f-Strings &
Intro to Recursion

Reminders

- **Quiz 01** on Thursday
  - Practice questions under Quiz 01 on the site!
- **LS08: Intro to Recursion** due tonight at 11:59pm

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

f-strings (formatted string literals)

A helpful way to embed expressions directly into strings!

Without f-strings:

```
print("They are " + str(30 + 1))
```

With f-strings:

```
print(f"They are {30 + 1}")
```

Both will output the string:

```
They are 31
```

f-strings (formatted string literals)

```python
def get_class(subject: str, num: int) -> None:
    print(
        "I'm currently in "
        + subject
        + str(num)
        + ", but next semester I'm taking "
        + subject
        + str(num + 100)
        + "!"
    )


get_class(subject="COMP", num=110)
```

Will these two versions of the get_class function print the exact same phrase?

```python
def get_class(subject: str, num: int) -> None:
    print(f"I'm currently in {subject}{num}, but next semester I'm taking {subject}{num+100}!")


get_class(subject="COMP", num=110)
```

```
1  """Examples of conditionals."""
2
3
4  def number_report(x: int) -> None:
5      """Print some numerical properties of x"""
6      if x % 2 == 0:
7          print("Even")
8      else:
9          print("Odd")
10
11     if x % 3 == 0:
12         print("Divisible by 3")
13
14     if x == 0:
15         print("Zero")
16     else:
17         if x > 0:
18             print("Positive")
19         else:
20             print("Negative")
21
22     print("x is " + str(x))
23
24
25 number_report(x=110)
```

How could we convert the print statement on line 22 to use an f-string?

Your job: Diagram *at least* 2 function call frames…

But stop when you get tired or run out of lead!

```
1    def icarus(x: int) -> int:
2        """Unbound aspirations!"""
3        print(f"Height: {x}")
4        return icarus(x=x + 1)
5
6
7    print(icarus(x=0))
```

Questions to discuss with your neighbor(s):

What seems *problematic* about this function?

How might you prevent it?

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


print(icarus(x=0))
```

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
```

Base Cases and Recursive Cases

The key to writing recursive functions that are non-infinite!

To avoid StackOverflow Errors and infinite recursion:

1. You must have at least one **base case**
    a. Base case: a branch in a recursively defined function that **does not recur**
2. **Recursive cases** must change the arguments of recursive calls such that they make progress toward a base case

Trace the following program in a diagram:

```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))
```

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?

So… when is recursion useful?

Recursion is useful when a problem can be defined in terms of smaller versions of itself.

Hints that recursion may be useful:

- The problem has a natural “stop” (a base case)
  - E.g., “stop when n is 0!”
- The data or process is hierarchical
  - E.g., a family tree, or computer files and folders
- The same operation repeats on smaller inputs
  - E.g, fibonacci, factorial

`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**

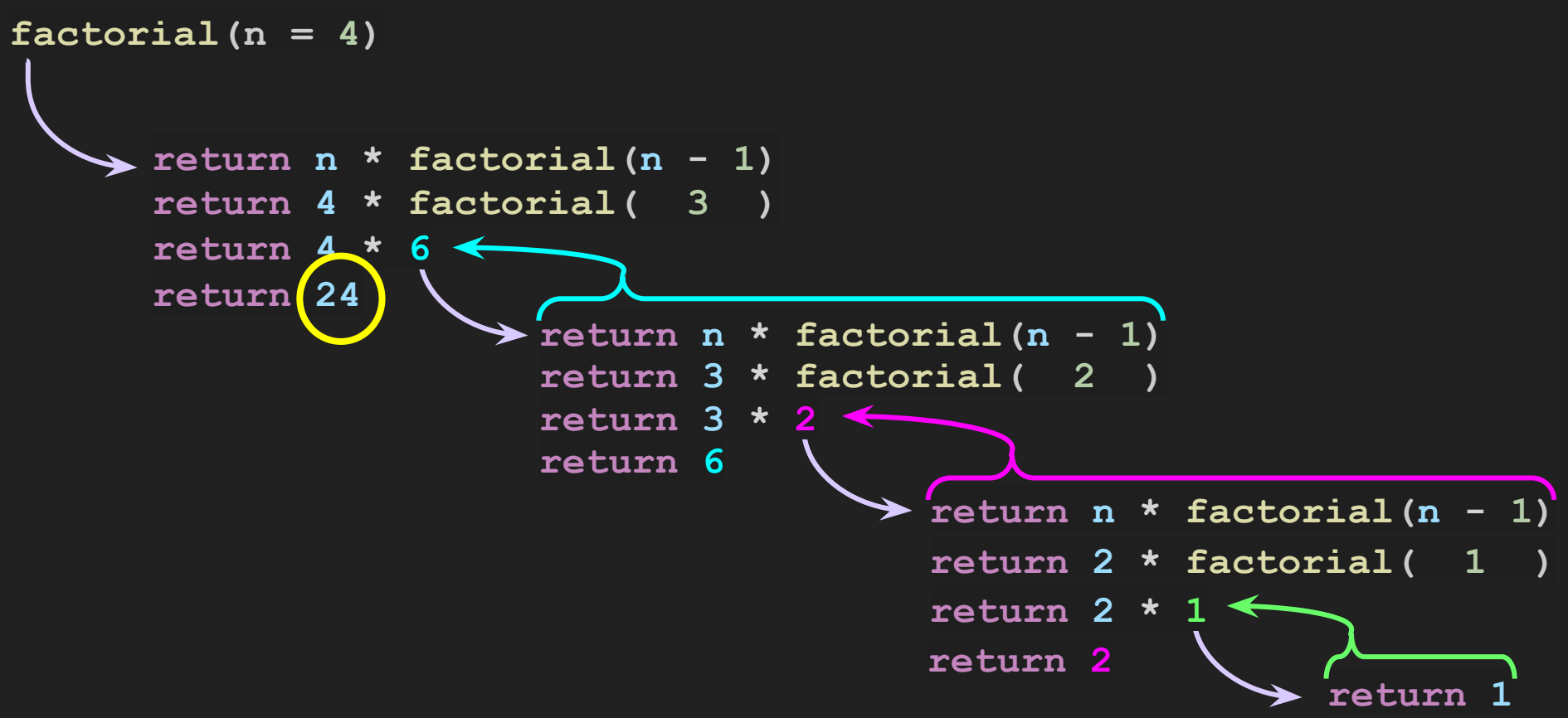
Conceptually, what will our **base case** be?

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`



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

Memory diagram (without considering the edge case, for space reasons)

```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))
```

Weekly Tutoring + Office Hours

Office Hours (Sitterson Hall (SN) 008):

- Mondays–Fridays: 11am-5pm
- Sundays: 1-5pm

Tutoring (see CSXL site for location):

- Mondays, Wednesdays, Thursdays: 5-7pm

Reminder: No class, office hours, or tutoring on Monday, Feb 9 (Wellbeing Day)!

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 an `int`
- 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`'s literal value
- The function should keep calling itself, increasing the argument by 1 each time, until we finally reach a "fizzbuzz" number, when we'll return `n`
- Explicitly type your parameter and return type.

Solution

```
def fizzbuzz(n: int) -> int:
    if n % 3 == 0 and n % 5 == 0:  # Base case
        print("fizzbuzz")
        return n
    elif n % 3 == 0:  # If n is divisible by 3 but NOT 5
        print("fizz")
    elif n % 5 == 0:  # If n is divisible by 5 but NOT 3
        print("buzz")
    else:  # If n is not divisible by 3 OR 5
        print(n)
    # If fizzbuzz wasn't reached this time, call function again with n+1
    return fizzbuzz(n=n + 1)
```