Memory Diagrams

Tracing Programs by Hand

- Understanding how a program will evaluate depends on systematically keeping track of many details.

- As your program is evaluated, there are many moving parts:
    1. The current line of code, or expression within a line, it will process next
    2. The trail of function call bookmarks that led to the current line
    3. The values of all variables and a map of variable "names" to the location of their values

- For humans, this is more than you can keep track of in your head!
    - Good news: diagrams will help you keep track of these things... just like the CPU

Environment Diagrams

- A program's *state* is made up of the *values stored* in memory.
- A program's *environment binds names* in your program to *values* in memory.
- Use *environment diagrams* to ***trace*** both *state* and naming *environment*.
- Additionally, they'll help you keep track of how function calls are processed.
- In the 2018-2019 academic year we began teaching with these diagrams
  - On the final exam, students who made use of environment diagrams to trace code were over 50% less likely to make errors than students who did not.

Environment Diagram

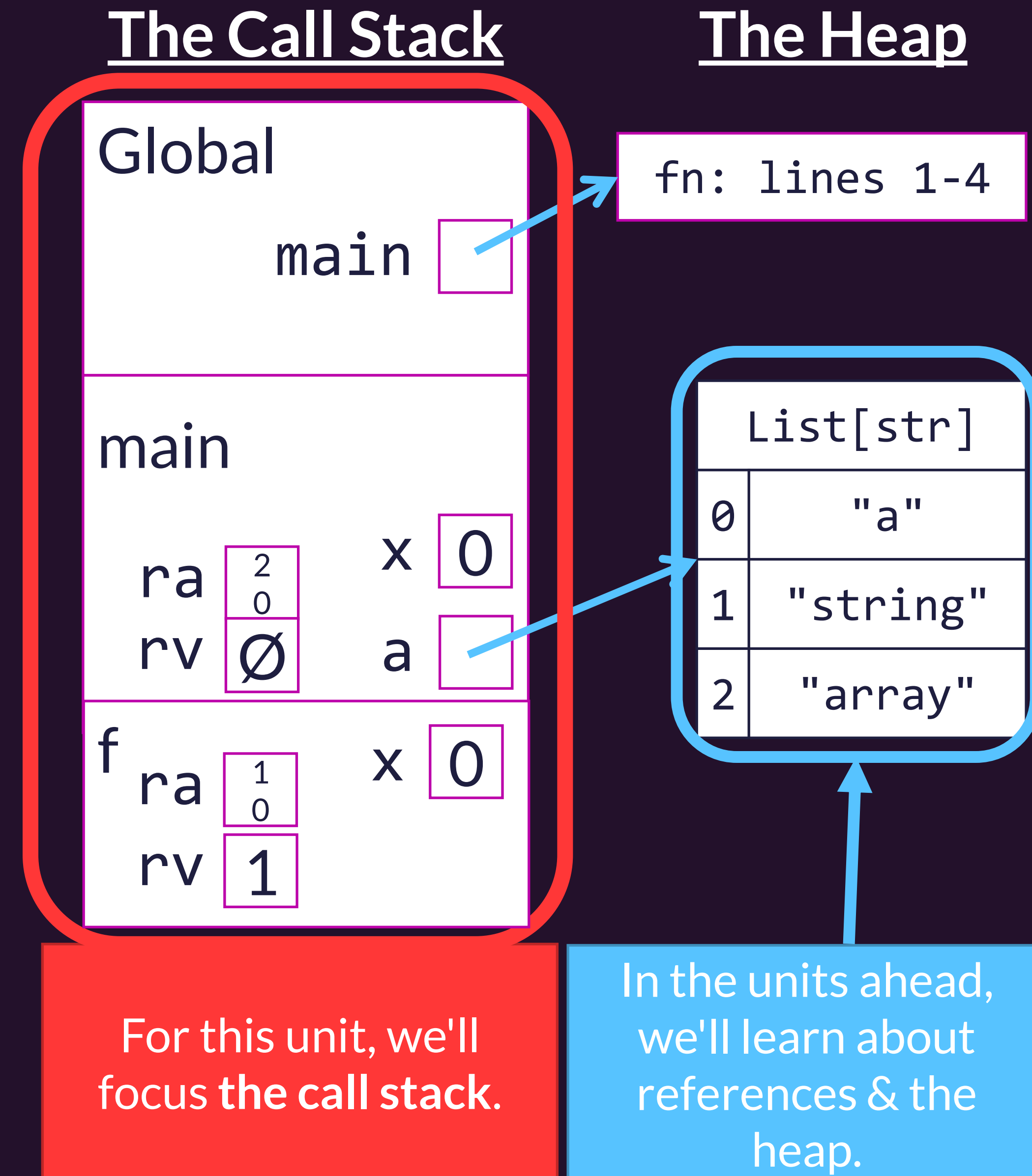
- There are two areas of an environment diagram:

1. Call Stack (or **"The Stack"**)
   - When a function is called, a new **Frame** is added
   - Every frame has:
     - The name of its function definition
     - A list of **variable names** and boxes holding their **bound values**
     - Variable values are stored in stack frames
     - A place to represent its return value (**rv**) when it returns.

2. Dynamic Memory Heap (or **"The Heap"**)
   - We'll come back to this in the next unit.

- This is *a rough approximation* of the model of how state in your programs is managed by the processor.

Example:



Environment Diagram Example

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

- Let's trace the example to the left using an environment diagram!
- In the process you will learn how to:
  - Establish a frame for **main**
  - Establish **local** variables (those declared *inside* of a function's body) in the frame
  - Call functions
    - Establish a **frame** for the function
    - Establish **parameters** as local variables, assigned their **argument's** values
    - Keep track of the value returned by a function call

Environment Diagram Example

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

- Let's trace the example to the left using an environment diagram!
- In the process you will learn how to:
  - Establish a frame for **main**
  - Establish **local** variables (those declared *inside* of a function's body) in the frame
  - Call functions
    - Establish a **frame** for the function
    - Establish **parameters** as local variables, assigned their **argument's** values
    - Keep track of the value returned by a function call

Module Evaluation

When Python loads a **module** (a file name ending in .py) the stack and heap are empty outside of Python's **built-ins**, such as the **print** function, which are outside our diagramming concern. A Globals **frame** is established and evaluation begins from the top of the file.

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

The Call Stack

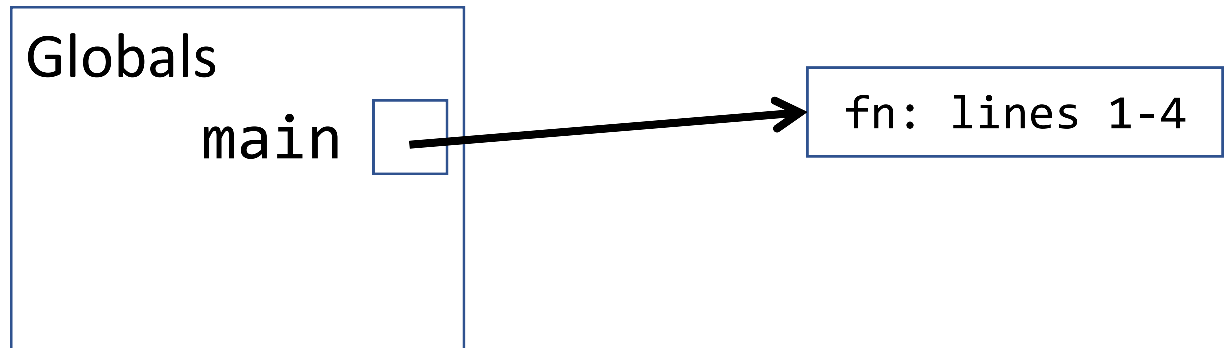
Globals

The Heap

Function Definition - `main`

When a function is defined, its name is bound in your current stack frame. It refers to an function object ("fn" shorthand) representing its code stored on the heap. We'll use its line #s.

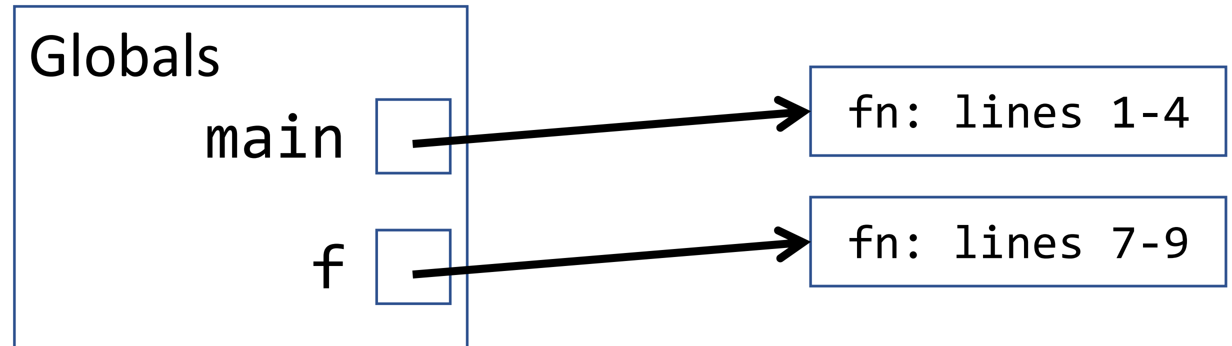
```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```



Function Definition - f

When a function is defined, its name is bound in your current stack frame. It refers to an function object ("fn" shorthand) representing its code stored on the heap. We'll use its line #s.

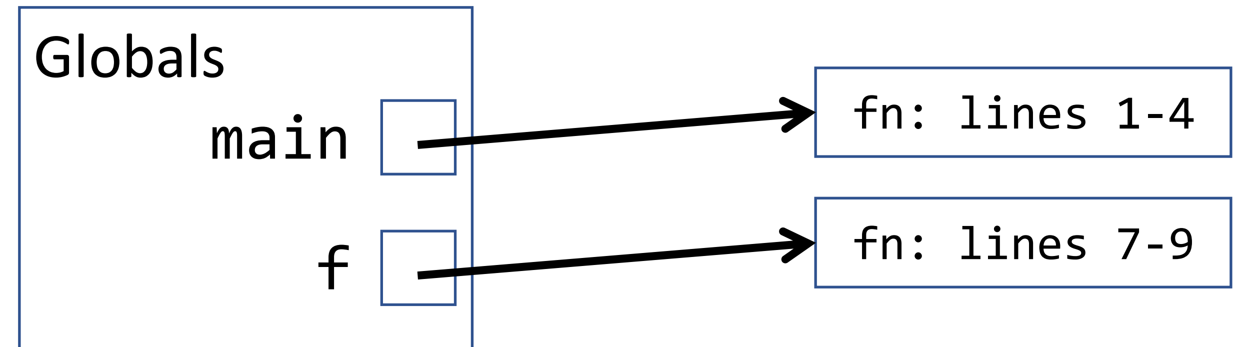
```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```



Name Resolution: What is *main*?

When a name is encountered in our program we must be able to resolve what it is bound to in memory. In this case, **main** is bound to the **function** defined on lines 1 through 4. The ()'s following the name main tell us this is a **function call**.

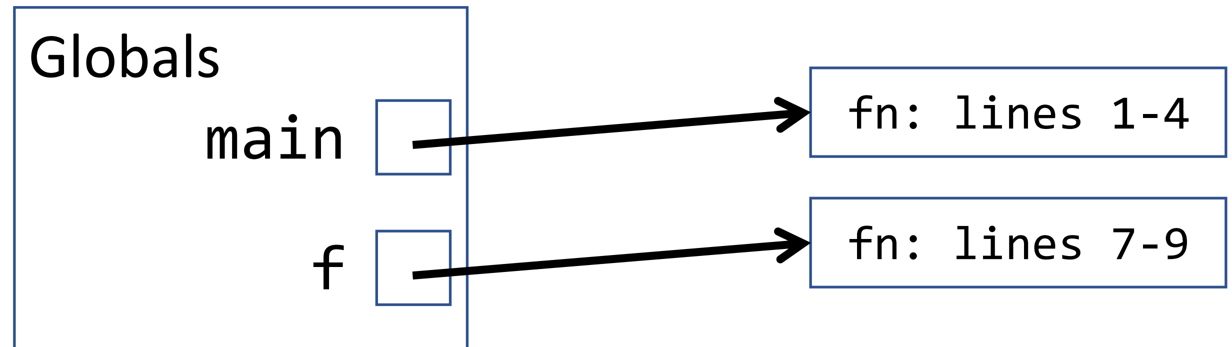
```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```



Aside: Why does the call to main occur at the end?

Remember: if you make use of a name is not yet defined, you get a NameError in Python.
Calling main at the end ensures all functions in the module are defined before main begins.

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

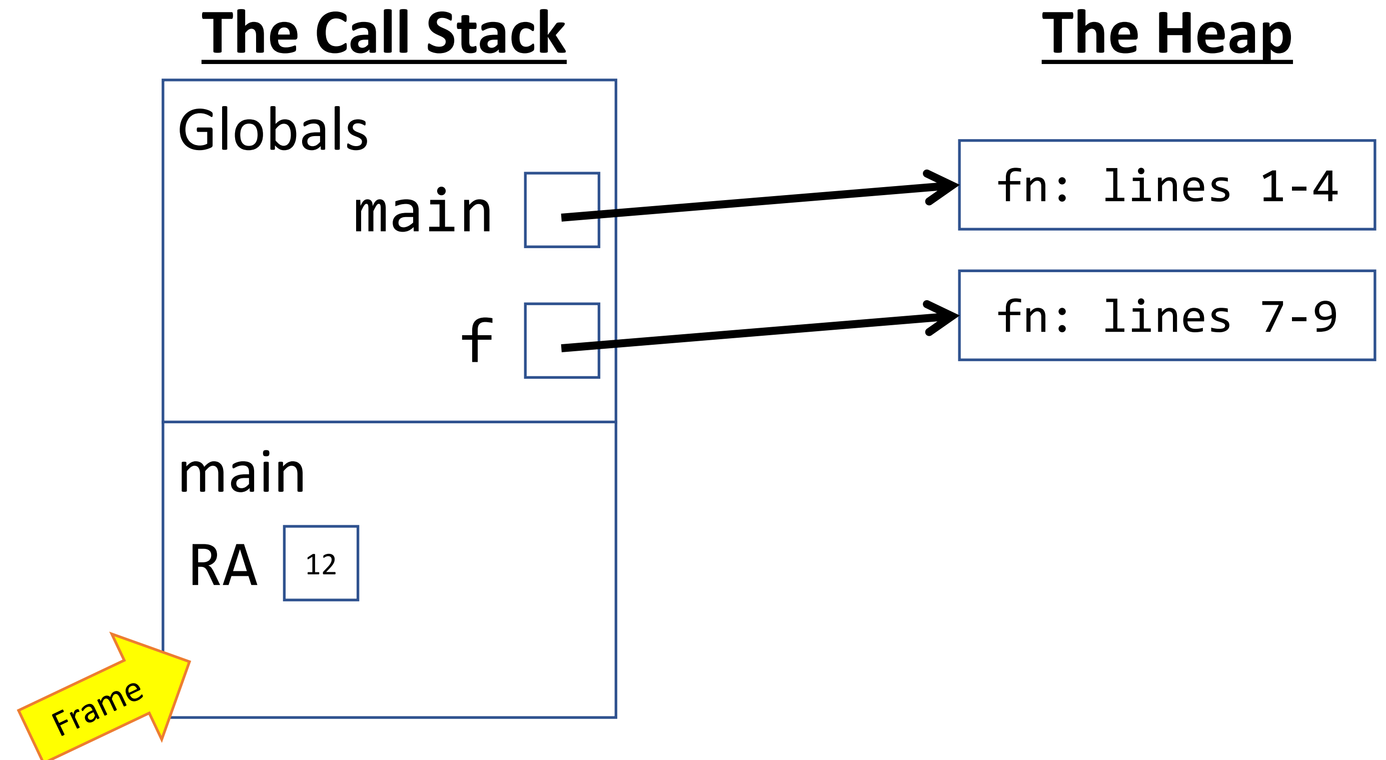


Function Call Process

First, evaluate the arguments in the call's parentheses. Confirm they match the definition's parameters. Here there are none!
Second, establish frame for the call on call stack (its namesake) with its:

1) name
2) the line number the call originated on and will return back to named return address ("RA")
3) parameters bound to argument values (main defines 0-parameters, so there are no parameters to bind)

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

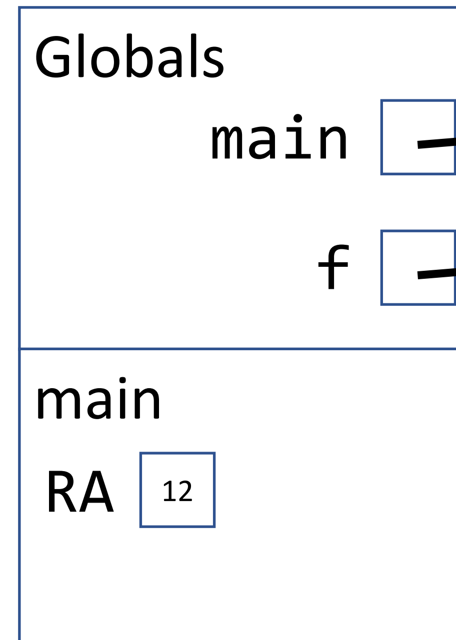


Function Call Jump

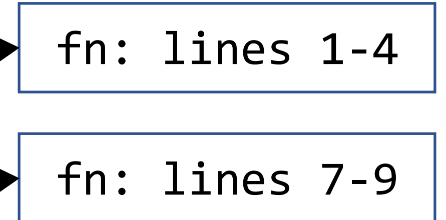
Once the process establishing a frame for a function call is complete, control *jumps* into the function and begins evaluating the statements in the function's body starting from the top statement. Notice the RA in main's frame maintains the bookmark control will return to.

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

The Call Stack



The Heap

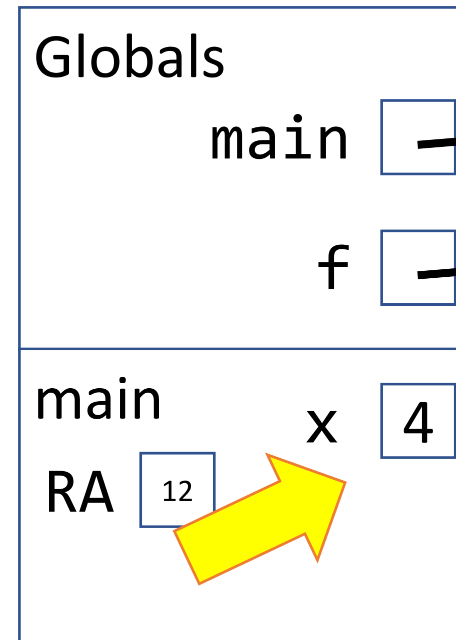


Variable Initialization - 1) Evaluate RHS, 2) Bind Name

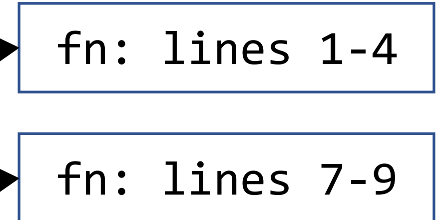
When a variable is initialized, *first* evaluate the value on the right. In this case it's the number literal 4, no more work is needed. Then, bind its name to its initial value in the current frame.

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

The Call Stack



The Heap

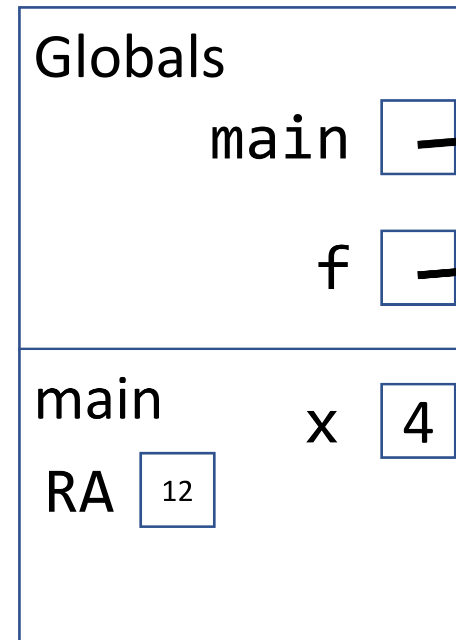


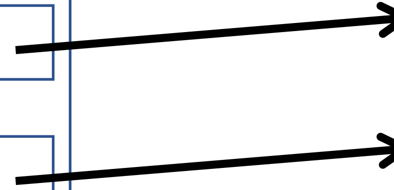
Variable Initialization - 1) Evaluate RHS

When a variable is initialized, *first* evaluate the ***expression*** on its right-hand side.
In this case it's a function call, so let's evaluate the function call.

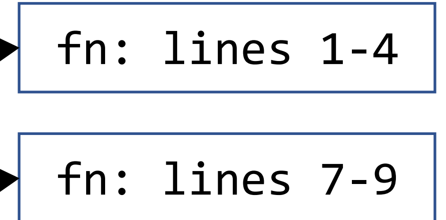
```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

The Call Stack





The Heap



Name Resolution: What is *f*?

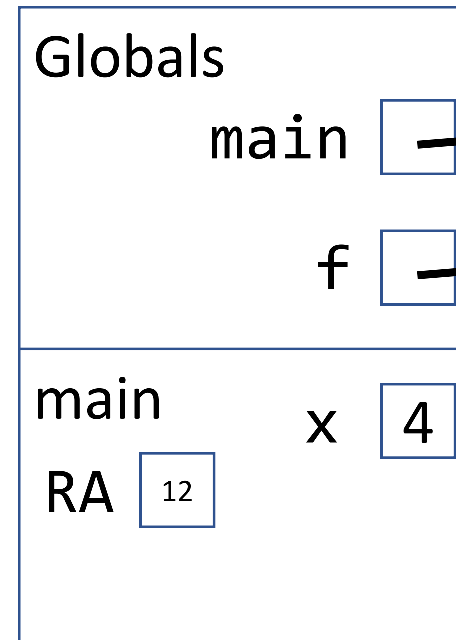
Look in the current frame (**main**) for the name. Is it bound there? No!

If the name is not in the current frame, next check the Globals frame. Is it bound there? Yes!

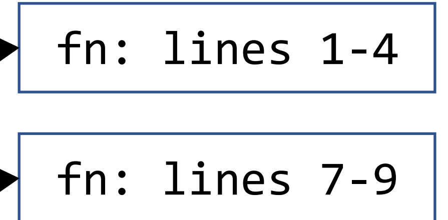
The name **f** is bound to the function defined on lines 7 through 9.

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

The Call Stack



The Heap



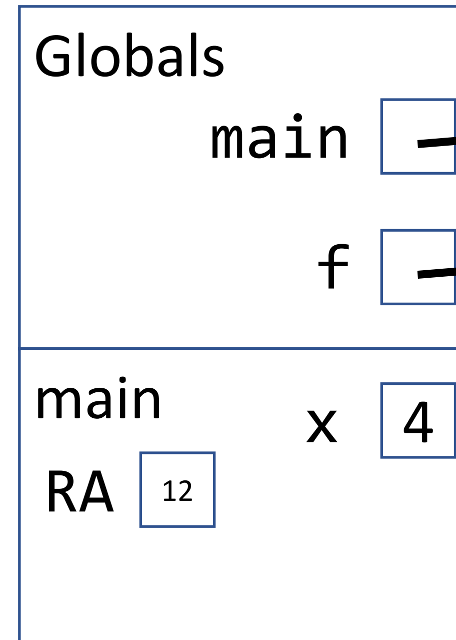
Function Call - Step 1) Evaluate Arguments

Before evaluating the function call to f, we must determine the values of each argument.
What is the name **x** bound to in **main**'s frame? We look in our diagram to see its value is 4.
Next, we confirm the number, types, and order of arguments match the parameters. They do.

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

x evaluates to 4

The Call Stack



The Heap

fn: lines 1-4

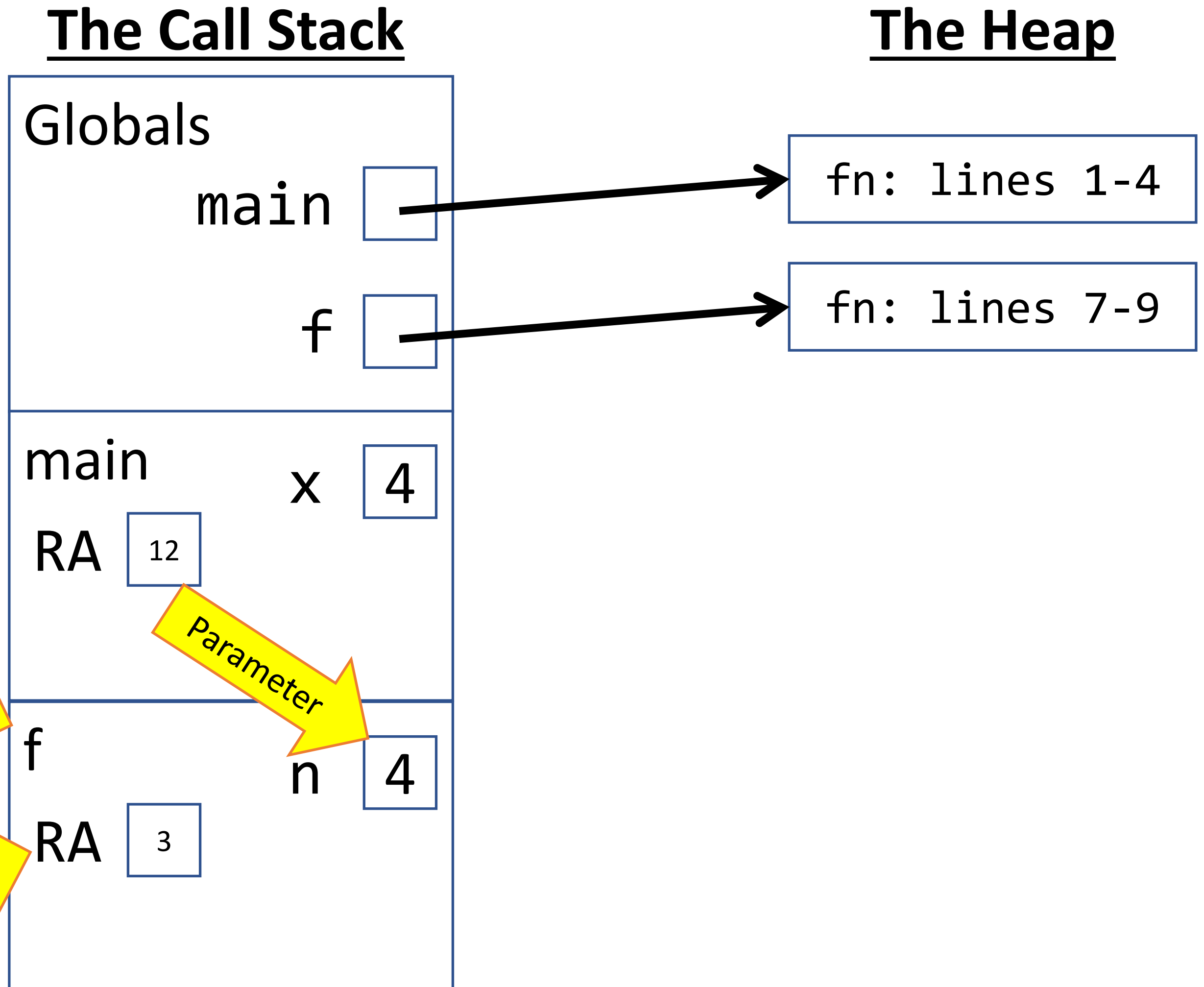
fn: lines 7-9

Function Call - Step 2) Establish a Frame

1. Give the frame the function's **name**. 2. Write down the line the function call occurred on as the frame's **Return Address (RA)**. 3. Bind argument values to the function's **parameters**.

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

x evaluated to 4

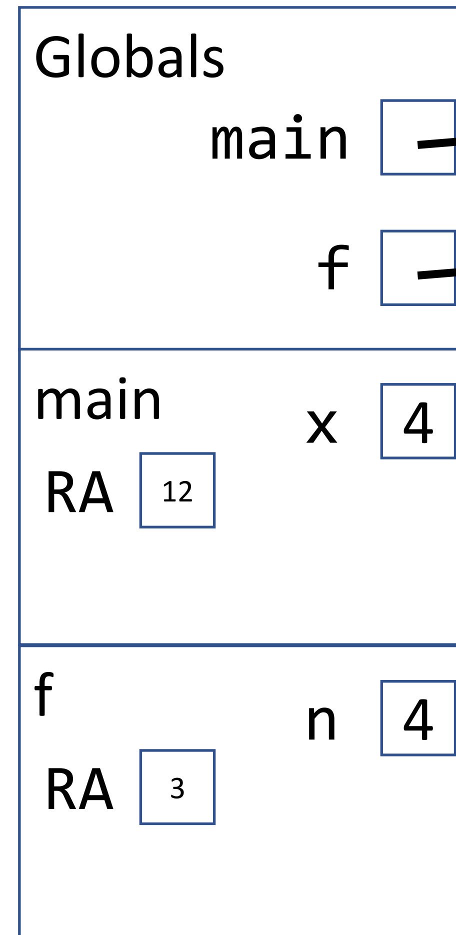


Function Call - Step 3) Jump to Function

Once the process establishing a frame for a function call is complete, control *jumps* into the function and begins evaluating the statements in the function's body starting from the top statement. Notice the RA in f's frame maintains the bookmark control will return to.

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

The Call Stack



The Heap

fn: lines 1-4

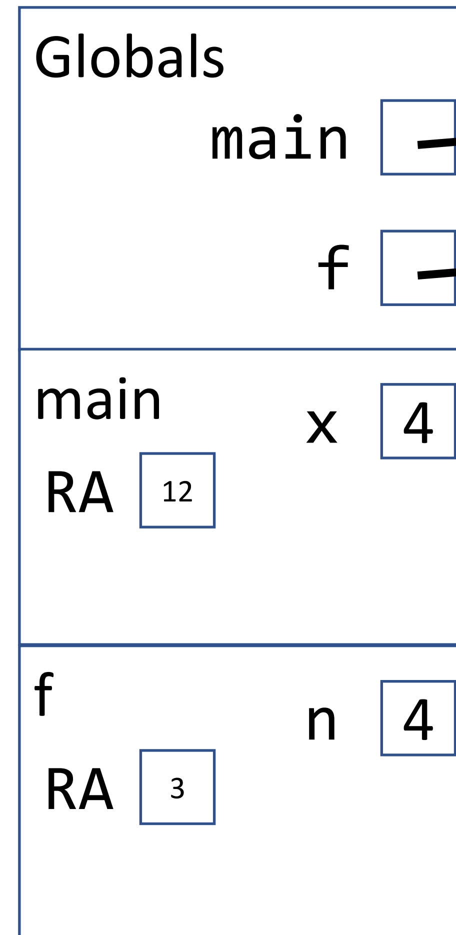
fn: lines 7-9

Variable Initialization - 1) Evaluate RHS

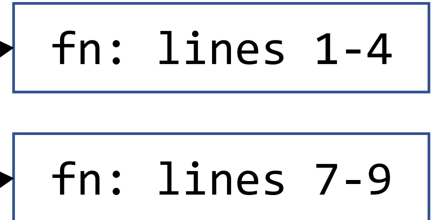
When a variable is initialized, *first* evaluate the expression on the right-hand side.
In this case it's an arithmetic `int` expression, so let's evaluate it first.

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

The Call Stack



The Heap



Name Resolution: What is *n*?

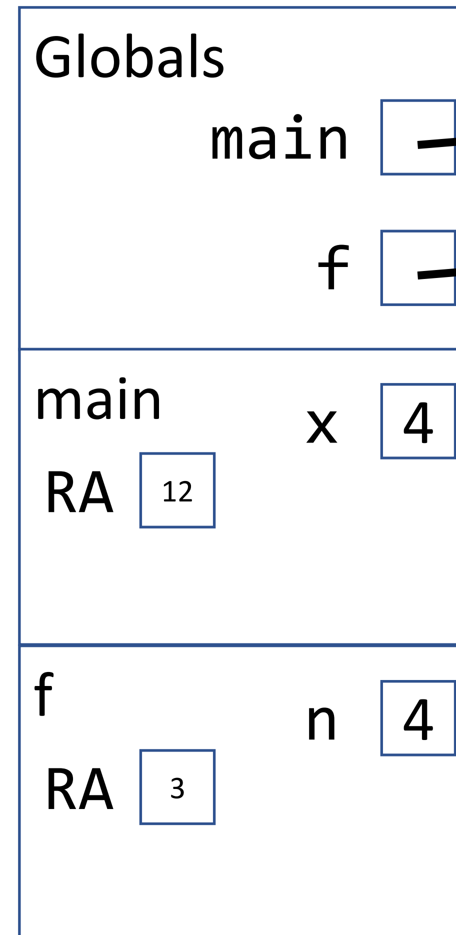
Look in the current frame (**f**) for the name **n**. Is it bound there? Yes!

The name **n** is bound to the int value **4**, so *accessing n evaluates to 4*.

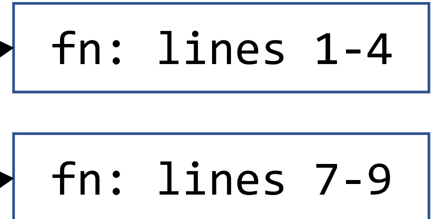
```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

n evaluates to 4

The Call Stack



The Heap



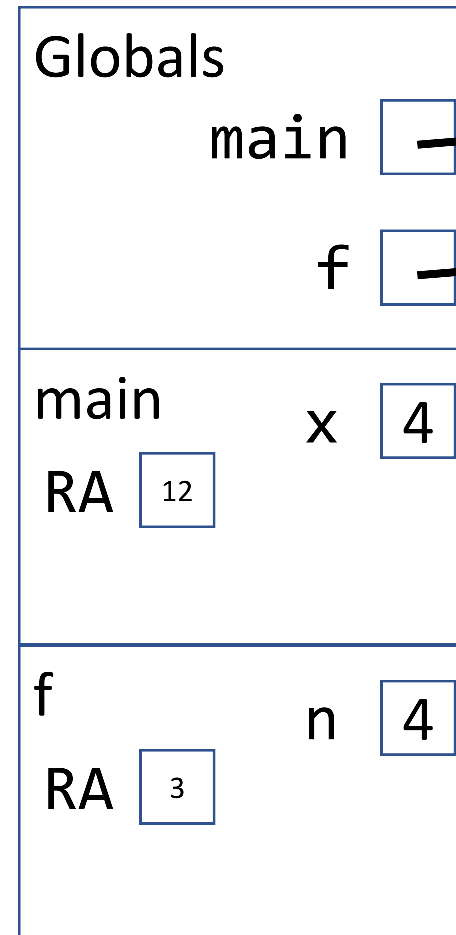
Expression Evaluation

Complete the evaluation of the right-hand side's expression: 4 + 1 is 5

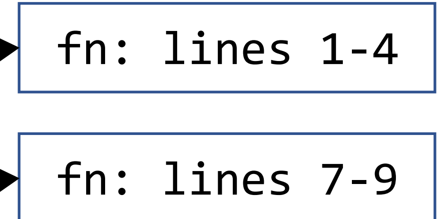
```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

n + 1 evaluates to 5

The Call Stack



The Heap



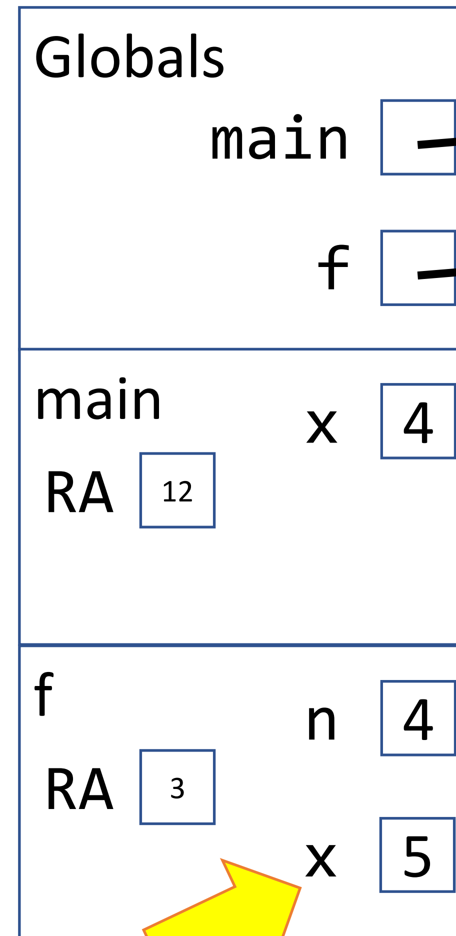
Variable Initialization - 2) Bind Name

After evaluating the right-hand side, bind the name x its initial value in the current frame. The current frame is **f**'s frame.

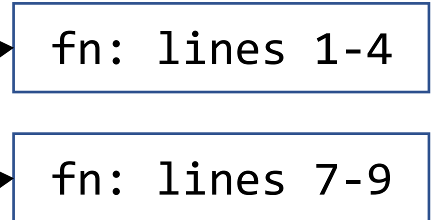
```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

n + 1 evaluated to 5

The Call Stack



The Heap



Notice the frame for *main* has its own variable x with a value of 4.

The frame for *f* also has its own variable x with *a different value*.

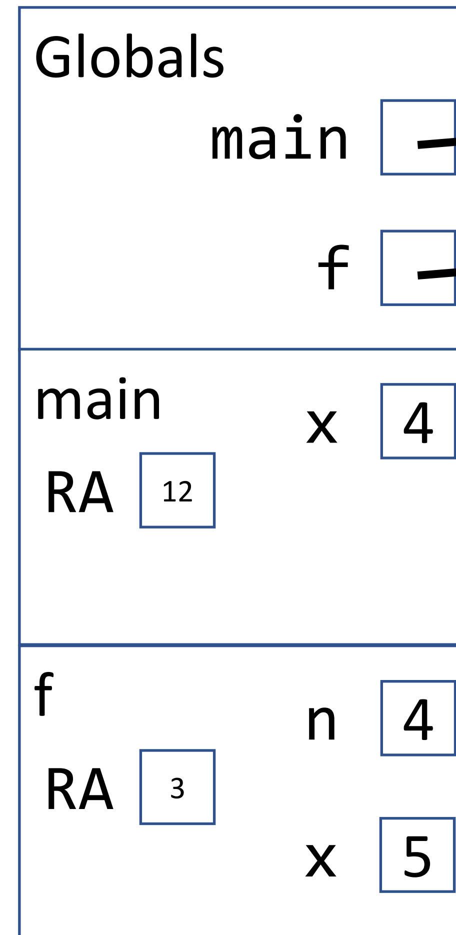
This is *entirely ok* and a *wonderful, powerful thing*. This means when you write functions you don't need to concern yourself with the variable names in other functions.

Return Statement - Step 1) Evaluate its Expression

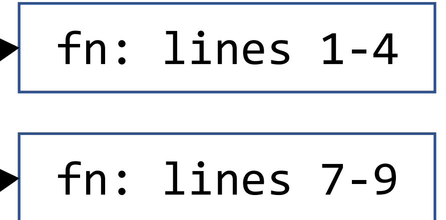
When a return statement is encountered, you must first evaluate expression it is returning. Let's focus on evaluating the expression **x**.

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

The Call Stack



The Heap



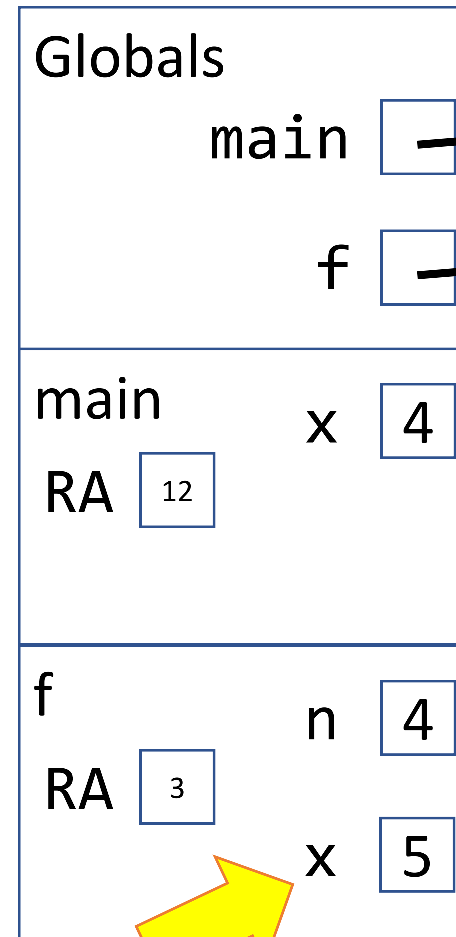
Name Resolution: What is *x*?

When a name is encountered in our program we look to the *current frame of the stack* for its value. In this case, **x**'s value in **f**'s frame is bound to **5.**

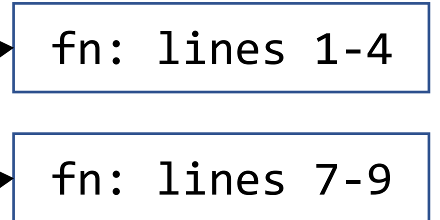
```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

x evaluates to 5

The Call Stack



The Heap



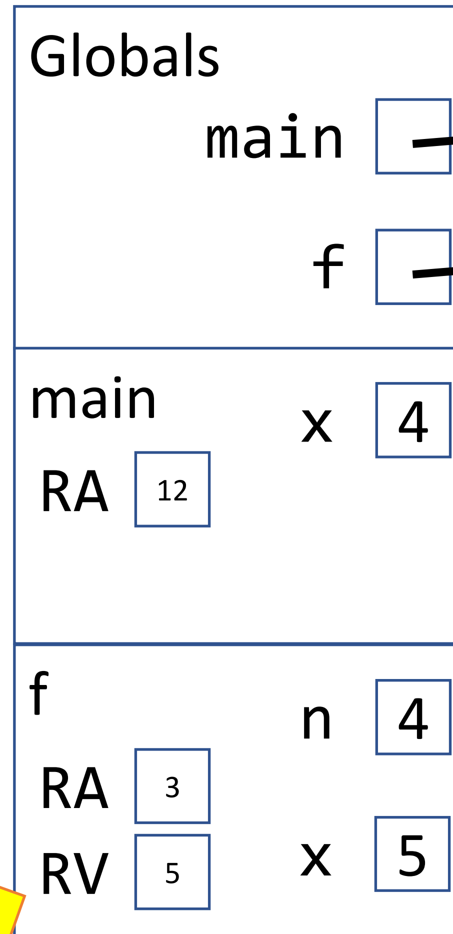
Return Statement - Step 2) Record its Value

When a return statement is encountered, once you know the value its expression evaluates to, enter the Return Value in a box named ***RV*** in the current frame.

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> 
08      x: int = n
09      return x
10
11
12  main()
```

x evaluated to 5

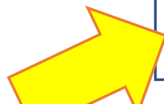
The Call Stack



The Heap

fn: lines 1-4

fn: lines 7-9



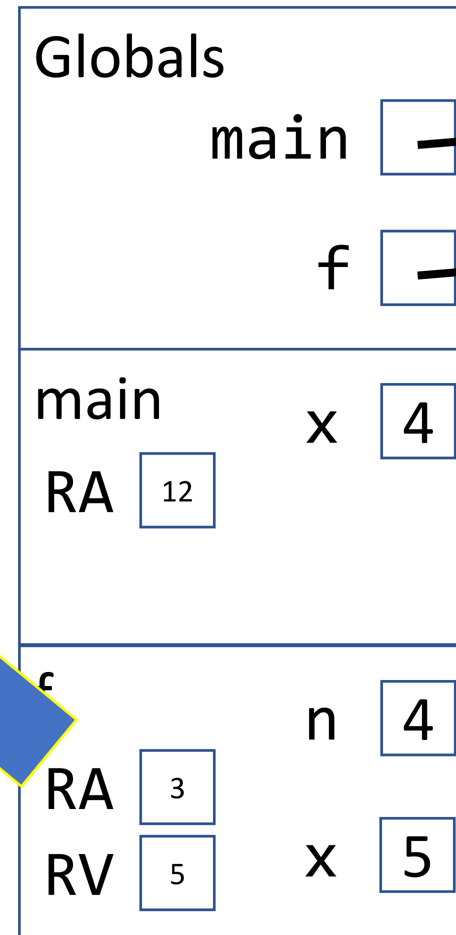
Return Statement - Step 3) Send RV back to RA

The returned value is then "*returned*" to the return address where the call originated. The originating call expression evaluates to RV. Back in main, this line is evaluated as **`y: int = 5`**

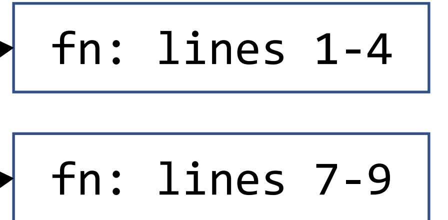
```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

f(x) evaluated to 5

The Call Stack



The Heap



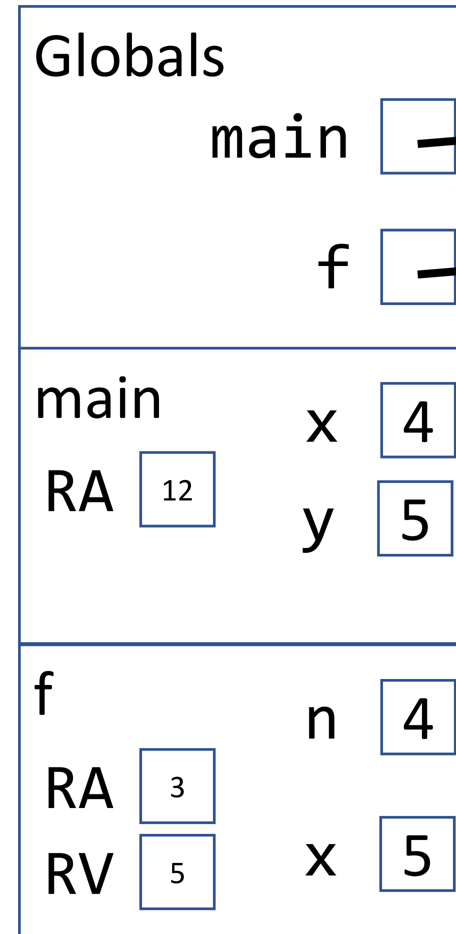
Variable Initialization - 2) Bind Name

Now that we've evaluated the right-hand side, we add an entry for the newly declared variable **y** to the current frame **main**.

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

f(x) evaluated to 5

The Call Stack



The Heap

How can you tell what the **current frame** of execution is?

The current frame is always the *lowest* frame that *has not returned.* So, if a frame has an *RV* entry, that frame is ignored.

Behind the scenes in your computer, once a function call returns its frame is deleted. When working on paper, though, it is helpful to keep track of all the work it took to arrive at a given position in our program.

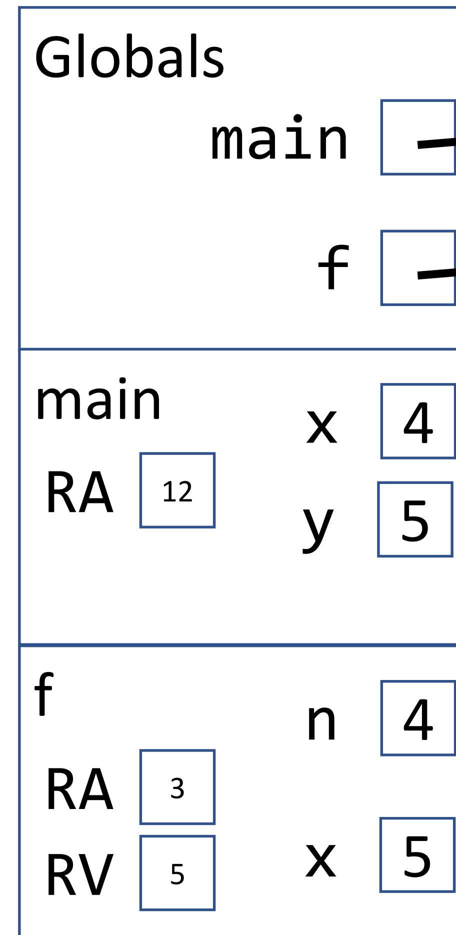
Print Function Call

A call to print goes through *the exact same steps* as the other function calls.

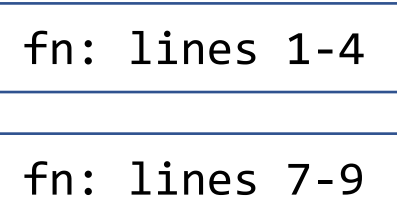
Name resolution? It's built-in! There are rules for resolving built-in functions, too. Not your concern for now.

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

The Call Stack



The Heap

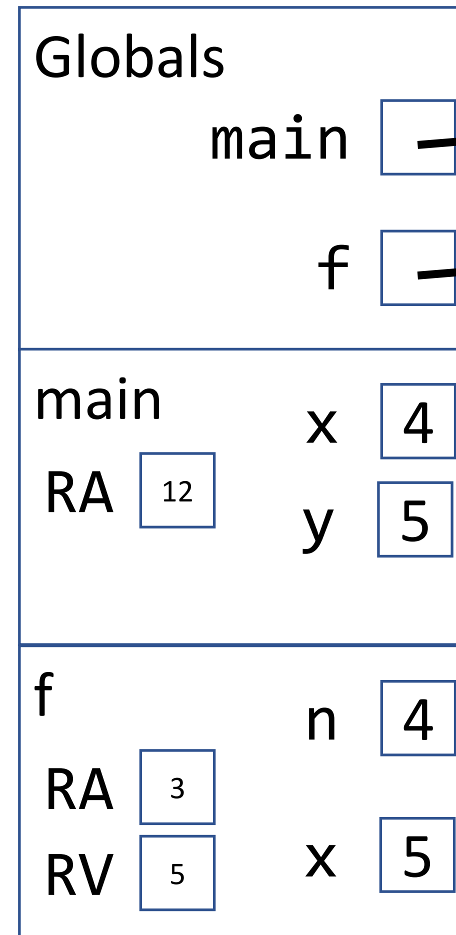


Print Function Call - 1) Evaluate Arguments

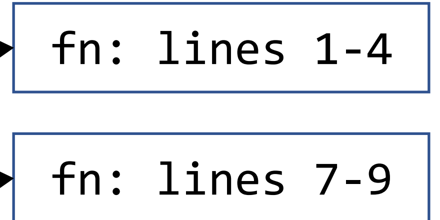
Before evaluating the function call to print, we must determine the values of each argument. What is the name **x** bound to in **main**'s frame? We look in our diagram to see its value is **4**! Convince yourself of it. Next, we look for y and see it is bound to 5.

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

The Call Stack



The Heap



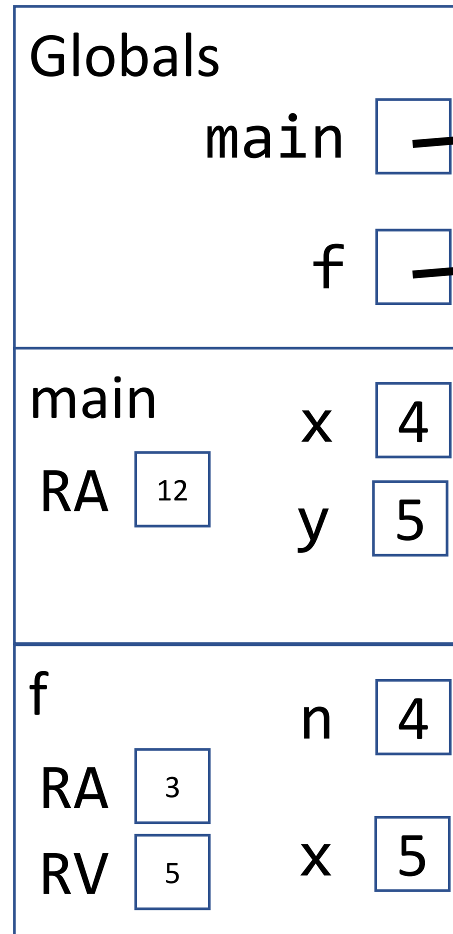
Print Function Call - 2) Establish a Frame

For functions defined outside of the program we are tracing we *will "abstract away"* their function call frames since we do not have their code to trace and we're confident in their correctness and purpose. The `print` function emits output, which is useful to keep track of.

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

evaluated to `print(4, 5)`

The Call Stack



The Heap

fn: lines 1-4

fn: lines 7-9

Output

4 5

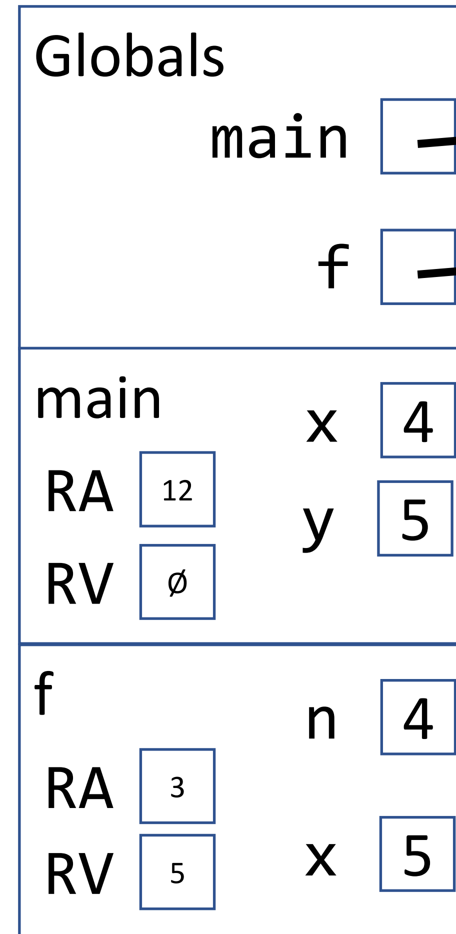
When you provide multiple arguments to the print function, separated by commas, they are printed on the same line and separated by a space.

End of `main` or any function that returns None

When our program reaches the end of a function that returns **None**, notice it has no *return statement.*
It's Return Value is *None.* We use the empty set notation Ø as a convention of representing *None*.
The processor would jump back to the return address at line 12 and reach the end of the program.

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

The Call Stack



The Heap

fn: lines 1-4

fn: lines 7-9

Output

4 5

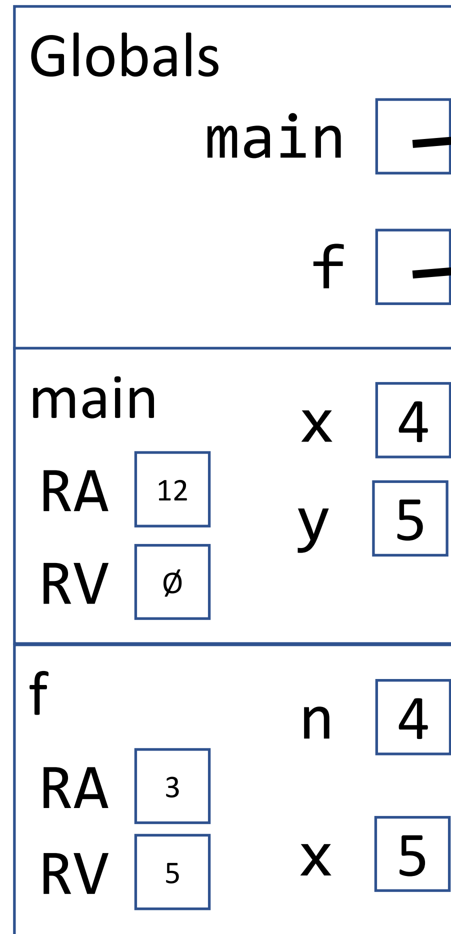
Functions that return None - Send RV back to RA

The returned value is then "*returned*" to the return address where the call originated. The originating call expression evaluates to RV. In this case, line 12 evaluates to None.

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

main() evaluated to None / Ø

The Call Stack



The Heap

fn: lines 1-4

fn: lines 7-9

Output

4 5

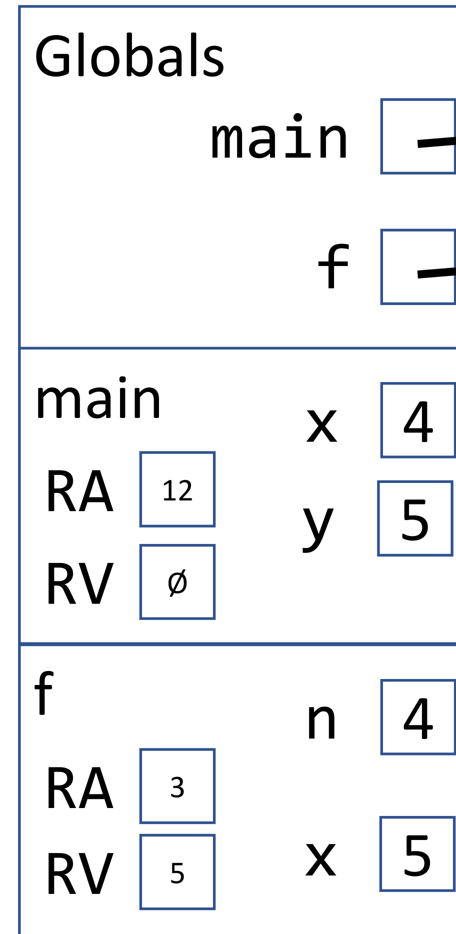
End of Program

Fin. The execution of this program is complete!

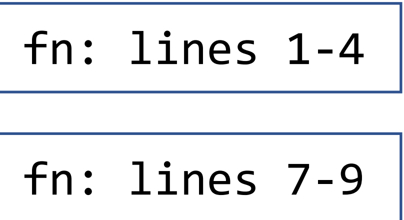
Note, if there *were* additional statements after the call to main()... they would evaluate just like anything else!

```
01  def main() -> None:
02      x: int = 4
03      y: int = f(x)
04      print(x, y)
05
06
07  def f(n: int) -> int:
08      x: int = n + 1
09      return x
10
11
12  main()
```

The Call Stack



The Heap



Output

4 5