Tree Recursion

Announcements

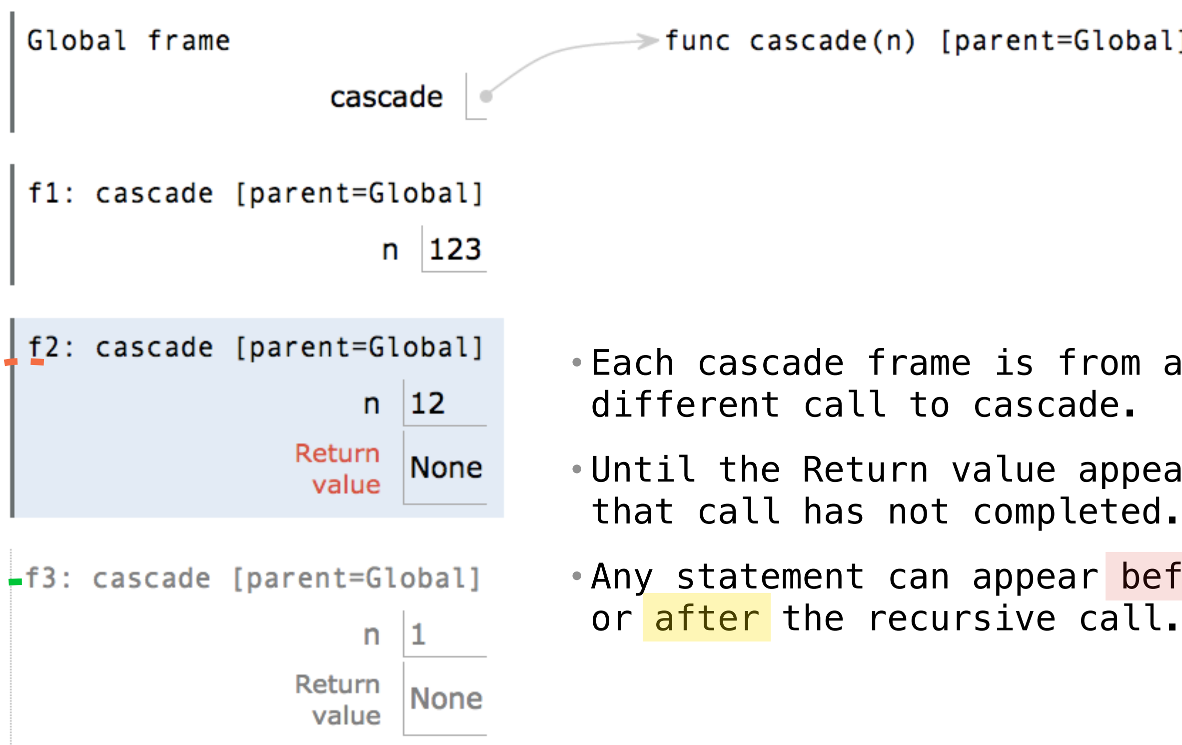
Order of Recursive Calls

The Cascade Function

(Demo)

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

cascade(123)
```



Program output:

```
123
12
1
12
```

- Each cascade frame is from a different call to cascade.
- Until the Return value appears, that call has not completed.
- Any statement can appear before or after the recursive call.

http://pythontutor.com/composingprograms.html#code=def%20cascade%28n%29%3A%20%20%20%20%0A%20%20%20%20if%20n%20%3C%2010%3A%20%20%20%20%20%20%20%20%0A%20%20%20%20%20%20%20%20print%28n%29%20%20%20%20%0A%20%20%20%20else%3A%20%20%20%20%20%20%20%20%0A%20%20%20%20%20%20%20%20print%28n%29%20%20%20%20%20%20%20%20%20%0A%20%20%20%20%20%20%20%20cascade%28n//10%29%20%20%20%20%20%20%20%20%0A%20%20%20%20%20%20%20%20print%28n%29%20%20%20%20%20%20%20%20%0A%20%20%20%20%20%20%20%20%0Acascade%28123%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

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

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(                    )
```

Tree Recursion

Tree Recursion

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

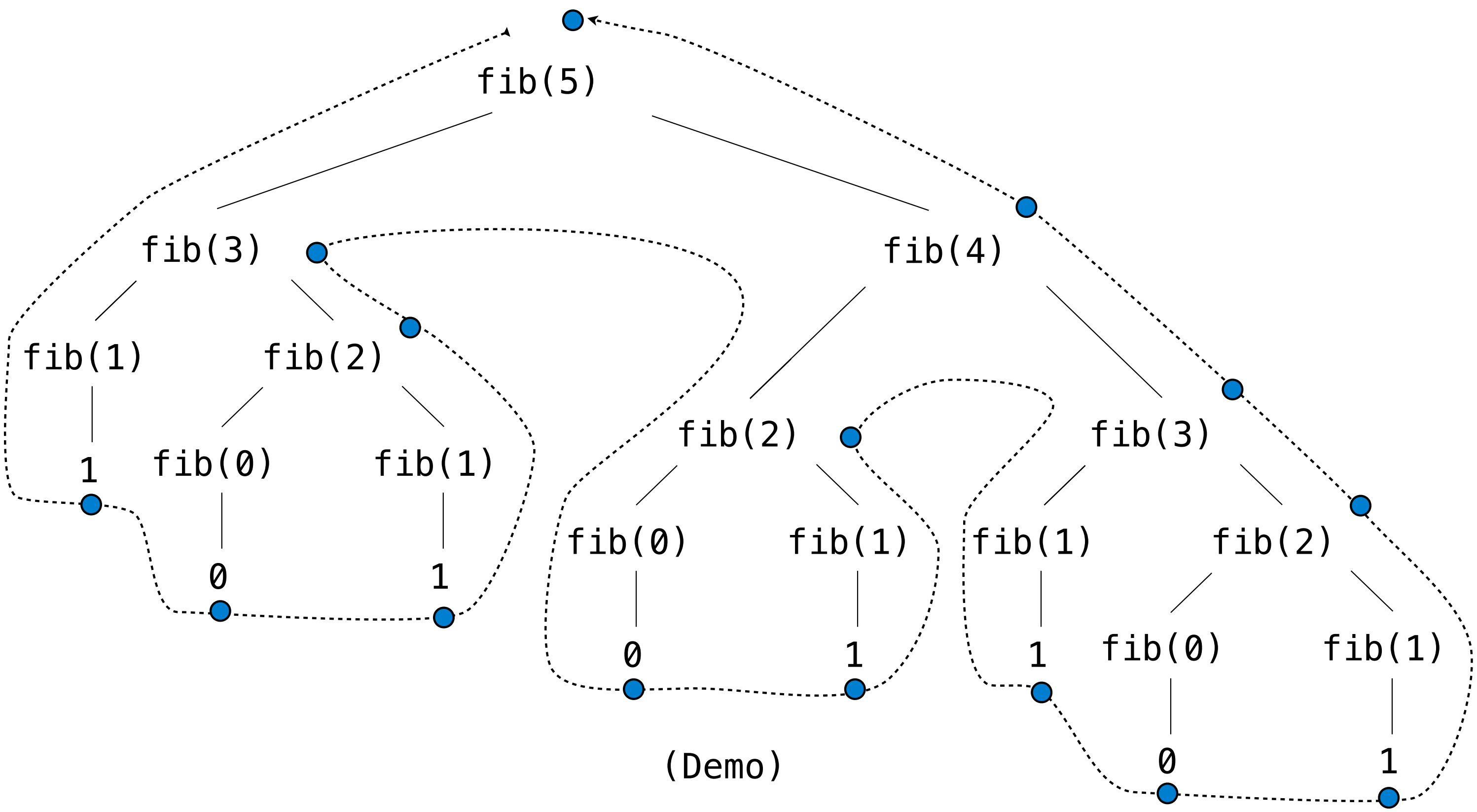
n:	0,	1,	2,	3,	4,	5,	6,	7,	8,	... ,	35
fib(n):	0,	1,	1,	2,	3,	5,	8,	13,	21,	... ,	9,227,465

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

http://en.wikipedia.org/wiki/File:Fibonacci.jpg

A Tree-Recursive Process

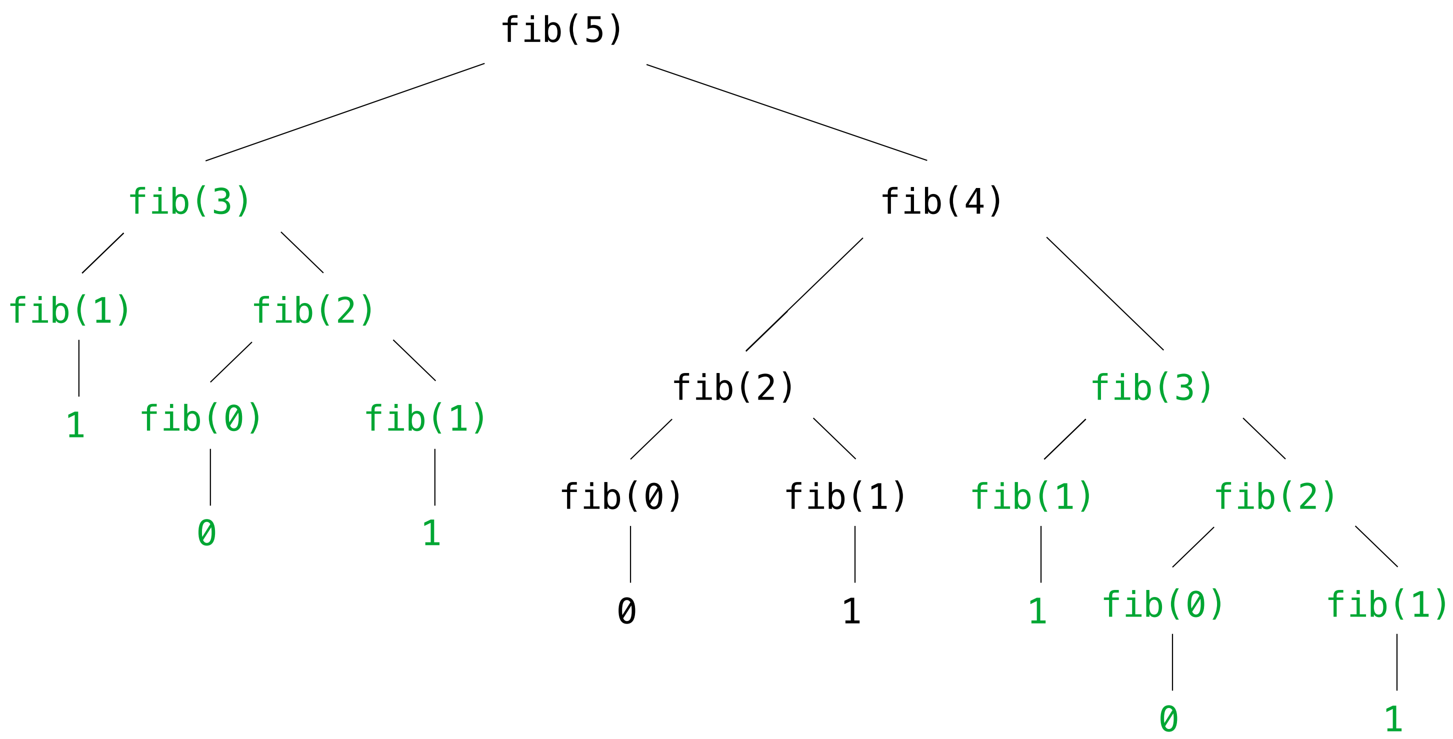
The computational process of fib evolves into a tree structure



(Demo)

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)

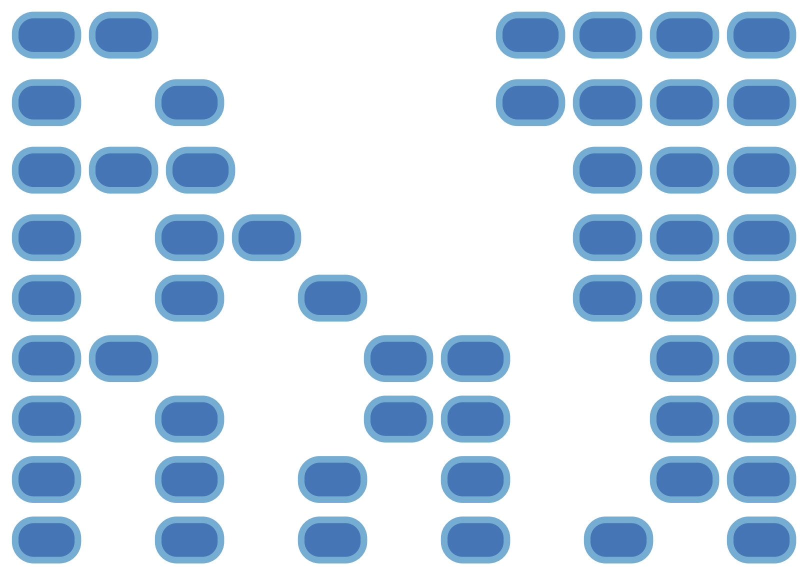
Example: Counting Partitions

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)`

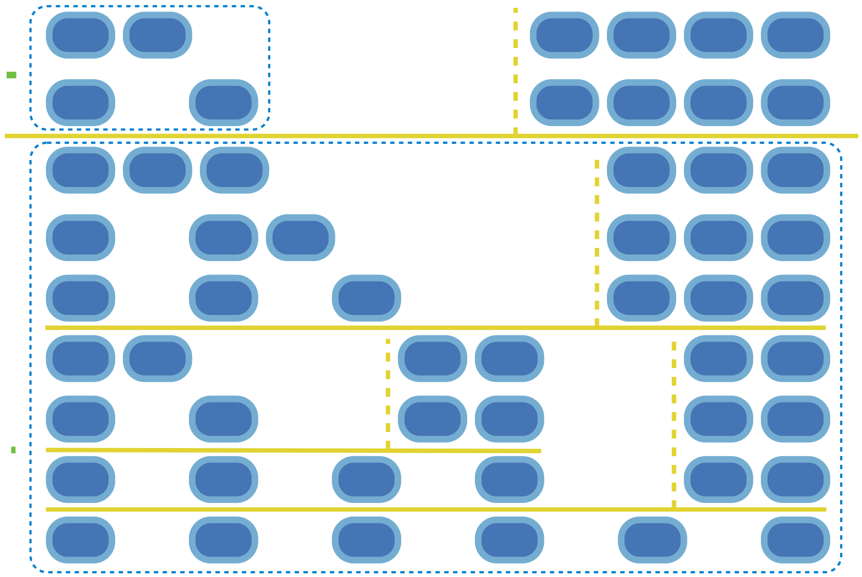
2 + 4 = 6
1 + 1 + 4 = 6
3 + 3 = 6
1 + 2 + 3 = 6
1 + 1 + 1 + 3 = 6
2 + 2 + 2 = 6
1 + 1 + 2 + 2 = 6
1 + 1 + 1 + 1 + 2 = 6
1 + 1 + 1 + 1 + 1 + 1 = 6



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 non-
decreasing 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)

pythontutor.com/composingprograms.html#code=def%20count_partitions%28n,%20m%29%3A%0A%20%20%20%20if%20n%20%3D%3D%200%3A%0A%20%20%20%20%20%20%20%20return%201%0A%20%20%20%20elif%20n%20<%200%3A%0A%20%20%20%20%20%20%20%20return%200%0A%20%20%20%20elif%20m%20%3D%3D%200%3A%0A%20%20%20%20%20%20%20%20return%200%0A%20%20%20%20else%3A%0A%20%20%20%20%20%20%20%20with_m%20%3D%20count_partitions%28n-m,
%20m%29%20%0A%20%20%20%20%20%20%20%20without_m%20%3D%20count_partitions%28n,
%20m-1%29%0A%20%20%20%20%20%20%20%20return%20with_m%20%2B%20without_m%0A%20%20%20%20%20%20%20%20%0Aresult%20%3D%20count_partitions%285,%203%29%0A%0A#%201%20%2B%201%20%2B%201%20%2B%201%20%2B%201%20%3D%205%0A#%201%20%2B%201%20%2B%201%20%2B%202%20%2B%20%20%20%3D%205%0A#%201%20%2B%202%20%2B%202%20%2B%20%20%20%20%20%20%20%3D%205%0A#%201%20%2B%201%20%2B%203%20%2B%20%20%20%20%20%20%20%20%3D%205%0A#
20%2B%203%20%2B%20%20%20%20%20%20%20%20%20%20%20%20%3D%205&mode=display&origin=composingprograms.js&cumulative=false&py=3&rawInputLstJSON=[]&curInstr=0