CSci 1133
Lab Exercise 2

Basics

In the previous lab exercise, you learned to start the Python interpreter and interact with it in *command* mode, one expression at a time. You also used a text editor to enter Python commands into a script file named `decay.py` Python *programs,* or *scripts,* are written and saved as text files with the extension `.py` They contain a number of Python statements that are subsequently executed one after another (automatically) by the interpreter operating in *script* mode. To "run" the radioactive decay program, you invoked the Python interpreter in script mode using the `python3` command followed by the name of the script file:

```
python3 decay.py
```

Note that the terms *program* and *script* are interchangeable in Python. All Python *programs* are written as *scripts* using a text editor (e.g., Atom) and subsequently executed by the interpreter in script mode.

Although using Python in command mode is very useful for trying things out and testing simple ideas, most of our efforts from now on will involve writing complete programs using a text editor and executing them separately as scripts.

Generally, you will find that the programs you write don't actually do what you intend when you first run them. Either you will make unintended typing mistakes (*syntax* errors), or something crazy will happen which results in the interpreter halting the execution of your program with a dire-sounding warning message (*runtime* errors). As you get better at writing Python programs, you will encounter fewer *syntax* and *runtime* errors, however you will often find that your program runs successfully but produces erroneous results. This, more common problem, is known as a *logic* or *semantic* error and will require that you *deduce* why your program isn't doing what you intended. It is a bit like detective work. We refer to these sorts of errors as "*bugs*", and the process of determining what is wrong with your program and fixing it: "*debugging*". Debugging involves isolating potential problems, editing your source file to remedy the problem, and then re-running your script. You may have to repeat this cycle many times to get your program to produce correct results. This is actually a very common scenario and, in fact, is one of the primary challenges for anyone who writes computer programs for a living!

Procedural Abstraction

Abstractions allow us to refer to more complex entities by *name*, ignoring much or all of the details of computation. For example,

```
>>> DegPerRadian = 180 / 3.14159 <ENTER>
```

provides a useful constant value for converting between radians and degrees without the need to continually remember and enter the expression. The assignment operator, = , (it's not an equal sign!) is used to *bind* the result of some expression to a label name.

The binding between a name and an object remains until the program ends or another explicit binding occurs. Consider:

```
>>> x = 3 <ENTER>
>>> x = x + 2 <ENTER>
>>> x <ENTER>
5
```

In this example, the first command *binds* the label x to the integer object with value: 3. Because assignment is of lower precedence than addition, the second expression will first obtain the stored value for the object bound to x (i.e., 3) add 2 and *re-bind* the label x to the resulting object (i.e., the integer with value 5).

Expressions also may include *function calls*. A *pure-function* in Python is a form of *procedural* abstraction: a named procedure that performs a specific sequence of computations and returns a result. Carefully note the use of the word "call". Unlike functions in mathematics, which are generally *declarative* (statements of fact), Python functions are *imperative*; a series of computational steps that are *dynamically* executed whenever the function name is encountered by the interpreter. We say that the function is "called" by the interpreter. Python provides a number of built-in functions. Try this:

```
>>> abs(-3.4)  <ENTER>
3.4
```

We supply a value as the function *argument,* and the abs() function executes the necessary computational steps to determine and return the absolute value of the value. Here's another:

```
>>> min(3,5,-8,1,2)  <ENTER>
-8
```

min( ) takes any number of arguments separated by commas and computes/returns the value of the smallest one.

Function arguments are *expressions* that are evaluated before the function is called:

```
>>> abs(x//3*43.61)  <ENTER>
43.61
```

[At this point, stop and explain to one of your TAs why this produced the value 43.61...]

Function calls are expressions and the arguments to functions are expressions, therefore it is logically consistent that function calls can be used as arguments to other functions! We refer to this situation as a *nested call* and it represents a very powerful *functional* form of computation:

```
>>> abs(min(3,5,-8,1,2))  <ENTER>
8
```

Python Modules

abs() and min() are members of a "built-in" collection that consists of functions considered "basic" to the language. In addition, a large assortment of other Python functions have been developed and provided by other people. These functions are gathered into specific library "modules" that must be explicitly loaded into your programs before you can use them.

The `math` module contains a particularly useful set of abstractions such as square root, sine, cosine, and other common mathematical operations. The `math` module also includes useful constant definitions such as `pi`. In order to use these functions and definitions, you must first *import* the `math` module into your program.

Let's import the math module functions:

```
>>> import math <ENTER>
```

By the way, if you ever forget which functions are included in a module, or are just curious what might be available, you can use the built-in Python function `help()`:

```
>>> help("math") <ENTER>
```

The `help` function takes a string argument and provides a complete listing of the contents of the specified module. Try it in interactive mode. You can scroll up/down using the terminal controls. If you just want a concise list of the function names, use the built-in `dir()` function:

```
>>> dir(math)
```

Now let's try to compute the square root of a number using the `sqrt()` function from the math module:

```
>>> sqrt(144) <ENTER>
```

Oops! What happened?

When you load a module using the `import` *modulename* command, the function names must be preceded with the *modulename* and a period when you call them:

modulename.functionname( )

Like this:

```
>>> math.sqrt(144) <ENTER>
```

Go ahead and try it.

That's more like it! You always need to provide the module name followed by the period before the function name.

The Turtle Module

The `turtle` module includes many fairly sophisticated functions that will do more than move turtles and draw things. For example, you can request a value to be entered into your program by the user. Let's try it! First, import the turtle module and then enter:

```
>>> turtle.numinput('','Enter a number: ') <ENTER>
```

....hmmm why didn't the interpreter respond? Take a look at the turtle window. A dialog box has appeared with the text 'Enter a number: ' and it is waiting for you to enter something. In effect, the interpreter has "called" the `numinput()` function which is now waiting for input. The call to `numinput()` won't return a value until you enter something.

Try entering the number 42 in the dialog box. The command mode interpreter should report that the result of the function call is the floating-point object whose value is 42.0:

```
    >>> turtle.numinput('','Enter a number: ')
    42.0
```

The `numinput()` function accepts a number of arguments. Let's have Python show us what they are:

```
    >>> help('turtle.numinput') <ENTER>
```

There are two required arguments that must be entered in the order shown. The first is a string that will appear as the title of the dialog box. The second one is a *prompting* message (string) that will be written to tell the user what to do.

If we want to use the input in our program, we will have to bind the returned value to a variable name. Try this:

```
    >>> thenumber = turtle.numinput('','Enter a number: ') <ENTER>
```

Enter 42 in the dialog box again and then take a look at the following turtle function:

```
    >>> help('turtle.write') <ENTER>
```

The `turtle.write()` function will output text to the turtle window starting at the current turtle location. The little turtle symbol will just be on the way, so let's hide it, and, since the number we entered is actually the meaning of life, the universe and everything, we should convert it to an integer value!

```
    >>> turtle.hideturtle() <ENTER>
    >>> thenumber = int(thenumber) <ENTER>
```

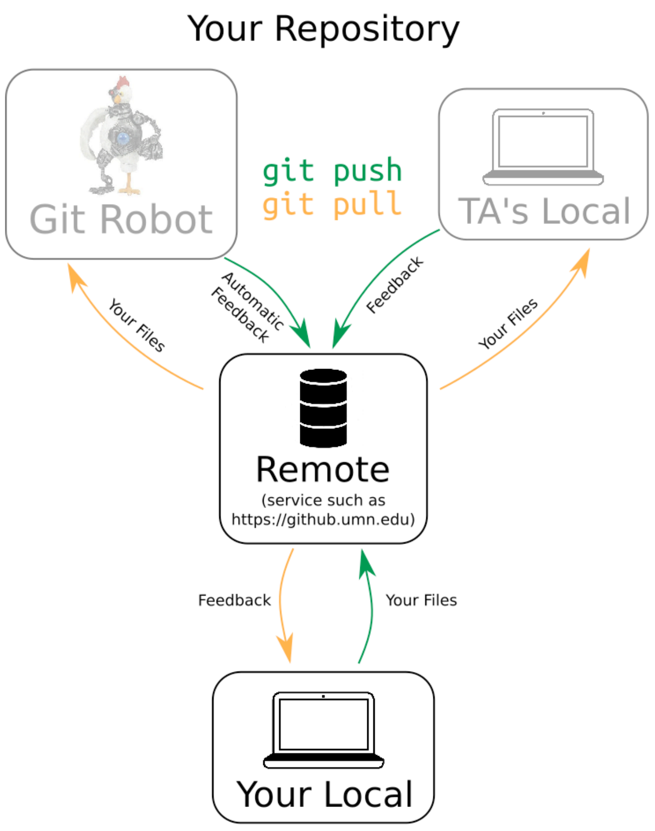
Now write `thenumber` to the display using the `turtle.write()` function. If you don't like the way it looks, you can try a different font= argument.

Saving Python Scripts

As mentioned in Lab 1, we're going to be using a system named `Git` to store and submit assignments during the semester. `Git` will also be used to provide you with automatic feedback on your exercises, and course feedback from your TAs. `Git` is an advanced software version-control system that is designed for programming projects. It works by storing 'snapshots' of your code so that you can go back to old versions if you mess something up, or just want to use your old code as a reference. `Git` has many great features, but you only need to understand a few of them to use it for storing your programs. Later, we'll post a comprehensive `Git` tutorial on our class website.

Your Repository

A *repository* is a place where program source code is saved in `Git`. Your repository (repo) has already been created for you at **https://github.umn.edu/umn-csci-1133S17**. Your specific repo will be named something like `repo-user0000` (where `user0000` is replaced with your Internet ID). If you discover that your specific repo does not exist, make sure you have a `GitHub` account, then talk to one of your TAs so one can be created for you. Right now, your repo only exists on a **remote** sever. In order to save your program code in your repo, you'll need to first create a **local** copy of it. Remotes and locals are explained in the following diagram:



Creating a local repo copy

To get a local copy of your repo on your computer, you'll first need to make a "clone" of it. To do this, run the `git clone` command from your terminal. For example, running the command:

```
git clone https://github.umn.edu/umn-csci-1133F17/repo-user0000.git
```

will create a local **clone** of the repo `repo-user0000` in your home directory. This local copy and the remote copy can be synchronized using commands we will learn later in this lab. Syncing these copies allows you to share your changes and get the feedback provided on your work.

Go ahead and try it; create a local clone of your own repo by running the command above, replacing the text, "`repo-user0000`" with the name of your repo. If you're working with a partner, select one person's repository to use for this lab. (When you type your password, you won't see anything, even though it is registering. Linux hides your password for security reasons.)

Configuring Git

Before we save anything on `GitHub`, we need to first configure the version of `Git` running on your computer. This process will tell `Git` who you are, and it only needs to be done **once** for your lab account. To use `Git` on your own computer, you will have to repeat these commands on your computer as well.

First, run the following command, replacing the text `user0000` with your own Internet ID:

```
git config --global user.email "user0000@umn.edu"
```

Next, run this command, substituting your real name for the text, `Your Name`:

```
git config --global user.name "Your Name"
```

`Git` is now configured, and you're ready to begin putting files on `GitHub`!

Creating files and directories in your repository

Now that you've cloned your repo, you can treat it as a normal directory on your computer. First, change your working directory to your repo. Once inside of your repo, create a directory called `lab2` and navigate into it. Now that you've created the directory for this lab, we can finally begin writing code. Open the `Atom` editor in your working directory by running `atom .` (including the period) and let's get started!

Scripted Turtles, Part 1

Recall the steps that you used to draw a square using the turtle graphics module in the previous lab. Instead of typing the commands interactively into the shell, write them as individual lines of code in a script file. Start by importing the turtle module and then adding the necessary Python statements to draw a square with sides of length 100. Refer back to Lab1 if you need to.

When you've finished entering your program, save it as: `drawsquare.py`. Note that it is always necessary to use the `.py` extension to indicate that it is a Python source file. Then run your program using Python in script mode. Using the Python script command:

```
python3 drawsquare.py
```

will work. However, because your program terminates immediately following the last drawing command, the graphics window will immediately close and you will not be able to admire your handiwork! In order to have the graphics display "stick around", you need to run your script in "interactive" mode. Interactive mode will run your program and then immediately put Python in command-line mode, allowing you examine variables, call functions or, in this case, view your graphics display before it closes. It's easy to do, simply include a `-i` flag to the `python3` command:

```
python3 -i drawsquare.py
```

Go ahead and try it. If you've done everything correctly, you should have a square drawn on the display. If not, you will need to `exit()` the Python command-line environment, figure out what your program is doing wrong and correct it.

Part 2

Modify your program to have a user input the length of each side. Remember to convert the value to an integer object and then modify your statements to use the value entered by the user.

Part 3

Now modify your program to draw three more squares, each rotated 30 degrees counterclockwise from the preceding one.

Part 4
This graphics stuff is fun! But writing out all of those commands starts getting very tiresome indeed. We can save ourselves a lot of effort by noticing all the similarities. Note that drawing each square requires *exactly* the same steps. It's a computational procedure in its own right. If we give a name to this procedure, like "Drawsquare", then the task gets a lot easier. We could just invoke the "Drawsquare" procedure, turn the turtle 30 degrees counterclockwise, invoke the Drawsquare procedure again, turn the turtle, invoke Drawsquare, etc.

We just need a procedural *abstraction*: a "Drawsquare" function.

It's easy to do in Python. We can create our own function using the Python `def` statement:

```
def Drawsquare( ):
    turtle.forward(100)
    turtle.left(90)
    turtle.forward(100)
    turtle.left(90)
    turtle.forward(100)
    turtle.left(90)
    turtle.forward(100)
    turtle.left(90)
```

The `def` statement "defines" a new function, binding the name `Drawsquare` to the procedure described by the lines of code that appear immediately following the definition.

There is a lot going on here that you need to understand. First, the syntax is very precise! The parentheses and colon character are required. The lines of code following the definition are the statements that will be executed *when the function is called* ! These lines of code constitute the function *body* and must be all indented using the tab key in the editor.

Replace the lines of code in your program with the function definition as shown above and run your program. (You will need to keep the `import` and `turtle.showturtle()` lines)

What happened?! When you run your program, nothing happens. Why not?

Remember that functions must be *called*. The function definition does nothing until the function is called. Then, and only then, will the statements in the function be executed.

From the shell, try your new function by entering it at the command prompt:

```
>>> Drawsquare()
```

The turtle should be dutifully drawing a square. Now type:

```
>>> turtle.left(30)
>>> Drawsquare()
```

Assuming you've done everything correctly, a second square should be drawn, 30 degrees counterclockwise from the first. Now rebuild your program to use the `Drawsquare` function to draw 4 squares, each rotated 30 degrees counter clockwise.

Part 5
The `Drawsquare` function is a useful abstraction, but pretty limited. We would probably like to draw squares of various sizes, not just 100 units on a side! Our *abstraction* is not *general*. It always does *exactly* the same thing each time it is called. Like the math functions we explored earlier, it would like be nice to pass a size value to the function as an *argument*.

In Python, we simply provide a variable name that will be automatically *bound* to the argument during the function call process. Once we've provided the variable name, Python takes care of the rest. When the statements of our function are executed, the *variable* will contain the value of the argument that was given in the function call:

```
def Drawsquare(sidelength ):
    turtle.forward(sidelength)
    turtle.left(90)
    turtle.forward(sidelength)
    turtle.left(90)
    turtle.forward(sidelength)
    turtle.left(90)
    turtle.forward(sidelength)
    turtle.left(90)
```

Modify your function definition to accept an argument containing the length of a side. Try testing your function using the interactive shell command mode to make sure it can draw squares of various sizes.

Now modify your program to both rotate the squares and provide sides of 100, 200 and 300 for the rotated squares, respectively.

Now that we've created the file `drawsquare.py`, we're at a good point to pause and save our file on `GitHub`.

Putting your files on GitHub
There are four important commands to remember for putting your code on `GitHub`:

1. `git status`:
   Tells you the status of files and directories in your repo. If you're unsure what to do next in your repo, this is a good place to start. Note that this is not a necessary step; it only gives you information, rather than telling Git what it needs to do.

2. `git add [files, directories...]`:
   Adds (**stages**) changed files and directories to be committed to your repo.

3. `git commit -m "Commit message"`:
   Creates a **commit** with all of your changed files and directories. The commit message you include should describe what changes you are including in this commit. This means all of your staged changes are now "saved" in your local repository, but ***are not yet on the remote***.

4. `git push`:

Pushes all of your active local commits to the remote server (https://github.umn.edu for this class)

Often, the four commands above will be executed right in a row. For example:

```
git status
git add test_file.py
git commit -m "Added a test Python script"
git push
```

Let's save the `drawsquare.py` file you've created on `GitHub`. First, *add* the file to `Git` by running the following command in your terminal:

`git add drawsquare.py` in your terminal.

Then, create a `Git` *commit* with this file in it by running:

```
git commit -m "Added square drawing".
```

Once you've done this, you're ready to *push* your local commit to the remote at `https://github.umn.edu`

You do this by running the command:

```
git push
```

`Git` will then ask you for your username and password – use the same username and password you use to log into the lab computers.

That's it! Your code is now safely stored on `GitHub`! You can verify this by visiting `https://github.umn.edu/umn-csci-1133S17/YOUR-REPO` using any web browser.

Fun with Circles!

OK, you're getting the hang of it. Many more (sophisticated) things can be done with turtle graphics. You've seen how to hide the "turtle" icon using `turtle.hideturtle()`. You can speed up the turtle using `turtle.speed()` and `turtle.delay()`.

Sometimes you want to move the turtle without leaving a line. This is accomplished by using the `turtle.penup()` function. Remember to put the pen back down when you want to draw something again!

Now write a new Python program named "`olympic.py`" that will do the following:

- Define a function that will accept a single radius argument and draw a circle (with the given radius)
- Obtain a radius value from the user
- Draw the Olympic rings using your defined function, with each ring having the user-provided radius. Use the correct colors! If you cannot remember what the Olympic rings look like, check out Wikipedia.

For a bigger challenge, "scale" the rings by setting the location of the circles based on their given radii.

Remember, once you finish writing your `olympic.py` program, save it to the remote server by using the `Git` commands introduced earlier in the lab.