# jax.numpy module

## Contents



Implements the NumPy API, using the primitives in **`jax.lax`**.

While JAX tries to follow the NumPy API as closely as possible, sometimes JAX cannot follow NumPy exactly.

- Notably, since JAX arrays are immutable, NumPy APIs that mutate arrays in-place cannot be implemented in JAX. However, often JAX is able to provide an alternative API that is purely functional. For example, instead of in-place array updates (`x[i] = y`), JAX provides an alternative pure indexed update function `x.at[i].set(y)` (see **`ndarray.at`**).
- Relatedly, some NumPy functions often return views of arrays when possible (examples are **`transpose()`** and **`reshape()`**). JAX versions of such functions will return copies instead, although such are often optimized away by XLA when sequences of operations are compiled using **`jax.jit()`**.
- NumPy is very aggressive at promoting values to `float64` type. JAX sometimes is less aggressive about type promotion (See Type promotion semantics).
- Some NumPy routines have data-dependent output shapes (examples include **`unique()`** and **`nonzero()`**). Because the XLA compiler requires array shapes to be known at compile time, such operations are not compatible with JIT. For this reason, JAX adds an optional `size` argument to such functions which may be specified statically in order to use them with JIT.

Nearly all applicable NumPy functions are implemented in the `jax.numpy` namespace; they are listed below.

| | |
|---|---|
| `ndarray.at` | Helper property for index update functionality. |
| `abs`(x, /) | Calculate the absolute value element-wise. |
| `absolute`(x, /) | Calculate the absolute value element-wise. |
| `add`(x1, x2, /) | Add arguments element-wise. |
| `all`(a[, axis, out, keepdims, where]) | Test whether all array elements along a given axis evaluate to True. |
| `allclose`(a, b[, rtol, atol, equal_nan]) | Returns True if two arrays are element-wise equal within a tolerance. |
| `alltrue`(a[, axis, out, keepdims, where]) | Test whether all array elements along a given axis evaluate to True. |
| `amax`(a[, axis, out, keepdims, initial, where]) | Return the maximum of an array or maximum along an axis. |
| `amin`(a[, axis, out, keepdims, initial, where]) | Return the minimum of an array or minimum along an axis. |
| `angle`(z[, deg]) | Return the angle of the complex argument. |
| `any`(a[, axis, out, keepdims, where]) | Test whether any array element along a given axis evaluates to True. |
| `append`(arr, values[, axis]) | Append values to the end of an array. |
| `apply_along_axis`(func1d, axis, arr, *args, ...) | Apply a function to 1-D slices along the given axis. |
| `apply_over_axes`(func, a, axes) | Apply a function repeatedly over multiple axes. |

| | |
|---|---|
| **arange**(start[, stop, step, dtype]) | Return evenly spaced values within a given interval. |
| **arccos**(x, /) | Trigonometric inverse cosine, element-wise. |
| **arccosh**(x, /) | Inverse hyperbolic cosine, element-wise. |
| **arcsin**(x, /) | Inverse sine, element-wise. |
| **arcsinh**(x, /) | Inverse hyperbolic sine element-wise. |
| **arctan**(x, /) | Trigonometric inverse tangent, element-wise. |
| **arctan2**(x1, x2, /) | Element-wise arc tangent of `x1/x2` choosing the quadrant correctly. |
| **arctanh**(x, /) | Inverse hyperbolic tangent element-wise. |
| **argmax**(a[, axis, out, keepdims]) | Returns the indices of the maximum values along an axis. |
| **argmin**(a[, axis, out, keepdims]) | Returns the indices of the minimum values along an axis. |
| **argsort**(a[, axis, kind, order]) | Returns the indices that would sort an array. |
| **argwhere**(a, *[, size, fill_value]) | Find the indices of array elements that are non-zero, grouped by element. |
| **around**(a[, decimals, out]) | Evenly round to the given number of decimals. |
| **array**(object[, dtype, copy, order, ndmin]) | Create an array. |

| | |
|---|---|
| `array_equal`(a1, a2[, equal_nan]) | True if two arrays have the same shape and elements, False otherwise. |
| `array_equiv`(a1, a2) | Returns True if input arrays are shape consistent and all elements equal. |
| `array_repr`(arr[, max_line_width, precision, ...]) | Return the string representation of an array. |
| `array_split`(ary, indices_or_sections[, axis]) | Split an array into multiple sub-arrays. |
| `array_str`(a[, max_line_width, precision, ...]) | Return a string representation of the data in an array. |
| `asarray`(a[, dtype, order]) | Convert the input to an array. |
| `atleast_1d`(*arys) | Convert inputs to arrays with at least one dimension. |
| `atleast_2d`(*arys) | View inputs as arrays with at least two dimensions. |
| `atleast_3d`(*arys) | View inputs as arrays with at least three dimensions. |
| `average`() | Compute the weighted average along the specified axis. |
| `bartlett`(M) | Return the Bartlett window. |
| `bincount`(x[, weights, minlength, length]) | Count number of occurrences of each value in array of non-negative ints. |
| `bitwise_and`(x1, x2, /) | Compute the bit-wise AND of two arrays element-wise. |
| `bitwise_not`(x, /) | Compute bit-wise inversion, or bit-wise NOT, element-wise. |

| | |
|---|---|
| `bitwise_or`(x1, x2, /) | Compute the bit-wise OR of two arrays element-wise. |
| `bitwise_xor`(x1, x2, /) | Compute the bit-wise XOR of two arrays element-wise. |
| `blackman`(M) | Return the Blackman window. |
| `block`(arrays) | Assemble an nd-array from nested lists of blocks. |
| `bool_`(x) | |
| `broadcast_arrays`(*args) | Broadcast any number of arrays against each other. |
| `broadcast_shapes`() | Broadcast the input shapes into a single shape. |
| `broadcast_to`(array, shape) | Broadcast an array to a new shape. |
| `c_` | Concatenate slices, scalars and array-like objects along the last axis. |
| `can_cast`(from_, to[, casting]) | Returns True if cast between data types can occur according to the casting rule. |
| `cbrt`(x, /) | Return the cube-root of an array, element-wise. |
| `cdouble` | alias of `jax.numpy.complex128` |
| `ceil`(x, /) | Return the ceiling of the input, element-wise. |
| `character`() | Abstract base class of all character string scalar types. |

| | |
|---|---|
| choose(a, choices[, out, mode]) | Construct an array from an index array and a list of arrays to choose from. |
| clip(a[, a_min, a_max, out]) | Clip (limit) the values in an array. |
| column_stack(tup) | Stack 1-D arrays as columns into a 2-D array. |
| complex_ | alias of jax.numpy.complex128 |
| complex128(x) | |
| complex64(x) | |
| complexfloating() | Abstract base class of all complex number scalar types that are made up of floating-point numbers. |
| ComplexWarning | The warning raised when casting a complex dtype to a real dtype. |
| compress(condition, a[, axis, out]) | Return selected slices of an array along given axis. |
| concatenate(arrays[, axis, dtype]) | Join a sequence of arrays along an existing axis. |
| conj(x, /) | Return the complex conjugate, element-wise. |
| conjugate(x, /) | Return the complex conjugate, element-wise. |
| convolve(a, v[, mode, precision]) | Returns the discrete, linear convolution of two one-dimensional sequences. |
| copy(a[, order]) | Return an array copy of the given object. |

| | |
|---|---|
| `copysign`(x1, x2, /) | Change the sign of x1 to that of x2, element-wise. |
| `corrcoef`(x[, y, rowvar]) | Return Pearson product-moment correlation coefficients. |
| `correlate`(a, v[, mode, precision]) | Cross-correlation of two 1-dimensional sequences. |
| `cos`(x, /) | Cosine element-wise. |
| `cosh`(x, /) | Hyperbolic cosine, element-wise. |
| `count_nonzero`(a[, axis, keepdims]) | Counts the number of non-zero values in the array `a`. |
| `cov`(m[, y, rowvar, bias, ddof, fweights, ...]) | Estimate a covariance matrix, given data and weights. |
| `cross`(a, b[, axisa, axisb, axisc, axis]) | Return the cross product of two (arrays of) vectors. |
| `csingle` | alias of `jax.numpy.complex64` |
| `cumprod`(a[, axis, dtype, out]) | Return the cumulative product of elements along a given axis. |
| `cumproduct`(a[, axis, dtype, out]) | Return the cumulative product of elements along a given axis. |
| `cumsum`(a[, axis, dtype, out]) | Return the cumulative sum of the elements along a given axis. |
| `deg2rad`(x, /) | Convert angles from degrees to radians. |
| `degrees`(x, /) | Convert angles from radians to degrees. |

| | |
|---|---|
| `delete`(arr, obj[, axis]) | Return a new array with sub-arrays along an axis deleted. |
| `diag`(v[, k]) | Extract a diagonal or construct a diagonal array. |
| `diag_indices`(n[, ndim]) | Return the indices to access the main diagonal of an array. |
| `diag_indices_from`(arr) | Return the indices to access the main diagonal of an n-dimensional array. |
| `diagflat`(v[, k]) | Create a two-dimensional array with the flattened input as a diagonal. |
| `diagonal`(a[, offset, axis1, axis2]) | Return specified diagonals. |
| `diff`(a[, n, axis, prepend, append]) | Calculate the n-th discrete difference along the given axis. |
| `digitize`(x, bins[, right]) | Return the indices of the bins to which each value in input array belongs. |
| `divide`(x1, x2, /) | Divide arguments element-wise. |
| `divmod`(x1, x2, /) | Return element-wise quotient and remainder simultaneously. |
| `dot`(a, b, *[, precision]) | Dot product of two arrays. |
| `double` | alias of `jax.numpy.float64` |
| `dsplit`(ary, indices_or_sections) | Split array into multiple sub-arrays along the 3rd axis (depth). |
| `dstack`(tup[, dtype]) | Stack arrays in sequence depth wise (along third axis). |
| `dtype`(dtype[, align, copy]) | Create a data type object. |

| | |
|---|---|
| **ediff1d**(ary[, to_end, to_begin]) | The differences between consecutive elements of an array. |
| **einsum**(*operands[, out, optimize, ...]) | Evaluates the Einstein summation convention on the operands. |
| **einsum_path**(subscripts, *operands[, optimize]) | Evaluates the lowest cost contraction order for an einsum expression by |
| **empty**(shape[, dtype]) | Return a new array of given shape and type, without initializing entries. |
| **empty_like**(prototype[, dtype, shape]) | Return a new array with the same shape and type as a given array. |
| **equal**(x1, x2, /) | Return (x1 == x2) element-wise. |
| **exp**(x, /) | Calculate the exponential of all elements in the input array. |
| **exp2**(x, /) | Calculate *2**p* for all *p* in the input array. |
| **expand_dims**(a, axis) | Expand the shape of an array. |
| **expm1**(x, /) | Calculate `exp(x) - 1` for all elements in the array. |
| **extract**(condition, arr) | Return the elements of an array that satisfy some condition. |
| **eye**(N[, M, k, dtype]) | Return a 2-D array with ones on the diagonal and zeros elsewhere. |
| **fabs**(x, /) | Compute the absolute values element-wise. |
| **finfo**(dtype) | Machine limits for floating point types. |

| | |
|---|---|
| `fix`(x[, out]) | Round to nearest integer towards zero. |
| `flatnonzero`(a, *[, size, fill_value]) | Return indices that are non-zero in the flattened version of a. |
| `flexible`() | Abstract base class of all scalar types without predefined length. |
| `flip`(m[, axis]) | Reverse the order of elements in an array along the given axis. |
| `fliplr`(m) | Reverse the order of elements along axis 1 (left/right). |
| `flipud`(m) | Reverse the order of elements along axis 0 (up/down). |
| `float_` | alias of `jax.numpy.float64` |
| `float_power`(x1, x2, /) | First array elements raised to powers from second array, element-wise. |
| `float16`(x) | |
| `float32`(x) | |
| `float64`(x) | |
| `floating`() | Abstract base class of all floating-point scalar types. |
| `floor`(x, /) | Return the floor of the input, element-wise. |
| `floor_divide`(x1, x2, /) | Return the largest integer smaller or equal to the division of the inputs. |
| `fmax`(x1, x2) | Element-wise maximum of array elements. |

| | |
|---|---|
| **fmin**(x1, x2) | Element-wise minimum of array elements. |
| **fmod**(x1, x2, /) | Returns the element-wise remainder of division. |
| **frexp**(x, /) | Decompose the elements of x into mantissa and twos exponent. |
| **frombuffer**(buffer[, dtype, count, offset]) | Interpret a buffer as a 1-dimensional array. |
| **fromfile**(*args, **kwargs) | Unimplemented JAX wrapper for jnp.fromfile. |
| **fromfunction**(function, shape, *[, dtype]) | Construct an array by executing a function over each coordinate. |
| **fromiter**(*args, **kwargs) | Unimplemented JAX wrapper for jnp.fromiter. |
| **fromstring**(string[, dtype, count]) | A new 1-D array initialized from text data in a string. |
| **from_dlpack**(x) | Create a NumPy array from an object implementing the `__dlpack__` |
| **full**(shape, fill_value[, dtype]) | Return a new array of given shape and type, filled with *fill_value*. |
| **full_like**(a, fill_value[, dtype, shape]) | Return a full array with the same shape and type as a given array. |
| **gcd**(x1, x2) | Returns the greatest common divisor of `\|x1\|` and `\|x2\|` |
| **generic**() | Base class for numpy scalar types. |
| **geomspace**(start, stop[, num, endpoint, ...]) | Return numbers spaced evenly on a log scale (a geometric progression). |

| | |
|---|---|
| `get_printoptions`() | Return the current print options. |
| `gradient`(f, *varargs[, axis, edge_order]) | Return the gradient of an N-dimensional array. |
| `greater`(x1, x2, /) | Return the truth value of (x1 > x2) element-wise. |
| `greater_equal`(x1, x2, /) | Return the truth value of (x1 >= x2) element-wise. |
| `hamming`(M) | Return the Hamming window. |
| `hanning`(M) | Return the Hanning window. |
| `heaviside`(x1, x2, /) | Compute the Heaviside step function. |
| `histogram`(a[, bins, range, weights, density]) | Compute the histogram of a dataset. |
| `histogram_bin_edges`(a[, bins, range, weights]) | Function to calculate only the edges of the bins used by the *histogram* |
| `histogram2d`(x, y[, bins, range, weights, ...]) | Compute the bi-dimensional histogram of two data samples. |
| `histogramdd`(sample[, bins, range, weights, ...]) | Compute the multidimensional histogram of some data. |
| `hsplit`(ary, indices_or_sections) | Split an array into multiple sub-arrays horizontally (column-wise). |
| `hstack`(tup[, dtype]) | Stack arrays in sequence horizontally (column wise). |
| `hypot`(x1, x2, /) | Given the "legs" of a right triangle, return its hypotenuse. |
| `i0`(x) | Modified Bessel function of the first kind, order 0. |

| | |
|---|---|
| identity(n[, dtype]) | Return the identity array. |
| iinfo(type) | Machine limits for integer types. |
| imag(val, /) | Return the imaginary part of the complex argument. |
| in1d(ar1, ar2[, assume_unique, invert]) | Test whether each element of a 1-D array is also present in a second array. |
| index_exp | A nicer way to build up index tuples for arrays. |
| indices() | Return an array representing the indices of a grid. |
| inexact() | Abstract base class of all numeric scalar types with a (potentially) inexact representation of the values in its range, such as floating-point numbers. |
| inner(a, b, *[, precision]) | Inner product of two arrays. |
| insert(arr, obj, values[, axis]) | Insert values along the given axis before the given indices. |
| int_ | alias of jax.numpy.int64 |
| int16(x) | |
| int32(x) | |
| int64(x) | |
| int8(x) | |
| integer() | Abstract base class of all integer scalar types. |

| | |
|---|---|
| **interp**(x, xp, fp[, left, right, period]) | One-dimensional linear interpolation for monotonically increasing sample points. |
| **intersect1d**(ar1, ar2[, assume_unique, ...]) | Find the intersection of two arrays. |
| **invert**(x, /) | Compute bit-wise inversion, or bit-wise NOT, element-wise. |
| **isclose**(a, b[, rtol, atol, equal_nan]) | Returns a boolean array where two arrays are element-wise equal within a |
| **iscomplex**(x) | Returns a bool array, where True if input element is complex. |
| **iscomplexobj**(x) | Check for a complex type or an array of complex numbers. |
| **isfinite**(x, /) | Test element-wise for finiteness (not infinity and not Not a Number). |
| **isin**(element, test_elements[, ...]) | Calculates `element in test_elements`, broadcasting over *element* only. |
| **isinf**(x, /) | Test element-wise for positive or negative infinity. |
| **isnan**(x, /) | Test element-wise for NaN and return result as a boolean array. |
| **isneginf**(x, /[, out]) | Test element-wise for negative infinity, return result as bool array. |
| **isposinf**(x, /[, out]) | Test element-wise for positive infinity, return result as bool array. |
| **isreal**(x) | Returns a bool array, where True if input element is real. |

| | |
|---|---|
| **isrealobj**(x) | Return True if x is a not complex type or an array of complex numbers. |
| **isscalar**(element) | Returns True if the type of *element* is a scalar type. |
| **issubdtype**(arg1, arg2) | Returns True if first argument is a typecode lower/equal in type hierarchy. |
| **issubsctype**(arg1, arg2) | Determine if the first argument is a subclass of the second argument. |
| **iterable**(y) | Check whether or not an object can be iterated over. |
| **ix_**(*args) | Construct an open mesh from multiple sequences. |
| **kaiser**(M, beta) | Return the Kaiser window. |
| **kron**(a, b) | Kronecker product of two arrays. |
| **lcm**(x1, x2) | Returns the lowest common multiple of `\|x1\|` and `\|x2\|` |
| **ldexp**(x1, x2, /) | Returns x1 * 2**x2, element-wise. |
| **left_shift**(x1, x2, /) | Shift the bits of an integer to the left. |
| **less**(x1, x2, /) | Return the truth value of (x1 < x2) element-wise. |
| **less_equal**(x1, x2, /) | Return the truth value of (x1 <= x2) element-wise. |
| **lexsort**(keys[, axis]) | Perform an indirect stable sort using a sequence of keys. |

| | |
|---|---|
| **linspace**() | Return evenly spaced numbers over a specified interval. |
| **load**(*args, **kwargs) | Load arrays or pickled objects from `.npy`, `.npz` or pickled files. |
| **log**(x, /) | Natural logarithm, element-wise. |
| **log10**(x, /) | Return the base 10 logarithm of the input array, element-wise. |
| **log1p**(x, /) | Return the natural logarithm of one plus the input array, element-wise. |
| **log2**(x, /) | Base-2 logarithm of *x*. |
| **logaddexp**(x1, x2, /) | Logarithm of the sum of exponentiations of the inputs. |
| **logaddexp2**(x1, x2, /) | Logarithm of the sum of exponentiations of the inputs in base-2. |
| **logical_and**(*args) | Compute the truth value of x1 AND x2 element-wise. |
| **logical_not**(*args) | Compute the truth value of NOT x element-wise. |
| **logical_or**(*args) | Compute the truth value of x1 OR x2 element-wise. |
| **logical_xor**(*args) | Compute the truth value of x1 XOR x2, element-wise. |
| **logspace**(start, stop[, num, endpoint, base, ...]) | Return numbers spaced evenly on a log scale. |
| **mask_indices**(*args, **kwargs) | Return the indices to access (n, n) arrays, given a masking function. |

| | |
|---|---|
| matmul(a, b, *[, precision]) | Matrix product of two arrays. |
| max(a[, axis, out, keepdims, initial, where]) | Return the maximum of an array or maximum along an axis. |
| maximum(x1, x2, /) | Element-wise maximum of array elements. |
| mean(a[, axis, dtype, out, keepdims, where]) | Compute the arithmetic mean along the specified axis. |
| median(a[, axis, out, overwrite_input, keepdims]) | Compute the median along the specified axis. |
| meshgrid(*xi[, copy, sparse, indexing]) | Return coordinate matrices from coordinate vectors. |
| mgrid | Return dense multi-dimensional "meshgrid". |
| min(a[, axis, out, keepdims, initial, where]) | Return the minimum of an array or minimum along an axis. |
| minimum(x1, x2, /) | Element-wise minimum of array elements. |
| mod(x1, x2, /) | Returns the element-wise remainder of division. |
| modf(x, /[, out]) | Return the fractional and integral parts of an array, element-wise. |
| moveaxis(a, source, destination) | Move axes of an array to new positions. |
| msort(a) | Return a copy of an array sorted along the first axis. |
| multiply(x1, x2, /) | Multiply arguments element-wise. |

| | |
|---|---|
| `nan_to_num`(x[, copy, nan, posinf, neginf]) | Replace NaN with zero and infinity with large finite numbers (default |
| `nanargmax`(a[, axis, out, keepdims]) | Return the indices of the maximum values in the specified axis ignoring |
| `nanargmin`(a[, axis, out, keepdims]) | Return the indices of the minimum values in the specified axis ignoring |
| `nancumprod`(a[, axis, dtype, out]) | Return the cumulative product of array elements over a given axis treating Not a |
| `nancumsum`(a[, axis, dtype, out]) | Return the cumulative sum of array elements over a given axis treating Not a |
| `nanmax`(a[, axis, out, keepdims, initial, where]) | Return the maximum of an array or maximum along an axis, ignoring any |
| `nanmean`(a[, axis, dtype, out, keepdims, where]) | Compute the arithmetic mean along the specified axis, ignoring NaNs. |
| `nanmedian`(a[, axis, out, overwrite_input, ...]) | Compute the median along the specified axis, while ignoring NaNs. |
| `nanmin`(a[, axis, out, keepdims, initial, where]) | Return minimum of an array or minimum along an axis, ignoring any NaNs. |
| `nanpercentile`(a, q[, axis, out, ...]) | Compute the qth percentile of the data along the specified axis, |
| `nanprod`(a[, axis, dtype, out, keepdims, ...]) | Return the product of array elements over a given axis treating Not a |
| `nanquantile`(a, q[, axis, out, ...]) | Compute the qth quantile of the data along the specified axis, |

| | |
|---|---|
| nanstd(a[, axis, dtype, out, ddof, ...]) | Compute the standard deviation along the specified axis, while |
| nansum(a[, axis, dtype, out, keepdims, ...]) | Return the sum of array elements over a given axis treating Not a |
| nanvar(a[, axis, dtype, out, ddof, ...]) | Compute the variance along the specified axis, while ignoring NaNs. |
| ndarray | alias of `jax.Array` |
| ndim(a) | Return the number of dimensions of an array. |
| negative(x, /) | Numerical negative, element-wise. |
| nextafter(x1, x2, /) | Return the next floating-point value after x1 towards x2, element-wise. |
| nonzero(a, *[, size, fill_value]) | Return the indices of the elements that are non-zero. |
| not_equal(x1, x2, /) | Return (x1 != x2) element-wise. |
| number() | Abstract base class of all numeric scalar types. |
| object_ | Any Python object. |
| ogrid | Return open multi-dimensional "meshgrid". |
| ones(shape[, dtype]) | Return a new array of given shape and type, filled with ones. |
| ones_like(a[, dtype, shape]) | Return an array of ones with the same shape and type as a given array. |
| outer(a, b[, out]) | Compute the outer product of two vectors. |

| | |
|---|---|
| **packbits**(a[, axis, bitorder]) | Packs the elements of a binary-valued array into bits in a uint8 array. |
| **pad**(array, pad_width[, mode]) | Pad an array. |
| **partition**(a, kth[, axis]) | Return a partitioned copy of an array. |
| **percentile**(a, q[, axis, out, ...]) | Compute the q-th percentile of the data along the specified axis. |
| **piecewise**(x, condlist, funclist, *args, **kw) | Evaluate a piecewise-defined function. |
| **place**(*args, **kwargs) | Change elements of an array based on conditional and input values. |
| **poly**(seq_of_zeros) | Find the coefficients of a polynomial with the given sequence of roots. |
| **polyadd**(a1, a2) | Find the sum of two polynomials. |
| **polyder**(p[, m]) | Return the derivative of the specified order of a polynomial. |
| **polydiv**(u, v, *[, trim_leading_zeros]) | Returns the quotient and remainder of polynomial division. |
| **polyfit**(x, y, deg[, rcond, full, w, cov]) | Least squares polynomial fit. |
| **polyint**(p[, m, k]) | Return an antiderivative (indefinite integral) of a polynomial. |
| **polymul**(a1, a2, *[, trim_leading_zeros]) | Find the product of two polynomials. |
| **polysub**(a1, a2) | Difference (subtraction) of two polynomials. |
| **polyval**(p, x, *[, unroll]) | Evaluate a polynomial at specific values. |
| **positive**(x, /) | Numerical positive, element-wise. |

| | |
|---|---|
| power(x1, x2, /) | First array elements raised to powers from second array, element-wise. |
| printoptions(*args, **kwargs) | Context manager for setting print options. |
| prod(a[, axis, dtype, out, keepdims, ...]) | Return the product of array elements over a given axis. |
| product(a[, axis, dtype, out, keepdims, ...]) | Return the product of array elements over a given axis. |
| promote_types(a, b) | Returns the type to which a binary operation should cast its arguments. |
| ptp(a[, axis, out, keepdims]) | Range of values (maximum - minimum) along an axis. |
| put(*args, **kwargs) | Replaces specified elements of an array with given values. |
| quantile(a, q[, axis, out, overwrite_input, ...]) | Compute the q-th quantile of the data along the specified axis. |
| r_ | Concatenate slices, scalars and array-like objects along the first axis. |
| rad2deg(x, /) | Convert angles from radians to degrees. |
| radians(x, /) | Convert angles from degrees to radians. |
| ravel(a[, order]) | Return a contiguous flattened array. |
| ravel_multi_index(multi_index, dims[, mode, ...]) | Converts a tuple of index arrays into an array of flat |
| real(val, /) | Return the real part of the complex argument. |

| | |
|---|---|
| `reciprocal`(x, /) | Return the reciprocal of the argument, element-wise. |
| `remainder`(x1, x2, /) | Returns the element-wise remainder of division. |
| `repeat`(a, repeats[, axis, total_repeat_length]) | Repeat elements of an array. |
| `reshape`(a, newshape[, order]) | Gives a new shape to an array without changing its data. |
| `resize`(a, new_shape) | Return a new array with the specified shape. |
| `result_type`(*args) | Returns the type that results from applying the NumPy |
| `right_shift`(x1, x2, /) | Shift the bits of an integer to the right. |
| `rint`(x, /) | Round elements of the array to the nearest integer. |
| `roll`(a, shift[, axis]) | Roll array elements along a given axis. |
| `rollaxis`(a, axis[, start]) | Roll the specified axis backwards, until it lies in a given position. |
| `roots`(p, *[, strip_zeros]) | Return the roots of a polynomial with coefficients given in p. |
| `rot90`(m[, k, axes]) | Rotate an array by 90 degrees in the plane specified by axes. |
| `round`(a[, decimals, out]) | Evenly round to the given number of decimals. |
| `round_`(a[, decimals, out]) | Evenly round to the given number of decimals. |

| | |
|---|---|
| `row_stack`(tup[, dtype]) | Stack arrays in sequence vertically (row wise). |
| `s_` | A nicer way to build up index tuples for arrays. |
| `save`(file, arr[, allow_pickle, fix_imports]) | Save an array to a binary file in NumPy `.npy` format. |
| `savez`(file, *args, **kwds) | Save several arrays into a single file in uncompressed `.npz` format. |
| `searchsorted`(a, v[, side, sorter, method]) | Find indices where elements should be inserted to maintain order. |
| `select`(condlist, choicelist[, default]) | Return an array drawn from elements in choicelist, depending on conditions. |
| `set_printoptions`([precision, threshold, ...]) | Set printing options. |
| `setdiff1d`(ar1, ar2[, assume_unique, size, ...]) | Find the set difference of two arrays. |
| `setxor1d`(ar1, ar2[, assume_unique]) | Find the set exclusive-or of two arrays. |
| `shape`(a) | Return the shape of an array. |
| `sign`(x, /) | Returns an element-wise indication of the sign of a number. |
| `signbit`(x, /) | Returns element-wise True where signbit is set (less than zero). |
| `signedinteger`() | Abstract base class of all signed integer scalar types. |
| `sin`(x, /) | Trigonometric sine, element-wise. |
| `sinc`(x, /) | Return the normalized sinc function. |
| `single` | alias of `jax.numpy.float32` |

| | |
|---|---|
| **sinh**(x, /) | Hyperbolic sine, element-wise. |
| **size**(a[, axis]) | Return the number of elements along a given axis. |
| **sometrue**(a[, axis, out, keepdims, where]) | Test whether any array element along a given axis evaluates to True. |
| **sort**(a[, axis, kind, order]) | Return a sorted copy of an array. |
| **sort_complex**(a) | Sort a complex array using the real part first, then the imaginary part. |
| **split**(ary, indices_or_sections[, axis]) | Split an array into multiple sub-arrays as views into *ary*. |
| **sqrt**(x, /) | Return the non-negative square-root of an array, element-wise. |
| **square**(x, /) | Return the element-wise square of the input. |
| **squeeze**(a[, axis]) | Remove axes of length one from *a*. |
| **stack**(arrays[, axis, out, dtype]) | Join a sequence of arrays along a new axis. |
| **std**(a[, axis, dtype, out, ddof, keepdims, where]) | Compute the standard deviation along the specified axis. |
| **subtract**(x1, x2, /) | Subtract arguments, element-wise. |
| **sum**(a[, axis, dtype, out, keepdims, ...]) | Sum of array elements over a given axis. |
| **swapaxes**(a, axis1, axis2) | Interchange two axes of an array. |
| **take**(a, indices[, axis, out, mode, ...]) | Take elements from an array along an axis. |

| | |
|---|---|
| `take_along_axis`(arr, indices, axis[, mode]) | Take values from the input array by matching 1d index and data slices. |
| `tan`(x, /) | Compute tangent element-wise. |
| `tanh`(x, /) | Compute hyperbolic tangent element-wise. |
| `tensordot`(a, b[, axes, precision]) | Compute tensor dot product along specified axes. |
| `tile`(A, reps) | Construct an array by repeating A the number of times given by reps. |
| `trace`(a[, offset, axis1, axis2, dtype, out]) | Return the sum along diagonals of the array. |
| `transpose`(a[, axes]) | Returns an array with axes transposed. |
| `trapz`(y[, x, dx, axis]) | Integrate along the given axis using the composite trapezoidal rule. |
| `tri`(N[, M, k, dtype]) | An array with ones at and below the given diagonal and zeros elsewhere. |
| `tril`(m[, k]) | Lower triangle of an array. |
| `tril_indices`(*args, **kwargs) | Return the indices for the lower-triangle of an (n, m) array. |
| `tril_indices_from`(arr[, k]) | Return the indices for the lower-triangle of arr. |
| `trim_zeros`(filt[, trim]) | Trim the leading and/or trailing zeros from a 1-D array or sequence. |
| `triu`(m[, k]) | Upper triangle of an array. |

| | |
|---|---|
| `triu_indices`(*args, **kwargs) | Return the indices for the upper-triangle of an (n, m) array. |
| `triu_indices_from`(arr[, k]) | Return the indices for the upper-triangle of arr. |
| `true_divide`(x1, x2, /) | Divide arguments element-wise. |
| `trunc`(x) | Return the truncated value of the input, element-wise. |
| `uint` | alias of `jax.numpy.uint64` |
| `uint16`(x) | |
| `uint32`(x) | |
| `uint64`(x) | |
| `uint8`(x) | |
| `union1d`(ar1, ar2, *[, size, fill_value]) | Find the union of two arrays. |
| `unique`(ar[, return_index, return_inverse, ...]) | Find the unique elements of an array. |
| `unpackbits`(a[, axis, count, bitorder]) | Unpacks elements of a uint8 array into a binary-valued output array. |
| `unravel_index`(indices, shape) | Converts a flat index or array of flat indices into a tuple |
| `unsignedinteger`() | Abstract base class of all unsigned integer scalar types. |
| `unwrap`(p[, discont, axis, period]) | Unwrap by taking the complement of large deltas with respect to the period. |
| `vander`(x[, N, increasing]) | Generate a Vandermonde matrix. |

| | |
|---|---|
| **var**(a[, axis, dtype, out, ddof, keepdims, where]) | Compute the variance along the specified axis. |
| **vdot**(a, b, *[, precision]) | Return the dot product of two vectors. |
| **vectorize**(pyfunc, *[, excluded, signature]) | Define a vectorized function with broadcasting. |
| **vsplit**(ary, indices_or_sections) | Split an array into multiple sub-arrays vertically (row-wise). |
| **vstack**(tup[, dtype]) | Stack arrays in sequence vertically (row wise). |
| **where**() | Return elements chosen from *x* or *y* depending on *condition*. |
| **zeros**(shape[, dtype]) | Return a new array of given shape and type, filled with zeros. |
| **zeros_like**(a[, dtype, shape]) | Return an array of zeros with the same shape and type as a given array. |

# jax.numpy.fft

| | |
|---|---|
| **fft**(a[, n, axis, norm]) | Compute the one-dimensional discrete Fourier Transform. |
| **fft2**(a[, s, axes, norm]) | Compute the 2-dimensional discrete Fourier Transform. |
| **fftfreq**(n[, d, dtype]) | Return the Discrete Fourier Transform sample frequencies. |
| **fftn**(a[, s, axes, norm]) | Compute the N-dimensional discrete Fourier Transform. |
| **fftshift**(x[, axes]) | Shift the zero-frequency component to the center of the spectrum. |
| **hfft**(a[, n, axis, norm]) | Compute the FFT of a signal that has Hermitian symmetry, i.e., a real |
| **ifft**(a[, n, axis, norm]) | Compute the one-dimensional inverse discrete Fourier Transform. |
| **ifft2**(a[, s, axes, norm]) | Compute the 2-dimensional inverse discrete Fourier Transform. |
| **ifftn**(a[, s, axes, norm]) | Compute the N-dimensional inverse discrete Fourier Transform. |
| **ifftshift**(x[, axes]) | The inverse of *fftshift*. |
| **ihfft**(a[, n, axis, norm]) | Compute the inverse FFT of a signal that has Hermitian symmetry. |
| **irfft**(a[, n, axis, norm]) | Computes the inverse of *rfft*. |
| **irfft2**(a[, s, axes, norm]) | Computes the inverse of *rfft2*. |
| **irfftn**(a[, s, axes, norm]) | Computes the inverse of *rfftn*. |
| **rfft**(a[, n, axis, norm]) | Compute the one-dimensional discrete Fourier Transform for real input. |
| **rfft2**(a[, s, axes, norm]) | Compute the 2-dimensional FFT of a real array. |

| | |
|---|---|
| `rfftfreq`(n[, d, dtype]) | Return the Discrete Fourier Transform sample frequencies |
| `rfftn`(a[, s, axes, norm]) | Compute the N-dimensional discrete Fourier Transform for real input. |

# jax.numpy.linalg

| | |
|---|---|
| `cholesky`(a) | Cholesky decomposition. |
| `cond`(x[, p]) | Compute the condition number of a matrix. |
| `det`(a) | Compute the determinant of an array. |
| `eig`(a) | Compute the eigenvalues and right eigenvectors of a square array. |
| `eigh`(a[, UPLO, symmetrize_input]) | Return the eigenvalues and eigenvectors of a complex Hermitian |
| `eigvals`(a) | Compute the eigenvalues of a general matrix. |
| `eigvalsh`(a[, UPLO]) | Compute the eigenvalues of a complex Hermitian or real symmetric matrix. |
| `inv`(a) | Compute the (multiplicative) inverse of a matrix. |
| `lstsq`(a, b[, rcond, numpy_resid]) | Return the least-squares solution to a linear matrix equation. |
| `matrix_power`(a, n) | Raise a square matrix to the (integer) power *n*. |
| `matrix_rank`(M[, tol]) | Return matrix rank of array using SVD method |
| `multi_dot`(arrays, *[, precision]) | Compute the dot product of two or more arrays in a single function call, |
| `norm`(x[, ord, axis, keepdims]) | Matrix or vector norm. |
| `pinv`(a[, rcond, hermitian]) | Compute the (Moore-Penrose) pseudo-inverse of a matrix. |
| `qr`() | Compute the qr factorization of a matrix. |
| `slogdet`(a, *[, method]) | Compute the sign and (natural) logarithm of the determinant of an array. |

| | |
|---|---|
| `solve`(a, b) | Solve a linear matrix equation, or system of linear scalar equations. |
| `svd`() | Singular Value Decomposition. |
| `tensorinv`(a[, ind]) | Compute the 'inverse' of an N-dimensional array. |
| `tensorsolve`(a, b[, axes]) | Solve the tensor equation `a x = b` for x. |

# JAX DeviceArray

The JAX `DeviceArray` is the core array object in JAX: you can think of it as the equivalent of a `numpy.ndarray` backed by a memory buffer on a single device. Like `numpy.ndarray`, most users will not need to instantiate `DeviceArray` objects manually, but rather will create them via `jax.numpy` functions like `array()`, `arange()`, `linspace()`, and others listed above.

## Copying and Serialization

`DeviceArray` objects are designed to work seamlessly with Python standard library tools where appropriate.

With the built-in `copy` module, when `copy.copy()` or `copy.deepcopy()` encounder a `DeviceArray`, it is equivalent to calling the `copy()` method, which will create a copy of the buffer on the same device as the original array. This will work correctly within traced/JIT-compiled code, though copy operations may be elided by the compiler in this context.

When the built-in `pickle` module encounters a `DeviceArray`, it will be serialized via a compact bit representation in a similar manner to pickled `numpy.ndarray` objects. When unpickled, the result will be a new `DeviceArray` object *on the default device.* This is because in general, pickling and unpickling may take place in different runtime environments, and there is no general way to map the device IDs of one runtime to the device IDs of another. If `pickle` is used in traced/JIT-compiled code, it will result in a `ConcretizationTypeError`.

## Class Reference

**jax.numpy.DeviceArray**

alias of `jaxlib.xla_extension.DeviceArrayBase`

***class*** **jaxlib.xla_extension.DeviceArrayBase**

## *class* jaxlib.xla_extension.DeviceArray

### *property* T: *jax.Array*

Returns an array with axes transposed.

LAX-backend implementation of `numpy.transpose()`.

The JAX version of this function may in some cases return a copy rather than a view of the input.

*Original docstring below.*

For a 1-D array, this returns an unchanged view of the original array, as a transposed vector is simply the same vector. To convert a 1-D array into a 2-D column vector, an additional dimension must be added, e.g., `np.atleast2d(a).T` achieves this, as does `a[:, np.newaxis]`. For a 2-D array, this is the standard matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided, then `transpose(a).shape == a.shape[::-1]`.

**Parameters:**
- **a** (*array_like*) – Input array.
- **axes** (*tuple or list of ints, optional*) – If specified, it must be a tuple or list which contains a permutation of [0,1,…,N-1] where N is the number of axes of *a*. The *i*'th axis of the returned array will correspond to the axis numbered `axes[i]` of the input. If not specified, defaults to `range(a.ndim)[::-1]`, which reverses the order of the axes.

**Returns:** **p** – *a* with its axes permuted. A view is returned whenever possible.

**Return type:** ndarray

### all(*axis=None, out=None, keepdims=False, *, where=None*)

Test whether all array elements along a given axis evaluate to True.

LAX-backend implementation of `numpy.all()`.

*Original docstring below.*

**Parameters:**
- **a** (*array_like*) – Input array or object that can be converted to an array.
- **axis** (*None or int or tuple of ints, optional*) – Axis or axes along which a logical AND reduction is performed. The default (`axis=None`) is to perform a logical AND over all the dimensions of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

- **keepdims** (*bool, optional*) –
  If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.
  If the default value is passed, then *keepdims* will not be passed through to the *all* method of sub-classes of *ndarray*, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.
- **where** (*array_like of bool, optional*) – Elements to include in checking for all *True* values. See *~numpy.ufunc.reduce* for details.
- **out** (**None**) –

**Returns:** **all** – A new boolean or array is returned unless *out* is specified, in which case a reference to *out* is returned.

**Return type:** ndarray, bool

### any(*axis=None, out=None, keepdims=False, *, where=None*)

Test whether any array element along a given axis evaluates to True.

LAX-backend implementation of **numpy.any()**.

*Original docstring below.*

Returns single boolean if *axis* is `None`

**Parameters:**

- **a** (*array_like*) – Input array or object that can be converted to an array.
- **axis** (*None or int or tuple of ints, optional*) – Axis or axes along which a logical OR reduction is performed. The default (`axis=None`) is to perform a logical OR over all the dimensions of the input array. *axis* may be negative, in which case it counts from the last to the first axis.
- **keepdims** (*bool, optional*) –
  If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.
  If the default value is passed, then *keepdims* will not be passed through to the *any* method of sub-classes of *ndarray*, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

- **where** (*array_like of bool, optional*) – Elements to include in checking for any *True* values. See *~numpy.ufunc.reduce* for details.
- **out** (**None**) –

**Returns:** **any** – A new boolean or *ndarray* is returned unless *out* is specified, in which case a reference to *out* is returned.

**Return type:** bool or ndarray

### argmax(*axis=None, out=None, keepdims=None*)

Returns the indices of the maximum values along an axis.

LAX-backend implementation of **numpy.argmax()**.

*Original docstring below.*

**Parameters:**
- **a** (*array_like*) – Input array.
- **axis** (*int, optional*) – By default, the index is into the flattened array, otherwise along the specified axis.
- **keepdims** (*bool, optional*) – If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the array.

**Returns:** **index_array** – Array of indices into the array. It has the same shape as *a.shape* with the dimension along *axis* removed. If *keepdims* is set to True, then the size of *axis* will be 1 with the resulting array having same shape as *a.shape*.

**Return type:** ndarray of ints

### argmin(*axis=None, out=None, keepdims=None*)

Returns the indices of the minimum values along an axis.

LAX-backend implementation of **numpy.argmin()**.

*Original docstring below.*

**Parameters:**
- **a** (*array_like*) – Input array.
- **axis** (*int, optional*) – By default, the index is into the flattened array, otherwise along the specified axis.
- **keepdims** (*bool, optional*) – If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the array.

**Returns:** **index_array** – Array of indices into the array. It has the same shape as *a.shape* with the dimension along *axis* removed. If *keepdims* is set to True, then the size of *axis* will be 1 with the resulting array having same shape as *a.shape*.

**Return type:** ndarray of ints

### `argpartition(**kwargs)`

Perform an indirect partition along the given axis using the

LAX-backend implementation of `numpy.argpartition()`.

*** This function is not yet implemented by jax.numpy, and will raise NotImplementedError ***

*Original docstring below.*

algorithm specified by the *kind* keyword. It returns an array of indices of the same shape as *a* that index data along the given axis in partitioned order.

> ***New in version 1.8.0.***

**Parameters:**

- **a** (*array_like*) – Array to sort.
- **kth** (*int or sequence of ints*) –
  Element index to partition by. The k-th element will be in its final sorted position and all smaller elements will be moved before it and all larger elements behind it. The order of all elements in the partitions is undefined. If provided with a sequence of k-th it will partition all of them into their sorted position at once.

  > ***Deprecated since version 1.22.0:*** Passing booleans as index is deprecated.

- **axis** (*int or None, optional*) – Axis along which to sort. The default is -1 (the last axis). If None, the flattened array is used.
- **kind** (*{'introselect'}, optional*) – Selection algorithm. Default is ‘introselect’
- **order** (*str or list of str, optional*) – When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all

fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

**Returns:** **index_array** – Array of indices that partition *a* along the specified axis. If *a* is one-dimensional, `a[index_array]` yields a partitioned *a*. More generally, `np.take_along_axis(a, index_array, axis=axis)` always yields the partitioned *a*, irrespective of dimensionality.

**Return type:** ndarray, int

### `argsort(axis=- 1, kind='stable', order=None)`

Returns the indices that would sort an array.

LAX-backend implementation of **`numpy.argsort()`**.

Only `kind='stable'` is supported. Other `kind` values will produce a warning and be treated as if they were `'stable'`.

*Original docstring below.*

Perform an indirect sort along the given axis using the algorithm specified by the *kind* keyword. It returns an array of indices of the same shape as *a* that index data along the given axis in sorted order.

**Parameters:**

- **a** (*array_like*) – Array to sort.
- **axis** (*int or None, optional*) – Axis along which to sort. The default is -1 (the last axis). If None, the flattened array is used.
- **kind** (*{'quicksort', 'mergesort', 'heapsort', 'stable'}, optional*) – Sorting algorithm. The default is ‘quicksort’. Note that both ‘stable’ and ‘mergesort’ use timsort under the covers and, in general, the actual implementation will vary with data type. The ‘mergesort’ option is retained for backwards compatibility.

  > **Changed in version 1.15.0.:** The ‘stable’ option was added.

- **order** (*str or list of str, optional*) – When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

| | |
|---|---|
| **Returns:** | **index_array** – Array of indices that sort *a* along the specified *axis*. If *a* is one-dimensional, `a[index_array]` yields a sorted *a*. More generally, `np.take_along_axis(a, index_array, axis=axis)` always yields the sorted *a*, irrespective of dimensionality. |
| **Return type:** | ndarray, int |

### `astype(dtype)`

Copy the array and cast to a specified dtype.

This is implemeted via **`jax.lax.convert_element_type()`**, which may have slightly different behavior than **`numpy.ndarray.astype()`** in some cases. In particular, the details of float-to-int and int-to-float casts are implementation dependent.

| | |
|---|---|
| **Parameters:** | • **arr** (**`Union`**[**`Array`**, **`ndarray`**, **`bool_`**, **`number`**, **`bool`**, **`int`**, **`float`**, **`complex`**]) –<br>• **dtype** (**`Union`**[**`Any`**, **`str`**, **`dtype`**, **`SupportsDType`**]) – |
| **Return type:** | **`Array`** |

### *property* `at`

Helper property for index update functionality.

The `at` property provides a functionally pure equivalent of in-place array modificatons.

In particular:

| Alternate syntax | Equivalent In-place expression |
|---|---|
| `x = x.at[idx].set(y)` | `x[idx] = y` |
| `x = x.at[idx].add(y)` | `x[idx] += y` |
| `x = x.at[idx].multiply(y)` | `x[idx] *= y` |
| `x = x.at[idx].divide(y)` | `x[idx] /= y` |
| `x = x.at[idx].power(y)` | `x[idx] **= y` |
| `x = x.at[idx].min(y)` | `x[idx] = minimum(x[idx], y)` |
| `x = x.at[idx].max(y)` | `x[idx] = maximum(x[idx], y)` |
| `x = x.at[idx].apply(ufunc)` | `ufunc.at(x, idx)` |
| `x = x.at[idx].get()` | `x = x[idx]` |

None of the `x.at` expressions modify the original `x`; instead they return a modified copy of `x`. However, inside a **`jit()`** compiled function, expressions like `x = x.at[idx].set(y)` are guaranteed to be applied in-place.

Unlike NumPy in-place operations such as `x[idx] += y`, if multiple indices refer to the same location, all updates will be applied (NumPy would only apply the last update, rather than applying all updates.) The order in which conflicting updates are applied is implementation-defined and may be nondeterministic (e.g., due to concurrency on some hardware platforms).

By default, JAX assumes that all indices are in-bounds. Alternative out-of-bound index semantics can be specified via the `mode` parameter (see below).

**Parameters:**

- **mode** (*str*) –
  Specify out-of-bound indexing mode. Options are:
  - `"promise_in_bounds"`: (default) The user promises that indices are in bounds. No additional checking will be performed. In practice, this means that out-of-bounds indices in `get()` will be clipped, and out-of-bounds indices in `set()`, `add()`, etc. will be dropped.
  - `"clip"`: clamp out of bounds indices into valid range.

- `"drop"`: ignore out-of-bound indices.
- `"fill"`: alias for `"drop"`. For *get()*, the optional `fill_value` argument specifies the value that will be returned.
  See `jax.lax.GatherScatterMode` for more details.

- **indices_are_sorted** (*bool*) – If True, the implementation will assume that the indices passed to `at[]` are sorted in ascending order, which can lead to more efficient execution on some backends.
- **unique_indices** (*bool*) – If True, the implementation will assume that the indices passed to `at[]` are unique, which can result in more efficient execution on some backends.
- **fill_value** (*Any*) – Only applies to the `get()` method: the fill value to return for out-of-bounds slices when *mode* is `'fill'`. Ignored otherwise. Defaults to `NaN` for inexact types, the largest negative value for signed types, the largest positive value for unsigned types, and `True` for booleans.

**Examples**

```
>>> x = jnp.arange(5.0)
>>> x
Array([0., 1., 2., 3., 4.], dtype=float32)
>>> x.at[2].add(10)
Array([ 0.,  1., 12.,  3.,  4.], dtype=float32)
>>> x.at[10].add(10)  # out-of-bounds indices are ignored
Array([0., 1., 2., 3., 4.], dtype=float32)
>>> x.at[20].add(10, mode='clip')
Array([ 0.,  1.,  2.,  3., 14.], dtype=float32)
>>> x.at[2].get()
Array(2., dtype=float32)
>>> x.at[20].get()  # out-of-bounds indices clipped
Array(4., dtype=float32)
>>> x.at[20].get(mode='fill')  # out-of-bounds indices filled with NaN
Array(nan, dtype=float32)
>>> x.at[20].get(mode='fill', fill_value=-1)  # custom fill value
Array(-1., dtype=float32)
```

## `block_until_ready()`

(self: xla::PyBuffer::pyobject) -> StatusOr[xla::PyBuffer::pyobject]

## `broadcast(sizes)`

Broadcasts an array, adding new leading dimensions

**Parameters:**

- **operand** (`Union`[`Array`, `ndarray`, `bool_`, `number`, `bool`, `int`, `float`, `complex`]) – an array

- **sizes** (Sequence[int]) – a sequence of integers, giving the sizes of new leading dimensions to add to the front of the array.

**Return type:** Array

**Returns:** An array containing the result.

> **See also**
>
> jax.lax.broadcast_in_dim : add new dimensions at any location in the array shape.

### broadcast_in_dim(*shape, broadcast_dimensions*)

Wraps XLA's BroadcastInDim operator.

**Parameters:**

- **operand** (Union[Array, ndarray, bool_, number, bool, int, float, complex]) – an array
- **shape** (Sequence[Union[int, Any]]) – the shape of the target array
- **broadcast_dimensions** (Sequence[int]) – to which dimension in the target shape each dimension of the operand shape corresponds to

**Return type:** Array

**Returns:** An array containing the result.

> **See also**
>
> jax.lax.broadcast : simpler interface to add new leading dimensions.

### choose(*choices, out=None, mode='raise'*)

Construct an array from an index array and a list of arrays to choose from.

LAX-backend implementation of `numpy.choose()`.

*Original docstring below.*

First of all, if confused or uncertain, definitely look at the Examples - in its full generality, this function is less simple than it might seem from the following code description (below ndi = *numpy.lib.index_tricks*):

`np.choose(a,c) == np.array([c[a[I]][I] for I in ndi.ndindex(a.shape)])`.

But this omits some subtleties. Here is a fully general summary:

Given an “index” array (*a*) of integers and a sequence of `n` arrays (*choices*), *a* and each choice array are first broadcast, as necessary, to arrays of a common shape; calling these *Ba* and *Bchoices[i], i = 0,…,n-1* we have that, necessarily, `Ba.shape == Bchoices[i].shape` for each `i`. Then, a new array with shape `Ba.shape` is created as follows:

- if `mode='raise'` (the default), then, first of all, each element of `a` (and thus `Ba`) must be in the range `[0, n-1]`; now, suppose that `i` (in that range) is the value at the `(j0, j1, ..., jm)` position in `Ba` - then the value at the same position in the new array is the value in `Bchoices[i]` at that same position;
- if `mode='wrap'`, values in *a* (and thus *Ba*) may be any (signed) integer; modular arithmetic is used to map integers outside the range *[0, n-1]* back into that range; and then the new array is constructed as above;
- if `mode='clip'`, values in *a* (and thus `Ba`) may be any (signed) integer; negative integers are mapped to 0; values greater than `n-1` are mapped to `n-1`; and then the new array is constructed as above.

**Parameters:**

- **a** (*int array*) – This array must contain integers in `[0, n-1]`, where `n` is the number of choices, unless `mode=wrap` or `mode=clip`, in which cases any integers are permissible.
- **choices** (*sequence of arrays*) – Choice arrays. *a* and all of the choices must be broadcastable to the same shape. If *choices* is itself an array (not recommended), then its outermost dimension (i.e., the one corresponding to `choices.shape[0]`) is taken as defining the “sequence”.
- **mode** (*{'raise' (default), 'wrap', 'clip'}, optional*) – Specifies how indices outside `[0, n-1]` will be treated:

  - ’raise’ : an exception is raised
  - ’wrap’ : value becomes value mod `n`
  - ’clip’ : values < 0 are mapped to 0, values > n-1 are mapped to n-1

- **out** (**None**) –

**Returns:** **merged_array** – The merged result.

**Return type:** array

## `clone()`

(self: xla::PyBuffer::pyobject) -> xla::PyBuffer::pyobject

## `conj()`

Return the complex conjugate, element-wise.

LAX-backend implementation of `numpy.conjugate()`.

*Original docstring below.*

The complex conjugate of a complex number is obtained by changing the sign of its imaginary part.

| | |
|---|---|
| **Parameters:** | **x** (*array_like*) – Input value. |
| **Returns:** | **y** – The complex conjugate of *x*, with same dtype as *y*. This is a scalar if *x* is a scalar. |
| **Return type:** | ndarray |

## `conjugate()`

Return the complex conjugate, element-wise.

LAX-backend implementation of `numpy.conjugate()`.

*Original docstring below.*

The complex conjugate of a complex number is obtained by changing the sign of its imaginary part.

| | |
|---|---|
| **Parameters:** | **x** (*array_like*) – Input value. |
| **Returns:** | **y** – The complex conjugate of *x*, with same dtype as *y*. This is a scalar if *x* is a scalar. |
| **Return type:** | ndarray |

## `copy(order=None)`

Return an array copy of the given object.

LAX-backend implementation of `numpy.copy()`.

This function will create arrays on JAX's default device. For control of the device placement of data, see `jax.device_put()`. More information is available in the JAX FAQ at Controlling data and computation placement on devices (full FAQ at https://jax.readthedocs.io/en/latest/faq.html).

*Original docstring below.*

| | |
|---|---|
| **Parameters:** | • **a** (*array_like*) – Input data. |

- **order** (*{'C', 'F', 'A', 'K'}, optional*) – Controls the memory layout of the copy. ‘C’ means C-order, ‘F’ means F-order, ‘A’ means ‘F’ if *a* is Fortran contiguous, ‘C’ otherwise. ‘K’ means match the layout of *a* as closely as possible. (Note that this function and `ndarray.copy()` are very similar, but have different default values for their order= arguments.)

**Returns:** **arr** – Array interpretation of *a*.

**Return type:** ndarray

### `copy_to_device()`

(self: xla::PyBuffer::pyobject, arg0: jaxlib.xla_extension.Device) -> StatusOr[object]

### `copy_to_host_async()`

(self: xla::PyBuffer::pyobject) -> Status

### `copy_to_remote_device()`

(self: xla::PyBuffer::pyobject, arg0: bytes) -> Tuple[Status, bool]

### `cumprod(axis=None, dtype=None, out=None)`

Return the cumulative product of elements along a given axis.

LAX-backend implementation of `numpy.cumprod()`.

*Original docstring below.*

**Parameters:**

- **a** (*array_like*) – Input array.
- **axis** (*int, optional*) – Axis along which the cumulative product is computed. By default the input is flattened.
- **dtype** (*dtype, optional*) – Type of the returned array, as well as of the accumulator in which the elements are multiplied. If *dtype* is not specified, it defaults to the dtype of *a*, unless *a* has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used instead.
- **out** (`None`) –

**Returns:** **cumprod** – A new array holding the result is returned unless *out* is specified, in which case a reference to out is returned.

**Return type:** ndarray

### `cumsum(axis=None, dtype=None, out=None)`

Return the cumulative sum of the elements along a given axis.

LAX-backend implementation of `numpy.cumsum()`.

*Original docstring below.*

**Parameters:**
- **a** (*array_like*) – Input array.
- **axis** (*int*, *optional*) – Axis along which the cumulative sum is computed. The default (None) is to compute the cumsum over the flattened array.
- **dtype** (*dtype*, *optional*) – Type of the returned array and of the accumulator in which the elements are summed. If *dtype* is not specified, it defaults to the dtype of *a*, unless *a* has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.
- **out** (**None**) –

**Returns:** **cumsum_along_axis** – A new array holding the result is returned unless *out* is specified, in which case a reference to *out* is returned. The result has the same size as *a*, and the same shape as *a* if *axis* is not None or *a* is a 1-d array.

**Return type:** ndarray.

### `delete()`

(self: xla::PyBuffer::pyobject) -> None

### `device()`

(self: xla::PyBuffer::pyobject) -> jaxlib.xla_extension.Device

### `diagonal(offset=0, axis1=0, axis2=1)`

Return specified diagonals.

LAX-backend implementation of **`numpy.diagonal()`**.

The JAX version of this function may in some cases return a copy rather than a view of the input.

*Original docstring below.*

If *a* is 2-D, returns the diagonal of *a* with the given offset, i.e., the collection of elements of the form `a[i, i+offset]`. If *a* has more than two dimensions, then the axes specified by *axis1* and *axis2* are used to determine the 2-D sub-array whose diagonal is returned. The shape of the resulting array can be determined by removing *axis1* and *axis2* and appending an index to the right equal to the size of the resulting diagonals.

In versions of NumPy prior to 1.7, this function always returned a new, independent array containing a copy of the values in the diagonal.

In NumPy 1.7 and 1.8, it continues to return a copy of the diagonal, but depending on this fact is deprecated. Writing to the resulting array continues to work as it used to, but a FutureWarning is issued.

Starting in NumPy 1.9 it returns a read-only view on the original array. Attempting to write to the resulting array will produce an error.

In some future release, it will return a read/write view and writing to the returned array will alter your original array. The returned array will have the same type as the input array.

If you don't write to the array returned by this function, then you can just ignore all of the above.

If you depend on the current behavior, then we suggest copying the returned array explicitly, i.e., use `np.diagonal(a).copy()` instead of just `np.diagonal(a)`. This will work with both past and future versions of NumPy.

**Parameters:**

- **a** (*array_like*) – Array from which the diagonals are taken.
- **offset** (*int, optional*) – Offset of the diagonal from the main diagonal. Can be positive or negative. Defaults to main diagonal (0).
- **axis1** (*int, optional*) – Axis to be used as the first axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults to first axis (0).
- **axis2** (*int, optional*) – Axis to be used as the second axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults to second axis (1).

**Returns:** **array_of_diagonals** – If *a* is 2-D, then a 1-D array containing the diagonal and of the same type as *a* is returned unless *a* is a *matrix*, in which case a 1-D array rather than a (2-D) *matrix* is returned in order to maintain backward compatibility.
If `a.ndim > 2`, then the dimensions specified by *axis1* and *axis2* are removed, and a new axis inserted at the end corresponding to the diagonal.

**Return type:** ndarray

## `dot(b, *, precision=None)`

Dot product of two arrays. Specifically,

LAX-backend implementation of **`numpy.dot()`**.

In addition to the original NumPy arguments listed below, also supports `precision` for extra control over matrix-multiplication precision on supported devices. `precision` may be set to `None`, which means default precision for the backend, a **`Precision`** enum value (`Precision.DEFAULT`, `Precision.HIGH` or `Precision.HIGHEST`) or a tuple of two **`Precision`** enums indicating separate precision for each argument.

*Original docstring below.*

- If both *a* and *b* are 1-D arrays, it is inner product of vectors (without complex conjugation).
- If both *a* and *b* are 2-D arrays, it is matrix multiplication, but using **`matmul()`** or `a @ b` is preferred.
- If either *a* or *b* is 0-D (scalar), it is equivalent to **`multiply()`** and using `numpy.multiply(a, b)` or `a * b` is preferred.
- If *a* is an N-D array and *b* is a 1-D array, it is a sum product over the last axis of *a* and *b*.
- If *a* is an N-D array and *b* is an M-D array (where `M>=2`), it is a sum product over the last axis of *a* and the second-to-last axis of *b*:

```
dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
```

It uses an optimized BLAS library when possible (see *numpy.linalg*).

| | |
|---|---|
| **Parameters:** | • **a** (*array_like*) – First argument.<br>• **b** (*array_like*) – Second argument. |
| **Returns:** | **output** – Returns the dot product of *a* and *b*. If *a* and *b* are both scalars or both 1-D arrays then a scalar is returned; otherwise an array is returned. If *out* is given, then it is returned. |
| **Return type:** | ndarray |

## `flatten(order='C')`

Return a contiguous flattened array.

LAX-backend implementation of **`numpy.ravel()`**.

The JAX version of this function may in some cases return a copy rather than a view of the input.

*Original docstring below.*

A 1-D array, containing the elements of the input, is returned. A copy is made only if needed.

As of NumPy 1.10, the returned array will have the same type as the input array. (for example, a masked array will be returned for a masked array input)

**Parameters:**
- **a** (*array_like*) – Input array. The elements in *a* are read in the order specified by *order*, and packed as a 1-D array.
- **order** (*{'C','F', 'A', 'K'}, optional*) – The elements of *a* are read using this index order. 'C' means to index the elements in row-major, C-style order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to index the elements in column-major, Fortran-style order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of axis indexing. 'A' means to read the elements in Fortran-like index order if *a* is Fortran *contiguous* in memory, C-like order otherwise. 'K' means to read the elements in the order they occur in memory, except for reversing the data when strides are negative. By default, 'C' index order is used.

**Returns:** **y** – y is an array of the same subtype as *a*, with shape `(a.size,)`. Note that matrices are special cased for backward compatibility, if *a* is a matrix, then y is a 1-D ndarray.

**Return type:** array_like

### *property* imag: *jax.Array*

Return the imaginary part of the complex argument.

LAX-backend implementation of **numpy.imag()**.

*Original docstring below.*

**Parameters:** **val** (*array_like*) – Input array.

**Returns:** **out** – The imaginary component of the complex argument. If *val* is real, the type of *val* is used for the output. If *val* has complex elements, the returned type is float.

**Return type:** ndarray or scalar

### is_deleted()

(self: xla::PyBuffer::pyobject) -> bool

### is_known_ready()

(self: xla::PyBuffer::pyobject) -> StatusOr[bool]

### `is_ready()`

(self: xla::PyBuffer::pyobject) -> StatusOr[bool]

### `max(axis=None, out=None, keepdims=False, initial=None, where=None)`

Return the maximum of an array or maximum along an axis.

LAX-backend implementation of `numpy.amax()`.

*Original docstring below.*

**Parameters:**

- **a** (*array_like*) – Input data.
- **axis** (*None or int or tuple of ints, optional*) – Axis or axes along which to operate. By default, flattened input is used.
- **keepdims** (*bool, optional*) –
  If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.
  If the default value is passed, then *keepdims* will not be passed through to the *amax* method of sub-classes of *ndarray*, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.
- **initial** (*scalar, optional*) – The minimum value of an output element. Must be present to allow computation on empty slice. See *~numpy.ufunc.reduce* for details.
- **where** (*array_like of bool, optional*) – Elements to compare for the maximum. See *~numpy.ufunc.reduce* for details.
- **out** (`None`) –

**Returns:** **amax** – Maximum of *a*. If *axis* is None, the result is a scalar value. If *axis* is an int, the result is an array of dimension `a.ndim - 1`. If *axis* is a tuple, the result is an array of dimension `a.ndim - len(axis)`.

**Return type:** ndarray or scalar

### `mean(axis=None, dtype=None, out=None, keepdims=False, *, where=None)`

Compute the arithmetic mean along the specified axis.

LAX-backend implementation of `numpy.mean()`.

*Original docstring below.*

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. *float64* intermediate and return values are used for integer inputs.

**Parameters:**

- **a** (*array_like*) – Array containing numbers whose mean is desired. If *a* is not an array, a conversion is attempted.
- **axis** (*None or int or tuple of ints, optional*) – Axis or axes along which the means are computed. The default is to compute the mean of the flattened array.
- **dtype** (*data-type, optional*) – Type to use in computing the mean. For integer inputs, the default is *float64*; for floating point inputs, it is the same as the input dtype.
- **keepdims** (*bool, optional*) –
  If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.
  If the default value is passed, then *keepdims* will not be passed through to the *mean* method of sub-classes of *ndarray*, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.
- **where** (*array_like of bool, optional*) – Elements to include in the mean. See *~numpy.ufunc.reduce* for details.
- **out** (**None**) –

**Returns:** **m** – If *out=None*, returns a new array containing the mean values, otherwise a reference to the output array is returned.

**Return type:** ndarray, see dtype parameter above

### `min(axis=None, out=None, keepdims=False, initial=None, where=None)`

Return the minimum of an array or minimum along an axis.

LAX-backend implementation of **`numpy.amin()`**.

*Original docstring below.*

**Parameters:**

- **a** (*array_like*) – Input data.
- **axis** (*None or int or tuple of ints, optional*) – Axis or axes along which to operate. By default, flattened input is used.
- **keepdims** (*bool, optional*) –
  If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then *keepdims* will not be passed through to the *amin* method of sub-classes of *ndarray*, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

- **initial** (*scalar, optional*) – The maximum value of an output element. Must be present to allow computation on empty slice. See *~numpy.ufunc.reduce* for details.
- **where** (*array_like of bool, optional*) – Elements to compare for the minimum. See *~numpy.ufunc.reduce* for details.
- **out** (`None`) –

**Returns:** **amin** – Minimum of *a*. If *axis* is None, the result is a scalar value. If *axis* is an int, the result is an array of dimension `a.ndim - 1`. If *axis* is a tuple, the result is an array of dimension `a.ndim - len(axis)`.

**Return type:** ndarray or scalar

### `nonzero(*, size=None, fill_value=None)`

Return the indices of the elements that are non-zero.

LAX-backend implementation of `numpy.nonzero()`.

Because the size of the output of `nonzero` is data-dependent, the function is not typically compatible with JIT. The JAX version adds the optional `size` argument which must be specified statically for `jnp.nonzero` to be used within some of JAX's transformations.

*Original docstring below.*

Returns a tuple of arrays, one for each dimension of *a*, containing the indices of the non-zero elements in that dimension. The values in *a* are always tested and returned in row-major, C-style order.

To group the indices by element, rather than dimension, use *argwhere*, which returns a row for each non-zero element.

> **Note**
>
> When called on a zero-d array or scalar, `nonzero(a)` is treated as `nonzero(atleast_1d(a))`.
>
> > ***Deprecated since version 1.17.0:*** Use *atleast_1d* explicitly if this behavior is deliberate.

**Parameters:**

- **a** (*array_like*) – Input array.
- **size** (*int, optional*) – If specified, the indices of the first `size` True elements will be returned. If there are fewer unique elements than `size` indicates, the return value will be padded with `fill_value`.
- **fill_value** (*array_like, optional*) – When `size` is specified and there are fewer than the indicated number of elements, the remaining elements will be filled with `fill_value`, which defaults to zero.

**Returns:** **tuple_of_arrays** – Indices of elements that are non-zero.

**Return type:** tuple

### `on_device_size_in_bytes()`

(self: xla::PyBuffer::pyobject) -> StatusOr[int]

### `platform()`

(self: xla::PyBuffer::pyobject) -> str

### `prod(axis=None, dtype=None, out=None, keepdims=False, initial=None, where=None, promote_integers=True)`

Return the product of array elements over a given axis.

LAX-backend implementation of **`numpy.prod()`**.

*Original docstring below.*

**Parameters:**

- **a** (*array_like*) – Input data.
- **axis** (*None or int or tuple of ints, optional*) – Axis or axes along which a product is performed. The default, axis=None, will calculate the product of all the elements in the input array. If axis is negative it counts from the last to the first axis.
- **dtype** (*dtype, optional*) – The type of the returned array, as well as of the accumulator in which the elements are multiplied. The dtype of *a* is used by default unless *a* has an integer dtype of less precision than the default platform integer. In that case, if *a* is signed then the platform integer is used while if *a* is unsigned then an unsigned integer of the same precision as the platform integer is used.
- **keepdims** (*bool, optional*) –

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.
If the default value is passed, then *keepdims* will not be passed through to the *prod* method of sub-classes of *ndarray*, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

- **initial** (*scalar, optional*) – The starting value for this product. See *~numpy.ufunc.reduce* for details.
- **where** (*array_like of bool, optional*) – Elements to include in the product. See *~numpy.ufunc.reduce* for details.
- **promote_integers** (*bool, default=True*) – If True, then integer inputs will be promoted to the widest available integer dtype, following numpy's behavior. If False, the result will have the same dtype as the input. `promote_integers` is ignored if `dtype` is specified.
- **out** (`None`) –

**Returns:** **product_along_axis** – An array shaped as *a* but with the specified axis removed. Returns a reference to *out* if specified.

**Return type:** ndarray, see *dtype* parameter above.

### `ptp(axis=None, out=None, keepdims=False)`

Range of values (maximum - minimum) along an axis.

LAX-backend implementation of `numpy.ptp()`.

*Original docstring below.*

The name of the function comes from the acronym for 'peak to peak'.

> **Warning**
>
> *ptp* preserves the data type of the array. This means the return value for an input of signed integers with n bits (e.g. *np.int8*, *np.int16*, etc) is also a signed integer with n bits. In that case, peak-to-peak values greater than `2**(n-1)-1` will be returned as negative values. An example with a work-around is shown below.

**Parameters:**

- **a** (*array_like*) – Input values.

- **axis** (*None or int or tuple of ints, optional*) – Axis along which to find the peaks. By default, flatten the array. *axis* may be negative, in which case it counts from the last to the first axis.
- **keepdims** (*bool, optional*) –
  If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.
  If the default value is passed, then *keepdims* will not be passed through to the *ptp* method of sub-classes of *ndarray*, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.
- **out** (`None`) –

**Returns:** **ptp** – The range of a given array - *scalar* if array is one-dimensional or a new array holding the result along the given axis

**Return type:** ndarray or scalar

### `ravel(order='C')`

Return a contiguous flattened array.

LAX-backend implementation of `numpy.ravel()`.

The JAX version of this function may in some cases return a copy rather than a view of the input.

*Original docstring below.*

A 1-D array, containing the elements of the input, is returned. A copy is made only if needed.

As of NumPy 1.10, the returned array will have the same type as the input array. (for example, a masked array will be returned for a masked array input)

**Parameters:**

- **a** (*array_like*) – Input array. The elements in *a* are read in the order specified by *order*, and packed as a 1-D array.
- **order** (*{'C','F', 'A', 'K'}, optional*) – The elements of *a* are read using this index order. 'C' means to index the elements in row-major, C-style order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to index the elements in column-major, Fortran-style order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of axis

indexing. ‘A’ means to read the elements in Fortran-like index order if *a* is Fortran *contiguous* in memory, C-like order otherwise. ‘K’ means to read the elements in the order they occur in memory, except for reversing the data when strides are negative. By default, ‘C’ index order is used.

**Returns:** **y** – y is an array of the same subtype as *a*, with shape `(a.size,)`. Note that matrices are special cased for backward compatibility, if *a* is a matrix, then y is a 1-D ndarray.

**Return type:** array_like

### *property* real: *jax.Array*

Return the real part of the complex argument.

LAX-backend implementation of **numpy.real()**.

*Original docstring below.*

**Parameters:** **val** (*array_like*) – Input array.

**Returns:** **out** – The real component of the complex argument. If *val* is real, the type of *val* is used for the output. If *val* has complex elements, the returned type is float.

**Return type:** ndarray or scalar

### repeat(*repeats, axis=None, *, total_repeat_length=None*)

Repeat elements of an array.

LAX-backend implementation of **numpy.repeat()**.

JAX adds the optional *total_repeat_length* parameter which specifies the total number of repeat, and defaults to sum(repeats). It must be specified for repeat to be compilable. If *sum(repeats)* is larger than the specified *total_repeat_length* the remaining values will be discarded. In the case of *sum(repeats)* being smaller than the specified target length, the final value will be repeated.

*Original docstring below.*

**Parameters:**

- **a** (*array_like*) – Input array.
- **repeats** (*int or array of ints*) – The number of repetitions for each element. *repeats* is broadcasted to fit the shape of the given axis.
- **axis** (*int, optional*) – The axis along which to repeat values. By default, use the flattened input array, and return a flat output array.

- **total_repeat_length** (`Optional`[`int`]) –

**Returns:** **repeated_array** – Output array which has the same shape as *a*, except along the given axis.

**Return type:** ndarray

### `round`(*`decimals=0, out=None`*)

Evenly round to the given number of decimals.

LAX-backend implementation of `numpy.around()`.

*Original docstring below.*

**Parameters:**
- **a** (*array_like*) – Input data.
- **decimals** (*int, optional*) – Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.

**Returns:** **rounded_array** – An array of the same type as *a*, containing the rounded values. Unless *out* was specified, a new array is created. A reference to the result is returned.
The real and imaginary parts of complex numbers are rounded separately. The result of rounding a float is a float.

**Return type:** ndarray

#### References

1 "Lecture Notes on the Status of IEEE 754", William Kahan, https://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF

**Parameters:** **out** (`None`) –

### `searchsorted`(*`v, side='left', sorter=None, *, method='scan'`*)

Find indices where elements should be inserted to maintain order.

LAX-backend implementation of `numpy.searchsorted()`.

*Original docstring below.*

Find the indices into a sorted array *a* such that, if the corresponding elements in *v* were inserted before the indices, the order of *a* would be preserved.

Assuming that *a* is sorted:

| *side* | **returned index *i* satisfies** |
|---|---|
| left | `a[i-1] < v <= a[i]` |
| right | `a[i-1] <= v < a[i]` |

**Parameters:**
- **a** (*1-D array_like*) – Input array. If *sorter* is None, then it must be sorted in ascending order, otherwise *sorter* must be an array of indices that sort it.
- **v** (*array_like*) – Values to insert into *a*.
- **side** (*{'left', 'right'}, optional*) – If ‘left’, the index of the first suitable location found is given. If ‘right’, return the last such index. If there is no suitable index, return either 0 or N (where N is the length of *a*).
- **method** (*str*) – One of ‘scan’ (default) or ‘sort’. Controls the method used by the implementation; ‘scan’ tends to be more performant on CPU (particularly when `a` is very large), while ‘sort’ is often more performant on accelerator backends like GPU and TPU (particularly when `v` is very large).
- **sorter** (**None**) –

**Returns:** **indices** – Array of insertion points with the same shape as *v*, or an integer if *v* is a scalar.

**Return type:** int or array of ints

## sort(*axis=- 1, kind='quicksort', order=None*)

Return a sorted copy of an array.

LAX-backend implementation of **numpy.sort()**.

*Original docstring below.*

**Parameters:**
- **a** (*array_like*) – Array to be sorted.
- **axis** (*int or None, optional*) – Axis along which to sort. If None, the array is flattened before sorting. The default is -1, which sorts along the last axis.
- **kind** (*{'quicksort', 'mergesort', 'heapsort', 'stable'}, optional*) – Sorting algorithm. The default is ‘quicksort’. Note that both ‘stable’ and ‘mergesort’ use timsort or radix sort under the covers and, in general, the actual implementation will vary with data

type. The ‘mergesort’ option is retained for backwards compatibility.

> **Changed in version 1.15.0.:** The ‘stable’ option was added.

- **order** (*str or list of str, optional*) – When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

**Returns:** **sorted_array** – Array of the same type and shape as *a*.

**Return type:** ndarray

### `split(indices_or_sections, axis=0)`

Split an array into multiple sub-arrays as views into *ary*.

LAX-backend implementation of `numpy.split()`.

The JAX version of this function may in some cases return a copy rather than a view of the input.

*Original docstring below.*

**Parameters:**

- **ary** (*ndarray*) – Array to be divided into sub-arrays.
- **indices_or_sections** (*int or 1-D array*) –
  If *indices_or_sections* is an integer, N, the array will be divided into N equal arrays along *axis*. If such a split is not possible, an error is raised.
  If *indices_or_sections* is a 1-D array of sorted integers, the entries indicate where along *axis* the array is split. For example, `[2, 3]` would, for `axis=0`, result in

  - ary[:2]
  - ary[2:3]
  - ary[3:]

  If an index exceeds the dimension of the array along *axis*, an empty sub-array is returned correspondingly.
- **axis** (*int, optional*) – The axis along which to split, default is 0.

**Returns:** **sub-arrays** – A list of sub-arrays as views into *ary*.

**Return type:** list of ndarrays

### `squeeze(axis=None)`

Remove axes of length one from *a*.

LAX-backend implementation of `numpy.squeeze()`.

The JAX version of this function may in some cases return a copy rather than a view of the input.

*Original docstring below.*

**Parameters:**
- **a** (*array_like*) – Input data.
- **axis** (*None or int or tuple of ints, optional*) –

**Returns:** **squeezed** – The input array, but with all or a subset of the dimensions of length 1 removed. This is always *a* itself or a view into *a*. Note that if all axes are squeezed, the result is a 0d array and not a scalar.

**Return type:** ndarray

### `std(axis=None, dtype=None, out=None, ddof=0, keepdims=False, *, where=None)`

Compute the standard deviation along the specified axis.

LAX-backend implementation of `numpy.std()`.

*Original docstring below.*

Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

**Parameters:**
- **a** (*array_like*) – Calculate the standard deviation of these values.
- **axis** (*None or int or tuple of ints, optional*) – Axis or axes along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.
- **dtype** (*dtype, optional*) – Type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.
- **ddof** (*int, optional*) – Means Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements. By default *ddof* is zero.
- **keepdims** (*bool, optional*) –

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.
If the default value is passed, then *keepdims* will not be passed through to the *std* method of sub-classes of *ndarray*, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

- **where** (*array_like of bool, optional*) – Elements to include in the standard deviation. See *~numpy.ufunc.reduce* for details.
- **out** (`None`) –

**Returns:** **standard_deviation** – If *out* is None, return a new array containing the standard deviation, otherwise return a reference to the output array.

**Return type:** ndarray, see dtype parameter above.

### `sum(axis=None, dtype=None, out=None, keepdims=False, initial=None, where=None, promote_integers=True)`

Sum of array elements over a given axis.

LAX-backend implementation of `numpy.sum()`.

*Original docstring below.*

**Parameters:**

- **a** (*array_like*) – Elements to sum.
- **axis** (*None or int or tuple of ints, optional*) – Axis or axes along which a sum is performed. The default, axis=None, will sum all of the elements of the input array. If axis is negative it counts from the last to the first axis.
- **dtype** (*dtype, optional*) – The type of the returned array and of the accumulator in which the elements are summed. The dtype of *a* is used by default unless *a* has an integer dtype of less precision than the default platform integer. In that case, if *a* is signed then the platform integer is used while if *a* is unsigned then an unsigned integer of the same precision as the platform integer is used.
- **keepdims** (*bool, optional*) –
  If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then *keepdims* will not be passed through to the *sum* method of sub-classes of *ndarray*, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

- **initial** (*scalar, optional*) – Starting value for the sum. See *~numpy.ufunc.reduce* for details.
- **where** (*array_like of bool, optional*) – Elements to include in the sum. See *~numpy.ufunc.reduce* for details.
- **promote_integers** (*bool, default=True*) – If True, then integer inputs will be promoted to the widest available integer dtype, following numpy's behavior. If False, the result will have the same dtype as the input. `promote_integers` is ignored if `dtype` is specified.
- **out** (`None`) –

**Returns:** **sum_along_axis** – An array with the same shape as *a*, with the specified axis removed. If *a* is a 0-d array, or if *axis* is None, a scalar is returned. If an output array is specified, a reference to *out* is returned.

**Return type:** ndarray

### `swapaxes(axis1, axis2)`

Interchange two axes of an array.

LAX-backend implementation of `numpy.swapaxes()`.

The JAX version of this function may in some cases return a copy rather than a view of the input.

*Original docstring below.*

**Parameters:**

- **a** (*array_like*) – Input array.
- **axis1** (*int*) – First axis.
- **axis2** (*int*) – Second axis.

**Returns:** **a_swapped** – For NumPy >= 1.10.0, if *a* is an ndarray, then a view of *a* is returned; otherwise a new array is created. For earlier NumPy versions a view of *a* is returned only if the order of the axes is changed, otherwise the input array is returned.

**Return type:** ndarray

### `take(indices, axis=None, out=None, mode=None, unique_indices=False, indices_are_sorted=False, fill_value=None)`

Take elements from an array along an axis.

LAX-backend implementation of **`numpy.take()`**.

By default, JAX assumes that all indices are in-bounds. Alternative out-of-bound index semantics can be specified via the `mode` parameter (see below).

*Original docstring below.*

When axis is not None, this function does the same thing as “fancy” indexing (indexing arrays using arrays); however, it can be easier to use if you need elements along a given axis. A call such as `np.take(arr, indices, axis=3)` is equivalent to `arr[:,:,:,indices,...]`.

Explained without fancy indexing, this is equivalent to the following use of *ndindex*, which sets each of `ii`, `jj`, and `kk` to a tuple of indices:

```
Ni, Nk = a.shape[:axis], a.shape[axis+1:]
Nj = indices.shape
for ii in ndindex(Ni):
    for jj in ndindex(Nj):
        for kk in ndindex(Nk):
            out[ii + jj + kk] = a[ii + (indices[jj],) + kk]
```

**Parameters:**

- **a** (*array_like (Ni..., M, Nk...)*) – The source array.
- **indices** (*array_like (Nj...)*) – The indices of the values to extract.
- **axis** (*int, optional*) – The axis over which to select values. By default, the flattened input array is used.
- **mode** (*string, default="fill"*) – Out-of-bounds indexing mode. The default mode=”fill” returns invalid values (e.g. NaN) for out-of bounds indices (see also `fill_value` below). For more discussion of mode options, see **`jax.numpy.ndarray.at`**.
- **fill_value** (*optional*) – The fill value to return for out-of-bounds slices when mode is ‘fill’. Ignored otherwise. Defaults to NaN for inexact types, the largest negative value for signed types, the largest positive value for unsigned types, and True for booleans.
- **unique_indices** (*bool, default=False*) – If True, the implementation will assume that the indices are unique, which can result in more efficient execution on some backends.
- **indices_are_sorted** (*bool, default=False*) – If True, the implementation will assume that the indices are sorted in ascending order, which can lead to more efficient execution on some backends.

**Returns:** **out** – The returned array has the same type as *a*.

**Return type:** ndarray (Ni…, Nj…, Nk…)

### `trace(offset=0, axis1=0, axis2=1, dtype=None, out=None)`

Return the sum along diagonals of the array.

LAX-backend implementation of `numpy.trace()`.

*Original docstring below.*

If *a* is 2-D, the sum along its diagonal with the given offset is returned, i.e., the sum of elements `a[i,i+offset]` for all i.

If *a* has more than two dimensions, then the axes specified by axis1 and axis2 are used to determine the 2-D sub-arrays whose traces are returned. The shape of the resulting array is the same as that of *a* with *axis1* and *axis2* removed.

**Parameters:**

- **a** (*array_like*) – Input array, from which the diagonals are taken.
- **offset** (*int, optional*) – Offset of the diagonal from the main diagonal. Can be both positive and negative. Defaults to 0.
- **axis1** (*int, optional*) – Axes to be used as the first and second axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults are the first two axes of *a*.
- **axis2** (*int, optional*) – Axes to be used as the first and second axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults are the first two axes of *a*.
- **dtype** (*dtype, optional*) – Determines the data-type of the returned array and of the accumulator where the elements are summed. If dtype has the value None and *a* is of integer type of precision less than the default integer precision, then the default integer precision is used. Otherwise, the precision is the same as that of *a*.
- **out** (`None`) –

**Returns:** **sum_along_diagonals** – If *a* is 2-D, the sum along the diagonal is returned. If *a* has larger dimensions, then an array of sums along diagonals is returned.

**Return type:** ndarray

### `unsafe_buffer_pointer()`

(self: xla::PyBuffer::pyobject) -> StatusOr[int]

### `var(axis=None, dtype=None, out=None, ddof=0, keepdims=False, *, where=None)`

Compute the variance along the specified axis.

LAX-backend implementation of `numpy.var()`.

*Original docstring below.*

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

**Parameters:**

- **a** (*array_like*) – Array containing numbers whose variance is desired. If *a* is not an array, a conversion is attempted.
- **axis** (*None or int or tuple of ints, optional*) – Axis or axes along which the variance is computed. The default is to compute the variance of the flattened array.
- **dtype** (*data-type, optional*) – Type to use in computing the variance. For arrays of integer type the default is *float64*; for arrays of float types it is the same as the array type.
- **ddof** (*int, optional*) – “Delta Degrees of Freedom”: the divisor used in the calculation is `N - ddof`, where `N` represents the number of elements. By default *ddof* is zero.
- **keepdims** (*bool, optional*) –
  If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.
  If the default value is passed, then *keepdims* will not be passed through to the *var* method of sub-classes of *ndarray*, however any non-default value will be. If the sub-class’ method does not implement *keepdims* any exceptions will be raised.
- **where** (*array_like of bool, optional*) – Elements to include in the variance. See *~numpy.ufunc.reduce* for details.
- **out** (`None`) –

**Returns:** **variance** – If `out=None`, returns a new array containing the variance; otherwise, a reference to the output array is returned.

**Return type:** ndarray, see dtype parameter above

### `xla_dynamic_shape()`

(self: xla::PyBuffer::pyobject) -> StatusOr[jaxlib.xla_extension.Shape]

### `xla_shape()`

(self: xla::PyBuffer::pyobject) -> jaxlib.xla_extension.Shape

---