Enthought

Introduction to Numerical Computing with Numpy

Enthought, Inc. 200 W Cesar Chavez Suite 202 Austin, TX 78701
www.enthought.com

Q2-2023
letter
3.7.0

Introduction to Numerical Computing with Numpy

Enthought, Inc.
www.enthought.com

Introduction to Numerical Computing with NumPy

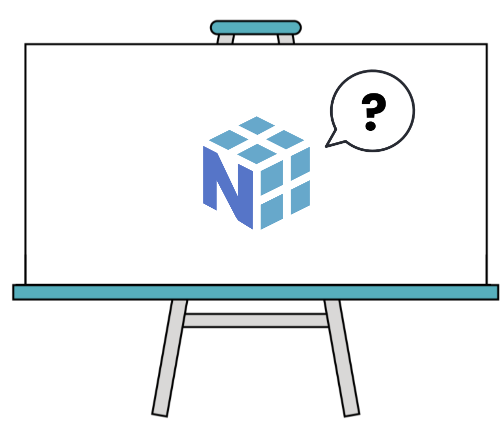
SciPy Conference Tutorial
2023

NumPy

The Standard Numerical Library for Python

NumPy: The Standard Numerical Library for Python

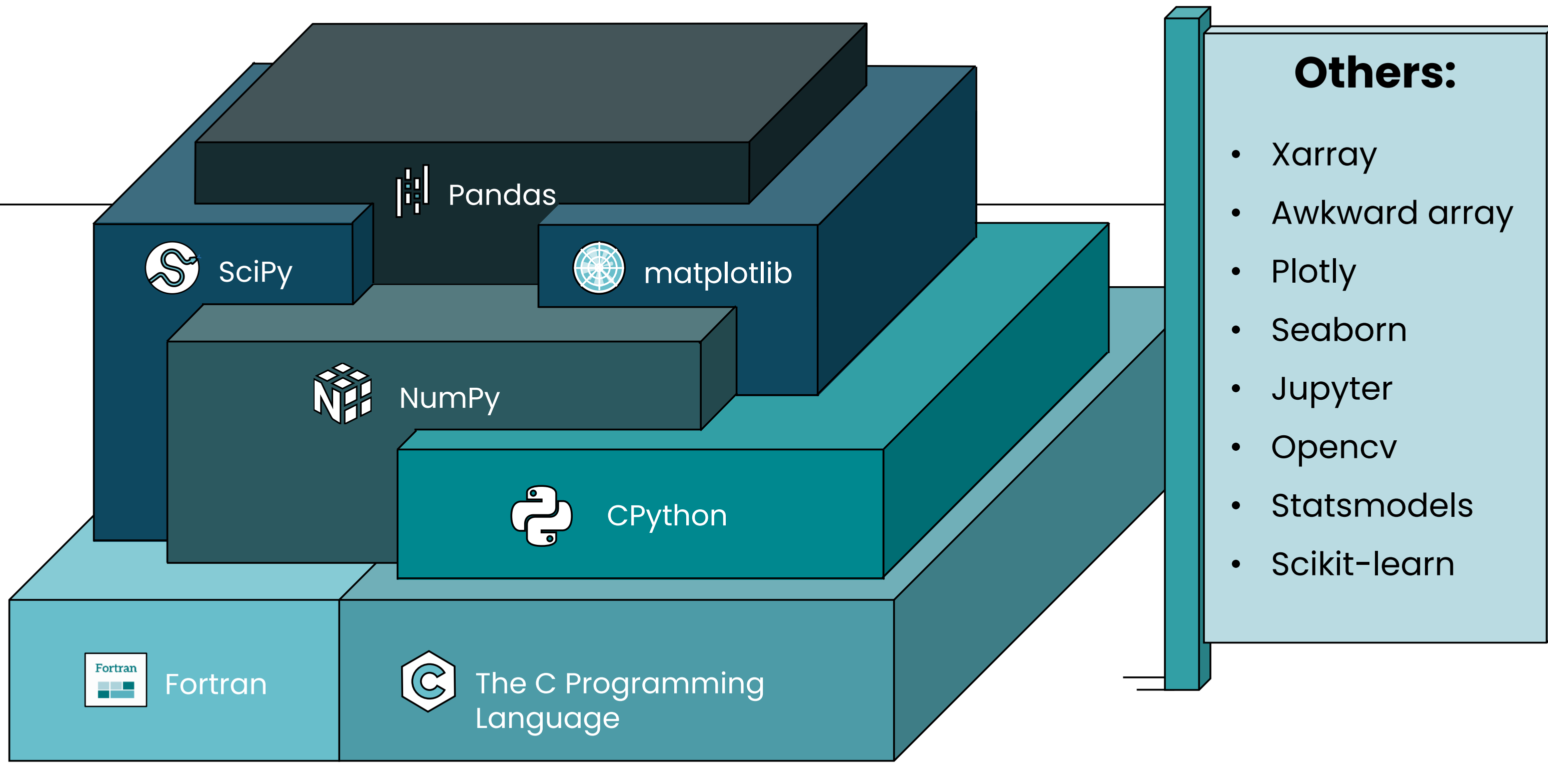
Syllabus



NumPy

Why NumPy?

NumPy : Foundation of Scientific Python



Python Lists and Math

ARITHMETIC

- Python lists are not made for arithmetic.
- For example, list addition is concatenation.

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a + b
[1, 2, 3, 4, 5, 6]
```

- To mathematically add two lists by position (like we would do with two vectors), we have to use loops.

```
>>> r = []
>>> for v_a, v_b in zip(a, b):
...     r.append(v_a + v_b)
>>> print(r)
[5, 7, 9]
```

- For large lists, Python loops are slow.

MATHMATICAL FUNCTIONS

- Python mathematical functions handle numeric types, not lists.

```
>>> import math
>>> c = [1.1, 2.2, 3.3]
>>> math.trunc(c)
TypeError: type list doesn't define
__trunc__ method
```

- To apply a mathematical function, we have to use loops.

```
>>> r = []
>>> for v_c in c:
...     r.append(math.trunc(v_c))
>>> print(r)
[1, 2, 3]
```

NumPy Arrays and Math

ARITHMETICAL OPERATIONS

- Array addition applies to the entire array and is **addition by position**.
- NumPy arrays are **vectorized**; no loops needed.

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> b = np.array([4, 5, 6])
>>> a + b
array([5, 7, 9])
```

- No loops needed
  (and really fast even for large arrays).

```
# Other Operations
>>> a * b
array([4, 10, 18])

>>> a / b
array([0.25, 0.4 , 0.5 ])
```

SCALAR AND IN-PLACE OPERATIONS

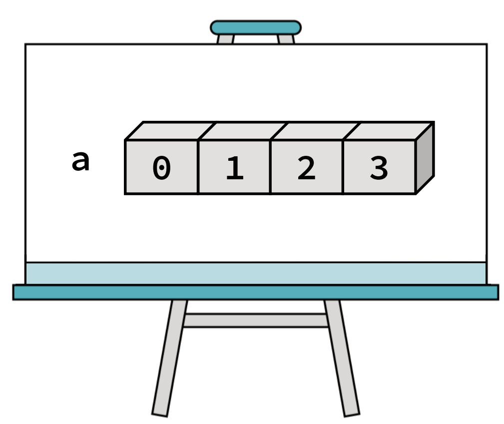
```
>>> a * 3
array([3, 6, 9])

>>> a += 7
>>> a
array([ 8,  9, 10])
```

MATHMATICAL FUNCTIONS

- NumPy mathematical functions handle arrays.

```
>>> c = np.array([1.1, 2.2, 3.3])
>>> np.trunc(c)
array([1., 2., 3.])

# Other Functions
>>> np.mean(a)
9.0

>>> np.exp(a / b)
array([7.3890561 , 6.04964746, 5.29449005])
```

Enthought



NumPy

Introducing Arrays

Enthought

Introducing NumPy Arrays

SIMPLE ARRAY CREATION

```
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
```

CHECKING THE TYPE

```
>>> type(a)
numpy.ndarray
```

NUMERIC "TYPE" OF ELEMENTS

```
>>> a.dtype
dtype('int32')
```

NUMBER OF DIMENSIONS

```
>>> a.ndim
1
```

ARRAY SHAPE

```
# Shape returns a tuple listing the
# length of the array along each
# dimension.
>>> a.shape
(4,)
```

BYTES PER ELEMENT

```
>>> a.itemize
4
```

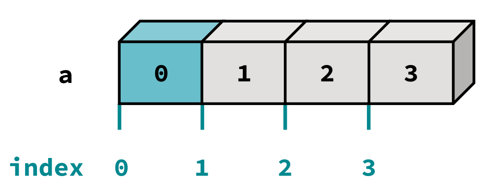
BYTES OF MEMORY USED

```
# Return the number of bytes used by
# the data portion of the array.
>>> a.nbytes
16
```

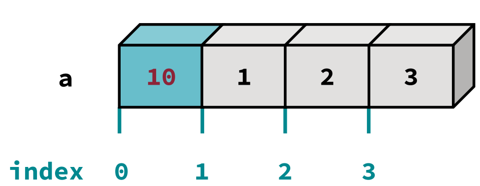

Array Indexing

ARRAY INDEXING

```
>>> a[0]
0
```




```
>>> a[0] = 10
>>> a
array([10, 1, 2, 3])
```



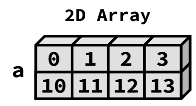
BEWARE OF TYPE COERCION

```
>>> a.dtype
dtype('int32')

# assigning a float into an int32
# array truncates the decimal part
>>> a[0] = 11.6
>>> a
array([11, 1, 2, 3])

# fill has the same behavior
>>> a.fill(-4.8)
>>> a
array([-4, -4, -4, -4])
```

Multi-Dimensional Arrays

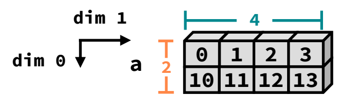
MULTI-DIMENSIONAL ARRAYS

```
>>> a = np.array([[ 0, 1, 2, 3],
...               [10,11,12,13]])
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,13]])
```



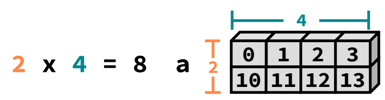
SHAPE = (ROWS, COLUMNS)

```
>>> a.shape
(2, 4)
```



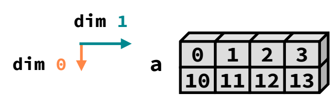
ELEMENT COUNT

```
>>> a.size
8
```



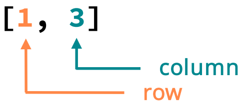
NUMBER OF DIMENSIONS

```
>>> a.ndim
2
```

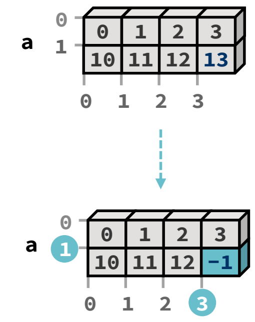


GET / SET ELEMENTS

```
>>> a[1, 3]
13
```

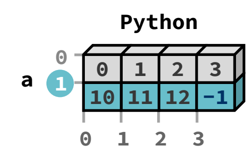



```
>>> a[1, 3] = -1
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,-1]])
```



ADDRESS 2ND (ONETH) ROW USING SINGLE INDEX

```
>>> a[1]
array([10, 11, 12, -1])
```



Enthought

Formatting Numeric Display

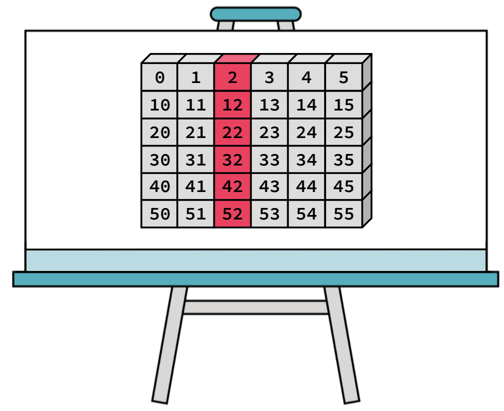
DEFAULT FORMATTING

```
>>> a = np.arange(1.0, 3.0, 0.5)
>>> a
array([1. , 1.5, 2. , 2.5])

>>> a * np.pi
array([3.14159265, 4.71238898,
6.28318531, 7.85398163])

>>> a * np.pi * 1e8
array([3.14159265e+08, 4.71238898e+08,
6.28318531e+08, 7.85398163e+08])

>>> a * np.pi * 1e-6
array([3.14159265e-06, 4.71238898e-06,
6.28318531e-06, 7.85398163e-06])
```

USER FORMATTING

```
# set precision
>>> np.set_printoptions(precision=2)

>>> a
array([1. , 1.5, 2. , 2.5])

>>> a * np.pi
array([3.14, 4.71, 6.28, 7.85])

>>> a * np.pi * 1e8
array([3.14e+08, 4.71e+08, 6.28e+08,
7.85e+08])

>>> a * np.pi * 1e-6
array([3.14e-06, 4.71e-06, 6.28e-06,
7.85e-06])

# suppress scientific notation
>>> np.set_printoptions(suppress=True)
>>> a * np.pi * 1e-6
array([0., 0., 0., 0.])
```

Enthought



NumPy

Indexing and Slicing

Array Slicing

`var[lower:upper:step]`

Extracts a portion of a sequence by specifying a lower and upper bound.
The lower-bound element is included, but the upper-bound element is **not** included.
Mathematically: [lower, upper). The step value specifies the stride between elements.

SLICING ARRAYS

```
#                  -5  -4  -3  -2  -1
# indices:          0   1   2   3   4
>>> a = np.array([10, 11, 12, 13, 14])

# [10, 11, 12, 13, 14]
>>> a[1:3]
array([11, 12])

# negative indices work also
>>> a[1:-2]
array([11, 12])
>>> a[-4:3]
array([11, 12])
```

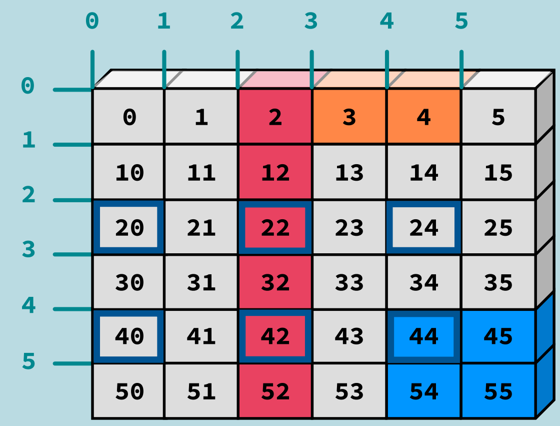
OMITTNG INDICES

```
# omitted boundaries are assumed to be the
# beginning (or end) of the array

# grab first three elements
>>> a[:3]
array([10, 11, 12])

# grab last two elements
>>> a[-2:]
array([13, 14])

# every other element
>>> a[::2]
array([10, 12, 14])
```

Array Slicing in 2-D

SLICING WORKS MUCH LIKE STANDARD PYTHON SLICING

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 45],
       [54, 55]])

>>> a[:, 2]
array([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```

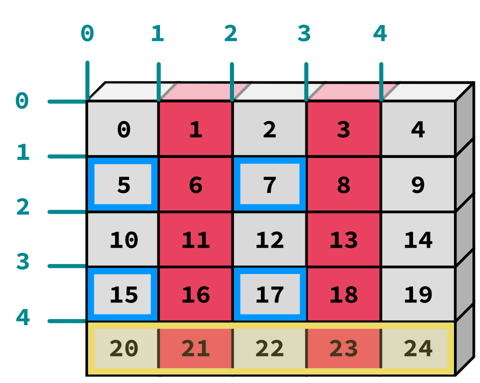


Give it a try! Slicing

1. Create the array below with the command

```
a = np.arange(25).reshape(5, 5)
```

2. Extract the slices as indicated



Assigning to a Slice

Slices are references to locations in memory.
These memory locations **can be used in assignment operations**.

```
>>> a = np.array([0, 1, 2, 3, 4])

# slicing the last two elements returns the data there
>>> a[-2:]
array([3, 4])

# we can insert an iterable of length two
>>> a[-2:] = [-1, -2]
>>> a
array([ 0,  1,  2, -1, -2])

# or a scalar value
>>> a[-2:] = 99
>>> a
array([ 0,  1,  2, 99, 99])
```


Sliced Arrays Share Data

Arrays created by slicing share data with the originating array.
Changing values in a slice also changes the original array.

```
>>> a = np.array([0, 1, 2, 3, 4])

# create a slice containing two elements of a
>>> b = a[2:4]
>>> b
array([2, 3])

>>> b[0] = 10

# changing b changed a!
>>> a
array([ 0,  1, 10, 3, 4])

>>> np.shares_memory(a, b)
True
```

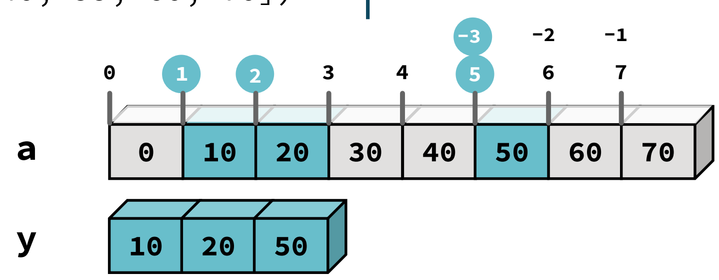
Fancy Indexing

INDEXING BY POSITION

```
>>> a = np.arange(0, 80, 10)

# fancy indexing
>>> indices = [1, 2, -3]
>>> y = a[indices]
>>> y
array([10, 20, 50])

# this also works with setting
>>> a[indices] = 99
>>> a
array([ 0, 99, 99, 30, 40, 99, 60, 70])
```

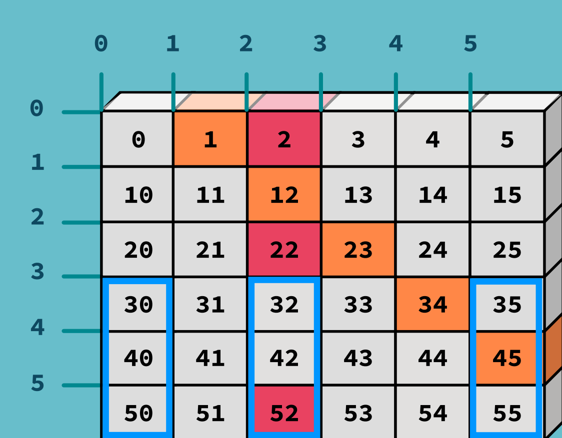
INDEXING WITH BOOLEANS

```
# manual creation of masks
>>> mask = np.array(
...     [0, 1, 1, 0, 0, 1, 0, 0],
...     dtype=bool)

# fancy indexing
>>> y = a[mask]
>>> y
array([99, 99, 99])
```



Fancy Indexing in 2-D

```
>>> a[[0, 1, 2, 3, 4],
...    [1, 2, 3, 4, 5]]
array([ 1, 12, 23, 34, 45])

>>> a[3:, [0, 2, 5]]
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])

>>> mask = np.array(
...     [1, 0, 1, 0, 0, 1],
...     dtype=bool)

>>> a[mask, 2]
array([2, 22, 52])
```



Unlike slicing, **fancy indexing creates copies** instead of a view into original array.

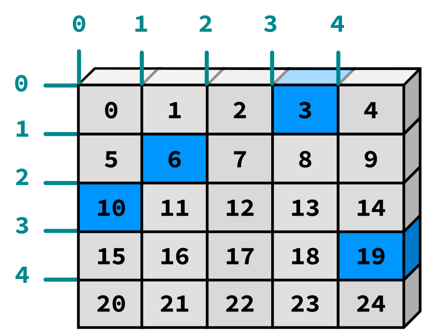
Give it a try! Fancy Indexing

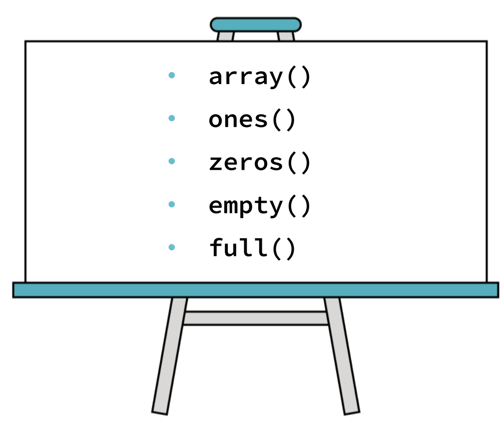
1. Create the array below with

    `a = np.arange(25).reshape(5, 5)`

    and extract the elements indicated in blue.

2. Extract all the numbers divisible by 3 using a boolean mask.





NumPy

Creating Arrays

Array Constructor Examples

FLOATING POINT ARRAYS

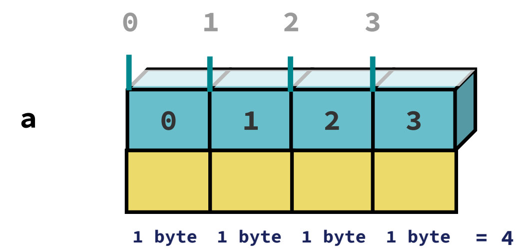
```
# Default to double precision
>>> a = np.array([0, 1.0, 2, 3])
>>> a.dtype
dtype('float64')

>>> a.nbytes
32
```

REDUCING PRECISION

```
>>> a = np.array([0, 1., 2, 3],
...              dtype='float32')
>>> a.dtype
dtype('float32')

>>> a.nbytes
16
```

UNSIGNED INTEGER BYTE

```
>>> a = np.array([0, 1, 2, 3],
...               dtype='uint8')
>>> a.dtype
dtype('uint8')

>>> a.nbytes
4
```




```
Base 2            Base 10

00000000   ->   0 = 0*2**0 + 0*2**1 + ... + 0*2**7
00000001   ->   1 = 1*2**0 + 0*2**1 + ... + 0*2**7
00000010   ->   2 = 0*2**0 + 1*2**1 + ... + 0*2**7
...
11111111   ->   255 = 1*2**0 + 1*2**1 + ... + 1*2**7
```

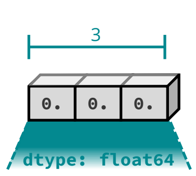

Array Creation Functions

ONES, ZEROS

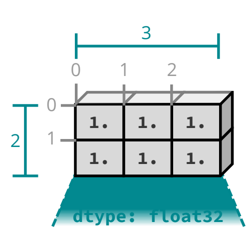
```
ones(shape, dtype='float64')
zeros(shape, dtype='float64')
```

shape is a number or sequence specifying the dimensions of the array. If **dtype** is not specified, it defaults to **float64**.

```
>>> np.ones((2, 3), dtype='float32')
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]],
       dtype=float32)

>>> np.zeros(3)
array([ 0., 0., 0.])
```



zeros(3) is equivalent to **zeros((3,))**



IDENTITY

```
# Generate an n by n identity array.
# The default dtype is float64.
>>> a = np.identity(4)
>>> a
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])

>>> a.dtype
dtype('float64')

>>> np.identity(4, dtype=int)
array([[ 1, 0, 0, 0],
       [ 0, 1, 0, 0],
       [ 0, 0, 1, 0],
       [ 0, 0, 0, 1]])
```

Generating Sequences

ARANGE

```
arange([start,] stop[, step],
                dtype=None)
```

Nearly identical to Python's **range()**. Creates an array of values in the range [start,stop) with the specified step value. Allows non-integer values for start, stop, and step. Default **dtype** is derived from the start, stop, and step values.

```
>>> np.arange(4)
array([0, 1, 2, 3])
>>> np.arange(0, 2*pi, pi/4)
array([ 0.000, 0.785, 1.571, 2.356, 3.142,
3.927, 4.712, 5.497])

# Be careful…
>>> np.arange(1.5, 2.1, 0.3)
array([ 1.5, 1.8, 2.1])
```

LINSPACE & LOGSPACE

```
# Generate N evenly spaced elements between
# (and including) start and stop values.
>>> np.linspace(0, 1, 5)
array([0., 0.25, 0.5, 0.75, 1.0])

# Generate N evenly spaced elements on a
# log scale between base**start and
# base**stop (default base=10)
>>> np.logspace(0, 1, 5)
array([1., 1.78, 3.16, 5.62, 10.])

>>> np.logspace(0, 1, 5, base=np.e)
array([1., 1.28, 1.65, 2.12, 2.72])
```


Generating Random Numbers (Old Style)

INTEGERS (OLD STYLE)

```
# Discrete uniform distribution
# over [0, 5) with N samples
>>> np.random.randint(5, size=3)
array([0, 3, 4])

# Discrete uniform distribution
# over [1, 5) with N samples
>>> np.random.randint(1, 5, 3)
array([2, 2, 3])
```

FLOATS (OLD STYLE)

```
# Uniform distribution over [0.0, 1.0)
# with samples
>>> np.random.rand(3)
array([0.567, 0.183, 0.586])

# Standard normal distribution with
# mean 0 and std 1 with N samples
>>> np.random.standard_normal(3)
array([-0.244, 1.542,  0.596])
```

Generating Random Numbers (New Style)

CREATE RANDOM NUMBER GENERATOR

```
# Create a default random generator
>>> rng = np.random.default_rng(seed=0)
```

INTEGERS (NEW STYLE)

```
# Discrete uniform distribution
# over [0, 5) with N samples
>>> rng.integers(5, size=3)
array([0, 0, 4])

# Discrete uniform distribution
# over [1, 5) with N samples
>>> rng.integers(1, 5, 3)
array([4, 3, 3])
```

FLOATS (NEW STYLE)

```
# Uniform distribution over [0.0, 1.0)
# with N samples
>>> rng.random(3)
array([0.078, 0.588, 0.621])

# Uniform distribution over
# [-1.0, 1.0) with N samples
>>> rng.uniform(-1, 1, 3)
array([-0.595, -0.764, 0.936])

# Standard normal distribution with
# mean 0 and std 1 with N samples
>>> rng.standard_normal(3)
array([-0.244, -0.713,  2.592])
```


Give it a try! Generating Array Data

`import numpy as np`

1. Generate 8 equally spaced points across the interval **`[-10.0, 10.0]`**. Call this **`x`**.
2. Scale this data by **`np.pi / 10`**
   Save this to **x**
3. Calculate **`y = np.cos(x)`**

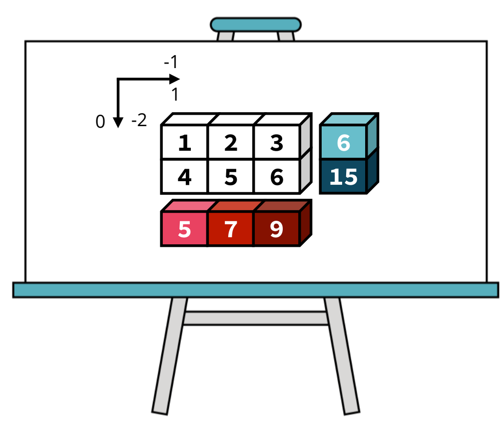
Arrays and Text Files

ARRAYS FROM/TO TXT FILES

```
# loadtxt() automatically generates an array
# from the txt file
arr = np.loadtxt('Data.txt',
...       skiprows=1,
...       dtype=int, delimiter=",",
...       usecols = (0,1,2,4),
...       comments = "%")

# Save an array into a txt file
np.savetxt('filename.txt', arr)
```

```
BEGINNING OF THE FILE
% Day,  Month,  Year, Skip, Avg Power
01, 01, 2000, x876, 13 % crazy day!
% we don't have Jan 03rd
04, 01, 2000, xfed, 55
```

Data.txt



NumPy

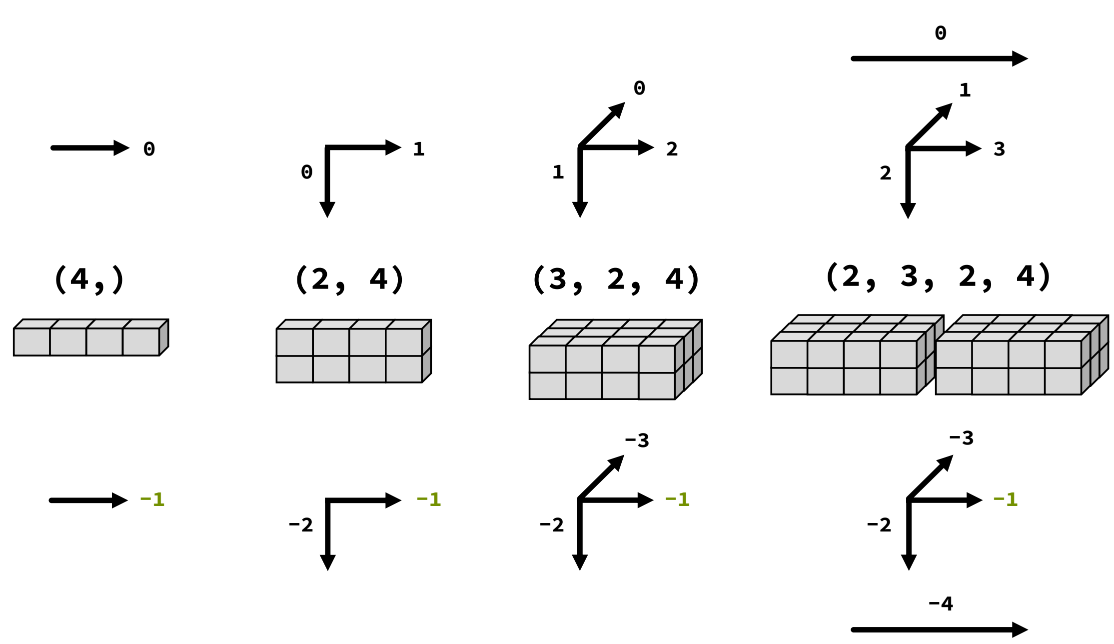
Array Calculation Methods

Computations with Arrays

Rule 1:	Operations between multiple array objects are first checked for proper shape match.
Rule 2:	Mathematical operators (`+ - * / exp`, `log`, …) apply element by element, on the values.
Rule 3:	Reduction operations (`mean`, `std`, `skew`, `kurt`, `sum`, `prod`, …) apply to the whole array, unless an axis is specified.
Rule 4:	Missing values propagate unless explicitly ignored (`nanmean`, `nansum`, …).

Multi-Dimensional Arrays

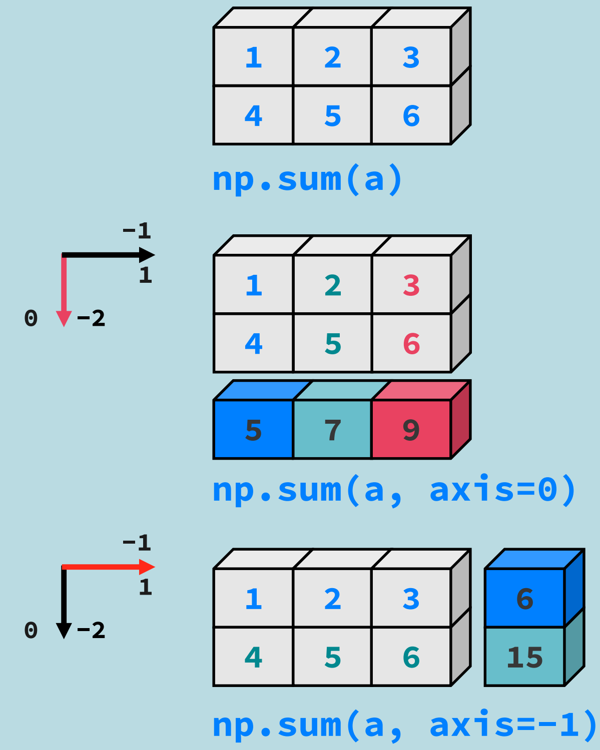
VISUALIZING MULTI-DIMENSIONAL ARRAYS



Array Calculation Methods

SUM METHODS

```
# Methods act on data stored in the array
>>> a = np.array([[1, 2, 3],
                  [4, 5, 6]])

# .sum() defaults to adding up all the
# values in an array.
>>> a.sum()
21

# supply the keyword axis to sum along the
# 0th axis
>>> a.sum(axis=0)
array([5, 7, 9])

# supply the keyword axis to sum along the
# last axis
>>> a.sum(axis=-1)
array([ 6, 15])
```



Other Operations on Arrays

PROD FUNCTION

```
# Functions work on data passed to it
>>> a = np.array([[1, 2, 3],
                  [4, 5, 6]])

# prod() defaults to calculating
# product of all values in an array.
>>> np.prod(a)
720

# supply an axis argument to
# get prod along a specific axis
>>> np.prod(a, axis=0)
array([4, 10, 18])
```

OTHER METHODS AND FUNCTIONS

Mathematical functions

- `sum, prod`
- `min, max, argmin, argmax`
- `ptp (max - min)`

Statistics

- `mean, std, var`

Truth value testing

- `any, all`

Min/Max

MIN

```
>>> a = np.array([[5, 4, 1], [2, 3, 6]])
# Prefer NumPy functions to builtins
# when working with arrays
>>> np.min(a)
1
# Most NumPy reducers can be used
# as methods as well as functions
>>> a.min()
1
```

MAX

```
# Use the axis keyword to find max
# values for one dimension
>>> a.max(axis=0)
array([5, 4, 6])
# as a function
>>> np.max(a, axis=1)
array([5, 6])
```

ARGMIN/ARGMAX

```
# Many tasks (like optimization) are
# interested in the location of a
# min/max, not the value
>>> a.argmax()
5

# arg methods return the location in a
# 1D, flattened version of the original
# array
>>> np.argmin(a)
2
```

UNRAVELING

```
# NumPy includes a function to
# un-flatten 1D locations
>>> np.unravel_index(
...     a.argmax(), a.shape)
(1, 2)
```


Statistics Array Methods

MEAN

```
>>> a = np.array([[1, 2, 3],
...               [4, 5, 6]])

# mean value of each column
>>> a.mean(axis=0)
array([ 2.5,  3.5,  4.5])

>>> np.mean(a, axis=0)
array([ 2.5,  3.5,  4.5])
```

STANDARD DEV./VARIANCE

```
# Standard Deviation
>>> a.std(axis=0)
array([ 1.5,  1.5,  1.5])

# For sample, set ddof=1
>>> a.std(axis=0, ddof=1)
array([ 2.12,  2.12,  2.12])


# variance
>>> a.var(axis=0)
array([2.25, 2.25, 2.25])

>>> np.var(a, axis=0)
array([2.25, 2.25, 2.25])
```

NumPy Where

COORDINATE LOCATIONS

```
# NumPy's where function has two
# distinct uses. One is to provide
# coordinates from masks.
>>> a = np.arange(-2, 2) ** 2
>>> a
array([4, 1, 0, 1])

>>> mask = a % 2 == 0
>>> mask
array([ True, False,  True, False])

# Coordinates are returned as a tuple
# of arrays, one for each axis.
>>> np.where(mask)
(array([0, 2]),)
```

CONDITIONAL ARRAY CREATION

```
# Where can also be used to construct
# a new array by choosing values from
# other arrays of the same shape.
>>> positives = np.arange(1, 5)
>>> negatives = -positives
>>> np.where(mask, positives,
...          negatives)
array([ 1, -2,  3, -4])

# Or from scalar values. This can be
# useful for recoding arrays.
>>> np.where(mask, 1, 0)
array([1, 0, 1, 0])

# Or from both.
>>> np.where(mask, positives, 0)
array([1, 0, 3, 0])
```


Give it a try! Array Calculation Methods

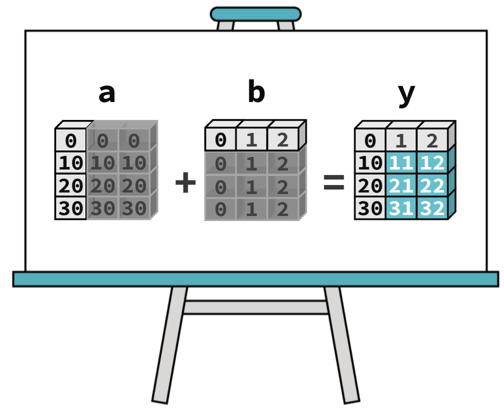
1. Create the array below with

```
a = np.arange(-15, 15).reshape(5, 6) ** 2
```

2. and compute:

- The maximum of each row
- The mean of each column
- The position of the overall minimum

	0	1	2	3	4	5
0	225	196	169	144	121	100
1	81	64	49	36	25	16
2	9	4	1	0	1	4
3	9	16	25	36	49	64
4	81	100	121	144	169	196



NumPy

Array Broadcasting

Array Broadcasting

NumPy arrays of different dimensionality can be combined in the same expression. Arrays with smaller dimension are **broadcasted** to match the larger arrays, *without copying data*.

Broadcasting has **two rules**.

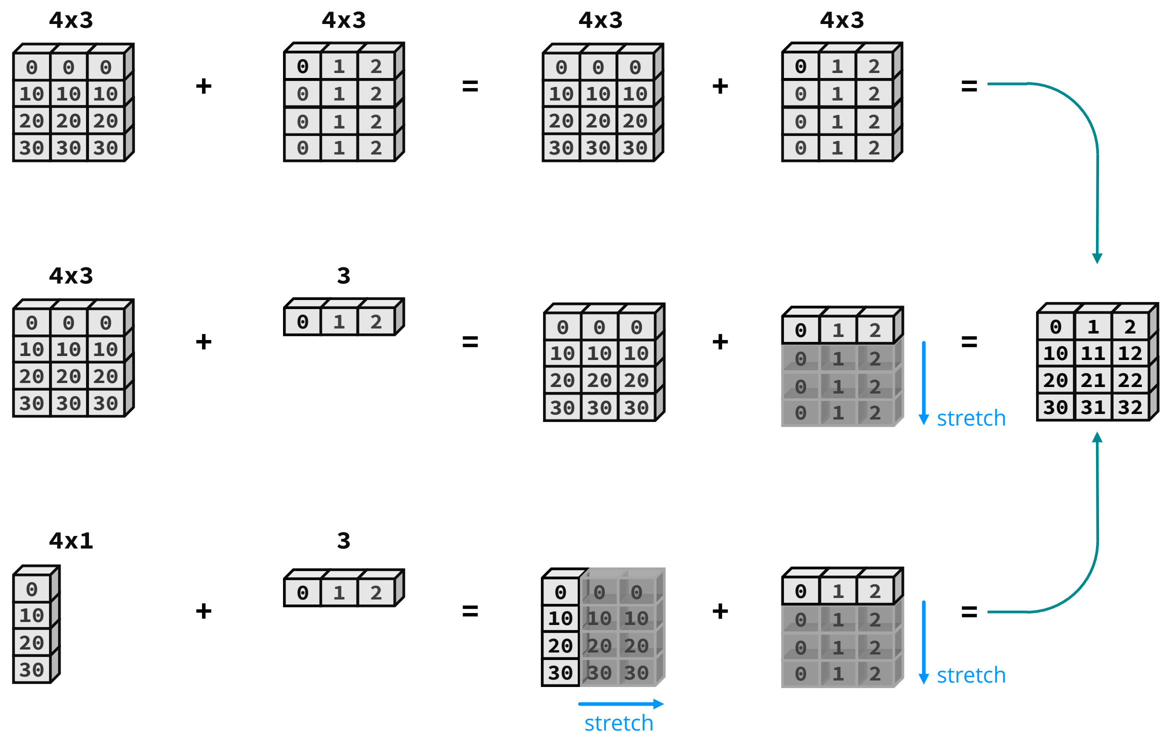
RULE 1: PREPEND ONES TO SMALLER ARRAY'S SHAPE

```
>>> import numpy as np
>>> a = np.ones((3, 5)) # a.shape == (3, 5)
>>> b = np.ones((5,)) # b.shape == (5,)
>>> b.reshape(1, 5) # result is a (1,5)-shaped array
>>> b[np.newaxis, :] # equivalent, more concise
```

RULE 2: DIMENSIONS OF SIZE 1 ARE REPEATED WITHOUT COPYING

```
>>> c = a + b # c.shape == (3, 5)
# is logically equivalent to...
>>> tmp_b = b.reshape(1, 5)
>>> tmp_b_repeat = tmp_b.repeat(3, axis=0)
>>> c = a + tmp_b_repeat
# But broadcasting makes no copies of "b"s data!
```

Array Broadcasting



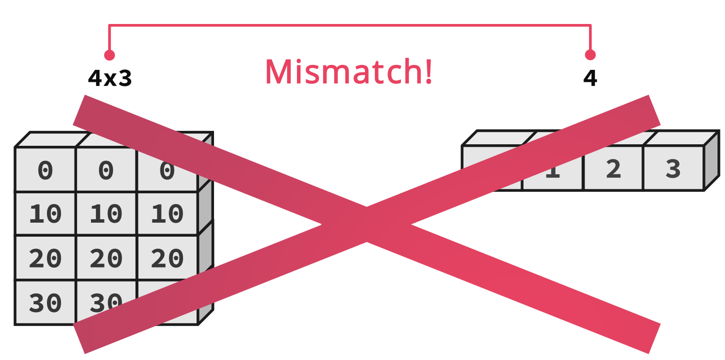
Broadcasting Rules

The **trailing axes** of either arrays **must be 1 or both must have the same size** for broadcasting to occur.

Otherwise, a

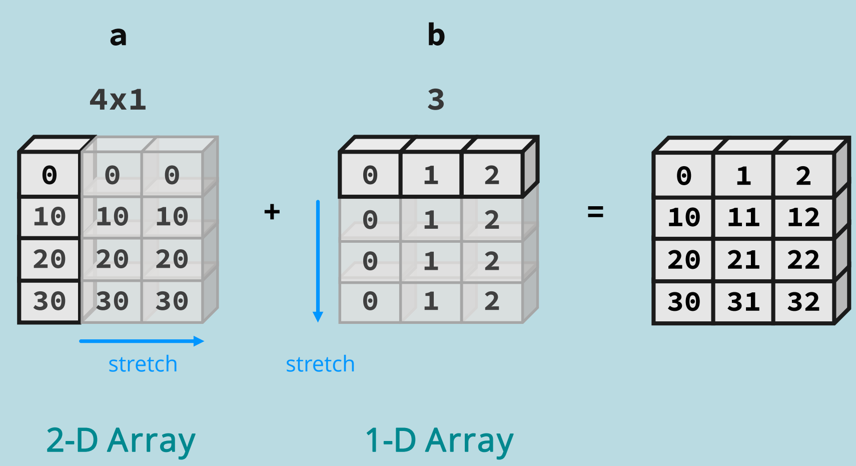
`"ValueError: shape mismatch: objects cannot be broadcast to a single shape"`

exception is thrown.

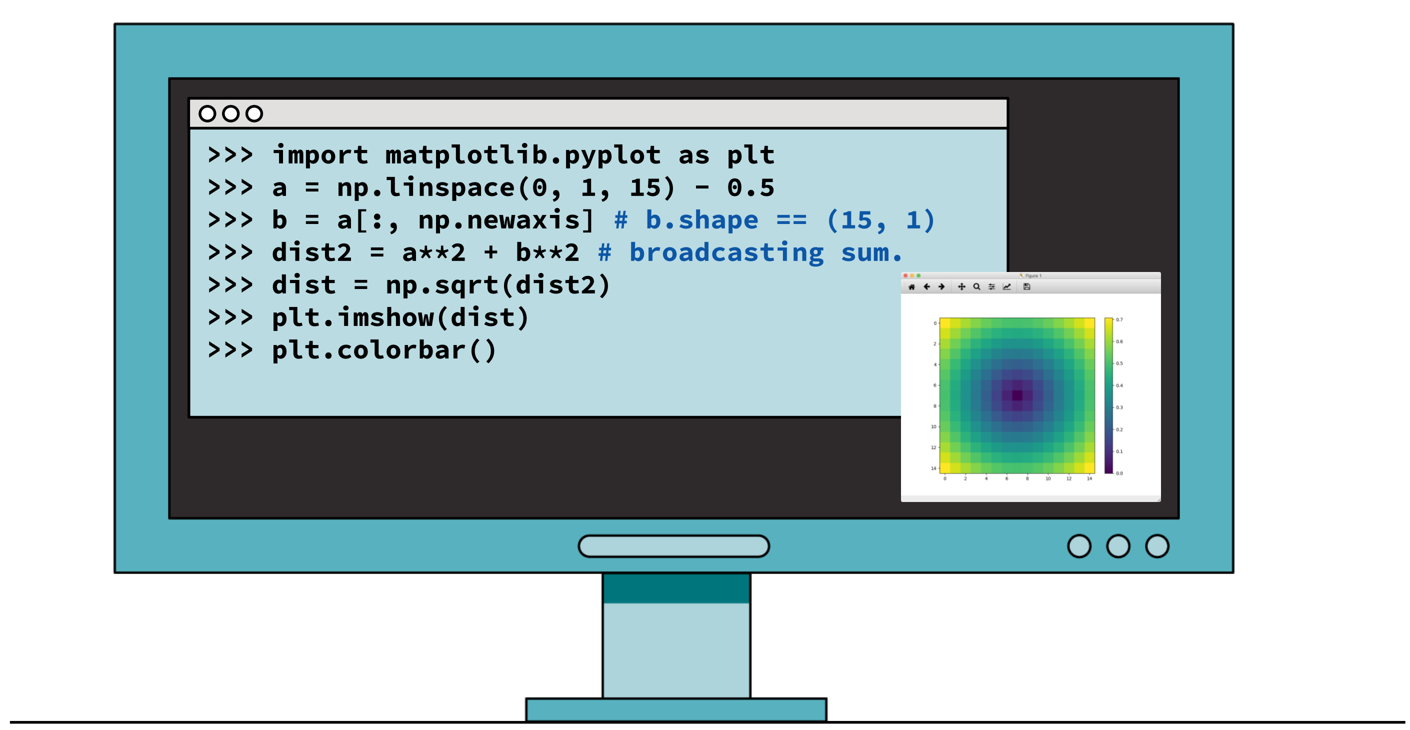


Broadcasting in Action

```
>>> a = array([0, 10, 20, 30])
>>> b = array([0, 1, 2])
>>> y = a[:, newaxis] + b
```



Application: Distance from Center



Broadcasting's Usefulness

Broadcasting can often be used to replace needless data replication inside a NumPy array expression.

`np.meshgrid()` – use **`newaxis`** appropriately in broadcasting expressions.

`np.repeat()` – broadcasting makes repeating an array along a dimension of size 1 unnecessary.

MESHGRID: COPIES DATA

```
>>> x, y = \
...     np.meshgrid([1,2],
...                 [3,4,5])
>>> z = x + y
```

BROADCASTING: NO COPIES

```
>>> x = np.array([1, 2])
>>> y = np.array([3, 4, 5])
>>> z = x[np.newaxis, :] \
...     + y[:, np.newaxis]
```

Enthought

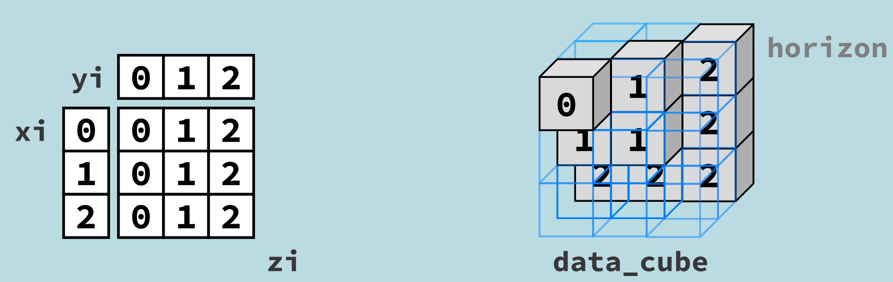
Broadcasting Indices

Broadcasting can also be used to **slice elements from different "depths"** in a 3-D (or any other shape) array. This is a very powerful feature of indexing.

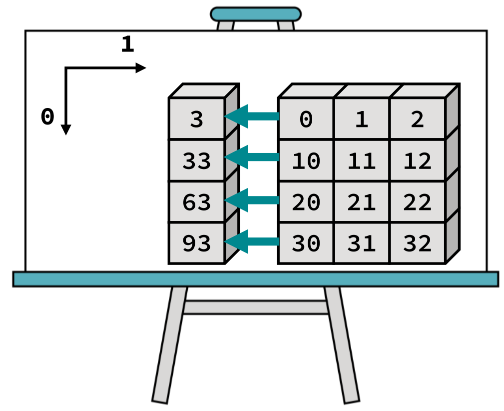
```
>>> data_cube = np.arange(27).reshape(3, 3, 3)
>>> yi, xi = np.meshgrid(np.arange(3), np.arange(3),
...                      sparse=True)
>>> zi = np.array([[0, 1, 2],
...                [1, 1, 2],
...                [2, 2, 2]])
>>> horizon = data_cube[xi, yi, zi]
```

Indices

Selected Data



Enthought



NumPy

Universal Function Methods

Universal Function Methods

The mathematical, comparative, logical, and bitwise operators *op* that take two arguments (binary operators) have special methods that operate on arrays:

```
>>> op.reduce(a,axis=0)
>>> op.accumulate(a,axis=0)
>>> op.outer(a,b)
>>> op.reduceat(a,indices)
```

op.reduce()

`op.reduce(a)` applies **`op`** to all the elements in a 1-D array **`a`** reducing it to a single value.

For example:

`y = add.reduce(a)`

$$= \sum_{n=0}^{N-1} a[n]$$

$$= a[0] + a[1] + ... + a[N-1]$$

ADD EXAMPLE

```
>>> a = np.array([1,2,3,4])
>>> np.add.reduce(a)
10
```

STRING LIST EXAMPLE

```
>>> a = np.array(['ab','cd','ef'],
...              dtype='object')
>>> np.add.reduce(a)
'abcdef'
```

LOGICAL OP EXAMPLES

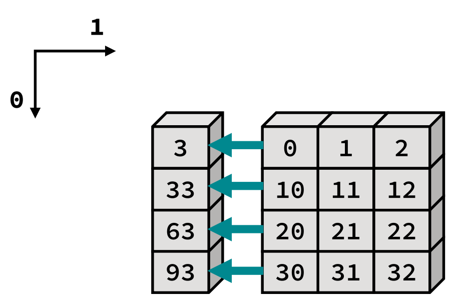
```
>>> a = np.array([1,1,0,1])
>>> np.logical_and.reduce(a)
False
>>> np.logical_or.reduce(a)
True
```


op.reduce()

For multidimensional arrays, **`op.reduce(a,axis)`** applies **`op`** to the elements of **`a`** along the specified **`axis`**. The resulting array has dimensionality one less than **`a`**. The default value for **`axis`** is 0.

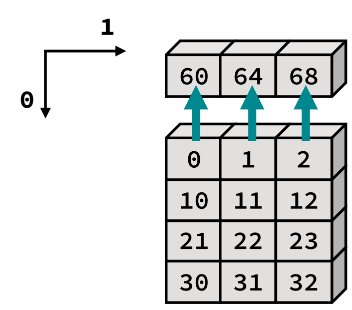
SUMMING UP EACH ROW

```
>>> a = np.arange(3) + np.arange(0, 40,
...                  10).reshape(-1, 1)
>>> np.add.reduce(a,1)
array([ 3, 33, 63, 93])
```



SUM COLUMNS BY DEFAULT

```
>>> np.add.reduce(a)
array([60, 64, 68])
```



op.accumulate()

`op.accumulate(a)` creates a new array containing the intermediate results of the **`reduce`** operation at each element in **a**.

For example:

`y = add.accumulate(a)`

$$= \left[\sum_{n=0}^{0} a[n], \sum_{n=0}^{1} a[n], \ldots, \sum_{n=0}^{N-1} a[n] \right]$$

ADD EXAMPLE

```
>>> a = np.array([1,2,3,4])
>>> np.add.accumulate(a)
array([ 1,  3,  6, 10])
```

STRING LIST EXAMPLE

```
>>> a = np.array(['ab','cd','ef'],
...                  dtype='object')
>>> np.add.accumulate(a)
array([ab,abcd,abcdef],
      dtype=object)
```

LOGICAL OP EXAMPLES

```
>>> a = np.array([1,1,0])
>>> np.logical_and.accumulate(a)
array([True, True, False])
>>> np.logical_or.accumulate(a)
array([True, True, True])
```


op.reduceat()

`op.reduceat(a,indices)` applies **op** to ranges in the 1-D array `a` defined by the values in `indices`. The resulting array has the same length as `indices`.

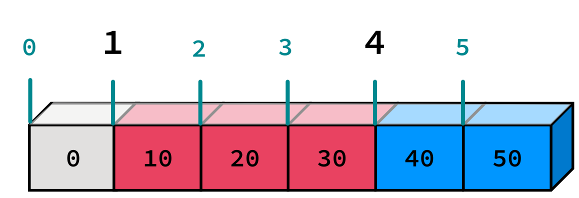
For example:

`y = add.reduceat(a, indices)`

$$\texttt{y[i]} = \sum_{n=indices[i]}^{indices[i+1]} a[n]$$

EXAMPLE

```
>>> a = np.array([0,10,20,30,40,50])
>>> indices = np.array([1,4])
>>> np.add.reduceat(a,indices)
array([60, 90])
```



For multidimensional arrays, **`reduceat()`** is always applied along the last axis (sum of rows for 2-D arrays). This is different from the default for **`reduce()`** and **`accumulate()`**.

`op.outer()`

`op.outer(a,b)` forms all possible combinations of elements between **a** and **b** using **op**. The shape of the resulting array results from concatenating the shapes of **a** and **b**. (Order matters.)

a

a[0]	a[1]	a[2]	a[3]

b

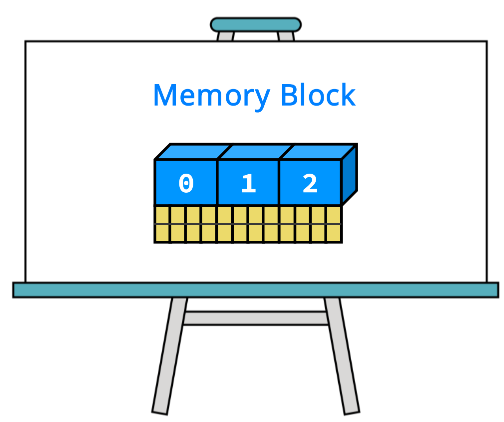
b[0]	b[1]	b[2]

```
>>> np.add.outer(a,b)
```

a[0]+b[0]	a[0]+b[1]	a[0]+b[2]
a[1]+b[0]	a[1]+b[1]	a[1]+b[2]
a[2]+b[0]	a[2]+b[1]	a[2]+b[2]
a30]+b[0]	a[3]+b[1]	a[3]+b[2]

```
>>> np.add.outer(b,a)
```

b[0]+a[0]	b[0]+a[1]	b[0]+a[2]	b[0]+a[3]
b[1]+a[0]	b[1]+a[1]	b[1]+a[2]	b[1]+a[3]
b[2]+a[0]	b[2]+a[1]	b[2]+a[2]	b[2]+a[3]

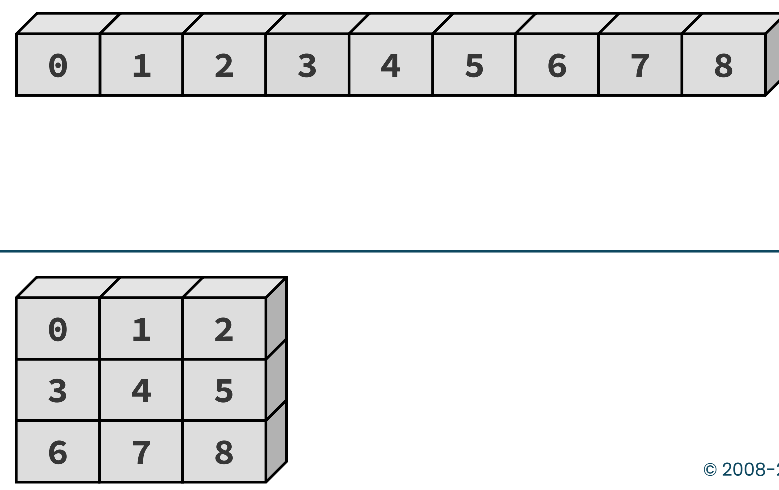


NumPy

The Array Data Structure

Array Data Structure

Memory Block



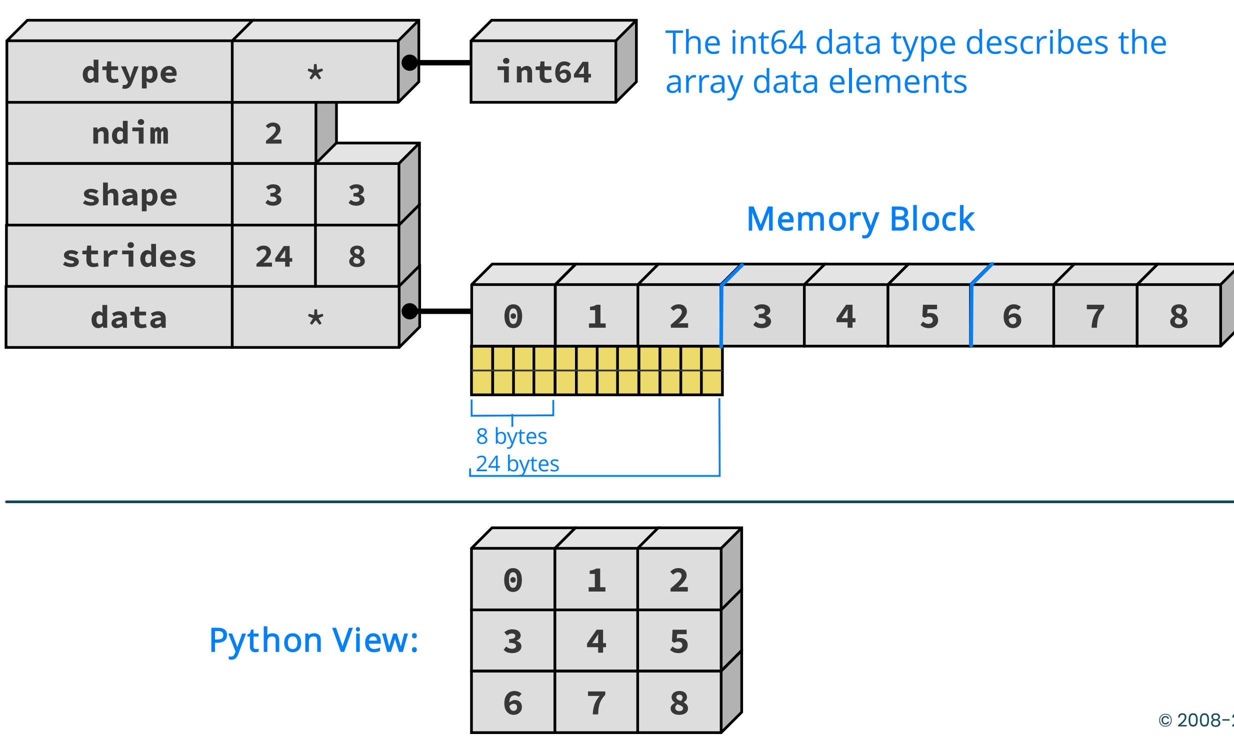
Python View:

Array Data Structure

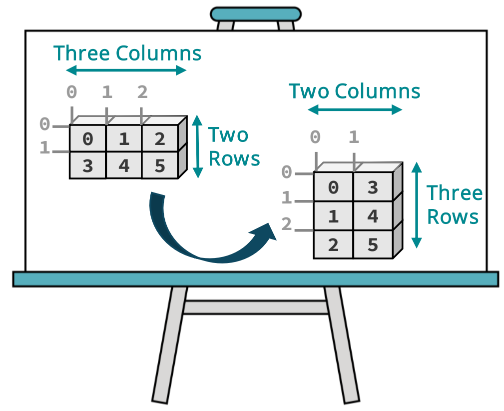
NDArray Data Structure

The int64 data type describes the array data elements

Memory Block



Python View:



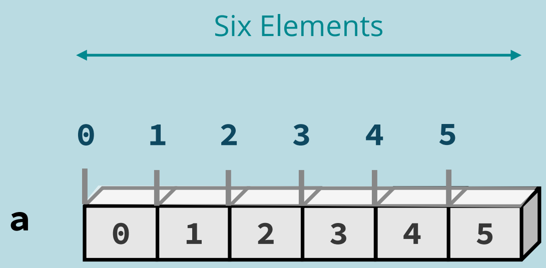
NumPy

Structure Operations

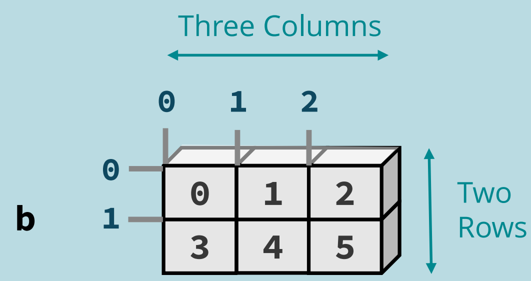
Operations on the Array Structure

Operations that only affect the array structure, not the data, can usually be executed **without copying memory**.

```
>>> a = np.arange(6)
>>> a
```




```
>>> b = a.reshape(2, 3)
>>> b
```

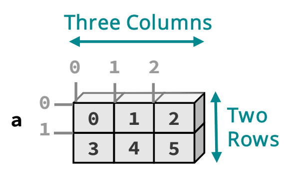


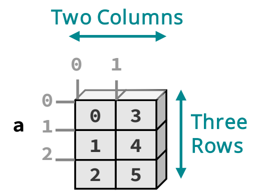
This is not a new copy of the data.
The original data does not get reordered.

Transpose

TRANSPOSE

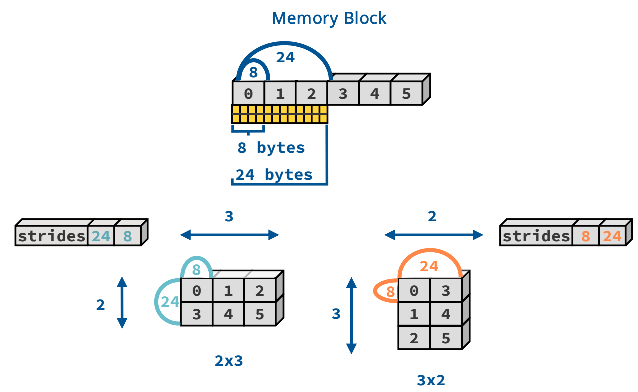
```
>>> a = np.array([[0, 1, 2],
...               [3, 4, 5]])
>>> a.shape
(2,3)

# Transpose swaps the order of axes.
>>> a.T
array([[0, 3],
       [1, 4],
       [2, 5]])

>>> a.T.shape
(3,2)
```





TRANSPOSE RETURNS VIEWS

```
# Transpose does not move values
# around in memory. It only changes
# the order of "strides" in the array
>>> a.strides
(24, 8)
>>> a.T.strides
(8, 24)
```



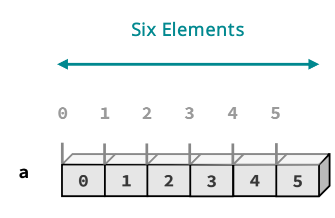
Reshaping Arrays

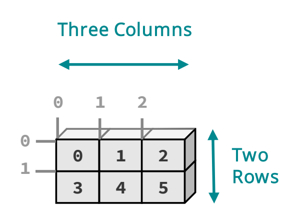
RESHAPE

```
>>> a = np.array([[0,1,2],
...               [3,4,5]])

# Return a new array with a different
# shape (a view where possible)
>>> a.reshape(3,2)
array([[0, 1],
       [2, 3],
       [4, 5]])

# Reshape cannot change the number of
# elements in an array
>>> a.reshape(4,2)
ValueError: total size of new array
must be unchanged
```

SHAPE

```
>>> a = np.arange(6)
>>> a
array([0, 1, 2, 3, 4, 5])
>>> a.shape
(6,)

# Reshape array in-place to 2x3
>>> a.shape = (2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
```



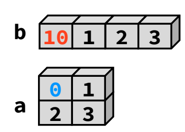


Flattening Arrays

FLATTEN (SAFE)

`a.flatten()` converts a multi-dimensional array into a 1-D array. The new array is a **copy** of the original data.

```
# Create a 2D array
>>> a = np.array([[0,1],
...               [2,3]])

# Flatten out elements to 1D
>>> b = a.flatten()
>>> b
array([0,1,2,3])

# Changing b does not change a
>>> b[0] = 10
>>> b
array([10,1,2,3])
>>> a                no change
array([[0, 1],
       [2, 3]])
```

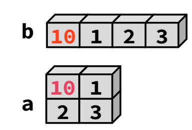


RAVEL (EFFICIENT)

`a.ravel()` is the same as **`a.flatten()`**, but returns **a reference (or view)** of the array if possible (i.e., the memory is contiguous).

Otherwise the new array copies the data. **`np.ravel()`** can be applied to non-array objects.

```
# Flatten out elements to 1-D
>>> b = a.ravel()
>>> b
array([0,1,2,3])

# Changing b does change a
>>> b[0] = 10
>>> b
array([10,1,2,3])
>>> a                changed!
array([[10, 1],
       [ 2, 3]])
```



Stay in touch!

Enthought

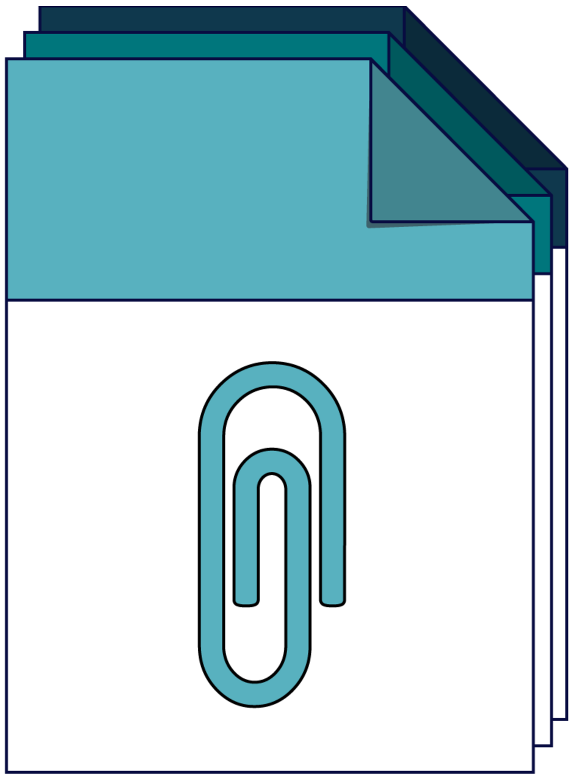
Enthought Media

@enthought

SciPy

EuroSciPy

Please complete the online survey!
(link on course web page)



Appendix

Additional Material

Enthought

Enthought

Enthought powers digital transformation for science.

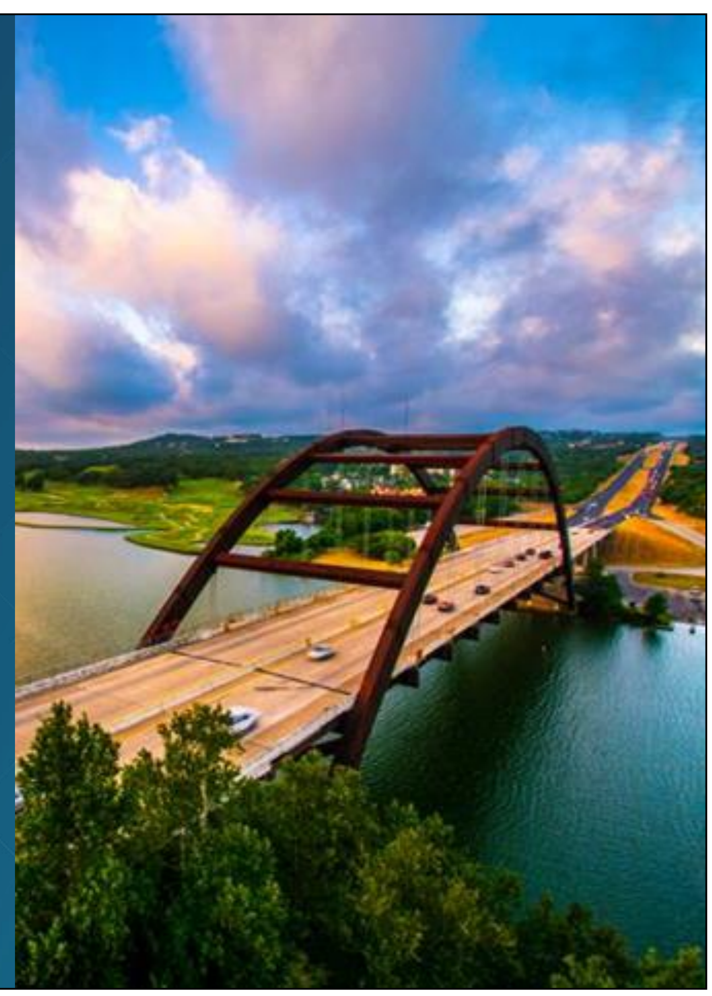
We Partner with Industry Leaders

TotalEnergies
TEL TOKYO ELECTRON
Shell
Roche
TESLA
Schlumberger
ExxonMobil
devon
Microsoft
Anadarko
BARCLAYS
NXP
Dow
aramco
ThermoFisher SCIENTIFIC
idemitsu
DE Shaw & Co
JSR
JOHN DEERE
Daiichi-Sankyo
Pfizer
SAMSUNG
gsk GlaxoSmithKline
MAERSK
SIEMENS
P&G
LLOYDS BANK
ConocoPhillips
J.P.Morgan
KBI BIOPHARMA
BROADCOM
Qualcomm
AIG

ABOUT ENTHOUGHT

Foundation for Success

- Global, privately-owned company with headquarters in Austin
- Offices in Houston, Texas; Cambridge, United Kingdom; Zürich, Switzerland; and Tokyo, Japan
- Deep roots in the Python community of scientists and engineers
- 80% of employees have graduate degrees or PhDs in science and engineering



About Enthought

Partner with Enthought

Science is at the core of your business. And ours.
Designed by scientists, for scientists

The Enthought Approach is proven.
Catalyzing innovation by transforming technology, people, and processes

We're at the intersection of science and technology.
Deep scientific expertise and computational knowledge

We empower scientists.
Equipping scientists with digital skills and technology

Enthought