61A LECTURE 6 – RECURSION

Steven Tang and Eric Tzeng

July 2, 2013

Announcements

- Homework 2 solutions are up!
- Homework 3 *and* 4 are out
  - Remember, hw3 due date pushed to Saturday
- Come to the potluck on Friday! It’ll be fun :)

Factorial

The factorial of a non-negative integer *n* is

$$n! = \begin{cases} 1, & n = 0 \text{ or } n = 1 \\ n * \underbrace{(n-1) * \cdots * 1,}_{(n-1)!} & n > 1 \end{cases}$$

Factorial

The factorial of a non-negative integer *n* is

$$n! = \begin{cases} 1, & n = 0 \text{ or } n = 1 \\ n * \boxed{(n-1)!}, & n > 1 \end{cases}$$

This is called a *recurrence relation*;

Factorial is defined in terms of itself

Can we write code to compute factorial using the same pattern?

Computing Factorial

We can compute factorial using the direct definition

$$n! = \begin{cases} 1, & n = 0 \text{ or } n = 1 \\ n * (n-1) * \cdots * 1, & n > 1 \end{cases}$$

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    total = 1
    while n >= 1:
        total, n = total * n, n - 1
    return total
```

Computing Factorial

Can we compute it using the recurrence relation?

$$n! = \begin{cases} 1, & n = 0 \text{ or } n = 1 \\ n * (n-1)!, & n > 1 \end{cases}$$

This is much shorter! But can a function call itself?

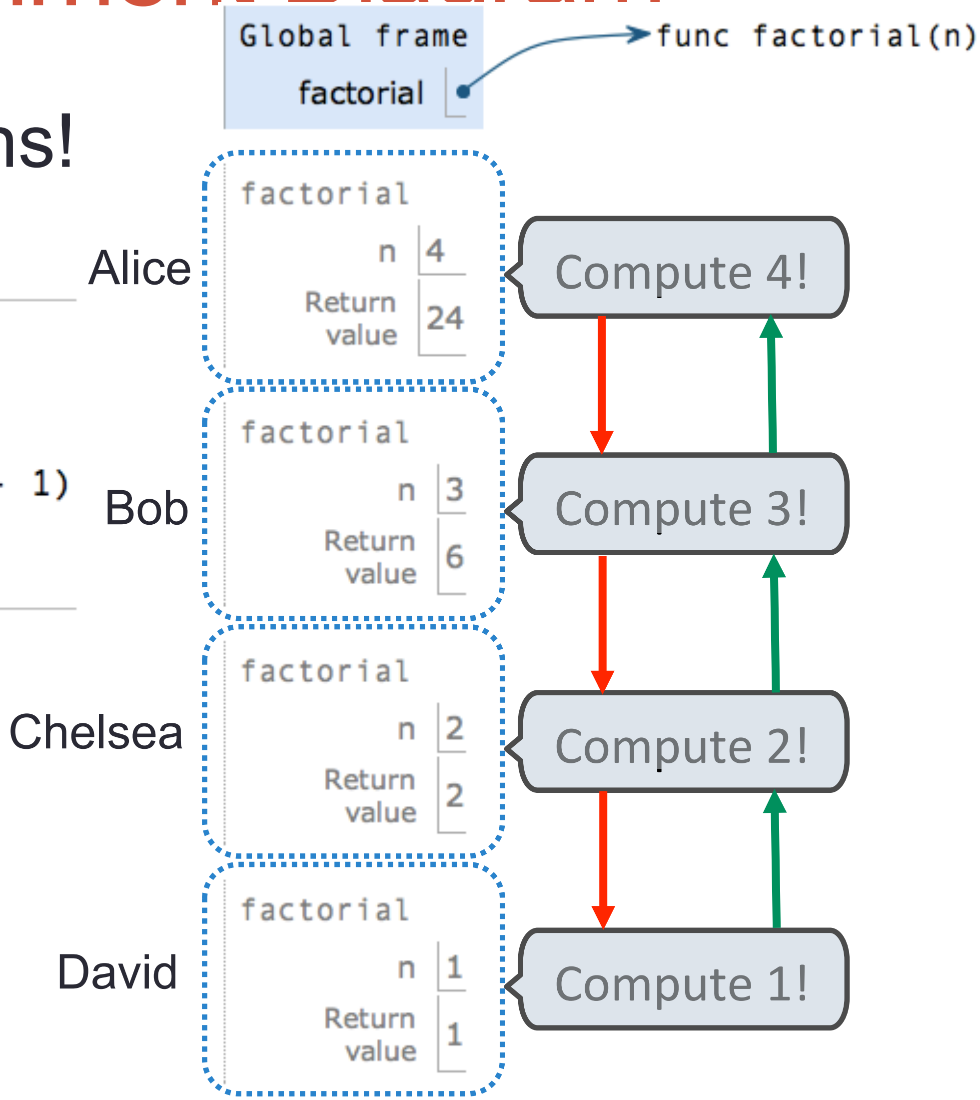
```python
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)
```

Factorial Environment Diagram

Let’s see what happens!

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)

factorial(4)
```



Recursive Functions

A function is *recursive* if the body calls the function itself, either directly or indirectly

Recursive functions have two important components:

1. *Base case(s)*, where the function directly computes an answer without calling itself
2. *Recursive case(s)*, where the function calls itself as part of the computation

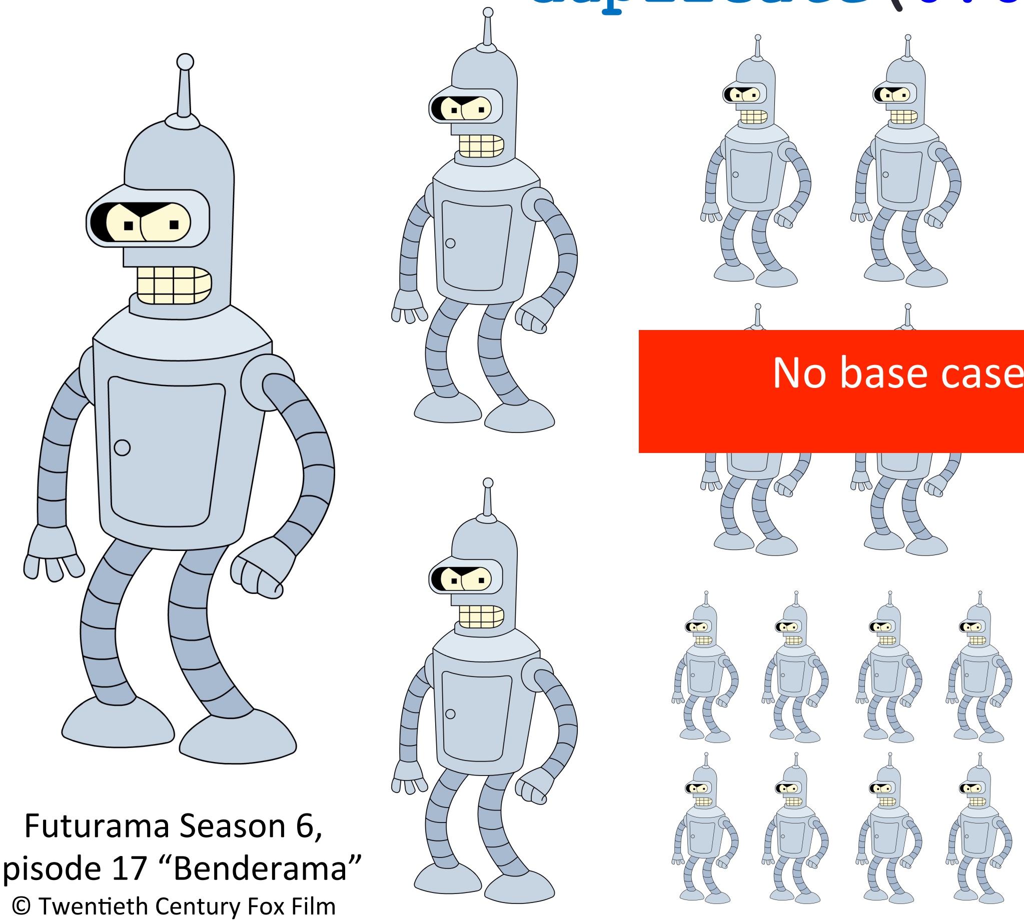
```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)
```

Base case

Recursive case

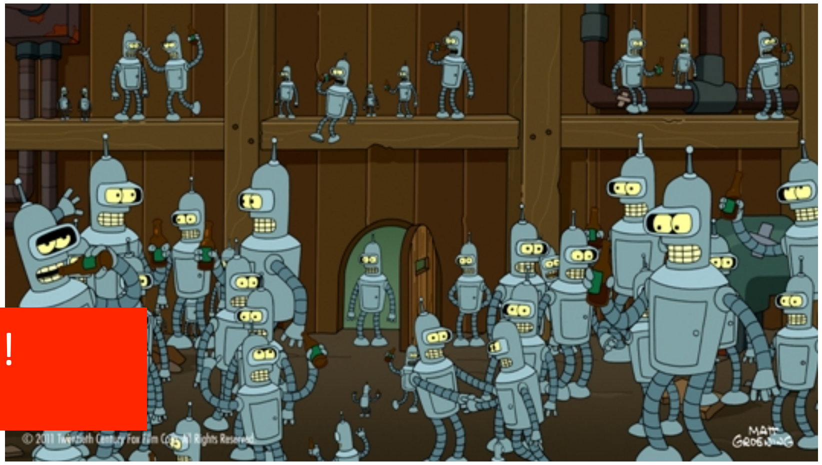
A warning: don’t forget your base case!

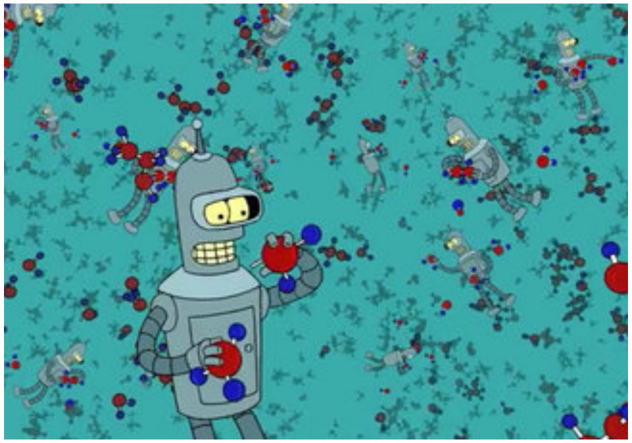
```
def duplicate(size):
    return (duplicate(0.6 * size) +
            duplicate(0.6 * size))
```



Futurama Season 6,
Episode 17 “Benderama”
© Twentieth Century Fox Film
Corporation

No base case!





Recursive leap of faith

- Most important slide in the lecture!
- Closest you'll get to a "program" for writing recursive functions
- Follow these steps:
  1. Figure out your base case – this is the simplest possible question someone could ask you about this function
  2. Now consider the general case, and treat a slightly simpler recursive call as a functional abstraction
     a. Leap of faith! Assume that the simpler call *just works*.
     b. Use the simpler call to make the general case work.

The leap of faith in action for factorial

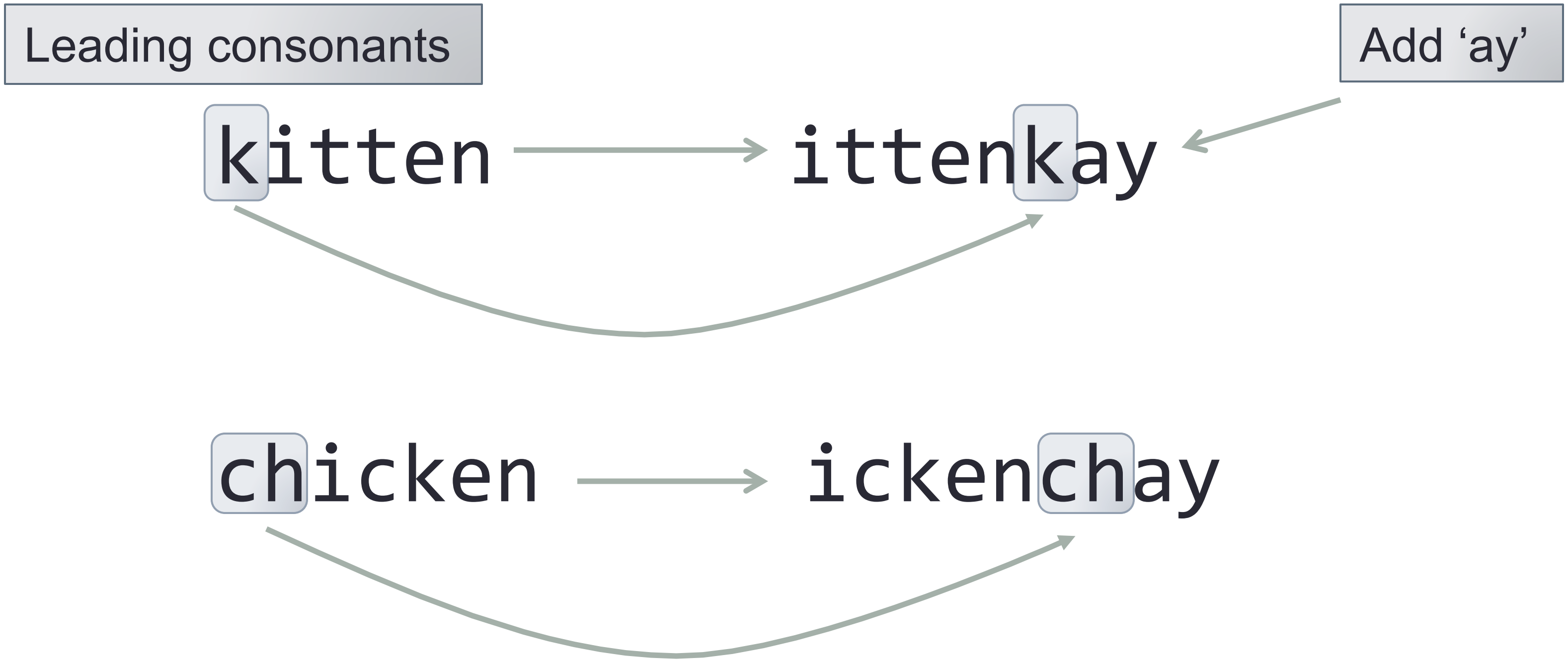
1. Figure out your base case – this is the simplest possible question someone could ask you about this function
   - What's the simplest possible factorial?
   - `fact(0)` is just 1
2. Now consider the general case, and treat a slightly simpler recursive call as a functional abstraction
   - General case – `factorial(n)` for some arbitrary n
   - Simpler recursive call – `factorial(n-1)`

   a. Leap of faith! Assume that the simpler call *just works*.
      - So we're assuming `factorial(n-1)` will correctly return `(n-1)`!

   b. Use the simpler call to make the general case work.
      - We can write `factorial(n)` as `n*factorial(n-1)`

Pig Latin

- Yes, there is a slide on Pig Latin in this lecture
- Yes, it’s okay if you tell your friends



Leading consonants

Add ‘ay’

kitten → ittenkay

chicken → ickenchay

- Don’t worry about if there aren’t any vowels!

Eaplay of aithfay, step 1

Step 1: Figure out your base case – this is the simplest possible question someone could ask you about this function

Pig latin-ize a word that already begins with a vowel!

How do you solve this problem?

Add ‘ay’ to the end of the word!

- This tells us two things:
  1. The condition that indicates the base case – word begins with a vowel
  2. What to do when you encounter the base case – just add ‘ay’ to the end

Eaplay of aithfay, step 2

Step 2: Now consider the general case, and treat a slightly simpler recursive call as a functional abstraction

General case: `piglatin(word)` for some arbitrary `word`

Slightly simpler: `word`, but with the first consonant moved to the end

a. Leap of faith! Assume that the simpler call *just works*.
   - Assume that calling `piglatin` on `word` with the first consonant moved to the end works
   - For example, given `‘hello’` as our `word`, assume `piglatin(‘elloh’)` works

b. Use the simpler call to make the general case work
   - `piglatin(‘hello’) == piglatin(‘elloh’) == ‘ellohay’`

Your turn: reverse a string (recursively)

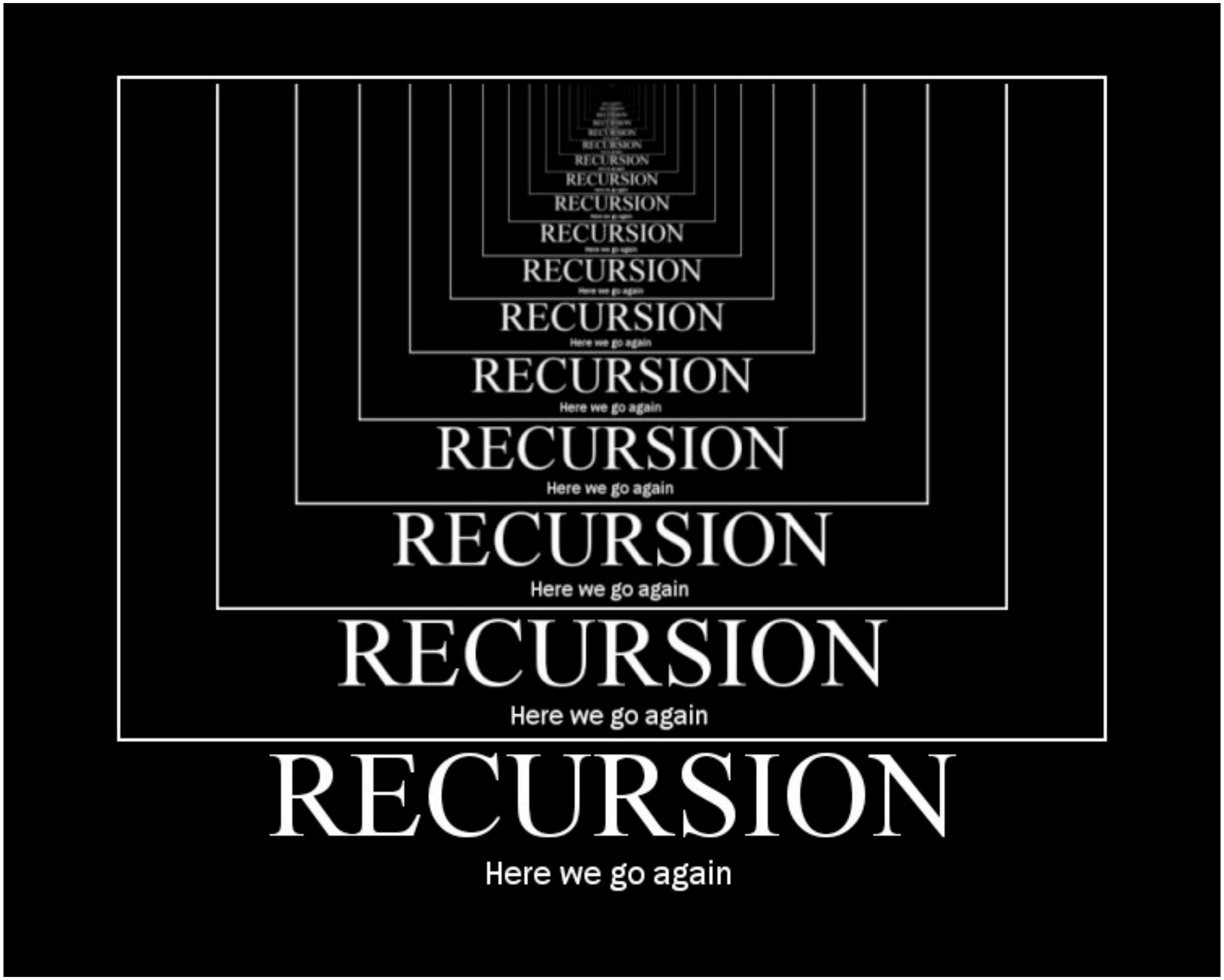
- Write a function `reverse` that takes a string and returns that string, but in reverse:

```
>>> reverse('steven tang')
'gnat nevets'
```

- You may find the following tricks useful:

```
>>> 'hello'[0]  # first letter
'h'
>>> 'hello'[1:] # everything but first letter
'ello'
```

Break!



Fibonacci sequence

- The Fibonacci sequence is defined as

$$\mathrm{fib}(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2), & n > 1 \end{cases}$$

```python
def fib_iter(n):
    if n == 0:
        return 0
    fib_n, fib_n_1 = 1, 0
    k = 1
    while k < n:
        fib_n, fib_n_1 = fib_n_1 + fib_n, fib_n
        k += 1
    return fib_n
```

Fibonacci sequence

- The Fibonacci sequence is defined as

$$\mathrm{fib}(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2), & n > 1 \end{cases}$$

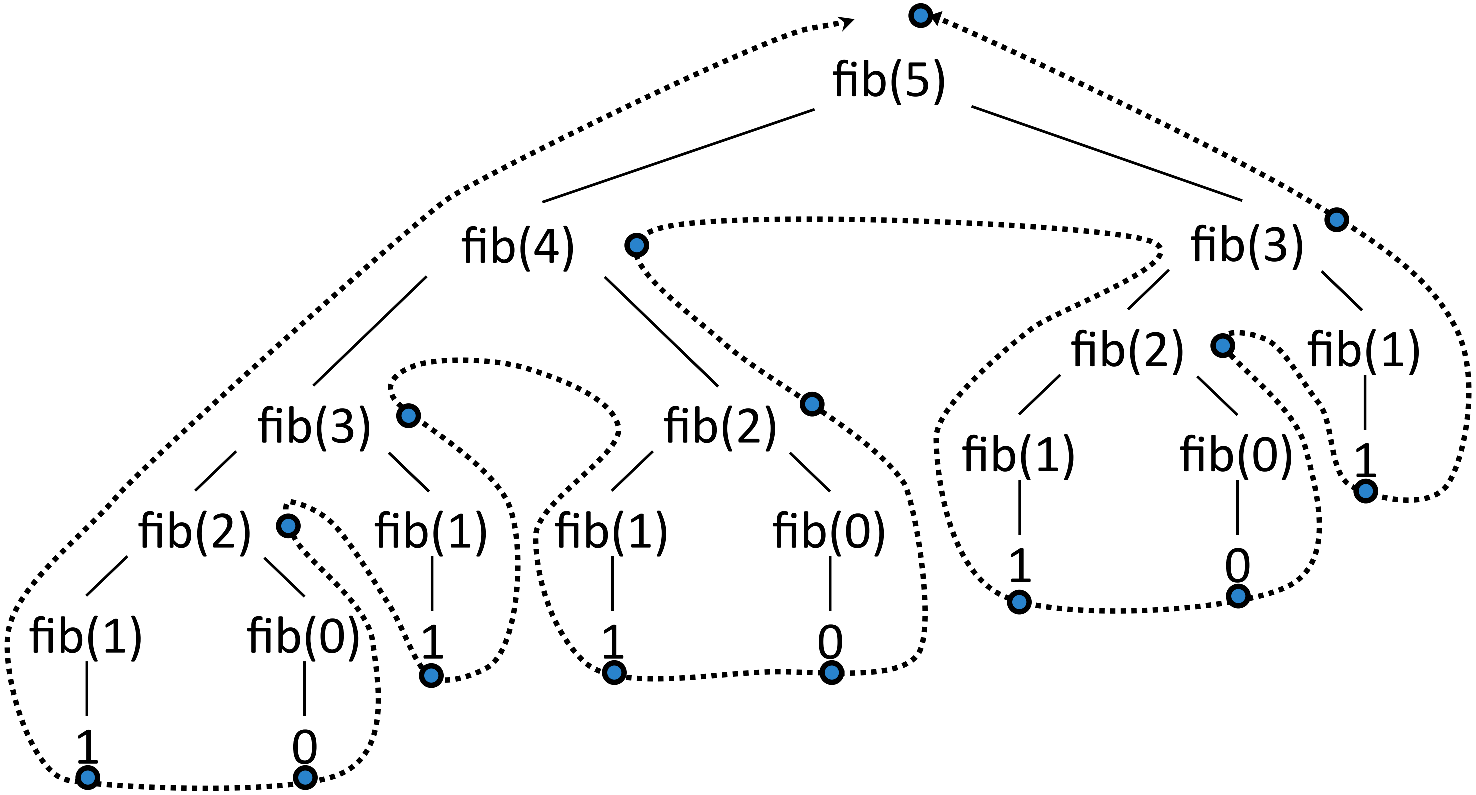
```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

Two recursive calls!

Tree recursion

Executing the body of a function may entail more than one recursive call to that function

This is called *tree recursion*

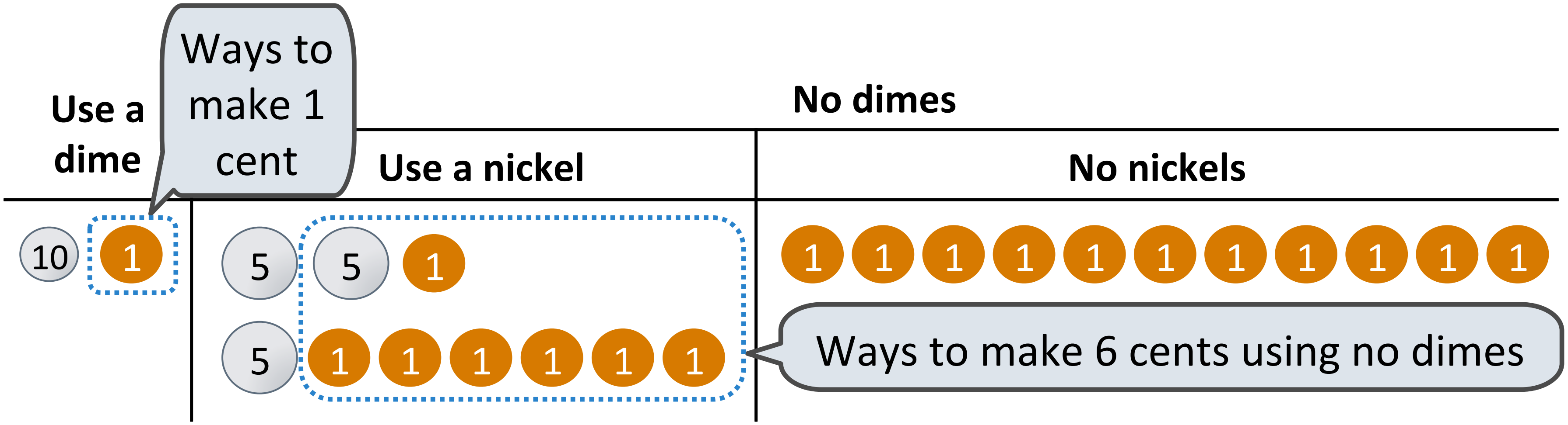


What changes with tree recursion?

- Not much!
- Multiple base cases are more common
  - But you can have multiple base cases in non-tree recursion too!
- Need to take the leap of faith multiple times
  - Think of multiple simpler recursive calls
  - But each simpler call is treated the same way as before

Example: counting change

- $1.00 = $0.50 + $0.25 + $0.10 + $0.10 + $0.05
- $1.00 = 1 half dollar, 1 quarter, 2 dimes, 1 nickel
- $1.00 = 2 quarters, 2 dimes, 30 pennies
- $1.00 = 100 pennies



Use a dime

Ways to make 1 cent

No dimes

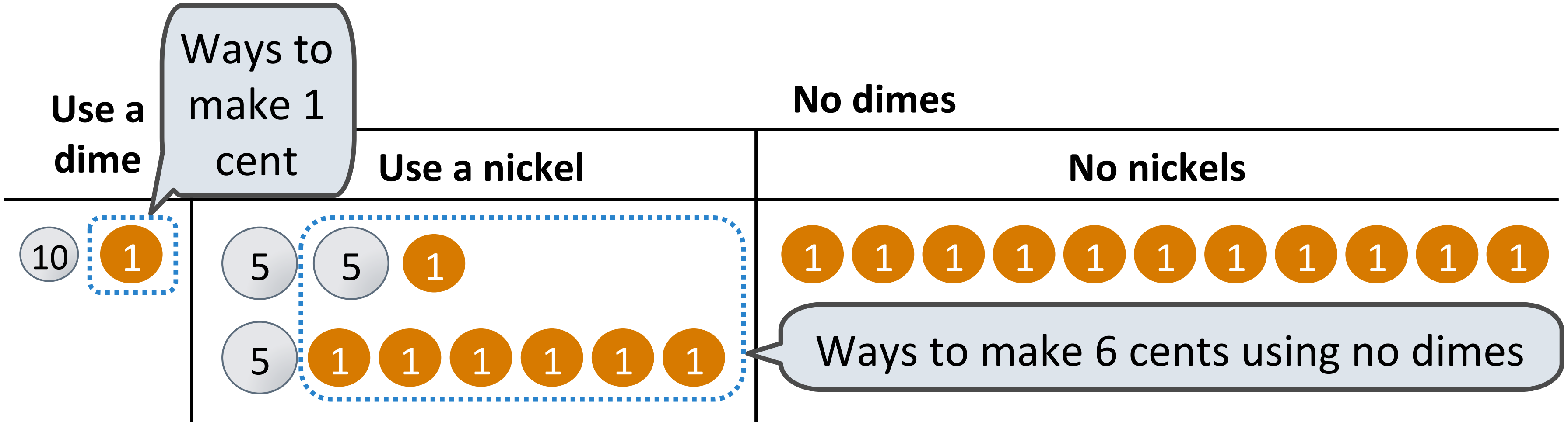
Use a nickel

No nickels

Ways to make 6 cents using no dimes

Example: counting change

- The number of ways to change an amount using a fixed set of kinds of coins is the sum of…
    1. The number of ways to change the remaining amount if you use your largest coin once
    2. The number of ways to change the full amount, without using your largest type of coin



Use a dime

Ways to make 1 cent

No dimes

Use a nickel

No nickels

Ways to make 6 cents using no dimes

Example: counting change

- The number of ways to change an amount using a fixed set of kinds of coins is the sum of…
  1. The number of ways to change the remaining amount if you use your largest coin once
  2. The number of ways to change the full amount, without using your largest type of coin

```
def count_change(a, d):
    if a == 0:
        return 1
    if a < 0 or d == 0:
        return 0
    return (count_change(a-d, d) +
            count_change(a, next_coin(d)))
```

One way to make no amount

Can't make negative amount, or any amount with no coins

Functional abstraction to get next coin