

Hack110 Interest Form!

When? Saturday, November 8th from 10 AM - 12 AM (Midnight)

Where? In Sitterson Lower Lobby

Who can join? Anyone in COMP 110! No prior experience required. Bring a partner, find one there, or go solo!

Come for a fun day of coding, workshops and events (also food and CLE credit will be provided):

- Choose between web development or game development track
- Go to various workshops & events such as: Navigating the CS Major, Resume workshop, ice cream station, and kahoot trivia and MORE!
- Link: [Interest Form Here!](#) Or via the QR code
- Interest form will close Sunday, Sept 21st at 11:59 pm
 - Fill out this form to get **priority notice** of when we release the sign-up form.



Interest Form!





Positional Arguments & Recursion Practice

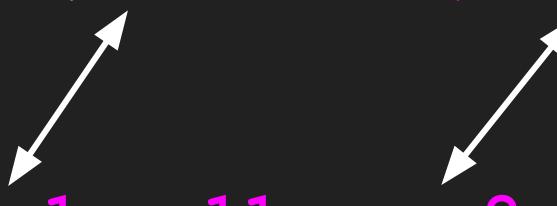
Reminders

- **Quiz 00:** Regrade requests will be open till 11:59pm tonight!
 - Please submit a regrade request if you believe your quiz was not graded correctly according to the rubric
- **EX02: Wordle** due Tuesday, Sep 16 at 11:59pm

Want extra support? We're here and *want* to help!

Recall: Signature vs Call

```
def divide(num1: int, num2: int) -> float:  
    divide(num1 = 11, num2 = 3)
```



The diagram illustrates the mapping between parameters and their values. Two white arrows point from the parameter names 'num1' and 'num2' in the function definition to their respective values '11' and '3' in the call expression. A curly brace groups the two parameter-value pairs.

These are called **keyword arguments**, since you are assigning values based on the parameter names.

Keyword arguments

```
def divide(num1: int, num2: int) -> float:
```

```
divide(num1 = 11, num2 = 3)
```

Benefit of keyword arguments:
order of arguments doesn't
matter.

Keyword arguments

```
def divide(num1: int, num2: int) -> float:
```

```
divide(num1 = 11, num2 = 3)
```

```
divide(num2 = 3, num1 = 11)
```

Benefit of keyword arguments:
order of arguments doesn't
matter.

Positional Arguments

```
def divide(num1: int, num2: int) -> float:
```

```
divide(11, 3)
```

For **positional arguments**, values are assigned based on the order (*position*) of the arguments.

Review: Checklist for developing a recursive function:

Base case:

- Does the function have a clear base case?
 - Ensure the base case returns a result directly (without calling the function again).
- Will the base case *always* be reached?

Recursive case:

- Ensure the function moves closer to the base case with each recursive call.
- Combine returned results from recursive calls where necessary.
- Test the function with edge cases (e.g., empty inputs, smallest and largest valid inputs, etc.). Does the function account for these cases?

factorial Algorithm

Create a recursive function called **factorial** that will calculate the product of all positive integers less than or equal to an int, **n**. E.g.,

factorial(n=5) would return: $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$

factorial(n=2) would return: $2 \cdot 1 = 2$

factorial(n=1) would return: $1 = 1$

factorial(n=0) would return: 1

Conceptually, what will our **base case** be?

What will our **recursive case** be?

What is an **edge case** for this function? How could we account for it?

Visualizing recursive calls to `factorial`

Visualizing recursive calls to factorial

```
factorial(n = 4)
```

```
return n * factorial(n - 1)  
return 4 * factorial( 3 )  
return 4 * 6  
return 24
```

```
return n * factorial(n - 1)  
return 3 * factorial( 2 )  
return 3 * 2  
return 6
```

```
return n * factorial(n - 1)  
return 2 * factorial( 1 )  
return 2 * 1  
return 2
```

```
return 1
```

Let's write the **factorial** function in VS Code!



Memory diagram

```
1 # Factorial
2 def factorial(n: int) -> int:
3     """Calculates factorial of int n."""
4     # Base case
5     if n == 0 or n == 1:
6         return 1
7     # Recursive case
8     else:
9         return n * factorial(n - 1)
10
11 # Example usage
12 print(factorial(3))
```

Hand-writing code: An adaptation of **fizzbuzz**

A group of students start counting up from 1, taking turns saying either a number or a phrase.

If their number is divisible by 3, the student says “fizz” rather than the number.

If their number is divisible by 5, they say “buzz” rather than the number.

If their number is divisible by both 3 and 5, they say “fizzbuzz”

Example:

1, 2, fizz, 4, buzz, fizz, 7, 8, fizz, buzz, 11, fizz, 13, 14, fizzbuzz, 16, ...

Hand-writing code: An adaptation of **fizzbuzz**

Our function definition should meet the following specifications:

- The function should be named **fizzbuzz**, have one **int** parameter named **n**, and return a **str**
- If **n** is divisible by 3 and not 5, the function should print "fizz"
- If **n** is divisible by 5 and not 3, the function should print "buzz"
- If **n** is divisible by 3 AND 5, the function should print "fizzbuzz"
- If **n** is not divisible by 3 OR 5, the function should print **n** as a string
- The function should keep calling itself, increasing the argument by 1 each time, until we finally reach a “fizzbuzz” number, when we’ll return “fizzbuzz”
- Explicitly type your parameter and return type.