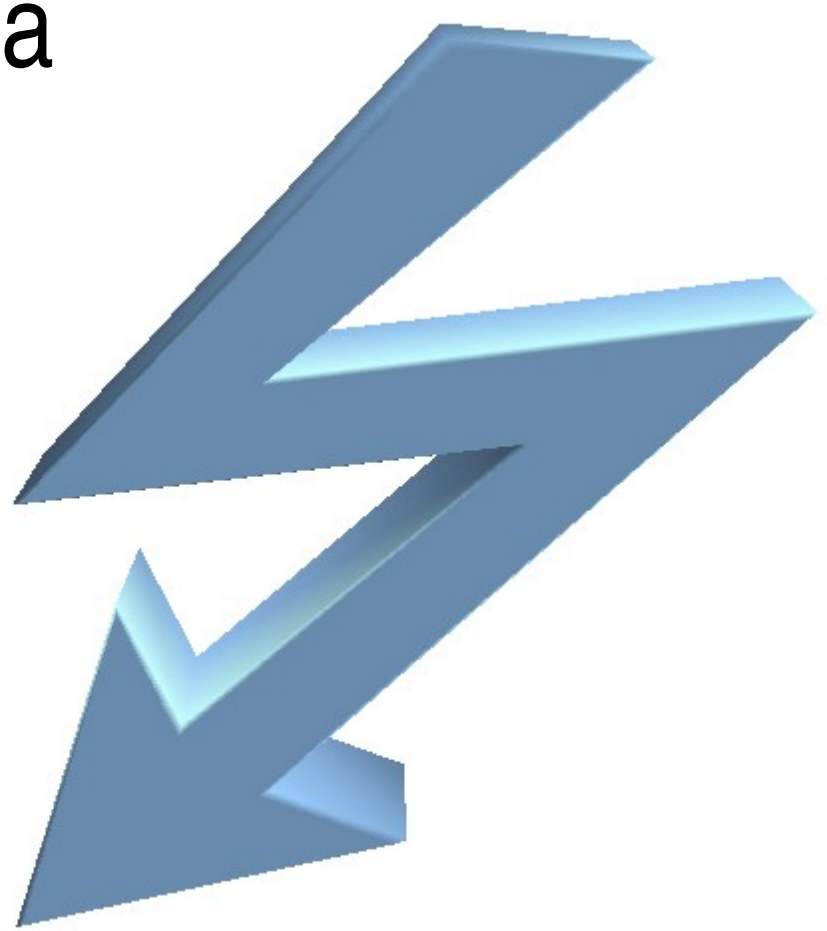
EECS 12: Lecture 5
Advanced Information about Functions

Mark E. Phair
mphair@gmail.com
UC Irvine EECS

July 17th, 2006

Agenda

- Extra Credit Project
- Midterm demo
- Function objects
- Functional programming
- How function calls work
- Recursion
- Dictionaries and hints
- Python factoid of the day



A moose once bit my sister. Well, actually it wasn't a moose, it was a goose. And it wasn't my sister, because I don't have a sister; it was me.

Extra Credit Project

- 50 (Midterm) Bonus Points
- Will be difficult
- Will require 5 minute presentation to class
- Still interested?

Midterm Demo

Big `if` statement solution: Function Objects

- In python, functions can be objects just like anything else
  - Pass to other functions
  - Store in variables
  - Think of function names as variables that hold functions
- `apply(function, arguments)`
- Let's use this to clean up the midterm even more
  - store an add function in a dictionary under key '+'
  - use apply to call the function with arguments to add

Functional Programming: `map`

```
newLst = map(function, lst)


# What map does
newLst = []
for item in lst:
 newLst.append(apply(function,item))
```

Functional Programming: `map`

```
>>> def times2(a):
        return a*2
>>> map(times2, [1,7,2])
[2, 14, 4]
```

Functional Programming: `reduce`

```
result = reduce(function, lst)


# what reduce does
result = apply(function, lst[0:2])
for item in lst[2:]:
 result = apply(function,\
                [result, item])
```

Functional Programming: `reduce`

```
>>> def add(a,b):
        return a+b

>>> reduce(add, [1,2,3,4])
10
```

Functional Programming: `filter`

```
newLst = filter(function, lst)


# what filter does
newLst = []
for item in lst:
    if apply(function, [item]):
        newList.append(item)
```

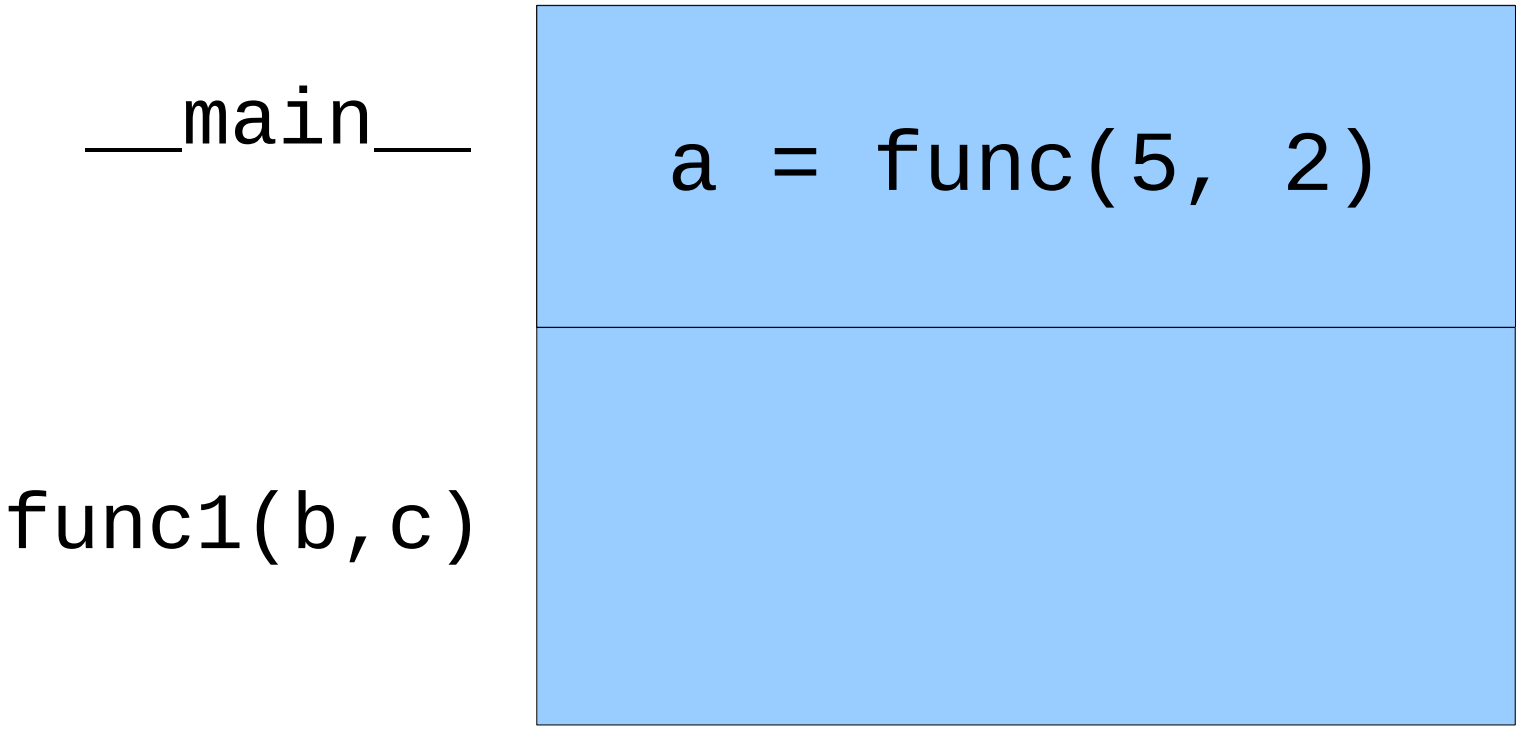
Functional Programming: `filter`

```
>>> def even(n):
        return n%2 == 0
>>> filter(even, range(5))
[0, 2, 4]
```

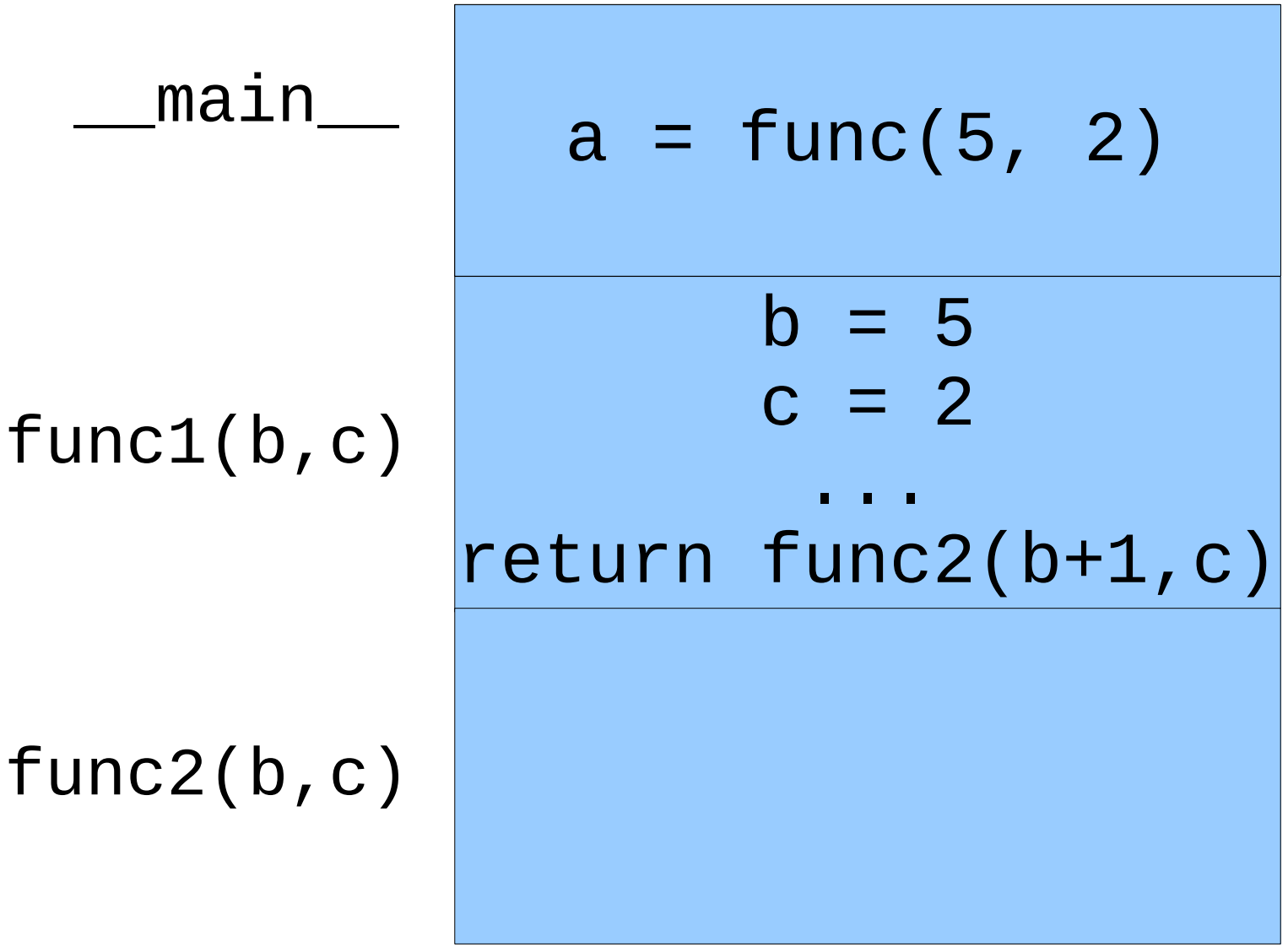
Let's explore: summing squares of odd numbers

- Using reduce, map, and filter, create a function that sums the squares of the odd numbers from 0 to n
- functions needed:
  - sumEvenSquares(n)
  - add(x,y)
  - square(x)
  - isEven(x)

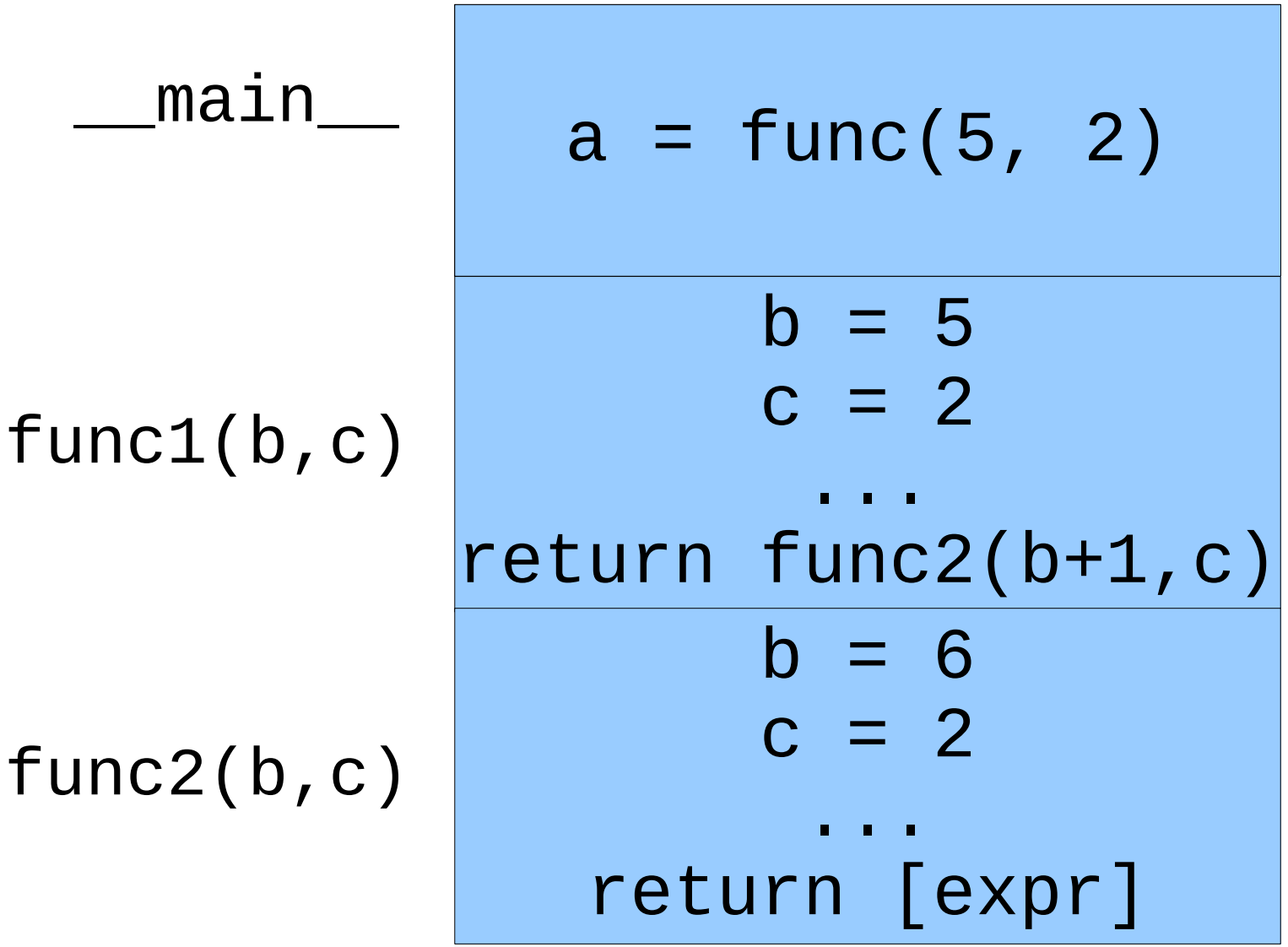
How Function Calls Work



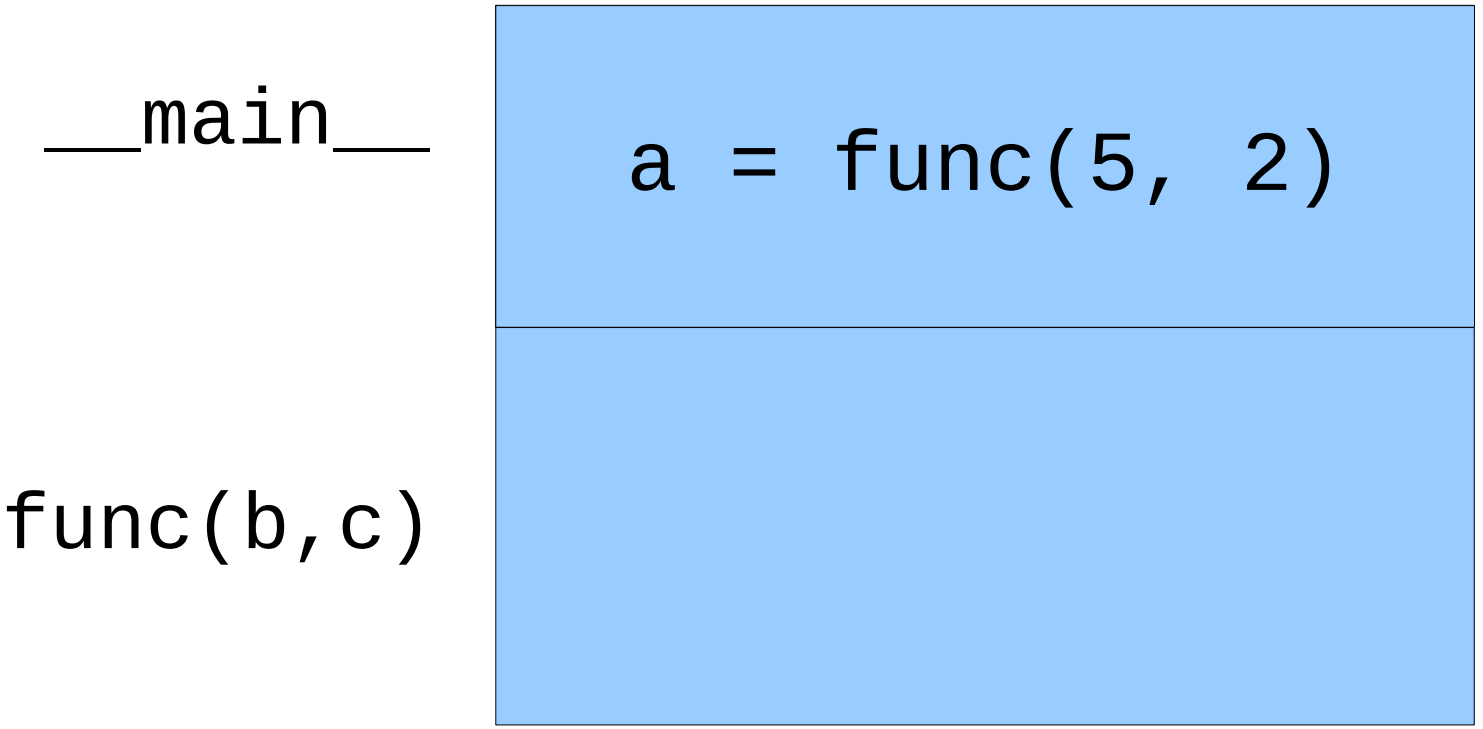
How Function Calls Work



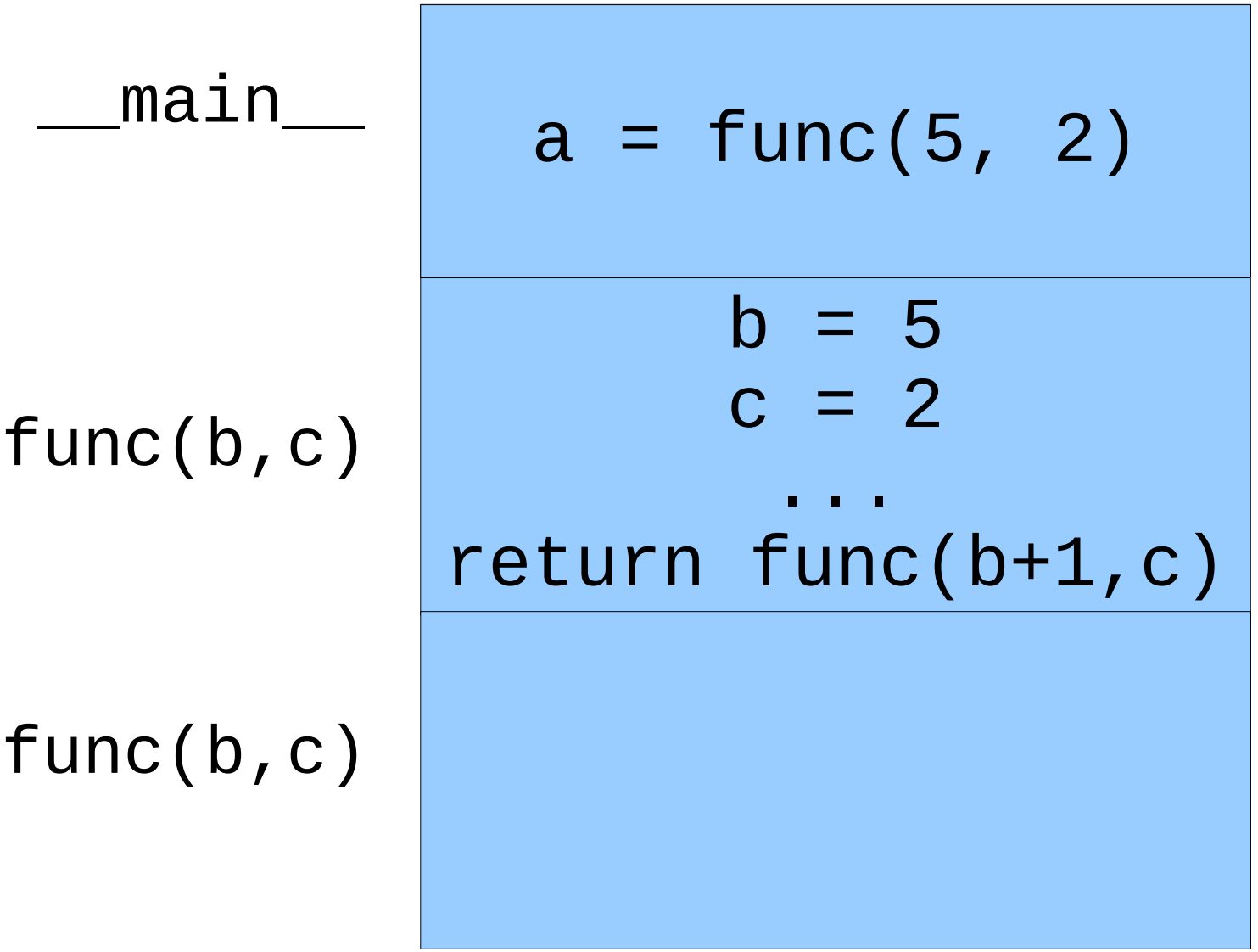
How Function Calls Work



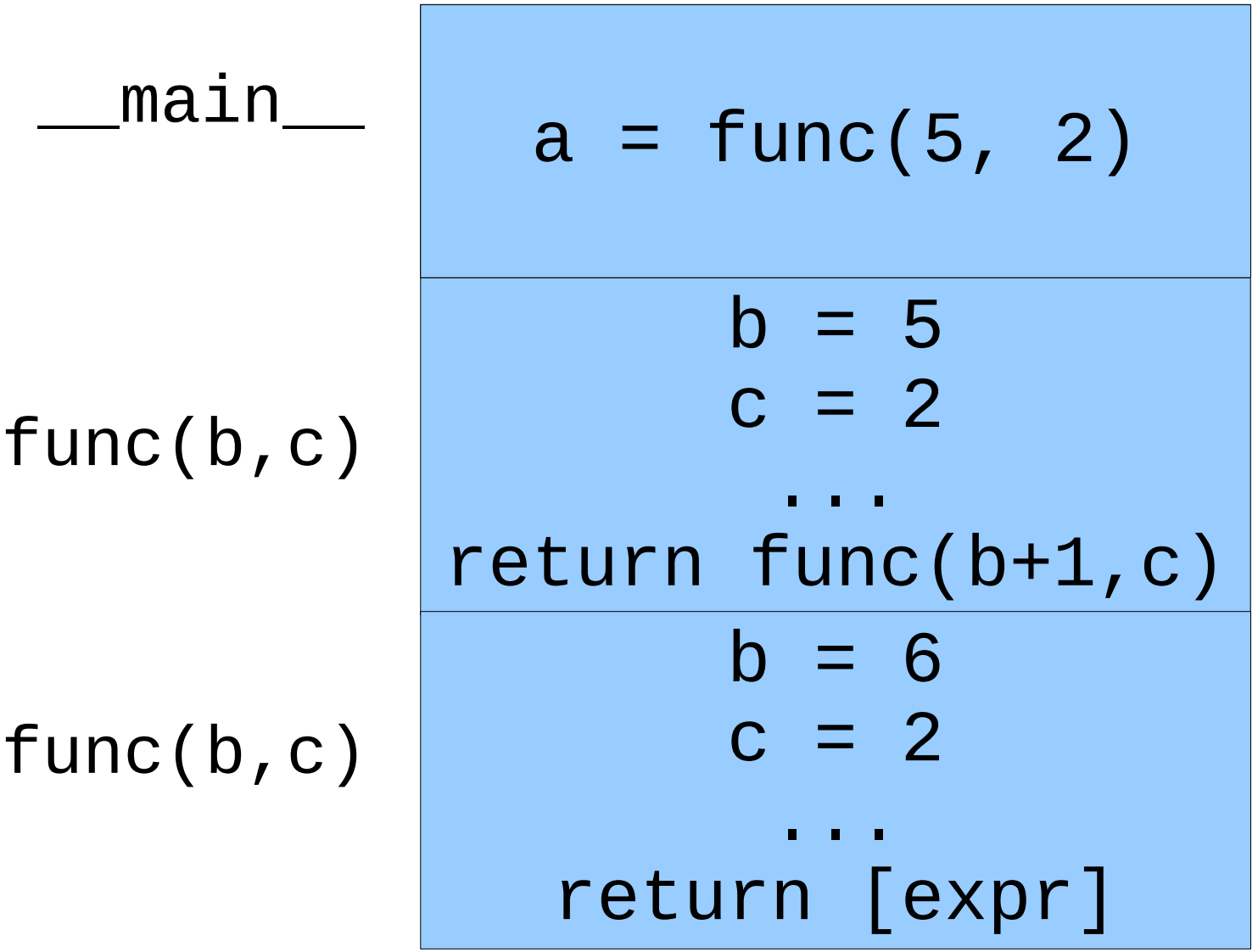
How Function Calls Work: Recursion



How Function Calls Work: Recursion



How Function Calls Work: Recursion



Forget the “Leap of Faith”

- The book describes a “leap of faith” about recursion which is not required
- Recursion can be understood and believed with induction
- Induction has two steps
  - Show a base case to be correct
  - Show that if it is true for some `i`, then it is true for `i+1`

Inductive Proof Example: Factorial

Claim: The factorial function below yields the correct result when given a positive integer as input.

```
def factorial(n):
    "Computes n!"
    if (n == 0 or n == 1): return 1
    else: return n * factorial(n-1)
```

Step 1: Base Case

A base case tells us how to start the induction. Equivalently, it tells us where the recursion ends.

```
def factorial(n):
    "Computes n!"
    if (n == 0 or n == 1): return 1
    else: return n * factorial(n-1)
```

Step 2: Induction Step

The induction step tells us how to progress from one stage to the next. In the case of induction, n is getting larger. In the case of recursion, n is getting smaller.

```
def factorial(n):
    "Computes n!"
    if (n == 0 or n == 1): return 1
    else: return n * factorial(n-1)
```

Step 2: Induction Step (cont)

Assuming that it works for i, show that it works for i+1

```
def factorial(n):
    ...
    else: return n * factorial(n-1)
```

Let `i = n-1`. Assume that `factorial(i)` yields `i!`, which means that `factorial(n-1)` is correct.

Now, show that it works for `i+1==n`:

`(i+1) * (i!) == (i+1)!` (by definition)

`n * factorial(n-1) == factorial(n)`

Caveat Recursor

- Beware of infinite recursion: caused by making mistakes in the base case
- Because of the ways function calls work, recursion can run into performance problems
  - "Function call overhead"
  - Tail recursion can help (recursive call is last thing in function)

Dictionaries and Hints

- For functions that do not depend on outside data structures (e.g., “Mathematical functions”), we can use *hints* to speed them up
- Calculated values can be stored in a dictionary

Fibonacci Example

```
def fib(n):
    """Computes Fibonacci sequence
  with bases fib(0)=0,fib(1)=1"""
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Fibonacci Example: With Hints

```
fibHint = {0:0, 1:1}
def fib(n):
    """Computes Fibonacci sequence
  with bases fib(0)=0,fib(1)=1"""
    if fibHint.has_key(n):
        return fibHint[n]
    else:
        return fib(n-1) + fib(n-2)
```

Python factoid of the day: `global`

The `global` keyword tells python to use an existing global variable instead of creating a new local one on assignment.

```
myD = {}
def doSomething():
    global myD
    myD = {'cow':'Moo!'}
```

If you are using it often, then you are abusing it. Global variables tend to be harmful!