SECTION 2: VECTORS AND MATRICES

ENGR 103 – Introduction to Engineering Computing

Vectors and Matrices

Vectors and Matrices

- ***Vectors and matrices*** are used extensively in many areas of engineering, e.g.:
  - Systems of equations
  - Dynamic system modeling and analysis
  - Feedback control system design
  - Signal processing
  - Automated test and measurement
  - Data analysis and plotting
- Here, we will briefly introduce vectors and matrices
  - Matrix math – linear algebra fundamentals
  - You'll cover this in much more detail in your Linear Algebra course

Matrices

- **Matrix**
  - Array of numerical values, e.g.:

$$\mathbf{A} = \begin{bmatrix} -7 & 0 & 1 & 4 \\ 4 & -2 & 9 & 5 \\ 8 & 3 & 4 & 0 \end{bmatrix}$$

  - The variable, $\mathbf{A}$, is a ***matrix***
- An $m \times n$ matrix has m ***rows*** and n ***columns***
- These are the ***dimensions*** of the matrix
  - $\mathbf{A}$ is a 3×4 matrix

Matrix Dimensions and Indexing

- An $m \times n$ matrix:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

- Use indices to refer to individual elements of a matrix
  - a_{ij}: the element of $\mathbf{A}$ in the i^{th} row and the j^{th}column

Vectors

- ***Vectors***
  - A matrix with one dimension equal to one
  - A matrix with ***one row*** or ***one column***
- ***Row vector***
  - One row – a $1 \times n$ matrix, e.g.:

$$x = \begin{bmatrix} -9 & 1 & -4 \end{bmatrix}$$

  - A 1×3 row vector
- ***Column vector***
  - One column – an $m \times 1$ matrix, e.g.:

$$x = \begin{bmatrix} 5 \\ 1 \\ 8 \end{bmatrix}$$

  - A 3×1 column vector

Scalars

- **Scalar**
  - A 1×1 matrix
  - The numbers we are we are familiar with, e.g.:

$$b = 4, \qquad x = -3 + j5.8, \qquad y = -1 \times 10^{-9}$$

- We understand simple mathematical operations involving scalars
  - Can add, subtract, multiply, or divide any pair of scalars
  - Not true for matrices
    - Depends on the matrix dimensions

Mathematical Matrix Operations

Matrix Addition and Subtraction

- As long as matrices have the ***same dimensions***, we can add or subtract them
  - ***Addition*** and ***subtraction*** are done ***element-by-element***, and the ***resulting matrix is the same size***

$$\begin{bmatrix} 4 & 8 \\ 0 & 3 \end{bmatrix} + \begin{bmatrix} 1 & -4 \\ 6 & -1 \end{bmatrix} = \begin{bmatrix} 5 & 4 \\ 6 & 2 \end{bmatrix}$$

$$\begin{bmatrix} 4 & 8 \\ 0 & 3 \end{bmatrix} - \begin{bmatrix} 1 & -4 \\ 6 & -1 \end{bmatrix} = \begin{bmatrix} 3 & 12 \\ -6 & 4 \end{bmatrix}$$

- We can also add ***scalars*** to (or subtract from) matrices

$$\begin{bmatrix} 1 & -4 \\ 6 & -1 \end{bmatrix} + 5 = \begin{bmatrix} 6 & 1 \\ 11 & 4 \end{bmatrix}$$

Matrix Addition and Subtraction

- If matrices are not the same size, and neither is a scalar, addition/subtraction are not defined
  - The following operations cannot be done

$$\begin{bmatrix} 4 & 8 \\ 0 & 3 \end{bmatrix} + \begin{bmatrix} 1 & -4 & 6 \\ 6 & -1 & 9 \end{bmatrix} = ?$$

$$\begin{bmatrix} 8 \\ 3 \end{bmatrix} - \begin{bmatrix} 1 & -4 \\ 6 & -1 \end{bmatrix} = ?$$

- Addition is commutative (order does not matter):

$$\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A} = \mathbf{C}$$

$$\begin{bmatrix} 4 & 8 \\ 0 & 3 \end{bmatrix} + \begin{bmatrix} 1 & -4 \\ 6 & -1 \end{bmatrix} = \begin{bmatrix} 1 & -4 \\ 6 & -1 \end{bmatrix} + \begin{bmatrix} 4 & 8 \\ 0 & 3 \end{bmatrix} = \begin{bmatrix} 5 & 4 \\ 6 & 2 \end{bmatrix}$$

Matrix Multiplication

- □ In order to multiply matrices, their ***inner dimensions*** must agree
- □ We can multiply $\mathbf{A} \cdot \mathbf{B}$ only if the ***number of columns*** of $\mathbf{A}$ is equal to the ***number of rows*** of $\mathbf{B}$
- □ Resulting Matrix has same number of rows as $\mathbf{A}$ and same number of columns as $\mathbf{B}$

$$\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$$

$$\nearrow \qquad \uparrow \qquad \nwarrow$$

$$(\text{m x n}) \cdot (\text{n x p}) = (\text{m x p})$$

Matrix Multiplication – $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{np} \end{bmatrix} = \begin{bmatrix} c_{11} & \cdots & c_{1p} \\ \vdots & \ddots & \vdots \\ c_{m1} & \cdots & c_{mp} \end{bmatrix}$$

- The (i, j^{th}) entry of $\mathbf{C}$ is the ***dot product*** of the i^{th} row of $\mathbf{A}$ with the j^{th} column of $\mathbf{B}$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

- Consider the multiplication of two 2×2 matrices:

$$\begin{bmatrix} \boxed{a_{11} \quad a_{12}} \\ a_{21} \quad a_{22} \end{bmatrix} \cdot \begin{bmatrix} \boxed{\begin{matrix} b_{11} \\ b_{21} \end{matrix}} & \begin{matrix} b_{12} \\ b_{22} \end{matrix} \end{bmatrix} = \begin{bmatrix} \boxed{a_{11}b_{11} + a_{12}b_{21}} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

Matrix Multiplication – Examples

- A 2×2 and a 2×3 yield a 2×3

$$\begin{bmatrix} 1 & 4 \\ 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & -1 & 5 \\ 6 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 27 & 7 & 5 \\ 12 & 0 & 10 \end{bmatrix}$$

- A 3×3 and a 3×1 result in a 3×1

$$\begin{bmatrix} 1 & 5 & 0 \\ 0 & 4 & 8 \\ 2 & 7 & 3 \end{bmatrix} \cdot \begin{bmatrix} 6 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 11 \\ 20 \\ 25 \end{bmatrix}$$

Matrix Multiplication – Properties

- ***Matrix multiplication is not commutative***
  - Order matters
  - Unlike scalars
- In general,

$$\mathbf{A} \cdot \mathbf{B} \neq \mathbf{B} \cdot \mathbf{A}$$

- If A and/or B is not square then one of the above operations may not be possible anyway
  - Inner dimensions may not agree for both product orders

Matrix Multiplication – Properties

- ***Matrix multiplication is associative***
  - Insertion of parentheses anywhere within a product of multiple terms does not affect the result:

$$(\mathbf{A}\cdot\mathbf{B})\cdot\mathbf{C}=\mathbf{D}$$

$$\mathbf{A}\cdot(\mathbf{B}\cdot\mathbf{C})=\mathbf{D}$$

- ***Matrix multiplication is distributive***
  - Multiplication distributes over addition
  - Must maintain correct order, i.e. left- or right-multiplication

$$\mathbf{A}(\mathbf{B}+\mathbf{C})=\mathbf{AB}+\mathbf{AC}$$

$$(\mathbf{B}+\mathbf{C})\mathbf{A}=\mathbf{BA}+\mathbf{CA}$$

Identity Matrix

- Multiplication of a scalar by 1 results in that scalar

$$a \cdot 1 = 1 \cdot a = a$$

- The matrix version of 1 is the ***identity matrix***
  - Ones along the diagonal, zeros everywhere else
  - Square $(n \times n)$ matrix
  - Denoted as $\mathbf{I}$ or $\mathbf{I_n}$, where $\mathbf{n}$ is the matrix dimension, e.g.

$$\mathbf{I_3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Left- or right-multiplication by an identity matrix results in that matrix, unchanged

$$\mathbf{A} \cdot \mathbf{I} = \mathbf{I} \cdot \mathbf{A} = \mathbf{A}$$

Identity Matrix

- Right-multiplication of an $n \times n$ matrix by an $n \times n$ identity matrix, $\mathbf{I_n}$

$$\begin{bmatrix} 1 & 5 & 0 \\ 0 & 4 & 8 \\ 2 & 7 & 3 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 5 & 0 \\ 0 & 4 & 8 \\ 2 & 7 & 3 \end{bmatrix}$$

- Same result if we left-multiply by $\mathbf{I_n}$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 5 & 0 \\ 0 & 4 & 8 \\ 2 & 7 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 5 & 0 \\ 0 & 4 & 8 \\ 2 & 7 & 3 \end{bmatrix}$$

Identity Matrix

- Right-multiplication of an $m \times n$ matrix by an $n \times n$ identity matrix

$$\begin{bmatrix} 1 & 5 & 0 \\ 0 & 4 & 8 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 5 & 0 \\ 0 & 4 & 8 \end{bmatrix}$$

- Same result if we left-multiply the $m \times n$ matrix by an $m \times m$ identity matrix

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 5 & 0 \\ 0 & 4 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 5 & 0 \\ 0 & 4 & 8 \end{bmatrix}$$

Vector Multiplication

- Vectors *are* matrices, so inner dimensions must agree
- Two types of vector multiplication:
- ***Inner product*** (***dot product***)
  - Result is a scalar

$$\begin{bmatrix} a_{11} & a_{12} \end{bmatrix} \cdot \begin{bmatrix} b_{11} \\ b_{21} \end{bmatrix} = a_{11}b_{11} + a_{12}b_{21}$$

- ***Outer product***
  - Result for n-vectors is an n x n matrix

$$\begin{bmatrix} a_{11} \\ a_{21} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} \\ a_{21}b_{11} & a_{21}b_{12} \end{bmatrix}$$

Exponentiation

- As with scalars, raising a matrix to the power, n, is the multiplication of that matrix by itself n times

$$\mathbf{A}^3 = \mathbf{A} \cdot \mathbf{A} \cdot \mathbf{A}$$

- What must be true of a matrix for exponentiation to be allowable?
  - Inner matrix dimensions must agree
  - Rows of $\mathbf{A}$ must equal columns of $\mathbf{A}$ – n x n
  - ***Matrix must be square***

Matrix 'Division' – Multiplication by the Inverse

- Scalar division that we are accustomed to can be thought of as multiplication by an inverse:

$$a \div b = a \cdot \frac{1}{b} = a \cdot b^{-1}$$

- This is how we 'divide' matrices as well

$$\mathbf{A} \cdot \mathbf{B} \cdot \mathbf{B}^{-\mathbf{1}} = \mathbf{A}$$

- Multiplication of a scalar by its inverse is equal to 1.
  - For a matrix, the result is the ***identity matrix***

$$\mathbf{A} \cdot \mathbf{A}^{-\mathbf{1}} = \mathbf{I} = \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix}$$

Matrix Inverse

- Recall that matrix multiplication is not commutative
  - ***Right-*** and ***left-multiplication*** are different operations

$$\mathbf{A}\cdot\mathbf{B}\cdot\mathbf{B}^{-1}=\mathbf{A}\neq\mathbf{B}^{-1}\cdot\mathbf{A}\cdot\mathbf{B}$$

- The inverse does not exist for all matrices
  - ***Non-invertible*** matrices are referred to as ***singular***
  - Matrix must be ***square*** for its inverse to exist

Matrix Inverse

- Possible to calculate matrix inverses by hand
  - Simple for small matrices
  - Quickly becomes tedious as matrices get larger
- For example, the inverse of a 2 × 2 matrix:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

- For example:

$$\mathbf{A} = \begin{bmatrix} 2 & 5 \\ 2 & 4 \end{bmatrix}$$

$$\mathbf{A}^{-\mathbf{1}} = \frac{1}{8 - 10} \begin{bmatrix} 4 & -5 \\ -2 & 2 \end{bmatrix} = \begin{bmatrix} -2 & 2.5 \\ 1 & -1 \end{bmatrix}$$

Matrix Inverse - Example

- Multiplication of a matrix by its inverse yields the identity matrix
- For example:

$$\mathbf{A}\cdot\mathbf{A}^{-1}=\begin{bmatrix}2 & 5\\ 2 & 4\end{bmatrix}\cdot\begin{bmatrix}-2 & 2.5\\ 1 & -1\end{bmatrix}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$$

- Or, for a 3×3:

$$\mathbf{A}=\begin{bmatrix}2 & 0 & 2\\ 0 & 1 & 2\\ 0 & 0 & 2\end{bmatrix},\quad \mathbf{A}^{-1}=\begin{bmatrix}0.5 & 0 & -0.5\\ 0 & 1 & -1\\ 0 & 0 & 0.5\end{bmatrix}$$

$$\begin{bmatrix}2 & 0 & 2\\ 0 & 1 & 2\\ 0 & 0 & 2\end{bmatrix}\cdot\begin{bmatrix}0.5 & 0 & -0.5\\ 0 & 1 & -1\\ 0 & 0 & 0.5\end{bmatrix}=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

- You’ll learn more about this in Linear Algebra – not critical here

Matrix Transpose

- The ***transpose*** of a matrix is that matrix with ***rows and columns swapped***
  - First row becomes the first column, second row becomes the second column, and so on
- For example:

$$\mathbf{A} = \begin{bmatrix} 0 & 9 \\ 2 & 7 \\ 6 & 3 \end{bmatrix} \quad \mathbf{A}^{\mathrm{T}} = \begin{bmatrix} 0 & 2 & 6 \\ 9 & 7 & 3 \end{bmatrix}$$

- Row vectors become column vectors and vice versa

$$\mathbf{x} = \begin{bmatrix} 7 \\ -1 \\ -4 \end{bmatrix} \qquad \mathbf{x}^{\mathrm{T}} = \begin{bmatrix} 7 & -1 & -4 \end{bmatrix}$$

Why Do We Use Matrices?

- Vectors and matrices are used extensively in many engineering fields, for example:
  - Modeling, analysis, and design of dynamic systems
  - Controls engineering
  - Image processing
  - Etc. …
- Very common usage of vectors and matrices is to represent ***systems of equations***
  - These regularly occur in *all* fields of engineering

Systems of Equations

- Consider a system of three equations with three unknowns:

$$3x_1 + 5x_2 - 9x_3 = 6$$
$$-3x_1 + 7x_3 = -2$$
$$-x_2 + 4x_3 = 8$$

- Can represent this in ***matrix form***:

$$\begin{bmatrix} 3 & 5 & -9 \\ -3 & 0 & 7 \\ 0 & -1 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ -2 \\ 8 \end{bmatrix}$$

- Or, more compactly as:

$$\mathbf{Ax} = \mathbf{b}$$

- Perform algebra operations as we would if $\mathbf{A}$, $\mathbf{x}$, and $\mathbf{b}$ were scalars
  - Observing matrix-specific rules, e.g. multiplication order, etc.

Matrix Multiplication

EXERCISE

If $\mathbf{A} = \begin{bmatrix} 2 & 3 \\ 1 & -5 \\ 4 & 1 \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} 4 & 3 & 6 \\ 1 & -2 & 3 \end{bmatrix}$ find (a) the size of C when $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$ and (b) the value of $\mathbf{C}_{22}$.

Systems of Equations

EXERCISE

Determine the values of x_1 and x_2 if

$$4x_1 + x_2 = 7$$
$$-x_1 + 5x_2 = -7$$

Step 1: express this system of equations in matrix form $\mathbf{Ax} = \mathbf{b}$

Systems of Equations

EXERCISE

Determine the values of x_1 and x_2 if

$$\begin{aligned} 4x_1 + x_2 &= 7 \\ -x_1 + 5x_2 &= -7 \end{aligned}$$

Step 2: find $\mathbf{A}^{-1}$

Systems of Equations

EXERCISE

Determine the values of x_1 and x_2 if

$$\begin{aligned} 4x_1 + x_2 &= 7 \\ -x_1 + 5x_2 &= -7 \end{aligned}$$

Step 3: If you multiply $\mathbf{A}$ by $\mathbf{A}^{-1}$ ($\mathbf{A}^{-1}\mathbf{A}$), what do you get?

Step 4: Find the values x by multiplying both sides of $\mathbf{Ax} = \mathbf{b}$ by $\mathbf{A}^{-1}$

Vectors & Matrices in Python

NumPy

- Python, itself, does not have a built-in data type for matrices
  - Lists are like vectors
  - Lists of lists are like matrices
  - But, cannot operate on them like we would like to operate on vectors and matrices
- Instead, we will use the ***NumPy*** package when working with matrices

NumPy

- We will use the NumPy (**Num**erical **Py**thon) package extensively
- Fundamental data type:
  - Multi-dimensional array object – **`ndarray`**
    - These are ***matrices***
    - Useful for engineering computation
- Many built-in functions
  - Mathematical operations, e.g.:
    - Trigonometric functions
    - Exponents and logarithms
    - Complex number operations
  - Array creation and manipulation routines
  - Polynomial creation, manipulation, fitting, etc.
  - Much more …

Defining Vectors and Matrices – `np.array()`

- Let’s say we want to assign the following matrix variable in Python:

$$\mathbf{A} = \begin{bmatrix} 2 & 5 & 1 \\ -4 & 6 & 0 \end{bmatrix}$$

- Use NumPy's `array()` function

```
np.array(object)
```

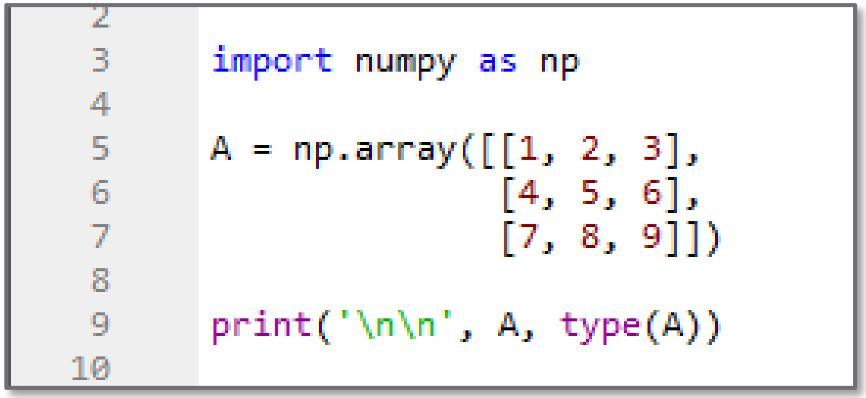
  - *`object`*: the array data – a nested list – one list for each row

- For example:

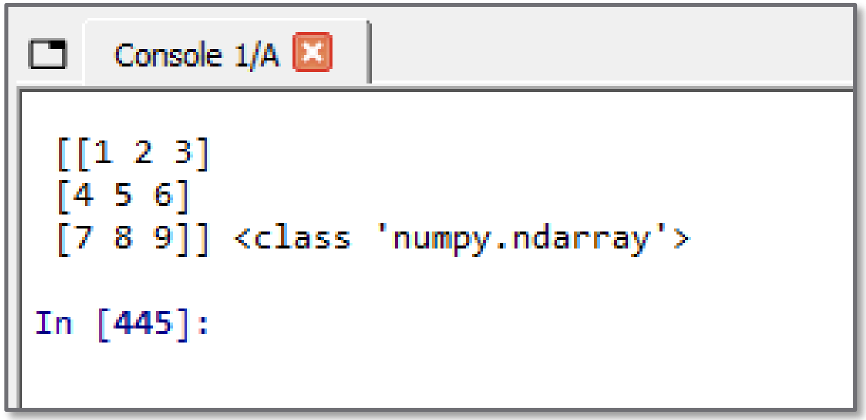
```
A = np.array([[2, 5, 1], [-4, 6, 0]])
```

Line Continuation

- You can continue a single Python command across multiple lines
  - Improves readability
- Useful when explicitly defining `ndarrays`
  - Indent continued lines to align leading delimiters (i.e. square brackets)



```python
import numpy as np

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

print('\n\n', A, type(A))
```



```

[[1 2 3]
 [4 5 6]
 [7 8 9]] <class 'numpy.ndarray'>

In [445]:
```

Vector and Matrix Generation

- Often want to automatically generate vectors and matrices without having to enter them element-by-element

- A few of NumPy’s ***array-generation*** functions:
  - `arange()`
  - `linspace()`
  - `logspace()`
  - `ones()`
  - `zeros()`
  - `empty()`
  - `diag()`
  - `eye()`

Vector Generation – arange( )

- Create vector of evenly-spaced values
  - Values are on ***half-open interval***: `[start, stop)`

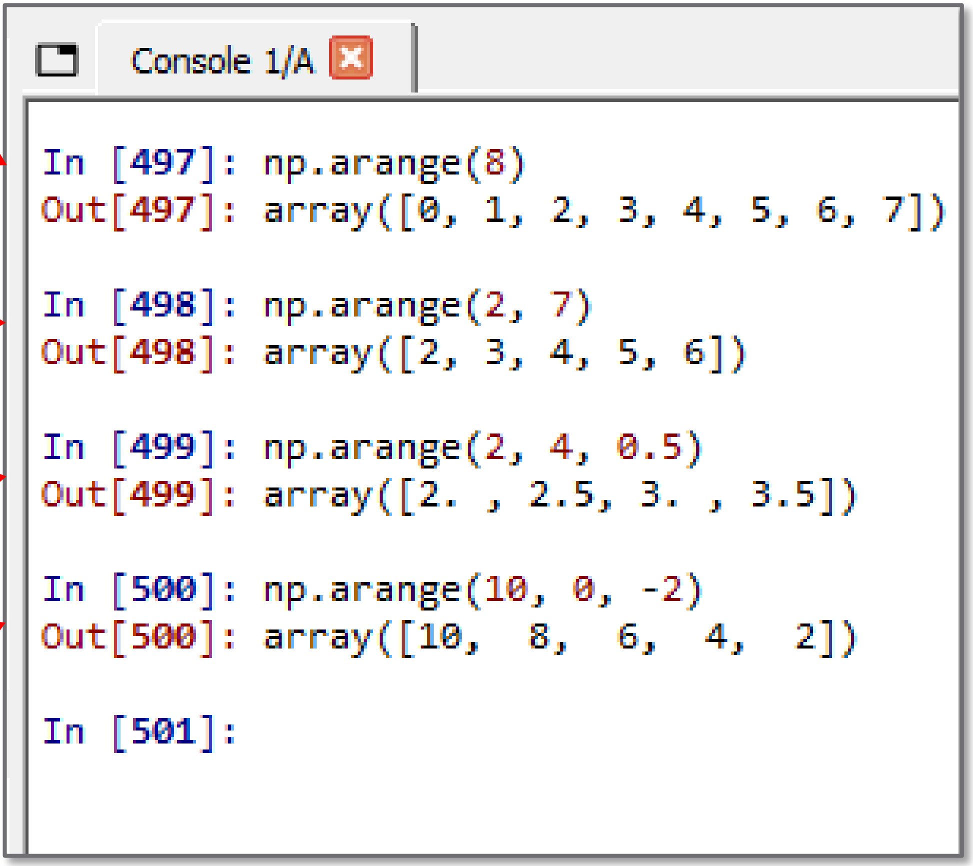
```
x = np.arange(start, stop, step)
```

  - `start`: *optional* start of interval – default: `0`
  - `stop`: end of interval
  - `step`: *optional* increment value – default: `1`
  - `x`: resulting vector of points

- Half-open interval: `[start, stop)`
  - `start` *is* the first value in x
  - `stop` is *not* the last value in x

Vector Generation – arange( )

- □ Default start is 0, default step is 1
- □ Specify start and stop
- □ Specify start, stop, and step
- □ step may be negative



```
In [497]: np.arange(8)
Out[497]: array([0, 1, 2, 3, 4, 5, 6, 7])

In [498]: np.arange(2, 7)
Out[498]: array([2, 3, 4, 5, 6])

In [499]: np.arange(2, 4, 0.5)
Out[499]: array([2. , 2.5, 3. , 3.5])

In [500]: np.arange(10, 0, -2)
Out[500]: array([10,  8,  6,  4,  2])

In [501]:
```

Vector Generation – `linspace()`

```
x = np.linspace(start,stop,N)
```

- `start`: first element in the vector
- `stop`: last element in the vector
- `N`: *optional* number of elements – default: `50`
- `x`: resulting vector of linearly spaced points

- `arange()`:
  - `stop` is *not* in `x`
  - Number of points not directly specified
- `linspace()`:
  - `stop` *is* the last value in `x`
  - Increment value not directly specified

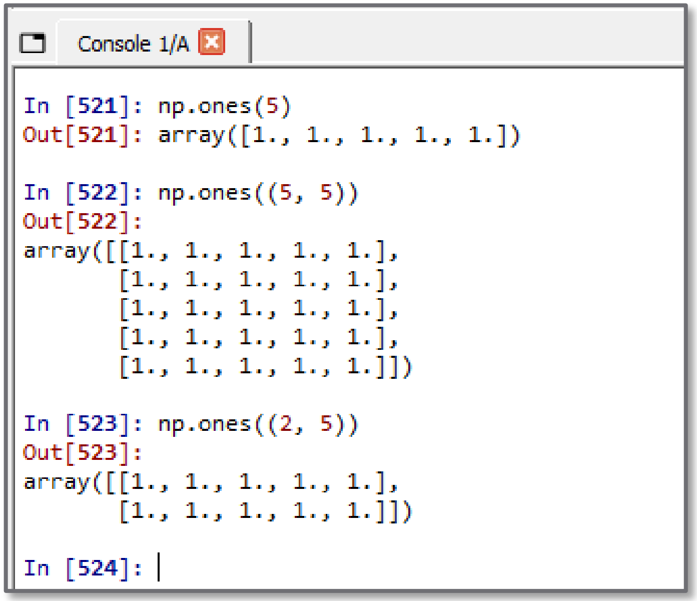
Array Generation – ones(), zeros()

- Generate an N-vector of all 1's or all 0's:

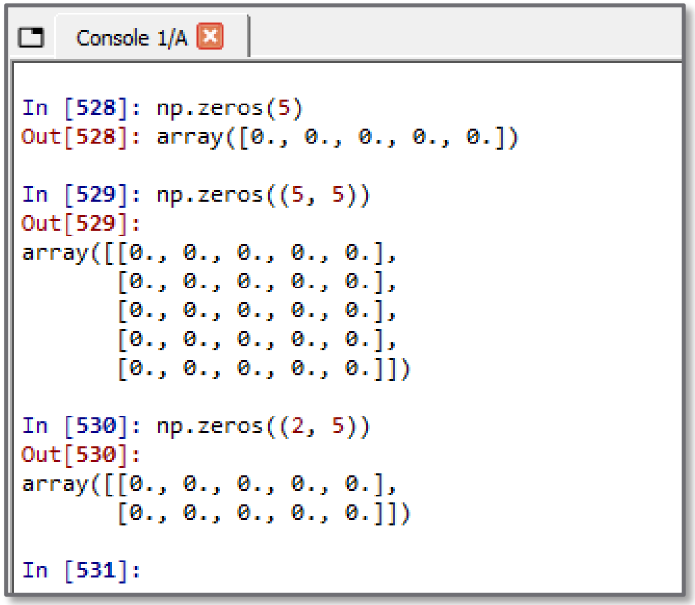
```
A = np.ones(N)   or   A = np.zeros(N)
```

- Generate an $m \times n$ matrix of all 1's or 0's

```
A = np.ones((m,n))   or   A = np.zeros((m,n))
```



```
In [521]: np.ones(5)
Out[521]: array([1., 1., 1., 1., 1.])

In [522]: np.ones((5, 5))
Out[522]:
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])

In [523]: np.ones((2, 5))
Out[523]:
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])

In [524]:
```

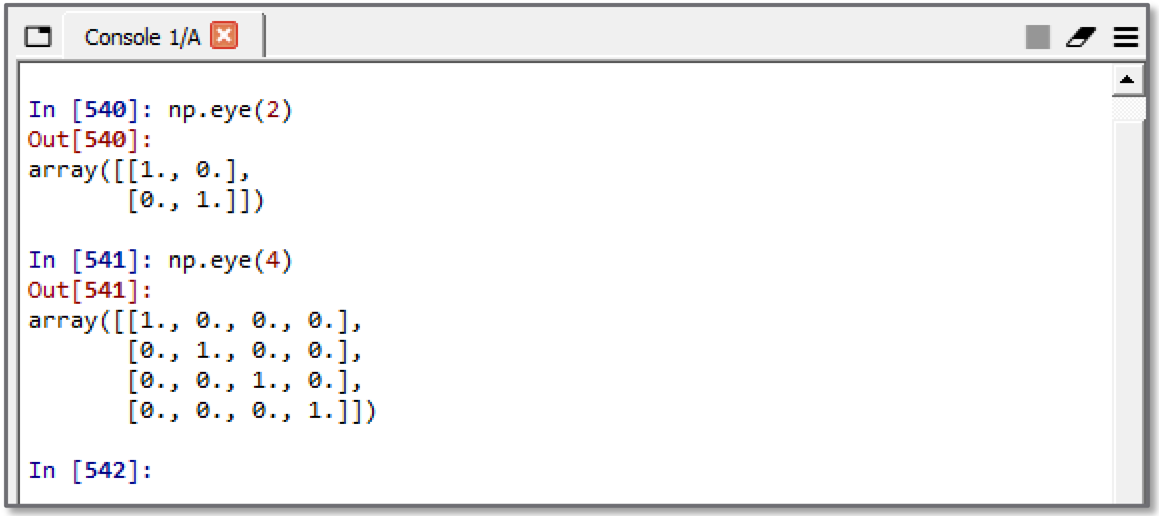


```
In [528]: np.zeros(5)
Out[528]: array([0., 0., 0., 0., 0.])

In [529]: np.zeros((5, 5))
Out[529]:
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])

In [530]: np.zeros((2, 5))
Out[530]:
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])

In [531]:
```

Identity Matrix – eye( )

```
I = np.eye(N)
```

- N: identity matrix dimension
- `I`: $N \times N$ identity matrix



```
In [540]: np.eye(2)
Out[540]:
array([[1., 0.],
       [0., 1.]])

In [541]: np.eye(4)
Out[541]:
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])

In [542]:
```

Random Number Generation – `default_rng()`

- Very often useful to generate ***random numbers***
  - Simulating the effect of noise
  - Monte Carlo simulation, etc.

- First, construct a random-number generator object:

```
rng = np.random.default_rng(seed)
```

  - `seed`: *optional* initialization seed for generator
  - `rng`: initialized generator object – will run methods on this object to generate random numbers

Normally-Distributed Random Numbers

- Generate random values from a normal (Gaussian) distribution

```
x = rng.normal(loc=0, scale=1, size=1)
```

  - `rng`: generator object created with `default_rng()`
  - `loc`: *optional* mean of distribution – default: 0.0
  - `scale`: *optional* standard deviation – default: 1.0
  - `size`: *optional* dimension of resulting array
  - `x`: resulting array of random values

- Note that `normal()` is a method that operates on the random-number generator object, rng

Uniformly-Distributed Random Numbers

- Generate random values from a uniform distribution on the interval `[low, high)`

```
x = rng.uniform(low=0, high=1, size=1)
```

  - `rng`: generator object created with `default_rng()`
  - `low`: *optional* lower bound of interval – default: 0.0
  - `high`: *optional* upper bound of interval – default: 1.0
  - `size`: *optional* dimension of resulting array – default: 1
  - `x`: resulting array of random values

- Half-open interval:
  - Resulting values are ≥ low and < high

Uniformly-Distributed Random Integers

- Generate random values from a uniform distribution on the interval `[low, high)`

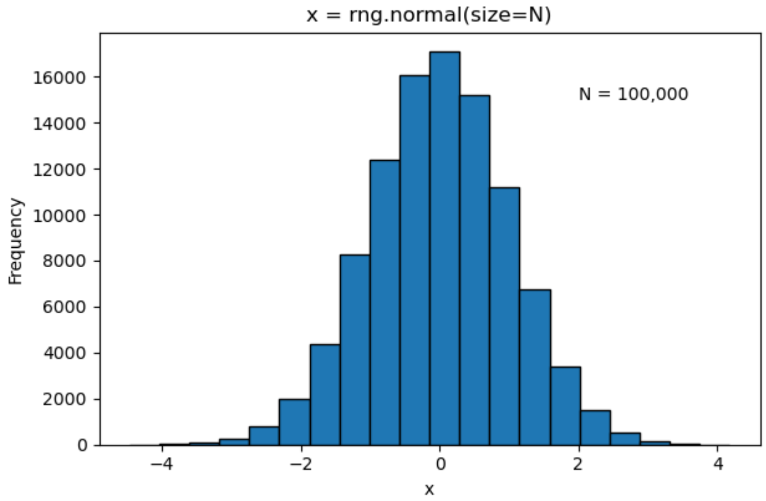
```
x = rng.integers(low, high, size=1)
```

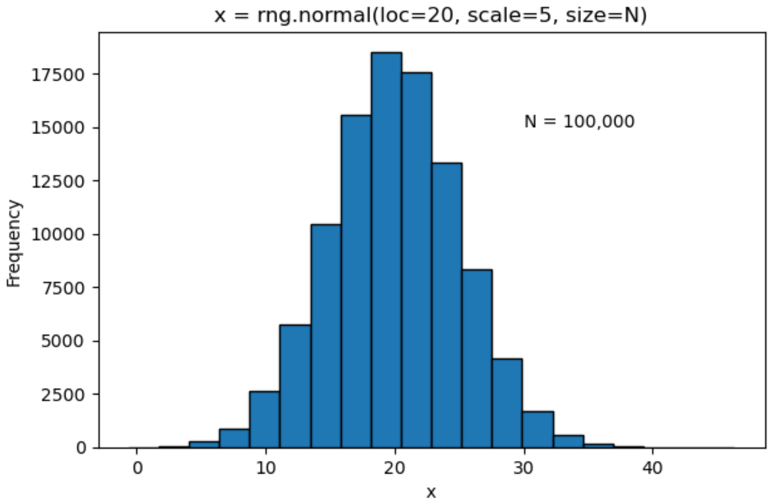
  - `rng`: generator object created with `default_rng()`
  - `low`: minimum possible resulting integer
  - `high`: *one more than* the maximum possible integer
  - `size`: *optional* dimension of resulting array – default: 1
  - `x`: resulting array of random integers

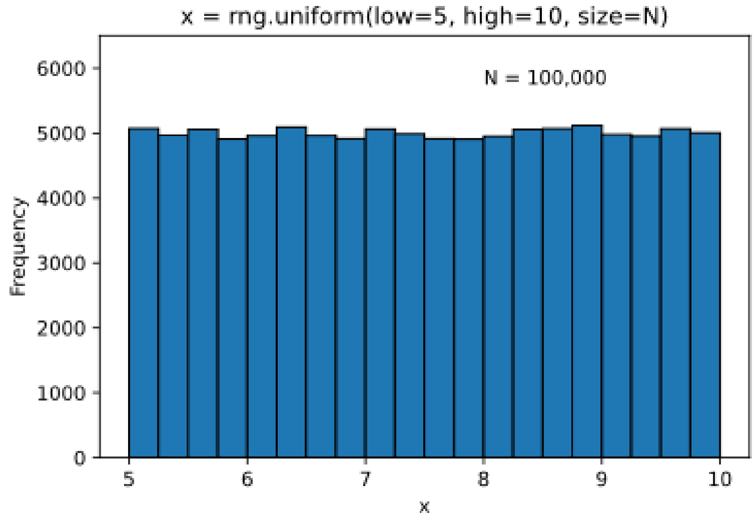
- Or

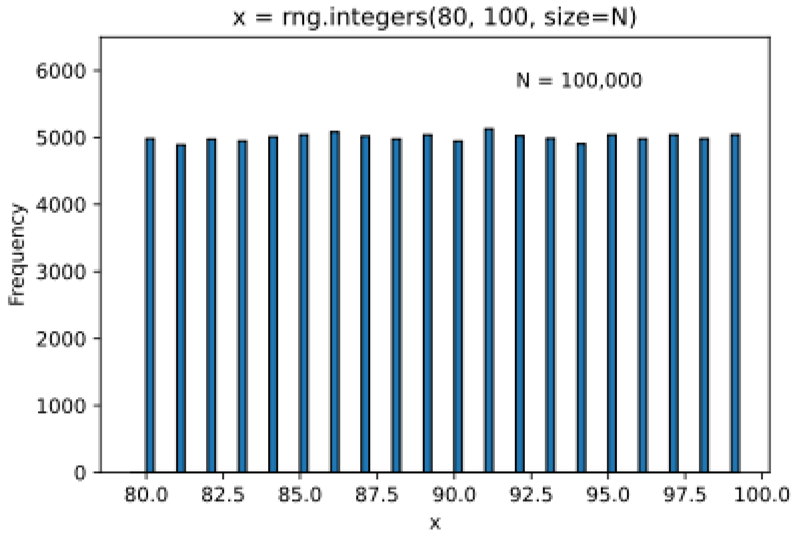
```
x = rng.integers(high, size)

x = rng.integers(high)
```

Random Numbers – Examples









Array Indexing and Slicing

Array Indexing

- We've seen how we can refer to specific elements in an array by their ***row, column indices***, a_{ij}:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

- Python allows us to do the same thing
  - Indices specified in square brackets immediately following the array variable name
  - ***Numbering begins at 0***
  - Applies to any Python ***iterable***: `list`, `str`, `tuple`, `dict`, `ndarray`, …
- For example:
  - `B[1,4]`: element in the 2nd row, 5th column of the array B

Array Indexing – Vectors, Lists, Tuples …

- Consider a 1-dimensional array, or vector
  - Two indexing methods:
    - Positive indexing
    - Negative indexing

Positive Index:	0	1	2	3	4	5	
x =	[1,	3,	5,	7,	9,	11]	
	-6	-5	-4	-3	-2	-1	:Negative Index

```
>>> x[0]
1
>>> x[3]
7
```

```
>>> x[-1]
11
>>> x[-4]
5
```

Array Indexing – ndarray

- Pass row and column indices to index ndarrays
  - In square brackets, separated by commas
  - Positive or negative indexing

$$A = \begin{array}{c|ccc|c}
\text{Positive Index:} & 0 & 1 & 2 & \\
\hline
0 & 1 & 2 & 3 & -3 \\
1 & 4 & 5 & 6 & -2 \\
2 & 7 & 8 & 9 & -1 \\
\hline
 & -3 & -2 & -1 & \text{:Negative Index}
\end{array}$$

```
>>> A[0,1]
2
```

```
>>> A[1,-2]
5
```

Array Slicing

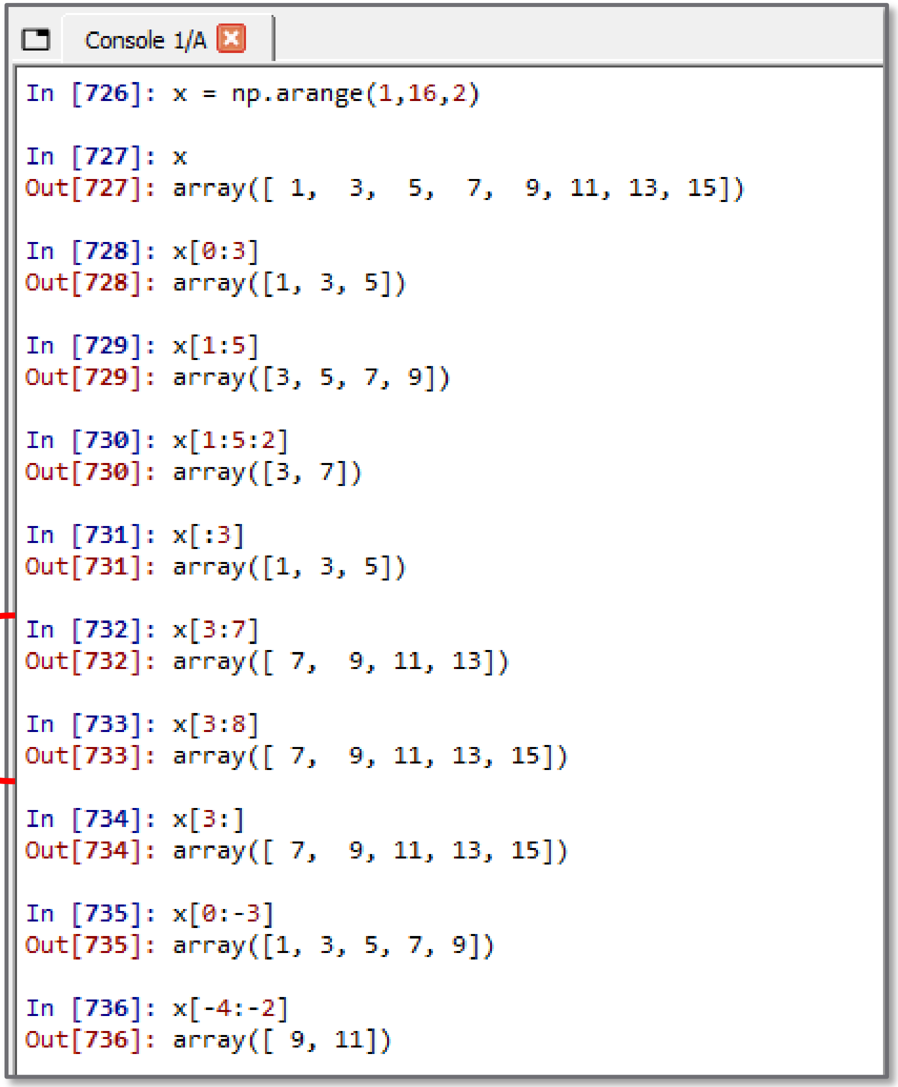
- ***Slicing***
  - Access a range of values within a Python iterable, or NumPy ndarray
- Slicing index syntax:

```
[start:stop:step]
```

  - `start`: index of the first value to access – default: `0`
  - `stop`: *one past* the index of the last value – default: `-1`
  - `step`: index increment value – default: `1`
- For example:
  - `x[1:4]` refers to the 2nd through 4th elements of x

Array Slicing

- First index is `0`
- `stop` (here, `x[5]`) is not included
- Increment by `step`
- Default `start` is `0`
- Index to `x[8]` to get last element at `x[7]`
- Omit `stop` to index through the end
- Negative indexing



```
In [726]: x = np.arange(1,16,2)

In [727]: x
Out[727]: array([ 1,  3,  5,  7,  9, 11, 13, 15])

In [728]: x[0:3]
Out[728]: array([1, 3, 5])

In [729]: x[1:5]
Out[729]: array([3, 5, 7, 9])

In [730]: x[1:5:2]
Out[730]: array([3, 7])

In [731]: x[:3]
Out[731]: array([1, 3, 5])

In [732]: x[3:7]
Out[732]: array([ 7,  9, 11, 13])

In [733]: x[3:8]
Out[733]: array([ 7,  9, 11, 13, 15])

In [734]: x[3:]
Out[734]: array([ 7,  9, 11, 13, 15])

In [735]: x[0:-3]
Out[735]: array([1, 3, 5, 7, 9])

In [736]: x[-4:-2]
Out[736]: array([ 9, 11])
```

Array Slicing – `ndarray`

- Can extend all slicing concepts to ***multi-dimensional arrays, or matrices***
  - Access a multi-dimensional range of values from within a NumPy ndarray
  - Add an index range for each dimension
- For a 2-D array, or matrix:

```
[r_start:r_stop:r_step, c_start:c_stop:c_step]
```

  - `r_start`, `r_stop`, and `r_step`: ***row range***
  - `c_start`, `c_stop`, and `r_step`: ***column range***

- For example, B`[0:3,1:4]` refers elements of B in the
  - 1st through 3rd row (rows 0, 1, and 2)
  - 2nd through 4th column (columns 1, 2, and 3)

Array Slicing – ndarray

$$B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

- B[0:2,0:2]

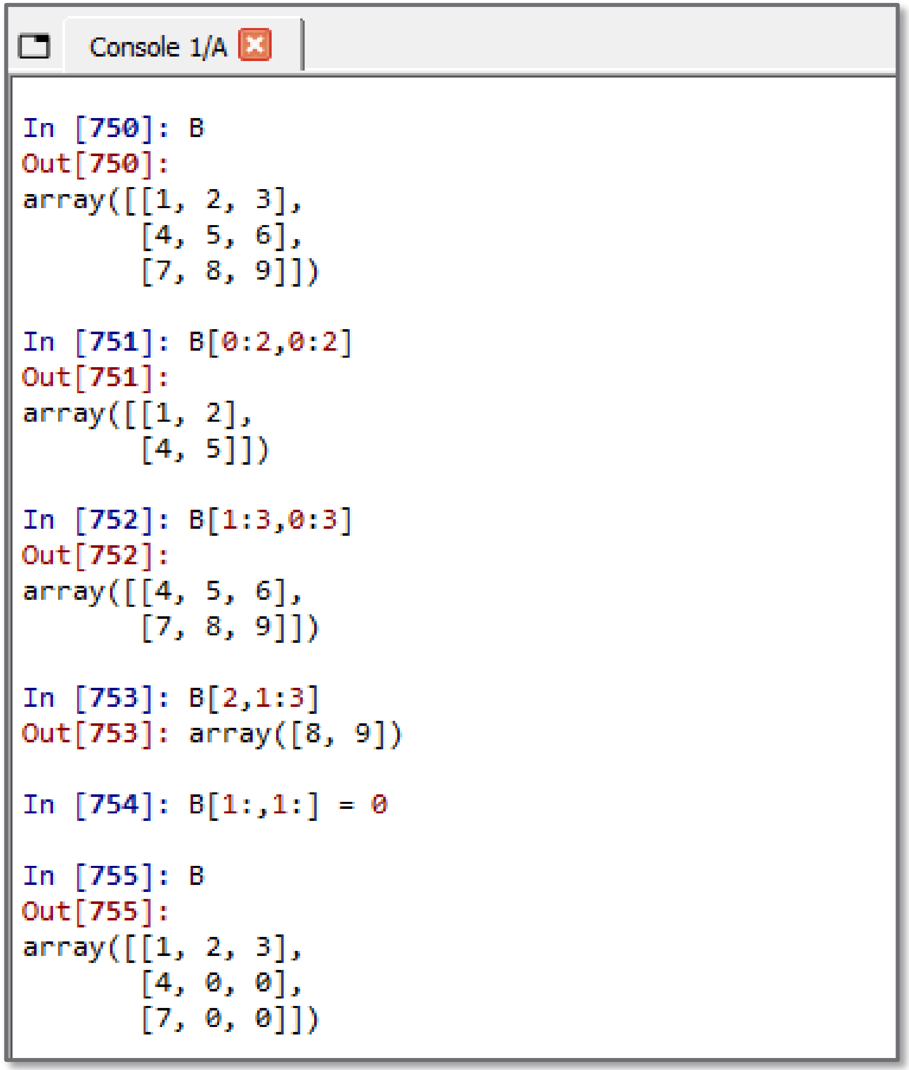
  $$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}$$

- B[1:3,0:3]

  $$\begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

- B[2,1:3]

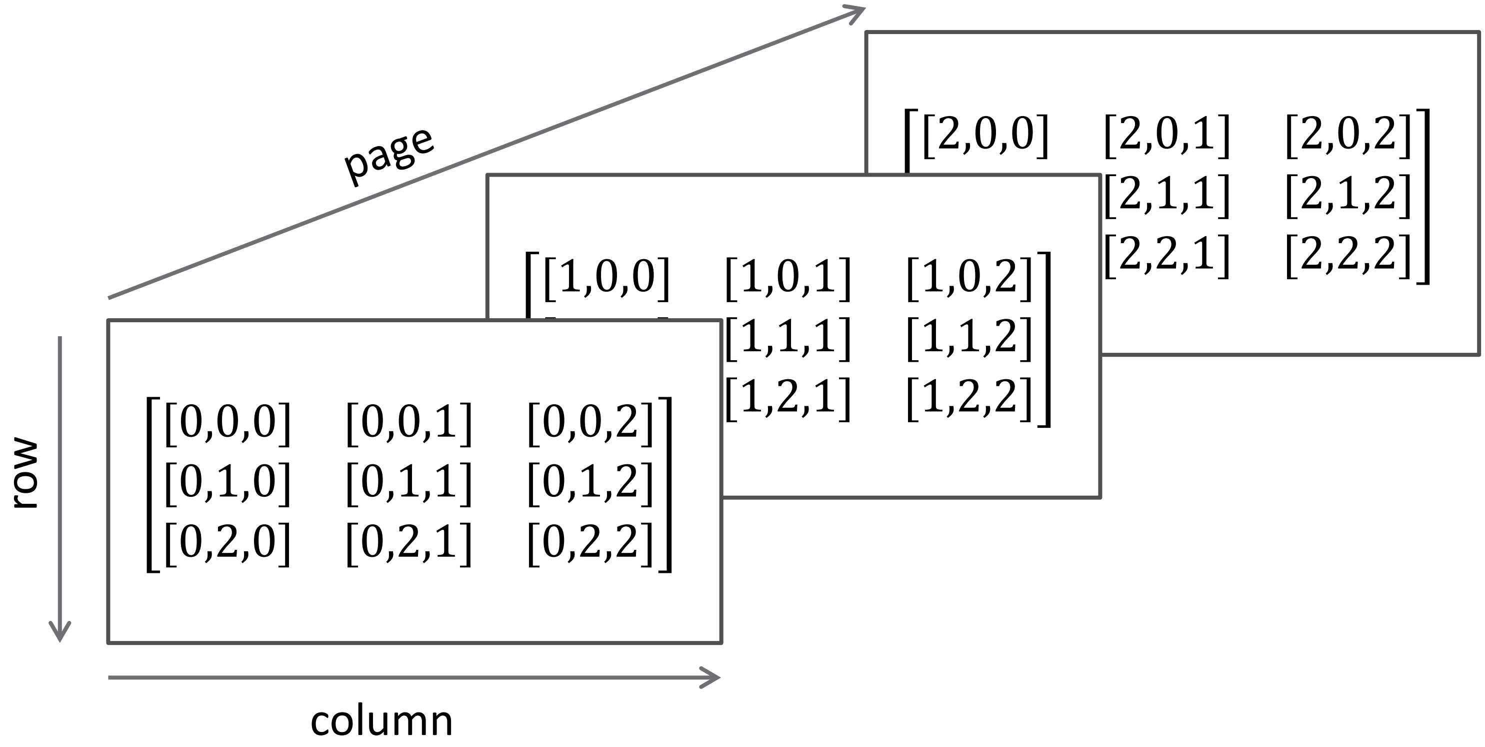
  $$\begin{bmatrix} 8 & 9 \end{bmatrix}$$



```
In [750]: B
Out[750]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [751]: B[0:2,0:2]
Out[751]:
array([[1, 2],
       [4, 5]])

In [752]: B[1:3,0:3]
Out[752]:
array([[4, 5, 6],
       [7, 8, 9]])

In [753]: B[2,1:3]
Out[753]: array([8, 9])

In [754]: B[1:,1:] = 0

In [755]: B
Out[755]:
array([[1, 2, 3],
       [4, 0, 0],
       [7, 0, 0]])
```

Multidimensional Arrays

- NumPy allows for the definition of arrays with more than two dimensions
  - Arbitrary number of dimensions allowed
  - Three dimensional arrays are common
  - Index an N-dimensional array with N indices
- For example, a $3 \times 3 \times 3$ array looks like this:



$$\begin{bmatrix} [2,0,0] & [2,0,1] & [2,0,2] \\ & [2,1,1] & [2,1,2] \\ & [2,2,1] & [2,2,2] \end{bmatrix}$$

$$\begin{bmatrix} [1,0,0] & [1,0,1] & [1,0,2] \\ & [1,1,1] & [1,1,2] \\ & [1,2,1] & [1,2,2] \end{bmatrix}$$

$$\begin{bmatrix} [0,0,0] & [0,0,1] & [0,0,2] \\ [0,1,0] & [0,1,1] & [0,1,2] \\ [0,2,0] & [0,2,1] & [0,2,2] \end{bmatrix}$$

Multidimensional Arrays – Indexing

- Indices for additional dimensions are ***prepended*** to the index list:
  - 1-D array (vector):

    ```
    x[index]
    ```

    $$\left[[0], [1], [2], \dots, x[N-1]\right]$$

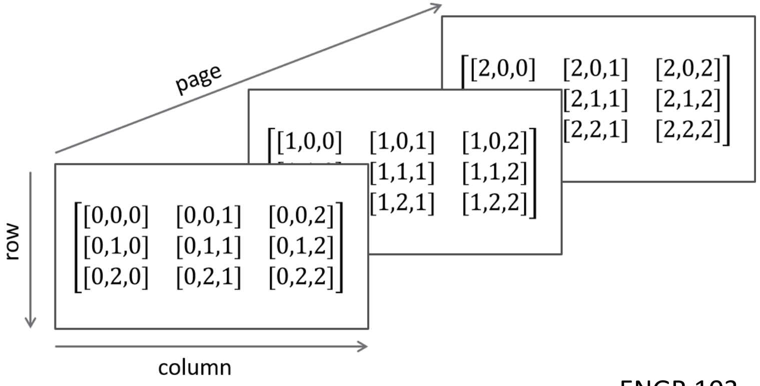
  - 2-D array (matrix):

    ```
    A[row, col]
    ```

    $$\begin{bmatrix} [0,0] & \cdots & [0, N-1] \\ \vdots & \ddots & \vdots \\ [N-1,0] & \cdots & [N-1, N-1] \end{bmatrix}$$

  - 3-D array

    ```
    B[page, row, col]
    ```



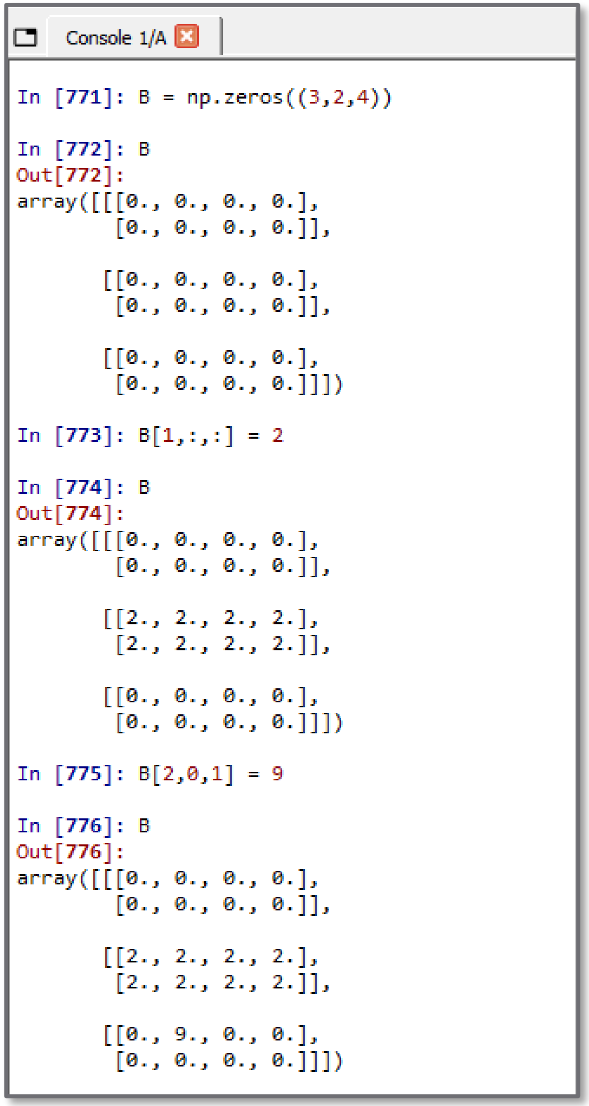
$$\begin{bmatrix} [2,0,0] & [2,0,1] & [2,0,2] \\ & [2,1,1] & [2,1,2] \\ & [2,2,1] & [2,2,2] \end{bmatrix}$$

$$\begin{bmatrix} [1,0,0] & [1,0,1] & [1,0,2] \\ & [1,1,1] & [1,1,2] \\ & [1,2,1] & [1,2,2] \end{bmatrix}$$

$$\begin{bmatrix} [0,0,0] & [0,0,1] & [0,0,2] \\ [0,1,0] & [0,1,1] & [0,1,2] \\ [0,2,0] & [0,2,1] & [0,2,2] \end{bmatrix}$$

Multidimensional Arrays – Indexing

- Create a 3-D array of zeros
  - 3 pages, 2 rows, 4 columns
- Set the 2nd page, all rows, all columns equal to 2
- Set the element on the third page, 1st row, 2nd column to 9



```
In [771]: B = np.zeros((3,2,4))

In [772]: B
Out[772]:
array([[[0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.]]])

In [773]: B[1,:,:] = 2

In [774]: B
Out[774]:
array([[[0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[2., 2., 2., 2.],
        [2., 2., 2., 2.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.]]])

In [775]: B[2,0,1] = 9

In [776]: B
Out[776]:
array([[[0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[2., 2., 2., 2.],
        [2., 2., 2., 2.]],

       [[0., 9., 0., 0.],
        [0., 0., 0., 0.]]])
```

Array Dimensions – `len()`, `shape()`, `size()`

- **Length of a vector**
  - Built-in Python function
  - Returns an integer

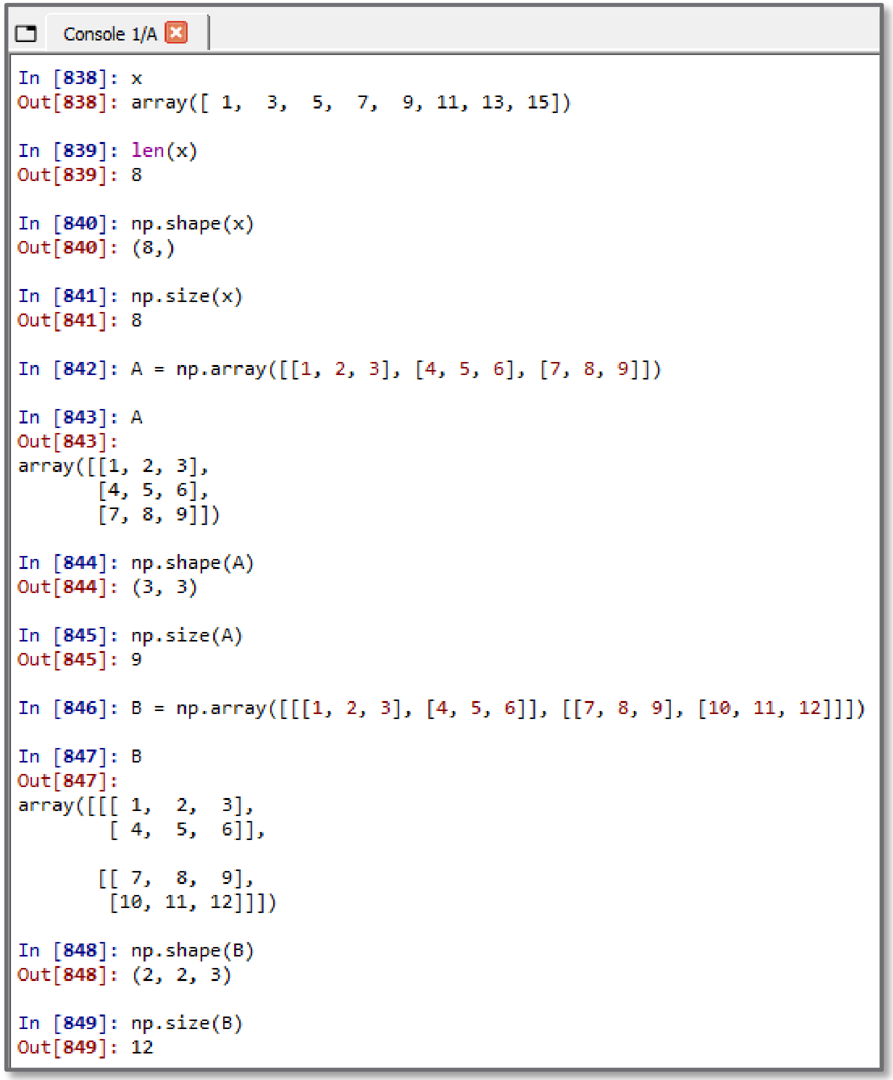
```
len(x)
```

- **Dimensions of an array**
  - Tuple: (…, pages, rows, cols)
  - NumPy function

```
np.shape(A)
```

- **Number of elements in an array**
  - Integer: product of dimensions
  - NumPy function

```
np.size(B)
```



```
In [838]: x
Out[838]: array([ 1,  3,  5,  7,  9, 11, 13, 15])

In [839]: len(x)
Out[839]: 8

In [840]: np.shape(x)
Out[840]: (8,)

In [841]: np.size(x)
Out[841]: 8

In [842]: A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [843]: A
Out[843]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [844]: np.shape(A)
Out[844]: (3, 3)

In [845]: np.size(A)
Out[845]: 9

In [846]: B = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

In [847]: B
Out[847]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])

In [848]: np.shape(B)
Out[848]: (2, 2, 3)

In [849]: np.size(B)
Out[849]: 12
```

Matrix & Array Operations

Matrix & Array Operations

- Python/NumPy operations and functions can operate on arrays
  - Element-by-element (array operations) by default
  - Special operators for matrix math
- For example:
  - Addition:

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}+\begin{bmatrix}5 & 6\\7 & 8\end{bmatrix}=\begin{bmatrix}(1+5) & (2+6)\\(3+7) & (4+8)\end{bmatrix}=\begin{bmatrix}3 & 8\\10 & 12\end{bmatrix}$$

  - Multiplication:

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}*\begin{bmatrix}5 & 6\\7 & 8\end{bmatrix}=\begin{bmatrix}(1*5) & (2*6)\\(3*7) & (4*8)\end{bmatrix}=\begin{bmatrix}2 & 12\\21 & 32\end{bmatrix}$$

  - Note, this is ***not matrix multiplication***

Array Operations

- More array operations:
  - Division:

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}/\begin{bmatrix}5 & 6\\7 & 8\end{bmatrix}=\begin{bmatrix}(1/5) & (2/6)\\(3/7) & (4/8)\end{bmatrix}=\begin{bmatrix}0.2 & 0.333\\0.429 & 32\end{bmatrix}$$

  - Exponentiation:

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} ** \ 3=\begin{bmatrix}(1 ** 3) & (2 ** 3)\\(3 ** 3) & (4 ** 3)\end{bmatrix}=\begin{bmatrix}1 & 8\\27 & 64\end{bmatrix}$$

Matrix Operations

- ***Vector multiplication***:
  - Use the NumPy @ operator

$$[1 \quad 2]@[3 \quad 4] = (1 * 3) + (2 * 4) = 11$$

  - Note that 1-D ndarrays are neither row nor column vectors
  - For vectors (1-D ndarrays), @ performs an ***inner product***:

$$[1 \quad 2]@[3 \quad 4] = [1 \quad 2] * \begin{bmatrix} 3 \\ 4 \end{bmatrix} (1 * 3) + (2 * 4) = 11$$

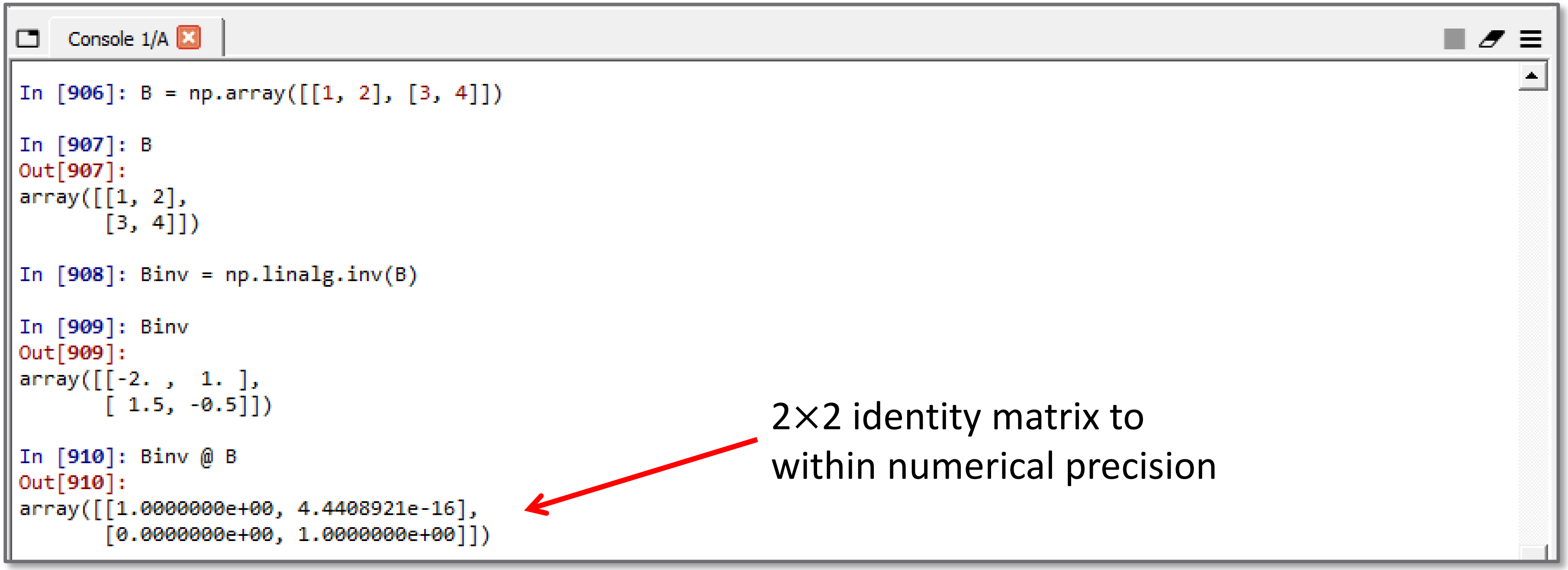
- ***Matrix multiplication***:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} @ \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} (1 * 5 + 2 * 7) & (1 * 6 + 2 * 8) \\ (3 * 5 + 4 * 7) & (3 * 6 + 4 * 8) \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Matrix Operations

- ***Matrix inverse***
  - Use NumPy's `linalg` module:

```
np.linalg.inv(A)
```

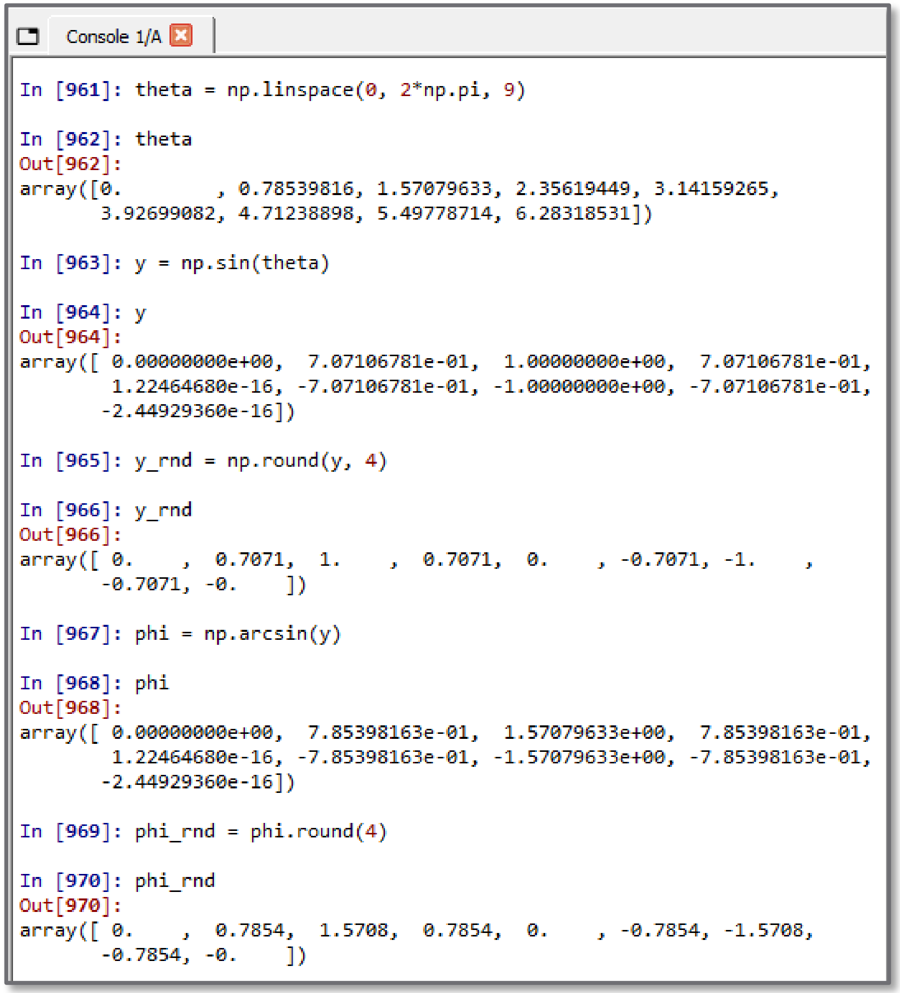


```
In [906]: B = np.array([[1, 2], [3, 4]])

In [907]: B
Out[907]:
array([[1, 2],
       [3, 4]])

In [908]: Binv = np.linalg.inv(B)

In [909]: Binv
Out[909]:
array([[-2. ,  1. ],
       [ 1.5, -0.5]])

In [910]: Binv @ B
Out[910]:
array([[1.0000000e+00, 4.4408921e-16],
       [0.0000000e+00, 1.0000000e+00]])
```

2×2 identity matrix to within numerical precision

Passing Arrays to Functions

- Can pass arrays to most functions, just as we would a scalar
- The sine of a vector of angles calculated all at once
  - No need to pass one-at-a-time
  - Result is a vector of the same size
- y passed as an input to the function round( )
- round( ) run as a ***method*** applied to the ndarray ***object***, phi



```
In [961]: theta = np.linspace(0, 2*np.pi, 9)

In [962]: theta
Out[962]:
array([0.        , 0.78539816, 1.57079633, 2.35619449, 3.14159265,
       3.92699082, 4.71238898, 5.49778714, 6.28318531])

In [963]: y = np.sin(theta)

In [964]: y
Out[964]:
array([ 0.00000000e+00,  7.07106781e-01,  1.00000000e+00,  7.07106781e-01,
        1.22464680e-16, -7.07106781e-01, -1.00000000e+00, -7.07106781e-01,
       -2.44929360e-16])

In [965]: y_rnd = np.round(y, 4)

In [966]: y_rnd
Out[966]:
array([ 0.    ,  0.7071,  1.    ,  0.7071,  0.    , -0.7071, -1.    ,
       -0.7071, -0.    ])

In [967]: phi = np.arcsin(y)

In [968]: phi
Out[968]:
array([ 0.00000000e+00,  7.85398163e-01,  1.57079633e+00,  7.85398163e-01,
        1.22464680e-16, -7.85398163e-01, -1.57079633e+00, -7.85398163e-01,
       -2.44929360e-16])

In [969]: phi_rnd = phi.round(4)

In [970]: phi_rnd
Out[970]:
array([ 0.    ,  0.7854,  1.5708,  0.7854,  0.    , -0.7854, -1.5708,
       -0.7854, -0.    ])
```