Lab3_Notebook

September 26, 2020

1 Lab 3: Lists and Numerical Methods

In this notebook you will learn how to store data within lists and how Python can be used to numerically solve mathematical problems.

1.1 Lists in Python

So far we have only worked with variables that have a single stored values. However, it may be the case that we want to store a collection (i.e. list) of values - for example a list of data points we wish to plot, or simply a list of values that we want to store a loop through at a later date. Python knows a number of compound data types, used to group together many values, and store them "together". The most versatile is the `list`, which can be written as a list of comma-separated values (items) between square brackets `[ ]`.

Using the square brackets on the right side of an assignment statement defines a python list. Then individual "items" in the list can be accessed and manipulated as in the examples below.

[1]:
```
squares = [1, 4, 9, 16, 25]
```

Here we have defined a set called `squares`, and it has five integer values stored in it.

[2]:
```
print(squares)
print(type(squares))
```

```
[1, 4, 9, 16, 25]
<class 'list'>
```

[3]:
```
print(squares[3])
```

```
16
```

In the above statement, `squares`[3] calls the 4th member of the set (remembering 0 calls the 1st member). This indexing starting at 0 is common to lists and arrays (see Lab 4)

[4]:
```
print(squares[0:2])
```

```
[1, 4]
```

In the above case, `squares`$[0:2]$ calls for items 0 and 1 (but not 2!)

[5]:
```
squares[4]=0
```

squares[4] = 0 changes (overwrites) the 5th item of the list to 0

[6]:
```
print(squares)
```

```
[1, 4, 9, 16, 0]
```

Finally, we can add more items to the list using the append command..

[7]:
```
squares.append(8)
```

...will add the value 8 to the list - i.e. there are now 6 items in the list

[8]:
```
print(squares)
```

```
[1, 4, 9, 16, 0, 8]
```

Once we have data in lists, we can do all kinds of things, including plotting - see later. We can also loop through lists, searching for items or performing operations...

There is another version of a for loop for lists. It has the form:

`for <variable name > in <name of list> :`

which loops through all items in the lits, and `<variable name >` will have the value of each item of this list in turn. Look at the following examples....

[9]:
```
my_list = [2,4,6,8,10,1,3,5,7,9]
for x in my_list:
    print(x)
```

```
2
4
6
8
10
1
3
5
7
9
```

[10]:
```
total = 0
for x in my_list:
    total=total+x
print("Sum of all elements in list is",total)
```

```
Sum of all elements in list is 55
```

...by the way, you can do this using the `sum()` function - one of a number of functions that can operate on lists...

[11]:
```
print(sum(my_list))
print(max(my_list))
print(min(my_list))
```

```
55
10
1
```

1.2 Numerical Methods in Python

This section introduces the use of the Euler method to solve ordinary differential equations (ODE's). For many of the differential equations we need to solve in the real world, there is no "nice" algebraic solution. That is, we can't solve it using the integration techniques that are available to us (e.g. separation of variables, integrable combinations, or using an integrating factor).

As a result, we need to resort to using numerical methods for solving such ODE's. A numerical approach gives us a good approximate solution and the Euler method is a first-order numerical procedure for solving differential equations with given initial conditions.

We want to solve a problem of the following form :

$$\frac{dy}{dx} = f(x, y)$$

$$y(x_0) = y_0$$

where $f(x, y)$ is some function of the variables x and y that are involved in the problem and x_0 and y_0 are the initial conditions. Imagine that we want to know what the value of y will be for some specific value of x. In Solving this numerically, we start with our pair of values (x_0, y_0), and use these to generate another pair of values (x_1, y_1) where x_1 is x_0 plus a small step h. This is explained below…

1.2.1 The Euler Method

Euler's Method assumes our solution is written in the form of a Taylor's Series. That is, we'll have a function of the form:

$$y(x + h) \approx y(x) + hy'(x) + \frac{h^2 y''(x)}{2!} + \frac{h^3 y'''(x)}{3!} \ldots$$

Here and from now on $\frac{dy}{dx}$ is written as $y'(x)$ and $y''(x)$ corresponds to $\frac{d^2y}{dx^2}$ etc. This expression gives us a reasonably good approximation if we take plenty of terms, and if the value of h is reasonably small. For Euler's Method, we just take the **first 2 terms only**:

$$y(x + h) \approx y(x) + hy'(x)$$

The last term is just h times our $\frac{dy}{dx}$ expression, so we can write Euler's Method as follows:

$$y(x+h) \approx y(x) + hf(x,y)$$

So how do we use this formula? We start with some known value for y which we have called y_0. It has this value when $x = x_0$ (we are therefore making use of the initial values).

The result of using this formula is the value for y one h step to the right of the current value. Let's call it y_1. So we have:

$$y_1 \approx y_0 + hf(x_0, y_0)$$

where y_1 is the next estimated solution value; y_0 is the current value; h is the interval between steps; and $f(x_0, y_0)$ is the value of the derivative at the starting point (x_0, y_0).

To get the next value y_2 we would use the value we just found for y_1 as follows:

$$y_2 \approx y_1 + hf(x_1, y_1)$$

where y_2 is the next estimated solution value; y_1 is the current value; h is the interval between steps; $x_1 = x_0 + h$ and $f(x_1, y_1)$ is the value of the derivative at the current (x_1, y_1) point.

We can write Euler's method in terms of iterative formulas :

$$x_{n+1} = x_n + h$$

$$y_{n+1} = y_n + hf(x_n, y_n)$$

to find the points in our numerical solution. Rather than doing these calculations by hand, we can use Python to repeating these lines of code many times (in a loop) and create an algorithm that outputs corresponding values of x and y in a step-by-step approximate method until we have reached the end of the desired interval.

1.2.2 Euler method example :

You are asked to solve the initial value problem numerically for

$$\frac{dy}{dx} = x + y$$

$$y(0) = 0$$

finding a value for the solution at x = 5.0 and using step sizes h = 0.01. If you were doing this by hand, you would start x_0, y_0, and then keep doing Euler steps, find corresponding x_n and y_n

values. Obviously, here we have chosen a small value of h to make the answer as accurate as possible! However, this means 500 calculations! So we'd better write a code to do it...

Here is the Euler method for solving the problem above with h = 0.01 and iterated 500 times.

[12]:
```
h = 0.01
N_steps = 500
y=0                                 # Sets the initial value of x
x=0                                 # Sets the initial value of y
for i in range(1,N_steps+1):        # Goes round the loop N_steps times
    y_prime = x + y                 # calculates the current dy/dx for the current␣
 ↪values of x and y
    y = y + y_prime * h             # increments the value of y with the Euler␣
 ↪method
    x = x + h                       # increments the value of x with h

# Now the loop is finished, we can print out the final results...

print("for x =", x,"the numerical solution of the ODE is y =", y )
```

```
for x = 4.999999999999938 the numerical solution of the ODE is y =
138.7727724325731
```

In this particular case, this ODE, with y(0) = 0, has an analytical solution, of the form...

$$y = e^x - x - 1$$

...and so we can check our answer....

[13]:
```
import math
x = 5
true_answer = math.exp(x) - x -1
print("for x =", x,"the true solution of the ODE is y =", true_answer )
```

```
for x = 5 the true solution of the ODE is y = 142.4131591025766
```

... so we can see that the results are close, but not exacly the same

Let's structure this programme a little better using functions...

[14]:
```
def func(x,y):                         # function returns dy/dx for the current␣
 ↪values of x and y
    return x+y                         # The expression for dy/dx

h = 0.01
N_steps=500
x = 0                                  # Sets the initial value of x
y = 0                                  # Sets the initial value of y

for i in range(1,N_steps+1):           # Goes round the loop N_steps times
```

```
    y = y + h*func(x,y)              # increments the value of y with the Euler␣
 ↪method
    x = x + h                        # increments the value of x with h

print("for x =", x,"the numerical solution of the ODE is y =", y )
```

```
for x = 4.999999999999938 the numerical solution of the ODE is y =
138.7727724325731
```

1.3 Using lists in numerical methods

When doing numerical methods like this we often need to keep track of the values as we go. This is so that we can analyse the results afterwards, plot the results or simply store the output for later use. This is where lists and arrays (see Lab 4) can really help us.

Lets repeat the above exercise, but this time we are going to store the successive values of x and y in lists. We saw earlier the `append` command, which adds a value to an exisiting list. We can therefore do this in our Euler method, storing the successive values of x and y as we go...

First we need to define x and y as lists, and then we use the `append` command, each time we go round the loop....

```
[15]: def func(x,y):                   # function returns dy/dx for the current values␣
 ↪of x and y
    return x+y                   # The expression for dy/dx

h = 0.01
N_steps=500
x = 0                            # Sets the initial value of x
y = 0                            # Sets the initial value of y
x_list=[x]                       # Creates a list and sets the first value to be␣
 ↪the initial value of x
y_list=[y]                       # Creates a list and sets the first value to be␣
 ↪the initial value of y
for i in range(1,N_steps+1):
    y = y + h*func(x,y)
    x = x + h
    x_list.append(x)             # Adds (appends) the new value of x into the list
    y_list.append(y)             # Adds (appends) the new value of y into the list
```

Now let's look at these lists and see what we have. We can use `len(x_list)` to look at how many entries the list has. There should be 501 since we had initial values of x and y and then 500 increments.

We can also examine specific elements of the list, using `print(x_list[0])` etc.

It is useful to know that `y_list[-1]` will be final entry of the list, which of course is our final answer....

[16]:
```
print("Length of x list is", len(x_list))
print("First entry of x list is", x_list[0])
print("First entry of y list is", y_list[0])
print("100th entry of x list is", x_list[99])
print("100th  entry of y list is", y_list[99])
print("")
print("For x =", x_list[-1],"the numerical solution of the ODE is y =",␣
 ↪y_list[-1])
```

```
Length of x list is 501
First entry of x list is 0
First entry of y list is 0
100th entry of x list is 0.9900000000000007
100th  entry of y list is 0.6880334944767588

For x = 4.999999999999938 the numerical solution of the ODE is y =
138.7727724325731
```

1.3.1 Plotting using lists

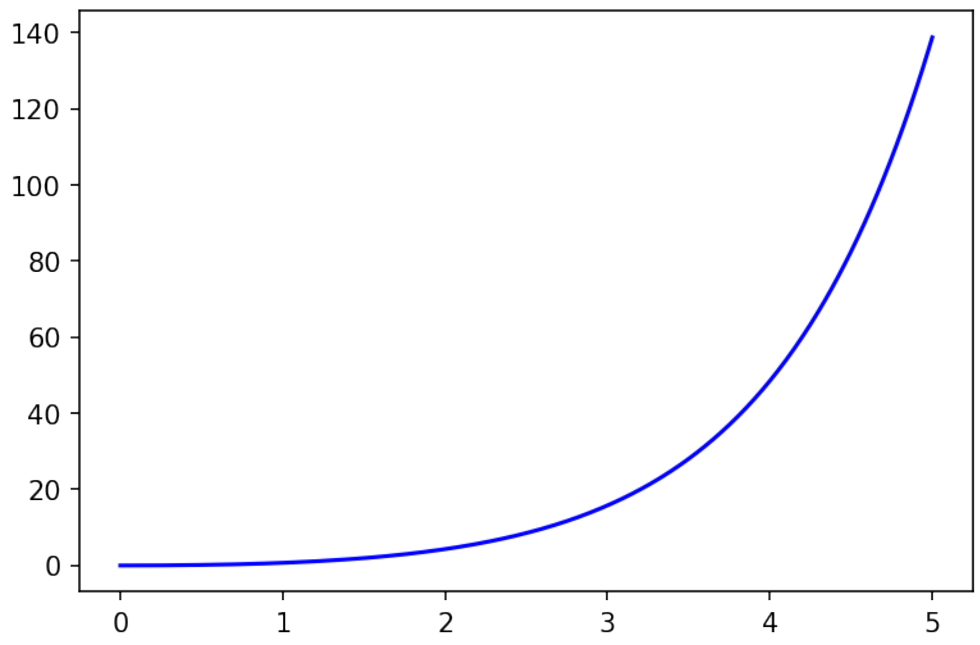
One of the main benefits of storing values in lists is plotting. In the next lab session you'll learn about plotting in more detail. For now, you can just use the lines of code below, which simply plots one list against another and draws a blue line between the data points. Now we can see the shape of our approximate solution…

Run the cell below and see how easy it is…

[17]:
```
import matplotlib.pyplot as plt        # Imports the matplotlib library so we␣
 ↪can draw graphs
plt.rcParams['figure.dpi'] = 150       # Sets the resolution (in dpi) of the␣
 ↪figure

plt.plot(x_list, y_list, 'b')          # Sets up a plot of one list against␣
 ↪another, blue line

plt.show()                             # Shows the plot
```



Since we know the true solution here, $ y=e^x - x - 1 $, we could also create another list, and store all the "true" values of y in that list, for each value of x (we need to append to that new list inside the loop of course). We can then plot the approximate solution and the true solution side by side...

Run the cells below ro see an example of this:

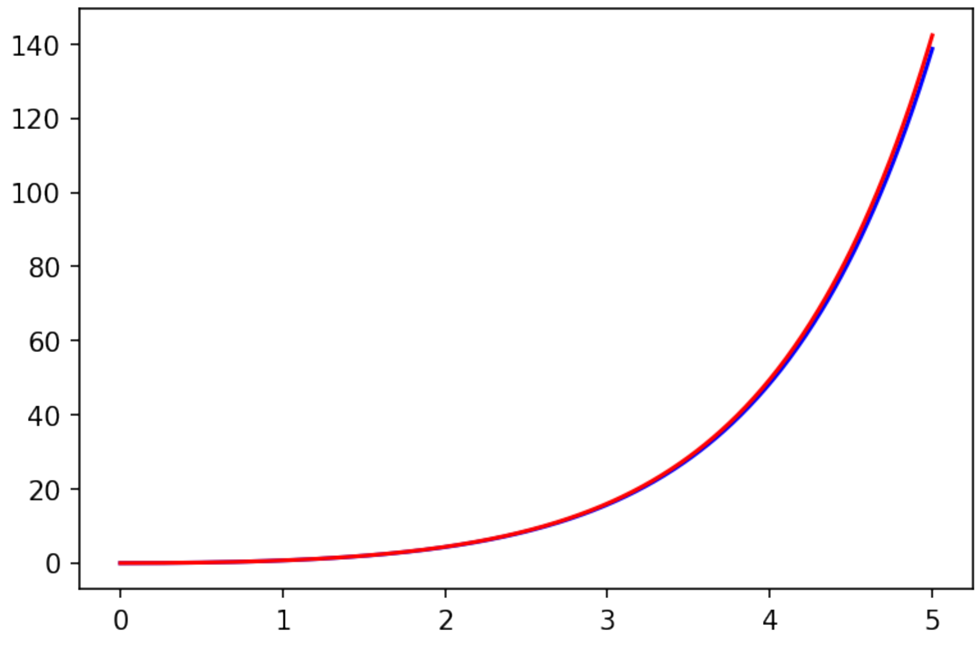
[18]:
```
def func(x,y):                          # function returns dy/dx for the current values␣
 ↪of x and y
    return x+y                          # The expression for dy/dx

h = 0.01
N_steps=500
x = 0                                   # Sets the initial value of x
y = 0                                   # Sets the initial value of y
x_list=[x]                              # Creates a list and sets the first value to be␣
 ↪the initial value of x
y_list=[y]                              # Creates a list and sets the first value to be␣
 ↪the initial value of y
true = [y]
for i in range(1,N_steps+1):
    y = y + h*func(x,y)
    x = x + h
    x_list.append(x)                    # Adds (appends) the new value of x into the list
    y_list.append(y)                    # Adds (appends) the new value of y into the list
    true.append(math.exp(x)-x-1)
```

```
[19]: import matplotlib.pyplot as plt          # Imports the matplotlib library so we␣
       ↪can draw graphs
      plt.rcParams['figure.dpi'] = 150        # Sets the resolution (in dpi) of the␣
       ↪figure

      plt.plot(x_list, y_list, 'b')           # Sets up a plot of one list against␣
       ↪another, blue line
      plt.plot(x_list, true, 'r')             # Sets up a plot of one list against␣
       ↪another, red line

      plt.show()                              # Shows the plot
```



This plot shows the Euler solution in blue and the 'real' solution in red. You can see that for small values of x the y values are quite close, however as 'x' increases so does the difference between the solutions.

1.4 Comments in Coding

You will see that as programs get more complex it is important to add comments to our code to explain what we have done. The point of these commments is so that when you look at your complex code at some later date you will understand what you have done! This is, in fact, a really important element of coding, and for complex codes it can be virtually impossible to understand what the code is doing without comments.

Look at the code in the above few cells as an example.

For this, reason all your codes submitted (i.e. assignments) from this point on should be commented. The comments need to be sufficient to enable the reader to understand the code, but don't over do it. For all the assignments from now, 2 marks will be awarded for good comments!!

```
[ ]:
```