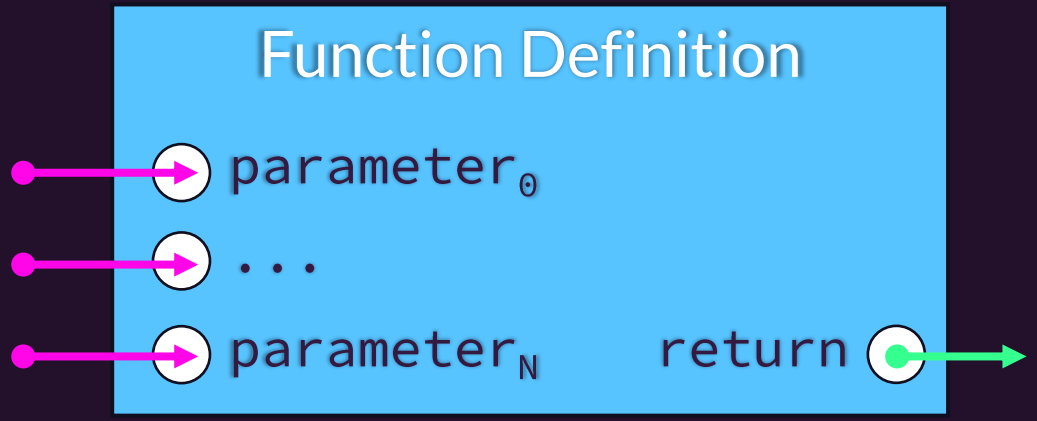
Function!

Definitions and Calls

in Python

Function Definition Overview

- A **function definition** is a subprogram
  - It has a **name**
  - **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



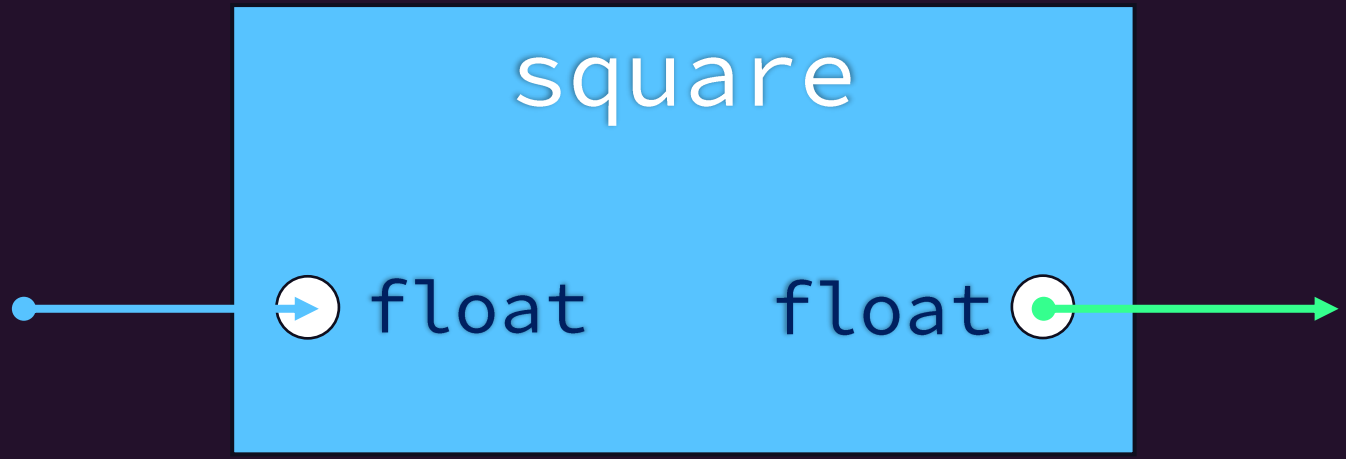
* *Defining* a function is like *writing down* a recipe. The definition has no immediate result. It is not until you *call* a function or *follow* a recipe that its steps are carried out.

Visualizing: The `square` Function Definition

- Imagine a function that takes in a float value and returns its square.
  - Disclaimer: Yes, this is a *very silly* function with the power operator `**` built-in! It's chosen to highlight the *shape* and *mechanisms* of a function definition and call.

- We can visualize it like the block below:
  - One ***parameter***, of type **float**
  - The ***function body*** is the ***named*** 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?



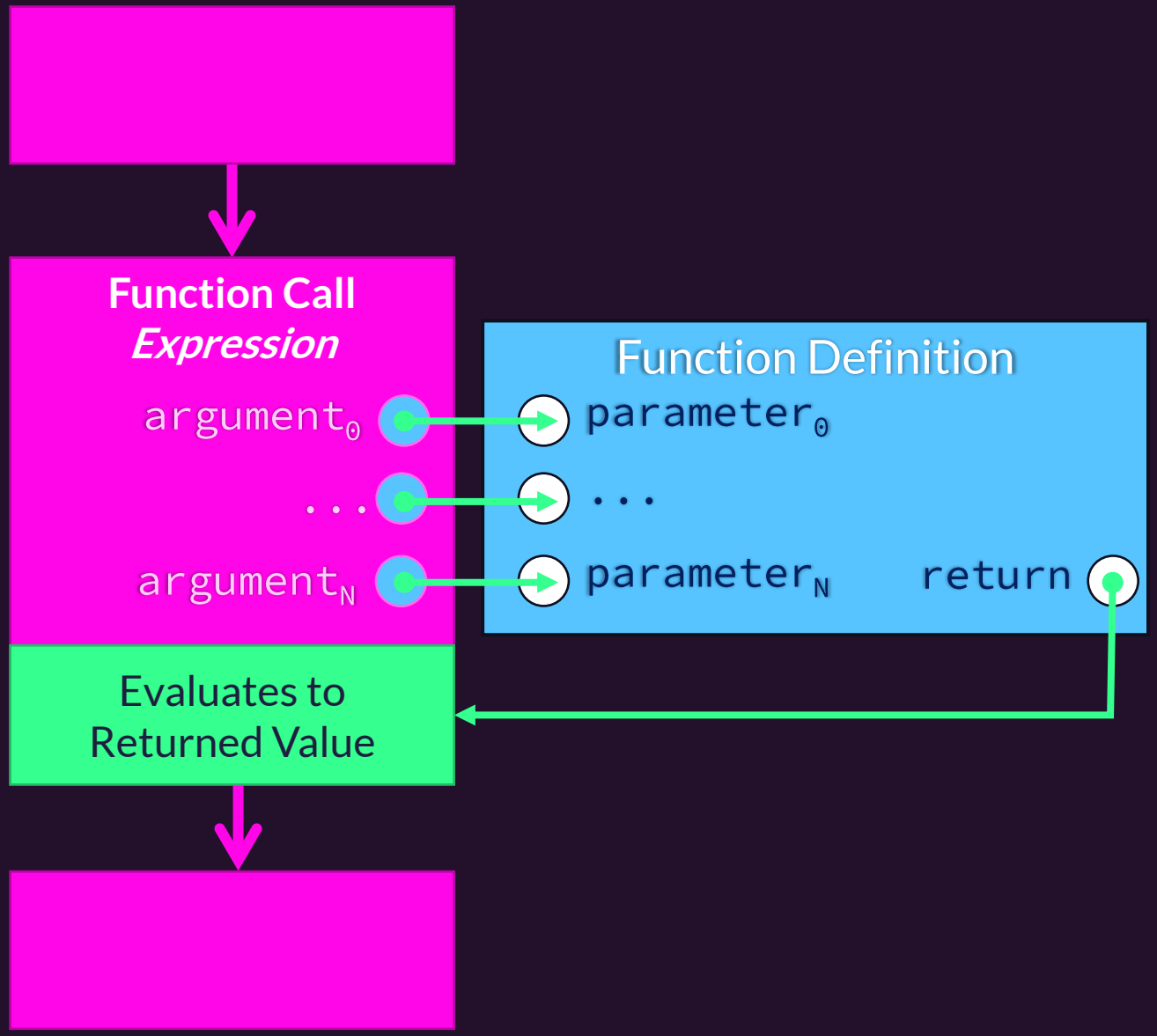
Function Call Expression Overview

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.**



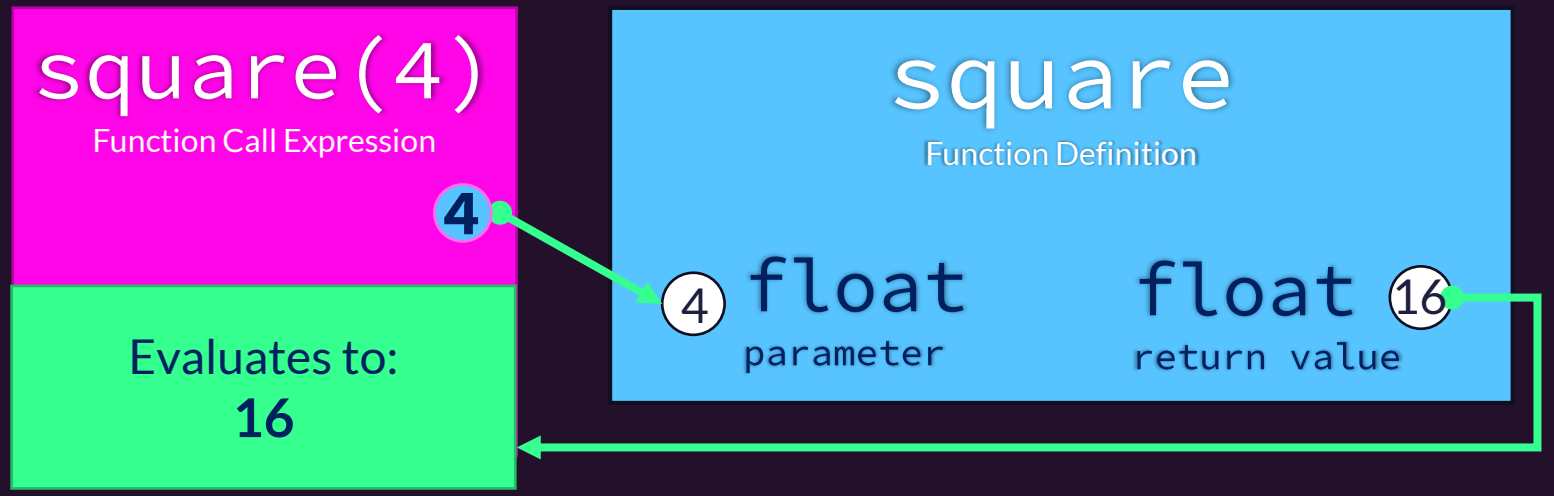
Visualizing: A `square` Function Call Expression

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

```
four_squared: float = square(4)
```

  - We *know* the expression `four_squared(4)` must evaluate to a single `float` value.

1. A function call expression needs to be evaluated
2. The call's argument 4 is used as definition's input **parameters**
3. The square "algorithm" results in the value 16 returning
4. The function call expression evaluates to 16



Function Definition Syntax

```
def [name]([parameter_0], ..., [parameter_N]) -> [return_type]:
    [function body statement_0]
    ...
    [function body statement_S-1]
    return [expression of type return_type]
```

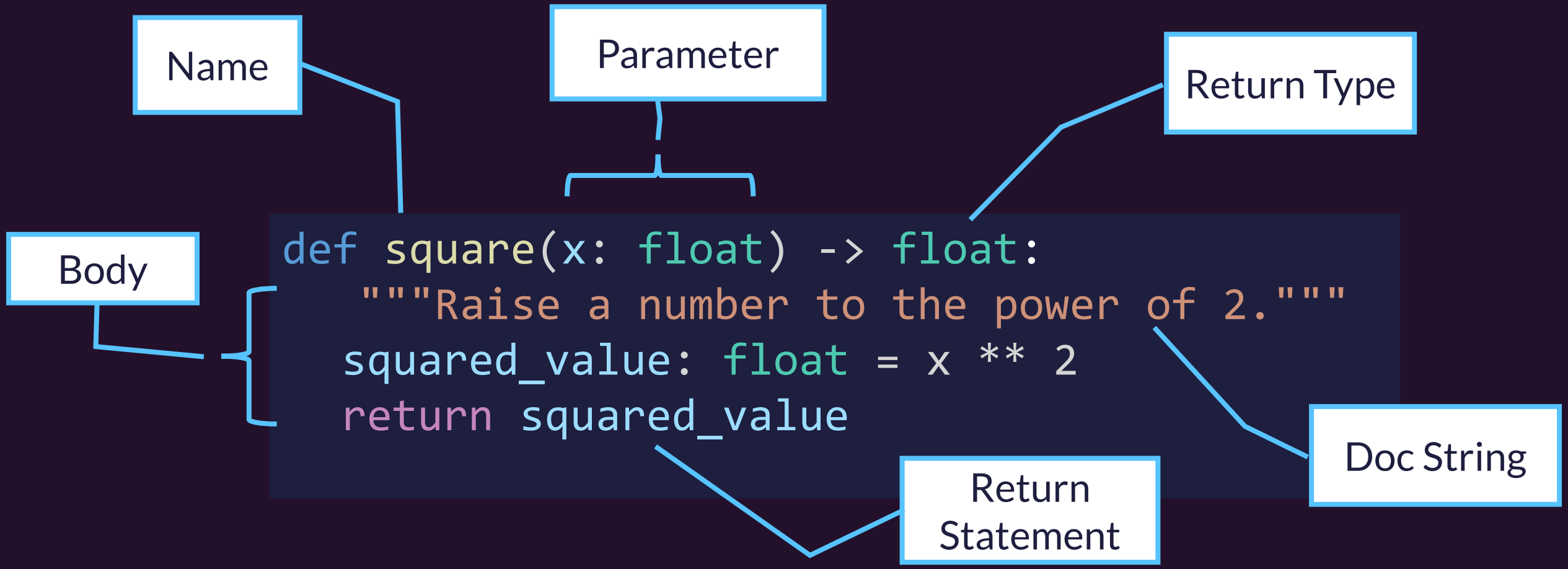
- Like variables, functions are given a **name**.
  - Function names are governed by the same *identifier* rules as variables.

- **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.
  - Statements in the same block must be consistently indented one tab
  - Functions must have at least one return statement which return an expression of type return_type

Function Definition Example



The **square** function can be given a **float** value and returns the square of its parameter.

Function Call Syntax

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

Example

```
square(4.0)
```

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 `four_squared: float = square(4.0)`
   Since the **`square`** function's return type is **`float`**, a function call to **`square`** is a float expression.

3. When control reaches a function call, it follows rules to jump into to the function call with input arguments and returns with the return value.
   - We'll continue exploring 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

Functions with Multiple Parameters

- Let's declare the function with multiple parameters shown right!
  - Define it after square
  - Notice the parameters are separated by a comma

```
def power(x: float, exp: int) -> float:
    """Returns x raised to the exp."""
    raised_value: float = x ** exp
    return raised_value
```

- To *call* your function:

1. Save your file
2. Begin a new REPL
3. Import it (as shown right)
4. Call it!

```
>>> from lessons.ls08_functions import power
>>> power(4.0, 2)
16.0
>>> x: float = power(3.0, 4)
>>> x
81.0
>>> power(3.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: power() missing 1 required
positional argument: 'exp'
>>> quit()
```

- Notice: To call a function with multiple parameters requires multiple arguments!
  - Ordering and types matter!

Up Next House Challenge & Async

- We will transition to a small challenge and then you should go complete LS09: Named Constants at your own pace. Hand-in questions before midnight tonight.

- Challenge: How could you change this line of code to make use of a *function call expression* to your `power` function, rather than `x ** 2`?

```
def square(x: float) -> float:
    """Raise a number to the power of 2."""
    squared_value: float = x ** 2
    return squared_value
```