# MATH2070: LAB 2: Beginning Python

## 1 Introduction

In the last lab, we learned how to start up Python and a little bit about using Python. This lab is intended to introduce you to the basics of the Python programming language. We will not be concerned with all of the details of this language, we will focus on those aspects of the language that make it ideal for numerical calculations. But Python *is* a programming language and it is important to learn the basics of good programming so that you will be able to use Python for research and applications. This lab will take two sessions.

## 2 Python files

The best way to use Python is to use its scripting (programming) facility. With sequences of Python commands contained in files, it is easy to see what calculations were done to produce a certain result, and it is easy to show that the correct values were used in producing a graph. It is terribly embarrassing to produce a very nice plot that you show to your advisor only to discover later that you cannot reproduce it or anything like it for similar conditions or parameters. When the commands are in clear text files, with easily read, well-commented code, you have a very good idea of how a particular result was obtained. And you will be able to reproduce it and similar calculations as often as you please.

The Python comment character is a "pound" or "hash" sign (#). That is, lines starting with # are not read as Python commands and can contain any text. Similarly, any text on a line following a # can contain textual comments and not Python commands. It is important to include comments in .py files to explain what is being accomplished.

## 3 Variables, numpy, and scipy

Python uses variable names to represent data. A variable can represent some important value in a program, or it can represent some sort of dummy or temporary value. Important quantities should be given names longer than a few letters, and the names should indicate the meaning of the quantity. For example, if you were using Python to generate a matrix containing a table of squares of numbers, you might name the table `tableOfSquares`. (The convention I am using here is that the first part of the variable name should be a noun and it should be lower case. Modifying words follow with upper case letters separating the words. This rule comes from the officially recommended naming of Java variables.) Once you have used a variable name, it is bad practice to re-use it to mean something else.

To enable the level of scientific computing needed for the class we will make use of several open-source Python libraries. Many of these are installed by default via anaconda. Two of the most important are the packages NumPy and SciPy. Roughly, you can think of the NumPy package as containing the base data types and operations which we will need. Some examples include the ability to make vectors and matricies, matrix and vector addition, define commonly used function such as $sin(x)$. SciPy builds on top of NumPy and is used for actual numerical code such as methods for solving linear systems.

**Exercise 1**: Being by starting up an interactive Python session by typing python into the command line.

(a) Import the numpy library using the command

```
import numpy as np
```

(b) What value does the variable `np.pi` return?

(c) Print the variable `np.pi` to 4 decimal places using the command

```
print(f"{np.pi:.4f}")
```

(d) Do a quick google search for "python fstrings" and explain what `f"np.pi:.4f"` is doing.

(e) In your own words describe what the `float` function is doing.

(f) calculate and print the double precision machine epsilon value in Python using the below code snippet

```
eps = np.finfo(float).eps
print(f"machine eps = {eps}")
```

(g) Set the variable `a=1` and the variable `b=1+eps`. What is the difference in the way that Python displays these values? Can you tell from the form of the printed value that `a` and `b` are different?

(h) Set the variable `c=2` and the variable `d=2+eps`. Are the values of `c` and `d` different? Explain why or why not.

(i) Choose a value and set the variable `x` to that value.

(j) What is the square of `x`? Its cube? In Python `**` is used for taking a variable to a power. For example `2 ** 3 = 8`.

(k) Choose an angle $\theta$ and set the variable `theta` to its value (a number).

(l) What values does $np.\sin\theta$? $np.\cos\theta$ return? Angles can be measured in degrees or radians. Which of these has Python used?

(m) Python variables can also be given "character" or "string" values. A string is a sequence of letters, numbers, spaces, etc., surrounded by single quotes (`'`). In your own words, what is the difference between the following two expressions?

```
a1='np.sqrt(4)'
a2=np.sqrt(4)
```

# 4 Vectors and matrices

Thoughout this course we will need to make use of vectors and matrices over and over again. These vectors and matrices will be defined using numpy arrays. Below we provide some basic details about these.

The easiest way to define a numpy row vector is the `np.array()` command with the values listed in square brackets

```
rowVec1 = np.array([ 0, 1, 3, -6, np.pi ])
```

A column vector can be described similarly except there is another layer of square brackets for each individual column entry.

```
columnVector1 = np.array([[0],[1],[3],[6],[np.pi]])
```

Matrices can be written by simply adding more entries to the inner square brackets appearing in the columVector1 example above. For instance The matrix

$$\mathcal{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \tag{1}$$

can be generated with the expression

```
A = np.array([[1,2,3],[4,5,6],[7,8,9]])
```

Often we will need to generate a set of equally spaced values, which can be useful for plotting and other tasks. This can be done using the numpy function `np.linsapce()`. This function has the form:

`np.linspace( firstValue, lastValue, numberOfValues )`

For example we could generate 6 evenly spaced numbers in the interval `[10,20]` using the command:

```
evens = np.linspace ( 10, 20, 6 )
```

or fifty evenly-spaced points with

```
evens = np.linspace ( 10, 20, 50 )
```

As a general rule, use the colon notation when `firstValue`, `lastValue` and `increment` are integers, or when you would have to do mental arithmetic to get `increment`, and `linspace` otherwise.

**Exercise 2**:

(a) In a terminal type python to begin an interactive session. Import the numpy library using the command

```
import numpy as np
```

(b) Use the `np.linspace` function to create a numpy array called `meshPoints` containing exactly 1000 values with values evenly spaced between -1 and 1.

```
meshPoints = np.linspace(-1,1,1000)
```

(c) What expression will yield the value of the $95^{\text{th}}$ element of `meshPoints` (remember Python arrays start at index 0)? What is this value?

(d) Use the np.size function to again confirm the vector has length 1000 e.g.

```
np.size(meshPoints)
```

(e) Next, we will produce a plot of a sinusoid on the interval $[-1, 1]$. This can be done with the sequence of commands

```
import matplotlib.pyplot as plt
plt.plot(meshPoints,np.sin(2*np.pi*meshPoints))
plt.show()
```

Please save this plot along with your summary.

(f) In your own words describe what each of the above commands does.

(g) Create a file named `exer2.py`. The first lines of the file should be the following:

```
# Lab 2, exercise 2
# A sample script file.
# Your name and the date
```

Follow the header comments with the commands containing exactly the commands you used in the earlier parts of this exercise. Load the provided conda environment as done in Lab01 and test your .py script by typing into the terminal

```
python exer2.py
```

Remember your terminal needs to be in the directory containing exer2.py.

# 5 Vector and matrix operations

The Python packages numpy and scipy provide a large assembly of tools for matrix and vector manipulation. We will investigate a few of these by trying them out.

**Exercise 3**: In an interactive Python session define the following vectors and matrices after:

```
import numpy as np
rowVec1 = np.array([-1,-4,-9])
colVec1 = np.array([[2],[8],[9]])
mat1 = np.array([[1,3,5],[7,9,0],[2,4,6]])
```

(a) You can multiply vectors by constants. Compute

```
colVec2 = (np.pi/4) * colVec1
```

(b) The cosine function can be applied to a vector to yield a vector of cosines. Compute

```
colVec2 = np.cos ( colVec2 )
```

Print these new values using the command

```
print(colVec2)
```

Note that the values of `colVec2` have been overwritten. Are these the values you expect?

(c) You can add vectors. Compute

```
colVec3 = colVec1 + colVec2
```

and print these values.

(d) The Euclidean norm of a matrix or a vector is available using `np.linalg.norm`. Compute

```
np.linalg.norm(colVec3)
```

(e) You can do row-column matrix multiplication using `np.dot`. Compute

```
colvec4 = np.dot(mat1, colVec1)
```

print these values.

(f) We can take the transpose of a matrix using the command `np.transpose`

```
mat1Transpose = np.transpose(mat1)
rowVec2 = np.transpose(colVec3)
```

print these values.

(g) Similarly we can take the transpose using the matrix and columns method rather than calling a numpy function

```
mat1Transpose = mat1.T
rowVec2 = colVec3.T
```

(h) Matrix operations such as determinant and trace are available, too.

```
determinant = np.linalg.det(mat1)
tr = np.trace( mat1 )
```

Print these values.

(i) You can pick certain elements out of a vector, too. Use the following command to find the smallest element in a vector `rowVec1`.

```
np.min(rowVec1)
```

or by using the numpy array method

```
rowVec1.min()
```

(j) For matrices the `np.min` and `np.max` functions can be set to work along one dimension at a time. Therefore they will produce vectors rather than a single scalar if desired.

```
np.max(mat1, axis=0)
np.max(mat1, axis=1)
```

In your own words describe what the axis option does.

(k) You can compose different function operations. For example, use the following expression to compute the max norm of a vector.

```
np.max(abs(rowVec1))
```

# 6 Flow control

It is critical to be able to ask questions and to perform repetitive calculations in .py files. These topics are examples of "flow control" constructs in programming languages. Python provides two basic looping (repetition) constructs: `for` and `while`, and the `if` construct for asking questions. These statements each precede an indented block of Python statements to be repeated as necessary. Python loops are very flexible; generally in this course we will be looping over integers and making use of the `range` function. In other languages there might be a statement which signifies the end of the loop/if statement e.g. the `end` statement in Matlab. However, in Python white space and indentation have an actual meaning. In the case of loops and if statements whatever is immediately indented afterwards is considered a part of the loop/if statement. The end of indented statements signifies the end of the loop.

The `range command` produces a sequence of numbers starting from 0 by default and increments by 1 by default. These defaults can modified. The range function is inclusive of the first value, but stops before the specified end value.

| | Syntax | Example |
|---|---|---|
| `for` loop | `for` *control-variable in sequence:*<br>*Python statement ...*<br>...<br>`end` | `nFactorial=1;`<br>`for i in range(1,n+1):`<br>`  nFactorial=nFactorial*i` |
| `while` loop | *Python statement initializing a control variable*<br>`while` *logical condition involving the control variable:*<br>*Python statement ...*<br>...<br>*Python statement changing the control variable* | `nFactorial=1`<br>`i=1;  # initialize i`<br>`while i <= n:`<br>`  nFactorial=nFactorial*i`<br>`  i=i+1` |
| a simple `if` | `if` *logical condition:*<br>*Python statement ...*<br>... | `if x ~= 0:   # ~ means "not"`<br>`  y = 1/x` |
| a compound `if` | `if` *logical condition:*<br>*Python statement ...*<br>...<br>`elif:` *logical condition*<br>...<br>`else:`<br>... | `if x ~= 0:`<br>`  y=1/x`<br>`elif np.sign(x) > 0:`<br>`  y = 1`<br>`else:`<br>`  y = -1` |

**Exercise 4**: The trapezoid rule for the approximate value of the integral of $e^x$ on the interval $[0, 1]$ can be written as

$$\int_0^1 e^x dx \approx \frac{h}{2}e^{x_0} + h\sum_{k=1}^{N-1} e^{x_k} + \frac{h}{2}e^{x_N}$$

where $h = 1/N$ and $x_k = 0, h, 2h, \ldots, 1$.

The following Python code computes the trapezoid rule for the case that $N = 40$.

```
import numpy as np

# Use the trapezoid rule to approximate the integral from 0 to 1
# of exp(x), using N intervals
# Your name and the date

N = 40
h = 1 / N
x = -h  # look at this trick
approxIntegral = 0
for k in range(0, N+1):
  # compute current x value
  x = x + h

  # add the terms up
  if k == 0 or k == N:
    approxIntegral = approxIntegral + (h / 2) * np.exp(x)
    # ends of interval
  else:
    approxIntegral = approxIntegral + h * np.exp(x)
    # middle of interval

print(f"The approximate integral is {approxIntegral}")
```

(a) Cut and paste this code into a Python file exer4.py then run the code and report the outputted values. Is the final value for `approxIntegral` nearly equal to $e^1 - e^0$?

(b) What is the complete sequence of all values taken on by the variable `x`?

(c) How many times is the following statement executed?

```
approxIntegral=approxIntegral + (h / 2) *np.exp(x)   # ends of interval
```

(d) How many times is the following statement executed?

```
approxIntegral=approxIntegral + h * np.exp(x)       # middle of interval
```

# 7 Scripts, functions and graphics

If you have to type everything at the command line, you will not get very far. You need some sort of scripting capability to save the trouble of typing, to make editing easier, and to provide a record of what you have done. You also need the capability of making functions or your scripts will become too long to understand. In this section we will consider first a pure script file and later introduce functions. We will be using graphics in the script file, so you can pick up how graphics can be used in our work.

The Fourier series for the function $y = x^3$ on the interval $-1 \leq x \leq 1$ is

$$y = 2\sum_{k=1}^{\infty}(-1)^{k+1}\left(\frac{\pi^2}{k} - \frac{6}{k^3}\right)\sin kx. \tag{2}$$

We are going to look at how this series converges.

**Exercise 5**: Copy and paste the following text into a file named `exer5.py` and then answer the questions about the code.

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

# compute NTERMS terms of the Fourier Series for y=x^3
# plot the result using NPOINTS points from -1 to 1.
# Your name and the date

NTERMS = 20
NPOINTS = 1000
x = np.linspace(-1, 1, NPOINTS)
y = np.zeros(np.size(x))
for k in range(1, NTERMS + 1):
  term = 2 * (-1)**(k + 1) * (np.pi ** 2 / k - 6 / k ** 3) * np.sin(k * x)
  y = y + term

plt.plot(x, y, "b")  # 'b' is for blue line
plt.plot(x, x ** 3, "g")  # 'g' is for a green line
plt.show()
```

It is always good programming practice to define constants symbolically at the beginning of a program and then to use the symbols within the program. Sometimes these special constants are called "magic numbers." By convention, symbolic constants are named using all upper case.

(a) Add your name and the date to the comments at the beginning of the file.

(b) How is the Python variable `x` related to the dummy variable $x$ in Equation (2)? (Please use no more than one sentence for the answer.)

(c) How is the Python statement that begins `y=y...` inside the loop related to the summation in Equation (2)? (Please use no more than one sentence for the answer.)

(d) In your own words, what does the line

```
y = np.zeros(np.size(x))
```

do? **Hint**: You can google np.zeros and np.size to find the documentation for these functions.

(e) Execute the script by typing `python exer5.py` at the command line. You should see a plot of two lines, one representing a partial sum of the series and the green line a plot of $x^3$, the limit of the partial sums. You do not have to send me this plot.

In the following exercise you are going to modify the calculation so that it continues to add terms so long as the largest component of the next term remains larger in absolute value than, say, 0.05. Since the series is of alternating sign, this quantity is a legitimate estimate of the difference between the partial sum and the limit. The `while` statement is designed for this case. It would be used in the following way, replacing the `for` statement.

```
TOLERANCE= 0.05;     # the chosen tolerance value
<<some lines of code from above>>
k=0
term = TOLERANCE + 1  # bigger than TOLERANCE
while (np.max(abs(term)) > TOLERANCE):
  k = k + 1
  <<some lines of code from above>>

print(f"Number of iterations = {k}")
<<some lines of code to plot results>>
```

**Exercise 6**:

(a) Copy the file `exer5.py` to a new file called `exer6.py` and change the comments at the beginning of the file to reflect the objective of this exercise.

(b) Modify `exer6.py` by a replacing the `for` loop with a `while` loop as outlined above.

(c) What is the purpose of the statement

```
term= TOLERANCE + 1
```

(d) What is the purpose of the statement

```
k = k + 1
```

(e) Try the script to see how it works. How many iterations are required? Does it generate a plot similar to the one from the previous exercise using a `for` loop?

**Hint:** If you try this script and it does not quit (it stays "busy") you can interrupt the calculation by holding the Control key ("CTRL") and pressing "C".

If you find that your code does not quit, you have a bug. For some reason, the value of the variable `term` is not getting small. Look at your code carefully to find out why. If you cannot see it, use the debugger to watch execution and see what is happening to the value of `term`.

The next task is to make a function out of `exer6.py` in such a way that it takes as argument the desired tolerance and returns the number of iterations required to meet that tolerance.

**Exercise 7**:

(a) Copy the file `exer6.py` to a new file called `exer7.py`. We will turn it into a function by using the line below

```
def exer7 (tolerance):
# comments
# Your name and the date
```

all code within the function must be indented.

(b) Replace the upper-case `TOLERANCE` with a lower-case `tolerance` because it no longer is a constant, and throw away the line giving it a value.

(c) Add comments just after the signature and usage lines to indicate what the function does.

(d) Delete the lines that create the plot and print so that the function does its work silently.

(e) Add the statement `return k` at the end of the function. This tells Python that when the function is called it should return the value k.

(f) Start up a python interaction session and type the command

```
from exer7 import exer7
```

this line is saying from the file exer7 load the function exer7 into my Python environment. Next, invoke this function from the command line by choosing a tolerance and calling the function. Using the command

```
numItsRequired=exer7(0.05)
```

(g) Print this value. How many iterations are required for a tolerance of 0.05? This value should agree with the value you saw in Exercise 6.

(h) To observe convergence, how many iterations are required for tolerances of 0.1, 0.05, 0.025, and 0.0125?

Unlike ordinary mathematical notation, Python allows a function to return two or more values instead of a single value. The syntax to accomplish this trick is simple we just add it to the return statement with a comma, e.g.,

```
return y, z
```

would now return the values `y and z`. For a function named "`funct`" that returns two variables depending on a single variable as input, the function call would look like:

```
y,z = funct( x )
```

If you have the same function but wish only the first output variable, `y`, you would write

```
y,_ = funct( x )
```

and if you wish only the second output variable, `z`, you would write

```
_,z = funct( x )
```

**Exercise 8**:

(a) Copy the file `exer7.py` to a new file called `exer8.py`. Modify the function so that it returns first the converged value of the sum (a vector) and, second, the number of iterations required. Be sure to change the comments to include a description of all input and output parameters.

(b) What form of the command line would return **only** the (vector) value of the sum to a tolerance of 0.03. What is the norm of this vector to 14 digits of accuracy. **Hint**: The command `print(f'{a:.10f}')` would print out the variable `a` to 10 digits of accuracy.

(c) What form of the command line would return the (vector) value of the sum and the number of iterations required to achieve a tolerance of 0.02? How many iterations were taken, and what is the norm of the (vector) value of the sum, to 14 digits of accuracy?

**Exercise 9**: We will often find it useful in this course to have function names as variables. For example, the sine function in `exer8` could be replaced with an arbitrary function. While the series might not converge for some choices of functions, it would converge for others

(a) Copy the file `exer8.py` to a new file called `exer9.py`. Modify the function so that it accepts a second parameter: `exer9(tolerance, func)` and replace the `np.sin` function inside the sum with `func`. Do not forget to modify the comments in the file to reflect this change.

(b) Test that `exer9` and `exer8` give equivalent results when `func` is really `sin` with the following code in an interactive python terminal:

```
import numpy as np
from exer8 import exer8
from exer9 import exer9
y8,k8=exer8(0.02);
y9,k9=exer9(0.02, np.sin);
#the following difference should be small
np.linalg.norm(y8-y9)
```

(c) Use `exer9` to compute the (vector) sum of the series for `exer9(.02, np.cos)` and plot the result. Please include this plot with your summary.

You have now completed all the required work for this lab. Be sure to send it to me in .tar.gz or .zip file. It should include your summary file, each of the .py files `exer2.py`, `exer5.py`, `exer6.py`, `exer7.py`, `exer8.py` and the plot files you created. Additionally, remember that all of .py files must pass flake8 and python Black for full credit.