FUNctions

Remember to **`startlab`** and follow the directions. We are working on module 5!

Remember to **`endlab`** and follow the directions when you are done.

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!

```
c = 0
for number in range(1, 100 + 1):
    print(number)
    c = c + number
print(c)
```

2. Run the file. It should print the **first 100 numbers**, and another number: **5050**

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 **5050**.
Question - What did we change?

4. Farah needs help with her Math homework, and we want to help her! She needs to do a long calculation, but she forgot her **calculator**. She needs to find this answer:

$$333 + 334 + 335 + 336 + ... + 775 + 776 + 777 = \ ?$$

we're going to help her 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:

$$(start) + (start + 1) + (start + 2) + ... + (end - 2) + (end - 1) + (end)$$

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
    # write your code here

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 Farah solve her 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 - Star* Rectangles

Congratulations! You finished the first part of the lab, and you are awesome!
Now we're going to use **fun**ctions for **drawing**.

1. Make a new file called "**fun2.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 **6×10** rectangle (think about why). Now try making **5×12** and **3×24** rectangles.

8. Make a function called **draw_2d** that takes in **three** arguments:
    a. **n** (int) - the number of rows the rectangle has
    b. **m** (int) - the number of columns the rectangle has
    c. **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 2 - 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:

$$1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ 89,\ 144,\ 233,\ 377,\ ...$$

You can see that the 1st fibonacci number is 1, the 2nd is 1, the 3rd is 2, and so on. 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**. These are the factorials of a few numbers. Can you find the pattern?

$$1! = 1$$
$$2! = 1 \times 2 = 2$$
$$3! = 1 \times 2 \times 3 = 6$$
$$4! = 1 \times 2 \times 3 \times 4 = 24$$
$$5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$$
$$6! = 1 \times 2 \times 3 \times 4 \times 5 \times 6 = 720$$

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!