

# Warmup (L22)

Finish the class `Rational`:

```
import math # math.gcd is greatest common divisor

class Rational:
    """
    Represents a rational number by its (integer) numerator and
    denominator, reduced.
    """
    num: int
    den: int

    def __init__(self, num: int, den: int) -> None:
        gcd = math.gcd(num, den)
        self.num = num // gcd
        self.den = den // gcd

    def __repr__(self) -> str:
        return f"{self.num}/{self.den}"

    def __add__(self, y):
        ...
    def __sub__(self, y):
        ...
    def __mul__(self, y):
        ...
    def __truediv__(self, y):
        ...
    def toFloat(self) -> float:
        ...
```

# Recursion

---

CS114 M10

# When Python gets angry

- Many of you have seen Python get stuck and run forever (or for a while then crash)
- One common mistake that causes this:

```
def silly(x: int) -> int:  
    return silly(x-1)
```

- This gets stuck because silly keeps calling itself forever, and can never get out

# Recursion!

---

- A function calling itself *is* allowed...
- ... but it has to be conditional, or it'll go on forever
- This technique is called *recursion*
- Recursion is no more powerful than looping, but sometimes it's clearer to express something with recursion

# Recursion!

---

- No new syntax for recursion
- This is a new technique, but it's just function calls as we've already seen them

# First example: factorial

---

- Classic example of recursion: factorial
  - E.g.,  $5! = 5 * 4 * 3 * 2 * 1$

# First example: factorial

---

- Classic example of recursion: factorial
  - E.g.,  $5! = 5 * 4 * 3 * 2 * 1$


```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
```

# First example: factorial

---

*Base case:* If  $n$  is 1 (or less), `factorial` does not call itself (does not recurse). This is important, because this is how `factorial` stops!

```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
```



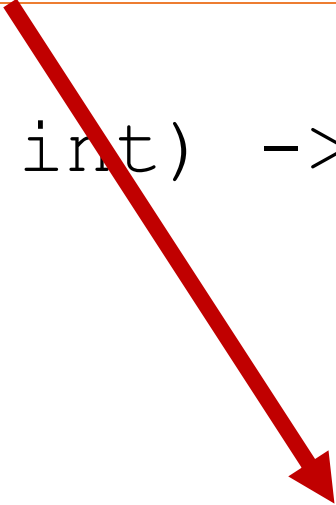


# First example: factorial

---

*Recursive case:* We build our definition of factorial( $n$ ) for any value of  $n$  greater than 1 by calling factorial with a smaller  $n$

```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
```



# Understanding recursion

---

- Let's add some prints to see where our code goes

```
def factorial(n: int) -> int:
    print("factorial started ({n})")
    if n <= 1:
        print(f"base case {n}")
        return 1
    else:
        print(f"recursive case {n} started")
        r = n * factorial(n-1)
        print(f"recursive case {n} finished")
        return r
```

# Recursion vs. looping

---

- Recursion is no more powerful than looping! Here's factorial with loops (we developed this in Module 3):

```
def factorial(n: int) -> int:
    r = 1
    for i in range(n, 1, -1):
        r *= i
    return r
```

# Recursion concepts

---

CS114 (M10)

# Recursion in math

---

- Functions are often described recursively in math, not just programming
- Here's how a mathematician might describe factorial:

$$n! = \begin{cases} 1, & \text{if } n \leq 1 \\ (n - 1)! \cdot n, & \text{if } n > 1 \end{cases}$$

# Recursion concepts

---

- For a recursive function to be meaningful (in math or programming), it needs
  - At least one *base case*: Some case(s) where the result does not depend on a recursive call
  - At least one *recursive case*: Some case(s) where the result does depend on a recursive call
  - A correctly implemented *descent condition*: Each call to the recursion gets closer to (or “approaches”) a base case

# Back to factorial

---

```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
```

- Base case:  $n \leq 1$ . There is no recursive call when  $n$  is less than or equal to 1.

# Back to factorial

---

```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
```

- Recursive case:  $n > 1$ . We didn't specifically write " $n > 1$ " in the code, but the **else** case only happens when that's true.



# Back to factorial

---

```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    return n * factorial(n-1)
```

- The recursive case doesn't need to be part of an **if** or **else**! All that matters is what actually *happens*.

# Back to factorial

---

```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
```

- The way to think about the recursive case is “Imagine we’ve already solved this for  $n-1$ . What’s the solution for  $n$  given that?”

# Back to factorial

---

```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
```

- Descent condition:  $n-1$  is smaller than  $n$ , so will eventually be less than or equal to 1.

# In-lecture quiz (L22)

---

- <https://student.cs.uwaterloo.ca/~cs114/F25/quiz/>
- Q1: Are there any conditions under which this version of `factorial` would recurse forever?

```
def factorial(n: int) -> int:
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

- A. This code crashes
- B. Yes (but it doesn't otherwise crash)
- C. No (and it doesn't crash either)

# In-lecture quiz (L22)

---

- <https://student.cs.uwaterloo.ca/~cs114/F25/quiz/>
- Q2: Are there any conditions under which this version of `factorial` would recurse forever?

```
def factorial(n: int) -> int:
    if n > 1:
        return n * factorial(n-1)
    else:
        return 1
```

- A. This code crashes
- B. Yes (but it doesn't otherwise crash)
- C. No (and it doesn't crash either)

# More recursion

---

- Another classic example of recursion (and another case that's easy with loops):  
Fibonacci sequence
  - $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$
  - $\text{fib}(1) = 1$
  - $\text{fib}(0) = 0$
- Again, easily done with loops, but we're just starting to learn recursion here!

# Fibonacci

---

- Descent condition:  $n$  gets smaller (approaches 0)
- Base case:  $n == 0$  **or**  $n == 1$
- Recursive thought process: If we already know `fib(n-1)` and `fib(n-2)`, then `fib(n)` is just their sum

# Fibonacci

---

```
def fib(n: int) -> int:
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

- When we talk about efficiency, we'll see how bad this is, but think about how many times it'll call `fib(1)` for some  $n$ .



# Fibonacci

---

```
def fib(n: int) -> int:
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

- Recursion is not inherently inefficient. But, it's easier to be accidentally inefficient than it is with loops.

# Fibonacci

---

```
def fib(n: int) -> int:
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

- We'll be talking about efficiency in loose terms in this module, but we *will* be talking about it.

# Divide and conquer

---

CS114 (M10)

# Divide and conquer

---

- Most common reason why recursion is good: *divide and conquer*
- Many problems are done best by dividing the problem (in half or otherwise into subparts), then solving each subproblem
- We'll start with search

# in

---

- How does **in** work with a list?
- It's just a loop:

```
def contains(lst: list, needle: typing.Any) -> bool:
    for val in lst:
        if val == needle:
            return True
    return False
```

- With nothing known about the list, this is the best we can do

# Sorted search

---

- If you're looking at a library shelf for a book by a specific author, you don't go one-by-one through every book
- If the list is sorted, you can make a decent guess as to where the value is
- Guesswork is human behavior, so in programming, we split the list in half and look in the appropriate half