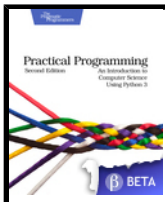


- 3.5 Tracing Function Calls in the Memory Model
- *Optional reading*

Functions, Variables, and the Call Stack

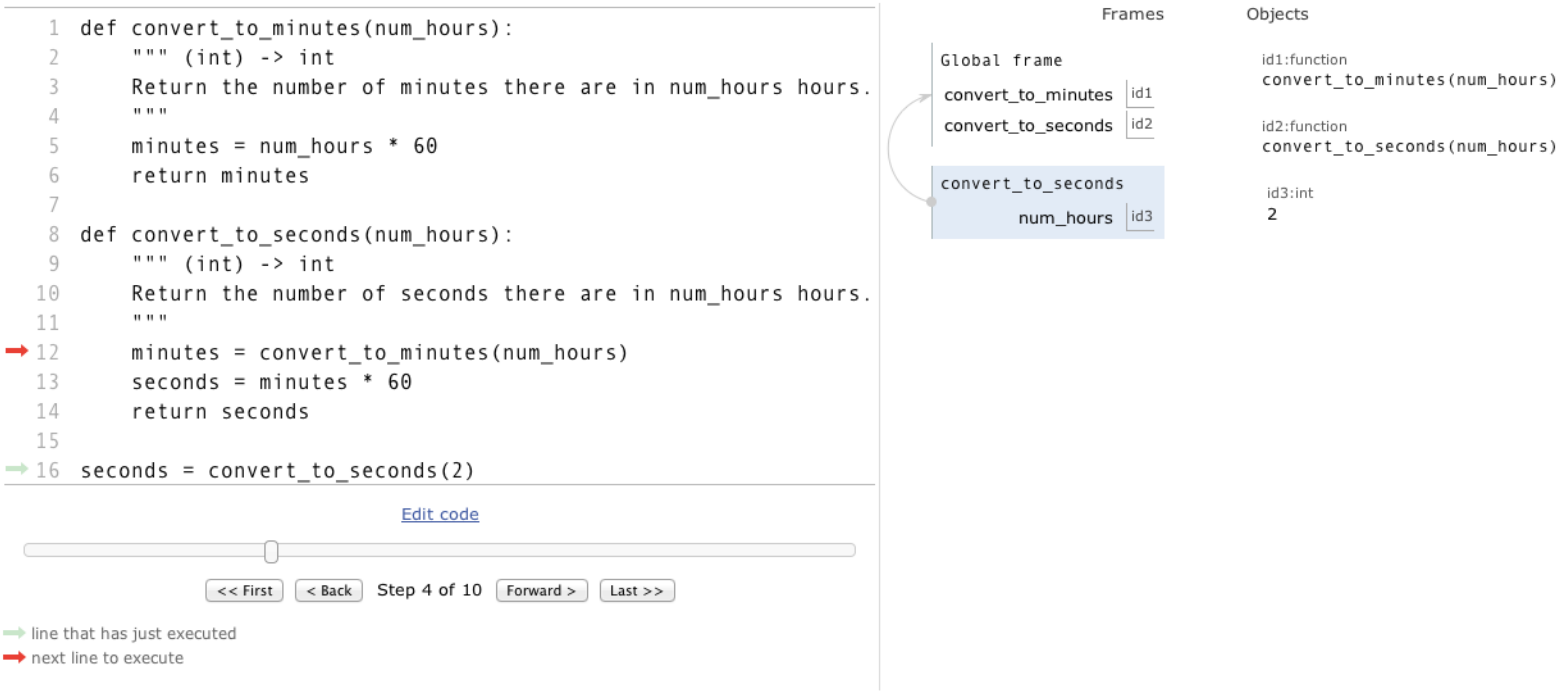
Understanding Scope

Below is an explanation and review of the example used in the video.

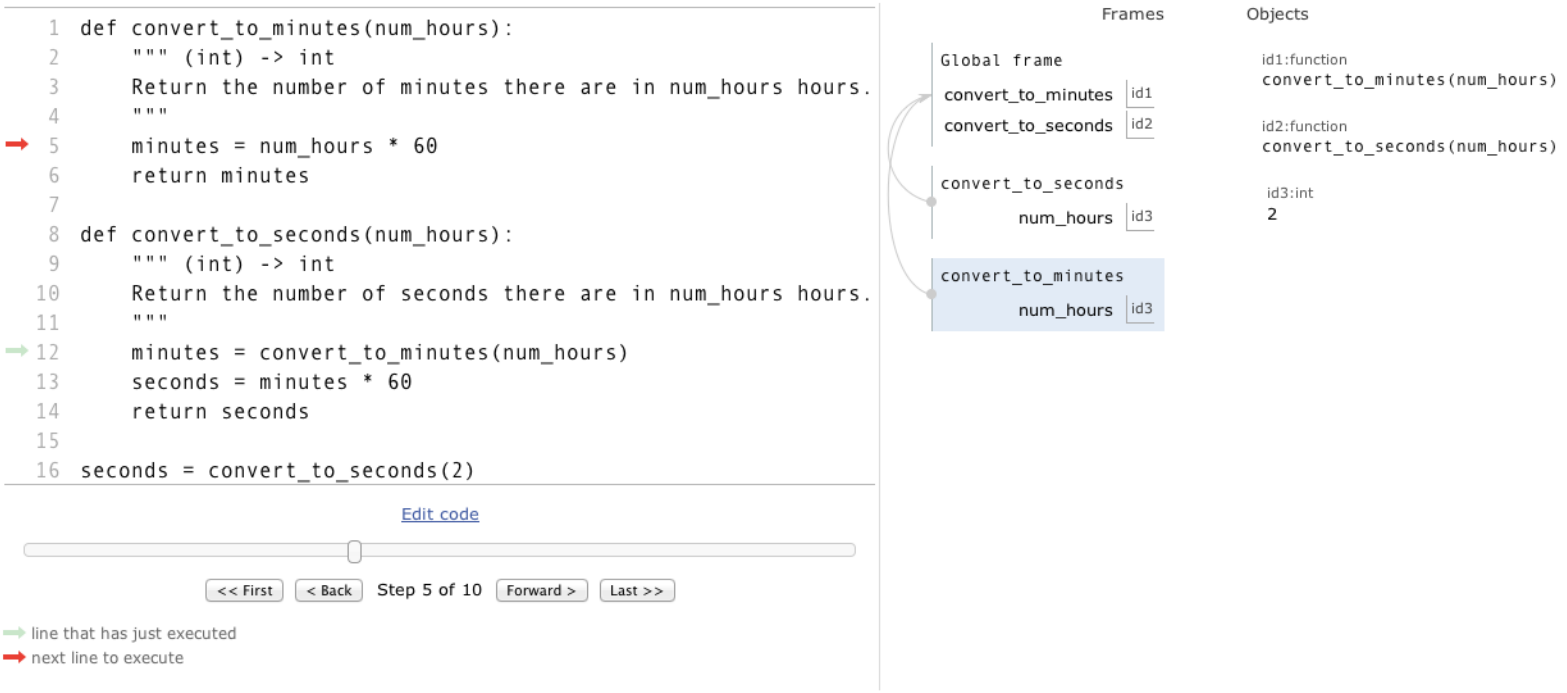
```
def convert_to_minutes(num_hours):
    """ (int) -> int
    Return the number of minutes there are in num_hours hours.
    """
    minutes = num_hours * 60
    return minutes
def convert_to_seconds(num_hours):
    """ (int) -> int
    Return the number of seconds there are in num_hours hours.
    """
    minutes = convert_to_minutes(num_hours)
    seconds = minutes * 60
    return seconds
seconds = convert_to_seconds(2)
```

Python defines the first two functions, creates objects for them in the heap, and, in the stack frame for the main program, creates variables that refer to those function objects.

After that, it executes the assignment statement on line 16. The right-hand side of the assignment statement is a function call so we evaluate the argument, 2, first. The frame for `convert_to_seconds` will appear on the call stack. The parameter, `num_hours`, will refer to the value 2.

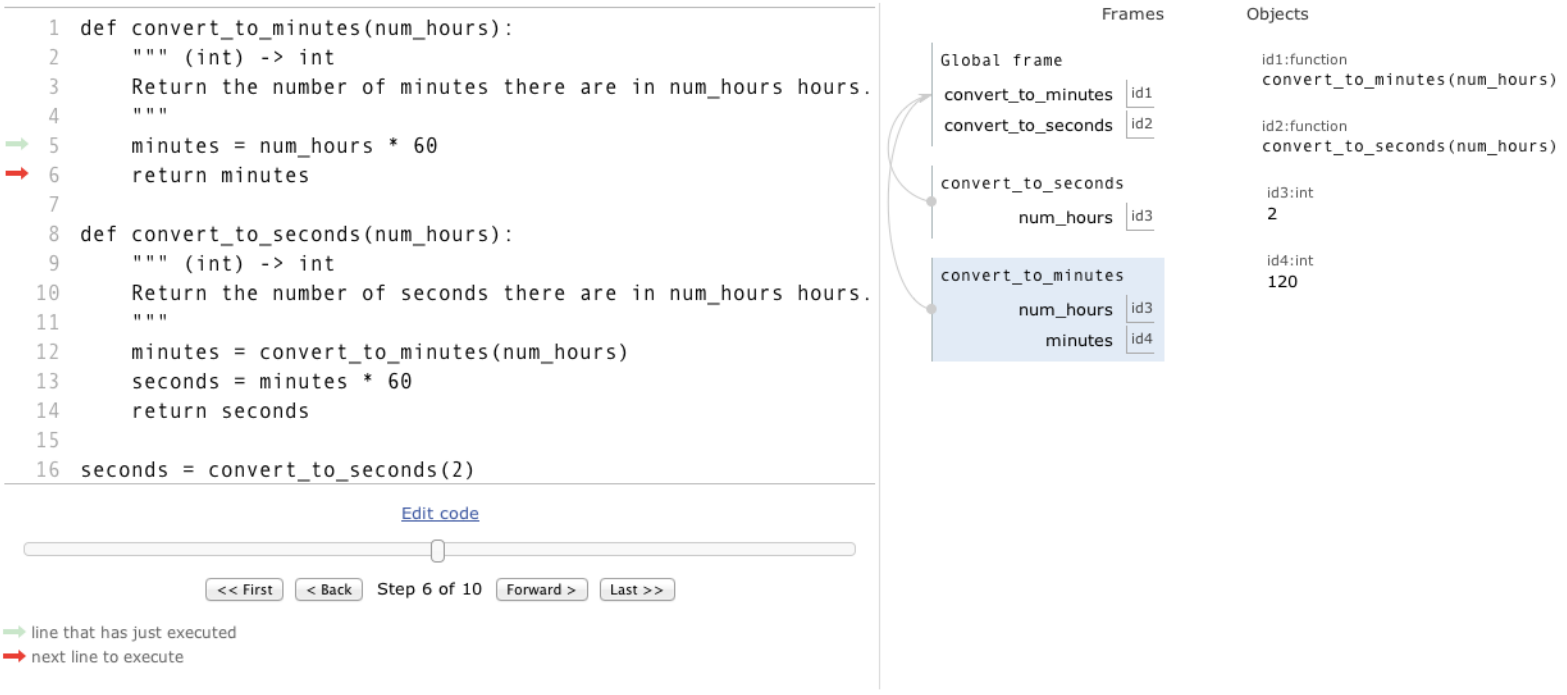


The first statement in function `convert_to_seconds` is an assignment statement. Again, we evaluate the expression on the right-hand side. This is a function call so we evaluate the argument, `num_hours`. This produces the value 2. A stack frame for function `convert_to_minutes` is created on the call stack. Python stores the memory address of 2 in the parameter for `convert_to_minutes`, which also happens to be called `num_hours`.

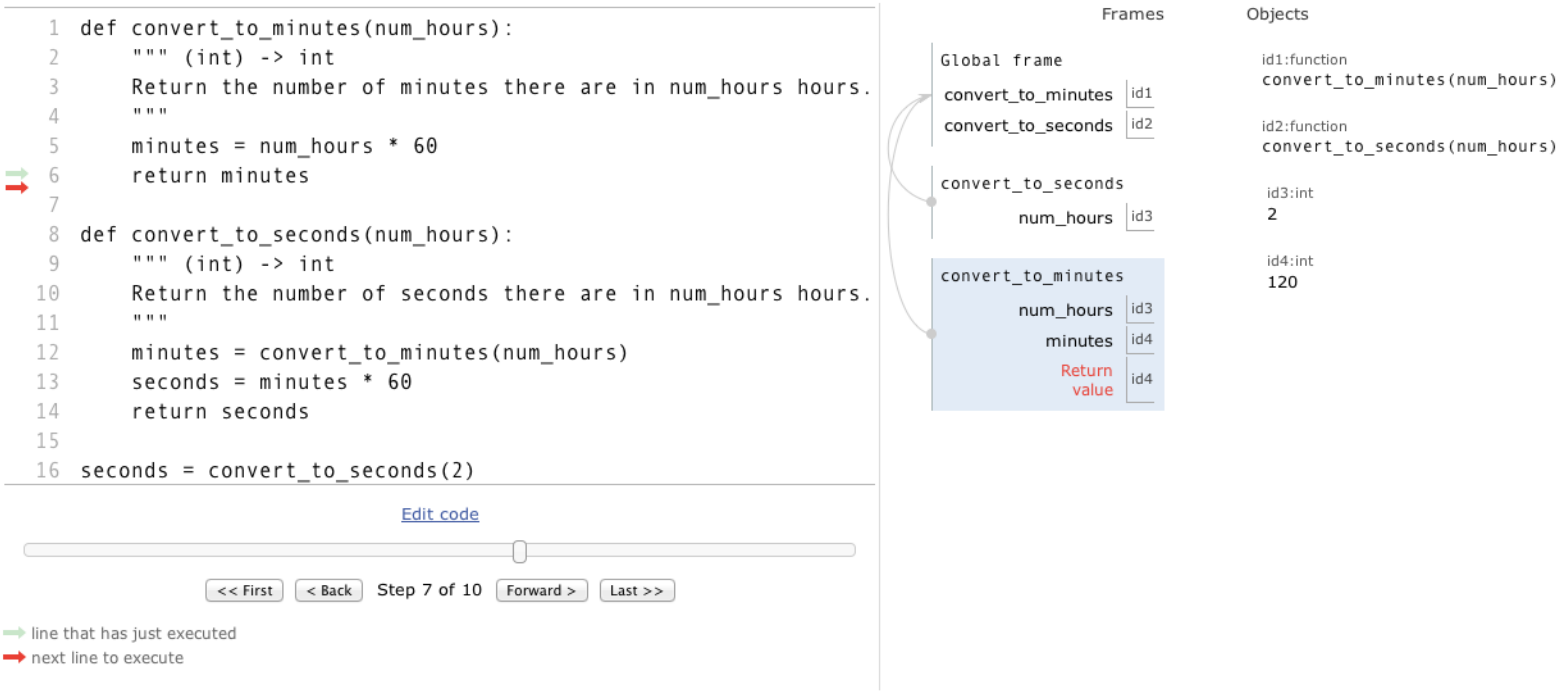


We now see that there are two variables called `num_hours` in the call stack; one is in `convert_to_minutes` and the other is in `convert_to_seconds`.

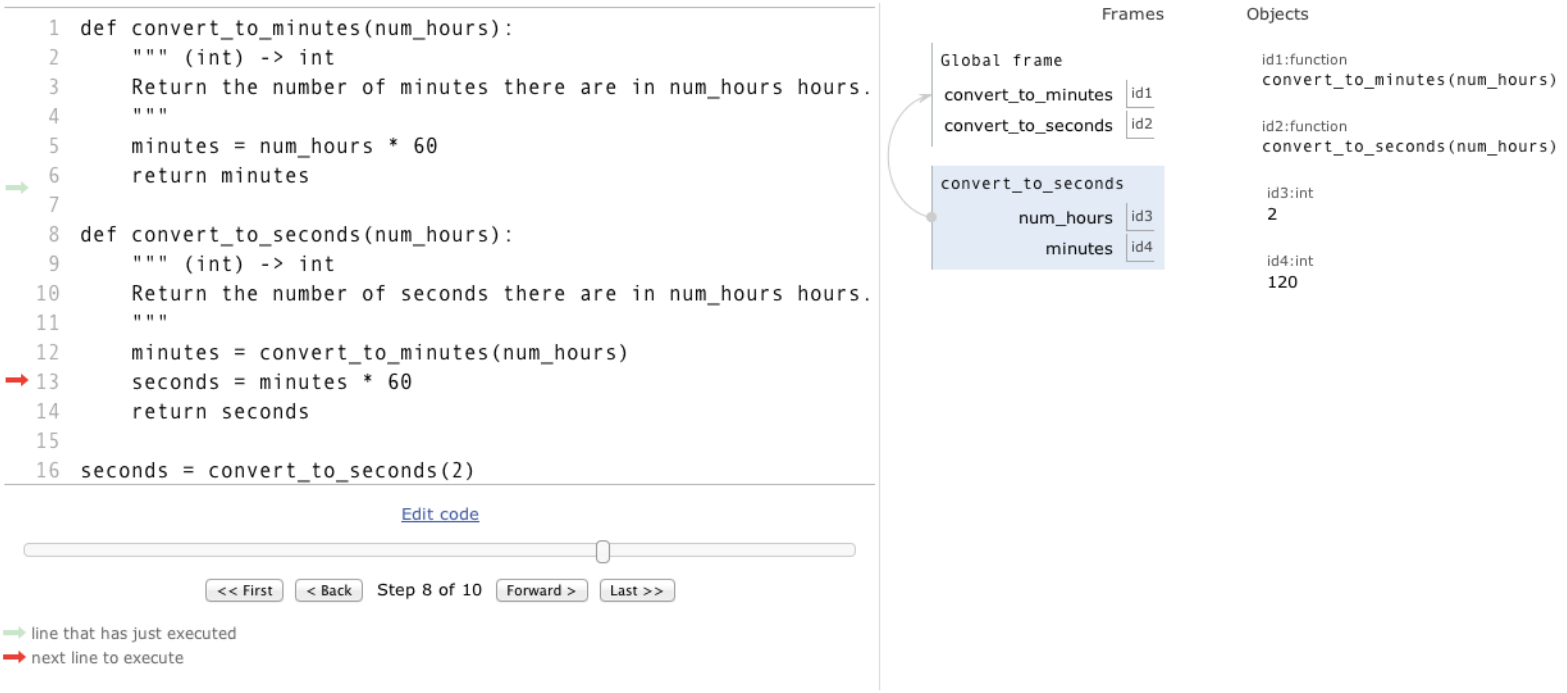
The next line of code Python executes is `minutes = num_hours * 60`. However, which instance of `num_hours` will be used? Python always uses the variable in the current stack frame. With an assignment statement, if the variable does not exist in the current stack frame, Python creates it. So, once `num_hours * 60` is evaluated, variable `minutes` is created in the current stack frame.



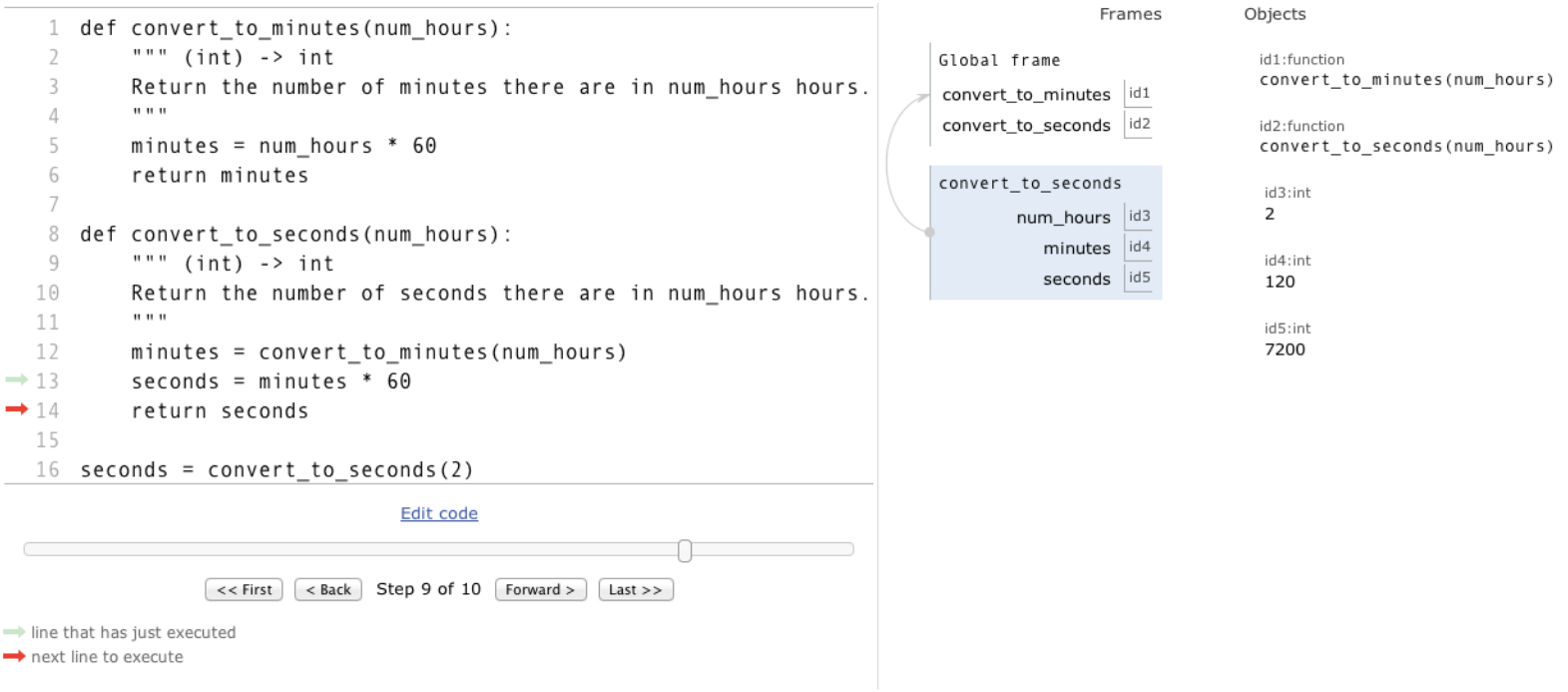
The last line of the function is `return minutes`. Once this statement is complete, Python will return to the frame just underneath the top of the call stack.



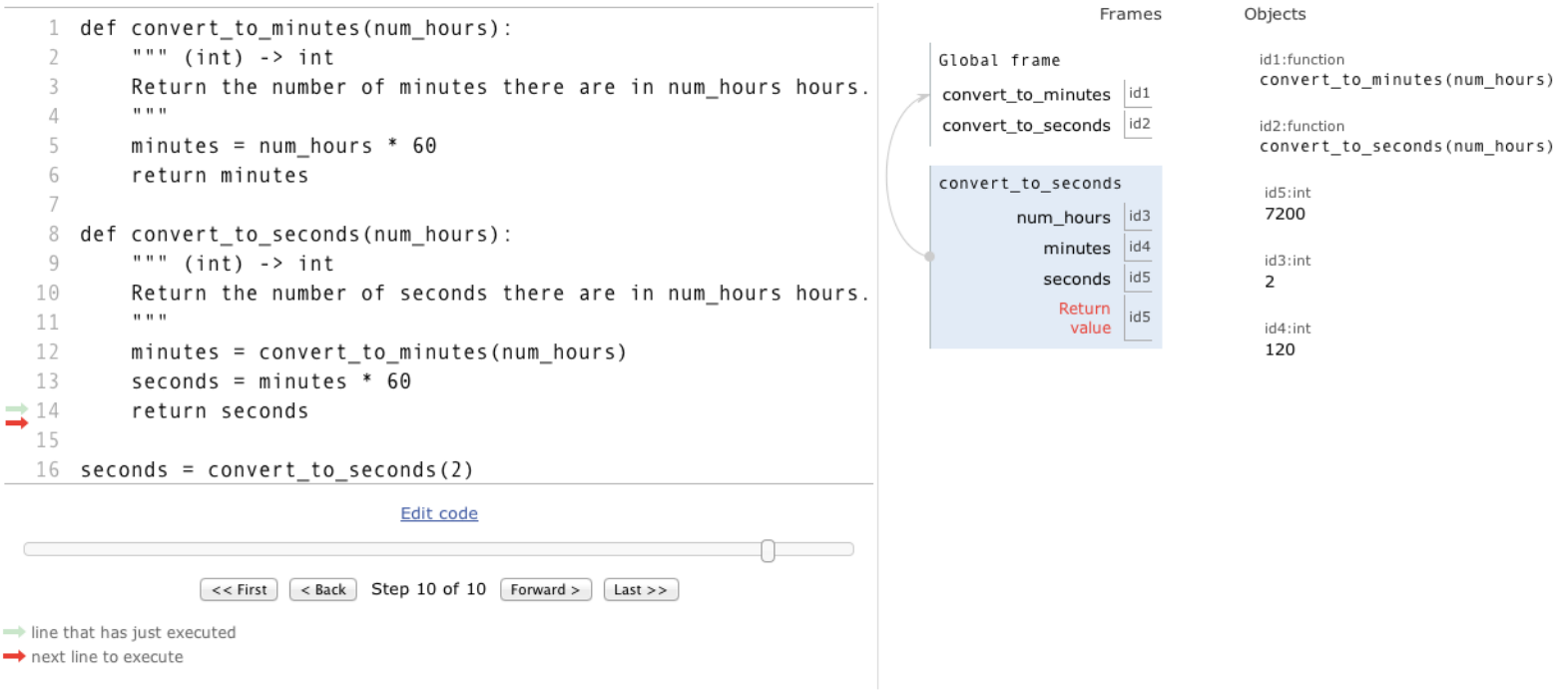
So, Python is going to produce the value `120`, remove the current stack frame, create a new variable called `minutes` in the stack frame for `convert_to_seconds`, and store the memory adress of `120` in that variable.



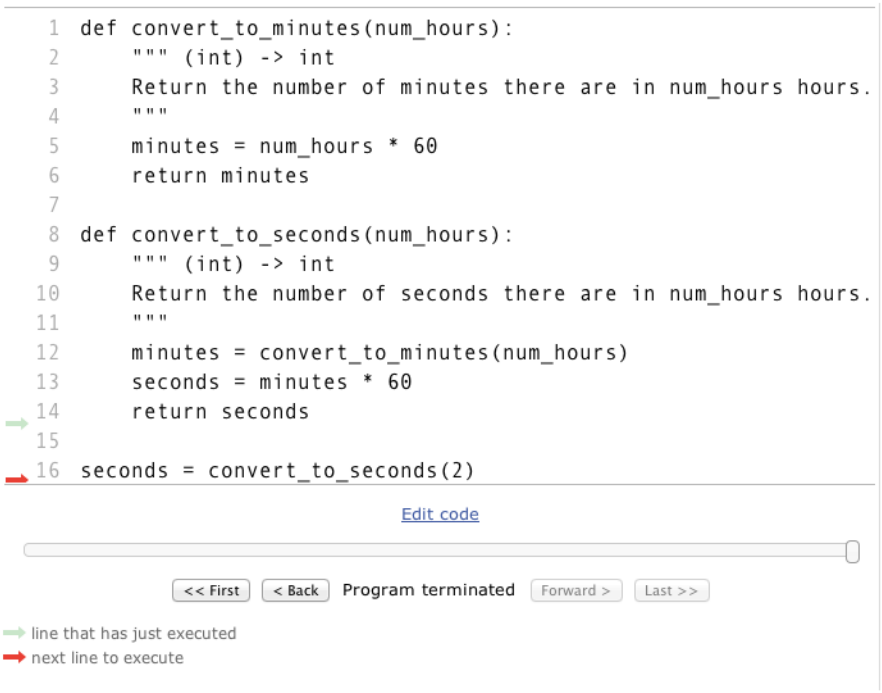
Python then executes `seconds = minutes * 60`. Python evaluates the right-hand side, which produces `7200`, and stores the memory address of that value in variable `seconds`. Since this variable does not exist yet, Python creates it in the current stack frame.

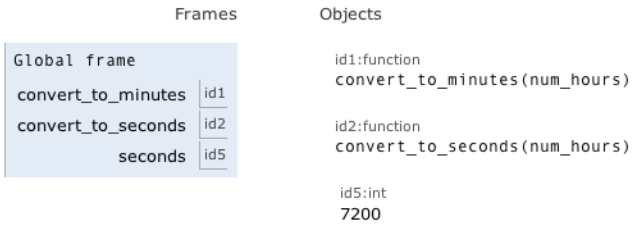


Next is a return statement. Like we saw above, that is going to return control back to the the main module.



Once the frame for `convert_to_seconds` is removed, the assignment statement on line 16 (which has been paused a long time!) is completed, and a new variable `seconds` is created in the stack frame for the main program.





Notes and assignment and return statements

Assignment statement and computer memory

```
variable = expression
```

If a variable does not exist in the current stack frame, Python creates it.

Return statement and computer memory

```
return expression
```

In addition to evaluating the expression and yielding its value, `return` also erases the stack frame on top of the call stack.

Jennifer Campbell ? Paul Gries
University of Toronto
