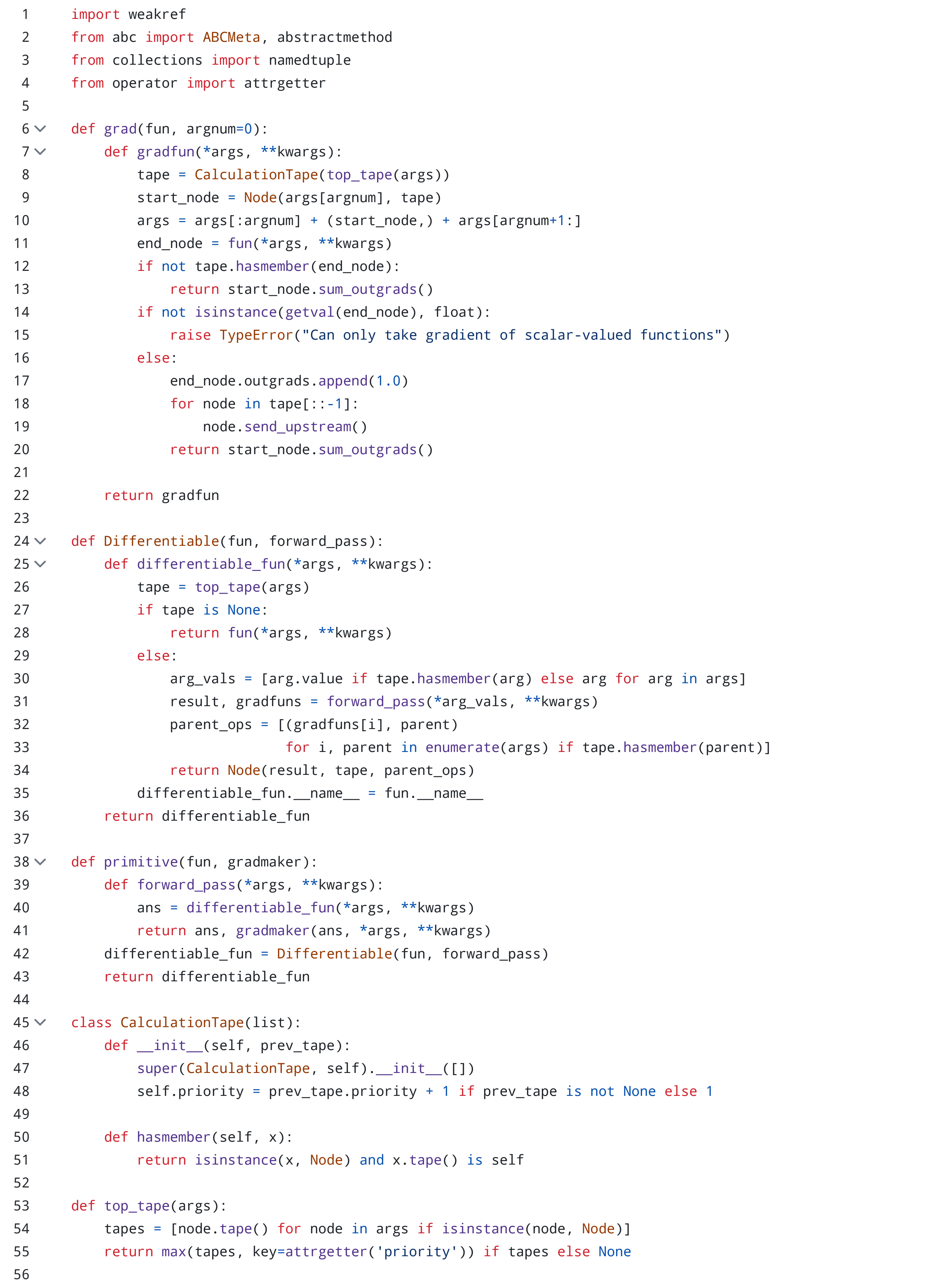

```python
import weakref
from abc import ABCMeta, abstractmethod
from collections import namedtuple
from operator import attrgetter

def grad(fun, argnum=0):
    def gradfun(*args, **kwargs):
        tape = CalculationTape(top_tape(args))
        start_node = Node(args[argnum], tape)
        args = args[:argnum] + (start_node,) + args[argnum+1:]
        end_node = fun(*args, **kwargs)
        if not tape.hasmember(end_node):
            return start_node.sum_outgrads()
        if not isinstance(getval(end_node), float):
            raise TypeError("Can only take gradient of scalar-valued functions")
        else:
            end_node.outgrads.append(1.0)
            for node in tape[::-1]:
                node.send_upstream()
            return start_node.sum_outgrads()

    return gradfun

def Differentiable(fun, forward_pass):
    def differentiable_fun(*args, **kwargs):
        tape = top_tape(args)
        if tape is None:
            return fun(*args, **kwargs)
        else:
            arg_vals = [arg.value if tape.hasmember(arg) else arg for arg in args]
            result, gradfuns = forward_pass(*arg_vals, **kwargs)
            parent_ops = [(gradfuns[i], parent)
                          for i, parent in enumerate(args) if tape.hasmember(parent)]
            return Node(result, tape, parent_ops)
        differentiable_fun.__name__ = fun.__name__
    return differentiable_fun

def primitive(fun, gradmaker):
    def forward_pass(*args, **kwargs):
        ans = differentiable_fun(*args, **kwargs)
        return ans, gradmaker(ans, *args, **kwargs)
    differentiable_fun = Differentiable(fun, forward_pass)
    return differentiable_fun

class CalculationTape(list):
    def __init__(self, prev_tape):
        super(CalculationTape, self).__init__([])
        self.priority = prev_tape.priority + 1 if prev_tape is not None else 1

    def hasmember(self, x):
        return isinstance(x, Node) and x.tape() is self

def top_tape(args):
    tapes = [node.tape() for node in args if isinstance(node, Node)]
    return max(tapes, key=attrgetter('priority')) if tapes else None

```

```python
class Node(object):
    __slots__ = ['value', 'tape', 'parent_ops', 'outgrads']
    __metaclass__ = ABCMeta
    def __new__(cls, value, *args, **kwargs):
        try:
            node_type = node_types.type_mappings[type(value)]
            return super(Node, cls).__new__(node_type, value, *args, **kwargs)
        except KeyError:
            raise TypeError("Can't differentiate wrt {0}".format(type(value)))

    def __init__(self, value, tape, parent_ops=[]):
        self.value = value
        self.tape = weakref.ref(tape)
        tape.append(self)
        self.parent_ops = parent_ops
        self.outgrads = []

    def send_upstream(self):
        if self.outgrads:
            outgrad_sum = self.sum_outgrads()
            for gradfun, parent in self.parent_ops:
                parent.outgrads.append(gradfun(outgrad_sum))

    def sum_outgrads(self):
        if len(self.outgrads) is 1 and not isinstance(getval(self.outgrads[0]), Setter):

            return self.outgrads[0]
        else:
            outgrad_sum = self.zeros()
            for new in self.outgrads:
                outgrad_sum = mutating_add(outgrad_sum, new)
            return outgrad_sum

    def __getitem__(self, idx):
        return take(self, idx)

    @abstractmethod
    def zeros(self):
        pass

def getval(x):
    return getval(x.value) if isinstance(x, Node) else x

def zeros_like(x):
    return Node(x, CalculationTape(None)).zeros()

Setter = namedtuple('Setter', ('idx', 'val'))

import node_types # Can only import after defining Node and Setter

def mutating_add(old, new):
    if isinstance(new, Setter):
        if old[new.idx] is 0:
            old[new.idx] = new.val
        else:
            old[new.idx] += new.val
    else:
        old += new
    return old
mutating_add = primitive(mutating_add, lambda ans, old, new: [lambda g : g] * 2)

def take(A, idx): return A[idx]
take = primitive(take, lambda ans, A, idx : [lambda g : untake(g, idx)])

def untake(x, idx): return Setter(idx, x)
untake = primitive(untake, lambda ans, x, idx : [lambda g : take(g, idx)])
```