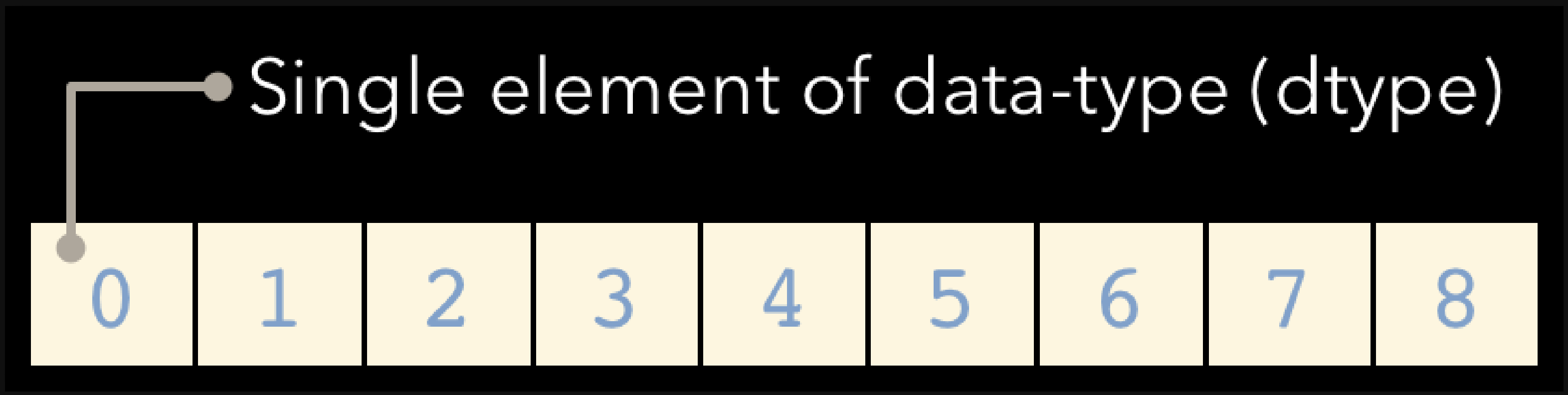
# Introduction to NumPy

# What is NumPy

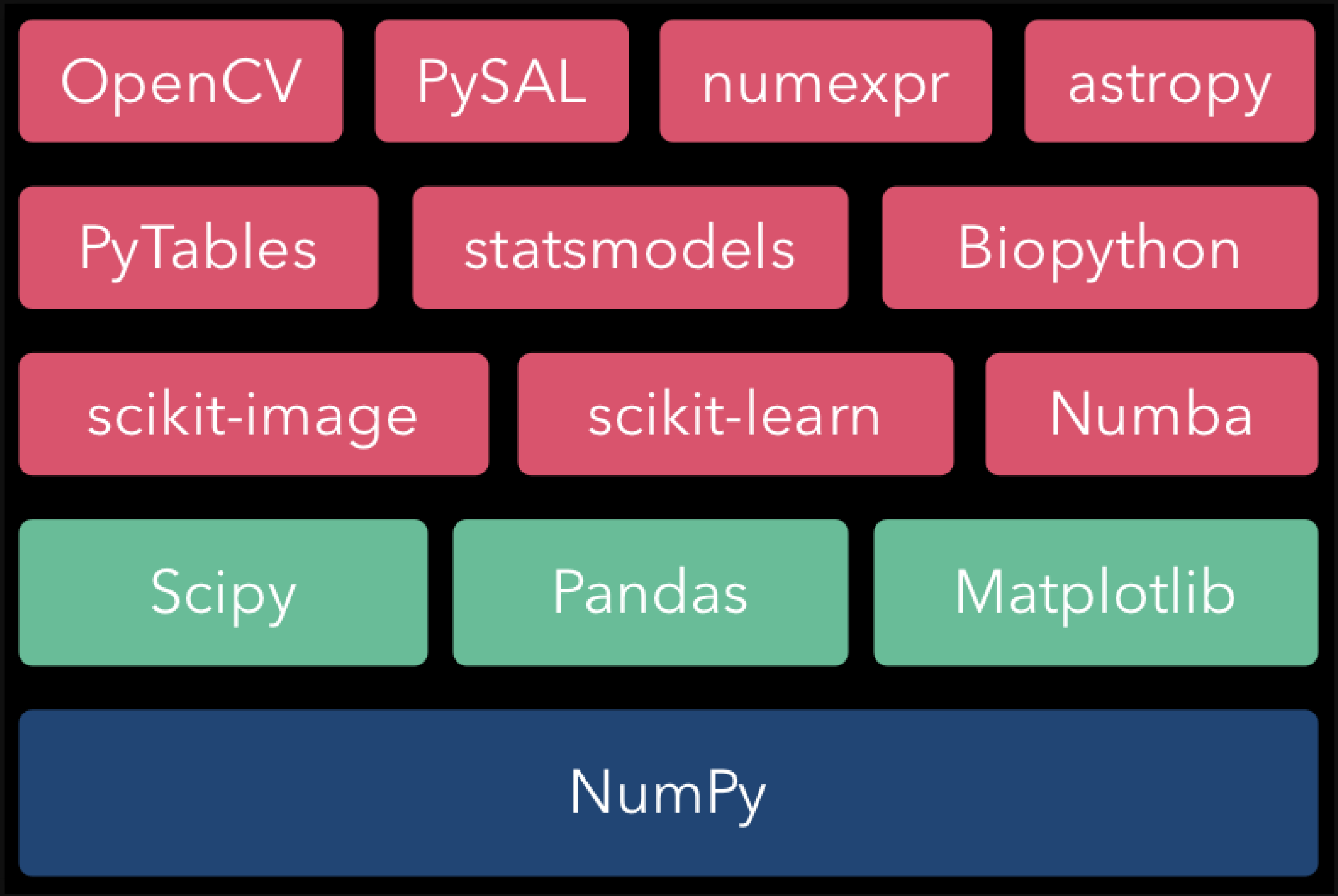
- NumPy is a Python C extension library for array-oriented computing
  - Efficient
  - In-memory
  - Contiguous (or Strided)
  - Homogeneous (but types can be algebraic)




- NumPy is suited to many applications
  - Image processing
  - Signal processing
  - Linear algebra
  - A plethora of others

# NumPy is the foundation of the python scientific stack

# NumPy Ecosystem

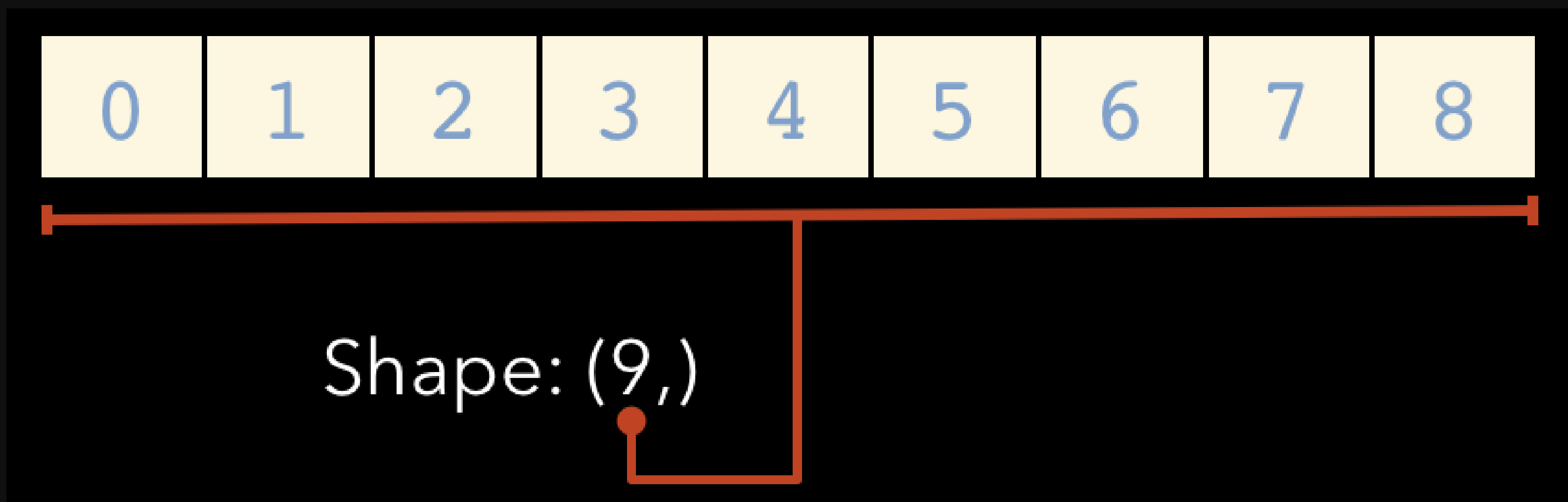


# Quick Start

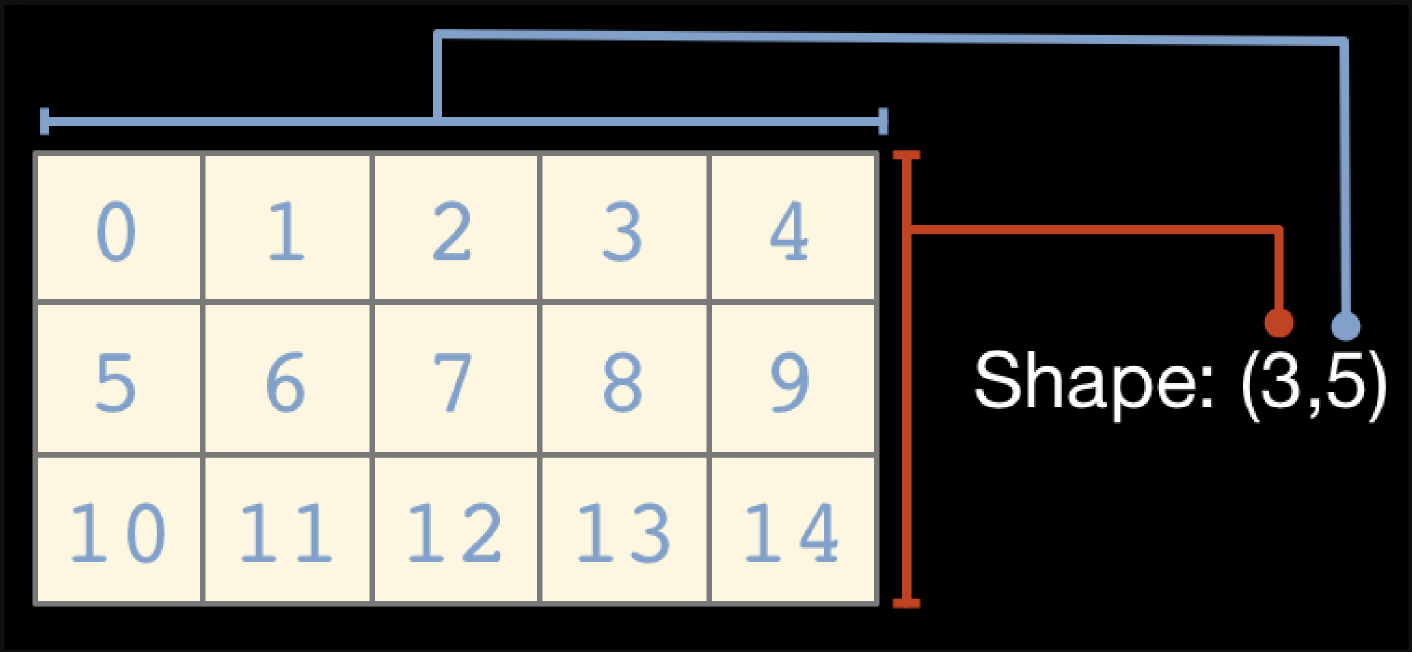
```
In [1]: import numpy as np

In [2]: a = np.array([1,2,3,4,5,6,7,8,9])

In [3]: a
Out[3]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])

In [4]: b = a.reshape((3,3))

In [5]: b
Out[5]:
array([[1, 2, 3],
[4, 5, 6],
[7, 8, 9]])

In [6]: b * 10 + 4
Out[6]:
array([[14, 24, 34],
[44, 54, 64],
[74, 84, 94]])
```

# Array Shape

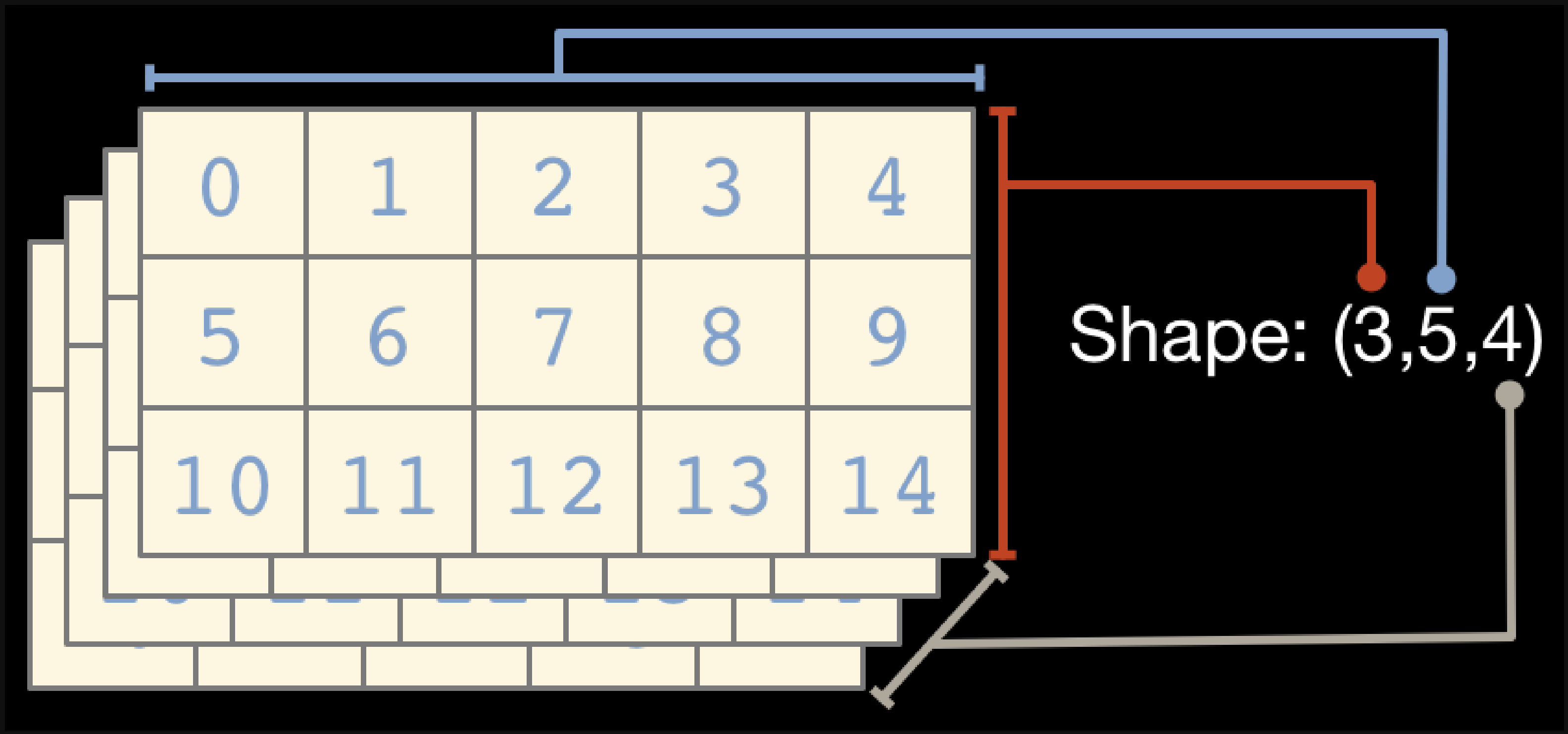
## One dimensional arrays have a 1-tuple for their shape



# ...Two dimensional arrays have a 2-tuple



# ...And so on



# Array Element Type (dtype)

- NumPy arrays comprise elements of a single data type
- The type object is accessible through the `.dtype` attribute

Here are a few of the most important attributes of dtype objects

- `dtype.byteorder` — big or little endian
- `dtype.itemsize` — element size of this dtype
- `dtype.name` — a name for this dtype object
- `dtype.type` — type object used to create scalars

There are many others...

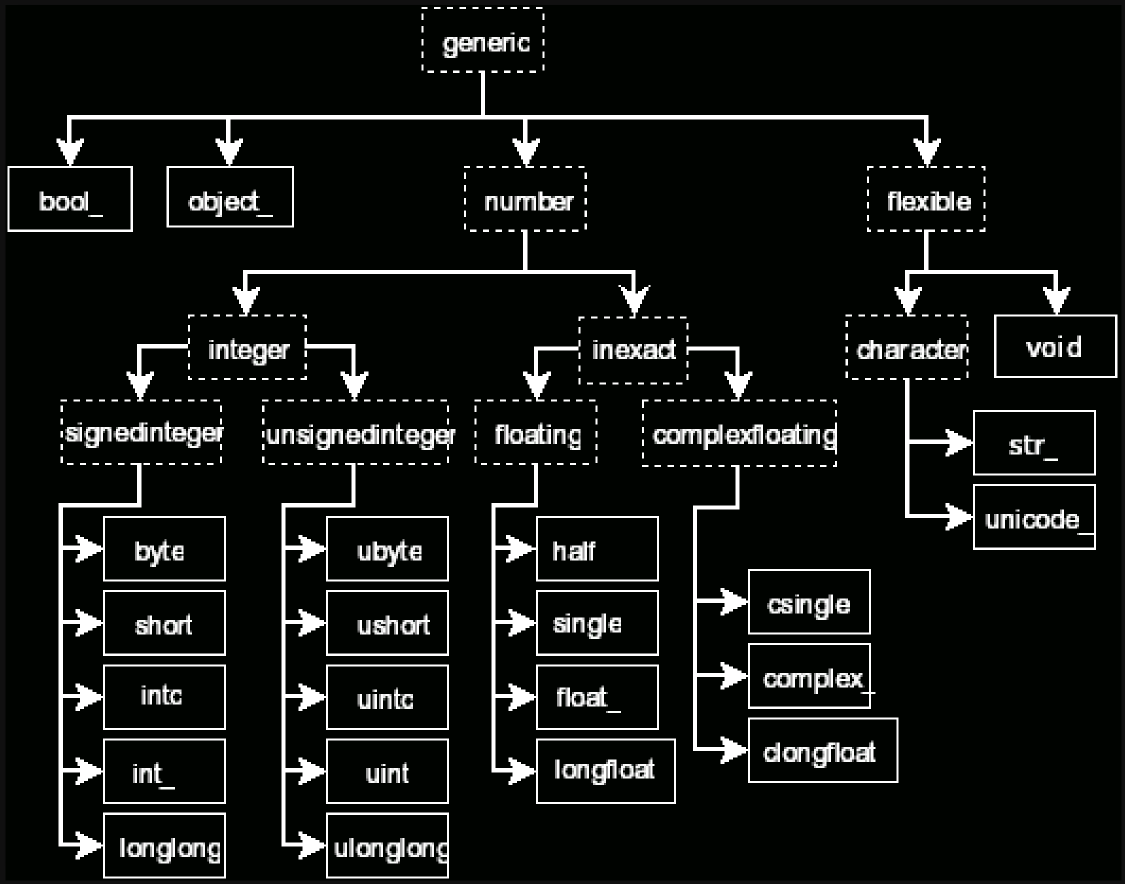
## Array dtypes are usually inferred automatically

```
In [16]: a = np.array([1,2,3])

In [17]: a.dtype
Out[17]: dtype('int64')

In [18]: b = np.array([1,2,3,4.567])

In [19]: b.dtype
Out[19]: dtype('float64')
```

## But can also be specified explicitly

```
In [20]: a = np.array([1,2,3], dtype=np.float32)

In [21]: a.dtype
Out[21]: dtype('int64')

In [22]: a
Out[22]: array([ 1.,  2.,  3.], dtype=float32)
```

# NumPy Builtin dtype Hierarchy




`np.datetime64` is a new addition in NumPy 1.7

# Array Creation

Explicitly from a list of values

```
In [2]: np.array([1,2,3,4])
Out[2]: array([1, 2, 3, 4])
```

As a range of values

```
In [3]: np.arange(10)
Out[3]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

By specifying the number of elements

```
In [4]: np.linspace(0, 1, 5)
Out[4]: array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
```

## Zero-initialized

```
In [4]: np.zeros((2,2))
Out[4]:
array([[ 0.,  0.],
[ 0.,  0.]])
```

## One-initialized

```
In [5]: np.ones((1,5))
Out[5]: array([[ 1.,  1.,  1.,  1.,  1.]])
```

## Uninitialized

```
In [4]: np.empty((1,3))
Out[4]: array([[  2.12716633e-314,   2.12716633e-314,   2.15203762e-314]])
```

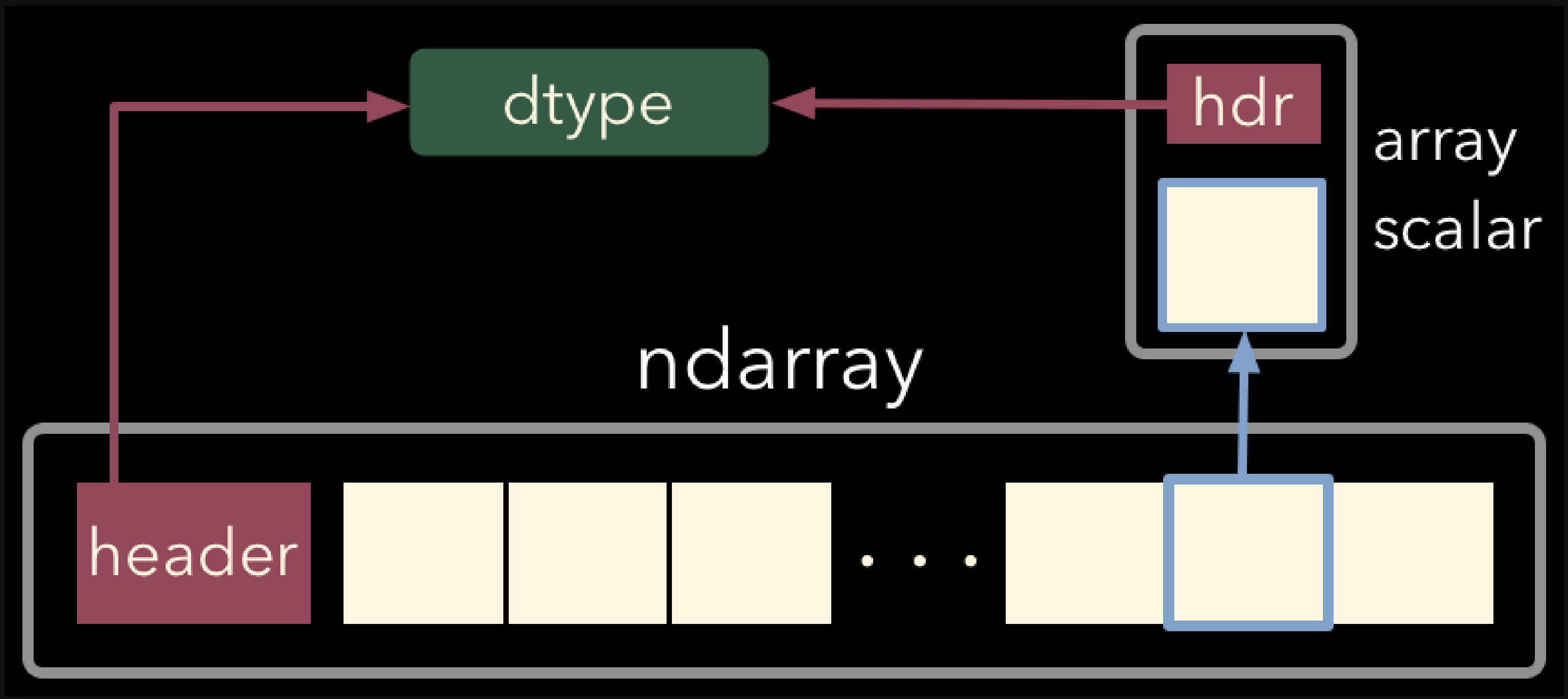
# Constant diagonal value

```
In [6]: np.eye(3)
Out[6]:
array([[ 1.,  0.,  0.],
[ 0.,  1.,  0.],
[ 0.,  0.,  1.]])
```

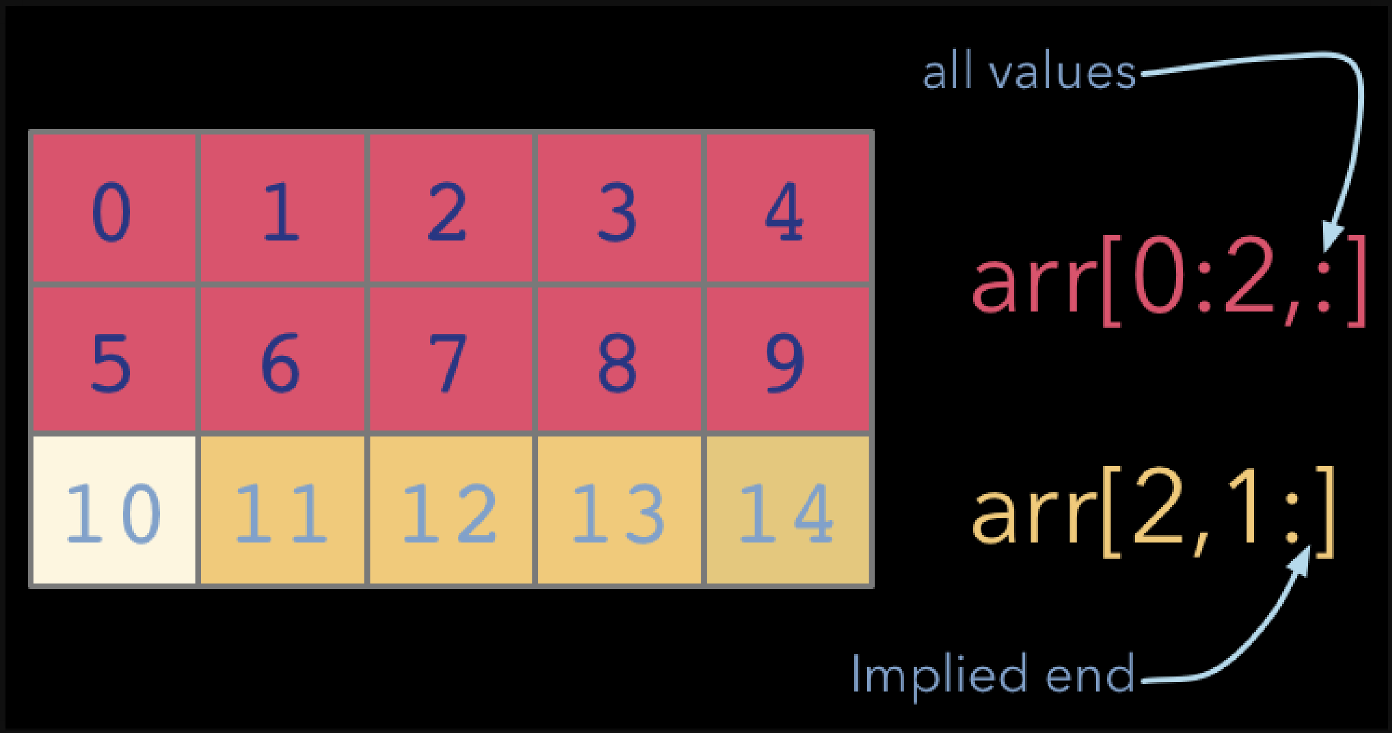
# Multiple diagonal values

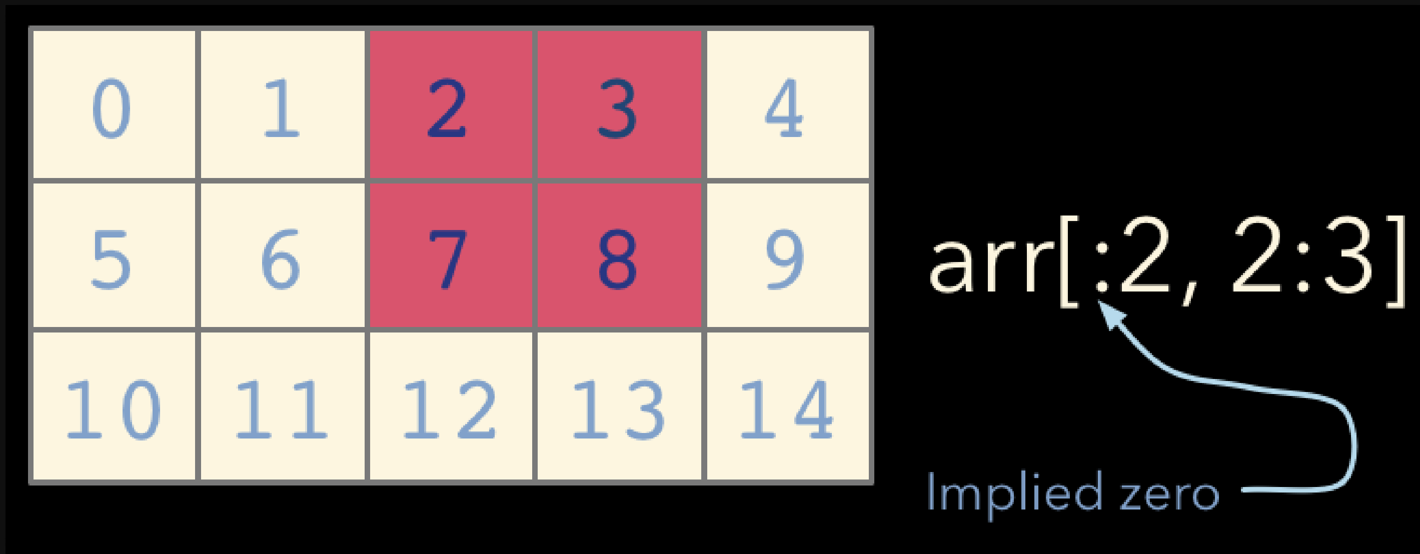
```
In [7]: np.diag([1,2,3,4])
Out[7]:
array([[1, 0, 0, 0],
[0, 2, 0, 0],
[0, 0, 3, 0],
[0, 0, 0, 4]])
```

# Array Memory Layout

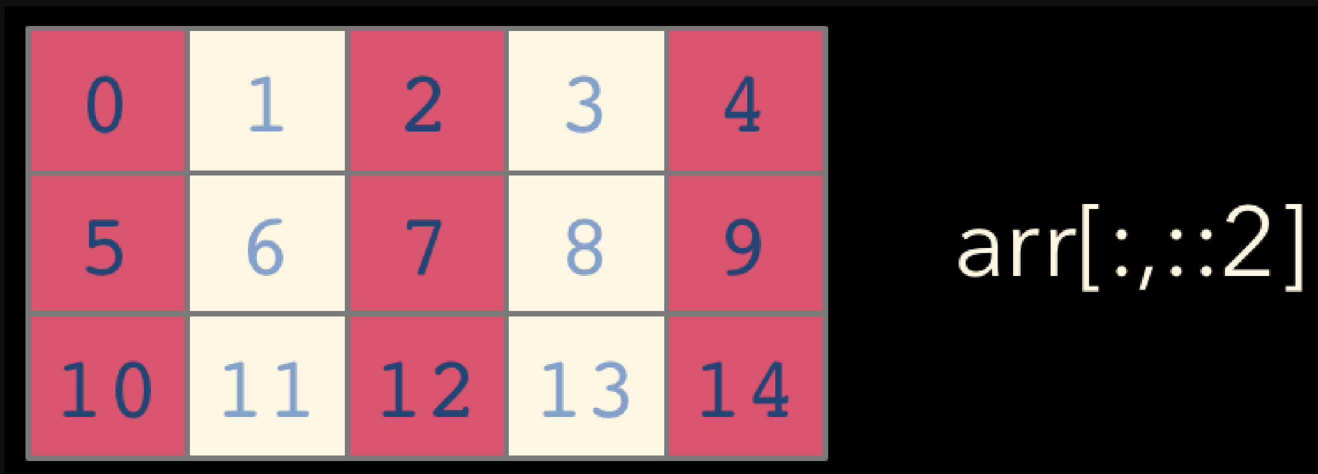


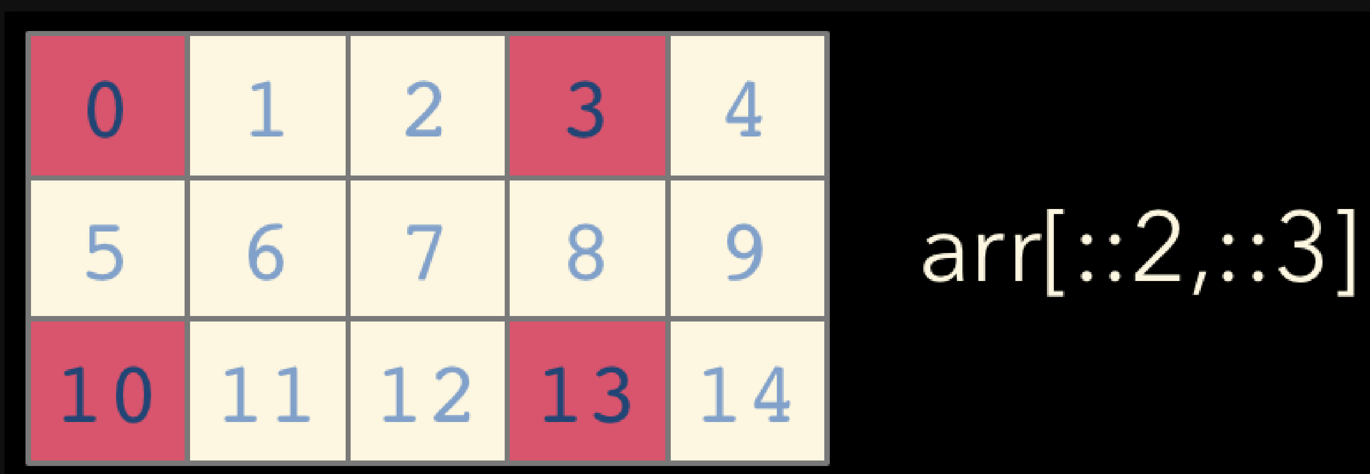
# Indexing and Slicing




0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
arr[:2, 2:3]
Implied zero

NumPy array indices can also take an optional stride



# Array Views

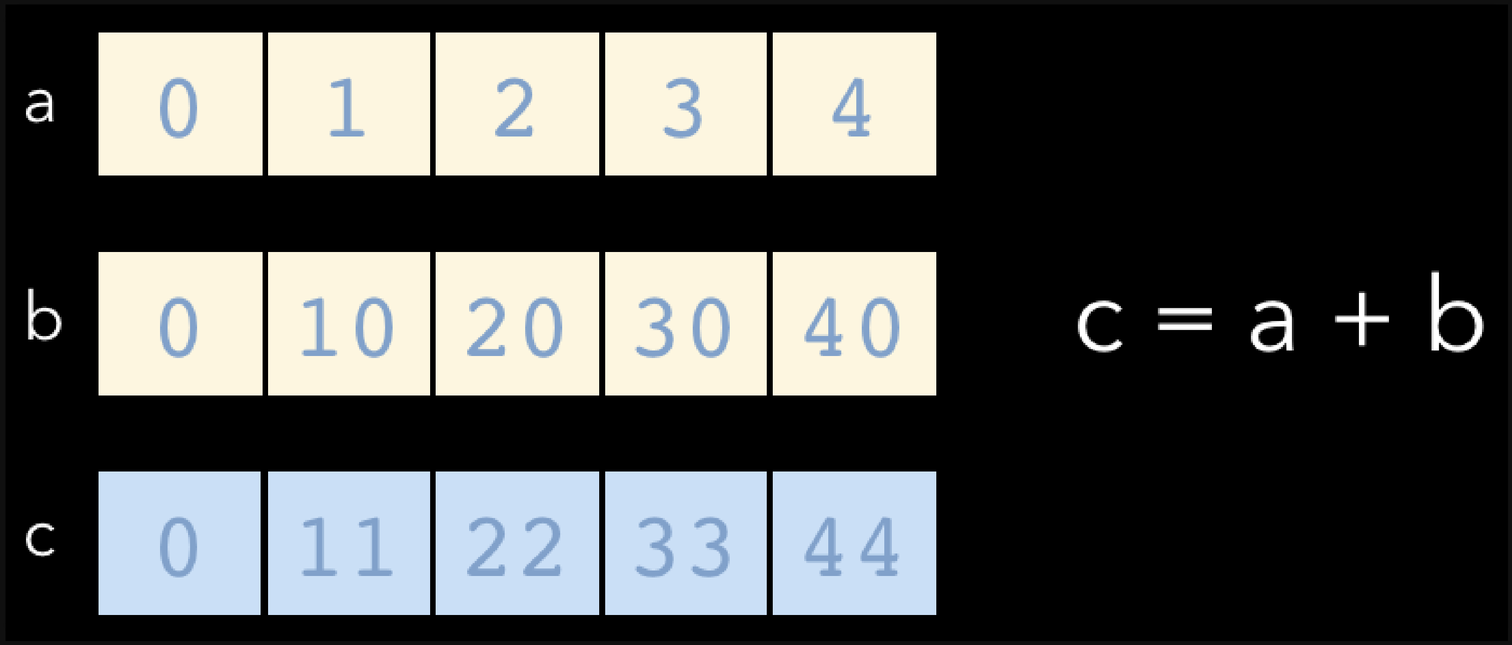
Simple assigments do not make copies of arrays (same semantics as Python). Slicing operations do not make copies either; they return views on the original array.

```
In [2]: a = np.arange(10)

In [3]: b = a[3:7]

In [4]: b
Out[4]: array([3, 4, 5, 6])

In [5]: b[:] = 0

In [6]: a
Out[6]: array([0, 1, 3, 0, 0, 0, 0, 7, 8, 9])

In [7]: b.flags.owndata
Out[7]: False
```

Array views contain a pointer to the original data, but may have different shape or stride values. Views always have `flags.owndata` equal to `False`.

# Universal Functions (ufuncs)

NumPy ufuncs are functions that operate element-wise on one or more arrays




ufuncs dispatch to optimized C inner-loops based on array dtype

# NumPy has many built-in ufuncs

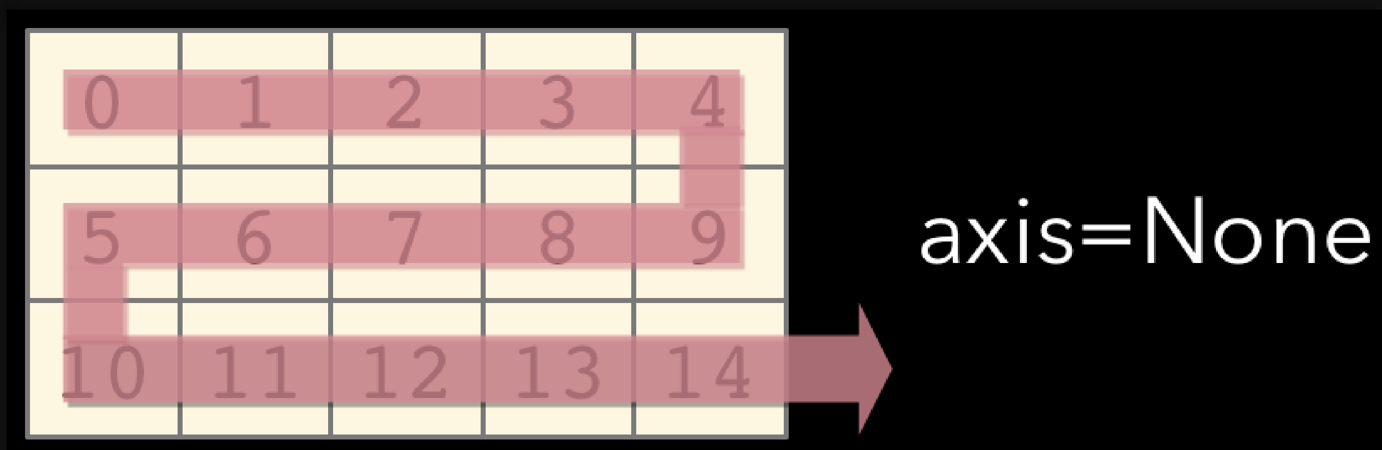
- comparison: `<, <=, ==, !=, >=, >`
- arithmetic: `+, -, *, /, reciprocal, square`
- exponential: `exp, expm1, exp2, log, log10, log1p, log2, power, sqrt`
- trigonometric: `sin, cos, tan, acsin, arccos, atctan`
- hyperbolic: `sinh, cosh, tanh, acsinh, arccosh, atctanh`
- bitwise operations: `&, |, ~, ^, left_shift, right_shift`
- logical operations: `and, logical_xor, not, or`
- predicates: `isfinite, isinf, isnan, signbit`
- other: `abs, ceil, floor, mod, modf, round, sinc, sign, trunc`

# Axis

Array method reductions take an optional `axis` parameter that specifies over which axes to reduce

`axis=None` reduces into a single scalar

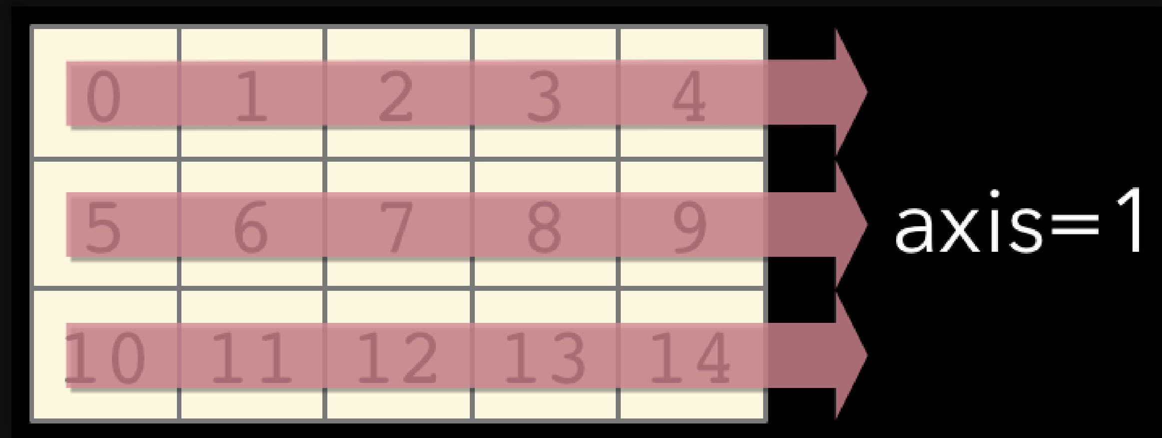
```
In [7]: a.sum
()
Out[7]: 105
```



`axis=None` is the default

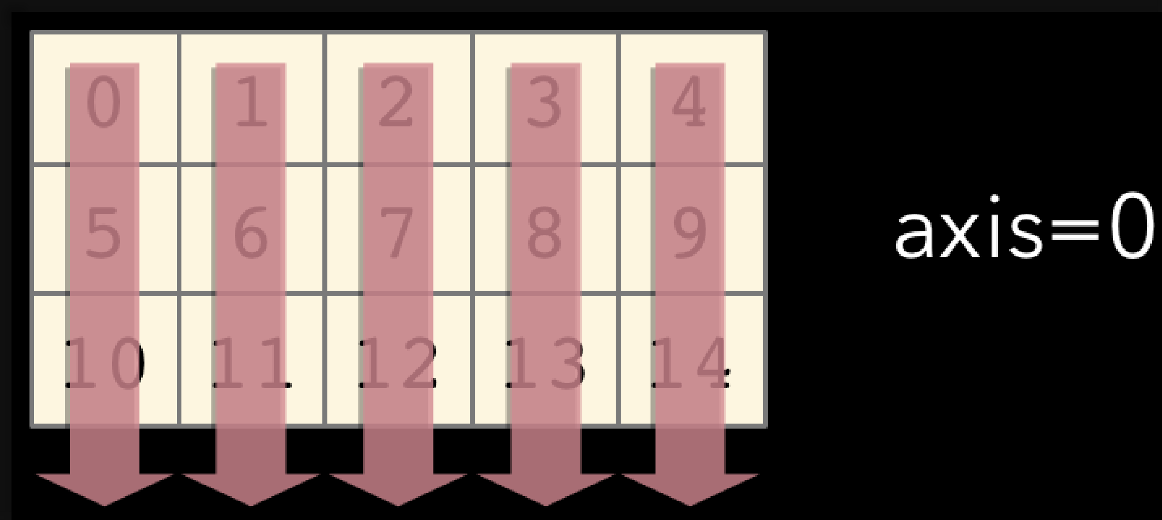
## `axis=0` reduces into the zeroth dimension

```
In [8]: a.sum(axis=0)
Out[8]: array([15, 18, 21, 24,
27])
```



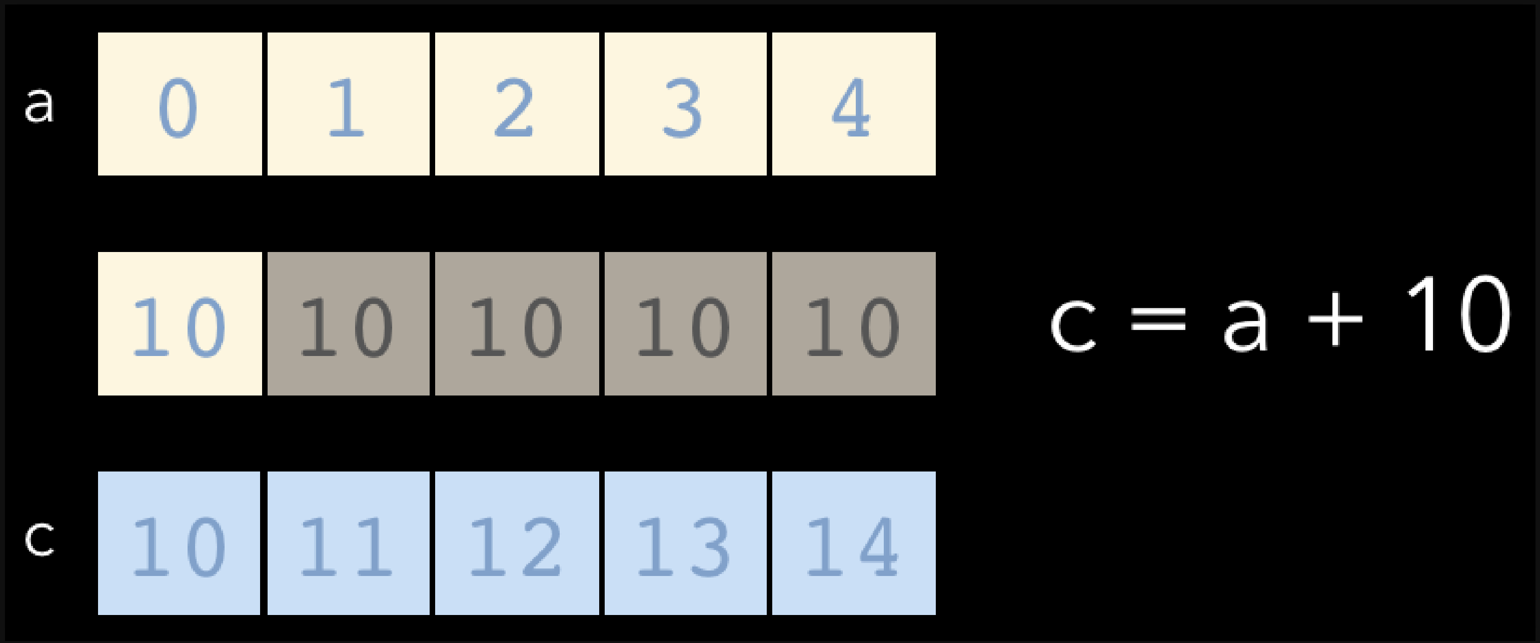

## `axis=0` reduces into the first dimension

```
In [9]: a.sum(axis=1)
Out[9]: array([10, 35, 60])
```



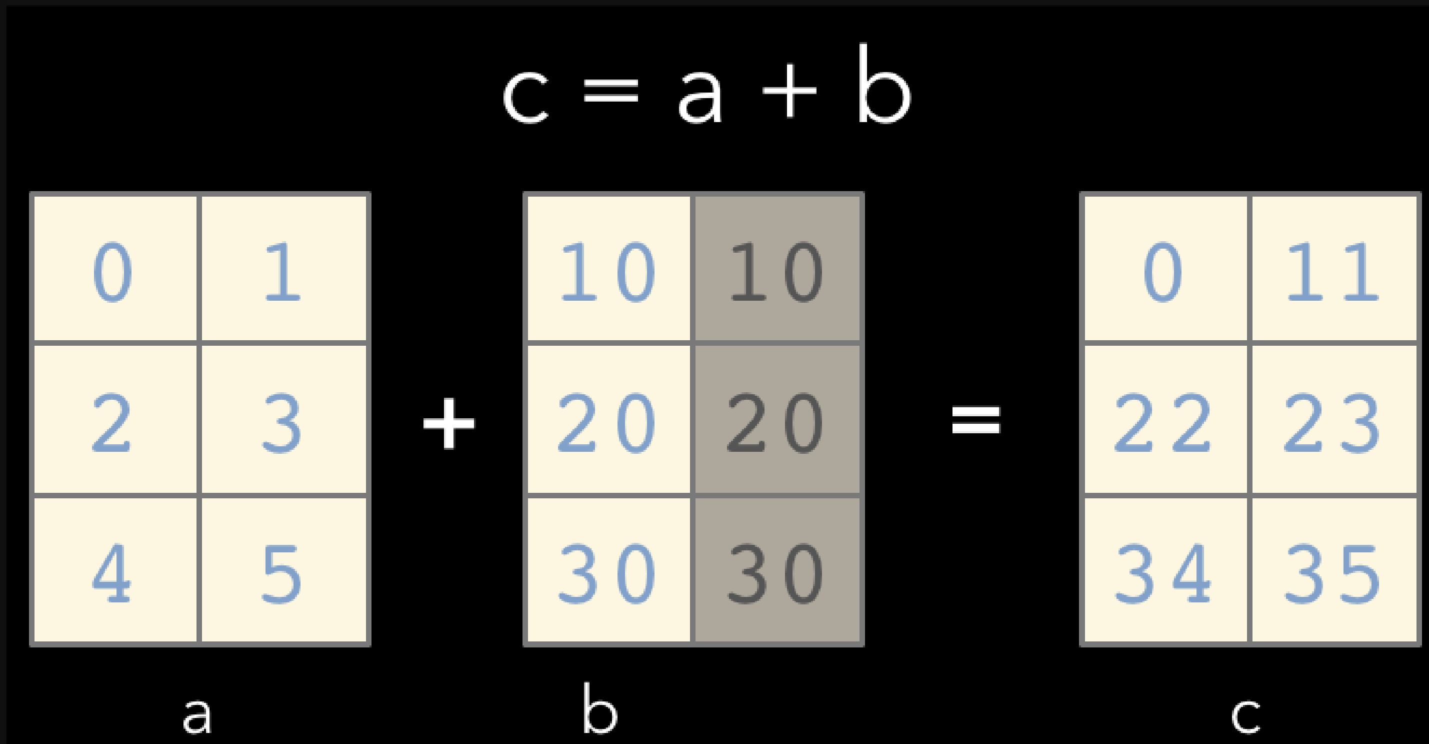
# Broadcasting

A key feature of NumPy is broadcasting, where arrays with different, but compatible shapes can be used as arguments to ufuncs




In this case an array scalar is broadcast to an array with shape `(5, )`

# A slightly more involved broadcasting example in two dimensions




Here an array of shape `(3, 1)` is broadcast to an array with shape `(3, 2)`

# Broadcasting Rules

In order for an operation to broadcast, the size of all the trailing dimensions for both arrays must either:

be **equal** OR be **one**

```
A      (1d array):                3
B      (2d array):            2 x 3
Result (2d array):            2 x 3
```

```
A      (2d array):            6 x 1
B      (3d array):        1 x 6 x 4
Result (3d array):        1 x 6 x 4
```

```
A      (4d array):    3 x 1 x 6 x 1
B      (3d array):        2 x 1 x 4
Result (4d array):    3 x 2 x 6 x 4
```

# Square Peg in a Round Hole

If the dimensions do not match up, `np.newaxis` may be useful

```
In [16]: a = np.arange(6).reshape((2, 3))

In [17]: b = np.array([10, 100])

In [18]: a * b
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
 in ()
----> 1 a * b

ValueError: operands could not be broadcast together with shapes (2,3) (2)

In [19]: b[:,np.newaxis].shape
Out[19]: (2, 1)

In [20]: a *b[:,np.newaxis]
Out[20]:
array([[  0,  10,  20],
[300, 400, 500]])
```

# Array Methods

- Predicates
  - `a.any(), a.all()`
- Reductions
  - `a.mean(), a.argmin(), a.argmax(), a.trace(), a.cumsum(), a.cumprod()`
- Manipulation
  - `a.argsort(), a.transpose(), a.reshape(...), a.ravel(), a.fill(...), a.clip(...)`
- Complex Numbers
  - `a.real, a.imag, a.conj()`

# Fancy Indexing

NumPy arrays may be used to index into other arrays

```
In [2]: a = np.arange(15).reshape((3,5))

In [3]: a
Out[3]:
array([[ 0,  1,  2,  3,  4],
[ 5,  6,  7,  8,  9],
[10, 11, 12, 13, 14]])
In [4]: i = np.array([[0,1], [1, 2]])

In [5]: j = np.array([[2, 1], [4, 4]])

In [6]: a[i,j]
Out[6]:
array([[ 2,  6],
[ 9, 14]])
```

# Boolean arrays can also be used as indices into other arrays

```
In [2]: a = np.arange(15).reshape((3,5))

In [3]: a
Out[3]:
array([[ 0,  1,  2,  3,  4],
[ 5,  6,  7,  8,  9],
[10, 11, 12, 13, 14]])

In [4]: b = (a % 3 == 0)

In [5]: b
Out[5]:
array([[ True, False, False,  True, False],
[False,  True, False, False,  True],
[False, False,  True, False, False]], dtype=bool)

In [6]: a[b]
Out[6]: array([ 0,  3,  6,  9, 12])
```

# NumPy Functions

- Data I/O
  - `fromfile`, `genfromtxt`, `load`, `loadtxt`, `save`, `savetxt`
- Mesh Creation
  - `mgrid`, `meshgrid`, `ogrid`
- Manipulation
  - `einsum`, `hstack`, `take`, `vstack`

# Array Subclasses

- `numpy.ma` — Masked arrays
- `numpy.matrix` — Matrix operators
- `numpy.memmap` — Memory-mapped arrays
- `numpy.recarray` — Record arrays

# Other Subpackages

- `numpy.fft` — Fast Fourier transforms
- `numpy.polynomial` — Efficient polynomials
- `numpy.linalg` — Linear algebra
  - `cholesky, det, eig, eigvals, inv, lstsq, norm, qr, svd`
- `numpy.math` — C standard library math functions
- `numpy.random` — Random number generation
  - `beta, gamma, geometric, hypergeometric, lognormal, normal, poisson, uniform, weibull`

# Examples

# FFT

```
import numpy as np
t = np.linspace(0,120,4000)
PI = np.pi
signal = 12*np.sin(3 * 2*PI*t)          # 3 Hz
signal += 6*np.sin(8 * 2*PI*t)          # 8 Hz
signal += 1.5*np.random.random(len(t)) # noise
FFT = abs(np.fft.fft(signal))
freqs = np.fft.fftfreq(signal.size, t[1]-t[0])
```