Module 5

User-Defined Functions

Purpose of this Video

- **Series Goal**: Create your own functions
  - Not same as designing (a larger course goal)
  - Focusing on technical details of writing code
- But need to introduce a lot of terminology
  - If you do not know cannot follow lectures
  - Will have a glossary on the course web page
- Will also *standardize* some terminology
  - People use words in slightly different ways

Basic Terminology

- Assume familiarity with a *function call*
  - May not remember the exact term
  - The name for using a function in python
  - **Example**: round(26.54)
- *Arguments* are expressions in parentheses
  - **Example**: round(26.54) has one argument
  - **Example**: round(26.54,1) has two arguments

Procedures vs. Functions

- Most functions are *expressions*
  - The call evaluates to a value
  - Can nest or use in an assignment statement
  - **Example**: `x = round(26.54)` puts 2.7 in x
- But some functions are *statements*
  - **Example**: `print('Hello')` by itself
  - **Example**: `x = print('Hello')` makes x empty
- Latter type of functions are called *procedures*
  - All procedures are function, reverse not true

Fruitful Functions

- What to call functions that are not procedures?
  - Historically they were called functions
  - So functions and procedures **distinct**
  - But the C language called both types functions
  - Python kept this terminology
- We will use the term *fruitful function*
  - Because the function is producing a value
  - Taken from Allen Downey' *Think Python*

Procedure Definitions

- **Goal**: Learn to write a *function definition*
  - You know how to call a function
  - Python does something when you call it
  - How does it know what to do?
- Built-in functions have definitions, but hidden
- In this video, we will focus on procedures
  - Procedures are the easier of the two types
  - But most of what we say applies to all

Anatomy of a Procedure Definition

```
def greet(n):
    """Prints a greeting to the name n

    Precondition: n is a string
    representing a person's name"""
    text = 'Hello '+n+'!'
    print(text)
```

Function Header

Function Body

Anatomy of the Body

```
def greet(n):
    """Prints a greeting to the name n

    Precondition: n is a string
    representing a person's name"""
    text = 'Hello '+n+'!'
    print(text)
```

Docstring **Specification**

Statements to execute

Anatomy of the Header

name parameter(s)

```
def greet(n):
    """Prints a greeting to the name n

    Precondition: n is a string
    representing a person's name"""
    text = 'Hello '+n+'!'
    print(text)
```

keyword

- **Parameter**: variable listed within the parentheses of a header
- Need one parameter per argument you expect

Anatomy of the Header

name

parameter(s)

```
def greet(n):
    """Prints a greeting to the name n
    Precondition: n is a string
    representing a person's name"""
    text = 'Hello '+n+'!'
    print(text)
```

keyword

Function Call:

`greet('Walker')`

One *argument*

- **Parameter**: variable listed within the parentheses of a header
- Need one parameter per argument you expect

When You Call a Procedure

- Calling a procedure does the following
  - It evaluates each argument
  - It plugs each value in the relevant parameter
  - It executes each statement in the body
- **DEMO**: Copy from file into prompt

```
>>> greet('Walker')
'Hello Walker!'
```

When You Call a Procedure

- Calling a procedure does the following
  - It evaluates each argument
  - It p... meter
  - It e...
- **DEM**... into prompt

```
>>> greet('Walker')
'Hello Walker!'
```

Must enter procedure definition *before* you call the procedure

Parameter vs. Local Variables

parameter(s)

Last aside

```
def greet(n):
    """Prints a greeting to the name n

    Precondition: n is a string
    representing a person's name"""
    text = 'Hello '+n+'!'
    print(text)
```

local variable

- **Parameter**: variable listed within the parentheses of a header
- **Local Variable**: variable first assigned in function body

Modules: Python Files

- **Recall**: module is a file with Python code
  - Typically ends in .py
  - Edited with a code editor
  - Will use Atom Editor for my videos
- You use a module by *importing* it
  - Executes the statements in the file
  - You can access any variables in that file
  - **DEMO**: File with a single variable

Modules Contain Function Definitions

- Modules also allow you to access functions
  - Should be familiar with basic Python modules
  - **Example**: `math` and `math.cos`
  - Those modules have function definitions
- Importing causes Python to read definition
  - You can then call the procedure
  - But must follow the standard import rules
- **DEMO:** `procedure.greet('Walker')`

A Good Workflow to Use

1. Write a procedure (function) in a module
2. Open up the Terminal
3. Move to the directory with this file
4. Start Python (type `python`)
5. Import the module
6. Call the procedure (function)

Recall: Fruitful Function vs. Procedure

- **Procedure:** Function call is a statement
  - **Example**: `print('Hello')`
- **Fruitful Function:** Call is expression
  - **Example**: `round(2.64)`
- Definitions are (almost) exactly the same
  - Only difference is a minor change to body
  - Fruitfuls have a new type of statement
  - This is the **return statement**

The **return** Statement

- **Format**: return <*expression*>
  - Used to evaluate ***function call*** (as expression)
  - Also stops executing the function!
  - Any statements after a **return** are ignored
- **Example**: temperature converter function

```
def to_centigrade(x):
    """Returns: x converted to centigrade"""
    return 5*(x-32)/9.0
```

Combining Return with Other Statements

```
def plus(n):
    """Returns the number n+1

       Parameter n: number to add to
       Precondition: n is a number"""
    x = n+1
    return x
```

Creates variable x w/ answer

Makes value of x the result

Math Analogy:

- On a math exam, do your work and circle final answer.
- Return is same idea as indicating your final answer

Combining Return with Other Statements

```
def plus(n):
    """Returns the number n+1

    Parameter n: number to add to
    Precondition: n is a number"""
    x = n+1
    return x
```

Return should be placed last!

- `x = n+1` — Creates variable x w/ answer
- `return x` — Makes value of x the result

Math Analogy:

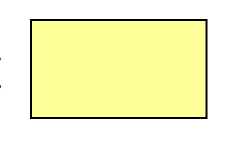
- On a math exam, do your work and circle final answer.
- Return is same idea as indicating your final answer

Print vs. Return

Print

- Displays value on screen
  - Useful for **testing**
  - Not for calculations

```
def print_plus(n):
    print(n+1)
>>> x = print_plus(2)
3
>>>
```



x

Return

- Defines function's value
  - Needed for **calculations**
  - But does not display

```
def return_plus(n):
    return (n+1)
>>> x = return_plus(2)
>>>
```

x 3

Visualization

- You must to learn to think like Python does
  - Else you and Python will miscommunicate
  - Like a coworker with language/cultural issues
  - Good programmers see from Python's persp.
- Need to build **visual models** of Python
  - You imagine what Python is doing invisibly
  - Not exactly accurate; more like **metaphores**
  - We call this skill **visualization**

A Motivating Example

Function Definition	Function Call

```
8.  def plus(n):
9.      """Returns n+1"""
10.     x = n+1        # local var
11.     return x
```

```
>>> x = 2              # global var
>>> y = plus(4)
```

A Motivating Example

Function Definition

```
8.  def plus(n):
9.      """Returns n+1"""
10.     x = n+1
11.     return x
```

local var

Function Call

```
>>> x = 2
>>> y = plus(4)
```

global var

Visualization

```
>>> x = 2
```

Global Space

x | 2

A Motivating Example

Function Definition

```
8.  def plus(n):
9.      """Returns n+1"""
10.     x = n+1
11.     return x
```

Function Call

```
>>> x = 2
>>> y = plus(4)
```

x: ?

What is in the box?

A: 2
B: 4
C: 5

A Motivating Example

Function Definition	Function Call

```
8.  def plus(n):
9.      """Returns n+1"""
10.     x = n+1
11.     return x
```

```
>>> x = 2
>>> y = plus(4)
```

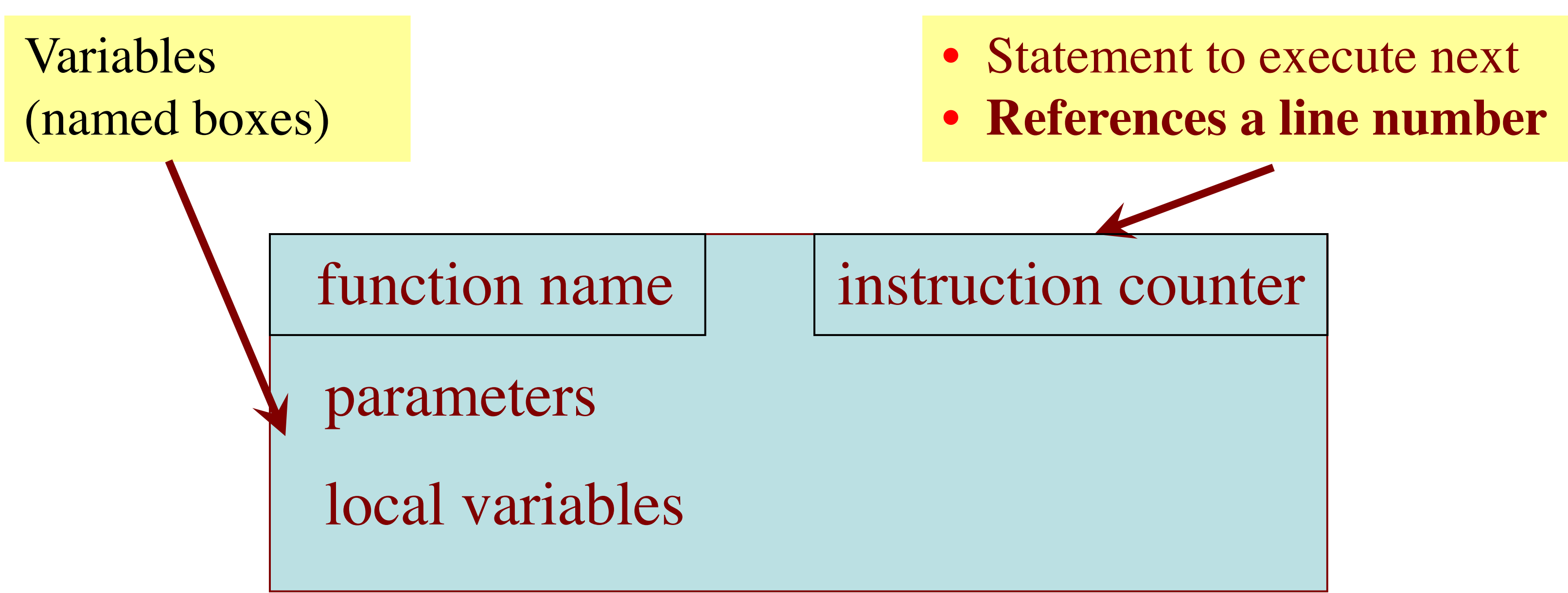
x: ?

What is in the box?

A: 2 **Correct**
B: 4
C: 5

Understanding How Functions Work

- **Call Frame**: Representation of function call
- A **conceptual model** of Python



When You Call a Function It…

- Creates a new call frame
- Evaluates the arguments
- Creates a variable for each parameter
- Stores the argument in each parameter
- Puts counter at first line after specification (or first of body if no specification)

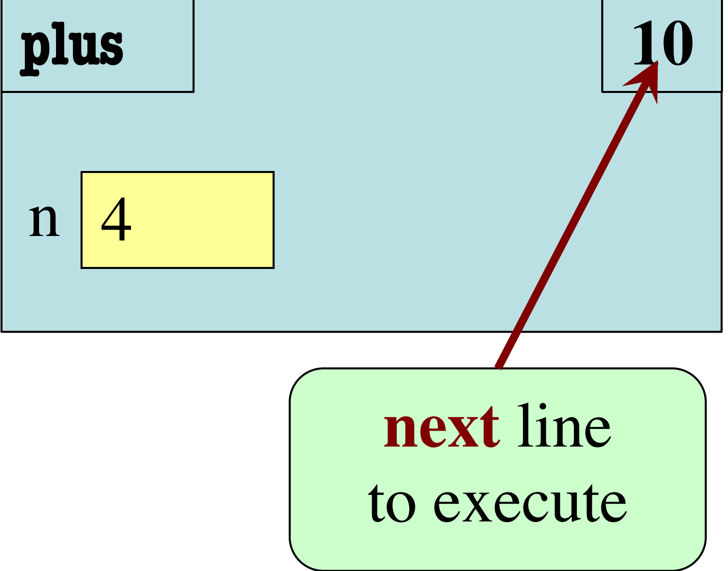
An Example

Function Definition

```
8.  def plus(n):
9.      """Returns n+1"""
10.     x = n+1
11.     return x
```

Function Call

-



Next: Execute the Body Until the End

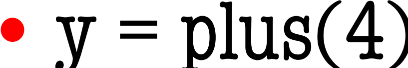
- Process one line of code at a time
  - Each time you read a line redraw the frame
  - Not a *new* frame; the frame is changing
  - Think of it as "animating" the frame
- How to process each type of statement:
  - **Print**: Nothing (on screen, not frame)
  - **Assignment**: Put variable in frame
  - **Return**: create a special "RETURN" variable
- Move the instruction counter forward

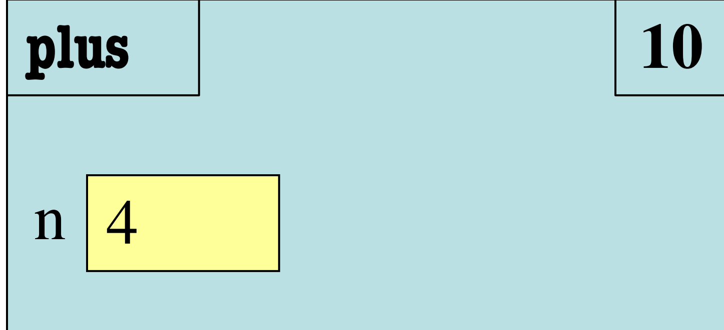
An Example

Function Definition

```
8.  def plus(n):
9.      """Returns n+1"""
10.     x = n+1
11.     return x
```

Function Call

- 



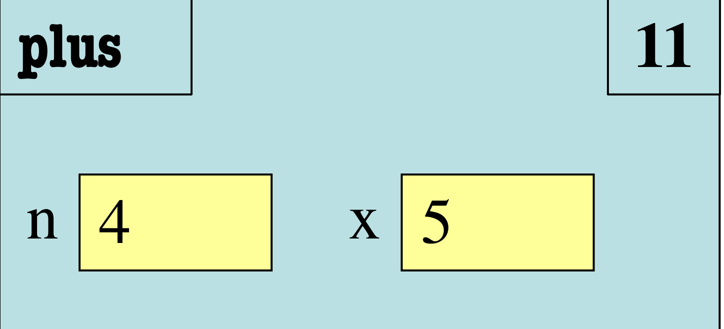
An Example

Function Definition

```
8.  def plus(n):
9.      """Returns n+1"""
10.     x = n+1
11.     return x
```

Function Call

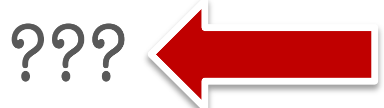
- y = plus(4)



An Example

Function Definition

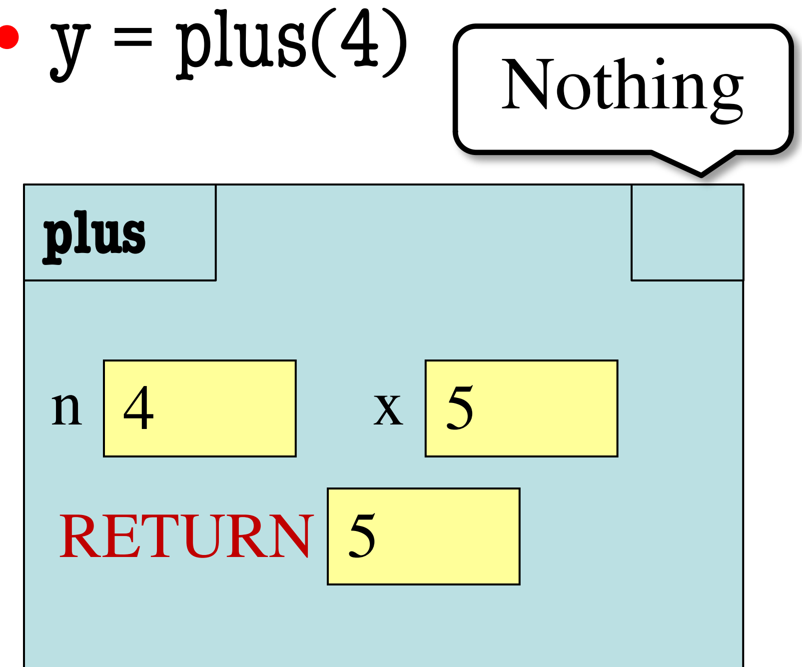
```
8.  def plus(n):
9.      """Returns n+1"""
10.     x = n+1
11.     return x
```



Function Call

- y = plus(4)

Nothing



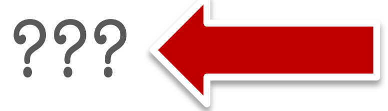
When You are Done

- Look if there is a RETURN variable
  - Might not be if a procedure
  - If so, remember that
- Erase the frame entirely
  - All variables inside of frame are deleted
  - Including the RETURN
- Function call turns into a value (RETURN)
  - Use that in the calling statement

An Example

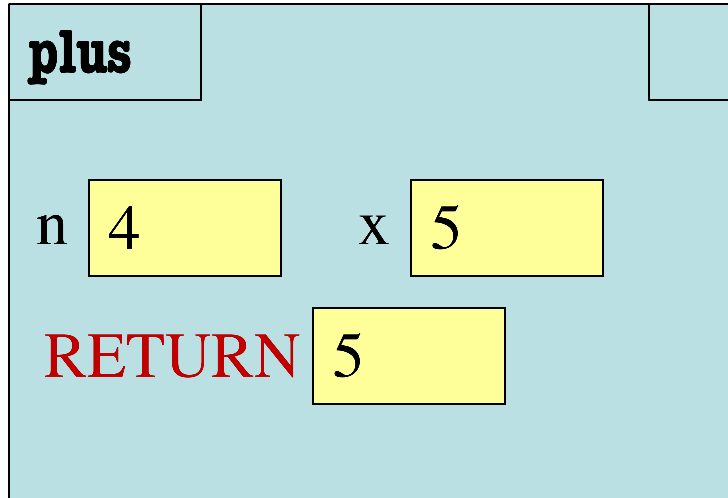
Function Definition

```
8.  def plus(n):
9.      """Returns n+1"""
10.     x = n+1
11.     return x
```



Function Call

- y = plus(4)



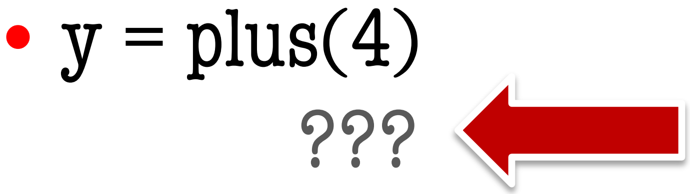
An Example

Function Definition

```
8.  def plus(n):
9.      """Returns n+1"""
10.     x = n+1
11.     return x
```

Function Call

- y = plus(4)



Global Space

Variables here are not erased

x | 2

y | 5

The Python Tutor

tab1 x +

```
1  def plus(n):
2      """Returns n+1"""
3      x = n+1
4      return x
5
6  x = 2
7  y = plus(4)
8
```

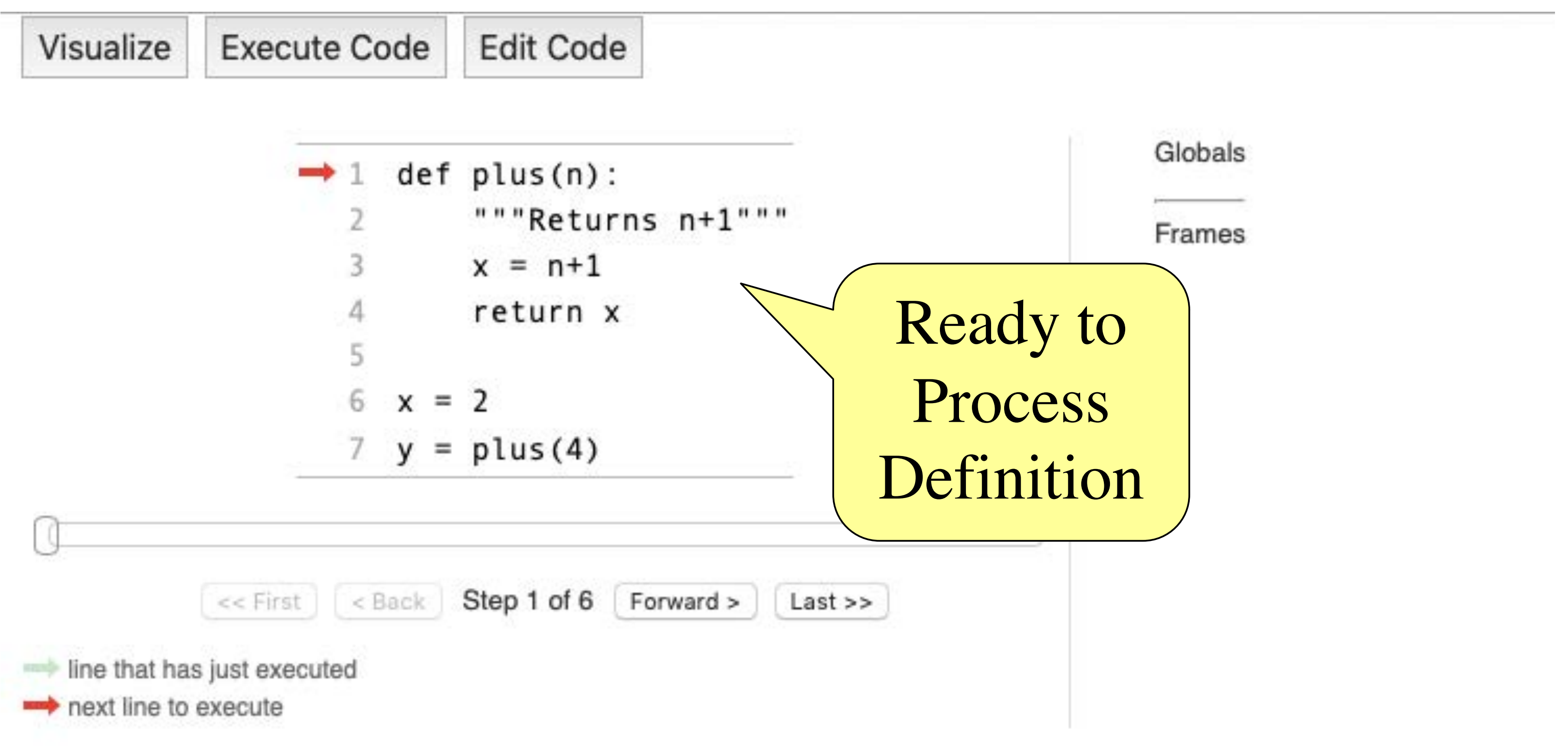
Definition

Global Assignment

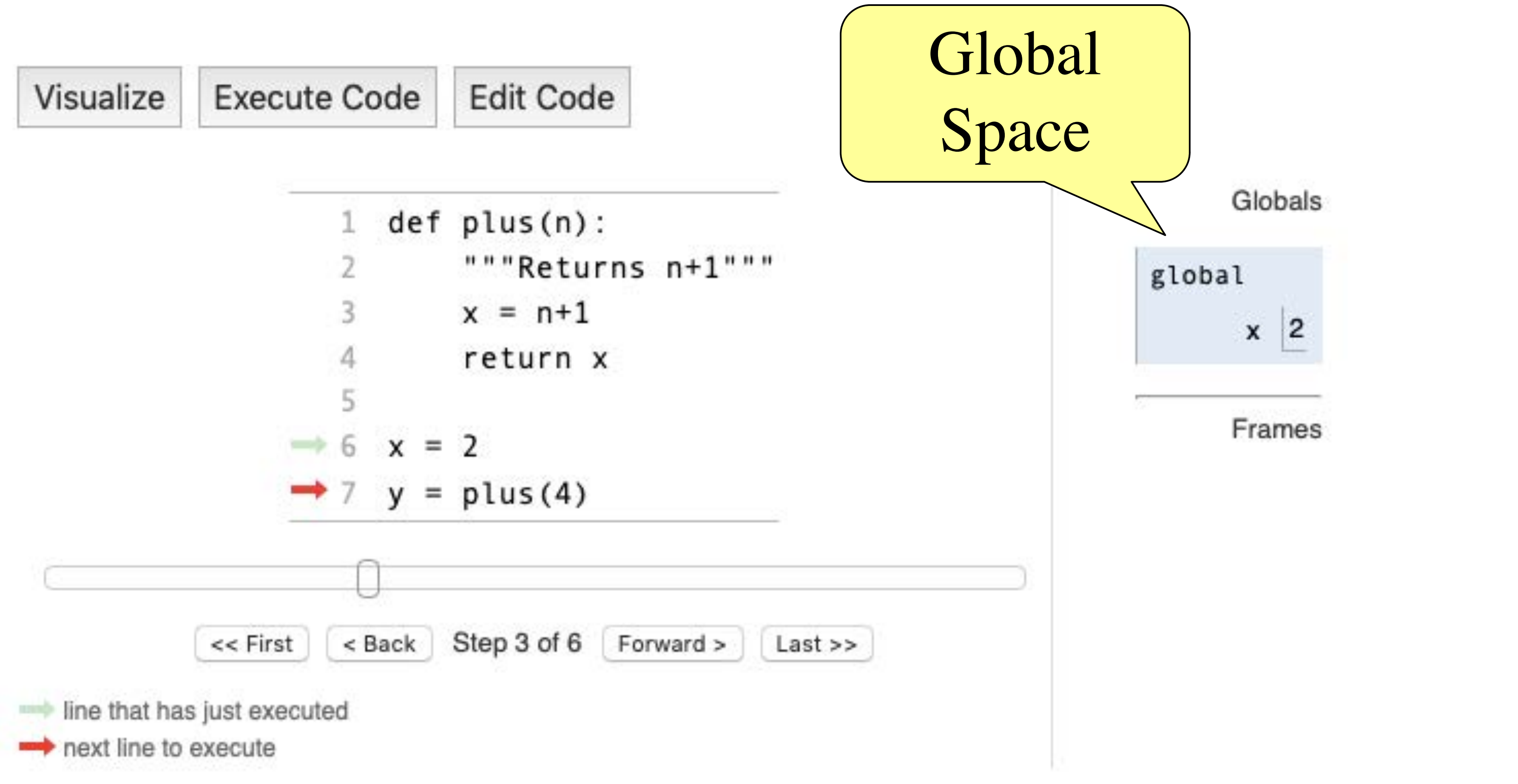
Function Call

Double click the tab to change name, press enter when done.

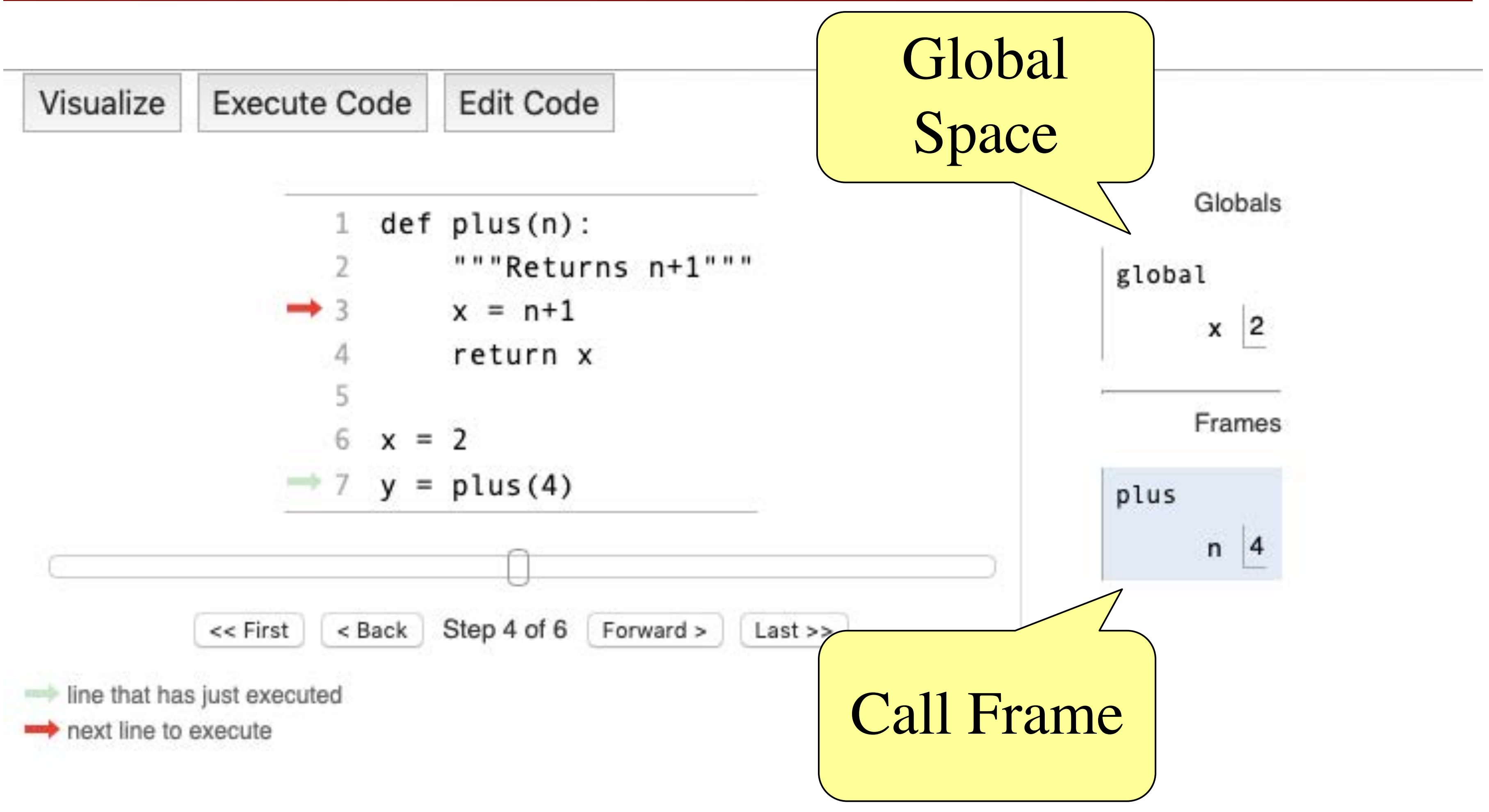
First Step of Visualization



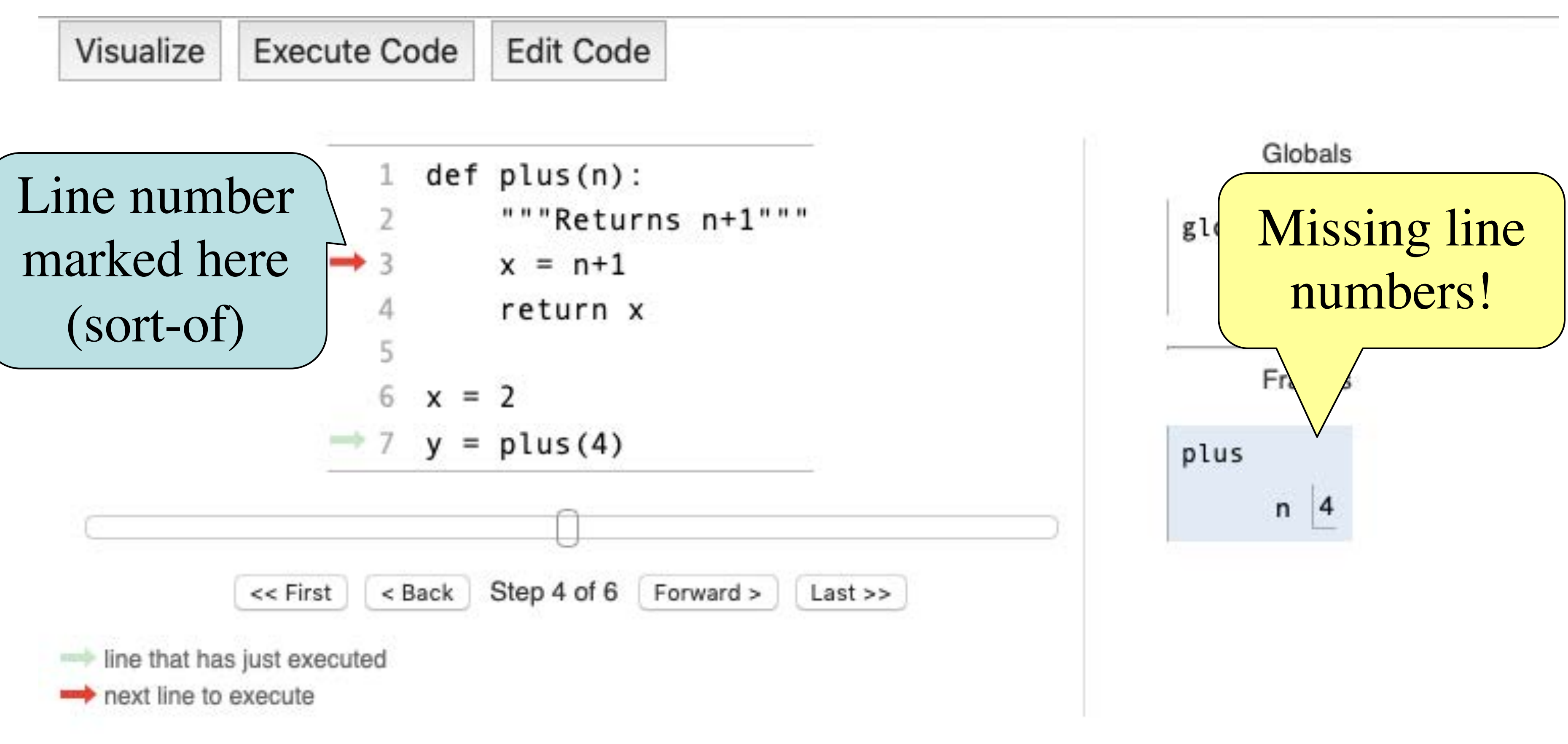
Processing the Global Assignment



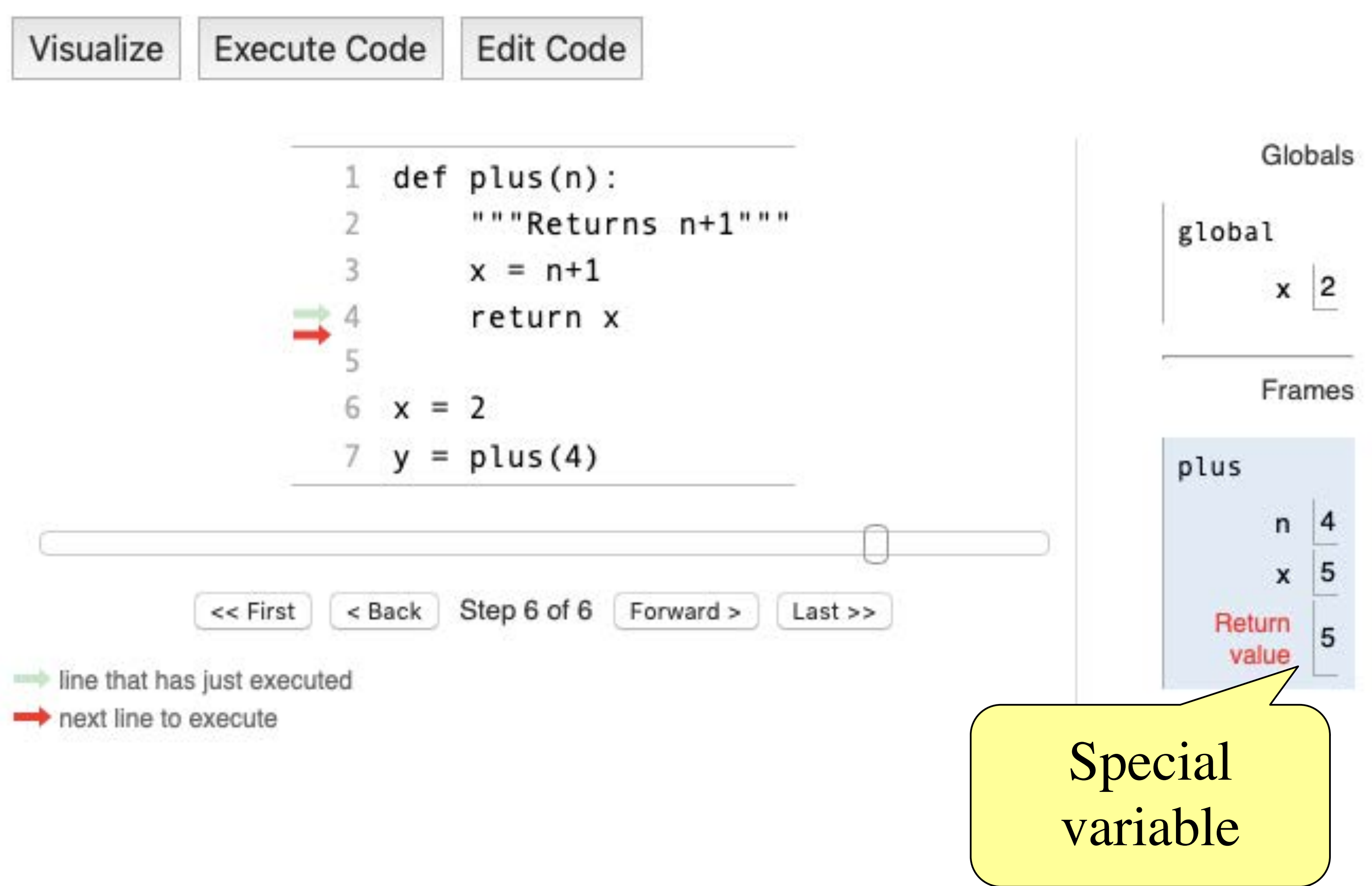
Starting The Function Call



Starting The Function Call

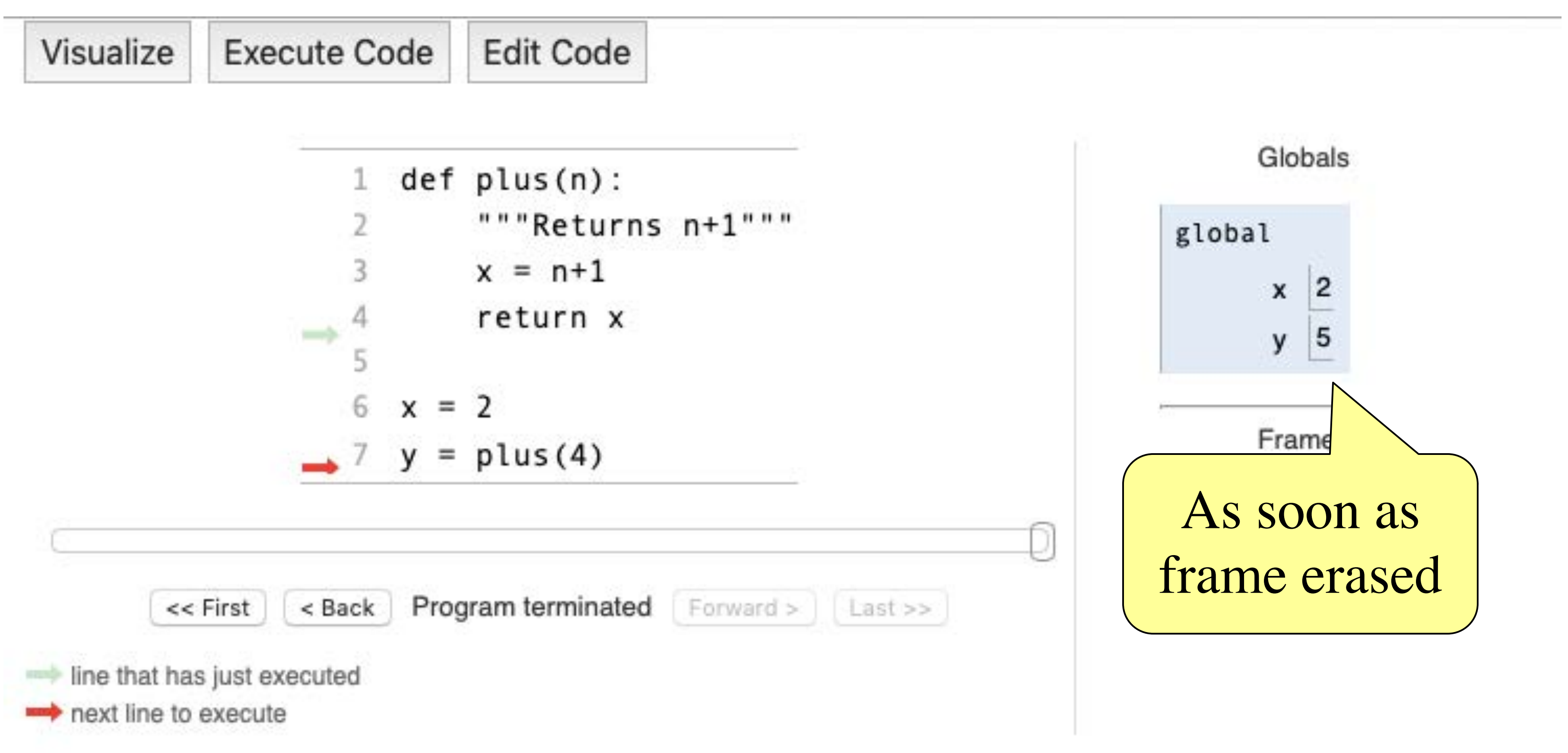


Executing the Function Call



Special variable

Erasing the Frame



Working With Tabs

- You can use tabs to simulate modules
  - Put function definition in one tab
  - Import and call in another
- But visualizer will not show frame
  - Can only show a call frame if in same tab
  - This is a limitation of visualizer
  - Under hood, call frame still made
- **DEMO**: Split up code from last example