UNIVERSITY OF SASKATCHEWAN

**Department of Computer Science**
176 Thorvaldson Building
110 Science Place, Saskatoon, SK, S7N 5C9, Canada
Telephine: (306) 966-4886, Facimile: (306) 966-4884



# Assignment 3

## Modules, Control Flow

**Date Due: Friday, October 4, 6:00pm** **Total Marks: 22**

### General Instructions

- **This assignment is individual work.** You may discuss questions and problems with anyone, but the work you hand in for this assignment must be your own work.
- Each question indicates what to hand in. You must give your document the name we prescribe for each question, usually in the form aNqM, meaning Assignment N, Question M. Put name and student number appear at the top of every document you hand in. These conventions assist the markers in their work. Failure to follow these conventions will result in needless effort by the markers, and a deduction of grades for you. Do not submit folders, zip documents, even if you think it will help.
- **Programs must be written in Python 3.** Marks will be deducted with severity if you submit code for Python 2.
- **Assignments must be submitted to Moodle.** There is a link on the course webpage that shows you how to do this.
- **Moodle will not let you submit work after the assignment deadline.** It is advisable to hand in each answer that you are happy with as you go. You can always revise and resubmit as many times as you like before the deadline; only your most recent submission will be graded.
- Questions are annotated use descriptors like "easy" "moderate" and "tricky". All students should be able to obtain perfect grades on "easy" problems. Most students should obtain perfect grades on "moderate" problems. The problems marked "tricky" may require significantly more time, and only the top students should expect to get perfect grades on these. We use these annotations to help students be aware of the differences, and also to help students allocate their time wisely. Partial credit will be given as appropriate, so hand in all attempts.

## Question 1 (7 points):

**Purpose:** To practice loops, and looking up and using module functions.

**Degree of Difficulty:** Moderate. There's a lot of reading here for 7 marks, but most of the coding is pretty easy.

### Intro: Turtle Graphics

In this question we will introduce you to turtle graphics. A turtle is a cute name for an entity that we can move around a drawing canvas to create pictures. The turtle has a pen which can be in either the up or down positions. When the turtle's pen is in the down position, the pen draws whenever we move the turtle. We can draw a line by, say, moving the turtle 50 pixels to the right while the pen is down. We can also change the colour of the lines that the turtle draws, and we can even ask the turtle to draw specific shapes, such as a circle. When the pen is up, we can move the turtle without it drawing anything.

We can add turtle graphics to a program by importing the `turtle` module.

```
import turtle as t
```

Now we can access the turtle module functions though the object `t`. For example, we can tell the turtle to put its pen down, move 50 pixels in the direction it is currently facing (drawing a 50-pixel long line in the process), and then lift its pen up:

```
t.down()
t.forward(50)
t.up()
```

You can also move the turtle to a specific location specified by an $(x, y)$ coordinate: $x$ is the horizontal position, $y$ is the vertical position. The center of the canvas window is $(0, 0)$. For example, we could move the turtle directly to position (20, 20) (down and to the right of the center) by calling:

```
t.goto(20, 20)
```

If the pen is down, then this will also draw a line from the turtle's current position to the position $(20, 20)$.

We can tell the turtle to draw lines of a given width (if the pen is down) like this:

```
t.pensize(3)
```

The line colour can be changed by calling:

```
t.pencolor(r, g, b)
```

where `r`, `g`, and `b` are values between 0 and 1 indicating the red, green, and blue values of the colour, 0.0 being none of that colour component, 1.0 the maximum amount of that colour component. E.g. `t.pencolour(1.0, 1.0, 0.0)` gives bright yellow (since red + green = yellow).

If you'd like another example, Lecture 6, demo 2 shows a complete program that uses turtle graphics to draw a circle wherever the user clicks the mouse. It uses all of the turtle functions described here.

For this question you will draw some random spirals using turtle graphics.

## Problem Overview: Drawing Spirals

We will draw spirals as an exercise to draw pictures using turtle graphics.

## How to get started

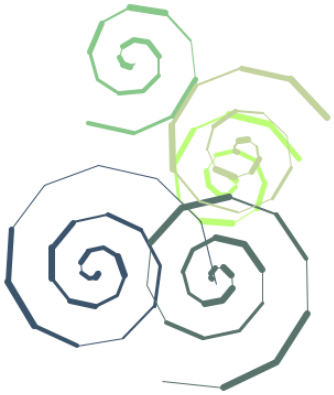
We have provided a partially complete Python program to get you started: `a3q1-starter.py`. You can find it on the Moodle page as a resource for Assignment 3. Download it and add it to your Python project.

The starter file `a3q1-starter.py` defines a function called `draw_spiral()` where you'll add your code to draw a spiral. The rest of the file creates a few variables and determines a few necessary values.

Near the bottom of the file there is a place where you need to create a loop to make the correct number of function calls to `draw_spiral()`. When you call `draw_spiral(segments,size)` you will need to pass the arguments indicating the number of segments in the current spiral and the length of the first segment. The first spiral should have 20 segments, and 2nd should have a random number of segments between 20 and 30, and the third should a random number of segments between 20 and 40, and so on. In the `draw_spiral()` function, there are comments that tells you to write some code to produce a spiral.

## How to colour your spirals

Each spiral should be drawn with a random color



**Getting random colours** Colours can be described by numbers. Perhaps you've learned something about how primary colours combine (red + yellow make orange, etc). One way to do this on a computer is to use numbers between 0 and 1 (this is not the only way). These numbers represent the "amount" of colour you want to combine; zero means none, and 1 means all. We need three such numbers to represent all the colours a computer can use, in terms of how much red, green, and blue to combine. Red is 1,0,0; green is 0,1,0; blue is 0,0,1; black is 0,0,0; white is 1,1,1.

You can set a turtle's pen-colour using three random numbers. You can ask the `random` module to return a number between 0 and 1 (inclusive) using its `random()` function (just to be clear, the module's name is `random`, and it contains a function of the same name `random()`). The number is random in the sense that you cannot predict what it will return, and it will change every time you ask for one. The function `randint(A,B)` from the `random` module yields a random integer between A and B (inclusive). The starter code does not import the `random` module, you'll have to do that yourself. The documentation for functions in the `random` module can be found here (click to link).

## Start a spiral at a random location

Each spiral should begin at a random location with its x coordinate being an integer between -100 and 100, and likewise its y coordinate should be an integer between -100 and 100.

## Drawing a spiral

You should draw a spiral by drawing a series of line segments one at a time in a loop. For each segment you should select a random pensize between 1 and 6. To draw the first segment you move the turtle forward length units, then turn the turtle to the right by 45 degrees. To draw the 2nd segment, you move the turtle forward 2 $\times$ length units, and then turn the turtle to the right by 45 - 1/2 = 44.5 degrees. To draw the ith segment you move the turtle forward length $\times$ i units, and then turn the turtle to the right by 45 - (i-1)/2 degrees

### Other Hints

The documentation for the `turtle` module containing descriptions of how to use the turtle module functions can be found here (click to link). Click on the name of a function to get more details.

When you run a program that uses turtle graphics, it pops up a canvas window on the screen. This window will often hide behind other windows (such as PyCharm) so if you think nothing is happening when you run your program, make sure you move your windows and look behind them. Also, the program is only terminated when you close the canvas window, so make sure you close the window between runs or you may get confused by having multiple drawing canvases open at the same time.

### What to Hand In

- Rename your completed `a3q1-starter.py` file to `a3q1.py` and submit it.

## Evaluation

- 1 mark for correctly importing the `random` module;
- 1 mark for calling `draw_spiral()` with the correct random number of segments each time;
- 1 mark for starting each spiral at the correct random location;
- 1 mark for drawing each spiral with a random color
- 1 mark for using a random pensize between 1 an 6 for each segment in a spiral
- 2 marks for correctly drawing a spiral

## Question 2 (7 points):

**Purpose:** To practice using loops to repeat actions until a condition is met (and solve a practical problem at the same time!).

**Degree of Difficulty:** Easy

A friend of yours has gotten the new "Rocket" credit card. Every dollar purchased gets you 1000 meters towards a free one-way trip to the planet Venus! To attract customers the card offers a novel interest rate. For each month in debt, the percentage owed in interest increases by a power of 2. For example, after 1 month, the interest owed is equal to 2% ($2^1$) of the **original** debt. After 2 months, the interest is 4% ($2^2$) of the **original** debt and so on. In general:

$$interest(m) = d \times (2^m/100)$$

Note: This formula is not a Python assignment statement! It is an equation that says that if you know the quantities $m$ (the number of months) and $d$ (the original debt), you can calculate a value using the right side of the equation, giving you a quantity that tells you the the total interest after $m$ months.

Write a program which does the following:

- Prompt the user to enter an initial amount of Rocket credit card debt. You must make sure that the user enters a positive number. If they do not enter a positive number, print a message informing them so, and prompt them to enter the initial amount of debt again. Keep doing this until they enter a positive number.
- Starting from month 1, print to the console the total payment owed at 1-month intervals. Thus, for month 1, month 2, etc., you should print out the total payment owing, which consists of the interest **plus** the original debt. Your program should stop after the first month where the total payment owing is greater than **100 times** the original debt (that's the point at which the credit company calls in its debts, whether you like it or not!).

Don't forget to import the math module if you need math functions. Using the math module is not required and there are correct solutions that don't use it.

Hint: A correct solution should make use of two separate while-loops.

## Sample Run

Sample input and output (input typed by the user is shown in green text):

```
Enter amount of debt in dollars: -7
Error!  Debt must be a positive number!
Enter amount of debt in dollars: 1
After 1 months, you owe: $1.02
After 2 months, you owe: $1.04
After 3 months, you owe: $1.08
After 4 months, you owe: $1.16
After 5 months, you owe: $1.32
After 6 months, you owe: $1.6400000000000001
After 7 months, you owe: $2.2800000000000002
After 8 months, you owe: $3.56
After 9 months, you owe: $6.12
After 10 months, you owe: $11.24
After 11 months, you owe: $21.48
After 12 months, you owe: $41.96
After 13 months, you owe: $82.92
After 14 months, you owe: $164.84
Time to pay up!
```

### Testing

Test your program first using $d = 1$, and then using two more inputs of your choice — these should be unique to you and unlikely to be chosen by two different students. Make sure to look closely at the results and do a few calculations by hand to convince yourself that your program is correct! Copy the console output of your three test runs to a document, and hand it in with your program code (see below).

### What to Hand In

- A file called `a3q2.py` containing your finished program, as described above.
- A document called `a3q2.txt` (.rtf, .pdf and .doc are also okay) containing the console output from the testing that you did to verify that your program works correctly.

## Evaluation

- 2 marks for correctly verifying the console input
- 4 marks for producing the correct console output for the given inputs
- 1 mark for the console output of your tests.

## Question 3 (8 points):

**Purpose:** To practice using modules. Practice with mixing loops and conditionals.

**Degree of Difficulty:** Moderate.

A video game developer is ready to publish their new game, Meteors II (sequel to their breakout hit Meteors). In addition to the regular edition, the developers want to offer a special edition of the game containing extra goodies (e.g. squishy meteor plushies!). They are trying to set the selling price of the game's special edition, and determine how many copies of it they should make.

Suppose the number of units the publisher can manufacture is determined by the *supply* function $S(P)$ if they sell the game for price $P$:

$$S(P) = 500 + 90P$$

Suppose the number of copies they expect to sell is determined by the *demand* function $D(P)$ if they set the selling price as $P$:

$$D(P) = 10000 - 35P$$

If the game is priced lower than the optimal price then they would not make enough copies for the number of fans who want to buy the special edition of the game (demand would exceed supply) and lose out on potential sales. If they were to set the selling price of the game higher than the optimal price, then they would saturate the market with too many copies of the game (supply would exceed demand) and might fail to make a profit.

This means that the optimal price for the company to sell the special edition for is the price at which the number of copies produced by the publisher is the same amount as the number of copies demanded by the consumers, that is, the price at which supply equals demand:

$$S(P) = D(P)$$

In economics, this optimal selling price is known as the *equilibrium price*.

Since the functions $S(P)$ and $D(P)$ given above are quite simple, we could set the two expressions equal to each other and solve for $P$ using Grade 10 algebra. But supply and demand functions are not always so simple, and for more complicated functions, it might be hard to solve for $P$ using algebra.

A very handy way to solve problems requiring complicated algebra is to write a Python program to find the optimal value for $P$ *numerically*, that is, by searching for a numeric value of $P$ that makes $S(P)$ equal to $D(P)$. Solving complex algebraic problems numerically is a common problem solving paradigm in computer science. In fact, numerical calculations like this dominated the early use of computers in the dark ages before computers had screens.

Here's a guide for getting this program working. You should try your program out every time you reach a point where you've accomplished one of the items in the list. Don't wait to the end to try it out. This should be a very gradual process of getting something working, then moving on to the next part.

(a) Write a function called `supply()` with parameter $P$, returning the value of $S(P)$, as above. Document it with a docstring.

(b) Write a function called `demand()` with parameter $P$, returning the value of $D(P)$, as above. Document it with a docstring.

(c) Print the three columns ($p$, $S(p)$, and $D(p)$) in the console. Do this using a for-loop, for each price $p$ between \$10 and \$160 at intervals of \$1. Start a new line for each new $p$.

(d) Add an if-statement to your for-loop from part (c) that you will use to remember the price $p$ for which $S(p) = D(p)$. After the loop is finished, have your program print the optimal price. Make sure you check that the price is indeed the right price!

(e) Now we're going to make modifications to your program so that it displays a nice graph instead of a table of numbers. The first step will be to add a line of code at the top of your program to import `matplotlib.pyplot`. Nothing that follows in this list will work if you skip this step!

(f) Call the `figure()` function of `matplotlib.pyplot` to prepare Python for the creation of a new figure. This function should be called before computing any of the data you want to plot. It has no actions you can see, but if you don't call this function, none of the remaining steps will work!

(g) Call the function `show()` of `matplotlib.pyplot`, near the end of your program. This function will display all the stuff that gets plotted since the call to `figure()`. Since we haven't plotted any data yet, if you run the program, Python will pop-up a window that is mostly empty. While this window is on the screen, your program is still running; so, to allow your program to finish normally, close this window when you are done admiring it.

(h) In the for-loop that creates the table of data, call the `plot()` function from `matplotlib.pyplot` to put the data into the graph. Call `plot()` once to plot the point $(p, S(p))$ and again to plot the point $(p, D(p))$. The `plot()` function has three parameters in this order: the x-coordinate of the point to plot (price), the y-coordinate of the point to plot (quantity), and a *style string*. Use the style string literal `'og'` ('g' meaning green, 'o' meaning circles) when plotting the supply value, and `'ob'` ('b' meaning blue, 'o' meaning circles) for the demand value. When you get this going, you can delete the line of code that displayed the data in tabular form. It was only useful as a stepping point to our graph!

(i) Change the if-statement in the loop so that in addition to printing the optimal price, it plots the point $(p, S(p))$ for the optimal price $p$. Use the style string 'ro' so that the point is a red circles (the order of the characters is irrelevant, so you could use 'or' too).

(j) Now, make your graph look nicer. Look up the `xlim()` and `ylim()` functions from the `matplotlib.pyplot` module. Use them to set the minimum and maximum values on the x-axis of your figure to be between 0 and 170, and the y-axis values to be between 0 and 16000, as in the sample output figure below.

(k) Look up the `xlabel()`, `ylabel()`, and `title()` functions from the `matplotlib.pyplot` module. Use them to add a plot title and labels to the x- and y-axes as in the sample output figure below.

(l) Look up the `annotate()` function from the `matplotlib.pyplot` module. Use it to label the point (`optimal_price`, `supply(optimal_price)`) with the string `"Optimal Price"` as in the sample output figure below.

(m) Be sure that the optimal price and the quantity to be produced at that price (i.e. the value of `supply(optimal_price)`) is still displayed on the console. This is the solution to the problem we set out to solve!

## matplotlib.pyplot

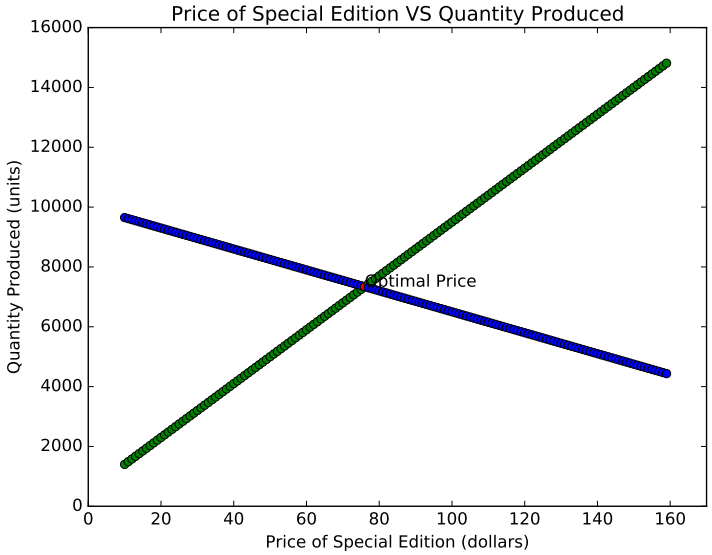
Documentation for `matplotlib.pyplot` is available at http://matplotlib.org/api/pyplot_api.html. You can find information on how to use `xlim()`, `ylim()`, `annotate()`, `xlabel()`, `ylabel()`, and `title()` there.

## Sample Output

Your console output should look something like this:

```
The optimal selling price of $76 will result in the sale of 7340 units.
```

Your graph should look more or less like this:




### What to Hand In

- A file called `a3q3.py` containing your finished program, as described above.

## Evaluation

- 2 marks for correct implementation of the `supply()` and `demand()` functions.
- 1 mark for creating good docstrings of the `supply()` and `demand()` functions.
- 1 mark for correctly plotting the values of the `supply()` and `demand()` functions (using green and blue points, respectively) using the loop in part (h).
- 2 marks for correctly recording the optimal price, and plotting it with a red dot within the loop in part(d).
- 2 marks for acceptable figure labelling and appearance as in the sample output.

### Side Note: A Word of Caution about Solving Equations Numerically

We have carefully crafted the `supply()` and `demand()` functions so that an integer value of $P$ is the optimal price. In general, numerical solutions use floating-point values, and determining whether two floating-point values are equal requires more care, because of the limited precision of floating-point numbers. If we aren't careful, a program can fail to discover when two numbers are equal (for all intents and purposes), because of tiny errors in their calculation. In such situations, instead of equality, we check if two numbers are "close enough." But you don't worry about that for this question.