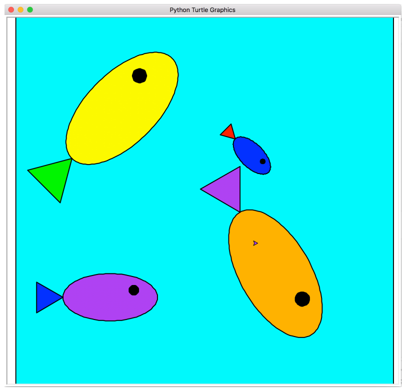
Abstracting with Functions



CS111 Computer Programming

Department of Computer Science
Wellesley College

FUNCTION BASICS

Concepts in this slide:
functions,
input & output

A function is a block of code that performs a sequence of instructions.

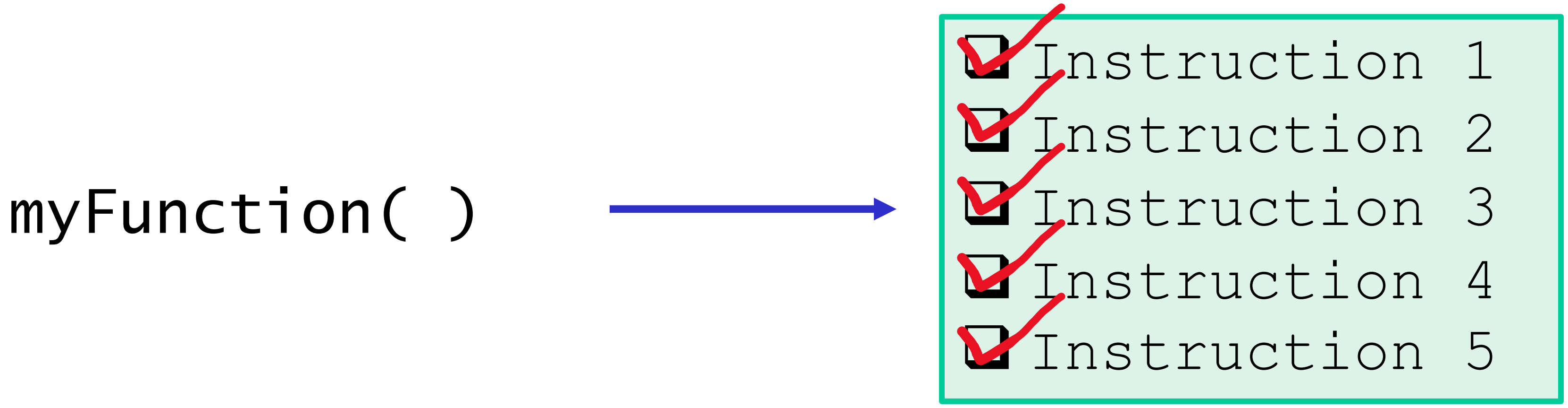
```
myFunction( ):
```

- ❑ Instruction 1
- ❑ Instruction 2
- ❑ Instruction 3
- ❑ Instruction 4
- ❑ Instruction 5

Concepts in this slide:
functions,
input & output

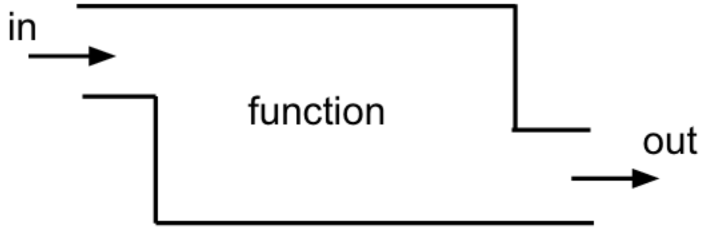
A function is a block of code that performs a sequence of instructions.

Whenever the function is "called", the sequence of instructions is executed.



Concepts in this slide: input & output

Functions can take inputs and return outputs based on those inputs



Here are examples of **built-in** functions you have seen:

In [...]

```
max(7,3)
min(7,3,2,9)
type(123)
len('CS111')
str(4.0)
int(-2.978)
float(42)
round(2.718, 1)
```

Out […]

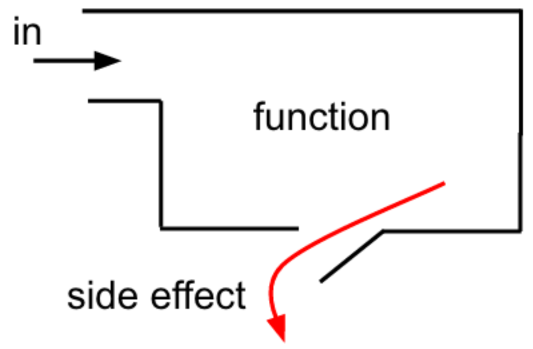
```
7
2
int
5
'4.0'
-2
42.0
2.7
```

Concepts in this slide:
side effects

Some functions perform actions instead of returning outputs

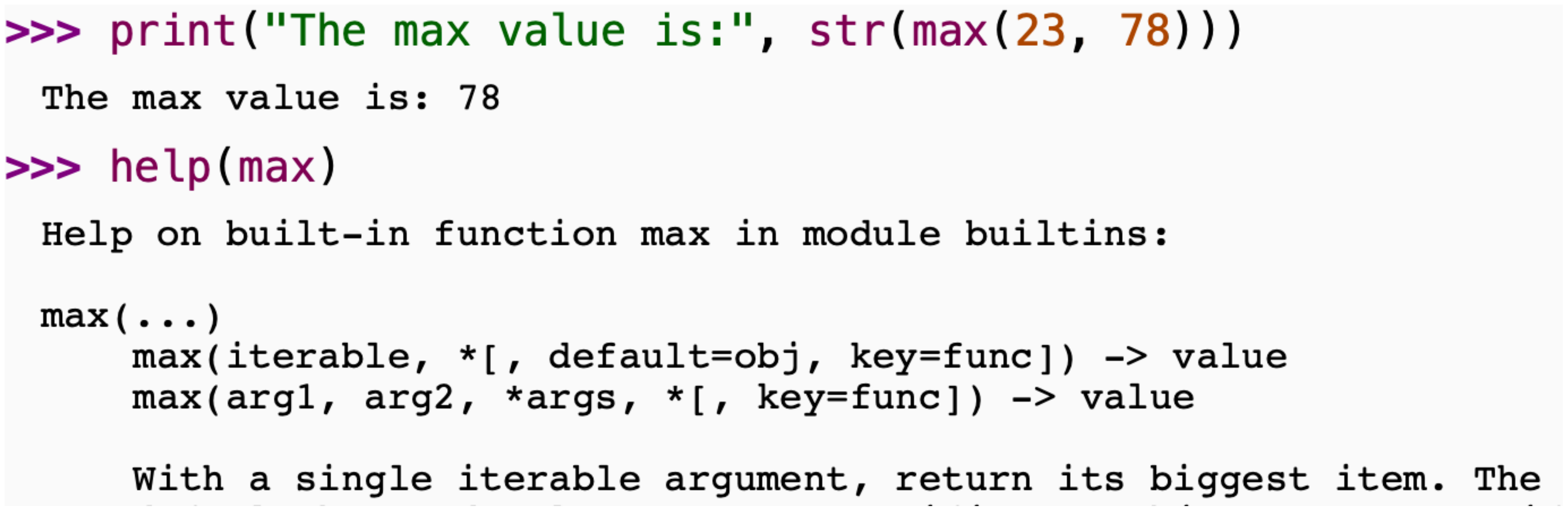
These actions are called **side effects**.

For example, displaying text in the interactive console is a **side effect** of the **`print`** and **`help`** functions:



Concepts in this slide:
side effects

The text outputs in the console are examples of a side effect. There is no value that is produced by calling **`print()`** or **`help()`**.

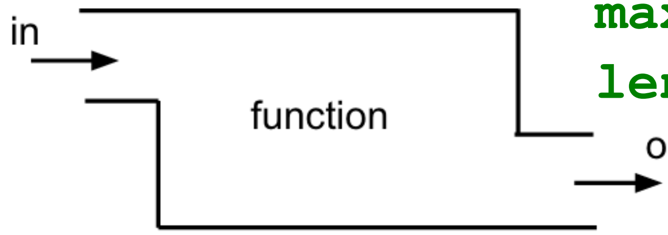


```
>>> print("The max value is:", str(max(23, 78)))
 The max value is: 78
>>> help(max)
 Help on built-in function max in module builtins:

 max(...)
     max(iterable, *[, default=obj, key=func]) -> value
     max(arg1, arg2, *args, *[, key=func]) -> value

     With a single iterable argument, return its biggest item. The
```

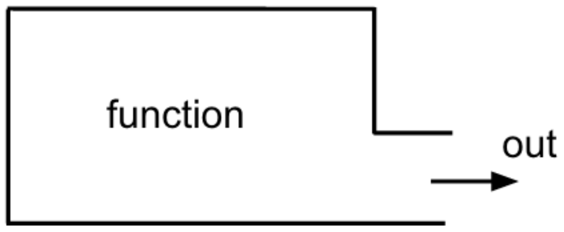
Function diagrams summarize what functions do

Concepts in this slide:
function diagrams

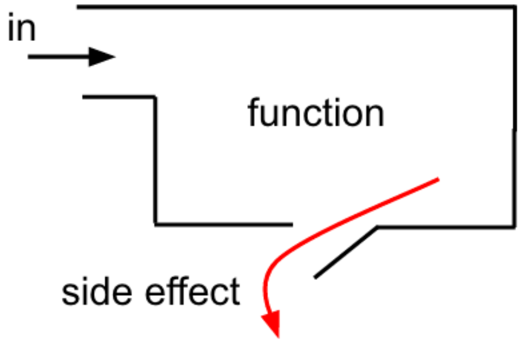
We will see examples of these soon!



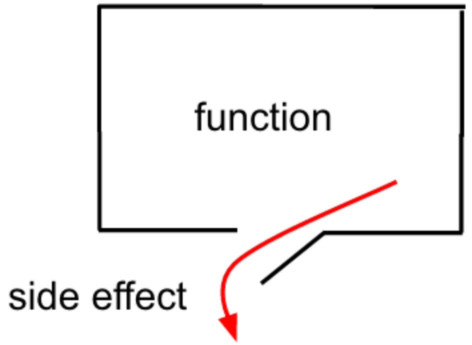
`max()`, `min()`
`len()`, `int()`



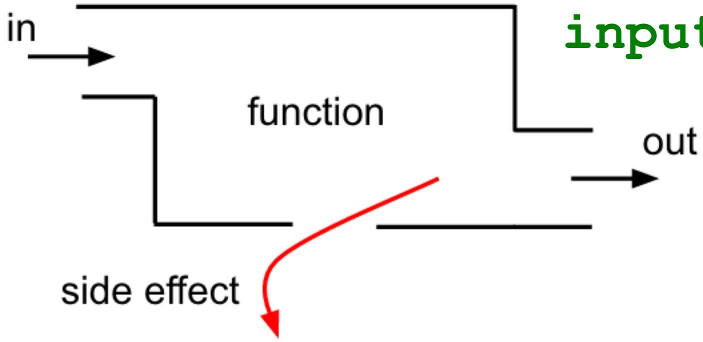
?



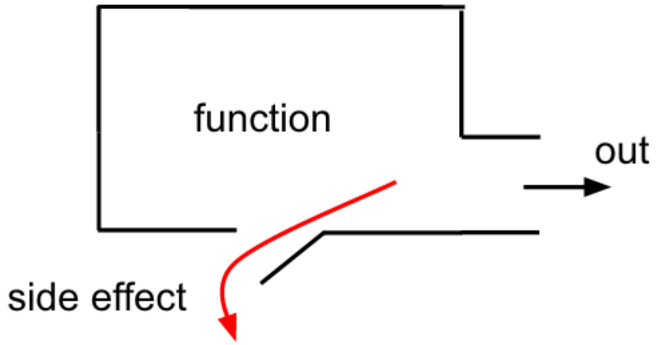
`print()`,
`help()`



?



`input()`



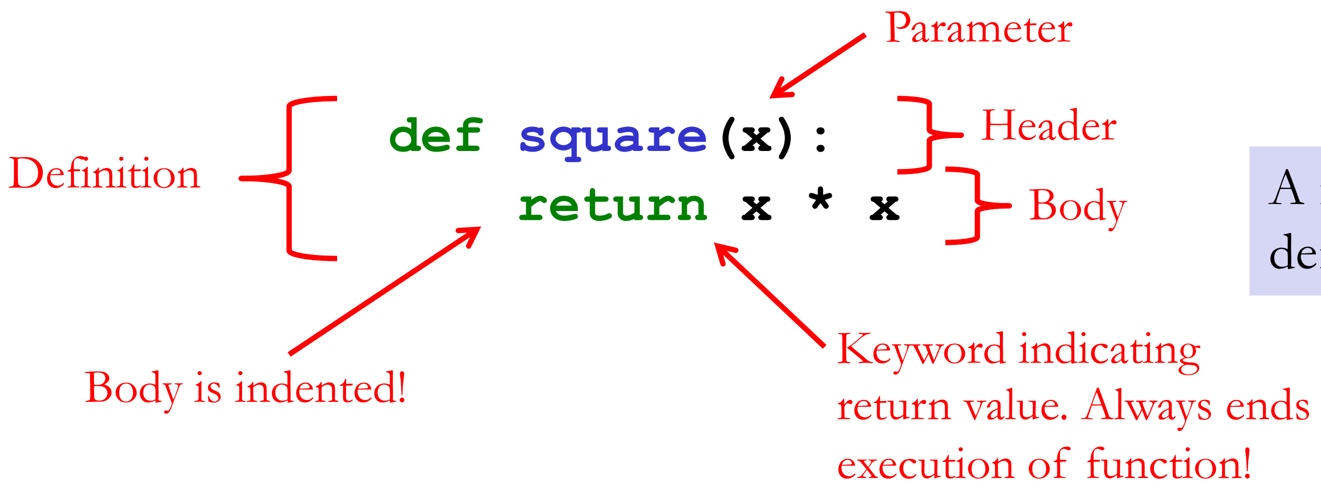
?

Anatomy of a User-defined Function

Concepts in this slide:
function definition,
function call,
parameter and argument

Functions are a way of abstracting over computational processes by capturing common patterns.

Function definitions

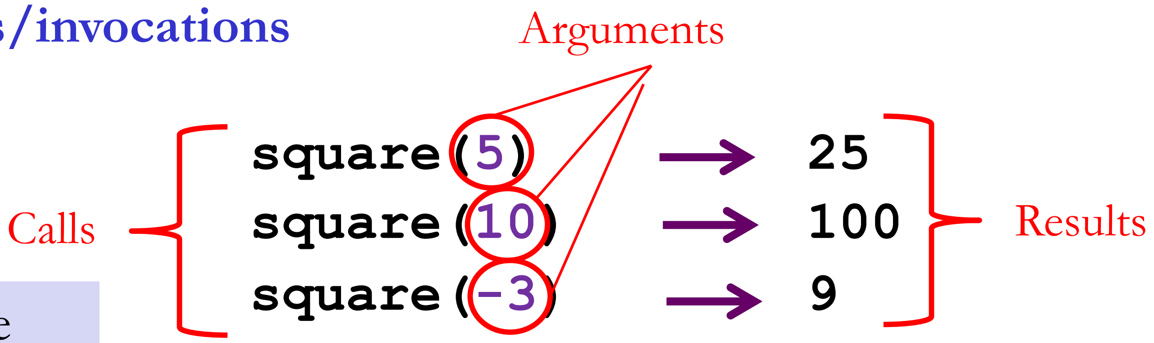


```
def square(x):
    return x * x
```

Parameter
Header
Definition
Body
Body is indented!
Keyword indicating return value. Always ends execution of function!

A function is defined **once**.

Function calls/invocations



```
square(5)   → 25
square(10)  → 100
square(-3)  → 9
```

A function can be called many times.

Parameters and Arguments

Concepts in this slide: Difference between parameters and arguments.

A **parameter** is a variable used in the definition of a function, which will be initialized with an **argument** value during a function call.

The particular name we use for a parameter is irrelevant, as long as we use the name consistently in the function definition.

```
def square(a):
    return a * a
```

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

```
def square(num):
    return num * num
```

```
def square(aLongParameterName):
    return aLongParameterName * aLongParameterName
```

The different parameter names: `a`, `x`, `num`, `aLongParameterName`, used for defining the function `square` do not affect its behavior.

Unindented function body

GOTCHA!

Python is unusual among programming languages in that it uses indentation to determine what's in the body of a function.

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

You can indent by using the TAB character in the keyboard. Alternatively, you can use a consistent number of spaces (e.g. 4).

The following definition is *incorrect* because the body isn't indented:

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

```
SyntaxError: 'return'
outside function
```

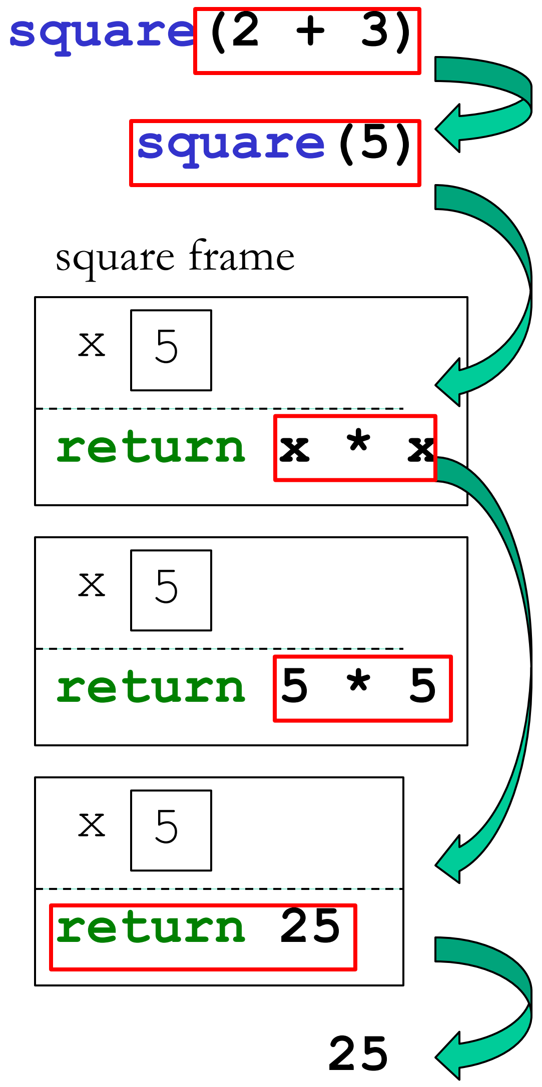
In general, when the indentation is wrong, you'll see error messages that point you to the problem, e.g.:

```
IndentationError: expected an indented block

IndentationError: unindent does not match
any outer indentation level
```

Python Function Call Model

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

We need a model to understand how function calls work.



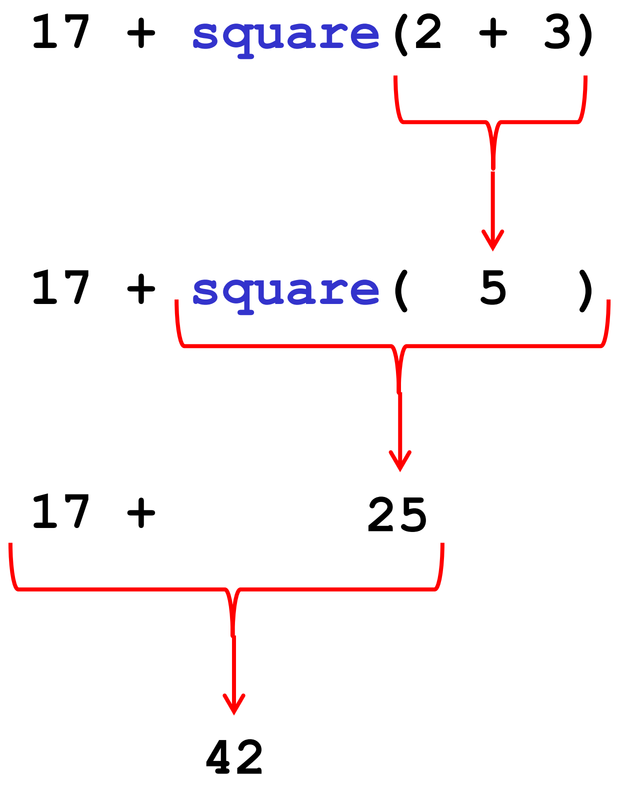
Step 1: Evaluate all argument expressions to values (e.g., numbers, strings, objects …)

Step 2: Create a **function call frame** with (1) a variable box named by each parameter and filled with the corresponding argument value; and (2) the body expression(s) from the function definition.

Step 3: Evaluate the body expression(s), using the values in the parameter variable boxes any time a parameter is referenced. (Do you see why parameter names don't matter as long as they're consistent?)

Step 4: The frame is discarded after the value returned by the frame "replaces" the call

A function call is “replaced” by its returned value



```
17 + square(2 + 3)

17 + square(  5  )

17 +       25

     42
```

Multiple parameters

Concepts in this slide:
Defining multiple parameters.
Using a function from an imported module

A function can take as many parameters as needed. They are separated via comma.

```
def energy(m, v):
    """Calculate kinetic energy"""
    return 0.5 * m * v**2
```

** is Pythons's raise-to-the-power operator

```
def pyramidVolume(len, wid, hgh):
    """Calculate volume rectangular pyramid"""
    return (len * wid * hgh)/3.0
```

Multiple parameters cont.

Concepts in this slide:
Defining multiple parameters.
Using a function from an imported module

A function can take as many parameters as needed. They are separated via comma.

```
import math

def distanceBetweenPoints(x1, y1, x2, y2):
    """Calculate the distance between points """
    return math.sqrt((x2-x1)**2 + (y2-y1)**2)
```

import declaration allows use of Python's `math` module

`math.sqrt` means use the `sqrt` function from Python's `math` module.

FUNCTIONS THAT CALL OTHER FUNCTIONS

Functions calling functions

Concepts in this slide:
User-defined functions can call other user-defined functions.

The function **hypotenuse** calls the **square** function we just defined.

```
import math

def hypotenuse(a, b):
    return math.sqrt(square(a) + square(b))

hypotenuse(3, 4) ⟶ 5.0
hypotenuse(1, 1) ⟶ 1.4142135623730951
```

Functions calling functions

Concepts in this slide: User-defined functions can call other user-defined functions.

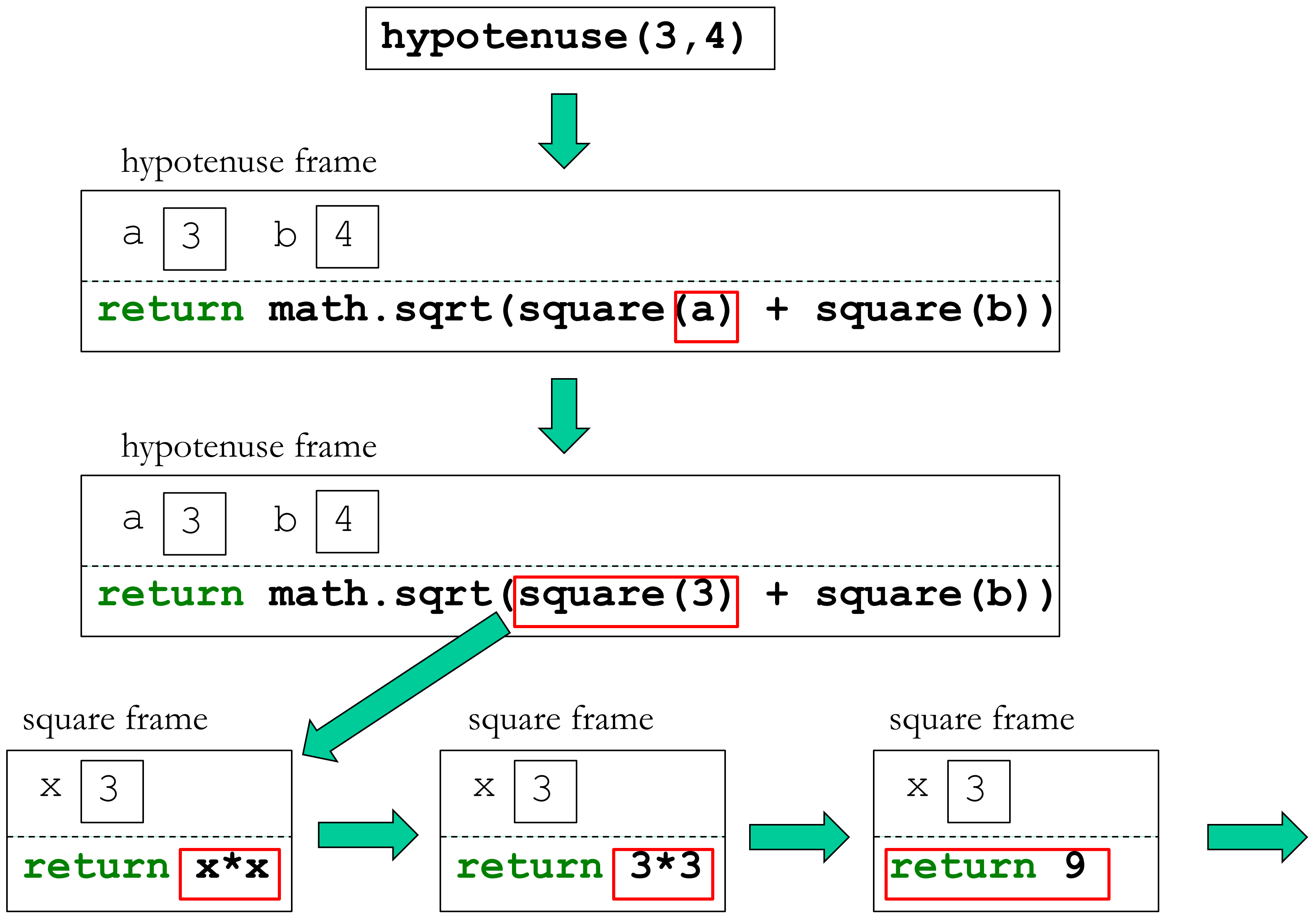
The function **hypotenuse** calls the **square** function we just defined.

```
import math

def hypotenuse(a, b):
    return math.sqrt(square(a) + square(b))

hypotenuse(3, 4) ⟶ 5.0
hypotenuse(1, 1) ⟶ 1.4142135623730951
```

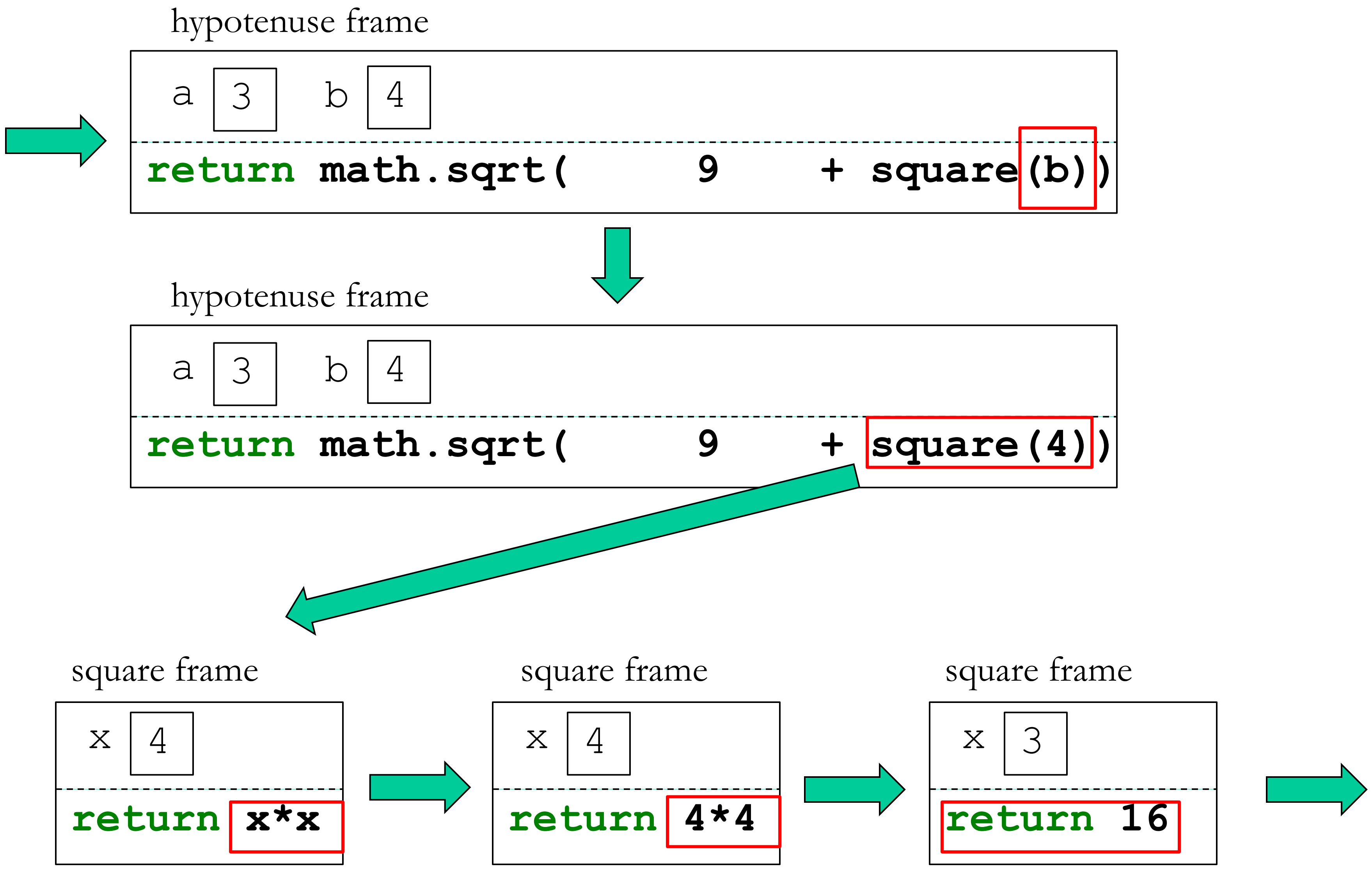
The function **distanceBetweenPoints** calls the **hypotenuse** function defined above.

```
def distanceBetweenPoints(x1, y1, x2, y2):
    """Calculate the distance between points """
    return hypotenuse(x2-x1, y2-y1)
```

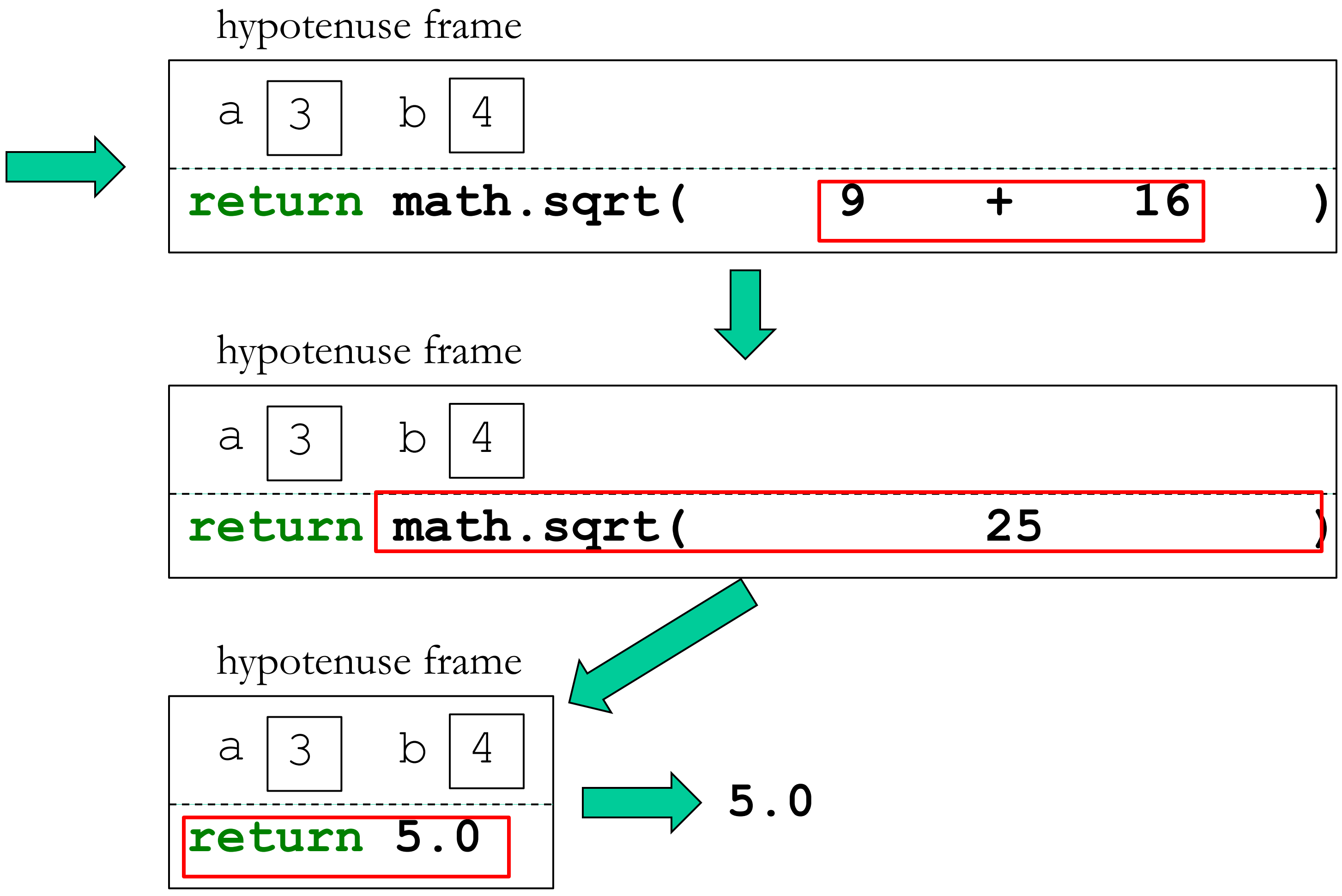
Function call model for `hypotenuse(3,4)` [1]



Function call model for `hypotenuse(3,4)` [2]



Function call model for `hypotenuse(3,4)` [3]



LOCAL VARIABLES

Local variables

Concepts in this slide:
local variables

An assignment to a variable within a function definition creates/changes a **local variable**.

Local variables exist only within a function's body. They cannot be referred outside of it.

Parameters are also local variables that are assigned a value when the function is invoked. They also cannot be referred outside the function.

```
def rightTrianglePerim(a, b):
    c = hypotenuse(a, b)
    return a + b + c

In [1]: rightTrianglePerim(3, 4)
Out [1]: 12.0

In [2]: c
NameError: name 'c' is not defined

In [3]: a
NameError: name 'a' is not defined

In [4]: b
NameError: name 'b' is not defined
```

Local variables in the Frame Model

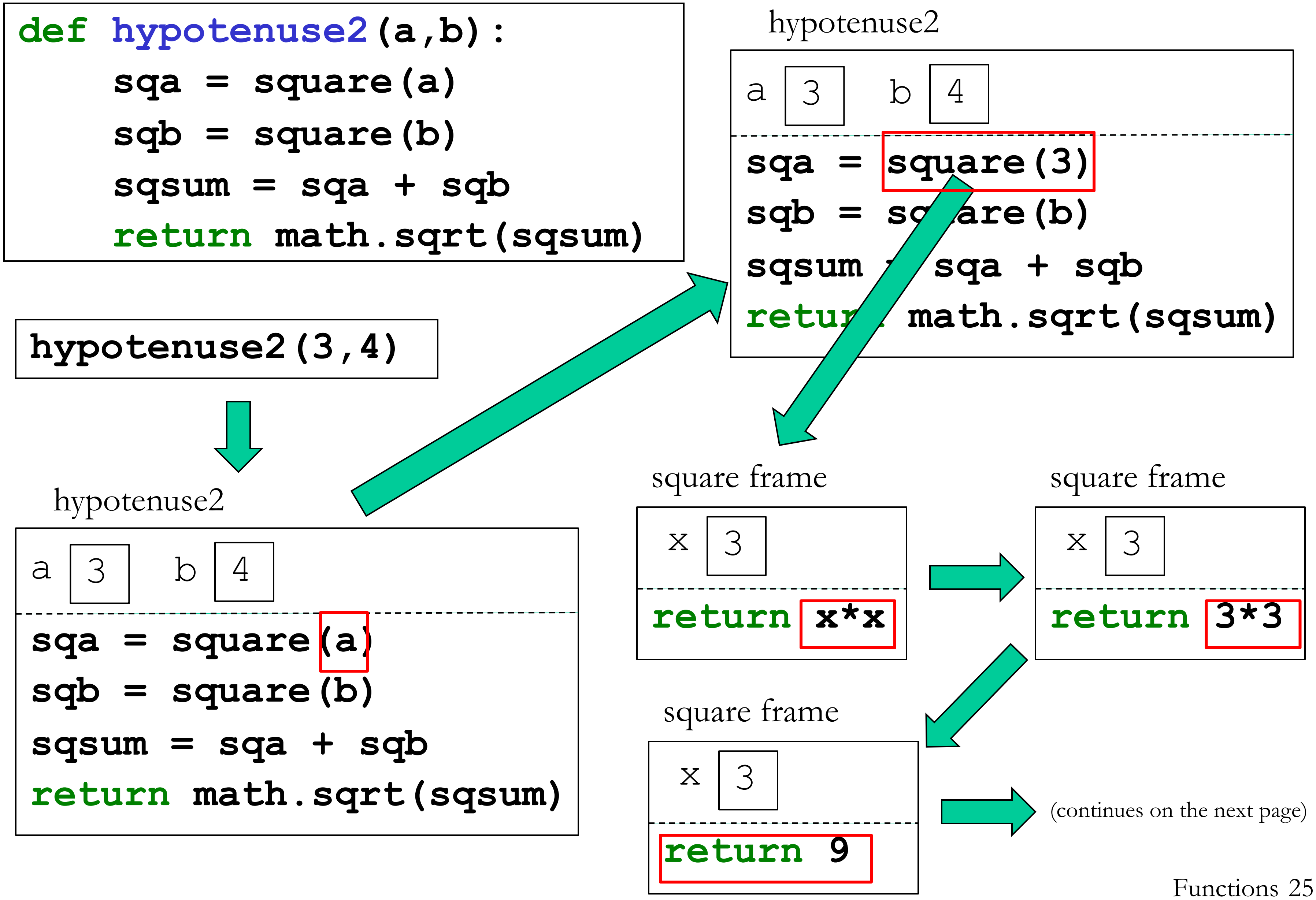
How do local variables work within the function frame model?

Consider the function below which calculates the length of the hypotenuse of a right triangle given the lengths of the two other sides.

```
def hypotenuse2(a,b):
    sqa = square(a)
    sqb = square(b)
    sqsum = sqa + sqb
    return math.sqrt(sqsum)
```

Functions w/local variables: hypotenuse2 [1]

```
def hypotenuse2(a,b):
     sqa = square(a)
     sqb = square(b)
     sqsum = sqa + sqb
     return math.sqrt(sqsum)
```



```
hypotenuse2(3,4)
```

hypotenuse2

```
a 3    b 4
----------------------
sqa = square(a)
sqb = square(b)
sqsum = sqa + sqb
return math.sqrt(sqsum)
```

hypotenuse2

```
a 3    b 4
----------------------
sqa = square(3)
sqb = square(b)
sqsum = sqa + sqb
return math.sqrt(sqsum)
```

square frame

```
x 3
-----------
return x*x
```

square frame

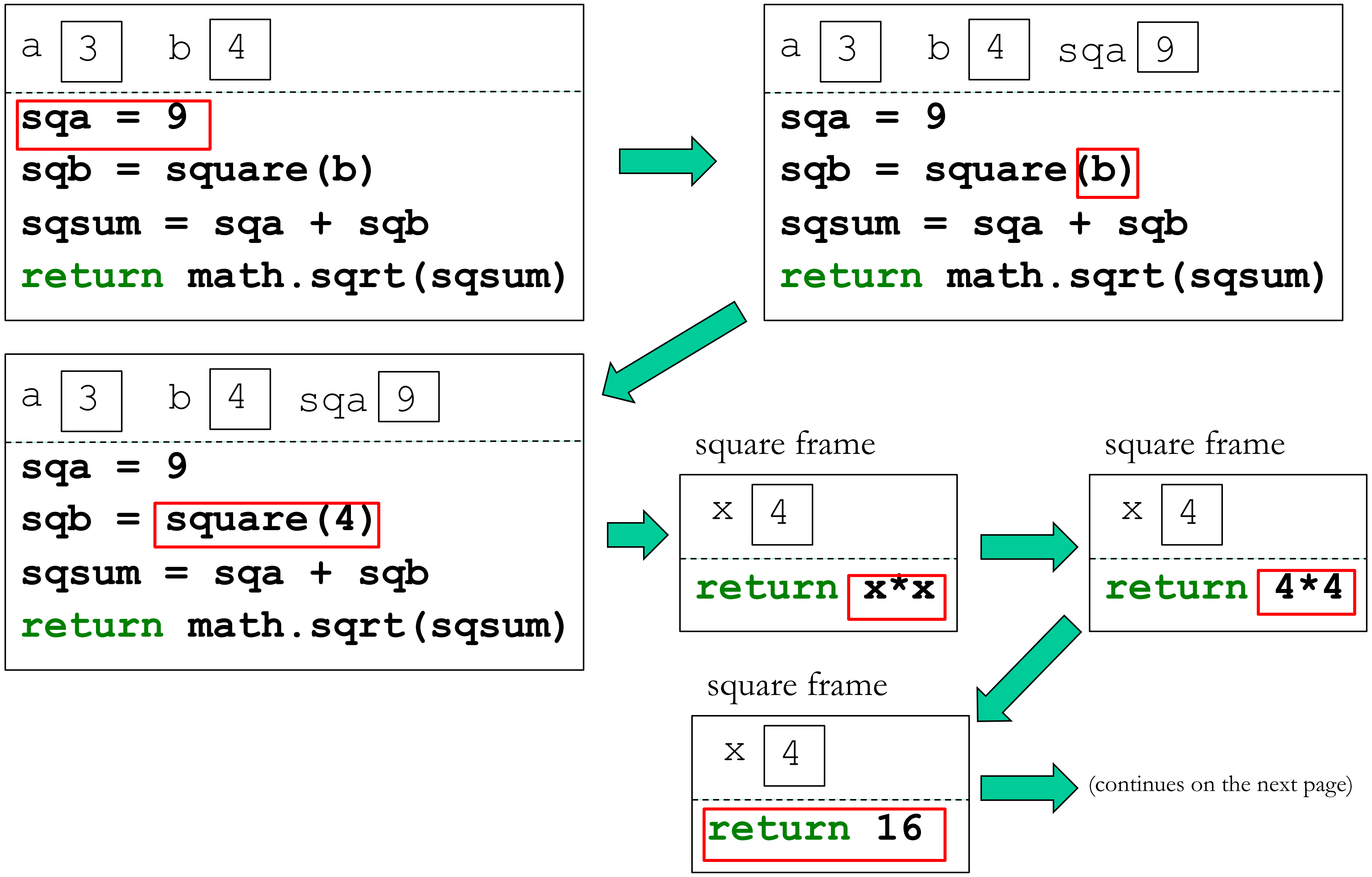
```
x 3
-----------
return 3*3
```

square frame

```
x 3
-----------
return 9
```

(continues on the next page)

Functions w/local variables: hypotenuse2 [2]



```
a 3   b 4
sqa = 9
sqb = square(b)
sqsum = sqa + sqb
return math.sqrt(sqsum)
```

```
a 3   b 4   sqa 9
sqa = 9
sqb = square(b)
sqsum = sqa + sqb
return math.sqrt(sqsum)
```

```
a 3   b 4   sqa 9
sqa = 9
sqb = square(4)
sqsum = sqa + sqb
return math.sqrt(sqsum)
```

square frame

```
x 4
return x*x
```

square frame

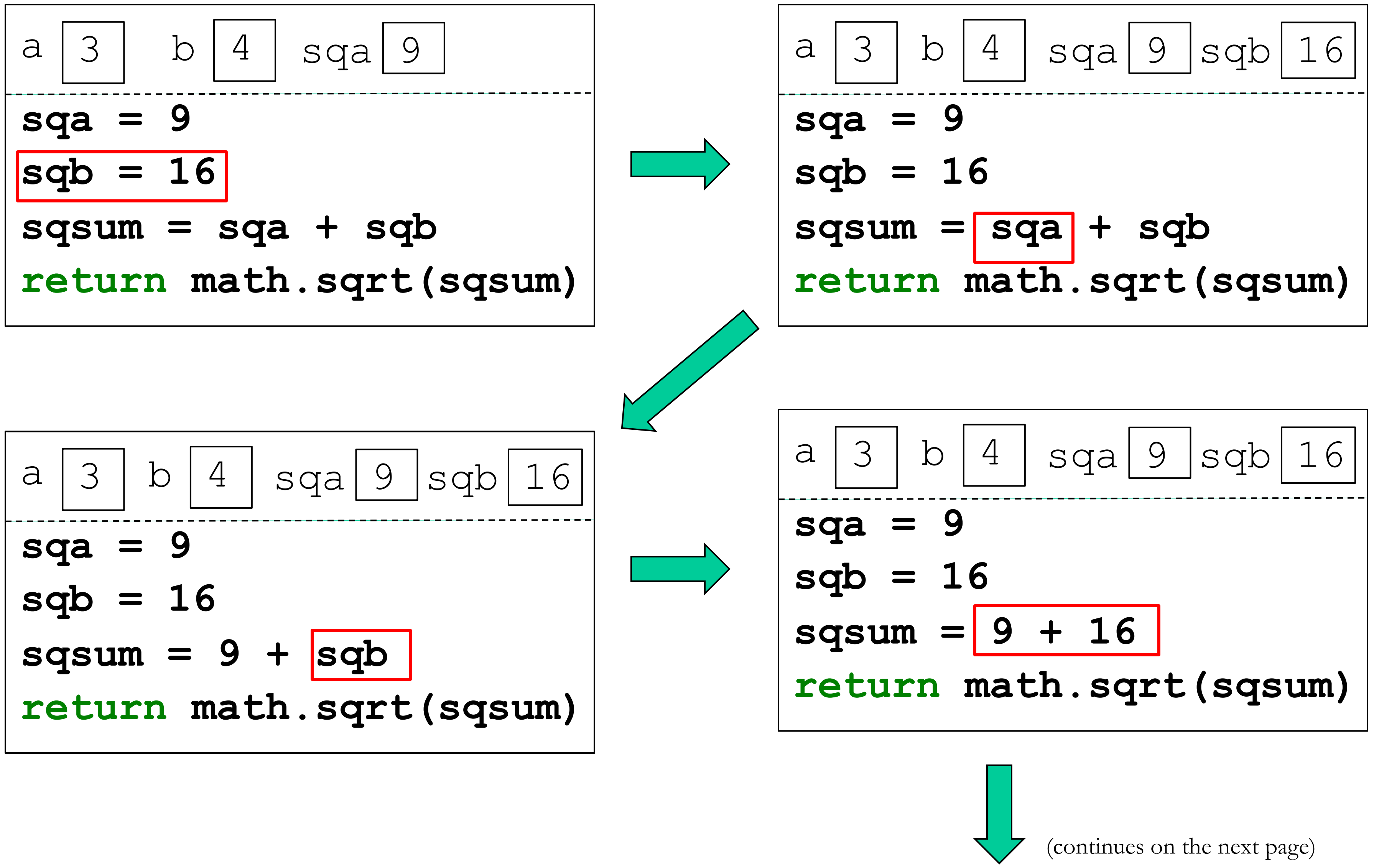
```
x 4
return 4*4
```

square frame

```
x 4
return 16
```

(continues on the next page)

Functions w/local variables: hypotenuse2 [3]



```
a  3   b  4   sqa  9
-----------------------------
sqa = 9
sqb = 16
sqsum = sqa + sqb
return math.sqrt(sqsum)
```

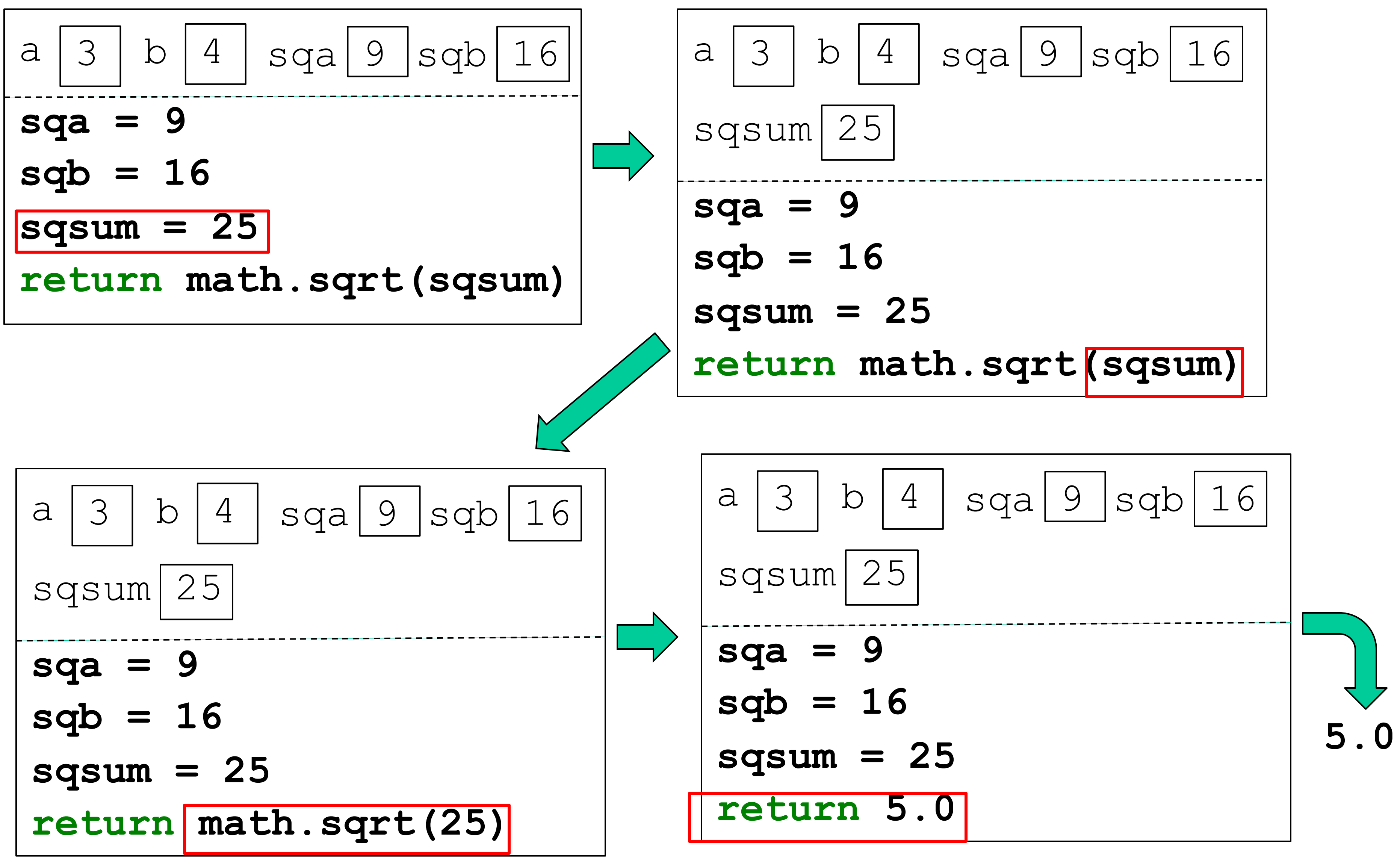
```
a  3   b  4   sqa  9   sqb  16
-----------------------------
sqa = 9
sqb = 16
sqsum = sqa + sqb
return math.sqrt(sqsum)
```

```
a  3   b  4   sqa  9   sqb  16
-----------------------------
sqa = 9
sqb = 16
sqsum = 9 + sqb
return math.sqrt(sqsum)
```

```
a  3   b  4   sqa  9   sqb  16
-----------------------------
sqa = 9
sqb = 16
sqsum = 9 + 16
return math.sqrt(sqsum)
```

(continues on the next page)

Functions w/local variables: hypotenuse2 [4]


a 3 b 4 sqa 9 sqb 16
a 3 b 4 sqa 9 sqb 16 sqsum 25
a 3 b 4 sqa 9 sqb 16 sqsum 25
a 3 b 4 sqa 9 sqb 16 sqsum 25
5.0

```
sqa = 9
sqb = 16
sqsum = 25
return math.sqrt(sqsum)
```

```
sqa = 9
sqb = 16
sqsum = 25
return math.sqrt(sqsum)
```

```
sqa = 9
sqb = 16
sqsum = 25
return math.sqrt(25)
```

```
sqa = 9
sqb = 16
sqsum = 25
return 5.0
```

RETURN VS. PRINT

Concepts in this slide: return and print are different!

Output of a function: `return` vs. `print`:

- `return` specifies the result of the function invocation
- `print` causes characters to be displayed in the shell (side effect).

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

def squarePrintArg(x):
    print('The argument of square is ' + str(x))
    return x*x
```

```
In [2]: square(3) + square(4)
Out[2]: 25

In [3]: squarePrintArg(3) + squarePrintArg(4)
The argument of square is 3
The argument of square is 4
Out[3]: 25
```

Concepts in this slide: return and print are different!

Don't confuse **return** with **print**!

```
def printSquare(a):
  print('square of ' + str(a) + ' is ' + str(square(a)))
```

```
In [4]: printSquare(5)
square of 5 is 25

In [5]: printSquare(3) + printSquare(4)
square of 3 is 9
square of 4 is 16
---------------------------------------------------------------------------
TypeError                         Traceback (most recent call last)
<ipython-input-10-ff81dee8cf8f> in <module>()
----> 1 printSquare(3) + printSquare(4)
```

`printSquare` **does not return** a number, so it doesn't make sense to add the two invocations!

The None value and NoneType

Concepts in this slide:
None is a type in Python

Python has a special **None** value (of type **NoneType**), which Python normally doesn't print.

```
In [2]: None

In [3]: type(None)
Out[3]: NoneType

In [4]: None + None
---------------------------------------------------------------------------
TypeError                     Traceback (most recent call last)
<ipython-input-7-28a1675638b9> in <module>()
----> 1 None + None

TypeError: unsupported operand type(s) for +: 'NoneType' and
'NoneType'
```

The None value and NoneType

Concepts in this slide:
None is a type in Python

A function without an explicit **return** statement actually returns the **None** value!

```
def printSquare(a):
  print('square of ' + str(a) + ' is ' + str(square(a)))
```

Is treated as if it were written as

```
def printSquare(a):
  print('square of ' + str(a) + ' is ' + str(square(a)))
  return None
```

This is the real reason that the expression
`printSquare(3) + printSquare(4)` causes an error.

Fruitful vs. None Functions

Concepts in this slide:
Two classes of functions

We call functions that return the None value None functions*. None functions are invoked to perform an action (e.g. print characters), not to return a result.

We will call functions that return a value (other than None) fruitful functions. **Fruitful functions return a meaningful value**. Additionally, they may also perform an action.

Fruitful functions	None functions
`int`	**`print`**
`square`	**`help`**
`square_print`	**`printSquare`**
`hypotenuse`	

* In Java (another programming language), methods that don't return a value are void methods. We sometimes use "void functions" as a synonym for "None functions"

Incremental Development

When writing your own functions or any other type of code, do not attempt to write it all at once!

Instead, develop code in a sequence of incremental steps, each of which makes a small amount of progress toward the final goal. Test each step to make sure it works before proceeding to the next step.

Store longer expressions into variables with meaningful names, and reference those variables later in your code. (Examples on the next slide)

Concepts in this slide:
Incremental Development

Incremental Development

Example: create a function named **`numStats`** that takes in two numbers, prints out the two numbers with their average, and returns the product of the two numbers.

Step 1: First, create the function header and print the arguments

```
def numStats(num1, num2):
  # print the two numbers
  print("num1 is", num1, "and num2 is", num2)
```

Step 2: Next, calculate and print the average of the two numbers

```
def numStats(num1, num2):
    # print the two numbers with their average
    average = (num1+num2)/2
    print("The average of", num1, "and", num2,
          "is", average + "."))
```

Step 3: Finally, return the product of the two numbers

```
def numStats(num1, num2):
    # print the two numbers with their average
    average = (num1+num2)/2
    print("The average of", num1, "and", num2,
          "is", average + "."))
    # return the product of the two numbers
    product = num1 * num2
    return product
```

FUNCTIONS AND TURTLES

Concepts in this slide: Useful functions from the turtle module.

Turtle Graphics

Python has a built-in module named **`turtle`**. See the Python turtle module API for details.

Use **`from turtle import *`** to use these commands:

`fd(`*`dist`*`)`	turtle moves forward by *dist*
`bk(`*`dist`*`)`	turtle moves backward by *dist*
`lt(`*`angle`*`)`	turtle turns left *angle* degrees
`rt(`*`angle`*`)`	turtle turns right *angle* degrees
`pu()`	(pen up) turtle raises pen in belly
`pd()`	(pen down) turtle lower pen in belly
`pensize(`*`width`*`)`	sets the thickness of turtle's pen to *width*
`pencolor(`*`color`*`)`	sets the color of turtle's pen to *color*
`shape(`*`shp`*`)`	sets the turtle's shape to *shp*
`home()`	turtle returns to (0,0) (center of screen)
`clear()`	delete turtle drawings; no change to turtle's state
`reset()`	delete turtle drawings; reset turtle's state
`setup(`*`width,height`*`)`	create a turtle window of given *width* and *height*

turtleBeads

Concepts in this slide: Useful functions from the turtle module.

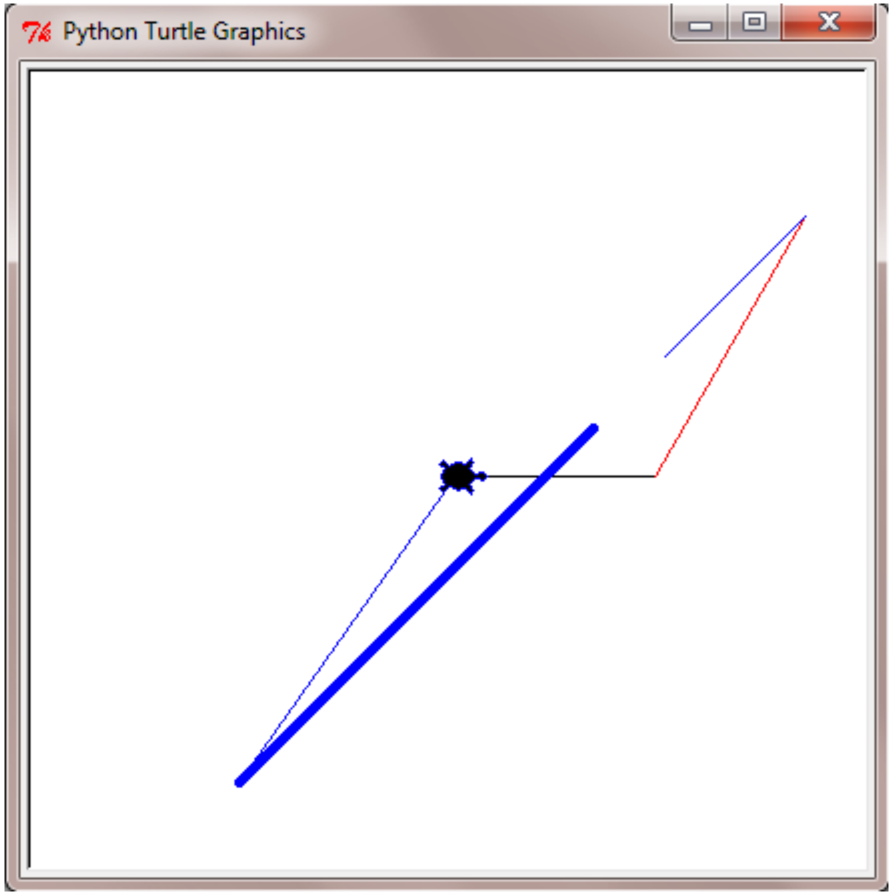
In CS111, we use a custom module called **`turtleBeads`** which allow us to teleport, draw simple shapes, and more:
https://cs111.wellesley.edu/reference/quickref/#turtle

Use `from turtleBeads import *` to use these commands:

`teleport(x,y)`	turtle moves to *x,y* coordinate without drawing
`leap(dist)`	turtle moves forward by *dist* without drawing
`hop(dist)`	turtle along the x-axis by *dist* without drawing
`drawCircle(radius)`	draws a circle centered on the current turtle position
`drawEllipse(r1, r2)`	draws ellipse with x radius *r1* and y radius *r2*
`drawDot(radius)`	draws filled with radius size *radius*
`fontsize(size)`	sets font size to *size*
`setupTurtle()`	resets everything including window title and background
`noTrace()`	turns off animation
`doTrace()`	turns on animation
`showPicture()`	updates the display
`randomPastelColor()`	returns random color name as a string from a fixed set
`randomWarmColor()`	returns random color name as a string from fixed warm hues

A Simple Example with Turtles

Concepts in this slide: The only two commands that draw lines are **fd** and **bk**.



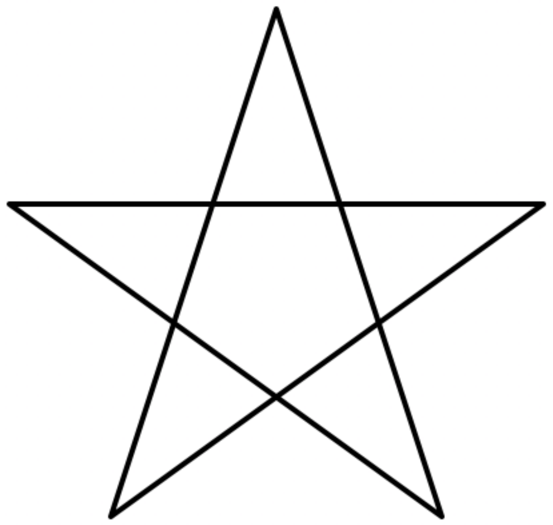
```
from turtle import *

setup(400,400)
fd(100)
lt(60)
shape('turtle')
pencolor('red')
fd(150)
rt(15)
pencolor('blue')
bk(100)
pu()
bk(50)
pd()
pensize(5)
bk(250)
pensize(1)
home()
exitonclick()
```

Tk window
The turtle module has its own graphics environment that is created when we call the function **setup**. All drawing happens in it.

Turtle Functions

Functions help make code for turtle graphics more concise and simple.

```
def star(startX, startY, length):
    teleport(startX, startY)
    rt(72)
    fd(length)
    rt(144)
    fd(length)
    rt(144)
    fd(length)
    rt(144)
    fd(length)
    rt(144)
    fd(length)
    rt(72)
```



Making more stars is as simple as calling the function multiple times.

```
star(0, 100, 100)
star(200, 100, 200)
star(-200, 100, 200)
```

The body of the function captures the similarities of all stars while the parameters express the differences.

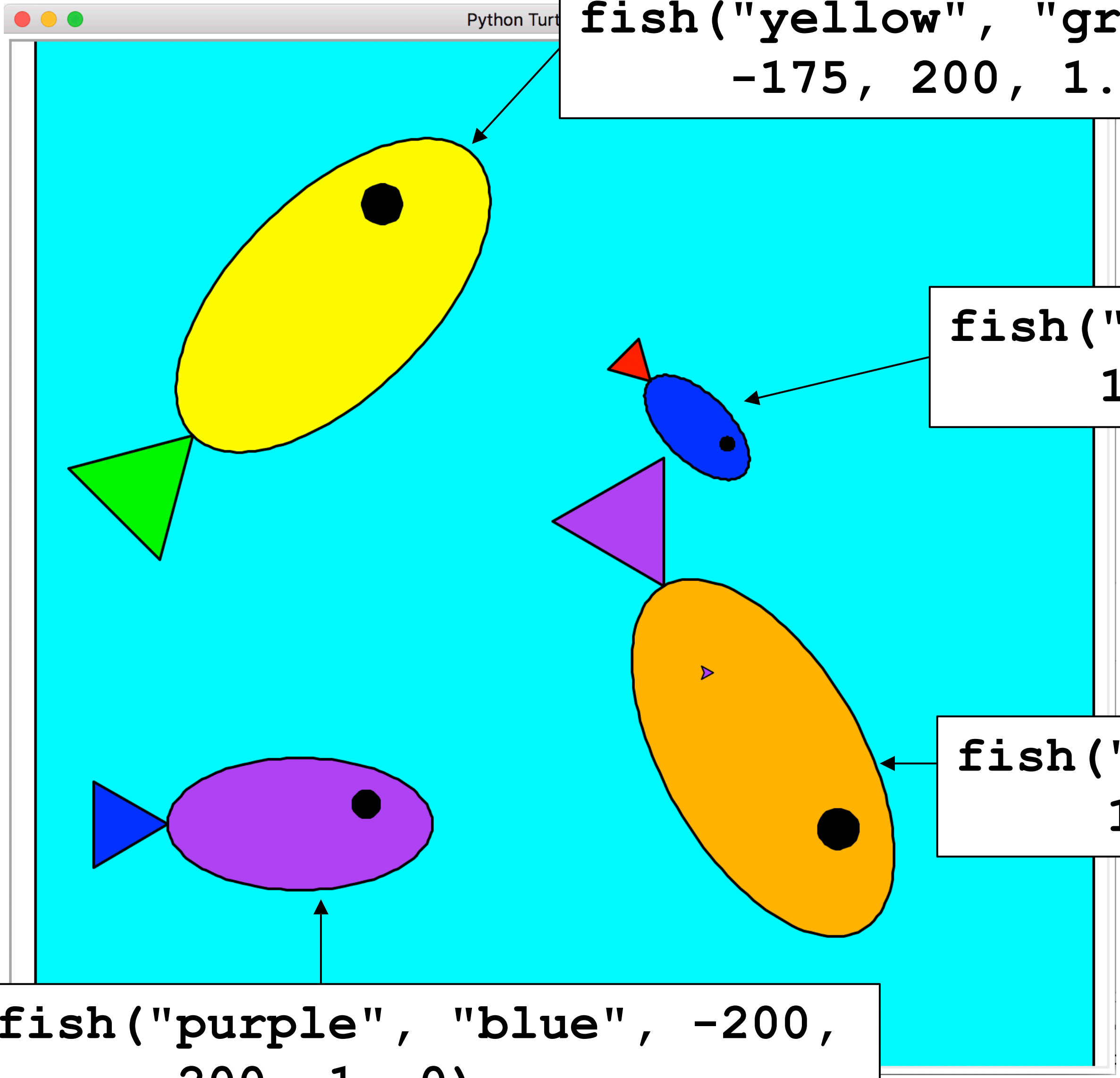
Fish Tank

```
def staticFish():
    # Make the body
    fillcolor("yellow")
    begin_fill()
    drawEllipse(50, 2)
    end_fill()
    # Make the eye
    penup()
    fd(50)
    lt(90)
    fd(15)
    rt(90)
    pendown()
    fillcolor("black")
    begin_fill()
    drawCircle(10)
    end_fill()
    # SOME CODE OMITTED.
    # SEE NOTEBOOK.
```

To make the fish tank shown on the opening slide and the next slide, we need to amend the code on the left so that it can produce fishes of different size, orientation and color. How can we do that? Use parameters to capture the differences and keep the body of the code that captures the similarities. See lecture code solution for answers! The new function header is given below as a start!

```
def fish(bodyColor, tailColor, x, y, scale, angle):
```

Fish Tank



```
fish("yellow", "green",
     -175, 200, 1.5, 45)
```

```
fish("blue", "red", 100,
     100, 0.5, -45)
```

```
fish("orange", "purple",
     150, -150, 1.5, -60)
```

```
fish("purple", "blue", -200,
     -200, 1, 0)
```

Fruitful Turtles

We say a function is fruitful if it returns a value. See slide 34 for more info!

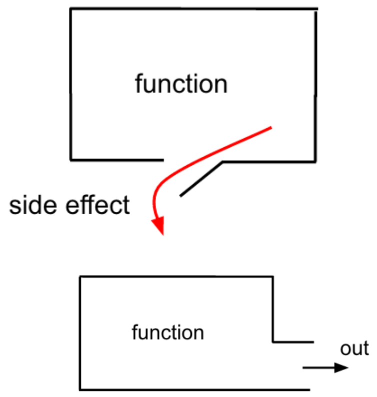
With turtle graphics, we often make a function fruitful if we want to return some statistic about the shape or picture we drew. The code on the right draws a triangle but also returns the perimeter of the triangle!

```
def trianglePlusPerimeter(size):
    rt(60)
    fd(size)
    rt(120)
    fd(size)
    rt(120)
    fd(size)
    rt(60)
    return size * 3

reset()
setupTurtle()
trianglePlusPerimeter(50)
```

Return the perimeter of the triangle after it has been drawn.

OTHER TYPES OF FUNCTIONS

Zero-Parameter Functions



Sometimes it's helpful to define/use functions that have zero parameters. Note: you still need parentheses after the function name when defining and invoking the function.

```
def rocks():
    print('CS111 rocks!')
```

Invoking **`rocks()`**

```
CS111 rocks!
```

```
def rocks3():
    rocks()
    rocks()
    rocks()
```

Invoking **`rocks3()`**

```
CS111 rocks!
CS111 rocks!
CS111 rocks!
```

Python libraries have useful built-in functions with zero parameters and a return value:

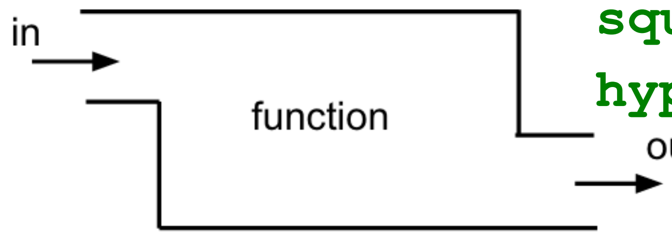
```
import random
random.random()
```

Out […]

```
0.72960321
```

A random float value between 0 and 1.

Updated Function diagrams

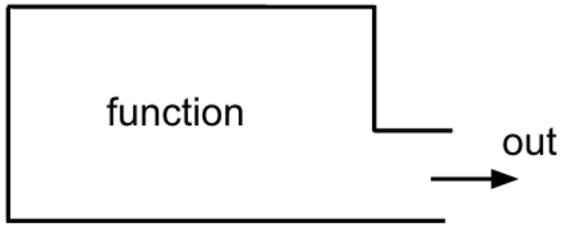


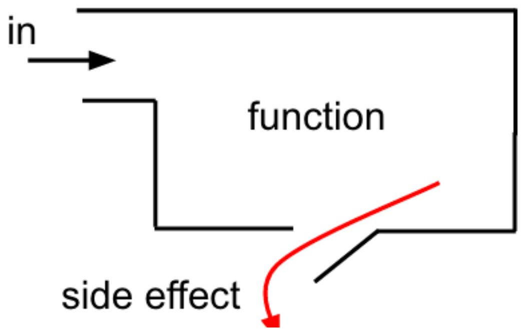
`max(), min()`

`square()`

`hypotenuse()`

`random.random()`



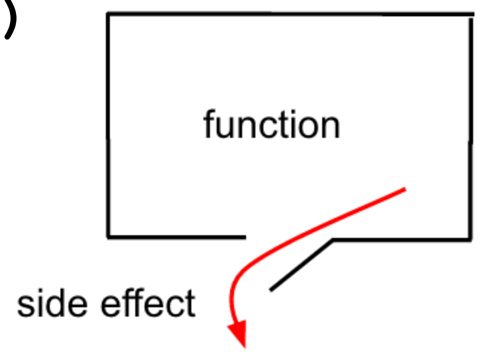


`print(), help()`

`verse()`

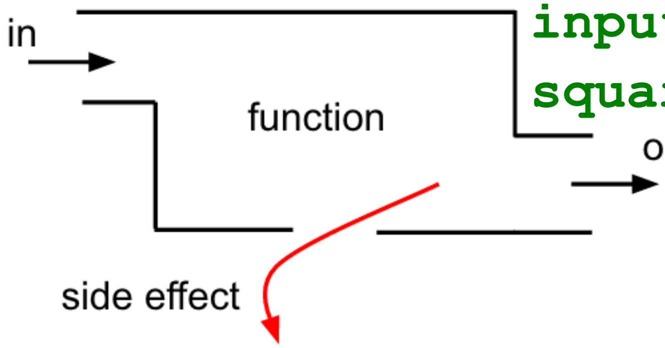
`printBanner()`

`printSquare()`



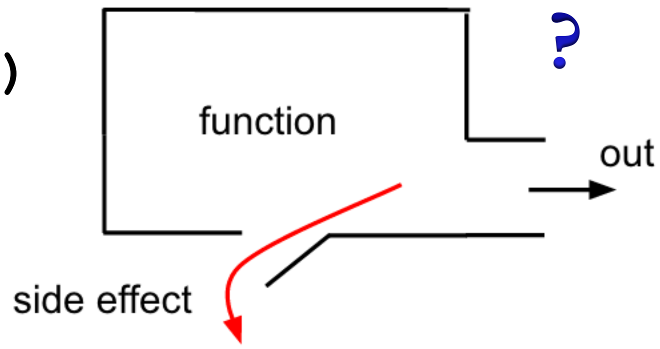
`rocks()`

`rocks3()`



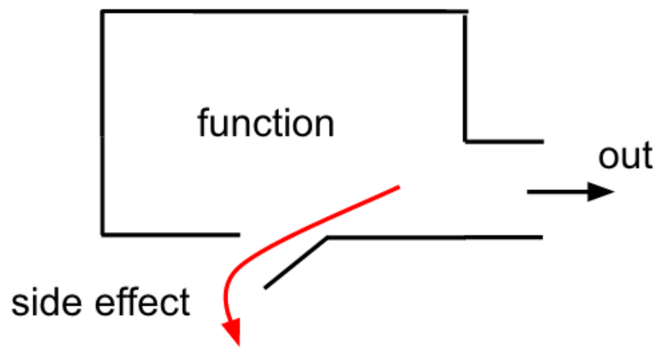
`input(),`

`squarePrintArg()`



Zero-Parameter Functions (continued)

We haven't seen an example yet of our last function diagram. There are no built-in functions that fulfill this contract.



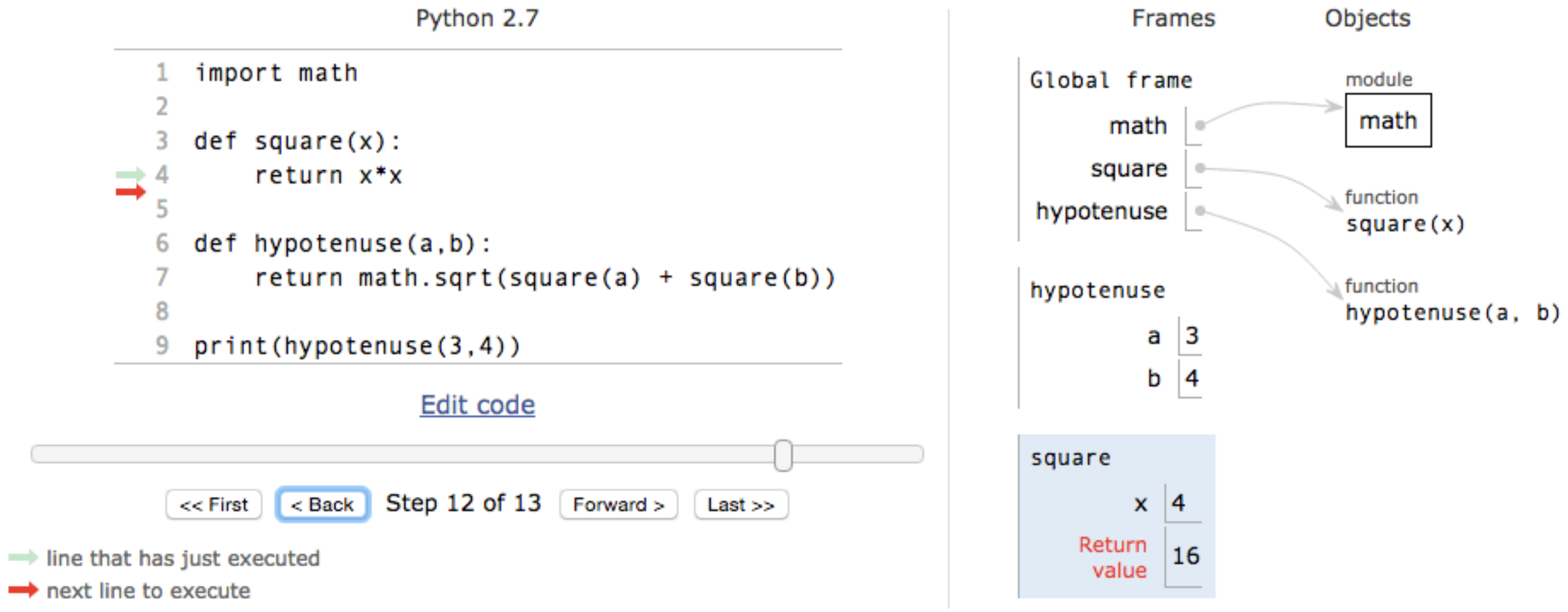
Exercise: Can you write a function that takes no input and produces a side-effect while returning a value?

Hint: printing is always a good way to produce a side-effect! Try and write a meaningful function that would fulfill these two criteria.

Visualizing Code Execution with the Python Tutor

Python Tutor: http://www.pythontutor.com/visualize.html

It automatically shows many (but not all) aspects of our CS111 Python function call model.



Python 2.7

```
import math

def square(x):
    return x*x

def hypotenuse(a,b):
    return math.sqrt(square(a) + square(b))

print(hypotenuse(3,4))
```

Test your knowledge

1. What is the difference between a function definition and a function call?
2. What is the difference between a parameter and an argument? In what context is each of them used?
3. Is it OK to use the same parameter names in more than one function definition? Why or why not?
4. Can a function have a return value and no side effects? Side effects and no return value? Both side effects and a return value?
5. Can a function whose definition lacks a **`return`** statement be called within an expression?
6. What is the value of using the function call model?
7. What is indentation and where it is used within Python?
8. Can a turtle function both draw and return a value?
9. How do functions relate to the idea of abstraction?