Functions that return values

Most functions require arguments, values that control how the function does its job. For example, if you want to find the absolute value of a number, you have to indicate what the number is. Python has a built-in function for computing the absolute value:

```
print(abs(5))

print(abs(-5))

```

ActiveCode: 1 (ch04_4)

Run

```
5
5
```

In this example, the arguments to the `abs` function are 5 and -5.

Some functions take more than one argument. For example the math module contains a function called `pow` which takes two arguments, the base and the exponent.

```
import math
print(math.pow(2, 3))

print(math.pow(7, 4))

```

ActiveCode: 2 (ch04_5)

Run

```
8.0
2401.0
```

Note

Of course, we have already seen that raising a base to an exponent can be done with the ** operator.

Another built-in function that takes more than one argument is `max`.

```
print(max(7, 11))
print(max(4, 1, 17, 2, 12))
print(max(3 * 11, 5**3, 512 - 9, 1024**0))

```

ActiveCode: 3 (ch04_6)

Run

```
11
17
503
```

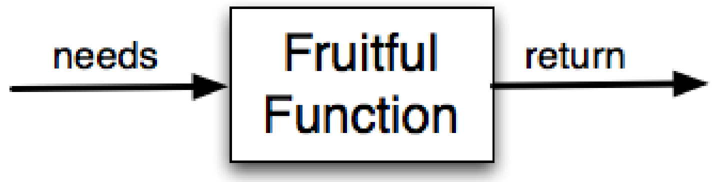
`max` can be sent any number of arguments, separated by commas, and will return the maximum value sent. The arguments can be either simple values or expressions. In the last example, 503 is returned, since it is larger than 33, 125, and 1. Note that `max` also works on lists of values.

Furthermore, functions like `range`, `int`, `abs` all return values that can be used to build more complex expressions.

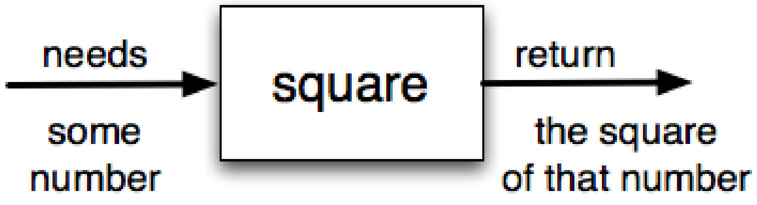
So an important difference between these functions and one like `drawSquare` is that `drawSquare` was not executed because we wanted it to compute a value — on the contrary, we wrote `drawSquare` because we wanted it to execute a sequence of steps that caused the turtle to draw a specific shape.

Functions that return values are sometimes called **fruitful functions**. In many other languages, a chunk that doesn't return a value is called a **procedure**, but we will stick here with the Python way of also calling it a function, or if we want to stress it, a *non-fruitful* function.

Fruitful functions still allow the user to provide information (arguments). However there is now an additional piece of data that is returned from the function.



How do we write our own fruitful function? Lets start by creating a very simple mathematical function that we will call `square`. The square function will take one number as a parameter and return the result of squaring that number. Here is the black-box diagram with the Python code following.



```python
def square(x):
    return x* x

toSquare = 10
result = square(toSquare)
print("The result of ", toSquare, " squared is ", result)

```

ActiveCode: 4 (ch04_square)

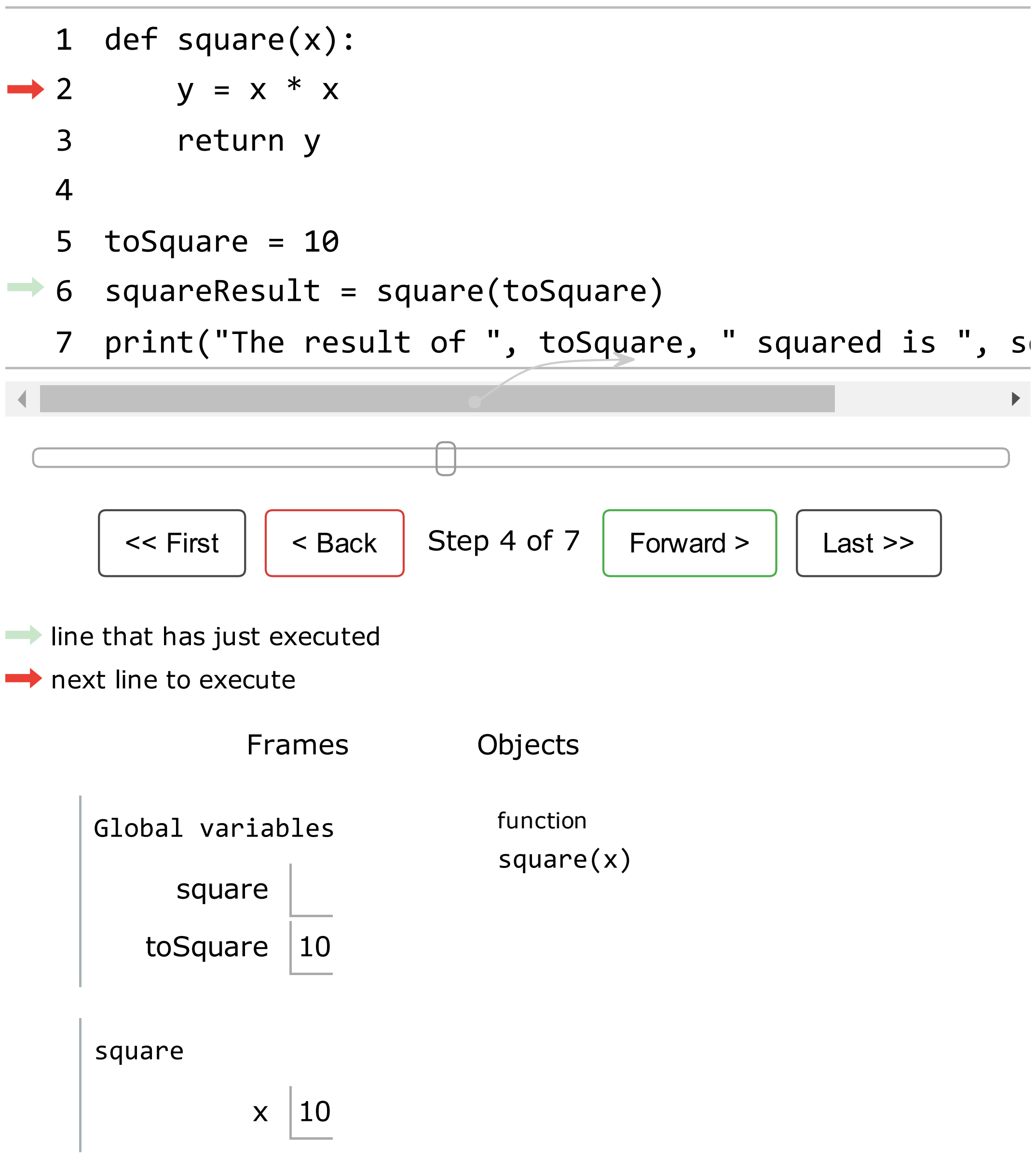
Run

```
The result of  10  squared is  100
```

The **return** statement is followed by an expression which is evaluated. Its result is returned to the caller as the “fruit” of calling this function. Because the return statement can contain any Python expression we could have avoided creating the **temporary variable** `y` and simply used `return x*x`. Try modifying the square function above to see that this works just the same. On the other hand, using **temporary variables** like `y` in the program above makes debugging easier. These temporary variables are referred to as **local variables**.

Notice something important here. The name of the variable we pass as an argument — `toSquare` — has nothing to do with the name of the formal parameter — `x`. It is as if `x = toSquare` is executed when `square` is called. It doesn’t matter what the value was named in the caller. In `square`, it’s name is `x`. You can see this very clearly in codelens, where the global variables and the local variables for the square function are in separate boxes.

As you step through the example in codelens notice that the **return** statement not only causes the function to return a value, but it also returns the flow of control back to the place in the program where the function call was made.



```
1  def square(x):
2      y = x * x
3      return y
4
5  toSquare = 10
6  squareResult = square(toSquare)
7  print("The result of ", toSquare, " squared is ", s
```

➡ line that has just executed
➡ next line to execute

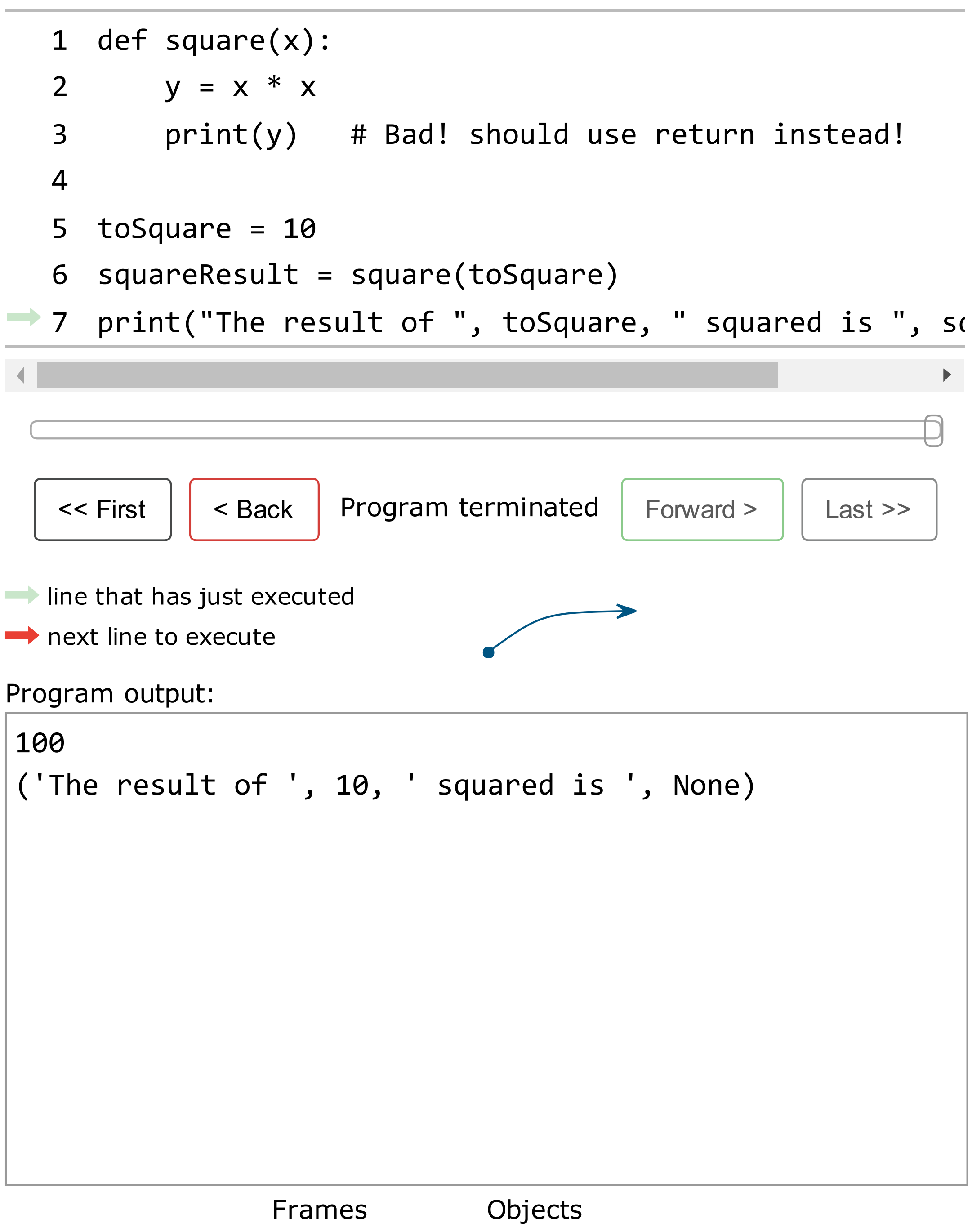
CodeLens: 1 (ch04_clsquare)

Another important thing to notice as you step through this codelens demonstration is the movement of the red and green arrows. Codelens uses these arrows to show you where it is currently executing. Recall that the red arrow always points to the next line of code that will be executed. The light green arrow points to the line that was just executed in the last step.

When you first start running this codelens demonstration you will notice that there is only a red arrow and it points to line 1. This is because line 1 is the next line to be executed and since it is the first line, there is no previously executed line of code.

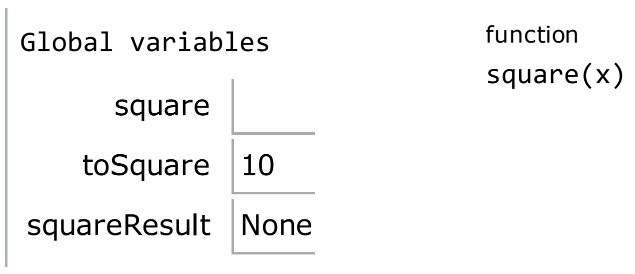
When you click on the forward button, notice that the red arrow moves to line 5, skipping lines 2 and 3 of the function (and the light green arrow has now appeared on line 1). Why is this? The answer is that function definition is not the same as function execution. Lines 2 and 3 will not be executed until the function is called on line 6. Line 1 defines the function and the name `square` is added to the global variables, but that is all the `def` does at that point. The body of the function will be executed later. Continue to click the forward button to see how the flow of control moves from the call, back up to the body of the function, and then finally back to line 7, after the function has returned its value and the value has been assigned to `squareResult`.

Finally, there is one more aspect of function return values that should be noted. All Python functions return the value `None` unless there is an explicit return statement with a value other than `None`. Consider the following common mistake made by beginning Python programmers. As you step through this example, pay very close attention to the return value in the local variables listing. Then look at what is printed when the function returns.



```
1  def square(x):
2      y = x * x
3      print(y)   # Bad! should use return instead!
4
5  toSquare = 10
6  squareResult = square(toSquare)
7  print("The result of ", toSquare, " squared is ", s
```

```
100
('The result of ', 10, ' squared is ', None)
```



Global variables

square	
toSquare	10
squareResult	None

CodeLens: 2 (ch04_clsquare_bad)

The problem with this function is that even though it prints the value of the square, that value will not be returned to the place where the call was done. Since line 6 uses the return value as the right hand side of an assignment statement, the evaluation of the function will be `None`. In this case, `squareResult` will refer to that value after the assignment statement and therefore the result printed in line 7 is incorrect. Typically, functions will return values that can be printed or processed in some other way by the caller.

Check your understanding

func-8: What is wrong with the following function definition:

```
def addEm(x, y, z):
    return x+y+z
    print('the answer is', x+y+z)
```

a) You should never use a print statement in a function definition.

[✓ b) You should not have any statements in a function after the return statement. Once the function gets to the return statement it will immediately stop executing the function.]

c) You must calculate the value of x+y+z before you return it.

d) A function cannot return a number.

Check Me Compare Me

Correct!! This is a very common mistake so be sure to watch out for it when you write your code!

func-9: What will the following function return?

```python
def addEm(x, y, z):
    print(x+y+z)
```

[✓ a) Nothing (no value)]

b) The value of x+y+z

c) The string 'x+y+z'

Check Me Compare Me

Correct!! We have accidentally used print where we mean return. Therefore, the function will return the value None by default. This is a VERY COMMON mistake so watch out! This mistake is also particularly difficult to find because when you run the function the output looks the same. It is not until you try to assign its value to a variable that you can notice a difference.

Variables and parameters are local

An assignment statement in a function creates a **local variable** for the variable on the left hand side of the assignment operator. It is called local because this variable only exists inside the function and you cannot use it outside. For example, consider again the `square` function:

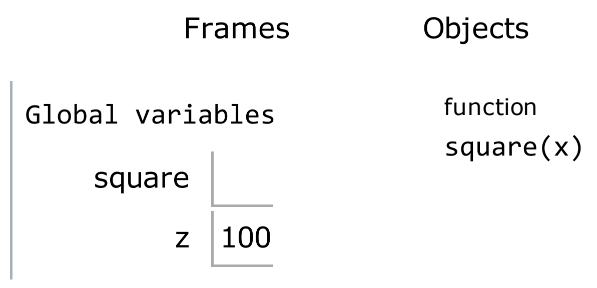
```python
def square(x):
    y = x * x
    return y

z = square(10)
print(y)
```

<< First < Back Program terminated Forward > Last >>

```
NameError: name 'y' is not defined
```

➡ line that has just executed

➡ next line to execute



CodeLens: 3 (bad_local)

If you press the ‘last >>’ button you will see an error message. When we try to use `y` on line 6 (outside the function) Python looks for a global variable named `y` but does not find one. This results in the error: `Name Error: 'y' is not defined.`

The variable `y` only exists while the function is being executed — we call this its **lifetime**. When the execution of the function terminates (returns), the local variables are destroyed. Codelens helps you visualize this because the local variables disappear after the function returns. Go back and step thru the statements paying particular attention to the variables that are created when the function is called. Note when they are subsequently destroyed as the function returns.

Formal parameters are also local and act like local variables. For example, the lifetime of `x` begins when `square` is called, and its lifetime ends when the function completes its execution.

So it is not possible for a function to set some local variable to a value, complete its execution, and then when it is called again next time, recover the local variable. Each call of the function creates new local variables, and their lifetimes expire when the function returns to the caller.

On the other hand, it is legal for a function to access a global variable. However, this is considered **bad form** by nearly all programmers and should be avoided. Look at the following, nonsensical variation of the square function.

```
def badsquare(x):
    y = x ** power
    return y

power = 2
result = badsquare(10)
print(result)

```

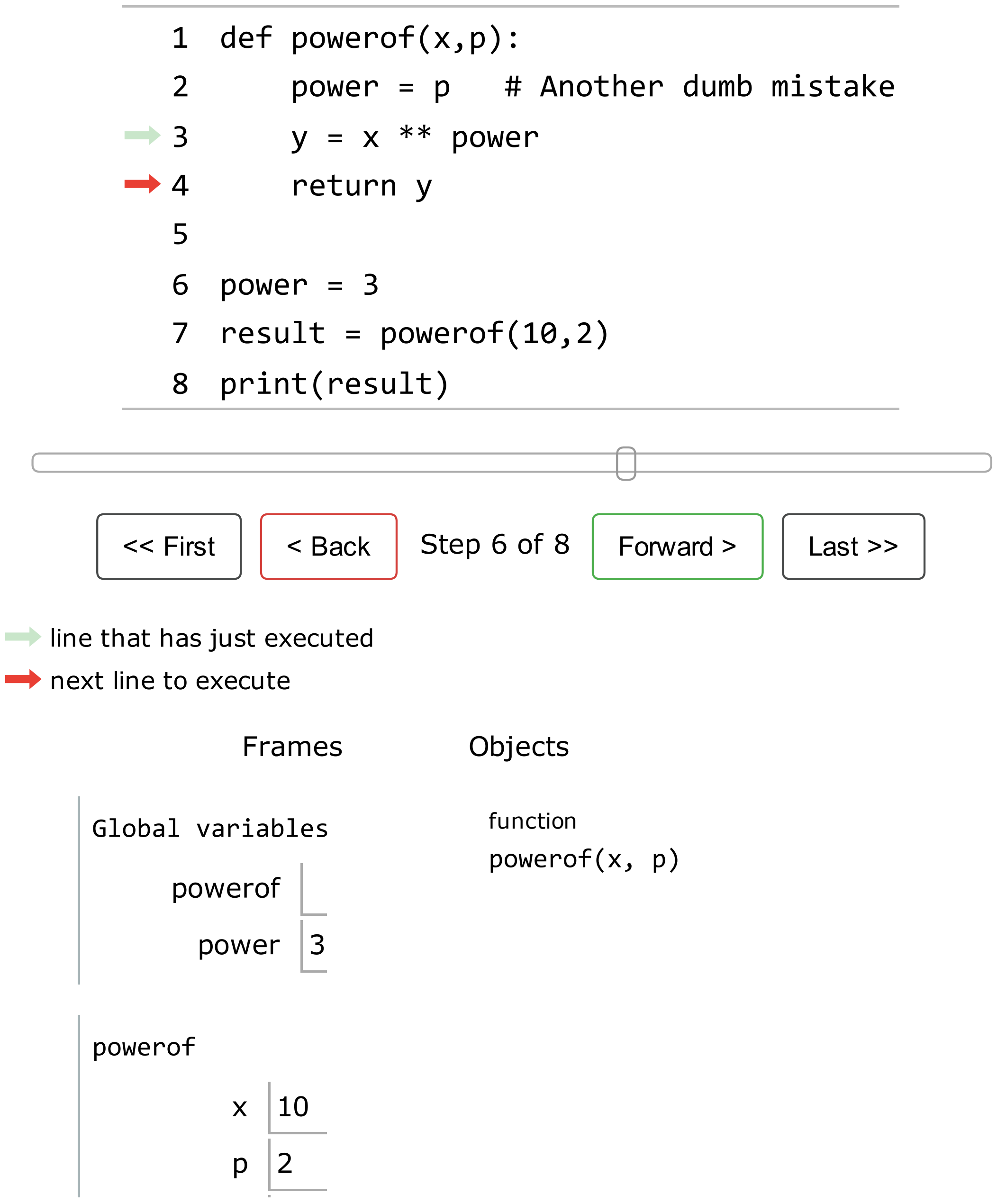
ActiveCode: 5 (badsquare_1)

Run

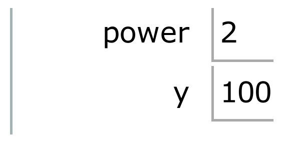
```
100
```

Although the `badsquare` function works, it is silly and poorly written. We have done it here to illustrate an important rule about how variables are looked up in Python. First, Python looks at the variables that are defined as local variables in the function. We call this the **local scope**. If the variable name is not found in the local scope, then Python looks at the global variables, or **global scope**. This is exactly the case illustrated in the code above. `power` is not found locally in `badsquare` but it does exist globally. The appropriate way to write this function would be to pass power as a parameter. For practice, you should rewrite the badsquare example to have a second parameter called power.

There is another variation on this theme of local versus global variables. Assignment statements in the local function cannot change variables defined outside the function. Consider the following codelens example:



```
def powerof(x,p):
    power = p    # Another dumb mistake
    y = x ** power
    return y

power = 3
result = powerof(10,2)
print(result)
```



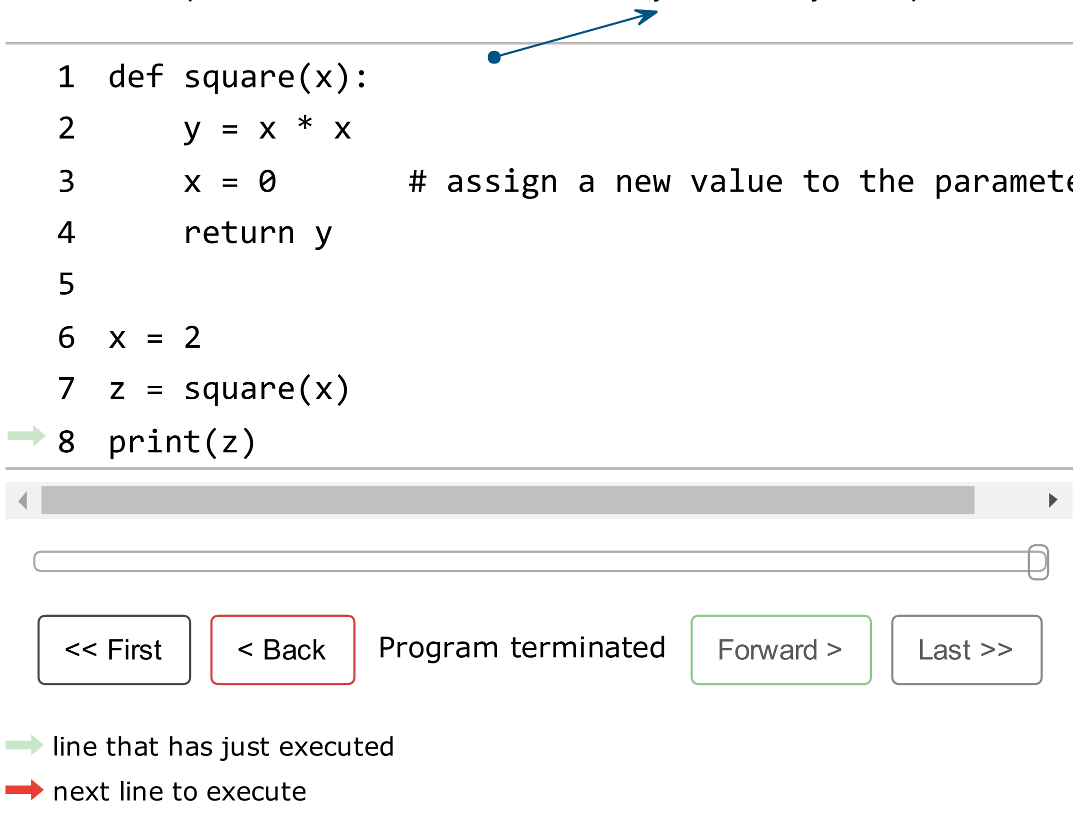
power	2
y	100

CodeLens: 4 (cl_powerof_bad)

Now step through the code. What do you notice about the values of variable `power` in the local scope compared to the variable `power` in the global scope?

The value of `power` in the local scope was different than the global scope. That is because in this example `power` was used on the left hand side of the assignment statement `power = p`. When a variable name is used on the left hand side of an assignment statement Python creates a local variable. When a local variable has the same name as a global variable we say that the local shadows the global. A **shadow** means that the global variable cannot be accessed by Python because the local variable will be found first. This is another good reason not to use global variables. As you can see, it makes your code confusing and difficult to understand.

To cement all of these ideas even further lets look at one final example. Inside the `square` function we are going to make an assignment to the parameter `x` There's no good reason to do this other than to emphasize the fact that the parameter `x` is a local variable. If you step through the example in codelens you will see that although `x` is 0 in the local variables for `square`, the `x` in the global scope remains 2. This is confusing to many beginning programmers who think that an assignment to a formal parameter will cause a change to the value of the variable that was used as the actual parameter, especially when the two share the same name. But this example demonstrates that that is clearly not how Python operates.

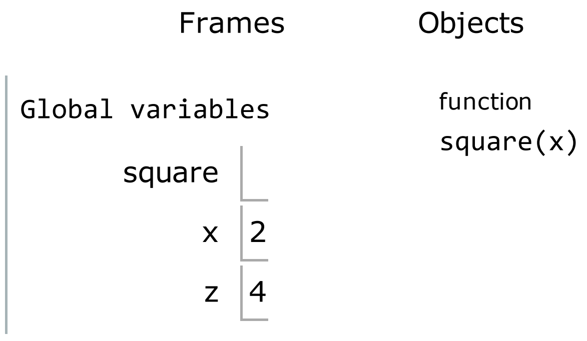


```
def square(x):
    y = x * x
    x = 0        # assign a new value to the paramet
    return y

x = 2
z = square(x)
print(z)
```

Program terminated

Program output:

```
4
```



CodeLens: 5 (cl_change_parm)

Check your understanding

func-10: What is a variable’s scope?

- a) Its value
- [✓ b) The range of statements in the code where a variable can be accessed.]
- c) Its name

Check Me Compare Me

Correct!!

func-11: What is a local variable?

- [✓ a) A temporary variable that is only used inside a function]

b) The same as a parameter

c) Another name for any variable

Check Me Compare Me

Correct!! Yes, a local variable is a temporary variable that is only known (only exists) in the function it is defined in.

func-12: Can you use the same name for a local variable as a global variable?

a) Yes, and there is no reason not to.

[✓ b) Yes, but it is considered bad form.]

c) No, it will cause an error.

Check Me Compare Me

Correct!! it is generally considered bad style because of the potential for the programmer to get confused. If you must use global variables (also generally bad form) make sure they have unique names.