COMP2002 - Artificial Intelligence

Week 1 - Python Exercises - Model Answer

Introduction

This set of model answers is intended to show you possible ways of solving the Python exercises in week 1 – there are others. Please attempt the exercises before looking at the answers.

Activities

Your task is to go through the following tasks. Please note, you are expected to complete some work on this outside of the timetabled sessions.

Exercise 1 - Bubble Sort

This question requires you to implement a Bubble Sort algorithm. This is a simple sorting algorithm that is known for its inefficiency, and the version I've included here is bubble sort in it its most basic form. You can improve its efficiency (a little) but if you want to sort a large list I'd recommend something like Quicksort or merge sort.

```
def bubble(L):
  N = len(L)  # Number of elements in L.
  for i in range(N):
    for j in range(N):
      if L[i] < L[j]:
         L[i], L[j] = L[j], L[i]
```

The function I've implemented begins by extracting the number of elements in the array using the len function. The bulk of the work is done in a double nested for loop, with each loop iterating over every element in the list. For each pair, if the pairs are in the wrong order their order is reversed. In some languages this means introducing a temporary variable so that you can avoid overwriting the values in the elements you're swapping. Python allows us to do this in a single line without introducing extraneous variables, as you can see.

The function I've implemented is an **in-place** sort. This takes advantage of the fact that most Python variables are **mutable** – which means they can be changed in place, and means we don't need to return the list. Instead, we pass the list to the function, and having run the sort can refer to the original variable, as follows:

```
L = [1, 5, 3, 2, 4, 6]
bubble(L)
print("Sorted list:", L)
```

Exercise 2 - Fibonacci Sequence

The task here is to implement the Fibonacci sequence, in which each element in the sequence is the sum of the previous two elements. Therefore the first 10 Fibonacci numbers are:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Here is a function that computes an arbitrary number of Fibonacci numbers and returns them in a list:

```
def fibonacci(N):
  sequence = [0, 1]

  for n in range(N-2):
    sequence.append(sequence[-1] + sequence[-2])

   return sequence
```

The function begins by initializing a list containing the first two values – 0 and 1. We then have a for loop; the sequence needs to be N elements long, and since we've done the first two we will loop for $N-2$ iterations. At each iteration we add the sum of the previous two elements, using the append method, which adds the passed as an argument to the list, which I've called sequence. I'm making use of a Python indexing feature, wherein the last element in the list has index -1, the penultimate element is -2, and so on. Finally, the function returns the sequence.

Here is some code that calls the function and prints the result to the screen:

```
seq = fibonacci(10)
print(seq)
```

Exercise 3 - Numpy and Matplotlib

During the module you'll be producing various plots to show what you've been doing, as well as using the mathematical tools provided by Numpy, so it's good to get used to using them from the off. This exercise requires you to produce the graph of

$$y = sin(x)$$

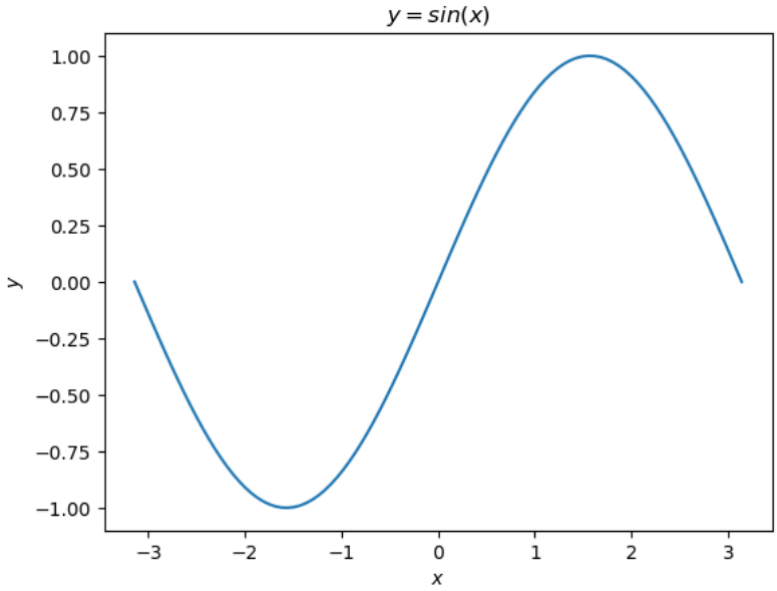
for a range of values of x between $-\pi$ and π. The following code does this:

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

x = np.linspace(-np.pi, np.pi, 100)    # Sample 100 points between -pi and pi.
y = np.sin(x)                          # Compute sin of the 100 points.

plt.figure()
plt.plot(x, y)
plt.title("$y=sin(x)$")
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.savefig("sin.png", bbox_inches="tight")
plt.show()
```

The first step is to import the two packages we'll need – Numpy, and Matplotlib's Pyplot. Then, generate the data we'll be using by sampling 100 points between $-\pi$ and π using the `np.linspace` function. This gives us a Numpy array of evenly-spaced values between $-\pi$ and π. Then, use `np.sin` to compute the sine of the values in x – passing in a single value computes a single sine value, and passing an array gives us an array of results, so we can do this in one go.

Having generated the data it's time to produce the plot. This is a case of using `plt.plot` to plot the x

and y values against each other. Then we tidy up the plot with a title, x-axis and y-axis labels, save it to a file called sin.png, and display it on the screen with the plt.show command. Your plot should look like this:



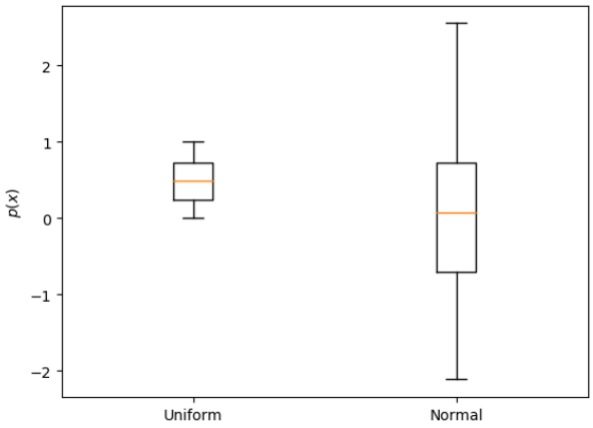
Exercise 4 - More Numpy and Matplotlib

The final exercise this week demonstrates how to generate random numbers within Numpy, and a useful method for plotting distributions using Matplotlib's boxplots. First, here is the code:

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

uniform = np.random.rand(100)    # Generate 100 uniform random points.
normal = np.random.randn(100)    # Generate 100 normal random points.

plt.figure()
plt.boxplot([uniform, normal])
plt.xticks([1, 2], ["Uniform", "Normal"])
plt.ylabel("$p(x)$")
plt.savefig("boxplot.png", bbox_inches="tight")
plt.show()
```

We again begin by importing Numpy and Matplotlib. Then we use the `np.random.rand` function to generate 100 uniform random values – these will all lie between 0 and 1 – and 100 normal values `np.random.randn`. The normal values will be centred around 0, and will follow the bell curve distribution. The majority of values (in the region of 95%) will lie between -2 and 2.

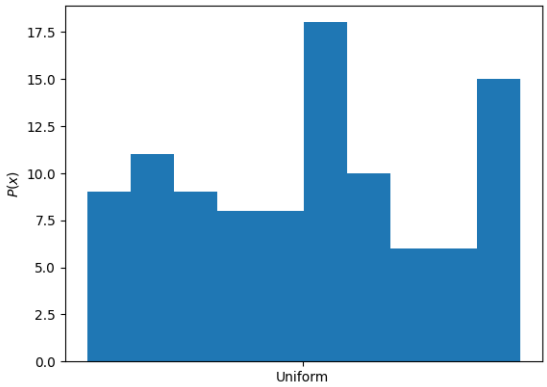
Then it is a case of producing another plot. This time we use the `plt.boxplot` command, which takes two Numpy arrays – the uniform and normal random values, comma separated. The spread of the random values is shown in the y axis. Finally, add a legend, label the axes, save and show the plot. Here it is:

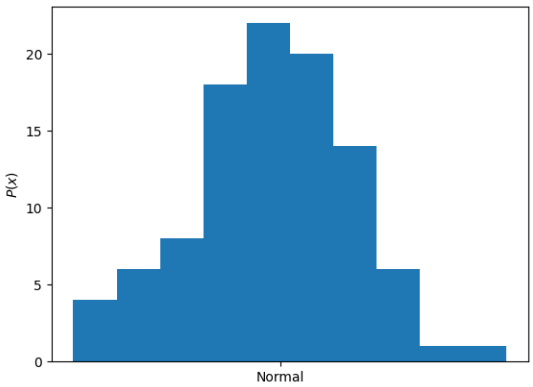


Extension task

To produce histograms we need the `plt.hist` function

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

uniform = np.random.rand(100)  # Generate 100 uniform random points
normal = np.random.randn(100)  # Generate 100 normal random points


plt.figure()
plt.hist([uniform])
plt.xticks([0.5],["Uniform"])
plt.ylabel("$P(x)$")
plt.savefig("histogramuniform.png",bbox_inches="tight")
plt.show()
plt.figure()
plt.hist([normal])
plt.xticks([0],["Normal"])
plt.ylabel("$P(x)$")
plt.savefig("histogramnormal.png",bbox_inches="tight")
plt.show()
```




20
15
10
5
0
P(x)
Normal