ENED1100 – ACTIVITY
PYTHON 4: FUNCTIONS

TASK: WARMUP

OBJECTIVES: Warm-up task to review Python topics from last semester: conditional structures and repetition structures.

TIMEKEEPER: This task should take approximately 20 minutes. The time estimate is only a guideline (some teams may use less time, while others may use more).

Remember that you can (and should!) use commenting in your Python scripts to help you remember what you were thinking and what your solution is doing.

Unit conversion is a common task required in engineering projects. Develop a Python script that will prompt the user for the following:

- A numerical value for pressure
- The units for the pressure value that was entered by the user.
- The desired units for the pressure value

For this task, limit the allowable units to Pascals (Pa) and pounds per square inch (psi).

The script should then determine if the units were entered properly (either Pa to psi or psi to Pa). If so, the pressure should be converted to the new set of units and the script should output (print) both the converted pressure and the new set of units. If units were not entered correctly, the script should output (print) an error message to the user indicating that the units were not entered correctly.

The script should then ask the user if he/she would like to do another conversion. If so, the process above should be repeated. If not, the script should terminate.

Test your script. You can readily obtain test values for your script by googling a psi to Pa calculator.

Save your program as **ACT_Python_4pW_*UCusername*.py**

ENED1100 – ACTIVITY
PYTHON 4: FUNCTIONS

TASK 1 (OF 4)

OBJECTIVES: Practice creating and calling functions in Python.
TIMEKEEPER: This task should take approximately 15 minutes. The time estimate is only a guideline (some teams may use less time, while others may use more).

Remember that you can (and should!) use commenting in your Python scripts to help you remember what you were thinking and what your solution is doing.

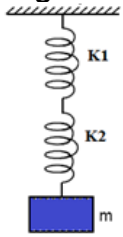
Hooke's law is an equation for how much force is required to compress or stretch a spring by a certain amount:

$$F = Kx$$

F Force (N)
x Distance spring is compressed or stretched (m)
K Spring Constant (N/m)

A higher spring constant means that the spring is more difficult to compress or stretch. Springs can be combined in series or in parallel as shown in the diagram below.

Springs in Series:

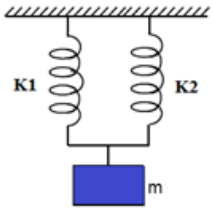


$$\frac{1}{K_{eq}} = \frac{1}{K_1} + \frac{1}{K_2}$$

$$F_{total} = F_1 = F_2$$

$$x_{total} = x_1 + x_2$$

Springs in Parallel:



$$K_{eq} = K_1 + K_2$$

$$F_{total} = F_1 + F_2$$

$$x_{total} = x_1 = x_2$$

Note: You should have a trace table of test values for this from ACT_Python_2_Conditional. If not, create one.

Develop a Python function, **Springs**, that has the following input arguments: spring constants, the total force, and the configuration (series or parallel). The function should check to make sure that K1, K2, and Ftotal are positive numbers. For invalid inputs, the function should output an error message to the user. For valid inputs, the program should return Keq, F1, F2, x1, x2, and xtotal.

Save your function as **Springs.py.** Run your function module then test the function in the Python shell.

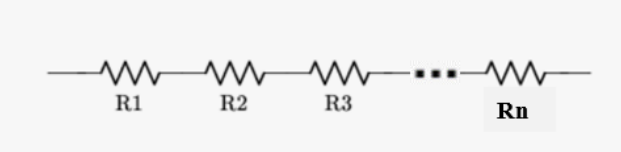
ENED1100 – ACTIVITY
PYTHON 4: FUNCTIONS

TASK 2 (OF 4)

OBJECTIVES: Practice creating and calling functions in Python.
TIMEKEEPER: This task should take approximately 25 minutes. The time estimate is only a guideline (some teams may use less time, while others may use more).

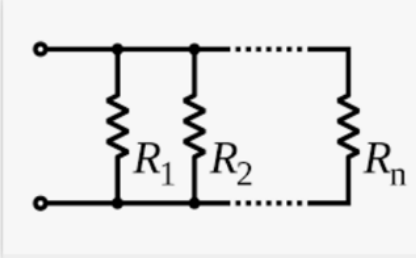
Resistors are very common components in any type of electronic devices. Resistors can be connected in series, in parallel, or a combination of both. The diagram below shows a series connection and a parallel connection along with the equations for determining the total resistance.

Resistors in Series



$$\text{Rtotal} = \text{R1} + \text{R2} + \ldots \text{Rn}$$

Resistors in Parallel



$$\frac{1}{\text{Rtotal}} = \frac{1}{\text{R1}} + \frac{1}{\text{R2}} + \ldots \frac{1}{\text{Rn}}$$

$$\text{Rtotal} = \frac{1}{\frac{1}{\text{R1}} + \frac{1}{\text{R2}} + \ldots \frac{1}{\text{Rn}}}$$

PART A

Create a set of test values by calculating the total resistance for a series circuit and for a parallel circuit assuming R1 = 200 Ω, R2 = 470 Ω, and R3 = 360 Ω.

PART B

- Develop a Python function, **Circuits**, which has 2 input arguments: the number of resistors and the configuration (series or parallel).
- The function should prompt the user for the resistor values – one resistor value at a time.
- If a resistor value is not positive, the function should prompt the user to re-enter the resistor value until a positive value is added.
- The function should calculate and return the total resistance.

ENED1100 – ACTIVITY
PYTHON 4: FUNCTIONS

TASK 3 (OF 4)

OBJECTIVES: Practice creating and calling functions in Python.

TIMEKEEPER: This task should take approximately 35 minutes. The time estimate is only a guideline (some teams may use less time, while others may use more).

PART A

- Develop a Python function, **Forces**, which has 5 input arguments: the magnitude and angle of Force 1, the magnitude and angle of Force 2, and units for the angles (degrees or radians).
- The function should first check for valid inputs: The magnitudes must be positive and the units must be 'degrees' or 'radians' (could use 'D' or 'R' instead).
- If any of the inputs are invalid, the function should print a message to the user and terminate without returning any values.
- If the inputs are valid, the function should compute the resultant force and return the magnitude and angle of the resultant force. The angle should be in the same units as the two forces were originally in.

To compute the resultant force:

1. Convert each force to rectangular form (remember Python trig functions use radians):
$$Fx = F\cos(\theta) \text{ and } Fy = F\sin(\theta)$$
2. Add the x-components and y-components of the two forces to get the x and y components of the resultant force, Frx and Fry.
3. Convert the resultant force from rectangular form to polar form:
$$Fr = \sqrt{Frx^2 + Fry^2} \quad \text{and } \theta r = \begin{cases} \operatorname{atan}\left(\frac{Fry}{Frx}\right) & Frx \geq 0 \\ \operatorname{atan}\left(\frac{Fry}{Frx}\right) + \pi & Frx < 0 \end{cases}$$

   Remember that arc tan only returns angles in the first and fourth quadrant. If Frx is negative you need to add pi to the atan value as indicated in the formula above.

Save your function as **Forces.py**. Run your function module then test the function in the Python shell.

Sample Output:

Assume the following order for input arguments: Mag of F1, Angle of F1, Mag of F2, Angle of F2, Units ('D' or 'R')

```
>>> Forces(10,180,5,90,'D')
(11.180339887498949, 153.43494882292202)

>>>  Forces(10,45,5,200,'D')
(5.862526869561879, 66.12724263592911)

>>>  Forces(10,0.7854,5,3.4907,'R')
(5.862383925381395, 1.1541165024299962)
```

ENED1100 – ACTIVITY
PYTHON 4: FUNCTIONS

PART B

- Write a Python script that will allow a user to enter any number of forces and will compute and output the resultant force.
- The script should first prompt the user for the magnitude and angle of Force 1, the magnitude and angle of Force 2, and units for the angles (degrees or radians).
- The script should check if the inputs are valid (magnitudes must be positive and units should be degrees or radians). If any inputs are invalid, the script should continue to prompt for these inputs until they are valid.
- The script should then call the function, **Forces**, to compute and return the resultant force for the first two forces.
- The script should then ask the user if they would like to add another force. If so, the user should be prompted for the magnitude and angle of the new force. (Assume units for angle will be consistent with the original set of units for angles.)
- The script should call the function, **Forces**, to find the new resultant by combining the previous resultant with the new force.
- This process should continue until the user has no more forces to add.
- The script should then print the final resultant force, both magnitude and angle, using the original units for angle.
- Test your script.

Sample Output:

```
Enter the magnitude of Force 1:  10
Enter the angle of Force 1: 180
Enter the magnitude of Force 2:  5
Enter the angle of Force 2: 90
Enter units for angles (D or R): D
Would you like to add another force (Y or N): Y
Enter the magnitude of the next force:  10
Enter the angle of the next force: 45
Would you like to add another force (Y or N): Y
Enter the magnitude of the next force:  5
Enter the angle of the next force: 200
Would you like to add another force (Y or N): N

Magnitude of Resultant = 12.8657
Angle of Resultant = 126.3592 D
```

Save your program as **ACT_Python_4p3_ *UCusername*.py**

ENED1100 – ACTIVITY
PYTHON 4: FUNCTIONS

TASK 4 (OF 4)

OBJECTIVES: Practice creating and calling functions in Python.
TIMEKEEPER: This task should take approximately 20 minutes. The time estimate is only a guideline (some teams may use less time, while others may use more).

(**See also**: ACT_Python_3_Repetition Task 3.) At its most basic level, the only mathematical operations a computer can perform are addition, subtraction, multiplication, and division. In order to implement more complex mathematical operations, iterative or numerical methods have been developed that only utilize these basic operations.

One example of this is the computation of the n^{th} root of a number. There is a method known as the Newton-Raphson algorithm that allows you to determine the n^{th} root of a number using only subtraction, multiplication, and division.

Suppose we want to find $x = \sqrt[n]{N}$ (i.e. the nth root of N). The algorithm works as follows:

- Make an initial guess for x: Est
- See how close you are: $\text{Error} = |\text{Est}^n - N|$
- If the Error is too big, first save the latest estimate then update the estimate as follows

$$\text{Est}_{\text{Previous}} = \text{Est}$$

$$\text{Est} = \text{Est} - \frac{\text{Est}^n - N}{n * \text{Est}^{(n-1)}}$$

- Calculate the Error as follows (different equation than before for Error):

$$\text{Error} = |\text{Est} - \text{Est}_{\text{Previous}}|$$

- Repeat this process until the Error is small enough.

PART A

Create a Python function, **nRoot**, which implements this algorithm to determine the n^{th} root of a number. The input arguments are the number N, the root n, a tolerance for the error, and the initial guess. The function should continue to iterate until the Error is smaller than the tolerance value. The output arguments are the estimated n^{th} root of N and the number of iterations required.

You need to check for invalid inputs. In the case of invalid inputs, you should output a message to the user and terminate the program without returning values. Invalid inputs would be:

- Negative value or zero for tolerance
- Negative value for number, N, and even number for root, n (see hint below).

Hint: An easy way to check if n is odd or even is to use the modulus operator. The modulus operator does division and returns the remainder of the division. So,

$$n\%2 = \begin{cases} 0 & \text{if n is even} \\ 1 & \text{if n is odd} \end{cases}$$

Save your function as **nRoot.py**. Run your function module then test the function in the Python shell.

Sample Output:
Assume the following order for input arguments: N, n, Est, Tol

```
>>> nRoot(10000,4,1,0.01)
(10.000000000394822, 24)

>>> nRoot(-100,2,1,0.001)
Root is not a real number when N is negative and n is even

>>> nRoot(10000,4,1,0)
Tolerance must be a positive value
```

PART B

Create a Python script that prompts the user for the number, N; the root, n an initial estimate, Est; and a tolerance for the error. Your script should check for invalid values and continue to prompt the user until valid values are entered. The script should then call the function and print both the estimate and the number of iterations required. Test your script to make sure it is working properly.

Save your program as **ACT_Python_4p4_ *UCusername*.py**