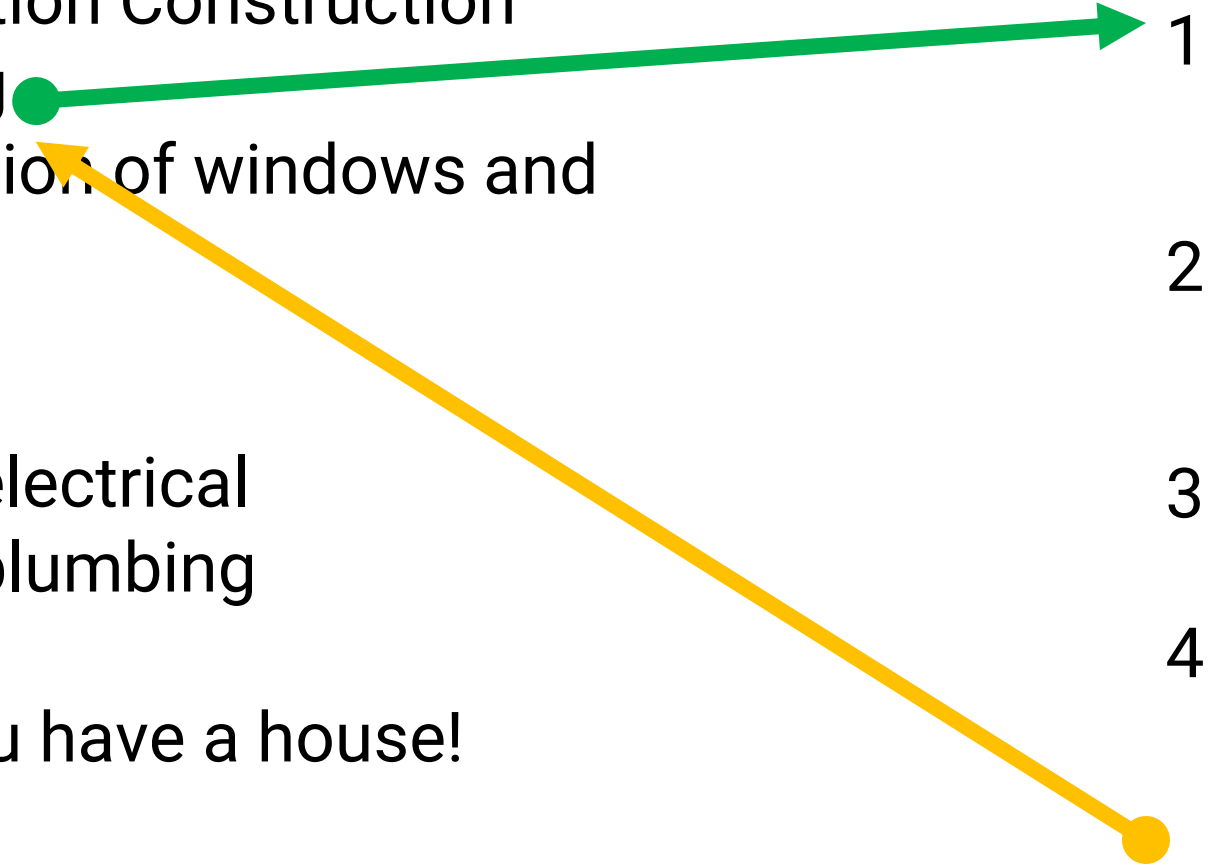
Function Intuition: How-to Build a House

1. Site Preparation and Grading
2. Foundation Construction
3. Framing
4. Installation of windows and doors
5. Roofing
6. Siding
7. Rough electrical
8. Rough plumbing
9. ...
10. Now you have a house!

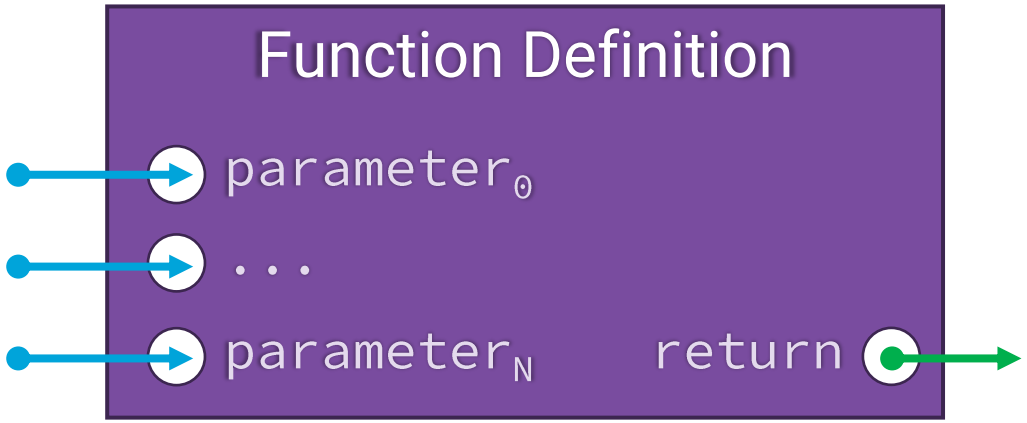


A Framing "Function"

1. Pre-build the outside frame in 8-foot sections
2. Stand each 8-foot section of the frame up
3. Insert braces for support
4. Repeat steps 3 and 4 until entire perimeter is complete

Function Definition Overview

- A **function definition** is a subprogram
  - **Parameters** are placeholders for inputs
  - The **function body** is the algorithm, or sequence of steps, the function will follow when it is used
  - A function may **return** a resulting value
    - The function *declares* the *type* of return value



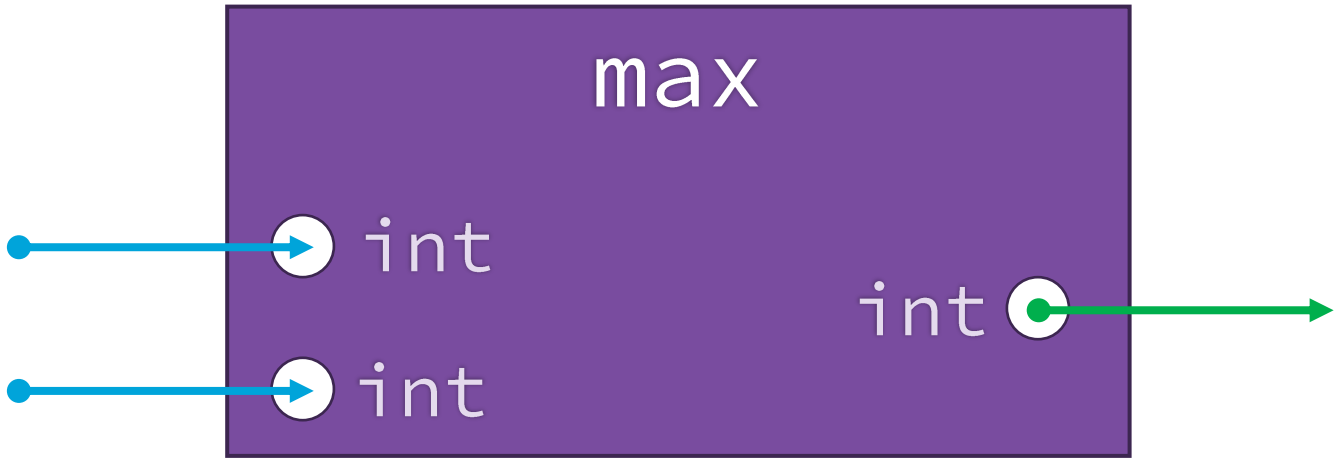
* *Defining* a function is like *writing down* a recipe. The definition has no immediate result. It is not until you *call* a function or *follow* a recipe that its steps are actually carried out.

Visualizing: A `max` Function Definition

- Imagine a function that takes in two int values and returns the largest.

- We can visualize it like the block below:
  - Two ***parameters***, both need to be type **int**
  - The ***function body*** is the purple box, its algorithm is opaque *"abstracted away"*
  - The ***return type*** is an **int**

- So, how can we *use* of this building block in our program?



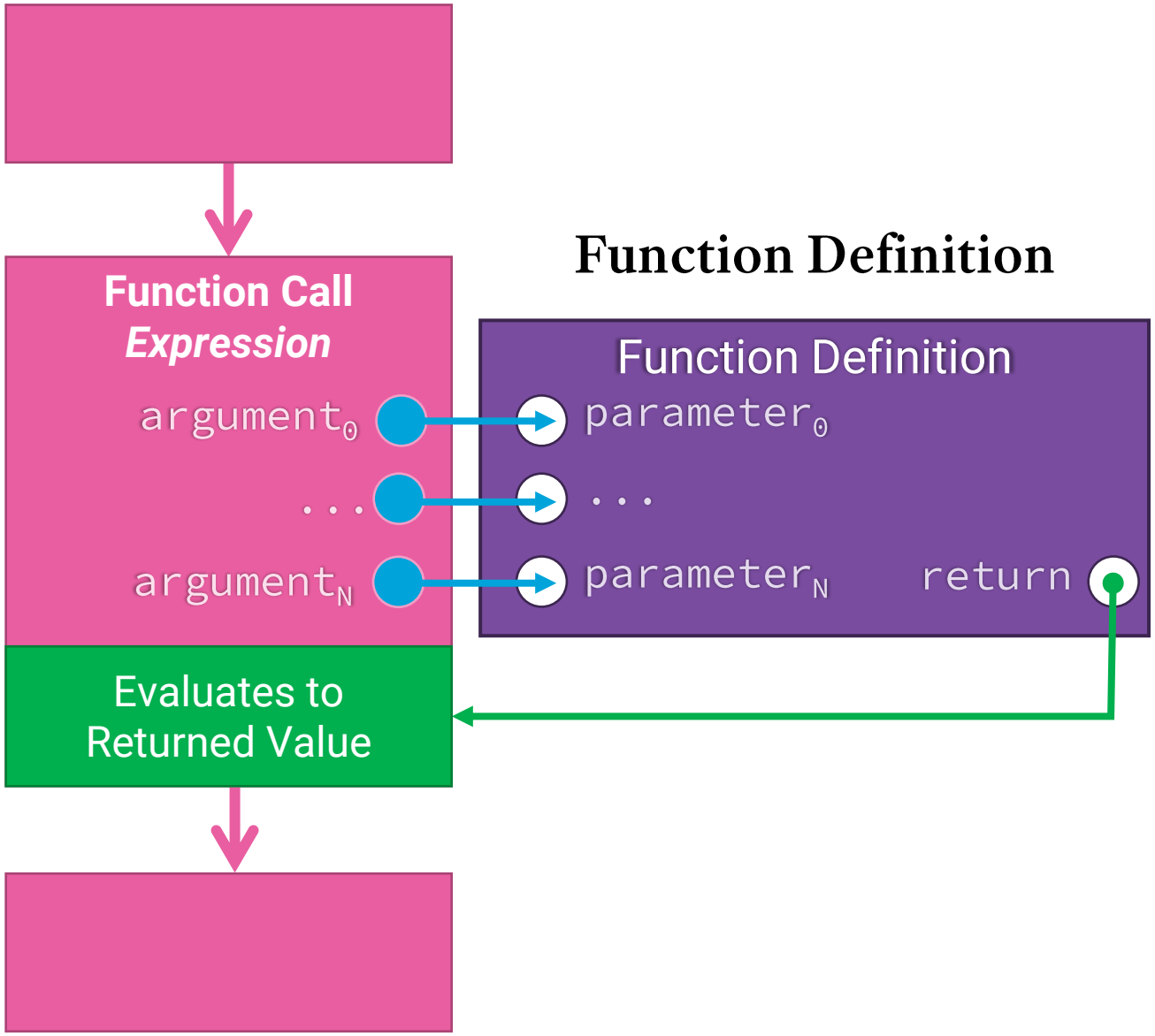
Function Call Expression Overview

1. A **function call** is an *expression* that will carry out a function's definition and evaluate to its returned value.

2. **Arguments** are the actual input values assigned to the definition's parameters.

3. A bookmark is left at the function call expression. Control **jumps into** the function definition.

4. When control reaches the function's return statement, the **returned result is substituted** for the function call and control **jumps back.**



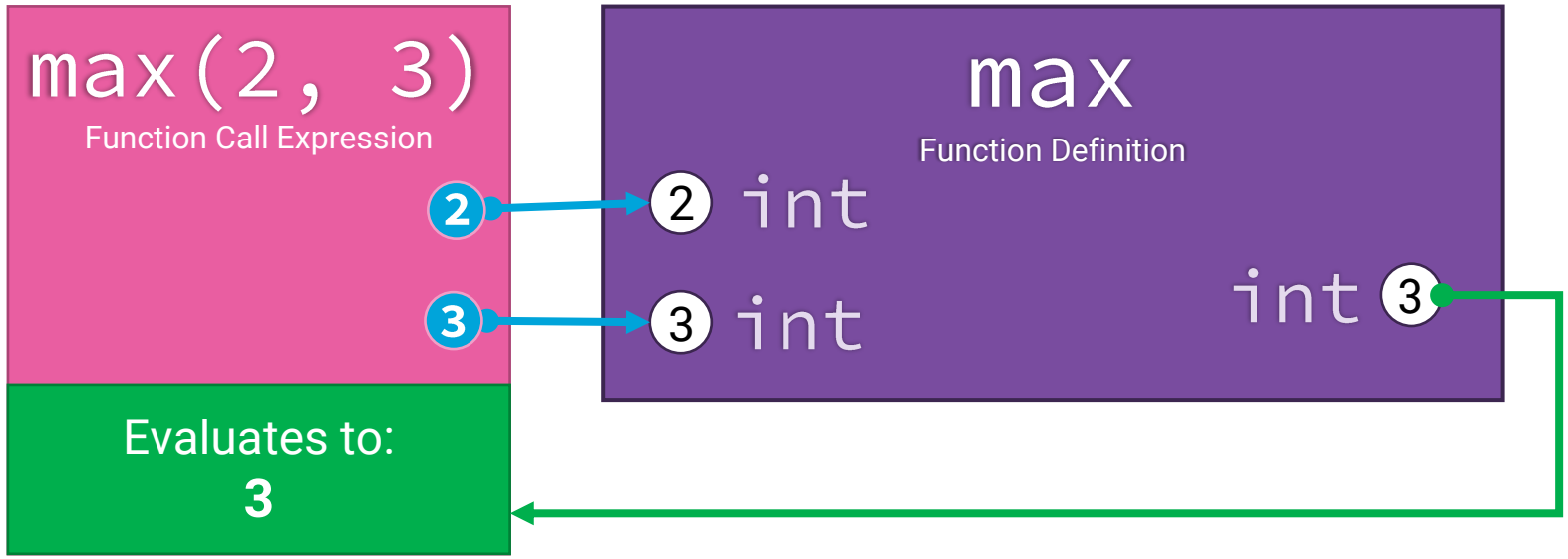
Visualizing: A `max` Function Call Expression

- Imagine the *function call expression* on the right-hand side of this variable initialization statement.

```
biggest: int = max(2, 3)
```

  - We *know* the expression `max(2, 3)` must evaluate to a single `int` value.

1. A function call expression needs to be evaluated
2. The call's arguments (2 and 3) are used as definition's input parameters
3. The max algorithm results in the value 3 returning
4. The function call expression evaluates to 3



Function Definition Syntax

```
def [name]([parameter_0], ..., [parameter_N]) -> [return_type]:
    [function body statement_0]
    ...
    [function body statement_N]
```

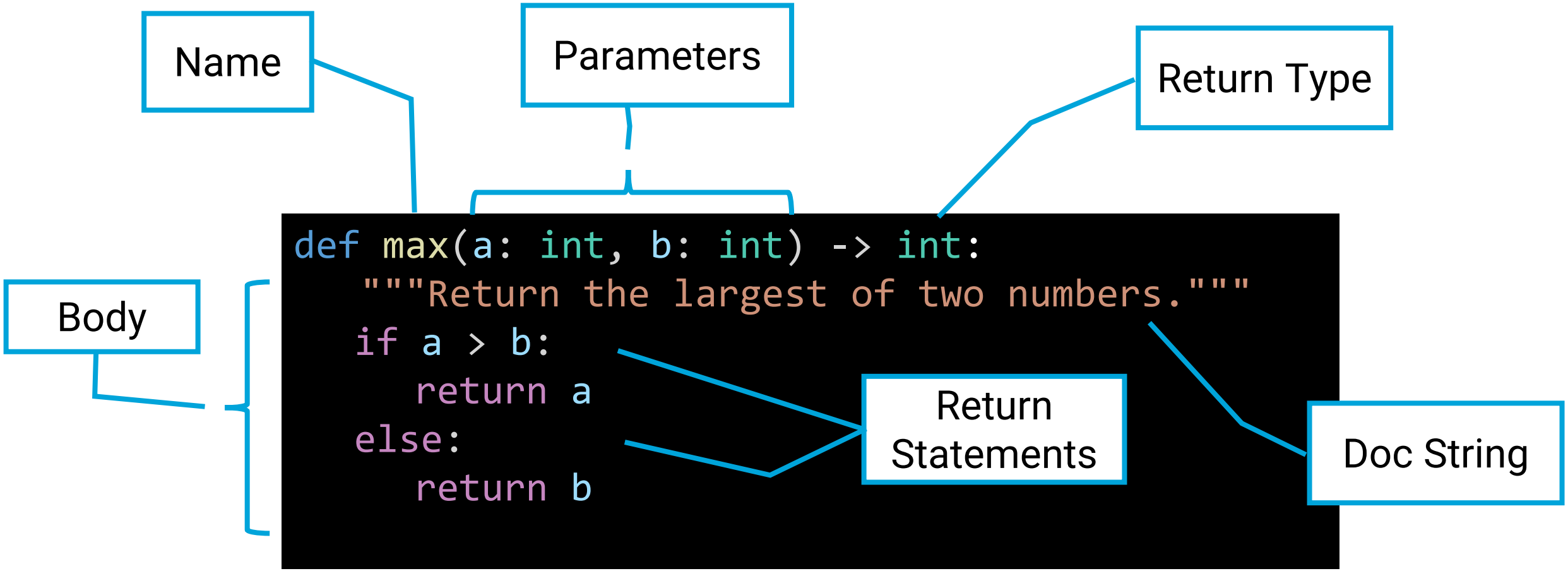
- Like variables, functions are given a **name.**
  - Function names are governed by the same *identifier* rules as variables.

- **Parameters** are special variable declarations.
  - Each parameter declared has the following syntax `[name]: [type]`
  - Parameters are placeholders for the inputs a function needs.

- **Return type** specifies the data type the function will return.

- **Statements** in the **body** block run *only* when a function is called.

Function Definition Example



The `max` function can be given two `int` values and will return the larger of the two.

Function Call Syntax

```
[name]([argument_0], ..., [argument_N])
```

Example:

```
max(2, 3)
```

1. When a function call is encountered the processor **drops a bookmark**.

2. A **function call's data type** is its function definition's return type

   For example: `biggest: int = max(2, 3)`

   Since the **`max`** function's return type is **`int`**, a function call to **`max`** is an **int** expression. What it evaluates to will be assigned to **biggest**.

3. When control reaches a function call, it follows rules to jump into to the function call with input arguments and return with a result.
   - We'll explore these rules in depth in upcoming lessons.

What purpose do functions serve?

- Functions are a fundamental unit of **process abstraction**
  - Learning to tie your shoe was process abstraction
    - As a child, you struggled to learn the right series of steps
    - Nowadays you can just "tie your shoe" without worrying about each step
  - Defining a function is process abstraction
    - Defining functions takes thoughtful effort to get the right series of steps
    - Once correct, you can reuse your function by "calling" it, without worrying about its steps
- Functions help you break down and logically organize your programs
- Functions make it easy to reuse computations or sequences of steps
  - Functions help you avoid repetitive, redundant code

Example Setup

In VSCode:

1. Open your COMP110 Workspace
   - File > Open Recent > comp110-workspace
2. Open the File Explorer Pane
   - comp110 > lessons
3. Create a new Python module in lessons directory
   - Right click lessons
   - Select new file
   - Name it "ls11_function.py"
4. Copy over the program to the right
5. Run the program, experiment with some different argument values.

```
def max(a: int, b: int) -> int:
    """Return the largest of two numbers."""
    if a > b:
        return a
    else:
        return b

biggest: int = max(2, 3)
print(biggest)

arg0: int = int(input("arg0: "))
arg1: int = int(input("arg1: "))
print(max(arg0, arg1))
```