Lecture #8: More Recursion

Announcements:

- Project #1 due next Thursday (13 Feb).
- Test #1 Tuesday, 18 Feb at 8PM.
- AWE 61A Party this Sunday (9 Feb) in the Woz, 1–3PM.
- Guerilla Sections this weekend (see Piazza).
- Self-assessment quiz will be released tonight, due Monday. Watch the website and Piazza.

A Simple Recursion

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

cascade(123)
```

Program output:

```
123
12
1
12
```

Frames | Objects

Global frame
cascade → func cascade(n)

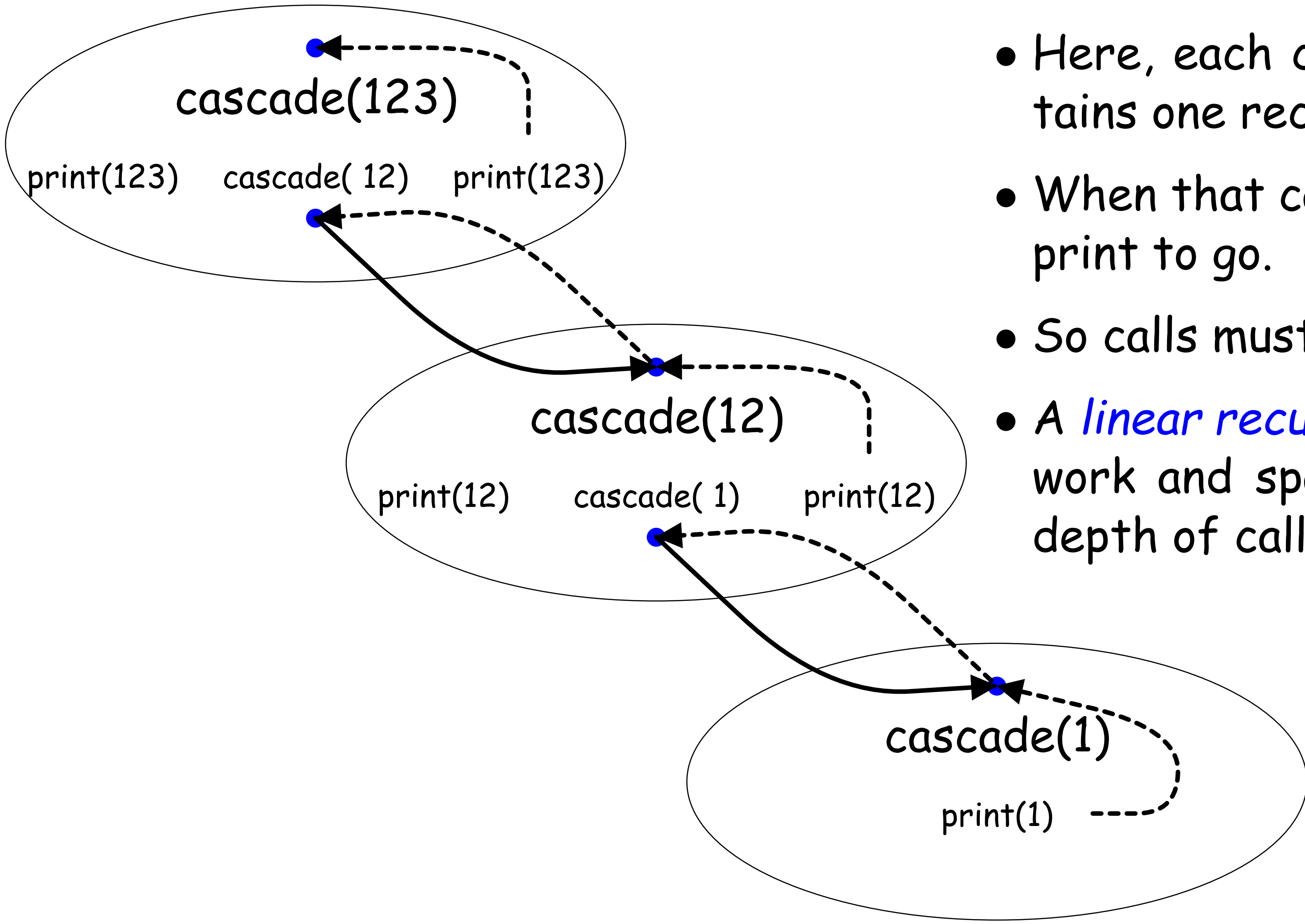
cascade
n 123

cascade
n 12
Return value None

cascade
n 1
Return value None

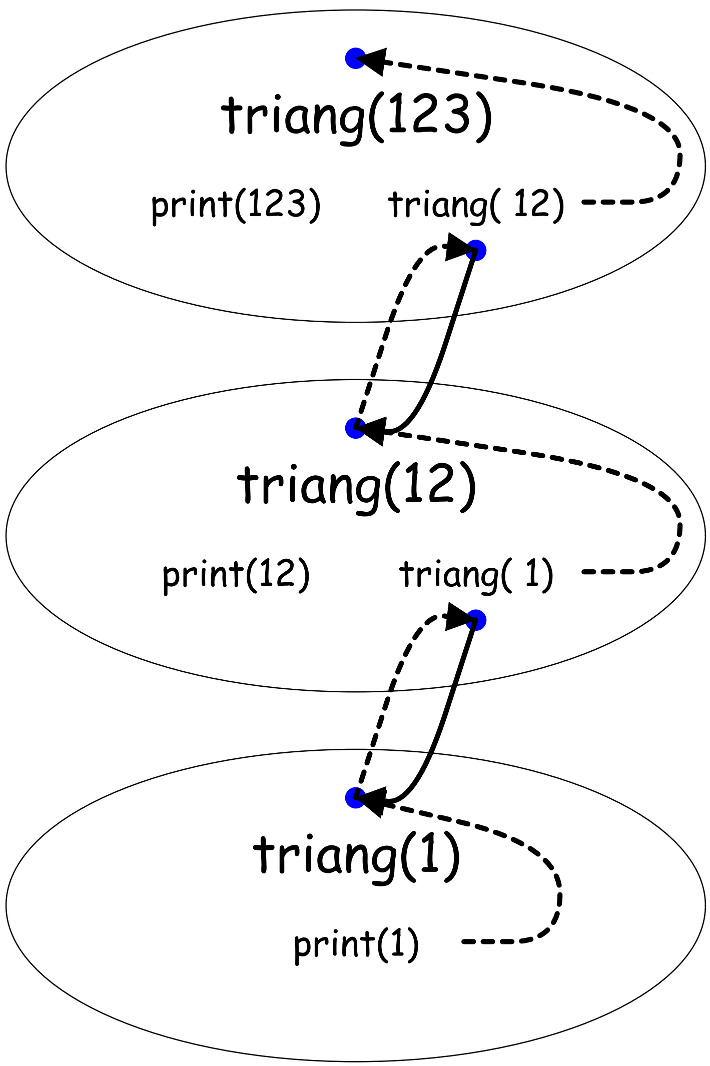
- Each frame connects to the global frame.
- Frames without "Return value" are still active
- Each recursive call has its own *n* value.
- That's how it works, but try *not* to think of it this way!
- Think recursively instead.

Classifying Recursions: Linear Recursions



- Here, each call of cascade contains one recursive call.
- When that call completes, still a print to go.
- So calls must remain pending.
- A *linear recursive process:* total work and space proportional to depth of calls.

Classifying Recursions: Iterative Processes




```
def triang(n):
    print(n)
    if n < 10: triang(n-1)
```

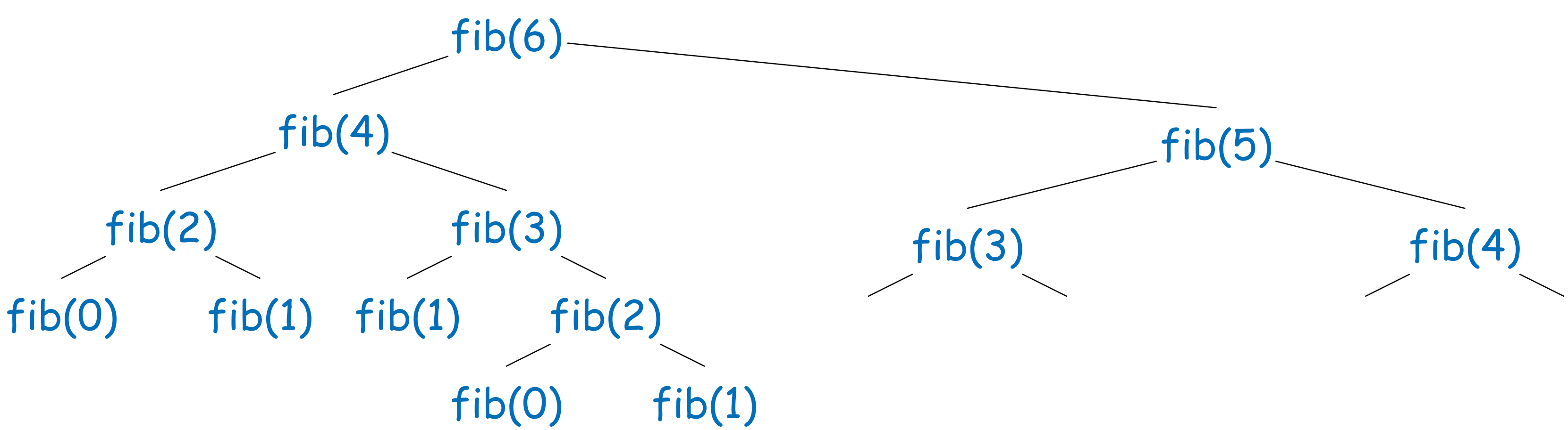
- Again, each call of triang contains one recursive call.
- So this is a type of linear recursive process.
- But there's no more to do when that call completes (*tail recursive*)
- So in principle, calls need not remain pending.
- An *iterative process:* total work still proportional to depth of calls, but total space need not be.
- This kind is suitable for a loop.

Classifying Recursion: Tree Recursions

- Previously, we looked at a program for computing values in the Fibonacci sequence:

```
def fib(n):
    """The Nth Fibonacci number, N>=0."""
    assert n >= 0
    if n <= 1:
        return n
    else:
        return fib(n-2) + fib(n-1)
```

Here, each invocation of fib makes *two* calls: work is exponential in depth of calls: A *tree-recursive process*.



A Tree Recursion: Partitions

- *partitions(n, k):* The number of non-decreasing sequences of two or more positive integers between 1 and k that add up to n.
- For example, *partitions(6, 4)* is 9:

```
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
```

Computing Partitions

- Observation: can choose sizes 1–k for the last partition.
- If we choose size k for the last partition, then how many ways are there to partition the rest?

  .

- Suppose we choose not to use size k for the last partition, then how many choices are there?

  .

- Finally, there is only one way to partition 0 items or to partition a negative number of items or a positive number of items with maximum partition size of 0.

Computing Partitions

- Observation: can choose sizes 1–k for the last partition.
- If we choose size k for the last partition, then how many ways are there to partition the rest?
- The number of ways of partitioning $n - k$ items of maximum size k.
- Suppose we choose not to use size k for the last partition, then how many choices are there?

.

- Finally, there is only one way to partition 0 items or to partition a negative number of items or a positive number of items with maximum partition size of 0.

Computing Partitions

- Observation: can choose sizes 1–k for the last partition.
- If we choose size k for the last partition, then how many ways are there to partition the rest?
- The number of ways of partitioning $n - k$ items of maximum size k.
- Suppose we choose not to use size k for the last partition, then how many choices are there?
- The number of ways of partitioning n items of maximum size $k - 1$.
- Finally, there is only one way to partition 0 items or to partition a negative number of items or a positive number of items with maximum partition size of 0.

Partitions, concluded

This leads to the following program:

```
def partitions(n, k):
    """The number of ways of partitioning N items into partitions of si
    <=K."""
    if n == 0:
        return 1
    elif n < 0 or k <= 0:
        return 0
    else:
        with_k =
        without_k =
        return with_k + without_k
```

Partitions, concluded

This leads to the following program:

```
def partitions(n, k):
    """The number of ways of partitioning N items into partitions of si
    <=K."""
    if n == 0:
        return 1
    elif n < 0 or k <= 0:
        return 0
    else:
        with_k = partitions(n-k, k)
        without_k = partitions(n, k-1)
        return with_k + without_k
```