4 More advanced use of lists for calculations and plotting.

In this notebook we will look at some more advanced uses of lists for numerical calculations and plotting functions.

At the end of ths notebook you should be able to:

- learn how to create large lists with a fixed number of elements,
- learn how to use the scipy and numpy libraries to do numerical integration using a standard method,
- learn how to create lists to plot more advanced functions,
- learn how to use loops and lists to plot a Fourier series.
- see how mathematical text may be included in Jupyter notebooks.

4.1 Creating lists

4.1.1 Creating a list of fixed size.

In our introduction to lists we created a simple list of numbers in the following way:

In [ ]:

```
x=[1,2,3,4,5,6,7,8]
print(x)
```

Although this is straight forward it becomes impractical if we wish to create lists with a large number of elements. How do we do this?

In python the easiest way to create, handle and manipulate large lists is to use the 'numpy' library. The most common way to include this in your programme is shown below. If you wish to look a list of attributes and methods for numpy uncomment the dir(np) line. However, for details you may find it easier to look it up on line at https://numpy.org/ (https://numpy.org/) where you will find extensive resources and explanations. We will use a small set of the numpy functions here. Note how we have used the import statement to shorten the name by which we will refer to the numpy library i.e. 'np'

In [ ]:

```
import numpy as np
#dir(np)
```

Some examples of how to create lists of numbers in different ways using numpy is shown below. Run the cell and look at the output foreach variable carefully.

In [ ]:

```
import numpy as np
#
# examples of creating lists in numpy
#
a = np.arange(10)
print("a =",a)
b = np.arange(1,11)
print("b =",b)
c=np.zeros(10)
print("c =",c)
d=np.arange(1,11,dtype=float)
print("d=",d)
e=np.zeros(10,dtype=complex)
print("e =",e)
#
# or creating a list of size defined by variable n
#
n=5
f=np.arange(1,n+1,dtype=complex)
print("f =",f)
g=np.arange(1,22,5)
print("g =",g)
```

In these examples we have used the numpy functions 'arange' and 'zeros' to create a list and pre-populate it with values.

As before, note the difference between arange(10) that creates a list with 10 elements with numbers 0-9 and arange(1,11) that creates a list with 10 elements with numbers 1-10. You can also see how it is possible to populate the list with numbers of different types. The last example (list g) shows how lists with numbers at different intervals may be created. Why did we need to use 22 as the end point in the range statement if we want the final element in the list to be 21?

'zeros' is similar to arange, except it pre-populates the list with the number of elements required but all set to zero. The default is float but as you can see you can also define complex zeros or integer zeros. To check you have understood these functions try adding a new list of 10 integer zeros to the code above.

Note however, that you cannot create lists of strings in this way.

4.2 Using lists - function integration using numpy and scipy

In the previous notebook we looked at some simple code to do numerical integration using the column method. We mentioned that in general it is easier and better to use the numerical integration routines that are already available in python. To use some of these routines we need to make use of lists that we can create using the methods described above. The routine we are going to use is found in the scipy library. As we showed in the previous notebook you can get help on the scipy routines within python but as before it is probably easiet to look on the https://www.SciPy.org (https://www.SciPy.org) web site for details. There, you may find plenty of examples and tutorials. This web site also has web links to information about NumPy, Matplotlib as well SymPy - a very useful library for doing symbolic algebra calculations that we, unfortunately, do not have time to cover in this course.

In the example below we will use the scipy library to do Simpson's rule integration. Look at carefully and run the code below

In [ ]:

```
# import numpy and scipy.integrate
import numpy as np
from scipy import integrate      # Note we have only imported the integration routines from
import matplotlib.pyplot as plt

x = np.arange(2, 5.25 , 0.25)      # Note how we need to use arange to get the upper limit
y = x*x  # Note how we can create the y list from the x list using this very simple line
print("x values",x)
print("y values",y)

# plot the function to see what we are integrating

plt.plot(x,y)
plt.show

# use the scipy.integrate.simps() method

simpsint = integrate.simps(y, x)

print("Simpsons rule integration gives",simpsint)
```

There are several points to note.

As with many of the programmes you will write, you need to import the numpy, scipy and matplotlib libraries for this program. If you get errors in your code, make sure you have imported the correct libraries as one of the first things you might check.

Simpson's rule for integration is not a particular complex algorithm, indeed, you could have a go at writing it for yourself. Nevertheless, it should produce better accuracy for fewer sampling points than the column integration we used previously. In this example we have used the integral $\int_2^5 x^2 dx$ that we calculated before.

In order to use the Simpson's rule integration method properly we need, as we did before to define points in the range (a,b) including a and b. A key advantage of Simpson's rule is we should get greater accuracy for fewer sampling points. How many sampling points did we use here and why did we need to use 5.25 as the last point in the arange statement?

The following line

y=x*x

is an easy way of generating a list of y values ($y = x^2$) for each value of x in the x list. This is particularly easy in python compared to other languages. The two lists have been printed out in order for you to inspect the results. These print lines can be deleted or commented out if you wish. Although it is not necessary, we have also taken the time to plot the curve we are integrating. Plotting functions or results at intermediate stages in your calculations is a useful habit to get into and can often be an easy way to spot mistakes for example, if you had got your range incorrect.

As you can see, once we have the x and y values as lists for our function, it is easy to use the scipy routine to do the Simpson's rule integration.

It should be apparent that even with few points we get a very good result. (There is a reason why this is almost perfect for this function (x^2) - you may wish to look up Simpson's rule to see if you can understand why).

As an exercise, you might wish to modify the function by changing the code line 'y =' and changing the integration limits and the number of points using the arange function. Once yhou have done this move on to the next step.

4.3 Addressing elements in the list using the index.

Each item in a list can be treated as single variable by use of the array index. Let's look at an example with a list of strings. Look and read carefully the code below. Can you see what each line does?

In [ ]:

```
fruit = ["apples","pears","grapes","plums","pears","oranges"]
print("There are ", len(fruit),"items in the list") # check to see how many items in the li

print("item 1 is", fruit[0])
print("item 5 is", fruit[4])
print("the last item in the list is",fruit[len(fruit)-1]) # Note the use of the len() funct

combin=fruit[1]+" and "+fruit[2] # Note how we are able to concatenate (join) to strings to
print("item 2 + item 3 is", combin)
```

From this code you can see how individual items in the list may be selected by their index and how operations on different items in the list may be carried out. Note also, how we have found the number of items in the list using the len function.

IMPORTANT. You should remember that the index always starts at 0, so for a list of n items the maximum index number is $n - 1$.

We can add items to lists in our programme using the append, insert and extend functions. Examples of how these work are shown below.

In [ ]:

```
fruit.append("lemons")
print ("lemons appended",fruit)  # items added at the end of the list.
fruit.insert (2,"bananas")       #  we have inserted 'bananas at position 3 in our list'
print("bananas inserted",fruit)
fruit.extend(("peaches","mangos")) # more than one item has beebnd added at the end of our
print("there are now ",len(fruit),"items in the list")
print(fruit)
```

BEWARE It is easy to get confusing output when appending, inserting or extending lists, especially when using Jupyter Notebooks and repeating code blocks. Try running the code block above again and note what happens. If in doubt you should regularly restart the kernel in the notebook to put the python intepreter back into its initial state. However, you will note that the code block above will not run until the numpy library has been imported so you will need to run some of the previous code blocks or import the library in this code block.

You may note that we can make explicit use of the index in our for loop to list each item in the list of fruits. Look at carefully and run the code below to see how this works.

In [ ]:

```
for i in range(len(fruit)):
    print(fruit[i])
print()
print("Now list every 3rd fruit")
print()
for i in range(0,len(fruit)-1,3):
    print(fruit[i])
```

Note, by using the index of the items in the list we can take more control of the operations in the list. We will be using this a lot, especially when doing summations, products etc. later.

Note also that, although the first loop prints all the items in the list in sequence as we have done before, it would be difficult to list every third item as we have done here without using the index.

The methods that we have applied for inserting into, appending and extending lists may be applied equally to lists of numbers, see for example the code below.

In [ ]:

```
a=[1.0,2.0,3.0,4.0,5.0]
a.append(6)
print (a)
a.extend((7,8))
a.extend([9.0,10.0])
print(a)
print(a[5],type(a[5]))
print (a[7],type(a[7]))
print (a[9],type(a[9]))
```

WARNING. There are several pitfalls you may come across using these methods if you are not careful. Most notably as demonstrated above, it is possible to have lists with mixed data types. This is easily done when adding items to lists and if you are not careful, could produce unexpected results. Note, also how both additional lists _OR_ tuples may be used to extend lists, the effect is the same but how you use them in programmes may be different.

If you are still unsure about the difference between a list and a tuple, look at carefully and run the code snippet below.

In [ ]:

```
a=[2,3]
b=(2,3)
print("a[0] is in a list, a[0] = ",a[0],type(a))
print("b[0] is in a tuple, b[0] = ",b[0],type(b))
a[0]=5
print("We have changed the value of a[0], new value = ",a[0],type(a))
b[0]=5
print("This line will not be executed as we cannot change a tuple, b[0] =",b[0],type(b))
```

In this case 'a' is a list and 'b' is a tuple. As you can see, python gives an error if you try to assign a new value to an item in the tuple - it is immutable.

In general it is better to define your arrays carefully and fix the size you need. Uncontrolled addition to lists, especially with unbreakable loops (more of which we will see later) may lead to excessive memory use, lead to a reduction in execution speed or, in dire cases, complete programme failure (a crash).

4.4 Plotting Fourier series stored in lists

Now that we have used and learnt how to use lists and how to compute values in list, we will move on to something practical. In this section we will take what we have learnt so far, loops, functions, plots etc. to plot the fourier series of a function. In your mathematics course you will have learnt about Fourier series and how you may represent any periodic function in terms of an expansion in $\sin$ and $\cos$ coefficients. A complete description of Fourier Series may be found in "Mathematical Techniques", by Jordan and Smith. The relevant parts in th 4th edition are from pages 562 to 585.

4.4.1 Fourier Series

For any periodic function $P(t)$ with period T, we may write $P(t)$ as a Fourier series as follows:

$P(t) = \frac{1}{2}a_0 + \sum_{n=1}^{\infty}(a_i \cos(n\omega t) + b_i \sin(n\omega t))$,

where $\omega = \frac{2\pi}{T}, -\infty < t < \infty$.

The values a_i and b_i are known as the Fourier coefficients of the function $P(t)$. In a general function $P(t)$, you will expect values for both a_i and b_i. However, if $P(t)$ is an even function you will only have coefficients a_i and if $P(t)$ is an odd function you will only have b_i terms. The Fourier coefficients may be detemined using the following relationships:

$a_N = \frac{\omega}{\pi}\int_{-\pi/\omega}^{\pi/\omega} P(t)\cos(N\omega t)dt$

$b_N = \frac{\omega}{\pi}\int_{-\pi/\omega}^{\pi/\omega} P(t)\sin(N\omega\ t)dt$

for $N = 1, 2, 3, \dots,$

You should have practised how to find the Fourier coefficients for several waveforms in your maths course, so we will not repeat the calculations here. However we will show below, a method for calculating and summing a known Fourier series with python. We will choose the particular case of square wave that is forced to be an *odd* function.

4.4.2 The Fourier coefficients of a square wave represented as an odd function.

We will follow example 26.8 from Jordan and Smith. In this case we take a function $P(t)$, with period 2 so that $\omega = \pi$ with a basic interval $t = -1$ to 1. i.e

$P(t) = -1\ ; -1 \leq t < 0$

and

$P(t) = 1\ ; 0 \leq t \leq 1$

The Fourier coffients are then:

$b_i = \frac{1}{i}$ when i is an odd number and

$b_i = 0$ if i is an even number.

In the following code we show how we can calculate the Fourier coefficent up to any finite value of N. You should look carefully at and run the code below. You should start with relatively small numbers of N i.e. < 10 and then increase N gradually to see the effect of including more terms in the expansion.

In [ ]:

```
import numpy as np
import math as mp
import matplotlib.pyplot as plt


N = int(input('input N>')) # The program will prompt us for maxium N in our series
b=np.zeros(N+1)
#
# calculate the coefficients of b and store them in the list
# note that we have set the list 'b' to zero already so by carefully
# using the range function we can avoid calculating values for even values
# of b.
#

for i in range(1,N+1,2):
    b[i]=1.0/i
#
# we now wish to determine and plot the function.
# npts just defines the number of t points to use when plotting.
#

npts = 1000    # this is the number of 't' points we have chosen to plot.
t = np.linspace(-2.0,2.0,npts)  #define the t point s for the plot.
Pt = np.zeros(npts) # the x and y values for the plot will be in t and Pt respectively.

for j in range (npts):
    for i in range (1,N+1,2):
        Pt[j]=Pt[j]+b[i]*mp.sin(i*mp.pi*t[j])

plt.plot(t,Pt)
plt.show()
```

There are a number of things to observe in this code.

We've introduced the idea of asking for input to the programme (make sure you hit \< RETURN > when you enter the number ! This may not be entirely necessary for this code fragment - we could just have assigned N in the code as we have done before.However, it is more useful to input data when required rather than trying to remember where to change the code. It is also less prone to error.

Note, that when we calculated the b coefficients we deliberately avoided the even values of i using the range function. Be careful if you adapt the code later for a different Fourier series where the even coefficients might be non zero!

We have introduced a new function from the numpy library, namely 'linspace'. This is easier to use than the range function in some cases. The point is that it generates a list that includes definitely the first and last points in the list and with an even spacing between the first and last points according to the number of points required. In this example we have calculated $P(t)$ for a 1000 t points. You might experiment with fewer or more points but recognising there is no point in trying to plot more points than the resolution of your screen/printer.

Make sure that you understand how the values of $P(t)$ have been calculated. We have used what is known as a 'nested' for loop, i.e. a loop within a loop. In particular, for this example we have summed the Fourier component at a particular of t (the inner loop) and then we have calculated it for each value of t (the outer loop). We could have executed the loops in the reverse order if we had wished.

5 Writing equations

You may wish to look carefully at the markdown cells in this notebook. We have been able to introduce mathematically formatted text (integrals and summations) without which the notebook would be very difficult to follow. Jupyter notebooks use the $L_A T_E X$ text processing language. The easiest thing to remember is equations are enclosed between $ signs. By and large writing equations in $L_A T_E X$ is not difficult and is a useful skill to learn. It is quicker to generate equations in text in this way than, for example, using the pull down menus in equation editor in Micorsoft Word. (Indeed, the latest equation editor in word allows you to type $L_A T_E X$ type entries which are reformatted on the fly)! You may well find it is also easier to produce well formatted documents.

Look at the markdown language in this cell to see some common uses. Some you may guess quite easily afterwards.

$\int_0^\infty$: note how we write ∞

$\sum_{i=1}^N$: note how we need to include curly brackets for the subscript

$\prod_{i=1}^N$

$\alpha, \gamma, \Gamma, \omega, \Omega$

$\sin, \cos, \sinh, \exp$

$\frac{sin(\theta)}{\cos(\theta)} = \tan(\theta)$

$\simeq, >, \geq, \leq, <, \sim$

$\rightarrow, \leftarrow$

6 Summary

In this notebook you have learnt how to:

- create and populate lists using the numpy library,
- insert into, append or extend lists, dynamically in your programme,
- use an integration method from the scipy library (Simpson's rule),
- use the list index to do calculations on an item in the list,
- use 'nested' loops,
- use lists and loops to plot a Fourier series,
- use $L_A T_E X$ to include mathematical equations in your notebook.