YSPA 2018
Problem Set 1 (mostly Python)
Due by 11:59:59pm on Monday, July 2, 2018

Type up or scan your solutions (you can use a scanner app on a smartphone if you'd like) and send them to **yalesummerastro@gmail.com** before the due date. Write your solutions in complete sentences with explanation of how you got your answer. Be sure to include your source code (with comments) and the results.

You should have already done the basic python tutorial on Code Academy (**https://www.codecademy.com/learn/python**) before attempting this problem set... you'll need to know about variable types, lists, conditional statements (if/then/else statements), while loops, and functions, all of which is covered in this tutorial.

You need to have python installed on your computer to do this problem set. If you are using a mac or linux operating system, Python is likely already installed on your computer, but the libraries we need (visual, numpy) probably are not. There are lots of ways of getting the python language onto your computer, but an easy way to get both the visual python (vpython) and numpy packages onto your computer us to use the "classic" vpython 6 install, which is here: **http://vpython.org/index6.html**. If you are already python savvy, make sure that you have python 2.7 and vpython 6 installed for your distribution and IDE. Do not install vpython 7 or glowscript[1].

You should run through the vpython tutorial here:
http://www.vpython.org/contents/docs/VPython_Intro.pdf , and the full documentation is here:
http://www.vpython.org/contents/docs/index.html . Use the VIDLE application to write your code and run your code in the built-in python shell.

Problem 1. Leibnitz's Formula for π is: $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$

Write a function that estimates the value of pi by summing this series. The function should take as an argument the number of terms to include in the sum and return the value of the sum. Then write a program that uses your function to calculate this sum with 10 terms, 100 terms, and 1000 terms, and compare each of those sums with the value of math.pi/4.

Problem 2. A standard problem in intro programming is to devise an algorithm to sort a list of numbers into increasing order (Even President Obama has an opinion on this problem: **http://bit.ly/1ttLvr3**).

Suppose we need to sort this list into ascending order:
list = [5, 0, 9, 8, 7, 1, 4, 3, 2, 6]

Let's try the "Bubble" sort (that Obama doesn't like) as a warm up. Here's the basic algorithm, written as pseudocode:
1) initialize a flag to record that position swaps have been made (to start the loop), and initialize a loop count at 0;

[1] We don't use vpython 7 because it only runs in a browser, and so it's slower than vpython 6 (which works fine) on our computers.

2) check the swap flag to see if we've made it through the list with no swaps... if so, the list is in order and we can stop (break the loop);
3) else, check the first value and see if it is bigger than the value that follows it in the list, and if it is, swap those two values and set the flag to indicate a swap;
4) Step through all of the values in the list, checking for adjacent pairs that are out of order and swapping those that are (and setting the flag if any swaps are made);
5) Add one to the counts through the loop and go back to Step 2.

As a quick example of the algorithm, let's do the first loop by hand.
5 is more than 0, so swap them. new list > [0, 5, 9, 8, 7, 1, 4, 3, 2, 6]
5 is less than 9, so leave them alone!
9 is greater than 8, so swap them. new list > [0, 5, 8, 9, 7, 1, 4, 3, 2, 6]
9 is greater than 7, so swap them. new list > [0, 5, 8, 7, 9, 1, 4, 3, 2, 6]
9 is greater than 1, so swap them. new list > [0, 5, 8, 7, 1, 9, 4, 3, 2, 6]
9 is greater than 4, so swap them. new list > [0, 5, 8, 7, 1, 4, 9, 3, 2, 6]
9 is greater than 3, so swap them. new list > [0, 5, 8, 7, 1, 4, 3, 9, 2, 6]
9 is greater than 2, so swap them. new list > [0, 5, 8, 7, 1, 4, 3, 2, 9, 6]
9 is greater than 6, so swap them. new list > [0, 5, 8, 7, 1, 4, 3, 2, 6, 9]

(You can see that big values will quickly rise to the top of the list, but lots of swaps are required, and small values will take a long time to bubble up to the front.)

Write a program that implements this sort on a given list and outputs the result. Have the program also output the number of steps through the sorting loop required. Try your program on this list. How many steps through the loop are required to sort it? What's the best possible case and the worst possible case for using the algorithm (what sequence of numbers sorts the fastest, and what sequence sorts the slowest)?

An improvement on the bubble sort is the "comb" sort. It is very similar to the bubble sort, but instead of comparing adjacent values and swapping them if necessary, you compare two values separated by a gap. This can help get the small values to the front of the list more quickly. Each time through the list, you reduce the gap by a scaling factor, and you repeat until the gap is zero and you are just back to doing the bubble sort.

For example, let's start sorting the list above using the comb sort with a scaling factor of 1.3.
The list is 10 elements long, so start with a gap of 10/1.3 = 7:
5 is greater than 3, so swap those two;
0 is less than 2, so leave those two as is;
9 is greater than 6, so swap those two.

The first time through the sort, the new list is: [3, 0, 6, 8, 7, 1, 4, 5, 2, 9].
Now we reduce the gap by 1.3; 7/1.3 = 5:
3 is greater than 1, so swap those;
0 is less than 4, so do nothing!
6 is greater than 5, so swap those;
8 is greater than 2, so swap those;
7 is less than 9, so do nothing!

The second time through the sort, the new list is [1, 0, 5, 2, 7, 3, 4, 6, 8, 9].
... and so on.

Write a new program that implements the comb sort (start with pseudocode for this algorithm), and sort the above list. How many steps did it take compared to the bubble sort?

Optional Extra Challenge (We don't have grades, so there's no "extra credit", but this part of the problem is very challenging... expect to ask for help, and don't worry if you don't get it!): Generate a thousand lists of 10 integers in random order and compare the average sort time between bubble sort and comb sort for all of the lists.

Sorting is an important problem in computer science, and there are many different sorting algorithms that work faster or more efficiently in different situations. This Timo Bingmann visualization of different algorithms is worth checking out: **http://bit.ly/1g8XrEF**

Problem 3. Astronomers measure positions on the Celestial Sphere using angles, just like we measure positions on the surface of the earth using the angles of latitude and longitude. The units of the angles could be in degrees (360° in a circle), radians (2π radians in a circle), or hours of time (24 hours in a circle, such that 1 hour = 15°, a way of measuring Right Ascension angle or Hour Angle).

Furthermore, angles are sometimes written in decimal notation, such as: 23.4712°, but for historical reasons, astronomers also sometimes use "sexagesimal", or base-60, notation. For example: 23° 28′ 16″ (degrees, arcminutes, arcseconds).

These are the same angle, just in different bases. In decimal, the number is:

$23 + \frac{4}{10^1} + \frac{7}{10^2} + \frac{1}{10^3} + \frac{2}{10^4}$.
In sexagesimal, the number is $23 + \frac{28}{60^1} + \frac{16}{60^2}$.

(You can punch that into your calculator to check that it matches the decimal notation.)

This number could be more precise in base 60 by including places of $1/60^3$ (arcthirds) and so on, but astronomers usually switch back to decimal there and use tenths and hundredth of an arcsecond. We still keep time in base-60 notation... there are 60 minutes of time in an hour, and 60 seconds of time in a minute.

Write a python function that converts sexagesimal angles to decimal. Use it to convert these angles to decimal (also, consider how many significant figures you should report in your answer!).
a) 11° 54′
b) – 60° 31′ 10″ (be careful about the sign of the arcminutes and arcseconds when you convert!)
c) – 8° 45′ 15.94″

Write a python function that converts decimal angles to sexagesimal. Use it to covert these angles to degrees, arcminutes, and arcseconds. (Be careful about your significant figures! Maintain the same degree of precision when you change base.)
d) 60.04°
e) 89.99999°
f) – 23.43715°

Problem 4. The determinant of a 2x2 matrix can be written as:

$$\det\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

.

Write a function that will return the determinant of a 2x2 matrix, passed to the function as a 2x2 numpy array. Put some test data into your function to show that it works.

The determinant of a 3x3 matrix can be written as

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a\times\begin{vmatrix} e & f \\ h & i \end{vmatrix} - b\times\begin{vmatrix} d & f \\ g & i \end{vmatrix} + c\times\begin{vmatrix} d & e \\ g & h \end{vmatrix}$$

.

Write another function that finds the determinant of a 3x3 matrix using the above equation and calling your 2x2 determinant function three times.

Optional Extra Challenge: Write a function that returns the determinant of an NxN matrix by breaking the determinant down into the sum of the scaled determinants of N, (N-1) x (N-1) matrices, then calling itself again and again recursively until a 2x2 matrix remains, which can be solved using your 2x2 matrix determinant function.

Problem 5. When you went through the vpython 6 tutorial you made a visualization of a ball bouncing around inside a box. Each step through the loop of the program, time advances by a small time step, and the position advances by the velocity times the time step:

```
ball.pos = ball.pos + ball.velocity*deltat
```

Since there was no acceleration in the simulation, the ball continues in a straight line until it hits a wall, when either the x component (if it hits a side) or y-component (if it hits the top or bottom) reverses instantaneously, in an elastic collision.

You could also image a situation where there is a constant acceleration in a certain direction, so that the velocity changes with every time step, by defining the constant acceleration vector as ball.accel and including this line in the loop:

```
ball.velocity = ball.velocity + bell.accel*deltat
```

Using this principle, modify your ball-in-a-box simulation (including the 6 walls, the velocity vector, and the trail) so that gravity is constantly pulling it down towards the bottom. Try running the simulation with a variety of different start positions and velocities. What did you use for the constant acceleration? How does gravity change the motion of the ball?

Optional Extra Challenges: What if the acceleration were not constant, but suppose it depended on the distance of the ball from the very center of the box, and it was always accelerating towards the center of the box (like a mass on a spring?). How would this affect the ball's motion? (Try it, and use different acceleration constants and different starting positions and speeds.) Or what if there were a micro-black hole in the center of the box, which pulled the ball towards the center with an acceleration inversely proportional to the square of the distance?