CL10 – Recursion and Positional Arguments

Reminders

Quiz 00:

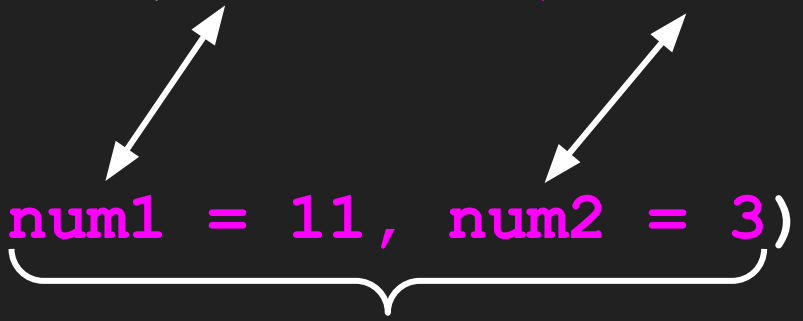
- *Regrade requests will be open **till 11:59pm tonight!***
  - Please submit a regrade request if you believe your quiz was not graded correctly according to the rubric
  - Please do not ask questions about content in regrade requests. Instead, come see us in office hours/tutoring!

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

- Visit Office Hours (11am–5pm in SN008)!
- Visit Tutoring (5–7pm in SN011 today)!

Recall: Signature vs Call

```
def sum(num1: int, num2: int) -> int:
```



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

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

Keyword arguments

```
def sum(num1: int, num2: int) -> int:
```

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

Benefit of keyword arguments: order doesn’t matter.

Keyword arguments

```
def sum(num1: int, num2: int) -> int:
```

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

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

Benefit of keyword arguments: order doesn’t matter.

Positional Arguments

```
def sum(num1: int, num2: int) -> int:
```

```
sum(11, 3)
```

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

Reviewing the memory diagram in the last lecture

```
1   def celebrate(winner: str) -> None:
2       print(f"Yay, {winner}!")
3
4
5   def get_votes(beyonce: int, kendrick: int, other: int) -> str:
6       """Find RoTY winner."""
7       if other > beyonce and other > kendrick:
8           return "Someone else!"
9       elif kendrick > beyonce:
10          return "Kendrick"
11      else:
12          return "Beyonce"
13      return "Charli"
14
15
16  celebrate(get_votes(beyonce=6000, kendrick=3000, other=4000))
```

On **line 16**, which function call uses *keyword argument(s)*, and which uses *positional argument(s)*?

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 *wrong* with 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:

```
1   def icarus(x: int) -> int:
2       """Unbound aspirations!"""
3       print(f"Height: {x}")
4       return icarus(x=x + 1)
5
6   def safe_icarus(x: int) -> int:
7       """Bound aspirations!"""
8       if x >= 2:
9           return 1
10      else:
11          return 1 + safe_icarus(x=x + 1)
12
13  print(safe_icarus(x=0))
```

When 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?