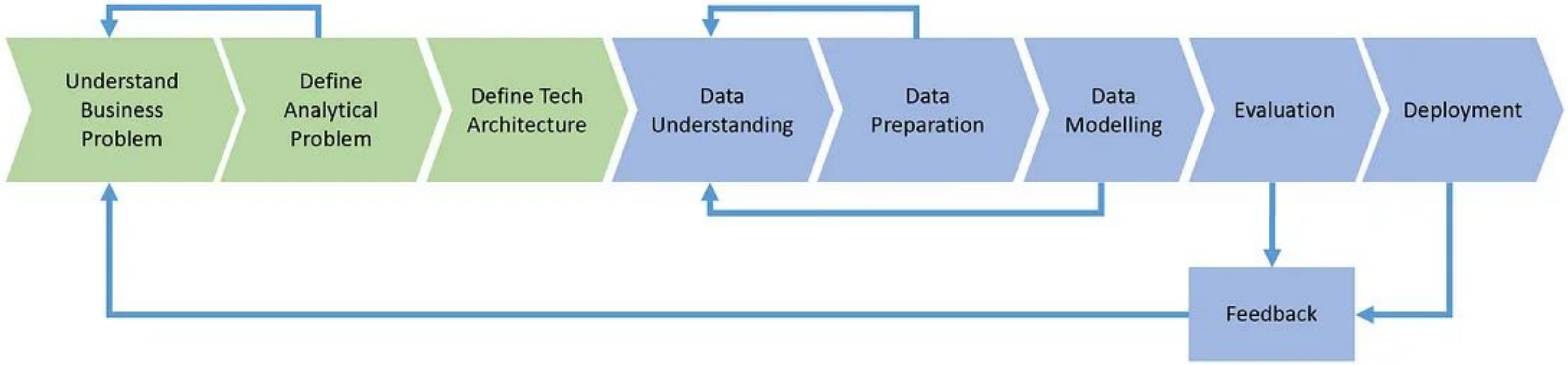
رواد مصر الرقمية

وزارة الاتصالات وتكنولوجيا المعلومات

# Data Preprocessing & Visualization

Session 1: (Numpy)


Understand Business Problem
Define Analytical Problem
Define Tech Architecture
Data Understanding
Data Preparation
Data Modelling
Evaluation
Deployment
Feedback

PyTorch

Pandas

NumPy

ملفات csv (ML )
صور او فيديوهات (DL,CV)
نصوص او صوت (DL,NLP)

plotly

matplotlib

theano

TensorFlow

scikit learn

Keras

python™
Natural Language Analyses with NLTK

# PYTHON DATA SCIENCE LIBRARIES

## Data Analysis & Visualization

- Pandas
- Matplotlib
- Seaborn
- Plotly

## Web Scraping & APIs

- Requests
- Beautiful Soup

## Machine Learning

- Scikit-Learn
- XGBoost
- Statsmodels

## API development

FastAPI

## Agenda:

| | |
|---|---|
| 1 | Introduction to Numpy |
| 2 | Numpy Arrays |
| 3 | Practical Applications |

## Introduction to Numpy

## Introduction to Numpy

What is Numpy?

Numpy is a fundamental package for scientific computing in Python. It is used for performing numerical operations on large, multi-dimensional arrays and matrices. It also provides a collection of mathematical functions to operate on these arrays.
Numpy is essential for data science, machine learning, and scientific computing because of its efficient array operations and broadcasting capabilities.

## Why NumPy?

### Speed

- Implemented in **C and Fortran** → much faster than pure Python.
- Efficient memory handling.
- Handles **large datasets** easily.

### Array Operations

- Provides the ndarray object for multi-dimensional arrays.
- Supports vectorized operations (no need for loops).

```
import numpy as np

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

b = np.array([4,5,6])

print(a + b)   # [5 7 9]
```

## Why NumPy?

### built-in Functions

- Mathematical: sin, cos, exp, log
- Statistical: mean, variance, std
- Linear Algebra: dot product, determinant, eigenvalues

### Integration

- Works with almost every DS/ML library:
  - **Pandas** → built on NumPy
  - **Scikit-learn** → ML algorithms
  - **TensorFlow / PyTorch** → Deep Learning

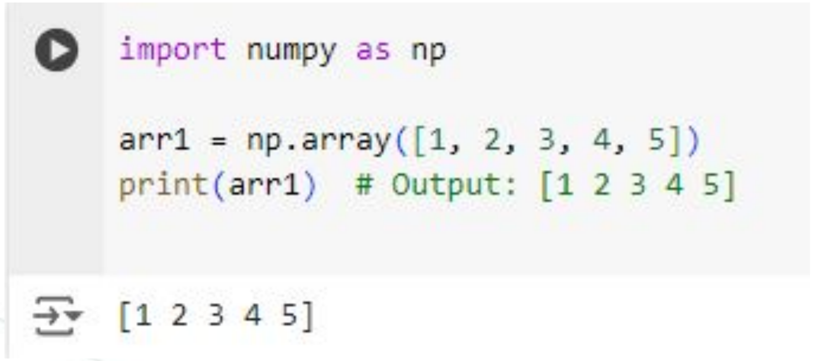
رواد مصر الرقمية

وزارة الاتصالات وتكنولوجيا المعلومات

## Numpy Arrays

## Creating Numpy Arrays

Creating a 1D Numpy Array from a List



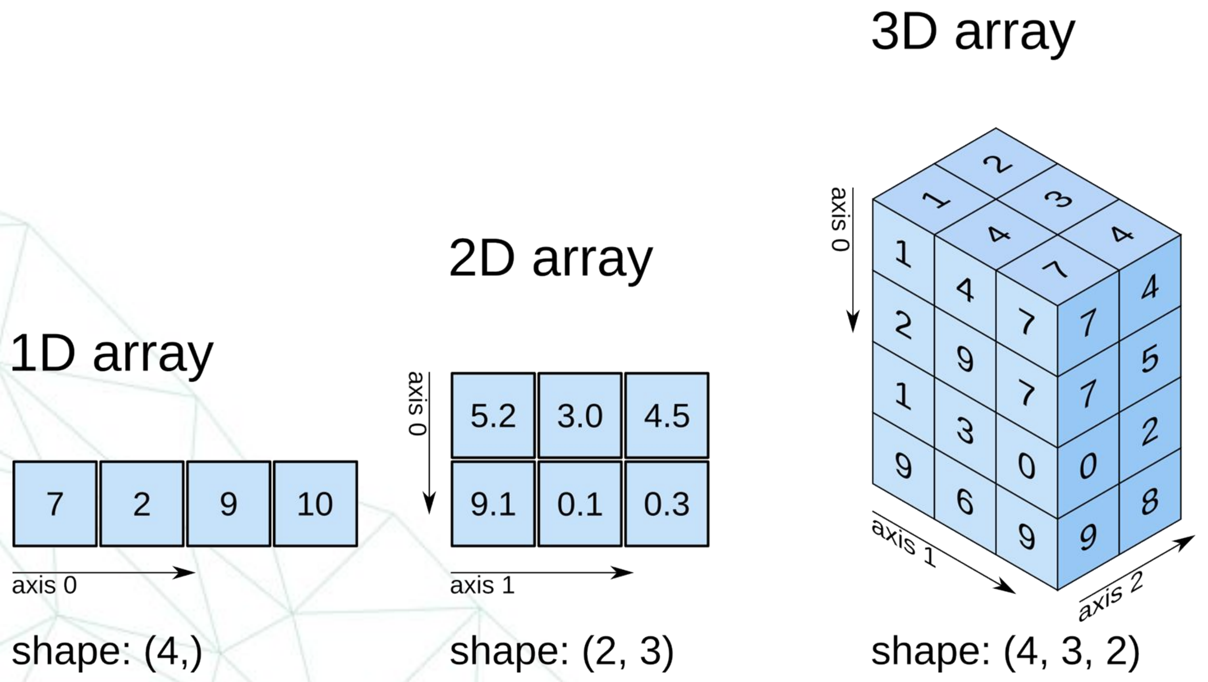
```
import numpy as np

arr1 = np.array([1, 2, 3, 4, 5])
print(arr1)  # Output: [1 2 3 4 5]
```

```
[1 2 3 4 5]
```

This creates a one-dimensional Numpy array arr1 from a Python list. Numpy arrays are more efficient for numerical operations than Python lists.

## Shape of NumPy Arrays

The shape of a NumPy array is represented as a tuple, indicating the number of elements present in each dimension. It provides information about the size of the array across its various dimensions.

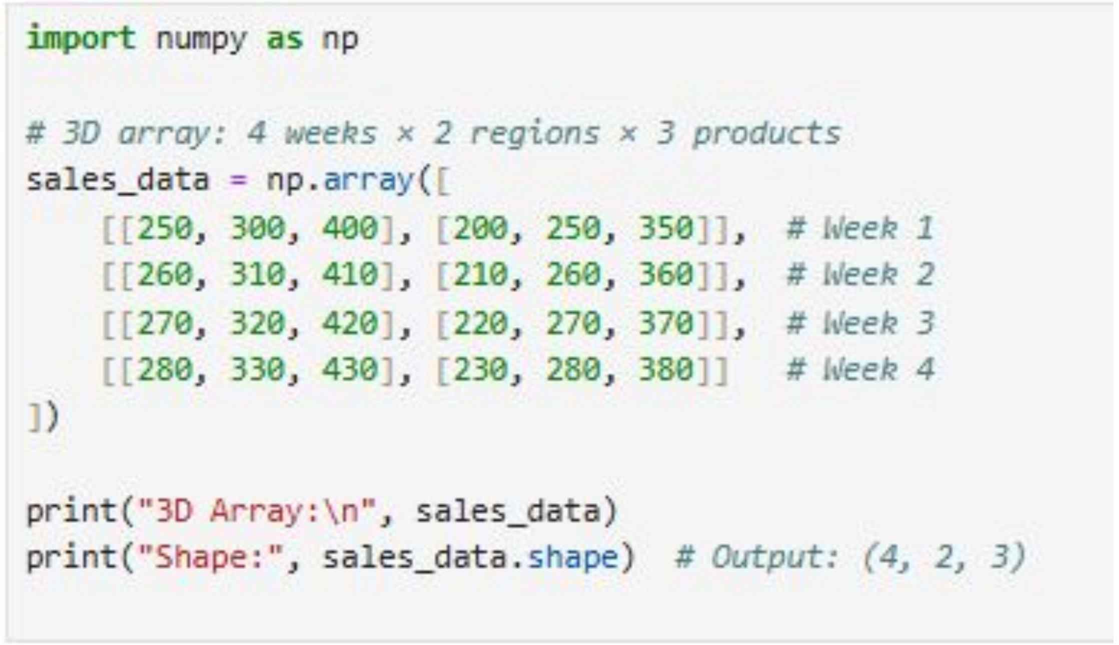


## • N-dimensional arrays:

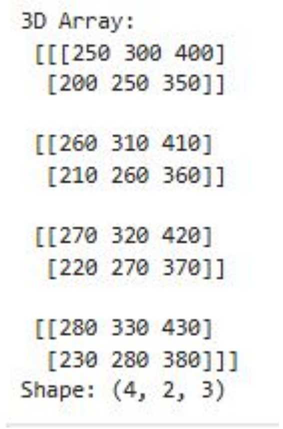
**Suppose you want to store sales data for three products over four weeks across two regions. This can be represented as a 3D array where:**

- Axis 0: Weeks (4 weeks).
- Axis 1: Regions (2 regions).
- Axis 2: Products (3 products).

Using NumPy, you can create this 3D array as follows:



```python
import numpy as np

# 3D array: 4 weeks × 2 regions × 3 products
sales_data = np.array([
    [[250, 300, 400], [200, 250, 350]],  # Week 1
    [[260, 310, 410], [210, 260, 360]],  # Week 2
    [[270, 320, 420], [220, 270, 370]],  # Week 3
    [[280, 330, 430], [230, 280, 380]]   # Week 4
])

print("3D Array:\n", sales_data)
print("Shape:", sales_data.shape)  # Output: (4, 2, 3)
```



```
3D Array:
 [[[250 300 400]
  [200 250 350]]

 [[260 310 410]
  [210 260 360]]

 [[270 320 420]
  [220 270 370]]

 [[280 330 430]
  [230 280 380]]]
Shape: (4, 2, 3)
```

# Where Data Scientists Use nD Arrays?

- **Direct Use (Explicit):**
  - To represent multi-dimensional data (e.g., Time × Region × Product).
  - For slicing/aggregation across dimensions.
  - When preparing custom features before modeling.
- **Indirect Use (Implicit / Built-in):**
  - Most ML/DL libraries (TensorFlow, PyTorch, scikit-learn) already handle 3D arrays internally.
  - Examples:
    - **Computer Vision:** Images = (Height × Width × Channels) (256, 256, 3).
    - **NLP:** Text embeddings = (Batch × Sequence × Embedding).
    - **Time Series:** Input shape = (Samples × Timesteps × Features).

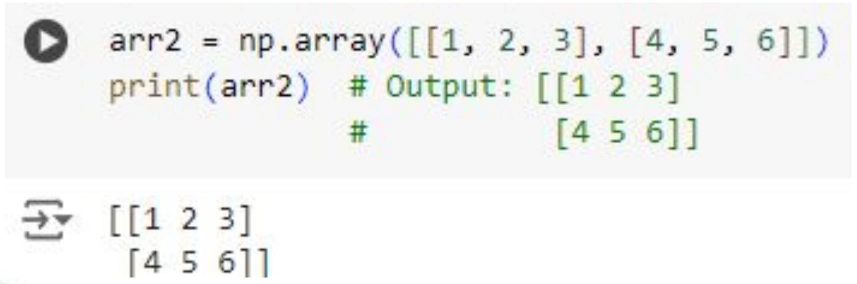
👉 In short: **You rarely build 3D arrays from scratch daily, but you must understand them, since models rely on them under the hood.**

## Do modern models reshape data automatically, or do you need to do it?

- **Models don’t reshape data for you**: Neural networks expect input with a specific structure—if your data doesn’t match that exact shape, you'll get an error
- **You’re responsible for reshaping**: As a Data Scientist, it’s your job to adjust your data (using functions like `reshape`, `resize`, etc.) to exactly match the model’s expected input format.
- **No automatic shape correction**: Most models won’t rearrange your data—except for minimal cases (like adding a batch dimension)—so you must ensure it’s formatted correctly

## Creating Numpy Arrays

Creating a 2D Numpy Array from a List of Lists



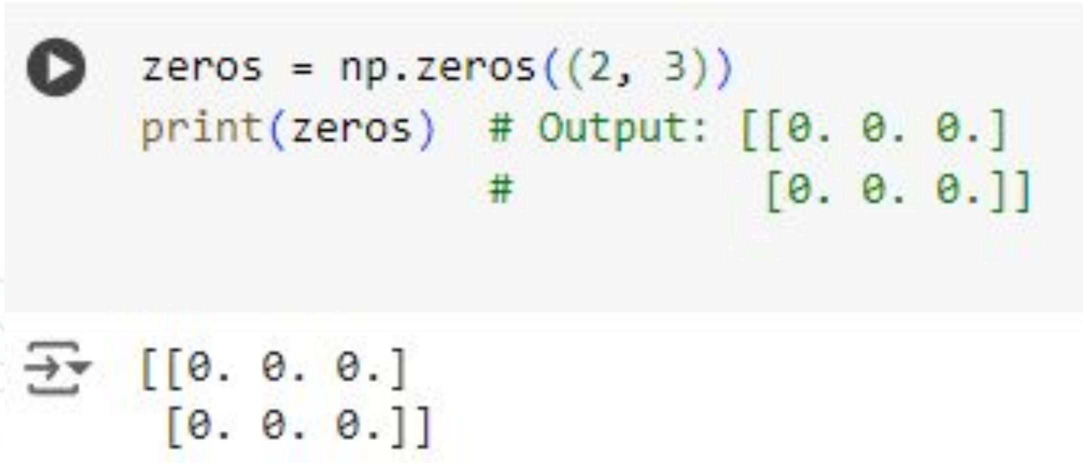
```
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2)  # Output: [[1 2 3]
             #          [4 5 6]]
```

```
[[1 2 3]
 [4 5 6]]
```

This creates a two-dimensional Numpy array arr2 from a list of lists. Numpy arrays can handle multi-dimensional data easily.

## Creating Numpy Arrays

Creating an Array of Zeros



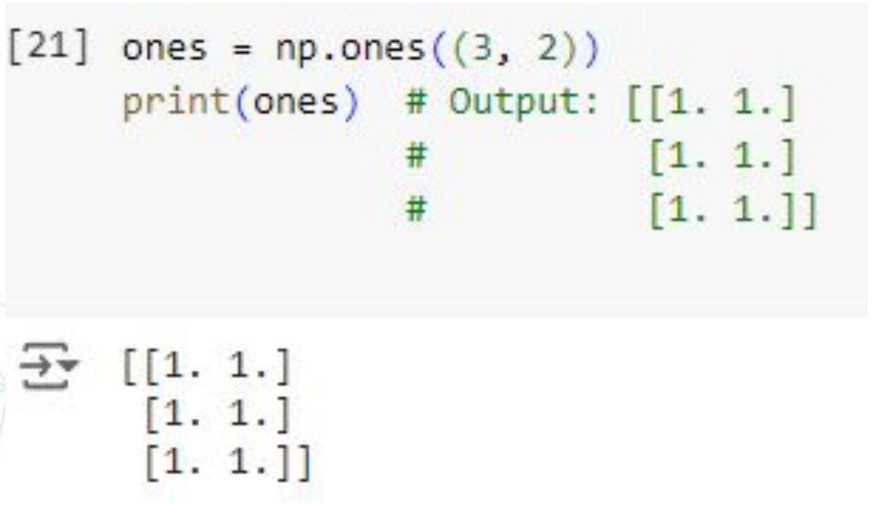
```
zeros = np.zeros((2, 3))
print(zeros)  # Output: [[0. 0. 0.]
              #          [0. 0. 0.]]
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
```

This creates a 2x3 Numpy array filled with zeros. It is useful for initializing arrays with default values.

## Creating Numpy Arrays

Creating an Array of Ones



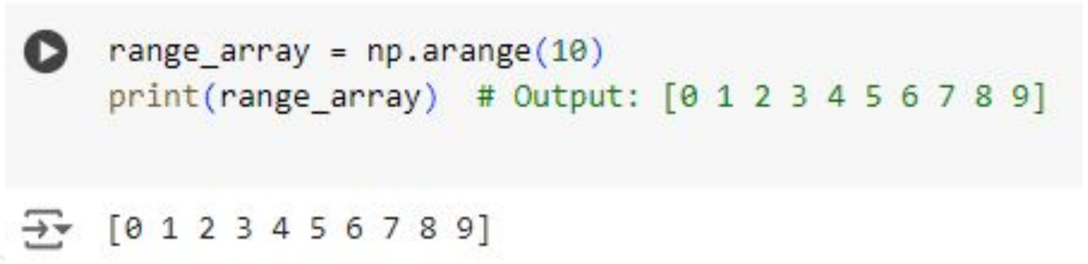
```
[21] ones = np.ones((3, 2))
     print(ones)  # Output: [[1. 1.]
                  #          [1. 1.]
                  #          [1. 1.]]
```

```
[[1. 1.]
 [1. 1.]
 [1. 1.]]
```

This creates a 3x2 Numpy array filled with ones. It’s useful for creating arrays where all elements start with a value of one.

## Creating Numpy Arrays

Creating an Array with a Specific Range



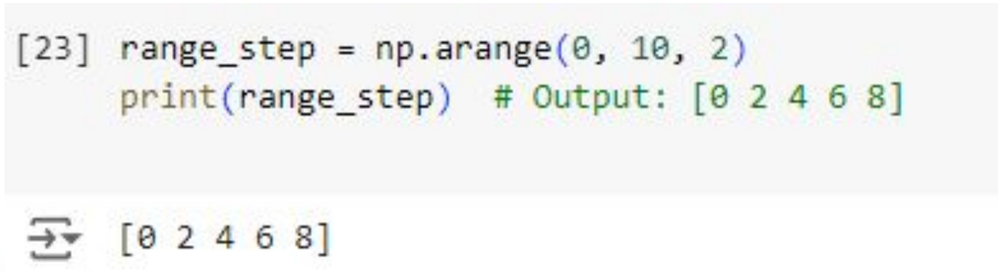
```
range_array = np.arange(10)
print(range_array)  # Output: [0 1 2 3 4 5 6 7 8 9]
```

```
[0 1 2 3 4 5 6 7 8 9]
```

This creates a one-dimensional array of integers from 0 to 9 using np.arange(). It’s similar to Python’s built-in range() but returns a Numpy array.

## Creating Numpy Arrays

Creating an Array with a Range of Values and Specific Step Size



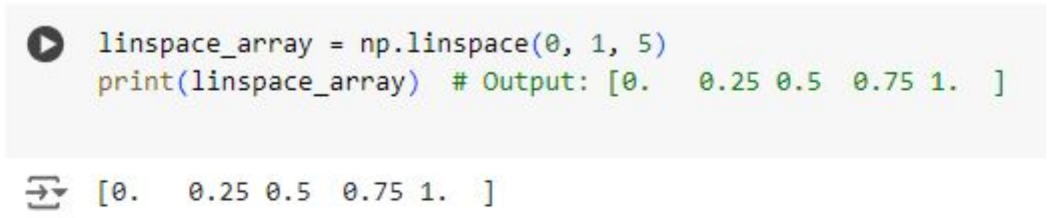
```
[23] range_step = np.arange(0, 10, 2)
     print(range_step)  # Output: [0 2 4 6 8]
```

```
[0 2 4 6 8]
```

This creates an array with values starting from 0 up to (but not including) 10, with a step size of 2. Useful for generating sequences with a specific interval.

## Creating Numpy Arrays

Creating an Array of Evenly Spaced Values Between Two Numbers



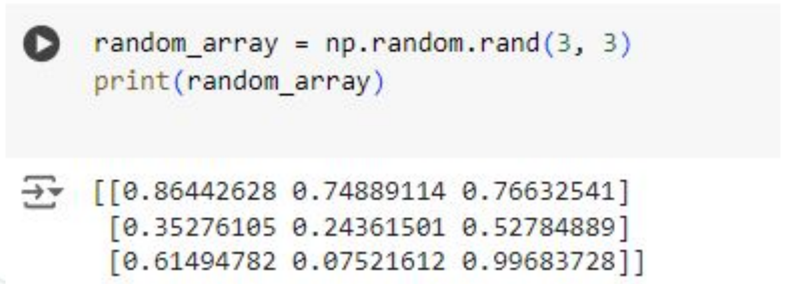
```
linspace_array = np.linspace(0, 1, 5)
print(linspace_array)  # Output: [0.   0.25 0.5  0.75 1.  ]
```

```
[0.   0.25 0.5  0.75 1.  ]
```

This creates an array of 5 evenly spaced values between 0 and 1. np.linspace() is often used for generating a sequence of numbers over a specified interval.

## Creating Numpy Arrays

Creating an Array with Random Values Between 0 and 1



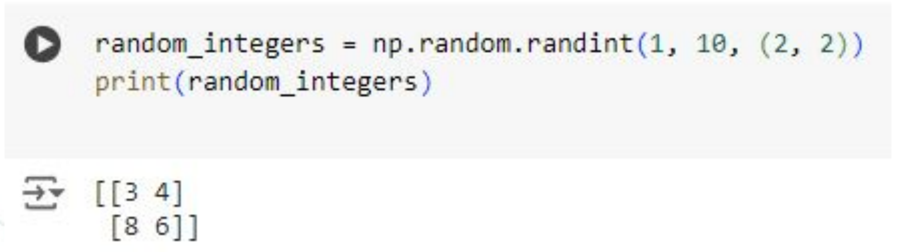
```
random_array = np.random.rand(3, 3)
print(random_array)
```

```
[[0.86442628 0.74889114 0.76632541]
 [0.35276105 0.24361501 0.52784889]
 [0.61494782 0.07521612 0.99683728]]
```

This creates a 3x3 array with random floating-point numbers between 0 and 1. Useful for simulations and generating random datasets.

## Creating Numpy Arrays

Creating an Array with Random Integers



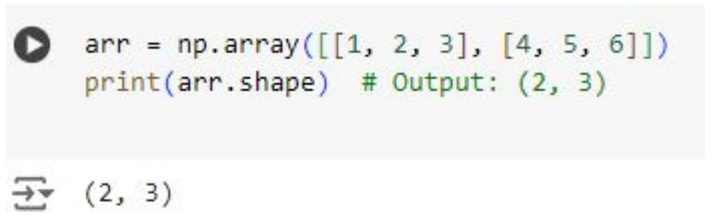
```python
random_integers = np.random.randint(1, 10, (2, 2))
print(random_integers)
```

```
[[3 4]
 [8 6]]
```

This creates a 2x2 array with random integers between 1 and 9. Ideal for testing and generating sample data.

## Creating Numpy Arrays

Checking the Shape of an Array



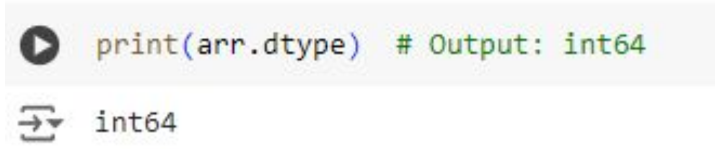
```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.shape)  # Output: (2, 3)
```

```
(2, 3)
```

This returns the shape of the array, showing it has 2 rows and 3 columns. shape is useful for understanding the structure of multi-dimensional arrays.

## Creating Numpy Arrays

Accessing Data Type of Elements



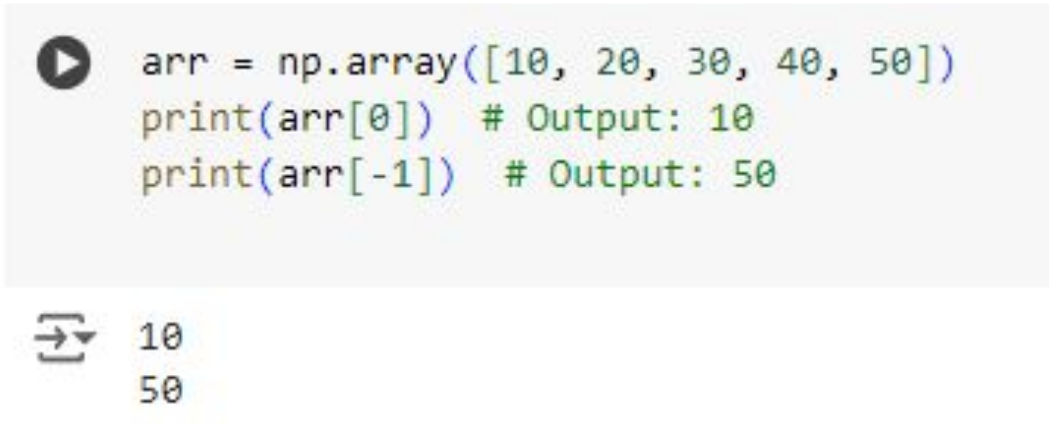
```
print(arr.dtype)  # Output: int64
```

```
int64
```

This checks the data type of elements in the array, showing that they are 64-bit integers. Knowing the data type helps optimize storage and computation.

## Creating Numpy Arrays

Indexing a 1D Array



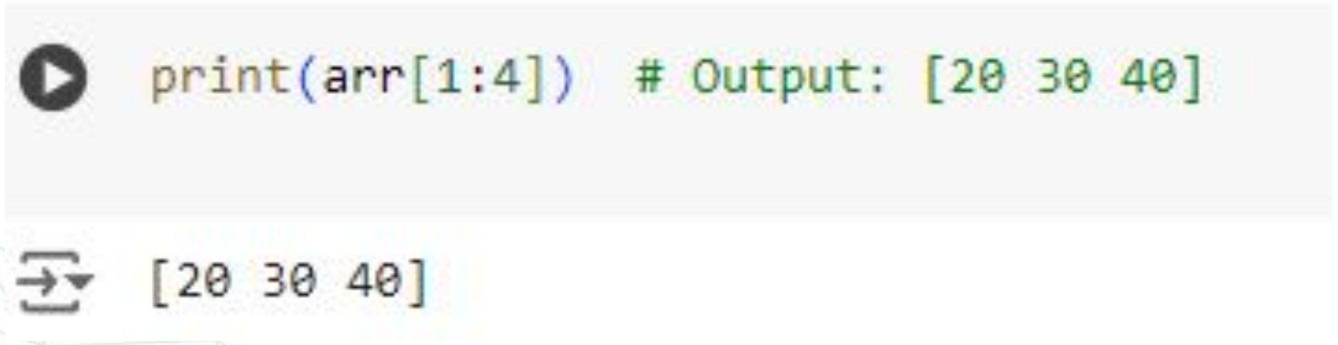
```
arr = np.array([10, 20, 30, 40, 50])
print(arr[0])  # Output: 10
print(arr[-1])  # Output: 50
```

```
10
50
```

This accesses elements in the array by their index. Positive indices start from the beginning, while negative indices start from the end.

## Creating Numpy Arrays

Slicing a 1D Array



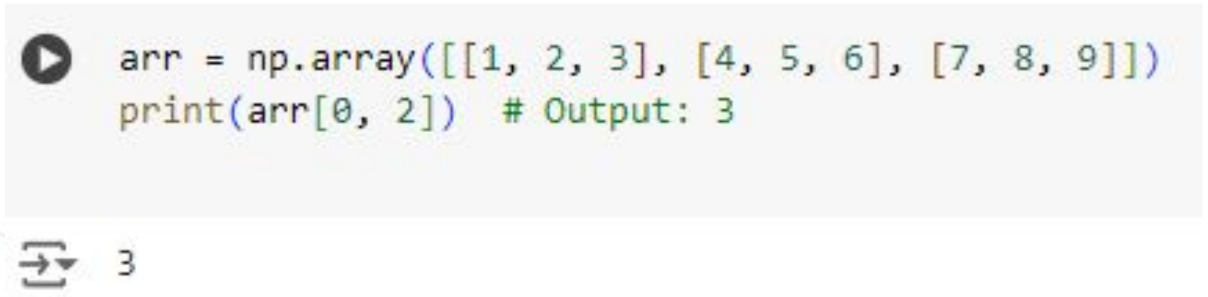
```
print(arr[1:4])  # Output: [20 30 40]
```

```
[20 30 40]
```

This slices the array to get a subarray from index 1 to 3.
Slicing is useful for extracting parts of an array.

## Creating Numpy Arrays

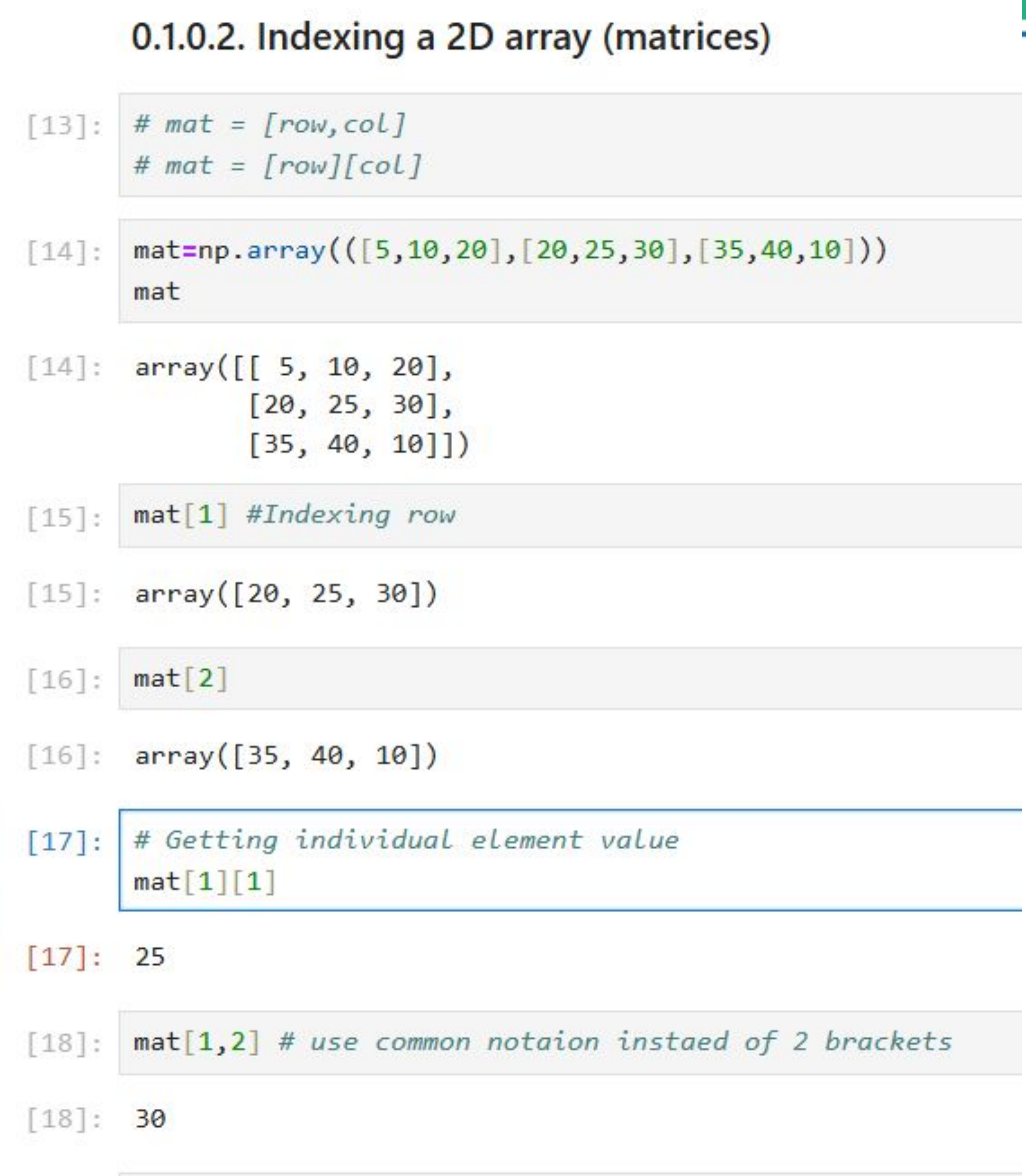
Indexing a 2D Array



```
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr[0, 2])  # Output: 3
```

```
3
```

This accesses the element in the first row and third column of the array. 2D indexing requires specifying both row and column indices.

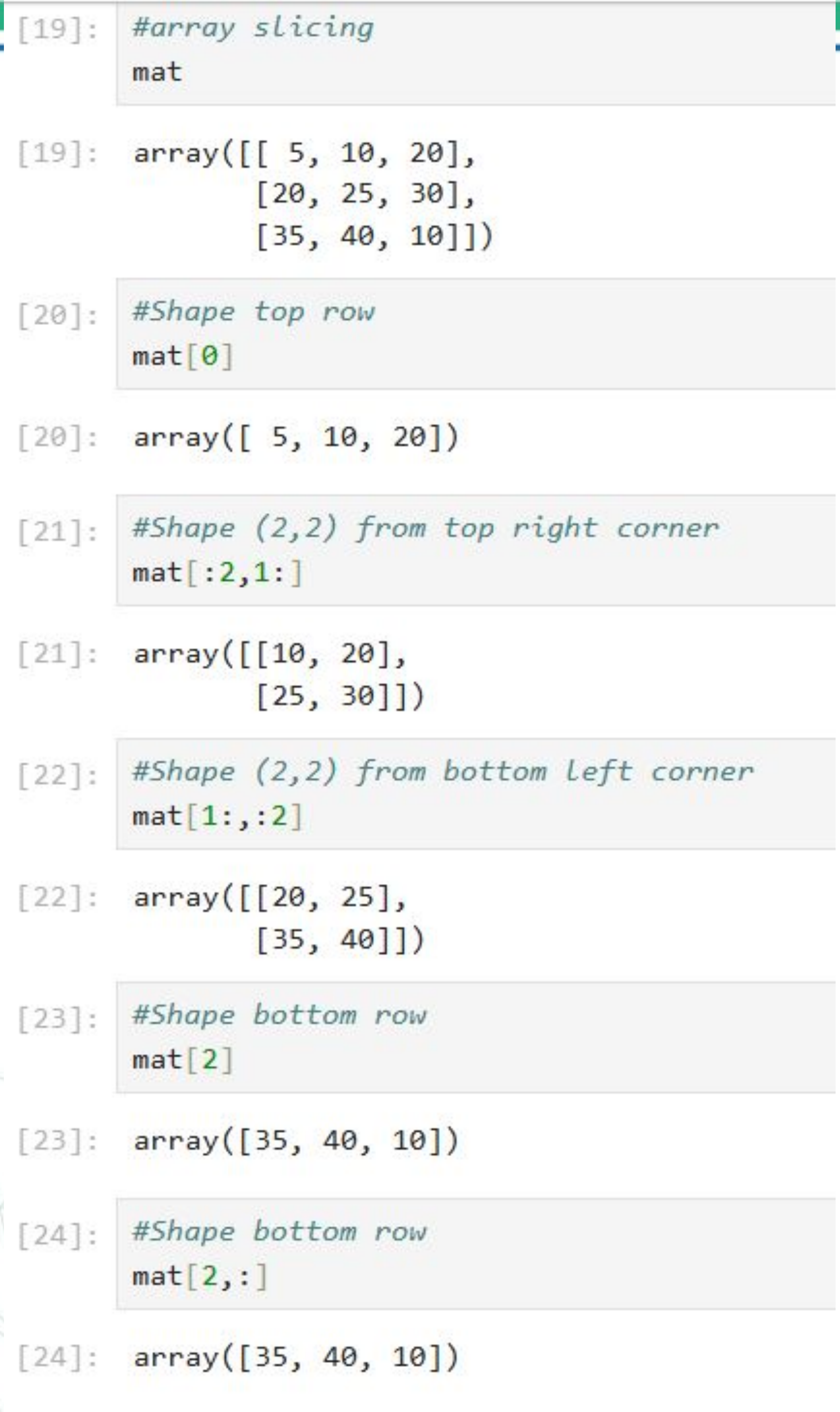


## 0.1.0.2. Indexing a 2D array (matrices)

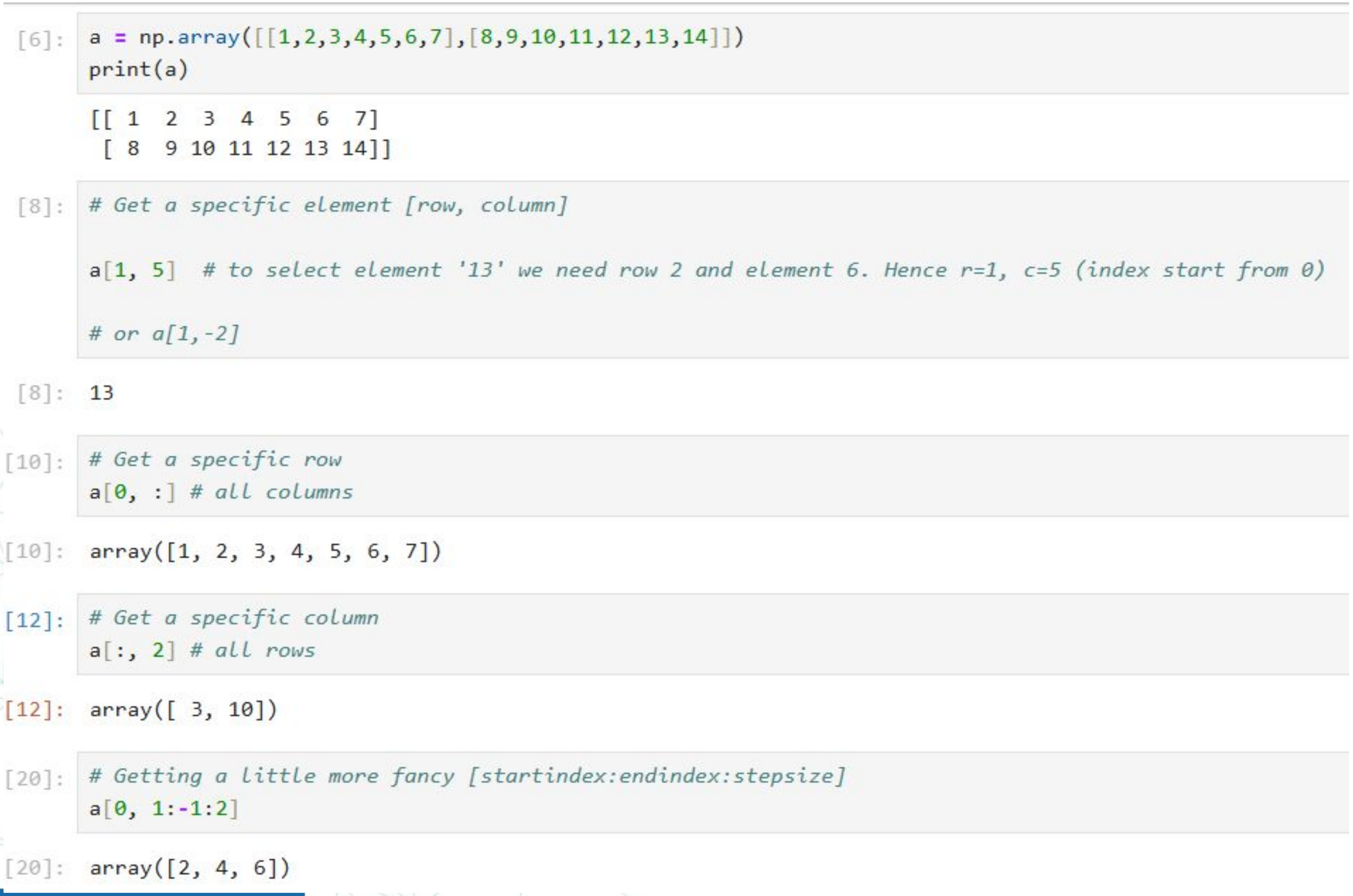
```
[13]: # mat = [row,col]
      # mat = [row][col]

[14]: mat=np.array(([5,10,20],[20,25,30],[35,40,10]))
      mat

[14]: array([[ 5, 10, 20],
             [20, 25, 30],
             [35, 40, 10]])

[15]: mat[1] #Indexing row

[15]: array([20, 25, 30])

[16]: mat[2]

[16]: array([35, 40, 10])

[17]: # Getting individual element value
      mat[1][1]

[17]: 25

[18]: mat[1,2] # use common notaion instaed of 2 brackets

[18]: 30
```

## Creating Numpy Arrays

Slicing a 2D Array



```
[19]: #array slicing
      mat

[19]: array([[ 5, 10, 20],
             [20, 25, 30],
             [35, 40, 10]])

[20]: #Shape top row
      mat[0]

[20]: array([ 5, 10, 20])

[21]: #Shape (2,2) from top right corner
      mat[:2,1:]

[21]: array([[10, 20],
             [25, 30]])

[22]: #Shape (2,2) from bottom left corner
      mat[1:,:2]

[22]: array([[20, 25],
             [35, 40]])

[23]: #Shape bottom row
      mat[2]

[23]: array([35, 40, 10])

[24]: #Shape bottom row
      mat[2,:]

[24]: array([35, 40, 10])
```



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

     [[ 1  2  3  4  5  6  7]
      [ 8  9 10 11 12 13 14]]

[8]: # Get a specific element [row, column]

     a[1, 5]  # to select element '13' we need row 2 and element 6. Hence r=1, c=5 (index start from 0)

     # or a[1,-2]

[8]: 13

[10]: # Get a specific row
      a[0, :] # all columns

[10]: array([1, 2, 3, 4, 5, 6, 7])

[12]: # Get a specific column
      a[:, 2] # all rows

[12]: array([ 3, 10])

[20]: # Getting a little more fancy [startindex:endindex:stepsize]
      a[0, 1:-1:2]

[20]: array([2, 4, 6])
```

## 0.1.3. Boolean array indexing

Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. Here is an example:

```
[39]: a = np.arange(1,11)
      a
```

```
[39]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

```
[40]: a > 4
```

```
[40]: array([False, False, False, False,  True,  True,  True,  True,  True,
              True])
```

```
[41]: bool_a = a>4
```

```
[42]: bool_a
```

```
[42]: array([False, False, False, False,  True,  True,  True,  True,  True,
              True])
```

```
[43]: a[bool_a]
```

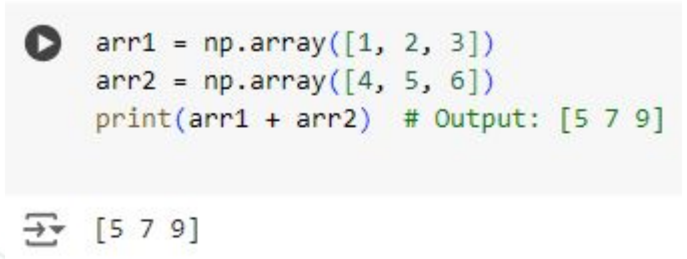
```
[43]: array([ 5,  6,  7,  8,  9, 10])
```

```
[44]: a[a>2]
```

```
[44]: array([ 3,  4,  5,  6,  7,  8,  9, 10])
```

## Creating Numpy Arrays

Element-wise Addition



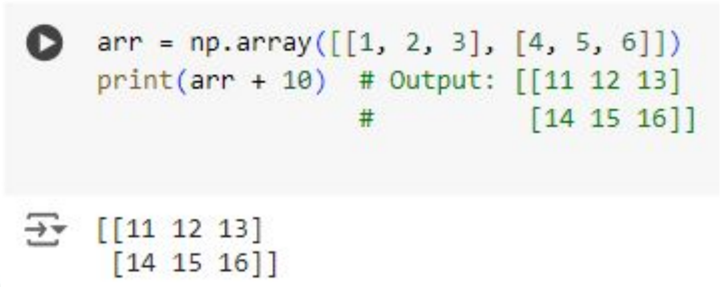
```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
print(arr1 + arr2)  # Output: [5 7 9]
```

```
[5 7 9]
```

This performs element-wise addition of two arrays. Numpy automatically adds corresponding elements, making operations concise and efficient.

## Creating Numpy Arrays

Array Broadcasting



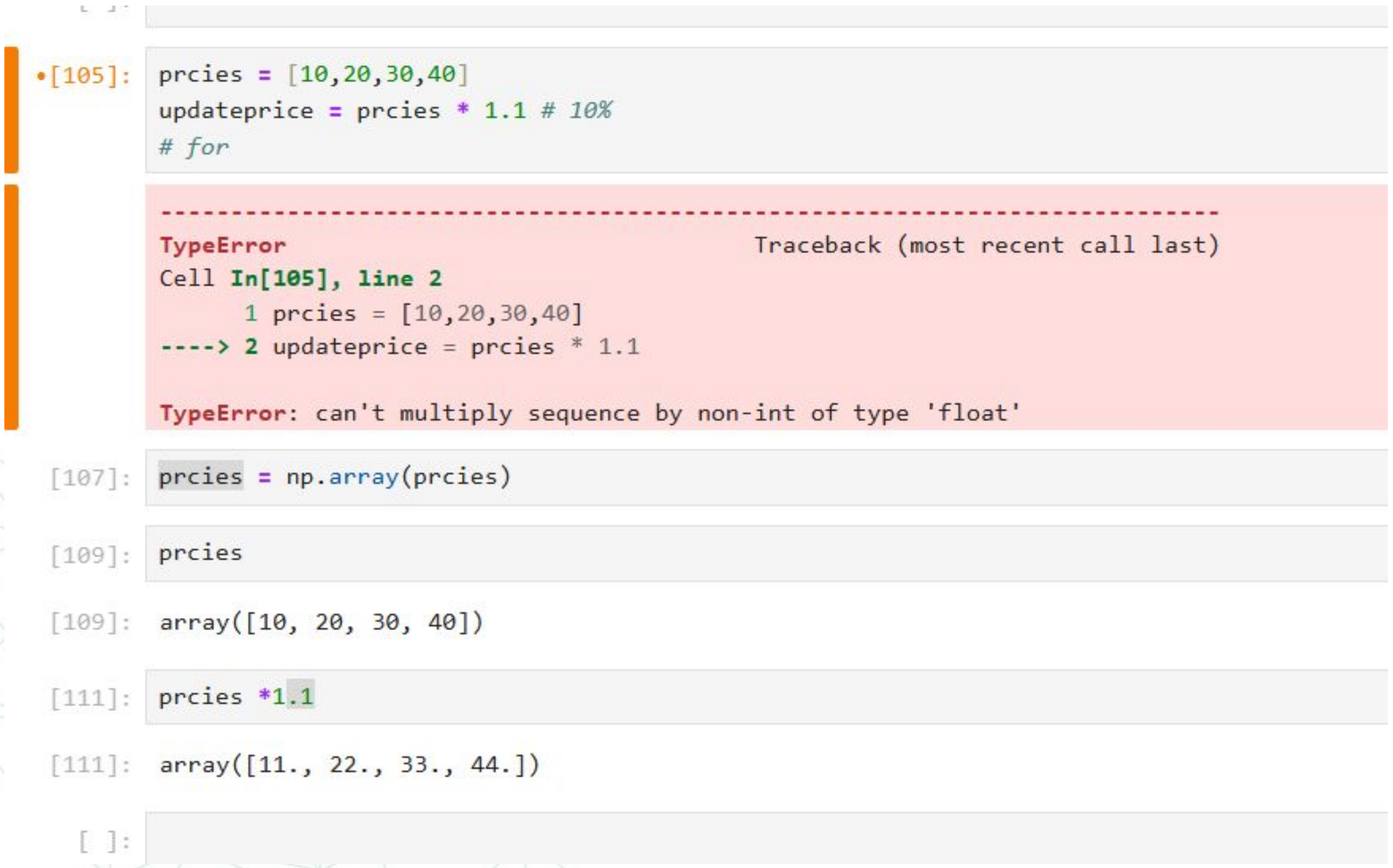
```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr + 10)  # Output: [[11 12 13]
                 #           [14 15 16]]
```

```
[[11 12 13]
 [14 15 16]]
```

Broadcasting allows arithmetic operations on arrays of different shapes. Here, 10 is added to each element of the array.

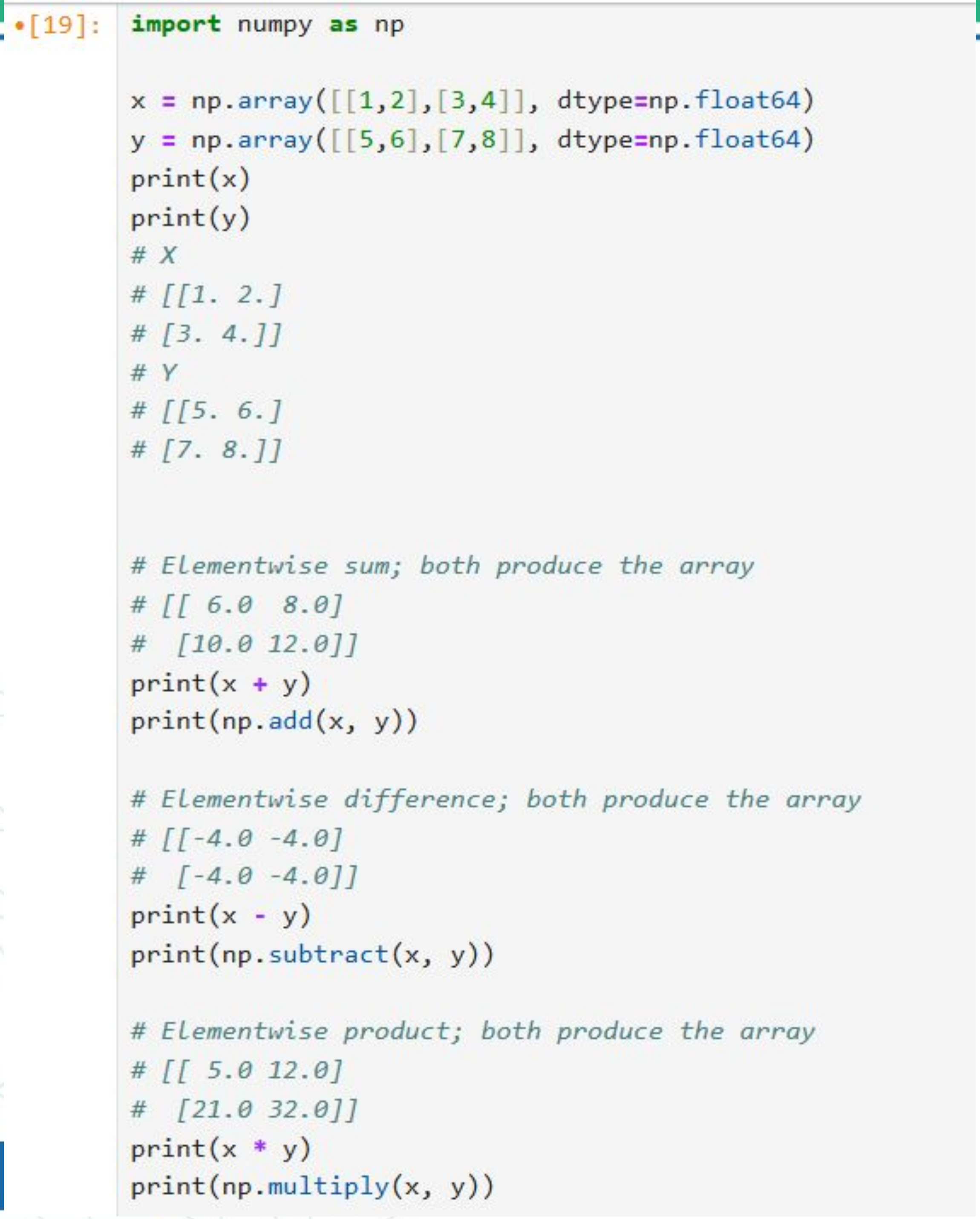
- If a mathematical operation fails due to shape incompatibility or when broadcasting rules don't apply, Numpy will raise an error. This happens because Numpy can't carry out the operation as there is an unresolved dimensional mismatch.

## Array Broadcasting

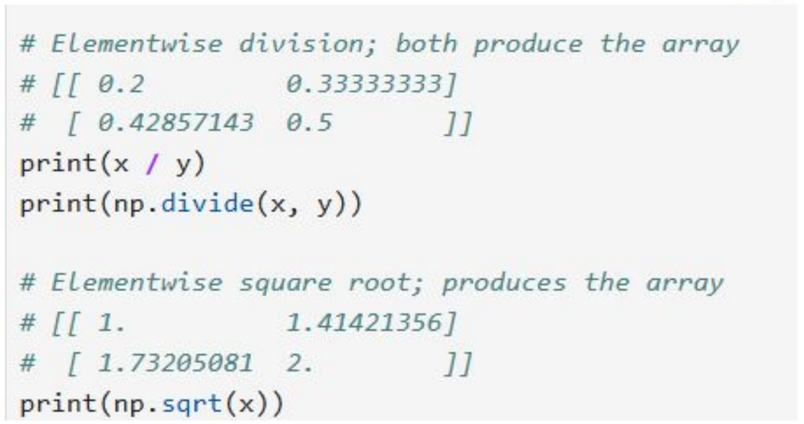


```
•[105]: prcies = [10,20,30,40]
        updateprice = prcies * 1.1 # 10%
        # for

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[105], line 2
      1 prcies = [10,20,30,40]
----> 2 updateprice = prcies * 1.1

TypeError: can't multiply sequence by non-int of type 'float'

[107]: prcies = np.array(prcies)

[109]: prcies

[109]: array([10, 20, 30, 40])

[111]: prcies *1.1

[111]: array([11., 22., 33., 44.])

[ ]:
```

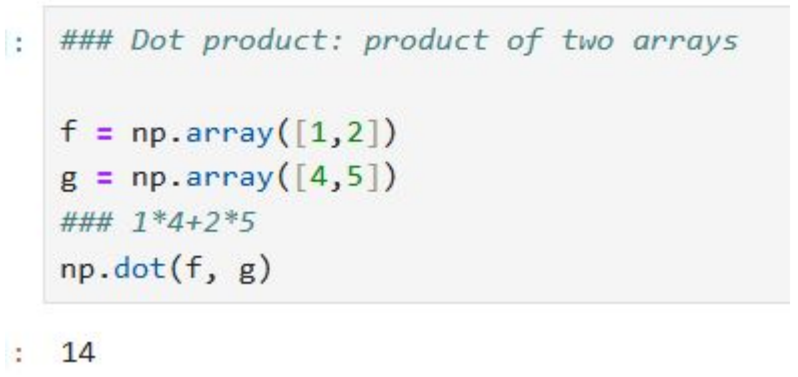
## 0.1.1. Universal Array Functions (Mathematical operations)



```
•[19]: import numpy as np

       x = np.array([[1,2],[3,4]], dtype=np.float64)
       y = np.array([[5,6],[7,8]], dtype=np.float64)
       print(x)
       print(y)
       # X
       # [[1. 2.]
       # [3. 4.]]
       # Y
       # [[5. 6.]
       # [7. 8.]]


       # Elementwise sum; both produce the array
       # [[ 6.0  8.0]
       #  [10.0 12.0]]
       print(x + y)
       print(np.add(x, y))

       # Elementwise difference; both produce the array
       # [[-4.0 -4.0]
       #  [-4.0 -4.0]]
       print(x - y)
       print(np.subtract(x, y))

       # Elementwise product; both produce the array
       # [[ 5.0 12.0]
       #  [21.0 32.0]]
       print(x * y)
       print(np.multiply(x, y))
```



```
# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```



```
:  ### Dot product: product of two arrays

   f = np.array([1,2])
   g = np.array([4,5])
   ### 1*4+2*5
   np.dot(f, g)

:  14
```

## 0.2. Broadcasting

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

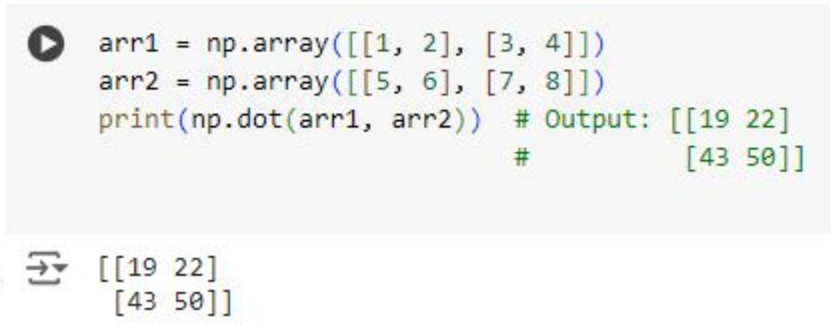
For example, suppose that we want to add a constant vector to each row of a matrix. We could do it like this:

[7]:
```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

# Now y is the following
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(x)
print("================")
print(y)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
================
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

## Creating Numpy Arrays

### Matrix Multiplication



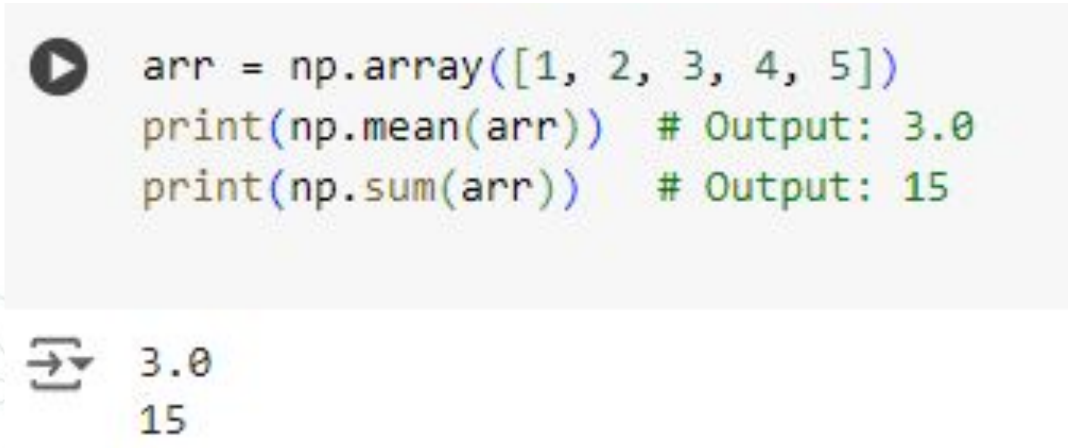
```
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])
print(np.dot(arr1, arr2))  # Output: [[19 22]
                           #          [43 50]]
```

```
[[19 22]
 [43 50]]
```

This performs matrix multiplication using np.dot(), which calculates the dot product of two arrays. Useful for linear algebra applications.

## Creating Numpy Arrays

Calculating Mean and Sum



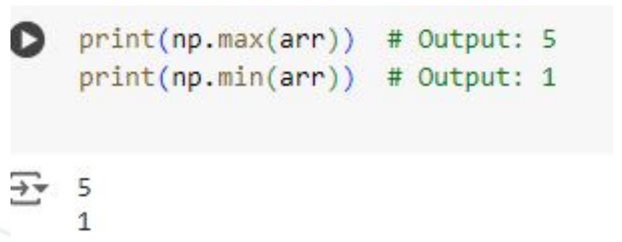
```
arr = np.array([1, 2, 3, 4, 5])
print(np.mean(arr))  # Output: 3.0
print(np.sum(arr))   # Output: 15
```

```
3.0
15
```

These operations calculate the mean (average) and sum of the array elements. Numpy provides efficient built-in functions for statistical calculations.

## Creating Numpy Arrays

### Finding Maximum and Minimum Values



```
print(np.max(arr))  # Output: 5
print(np.min(arr))  # Output: 1
```

```
5
1
```

np.max() finds the maximum value, and np.min() finds the minimum value in the array. Useful for identifying the range of data.

## **Eigenvalues** and **Eigenvectors**.

Now that we've seen how NumPy helps us create and manipulate matrices, let's move one step deeper into *Linear Algebra concepts* that are very important in Data Science and Machine Learning.

Two of the most fundamental ideas are **Eigenvalues** and **Eigenvectors**.

- They may sound a bit abstract at first, but they are actually everywhere in Data Science.

- From **PCA for dimensionality reduction**, to **understanding covariance in datasets**, to **optimizing machine learning models** – these concepts play a key role.

Think of them as the tools that tell us the *main directions of information* in our data, and how strong each direction is.

## 1. Eigenvalues & Eigenvectors

Imagine you have a **matrix (Matrix)** that represents a transformation (like rotation, stretching, or compression).

- **Eigenvector:** a direction (vector) that does not change its direction after the transformation, it may only get stretched or shrunk.

- **Eigenvalue:** the number that tells you how much the eigenvector is stretched or shrunk.

🔹 eigenvector = fixed direction, eigenvalue = scaling factor.

## 2. Eigenvalues & Covariance Matrix

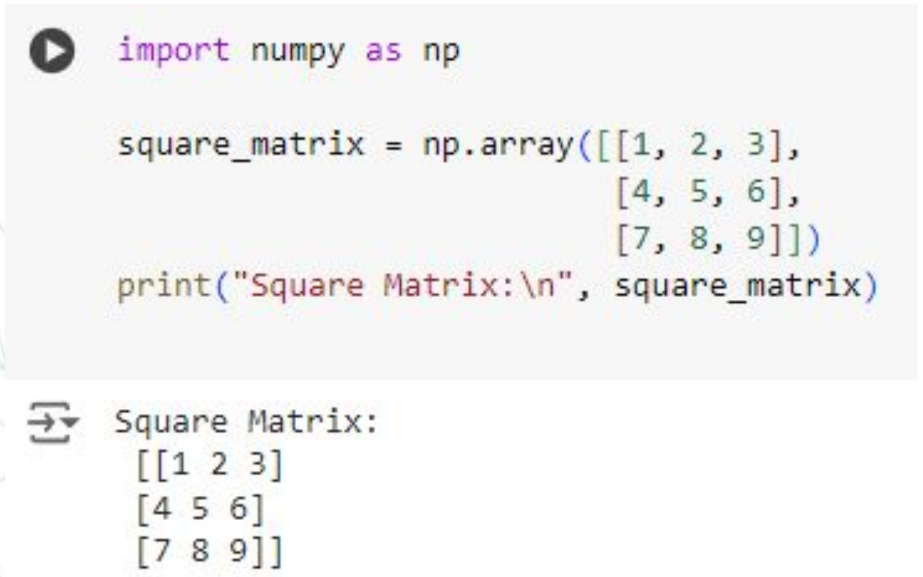
- The **Covariance Matrix** describes how features (variables) are correlated with each other.
- If we perform **Eigen Decomposition** on it:
  - **Eigenvectors =** the directions that explain the largest amount of variance in the data.
  - **Eigenvalues =** how much variance is explained along each direction.

This is exactly what happens in **PCA (Principal Component Analysis):**

1. Sort eigenvalues from largest to smallest.
2. Keep the top eigenvectors → reduce dimensions while preserving most of the variance.

## Square Matrix

A matrix with the same number of rows and columns.



```python
import numpy as np

square_matrix = np.array([[1, 2, 3],
                          [4, 5, 6],
                          [7, 8, 9]])
print("Square Matrix:\n", square_matrix)
```

```
Square Matrix:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
```

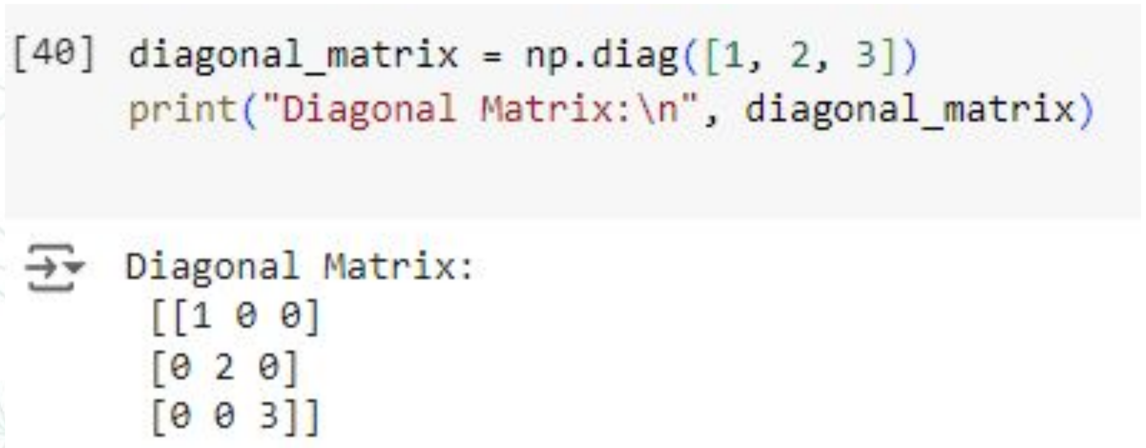
This is a 3x3 matrix where the number of rows equals the number of columns.

**Practical Uses**

- **Covariance Matrix** → always square.
- **Adjacency Matrix** (Social Networks / Graphs).
- **Image Filters** (3×3, 5×5 kernels in Computer Vision).

## Diagonal Matrix

A matrix in which all off-diagonal elements are zero



```
[40] diagonal_matrix = np.diag([1, 2, 3])
     print("Diagonal Matrix:\n", diagonal_matrix)
```

```
Diagonal Matrix:
 [[1 0 0]
 [0 2 0]
 [0 0 3]]
```

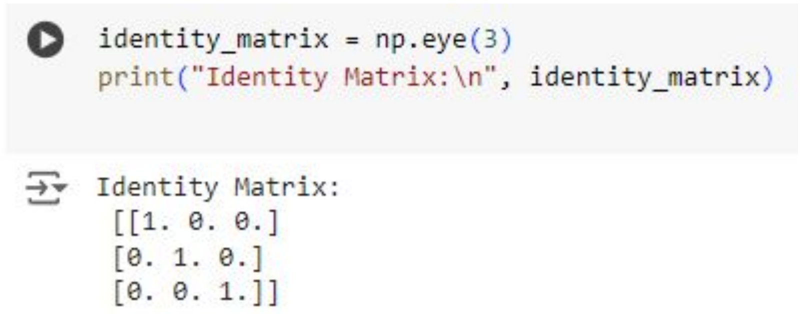
The np.diag() function creates a diagonal matrix where only the elements along the main diagonal are non-zero.

### Practical Uses

- **Covariance matrices** sometimes simplified to diagonal (no correlation between features).
- **Feature scaling** and **whitening** often use diagonal matrices.
- **Image processing filters** can be diagonal for certain effects.
- Simplifies many linear algebra operations.

## Identity Matrix

A square matrix with ones on the main diagonal and zeros elsewhere.



```
identity_matrix = np.eye(3)
print("Identity Matrix:\n", identity_matrix)
```

```
Identity Matrix:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

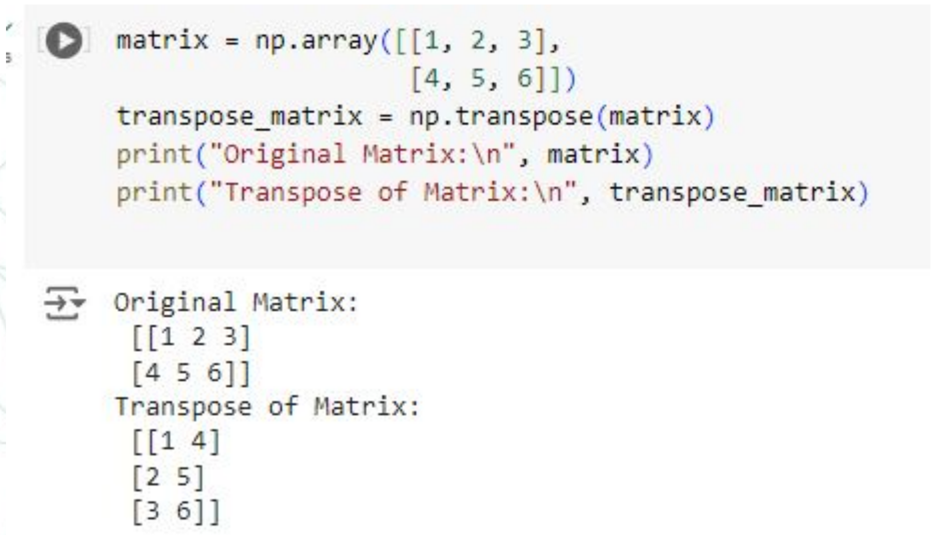
- A special type of diagonal matrix.
- Diagonal elements = 1.
- All off-diagonal elements = 0.
- The np.eye(3) function creates a 3x3 identity matrix. Identity matrices are used as the multiplicative identity in matrix algebra.

### Practical Uses

- **Linear Algebra:** neutral element in multiplication.
- **Machine Learning:** used in regularization (adding λI to covariance matrix in Ridge Regression).
- **Computer Vision:** identity filter keeps the image unchanged.
- **Optimization:** often appears in matrix factorizations.

## Transpose of a Matrix

Flipping a matrix over its diagonal, switching the row and column indices.



```
matrix = np.array([[1, 2, 3],
                   [4, 5, 6]])
transpose_matrix = np.transpose(matrix)
print("Original Matrix:\n", matrix)
print("Transpose of Matrix:\n", transpose_matrix)
```

```
Original Matrix:
 [[1 2 3]
 [4 5 6]]
Transpose of Matrix:
 [[1 4]
 [2 5]
 [3 6]]
```

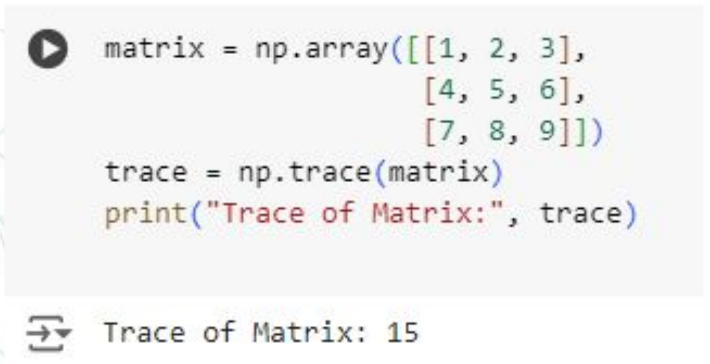
- The np.transpose() function is used to find the transpose of a matrix, which switches the rows and columns.
- Rows become columns, columns become rows.

### Practical Uses

- **Dot Product: can be written as** $x^T y$
- **Covariance Matrix:** built using transposed data vectors.
- **Machine Learning:** in linear regression $X^T X$
- **Deep Learning:** weight matrices often transposed in backpropagation.

## Trace of a Matrix

The sum of all the elements on the main diagonal of a square matrix.



```python
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])
trace = np.trace(matrix)
print("Trace of Matrix:", trace)
```

```
Trace of Matrix: 15
```

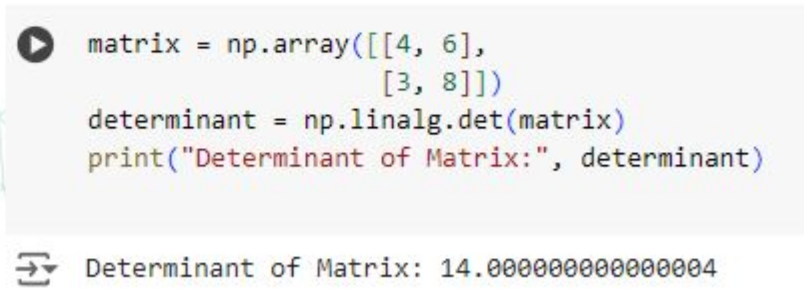
The np.trace() function calculates the trace of a matrix, which is often used in machine learning algorithms and various mathematical computations.

### Practical Uses

- **Machine Learning:** cost functions sometimes include trace terms (e.g., matrix norms).
- **Linear Algebra: trace =** sum of eigenvalues of a matrix.
- **PCA / Dimensionality Reduction:** trace of covariance matrix = total variance.
- **Optimization:** used in matrix calculus for gradients.

## Determinant of a Matrix

A scalar value that is a function of a square matrix, often used in solving linear equations, calculating matrix inverses, and more.



```
matrix = np.array([[4, 6],
                   [3, 8]])
determinant = np.linalg.det(matrix)
print("Determinant of Matrix:", determinant)
```

```
Determinant of Matrix: 14.000000000000004
```

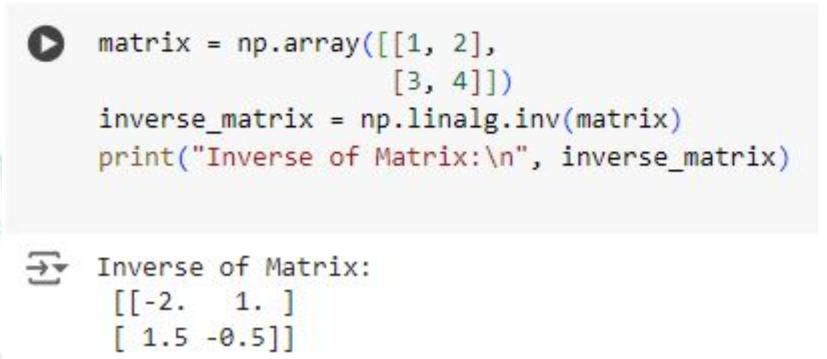
The np.linalg.det() function computes the determinant, which provides important information about the matrix properties, such as invertibility.

**Practical Uses**

- **Unique scalar value calculated from a square matrix.**
- **Tells us whether the matrix is invertible or not.**
- **In Data Science → used in solving linear equations.**

## Inverse of a Matrix

A matrix that, when multiplied with the original matrix, yields the identity matrix.



```python
matrix = np.array([[1, 2],
                   [3, 4]])
inverse_matrix = np.linalg.inv(matrix)
print("Inverse of Matrix:\n", inverse_matrix)
```

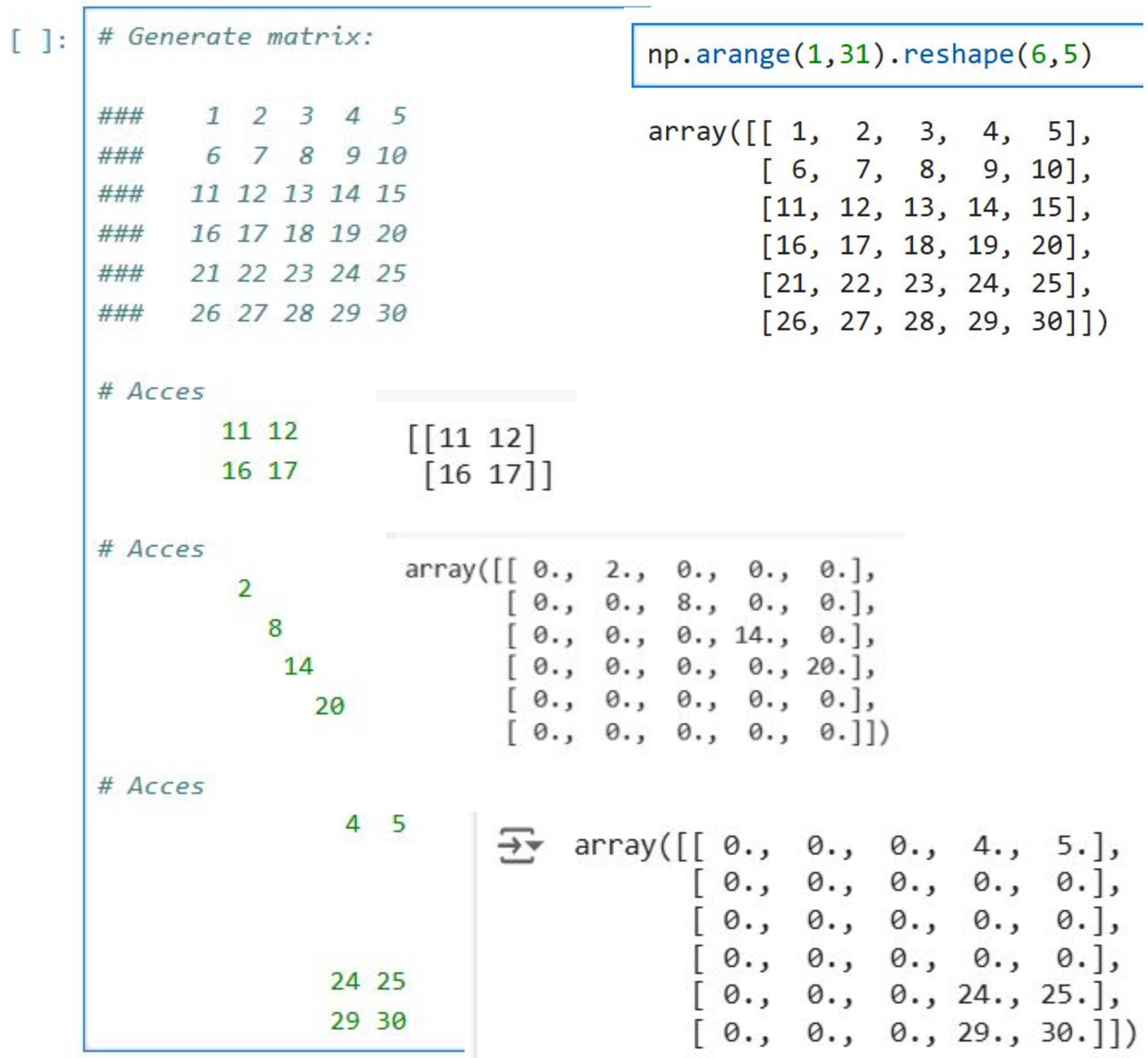
```
Inverse of Matrix:
 [[-2.   1. ]
 [ 1.5 -0.5]]
```

The np.linalg.inv() function finds the inverse of a matrix, which is useful for solving systems of linear equations.

**Practical Uses**

- **Works only for square matrices with non-zero determinant.**
- **Like an "undo" operation in mathematics.**
- **Important in regression and optimization problems.**

## (Numpy) :
## Task 1: Slicing



```
[ ]: # Generate matrix:

     ###    1  2  3  4  5
     ###    6  7  8  9 10
     ###   11 12 13 14 15
     ###   16 17 18 19 20
     ###   21 22 23 24 25
     ###   26 27 28 29 30

     # Acces
             11 12
             16 17

     # Acces
               2
                 8
                  14
                    20

     # Acces
                     4  5



                    24 25
                    29 30
```

```
np.arange(1,31).reshape(6,5)
```

```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15],
       [16, 17, 18, 19, 20],
       [21, 22, 23, 24, 25],
       [26, 27, 28, 29, 30]])
```

```
[[11 12]
 [16 17]]
```

```
array([[ 0.,  2.,  0.,  0.,  0.],
       [ 0.,  0.,  8.,  0.,  0.],
       [ 0.,  0.,  0., 14.,  0.],
       [ 0.,  0.,  0.,  0., 20.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

```
array([[ 0.,  0.,  0.,  4.,  5.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0., 24., 25.],
       [ 0.,  0.,  0., 29., 30.]])
```

## (Numpy) : Task 2: Exercises

**Part 1**

1. Create a one-dimensional NumPy array containing numbers from 0 to 9.
2. Create a 3x3 identity matrix using NumPy.
3. From a NumPy array containing numbers from 0 to 9, extract all the odd numbers.

**Part 2**

1. Convert a one-dimensional NumPy array containing numbers from 1 to 12 into a 3x4 two-dimensional array.
2. Create a 5x5 array with random values and the values range between 0 and 1.
3. Create a 5x3 matrix and a 3x2 matrix, then compute their matrix product.
4. In a one-dimensional array containing numbers from 0 to 10, replace all values between 3 and 8 with their negatives.