Exercise 1
Python Coding, Fourier Expansions

Computer Modelling 2017-18

1 Aims

In this exercise, you will re-familiarise yourself with the Python programming language. You will work through a series of short Python programs and fix their errors. You will then expand a program that plots a simple cosine function to allow a Fourier expansion of a slightly more complex periodic shape.

2 Tasks

2.1 Python Debugging

Extract the short Python codes in `python_errors.tar.gz`, interpret the runtime error messages for each source code file, and fix each source code to obtain a correctly working program.

2.2 A harmonic plotter Program

Write a Python program `harmonic_plotter.py` that

- Takes a file name from the command line and opens the specified file for writing;
- Plots the following function using `matplotlib`, but also writes its values over one period to the output file:

$$f(x) = \sin x + \frac{1}{3}\sin 3x + \frac{1}{5}\sin 5x + \frac{1}{7}\sin 7x \tag{1}$$

- Closes the output file.

Do not type in the entire $\sin(nx)$ series, but compute the function's value using a `for` loop in a separate method. Note that the series expansion above can be written as

$$f(x) = \sum_{i=1}^{N} \frac{1}{2i-1} \sin\left((2i-1)x\right) \tag{2}$$

How does the output change if instead of $N = 4$ you include the first $N = 20$ or $N = 200$ terms of the series?

These are the lowest four terms in the Fourier series for a square wave. You will cover Fourier series later this year in Fourier Analysis and Statistics. For the purposes of this checkpoint it is sufficient to see this as a first hint that an arbitrary function can be described by an appropriate combination of sines and cosines.

3 Detailed Instructions

You should refresh your memory of the Scientific Programming module of last years 'Programming and Data Analysis' course and associated documents. The Scientific Programming course booklet is available via the Computer Modelling LEARN page and will be a useful Python reference document for this course. You should pay particular attention to

- Structure of a Python program
- Declaration and initialization of variables
- Reading and writing files
- Loops and conditional constructs
- Functions and methods
- Objects

In the first part of this checkpoint, you are given a number of erroneous Python source code files. As a warm-up exercise for this course, correct the error(s) in each source code file. In the second part of this checkpoint, you will write numerical data into text files, and display them as graphs using the popular `matplotlib` module for Python.

3.1 Housekeeping

You should decide with your other group members on the way you are going to share files so that you can work on the exercises. The easiest option is probably to use a service such as Dropbox (http://www.dropbox.com) and share a folder between all group members – if you do not want to use Dropbox other mechanisms are suggested in the course notes.

Make a directory for this course, we would suggest

```
[user@cplab001 ~]$ mkdir CompMod
[user@cplab001 ~]$ cd CompMod
```

We recommend that you create separate subdirectories for each of the exercise in the course. This should make it easy to find the files you need when you are getting your work marked – it is also a good habit to get into. You may also want to create further subdirectories to hold the most recent working copy of your code so that you do not break it with a modification or by deleting a file by accident.

3.2 The `python_errors` Package

Download the compressed archive of erroneous Python source codes from LEARN: https://www.learn.ed.ac.uk/webapps/portal/frameset.jsp.

Unpack the archive:

```
[user@cplab001 ~]$ gunzip python_errors.tar.gz
[user@cplab001 ~]$ tar xvf python_errors.tar
```

Read the source code files with your preferred text editor. For example, use the emacs text editor with the command

```
[user@cplab001 ~]$ emacs error1.py &
```

(Recall that the ampersand '`&`' instructs Linux to run the program and then return to the command prompt.) Attempt to run each code using, e.g.

```
[user@cplab001 ~]$ python3 error1.py
```

Most example source codes provided will not compile; some will compile, but wont run. All errors in these codes are commonly encountered during programming, so you should pay attention to what the error messages say and how to fix the code in each instance.

3.3 The `cosine_plotter` Code

Download the Python source code file `cosine_plotter.py` from LEARN. `cosine_plotter.py` outputs the values of $\cos x$ one period. The cosine is evaluated in a separate function, `my_function()`, that is accessed from the main part of the code. Run it by typing

```
[user@cplab001 ~]$ python3 cosine_plotter.py
```

Make sure you understand how the output is generated. While the code produces a plot using `matplotlib`, it also writes to an output file. You can therefore plot the data produced by this program (in `cosine.dat`), for instance by using `xmGrace` with the command:

```
[user@cplab001 ~]$ xmgrace cosine.dat &
```

3.4 Reading String input from the command line

It is often very useful to be able to supply data and options to our programs via the Linux command line when we run the program rather than reading them in from a file

or Console (or even hardcoding them into the program). Fortunately, Python provides various extremely simple ways to do this.

The first option involves interaction with the user at runtime. The following Python code snippet

```
user_name = input("Your␣name:␣")
print("Your␣name␣is␣" + user_name)
```

will open a user prompt, wait for user input (finished with the Return key), and use that input to generate some output (in this case). If you save the above snippet in a file called `print_name.py`, the full interaction on Terminal will look something like this:

```
[user@cplab001 ~]$ python3 print_name.py
Your name: student1
Your name is student1
[user@cplab001 ~]$
```

The second option involves the passing of arguments at the time the Python script is executed. These arguments are accessible through the list `argv[]` in the Python `sys` module. The following code snippet

```
import sys
user_name = sys.argv[1]
user_town = sys.argv[2]
print("Your␣name␣is␣" + user_name + "and␣you␣are␣from␣" +
   user_town)
```

will result in the following behaviour:

```
[user@cplab001 ~]$ python3 print_name.py student1 edinburgh
Your name is student1 and you are from edinburgh
[user@cplab001 ~]$
```

For some purposes, for instance in an interactive game against the computer, the second option will not work at all. However, if both options *do* work for your purposes, the second is usually preferred. The user input can be tested faster and easier, and repeated execution of the script is faster because no interactive behaviour from the user is required.

4 Formative Feedback

1. Python debugging
   - Correct source codes for each erroneous source file
   - Explain the source of error in each case, and how to interpret the error message(s)
2. `HarmonicPlotter` code

- Program functions correctly
- Reads filename from command line
- Data plotted using `matplotlib`
- Clear and concise code, logical and well documented