```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

from six.moves import builtins

import six
import numpy as onp

from .. import core
from ..abstract_arrays import UnshapedArray, ShapedArray, ConcreteArray
from ..interpreters.xla import DeviceArray
from ..lib import xla_bridge
import jax.lax as lax

# To provide the same module-level names as Numpy, we need to redefine builtins
# and also use some common names (like 'shape' and 'dtype') at the top-level.
# pylint: disable=redefined-builtin,redefined-outer-name

# There might be a pylint bug with tuple unpacking.
# pylint: disable=unbalanced-tuple-unpacking

# We get docstrings from the underlying numpy functions.
# pylint: disable=missing-docstring


# We replace some builtin names to follow Numpy's API, so we capture here.
_all = builtins.all
_any = builtins.any
_max = builtins.max
_min = builtins.min
_sum = builtins.sum

# We need some numpy scalars
# TODO(mattjj): handle constants in an indirected, less explicit way?
pi = onp.pi
e = onp.e
inf = onp.inf
nan = onp.nan

# We want isinstance(x, np.ndarray) checks in user code to work with the our
# array-like types, including DeviceArray and UnshapedArray (i.e. the abstract
# array base class). We can override the isinstance behavior directly, without
# having the complexity of multiple inheritance on those classes, by defining
# the ndarray class to have a metaclass with special __instancecheck__ behavior.
_arraylike_types = (onp.ndarray, UnshapedArray, DeviceArray)

class _ArrayMeta(type(onp.ndarray)):
    """Metaclass for overriding ndarray isinstance checks."""

    def __instancecheck__(self, instance):
        try:
            return isinstance(instance.aval, _arraylike_types)
        except AttributeError:
            return isinstance(instance, _arraylike_types)

# pylint: disable=invalid-name
class ndarray(six.with_metaclass(_ArrayMeta, onp.ndarray)):
    pass
# pylint: enable=invalid-name


isscalar = onp.isscalar
iscomplexobj = onp.iscomplexobj
result_type = onp.result_type
shape = _shape = onp.shape
ndim = _ndim = onp.ndim
size = onp.size
_dtype = lax._dtype

uint32 = onp.uint32
int32 = onp.int32
uint64 = onp.uint64
int64 = onp.int64
float32 = onp.float32
float64 = onp.float64
complex64 = onp.complex64


### utility functions
```

```
def _promote_shapes(*args):
    """Prepend implicit leading singleton dimensions for Numpy broadcasting."""
    if len(args) < 2:
        return args
    else:
        shapes = [shape(arg) for arg in args]
        nd = len(_broadcast_shapes(*shapes))
        return [lax.reshape(arg, (1,) * (nd - len(shp)) + shp)
                        if len(shp) != nd else arg for arg, shp in zip(args, shapes)]


def _broadcast_shapes(*shapes):
    """Apply Numpy broadcasting rules to the given shapes."""
    if len(shapes) == 1:
        return shapes[0]
    ndim = _max(len(shape) for shape in shapes)
    shapes = onp.array([(1,) * (ndim - len(shape)) + shape for shape in shapes])
    result_shape = onp.max(shapes, axis=0)
    if not onp.all((shapes == result_shape) | (shapes == 1)):
        raise ValueError("Incompatible shapes for broadcasting: {}"
                                          .format(tuple(map(tuple, shapes))))
    return tuple(result_shape)


def _promote_dtypes(*args):
    """Convenience function to apply Numpy argument dtype promotion."""
    if len(args) < 2:
        return args
    else:
        all_scalar = _all(isscalar(x) for x in args)
        some_bools = _any(_dtype(x) == onp.dtype("bool") for x in args)
        keep_all = all_scalar or some_bools
        from_dtypes = (_dtype(x) for x in args if keep_all or not isscalar(x))
        to_dtype = xla_bridge.canonicalize_dtype(result_type(*from_dtypes))
        return [lax.convert_element_type(x, to_dtype)
                        if _dtype(x) != to_dtype else x for x in args]


def _promote_to_result_dtype(op, *args):
    """Convenience function to promote args directly to the op's result dtype."""
    to_dtype = _result_dtype(op, *args)
    return [lax.convert_element_type(arg, to_dtype) for arg in args]



def _result_dtype(op, *args):
    """Compute result dtype of applying op to arguments with given dtypes."""
    args = (onp.ones((0,) * ndim(arg), _dtype(arg)) for arg in args)
    return _dtype(op(*args))



def _check_arraylike(fun_name, *args):
    """Check if all args fit JAX's definition of arraylike (ndarray or scalar)."""
    not_array = lambda x: not isinstance(x, ndarray) and not onp.isscalar(x)
    if _any(not_array(arg) for arg in args):
        pos, arg = next((i, arg) for i, arg in enumerate(args) if not_array(arg))
        msg = "{} requires ndarray or scalar arguments, got {} at position {}."
        raise TypeError(msg.format(fun_name, type(arg), pos))



def _promote_args(fun_name, *args):
    """Convenience function to apply Numpy argument shape and dtype promotion."""
    _check_arraylike(fun_name, *args)
    return _promote_shapes(*_promote_dtypes(*args))



def _promote_args_like(op, *args):
    """Convenience function to apply shape and dtype promotion to result type."""
    _check_arraylike(op.__name__, *args)
    return _promote_shapes(*_promote_to_result_dtype(op, *args))



def _constant_like(x, const):
    return onp.array(const, dtype=_dtype(x))



def _wraps(fun):
    """Like functools.wraps but works with numpy.ufuncs."""
    docstr = """
    LAX-backed implementation of {fun}. Original docstring below.

    {np_doc}
    """.format(fun=fun.__name__, np_doc=fun.__doc__)
```

```
  def wrap(op):
    try:
      op.__name__ = fun.__name__
      op.__doc__ = docstr
    finally:
      return op
  return wrap


### implementations of numpy functions in terms of lax


def _one_to_one_op(numpy_fn, lax_fn, promote_to_result_dtype=False):
  if promote_to_result_dtype:
    promoted_lax_fn = lambda *args: lax_fn(*_promote_args_like(numpy_fn, *args))
  else:
    name = numpy_fn.__name__
    promoted_lax_fn = lambda *args: lax_fn(*_promote_args(name, *args))
  return _wraps(numpy_fn)(promoted_lax_fn)

absolute = abs = _one_to_one_op(onp.absolute, lax.abs)
add = _one_to_one_op(onp.add, lax.add)
bitwise_and = _one_to_one_op(onp.bitwise_and, lax.bitwise_and)
bitwise_not = _one_to_one_op(onp.bitwise_not, lax.bitwise_not)
bitwise_or = _one_to_one_op(onp.bitwise_or, lax.bitwise_or)
bitwise_xor = _one_to_one_op(onp.bitwise_xor, lax.bitwise_xor)
right_shift = _one_to_one_op(onp.right_shift, lax.shift_right_arithmetic)
left_shift = _one_to_one_op(onp.left_shift, lax.shift_left)
ceil = _one_to_one_op(onp.ceil, lax.ceil)
equal = _one_to_one_op(onp.equal, lax.eq)
expm1 = _one_to_one_op(onp.expm1, lax.expm1, True)
exp = _one_to_one_op(onp.exp, lax.exp, True)
floor = _one_to_one_op(onp.floor, lax.floor)
greater_equal = _one_to_one_op(onp.greater_equal, lax.ge)
greater = _one_to_one_op(onp.greater, lax.gt)
isfinite = _one_to_one_op(onp.isfinite, lax.is_finite)
less_equal = _one_to_one_op(onp.less_equal, lax.le)
less = _one_to_one_op(onp.less, lax.lt)
log1p = _one_to_one_op(onp.log1p, lax.log1p, True)
log = _one_to_one_op(onp.log, lax.log, True)
maximum = _one_to_one_op(onp.maximum, lax.max)
```

```
minimum = _one_to_one_op(onp.minimum, lax.min)
multiply = _one_to_one_op(onp.multiply, lax.mul)
negative = _one_to_one_op(onp.negative, lax.neg)
not_equal = _one_to_one_op(onp.not_equal, lax.ne)
power = _one_to_one_op(onp.power, lax.pow, True)
sign = _one_to_one_op(onp.sign, lax.sign)
subtract = _one_to_one_op(onp.subtract, lax.sub)
tanh = _one_to_one_op(onp.tanh, lax.tanh, True)
sort = _one_to_one_op(onp.sort, lax.sort)


def _logical_op(np_op, bitwise_op):
    @_wraps(np_op)
    def op(*args):
        zero = lambda x: lax.full_like(x, shape=(), fill_value=0)
        args = (x if onp.issubdtype(_dtype(x), onp.bool_) else lax.ne(x, zero(x))
                for x in args)
        return bitwise_op(*_promote_args(np_op.__name__, *args))
    return op

logical_and = _logical_op(onp.logical_and, lax.bitwise_and)
logical_not = _logical_op(onp.logical_not, lax.bitwise_not)
logical_or = _logical_op(onp.logical_or, lax.bitwise_or)
logical_xor = _logical_op(onp.logical_xor, lax.bitwise_xor)


@_wraps(onp.true_divide)
def true_divide(x1, x2):
    x1, x2 = _promote_shapes(x1, x2)
    result_dtype = _result_dtype(onp.true_divide, x1, x2)
    return lax.div(lax.convert_element_type(x1, result_dtype),
                   lax.convert_element_type(x2, result_dtype))


@_wraps(onp.divide)
def divide(x1, x2):
    # decide whether to perform integer division based on Numpy result dtype, as a
    # way to check whether Python 3 style division is active in Numpy
    result_dtype = _result_dtype(onp.divide, x1, x2)
    if onp.issubdtype(result_dtype, onp.integer):
        return floor_divide(x1, x2)
```

```
  else:
    return true_divide(x1, x2)


@_wraps(onp.floor_divide)
def floor_divide(x1, x2):
  x1, x2 = _promote_args("floor_divide", x1, x2)
  if onp.issubdtype(_dtype(x1), onp.integer):
    quotient = lax.div(x1, x2)
    select = logical_and(lax.sign(x1) != lax.sign(x2), lax.rem(x1, x2) != 0)
    # TODO(mattjj): investigate why subtracting a scalar was causing promotion
    return where(select, quotient - onp.array(1, _dtype(quotient)), quotient)
  else:
    return _float_divmod(x1, x2)[0]


@_wraps(onp.divmod)
def divmod(x1, x2):
  x1, x2 = _promote_args("divmod", x1, x2)
  if onp.issubdtype(_dtype(x1), onp.integer):
    return floor_divide(x1, x2), remainder(x1, x2)
  else:
    return _float_divmod(x1, x2)


def _float_divmod(x1, x2):
  # see float_divmod in floatobject.c of CPython
  mod = lax.rem(x1, x2)
  div = lax.div(lax.sub(x1, mod), x2)

  ind = lax.bitwise_and(mod != 0, lax.sign(x2) != lax.sign(mod))
  mod = lax.select(ind, mod + x1, mod)
  div = lax.select(ind, div - _constant_like(div, 1), div)

  return lax.round(div), mod


def logaddexp(x1, x2):
  x1, x2 = _promote_to_result_dtype(onp.logaddexp, *_promote_shapes(x1, x2))
  amax = lax.max(x1, x2)
  return lax.add(amax, lax.log(lax.add(lax.exp(lax.sub(x1, amax)),
                               lax.exp(lax.sub(x2, amax)))))


@_wraps(onp.remainder)
def remainder(x1, x2):
    x1, x2 = _promote_args("remainder", x1, x2)
    return lax.rem(lax.add(lax.rem(x1, x2), x2), x2)
mod = remainder
fmod = lax.rem


def sqrt(x):
    x, = _promote_to_result_dtype(onp.sqrt, x)
    return power(x, _constant_like(x, 0.5))


@_wraps(onp.transpose)
def transpose(x, axis=None):
    axis = onp.arange(ndim(x))[::-1] if axis is None else axis
    return lax.transpose(x, axis)


@_wraps(onp.sinh)
def sinh(x):
    x, = _promote_to_result_dtype(onp.sinh, x)
    return lax.div(lax.sub(lax.exp(x), lax.exp(lax.neg(x))), _constant_like(x, 2))


@_wraps(onp.cosh)
def cosh(x):
    x, = _promote_to_result_dtype(onp.cosh, x)
    return lax.div(lax.add(lax.exp(x), lax.exp(lax.neg(x))), _constant_like(x, 2))


@_wraps(onp.sin)
def sin(x):
    x, = _promote_to_result_dtype(onp.sin, x)
    return lax.sin(x)


@_wraps(onp.cos)
```

```
def cos(x):
    x, = _promote_to_result_dtype(onp.sin, x)
    return lax.cos(x)



@_wraps(onp.conjugate)
def conjugate(x):
    return lax.conj(x) if iscomplexobj(x) else x
conj = conjugate



@_wraps(onp.imag)
def imag(x):
    return lax.imag(x) if iscomplexobj(x) else x



@_wraps(onp.real)
def real(x):
    return lax.real(x) if iscomplexobj(x) else x



@_wraps(onp.angle)
def angle(x):
    if iscomplexobj(x):
        return lax.atan2(lax.imag(x), lax.real(x))
    else:
        return zeros_like(x)



@_wraps(onp.reshape)
def reshape(a, newshape, order="C"):    # pylint: disable=missing-docstring
    if order == "C" or order is None:
        dims = None
    elif order == "F":
        dims = onp.arange(ndim(a))[::-1]
    elif order == "A":
        dims = onp.arange(ndim(a))[::-1] if isfortran(a) else onp.arange(ndim(a))
    else:
        raise ValueError("Unexpected value for 'order' argument: {}.".format(order))

    dummy_val = onp.broadcast_to(0, a.shape)    # zero strides
```

```
  computed_newshape = onp.reshape(dummy_val, newshape).shape
  return lax.reshape(a, computed_newshape, dims)


@_wraps(onp.ravel)
def ravel(a, order="C"):
  if order == "K":
    raise NotImplementedError("Ravel not implemented for order='K'.")
  return reshape(a, (size(a),), order)


@_wraps(onp.squeeze)
def squeeze(a, axis=None):
  if 1 not in shape(a):
    return a
  if axis is None:
    newshape = [d for d in shape(a) if d != 1]
  else:
    axis = frozenset(onp.mod(axis, ndim(a)).reshape(-1))
    newshape = [d for i, d in enumerate(shape(a))
                if d != 1 or i not in axis]
  return lax.reshape(a, newshape)


@_wraps(onp.expand_dims)
def expand_dims(a, axis):
  shape = _shape(a)
  axis = axis % (ndim(a) + 1)  # pylint: disable=g-no-augmented-assignment
  return lax.reshape(a, shape[:axis] + (1,) + shape[axis:])


@_wraps(onp.swapaxes)
def swapaxes(a, axis1, axis2):
  perm = onp.arange(ndim(a))
  perm[axis1], perm[axis2] = perm[axis2], perm[axis1]
  return lax.transpose(a, perm)


@_wraps(onp.moveaxis)
def moveaxis(a, source, destination):
  source = onp.mod(source, ndim(a)).reshape(-1)
  destination = onp.mod(destination, ndim(a)).reshape(-1)
  if len(source) != len(destination):
    raise ValueError("Inconsistent number of elements: {} vs {}"
                     .format(len(source), len(destination)))
  perm = [i for i in range(ndim(a)) if i not in source]
  for dest, src in sorted(zip(destination, source)):
    perm.insert(dest, src)
  return lax.transpose(a, perm)


@_wraps(onp.isclose)
def isclose(a, b, rtol=1e-05, atol=1e-08):
  a, b = _promote_args("isclose", a, b)
  rtol = lax.convert_element_type(rtol, _dtype(a))
  atol = lax.convert_element_type(atol, _dtype(a))
  return lax.le(lax.abs(lax.sub(a, b)),
                lax.add(atol, lax.mul(rtol, lax.abs(b))))


@_wraps(onp.where)
def where(condition, x=None, y=None):
  if x is None or y is None:
    raise ValueError("Must use the three-argument form of where().")
  if not onp.issubdtype(_dtype(condition), onp.bool_):
    condition = lax.ne(condition, zeros_like(condition))
  condition, x, y = broadcast_arrays(condition, x, y)
  return lax.select(condition, *_promote_dtypes(x, y))


def broadcast_arrays(*args):
  """Like Numpy's broadcast_arrays but doesn't return views."""
  shapes = [shape(arg) for arg in args]
  if len(set(shapes)) == 1:
    return [arg if isinstance(arg, ndarray) or isscalar(arg) else array(arg)
            for arg in args]
  result_shape = _broadcast_shapes(*shapes)
  return [broadcast_to(arg, result_shape) for arg in args]


def broadcast_to(arr, shape):
  """Like Numpy's broadcast_to but doesn't necessarily return views."""
```

```
    arr = arr if isinstance(arr, ndarray) or isscalar(arr) else array(arr)
    if _shape(arr) != shape:
        # TODO(mattjj): revise this to call lax.broadcast_in_dim rather than
        # lax.broadcast and lax.transpose
        _broadcast_shapes(shape, _shape(arr))    # error checking
        nlead = len(shape) - len(_shape(arr))
        diff, = onp.where(onp.not_equal(shape[nlead:], _shape(arr)))

        new_dims = tuple(range(nlead)) + tuple(nlead + diff)
        kept_dims = tuple(onp.delete(onp.arange(len(shape)), new_dims))
        perm = onp.argsort(new_dims + kept_dims)

        broadcast_dims = onp.take(shape, new_dims)
        squeezed_array = squeeze(arr, diff)
        return lax.transpose(lax.broadcast(squeezed_array, broadcast_dims), perm)
    else:
        return arr


@_wraps(onp.split)
def split(ary, indices_or_sections, axis=0):
    dummy_val = onp.broadcast_to(0, ary.shape)    # zero strides
    subarrays = onp.split(dummy_val, indices_or_sections, axis)    # shapes
    split_indices = onp.cumsum([0] + [onp.shape(sub)[axis] for sub in subarrays])
    starts, ends = [0] * ndim(ary), shape(ary)
    _subval = lambda x, i, v: lax.subvals(x, [(i, v)])
    return [lax.slice(ary, _subval(starts, axis, start), _subval(ends, axis, end))
                    for start, end in zip(split_indices[:-1], split_indices[1:])]


@_wraps(onp.clip)
def clip(a, a_min=None, a_max=None):
    a_min = _dtype_info(_dtype(a)).min if a_min is None else a_min
    a_max = _dtype_info(_dtype(a)).max if a_max is None else a_max
    if _dtype(a_min) != _dtype(a):
        a_min = lax.convert_element_type(a_min, _dtype(a))
    if _dtype(a_max) != _dtype(a):
        a_max = lax.convert_element_type(a_max, _dtype(a))
    return lax.clamp(a_min, a, a_max)
```

```
def _dtype_info(dtype):
  """Helper function for to get dtype info needed for clipping."""
  if onp.issubdtype(dtype, onp.integer):
    return onp.iinfo(dtype)
  return onp.finfo(dtype)


@_wraps(onp.round)
def round(a, decimals=0):
  if onp.issubdtype(_dtype(a), onp.integer):
    return a  # no-op on integer types

  if decimals == 0:
    return lax.round(a)

  factor = _constant_like(a, 10 ** decimals)
  return lax.div(lax.round(lax.mul(a, factor)), factor)
around = round


### Reducers


def _make_reduction(np_fun, op, init_val):
  """Creates reduction function given a binary operation and monoid identity."""

  @_wraps(op)
  def reduction(a, axis=None, dtype=None, out=None, keepdims=False):
    if out is not None:
      raise ValueError("reduction does not support `out` argument.")

    a = a if isinstance(a, ndarray) else asarray(a)
    dims = _reduction_dims(a, axis)
    result_dtype = _dtype(np_fun(onp.ones((), dtype=_dtype(a))))
    if _dtype(a) != result_dtype:
      a = lax.convert_element_type(a, result_dtype)
    result = lax.reduce(a, _reduction_init_val(a, init_val), op, dims)
    if keepdims:
      shape_with_singletons = lax.subvals(shape(a), zip(dims, (1,) * len(dims)))
      result = lax.reshape(result, shape_with_singletons)
    if dtype and onp.dtype(dtype) != onp.dtype(result_dtype):
      result = lax.convert_element_type(result, dtype)
    return result

  return reduction


def _reduction_dims(a, axis):
  if axis is None:
    return onp.arange(ndim(a))
  elif isinstance(axis, (onp.ndarray, tuple, list)):
    return onp.mod(onp.asarray(axis), ndim(a))
  elif isinstance(axis, int):
    return onp.mod([axis], ndim(a))
  else:
    raise TypeError("Unexpected type of axis argument: {}".format(type(axis)))


def _reduction_init_val(a, init_val):
  a_dtype = xla_bridge.canonicalize_dtype(_dtype(a))
  try:
    return onp.array(init_val, dtype=a_dtype)
  except OverflowError:
    assert onp.issubdtype(a_dtype, onp.integer)
    sign, iinfo = onp.sign(init_val), onp.iinfo(a_dtype)
    return onp.array(iinfo.min if sign < 0 else iinfo.max, dtype=a_dtype)


sum = _make_reduction(onp.sum, lax.add, 0)
prod = _make_reduction(onp.prod, lax.mul, 1)
max = _make_reduction(onp.max, lax.max, -onp.inf)
min = _make_reduction(onp.min, lax.min, onp.inf)
all = _make_reduction(onp.all, logical_and, True)
any = _make_reduction(onp.any, logical_or, False)


@_wraps(onp.mean)
def mean(a, axis=None, keepdims=False):
  if axis is None:
    normalizer = size(a)
  else:
    normalizer = onp.prod(onp.take(shape(a), axis))
  if onp.issubdtype(_dtype(a), onp.bool_):
    a = lax.convert_element_type(a, onp.int32)
  return true_divide(sum(a, axis, keepdims=keepdims),
                     _constant_like(a, normalizer))


@_wraps(onp.var)
def var(a, axis=None, keepdims=False, ddof=0):
  if ddof != 0:
    raise NotImplementedError("Only implemented for ddof=0.")
  centered = subtract(a, mean(a, axis, keepdims=True))
  if iscomplexobj(centered):
    centered = lax.abs(centered)
  return mean(lax.mul(centered, centered), axis, keepdims=keepdims)


@_wraps(onp.std)
def std(a, axis=None, keepdims=False, ddof=0):
  return sqrt(var(a, axis, keepdims, ddof))


@_wraps(onp.allclose)
def allclose(a, b, rtol=1e-05, atol=1e-08):
  return all(isclose(a, b, rtol, atol))


### Array-creation functions


arange = onp.arange


@_wraps(onp.stack)
def stack(arrays):
  if not arrays:
    raise ValueError("Need at least one array to stack.")
  new_arrays = [reshape(x, (-1,) + onp.shape(x)) for x in arrays]
  return reshape(concatenate(new_arrays), (len(arrays),) + arrays[0].shape)


@_wraps(onp.concatenate)
```

```
def concatenate(arrays, axis=0):
    if not arrays:
        raise ValueError("Need at least one array to concatenate.")
    return lax.concatenate(_promote_dtypes(*arrays), axis % ndim(arrays[0]))


@_wraps(onp.vstack)
def vstack(tup):
    return concatenate([atleast_2d(m) for m in tup], axis=0)
row_stack = vstack


@_wraps(onp.hstack)
def hstack(tup):
    arrs = [atleast_1d(m) for m in tup]
    if arrs[0].ndim == 1:
        return concatenate(arrs, 0)
    return concatenate(arrs, 1)


@_wraps(onp.column_stack)
def column_stack(tup):
    arrays = []
    for v in tup:
        arr = array(v)
        if arr.ndim < 2:
            arr = arr.reshape((-1, 1))
        arrays.append(arr)
    return concatenate(arrays, 1)


@_wraps(onp.atleast_1d)
def atleast_1d(*arys):
    if len(arys) == 1:
        arr = array(arys[0])
        return arr if arr.ndim >= 1 else arr.reshape(-1)
    else:
        return [atleast_1d(arr) for arr in arys]


@_wraps(onp.atleast_2d)
```

```
def atleast_2d(*arys):
    if len(arys) == 1:
        arr = array(arys[0])
        return arr if arr.ndim >= 2 else arr.reshape((1, -1))
    else:
        return [atleast_2d(arr) for arr in arys]


# TODO(mattjj): can this be simplified?
@_wraps(onp.array)
def array(object, dtype=None, copy=True, order="K", ndmin=0):
    del copy    # Unused.
    if ndmin != 0 or order != "K":
        raise NotImplementedError("Only implemented for order='K', ndmin=0.")

    if isinstance(object, ndarray):
        if dtype and _dtype(object) != dtype:
            return lax.convert_element_type(object, dtype)
        else:
            return object
    elif isinstance(object, (list, tuple)):
        if object:
            subarrays = [expand_dims(array(elt, dtype=dtype), 0) for elt in object]
            return concatenate(subarrays)
        else:
            return onp.array([], dtype)
    elif isscalar(object):
        out = lax.reshape(object, ())
        if dtype and _dtype(out) != dtype:
            return lax.convert_element_type(out, dtype)
        else:
            return out
    else:
        raise TypeError("Unexpected input type for array: {}".format(type(object)))
asarray = array


@_wraps(onp.zeros_like)
def zeros_like(x, dtype=None):
    return zeros(_shape(x), dtype or _dtype(x))
```

```
@_wraps(onp.ones_like)
def ones_like(x, dtype=None):
  return ones(_shape(x), dtype or _dtype(x))


@_wraps(onp.full)
def full(shape, fill_value, dtype=None):
  if dtype:
    fill_value = lax.convert_element_type(fill_value, dtype)
  return lax.broadcast(fill_value, tuple(shape))


@_wraps(onp.zeros)
def zeros(shape, dtype=onp.dtype("float64")):
  shape = (shape,) if onp.isscalar(shape) else shape
  dtype = xla_bridge.canonicalize_dtype(dtype)
  return onp.broadcast_to(onp.zeros((), dtype), tuple(shape))


@_wraps(onp.ones)
def ones(shape, dtype=onp.dtype("float64")):
  shape = (shape,) if onp.isscalar(shape) else shape
  dtype = xla_bridge.canonicalize_dtype(dtype)
  return onp.broadcast_to(onp.ones((), dtype), tuple(shape))


### Tensor contraction operations


@_wraps(onp.dot)
def dot(a, b):  # pylint: disable=missing-docstring
  _check_arraylike("dot", a, b)
  a, b = _promote_dtypes(a, b)
  a_ndim, b_ndim = ndim(a), ndim(b)
  if a_ndim == 0 or b_ndim == 0:
    return lax.mul(a, b)
  if _max(a_ndim, b_ndim) <= 2:
    return lax.dot(a, b)
  a_reshaped = reshape(a, (-1, shape(a)[-1]))
  if _ndim(b) in {1, 2}:
    out = lax.dot(a_reshaped, b)
  else:
    b_reshaped = reshape(moveaxis(b, -2, 0), (shape(b)[-2], -1))
    out = lax.dot(a_reshaped, b_reshaped)
  return lax.reshape(out, a.shape[:-1] + b.shape[:-2] + b.shape[-2:][1:])



@_wraps(onp.matmul)
def matmul(a, b):  # pylint: disable=missing-docstring
  _check_arraylike("matmul", a, b)
  a_is_vec, b_is_vec = (ndim(a) == 1), (ndim(b) == 1)
  a = lax.reshape(a, (1,) + shape(a)) if a_is_vec else a
  b = lax.reshape(b, shape(b) + (1,)) if b_is_vec else b

  a, b = _promote_dtypes(a, b)
  batch_shape = _broadcast_shapes(shape(a)[:-2], shape(b)[:-2])
  a = broadcast_to(a, batch_shape + shape(a)[-2:])
  b = broadcast_to(b, batch_shape + shape(b)[-2:])
  batch_dims = tuple(range(len(batch_shape)))
  result = lax.dot_general(a, b, (((ndim(a) - 1,), (ndim(b) - 2,)),
                                  (batch_dims, batch_dims)))

  if a_is_vec or b_is_vec:
    m, n = shape(result)[-2:]
    new_m = () if a_is_vec else (m,)
    new_n = () if b_is_vec else (n,)
    return lax.reshape(result, batch_shape + new_m + new_n)
  else:
    return result



@_wraps(onp.vdot)
def vdot(a, b):
  if onp.issubdtype(_dtype(a), onp.complexfloating):
    a = conj(a)
  return dot(a.ravel(), b.ravel())



### Misc
```

```
@_wraps(onp.argmax)
def argmax(a, axis=None):
    if axis is None:
        a = ravel(a)
        axis = 0
    return _argminmax(max, a, axis)


@_wraps(onp.argmin)
def argmin(a, axis=None):
    if axis is None:
        a = ravel(a)
        axis = 0
    return _argminmax(min, a, axis)


# TODO(mattjj): redo this lowering with a call to variadic lax.reduce
def _argminmax(op, a, axis):
    shape = [1] * a.ndim
    shape[axis] = a.shape[axis]
    idxs = onp.arange(a.shape[axis]).reshape(shape)
    maxval = onp.iinfo(xla_bridge.canonicalize_dtype(idxs.dtype)).max
    mask_idxs = where(lax._eq_meet(a, op(a, axis, keepdims=True)), idxs, maxval)
    return min(mask_idxs, axis)


# TODO plan how to handle unsupported ops
def _not_implemented(fun):
    return None

argpartition = _not_implemented(onp.argpartition)
argsort = _not_implemented(onp.argsort)
compress = _not_implemented(onp.compress)
cumprod = _not_implemented(onp.cumprod)
cumsum = _not_implemented(onp.cumsum)
delete = _not_implemented(onp.delete)
diagonal = _not_implemented(onp.diagonal)
insert = _not_implemented(onp.insert)
linspace = _not_implemented(onp.linspace)
nonzero = _not_implemented(onp.nonzero)
ptp = _not_implemented(onp.ptp)
```

```
repeat = _not_implemented(onp.repeat)
searchsorted = _not_implemented(onp.searchsorted)
take = _not_implemented(onp.take)
trace = _not_implemented(onp.trace)


### Indexing


def _rewriting_take(arr, idx, axis=0):
  """A function like numpy.take that handles boxes and rewrites to LAX."""

  # Handle special indexers: (), Ellipsis, slice(None), and None.
  # TODO(mattjj): don't compare empty tuple identity (though works for CPython)
  if idx is () or idx is Ellipsis or _is_slice_none(idx):  # pylint: disable=literal-comparison
    return arr
  elif idx is None:
    return expand_dims(arr, 0)



  # Handle int index
  _int = lambda aval: not aval.shape and onp.issubdtype(aval.dtype, onp.integer)
  try:
    abstract_idx = core.get_aval(idx)
  except TypeError:
    abstract_idx = None

  if isinstance(abstract_idx, ConcreteArray) and _int(abstract_idx):
    return lax.index_in_dim(arr, idx, axis, False)
  elif isinstance(abstract_idx, ShapedArray) and _int(abstract_idx):
    idx = mod(idx, arr.shape[axis])
    return lax.dynamic_index_in_dim(arr, idx, axis, False)

  # Handle slice index (only static, otherwise an error is raised)
  elif isinstance(idx, slice):
    if not _all(elt is None or isinstance(core.get_aval(elt), ConcreteArray)
                for elt in (idx.start, idx.stop, idx.step)):
      msg = ("Array slice indices must have static start/stop/step to be used "
             "with Numpy indexing syntax. Try lax.dynamic_slice instead.")
      raise IndexError(msg)
    else:
        start, limit, stride, needs_rev = _static_idx(idx, arr.shape[axis])
        result = lax.slice_in_dim(arr, start, limit, stride, axis=axis)
        return lax.rev(result, [axis]) if needs_rev else result

# Handle non-advanced tuple indices by recursing once
elif isinstance(idx, tuple) and _all(onp.ndim(elt) == 0 for elt in idx):
    canonical_idx = _canonicalize_tuple_index(arr, idx)
    result, axis = arr, 0
    for elt in (elt for elt in canonical_idx if elt is not None):
        result = _rewriting_take(result, elt, axis=axis)
        axis += isinstance(elt, slice)      # advance axis index if not eliminated
    unexpanded_shape_itr = iter(result.shape)
    result_shape = tuple(1 if elt is None else next(unexpanded_shape_itr)
                         for elt in canonical_idx if not isinstance(elt, int))
    return lax.reshape(result, result_shape)

# Handle advanced indexing (non-tuple sequence, ndarray of dtype int or bool,
# or a tuple with at least one sequence object).
# https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html#advanced-indexing
# https://gist.github.com/seberg/976373b6a2b7c4188591

# Handle integer array indexing *without* ellipsis/slices/nones
# https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html#integer-array-indexing
if _is_advanced_int_indexer_without_slices(idx):
    if isinstance(idx, list):
        if _any(_shape(e) for e in idx):
            # At least one sequence element in the index list means broadcasting.
            idx = broadcast_arrays(*idx)
        else:
            # The index list is a flat list of integers.
            idx = [lax.concatenate([lax.reshape(e, (1,)) for e in idx], 0)]
    else:
        # The indexer is just a single integer array.
        idx = [idx]

    flat_idx = tuple(mod(ravel(x), arr.shape[i]) for i, x in enumerate(idx))
    out = lax.index_take(arr, flat_idx, tuple(range(len(idx))))
    return lax.reshape(out, idx[0].shape + _shape(arr)[len(idx):])

# Handle integer array indexing *with* ellipsis/slices/nones by recursing once
#
```

```
https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html#combining-advanced-and-basic
dexing
  elif _is_advanced_int_indexer(idx):
    canonical_idx = _canonicalize_tuple_index(arr, tuple(idx))
    idx_noadvanced = [slice(None) if _is_int(e) else e for e in canonical_idx]
    arr_sliced = _rewriting_take(arr, tuple(idx_noadvanced))

    advanced_pairs = ((e, i) for i, e in enumerate(canonical_idx) if _is_int(e))
    idx_advanced, axes = zip(*advanced_pairs)
    idx_advanced = broadcast_arrays(*idx_advanced)

    flat_idx = tuple(mod(ravel(x), arr_sliced.shape[i])
                     for i, x in zip(axes, idx_advanced))
    out = lax.index_take(arr_sliced, flat_idx, axes)
    shape_suffix = tuple(onp.delete(_shape(arr_sliced), axes))
    out = lax.reshape(out, idx_advanced[0].shape + shape_suffix)

    axes_are_contiguous = onp.all(onp.diff(axes) == 1)
    if axes_are_contiguous:
      start = axes[0]
      naxes = idx_advanced[0].ndim
      out = moveaxis(out, list(range(naxes)), list(range(start, start + naxes)))
    return out

  msg = "Indexing mode not yet supported. Open a feature request!\n{}"
  raise IndexError(msg.format(idx))


def _is_slice_none(idx):
  """Return True if idx is equal to slice(None), falsey otherwise."""
  if isinstance(idx, slice):
    return idx.start is None and idx.stop is None and idx.step is None


def _is_advanced_int_indexer(idx):
  """Returns True if idx should trigger int array indexing, False otherwise."""
  # https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html#advanced-indexing
  if isinstance(idx, (tuple, list)):
    # We assume this check comes *after* the check for non-advanced tuple index,
    # and hence we already know at least one element is a sequence
    return _all(e is None or e is Ellipsis or isinstance(e, slice) or _is_int(e)
                                for e in idx)
    else:
        return _is_int(idx)



def _is_advanced_int_indexer_without_slices(idx):
    """Returns True iff idx is an advanced int idx without slice/ellipsis/none."""
    if _is_advanced_int_indexer(idx):
        if isinstance(idx, (tuple, list)):
            return not _any(e is None or e is Ellipsis or isinstance(e, slice)
                                            for e in idx)
        else:
            return True



def _is_int(x):
    """Returns True if x is array-like with integer dtype, falsey otherwise."""
    return (isinstance(x, int) and not isinstance(x, bool)
                    or onp.issubdtype(getattr(x, "dtype", None), onp.integer)
                    or isinstance(x, (list, tuple)) and _all(_is_int(e) for e in x))



def _canonicalize_tuple_index(arr, idx):
    """Helper to remove Ellipsis and add in the implicit trailing slice(None)."""
    len_without_none = _sum(1 for e in idx if e is not None and e is not Ellipsis)
    if len_without_none > arr.ndim:
        msg = "Too many indices for array: {} non-None/Ellipsis indices for dim {}."
        raise IndexError(msg.format(len_without_none, arr.ndim))
    ellipses = (i for i, elt in enumerate(idx) if elt is Ellipsis)
    ellipsis_index = next(ellipses, None)
    if ellipsis_index is not None:
        if next(ellipses, None) is not None:
            msg = "Multiple ellipses (...) not supported: {}."
            raise IndexError(msg.format(list(map(type, idx))))
        colons = (slice(None),) * (arr.ndim - len_without_none)
        idx = idx[:ellipsis_index] + colons + idx[ellipsis_index + 1:]
    elif len_without_none < arr.ndim:
        colons = (slice(None),) * (arr.ndim - len_without_none)
        idx = tuple(idx) + colons
    return idx
```

```
def _static_idx(idx, size):
    """Helper function to compute the static slice start/limit/stride values."""
    indices = onp.arange(size)[idx]    # get shape statically
    if not len(indices):    # pylint: disable=g-explicit-length-test
        return 0, 0, 1, False    # sliced to size zero
    start, stop_inclusive = indices[0], indices[-1]
    step = 1 if idx.step is None else idx.step
    if step > 0:
        end = _min(stop_inclusive + step, size)
        return start, end, step, False
    else:
        end = _min(start - step, size)
        return stop_inclusive, end, -step, True



### add method and operator overloads to arraylike classes

# We add operator overloads to DeviceArray and ShapedArray. These method and
# operator overloads mainly just forward calls to the corresponding lax_numpy
# functions, which can themselves handle instances from any of these classes.



def _swap_args(f):
    return lambda x, y: f(y, x)

_operators = {
        "astype": lax.convert_element_type,
        "getitem": _rewriting_take,
        "neg": negative,
        "eq": equal,
        "ne": not_equal,
        "lt": less,
        "le": less_equal,
        "gt": greater,
        "ge": greater_equal,
        "abs": abs,
        "add": add,
        "radd": add,
        "sub": subtract,
        "rsub": _swap_args(subtract),
    "mul": multiply,
    "rmul": multiply,
    "div": divide,
    "rdiv": _swap_args(divide),
    "truediv": true_divide,
    "rtruediv": _swap_args(true_divide),
    "floordiv": floor_divide,
    "rfloordiv": _swap_args(floor_divide),
    "divmod": divmod,
    "rdivmod": _swap_args(divmod),
    "mod": mod,
    "rmod": _swap_args(mod),
    "pow": power,
    "rpow": _swap_args(power),
    "matmul": matmul,
    "rmatmul": _swap_args(matmul),
    "and": bitwise_and,
    "rand": bitwise_and,
    "or": bitwise_or,
    "ror": bitwise_or,
    "xor": bitwise_xor,
    "rxor": bitwise_xor,
    "invert": bitwise_not,
    "lshift": left_shift,
    "rshift": right_shift,
}

# These numpy.ndarray methods are just refs to an equivalent numpy function
_nondiff_methods = ["all", "any", "argmax", "argmin", "argpartition", "argsort",
                    "nonzero", "searchsorted", "round"]
_diff_methods = ["clip", "compress", "conj", "conjugate", "cumprod", "cumsum",
                 "diagonal", "dot", "max", "mean", "min", "prod", "ptp",
                 "ravel", "repeat", "reshape", "sort", "squeeze", "std", "sum",
                 "swapaxes", "take", "trace", "transpose", "var"]


# Set up operator, method, and property forwarding on Tracer instances containing
# ShapedArray avals by following the forwarding conventions for Tracer.
# Forward operators using a single-underscore-prefix naming convention:
for operator_name, function in _operators.items():
  setattr(ShapedArray, "_{}".format(operator_name), staticmethod(function))
```

```
# Forward methods and properties using core.aval_method and core.aval_property:
for method_name in _nondiff_methods + _diff_methods:
  setattr(ShapedArray, method_name, core.aval_method(globals()[method_name]))
setattr(ShapedArray, "flatten", core.aval_method(ravel))
setattr(ShapedArray, "T", core.aval_property(transpose))


# Forward operators, methods, and properies on DeviceArray to lax_numpy
# functions (with no Tracers involved; this forwarding is direct)
for operator_name, function in _operators.items():
  setattr(DeviceArray, "__{}__".format(operator_name), function)
for method_name in _nondiff_methods + _diff_methods:
  setattr(DeviceArray, method_name, globals()[method_name])
setattr(DeviceArray, "flatten", ravel)
setattr(DeviceArray, "T", property(transpose))


# Extra methods that are handy
setattr(DeviceArray, "broadcast", lax.broadcast)
```