : 0.511, "muon" : 105.7, "tau" : 1777}
```

- Again, we access/change the elements by `[]`, but this time passing the key not the index

```
print("Electron mass = ", mydict["electron"], "MeV")
```

```
Electron mass =  0.511 MeV
```

```
mydict["electron"] = 9.1e-31 # Switch to kg
print("Electron mass = ", mydict["electron"], "kg")
```

```
Electron mass =  9.1e-31 kg
```

- Dicts are often useful for look up tables, using one thing to find another
  - Above we have used strings as keys, which is often the case, but it can also be ints, floats, …
  - They are not so useful in calculations so we will not use them so extensively in the course

NumPy

- The flexibility of python in e.g. allowing data structures to contain mixed types comes at a price
  - Manipulations on these data structures can be relatively slow, particularly for big data sets

- The solution is the Numerical Python module
  - NumPy is a third-party python library
  - Once installed (as it is on our CoCalc system) we use it like any other module

```
[28]: import numpy as np
```

- It consists of:
  - Array structures for efficient storage of data
    - Can be 1D (vectors), 2D (matrices) or more (tensors)
  - Functions for rapid manipulation of the data (next week)

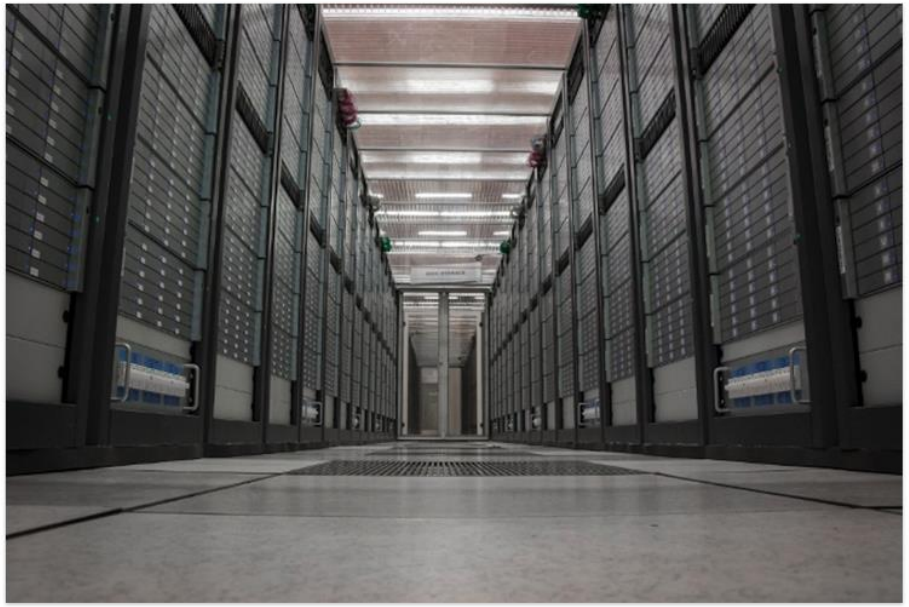
- NumPy is one of the fundamental packages for scientific computing in python
  - Many other python modules built on top of it
  - We will make heavy use of it throughout the course

An exabyte of disk storage at CERN

CERN disk storage capacity passes the threshold of one million terabytes of disk space

29 SEPTEMBER, 2023 | By Tim Smith

Article Link



A fraction of the 111 000 devices that form CERN's data storage capacity. (Image: CERN)

NumPy Arrays

- NumPy arrays can be created/filed in many ways e.g.
  - Note the `np.` to call `array` etc from the NumPy module

Create an array from a list

```
[43]: arrayFromList = np.array([0, 10.1, 20.2, 30.3, 40.4])
      print(arrayFromList)
```

```
[ 0.   10.1 20.2 30.3 40.4]
```

  - You will see other ways of creating arrays in the notebook

Create an array of zeros or ones

```
[44]: length = 10
      arrayOfZeros = np.zeros(length)
      arrayOfOnes = np.ones(length)
```

```
[45]: print(arrayOfZeros)
      print(arrayOfOnes)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

- The elements can be accessed/changed just like for a list (including slices or negative indices)
  - But the new elements must be of the same type (note, it converts int 100 to float 100.0 below)

```
[46]: print("Second element [1] = ", arrayFromList[1])
```

```
Second element [1] =  10.1
```

```
[47]: arrayFromList[1] = 100
      print("Second element [1] = ", arrayFromList[1])
```

```
Second element [1] =  100.0
```

NumPy Arrays (2)

- Multidimensional arrays are created similarly
  - Giving a tuple with the size of each dimension
  - E.g. 2D arrays by giving a tuple of rows and columns

```
[49]: nRows, nCols = 3, 2
      matrix = np.ones((nRows, nCols))
      print("matrix =\n",matrix)

matrix =
 [[1. 1.]
 [1. 1.]
 [1. 1.]]
```

Note: two sets of brackets

- The elements are accessed/changed using two indices
  - Specifying the row and column index (both from 0)
  - Can be in separate `[i][j]` or separated by a comma `[i,j]`

```
[54]: matrix[0][1] = 2
      print("matrix =\n",matrix)

matrix =
 [[1. 2.]
 [1. 1.]
 [1. 1.]]
```

```
[56]: print ("Element [0,1] = ", matrix[0, 1])

Element [0,1] =  2.0
```

- As we will see, there are a whole host of functions to manipulate NumPy arrays
  - Will look at these in more detail next week and use them throughout the course

```
[63]: np.sum(arrayFromList)

[63]: 190.9
```

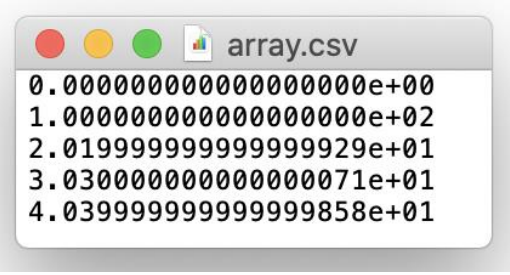
```
[115]: np.log(matrix)

[115]: array([[0.        , 0.69314718],
              [0.        , 0.        ],
              [0.        , 0.        ]])
```

Reading/writing data to/from arrays

- If you have data in an array (e.g. some results) you can write it to a file using `np.savetxt`
  - E.g. as comma-separated variables (csv), which can be read by e.g. Excel

```
[74]: print(arrayFromList)
      np.savetxt("array.csv", arrayFromList, delimiter = ",")
```

```
[  0.   100.    20.2   30.3   40.4]
```



- Conversely, data in a file (e.g. from a lab experiment) can be read into an array using `np.loadtxt`

```
[73]: arrayFromList2 = np.loadtxt("array.csv", delimiter = ",")
      print(arrayFromList2)
```

```
[  0.   100.    20.2   30.3   40.4]
```

- This also works for several columns of data
  - Giving a 2D array of nRows, nCols

```
[75]: boyle.shape
```

```
[75]: (10, 4)
```

```
[60]: boyle = np.loadtxt("boyle.csv", delimiter = ",")
      print(boyle)
```

```
[[5.7000e+01 1.0000e+00 1.0236e+03 1.0000e-01]
 [5.5000e+01 1.0000e+00 1.0580e+03 2.0000e+00]
 [5.3000e+01 1.0000e+00 1.0901e+03 8.0000e-01]
 [5.1000e+01 1.0000e+00 1.1276e+03 4.0000e-01]
 [4.9000e+01 1.0000e+00 1.1667e+03 9.0000e-01]
 [4.7000e+01 1.0000e+00 1.2060e+03 1.0000e+00]
 [4.5000e+01 1.0000e+00 1.2500e+03 2.0000e+00]
 [4.3000e+01 1.0000e+00 1.2960e+03 1.0000e+00]
 [4.1000e+01 1.0000e+00 1.3460e+03 3.0000e+00]
 [3.9000e+01 1.0000e+00 1.4086e+03 9.0000e-01]]
```

Histograms

- Once we have data we usually want to visualise it
- This can be done using the matplotlib module
  - Powerful package with many plotting types
    - Histograms, line graphs, much more
  - Takes NumPy arrays as input
- You will look at plotting some first, simple histograms in this week's notebook
  - `import matplotlib.pyplot as plt`
  - Create a `plt.figure()`
  - Fill using `plt.hist()`
  - Display using `plt.show()`

Figure is displayed inline in notebook

- Will go into more detail next week
  - And use often in subsequent weeks

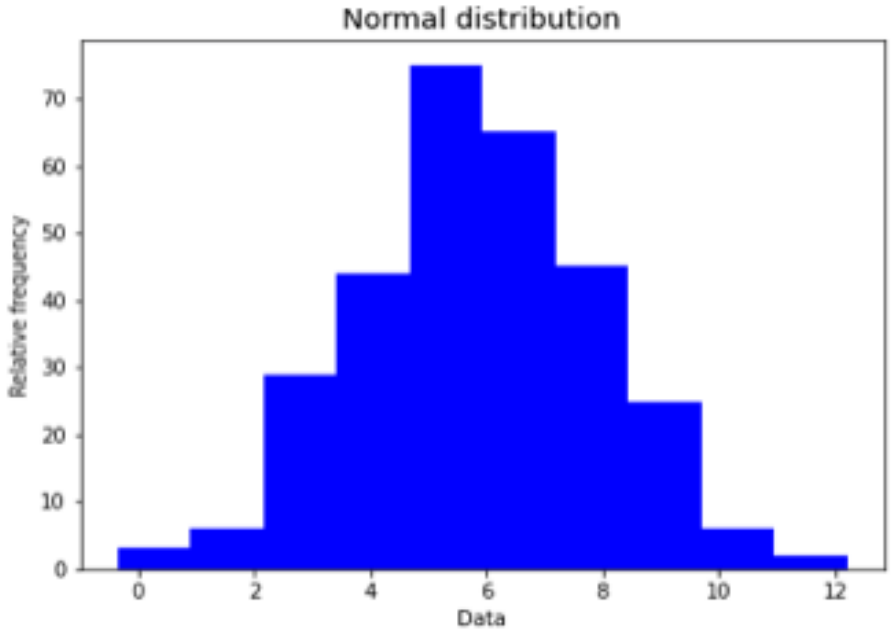
matplotlib

Load the data

[77]:
```
import numpy as np
gaussArr = np.loadtxt("normDistArr.csv", delimiter = ',')
```

Plot the data

[78]:
```
import matplotlib.pyplot as plt
%matplotlib inline

# Construct a figure with a title and axis labels
plt.figure(figsize = (7, 5))
plt.title('Normal distribution', fontsize = 14)
plt.xlabel('Data')
plt.ylabel('Relative frequency')

# Make a histogram and display it
plt.hist(gaussArr, color = 'b')
plt.show()
```



Summary

- This week we looked at how to store data in python
  - And the various data structures available to do this
- In particular, we introduced NumPy for efficient numerical storage and computations
  - This will be the bedrock of our scientific calculations
- We saw how to read and write data using NumPy
- Finally, we started to look at how we can visualise the resulting data using matplotlib
  - Again, this will be essential for scientific analysis
  - E.g. in PHSY106 labs
- You will use these in this week's problems class
  - Looking more in depth and practicing hands on