3 Functions and loops with numerical integration as an example

In the previous notebooks you have used python to execute a series of programme steps sequentially. In this notebook you will be introduced to two new *programming structures*: the *for* loop and *functions* that are used for carrying out repetive programming tasks. You will learn how to use these structures with reference to the task of basic numerical integration. Although the method we will use is simple, it is the basis around which better and more accurate methods can be built.

In this notebook you will learn and understand how to use:

- 'for' loops to carry out calculations for a fixed number of iterations,
- 'functions' as a way to simplify and clarify repetive calculations for different values of a variable,
- combine the use of for loops and functions for simple methods of numerical integration,
- the 'range' function to define the index for a loop.

3.1 For loops.

3.1.1 The structure of a 'for' loop

Look at and run the python code fragment below

In [ ]:

```
for i in range(3):
    print(i)
```

It is worth taking some time to look at this code. We note that we have introduced a variable _i_ (in this case it will be an integer) as an index. The programme interprets the first line as:

"for all the possible values of _i_ defined after _in_ , execute all the following indented code with each value of _i_ in turn, until all vlaues of _i_ have been used."

The programme will continue, after the indented code, once this 'loop' has been completed.

The code above shows the values of _i_ that were used in the loop.

IMPORTANT.

- The first line in the 'for' loop must end with a colon (:). If you forget this you will get an error. (Delete it in the code above and see what happens).
- The lines in the loop must be indented. If you don't have at least one indented line you will get an error (Remove the indentation in the code above and see what happens). You may also note that in Jupyter notebooks the colour of the code line changes after 4 spaces of indentation. 4 spaces is the convention in python but you can use more, or less spacing.
- You must be consistent in your indentation. Python (unlike other languages) uses indentation to identify code block.
- The range function used in the for loop, used in the example above, often leads to confusion. In this code we have 3 values of _i_ , as we might expect, but they correspond to the values 0,1,2 and NOT 1,2,3.

Look at and run the code fragment below

In [ ]:

```
for i in range(3,12,3):
    print(i)
```

This shows a more complete definition of the range function. In this case the first number in 'range' is the first value of _i_ to be used, the second number indicates the last value of _i_ to be used and the third number the amount _i_ should be incremented each time until it reaches the last value.

IMPORTANT. Note, we might expect the loop to use a value of *i=12* . However, it only continues the loop when the index is less that the last value in the range. If it is equal to the last number the execution of the loop ends. Again, this is common source of confusion and errors in calculations. Beware!

Note. The values used in the range function must be integers.

Look at and run the code fragment below

In [ ]:

```
for i in [2,1,4,5]:
    print(i)
```

From this you can see that the 'range' function actually defines a hidden, sequential list of numbers that the loop cycles through. In the example above you can see python goes through each successive element of a list of integers without using a range function.

To make matters perhaps more confusing, look at and run the code fragment below

In [ ]:

```
for fnum in [1.2,2.3,4.5]:
    print(fnum)

for astring in ["apples","pears","bananas",5.0]:
    print(astring)
```

From this we can see we can have a list of any data types to used in each iteration of our loop. Indeed, we can even mix the data types in the list! By and large in this course we will stick to using the *range* function to generate values for an index for our loop but you may see other forms on your travels!

3.1.2 An example of a 'for' loop for calculation - calculating factorials

The factorial of a number $n!$ is defined as (provided we assume $n \geq 1$)

$n! = 1 \times 2 \times 3 \ldots . \times (n-1) \times n$

How can we use our for loop to calculate the factorial of a number for arbitary n.

Look at and run the code fragment below

In [ ]:

```
n = 10
fac = 1
for i in range(1,n+1):
   fac=fac*i

print("The factorial of", n ,"is",fac)
```

Look very carefuly at this code and make sure that you understand how it works. You will be using loops of this form a lot.

Points to note are:

- We are calculating the factorial of the variable _n_ set in the first line. Try different values and check the result. Note, it must be integer > 1.
- We have set a value of fac before we execute the loop. Why did we need to do this? (try removing the line and see what happens).
- Why did we need to use (n+1) in our range rather than n?
- Why didn't we just use the 'range(n+1)' for our loop?
- Note how the unindented print statement is executed after the loop is completed.
- We have introduced a more complicated print statement that prints out useful text and two of the variables used in the programme.

3.2 Functions

3.2.1 What is a function?

You are already familiar with the idea of a function in python. For example, the sin() and cos() functions from the *math* package that you have seen earlier. The key point is that these will 'return' the value of sin etc. for any valid variable that we 'pass' to it. However, although python has many inbuilt functions and functions that can be imported, there are many times when we want to define our own. For example, suppose we have a polynomial function,

$$f(x) = 4x^2 + 3x + 1$$

that we wish to evaluate at many points in our programme. We could have a line, for example,

*a = 4*x**2 + 3*x+1*

at each point in our programme that we wish to evaluate the function. Each time we would need to set _x_ before caclulating _a_. This would not only be tedious but also likely to produce errors into you code.

The better and easier way is to define our own function in our programme. To see how this is done look at and run the following code fragment (it may be a good idea to restart the kernel at this point).

In [ ]:

```
def fpoly(x):
    ans = 4*x**2+3*x+1
    return(ans)

print(fpoly(1))
print(fpoly(2))
print(fpoly(2.5))
```

Take some care to look at this code fragment. You should note that our function has the name *fpoly* and that it returns a value for *fpoly* at the point _x_. Verify that it has evaluated the function as you expect.

IMPORTANT. Note, that like our loop the *def* line must end with a colon and the code for the function must be indented. The function should also be defined (best at the top of the code) before you need to call it.

You may try and change this code to generate a function of your own to test your understanding of functions.

You may also note that a function may passed any number of variables but it will only return one value. For example, look at the trivial function below,

In [ ]:

```
def fmult(a,b):
    ans=a*b**2
    return (ans)

print(fmult(2,4))
```

Aside. Python can be used as/is an *object oriented* language. A function definition is a much simpler version of a'class' definition we would write if writing properly object oriented code. In this unit we will not be using any user defined classes (that may contain functions) but, for example, when you import the maths package you are generating an 'instance' of the math 'class' that includes all the trig functions etc. You may use object orientation and python classes in more advanced courses.

3.3 Simple numerical integration - using for loops and functions

3.3.1 Numerical integration

In your calculus courses you will have become familiar with the idea of integration (anti-differentation) and the analytical methods by which you may evaluate definite integrals. For example, you should be able to evaluate the definite integral

$$\int_0^2 x^2 dx$$

to obtain the result 8/3. You have learnt many techniques for eavluating integrals of many forms (by parts etc) but occasionally you may have found an integral which you were not able to evaluate. This may be because your knowledge of calculus is not advanced enough or no analytical result exists. However, in cases where it is not possible or we are unable to evaluate a definite integral we may still use numerical methods to find a (approximate) solution. The methods used for numerical integration and where and when they are used would be a complete course in itself (for example, how to deal with infinite and semi-infinite integrals). Here we will look at a simple method, column integration as s demonstration of the ideas.

You should be familiar with the idea of 1D integration as evaluating the area under a curve. If not, I refer you to basic texts on calculus. Given this, the easiest way to approximate the area under a curve is to divide the curve to be integrated into evenly spaced regions on the x axis and then to draw a series of columns that are centred on each region and whose height is given by the value of the function evaluated at the centre of each region. The approximate area under the curve is then the sum of the areas of all these columns. Mathemtically we would write this as:

$$\int_a^b f(x)dx \simeq \sum_{i=1}^{n} f_i(x)\Delta x$$

where $\Delta x = (b-a)/n$ and $f_i(x) = f(a + \Delta x \times (i - 1/2))$ where n is the number of regions in to which we have divided the range (a, b).

Let's see how we can apply this for a simple function that we know. For example, to evaluate $\int_2^5 x^2 dx$

Look at and run the code below

In [ ]:

```
# A programme to integrate a function, f(x), numerically
# Firstly, let's define the f(x) as a python function.
#
def f(x):
    ans=x**2
    return(ans)
#
# The integration limits are a, b and the region is divided into n columns
# the width of each column will be (b-a)/n
#
n = 10
a=2.0
b=5.0
deltax=(b-a)/n # calculate delta x
#
# The answer to the integral will be stored in the variable 'result'.
#
result = 0
for i in range(1,n+1):
    result=result+f(a+deltax*(i-1/2.0))*deltax

print(result)
```

Hopefully you are able to find the analytical result for this calculation as $\int_2^5 x^2 dx = 39$

Now let's look at the code.

You will see that we have introduced comments into our code for the first time. The way to do this is to use the # symbol. Anything that appears after a # symbol in a line is ignored when python runs the code, so we can have comments as lines by themselves or we can include them after the end of code on a line. How you use and layout you comments is for you but they should be aimed to help you read and understand the code you have written, especially if you, or someone else needs to understand it some time later. Don't put in unnecessary comments that do nothing to help understand the code. For example,

n=10 # set n equal to 10

does not really tell us anything new!

The first thing we did was to write a python function that evaluates our function at the point x. Note, by defining a function, it is much easier to evaluate a different integral by changing the definition here.

The next lines set the limits we have required for the integration, the number of columns we are going to use to approximate the integral and the width of each column ($\Delta x). Note we have written n = 10, with no decimal point - why is this important? Try running the code with *n=10.0*.

As we are going to store the answer in the variable *result* it is important to make sure it is zero before we do the evaluation. Forcing it to be zero avoids making any mistake if a value for *result* has already been calculated somewhere else in the programme. You can see why this is important if you remove or 'comment out' the line

result =0

If you then run the code cell again and again you see that the answer continues to change as we are adding to *result* each time (as we have stored a value in the variable *result* the first time we ran it).

Inspect the *for* loop carefully. Why did we choose the range from 1 to (n+1)? Make sure you understand why it was done this way in this example. Can you see how the value of the function has been evaluated at the centre of the x region for each column?

Finally increase the value of _n_ (to 100, 1000 etc) and see what the effect on your result is? You should see that we get closer and closer to 39 as we _n_ increases. It is tempting to believe that we could keep increasing the number of columns to get a better result (as _n_ increases we appear to converge to a single value - 39). This is tempting, but smarter algorithms might demand less computer time and there is a limit, due to the precision in which the calculation can be done (rounding errors) that means this strategy would eventually fail.

The code fragment below is identical to the original version above except the 'for' loop starts with

for i in range(n)

If we used this line in the code fragment above (rather than *for i in range(1,n+1)*) we get the wrong result? Why? If you understand the range function properly you should be able to modify the result line in the code fragment below (where its says # +?) to obtain the same answer (without changing any other line). Try it.

In [ ]:

```
# A programme to integrate a function, f(x), numerically
# Firstly, let's define the f(x) as a python function.
#
def f(x):
    ans=x**2
    return(ans)
#
# The integration limits are a, b and the region is divided into n columns
# the width of each column will be (b-a)/n
#
n = 10
a=2.0
b=5.0
deltax=(b-a)/n # calculate delta x
#
# The answer to the integral will be stored in the variable 'result'.
#
result = 0
for i in range(n):
    result=result # +?

print(result)
```

The scientific library *scipy*, that we will be using later, has a number of simple and advanced routines for doing numerical integration. In practise you will be better off selecting the appropriate routine from this library rather than writing your own integration routine. If you need more than availabe in *scipy* you will be turning into a numerical specialist.

To get some idea of what is available uncomment the two lines below and execute the code fragment. You can see how large the help file for the scipy integration routines is! We will look at how to use one of these routines later.

In [ ]:

```
#import scipy.integrate as integrate
#help(integrate)
```

3.3 Summary

In this notebook you have:

- learnt how to use the python 'for' loop to do execute a series of repetitive steps,
- learnt about the pitfalls and difficulties in correctly using the python 'range' function,
- learnt how to create and use simple python functions,
- learnt how to use loops and functions in a python routine to carry out simple numerical intergration.

Exercises

You may wish to do this in code cells in this notebook, or by creating your own notebook.

- Experiment in writing your own functions. For example, see if you can write your own version of the *sinh* and *cosh* functions. (you need to use the *exp* functions from the *math* package). You can check your functions against the ones in the standard package.
- With the functions you have defined take the basic integration code and modify it do to definite integrals for your functions. i.e find $\int_1^3 \sinh(x)dx$ using your function fo *sinh*.
- Use a *for* loop to find the sum of the first _n_ integeters, i.e. $\sum_{i=1}^{n} i$. You can base you code on the code we used to calculate the factorial of a number. If you are feeling adventurous, modify this to find the sum of the square, or the cubes, of the first _n_ integers.

In [ ]: