

Some instrumentation to help see/show how recursion works. While this needs to be executed, you can ignore the details (unless you're curious!) and skip directly to "Recursive Patterns" below.

```
In [ ]: 1 from functools import wraps
2 import sys
3 def instrument(f):
4     """This is a helpful wrapper, to instrument a function to show the
5     call entry and exit from that function.
6     """
7     @wraps(f)
8     def wrapper(*args, **kwargs):
9         call_depth = wrapper.call_count
10        wrapper.call_count += 1
11        argstr = ', '.join([str(args[i]) for i in range(len(args))])
12        sys.stderr.write("    *call_depth + \"call to \" + f.__name__ + \": \" + argstr + \"\n\")
13        result = f(*args, **kwargs)
14        sys.stderr.write("    *call_depth + f.__name__ + \" returns: \" + str(result) + \"\n\")
15        wrapper.call_count -= 1
16        return result
17    wrapper.call_count = 0
18    return wrapper
```

Recursive Patterns

Let's start with some simple functions that recurse on lists...

Walk the list to find the first value satisfying function f

```
In [ ]: 1 @instrument
2 def walk_list(L, f):
3     """Walk a list -- in a recursive style. Note that this is done as a
4     stepping stone toward other recursive functions, and so does not
5     use easier/direct built-in list functions.
6
7     In this first version -- walk the list just to find/return the
8     FIRST item that satisfies some condition, where f(item) is true.
9
10    >>> walk_list([1, 2, 3], lambda x: x > 2)
11    3
12    """
13    pass
```

```
In [ ]: 1 walk_list([1, 2, 3], lambda x: x > 2)
```

Walk a list, but now returning a *list* of items that satisfy f -- uses stack

```
In [ ]: 1 @instrument
2 def walk_list_filter1(L, f):
3     """ Walk a list, returning a list of items that satisfy the
4     condition f.
5
6     This implementation uses the stack to hold intermediate results,
7     and completes construction of the return list upon return of
8     the recursive call.
9
10    >>> walk_list_filter1([1, 2, 3], lambda x: x % 2 == 1) #odd only
11    [1, 3]
12    """
13    pass
```

```
In [ ]: 1 walk_list_filter1([1, 2, 3], lambda x: x % 2 == 1)
```

Walk a list, returning a list of items that satisfy f -- uses helper with a "so_far" argument

```
In [ ]: 1 @instrument
2 def walk_list_filter2(L, f):
3     """ Walk a list, returning a list of items that satisfy the
4         condition f.
5
6         This implementation uses a helper with an explicit 'so far'
7         variable, that holds the return value as it is being built
8         up incrementally on each call.
9
10        >>> walk_list_filter2([1, 2, 3], lambda x: x % 2 == 1)
11        [1, 3]
12        """
13    @instrument
14    def helper(L, ans_so_far):
15        pass
16
17    return helper(L, [])
```

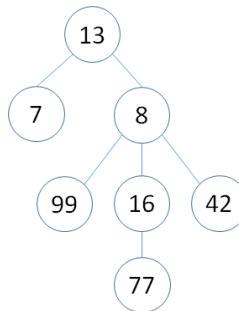
```
In [ ]: 1 walk_list_filter2([1, 2, 3], lambda x: x % 2 == 1)
```

Note the difference in how this works. `walk_list_filter2` builds up the result as an evolving argument to `helper`. When we're done, the stack does nothing more than keep passing that result back up the call chain (i.e., is written in a tail-recursive fashion). In contrast, `walk_list_filter1` uses the stack to hold partial results, and then does further work to build or complete the result after each recursive call returns.

Now consider some functions that recurse on trees...

We want to extend the basic idea of recursive walkers and builders for lists, now to trees. We'll see the same patterns at work, but now often with more base cases and/or more recursive branch cases.

For these examples, we need a simple tree structure. Here we'll represent a node in a tree as a list with the first element being the node value, and the rest of the list being the children nodes. That is to say, our tree structure is a simple nested list structure.



```
In [ ]: 1 tree1 = [13,
2             [7],
3             [8,
4             [99],
5             [16,
6             [77]],
7             [42]]]
8 tree1
```

```
In [ ]: 1 @instrument
2 def tree_max(tree):
3     """Walk a tree, returning the maximum value in the (assumed non-empty) tree. """
4     pass
```

```
In [ ]: 1 tree_max(tree1)
```

```
In [ ]: 1 @instrument
2 def depth_tree(tree):
3     """ Walk a tree, returning the depth of the tree
4     >>> depth_tree([13, [7], [8, [99], [16, [77]], [42]]])
5     """
6     pass
```

```
In [ ]: 1 depth_tree([13, [7], [8, [99], [16, [77]], [42]]])
```

Notice that the recursion structure is exactly the same in both cases? We could generalize to something like a `walk_tree` that took a tree *and* a function `f` (and perhaps some other base case values), and did that operation at each step. We'll leave that as an exercise for the reader.

Now a "builder" or "maker" function, that recursively creates a tree structure...

```
In [ ]: 1 @instrument
2 def make_tree(L):
3     """ Make and return a binary tree corresponding to the list. The
4     tree is "binary" in the sense that left and right branches are
5     balanced as much as possible, but no condition is imposed on the
6     left/right values under each node in the tree.
7
8     >>> make_tree([1,2,3])
9     [1, 2, 3]
10    """
11    pass
```

```
In [ ]: 1 tree2 = make_tree([1, 2, 3])
2 tree2
```

```
In [ ]: 1 tree3 = make_tree([1, 2]) #unbalanced tree case
2 tree3
```

How many recursive calls do you expect for a list of length `n`?

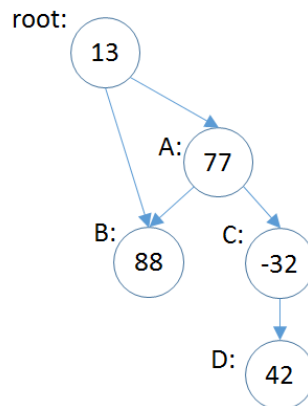
```
In [ ]: 1 tree4 = make_tree(list(range(8)))
2 tree4
```

```
In [ ]: 1 def show_tree(tree):
2         """ Return a formatted string representation to visualize a tree """
3         spaces = ' '
4         def helper(tree, level):
5             if not tree:
6                 return ""
7             val = tree[0]
8             children = tree[1:]
9             result = spaces*level + str(val) + '\n'
10            for child in children:
11                result += helper(child, level+1)
12            return result
13        return helper(tree, 0)
```

```
In [ ]: 1 print("tree4:", tree4, "\n", show_tree(tree4))
```

Finally, consider some functions that recurse on graphs...

For this, we need a more sophisticated structure, since a node may be referenced from more than one other node. We'll represent a (directed) graph as a dictionary with node names as keys, and associated with the key is a list holding the node value and a list of children node names. The special name 'root' is the root of the graph.

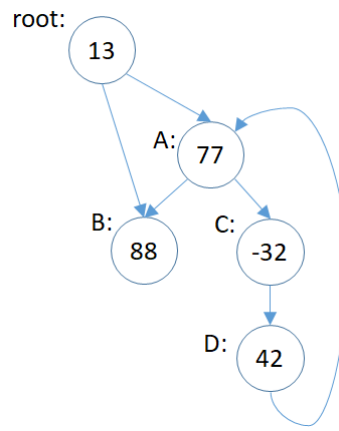


```
In [ ]: 1 graph1 = {'root': [13, ['A', 'B']],
2                 'A': [77, ['B', 'C']],
3                 'B': [88, []],
4                 'C': [-32, ['D']],
5                 'D': [42, []]}
```

```
In [ ]: 1 @instrument
2 def graph_max(graph):
3     """Walk a graph, returning the maximum value in a (non-empty) graph.
4     First, we'll assume there are no cycles in the graph.
5     """
6     @instrument
7     def node_max(node_name):
8         pass
9
10    return node_max('root')
```

```
In [ ]: 1 graph_max(graph1)
```

What do we do if there are cycles in the graph, e.g.



```
In [ ]: 1 graph2 = {'root': [13, ['A', 'B']],
2             'A': [77, ['B', 'C']],
3             'B': [88, []],
4             'C': [-32, ['D']],
5             'D': [42, ['A']]} #changed; now D -> A
```

```
In [ ]: 1 #graph_max(graph2) # breaks! (need to re-execute def graph_max afterwards for instrumentation)
```

```
In [ ]: 1 @instrument
2 def graph_max2(graph):
3     """Walk a graph, returning the maximum value in a (non-empty) graph.
4     Now, however, there might be cycles.
5     """
6     @instrument
7     def node_max(node_name):
8         pass
9
10    return node_max('root')
```

```
In [ ]: 1 graph_max2(graph2)
```

Recursive Lists

It's possible to create a simple python list that has itself as an element. In essence, that means that python lists themselves might be "graphs" and have cycles in them, not just have a tree-like structure!

```
In [ ]: 1 x = [0, 1, 2]
2 x[1] = x
3 print("x:", x)
4 print("x[1][1][1][1][1][1][1][1][1][1][2]:", x[1][1][1][1][1][1][1][1][1][1][2])
```

We'd like a version of `deep_copy` that could create a (separate standalone) copy of a recursive list, *with the same* structural sharing (including any cycles that might exist!) as in the original recursive list.

```
In [ ]: 1 @instrument
2 def deep_copy(old, copies=None):
3     if copies is None:
4         copies = {}
5
6     oid = id(old)      #get the unique python object-id for old
7
8     if oid in copies: #base case: already copied object, just return it
9         return copies[oid]
10
11    if not isinstance(old, list): #base case: not a list, remember & return it
12        copies[oid] = old
13        return copies[oid]
14
15    #recursive case
16    copies[oid] = []
17    for e in old:
18        copies[oid].append(deep_copy(e, copies))
19    return copies[oid]
```

```
In [ ]: 1 y = deep_copy(x)
2 y[0] = 'zero'
3 print("x:", x)
4 print("y:", y)
5 print("y[1][1][1][1][1][1][1][1][1][1][2]:", y[1][1][1][1][1][1][1][1][1][1][2])
```