Advanced Functions

We can write a function that calls another function, even itself. When a function calls itself, this is referred to as *recursion*:

In [13]:

```
def recursive_adder(first, *rest):
    print 'Call recursive_adder(%r, *%r)' % (first, rest)
    if rest:
        return first + recursive_adder(*rest)
    else:
        return first
recursive_adder(1,2,3)
```

```
Call recursive_adder(1, *(2, 3))
Call recursive_adder(2, *(3,))
Call recursive_adder(3, *())
```

Out[13]:

```
6
```

Functions are just regular Python objects (they are so-called *first class* functions). This means that they can be passed as arguments to other functions or assigned variable names:

In [14]:

```
def doubler(a):
    return a * 2

def my_map(mapf, sequence):
    result = []
    for item in sequence:
        result.append(mapf(item))
    return result

my_map(doubler, [1,2,3])
```

Out[14]:

```
[2, 4, 6]
```

As seen above, first class functions can be used to traverse data structures. Another common data structure is a tree. We can implement tree traversal functions to visit each node:

In [28]:

```
# Store the tree as nodes of (value, left, right)

mytree = ('root',
          ('child-L', (), ()),
          ('child-R',
           ('child-RL', (), ()),
           ('child-RR', (), ())))

def preorder_tree_map(function, node, level=0):
    value, left, right = node
    result = [function(level, value)]
    if left:
        result += preorder_tree_map(function, left, level+1)
    if right:
        result += preorder_tree_map(function, right, level+1)
    return result

def print_node(level, value):
    print ('    ' * level) + repr(value)
    return value

preorder_tree_map(print_node, mytree)
```

```
'root'
    'child-L'
    'child-R'
        'child-RL'
        'child-RR'
```

Out[28]:

```
['root', 'child-L', 'child-R', 'child-RL', 'child-RR']
```

In [29]:

```
def inorder_tree_map(function, node, level=0):
    value, left, right = node
    result = []
    if left:
        result += inorder_tree_map(function, left, level+1)
    result.append(function(level, value))
    if right:
        result += inorder_tree_map(function, right, level+1)
    return result

inorder_tree_map(print_node, mytree)
```

```
    'child-L'
'root'
        'child-RL'
    'child-R'
        'child-RR'
```

Out[29]:

```
['child-L', 'root', 'child-RL', 'child-R', 'child-RR']
```

Closures and lexical scoping

In [1]:
```
def make_adder(value):
    def adder(other_value):
        return value + other_value
    return adder

add5 = make_adder(5)
print add5(10)

add2 = make_adder(2)
print add2(10)
```

```
15
12
```

Exercise

- Write a function that traverses the tree above in *post*-order (recursing to the left and right children *before* running the function on the node's value itself.
- Write a version of `filter(function, sequence)` that returns the values in a sequence for which `function(item)` evaluates to `True`