From NumPy To NumCpp – A Quick Start Guide

For a complete listing of all of the NumCpp functionality, see the full html documentation located [here](https://dpilger26.github.io/NumCpp).

**namespace NC = NumCpp;**

# Containers

The main data structure in **NumCpp** is the **NdArray**. It is inherently a 2D array class, with 1D arrays being implemented as 1xN arrays. There is also a **DataCube** class that is provided as a convenience container for storing an array of 2D **NdArrays**, but it has limited usefulness past a simple container.

|  |  |
| --- | --- |
| NumPy | NumCpp |
| a = np.array([[3, 4], [5, 6]]) | **NC::NdArray<int> a{{3, 4}, {5, 6}}** |
| a.reshape([3, 4]) | **a.reshape(3, 4)** |
| a.astype(np.double) | **a.astype<double>()** |

# Initializers

Many initializer functions are provided that return **NdArrays** for common needs.

|  |  |
| --- | --- |
| NumPy | NumCpp |
| np.linspace(1, 10, 2) | **NC::Methods<int>::linspace(1, 10, 2)** |
| np.arange(3, 7) | **NC::Methods<int>::arrange(3, 7)** |
| np.eye(4) | **NC::Methods<int>::eye(4)** |
| np.zeros([3, 4]) | **NC::Methods<int>::zeros(3, 4)** |
| np.ones([3, 4]) | **NC::Methods<int>::ones(3, 4)** |
| np.nans([3, 4]) | **NC::Methods<int>::nans(3, 4)** |
| np.empty([3, 4]) | **NC::Methods<int>::empty(3, 4)** |

# Slicing/Broadcasting

NumCpp offers NumPy style slicing and broadcasting.

|  |  |
| --- | --- |
| NumPy | NumCpp |
| a[2, 3] | **a(2, 3)** |
| a[2:5, 5:8] | **a(NC::Slice(2, 5), NC::Slice(5, 8))** |
| a[:, 7] | **a(NC::Slice::all(), 7)** |
| a[a > 5] | **a[a > 5]** |
| a[a > 5] = 0 | **a.put(a > 5, 0)** |

# Random

The random module provides simple ways to create random arrays.

|  |  |
| --- | --- |
| NumPy | NumCpp |
| np.random.seed(666) | **NC::Random<>::seed(666)** |
| np.random.randn(3, 4) | **NC::Random<double>::randn(nc::Shape(3,4))** |
| np.random.randint(0, 10, [3, 4]) | **NC::Random<int>::randInt(nc::Shape(3,4),0,10)** |
| np.random.rand(3, 4) | **NC::Random<double>::randn(nc::Shape(3,4))** |
| np.random.choice(a, 3) | **NC::Random<dtype>::choice(a, 3)** |

# Concatenation

Many ways to concatenate **NdArray** are available.

|  |  |
| --- | --- |
| NumPy | NumCpp |
| np.stack([a, b, c], axis=0) | **NC::Methods<dtype>::stack({a,b,c},NC::Axis::ROW)** |
| np.vstack([a, b, c]) | **NC::Methods<dtype>::vstack({a, b, c})** |
| np.hstack([a, b, c]) | **NC::Methods<dtype>::hstack({a, b, c})** |
| np.append(a, b, axis=1) | **NC::Methods<dtype>::append(a,b,NC::Axis::COL)** |

# Diagonal, triangular, and flip

The following return new **NdArrays**.

|  |  |
| --- | --- |
| NumPy | NumCpp |
| np.diagonal(a) | **NC::Methods<dtype>::diagonal(a)** |
| np.triu(a) | **NC::Methods<dtype>::triu(a)** |
| np.tril(a) | **NC::Methods<dtype>::tril(a)** |
| np.flip(a, axis=0) | **NC::Methods<dtype>::flip(a, NC::Axis::ROW)** |
| np.flipud(a) | **NC::Methods<dtype>::flipud(a)** |
| np.fliplr(a) | **NC::Methods<dtype>::fliplr(a)** |

# Iteration

NumCpp follows the idioms of the C++ STL providing iterator pairs to iterate on arrays in different fashions.

|  |  |
| --- | --- |
| NumPy | NumCpp |
| for value in a | **for(auto it = a.begin(); a < a.end(); ++it)** |
|  | **for(auto& value : a)** |

# Logical

Logical functions in NumCpp behave the same as NumPy.

|  |  |
| --- | --- |
| NumPy | NumCpp |
| np.where(a > 5) | **NC::Methods<dtype>::where(a > 5)** |
| np.where(a > 5, a, b) | **NC::Methods<dtype>::where(a > 5, a, b)** |
| np.any(a) | **NC::Methods<dtype>::any(a)** |
| np.all(a) | **NC::Methods<dtype>::all(a)** |
| np.logical\_and(a, b) | **NC::Methods<dtype>::logical\_and(a, b)** |
| np.logical\_or(a, b) | **NC::Methods<dtype>::logical\_or(a, b)** |
| np.isclose(a, b) | **NC::Methods<dtype>::isclose(a, b)** |
| np.allclose(a, b) | **NC::Methods<dtype>::allclose(a, b)** |

# Comparisons

|  |  |
| --- | --- |
| NumPy | NumCpp |
| np.equal(a, b) | **NC::Methods<dtype>::equal(a, b)** |
| np.not\_equal(a, b) | **NC::Methods<dtype>::not\_equal(a, b)** |
| np.nonzero(a) | **NC::Methods<dtype>::nonzero(a)** |

# Minimum, Maximum, Sorting

|  |  |
| --- | --- |
| NumPy | NumCpp |
| np.min(a) | **NC::Methods<dtype>::min(a)** |
| np.max(a) | **NC::Methods<dtype>::max(a)** |
| np.argmin(a) | **NC::Methods<dtype>::argmin(a)** |
| np.argmax(a) | **NC::Methods<dtype>::argmax(a)** |
| np.sort(a, axis=0) | **NC::Methods<dtype>::sort(a, NC::Axis::ROW)** |
| np.argsort(a, axis=1) | **NC::Methods<dtype>::argsort(a, NC::Axis::COL)** |
| np.unique(a) | **NC::Methods<dtype>::unique(a)** |
| np.setdiff1d(a, b) | **NC::Methods<dtype>::setdiff1d(a, b)** |
| np.diff(a) | **NC::Methods<dtype>::diff(a)** |

# Reducers

Reducers accumulate values of **NdArrays** along specified axes. When no axis is specified, values are accumulated along all axes.

|  |  |
| --- | --- |
| NumPy | NumCpp |
| np.sum(a) | **NC::Methods<dtype>::sum(a)** |
| np.sum(a, axis=0) | **NC::Methods<dtype>::sum(a, NC::Axis::ROW)** |
| np.prod(a) | **NC::Methods<dtype>::prod(a)** |
| np.prod(a, axis=0) | **NC::Methods<dtype>::prod(a, NC::Axis::ROW)** |
| np.mean(a) | **NC::Methods<dtype>::mean(a)** |
| np.mean(a, axis=0) | **NC::Methods<dtype>::mean(a, NC::Axis::ROW)** |
| np.count\_nonzero(a) | **NC::Methods<dtype>::count\_nonzero(a)** |
| np.count\_nonzero(a, axis=0) | **NC::Methods<dtype>::count\_nonzero(a,NC::Axis::ROW)** |

# I/O

Print and file output methods. All NumCpp classes support a print() method and << stream operators.

|  |  |
| --- | --- |
| NumPy | NumCpp |
| print(a) | **a.print()** |
|  | **std::cout << a** |
| a.tofile(filename, sep=’\n’) | **a.tofile(filename, sep=’\n’)** |
| np.fromfile(filename, sep=’\n’) | **NC::Methods<dtype>::fromfile(filename, sep=’\n’)** |
| np.dump(a, filename) | **NC::Methods<dtype>::dump(a, filename)** |
| np.load(filename) | **NC::Methods<dtype>::load(filename)** |

# Mathematical Functions

NumCpp universal functions are provided for a large set number of mathematical functions.

## Basic Functions

|  |  |
| --- | --- |
| NumPy | NumCpp |
| np.absolute(a) | **NC::Methods<dtype>::absolute(a)** |
| np.sign(a) | **NC::Methods<dtype>::sign(a)** |
| np.remainder(a, b) | **NC::Methods<dtype>::remainder(a, b)** |
| np.clip(a, min, max) | **NC::Methods<dtype>::clip(a, min, max)** |
| np.interp(x, xp, fp) | **NC::Methods<dtype>::interp(x, xp, fp)** |

## Exponential Functions

|  |  |
| --- | --- |
| NumPy | NumCpp |
| np.exp(a) | **NC::Methods<dtype>::exp(a)** |
| np.expm1(a) | **NC::Methods<dtype>::expm1(a)** |
| np.log(a) | **NC::Methods<dtype>::log(a)** |
| np.log1p(a) | **NC::Methods<dtype>::log1p(a)** |

## Power Functions

|  |  |
| --- | --- |
| NumPy | NumCpp |
| np.power(a, p) | **NC::Methods<dtype>::power(a)** |
| np.sqrt(a) | **NC::Methods<dtype>::sqrt(a)** |
| np.square(a) | **NC::Methods<dtype>::square(a)** |
| np.cbrt(a) | **NC::Methods<dtype>::cbrt(a)** |

## Trigonometric Functions

|  |  |
| --- | --- |
| NumPy | NumCpp |
| np.sin(a) | **NC::Methods<dtype>::sin(a)** |
| np.cos(a) | **NC::Methods<dtype>::cos(a)** |
| np.tan(a) | **NC::Methods<dtype>::tan(a)** |

## Hyperbolic Functions

|  |  |
| --- | --- |
| NumPy | NumCpp |
| np.sinh(a) | **NC::Methods<dtype>::sinh(a)** |
| np.cosh(a) | **NC::Methods<dtype>::cosh(a)** |
| np.tanh(a) | **NC::Methods<dtype>::tanh(a)** |

## Classification Functions

|  |  |
| --- | --- |
| NumPy | NumCpp |
| np.isnan(a) | **NC::Methods<dtype>::isnan(a)** |
| np.isinf(a) | **NC::Methods<dtype>::isinf(a)** |
| np.isfinite(a) | **NC::Methods<dtype>::isfinite(a)** |

# Linear Algebra

|  |  |
| --- | --- |
| NumPy | NumCpp |
| np.linalg.norm(a) | **NC::Methods<dtype>::norm(a)** |
| np.dot(a, b) | **NC::Methods<dtype>::dot(a, b)** |
| np.linalg.det(a) | **NC::Linalg<dtype>::det(a)** |
| np.linalg.inv(a) | **NC::Linalg<dtype>::inv(a)** |
| np.linalg.lstsq(a, b) | **NC::Linalg<dtype>::lstsq(a, b)** |
| np.linalg.matrix\_power(a, 3) | **NC::Linalg<dtype>::matrix\_power(a, 3)** |
| Np.linalg..multi\_dot(a, b, c) | **NC::Linalg<dtype>::multi\_dot({a, b, c})** |
| np.linalg.svd(a) | **NC::Linalg<dtype>::svd(a)** |