

**FUNctions**

Remember to put your code in the unit 7 repository!

**Part 1 - Add the Numbers**

1. Using IDLE3, make a **new file** called **"*fun1.py*"** and copy the **block** of code ↓ into your file. Remember to save!

|  |
| --- |
| total = 0 for number in range(1, 10 + 1):  print(number)  total = total + number print(total) |

2. Run the file. It should print the **first 10 numbers**, and another number: **55** (Where did 55 come from)

**Question** - What did this block of code do? (Hint: look at the name of this part of the lab)

Now, make this block of code a comment.

3. Write a function called ***add\_numbers***which will run the block of code we just saw. Like this:

|  |
| --- |
| def add\_numbers():  #write the body of this  #function, similar to the block  #of code we just saw. Hint:  #don’t forget to use return  answer = add\_numbers() print(answer) |

Make sure to add the last two lines! If you run the program, it should still print **55**.

**Question** - What did we change?

4. Jamie needs help with his math homework. He needs to do a long calculation, but he forgot his **calculator**. He needs to find this answer:

we"re going to help him by **changing** our **add\_numbers** function to have **two arguments**:

1. **start** (int) - the number we will start from
2. **end** (int) - the number we will end with

The **add\_numbers** function should return the value:

Your function should look like the block of code below.

|  |
| --- |
| def add\_numbers(start, end):  #this function should add  #all the numbers from start  #to end, including both  #copy your code from above and edit it  test1 = add\_numbers(1,2) print(test1) test2 = add\_numbers(1, 100) print(test2) test3 = add\_numbers(1000, 5000) print(test3) |

To help you check if it works, use the three tests in the block of code below. When you run the program, it should output:

* **3**
* **5050**
* **12003000**

When you know your function works, help Jamie solve his question. What is the output when you run this in the shell:

add\_numbers(333, 777)

Make sure to show a TA or instructor before you move on.

**Part 1 - Bonus (Go to Part 2 first)**

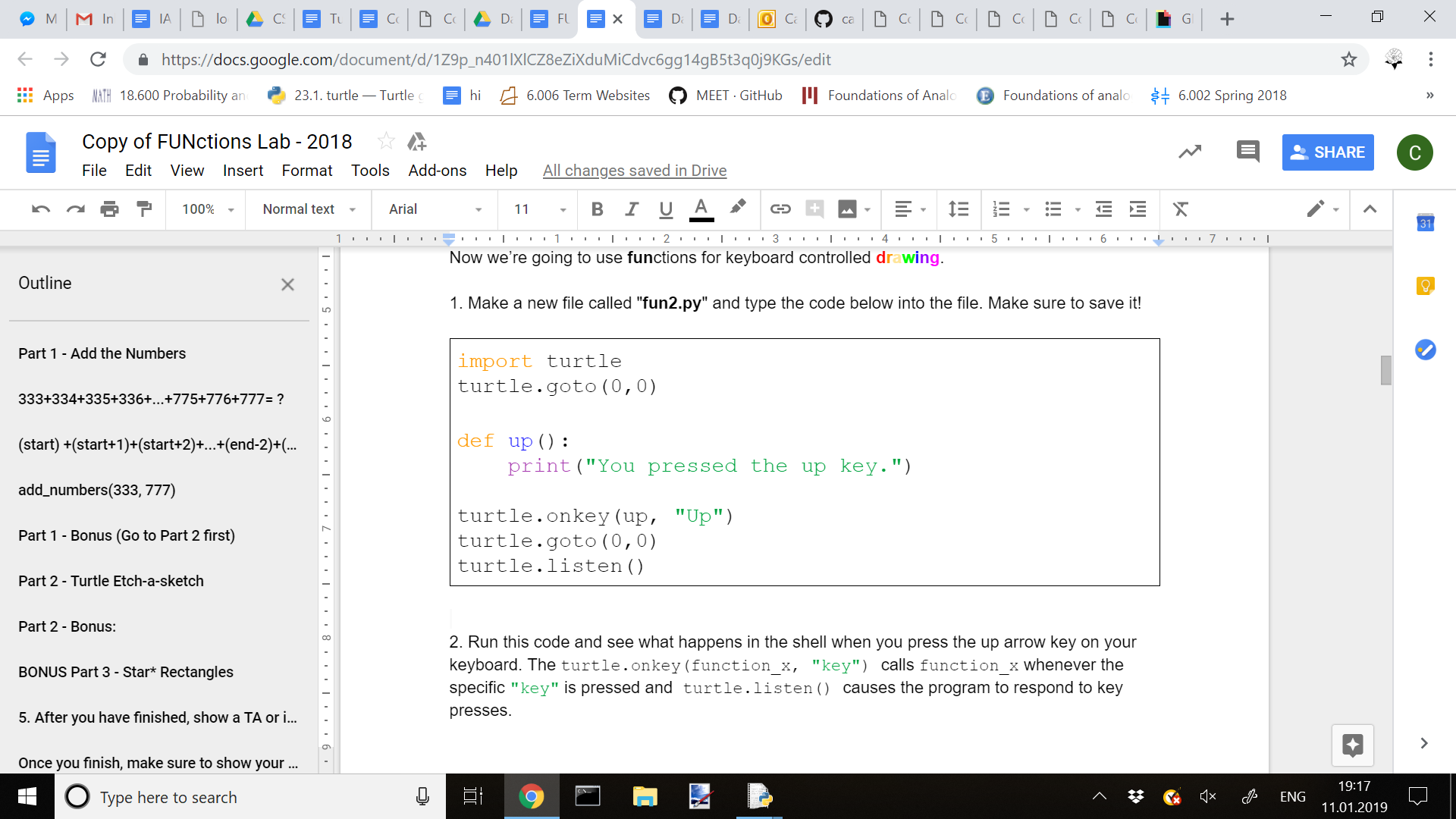
What if we want to add numbers from *a* to *b*that don’t go up by 1? What if we want them to go up by 2 or 3? What if we want them to go up by *n*?

**Part 2 - Turtle Etch-a-sketch**

Congratulations! You finished the first part of the lab, and you are awesome!

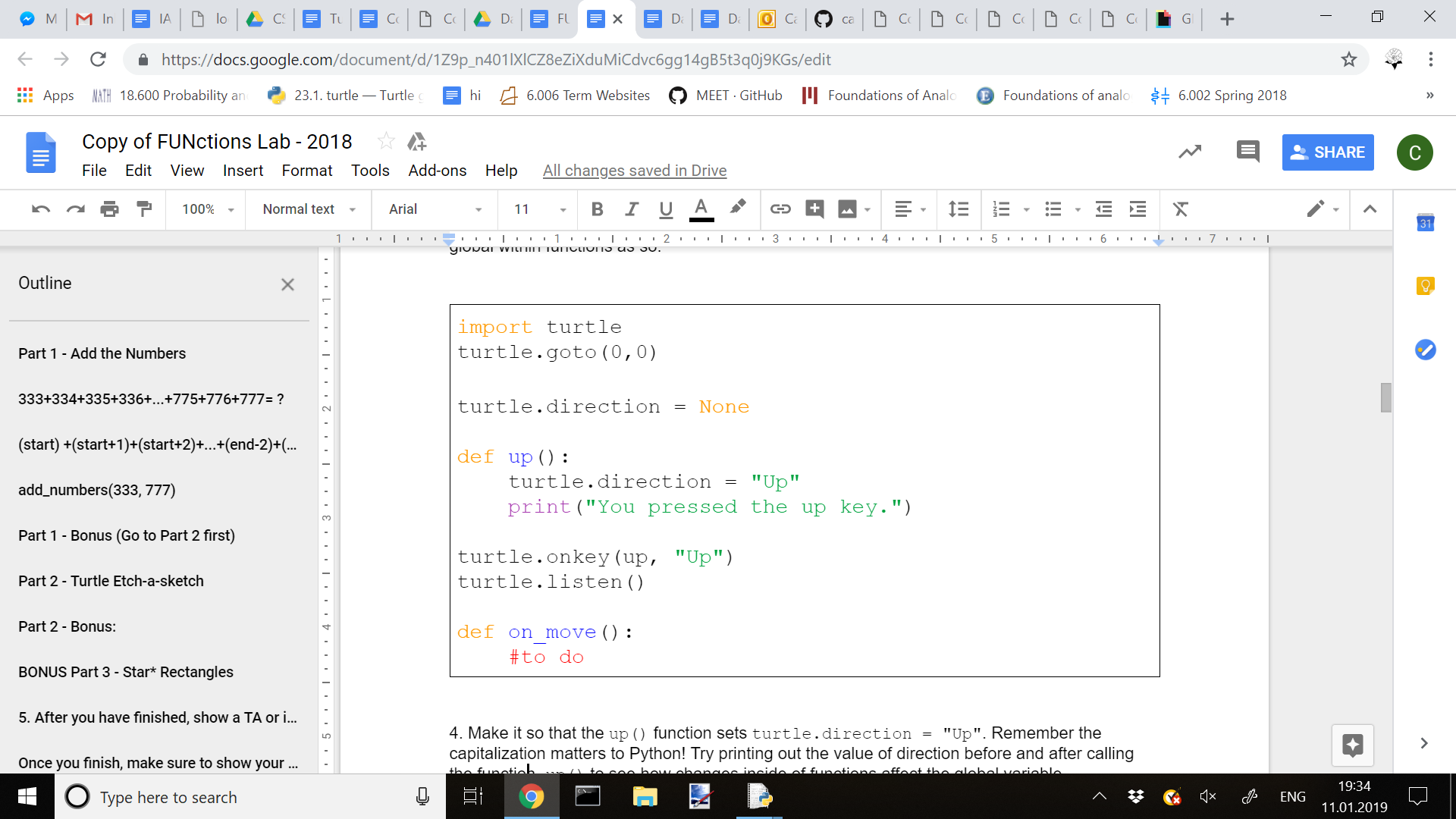
Now we’re going to use **fun**ctions for keyboard controlled **drawing**.

1. Make a new file called "**fun2.py**" and **type** the code below into the file. Make sure to save it!



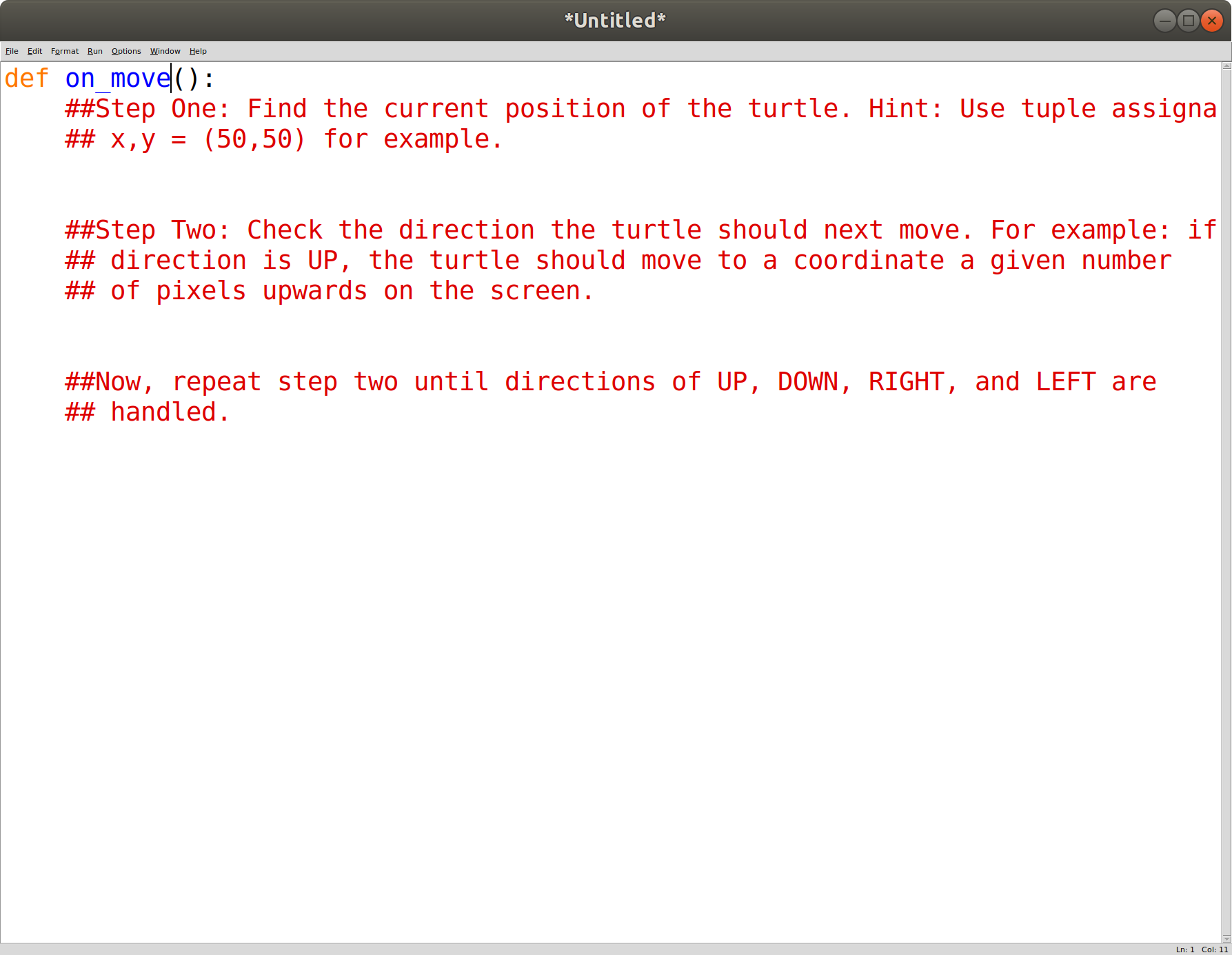
2. Run this code and see what happens in the shell when you press the up arrow key on your keyboard. The turtle.onkey(function\_x, "key") calls function\_x whenever the specific "key" is pressed and turtle.listen() causes the program to respond to key presses.

3. Now this isn’t the most interesting function, but it’s a good starting point. For the next step we’ll implement a direction. First we’ll declare the variable in the main program, then mark it as global within functions as so:



4. Make it so that the up() function sets turtle.direction = "Up". Remember the capitalization matters to Python! Try printing out the value of direction before and after calling the function up() to see how changes inside of functions affect the global variable.

6. Now we will fill in the on\_move() function. In order to move, we’ll want to assign determine how much to move the turtle horizontally and vertically. (Hint, you can check if turtle.direction == "Up": )



7. Then you can get the current x and y coordinates of the turtle using turtle.position(). turtle.position() returns a tuple (x,y). From there, you can calculate the new x and y coordinates and pass them in as arguments to turtle.goto(x\_new,y\_new)

8. Change the up function so that after it changes the global variable direction, it calls on\_move()

9. Add in helper functions for down, left, and right. Also, make sure to add in turtle.onkey()calls for each one.

10. Test out your turtle functions by drawing/writing something fun

**Part 2 - Bonus:**

Add in a feature so that pressing space bar causes the pen to be turned on/off. Handy functions:

* turtle.onkey(space, "space")
  + Calls the function space()
* turtle.isdown()
  + Returns a boolean - True if the pen is down
* turtle.up()
  + Turns off pen
* turtle.down()
  + Turns on pen

**BONUS Part 3 - Star\* Rectangles**

Congratulations! You finished the first part of the lab, and you are awesome!

Now we"re going to use **fun**ctions for more **drawing**.

1. Make a new file called "**fun3.py**" and type the code below into the file. Make sure to save it!

|  |
| --- |
| print("\*")  print("\*")  print("\*")  print("\*")  print("\*") |

2. Run the program. It should print 5 stars above each other, like this:

|  |
| --- |
| \*  \*  \*  \*  \* |

3. Now, think about how we can print the stars **next** to each other. Once you try it out, the output should look like this

|  |
| --- |
| \*\*\*\*\* |

4. Cool! At this point, write a function called **draw\_1d**, which will print the stars next to each other in one line. However, it should take in one argument: ***n***. This argument will be the number of stars the function **draw\_1d** will draw. For example:

|  |
| --- |
| >>> draw\_1d(3) \*\*\* >>> draw\_1d(8) \*\*\*\*\*\*\*\* |

**5. After you have finished, show a TA or instructor.**

6. The \* symbol can get very boring, what if we want to use a different one? You should now change your **draw\_1d** function to include another argument: **char**, which will be a string containing one character.

The function should print a line of char, and char should appear **n** times. **draw\_1d** should work like this.

|  |
| --- |
| >>> c1 = "~" >>> c2 = "!" >>> draw\_1d(12, c1) ~~~~~~~~~~~~ >>> draw\_1d(6, c2) !!!!!! >>> draw\_1d(24, "+") ++++++++++++++++++++++++ |

**Once you finish, make sure to show your work to a TA or instructor before moving on.**

7. Think about how you can draw a **rectangle** using the \* symbol. The output should look like this:

|  |
| --- |
| \*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* |

This was a **5×10** rectangle (think about why). Now try making **6×8** and **3×24** rectangles.

8. Make a function called **draw\_2d** that takes in **three** arguments:

1. **n** (int) - the number of rows the rectangle has
2. **m** (int) - the number of columns the rectangle has
3. **char** (string) - the symbol which the rectangle will be made of

**draw\_2d** should print an **n×m** rectangle made of the symbol **char**.

Hint 1: use your **draw\_1d** function inside your **draw\_2d** function when writing it.

Hint 2: use a for loop

Your function should look like this:

|  |
| --- |
| def draw\_2d(n, m, char1):  # draw\_2d should print out  # a rectangle of size n × m   # the rectangle must be made  # of the symbol in char  # Hint 1: use your draw\_1d function   # when writing this function   # Hint 2: use a for loop |

After you finish writing the function, run the code below and check that it correctly prints what you expect. Your code should look something like this:

|  |
| --- |
| >>> draw\_2d(3,23, "\*") \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* >>> draw\_2d(7,3, "x") xxx xxx xxx xxx xxx xxx xxx |

Congratulations you awesome person! You’re really smart and just finished the lab. You helped Farah with her math homework and she is very happy.

**Part 3 - Bonus**

We can draw shapes with a **border**. There are now **two** symbols in the rectangle, we will call them **border symbol** and **fill symbol**. For example, in the box below, @ is the border symbol, and x is the fill symbol.

|  |
| --- |
| @@@@@@@@@@@@  @xxxxxxxxxx@  @xxxxxxxxxx@  @xxxxxxxxxx@  @@@@@@@@@@@@ |

Write a function called **special\_draw\_2d** that takes in **four arguments**:

1. **n** (int) - the number of rows the rectangle has
2. **m** (int) - the number of columns the rectangle has
3. **border** (string) - the symbol along the rectangle"s border - the **border symbol**
4. **fill** (string) - the symbol inside the rectangle - the fill symbol

**special\_draw\_2d** must print an **n×m** rectangle with a border of the character stored in the **border** variable, and a filling of the character stored in the **fill** border. Your function must be able to do the following:

|  |
| --- |
| >>> special\_draw\_2d(5,12,"x","-") xxxxxxxxxxxx x----------x x----------x x----------x xxxxxxxxxxxx >>> special\_draw\_2d(7,24,"8",".") 888888888888888888888888 8......................8 8......................8 8......................8 8......................8 8......................8 888888888888888888888888 |

If you finish and still want more, you can move on to the challenge. You"re awesome for getting this far!

**Challenge - Fibonacci & Factorial**

**Challenge - Fibonacci**

The fibonacci are special numbers that are everywhere in nature, and they are very interesting to a lot of different people. These are the first few Fibonacci numbers:

What’s the pattern? Before moving forward, check your answer with a TA or instructor.

What you have to do is write a function called **fib**, which takes in only one argument:

* **n** (int) - the position of a Fibonacci number (for example, **fib(3)** would return 2, because 2 is the 3rd Fibonacci number)

**fib** should **return** the **nth** Fibonacci number.

Your **fib** function should give these results:

|  |
| --- |
| >>> fib(5) 5 >>> fib(10) 55 >>> fib(15) 610 >>> fib(20) 6765 |

After you have finished and made sure your answers are similar to the ones in the box of code above, show an instructor or TA.

**Challenge - Factorial**

Woohoo, you’re almost done with the challenge, you’re super duper awesome! **Factorial** (!) is something we use in math to **count things** (see end of problem if you want a better understanding of factorial). These are the factorials of a few numbers. Can you find the pattern?

Question: what is the value of n! for some number n?

When you find the pattern and answer this question, check with a TA or instructor.

After you do that, you should write a function called **fact** that takes in only one argument:

* **k** (int) - a number which **fact** will return the factorial of

**fact** should **return** the value k! (k factorial, also called the factorial of k).

Your **fact** function must give these results:

|  |
| --- |
| >>> fact(5) 120 >>> fact(7) 5040 >>> fact(9) 362880 >>> fact(11) 39916800 |

Congratulations! You are an awesome super duper cool coder!

(Optional)

Why do we care about factorials? ***n!* represents how many ways we can order *n* objects.** For example, say we wanted to know how many different ways 4 turtles could finish in a race. There are 4 different turtles that could come in first place.

For each of these options, there are 3 remaining turtles that could come in second. So far we have 4\*3 ways of choosing the turtles that come in first and second.

For each of these 4\*3 options, there are 2 remaining turtles that could come in third, so we have 4\*3\*2 ways to pick the turtles that come in first, second, and third.

For each of these options there is 1 remaining turtle and 1 remaining position, so in the end, there are 4\*3\*2\*1=4! Ways for the 4 turtles to finish.

# SAVE YOUR CODE USING ENDLAB!!!

# [LINK TO SURVEY!!!](https://goo.gl/forms/xkg9DLi4c7jpLEOm1)