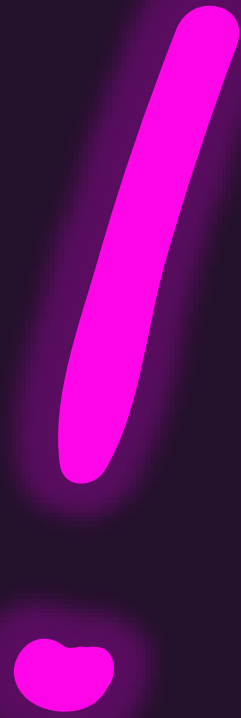


Parameter  
Passing



# Introducing Parameters

- Parameters allow functions to require additional pieces of information in order to be called
- Parameters are specified within the parenthesis of function definition
- Parameters look a lot like variable declarations... *because they are!*
- Parameters are local variables to the function. Their names are scoped inside of the function body's block.

*General Form*

```
# Function Definition
def <name>([parameters]) -> <return_type>:
    [statement0]
    ...
    [statementN]
```

*Example*

```
# Function Definition
def max2(x: int, y: int) -> int:
    if x > y:
        return x
    else:
        return y
```

# What effect does declaring parameters have?

## Function Definition

```
def max2(x: int, y: int) -> int:  
    if x > y:  
        return x  
    else:  
        return y
```

## OK Function Call Usage

```
max2(3, 4)
```

## Incorrect Function Call Usage

```
max2(3)
```

## Incorrect Function Call Usage

```
max2(3, 4, 50)
```

- When a function declares **parameters**, it is declaring:  
"you must give me these extra pieces of information in order to call me"
- The function *definition* on the left says:  
"in order to call **max2**, you must give me two number values"
- In the *usage* to the right, when we *call* max, we must give it two **int** values.

# Arguments vs Parameters

These are arguments.



`max2(3, 4)`

- Arguments are the *values* we assign to parameters
- The type of the arguments must match the types of the parameters
- We couldn't call max with str values: `max2("oh", "no")`

These are parameters.



```
def max2(x: int, y: int) -> int:  
    if x > y:  
        return x  
    else:  
        return y
```

# Function Calls: Step-by-Step (1 / 3)

## L1. max2(8, 9)

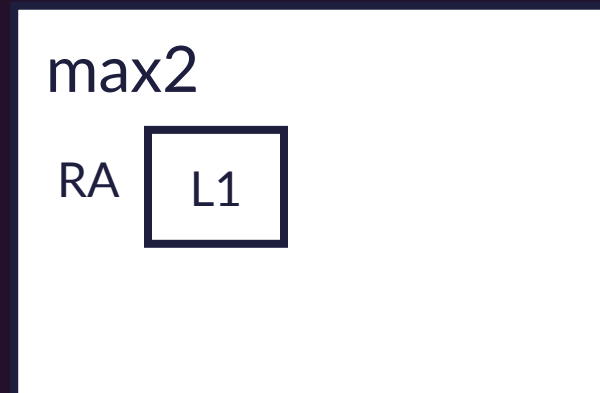
For each function call...

1. Is name defined and bound to a function?
  - NameError if not!
2. Does it have the correct # of arguments for function's parameters?
  - TypeError if not!
3. Its *argument expressions* are evaluated.
  - In this example, 8 and 9 are fully evaluated literals.
4. In memory, a frame is established on the call stack and a Return Address (RA) Line Number is recorded as a "bookmark" of where we'll come back to with a result.

```
def max2(x: int, y: int) -> int:  
    if x > y:  
        return x  
    else:  
        return y
```

Notice the argument matches the parameters in type (number) and count (2)!

Stack Memory:



# Function Calls: Parameter Passing (2 / 3)

L1. `max2(8, 9)`

```
def max2(x: int, y: int) -> int:
```

```
    x = 8
```

```
    y = 9
```

```
    if x > y:
```

```
        return x
```

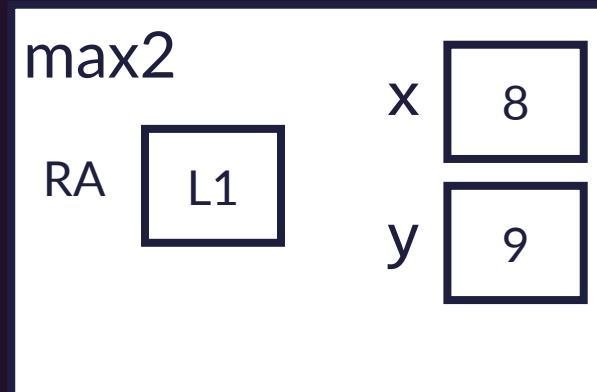
```
    else:
```

```
        return y
```

Argument values are assigned to parameters:

1. This happens invisibly when the code is running. You will never see the lines to the right.
2. However, each time a call happens, the processor assigns each argument value to its parameter.
3. This is called "parameter passing" because we are copying arguments from one point in code *into* another function's frame in memory.

Stack Memory:



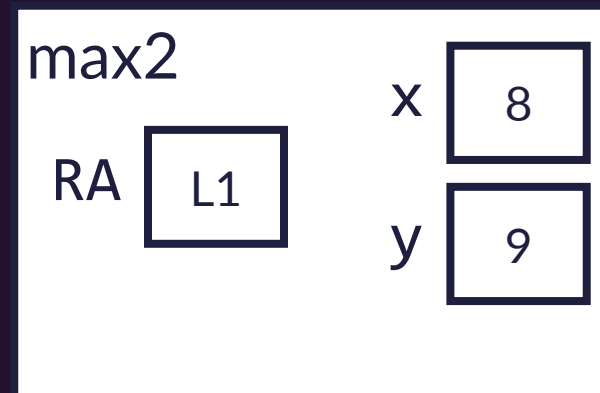
# Function Calls: Jumping into Function Body (3 / 3)

L1. `max2(8, 9)`

3. Finally, the processor then *jumps into* the function and continues onto the first line of the function body block

```
def max2(x: int, y: int) -> int:  
    x = 8  
    y = 9  
    if x > y:  
        return x  
    else:  
        return y
```

Stack Memory:



# Function Calls: Returning (3 / 3)

L1. `max2(8, 9)`

The return statement is discussed in full in another lesson, but for completeness, when a return statement is reached its expression is evaluated and added as the RV of the frame.

This value **9** is what the function call expression `max2(8, 9)` would evaluate to. Control would resume at the Return Address at L1.

```
def max2(x: int, y: int) -> int:  
    x = 8  
    y = 9  
    if x > y:  
        return x  
    else:  
        return y
```

Stack Memory:

