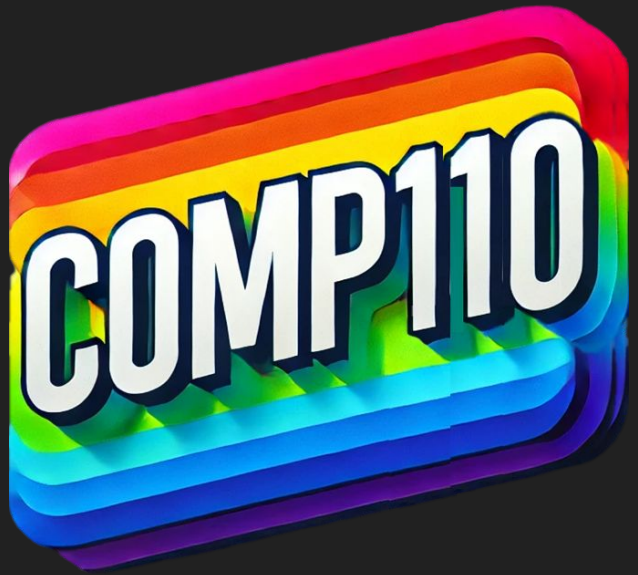


Functions and Memory Diagrams

Please be ready to write on a piece of paper or tablet!

Announcements

- EdStem is an optional way to ask questions/discuss concepts covered in this course
- Quiz 00 on Monday, May 19th
  - Ways to prepare:
    - Quiz expectations on course site
    - (Long) practice quiz and associated key
    - Office Hours
    - Ask a question on EdStem
- EX01 – Tea Party Planner – released tomorrow
    - Option: Complete parts 0-3 as quiz practice!
        - Submit to the autograder to confirm correctness

Functions by Intuition

Consider the following **function definition** (a new concept!):

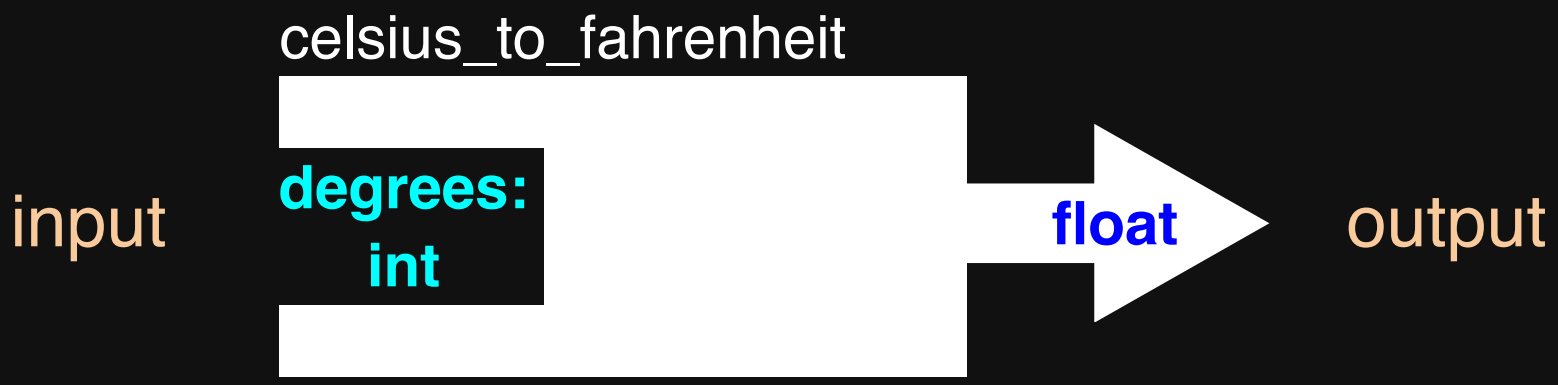
```
1 def celsius_to_fahrenheit(degrees: int) -> float:
2     """Convert degrees Celsius to degrees Fahrenheit."""
3     return (degrees * 9 / 5) + 32
```

Now, consider the following **function call expressions**, which use the definition:

```
1 celsius_to_fahrenheit(degrees=0)
2
3 celsius_to_fahrenheit(degrees=10)
```

What **value** and **type** does each function call expression evaluate to? How many connections between the ***definition*** and the ***call*** can you identify intuitively?

The fundamental pattern of functions




```
1 celsius_to_fahrenheit(degrees=10)
```

↓

50.0

Function definitions are like recipes

- A **recipe** in a book *does not result in a meal until you cook it*
- A **function definition** in your program *does not result in a value until you call it*
- An **adaptable recipe** is one where you can substitute ingredients, follow the same steps, and get different, but intentional, results
- A **parameterized function definition** is one where you can substitute input *arguments*, follow the same steps, and get different, but intentional, results.
  - Such as converting Celsius degree values to Fahrenheit!
- **Recipes** and **function definitions** are written down once with dreams of being cooked (called) tens, hundreds, thousands, ... billions of times over!

The anatomy of a function definition

define...

a function named this...

with this parameter list of "inputs"

which will return a value of this return type

```
def name_of_function(parameter: type) -> returnType:
    """Docstring description of the function."""
    return expression_of_type_returnType
```

function signature specifies how you and others will make use of the function from elsewhere in a program

- What is its **name**?
- What input **parameter(s) type(s)** does it need?
  - (Think: ingredients)
- What **type of return value** will calling it result in?
  - (Think: meal)

The anatomy of a function definition

define... a function named this... with this parameter list of "inputs" which will return a value of this return type

```
def name_of_function(parameter: type) -> returnType:
    """Docstring description of the function."""
    return expression_of_type_returnType
```

function body specifies the subprogram, or set of steps, which will be carried out every time a function calls the definition:

- Each statement in the body is **indented** by (at least) one level
- The **Docstring** describes the purpose and, often, usage of a function *for people*
- The function body contains one or more **statements**. For now, our definitions will be simple, one-statement functions
- **Return statements** are special and written inside of function definitions
  - When a function definition is called, a return statement indicates, **"stop following this function here and send my caller the result of evaluating this return expression!"**

The anatomy of a function definition

define... a function named this... with this parameter list of "inputs" which will return a value of this return type

```
def name_of_function(parameter: type) -> returnType:
    """Docstring description of the function."""
    return expression_of_type_returnType
```

function body

The anatomy of a function call

function name

argument list (a list of the input values for the parameters)

```
name_of_function(argument=value)
```

Fill in the blank to complete the missing expression

Say you want to hang string lights around your dorm room. How long of a strand of string lights will you need?

```
1 def perimeter(length: float, width: float) -> float:
2     """Calculate the perimeter of a rectangle."""
3     return _____
```

This is an example function call expression that calls the perimeter function definition above. What value and type will this expression evaluate to?

```
1 perimeter(length=10.0, width=8.0)
```

Practice: write down at least one line number for each:

```
1  """A simple program with a function call."""
2
3
4  def perimeter(length: float, width: float) -> float:
5      """Calculates the perimeter of a rectangle."""
6      return 2 * length + 2 * width
7
8
9  print(perimeter(length=10.0, width=8.0))
```

1. Docstring
2. Function call(s)
3. Return statement
4. Function definition
5. Use of a parameter's name in an *expression*

The `return` statement vs. calls to `print`

- **The return statement is *for your computer*** to send a result back to the function call's "bookmark" *within your program*
  - A bookmark is dropped when you *call* a function with a return type. When that function's body reaches a *return statement*, the returned value replaces the function call and the program continues on
- **Printing is *for humans* to see**. To share some data with the user of the program, you must output it in some way
- If you have a function, `my_func`, that returns some value, you can print the value it returns by:
  1. Printing its return value directly with `print(my_func())` or
  2. (Later in the course) by storing the returned value in a variable and *later* printing the variable

Tracing programs by hand: Intro to memory diagrams!

- The evaluation of a program depends on many interrelated values
- As any non-trivial program is evaluated, what needs to be kept track of includes:
  1. The current line of code, or expression within a line, being evaluated
  2. The trail of function calls that led to the current line and “frame of execution”
  3. The names of parameters/variables and a map of the values they are bound to
  4. More!
- As humans, this quickly becomes more information than we can mentally keep track of.
  - Good news: Memory diagrams will help you keep track of it all on paper!

Memory diagrams

- A program's runtime *environment* is the mapping of *names* in your program to their *locations* in memory
- A program's *state* is made up of the *values stored* in those locations
- You can use memory diagrams to visually keep track of both the environment and its state
- Memory diagrams will help you keep track of how function calls are processed.
  - Where was the function called?
  - What was the return value, and where was it returned to?
  - (and more!)

```python
"""A simple program with a function call."""


def perimeter(length: float, width: float) -> float:
    """Calculates the perimeter of a rectangle."""
    return 2 * length + 2 * width


print(perimeter(length=10.0, width=8.0))
```

```python
"""A program with *two* function calls."""

def perimeter(length: float, width: float) -> float:
    """Calculates the perimeter of a rectangle."""
    return 2 * length + 2 * width

def square_perimeter(side: float) -> float:
    """Calculates the perimeter of a square."""
    return perimeter(length=side, width=side)

print(square_perimeter(side=4.0))
```

(a second example, in case you want to try this on your own!)

CQ00: Submitting the memory diagram to Gradescope by 11:59pm

From your phone:

1. Open the CQ00 assignment and make a submission
2. Upload a photo of your memory diagram
3. Complete your submission (and please make sure your photo is in the right orientation!)

Other assignments due tonight:

- LS04: Introducing Functions & Function Syntax
- EX00: Hello, World! (programming assignment)