

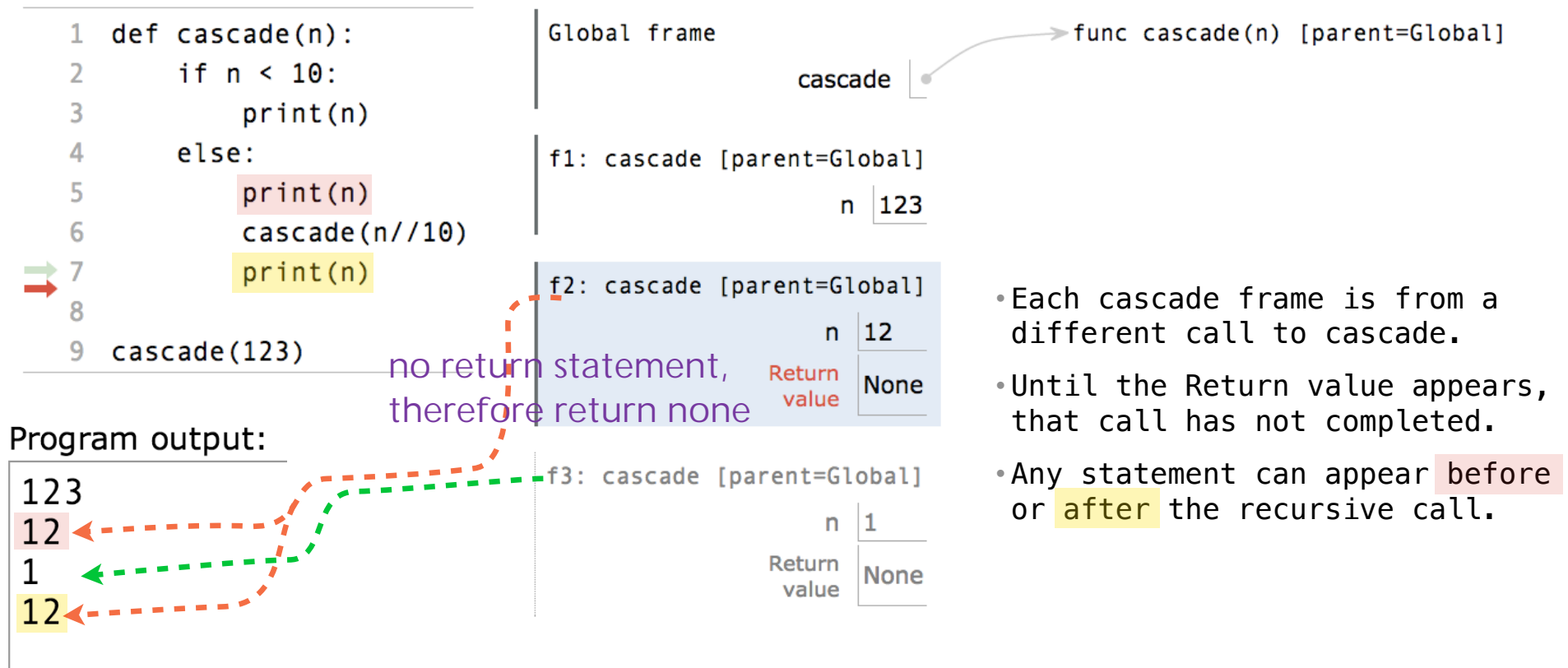
Tree Recursion

Announcements

Order of Recursive Calls

The Cascade Function

(Demo)



Interactive Diagram

Two Definitions of Cascade

(Demo)

```
def cascade(n):  
    if n < 10:  
        print(n)  
    else:  
        print(n)  
        cascade(n//10)  
        print(n)
```

```
def cascade(n):  
    print(n)  
    if n >= 10:  
        cascade(n//10)  
        print(n)
```

- If two implementations are equally clear, then shorter is usually better
- In this case, the longer implementation is more clear (at least to me)
- When learning to write recursive functions, put the base cases first
- Both are recursive functions, even though only the first has typical structure

Remember you write programs for other to read, only run by computers incidentally.

Example: Inverse Cascade

Inverse Cascade

Write a function that prints an inverse cascade:

```
1
12
123
1234
123
12
1
```

```
def inverse_cascade(n):
    grow(n)
    print(n)
    shrink(n)
```

```
def f_then_g(f, g, n):
    if n:
        f(n)
        g(n)
```

```
grow = lambda n: f_then_g(
shrink = lambda n: f_then_g(
```

```
grow = lambda n: f_then_g(grow, print, n//10)
shrink = lambda n: f_then_g(print, shrink, n//10)
```

difficult for me orz;

perhaps use the idea of abstraction, think of its function:

so grow has to grow first and then print, shrink the other way around

Tree Recursion

Tree Recursion

Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call

n: 0, 1, 2, 3, 4, 5, 6, 7, 8, ... , 35

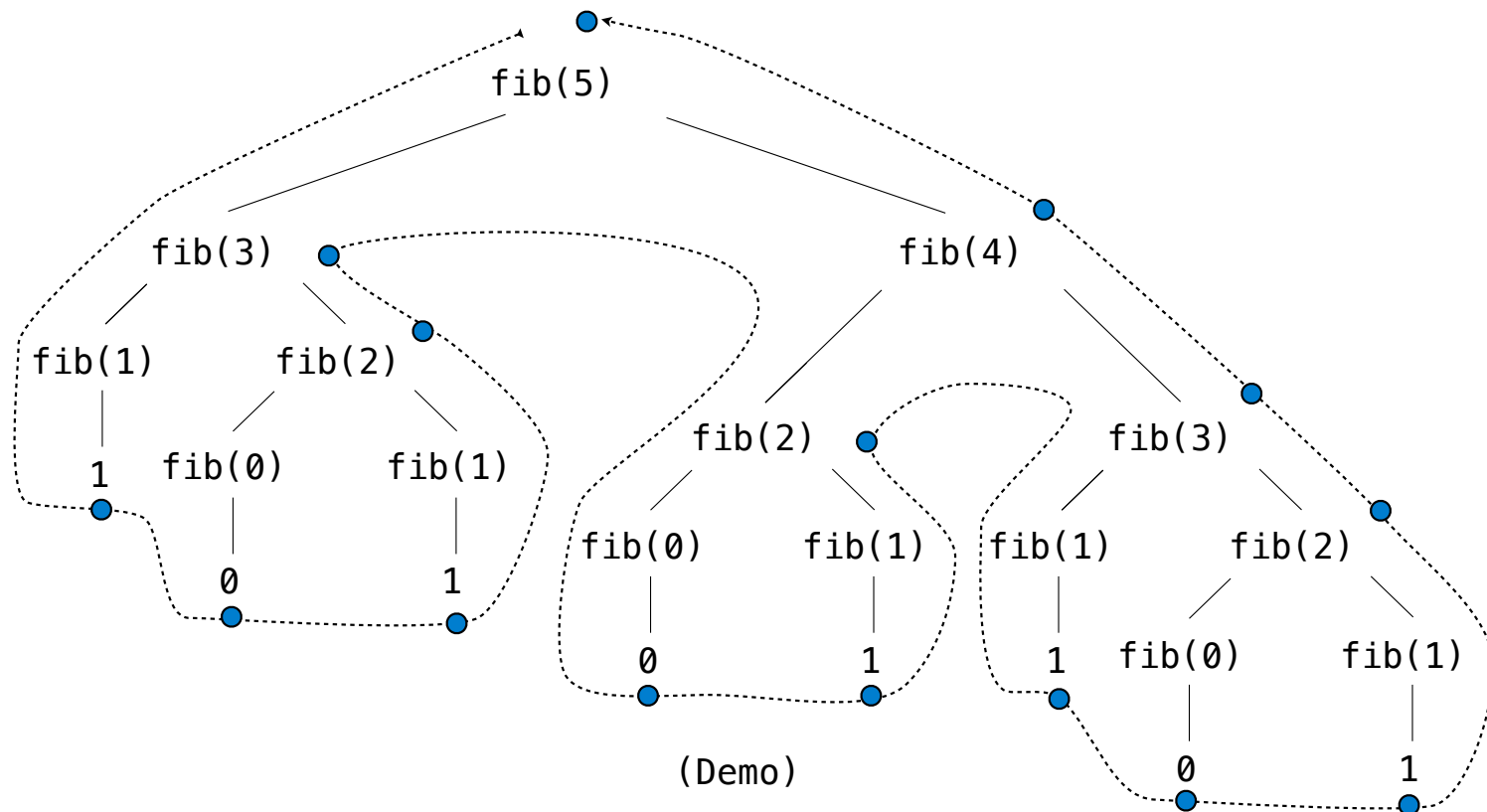
fib(n): 0, 1, 1, 2, 3, 5, 8, 13, 21, ... , 9,227,465

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



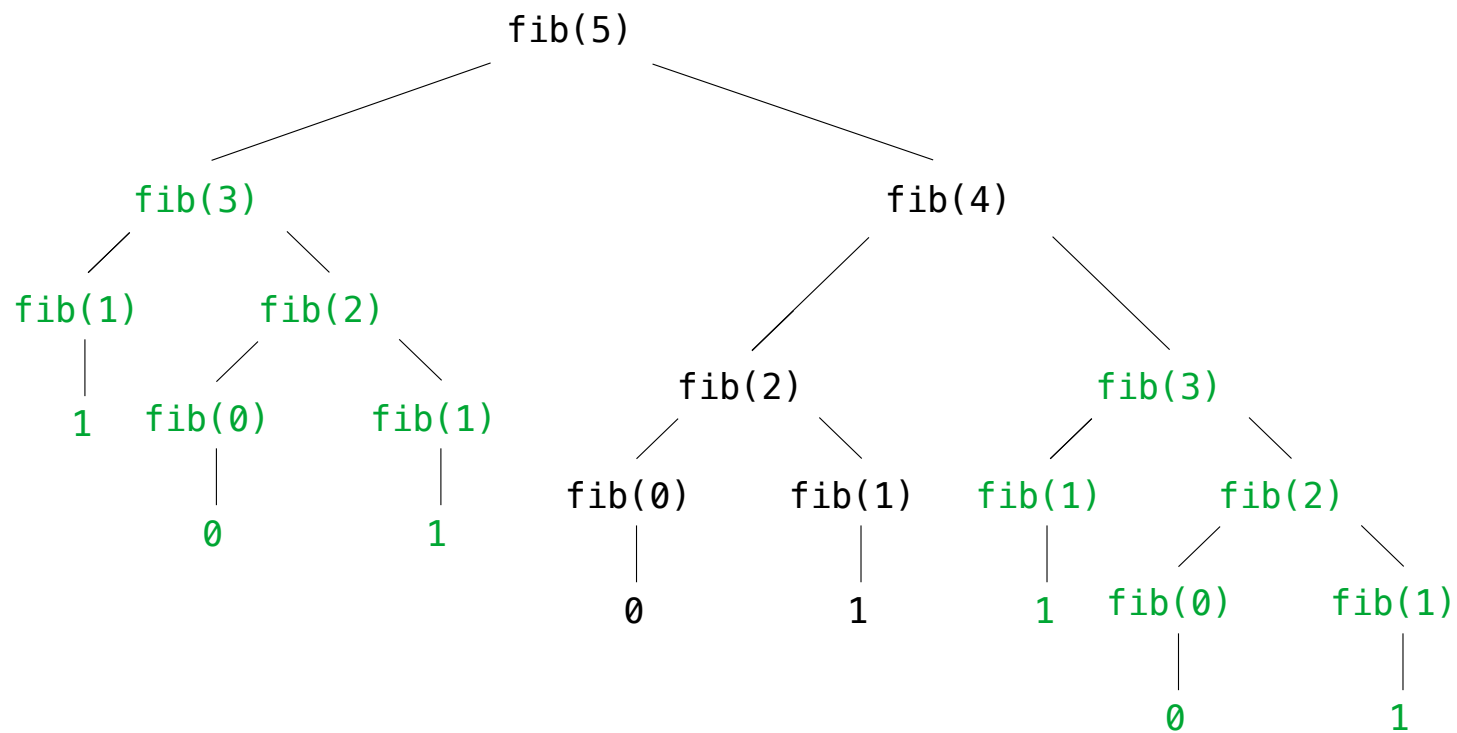
A Tree-Recursive Process

The computational process of fib evolves into a tree structure



Repetition in Tree-Recursive Computation

This process is highly repetitive; fib is called on the same argument multiple times



(We will speed up this computation dramatically in a few weeks by remembering results)

Example: Counting Partitions

Counting Partitions

The number of partitions of a positive integer n , using parts up to size m , is the number of ways in which n can be expressed as the sum of positive integer parts up to m in increasing order.

`count_partitions(6, 4)`

$$2 + 4 = 6$$

$$1 + 1 + 4 = 6$$

$$3 + 3 = 6$$

$$1 + 2 + 3 = 6$$

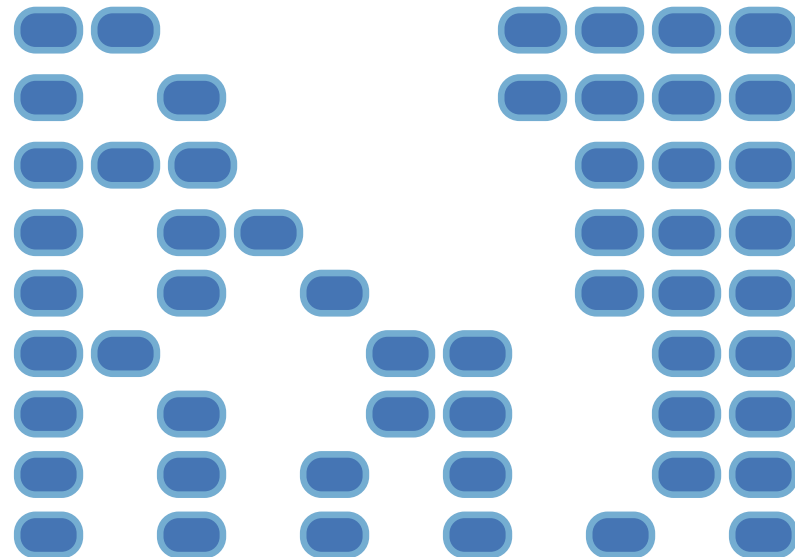
$$1 + 1 + 1 + 3 = 6$$

$$2 + 2 + 2 = 6$$

$$1 + 1 + 2 + 2 = 6$$

$$1 + 1 + 1 + 1 + 2 = 6$$

$$1 + 1 + 1 + 1 + 1 + 1 = 6$$

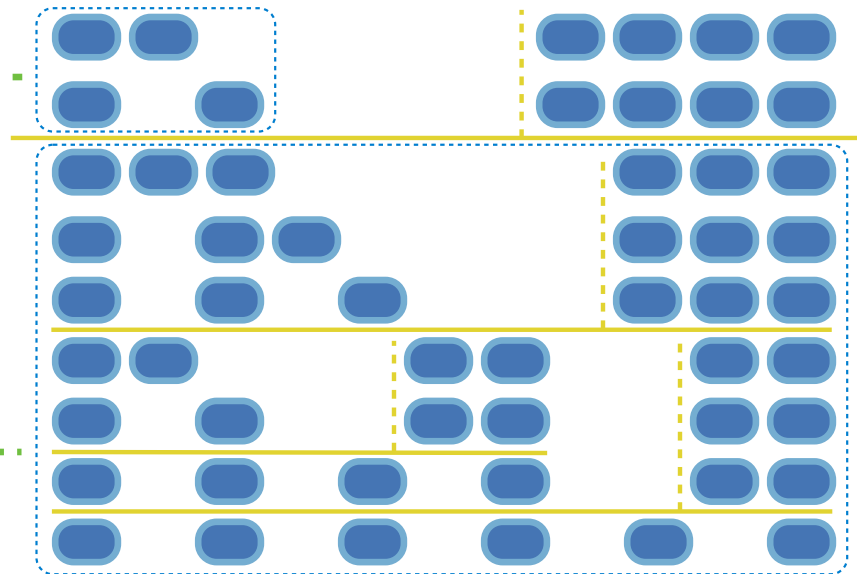


Counting Partitions

The number of partitions of a positive integer n , using parts up to size m , is the number of ways in which n can be expressed as the sum of positive integer parts up to m in increasing order.

`count_partitions(6, 4)`

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
 - Use at least one 4
 - Don't use any 4
- Solve two simpler problems:
 - `count_partitions(2, 4)`
 - `count_partitions(6, 3)`
- Tree recursion often involves exploring different choices.



Counting Partitions

The number of partitions of a positive integer n , using parts up to size m , is the number of ways in which n can be expressed as the sum of positive integer parts up to m in increasing order.

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
 - Use at least one 4
 - Don't use any 4
- Solve two simpler problems:
 - `count_partitions(2, 4)`
 - `count_partitions(6, 3)`
- Tree recursion often involves exploring different choices.

```
def count_partitions(n, m):  
    if n == 0:  
        return 1  
    elif n < 0:  
        return 0  
    elif m == 0:  
        return 0  
    else:  
        with_m = count_partitions(n-m, m)  
        without_m = count_partitions(n, m-1)  
        return with_m + without_m
```

(Demo)

Interactive Diagram