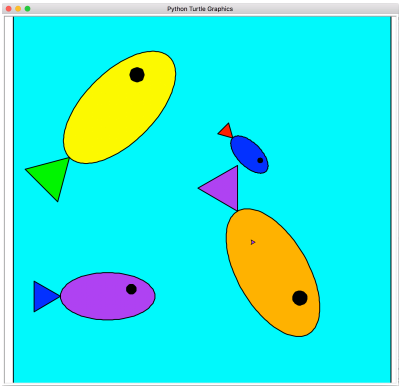
Abstracting with Functions



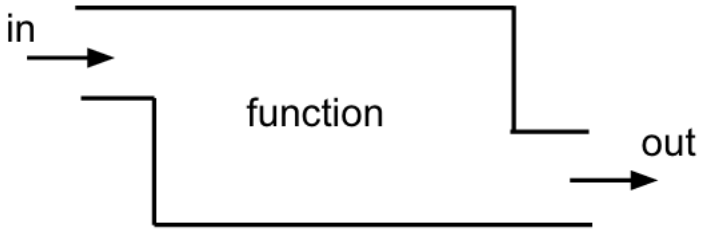
CS111 Computer Programming

Department of Computer Science
Wellesley College

FUNCTION BASICS

Concepts in this slide:
functions,
input & output

Functions take inputs and return outputs based on those inputs



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

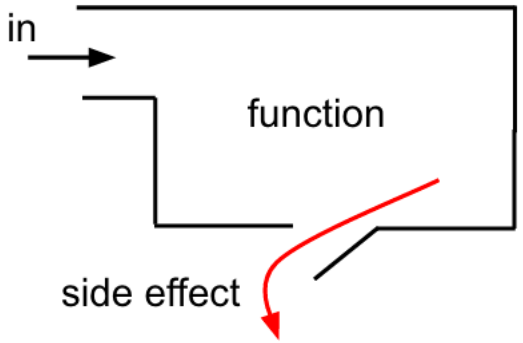
In [...]	Out [...]
`max(7,3)`	`7`
`min(7,3,2,9)`	`2`
`type(123)`	`int`
`len('CS111')`	`5`
`str(4.0)`	`'4.0'`
`int(-2.978)`	`-2`
`float(42)`	`42.0`
`round(2.718, 1)`	`2.7`

Some functions perform actions instead of returning outputs

Concepts in this slide:
side effects

These actions are called **side effects**.

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

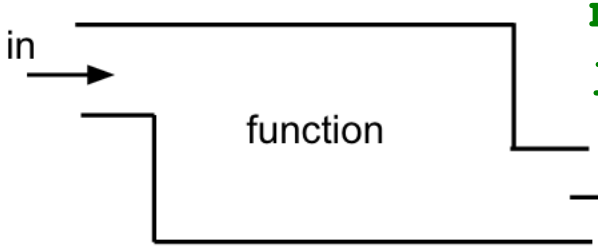



```
>>> print("The max value is:", str(max(23, 78)))
The max value is: 78
>>> help(max)
Help on built-in function max in module builtins:

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
```

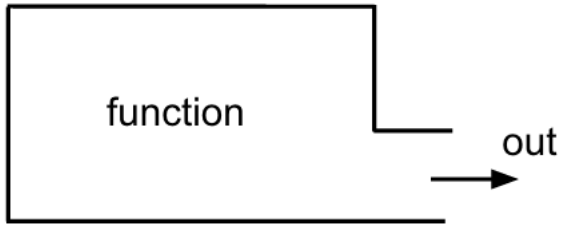
Function diagrams summarize what functions do

Concepts in this slide: function diagrams

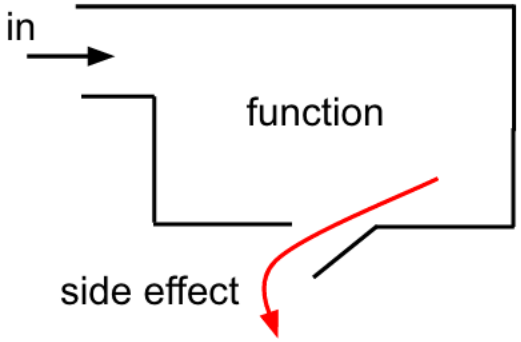
We will see examples of these soon!



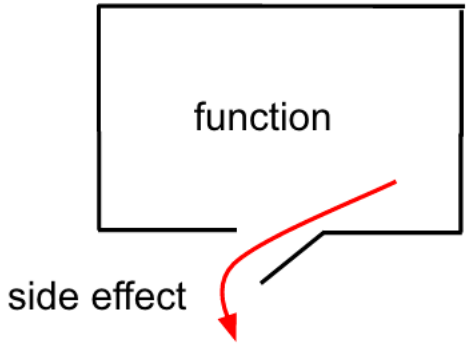
`max()`, `min()`
`len()`, `int()`



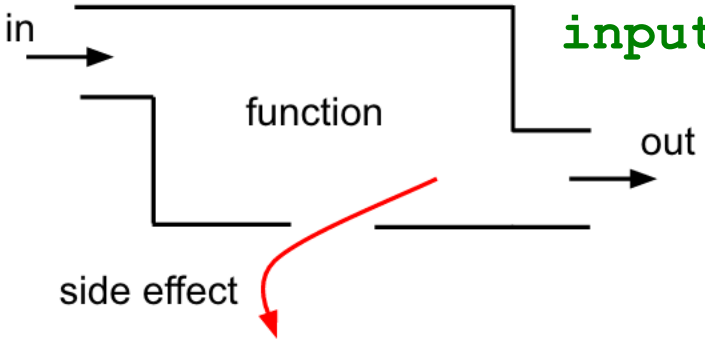
?



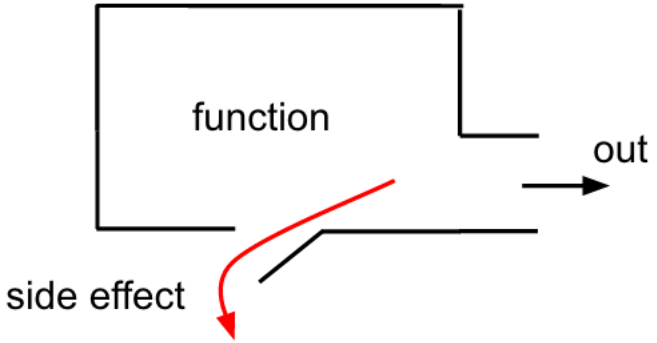
`print()`,
`help()`



?



`input()`



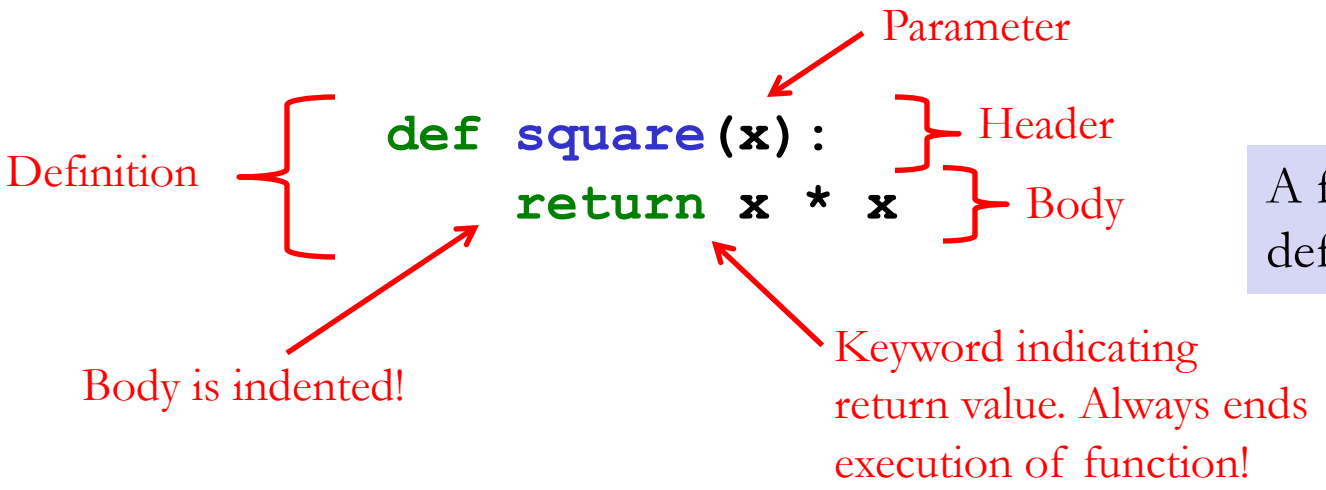
?

Anatomy of a User-defined Function

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

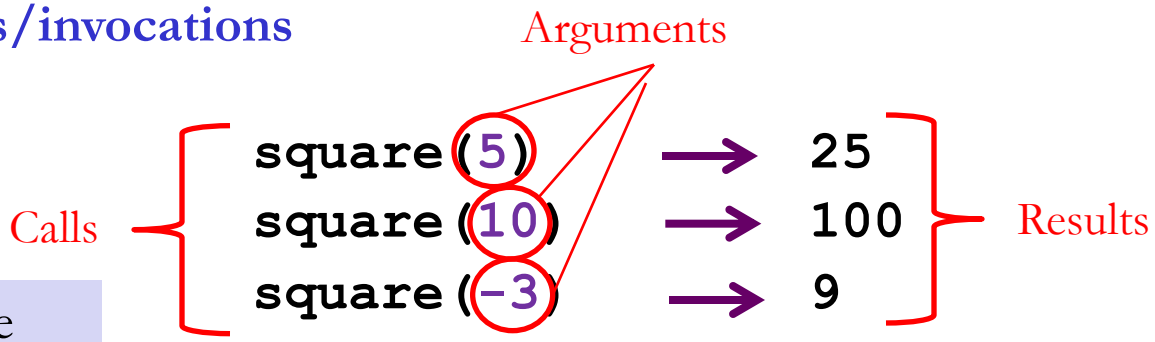
Functions are a way of abstracting over computational processes by capturing common patterns.

Function definitions



A function is defined **once**.

Function calls/invocations



A function can be called many times.

Concepts in this slide: Difference between parameters and arguments.

Parameters

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.:

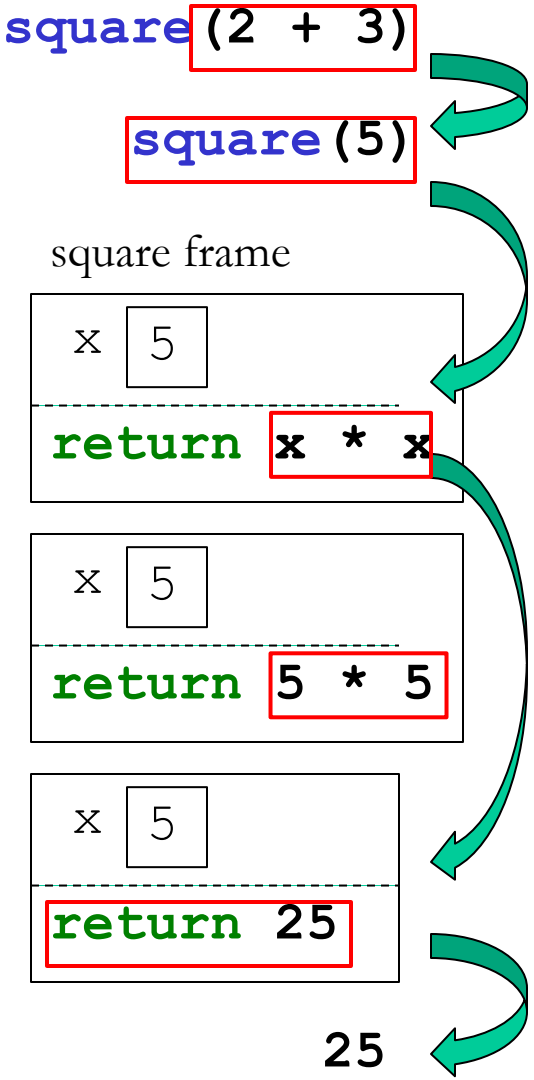
```
IndentationError: expected an indented block
```

```
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.



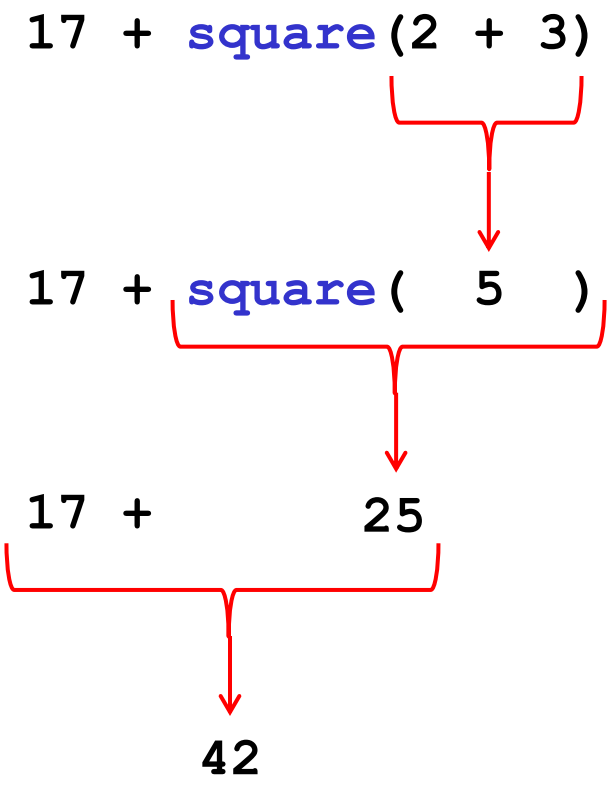
Step 1: evaluate all argument expressions to values (e.g., numbers, strings, objects …)

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



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
```

```
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

Calling other functions

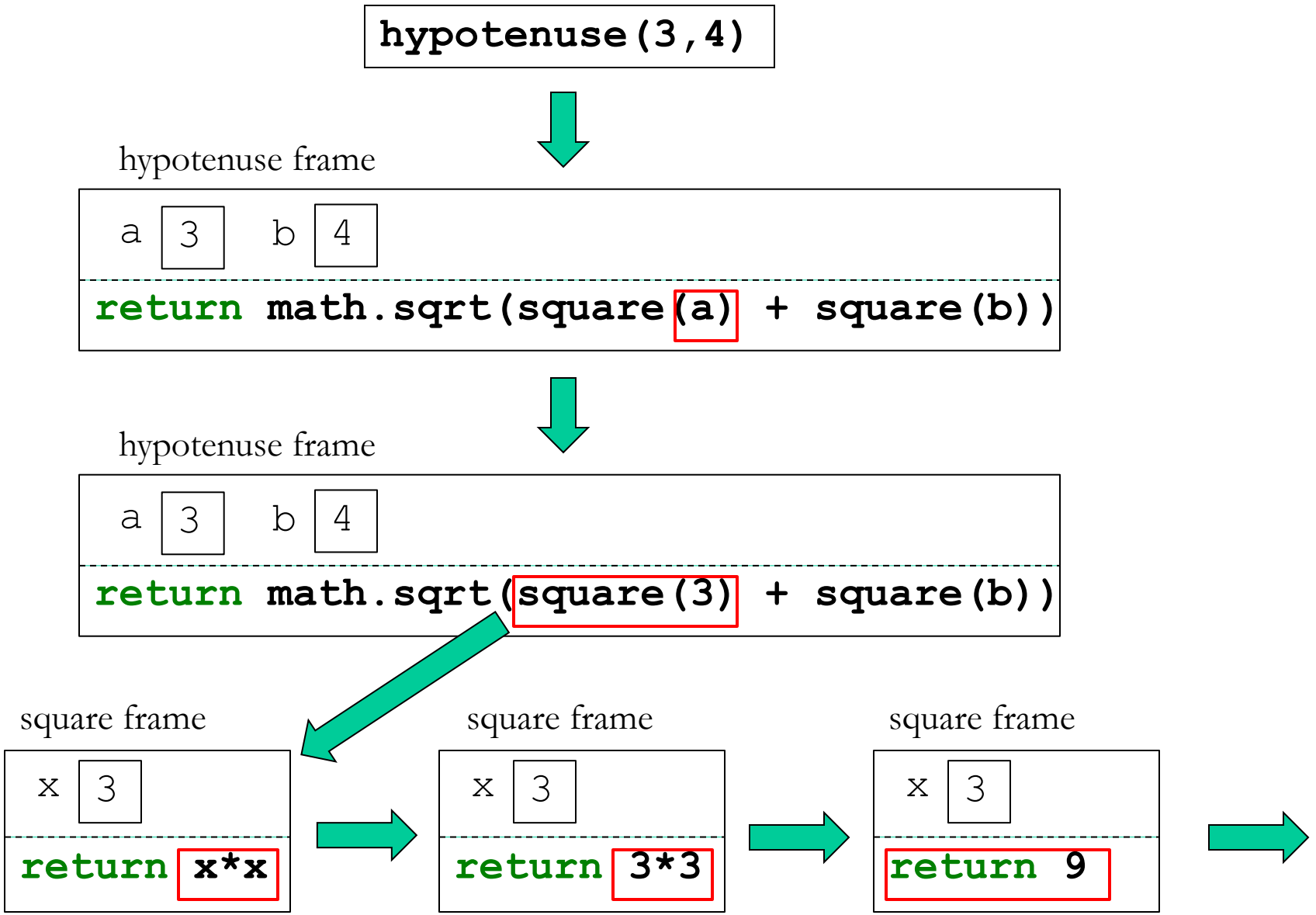
Concepts in this slide: User-defined functions can call other user-defined functions.

Functions can call other functions:

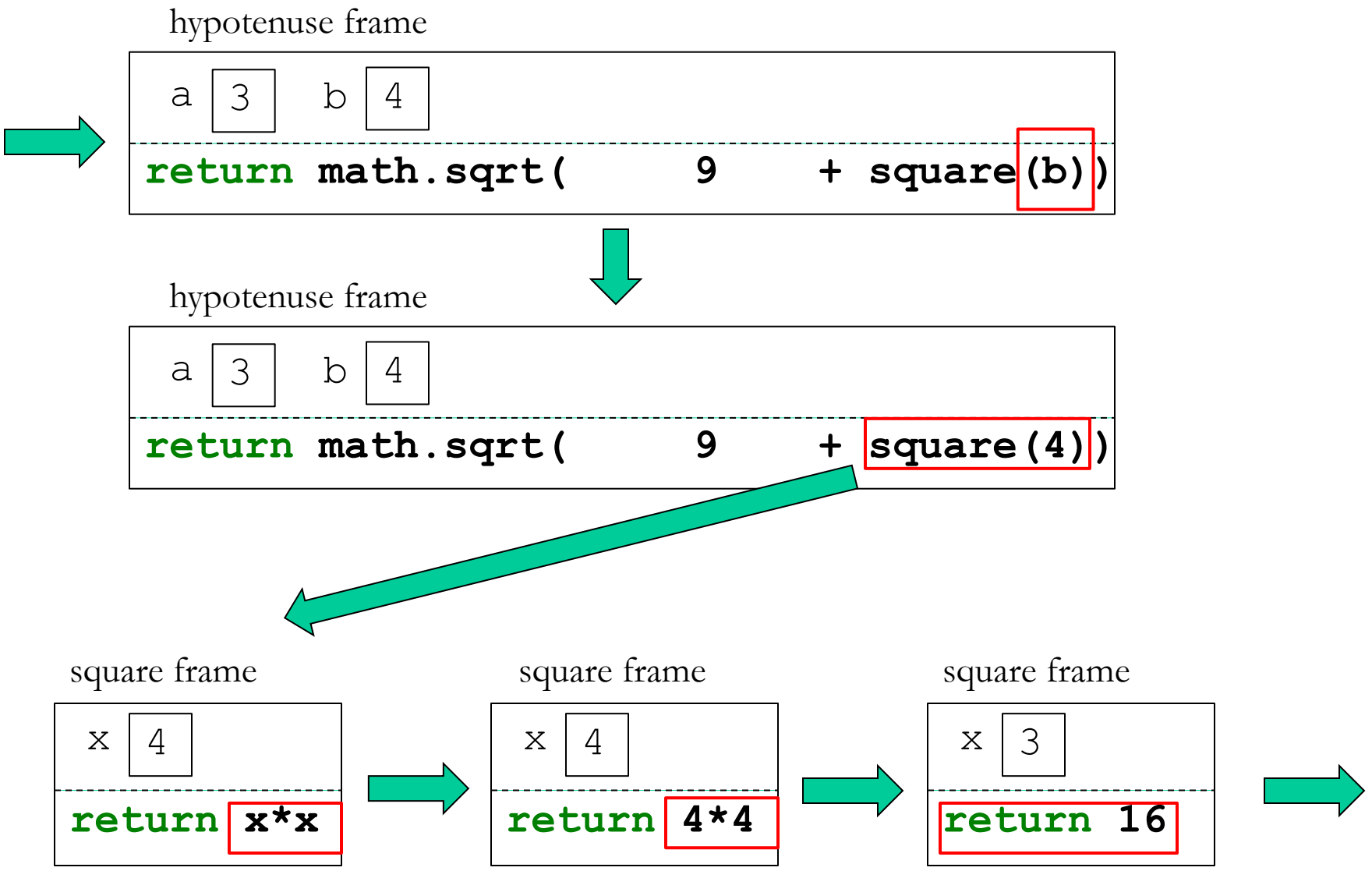
```
import math

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

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

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

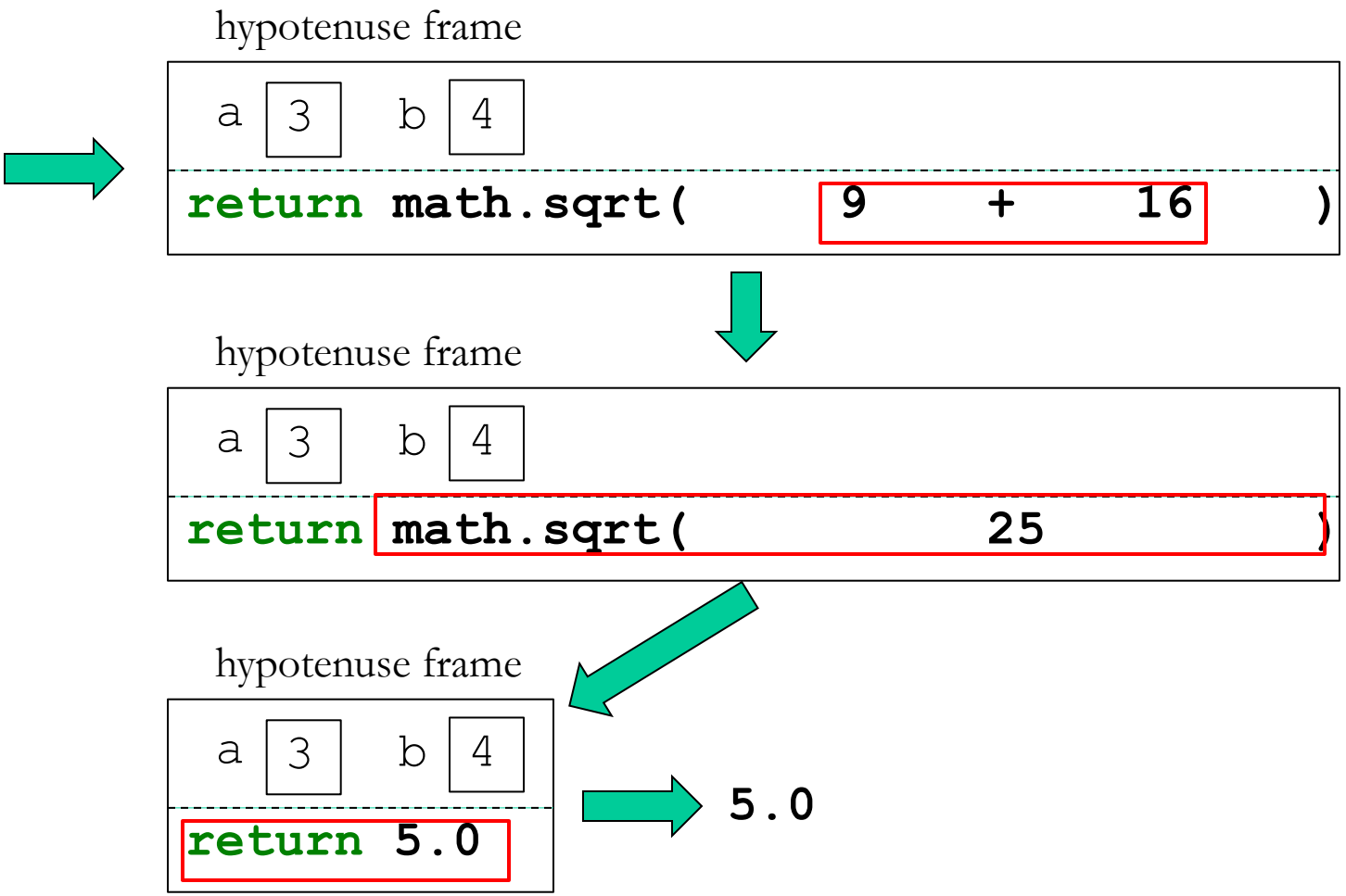
Function call model for `hypotenuse(3,4)` [1]



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?

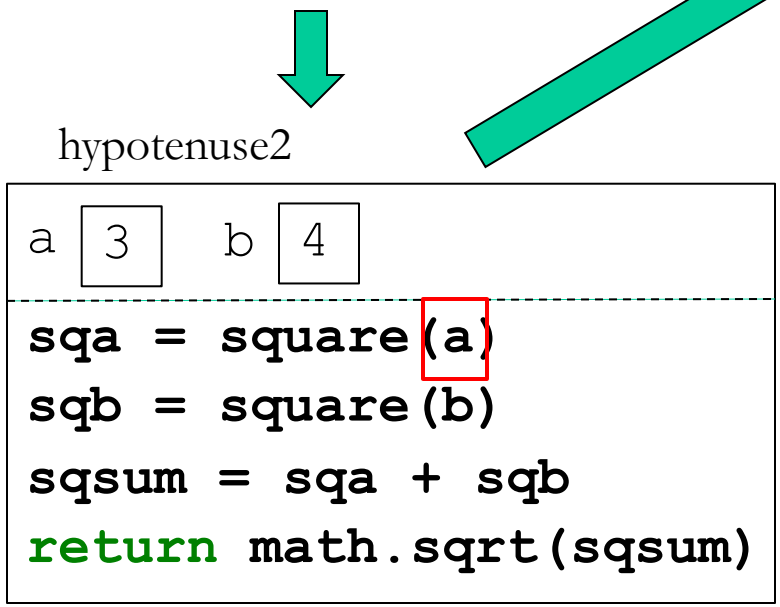
Consider the function below which calculates the length of the hypotenuse of a right triangle given the lengths of the two other sides.

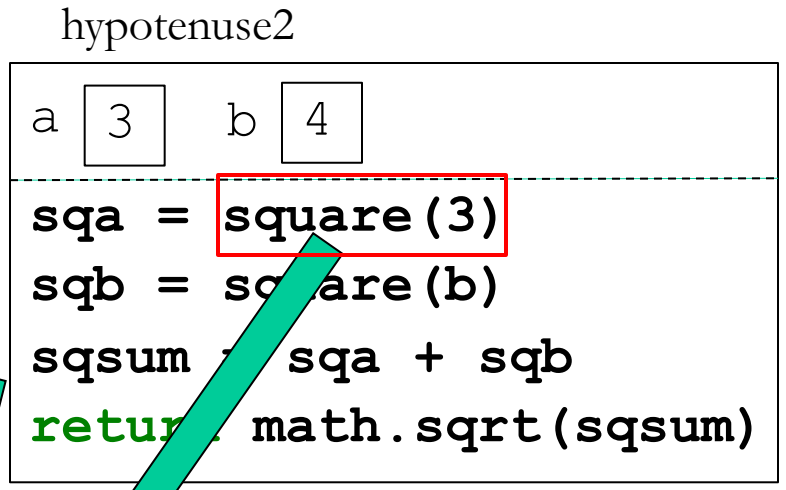
```
def hypotenuse2(a,b):
    sqa = square(a)
    sqb = square(b)
    sqsum = sqa + sqb
    return math.sqrt(sqsum)
```

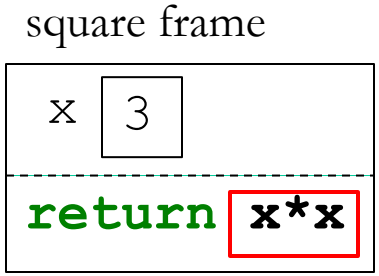
Functions w/local variables: hypotenuse2 [1]

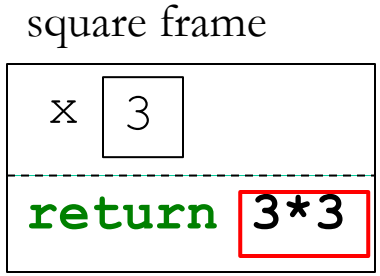
```
def hypotenuse2(a,b):
    sqa = square(a)
    sqb = square(b)
    sqsum = sqa + sqb
    return math.sqrt(sqsum)
```

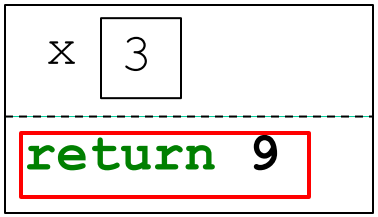
```
hypotenuse2(3,4)
```





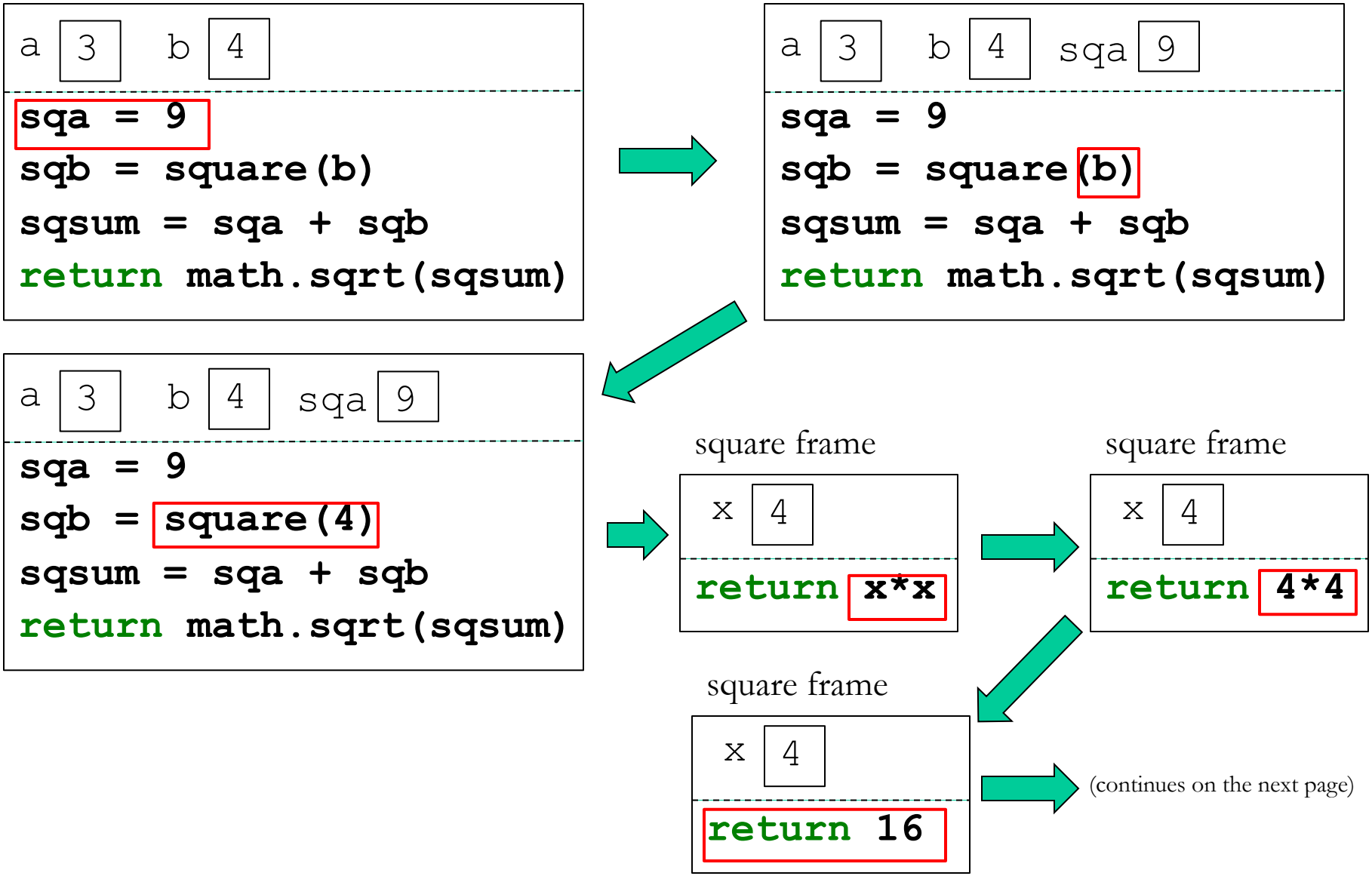




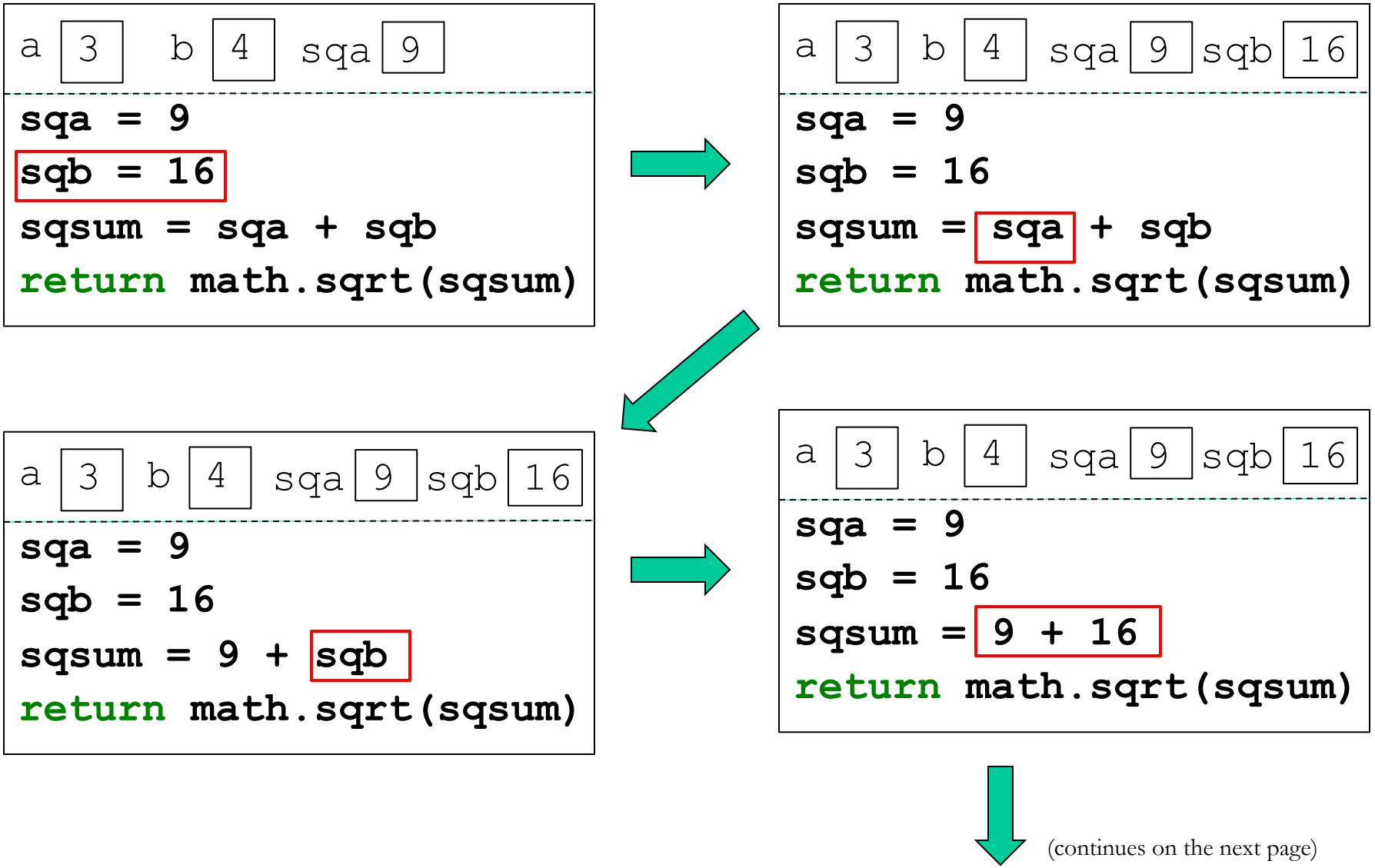


(continues on the next page)

Functions w/local variables: hypotenuse2 [2]

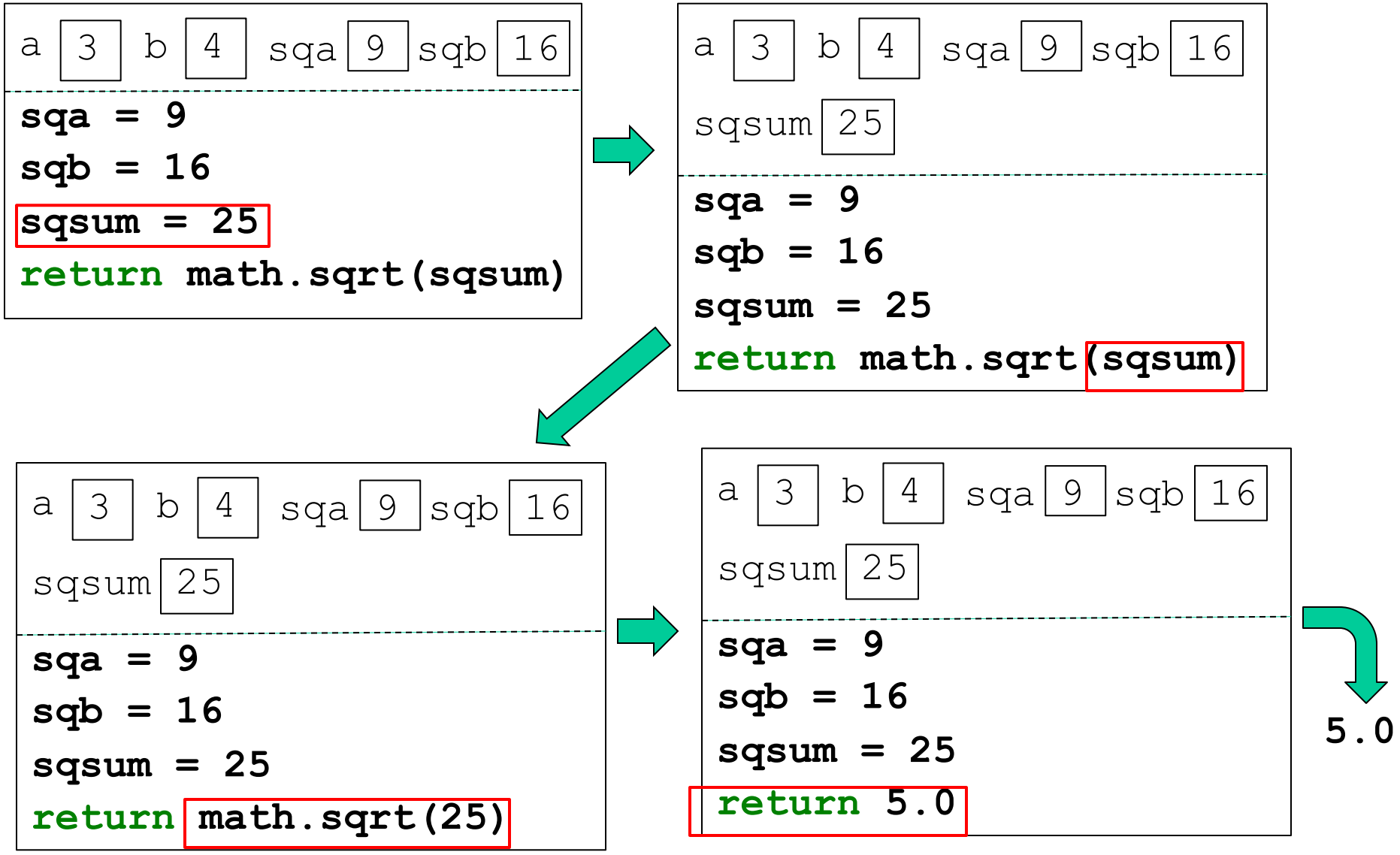


Functions w/local variables: hypotenuse2 [3]



(continues on the next page)

Functions w/local variables: hypotenuse2 [4]



RETURN VS. PRINT

Output of a function:

Concepts in this slide: return and print are different!

`return` vs. `print`:

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

```
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
```

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

Concepts in this slide: return and print are different!

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

```
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'
```

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

Concepts in this slide:
Two classes of functions

Fruitful vs. None Functions

We will 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, methods that don't return a value are void methods.
So we may sometimes use "void functions" as a synonym for "None functions"

Incremental Development

Concepts in this slide:
Incremental Development

When writing your own functions or any other type of code, do not attempt to write it all at once! Try to complete small tasks one at a time and test along the way.

Example: create a function called `numStats` that prints out the number, the number squared, and the remainder.

Step 1: create the header and print the num.

```
def numStats(num):
    print(num)
```

Step 2: properly print the first line.

```
def numStats(num):
    print("Your number is " + str(num) + "."))
```

Step 3: add the second print.

```
def numStats(num):
    print("Your number is " + str(num) + ".")
    print("Your number squared is " + str(num**2) + ".")
```

Step 4: properly print the final line.

```
def numStats(num):
    print("Your number is " + str(num) + ".")
    print("Your number squared is " + str(num**2) + ".")
    print("Your number has a remainder of " +
          str(num % 2) + " when divided by 2.")
```

FUNCTIONS AND TURTLES

Turtle Graphics

Concepts in this slide: A list of all useful functions from the turtle module.

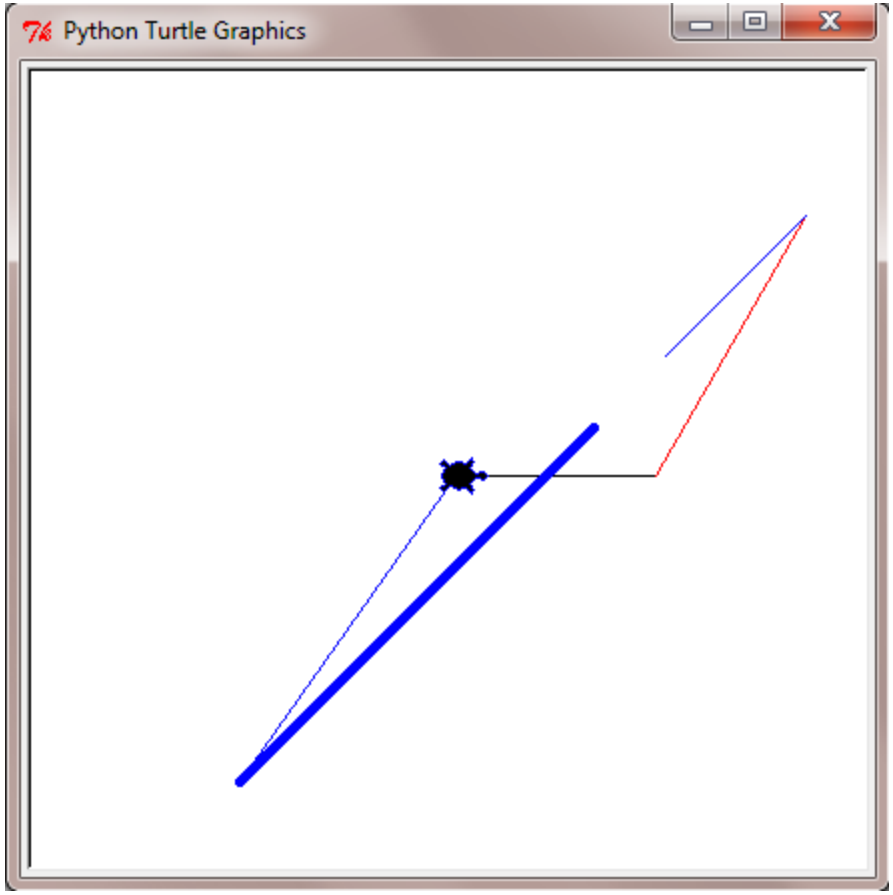
Python has a built-in module named **`turtle`**. See the Python turtle module API for details.

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*

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()
```

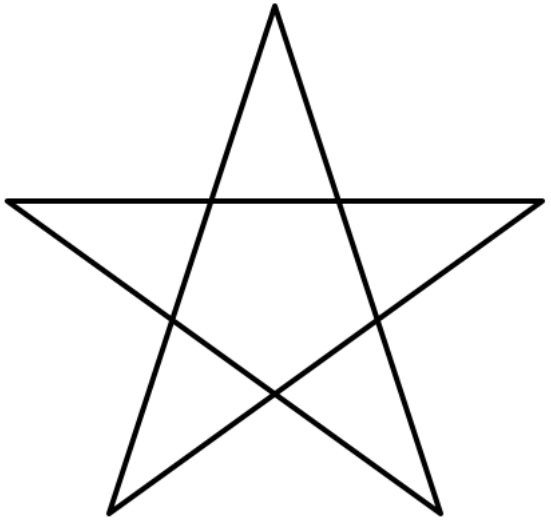
Tk window

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 can help make code for turtle graphics more concise and simpler.

```
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.

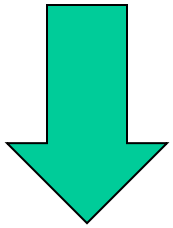
The body of the function captures the similarities of all stars while the parameters express the differences.

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

Fish Tank

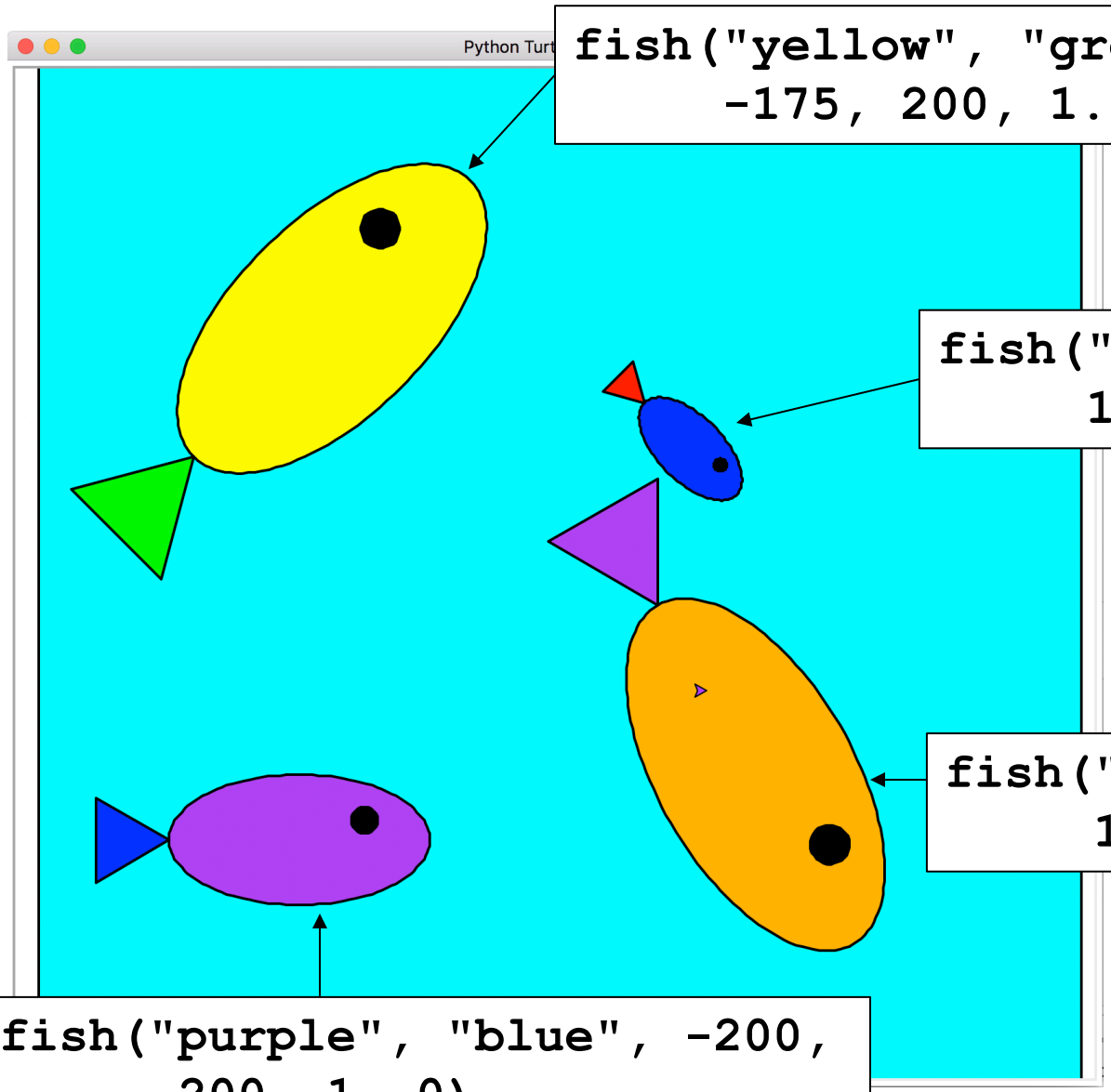
```
def staticFish():
    # Make the body
    fillcolor("yellow")
    begin_fill()
    drawEllipse(50, 2)
    end_fill()
    # Make the eye
    penup()
    fd(50)
    lt(90)
    fd(15)
    rt(90)
    pendown()
    fillcolor("black")
    begin_fill()
    drawCircle(10)
    end_fill()
    # SOME CODE OMITTED.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

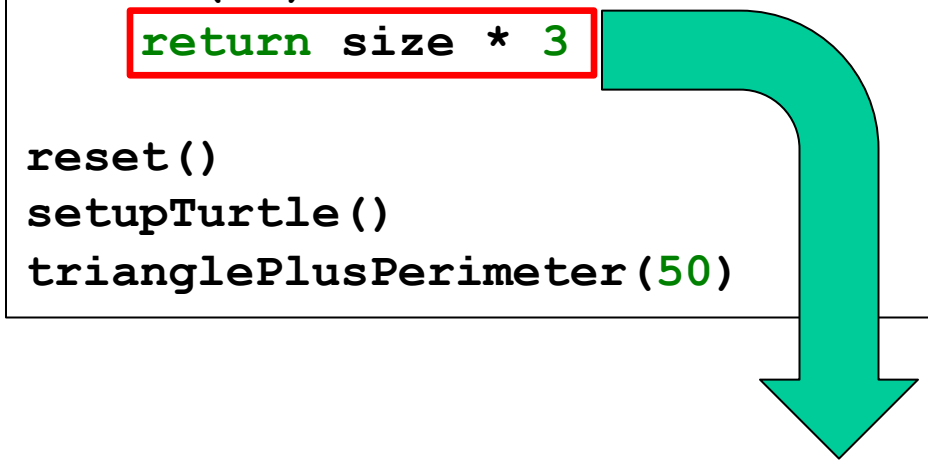
```
fish("purple", "blue", -200,
     -200, 1, 0)
```

Fruitful Turtles

We say a function is fruitful if it returns a value. See slide 28 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)
```

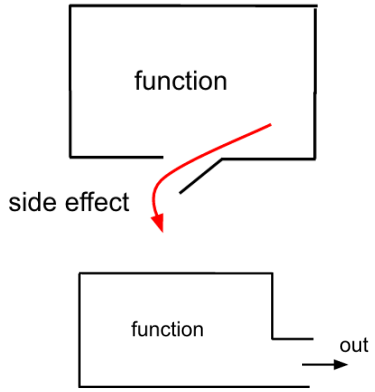


Return the perimeter of the triangle after it has been drawn.

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:

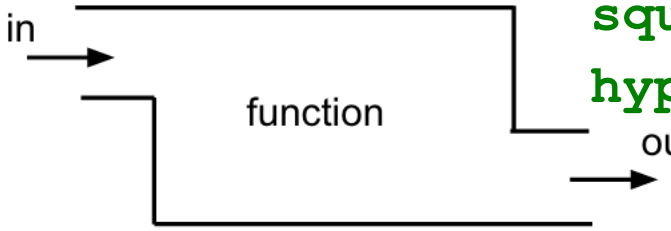
```
import random
random.random()
```

Out [...]

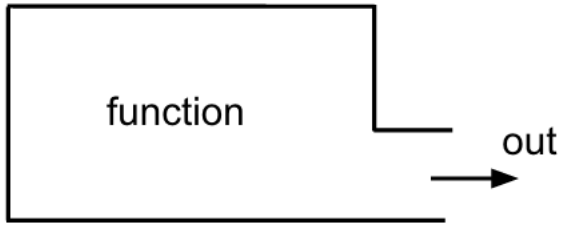
```
0.72960321
```

A random float value between 0 and 1.

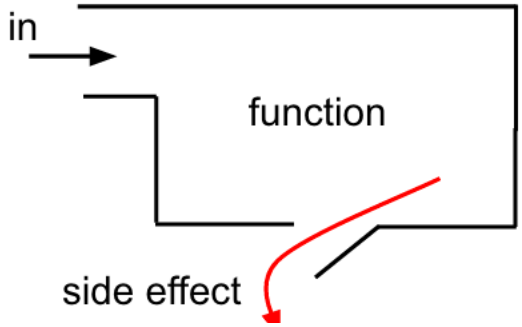
Updated Function diagrams



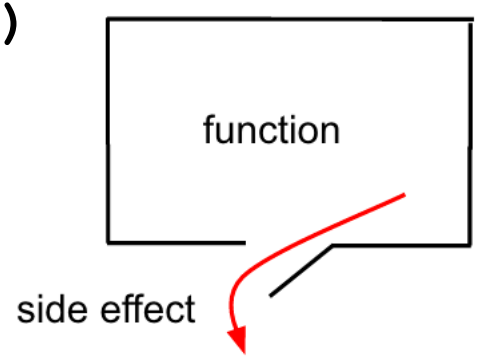
`max()`, `min()`
`square()`
`hypotenuse()`



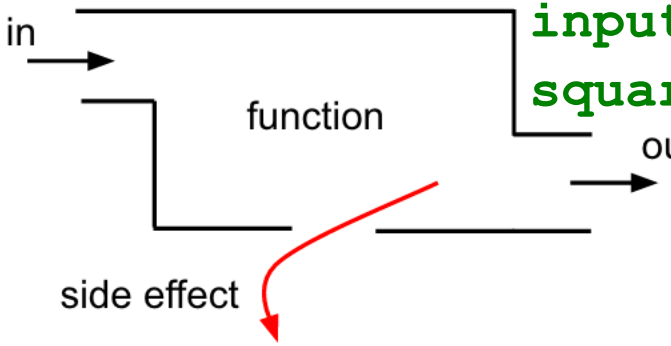
`random.random()`



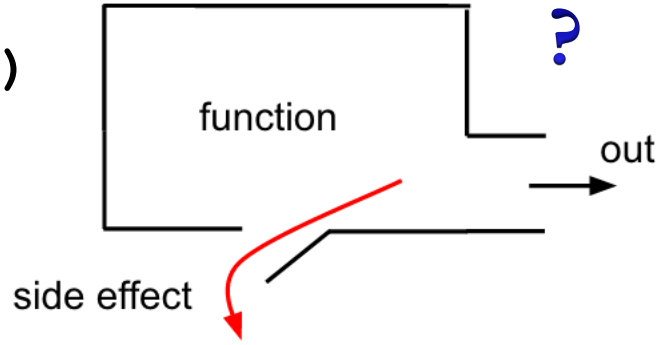
`print()`, `help()`
`verse()`
`printBanner()`
`printSquare()`



`rocks()`
`rocks3()`

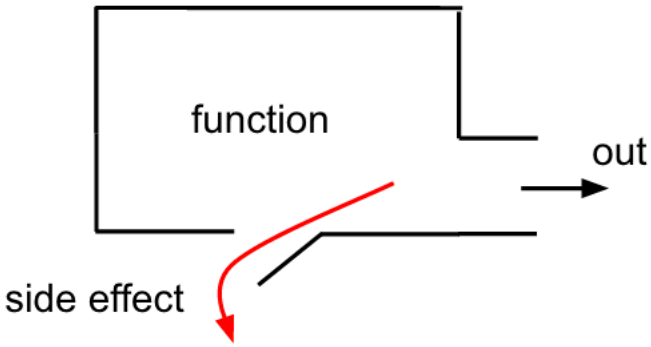


`input()`,
`squarePrintArg()`



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.

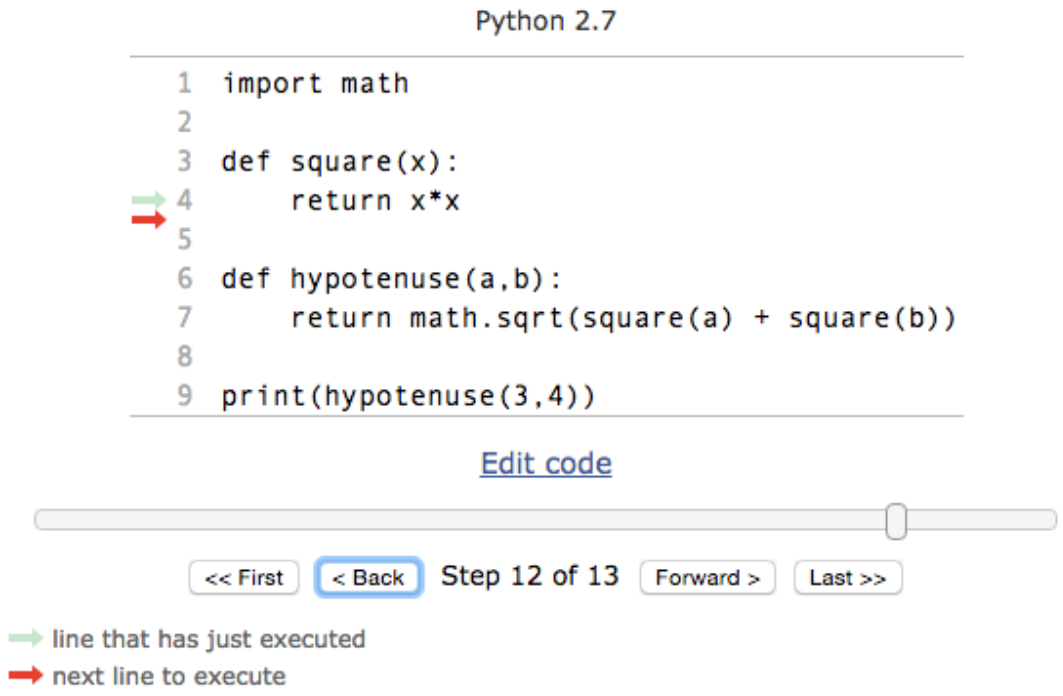


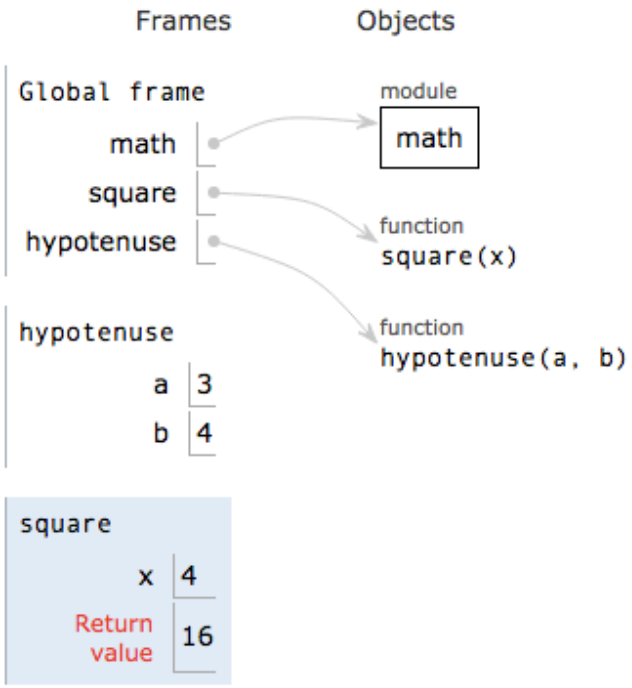
As an exercise, can you write a function that takes no input and produces a side-effect while returning a value? Hint: printing is always a good way to produce a side-effect! Try and write a meaningful function that would fulfill these two criteria.

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. **You'll use it in Lab.**





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?