Does your bestie (or enemy) need to take COMP110?

I still have seats available in my Summer Session I sections!

This is a great opportunity to:

- Take COMP110 in a small class environment
- Focus on COMP110 without having to balance other classes
- Not have to sit on the waitlist for COMP110 in a future semester

COMP110

CL29 – 🪄Magic Methods🪄 and Recursion Review

Announcements

Re: Quiz 03:

- *Regrade requests will be open **till 11:59pm on Wednesday!***
  - Please submit a regrade request if you believe your quiz was not graded correctly according to the rubric

Re: Quiz 04:

- Practice quiz will be available on the site today
  - Come to Tutoring to work through it with TAs today (5-7pm in SN011)!
- If you have a UAA and want to reschedule your quiz to another date, please let me know!

(Reminder: The River Simulation exercise is *due Friday at 11:59pm*. Start early!)

Warmup: Try printing a Point or Line object from last lecture!

```python
class Point:
    x: float
    y: float

    def __init__(self, x: float, y: float):
        self.x = x
        self.y = y

    def dist_from_origin(self) -> float:
        return (self.x**2 + self.y**2) ** 0.5

    def translate_x(self, dx: float) -> None:
        self.x += dx

    def translate_y(self, dy: float) -> None:
        self.y += dy

pt: Point = Point(2.0, 1.0)
```

What happens?

Warmup: Try printing a Point or Line object from last lecture!

```
class Point:
    x: float
    y: float

    def __init__(self, x: float, y: float):
        self.x = x
        self.y = y

    def dist_from_origin(self) -> float:
        return (self.x**2 + self.y**2) ** 0.5

    def translate_x(self, dx: float) -> None:
        self.x += dx

    def translate_y(self, dy: float) -> None:
        self.y += dy

pt: Point = Point(2.0, 1.0)
```

What happens?

Let’s implement some magic in VS Code!

Shifting gears… remember recursion?

Recall these functions: what was the issue with the icarus function?

```python
def icarus(x: int) -> int:
    """Unbound aspirations!"""
    print(f"Height: {x}")
    return icarus(x=x + 1)

def safe_icarus(x: int) -> int:
    """Bound aspirations!"""
    if x >= 2:
        return 1
    else:
        return 1 + safe_icarus(x=x + 1)

print(safe_icarus(x=0))
```

The dreaded **`Recursion Error`**!

Stack Overflow and Recursion Errors

When a programmer writes a function that calls itself indefinitely (*infinitely*), the **function call stack** will *overflow…*

This leads to a **`Stack Overflow`** or **`Recursion Error`**:

```
RecursionError: maximum recursion depth exceeded while
calling a Python object
```

Recursive function checklist:

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:

- ❏ Does the function have a recursive case that *progresses toward the base case*?
  - ❏ Does the recursive call have the right arguments? The function should call itself on a simpler or smaller version of the problem.
- ❏ Have you tested your function with multiple cases, including edge cases?

Another example of recursion: factorial!

To calculate the factorial of an int, `n`, we would multiply `n` by `(n-1)`, then `(n-2)`, and so on, until we reach `1`.

For instance, to calculate `5!`, we would do: `5 * 4 * 3 * 2 * 1`, which would evaluate to `120`.

```python
def factorial(n: int) -> int:
    # Base case: factorial of 0 or 1 is 1
    if n <= 1:
        return 1
    # Recursive case: n! = n × (n-1)!
    return n * factorial(n - 1)
```

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
```

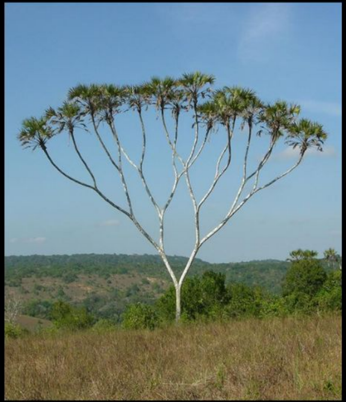
```
return n * factorial(n - 1)
return 2 * factorial(  1  )
return 2 * 1
return 2
```

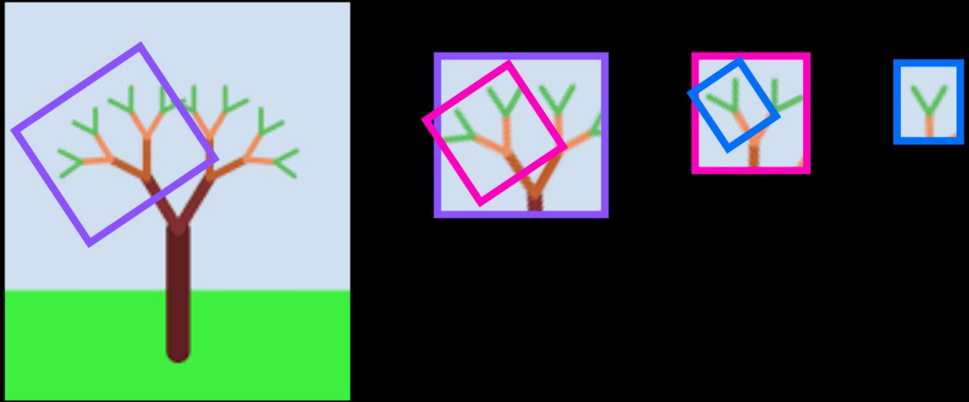
```
return 1
```

Recursion: defining an operation/object in terms of itself

A real-world phenomenon! Examples:

- **You** have **parents**, who have **parents**, who have **parents**, who have **parents**, who…
  … were the **first humans**
- A **tree** has **branches**, which have **branches**, which have **branches**, which…
  … have **leaves**

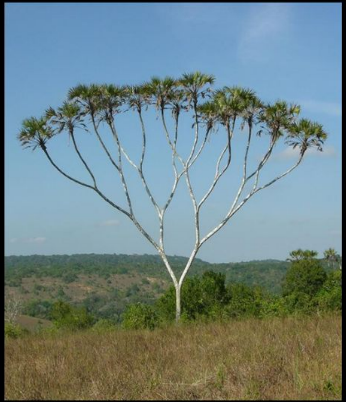


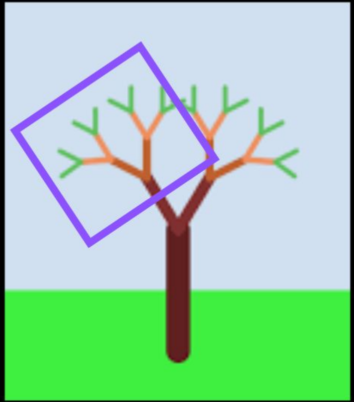


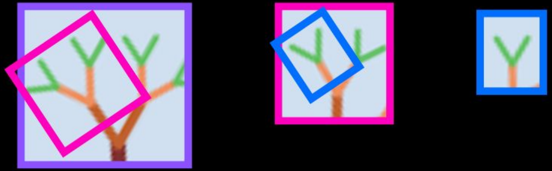
Recursion: defining an operation/object in terms of itself

A real-world phenomenon! Examples:

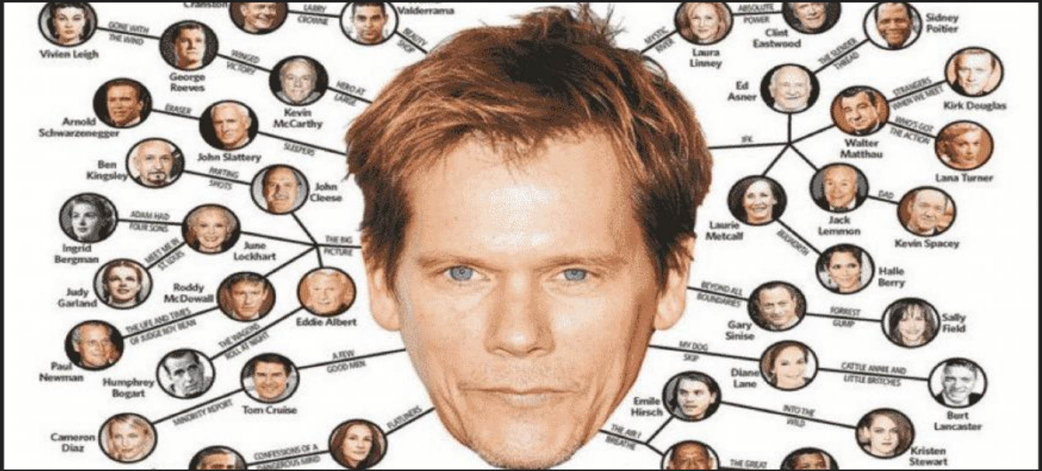
- **You** have **parents**, who have **parents**, who have **parents**, who have **parents**, who…
  … were the **first humans**
- A **tree** has **branches**, which have **branches**, which have **branches**, which…
  … have **leaves**





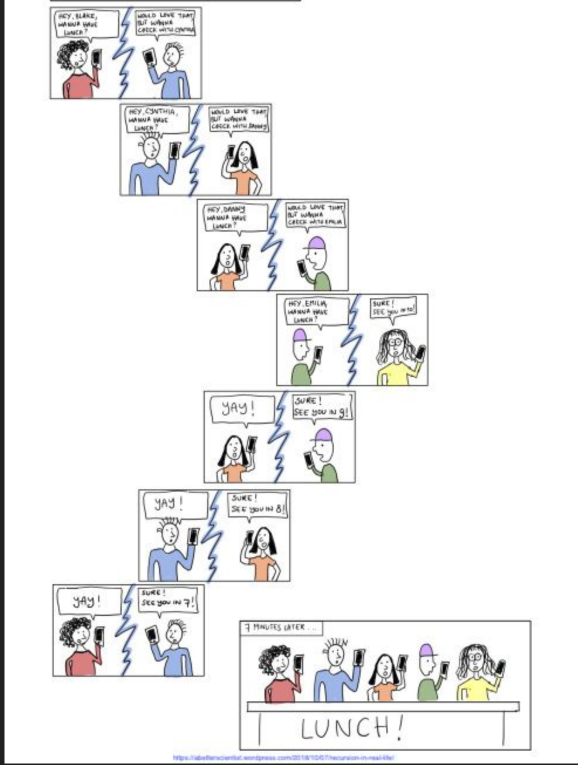


Different recursive structures for different purposes



Six degrees of Kevin Bacon

graph/network



Coordinating plans before 3-way calls were possible

linked list

`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*4*3*2*1 = **120**

`factorial(n=2)` would return: 2*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) returns 4 * 6 = 24

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

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

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

                                     return 1
```

Let’s write the **`factorial`** function in VS Code!

Memory diagram

```python
# Factorial
def factorial(n: int) -> int:
    """Calculates factorial of int n."""
    # Base case
    if n == 0 or n == 1:
        return 1
    # Recursive case
    else:
        return n * factorial(n - 1)

# Example usage
print(factorial(3))
```

```python
"""Mysterious 'rev' from source (src) to destination (dest)!"""


def rev(src: str, i: int, dest: str) -> str:
    """You happen upon a magical lil function..."""
    if i >= len(src):
        return dest
    else:
        return rev(src=src, i=i + 1, dest=src[i] + dest)


print(rev(src="lwo", i=0, dest=""))
```

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?