Phys105-Week02-Student

October 9, 2019

1 Introduction to Computational Physics - Week 2

1.1 Table of contents week 2

1.2 Introduction to week 2

This week, we will start by doing some calculations using Python, look at some of the ways data can be stored and manipulated. We will also look at one way of reading data into Python programs and see how to create hostograms, as you willneed these in Proctical Physics I (Phy106)!

1.3 Calculations with Python

The following cell shows how Python can be used to carry out a simple calculation, the addition of two integers. (Run the cell to get the answer; remember, the easiest way of doing this is to select the cell then press *Ctrl* + *Enter*.)

```
[1]: # <!-- Student -->
     333 + 666
```

```
[1]: 999
```

The answer should appear below the cell you have just run (and it should be 999!).

1.3.1 Week 2 exercise 1

Work out what the following operators do by trying them out on pairs of integers. (Some are obvious, some less so!) Write a very brief description of the function of each operator. 1. + 2. - 3. * 4. / 5. ** 6. // 7. %

As well as integers, we can enter and manipulate floating point numbers (`floats`). For example, we can calculate the area of a circle with radius 0.2 (in some unspecified units!) as follows:

```
[2]: # <!-- Student -->
     3.1415927*0.2**2
```

```
[2]: 0.125663708
```

We can make this calculation clearer by associating (*binding* in Python-speak) a name with each of the numbers, `r` for radius `pi` for π and `A` for area:

```
[3]: # <!-- Student -->
     r = 0.2
     pi = 3.1415927
     A = pi*r**2
     A
```

```
[3]: 0.125663708
```

Notice that writing a variable name without anything else at the end of a code cell results in the value of that variable being printed out (`A` in the example above).

In addition to the values of the variables, we can provide explanatory text by using Python's `print()` function, as shown below.

```
[4]: # <!-- Student -->
     print("Radius of circle",r)
```

```
Radius of circle 0.2
```

1.3.2 Week 2 exercise 2

Modify the cell above so that it prints out the string "Area of circle" and then the value of `A`.

1.4 Functions in Python

Suppose we want to calculate the sine of an angle, the logarithm of a number, or use some other mathematical function. Can we do this in Python? The answer is (of course) yes, but but we need to import these functions first. We shall use the versions of sine, cosine etc. from the Numerical

Python package, NumPy. As we have seen in Week 1, NumPy can be imported and the functions it provides used as shown below:

```
[5]: # <!-- Student -->
     import numpy as np
     #
     angle = 1.42 # radians
     sinAngle = np.sin(angle)
     x = 176.4
     logx = np.log(x) # logarithm to the base e
     print("The sine of",angle,"is",sinAngle)
     print("The natural logarithm of",x,"is",logx)
```

```
The sine of 1.42 is 0.9886517628517197
The natural logarithm of 176.4 is 5.172754143572691
```

The functions `np.sin`, `np.cos` etc. expect their arguments to be in radians, not degrees.

Python also allows is to create our own functions. We will see how to do this next week!

1.5 Lists and tuples

In addition to the integers and real numbers (`floats`) we have used above, Python has a large range of data types. These include `lists` and `tuples`. Lists can be created by using commas to separate their elements and enclosing them in square brackets. Various types of element are allowed, including integers, floats and strings (lists of characters, delimited by either double or single quotation marks). We can refer to an element in a list using its index (the position of the element, counting from zero). The index is enclosed in square brackets, as shown in the examples below:

```
[6]: # <!-- Student -->
     thisList = [1, 2.278, -8, "cheese", 'blue ', np.pi]
     print("thisList =",thisList)
     print("thisList[0] =",thisList[0])
     print("thisList[0] + thisList[1] =",thisList[0] + thisList[1])
     print("thisList[4] + thisList[3] =",thisList[4] + thisList[3])
```

```
thisList = [1, 2.278, -8, 'cheese', 'blue ', 3.141592653589793]
thisList[0] = 1
thisList[0] + thisList[1] = 3.278
thisList[4] + thisList[3] = blue cheese
```

As shown above, we can combine list (or tuple) elements in various ways. For example, we can add two elements. Note that the meaning of add depends on what the type of the element is. Adding numbers does just what you'd expect; adding strings concatenates the two elements (tags one onto the end of the other). You can't add elements where the notion of adding doesn't make sense (e.g. you can't add an integer to a string). Python will give you an error if you try! Note also that the string 'blue' includes a space after the 'e', this is part of the string. Without it,

```
print("thisList[4] + thisList[3] =",thisList[4] + thisList[3])
```

would give the result

```
thisList[4] + thisList[3] = bluecheese
```

The elements in a list can be changed (they are *mutable*). For example, we can assign a new value to element 3:

```
[7]: # <!-- Student -->
     thisList[3] = "car"
     #
     print("thisList[4] + thisList[3] =",thisList[4] + thisList[3])
```

```
thisList[4] + thisList[3] = blue car
```

A related data type is the tuple. This is similar to the list, but once defined its elements are fixed (they are *immutable*). They can be created as follows - notice that lists are indicated by [] and tuples use().

```
[8]: # <!-- Student -->
     thisTuple = (3.14, -9, "orange")
     #
     print("thisTuple =",thisTuple)
```

```
thisTuple = (3.14, -9, 'orange')
```

The elements in a tuple can be referred to using their indices in exactly the same way as those in a list:

```
[9]: # <!-- Student -->
     #
     print("thisTuple[2] =",thisTuple[2])
```

```
thisTuple[2] = orange
```

Because tuples are immutable, if we try and change one of the elements, we get an error:

```
[10]: # <!-- Student -->
      #
      thisTuple[2] = 27
```

```
    ---------------------------------------------------------------------------

    TypeError                                 Traceback (most recent call last)

    <ipython-input-10-e4e8c77ac3f5> in <module>
      1 # <!-- Student -->
      2 #
----> 3 thisTuple[2] = 27
```

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

An aside - tuples can also be created without using brackets, as in the example below. We will generally avoid this as it can be confusing! Notice that when it prints out a tuple, Python adds the brackets, even if we didn't put them in!

We will look at more at lists, tuples, strings, and the many other data types available in Python in future weeks. For now, we will concentrate on one of the most important data structures for scientific computing, the NumPy array.

1.6 NumPy arrays

NumPy arrays are one of the most useful ways of manipulating data when using for Python for scientific computing. They allow efficient storage of, and rapid computations with, vectors, matrices or larger dimension arrays of numbers. This efficency comes at a price: all the elements in a NumPy array have to be of the same type; `ints` and `floats` can't be put in the same NumPy array, for example.

1.6.1 One dimensional arrays

One dimensional arrays (vectors) can be created and assigned values in various ways, as is shown below. (Remember how we entered the data for the `leastsq` fit in Week 1!) Note how the index is used to refer to the array element, and that the first index is zero…

[11]:
```
# <!-- Student -->
import numpy as np
#
arrayFromList = np.array([0, 10, 20, 20, 40])
#
length = 10
arrayOfZeros = np.zeros(length)
arrayOfOnes = np.ones(length)
#
arrayOfNumbers = np.zeros(length)
arrayOfNumbers[0] = 3
arrayOfNumbers[1] = 1
arrayOfNumbers[2] = 4
arrayOfNumbers[3] = 1
arrayOfNumbers[4] = 5
arrayOfNumbers[5] = 9
arrayOfNumbers[6] = 2
arrayOfNumbers[7] = 6
arrayOfNumbers[8] = 5
arrayOfNumbers[9] = 4
#
print("arrayFromList",arrayFromList)
print("arrayOfZeros",arrayOfZeros)
print("arrayOfOnes",arrayOfOnes)
print("arrayOfNumbers",arrayOfNumbers)
```

```
arrayFromList [ 0 10 20 20 40]
arrayOfZeros [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
arrayOfOnes [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
arrayOfNumbers [3. 1. 4. 1. 5. 9. 2. 6. 5. 4.]
```

[12]:
```
# <!-- Student -->
length = 3
start = 10
stop = 30
arrayOfNumbers = np.linspace(start, stop, length)
print("arrayOfNumbers",arrayOfNumbers)
```

```
arrayOfNumbers [10. 20. 30.]
```

[13]:
```
# <!-- Student -->
print("arrayOfNumbers[1]",arrayOfNumbers[1])
```

```
arrayOfNumbers[1] 20.0
```

Another useful way of filling an array is using the function `np.linspace`. This allows beginning (`beg`) and end (`end`) values to be entered together with a number of steps (`nSteps`). The array starts at `beg` and `nStep` equally spaced values are added until `end` is reached. As there are `nStep` steps, the array contains `nStep + 1` values:

[14]:
```
# <!-- Student -->
beg = 0.0
end = 20.0
nSteps = 10
stepArray = np.linspace(beg, end, nSteps + 1)
print("stepArray =",stepArray)
```

```
stepArray = [ 0.  2.  4.  6.  8. 10. 12. 14. 16. 18. 20.]
```

If you want to know the size of the steps between in the array created by `np.linspace`, you can add the argument `retstep = True` as below (`retstep` is short for return step).

[15]:
```
# <!-- Student -->
beg = 0.0
end = 20.0
nSteps = 10
stepArray, stepSize = np.linspace(beg, end, nSteps + 1, retstep = True)
print("stepArray =",stepArray)
print("stepSize =",stepSize)
```

```
stepArray = [ 0.  2.  4.  6.  8. 10. 12. 14. 16. 18. 20.]
stepSize = 2.0
```

1.6.2 Week 2 exercise 3

Use `np.linspace` to create an array containing the numbers 10.0 to 22.0 with a step of 3.0 between each entry. How many steps will this array contain? Print out your array to check it is correct!

The values of the elements of `stepArray` can be retrieved (or changed) using their indices:

[16]:
```
# <!-- Student -->
print("stepArray[3] =",stepArray[3])
stepArray[3] = 76
print("stepArray[3] =",stepArray[3])
```

```
stepArray[3] = 6.0
stepArray[3] = 76.0
```

Because the first element in the array has index zero, the last element has index `nSteps`, even though there are `nSteps + 1` elements in the array:

[17]:
```
# <!-- Student -->
print("stepArray[nSteps] =",stepArray[nSteps])
```

```
stepArray[nSteps] = 20.0
```

The last (and next-to-last, and next-to-next-to-last) elements of the array can also be accessed by using the indices -1 (and -2, and -3). This can be useful, but can lead to confusion in some circumstances!

[18]:
```
# <!-- Student -->
print("stepArray[-1] =",stepArray[-1])
print("stepArray[-2] =",stepArray[-2])
print("stepArray[-3] =",stepArray[-3])
```

```
stepArray[-1] = 20.0
stepArray[-2] = 18.0
stepArray[-3] = 16.0
```

An alternative NumPy routine for filling arrays with evenly spaced values is `arange`. You can read about arange here.

1.6.3 Two and more dimensional arrays

NumPy arrays with two or more dimensions can also be created. The dimensions are defined by a tuple, e.g. `(3, 4)` for a matrix with three rows and four columns.

[19]:
```
# <!-- Student -->
matrix = np.ones((3, 4))
print("matrix =\n",matrix)    # \n in string "matrix =\n" starts a new line!
```

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

An aside - in defining our one dimensional arrays above, e.g. using `np.zeros(length)`*, we have used a "shorthand", which allows us to replace the tuple with a number. We could also use a tuple with only one element, as in the example below:*

[20]:
```
# <!-- Student -->
a1Darray = np.zeros((length))
print("a1Darray =",a1Darray)
```

```
a1Darray = [0. 0. 0.]
```

1.6.4 Week 2 exercise 4

Use the *.shape* method described here to check the dimensions of `matrix` are as you would expect.

We can access elements of this matrix using indices that label the row and column of the entry we want (again, both are counted from zero!). For example, we can set the values in our matrix:

[21]:
```
# <!-- Student -->
matrix[0, 0], matrix[0, 1], matrix[0, 2], matrix[0, 3] = 11, 12, 13, 14
matrix[1, 0], matrix[1, 1], matrix[1, 2], matrix[1, 3] = 21, 22, 23, 24
matrix[2, 0], matrix[2, 1], matrix[2, 2], matrix[2, 3] = 31, 32, 33, 34
print("matrix = \n",matrix)
```

```
matrix =
 [[11. 12. 13. 14.]
 [21. 22. 23. 24.]
 [31. 32. 33. 34.]]
```

And we can read them out:

[22]:
```
# <!-- Student -->
print("matrix[1, 3] =",matrix[1, 3])
```

```
matrix[1, 3] = 24.0
```

1.6.5 Slicing NumPy arrays

We have seen that we can access a single element of a 1D or 2D NumPy array using its index, We can also look at ranges of elements in an array. Here is a 1D example:

[23]:
```
# <!-- Student -->
countArr = np.linspace(0, 4, 5)
print("countArr =",countArr)
print("countArr[1:4] =",countArr[1:4])
```

```
countArr = [0. 1. 2. 3. 4.]
countArr[1:4] = [1. 2. 3.]
```

The syntax `countArr[iLow:iHigh]` returns the elements from `iLow` up to `iHigh`, including `iLow` but not `iHigh`.

The situation is similar for 2D (and higher dimensional) arrays.

[24]:
```
# <!-- Student -->
print("matrix \n",matrix)
print("matrix[1:3, 0:2] \n",matrix[1:3, 0:2])
```

```
matrix
 [[11. 12. 13. 14.]
 [21. 22. 23. 24.]
 [31. 32. 33. 34.]]
matrix[1:3, 0:2]
 [[21. 22.]
 [31. 32.]]
```

These "slices" of the arrays can be assigned new names. (Notice that Python doesn't need to be told the shape of the new array, it works it out from the slices we are requesting from the existing array.)

[25]:
```
# <!-- Student -->
square = matrix[1:3, 0:2]
print("square \n",square)
```

```
square
 [[21. 22.]
 [31. 32.]]
```

1.6.6 Week 2 exercise 5

Print out a 2×2 array consisting of the four elements in the lower right corner of the `matrix` array.

1.7 Reading data into a NumPy array

Supposing we have data that we have generated in an experiment and stored in a file. How can we read these into our Python programs and Jupyter Notebooks so we can do some analysis? There are many methods, and we will start with the simplest. This method can be used to read in files created in Excel, for example, as long as they are stored as *.csv* files, contain only numbers, all rows have the same number of elements and all columns have the same number of elements. (We will learn how to read in datasets containing strings etc. later in the module.)

The data set below (the file is called *normDistArr.csv*) consists of a column of 300 numbers from a gaussian (or normal) distribution, which is described by the function:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right].$$

Here, the mean $\mu = 6$ and the standard deviation $\sigma = 2$. (You can look at the file using MS Excel, or a text editor like notebook on windows computers or gedit on linux systems.) The file can be read into a NumPy array using the np.loadtxt function.

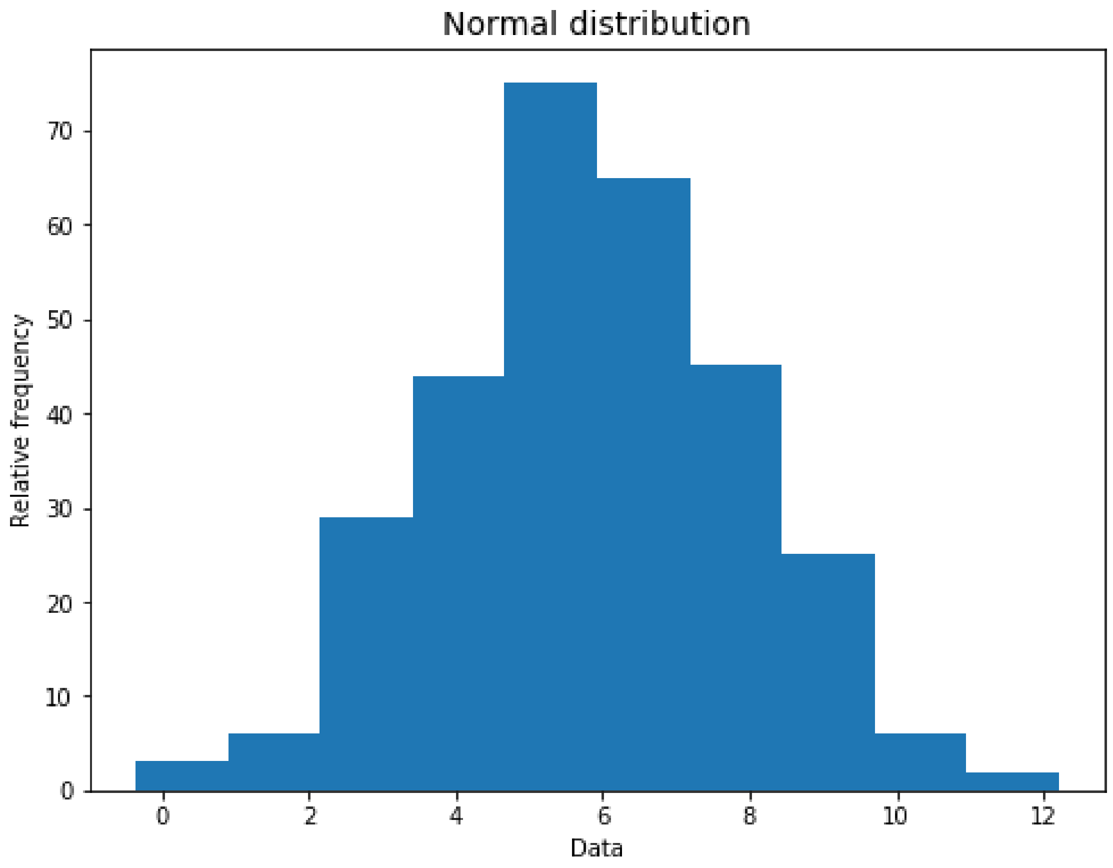
[26]:
```python
# <!-- Student -->
gaussArr = np.loadtxt("normDistArr.csv")
print("gaussArr\n",gaussArr)
```

```
gaussArr
 [ 3.5014433   5.47933717  6.7675866   5.22907705  3.82972654 10.65443803
  6.86158591  6.86463154  4.0399773   4.73606962  7.15488414  5.75048457
  7.95789573  9.18984308  3.59611096  3.24726231  8.10869102  5.92229303
  7.36057134  8.6583498   8.56689905  2.48349266  7.22861181  9.03271616
  5.60804518  4.36558745  4.10774459  6.44127782  4.79853297  5.6948679
  3.62511378  6.59827642  4.10447166  2.31323612  7.62117837  4.49548684
  5.12706197  6.09455328  5.49834473  6.33417478  7.3312924   9.39433
  8.10736917  6.58523964  4.35100202  6.20202905  6.49745134  9.43062431
  5.24541762  1.80269542  4.79409134  4.96218494  1.57450651  5.18788988
  5.40700754  6.54347574  5.08222575  6.92775565  3.9448419   6.34450136
  7.18737679  5.38369649  4.15255129  4.59442115  8.0268868   9.05722205
  7.05215193  9.37760719  6.0251228   7.68133534  5.8183938   8.06432098
  6.96847743  5.22003333  5.79602309  9.69539301  7.34922742  3.2144132
  4.84203536  5.5029985   4.95404743  6.93994268  4.15919042  5.93954612
  6.16333371  8.07109779  3.52280688  5.80863662  2.61595535  7.31916985
  6.73181965  6.1211189   7.11823602  2.55975618  6.82114215  4.76559297
  9.24198405  6.50301263  9.55627779  9.46653083  5.06120737  4.74921862
  4.75932214  6.97829563  9.18049307  4.65148433  5.6198033   2.46939583
  8.74472764  6.66242844 12.21078916  4.86476956  5.72872793  4.67499754
  6.62288208  5.84270498  8.01747028  5.90349489  5.34708765  4.10433877
  5.46534902  4.44129545  5.35805862  4.93005801  7.4901034   8.43872441
  4.31903879  7.21576365  6.8592109   3.97092601  3.70517352  9.73570103
  8.67298734  2.47989264  4.59118312  6.98954801  5.83942985  3.61886995
  6.93927333  5.18098169  9.8293508   9.40734128  3.15743925  5.74640845
  4.77251688  1.77935307  4.49123267  6.46548427  3.28770025  6.94158117
  6.02482704  5.83220032  9.28485031  3.98102275  4.45044275  7.080101
  5.50217123  7.00524574  4.30651325  6.72909345  9.31139884  3.9677962
  4.40708481  7.58343947  6.50906311  4.37824587  4.62031051  4.97531123
  6.67397461  6.11007134  2.82826125  4.47146843  3.71442391  4.88343037
  3.97236851  8.30632093  7.66675599 10.86080082  7.49370007  8.88488795
  2.34406955  5.93786302  3.17761805 11.3186519   8.20212677  8.11557512
  3.11876196  2.52056101  5.64579872  6.01504753 -0.35365821  5.06999963
  4.11278091  2.90743948  7.55989103  5.2181754   3.77263047  6.29771628
  5.90134206  5.5807144   5.31386005  7.0755266   2.20548159  8.39713388
  5.72573759  4.86551852  7.33281782  4.92018833  7.66287917  5.87098717
  8.2091381   2.92222266  3.28508851  5.32529887  4.94875765  6.84449899
  6.4626251   7.63660623  5.40145926  3.60865743  6.56378292  9.05144174
  4.39466424  8.10866394  6.64269189  5.54329987  9.82878768  7.735777
  5.82547646  9.99441802  1.88224863  4.93908206  6.32797967  4.96147939
  2.76168143  6.37126441  4.21215485  8.45380686  3.09195456  5.57870953
  4.27609472  8.26294927  6.5075924   7.39784152  3.5762947   6.47130925
  3.74229883  5.06993238  8.72788267  0.78754796  8.67055327  8.74535834
 2.52500603  6.69163292  5.22990293  3.51084521  6.85320998  8.16655779
 6.4884773   4.90286488  7.45080784  8.07877368  5.30835317  5.7599842
 6.33446856  7.61475072  2.86176607  3.36240794  3.06436815  6.26834572
 3.2174083   7.3489549   7.3391782   4.83426382  6.35267014  7.38449864
 5.81400879  6.7844552   1.49288972  7.33974267 -0.27577907  5.03217397
 7.8957643   2.97104752  5.63111399  2.39147666  7.09625438  6.01151981
 8.05050704  6.85493294  6.13088109  6.34145821  1.95683834  9.50284022
 4.89328951  7.17225687  3.26166645  5.60419237  4.51157989  7.24391362]
```

1.8 Creating a histogram

A histogram of the data stored in the `gaussArray` can be plotted as shown below:

[27]:
```python
# <!-- Student -->
import matplotlib.pyplot as plt
%matplotlib inline
#
plt.figure(figsize = (8, 6))
plt.title('Normal distribution', fontsize = 14)
plt.xlabel('Data')
plt.ylabel('Relative frequency')
plt.hist(gaussArr)
plt.show()
```

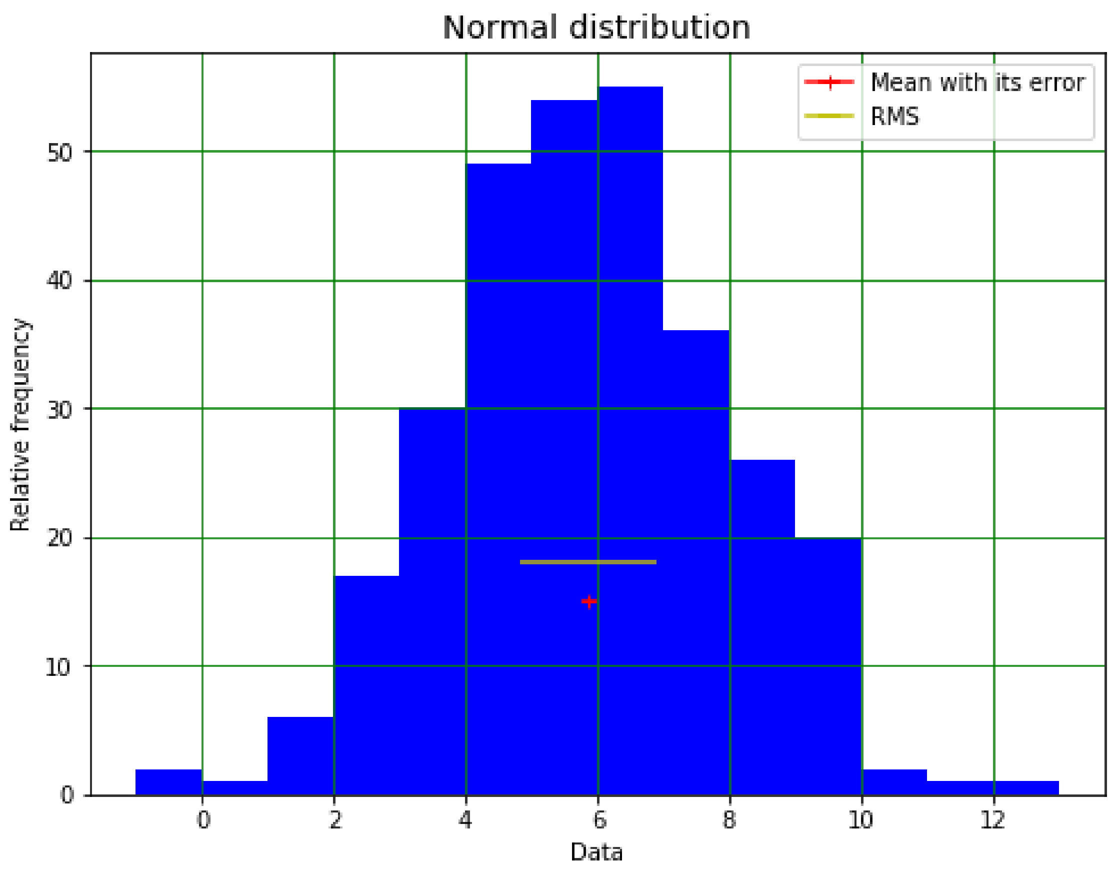


In a histogram, the data are split into "bins", the lower and upper limits of which are indicated by the edges of the bars in the plot. The area of each of the bars is proportional to the number of data points that fall within the bin. (Most histograms use bins of equal width, in which case the height of the bin indicates the number of entries it contains.) The above histogram shows that a gaussian distribution has a bell shape centred on its mean value. The width of the bell is given by the distribution's standard deviation (or root mean square, RMS).

Let's investigate the normal distribution in a little more detail. We'll plot a new histogram of the data, this time entering the bins we would like to use, rather than accepting those calculated by matplotlib, and making some other changes so the plot looks nicer. (A complete description of `plt.hist` is provided here.) We will also calculate the actual mean and standard deviation of the distribution, as well as the error on the mean, and add them to the plot.

[28]:
```python
# <!-- Student -->
#
binBot = -1.0
binTop = 13.0
binNumber = 14
binEdges, binWidth = np.linspace(binBot, binTop, binNumber + 1, retstep = True)
print("Histogram bins start at",binBot,"finish at",binTop)
print("Number of bins is",binNumber,"and width of bins is",binWidth)
#
nEvents = len(gaussArr)
mu = np.mean(gaussArr) # calculate arithmetic mean of numbers in array
sigma = np.std(gaussArr) # calculate standard deviation (error on single value)
muError = sigma/np.sqrt(nEvents) # calculate error of mean
yMu = nEvents/20
ySigma = 1.2*nEvents/20
#
plt.figure(figsize = (8, 6))
plt.title('Normal distribution', fontsize = 14)
plt.xlabel('Data')
plt.ylabel('Relative frequency')
plt.hist(gaussArr, bins = binEdges, color = 'b')
plt.errorbar(mu, yMu, xerr = muError, marker = '+', color = 'r', label = 'Mean␣
↪with its error')
plt.errorbar(mu, ySigma, xerr = sigma/2, marker = '', color = 'y', label = 'RMS')
plt.grid(color = 'g')
plt.legend()
plt.show()
```

```
Histogram bins start at -1.0 finish at 13.0
Number of bins is 14 and width of bins is 1.0
```



1.8.1 Week 2 exercise 6

Add a line to the histogram above that shows the full width at half maximum (FWHM) of the distribution. You can calculate the FWHM using the formula:

$$\text{FWHM} = 2\sqrt{2\log 2}\sigma.$$

Hint. You can use code similar to that used to show the RMS!