



CL05 – f-Strings, Positional Arguments, Optional Parameters, and an Intro to Recursion

Reminders

- **Quiz 00:** Regrade requests will be open **till 11:59pm tomorrow night!**
 - Please submit a regrade request if you believe your quiz was not graded correctly according to the rubric
- **LS07** and **CQ01** due tonight at 11:59pm
- **Quiz 01** on Friday
 - Practice quiz and key available on site

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

Warm-up

Write a function called `check_first_letter` that takes as input two `strs`: `word` and `letter`

It should return `"match!"` if the first character of `word` is `letter`

Otherwise, it should return `"no match!"`

Examples:

- `check_first_letter(word="happy", letter="h")` would return `"match!"`
- `check_first_letter(word="happy", letter="s")` would return `"no match!"`

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)


```
1 def get_class(subject: str, num: int) -> None:
2     print(
3         "I'm currently in "
4         + subject
5         + str(num)
6         + ", but next semester I'm taking "
7         + subject
8         + str(num + 100)
9         + "!"
10    )
11
12
13 get_class(subject="COMP", num=110)
```

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

```
1 def get_class(subject: str, num: int) -> None:
2     print(f"I'm currently in {subject}{num}, but next semester I'm taking
3         {subject}{num+100}!")
4
5 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)
```

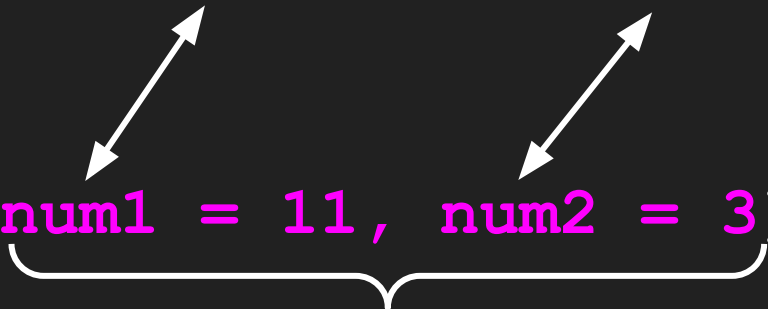
How could we convert the print statement on line 22 to use an f-string?



Recall: Signature vs Call

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

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




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)
```

Two white double-headed arrows are shown. The first arrow originates from the text 'num1 = 11' in the function call below and points to the parameter 'num1' in the function signature above. The second arrow originates from the text 'num2 = 3' in the function call below and points to the parameter 'num2' in the function signature above.

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)
```

Two white arrows originate from the function call 'divide(11, 3)'. The first arrow points from the argument '11' to the parameter 'num1' in the function definition above. The second arrow points from the argument '3' to the parameter 'num2' in the function definition above.

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

Named constants

- Variables that are meant to hold a value that *doesn't change* throughout the program's execution
- Naming convention: all uppercase letters, with underscores between any words

```
PI: float = 3.14
```


```
FREEZING_F: int = 32
```

```
FREEZING_C: int = 0
```

Default parameters

- A parameter in a function signature that is assigned a default value. If a function call does not provide a value for that parameter, the default value is used
- ******* Default parameters should come *after* non-default parameters in the function signature *******

```
def greet(name: str, greeting: str = "Hello") -> None:
```




default parameter

Default parameters

- A parameter in a function signature that is assigned a default value. If a function call does not provide a value for that parameter, the default value is used
- *** Default parameters should come *after* non-default parameters in the function signature ***

```
def greet(name: str, greeting: str = "Hello") -> None:
```



default parameter

Happy with the default greeting? `greet(name="Conor")`

Want to specify your own greeting? `greet(name="Conor", greeting="Hi")`

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?

```
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))
```

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))
```

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)`

`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`

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?

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))
```



```
1  """Mysterious 'rev' from source (src) to destination (dest)!"""
2
3
4  def rev(src: str, i: int, dest: str) -> str:
5      """You happen upon a magical lil function..."""
6      if i >= len(src):
7          return dest
8      else:
9          return rev(src=src, i=i + 1, dest=src[i] + dest)
10
11
12  print(rev(src="lwo", i=0, dest=""))
```