Practical 4
2-Dimensional Arrays, Functions and Plotting

Learning Objectives

1. Understand and use multi-dimensional arrays in Python using the Numpy library
2. Use sub-modules available the Scipy library
3. Define and use simple functions
4. Apply multi-dimensional arrays to multi-dimensional science data
5. Use matplotlib to plot multi-dimensional data

Overview

In this practical you will be exploring the use 2-D arrays. We will use the ndimage sub-module of the Scipy package. Arrays are very useful for storing the values of variables representing areas. In heat.py we will look at a model for heat diffusion. In the final two exercises, we will work with functions, firstly on strings and then on the heat diffusion calculation.

Tasks

1. Exploring 2-D Arrays

Type in the following code, zeros.py, for creating and resizing an array:

```
#
# zeros.py - creating and resizing an array
#
import numpy as np

print('\nZERO ARRAY\n')
zeroarray = np.zeros((3,3,3))

print('Zero array size: ', np.size(zeroarray))
print('Zero array shape: ', np.shape(zeroarray), '\n')
print(zeroarray)

zeroarray.resize((1,27))
print('\nZero array size: ', np.size(zeroarray))
print('Zero array shape: ', np.shape(zeroarray), '\n')
print(zeroarray)

zeroarray.resize((3,9))
print('\nZero array size: ', np.size(zeroarray))
print('Zero array shape: ', np.shape(zeroarray), '\n')
print(zeroarray)
```

Modify the code to set element [0,0,2] to 1, element [1,1,1] to 2 and [2,2,0] to 3. Run the code and look for where these values sit in each resized array.

2. Ndimage in Scipy

Type in the following code, prettyface.py – it's from the lecture…

```
#
# prettyface.py
#
import matplotlib.pyplot as plt
from scipy import ndimage
from scipy import misc

face = misc.face(gray=True)
plt.imshow(face)
plt.imshow(face, cmap=plt.cm.gray)
plt.show()
```

Go through the lecture slides and add in the code for shifting, rotating, cropping and pixelating.

Look at the documentation for colour maps and try a few of them with your code…
http://matplotlib.org/examples/color/colormaps_reference.html

3. Specifications and Pseudocode

The lecture slides included a description and pseudocode specification of a program for collecting competition scores.

Translate and test the first version of the program (competition_v1.py) and check how it handles incorrect data, and the impact of the dodgy data on the results (e.g. score of -100).

Make a copy of the code as competition_v2.py and adjust it to match the second version of the pseudocode from the slides. Test it again with bad input to see how it is handled.

Finally make another copy and modify it to match version three from the lecture slides. Try the same tests to check it is working.

4. Conversions

In this task we will create some functions to convert values between different units. To start, we will convert between Celsius, Fahrenheit and Kelvin. Below is a skeleton of how to start your code.

conversions.py

```
#
# conversions.py - module with functions to convert between units
#
#       fahr2cel : Convert from Fahrenheit to Celsius.
#
def fahr2cel(fahr):
    """Convert from Fahrenheit to Celsius.
    Argument:
    fahr - the temperature in Fahrenheit
    """
    celsius = (fahr - 32) * (5/9)
    return celsius
```

Note that we are using docstrings to document our functions. See the related PEP for more about docstrings - https://www.python.org/dev/peps/pep-0257/

Write some test code to test out your functions. You could start with something like the code below, or work with user input.

testconversions.py

```
#
# testConversions.py - tests the functions in conversions.py
#
from conversions import *
print("\nTESTING CONVERSIONS\n")
testF = 100
testC = fahr2cel(testF)
print("Fahrenheit is ", testF, " Celsius is ", testC)
print()
```

Extend conversions.py to include all six conversion functions, along with others you might find useful. Extend your test program to test the other conversions.

To see the docstring for a function, you access the __doc__ attribute. So to print the docstring for fahr2cel, use:

print(fahr2cel.__doc__)

5. Conversion Machine

Now we can write a program, converter.py, to convert between our temperature formats. Your program should:

1. print starting message
2. provide a menu of conversions to choose between
3. take the user input
4. do the conversion, or provide an error message
5. ask if they want to do another conversion
6. loop back to (2) if they want to continue
7. print closing message

This is very similar to the Bucket List Builder, so refer to Practical 2 to see that code.

6. Conversion Machine (2)

Create a different version of the conversion machine, converter2.py, that will ask for the conversion type, then will convert a list of numbers into the target unit. The loop should exit when the user enters an empty value (just presses return).

```
1. print starting message
2. provide a menu of conversions to choose between
3. take the user input
4. while input isn't an empty string
5.     print converted value
6.     ask for next value
7.     loop back to (4) if they want to continue
8. print closing message
```

7. Conversion Machine (3)

Think about the input you are giving to converter2.py. Could you automate that input?

We can redirect input in the same way that we redirected the output of history in the practical test (history > hist.txt). Create a file temps.txt with sample input for your converter2.py program, then try:

```
python3 converter2.py < temps.txt
```

To capture the results, you can also redirect the output:

```
python3 converter2.py < temps.txt >tempsout.txt
```

Make a larger input file to see how easy it is to process data using standard in (keyboard input) and standard out (screen output).

7. Testing your Module

In the lecture (slide 71), we saw how we can use the __main__ attribute/variable to check if our python code has been run directly (e.g. `python3 conversions.py`) or indirectly (e.g. `from conversions import *, temp = fahr2cel(100)`). Using this, we can create test code inside our module.

Modify conversions.py to include test code by implementing a main() function and putting the required if statement at the end of the module.

Test your changes by running the conversions.py from the command line.

```
python3 conversions.py
```

The additional code for conversions.py is:

```
def main():
    print('\nTesting textfun.py ')
    # put your testing code in here
    print('Testing complete')

if __name__ == '__main__':
    main()
```

Submission

Create a README file for Prac 4. Include the names and descriptions of all of your Python programs from today.

All of your work for this week's practical should be submitted via Blackboard using the link in the Week 4 unit materials. This should be done as a single "zipped" file.

Reflection

1. **Knowledge**: What are three benefits of using functions?
2. **Comprehension**: What is the purpose of the colour map (`cmap=`) in Task 2?
3. **Application**: How would you change the plot of the critter to be shades of purple and in reverse? (like a negative of a photo)
4. **Analysis**: Task 7 uses the Python variable "`__name__`" to support testing. What does name equal when the module code is run directly (as "`python conversions.py`"), and what is its value when the module is run as "`python converter2.py`"?
5. **Synthesis**: What happens when we resize an array to be smaller than the original? What happens when we make it larger? Does the data in the array change?
6. **Evaluation**: Compare the three versions of the competition code from task 3. How has the code improved from 1) the user perspective, and 2) the programmer perspective.

Challenge

For those who want to explore a bit more of the topics covered in this practical. Note that the challenges are not examinable.

1. Create a program to convert an inputted string to Pig Latin
2. Find a repetitive song and use functions like print_lyrics() to print out the complete song.