Program State and Program Execution

CSE 1310 – Introduction to Computers and Programming
Vassilis Athitsos
University of Texas at Arlington

Program State vs. Program History

- The state of the program contains all the information that we need to determine what the program will do next.
- The state of the program is typically **much more simple** than the history of the program, which describes everything than the program did from the beginning till now.
- The computer always keeps track of program state.
- As a rule (with rare exceptions) the computer does **NOT** keep track of program history.

Understanding Program States

- Understanding any piece of code (whether it is an if statement, while loop, function call) means understanding how that piece of code changes program state.
  - Code that does not change the program state is useless.
- Conversely, if you do not understand precisely how some piece of code changes program state, **you do not understand that piece of code.**

Defining a Program State

- A program state consists of:
  - **Namespaces**, that associate variable names with values.
    - Only one namespace is visible at each moment in the program execution, but multiple other namespaces may still be in memory, and can become visible later.
  - **A calling stack**, which describes what line(s) of code we are currently executing.

The Calling Stack

- The **calling stack** describes what line(s) of code we are executing right now. The calling stack contains a **sequence** of program lines:
  - The current line L_1 we are executing.
  - The line L_2 that made the last function call.
  - The line L_3 that made the second-to-last function call.
  - And so on, until some line L_N that belongs to the main code.
- Important: for every line in the calling stack, additional information is needed:
  - Exactly **which function call** is being evaluated (lines of code may include multiple function calls).
  - Which subexpressions **have already been evaluated** and what values they returned.

The Calling Stack

- The **calling stack** describes what line(s) of code we are executing right now. The calling stack contains a **sequence** of program lines:
  - The current line L_1 we are executing.
  - The line L_2 that made the last function call.
  - The line L_3 that made the second-to-last function call.
  - And so on, until some line L_N that belongs to the main code.
- Important: for every line in the calling stack, additional information is needed:
  - Exactly **which function call** is being evaluated (lines of code may include multiple function calls).
  - Which subexpressions **have already been evaluated** and what values they returned. WE WILL GET BACK TO THIS TOPIC, IT IS IMPORTANT.

Correspondences between Namespaces and Calling Stack

- At any point in the program, the number of namespaces is equal to the number of lines in the calling stack.
- Each line in the calling stack corresponds to a different namespace.
  - Why? Because each line in the calling stack corresponds to a different function call.

The Order of Evaluating Subexpressions

- For every line in the calling stack, we process it by evaluating its subexpressions, and using the resulting values.
- In what order do we evaluate subexpressions?
  - Evaluate simpler expressions before larger expressions that contain the simple ones.
- Does this specify a complete order? **NO**
- The order in which subexpressions are evaluated may matter (in bad code), but **should never matter in good code.**

The Order of Evaluating Subexpressions

- For every line in the calling stack, we process it by evaluating its subexpressions, and using the resulting values.
- In what order do we evaluate subexpressions?
  - Evaluate simpler expressions before larger expressions that contain the simple ones.
- Does this specify a complete order? **NO**
- The order in which subexpressions are evaluated may matter (in bad code), but **should never matter in good code.**
- Example: what will this print?

```
>>> list1 = [4, 5, 6]
>>> list1.pop() - list1.pop()
```

The Order of Evaluating Subexpressions

- For every line in the calling stack, we process it by evaluating its subexpressions, and using the resulting values.
- In what order do we evaluate subexpressions?
  - Evaluate simpler expressions before larger expressions that contain the simple ones.
- Does this specify a complete order? **NO**
- The order in which subexpressions are evaluated may matter (in bad code), but **should never matter in good code.**
- Example: what will this print?

```
>>> list1 = [4, 5, 6]
>>> list1.pop() - list1.pop()
```

It depends on whether the left pop or the right pop is evaluated first.

If left pop is evaluated first, result is 1.
If the right pop is evaluated first, result is -1.

THIS IS HORRIBLE CODE, DO NOT USE.

An Example of Program Execution

```
def cube(n):
    result = n*n*n
    return result

# start of main code
x = 3
result = cube(x) + cube(x+1)
print("result =", result)
```

An Example – Numbering Lines

To make it easy to refer to lines of code, we assign numbers to each line

```
Line 1:   def cube(n):
Line 2:       result = n*n*n
Line 3:       return result

# start of main code
Line 4:   x = 3
Line 5:   result = cube(x) + cube(x+1)
Line 6:   print("result =", result)
```

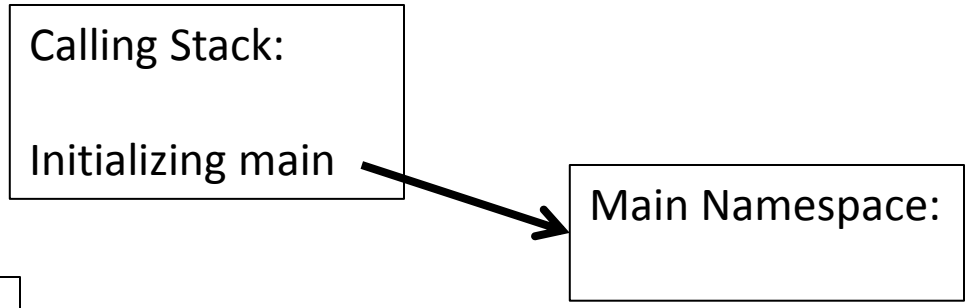
Program Execution

Where do we start from?

How do we initialize the program state?

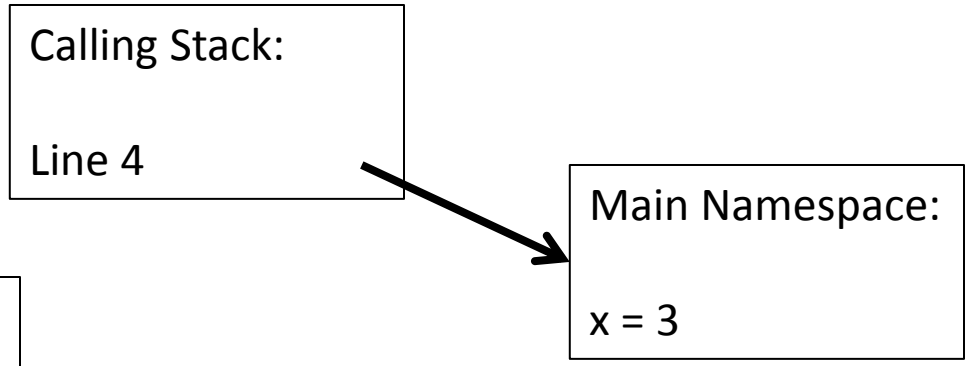
```
Line 1:   def cube(n):
Line 2:       result = n*n*n
Line 3:       return result

# start of main code
Line 4:   x = 3
Line 5:   result = cube(x) + cube(x+1)
Line 6:   print("result =", result)
```

Program Execution



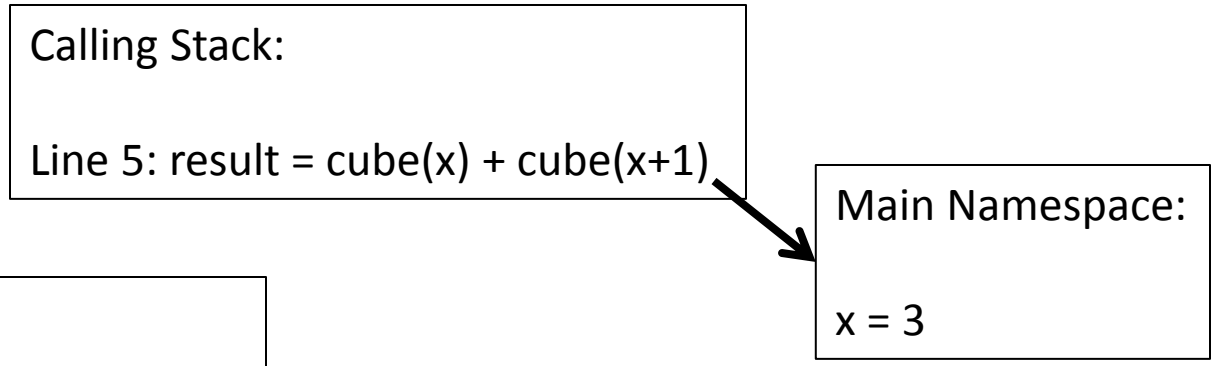

```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```

Program Execution



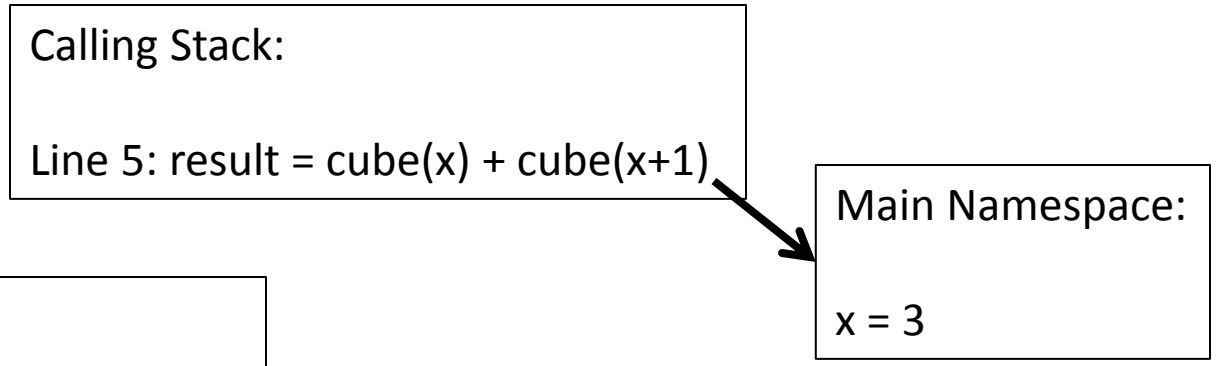

```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```

Program Execution




```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```

Program Execution

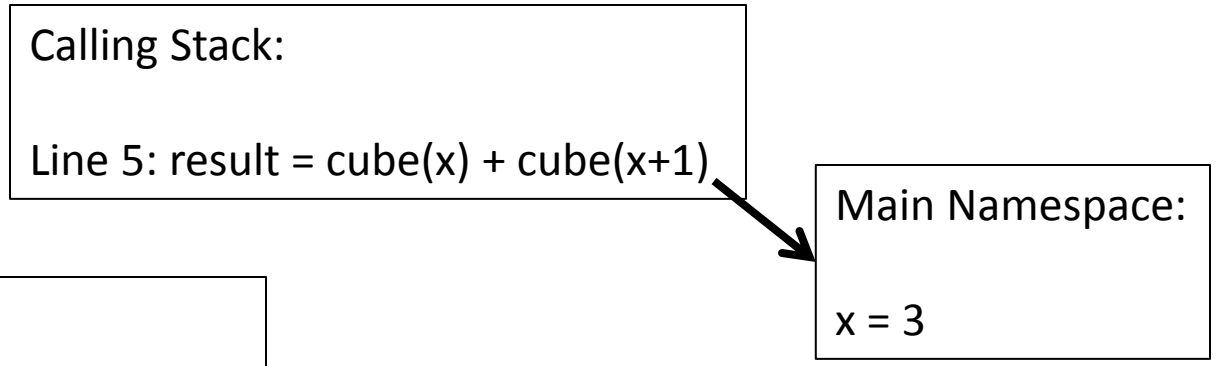



```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```

Note: it is not sufficient to just show Line 8 in the calling stack. We need to specify which subexpression we will work on.

Which subexpression should we choose?

Program Execution



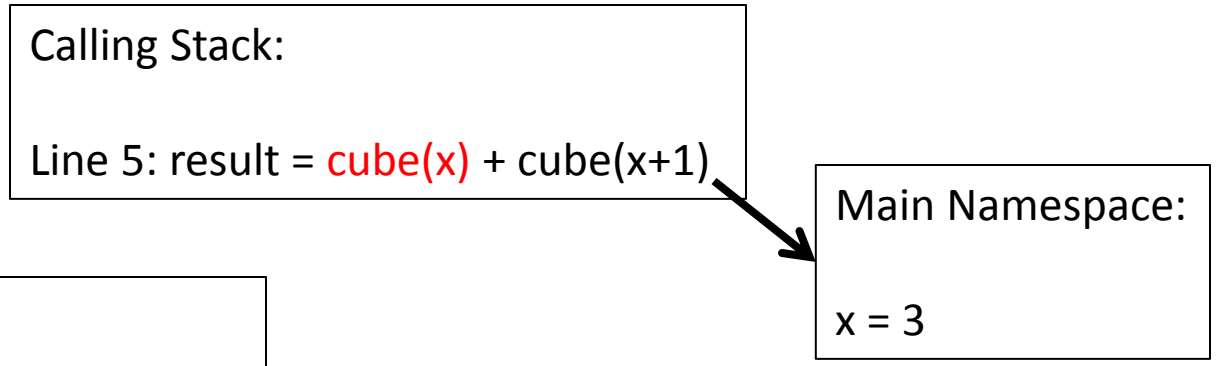

```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```

Note: it is not sufficient to just show Line 8 in the calling stack. We need to specify which subexpression we will work on.

Which subexpression should we choose?

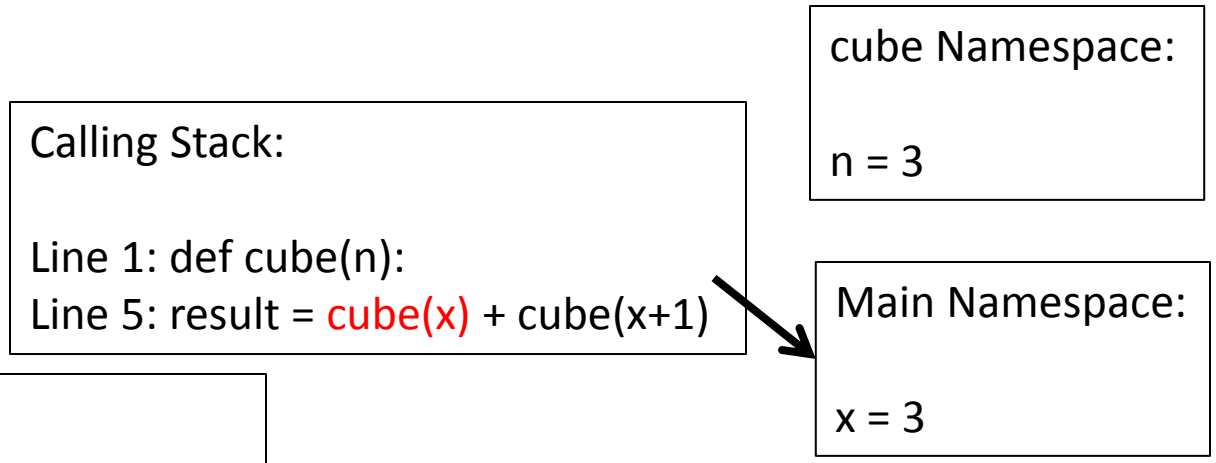
As long as the code follows good guidelines, the order does not matter.

Program Execution




```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```

Program Execution



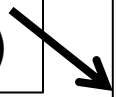

```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```

Program Execution

```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```

Calling Stack:

Line 2: result = n*n*n
Line 5: result = cube(x) + cube(x+1)



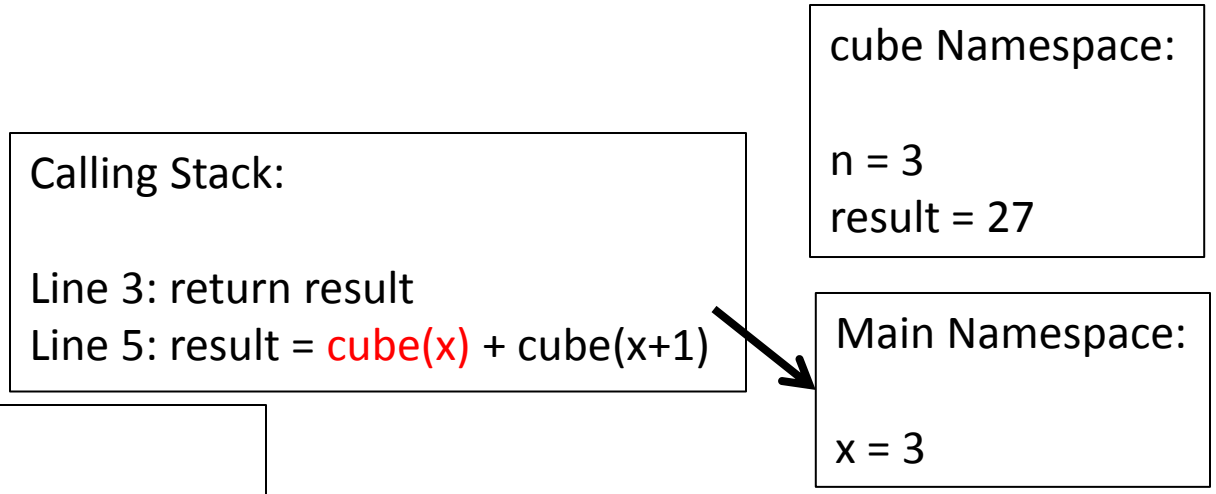
cube Namespace:

n = 3
result = 27

Main Namespace:

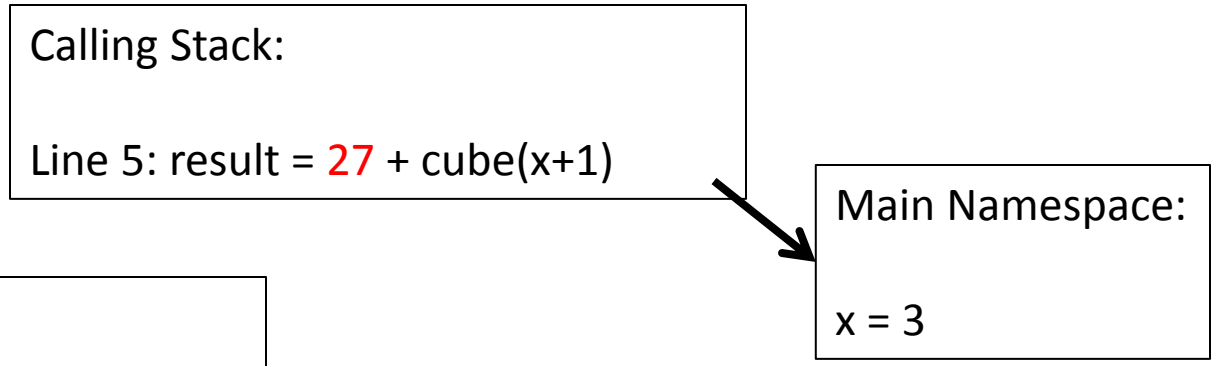
x = 3

Program Execution




```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```

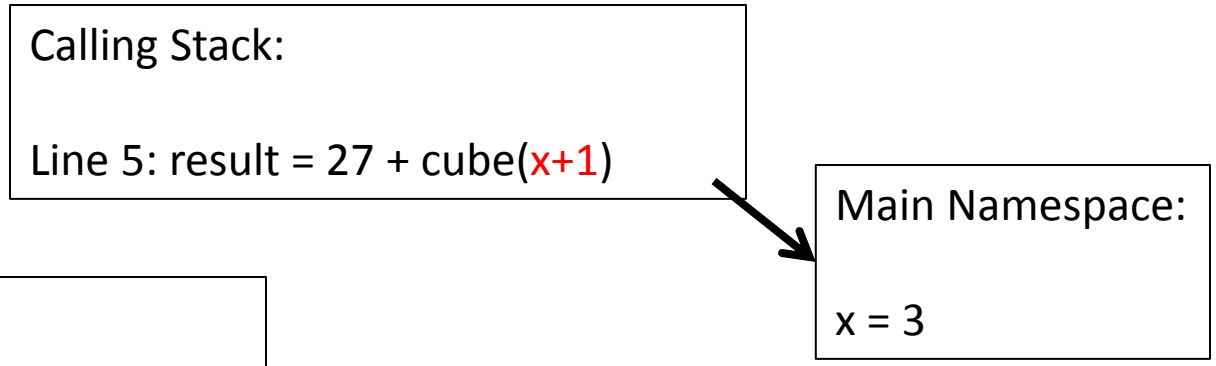
Program Execution




```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```

Note:
- the namespace for **cube** disappears.
- in the calling stack, the returned value replaces the function call in the calling line.

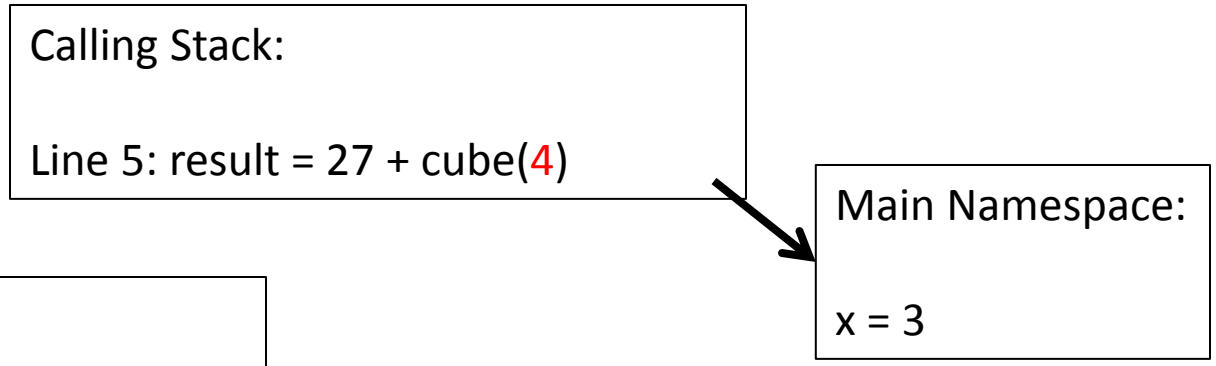
Program Execution




```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```

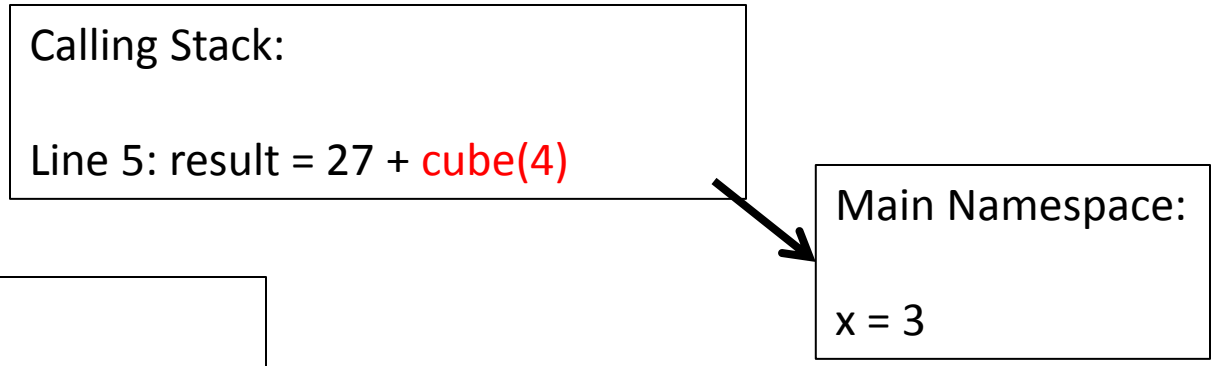
Next subexpression to evaluate in current line: x+1

Program Execution




```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```

Program Execution




```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```

Next subexpression to evaluate in current line: cube(4)

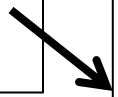
Program Execution

cube Namespace:

n = 4

Calling Stack:

Line 1: def cube(n):
Line 5: result = 27 + cube(4)

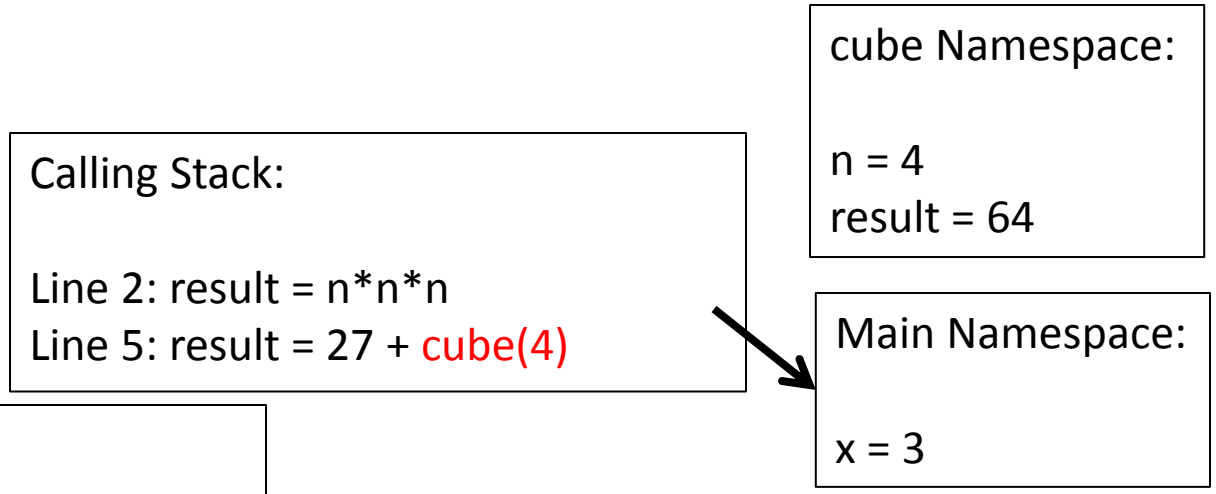


Main Namespace:

x = 3

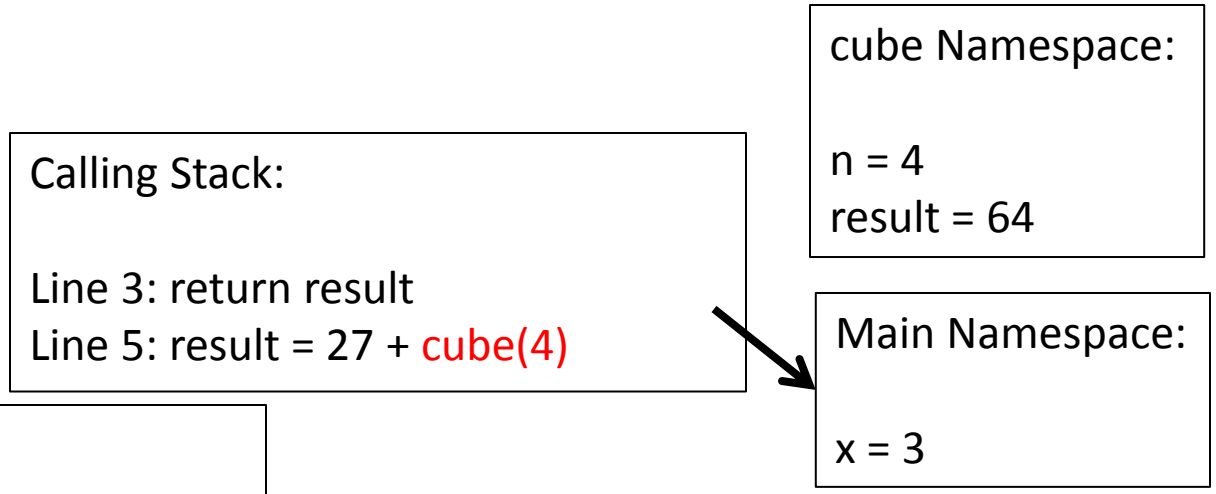
```
Line 1:   def cube(n):
Line 2:       result = n*n*n
Line 3:       return result

# start of main code
Line 4:   x = 3
Line 5:   result = cube(x) + cube(x+1)
Line 6:   print("result =", result)
```

Program Execution



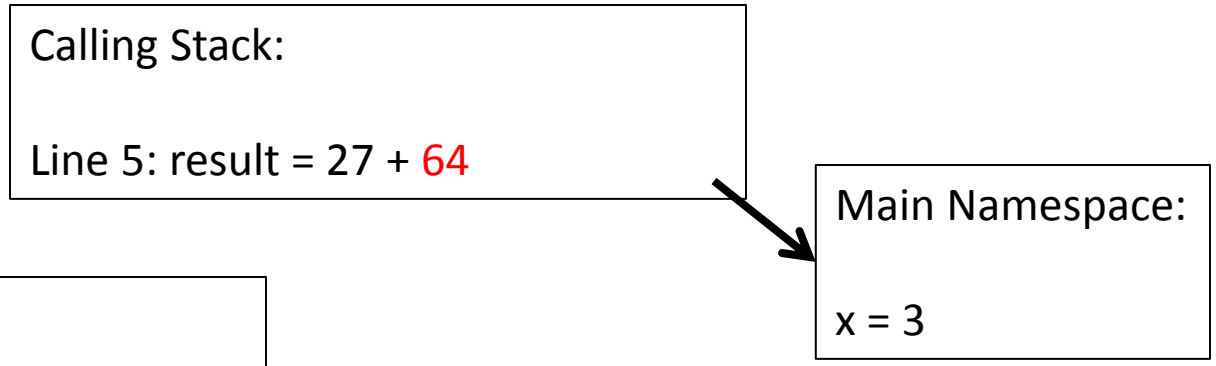

```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```

Program Execution




```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```

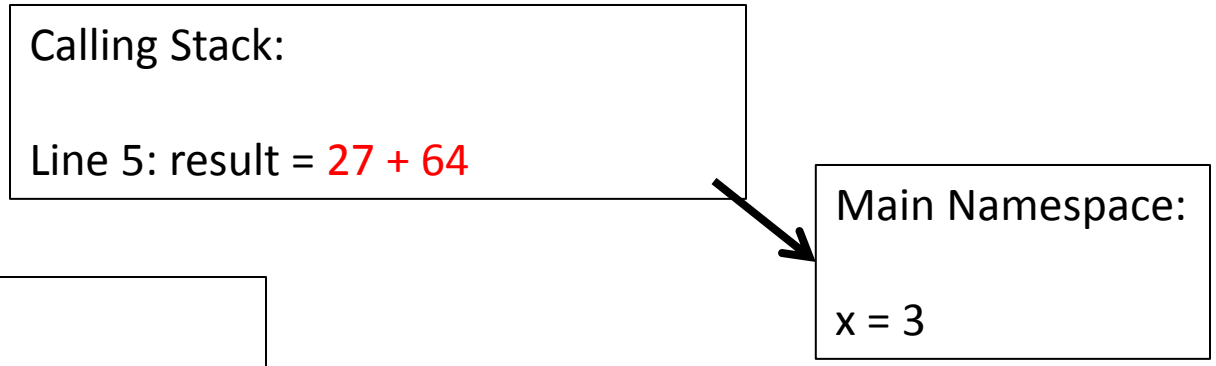
Program Execution




```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```

Note: in the calling stack, the returned value replaces the function call in the calling line.

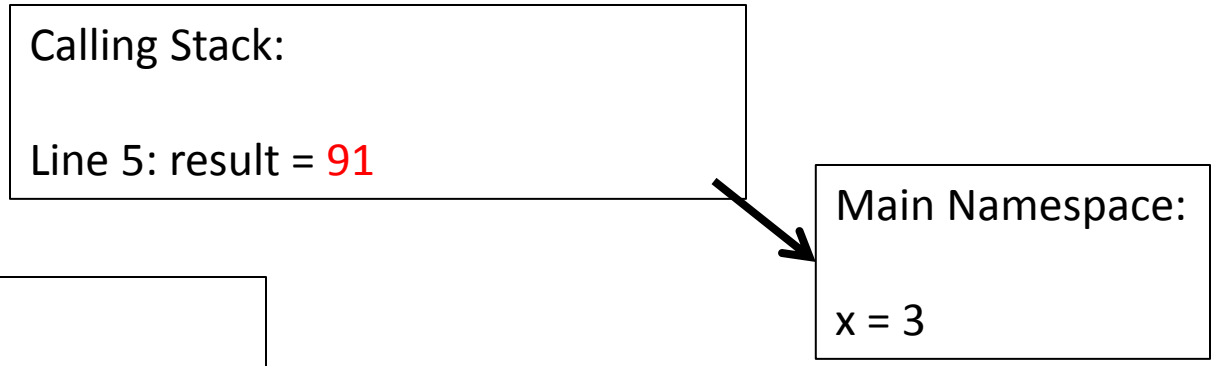
Program Execution




```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```

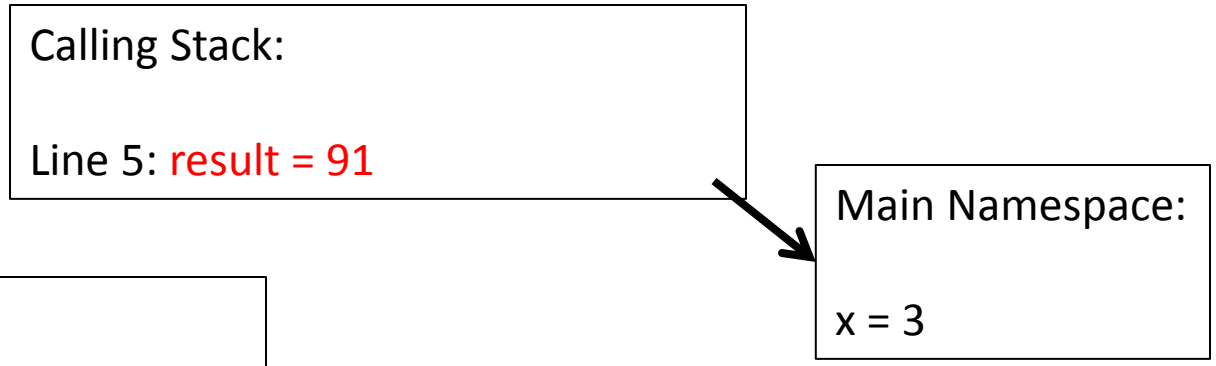
Next subexpression to evaluate in current line: 27 + 64

Program Execution




```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```

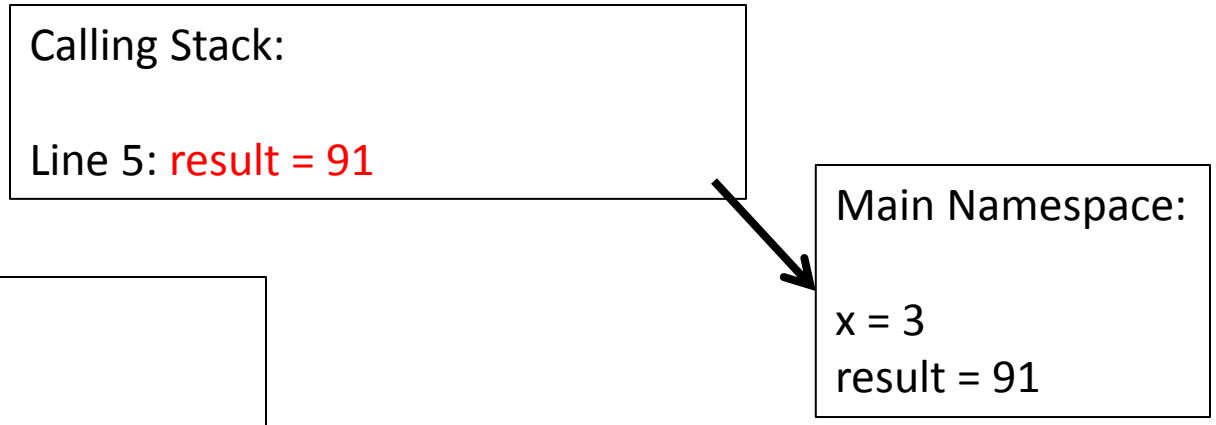
Program Execution




```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```

Next to be done in Line 5: use computed value(s) as prescribed by that line (i.e., do an assignment).

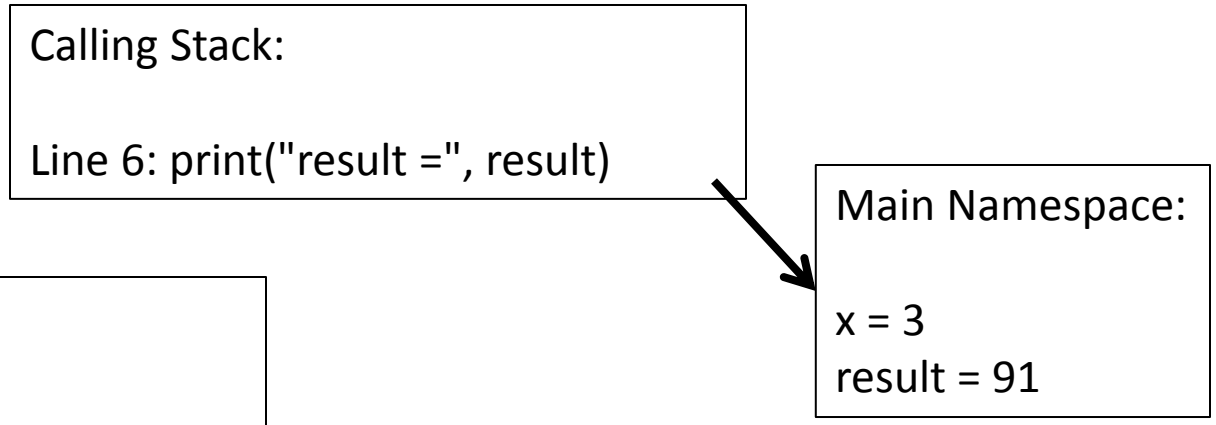
Program Execution




```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```

Next to be done in Line 5: use computed value(s) as prescribed by that line (i.e., do an assignment).

Program Execution




```
Line 1:  def cube(n):
Line 2:      result = n*n*n
Line 3:      return result

# start of main code
Line 4:  x = 3
Line 5:  result = cube(x) + cube(x+1)
Line 6:  print("result =", result)
```