Functions

Overview

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!

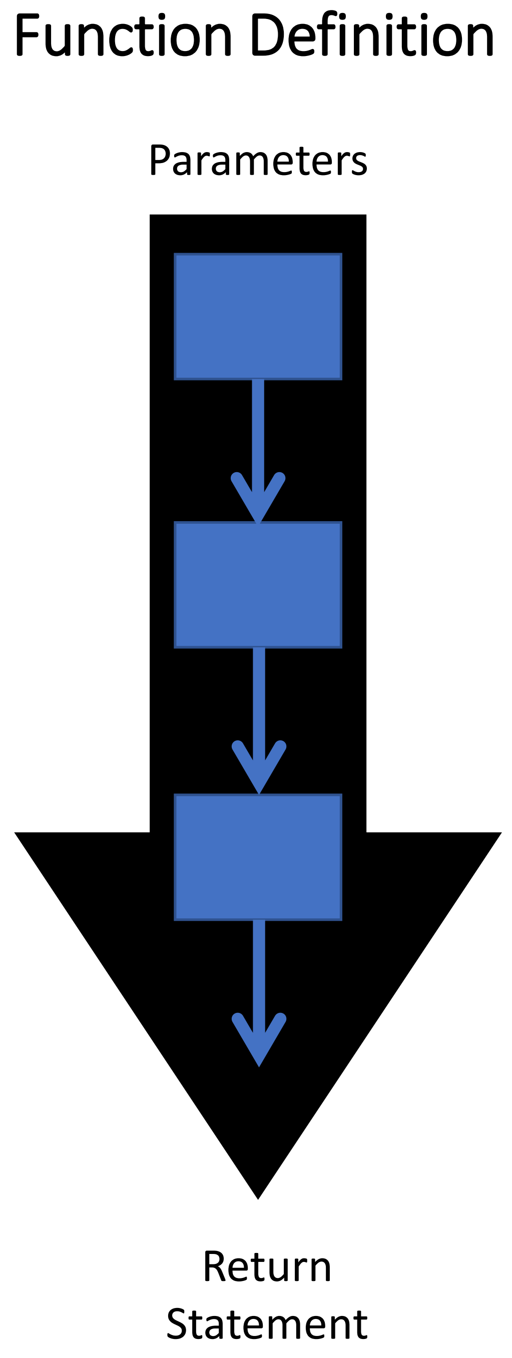
<u>A Framing "Function"</u>

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 needs
  - The **function body** is the algorithm, or sequence of steps, the function will follow when it is used
    - The function body is a block of statements
    - *Any* statement can be written inside the function body, including if-then-else, while loops, and so on
  - A **return statement** inside a function body specifies the final value a function results in

* *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.



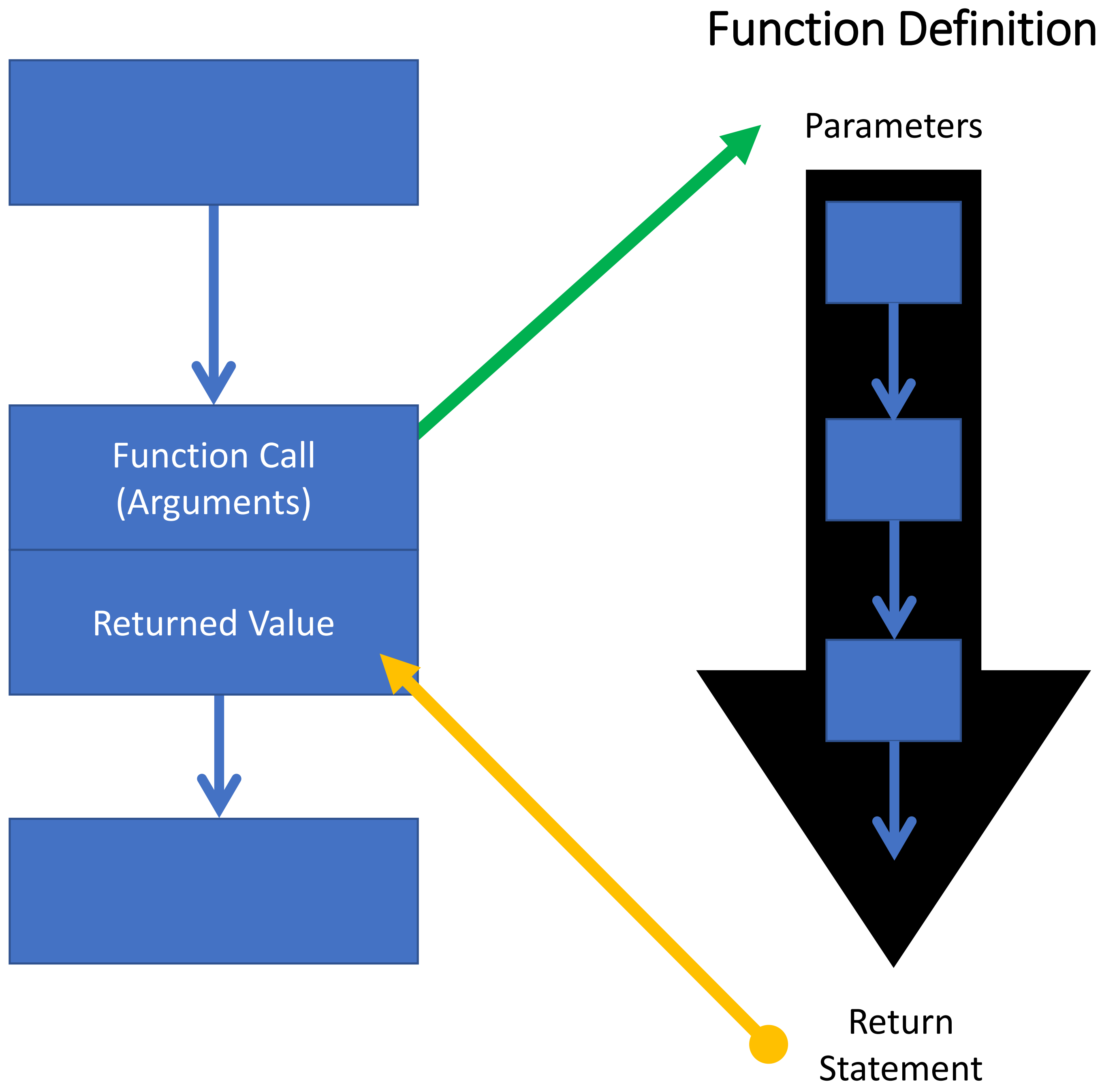
Function Call Overview

1. A **function call** instructs the processor to carry out a function's definition.

2. **Arguments** are the actual input values. They are assigned to the function definition's parameter placeholders.

3. The processor leaves a bookmark at the function call and **jumps into** the function definition.

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



Example Setup

In VSCode:

1. Start the Development Server
   - View Terminal
   - `npm run pull`
   - `npm start`

2. Open the File Explorer Pane
   - Right click on the src folder
     - Select "New folder"
     - Name it: **`x-functions`**
   - Right click on the x-functions folder
     - Select "New file"
     - Name it: **`functions-app.ts`**

3. In `functions-app.ts`, write out the code to the right. It has no errors, so review carefully if yours has any.

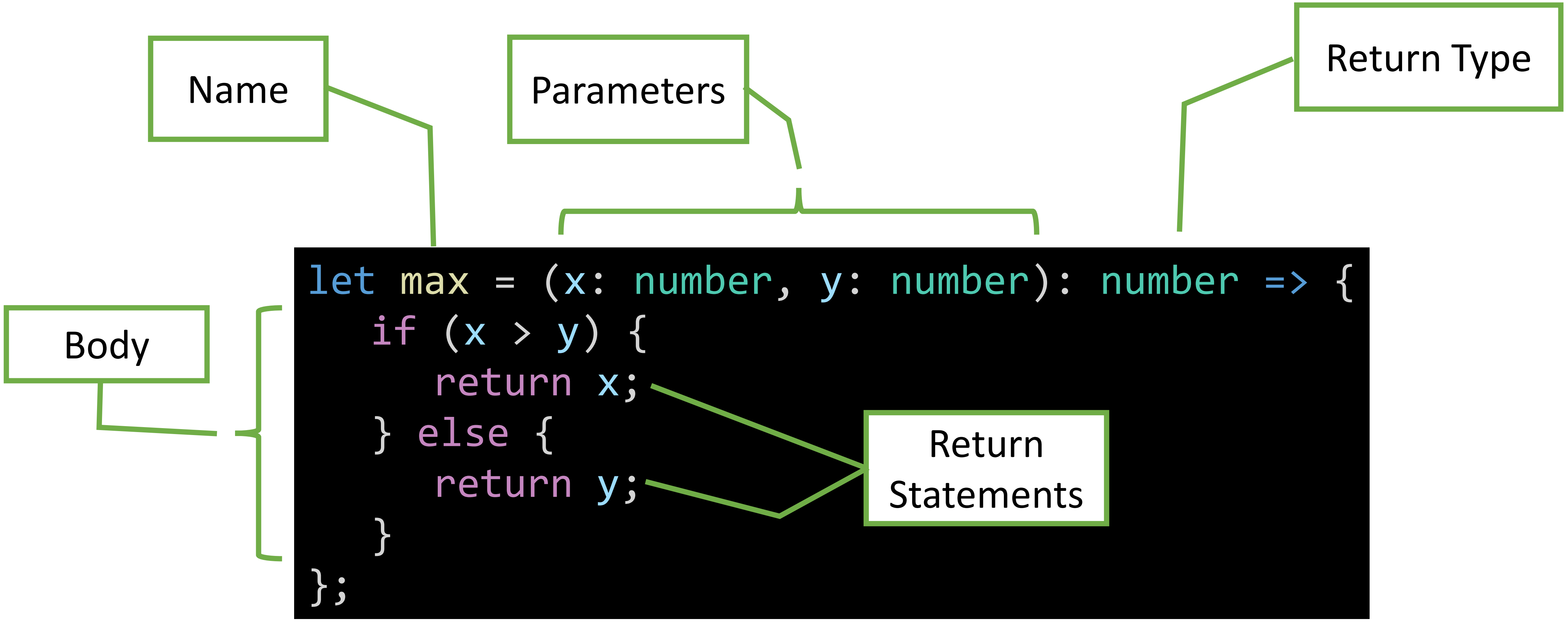
```typescript
import { print, promptNumber } from "introcs";

export let main = async () => {
    let a = await promptNumber("a");
    let b = await promptNumber("b");

    // Function Call
    let answer = max(a, b);

    print(answer + " is greatest!");
};

// Function Definition
let max = (x: number, y: number): number => {
    if (x > y) {
        return x;
    } else {
        return y;
    }
};

main();
```

Function Definition Syntax

```
let <name> = (<parameters>): <returnType> => {
  <function body statements>
};
```

- We will define functions outside of the `main` function, typically following it
- Like variables, functions can be given a **name.**
- **Parameters** are special variable declarations. They 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



```
let max = (x: number, y: number): number => {
    if (x > y) {
        return x;
    } else {
        return y;
    }
};
```

The **max** function can be given two **number** values and will return the larger of the two.

Function Call Syntax

```
<name>(<arguments>)
```

Example:

```
max(a, b)
```

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: `let answer: number = max(a, b);`

   Since the **`max`** function's return type is **`number`**, a function call to **`max`** can be assigned to the number variable **`answer`**.

3. When the processor reaches a function call, it follows a set of rules to jump over to the function call with input arguments and return back 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