

# NumPy Broadcasting & JAX

Lecture 06

Dr. Colin Rundel

# Basic file IO

# Reading and writing ndarrays

We will not spend much time on this as most data you will encounter is more likely to be a tabular format (e.g. data frame) and tools like Pandas are more appropriate.

For basic saving and loading of NumPy arrays there are the `save()` and `load()` functions which use a custom binary format.

```
1 x = np.arange(1e5)
2 np.save("data/x.npy", x)
```

```
1 new_x = np.load("data/x.npy")
2 np.all(x == new_x)
```

`np.True_`

Additional functions for saving (`savez()`, `savez_compressed()`, `savetxt()`) exist for saving multiple arrays or saving a text representation of an array.

# Reading delimited data

While not particularly recommended, if you need to read delimited (csv, tsv, etc.) data into a NumPy array you can use `genfromtxt()`,

```
1 with open("data/mtcars.csv") as file:
2     mtcars = np.genfromtxt(file, delimiter=";", skip_header=True)
3
4 mtcars
```

```
array([[ 6.   , 160.   , 110.   , 3.9   , 2.62  ,
        16.46 , 0.    , 1.    , 4.    , 4.    ],
       [ 6.   , 160.   , 110.   , 3.9   , 2.875 ,
        17.02 , 0.    , 1.    , 4.    , 4.    ],
       [ 4.   , 108.   , 93.    , 3.85  , 2.32  ,
        18.61 , 1.    , 1.    , 4.    , 1.    ],
       [ 6.   , 258.   , 110.   , 3.08  , 3.215 ,
        19.44 , 1.    , 0.    , 3.    , 1.    ],
       [ 8.   , 360.   , 175.   , 3.15  , 3.44  ,
        17.02 , 0.    , 0.    , 3.    , 2.    ],
       [ 6.   , 225.   , 105.   , 2.76  , 3.46  ,
        20.22 , 1.    , 0.    , 3.    , 1.    ],
       [ 8.   , 360.   , 245.   , 3.21  , 3.57  ,
        15.84 , 0.    , 0.    , 3.    , 4.    ],
       [ 4.   , 146.7  , 62.   , 3.69  , 3.19  ,
        20.   , 1.    , 0.    , 4.    , 1.    ]])
```

# Broadcasting

# Broadcasting

This is an approach for deciding how to generalize operations between arrays with differing shapes.

```
1 x = np.array([1, 2, 3])
```

```
1 x * 2
```

```
array([2, 4, 6])
```

```
1 x * np.array([2,2,2])
```

```
array([2, 4, 6])
```

```
1 x * np.array([2])
```

```
array([2, 4, 6])
```

# Efficiency

Using broadcasts can be more efficient as it does not copy the broadcast data,

```
1 x = np.arange(1e5)
2 y = np.array([2]).repeat(1e5)
```

```
1 %timeit x * 2
```

13.1  $\mu\text{s}$   $\pm$  297 ns per loop (mean  $\pm$  std. dev. of 7 runs, 100,000 loops each)

```
1 %timeit x * np.array([2])
```

18.3  $\mu\text{s}$   $\pm$  96.4 ns per loop (mean  $\pm$  std. dev. of 7 runs, 10,000 loops each)

```
1 %timeit x * y
```

61.4  $\mu\text{s}$   $\pm$  465 ns per loop (mean  $\pm$  std. dev. of 7 runs, 10,000 loops each)

```
1 %timeit x * np.array([2]).repeat(1e5)
```

99.2  $\mu\text{s}$   $\pm$  1.12  $\mu\text{s}$  per loop (mean  $\pm$  std. dev. of 7 runs, 10,000 loops each)

# Rules for Broadcasting

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimensions and works its way left. Two dimensions are compatible when

1. they are equal, or
2. one of them is 1

If these conditions are not met, a `ValueError: operands could not be broadcast together` exception is thrown, indicating that the arrays have incompatible shapes. The size of the resulting array is the size that is not 1 along each axis of the inputs.



# Example

Why does the code on the left work but not the code on the right?

```
1 x = np.arange(12).reshape((4,3)); x
```

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

```
1 x + np.array([1,2,3])
```

```
array([[ 1,  3,  5],
       [ 4,  6,  8],
       [ 7,  9, 11],
       [10, 12, 14]])
```

```
x      (2d array): 4 x 3
y      (1d array):    3
-----
x+y    (2d array): 4 x 3
```

```
1 x = np.arange(12).reshape((3,4)); x
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
1 x + np.array([1,2,3])
```

ValueError: operands could not be broadcast together with shapes (3,4) (3,)

```
x      (2d array): 3 x 4
y      (1d array):    3
-----
x+y    (2d array): Error
```

# A fix

```
1 x = np.arange(12).reshape((3,4)); x
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
1 x + np.array([1,2,3]).reshape(3,1)
```

```
array([[ 1,  2,  3,  4],
       [ 6,  7,  8,  9],
       [11, 12, 13, 14]])
```

```
x      (2d array): 3 x 4
y      (2d array): 3 x 1
-----
x+y    (2d array): 3 x 4
```

# Examples (2)

```
1 x = np.array([0,10,20,30]).reshape((4,1))
2 y = np.array([1,2,3])
```

```
1 x
```

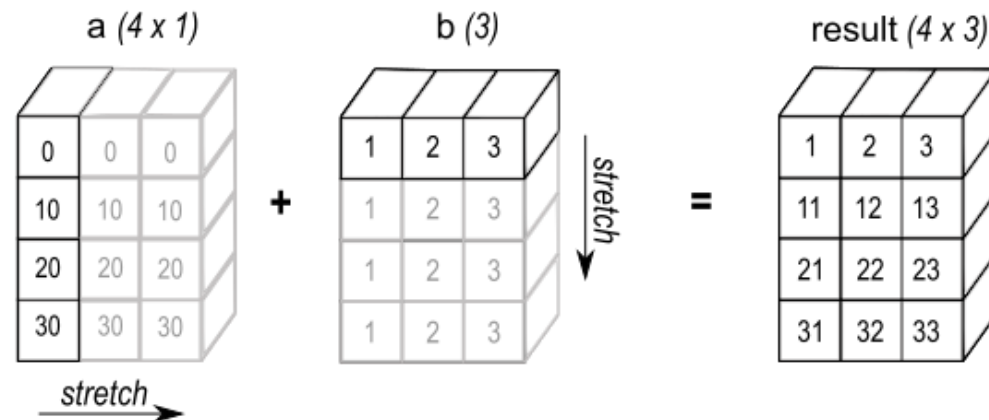
```
array([[ 0],
       [10],
       [20],
       [30]])
```

```
1 x+y
```

```
array([[ 1,  2,  3],
       [11, 12, 13],
       [21, 22, 23],
       [31, 32, 33]])
```

```
1 y
```

```
array([1, 2, 3])
```



# Exercise 1

For each of the following combinations determine what the resulting dimension will be using broadcasting

- $A [128 \times 128 \times 3] + B [3]$
- $A [8 \times 1 \times 6 \times 1] + B [7 \times 1 \times 5]$
- $A [2 \times 1] + B [8 \times 4 \times 3]$
- $A [3 \times 1] + B [15 \times 3 \times 5]$
- $A [3] + B [4]$

# Demo 1 - Standardization

Below we generate a data set with 3 columns of random normal values. Each column has a different mean and standard deviation which we can check with `mean()` and `std()`.

```
1 rng = np.random.default_rng(1234)
2 d = rng.normal(
3     loc=[-1,0,1],
4     scale=[1,2,3],
5     size=(1000,3)
6 )
```

```
1 d.shape
```

```
(1000, 3)
```

```
1 d.mean(axis=0)
```

```
array([-1.02944, -0.01396,  1.01242])
```

```
1 d.std(axis=0)
```

```
array([0.99675,  2.03223,  3.10625])
```

Lets use broadcasting to standardize all three columns to have mean 0 and standard deviation 1.

# Broadcasting and assignment

In addition to arithmetic operators, broadcasting can be used with assignment via array indexing,

```
1 x = np.arange(12).reshape((3,4))
2 y = -np.arange(4)
3 z = -np.arange(3)
```

```
1 x[:] = y
2 x
```

```
array([[ 0, -1, -2, -3],
       [ 0, -1, -2, -3],
       [ 0, -1, -2, -3]])
```

```
1 x[...] = y
2 x
```

```
array([[ 0, -1, -2, -3],
       [ 0, -1, -2, -3],
       [ 0, -1, -2, -3]])
```

```
1 x[:] = z
```

ValueError: could not broadcast input array from shape (3,) into shape (3,4)

```
1 x[:] = z.reshape((3,1))
2 x
```

```
array([[ 0,  0,  0,  0],
       [-1, -1, -1, -1],
       [-2, -2, -2, -2]])
```

**JAX**

# JAX

JAX is a library for array-oriented numerical computation (à la NumPy), with automatic differentiation and JIT compilation to enable high-performance machine learning research.

- JAX provides a unified NumPy-like interface to computations that run on CPU, GPU, or TPU, in local or distributed settings.
- JAX features built-in Just-In-Time (JIT) compilation via Open XLA, an open-source machine learning compiler ecosystem.
- JAX functions support efficient evaluation of gradients via its automatic differentiation transformations.
- JAX functions can be automatically vectorized to efficiently map them over arrays representing batches of inputs.

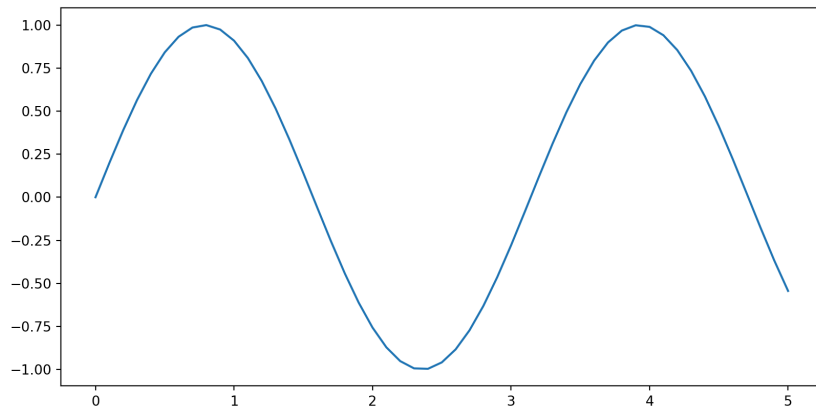
```
1 import jax
2 jax.__version__
```

'0.5.0'



# JAX & NumPy

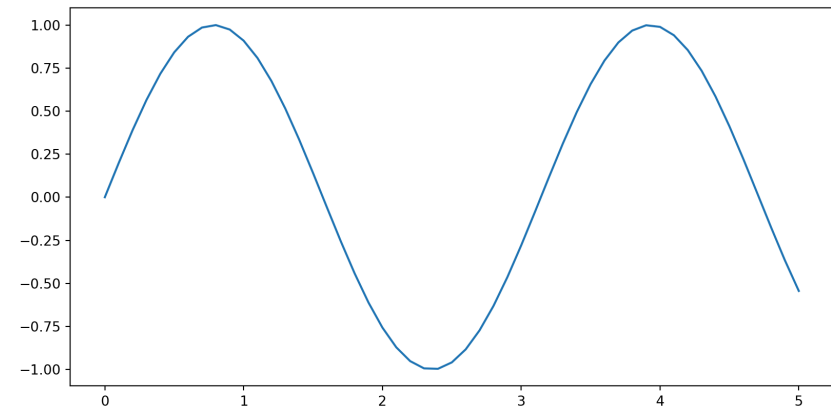
```
1 import numpy as np
2
3 x_np = np.linspace(0, 5, 51)
4 y_np = 2 * np.sin(x_np) * np.cos(x_np)
5 plt.plot(x_np, y_np)
```



```
1 type(x_np)
```

```
<class 'numpy.ndarray'>
```

```
1 import jax.numpy as jnp
2
3 x_jnp = jnp.linspace(0, 5, 51)
4 y_jnp = 2 * jnp.sin(x_jnp) * jnp.cos(x_jnp)
5 plt.plot(x_jnp, y_jnp)
```



```
1 type(x_jnp)
```

```
<class 'jaxlib.xla_extension.ArrayImpl'>
```

```
1 x_np
```

```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6,
       0.7, 0.8, 0.9,
        1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6,
       1.7, 1.8, 1.9,
        2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6,
       2.7, 2.8, 2.9,
        3. , 3.1, 3.2, 3.3, 3.4, 3.5, 3.6,
       3.7, 3.8, 3.9,
        4. , 4.1, 4.2, 4.3, 4.4, 4.5, 4.6,
       4.7, 4.8, 4.9,
        5. ])
```

```
1 x_np.dtype
```

```
dtype('float64')
```

```
1 x_jnp
```

```
Array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6,
       0.7, 0.8, 0.9,
        1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6,
       1.7, 1.8, 1.9,
        2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6,
       2.7, 2.8, 2.9,
        3. , 3.1, 3.2, 3.3, 3.4, 3.5, 3.6,
       3.7, 3.8, 3.9,
        4. , 4.1, 4.2, 4.3, 4.4, 4.5, 4.6,
       4.7, 4.8, 4.9,
        5. ], dtype=float32)
```

```
1 x_jnp.dtype
```

```
dtype('float32')
```

# Compatibility

```
1 y_mix = 2 * np.sin(x_jnp) * jnp.cos(x_np); y_mix
```

```
Array([ 0.          ,  0.19867,  0.38942,  0.56464,  0.71736,  
        0.84147,  0.93204,  0.98545,  0.99957,  0.97385,  
        0.9093 ,  0.8085 ,  0.67546,  0.5155 ,  0.33499,  
        0.14112, -0.05837, -0.25554, -0.44252, -0.61186,  
       -0.7568 , -0.87158, -0.9516 , -0.99369, -0.99616,  
       -0.95892, -0.88345, -0.77276, -0.63127, -0.4646 ,  
       -0.27942, -0.08309,  0.11655,  0.31154,  0.49411,  
        0.65699,  0.79367,  0.89871,  0.96792,  0.99854,  
        0.98936,  0.94073,  0.8546 ,  0.7344 ,  0.58492,  
        0.41212,  0.22289,  0.02478, -0.17433, -0.36648,  
       -0.54402], dtype=float32)
```

```
1 type(y_mix)
```

```
<class 'jaxlib.xla_extension.ArrayImpl'>
```

# JAX Arrays

As we've just seen a JAX array is very similar to a numpy array but there are some important differences.

- JAX arrays are immutable

```
1 x = jnp.array([3, 2, 1])
2 x[0] = 2
```

TypeError: JAX arrays are immutable and do not support in-place item assignment. Instead of `x[idx] = y`, use `x = x.at[idx].set(y)` or another `.at[]` method:  
[https://jax.readthedocs.io/en/latest/\\_autosummary/jax.numpy.ndarray.at.html](https://jax.readthedocs.io/en/latest/_autosummary/jax.numpy.ndarray.at.html)

- related to the above JAX does not support inplace operations - these functions now create and return a copy of the array

```
1 y = x.sort()
2 y
```

```
Array([1, 2, 3], dtype=int32)
```

```
1 x
```

```
Array([3, 2, 1], dtype=int32)
```

```
1 np.shares_memory(x,y)
```

```
False
```

- The default JAX array dtypes 32 bits not 64 bits (i.e. `float32` not `float64` and `int32` not `int64`)

```
1 jnp.array([1, 2, 3])
```

```
Array([1, 2, 3], dtype=int32)
```

```
1 jnp.array([1., 2., 3.])
```

```
Array([1., 2., 3.], dtype=float32)
```

```
1 jnp.array([1, 2, 3], dtype=jnp.float64)
```

UserWarning: Explicitly requested dtype `<class 'jax.numpy.float64'>` requested in array is not available, and will be truncated to dtype `float32`. To enable more dtypes, set the `jax_enable_x64` configuration option or the `JAX_ENABLE_X64` shell environment variable. See <https://github.com/jax-ml/jax#current-gotchas> for more.

```
Array([1., 2., 3.], dtype=float32)
```

64-bit dtypes can be enabled by setting `jax_enable_x64=True` in the JAX configuration.

```
1 jax.config.update("jax_enable_x64", True)
```

```
1 jnp.array([1, 2, 3])
```

```
Array([1, 2, 3], dtype=int64)
```

```
1 jnp.array([1., 2., 3.])
```

```
Array([1., 2., 3.], dtype=float64)
```

- JAX arrays are allocated to one *or more* devices

```
1 jax.devices()
```

```
[CpuDevice(id=0)]
```

```
1 x.devices()
```

```
{CpuDevice(id=0)}
```

```
1 x.sharding
```

```
SingleDeviceSharding(device=CpuDevice(id=0)  
memory_kind=unpinned_host)
```

- Using JAX interactively allows for the use of standard Python control flow (`if`, `while`, `for`, etc.) but this is not supported for some of JAX's more advanced operations (e.g. `jit` and `grad`)

There are replacements for most of these constructs in JAX, but they are beyond the scope of today.

# Random number generation

# JAX vs NumPy

Pseudo random number generation in JAX is a bit different than with NumPy - the latter depends on a global state that is updated each time a random function is called.

NumPy's PRNG guarantees something called sequential equivalence which amounts to sampling N numbers sequentially is the same as sampling N numbers at once (e.g. a vector of length N).

```
1 np.random.seed(0)
2 f"individually: {np.stack([np.random.uniform() for i in range(5)])}"
```

```
'individually: [0.54881 0.71519 0.60276 0.54488 0.42365]'
```

```
1 np.random.seed(0)
2 f"at once: {np.random.uniform(size=5)}"
```

```
'at once: [0.54881 0.71519 0.60276 0.54488 0.42365]'
```



# Parallelization & sequential equivalence

Sequential equivalence can be problematic in when using parallelization across multiple devices, consider the following code:

```
1 np.random.seed(0)
2
3 def bar():
4     return np.random.uniform()
5
6 def baz():
7     return np.random.uniform()
8
9 def foo():
10    return bar() + 2 * baz()
```

How do we guarantee that we get consistent results if we don't know the order that `bar()` and `baz()` will run?

# PRNG keys

JAX makes use of *random keys* which are just a fancier version of random seeds - all of JAX's random functions require a key as their first argument.

```
1 key = jax.random.PRNGKey(1234); key
```

```
Array([ 0, 1234], dtype=uint32)
```

```
1 jax.random.normal(key)
```

```
Array(-0.5402, dtype=float64)
```

```
1 jax.random.normal(key)
```

```
Array(-0.5402, dtype=float64)
```

```
1 jax.random.normal(key, shape=(3,))
```

```
Array([-0.5402 ,  0.43958, -0.01978], dtype=float64)
```

Note that JAX does not provide a sequential equivalence guarantee - this is so that it can support vectorization for the generation of PRN.

# Splitting keys

Since a key is essentially a seed we do not want to reuse them (unless we want an identical output). Therefore to generate multiple different PRN we can split a key to deterministically generate two (or more) new keys.

```
1 key11, key12 = jax.random.split(key)
2 f"{key=}"
```

```
'key=Array([  0, 1234], dtype=uint32)'
```

```
1 f"{key11=}"
```

```
'key11=Array([1264997412, 2518116175],
dtype=uint32)'
```

```
1 f"{key12=}"
```

```
'key12=Array([2877103387, 1697627890],
dtype=uint32)'
```

```
1 key3 = jax.random.split(key, num=3)
2 key3
```

```
Array([[1264997412, 2518116175],
       [2877103387, 1697627890],
       [2113592192,  603280156]], dtype=uint32)
```

```
1 key21, key22 = jax.random.split(key)
2 f"{key=}"
```

```
'key=Array([  0, 1234], dtype=uint32)'
```

```
1 f"{key21=}"
```

```
'key21=Array([1264997412, 2518116175],
dtype=uint32)'
```

```
1 f"{key22=}"
```

```
'key22=Array([2877103387, 1697627890],
dtype=uint32)'
```

```
1 jax.random.normal(key, shape=(3,))
```

```
Array([-0.5402 ,  0.43958, -0.01978], dtype=float64)
```

```
1 jax.random.normal(key11, shape=(3,))
```

```
Array([ 1.24104,  0.12019, -2.2399 ],  
dtype=float64)
```

```
1 jax.random.normal(key21, shape=(3,))
```

```
Array([ 1.24104,  0.12019, -2.2399 ],  
dtype=float64)
```

```
1 jax.random.normal(key12, shape=(3,))
```

```
Array([ 0.07627, -1.3035 ,  0.86524],  
dtype=float64)
```

```
1 jax.random.normal(key22, shape=(3,))
```

```
Array([ 0.07627, -1.3035 ,  0.86524],  
dtype=float64)
```

```
1 jax.random.normal(key3[0], shape=(3,))
```

```
Array([ 1.24104,  0.12019, -2.2399 ], dtype=float64)
```

```
1 jax.random.normal(key3[1], shape=(3,))
```

```
Array([ 0.07627, -1.3035 ,  0.86524], dtype=float64)
```

```
1 jax.random.normal(key3[2], shape=(3,))
```

```
Array([-0.02894,  1.05075, -2.22082], dtype=float64)
```

# JAX & jit

# JAX performance

```
1 key = jax.random.PRNGKey(1234)
2 x_jnp = jax.random.normal(key, (1000,1000))
3 x_np = np.array(x_jnp)
```

```
1 %timeit y = x_np @ x_np
```

1.09 ms ± 92.4 μs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)

```
1 %timeit y = x_jnp @ x_jnp
```

3.42 ms ± 122 μs per loop (mean ± std. dev. of 7 runs, 100 loops each)

```
1 %timeit y = 3*x_np + x_np
```

514 μs ± 41 μs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)

```
1 %timeit y = 3*x_jnp + x_jnp
```

413 μs ± 24.2 μs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)

# jit

```
1 def SELU_np(x,  $\alpha=1.67$ ,  $\lambda=1.05$ ):  
2     "Scaled Exponential Linear Unit"  
3     return  $\lambda * \text{np.where}(x > 0, x, \alpha * \text{np.exp}(x))$ 
```

```
1 x = np.arange(1e6)  
2 %timeit y = SELU_np(x)
```

4.08 ms  $\pm$  80  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 100 loops each)

```
1 SELU_np_jit = jax.jit(SELU_np)
```

```
1 %timeit y = SELU_np_jit(x)
```

TracerArrayConversionError: The numpy.ndarray conversion method `__array__()` was called on traced array with shape float32[1000000]

```
1 def SELU_jnp(x,  $\alpha=1.67$ ,  $\lambda=1.05$ ):  
2     "Scaled Exponential Linear Unit"  
3     return  $\lambda * \text{jnp.where}(x > 0, x, \alpha * \text{jnp.exp}(x))$ 
```

```
1 x = jnp.arange(1e6)  
2 %timeit y = SELU_jnp(x)
```

1.58 ms  $\pm$  68.8  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 1,000 loops each)

```
1 SELU_jnp_jit = jax.jit(SELU_jnp)
```

```
1 %timeit y = SELU_jnp_jit(x)
```

418  $\mu$ s  $\pm$  13  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 1,000 loops each)

# jit limitations

When it works the jit tool is fantastic, but it does have a number of limitations,

- Must use `pure functions` (no side effects)
- Must primarily use JAX functions
  - e.g. use `jnp.minimum()` not `np.minimum()` or `min()`
- Must generally avoid conditionals / control flow
- Issues around concrete values when tracing (static values)
- Check performance - there are not always gains + there is the initial cost of compilation



# Automatic differentiation

# Basics

The `grad()` function takes a numerical function, returning a scalar, and returns a function for calculating the gradient of that function.

```
1 def f(x):  
2     return x**2
```

```
1 f(3.)
```

9.0

```
1 jax.grad(f)(3.)
```

Array(6., dtype=float64,  
weak\_type=True)

```
1 jax.grad(  
2     jax.grad(f)  
3 )(3.)
```

Array(2., dtype=float64,  
weak\_type=True)

```
1 def g(x):  
2     return jnp.exp(-x)
```

```
1 g(1.)
```

Array(0.36788, dtype=float64,  
weak\_type=True)

```
1 jax.grad(g)(1.)
```

Array(-0.36788, dtype=float64,  
weak\_type=True)

```
1 jax.grad(  
2     jax.grad(g)  
3 )(1.)
```

Array(0.36788, dtype=float64,  
weak\_type=True)

```
1 def h(x):  
2     return jnp.maximum(0,x)
```

```
1 h(-2.)
```

Array(0., dtype=float64,  
weak\_type=True)

```
1 h(2.)
```

Array(2., dtype=float64,  
weak\_type=True)

```
1 jax.grad(h)(-2.)
```

Array(0., dtype=float64,  
weak\_type=True)

```
1 jax.grad(h)(2.)
```

Array(1., dtype=float64,  
weak\_type=True)

## Aside - `vmap()`

I would like to plot `h()` and `jax.grad(h)()` - lets see what happens,

```
1 x = jnp.linspace(-3,3,101)
2 y = h(x)
3 y_grad = jax.grad(h)(x)
```

TypeError: Gradient only defined for scalar-output functions. Output had shape: (101,).

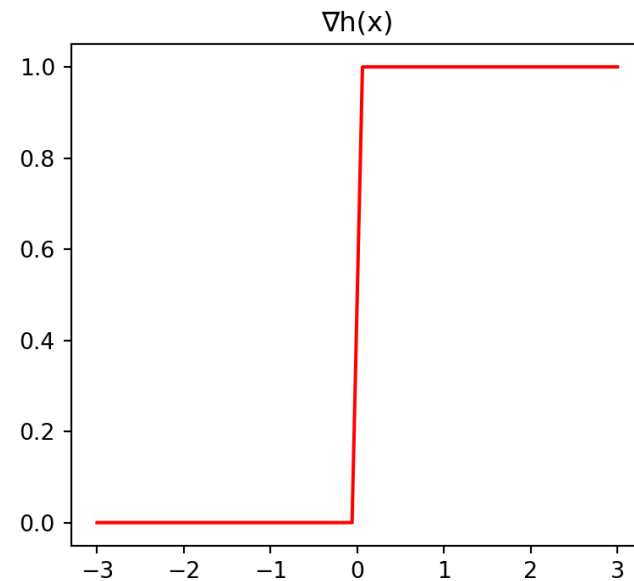
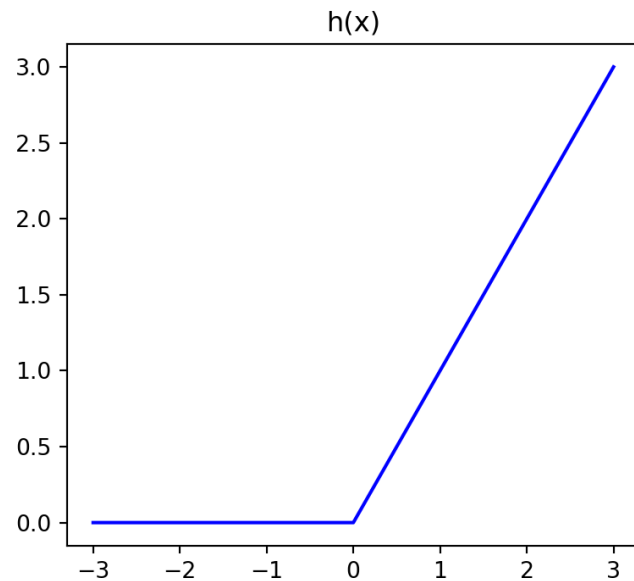
-----  
For simplicity, JAX has removed its internal frames from the traceback of the following exception. Set `JAX_TRACEBACK_FILTERING=off` to include these.

As just mentiond, we can only calculate the gradient for scalar valued functions. However, we can transform our scalar function into a vectorized function using `vmap()`.

```
1 h_grad = jax.vmap(
2     jax.grad(h)
3 )
4 y_grad = h_grad(x)
```

## 1 y\_grad

```
Array([0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ,  
       0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ,  
       0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ,  
       0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ,  
       0.5, 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. ,  
       1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. ,  
       1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. ,  
       1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. ,  
       1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. ,  
       1. ], dtype=float64)
```



# Another quick example

```
1 x = jnp.linspace(-6,6,101)
2 f = lambda x: 0.5 * (jnp.tanh(x / 2) + 1)
3 y = f(x)
4 y_grad = jax.vmap(jax.grad(f))(x)
```

