CS65K Robotics

Modelling, Planning and Control

Appendix 1

LINEAR ALGEBRA

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objective

- Scalar, Array, Vector, Matrix, and Tensor
- Since modelling and control of robot manipulators requires an extensive use of matrices and vectors as well as of matrix and vector operations, the goal of this appendix is to provide a brush-up of linear algebra.

Plotly

- Plotly's Python graphing library makes interactive, publication-quality graphs. Examples of how to make line plots, scatter plots, area charts, bar charts, error bars, box plots, histograms, heatmaps, subplots, multiple-axes, polar charts, and bubble charts.
- Installation: **pip install plotly==4.8.1**
  pip install chart-studio
- Web-site: https://plotly.com/python/

Plotly

SECTION 1

Imports

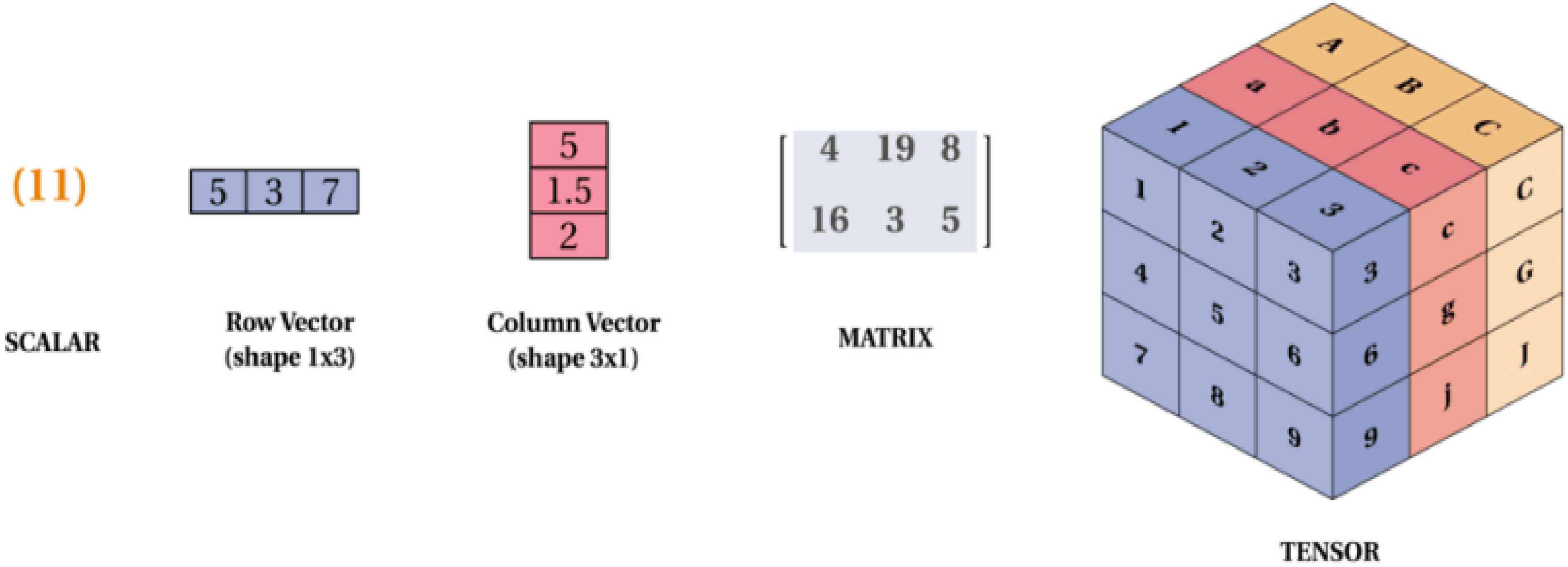
The tutorial below imports NumPy, Pandas, and SciPy.

```
import chart-studio.plotly as py
import plotly.graph_objs as go
from plotly import figure_factory as FF
import numpy as np
import pandas as pd
import scipy
```

Array

SECTION 2

Scalar	Vector	Matrix	Tensor
1	$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$	$\begin{bmatrix} \begin{bmatrix} 1 & 2 \end{bmatrix} & \begin{bmatrix} 3 & 2 \end{bmatrix} \\ \begin{bmatrix} 1 & 7 \end{bmatrix} & \begin{bmatrix} 5 & 4 \end{bmatrix} \end{bmatrix}$



Numpy Array

We'll cover a few categories of basic array manipulations here:

- **Attributes of arrays**: Determining the size, shape, memory consumption, and data types of arrays
- **Indexing of arrays**: Getting and setting the value of individual array elements
- **Slicing of arrays**: Getting and setting smaller subarrays within a larger array
- **Reshaping of arrays**: Changing the shape of a given array
- **Joining and splitting of arrays**: Combining multiple arrays into one, and splitting one array into many

Arrays

SECTION 1 DATA REPRESENTATION

Data Representation

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

Attriubutes

- **dim:** dimension
- **shape:** sizes for each dimension (width, length, height, plane)
- **Size:** net number of elements
- **Dtype:** data type
- **itemsize:** total number of items
- **nbytes:** total number of bytes

Numpy Array Attributes

```
>>> import numpy as np
>>> a = np.arange(6)        # NumPy arange returns an array object
>>> a
array([0, 1, 2, 3, 4, 5])
>>> a = a.reshape(2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> a.shape
(2, 3)                      # note: this returns a tuple
>>> a.ndim
2
>>> a.size
6
```

Array Creation

Demo Program: array1_Creation1.py

```
import numpy as np
a = np.arange(6)
print("1-D: ", a)
a.reshape(2, 3)
print("2-D: ", a)
print("Shape: ", a.shape)
print("Dimension: ", a.ndim)
print("Size: ", a.size)
```

```
1-D:  [0 1 2 3 4 5]
2-D:  [0 1 2 3 4 5]
Shape:  (6,)
Dimension:  1
Size:  6
```

Creating Arrays

Command	NumPy Array
`np.array([1,2,3])`	1 2 3

Creating Arrays

`np.ones(3)` →

1
1
1

`np.zeros(3)` →

0
0
0

`np.random.random(3)` →

0.5967
0.0606
0.2223

Example

In [1]:
```
import numpy as np
np.random.seed(0)  # seed for reproducibility

x1 = np.random.randint(10, size=6)  # One-dimensional array
x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array
```

Each array has attributes `ndim` (the number of dimensions), `shape` (the size of each dimension), and `size` (the total size of the array):

Example

```
In [2]: print("x3 ndim: ", x3.ndim)
        print("x3 shape:", x3.shape)
        print("x3 size: ", x3.size)

x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

```
In [3]: print("dtype:", x3.dtype)

dtype: int64
```

```
In [4]: print("itemsize:", x3.itemsize, "bytes")
        print("nbytes:", x3.nbytes, "bytes")

itemsize: 8 bytes
nbytes: 480 bytes
```

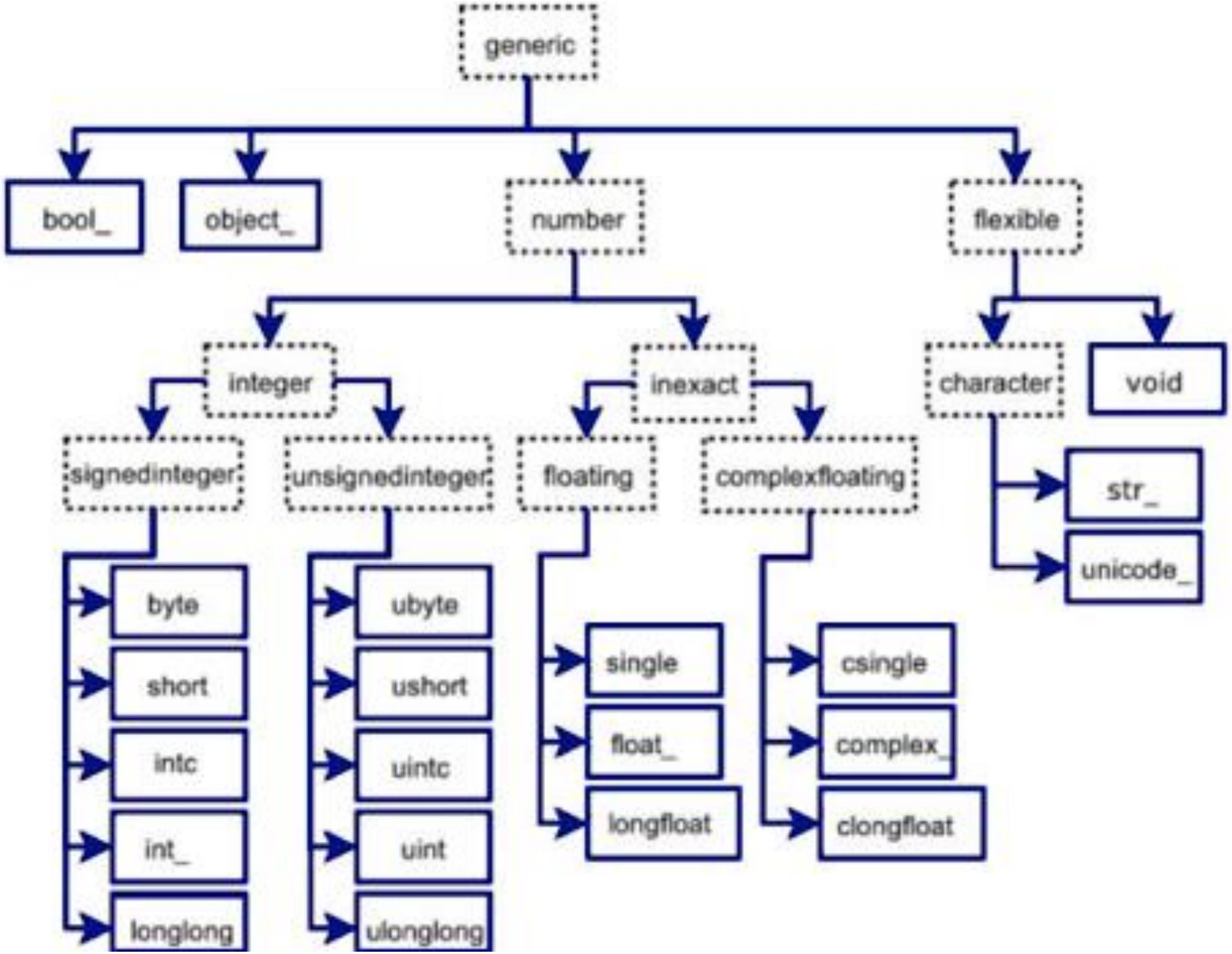
Array Creation

Demo Program: array1_Creation2.py

```
import numpy as np
np.random.seed(0)
x1 = np.random.randint(10, size=6)
x2 = np.random.randint(10, size=(3, 4))
x3 = np.random.randint(10, size=(3, 4, 5))
print("x3 ndim:", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size:", x3.size)
print("x3 dtype:", x3.dtype)
print("x3 itemsize:", x3.itemsize)
print("x3 nbytes:", x3.nbytes)
```

```
x3 ndim: 3
x3 shape: (3, 4, 5)
x3 size: 60
x3 dtype: int32
x3 itemsize: 4
x3 nbytes: 240
```

Name	Description	Syntax
empty()	Return a new array of given shape and type, without initializing entries.	empty(shape[, dtype, order])
empty_like	Return a new array with the same shape and type as a given array.	empty_like(a[, dtype, order, subok])
eye()	Return a 2-D array with ones on the diagonal and zeros elsewhere.	eye(N[, M, k, dtype])
identity()	Return the identity array.	identity(n[, dtype])
ones()	Return a new array of given shape and type, filled with ones.	ones(shape[, dtype, order])
ones_like	Return an array of ones with the same shape and type as a given array.	ones_like(a[, dtype, order, subok])
zeros	Return a new array of given shape and type, filled with zeros.	zeros(shape[, dtype, order])
zeros_like	Return an array of zeros with the same shape and type as a given array.	zeros_like(a[, dtype, order, subok])
full()	Return a new array of given shape and type, filled with fill_value.	full(shape, fill_value[, dtype, order])
full_like()	Return a full array with the same shape and type as a given array.	full_like(a, fill_value[, dtype, order, subok])


generic
bool_
object_
number
flexible
integer
inexact
character
void
signedinteger
unsignedinteger
floating
complexfloating
str_
unicode_
byte
short
intc
int_
longlong
ubyte
ushort
uintc
uint
ulonglong
single
float_
longfloat
csingle
complex_
clongfloat

Arrays

SECTION 2 ARITHMETIC

Array Arithmetic

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

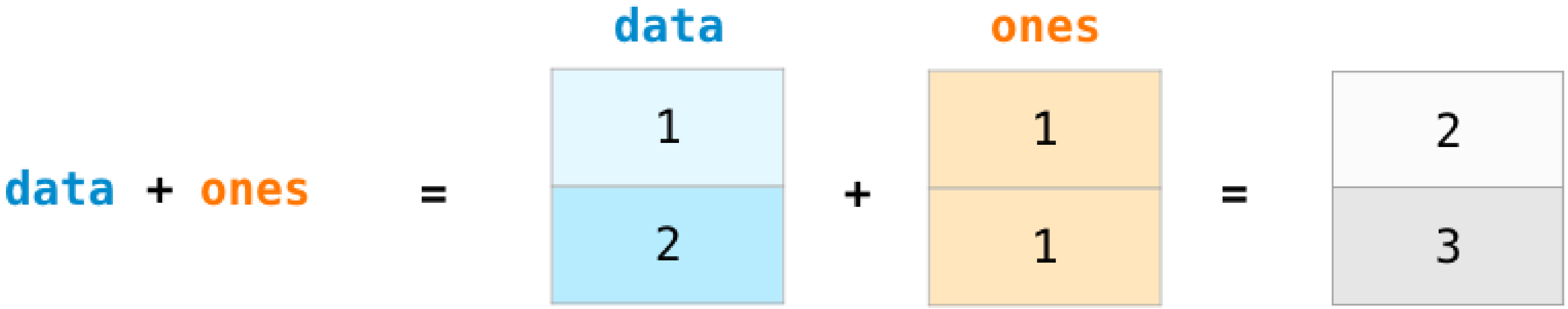
data
1
2

```
ones = np.ones(2)
```

ones
1
1

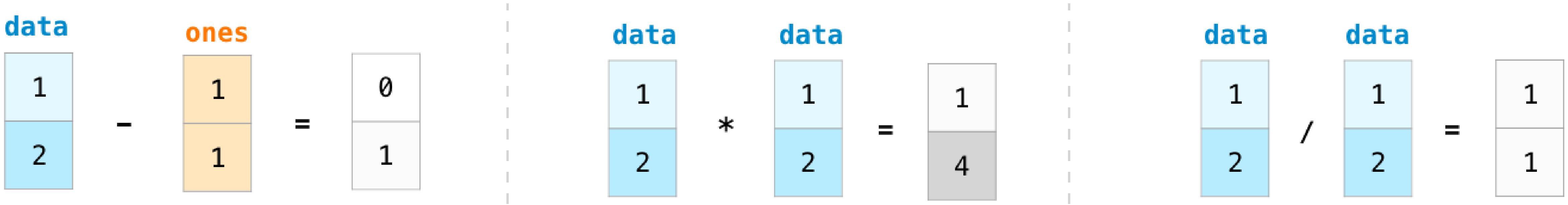
Array Arithmetic

Vector Addition (Parallel Processing)



Array Arithmetic

Vector Arithmetic (Parallel Processing)



Array Arithmetic

Scalar-Vector Arithmetic (Parallel Processing)

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} * 1.6 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} * \begin{bmatrix} 1.6 \\ 1.6 \end{bmatrix} = \begin{bmatrix} 1.6 \\ 3.2 \end{bmatrix}$$

Array Creation

Demo Program: array2_arithmetic.py

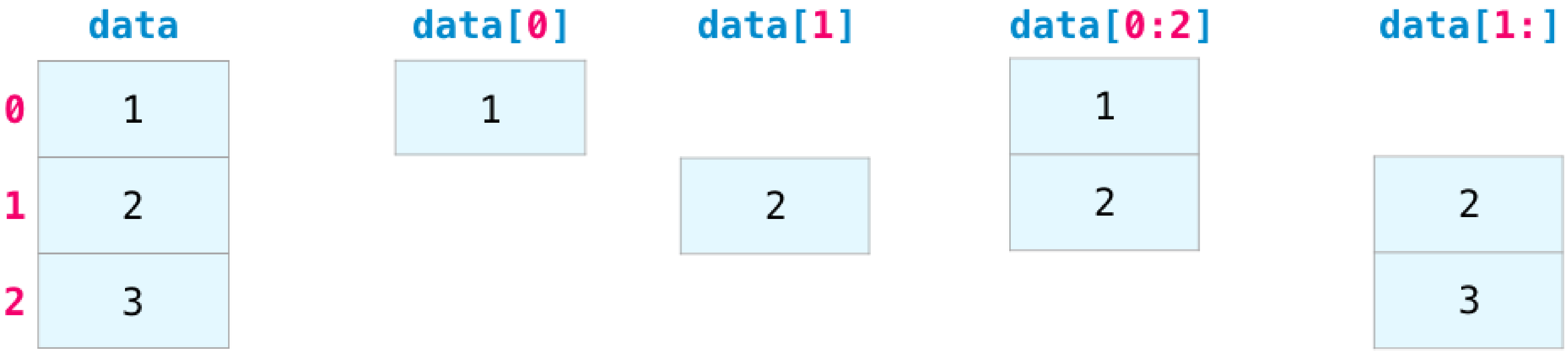
```
import numpy as np
data = np.array([1, 2])
ones = np.ones(2)
identity = np.eye(2)
zeros = np.zeros(2)
half = ones * 0.5
print("data: ", data)
print("ones: ", ones)
print("I: ", identity)
print("zeros: ", zeros)
print("Add: ", (data + ones))
print("Sub: ", (data - ones))
print("Mul: ", (data * half))
print("Div: ", (data + half))
print("Scalar-Vector: ", (0.6*data))
```

```
data:  [1 2]
ones:  [1. 1.]
I:  [[1. 0.]
     [0. 1.]]
zeros:  [0. 0.]
Add:  [2. 3.]
Sub:  [0. 1.]
Mul:  [0.5 1. ]
Div:  [1.5 2.5]
Scalar-Vector:  [0.6 1.2]
```

Arrays

SECTION 3 INDEXING AND SLICING

Indexing



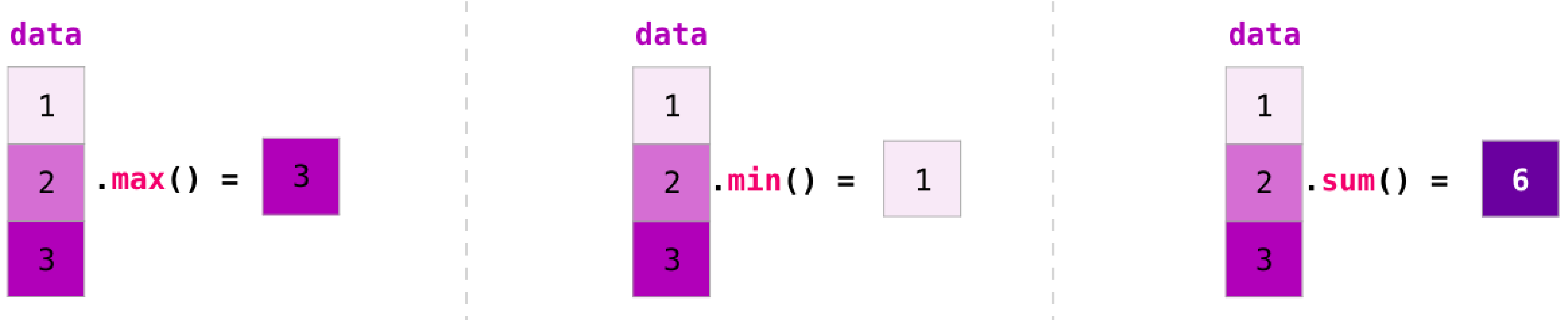
Array Creation

Demo Program: array3_index.py

```
import numpy as np
data = np.arange(0, 11)   # 0-10
print("data      : ", data)
print("data[1]  : ", data[1])
print("data[-3]: ", data[-3])
print("data[2:5]  : ", data[2:5])
print("data[-1:-9:-2]  : ", data[-1:-9:-2])
print("data[-1:0]  : ", data[-1:0])
print("data[-1:0:-1]  : ", data[-1:0:-1])
print("data[3:-2]  : ", data[3:-2])
print("data[:-2]  : ", data[:-2])
print("data[2:]  : ", data[2:])
```

```
data   : [ 0  1  2  3  4  5  6  7  8  9 10]
data[1] :  1
data[-3]:  8
data[2:5] :  [2 3 4]
data[-1:-9:-2] :  [10  8  6  4]
data[-1:0] :  []
data[-1:0:-1] :  [10  9  8  7  6  5  4  3  2  1]
data[3:-2] :  [3 4 5 6 7 8]
data[:-2] :  [0 1 2 3 4 5 6 7 8]
data[2:] :  [ 2  3  4  5  6  7  8  9 10]
```

Aggregates



Array Creation

Demo Program: array3_aggregates.py

```python
import numpy as np
data = np.arange(0, 11)   # 0-10
print("data       : ", data)
print("data.sum(): ", data.sum())
print("data.max(): ", data.max())
print("data.min(): ", data.min())
print("np.average(a): ", np.average(data))
print("np.mean(a): ", np.mean(data))
print("np.median(a): ", np.median(data))
print("np.std(a): ", np.std(data))
print("np.var(a): ", np.var(data))
```

```
data    : [ 0  1  2  3  4  5  6  7  8  9 10]
data.sum():  55
data.max():  10
data.min():  0
np.average(a):  5.0
np.mean(a):  5.0
np.median(a):  5.0
np.std(a):  3.1622776601683795
np.var(a):  10.0
```

Arrays

SECTION 4 2D ARRAY AS MATRIX/TABLE

Creating Matrices

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

1	2
3	4

2D Array Creation

`np.ones((3,2))` → 3 × 2

1	1
1	1
1	1

`np.zeros((3,2))` →

0	0
0	0
0	0

`np.random.random((3,2))` →

0.37	0.88
0.75	0.79
0.63	0.16

Matrix Creation

Demo Program: array4_2D_create.py

```
import numpy as np
m1 = np.array([[1, 2], [3, 4]])
print("m1: ", m1)
m2 = np.array(np.matrix("5 6; 7 8"))
print("m2: ", m2)
m3 = np.zeros((2, 2))
print("m3: ", m3)
m4 = np.ones((2, 2))
print("m4: ", m4)
m5 = np.eye(2)
print("m5: ", m5)
m6 = np.random.randint(10, size=(2, 2))
print("m6: ", m6)
m7 = np.random.random((2, 2))
print("m7: ", m7)
```

```
m1:  [[1 2]
      [3 4]]
m2:  [[5 6]
      [7 8]]
m3:  [[0. 0.]
      [0. 0.]]
m4:  [[1. 1.]
      [1. 1.]]
m5:  [[1. 0.]
      [0. 1.]]
m6:  [[7 9]
      [5 2]]
m7:  [[0.92441049 0.79840959]
      [0.057992   0.98685019]]
```

Matrix Arithmetic

Matrix Addition (Parallel Processing)

$$\texttt{data} + \texttt{ones} = \begin{array}{c} \texttt{data} \\ \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \end{array} + \begin{array}{c} \texttt{ones} \\ \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \end{array} = \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$$

Matrix Arithmetic

Scalar-Vector-Matrix Addition (Parallel Processing)

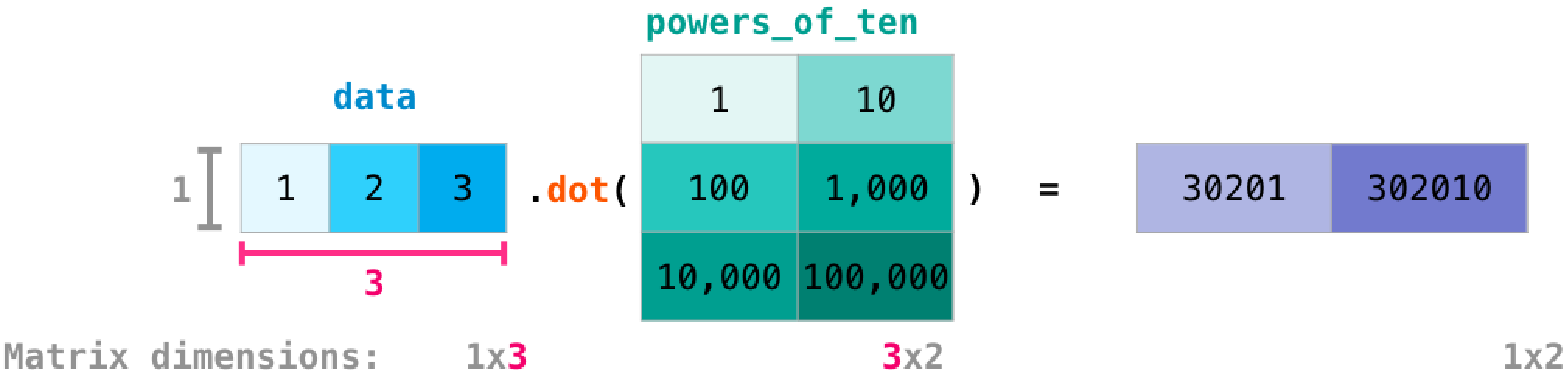
$$\texttt{data} + \texttt{ones\_row} = \overset{\texttt{data}}{\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}} + \overset{\texttt{ones\_row}}{\begin{bmatrix} 1 & 1 \end{bmatrix}} = \overset{\texttt{data}}{\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}} + \overset{\texttt{ones\_row}}{\begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}} = \begin{bmatrix} 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{bmatrix}$$

Matrix/Vector/Scalar Operations

Demo Program: array4_2D_Operations.py

```
import numpy as np
data = np.array([
 [1, 2],
 [3, 4],
 [5, 6]
])
print(data)
ones32 = np.ones((3, 2))
ones = np.ones(2)
one  = 1
print("data+ones32:\n", (data+ones32))
print("data+ones:\n", (data+ones))
print("data+one:\n", (data+one))
```

Dot Product



powers_of_ten
data
1
1
2
3
.dot(
1
10
100
1,000
10,000
100,000
)
=
30201
302010
3
Matrix dimensions:
1x3
3x2
1x2

Dot Product

Demo Program: array4_dot.py

```
import numpy as np

data = np.array([1, 2, 3])
power = [10**i for i in range(6)]
power_of_ten = np.array(power).reshape((3, 2))
print("data:\n", data)
print("power:\n", power_of_ten)
dot_product = data.dot(power_of_ten)
print("dot_product:\n", dot_product)
```

```
data:
 [1 2 3]
power:
 [[     1     10]
 [   100   1000]
 [ 10000 100000]]
dot_product:
 [ 30201 302010]
```

Sum

	1	100	10,000			10	1,000	100,000	
sum(	*	*	*	)	sum(	*	*	*	)
	1	2	3			1	2	3	

1x2

1*1 + 2*100 + 3*10,000	1*10 + 2*1,000 + 3*100,000	=	30201	302010

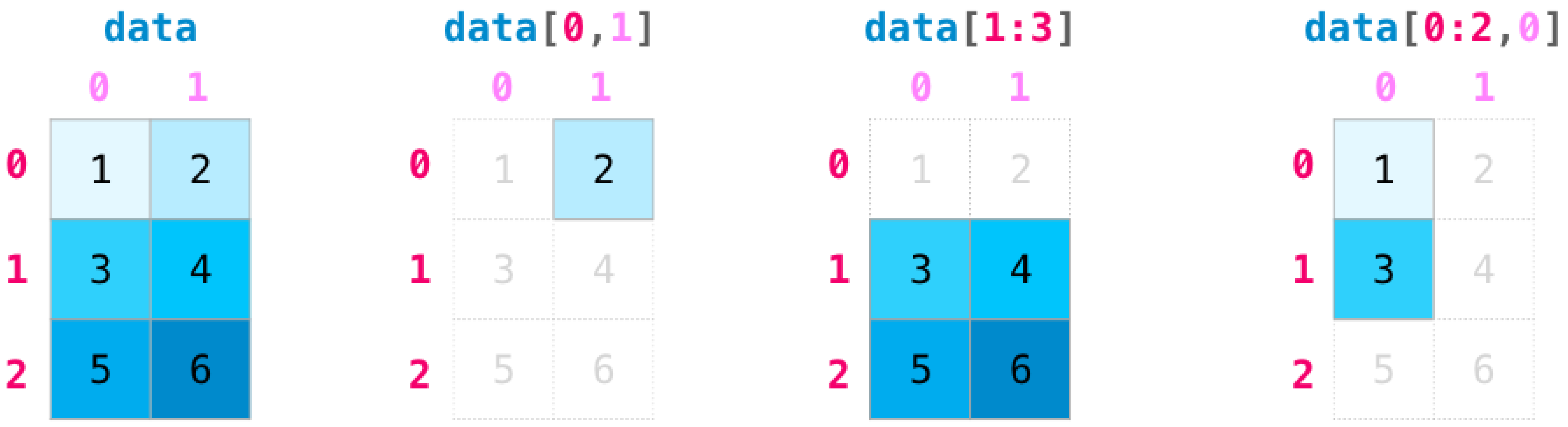
Dot Product

Demo Program: array4_dot2.py

```
import numpy as np
data = np.array([1, 2, 3])
power = [10**i for i in range(6)]
power_of_ten = np.array(power).reshape((3, 2))
c0 = power_of_ten[:, 0]
c1 = power_of_ten[:, 1]
print("data:", data)
print("c0:", c0)
print("c1:", c1)
x0 = data.dot(c0)
x1 = data.dot(c1)
print("sum of data * c0 : ", x0)
print("sum of data * c1 : ", x1)
s0 = sum(data * c0)
s1 = sum(data * c1)
print("sum of data * c0.T : ", s0)
print("sum of data * c1.T : ", s1)
```

```
data: [1 2 3]
c0: [     1    100 10000]
c1: [    10   1000 100000]
sum of data * c0 :  30201
sum of data * c1 :  302010
sum of data * c0.T :  30201
sum of data * c1.T :  302010
```

Matrix Indexing



Matrix Aggregation

data

1	2
3	4
5	6

`.max() =` 6

data

1	2
3	4
5	6

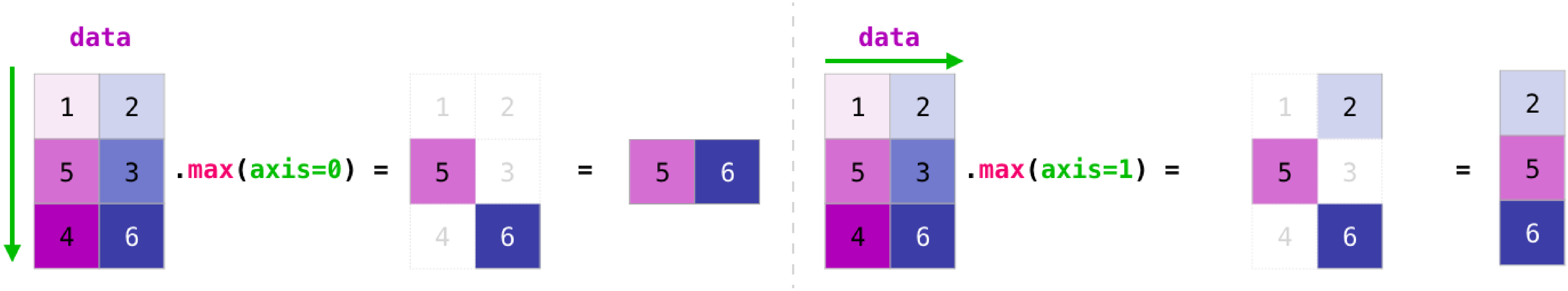
`.min() =` 1

data

1	2
3	4
5	6

`.sum() =` 21

Matrix Aggregation (Row/Column Level)



Arrays

SECTION 5 TRANSPOSE AND RESHAPE

Transposing and Reshaping

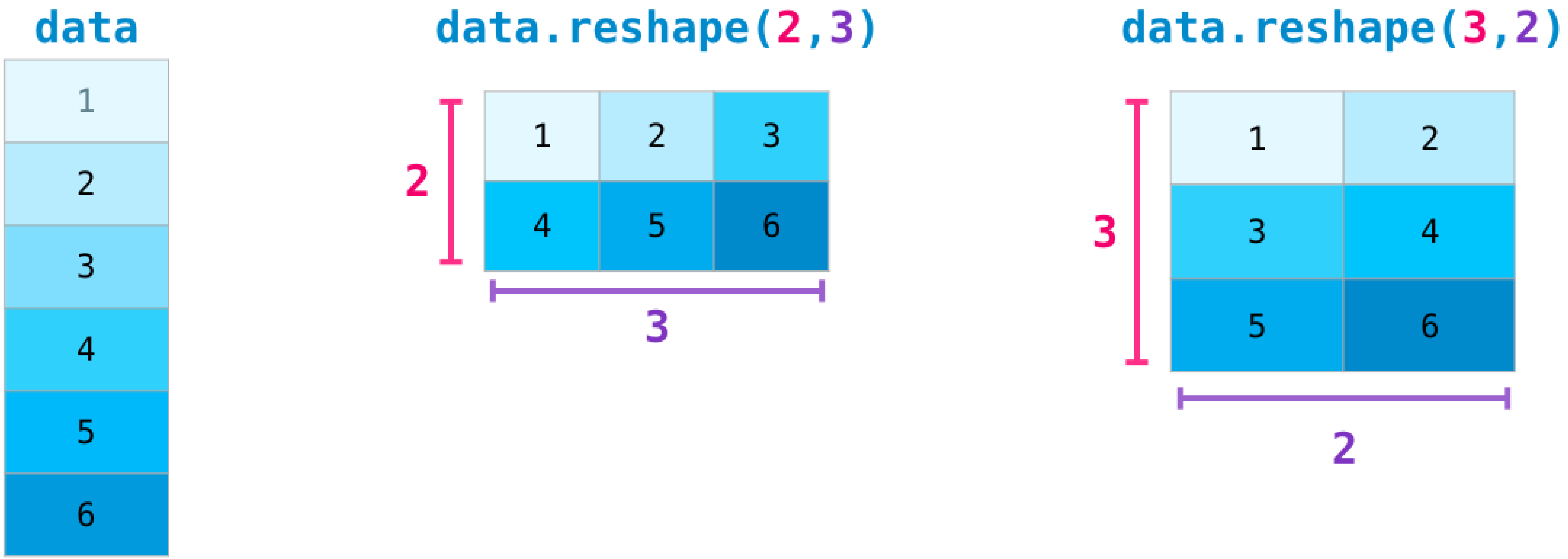
`data`

1	2
3	4
5	6

`data.T`

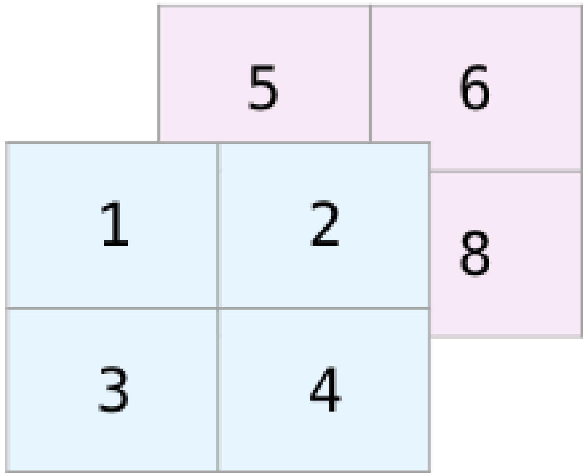
1	3	5
2	4	6

Transposing and Reshaping



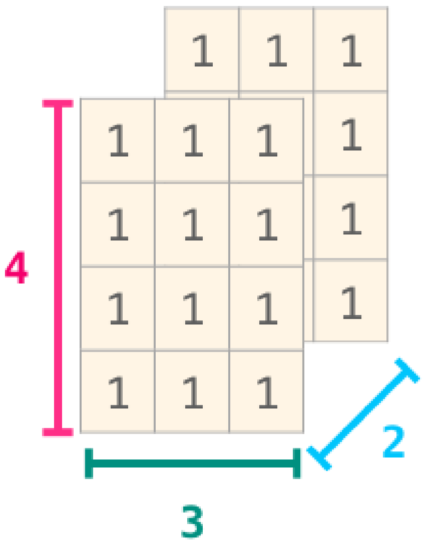
Multiple Dimension

```
np.array([ [[1,2],[3,4]],
           [[5,6],[7,8]] ])
```

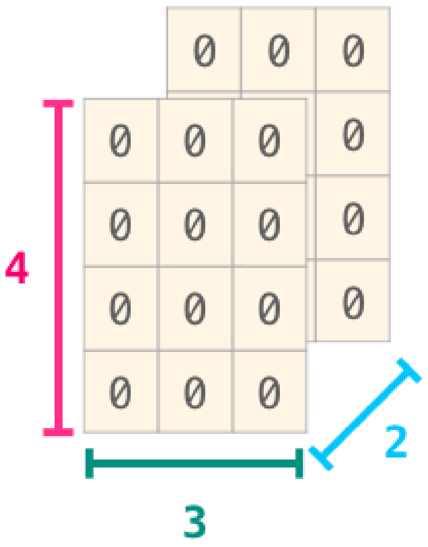


MD Array Creation

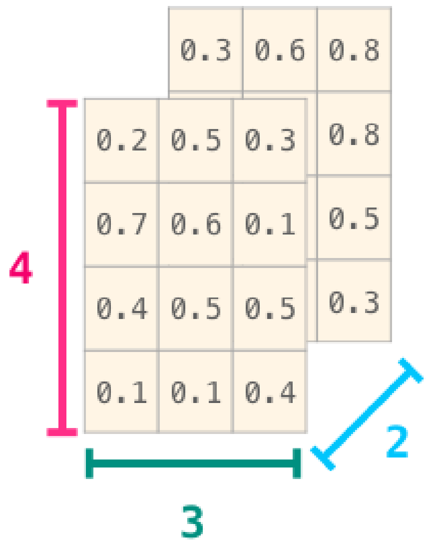
```
np.ones((4,3,2))
```




```
np.zeros((4,3,2))
```

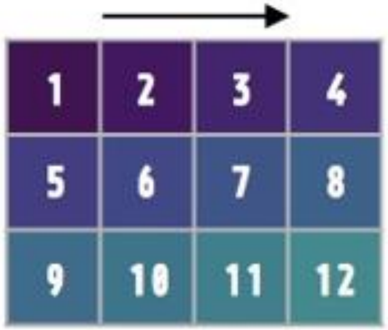



```
np.random.random((4,3,2))
```



reshape & ravel

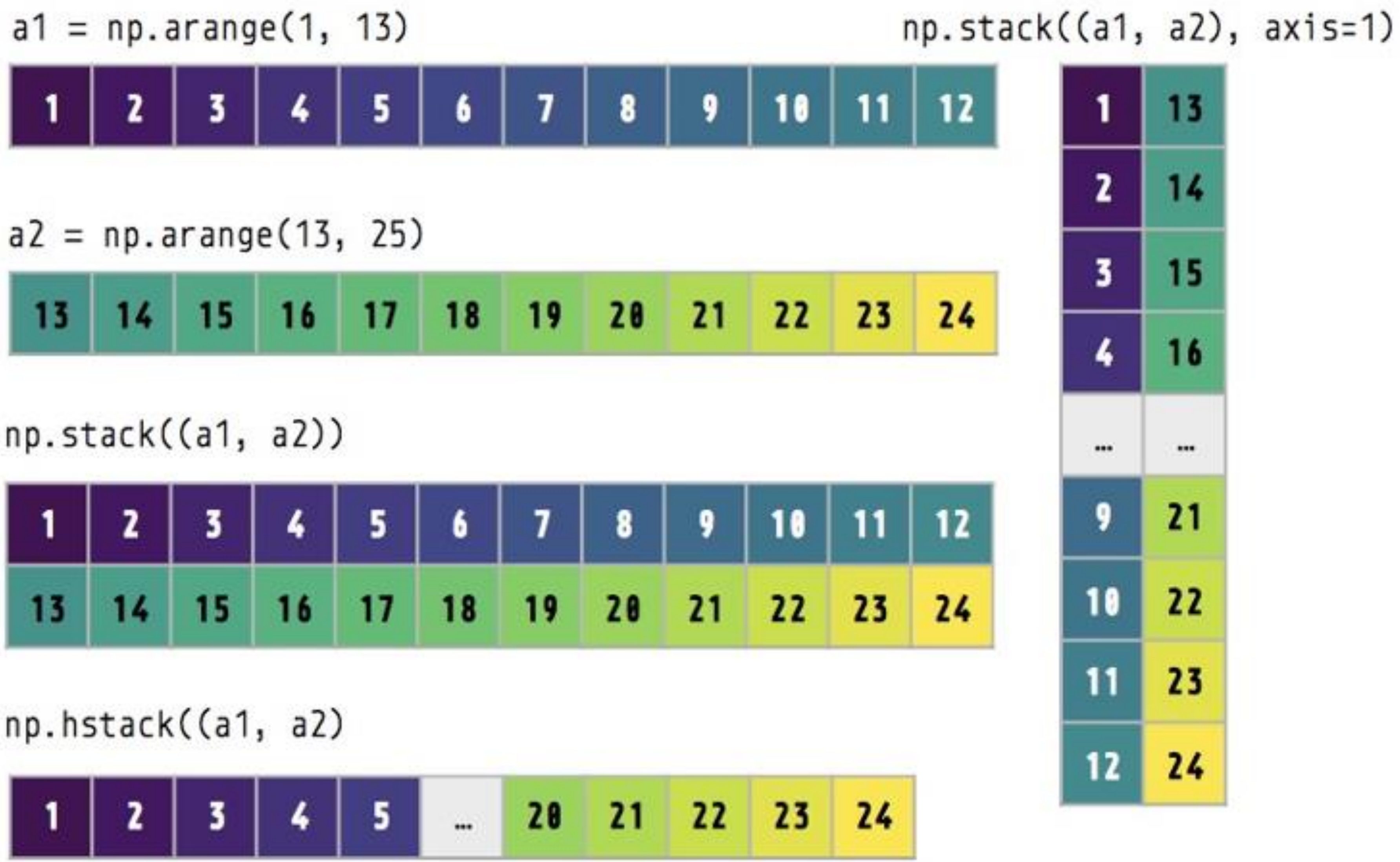
```
a1 = np.arange(1, 13)
```



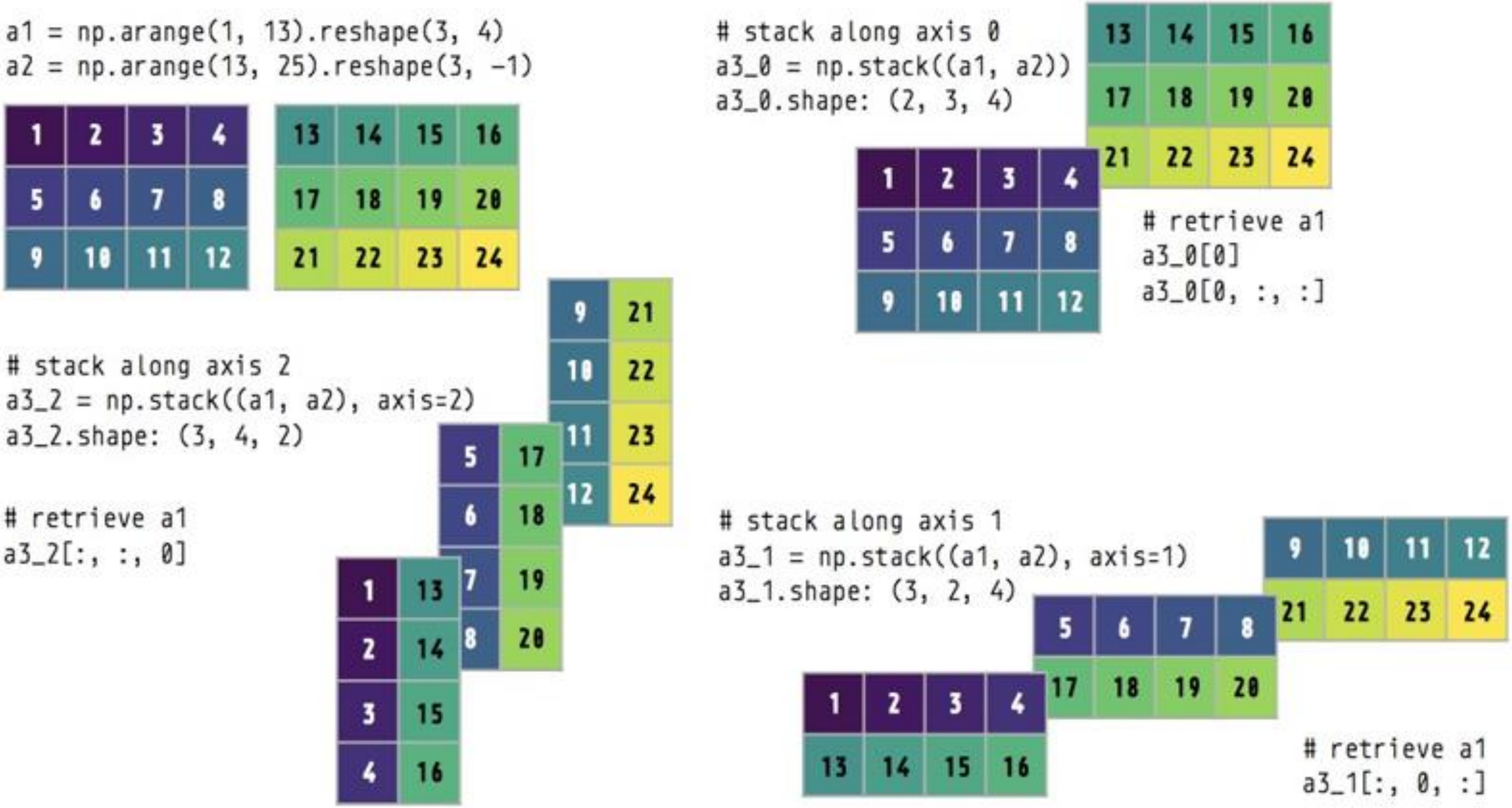

```
a1.reshape(3, 4)
a1.reshape(-1, 4)
a1.reshape(3, -1)
  .ravel()  # back to 1D
```

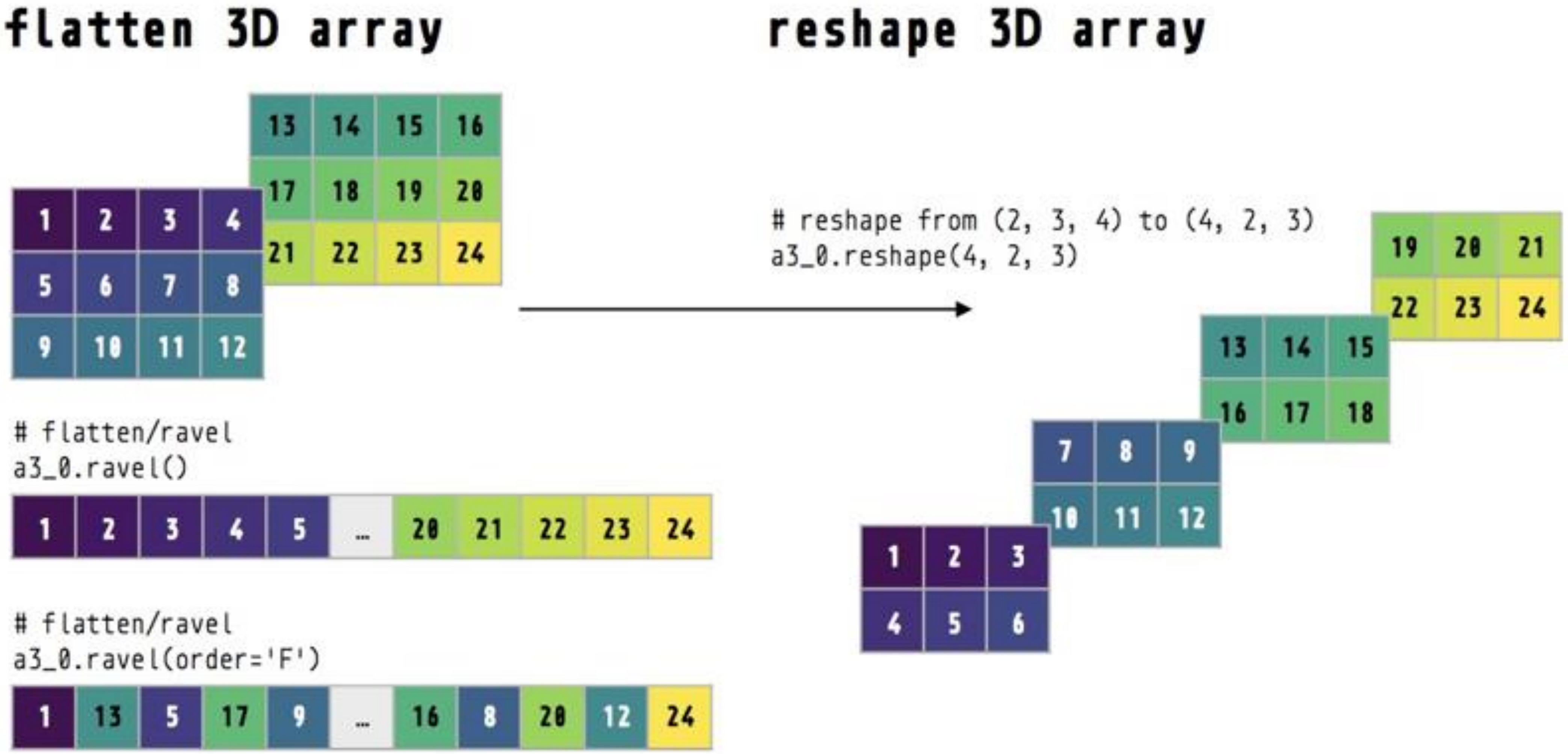

```
a1.reshape(3, -1, order='F')
  .ravel(order='F')  # back to 1D
```

stack


a1 = np.arange(1, 13)
1 2 3 4 5 6 7 8 9 10 11 12
a2 = np.arange(13, 25)
13 14 15 16 17 18 19 20 21 22 23 24
np.stack((a1, a2))
1 2 3 4 5 6 7 8 9 10 11 12
13 14 15 16 17 18 19 20 21 22 23 24
np.hstack((a1, a2)
1 2 3 4 5 ... 20 21 22 23 24
np.stack((a1, a2), axis=1)
1 13
2 14
3 15
4 16
... ...
9 21
10 22
11 23
12 24

3D array from 2D arrays




flatten 3D array
reshape 3D array
13 14 15 16
17 18 19 20
21 22 23 24
1 2 3 4
5 6 7 8
9 10 11 12
# flatten/ravel
a3_0.ravel()
1 2 3 4 5 … 20 21 22 23 24
# flatten/ravel
a3_0.ravel(order='F')
1 13 5 17 9 … 16 8 20 12 24
# reshape from (2, 3, 4) to (4, 2, 3)
a3_0.reshape(4, 2, 3)
19 20 21
22 23 24
13 14 15
16 17 18
7 8 9
10 11 12
1 2 3
4 5 6

Arrays

SECTION 6 INSERTION/DELETION OF ROW AND COLUMN

Array Creation

Demo Program: array5_InsDel1.py

```
import numpy as np

a = np.array([['Mon', 18, 20, 22, 17], ['Tue', 11, 18, 21, 18],
              ['Wed', 15, 21, 20, 19], ['Thu', 11, 20, 22, 21],
              ['Fri', 18, 17, 23, 22], ['Sat', 12, 22, 20, 18],
              ['Sun', 13, 15, 19, 16]])

m = np.reshape(a, (7, 5))
print(m)
```

Array Creation

Demo Program: array5_InsDel1.py

```
[['Mon' '18' '20' '22' '17']
 ['Tue' '11' '18' '21' '18']
 ['Wed' '15' '21' '20' '19']
 ['Thu' '11' '20' '22' '21']
 ['Fri' '18' '17' '23' '22']
 ['Sat' '12' '22' '20' '18']
 ['Sun' '13' '15' '19' '16’]]
```

Add a Row

Demo Program: array5_InsDel2.py

```python
import numpy as np

m = np.array([['Mon', 18, 20, 22, 17], ['Tue', 11, 18, 21, 18],
              ['Wed', 15, 21, 20, 19], ['Thu', 11, 20, 22, 21],
              ['Fri', 18, 17, 23, 22], ['Sat', 12, 22, 20, 18],
              ['Sun', 13, 15, 19, 16]])

m_r = np.append(m, [['Avg', 12, 15, 13, 11]], 0)

print(m_r)
```

Add a Row

Demo Program: array5_InsDel2.py

```
[['Mon' '18' '20' '22' '17']
 ['Tue' '11' '18' '21' '18']
 ['Wed' '15' '21' '20' '19']
 ['Thu' '11' '20' '22' '21']
 ['Fri' '18' '17' '23' '22']
 ['Sat' '12' '22' '20' '18']
 ['Sun' '13' '15' '19' '16']
 ['Avg' '12' '15' '13' '11']]
```

Add a Column

Demo Program: array5_InsDel3.py

```
import numpy as np

m = np.array([['Mon', 18, 20, 22, 17], ['Tue', 11, 18, 21, 18],
              ['Wed', 15, 21, 20, 19], ['Thu', 11, 20, 22, 21],
              ['Fri', 18, 17, 23, 22], ['Sat', 12, 22, 20, 18],
              ['Sun', 13, 15, 19, 16]])

m_c = np.insert(m, [5], [[1], [2], [3], [4], [5], [6], [7]], 1)

print(m_c)
```

Add a Column

Demo Program: array5_InsDel3.py

```
[['Mon' '18' '20' '22' '17' '1']
 ['Tue' '11' '18' '21' '18' '2']
 ['Wed' '15' '21' '20' '19' '3']
 ['Thu' '11' '20' '22' '21' '4']
 ['Fri' '18' '17' '23' '22' '5']
 ['Sat' '12' '22' '20' '18' '6']
 ['Sun' '13' '15' '19' '16' '7']]
```

Delete a Row

Demo Program: array5_InsDel4.py

```
import numpy as np

m = np.array([['Mon', 18, 20, 22, 17], ['Tue', 11, 18, 21, 18],
              ['Wed', 15, 21, 20, 19], ['Thu', 11, 20, 22, 21],
              ['Fri', 18, 17, 23, 22], ['Sat', 12, 22, 20, 18],
              ['Sun', 13, 15, 19, 16]])

m = np.delete(m, [2], 0)

print(m)
```

Delete a Row

Demo Program: array5_InsDel4.py

```
[['Mon' '18' '20' '22' '17']
 ['Tue' '11' '18' '21' '18']
 ['Thu' '11' '20' '22' '21']
 ['Fri' '18' '17' '23' '22']
 ['Sat' '12' '22' '20' '18']
 ['Sun' '13' '15' '19' '16’]]
```

Delete a Column

Demo Program: array5_InsDel5.py

```
import numpy as np

m = np.array([['Mon', 18, 20, 22, 17], ['Tue', 11, 18, 21, 18],
              ['Wed', 15, 21, 20, 19], ['Thu', 11, 20, 22, 21],
              ['Fri', 18, 17, 23, 22], ['Sat', 12, 22, 20, 18],
              ['Sun', 13, 15, 19, 16]])

m = np.delete(m, [2], 1)

print(m)
```

Delete a Column

Demo Program: array5_InsDel5.py

```
[['Mon' '18' '22' '17']
 ['Tue' '11' '21' '18']
 ['Wed' '15' '20' '19']
 ['Thu' '11' '22' '21']
 ['Fri' '18' '23' '22']
 ['Sat' '12' '20' '18']
 ['Sun' '13' '19' '16’]]
```

Modify a Row

Demo Program: array5_InsDel6.py

```
import numpy as np

m = np.array([['Mon', 18, 20, 22, 17], ['Tue', 11, 18, 21, 18],
              ['Wed', 15, 21, 20, 19], ['Thu', 11, 20, 22, 21],
              ['Fri', 18, 17, 23, 22], ['Sat', 12, 22, 20, 18],
              ['Sun', 13, 15, 19, 16]])

m[3] = ['Thu', 0, 0, 0, 0]

print(m)
```

Modify a Row

Demo Program: array5_InsDel6.py

```
[['Mon' '18' '20' '22' '17']
 ['Tue' '11' '18' '21' '18']
 ['Wed' '15' '21' '20' '19']
 ['Thu' '0' '0' '0' '0']
 ['Fri' '18' '17' '23' '22']
 ['Sat' '12' '22' '20' '18']
 ['Sun' '13' '15' '19' '16′]]
```

Arrays

SECTION 7 PRACTICAL USE

Formulas

- Implementing mathematical formulas that work on matrices and vectors is a key use case to consider NumPy for. It's why NumPy is the darling of the scientific python community. For example, consider the mean square error formula that is central to supervised machine learning models tackling regression problems:

$$MeanSquareError = \frac{1}{n} \sum_{i=1}^{n} (Y\_prediction_i - Y_i)^2$$

NumPy

- Implementing this is a breeze in NumPy:

```
error = (1/n) * np.sum(np.square(predictions - labels))
```

- The beauty of this is that numpy does not care if predictions and labels contain one or a thousand values (as long as they're both the same size). We can walk through an example stepping sequentially through the four operations in that line of code:

$$\texttt{error = (1/3) * np.sum(np.square(} \overset{\texttt{predictions}}{\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}} - \overset{\texttt{labels}}{\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}} \texttt{))}$$

Explanation

```
error = (1/3) * np.sum(np.square( [0, -1, -2] ))

error = (1/3) * np.sum( [0, 1, 4] )

error = (1/3) * [5]
```

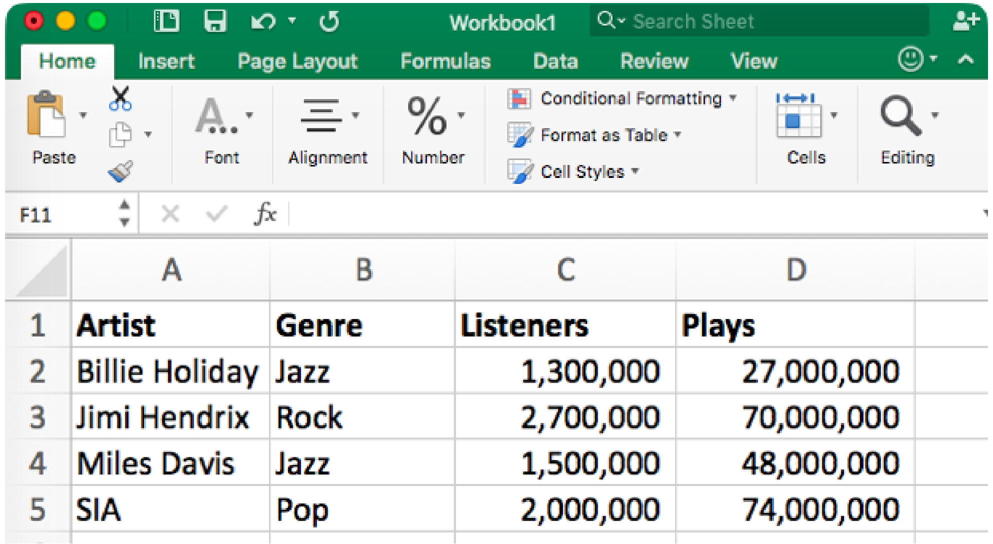
Mean Square Error

Demo Program: array6_MSE.py

```
import numpy as np

predictions = np.array([1, 1, 1])
labels      = np.array([1, 2, 3])
print("predictions: ", predictions)
print("label       : ", labels)
MSE         = np.sum(np.square(predictions-labels))
print("MSE         : ", MSE)
```

Arrays

SECTION 8: DATA MODELS - TABLES, AUDIO, IMAGE, AND LANGUAGE `music.csv`



	A	B	C	D
1	Artist	Genre	Listeners	Plays
2	Billie Holiday	Jazz	1,300,000	27,000,000
3	Jimi Hendrix	Rock	2,700,000	70,000,000
4	Miles Davis	Jazz	1,500,000	48,000,000
5	SIA	Pop	2,000,000	74,000,000

`pandas.read_csv('music.csv')`

	Artist	Genre	Listeners	Plays
0	Billie Holiday	Jazz	1,300,000	27,000,000
1	Jimi Hendrix	Rock	2,700,000	70,000,000
2	Miles Davis	Jazz	1,500,000	48,000,000

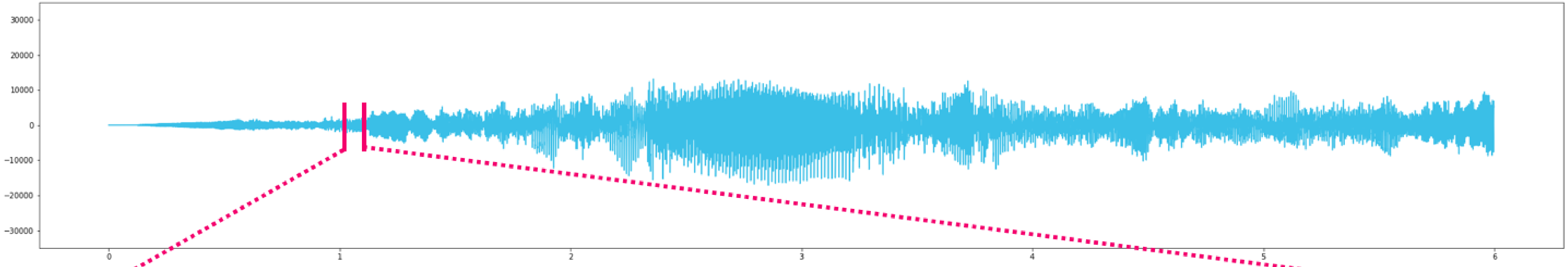
Data Representation
Tables and Spreadsheets

Load a .csv

Demo Program: music.py

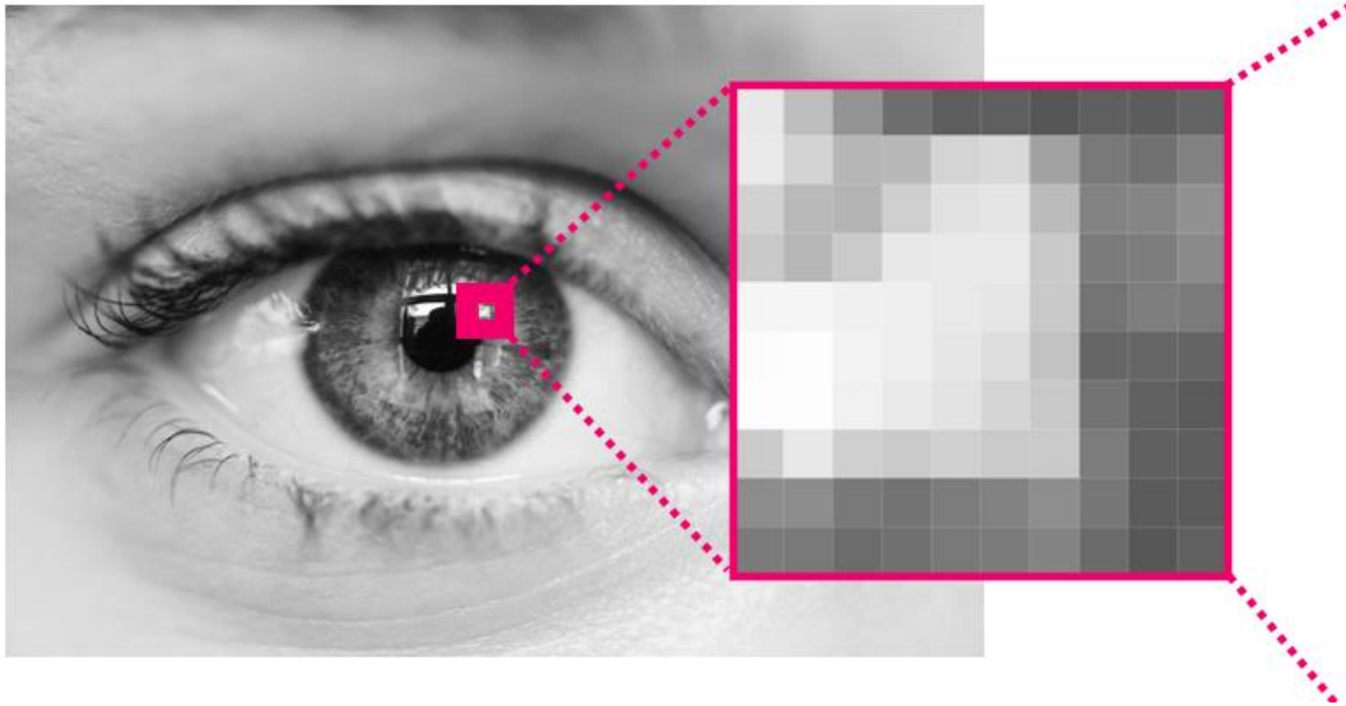
```
import numpy as np
import pandas as pd

m = pd.read_csv("music.csv")
print(type(m))
print(m)
print()
print(type(m.keys()))
header = list(m.keys())
print("Header: ", header)
print()
ma = m.values
print(type(ma))
print(ma)
```

```
<class 'pandas.core.frame.DataFrame'>
         Artist Genre  Listeners     Plays
0  Billie Holiday  Jazz    1300000  27000000
1    Jimi Hendrix  Rock    2700000  70000000
2     Miles Davis  Jazz    1200000  48000000
3             SIA   Pop    2000000  74000000

<class 'pandas.core.indexes.base.Index'>
Header:  ['Artist', 'Genre', 'Listeners', 'Plays']

<class 'numpy.ndarray'>
[['Billie Holiday' 'Jazz' 1300000 27000000]
 ['Jimi Hendrix' 'Rock' 2700000 70000000]
 ['Miles Davis' 'Jazz' 1200000 48000000]
 ['SIA' 'Pop' 2000000 74000000]]
```



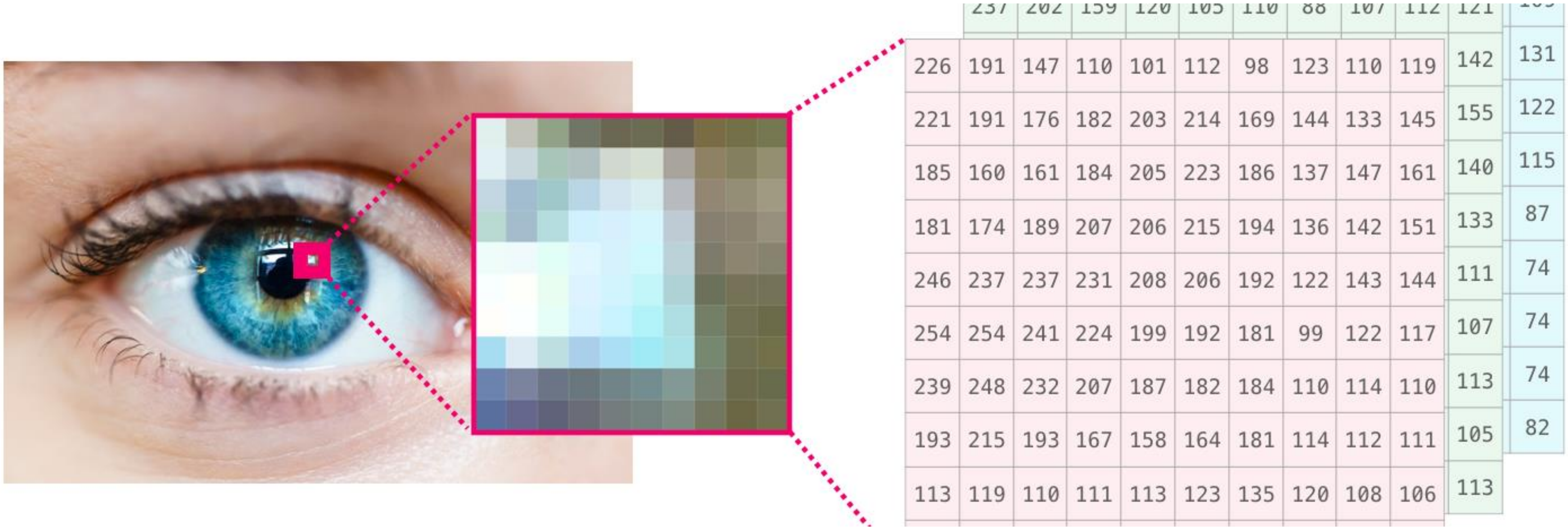
326	217	142	73	27	7	14	38	65	90	101	90	74	37	-12	-77	-154	-232	-304	-355	-433

Audio and Timeseries



230	194	147	108	90	98	84	96	91	101
237	206	188	195	207	213	163	123	116	128
210	183	180	205	224	234	188	122	134	147
198	189	201	227	229	232	200	125	127	135
249	241	237	244	232	226	202	116	125	126
251	254	241	239	230	217	196	102	103	99
243	255	240	231	227	214	203	116	95	91
204	231	208	200	207	201	200	121	95	95
144	140	120	115	125	127	143	118	92	91

Images


226 191 147 110 101 112 98 123 110 119
221 191 176 182 203 214 169 144 133 145
185 160 161 184 205 223 186 137 147 161
181 174 189 207 206 215 194 136 142 151
246 237 237 231 208 206 192 122 143 144
254 254 241 224 199 192 181 99 122 117
239 248 232 207 187 182 184 110 114 110
193 215 193 167 158 164 181 114 112 111
113 119 110 111 113 123 135 120 108 106
142 155 140 133 111 107 113 105 113
131 122 115 87 74 74 74 82

Color Image

Model Vocabulary

#	
0	the
1	of
2	and
…	…
71,289	dolophine

Language

Matrix (mat)

SECTION 3

Numpy Matrices

- Numpy matrices are strictly 2-dimensional, while numpy arrays (ndarrays) are N-dimensional. **Matrix** objects are a **subclass of ndarray**, so they inherit all the attributes and methods of ndarrays.
- The main advantage of numpy matrices is that they provide a convenient notation for matrix multiplication: if a and b are matrices, then a*b is their matrix product.

NumPy array creation: mat() function

mat() function:

numpy.mat(data, dtype=None)

Description:

The mat() function is used to interpret a given input as a matrix.

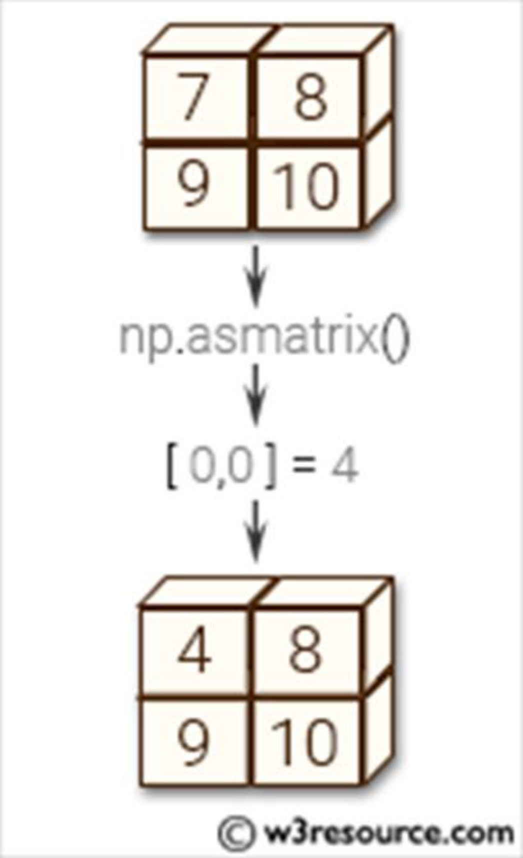
Unlike matrix, asmatrix does not make a copy if the input is already a matrix or an ndarray. Equivalent to matrix(data, copy=False).

NumPy array creation: mat() function

Demo Program: mat0.py

```
import numpy as np
a = np.mat([[7,8], [9,10]])
a[0,0] = 4

print(a)
```

```
[[ 4  8]
 [ 9 10]]
```



NumPy array asmatrix: asmatrix() function

asmatrix() function:

numpy.asmatrix(array)

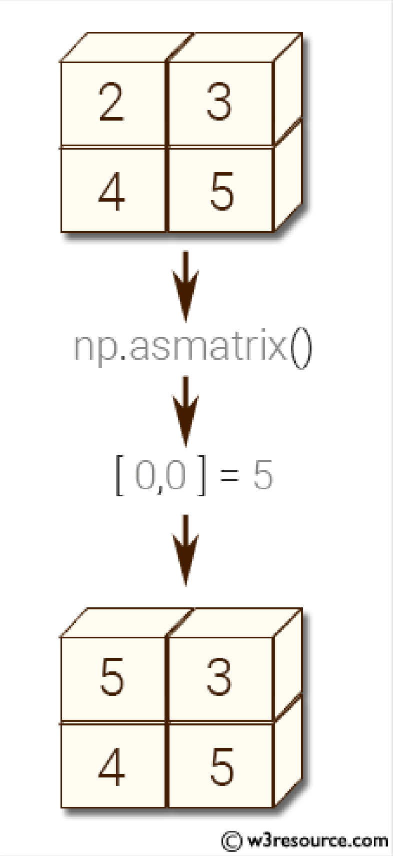
Description:

data interpreted as a matrix.

NumPy array as Matrix: asmatrix() function

Demo Program: mat1.py

```
import numpy as np
a = np.array([[2,3], [4,5]])
y = np.asmatrix(a)    # shallow copy

a[0,0] = 5

print(y)

[[5 3]
 [4 5]]
```



NumPy array creation: bmat() function

numpy.bmat() function:

numpy.bmat(obj, ldict=None, gdict=None)

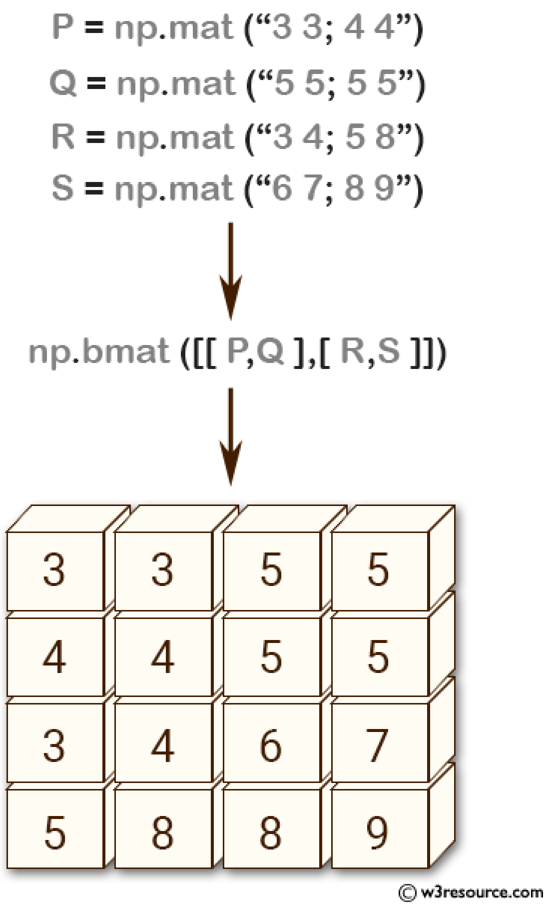
Description:

The numpy.bmat() function is used to built a matrix object from a string, nested sequence, or array.

NumPy Create Matrix: bmat() function

Demo Program: mat2.py

```
import numpy as np
P = np.mat('3 3; 4 4')
Q = np.mat('5 5; 5 5')
R = np.mat('3 4; 5 8')
S = np.mat('6 7; 8 9')
m1 = np.bmat([[P,Q], [R, S]])
print(m1)

[[3 3 5 5]
 [4 4 5 5]
 [3 4 6 7]
 [5 8 8 9]]
```



NumPy Create Matrix: bmat() function

Demo Program: mat3.py

```
import numpy as np
P = np.mat('3 3; 4 4')
Q = np.mat('5 5; 5 5')
R = np.mat('3 4; 5 8')
S = np.mat('6 7; 8 9')
m1 = np.bmat(np.r_[np.c_[P,Q], np.c_[R, S]])
print(m1)
m2 = np.bmat("P Q; R S")
print(m2)
```

```
[[3 3 5 5]
 [4 4 5 5]
 [3 4 6 7]
 [5 8 8 9]]
[[3 3 5 5]
 [4 4 5 5]
 [3 4 6 7]
 [5 8 8 9]]
```

Matrix (matrix)

SECTION 4

NumPy numpy.matrix()

numpy.matrix() function:

numpy.matrix(data, dtype = None)

Description:

This class returns a matrix from a string of data or array-like object. Matrix obtained is a specialised 2D array.

Parameters :

- data : data needs to be array-like or string
- dtype : Data type of returned array.

numpy.matrix

Note: It is no longer recommended to use this class, even for linear algebra. Instead use regular arrays. The class may be removed in the future.

- Returns a matrix from an array-like object, or from a string of data. **A matrix is a specialized 2-D array that retains its 2-D nature through operations**. It has certain special operators, such as * (matrix multiplication) and ** (matrix power).
- **Parameters**
    - **data*array_like or string***
        - If data is a string, it is interpreted as a matrix with commas or spaces separating columns, and semicolons separating rows.
    - **dtype*data-type***
        - Data-type of the output matrix.
    - **copy*bool***
        - If data is already an ndarray, then this flag determines whether the data is copied (the default), or whether a view is constructed.

NumPy Create Matrix: matrix() function

Demo Program: mat4.py

```
import numpy as np

# string input
a = np.matrix('1 2; 3 4')
print("Via string input : \n", a, "\n\n")

# array-like input
b = np.matrix([[5, 6, 7], [4, 6]])
print("Via array-like input : \n", b)
```

```
Via string input :
 [[1 2]
 [3 4]]

Via array-like input :
 [[list([5, 6, 7]) list([4, 6])]]
```

Basic Matrix Operations

SECTION 1

Basic Matrix Operations

1. **add() :-** This function is used to perform **element wise matrix addition**.
2. **subtract() :-** This function is used to perform **element wise matrix subtraction**.
3. **divide() :-** This function is used to perform **element wise matrix division**.
4. **multiply() :-** This function is used to perform **element wise matrix multiplication**.

Matrix Basic Operations

Demo Program: mat5.py

```
import numpy as np
# initializing matrices
x = np.array([[1, 2], [4, 5]])
y = np.array([[7, 8], [9, 10]])
# using add() to add matrices
print("The element wise addition of matrix is : ")
print(np.add(x, y))
# using subtract() to subtract matrices
print("The element wise subtraction of matrix is : ")
print(np.subtract(x, y))
# using divide() to divide matrices
print("The element wise division of matrix is : ")
print(np.divide(x, y))
# using multiply() to multiply matrices element wise
print ("The element wise multiplication of matrix is : ")
print (np.multiply(x,y))
```

Matrix Basic Operations

Demo Program: mat5.py

```
The element wise addition of matrix is :
[[ 8 10]
 [13 15]]
The element wise subtraction of matrix is :
[[-6 -6]
 [-5 -5]]
The element wise division of matrix is :
[[0.14285714 0.25      ]
 [0.44444444 0.5       ]]
The element wise multiplication of matrix is :
[[ 7 16]
 [36 50]]
```

Basic Matrix Operations

5. **dot() :-** This function is used to compute the **matrix multiplication, rather than element wise multiplication**.
6. **sqrt() :-** This function is used to compute the **square root of each element** of matrix.
7. **sum(x,axis) :-** This function is used to **add all the elements in matrix**. Optional "axis" argument computes the **column sum if axis is 0** and **row sum if axis is 1**.
8. **"T" :-** This argument is used to **transpose** the specified matrix.

Matrix Basic Operations

Demo Program: mat6.py

```
The product of matrices is :
[[25 28]
 [73 82]]
The element wise square root is :
[[1.         1.41421356]
 [2.         2.23606798]]
The summation of all matrix element is :
34
The column wise summation of all matrix  is :
[16 18]
The row wise summation of all matrix  is :
[15 19]
The transpose of given matrix is :
[[1 4]
 [2 5]]
```

```python
import numpy as np
# initializing matrices
x = np.array([[1, 2], [4, 5]])
y = np.array([[7, 8], [9, 10]])
# using dot() to multiply matrices
print ("The product of matrices is : ")
print (np.dot(x,y))
# using sqrt() to print the square root of matrix
print("The element wise square root is : ")
print(np.sqrt(x))
# using sum() to print summation of all elements of matrix
print("The summation of all matrix element is : ")
print(np.sum(y))
# using sum(axis=0) to print summation of all columns of matrix
print("The column wise summation of all matrix  is : ")
print(np.sum(y, axis=0))
# using sum(axis=1) to print summation of all columns of matrix
print("The row wise summation of all matrix  is : ")
print(np.sum(y, axis=1))
# using "T" to transpose the matrix
print("The transpose of given matrix is : ")
print(x.T)
```

Matrix Operations

SECTION 2

Arithmetic Operations

Recall that arithmetic array operations +, -, /, * and ** are performed elementwise on NumPy arrays. Let's create a NumPy array and do some computations:

```
M = np.array([[3,4],[-1,5]])
>>> print(M)
[[ 3 4]
 [-1 5]]

>>> M * M
array([[ 9, 16],
       [ 1, 25]])
```

Matrix Multiplication

We use the @ operator to do matrix multiplication with NumPy arrays:

M @ M

```
array([[ 5, 32],
       [-8, 21]])
```

Example

Demo Program: mat7.py

Let's compute $2I + 3A - AB$ for

$$A = \begin{bmatrix} 1 & 3 \\ -1 & 7 \end{bmatrix} \quad B = \begin{bmatrix} 5 & 2 \\ 1 & 2 \end{bmatrix}$$

```
I = np.eye(2)

2*I + 3*A - A@B

array([[-3.,  1.],
       [-5., 11.]])
```

Example

Demo Program: mat7.py

```
import numpy as np

A = np.array([[1,3],[-1,7]])
B = np.array([[5,2],[1,2]])
I = np.eye(2)

X = 2*I + 3*A - A@B
print(X)

[[-3.  1.]
 [-5. 11.]]
```

Matrix Powers

Demo Program: mat8.py

- There's no symbol for matrix powers and so we must import the function matrix_power from the subpackage numpy.linalg.

```
import numpy as np
from numpy.linalg import matrix_power as mpow
M = np.array([[3,4],[-1,5]])
print(M)
print(mpow(M,2))
```

Matrix Transpose M^T

Demo Program: mat9.py

```
import numpy as np
M = np.array([[3,4],[-1,5]])
print(M)
print(M.T)
print(M@M.T)
```

```
[[ 3  4]
 [-1  5]]
[[ 3 -1]
 [ 4  5]]
[[25 17]
 [17 26]]
```

Matrix Inverse

We can find the inverse using the function scipy.linalg.inv:
Demo Program: mat10.py

```
import numpy as np
import scipy.linalg as la
A = np.array([[1,2],[3,4]])
print(A)
print(la.inv(A))

[[1 2]
 [3 4]]
[[-2.   1. ]
 [ 1.5 -0.5]]
```

Trace

Demo Program: mat11.py

```
import numpy as np
A = np.array([[1,2],[3,4]])
# sum of the diagonal terms
print(np.trace(A))

5
```

Norm

Demo Program: mat12.py

The following norms can be calculated:

ord	norm for matrices	norm for vectors
None	Frobenius norm	2-norm
'fro'	Frobenius norm	–
'nuc'	nuclear norm	–
inf	max(sum(abs(x), axis=1))	max(abs(x))
-inf	min(sum(abs(x), axis=1))	min(abs(x))
0	–	sum(x != 0)
1	max(sum(abs(x), axis=0))	as below
-1	min(sum(abs(x), axis=0))	as below
2	2-norm (largest sing. value)	as below
-2	smallest singular value	as below
other	–	sum(abs(x)**ord)**(1./ord)

The Frobenius norm is given by [1]:

> $||A||_F = [\sum_{i,j} abs(a_{i,j})^2]^{1/2}$

The nuclear norm is the sum of the singular values.

Norm

Demo Program: mat13.py

```
import numpy as np
from numpy import linalg as la
# Using the axis argument to compute vector norms:
c = np.array([[ 1, 2, 3],
              [-1, 1, 4]])
print(la.norm(c, axis=0))
print(la.norm(c, axis=1))
print(la.norm(c, ord=1, axis=1))
# Using the axis argument to compute matrix norms:
m = np.arange(8).reshape(2,2,2)
print(la.norm(m, axis=(1,2)))
print(la.norm(m[0, :, :]), la.norm(m[1, :, :]))
```

```
[1.41421356 2.23606798 5.        ]
[3.74165739 4.24264069]
[6. 6.]
[ 3.74165739 11.22497216]
3.7416573867739413 11.224972160321824
```

```python
import numpy as np
from numpy import linalg as la
a = np.arange(9) - 4
print('a=', a)
b = a.reshape((3, 3))
print('b=', b)

print(la.norm(a))
print(la.norm(b))
print(la.norm(b, 'fro'))
print(la.norm(a, np.inf))
print(la.norm(b, np.inf))
print(la.norm(a, -np.inf))
print(la.norm(b, -np.inf))
print(la.norm(a, 1))
print(la.norm(b, 1))
print(la.norm(a, -1))
print(la.norm(b, -1))
print(la.norm(a, 2))
print(la.norm(b, 2))
print(la.norm(a, -2))
print(la.norm(b, -2))
print(la.norm(a, 3))
print(la.norm(a, -3))
```

```
a= [-4 -3 -2 -1  0  1  2  3  4]
b= [[-4 -3 -2]
 [-1  0  1]
 [ 2  3  4]]
7.745966692414834
7.745966692414834
7.745966692414834
4.0
9.0
0.0
2.0
20.0
7.0
0.0
6.0
7.745966692414834
7.3484692283495345
0.0
1.857033188519056e-16
5.848035476425731
0.0
```

Determinant

Demo Program: mat14.py

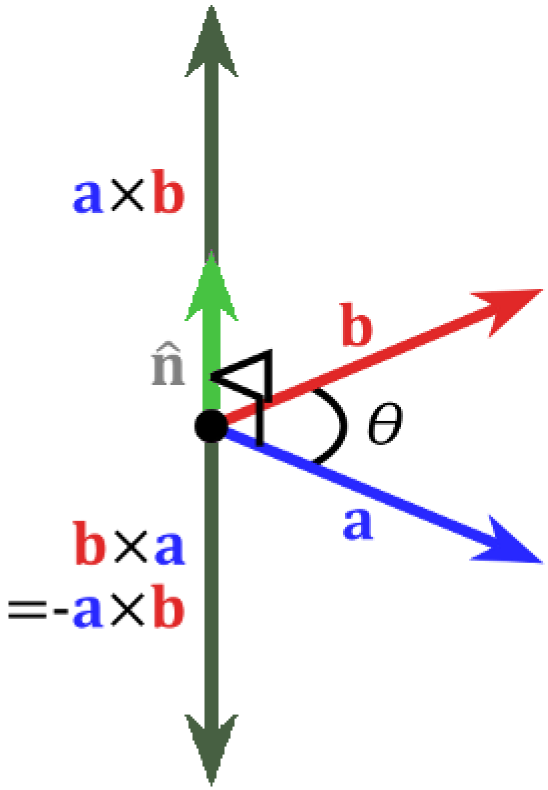
```
import numpy as np
import scipy.linalg as la

A = np.array([[1,2],[3,4]])
print(A)
print(la.det(A))

[[1 2]
 [3 4]]
-2.0
```

Dot Product

Demo Program: mat15.py

```
import numpy as np
x = np.array( ((2,3), (3, 5)) )
y = np.matrix( ((1,2), (5, -1)) )
print(np.dot(x,y))

# Alternatively, we can cast them into matrix
# objects and use the "*" operator:
z = np.mat(x) * np.mat(y)
print(z)

[[17  1]
 [28  1]]
[[17  1]
 [28  1]]
```

```python
import numpy as np
x = np.array([0,0,1])
y = np.array([0,1,0])

print(np.cross(x,y))
print(np.cross(y,x))

[-1  0  0]
[1   0  0]
```



Characteristic Polynomials and Cayley-Hamilton Theorem

Characteristic Polynomials and Cayley-Hamilton Theorem

The characteristic polynomial of a 2 by 2 square matrix A is

$$p_A(\lambda) = \det(A - \lambda I) = \lambda^2 - \mathrm{tr}(A)\lambda + \det(A)$$

The Cayley-Hamilton Theorem states that any square matrix satisfies its characteristic polynomial. For a matrix A of size 2, this means that

$$p_A(A) = A^2 - \mathrm{tr}(A)A + \det(A)I = 0$$

Verification

Demo Program: mat17.py

```
import numpy as np
import numpy.linalg as nla
import scipy.linalg as sla
A = np.array([[1, 2], [3, 4]])
print(A)

trace_A = np.trace(A)
det_A = nla.det(A)
I = np.eye(2)
SH = A @ A - trace_A * A + det_A * I
print(SH)
```

```
[[1 2]
 [3 4]]
[[-4.4408921e-16  0.0000000e+00]
 [ 0.0000000e+00 -4.4408921e-16]]
```

Projections

The formula to project a vector v onto a vector w is

$$\text{proj}_w(v) = \frac{v \cdot w}{w \cdot w} w$$

Let's a function called `proj` which computes the projection v onto w.

Projections

Demo Program: mat18.py

```
import numpy as np
def proj(v,w):
    '''Project vector v onto w.'''
    v = np.array(v)
    w = np.array(w)
    return np.sum(v * w)/np.sum(w * w) * w    # or (v @ w)/(w @ w) * w


print(proj([1,2,3],[1,1,1]))
```

```
[2. 2. 2.]
```

Advanced Matrix Operations

SECTION 3

Matrix Operations

- Add Two Matrices
- Multiply Two Matrices
- Solve Matrix Equation
- Find the Determinant
- Find the Inverse
- Find Eigenvalues
- Find Singular Value Decomposition (SVD)

Add Two Matrices

Demo Program: Add2Matrix.py

```
import numpy as np
matrix1 = np.matrix(
    [[0, 4],
     [2, 0]]
)
matrix2 = np.matrix(
    [[-1, 2],
     [1, -2]]
)

matrix_sum = matrix1 + matrix2

print(matrix_sum)
```

```
[[-1  6]
 [ 3 -2]]
```

Multiply Two Matrices

Demo Program: Mul2Matrix.py

```
import numpy as np
matrix1 = np.matrix(
    [[1, 4],
     [2, 0]]
)
matrix2 = np.matrix(
    [[-1, 2],
     [1, -2]]
)

matrix_prod = matrix1 * matrix2

print(matrix_prod)
```

```
[[ 3 -6]
 [-2  4]]
```

Solve Matrix Equation

Demo Program: SolveMatrix.py

How to find the solution of AX=B:

```
import numpy as np
A = np.matrix(
     [[1, 4],
      [2, 0]]
)
B = np.matrix(
     [[-1, 2],
      [1, -2]]
)

X = np.linalg.solve(A, B)

print(X)
```

```
[[ 0.5   -1.  ]
 [-0.375  0.75 ]]
```

Find the Matrix Determinant

Demo Program:Determinant.py

-7.9999999999999998

```
import numpy as np
matrix = np.matrix(
    [[1, 4],
     [2, 0]]
)

det = np.linalg.det(matrix)
print(det)
```

Find the Inverse

Demo Program: Inverse.py

```
[[ 0.    0.5  ]
 [ 0.25  -0.125]]
```

```
import numpy as np
matrix = np.matrix(
    [[1, 4],
     [2, 0]]
)

inverse = np.linalg.inv(matrix)
print(inverse)
```

Find Eigenvalues

Demo Program: eigenvalue.py

```
import numpy as np
matrix = np.matrix(
    [[1, 4],
     [2, 0]]
)

eigvals = np.linalg.eigvals(matrix)
print("The eigenvalues are %f and %f" %(eigvals[0], eigvals[1]))
```

```
The eigebvalues are 3.372281 and -2.372281
```

Find SVD

How to find the Singular Value Decomposition of a matrix, i.e. break up a matrix into the product of three matrices: U, Σ, V∗

```
import numpy as np
matrix = np.matrix(
     [[1, 4],
      [2, 0]]
)
svd = np.linalg.svd(matrix)
u = svd[0]
sigma = svd[1]
v = svd[2]
u = u.tolist()
sigma = sigma.tolist()
v = v.tolist()
matrix_prod = [
     ['U    ', '', '\u03A3    ', 'V*    ', ''],
     [u[0][0], u[0][1], sigma[0], v[0][0], v[0][1]],
     [u[1][0], u[1][1], sigma[1], v[1][0], v[1][1]]
]
```

```python
for i in range(len(matrix_prod)):
    row = matrix_prod[i]
    if (i==0):
        for col in row:
            print("%10s" % str(col), end=" ")
    else:
        for col in row:
            print("%10.6f" % col, end=" ")
    print()
```

```
     U                    Σ        V*
-0.988883  -0.148696  4.159415  -0.309244  -0.950983
-0.148696   0.988883  1.923347   0.950983  -0.309244
```

Exercises

SECTION 5

Exercises

1. Write a function which takes an input parameter A, i and j and returns the dot product of the ith and jth row (indexing starts at 0).

2. Compute the matrix equation $AB + 2B^2 - I$ for matrices

$$A = \begin{bmatrix} 3 & 4 \\ -1 & 2 \end{bmatrix} \text{ and } B = \begin{bmatrix} 5 & 2 \\ 8 & -3 \end{bmatrix}$$

Summary

SECTION 5

Summary

- In this lecture, we covered the Numpy arrays, matrix.
- Linear Algebra is the key pre-requisites for Robotics course. Therefore, we provide this lecture in a pre-recorded video lecture format.
- Please watch this video and download the sample Python code to study.