

## CSC 310 - Imperative Programming Languages, Spring, 2010, Dr. Dale E. Parson

Handout showing direct recursion, tail (direct) recursion, and into to Python objects and classes.

### objects\_minimake/example\_reduce.py

```
1 # example_reduce.py Examples of direct- and tail-recursive Python
2 # reduce functions. D. Parson, CSC 310, Spring 2010.
3
4 def direct_reduce(function, valuelist):
5     # Accumulate function applied to valuelist in left-to-right order.
6     # Assume valuelist has at least 1 element.
7     # Derek Columbus gets credit for this from CSC 237.
8     if (len(valuelist) == 1): # base case
9         return valuelist[0]
10    else:
11        # Reduce to the left side first, then reduce to the right as the
12        # stack unwinds. This function is not tail recursive because it
13        # does more than simply return the result of recursive calls.
14        return function(direct_reduce(function, valuelist[:-1]), valuelist[-1])
15
16 def __test_sub__(x, y): return x - y # a subtraction function
17 ll = [1, 2, 3, 4]
18
19 print "direct_reduce of [1, 2, 3, 4] with subtraction", \
20     direct_reduce(__test_sub__, ll)
21
22 def tail_reduce(function, valuelist):
23     # Parson's tail-recursive equivalent to direct_reduce.
24     # This is still direct recursion, but it is also tail recursion
25     # because the recursive call(s) are followed immediately by
26     # returning from the function. A compiler can compile tail recursion
27     # as re-assignment to parameters followed by a jump (goto) the start
28     # of the function without making a tail recursive call.
29     def help_reduce(function, accumulator, valuelist):
30         # Pretend label "start:" is here.
31         if (valuelist):
32             return help_reduce(function, function(accumulator, valuelist[0]),
33                               valuelist[1:])
34         # Compiler can compile away a tail call by re-assigning params.
35         # function = function It could skip this step.
36         # accumulator = function(accumulator, valuelist[0])
37         # valuelist = valuelist[1:]
38         # GOTO start
39     else:
40         return accumulator
```

```
41 # Make the initial, non-recursive call:
42 return help_reduce(function, valuelist[0], valuelist[1:])
43
44 print "tail_reduce of [1, 2, 3, 4] with subtraction", \
45     tail_reduce(__test_sub__, ll)
46
47 # A lambda expression is an anonymous function with the parameters
48 # to the left of the colon and the code to the right of the colon.
49 # It must be an expression (which results in a value), not a statement.
50 print "tail_reduce of [1, 2, 3, 4] with multiplication", \
51     tail_reduce(lambda x, y : x * y, ll)
```

**-bash-3.00\$ python example\_reduce.py**

**direct\_reduce of [1, 2, 3, 4] with subtraction -8**

**tail\_reduce of [1, 2, 3, 4] with subtraction -8**

**tail\_reduce of [1, 2, 3, 4] with multiplication 24**

### objects\_minimake/example\_classes.py

```
1 # example_classes.py Examples of some Python class definitions with
2 # inheritance. D. Parson, CSC 310, Spring 2010.
3
4 class binaryop(object): # object is the root base class in Python
5     def __init__(self, fieldval): # __init__ is the constructor
6         # self is equivalent to C++ or Java "this" pointer.
7         # You must declare it as the first parameter to methods and use it.
8         self.myvalue = fieldval # Copy parameter into a field.
9     def op(self, value):
10        # Here is how I define an "abstract" method for an abstract class
11        # or interface. This is a coding convention. C++ and Java use
12        # static type checking to make sure that you can only construct
13        # concrete classes that implement all methods. Python does not
14        # perform any compile-time checks for undefined methods, so you
15        # have to do it with run-time code.
16        raise AttributeError, "op undefined in abstract case class binaryop"
17
18 class binaryadd(binaryop): # add constructor fieldval and op value param.
19     def __init__(self, fieldval):
20         super(binaryadd, self).__init__(fieldval) # base class constructor.
21     def op(self, value):
22         return self.myvalue + value
23
24 class binarymul(binaryop): # multiply constructor fieldval and op value.
25     def __init__(self, fieldval):
26         super(binarymul, self).__init__(fieldval) # base class constructor.
27     def op(self, value):
28         return self.myvalue * value
29
30
31 obj1 = binaryadd(3) # call the constructor
32 print "Type of a binaryadd(3) object is", type(obj1)
33 print "Adding 7 to that yields", obj1.op(7)
34 print
35
36 obj2 = binarymul(5) # call the constructor
37 print "Type of a binarymul(5) object is", type(obj2)
38 print "Adding 7 to that yields", obj2.op(7)
39 print
40
41 print "Type of binaryadd.op is ", type(binaryadd.op) # unbound to object
42 print "Type of obj1.op is ", type(obj1.op) # bound to object
43
44 # unboundMethod binds a reference to the method (function), but does not
```

```
45 # bind a reference to a "self" object such as obj1. To use an unbound
46 # method reference you must bind the self parameter when making the call.
47 unboundMethod = binaryadd.op # Binds function but not the self parameter.
48 print "Adding 4 via an unbound method", unboundMethod(obj1, 4)
49
50 boundMethod = obj1.op # Binds obj1 as the self parameter.
51 print "Adding 6 via an unbound method", boundMethod(6)
52
53 # This last example is how you will form op codes for assignment 5.
54 # An opcode is a reference to a *bound* method, which consists of
55 # binding both a method's code (function) and its data (self reference).
56 # The textbook calls this an object closure.
```

```
-bash-3.00$ python example_classes.py
```

```
Type of a binaryadd(3) object is <class '__main__.binaryadd'>  
Adding 7 to that yields 10
```

```
Type of a binarymul(5) object is <class '__main__.binarymul'>  
Adding 7 to that yields 35
```

```
Type of binaryadd.op is <type 'instancemethod'>  
Type of obj1.op is <type 'instancemethod'>  
Adding 4 via an unbound method 7  
Adding 6 via an unbound method 9
```