```
def grad(fun, argnum=0):
    """
    Returns a function which computes the gradient of `fun` with respect to
    positional argument number `argnum`. The returned function takes the same
    arguments as `fun`, but returns the gradient instead. The gradient has
    the same type as the argument."""
    def gradfun(*args,**kwargs):
        return backward_pass(*forward_pass(fun,args,kwargs,argnum))

    try:
        gradfun.__name__     =     "grad_{fun}_wrt_argnum_{argnum}".format(fun=fun.__name__,
argnum=argnum)
        gradfun.__doc__ = "Gradient of function {fun} with respect to argument number {argnum}.
" \
                          "Has the same arguments as {fun} but the return value has type of" \
                          "argument {argnum}".format(fun=fun.__name__, argnum=argnum)
    except:
        pass

    return gradfun

def forward_pass(fun, args, kwargs, argnum=0):
        tape = CalculationTape()
        arg_wrt = args[argnum]
        start_node = new_node(safe_type(getval(arg_wrt)), [tape])
        args = list(args)
        args[argnum] = merge_tapes(start_node, arg_wrt)
        end_node = fun(*args, **kwargs)
        return start_node, end_node, tape

def backward_pass(start_node, end_node, tape):
    if not isinstance(end_node, Node) or tape not in end_node.tapes:
        warnings.warn("Output seems independent of input. Returning zero gradient.")
        return zeros_like(start_node)
    if not type(end_node) is FloatNode:
        try:
            end_node = FloatNode.cast(end_node, 1.0)
        except TypeError:
            raise TypeError("Output type {0} can't be cast to float. ".format(type(end_node.value))
                            + "Function grad requires a scalar-valued function. "
                              "Try jacobian or elementwise_grad.")
```

```
    end_node.tapes[tape].outgrads = [1.0]
    tape.complete = True
    while tape:
        node = tape.pop()
        if node.outgrads:
            cur_outgrad = node.sum_outgrads()
            assert type(new_node(getval(cur_outgrad))) == node.node_type, \
                "Types are {0} and {1}".format(type(new_node(getval(cur_outgrad))),
node.node_type)
            for gradfun, parent in node.parent_grad_ops:
                og = cast_to_node_type(gradfun(cur_outgrad), parent.node_type,
parent.node_value)
                parent.outgrads.append(og)
    return cur_outgrad

def cast_to_node_type(x, node_type, example):
    if type(new_node(getval(x))) is not node_type:
        return node_type.cast(x, example)
    else:
        return x

class primitive(object):
    """
    Wraps a function so that its gradient can be specified and its invocation
    can be recorded. For examples, see the docs."""
    def __init__(self, fun):
        self.fun = fun
        self.grads = {}
        self.zero_grads = set()
        self.__name__ = fun.__name__
        self.__doc__ = fun.__doc__

    def gradmaker(self, argnum, ans, args, kwargs):
        try:
            return self.grads[argnum](ans, *args, **kwargs)
        except KeyError:
            if self.grads == {}:
                raise NotImplementedError("Gradient of {0} not yet implemented."
                                          .format(self.fun, argnum))
            raise NotImplementedError("Gradient of {0} w.r.t. arg number {1} not yet
implemented."
```

```
                                        .format(self.fun, argnum))

def defgrad(self, gradmaker, argnum=0):
    self.grads[argnum] = gradmaker

def defgrads(self, gradmaker, argnums):
    for argnum in argnums:
        self.defgrad(functools.partial(gradmaker, argnum), argnum)

def defgrad_is_zero(self, argnums=(0,)):
    for argnum in argnums:
        self.zero_grads.add(argnum)

def __call__(self, *args, **kwargs):
    argvals = list(args)
    ops = []
    tapes = set()
    for i, arg in enumerate(args):
        if isinstance(arg, Node):
            argvals[i] = arg.value
            if i in self.zero_grads: continue
            for tape, parent_rnode in six.iteritems(arg.tapes):
                if not tape.complete:
                    ops.append((tape, i, parent_rnode))
                    tapes.add(tape)

    result = self.fun(*argvals, **kwargs)
    if result is NotImplemented: return result
    if ops:
        result = new_node(result, tapes)
        for tape, argnum, parent in ops:
            gradfun = self.gradmaker(argnum, result, args, kwargs)
            rnode = result.tapes[tape]
            rnode.parent_grad_ops.append((gradfun, parent))
    return result

if six.PY3:
    def __get__(self, obj, objtype):
        return types.MethodType(self, obj)
else:
    def __get__(self, obj, objtype):
```

```
            return types.MethodType(self, obj, objtype)

@primitive
def merge_tapes(x, y): return x
merge_tapes.defgrad(lambda ans, x, y : lambda g : g)
merge_tapes.defgrad(lambda ans, x, y : lambda g : g, argnum=1)

def new_node(value, tapes=[]):
    try:
        return Node.type_mappings[type(value)](value, tapes)
    except KeyError:
        raise TypeError("Can't differentiate wrt {0}".format(type(value)))

def zeros_like(value):
    if isinstance(value, Node):
        return value.zeros_like(value)
    else:
        return new_node(value, []).zeros_like(value)

class ReverseNode(object):
    __slots__ = ['parent_grad_ops', 'outgrads', 'node_type', 'node_value']
    def __init__(self, node_type, node_value):
        self.parent_grad_ops = []
        self.outgrads = []
        self.node_type = node_type
        self.node_value = node_value

    def sum_outgrads(self):
        return self.node_type.sum_outgrads(self.outgrads)

class Node(object):
    __slots__ = ['value', 'tapes']
    type_mappings = {}
    def __init__(self, value, tapes):
        self.value = value
        self.tapes = {}
        for tape in tapes:
            new_rnode = ReverseNode(type(self), value)
            tape.append(new_rnode)
            self.tapes[tape] = new_rnode
```

```
    @staticmethod
    def sum_outgrads(outgrads):
        return sum(outgrads[1:], outgrads[0])

    def __str__(self):
        return "Autograd {0} with value {1} and {2} tape(s)".format(
            type(self).__name__, str(self.value), len(self.tapes))

@primitive
def cast(value, caster):
    return caster(value)
cast.defgrad(lambda *args: I)

getval = lambda x : x.value if isinstance(x, Node) else x

class CalculationTape(list):
    def __init__(self):
        self.complete = False

    def __hash__(self):
        return id(self)

class FloatNode(Node):
    __slots__ = []
    @staticmethod
    def zeros_like(value):
        return 0.0
    @staticmethod
    def cast(value, example):
        return cast(value, cast_to_float)

Node.type_mappings[float] = FloatNode

def cast_to_float(x):
    if np.iscomplexobj(x):
        x = np.real(x)
    return float(x)

class ComplexNode(FloatNode):
    @staticmethod
    def zeros_like(value):
```

```
            return 0.0 + 0.0j
        @staticmethod
        def cast(value, example):
            return cast(value, cast_to_complex)

def cast_to_complex(value):
        if isinstance(value, np.ndarray):
            return complex(value[()])
        else:
            return complex(value)
Node.type_mappings[complex] = ComplexNode

def safe_type(value):
        if isinstance(value, int):
            warnings.warn("Casting int to float to handle differentiation.")
            return float(value)
        else:
            return value

if six.PY3:
        DIV = '__truediv__'
        RDIV = '__rtruediv__'
else:
        DIV = '__div__'
        RDIV = '__rdiv__'

differentiable_ops = ['__add__', '__sub__', '__mul__', '__pow__', '__mod__',
                                    '__neg__', '__radd__', '__rsub__', '__rmul__', '__rpow__',
                                    '__rmod__', DIV, RDIV]

nondifferentiable_ops = ['__eq__', '__ne__', '__gt__', '__ge__', '__lt__', '__le__',]
for float_op in differentiable_ops + nondifferentiable_ops:
        setattr(FloatNode, float_op, primitive(getattr(float, float_op)))

FloatNode.__dict__['__neg__'].defgrad(lambda ans, x : op.neg)




for comp_op in nondifferentiable_ops:
        FloatNode.__dict__[comp_op].defgrad_is_zero(argnums=(0, 1))
```

```
# These functions will get clobbered when autograd.numpy is imported.
# They're here to allow the use of autograd without numpy.
I = lambda g: g
FloatNode.__dict__['__add__'].defgrad(lambda ans, x, y : I)
FloatNode.__dict__['__add__'].defgrad(lambda ans, x, y : I, argnum=1)
FloatNode.__dict__['__mul__'].defgrad(lambda ans, x, y : lambda g : y * g)
FloatNode.__dict__['__mul__'].defgrad(lambda ans, x, y : lambda g : x * g, argnum=1)
FloatNode.__dict__['__sub__'].defgrad(lambda ans, x, y : I)
FloatNode.__dict__['__sub__'].defgrad(lambda ans, x, y : op.neg, argnum=1)
FloatNode.__dict__[DIV].defgrad(lambda ans, x, y : lambda g : g / y)
FloatNode.__dict__[DIV].defgrad(lambda ans, x, y : lambda g : - g * x / y**2, argnum=1)
FloatNode.__dict__['__pow__'].defgrad(lambda ans, x, y : lambda g : g * y * x ** (y - 1))
FloatNode.__dict__['__pow__'].defgrad(lambda ans, x, y : lambda g : g * log(x) * x ** y, argnum=1)
FloatNode.__dict__['__mod__'].defgrad(lambda ans, x, y : I)
FloatNode.__dict__['__mod__'].defgrad(lambda ans, x, y : lambda g : -g * floor(x/y), argnum=1)

log = primitive(math.log)
log.defgrad(lambda ans, x : lambda g : g / x)
floor = primitive(math.floor)
floor.defgrad_is_zero()

def swap_args(grads):
    grad_0, grad_1 = grads[1], grads[0]
    return {0 : lambda ans, y, x : grad_0(ans, x, y),
            1 : lambda ans, y, x : grad_1(ans, x, y)}

FloatNode.__dict__['__radd__'].grads = swap_args(FloatNode.__dict__['__add__'].grads)
FloatNode.__dict__['__rmul__'].grads = swap_args(FloatNode.__dict__['__mul__'].grads)
FloatNode.__dict__['__rsub__'].grads = swap_args(FloatNode.__dict__['__sub__'].grads)
FloatNode.__dict__[RDIV].grads = swap_args(FloatNode.__dict__[DIV].grads)
FloatNode.__dict__['__rpow__'].grads = swap_args(FloatNode.__dict__['__pow__'].grads)
FloatNode.__dict__['__rmod__'].grads = swap_args(FloatNode.__dict__['__mod__'].grads)
```