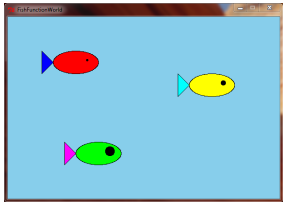
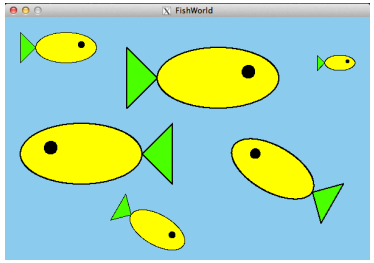
Abstracting with Functions



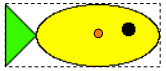
CS111 Computer Programming

Department of Computer Science
Wellesley College

Review: Abstracting with Layers

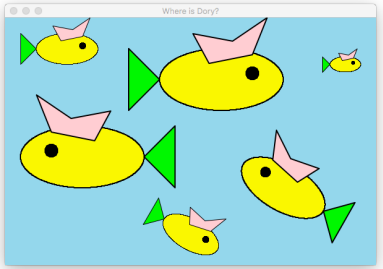


We've seen that layers are a means of abstraction. We can populate a fishtank by cloning and transforming a single prototype fish pattern expressed as a layer:



Then if we want every fish to have a hat, we just modify our one prototype fish before we clone it.

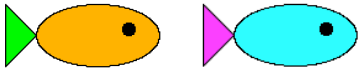
```
# Add pink hat *before* any clones are made
hat = Polygon(Point(-23,-37),Point(9,-31),
              Point(37,-50),Point(25,-20),
              Point(-10,-13))
hat.setFillColor('pink')
fish.add(hat)
```



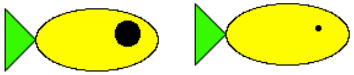
Review: Drawbacks of Layers

Although Layers are powerful, they do not let us abstract over all the properties of our fish that we might want to change.

What if we want different fish to have different body or tail colors?



What if we want different fish to have larger or smaller eyes?

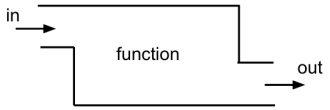


We cannot express these differences with Layers. Why not?

But we *can* express them with user-defined functions, a more powerful abstraction mechanism that we will study in this lecture.

Functions take inputs and return outputs based on those inputs

Concepts in this slide: functions, input & output

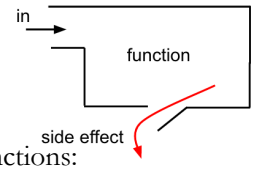


Here are examples of **built-in** functions you have seen:

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



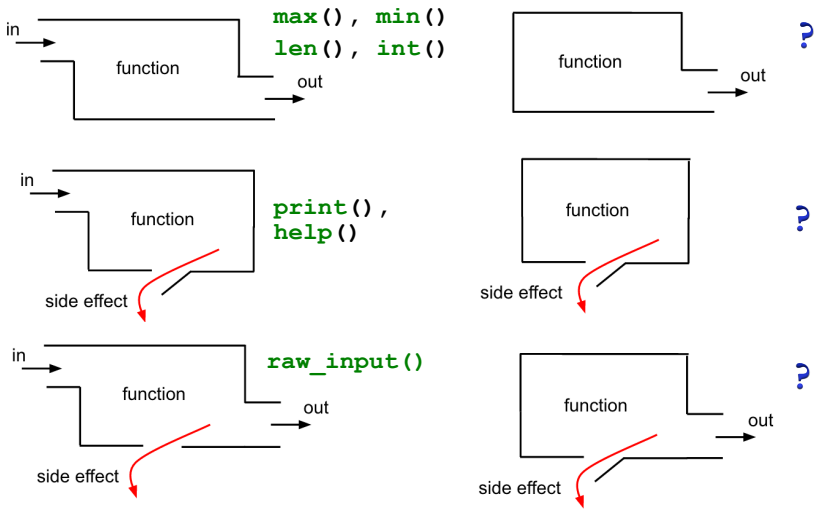
These actions are called **side effects**.

For example, displaying text in the interactive console (Canopy's Python pane) is a side effect of the **`print`** and **`help`** functions:

```
In [1]: print("The max value is: " + str(max(23,78)))
The max value is: 78

In [2]: help(max)
Help on built-in function max in module __builtin__:

max(...)
    max(iterable[, key=func]) -> value
    max(a, b, c, ...[, key=func]) -> value
```


Function diagrams summarize what functions do

Concepts in this slide: function diagrams

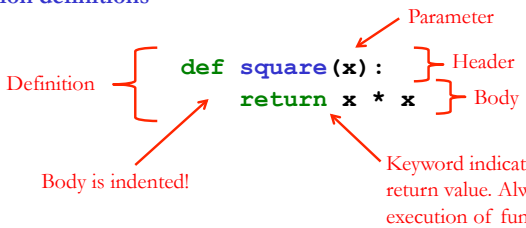


Anatomy of a User-defined Function

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

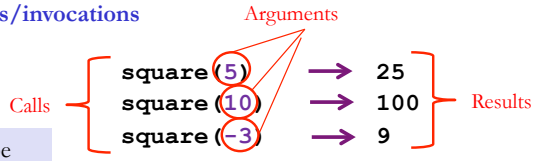
Functions are a way of abstracting over computational processes by capturing common patterns.

Function definitions



A function is defined **once**.

Function calls/invocations



A function can be called many times.

Parameters

Concepts in this slide: Difference between parameters and arguments.

A parameter **names** "holes" in the body that will be filled in with the **argument value** for each invocation.

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

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

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

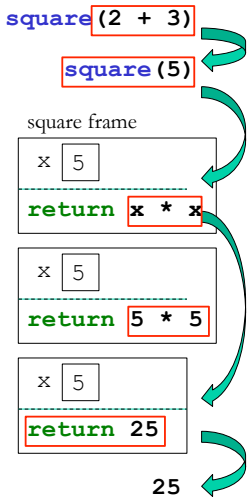
```
def square(num):
    return num * num
```

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

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

Concepts in this slide: Defining multiple parameters.

Multiple parameters

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
```

```
def pyramidVolume(l, w, h):
    """Calculate volume rectangular pyramid"""
    return (l * w * h)/3.0
```

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


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

Concepts in this slide: return and print are different!

- `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`!

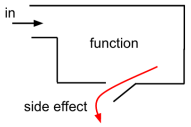
Concepts in this slide: return and print are different!

```
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!

Examples: Function with side-effect and no return value

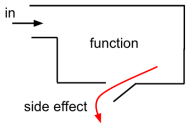



```
def printBanner(s):
    # 5 stars, 3 spaces, input string, 3 spaces, 5 stars
    banner_length = 5 + 3 + len(s) + 3 + 5
    print('*' * banner_length)
    print('*****' + '   ' + s + '   ' + '*****')
    print('*' * banner_length)
```

```
printBanner('CS111')

**********************
*****   CS111   *****
**********************
```

```
printBanner('Pied Piper')

**************************
*****   Pied Piper   *****
**************************
```


Example: Seconds to Days




```
def printTimeFromSeconds(s): # Total seconds
    seconds = s % 60 # Remaining seconds
    m = s / 60       # Total minutes
    minutes = m % 60 # Remaining minutes
    h = m / 60       # Total hours
    hours = h % 24   # Remaining hours
    days = h / 24    # Total days
    print(str(s) + ' seconds is equivalent to:')
    print(str(days) + ' days')
    print(str(hours) + ' hours')
    print(str(minutes) + ' minutes')
    print(str(seconds) + ' seconds')
```

```
In [1]: printTimeFromSeconds(1000000)
1000000 seconds is equivalent to:
11 days
13 hours
46 minutes
40 seconds
```


Calling other functions

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

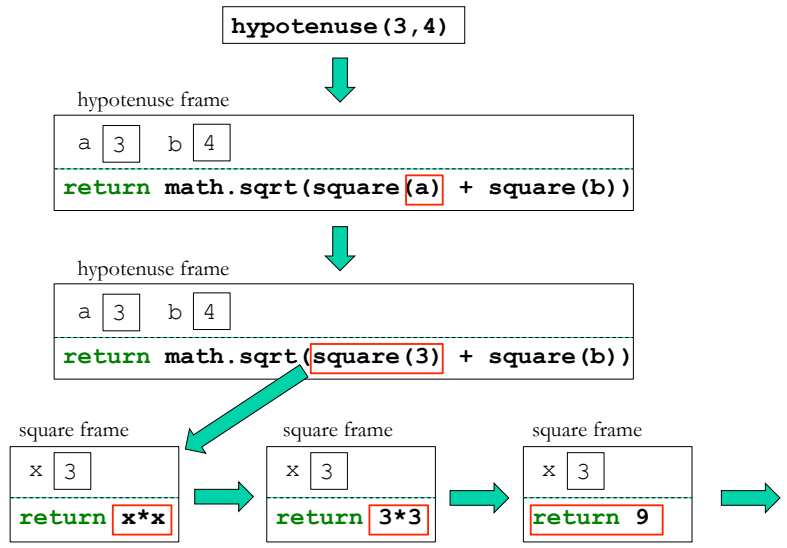
Functions call other functions:

```
import math

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

hypotenuse(3, 4) --> 5.0
hypotenuse(1, 1) --> 1.4142135623730951
```

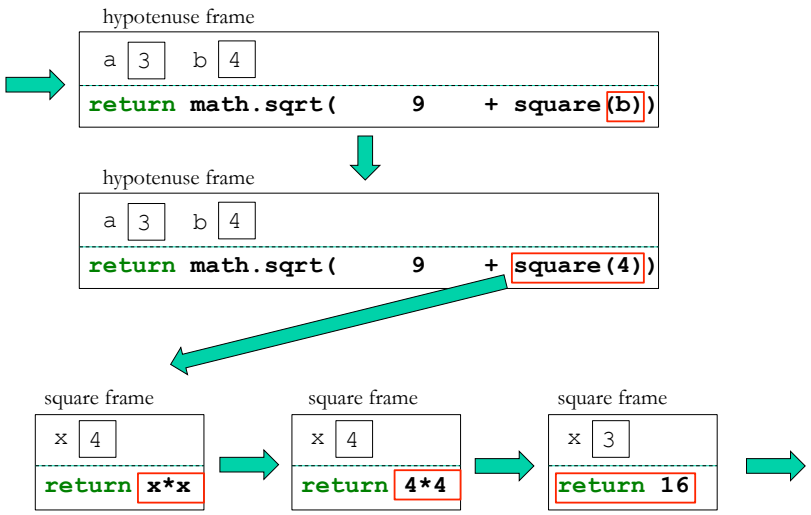
```
def distanceBetweenPoints(x1, y1, x2, y2):
    """Calculate the distance between points """
    return math.sqrt((x2-x1)**2 + (y2-y1)**2)
```

Note the use of Python's **math** module in both functions.
** is the power operator in Python.

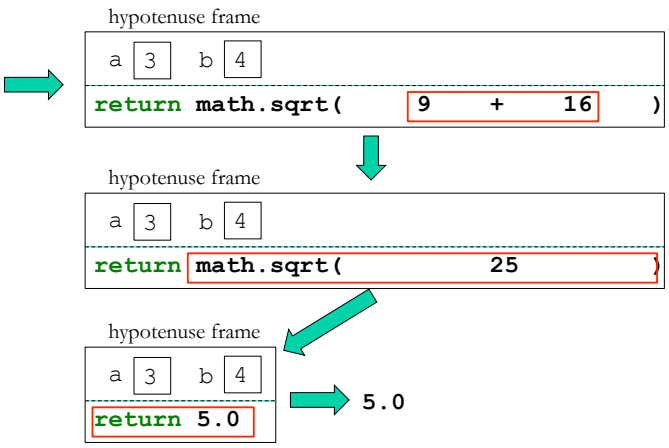
Function call model for `hypotenuse(3,4)` [1]



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

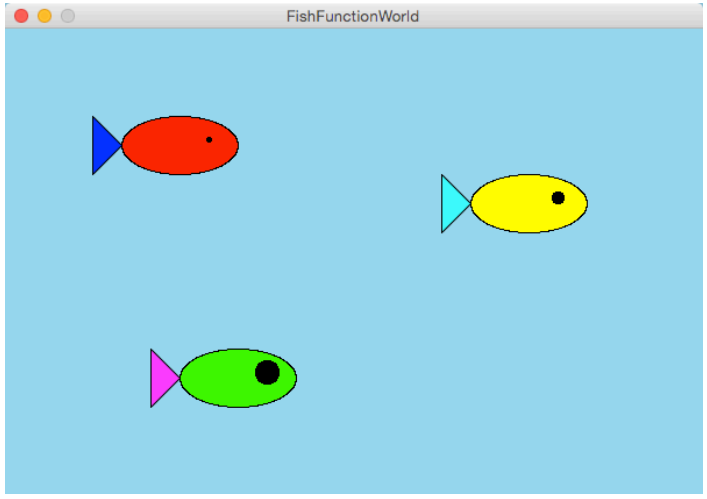


Function call model for `hypotenuse(3,4)` [3]



Function Abstraction: Fishtank Revisited

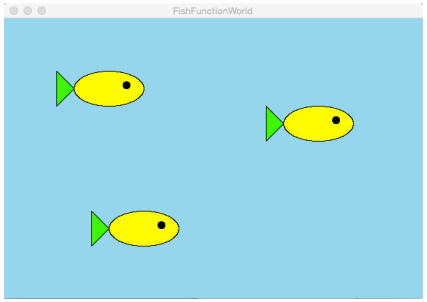
We cannot make these fish by cloning a fish layer. Why?



fish_with_functions.py

This makes a new fish Layer via a function call rather than a clone. With parameters (see next few slides), functions are more powerful than clones.

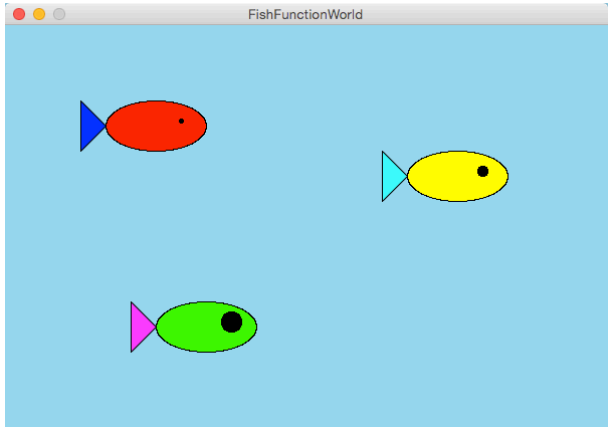
```
def makeFish():
    fish = Layer() # fish layer

    # body of the fish
    body = Ellipse(100,50,Point(0,0))
    body.setFillColor('yellow')
    fish.add(body)

    # green tail of the fish
    tail = Polygon()
    tail.addPoint(Point(-50,0))
    tail.addPoint(Point(-75,25))
    tail.addPoint(Point(-75,-25))
    tail.setFillColor('green')
    fish.add(tail)

    # black eye of the fish
    eye = Circle(5,Point(25,-5))
    eye.setFillColor('black')
    fish.add(eye)

    return fish
```



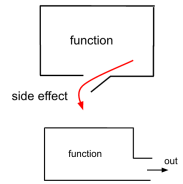

```
# Create a fishtank, and add three fish
tank = Canvas(600,400,'skyBlue',
              'FishFunctionWorld')

fish1 = makeFish()
tank.add(fish1)
fish1.moveTo(150,100)

fish2 = makeFish()
tank.add(fish2)
fish2.moveTo(450,150)

fish3 = makeFish()
tank.add(fish3)
fish3.moveTo(200,300)
```

makeFish with parameters

In lecture, you will modify the **makeFish** function definition and invocations to produce the fishtank picture shown below.



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!
```

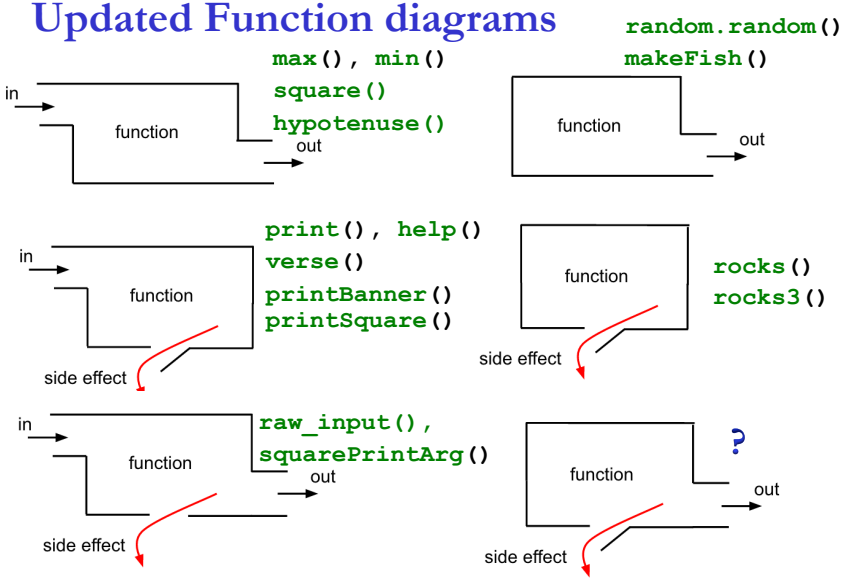
Python libraries have useful built-in functions with zero parameters and a return value:

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

```
Out [...]
0.72960321
```

A random float value between 0 and 1.

Updated Function diagrams



Unindented function body

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.:

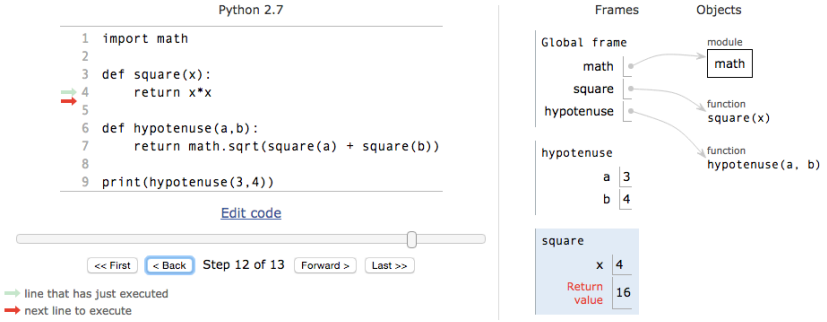
```
IndentationError: expected an indented block
```

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

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.**



The None value and NoneType

Digging Deeper

- Python has 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'
```

On slide 3-26, this is the real reason that the expression `print_square(3) + print_square(4)` causes an error.

Fruitful vs. None Functions

Digging Deeper

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

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

Fruitful functions	None functions
square	**test_square**
square_print	**printBanner**
hypotenuse	**verse**
hypotenuse2	**printTimeFromSeconds**

* 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"

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. Suppose the parameters of the `hypotenuse` function in 4-15 are renamed from `a` and `b` to `side1` and `side2`. Does the function still work as expected? Does any other part of the program "know" that the parameter names have been changed?
5. Can a function have a return value and no side effects? Side effects and no return value? Both side effects and a return value?
6. Can a function whose definition lacks a **return** statement be called within an expression?
7. What would happen if we swap the order or **print** and **return** in the definition of **squarePrintArg** in slide 4-11. Why? If you cannot imagine it, test it out in Canopy.
8. What is the value of using the function call model?
9. What is indentation and where it is used within Python?