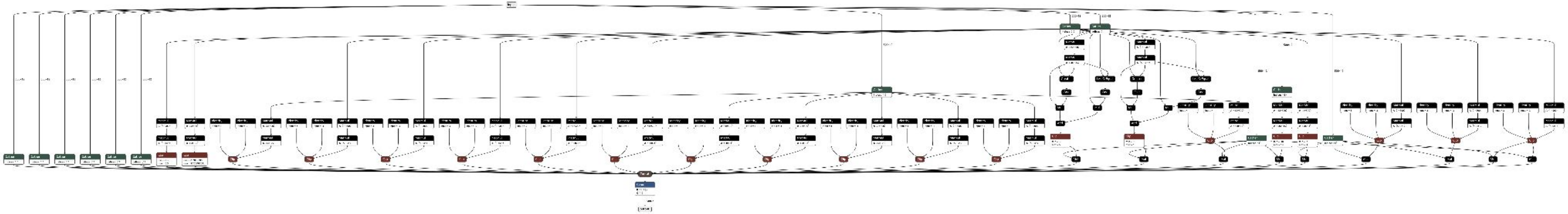
Numskull

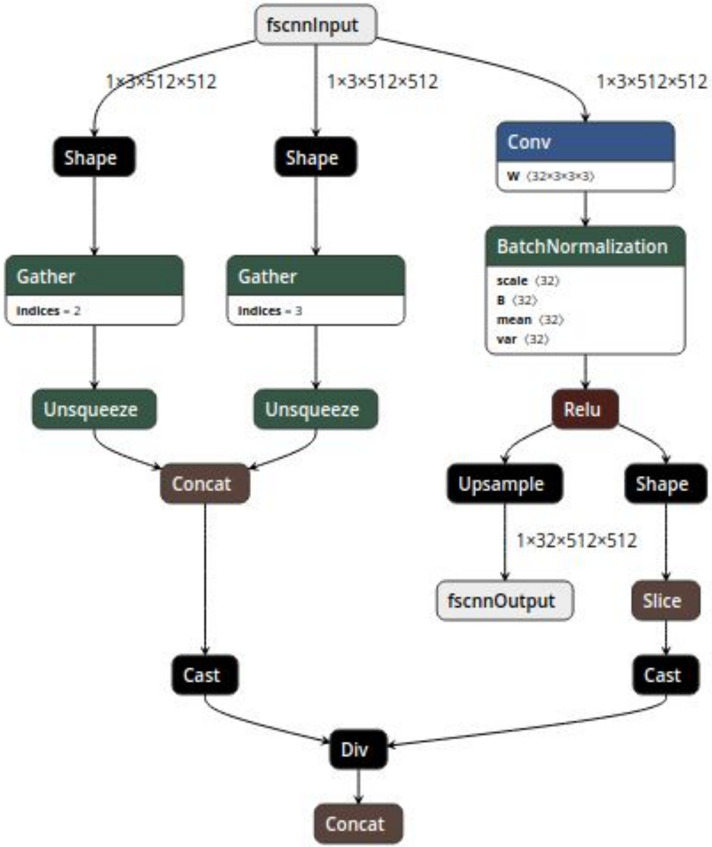
…

and why it’s not *so* daft.

Rowan Mather

ONNX




fscnnInput
1×3×512×512
1×3×512×512
1×3×512×512
Shape
Shape
Conv
W ⟨32×3×3×3⟩
Gather
indices = 2
Gather
indices = 3
BatchNormalization
scale ⟨32⟩
B ⟨32⟩
mean ⟨32⟩
var ⟨32⟩
Unsqueeze
Unsqueeze
Relu
Concat
Upsample
Shape
1×32×512×512
fscnnOutput
Slice
Cast
Cast
Div
Concat

Haskell likes types.

Existing Libraries

dynamic	Dynamic types
array	
repa	Multi-dimensional arrays
hmatrix	2D arrays with BLAS
accelerate	High-performance arrays
dense	The family favourite!

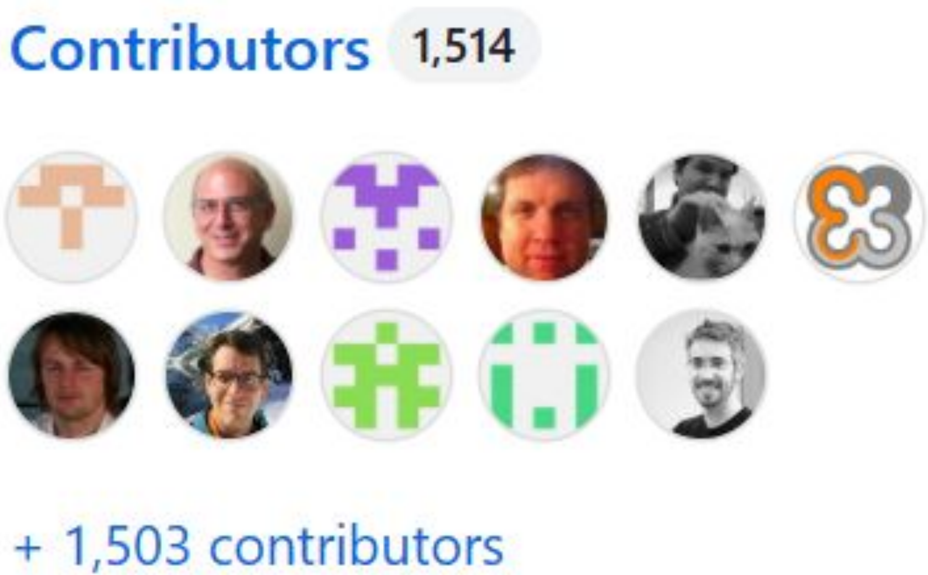
Everything is secretly a vector

NumPy

The fundamental package for scientific computing with Python

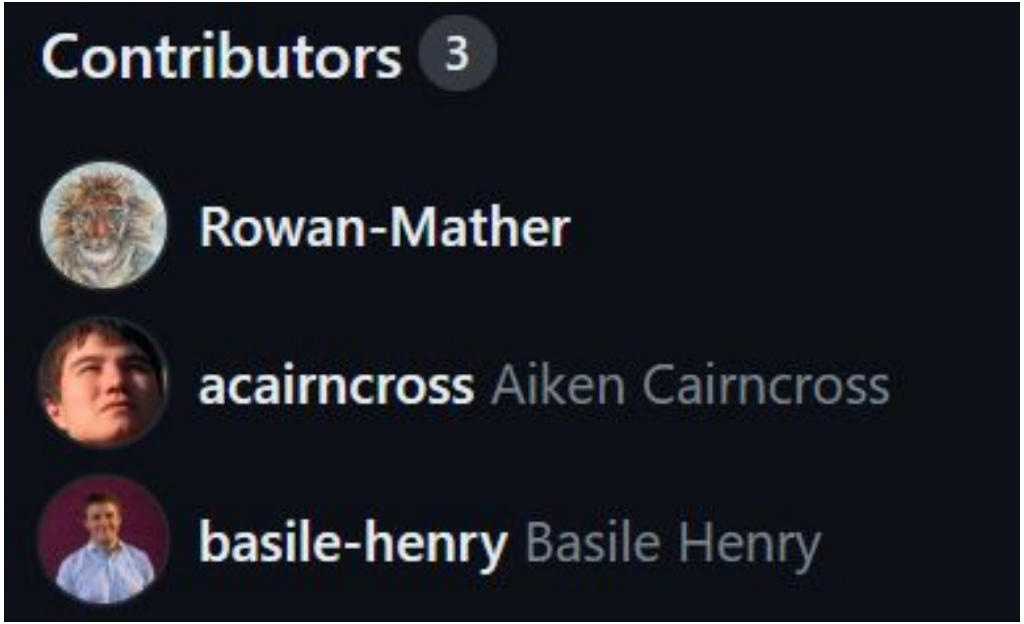
NumPy

The fundamental package for scientific computing with Python



Contributors 1,514

+ 1,503 contributors



Contributors 3

Rowan-Mather

acairncross Aiken Cairncross

basile-henry Basile Henry

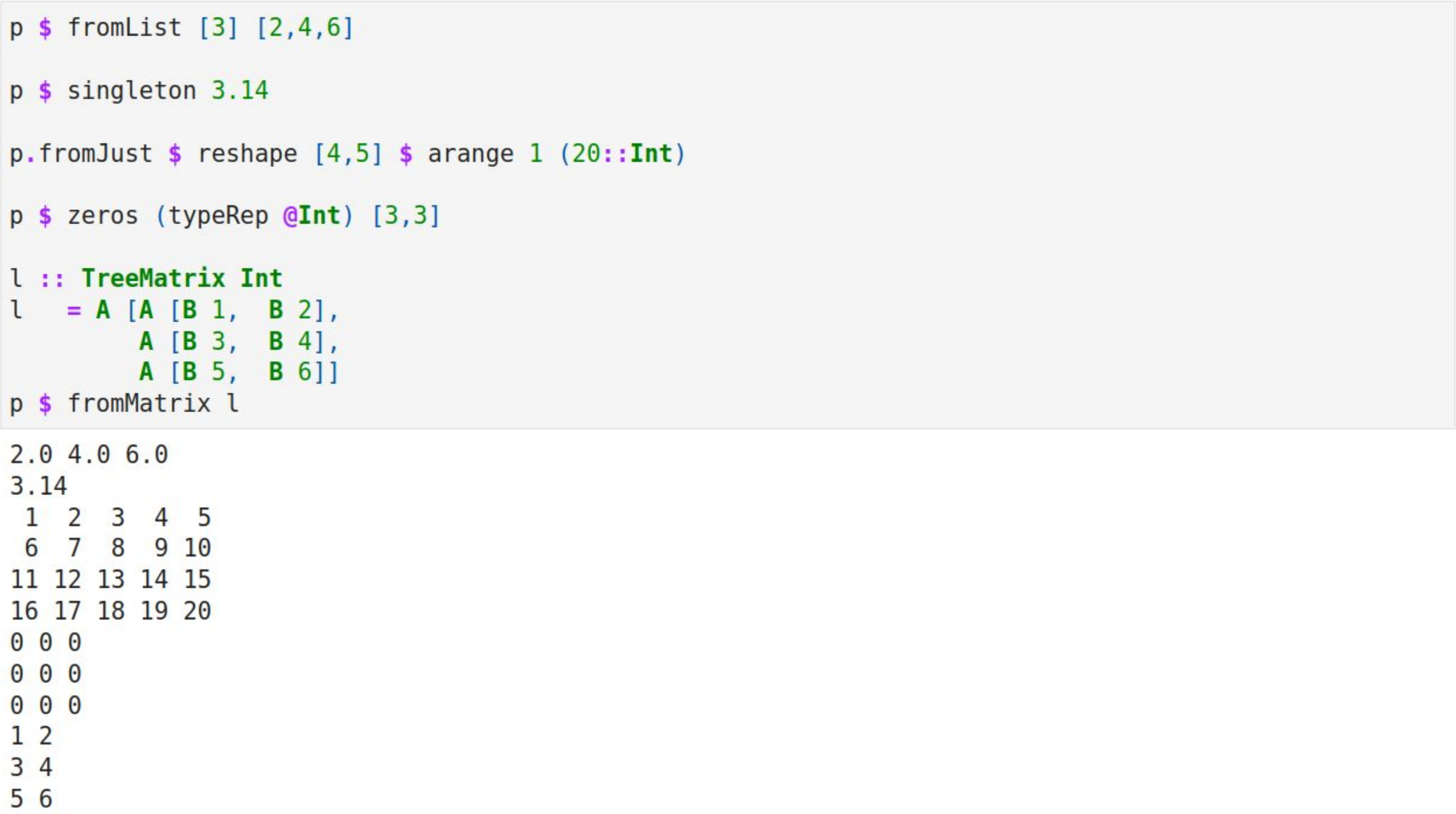
```
data NdArray where
        NdArray :: DType a => [Integer] -> Vector a -> NdArray
```

** In Numskull 1.0

```
class (Storable a, Typeable a, Show a, Eq a, Ord a) => DType a where
        addId :: a
        multId :: a
        add :: a -> a -> a
        subtract :: a -> a -> a
        multiply :: a -> a -> a
        ...
```

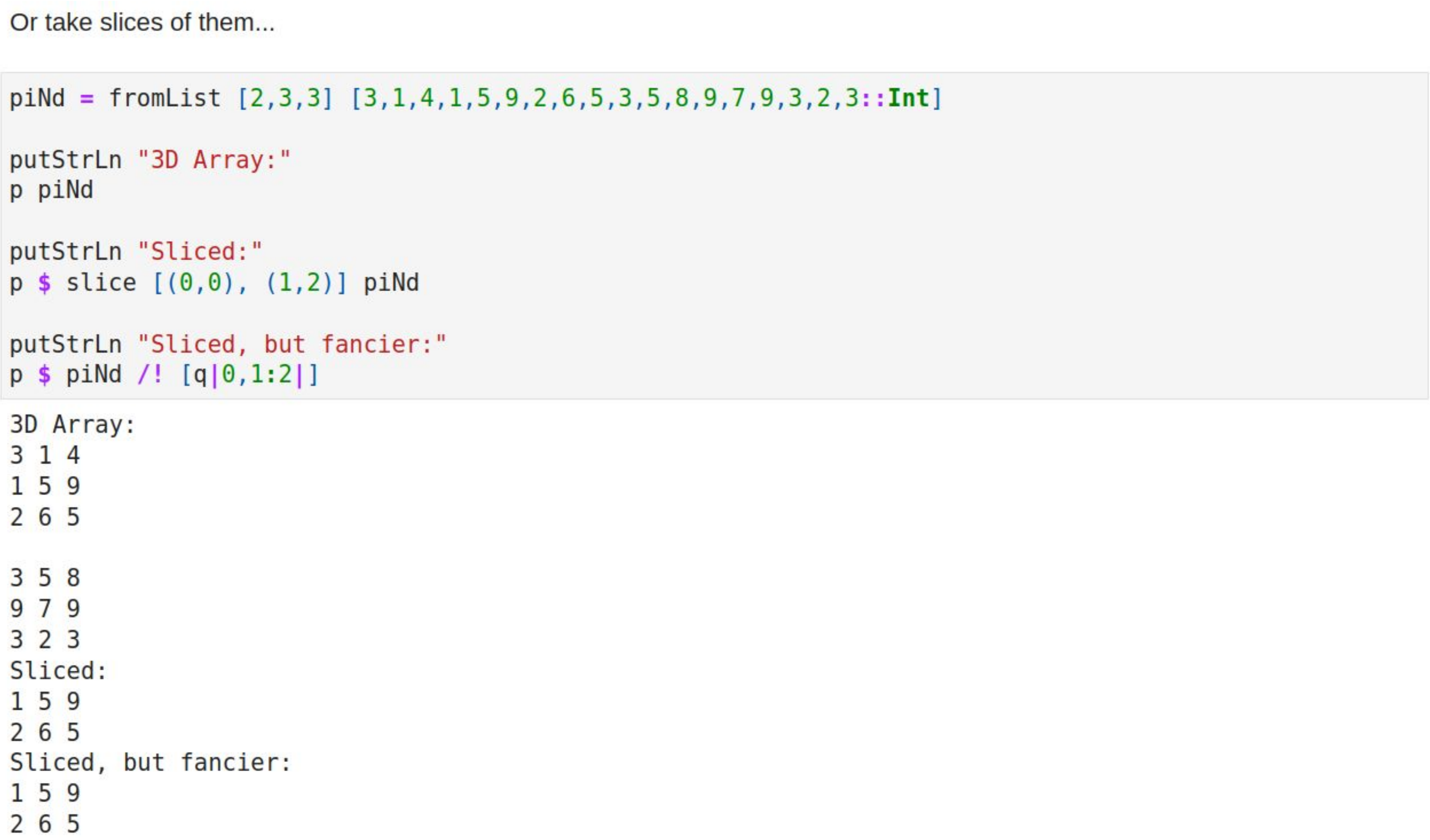
Demo!

https://github.com/MyrtleSoftware/rowan-ndarray
/tree/main/demo/notebook



```haskell
p $ fromList [3] [2,4,6]

p $ singleton 3.14

p.fromJust $ reshape [4,5] $ arange 1 (20::Int)

p $ zeros (typeRep @Int) [3,3]

l :: TreeMatrix Int
l   = A [A [B 1,  B 2],
         A [B 3,  B 4],
         A [B 5,  B 6]]
p $ fromMatrix l
```

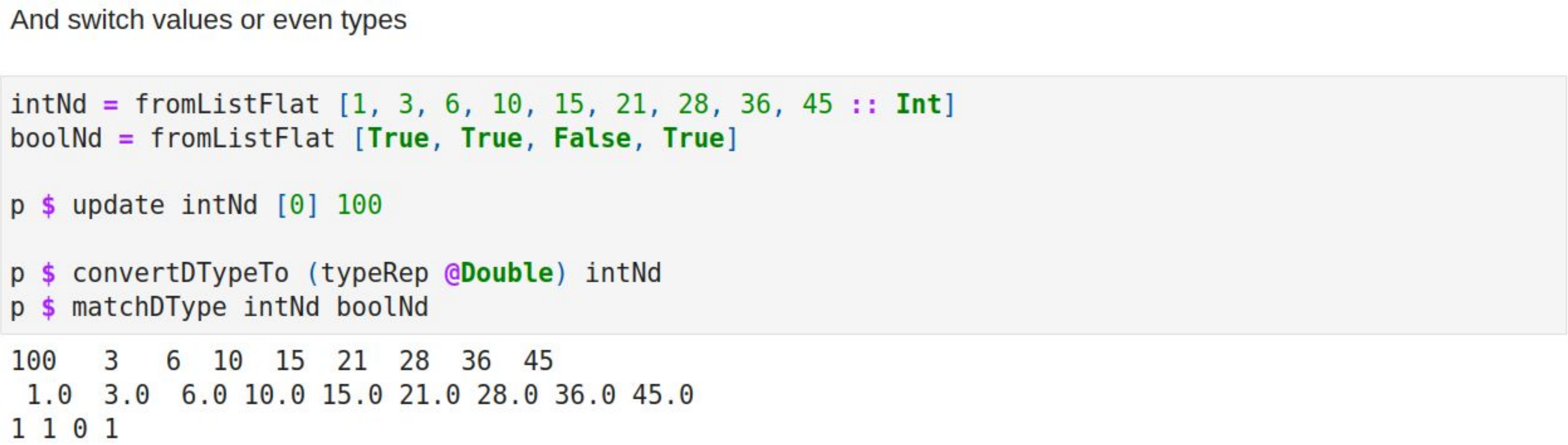
```
2.0 4.0 6.0
3.14
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
16 17 18 19 20
0 0 0
0 0 0
0 0 0
1 2
3 4
5 6
```



Or take slices of them...

```
piNd = fromList [2,3,3] [3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3,2,3::Int]

putStrLn "3D Array:"
p piNd

putStrLn "Sliced:"
p $ slice [(0,0), (1,2)] piNd

putStrLn "Sliced, but fancier:"
p $ piNd /! [q|0,1:2|]
```

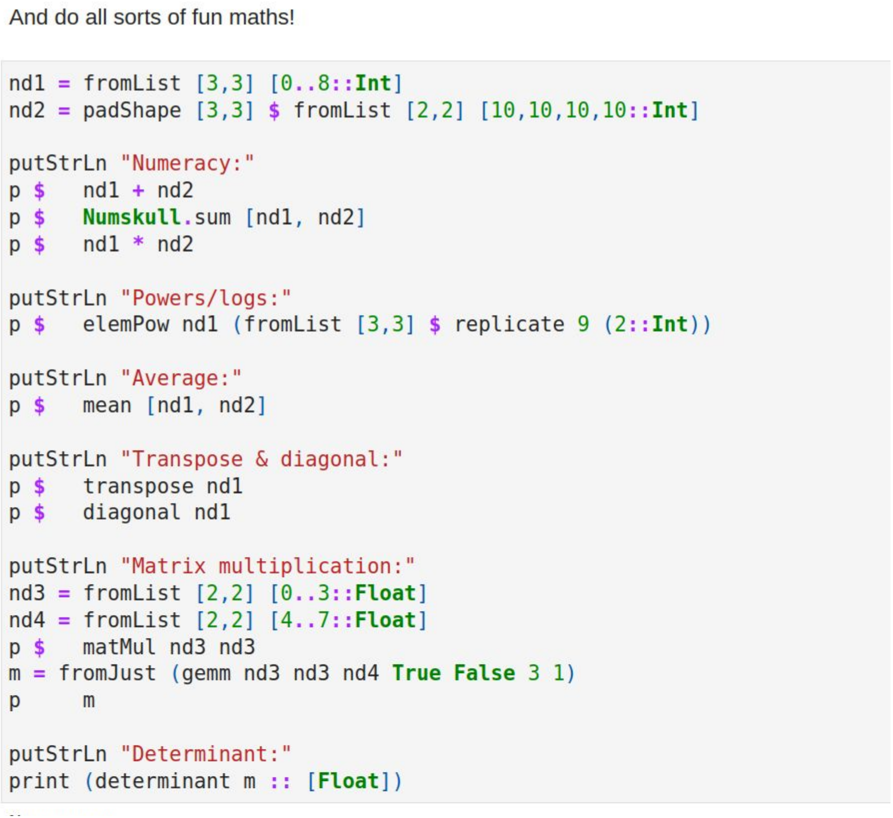
```
3D Array:
3 1 4
1 5 9
2 6 5

3 5 8
9 7 9
3 2 3
Sliced:
1 5 9
2 6 5
Sliced, but fancier:
1 5 9
2 6 5
```



And switch values or even types

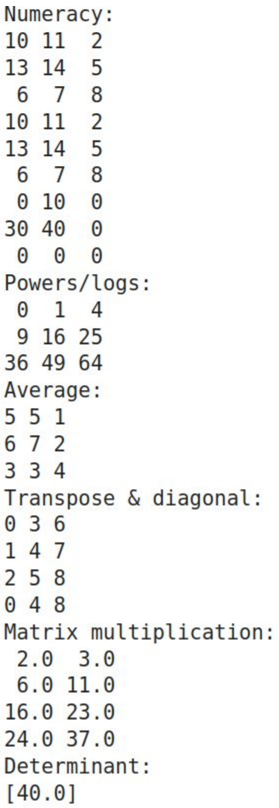
```
intNd = fromListFlat [1, 3, 6, 10, 15, 21, 28, 36, 45 :: Int]
boolNd = fromListFlat [True, True, False, True]

p $ update intNd [0] 100

p $ convertDTypeTo (typeRep @Double) intNd
p $ matchDType intNd boolNd
```

```
100   3   6  10  15  21  28  36  45
 1.0  3.0  6.0 10.0 15.0 21.0 28.0 36.0 45.0
1 1 0 1
```

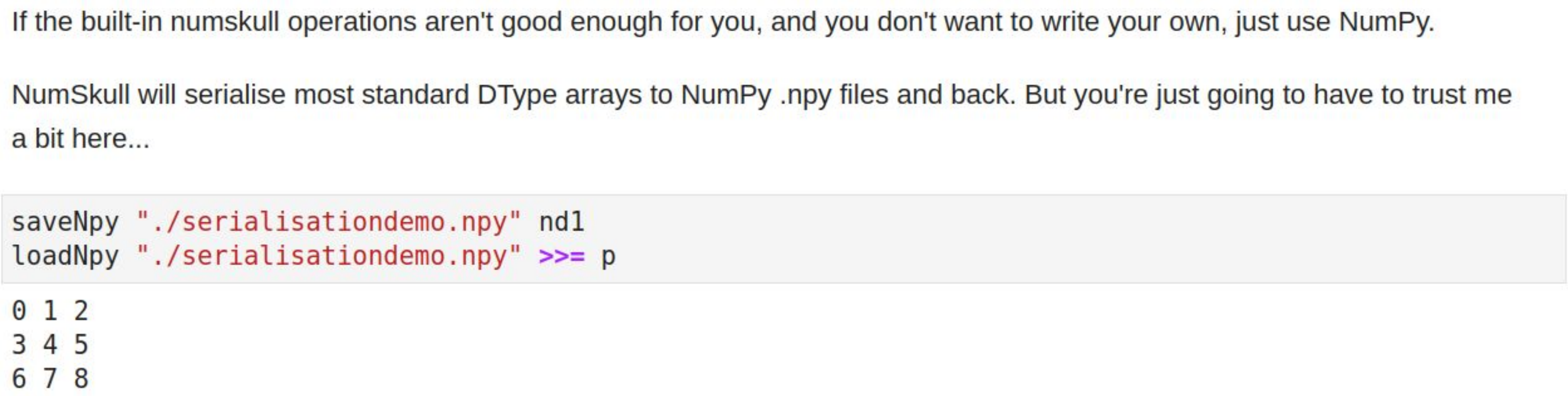


And do all sorts of fun maths!

```
nd1 = fromList [3,3] [0..8::Int]
nd2 = padShape [3,3] $ fromList [2,2] [10,10,10,10::Int]

putStrLn "Numeracy:"
p $   nd1 + nd2
p $   Numskull.sum [nd1, nd2]
p $   nd1 * nd2

putStrLn "Powers/logs:"
p $   elemPow nd1 (fromList [3,3] $ replicate 9 (2::Int))

putStrLn "Average:"
p $   mean [nd1, nd2]

putStrLn "Transpose & diagonal:"
p $   transpose nd1
p $   diagonal nd1

putStrLn "Matrix multiplication:"
nd3 = fromList [2,2] [0..3::Float]
nd4 = fromList [2,2] [4..7::Float]
p $   matMul nd3 nd3
m = fromJust (gemm nd3 nd3 nd4 True False 3 1)
p     m

putStrLn "Determinant:"
print (determinant m :: [Float])
```



```
Numeracy:
10 11  2
13 14  5
 6  7  8
10 11  2
13 14  5
 6  7  8
 0 10  0
30 40  0
 0  0  0
Powers/logs:
 0  1  4
 9 16 25
36 49 64
Average:
5 5 1
6 7 2
3 3 4
Transpose & diagonal:
0 3 6
1 4 7
2 5 8
0 4 8
Matrix multiplication:
 2.0  3.0
 6.0 11.0
16.0 23.0
24.0 37.0
Determinant:
[40.0]
```



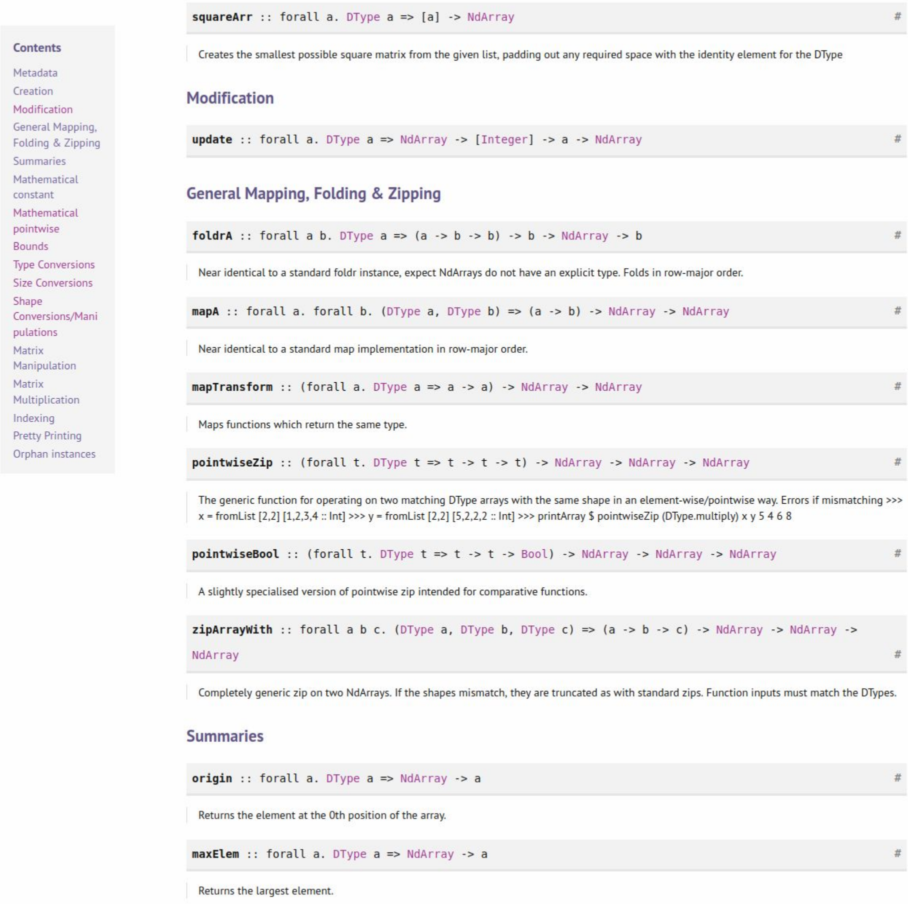
If the built-in numskull operations aren't good enough for you, and you don't want to write your own, just use NumPy.

NumSkull will serialise most standard DType arrays to NumPy .npy files and back. But you're just going to have to trust me a bit here...

```
saveNpy "./serialisationdemo.npy" nd1
loadNpy "./serialisationdemo.npy" >>= p
```

```
0 1 2
3 4 5
6 7 8
```

See the docs for more



```
squareArr :: forall a. DType a => [a] -> NdArray                                  #
```

Creates the smallest possible square matrix from the given list, padding out any required space with the identity element for the DType

Modification

```
update :: forall a. DType a => NdArray -> [Integer] -> a -> NdArray               #
```

General Mapping, Folding & Zipping

```
foldrA :: forall a b. DType a => (a -> b -> b) -> b -> NdArray -> b               #
```

Near identical to a standard foldr instance, expect NdArrays do not have an explicit type. Folds in row-major order.

```
mapA :: forall a. forall b. (DType a, DType b) => (a -> b) -> NdArray -> NdArray  #
```

Near identical to a standard map implementation in row-major order.

```
mapTransform :: (forall a. DType a => a -> a) -> NdArray -> NdArray               #
```

Maps functions which return the same type.

```
pointwiseZip :: (forall t. DType t => t -> t -> t) -> NdArray -> NdArray -> NdArray  #
```

The generic function for operating on two matching DType arrays with the same shape in an element-wise/pointwise way. Errors if mismatching >>> x = fromList [2,2] [1,2,3,4 :: Int] >>> y = fromList [2,2] [5,2,2,2 :: Int] >>> printArray $ pointwiseZip (DType.multiply) x y 5 4 6 8

```
pointwiseBool :: (forall t. DType t => t -> t -> Bool) -> NdArray -> NdArray -> NdArray  #
```

A slightly specialised version of pointwise zip intended for comparative functions.

```
zipArrayWith :: forall a b c. (DType a, DType b, DType c) => (a -> b -> c) -> NdArray -> NdArray ->
NdArray                                                                           #
```

Completely generic zip on two NdArrays. If the shapes mismatch, they are truncated as with standard zips. Function inputs must match the DTypes.

Summaries

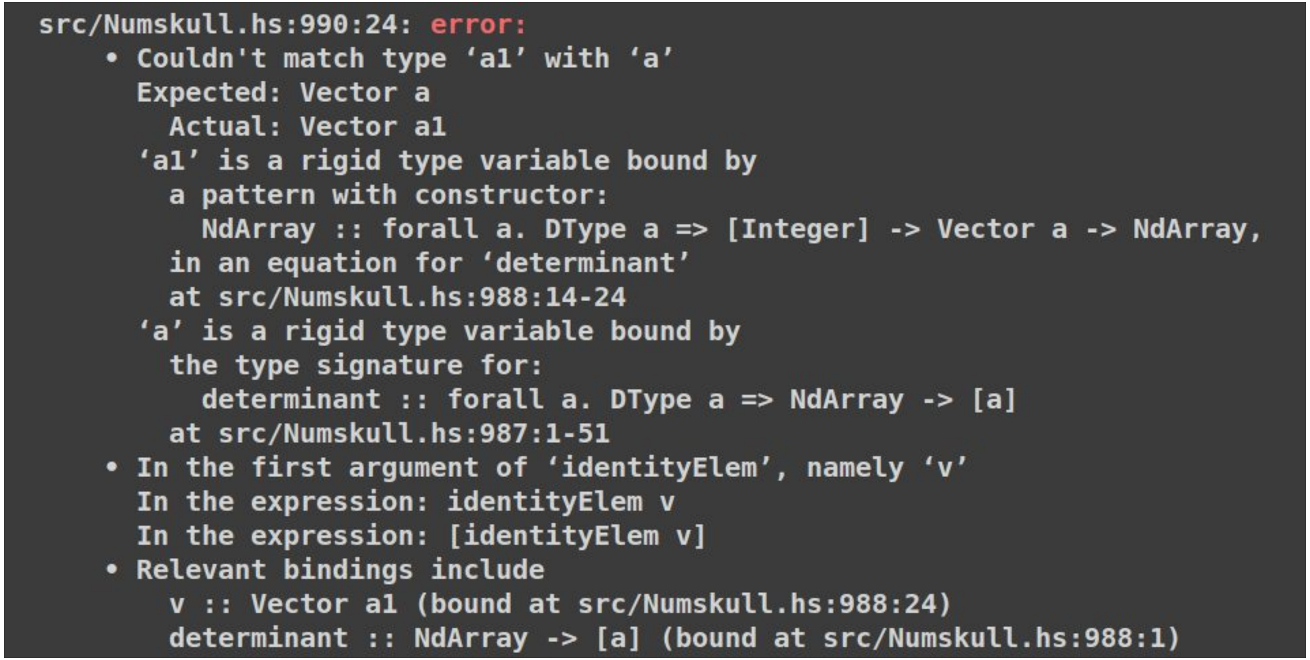
```
origin :: forall a. DType a => NdArray -> a                                       #
```

Returns the element at the 0th position of the array.

```
maxElem :: forall a. DType a => NdArray -> a                                      #
```

Returns the largest element.

An old friend...

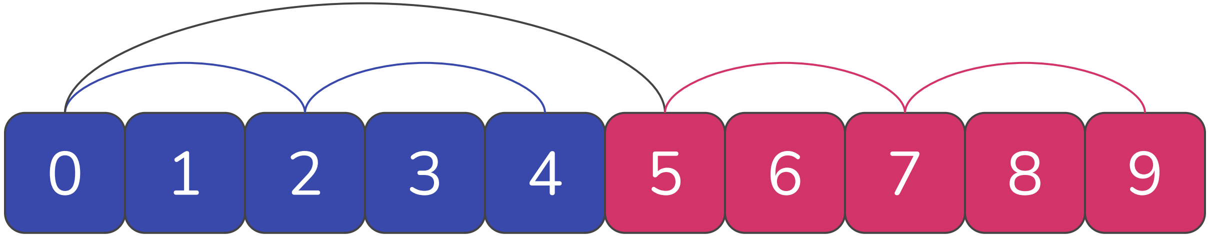


```
src/Numskull.hs:990:24: error:
    • Couldn't match type ‘a1’ with ‘a’
      Expected: Vector a
        Actual: Vector a1
      ‘a1’ is a rigid type variable bound by
        a pattern with constructor:
          NdArray :: forall a. DType a => [Integer] -> Vector a -> NdArray,
        in an equation for ‘determinant’
        at src/Numskull.hs:988:14-24
      ‘a’ is a rigid type variable bound by
        the type signature for:
          determinant :: forall a. DType a => NdArray -> [a]
        at src/Numskull.hs:987:1-51
    • In the first argument of ‘identityElem’, namely ‘v’
      In the expression: identityElem v
      In the expression: [identityElem v]
    • Relevant bindings include
        v :: Vector a1 (bound at src/Numskull.hs:988:24)
        determinant :: NdArray -> [a] (bound at src/Numskull.hs:988:1)
```

Numskull 2.0

Making Strides

```
data NdArray where
      NdArray :: DType a => Vector Int -> Vector Int -> Vector a -> NdArray
```



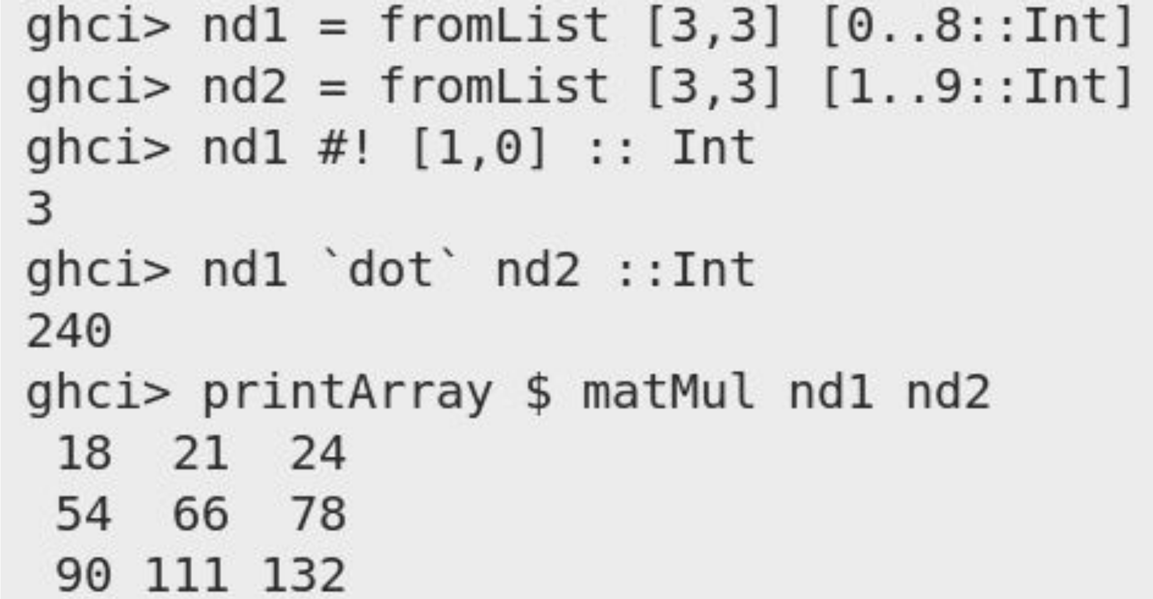
Stride: (5,2) →

0	2	4
5	7	9

Quick & Elegant

- Indexing
- Broadcasting
- Transposition

```
transpose :: NdArray -> NdArray
transpose (NdArray sh st v) = NdArray (reverse sh) (reverse st) v
```



```
ghci> nd1 = fromList [3,3] [0..8::Int]
ghci> nd2 = fromList [3,3] [1..9::Int]
ghci> nd1 #! [1,0] :: Int
3
ghci> nd1 `dot` nd2 ::Int
240
ghci> printArray $ matMul nd1 nd2
 18  21  24
 54  66  78
 90 111 132
```

Further Work

More of the...

- Onnx functions
- NumPy-like maths
- QuasiQuote elegance

Additional

- BLAS support
- Explicit parallelism

Thanks for looking after me!

Questions?