Do the following steps.

When done, submit each of your completed **.py** files in the same upload to the **L4 Assignment** tool on Canvas.

**[L4-1]** (**mypower.py**) Write a program that reads **int** values **n** and **r**, then prints out **n\*\*r**. Do this by defining and calling a function **power(base,exp)** which computes **base\*\*exp** using the accumulator variable **result** and the approach of **[L2-3]** as follows:

**result = 1**

**for counter in range (exp):**

**result = result \* base**  
**# return result, now having value base\*\*exp**

After calling your function **power(n,r)** and assigning its returned value to **returned** (**returned = power(n,r)**), print out the returned **result**, formatted like this example for **n==5** and **r==2**:

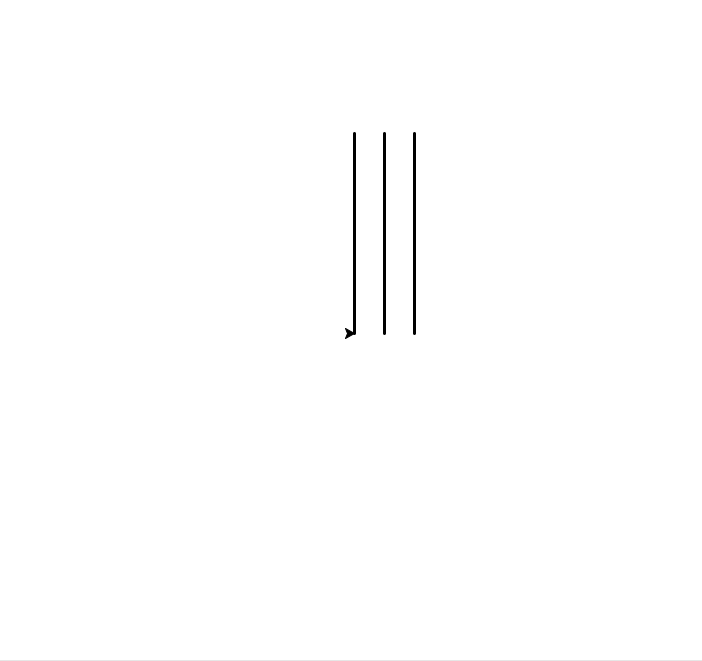
**5\*\*2 is 25**

Do this using this version of the **print()** function: **print(n,'\*\*',r,' is ',returned,sep='')**

Note how we customize **print**: by changing the default separator between **print()** arguments. Adding the argument **sep='** results in **print()** printing its arguments with no spaces (**''** is empty string) between them. If no **sep** is specified, the default separator between **print()** arguments is a single space **s: s = ' '**.

**[L4-2]** (**test\_mypower.py**) Write PyTest unit tests for your **power()** function of the previous problem. We will walk through in class how to set up PyCharm to run PyTest. It's also described in the Reading Lecture Videos for this class. You should test your function with at least ten (10) different unit test functions (PyTest requires test function names that begin with **test\_**...), each with an appropriate **assert**.

However... does your function give the correct results for all **power(n,r)**, with **n** and **r** integers? It doesn't - but don't worry about this now. For now, make sure you choose test values for **n** and **r** so that all of your tests do pass.

**[L4-3]** (**threelines2.py**) Write a program that draws the picture to the right (the three lines, but not the surrounding box). Each line is of length 200 and separated from the next line by 30 units. The bottom end of the leftmost line is at (0,0) (the turtle's home location), and all lines are of width 3.

Do this by defining a function **def draw\_line(a\_turtle)** that draws a single line of length 200 with the passed turtle reference **a\_turtle**. In the body of this function, start at the current turtle location, drop the pen, move forward 200 units, then raise the pen and return the turtle to its starting position. This restores the initial turtle state (location & heading) to what it was when you called the function. However, it leaves the pen up - which may not have been the original pen state. Don't worry about this for now.

Test your function by calling it after creating the turtle and screen, passing in your created **Turtle** reference: it should draw the single line of length 200 in the direction of the original turtle heading. It should also return the turtle to its original position when called; don't assume this position is necessarily **(0,0)**. And don't forget to invoke **exitonclick()** against your screen reference at the end of your program!

Then use your **draw\_line(a\_turtle)** function as part of drawing the given picture. Call it three times to draw each line; note that you'll need to turn and move the turtle between calls. At the end, restore the turtle to its original location and heading.

Once you have this working, put all of your drawing code into another function named **draw\_3\_lines(the\_turtle)**, so that when you call it, it draws the desired picture. Be sure that you restore the original turtle state (heading and location) when your function returns.

Finally, define a **main()** function that creates the turtle **larry** and screen, then calls **draw\_3\_lines(larry)**. It should draw the three-line picture shown above. Then call your **main()** method at the very end of your program.

**[L4-4]** (**threelines3.py**) Modify your **[L4-3]** function **draw\_line(a\_turtle)** and rename it to **draw\_line\_2(a\_turtle,length)**. This modification should have the turtle **a\_turtle** draw a line of length **length**, restoring the turtle location and position and leaving the pen up before returning.   
  
Then modify your **[L4-3]** function **draw\_3\_lines(the\_turtle)** to become **draw\_3\_lines\_2(the\_turtle,height,gap)**. When called, it should generalize drawing the 3-lines pattern: the length of each line should be **height**, and the distance between lines should be **gap**.   
  
Have your **draw\_3\_lines\_2** function body call **draw\_line\_2(the\_turtle,height)** to draw each line. As before, be sure to leave the turtle **the\_turtle**'s location and position unchanged from that at the time of call, and leave the pen up when you return.

Then define a **main()** method that reads line length **h** and line gap **g** from the user, creates a turtle **moe** then calls **draw\_3\_lines\_2(moe,h,g)**. Verify that you can draw different 3-line patterns of different sizes by varying your inputs.

What you've done here is to generalize the solution in **threelines2.py**, so that your new function can draw the three lines pattern but with lines of different lengths and different gaps. A powerful programming idea, made possible with parameterized functions!