Lecture 2: Introduction to numpy

1 What's Numpy?

- **NumPy** is a Python library used for scientific computing and data analysis.
- It provides efficient tools for working with large arrays and matrices of numerical data, In particular, NumPy provides objects such as n-dimensional arrays.
- Libraries written in lower-level languages, such as C, can operate on data stored in Numpy 'ndarray' without copying any data.
- NumPy also includes a wide range of mathematical functions for performing various mathematical operations, such as linear algebra, Fourier analysis, and statistics.
- NumPy facilitates and optimizes the storage and manipulation of numerical data, especially when dealing with large arrays. This is known as "array-oriented computing".
- In summary, the NumPy module is the basic tool used in all scientific and numerical calculations in Python due to its power, speed, and flexibility.

2 How to install NumPy:

You can use the pip package manager to install NumPy by running the command

```
pip install numpy
```

in the command prompt or Anaconda prompt. # How to import NumPy: To use NumPy in your Python code, you first need to import it using the `import` statement. You can import NumPy by adding the following line at the top of your Python script or Jupyter Notebook:

```
import numpy as np
```

This statement creates an alias "np" for NumPy, which is a common convention used by most developers working with NumPy. # Documentation - You can use https://docs.scipy.org/doc/. - Asking For Help:

```
np.info(np.ndarray.dtype)
```

- For interactive help, you can use the symbol `?`, for example:

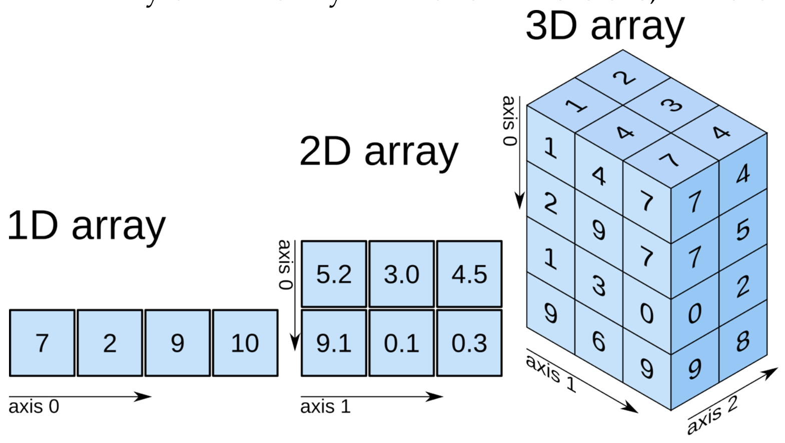
```
np.array?
```

```
Docstring:
array(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0, like=None)

Create an array.
...
```

3 NumPy N-dimensional array (ndarray):

- NumPy's ndarray is a multi-dimensional container for homogeneous numerical data, meaning that all elements in the array should be of the same data type.
- An ndarray can have any number of dimensions, and each dimension is called an "axis".



3.1 How to create ndarray:

Ndarrays are created using the `numpy.array` function, which takes a list or tuple of values as input and returns an ndarray object.

```
a = np.array([1,2,3]) #1D array
b = np.array([(1.5,2,3), (4,5,6)], dtype = float) #2D array
c = np.array([[(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]], dtype = float) #3D array
```

3.2 Some important NumPy array creation functions:

function	Description
`np.zeros`	Produce an array of all *0s* with the given shape and data type
`np.ones`	Produce an array of all *1s* with the given shape and data type
`np.arange`	Like the built-in range but returns an ndarray instead of a list
`np.linspace`	Create an array of evenly spaced values (number of samples)
`np.full`	Produce an array of the given shape and data type with all values set to the indicated 'fill value';
`np.eye`*or*`np.identity`	Create a square N × N identity matrix (1s on the diagonal and 0s elsewhere)
`np.random.random`	Create an array with random values
`np.empty`	Create new arrays by allocating new memory, but do not populate with any values like ones and zeros

examples:

```
np.zeros((3,4)) #Create an array of zeros
np.ones((2,3,4),dtype=np.int16) #Create an array of ones
np.arange(10,25,5) #Create an array of evenly spaced values (step value)
np.linspace(0,2,9) #Create an array of evenly spaced values (number of samples)
np.full((2,2),7) #Create a constant array
np.eye(2) #Create a 2X2 identity matrix
np.random.random((2,2)) #Create an array with random values
np.random.randint(2, size=(3,3))
np.empty((3,2)) #Create an empty array
```

3.3 Why not just use Python lists for calculations instead of creating a new type of array using numpy?

There are several reasons for this: - Python lists are very general (also called high-level objects). They can contain any object ⇒ dynamic typing. They do not support mathematical operations. - NumPy arrays are statically typed and homogeneous. - The type of the elements is determined when the array is created ⇒ no more dynamic typing. - Similarly, the size of the array is fixed at creation ⇒ optimized memory storage. - Due to static typing, mathematical functions such as matrix multiplication and addition can be implemented using a compiled language (C and Fortran).

Example:

```
%timeit [i**2 for i in range(1000)]
a = np.arange(1000)
%timeit a**2
```

3.4 Data types in numpy:

NumPy provides several data types that can be used to create arrays.

Type	*Code*	*Decription*
bool	?	Boolean (True or False) stored as a byte
*int*8	*i*1	Byte (-128 to 127)
*int*16	*i*2	Integer (-32768 to 32767)
*int*32	*i*4	Integer (-2 ** 31 to 2 ** 31 - 1)
*int*64	*i*8	Integer (-2 ** 63 to 2 ** 63 - 1)
*uint*8	*u*1	Unsigned integer (0 to 255)
*uint*16	*u*2	Unsigned integer (0 to 65535)
*uint*32	*u*4	Unsigned integer (0 to 2 ** 32 - 1)
*uint*64	*u*8	Unsigned integer (0 to 2 ** 64 - 1)
*float*16	*f*2	Half precision float
*float*32	*f*4	Single precision float
*float*64	*f*8	Double precision float
*complex*64	*c*8	Complex number, represented by two 32-bit floats
*complex*128	*c*16	Complex number, represented by two 64-bit floats
*string_*	*S*	Fixed-length ASCII string type (1 byte per character); for example, to create a string data type with length 10, use 'S10'

Example:

```
a = np.array([1, 2, 3], dtype=?)
print(a)
a = np.array([1, 2, 3], dtype=f4)
print(a)
```

3.5 Getting arrays dimensions and Data type in numpy

- To get the dimensions of an array, we can use the `shape` attribute. For example, for an array `a`, we can get its dimensions as follows:

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

- To get the number of dimensions of an array, we can use the `ndim` attribute. For example:

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

- To get the data type of an array, we can use the `dtype` attribute. For example:

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

- We can also explicitly convert the data type of an array using the `astype()` method. For example:

```
a = np.array([1, 2, 3], dtype=np.int32)
b = a.astype(np.float32)
print(b.dtype)
```

3.6 Array mathematics in numpy:

3.6.1 Arithmetic Operations

Subtraction, Addition, Division, Multiplication (item per item): NumPy provides element-wise arithmetic operations for arrays. You can perform addition, subtraction, multiplication, and division on arrays by simply using the appropriate operator (`+`, `-`, `*`, `/`) between them. The operations are performed on each corresponding element of the arrays. For example:

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

# Addition
c = a + b
print(c)  # Output: [5 7 9]

# Subtraction
d = a - b
print(d)  # Output: [-3 -3 -3]

# Multiplication
e = a * b
print(e)  # Output: [ 4 10 18]

# Division
f = a / b
print(f)  # Output: [0.25 0.4  0.5 ]
```

element-wise mathematical functions: NumPy provides several mathematical functions that operate element-wise on arrays. For example,

```
import numpy as np

a = np.array([1, 4, 9])

# Square root
b = np.sqrt(a)
print(b)  # Output: [1. 2. 3.]

# Sine
c = np.sin(a)
print(c)  # Output: [ 0.84147098 -0.7568025   0.41211849]

# Cosine
d = np.cos(a)
print(d)  # Output: [ 0.54030231 -0.65364362 -0.91113026]
```

3.6.2 Comparison

In NumPy, we can compare arrays element-wise using comparison operators such as `==`, `!=`, `<`, `>`, `<=`, and `>=`. When performing element-wise comparisons, the result is a boolean array of the same shape as the original arrays. For example:

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

print(a == b)  # [ True  True False]
print(a != b)  # [False False  True]
print(a < b)   # [False False  True]
print(a > b)   # [False False False]
print(a <= b)  # [ True  True  True]
print(a >= b)  # [ True  True False]
```

we can also perform element-wise comparisons between a scalar value and an array:

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

print(a == b)  # [False  True False]
print(a != b)  # [ True False  True]
print(a < b)   # [ True False False]
print(a > b)   # [False False  True]
print(a <= b)  # [ True  True False]
print(a >= b)  # [False  True  True]
```

any(), all(), and sum() with condition:

- `all()` returns True if all the elements in the input array evaluate to True. Otherwise, it returns False.
- `any()` returns True if any of the elements in the input array evaluate to True. Otherwise, it returns False.

Here are some examples:

```
import numpy as np

a = np.array([1, 2, 3, 4])
b = np.array([0, 2, 4, 6])
c = np.array([-1, 2, -3, 4])

# using all() method
print(np.all(a > 0))   # True
print(np.all(b > 0))   # False
print(np.all(c > 0))   # False

# using any() method
print(np.any(a == 2))  # True
print(np.any(b == 1))  # False
print(np.any(c == -3)) # True
```

- In NumPy, we can perform sum with condition using the `np.sum()` function with a boolean mask as input. The boolean mask is created using a comparison operator (`>`, `<`, `==`, etc.) on an array, which creates a boolean array of the same shape with True where the condition is met and False where it is not.

```
import numpy as np

# Create an array
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Sum all elements that are greater than 5
sum_greater_than_5 = np.sum(arr[arr > 5])

print(sum_greater_than_5) # Output: 30
```

3.6.3 Arrays statistics functions:

NumPy provides many built-in statistical functions that can be used to perform various statistical operations on arrays. Here are some of the commonly used statistical functions in NumPy:

function	*Description*
np.mean()	Calculates the arithmetic mean of an array along a specified axis.
np.median()	Computes the median of an array along a specified axis.
np.std()	Computes the standard deviation of an array along a specified axis.
np.var()	Computes the variance of an array along a specified axis.
np.min()	Computes the minimum value of an array along a specified axis.
np.max()	Computes the maximum value of an array along a specified axis.
np.argmin()	Return the index of the minimum value
np.argmax()	Return the index of the maximum value
np.sum()	Computes the sum of all the elements in an array along a specified axis.
np.prod()	Computes the product of all the elements in an array along a specified axis.
np.cumsum()	Computes the cumulative sum of the elements in an array along a specified axis.

Here's an example of how to use some of these functions:

```
import numpy as np

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

print(np.mean(a))           # 5.0
print(np.median(a))         # 5.0
print(np.std(a))            # 2.581988897471611
print(np.var(a))            # 6.666666666666667
print(np.min(a,axis=1))      # [1 4 7]
print(np.max(a))            # 9
print(np.sum(a))            # 45
print(np.prod(a))           # 362880
print(np.cumsum(a))         # [ 1  3  6 10 15 21 28 36 45]
```

3.6.4 Arrays algebraic and form operations:

Dot product The `dot()` function in NumPy performs matrix multiplication or dot product of two arrays.

```
A = np.array([[1, 2], [3, 4], [5, 6]])
B = np.array([[7, 8], [9, 10]])

# Matrix multiplication of A and B
C = np.dot(A, B )# or C = A @ B
print(C)
```

Transposition The `transpose()` function can be called on a NumPy array to create a new array with the rows and columns flipped. For example, the following code creates a 2D array arr and then transposes it using the `transpose()` function:

```
arr = np.array([[1, 2], [3, 4]])
transposed_arr = np.transpose(arr)
print(transposed_arr)
```

Alternatively, the `T` attribute can be used to transpose an array. For example:

```
arr = np.array([[1, 2], [3, 4]])
transposed_arr = arr.T
print(transposed_arr)
```

Reshaping: Changing the shape of an array while maintaining the same number of elements.

```
import numpy as np
a = np.array([[1, 2], [3, 4], [5, 6]])
print("Original array:\n", a)
# Output:
# Original array:
#  [[1 2]
#   [3 4]
#   [5 6]]

b = a.reshape((2, 3))
print("Reshaped array:\n", b)
# Output:
# Reshaped array:
#  [[1 2 3]
#   [4 5 6]]
```

Broadcasting: Performing operations on arrays with different shapes and sizes.

```
import numpy as np
a = np.array([1, 2, 3])
b = np.array([[4], [5], [6]])
c = a + b
print("Result array:\n", c)
# Output:
# Result array:
#  [[5 6 7]
#   [6 7 8]
#   [7 8 9]]
```

Concatenation: Joining two or more arrays along a specified axis.

```
import numpy as np
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
c = np.concatenate((a, b), axis=0)
print("Concatenated array:\n", c)
# Output:
```

```
# Concatenated array:
#  [[1 2]
#   [3 4]
#   [5 6]]
```

Stacking: Joining two or more arrays along a new axis.

```
import numpy as np
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = np.stack((a, b), axis=0)
print("Stacked array:\n", c)
# Output:
# Stacked array:
#  [[1 2 3]
#   [4 5 6]]
```

3.6.5 Splitting:

Splitting an array into smaller arrays along a specified axis.

```
import numpy as np
a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
b, c = np.split(a, 2, axis=0)
print("First split array:\n", b)
print("Second split array:\n", c)
# Output:
# First split array:
#  [[1 2 3]
#   [4 5 6]]
# Second split array:
#  [[ 7  8  9]
#   [10 11 12]]
```

3.6.6 flattened array

`ravel()` is a function in NumPy that returns a flattened array. It means that it returns a 1-dimensional array that contains all the elements of the input array. The returned array is always a copy of the original array, so modifying it does not affect the original array.

Here is an example of how to use `ravel()`:

```
import numpy as np

# Create a 2-dimensional array
arr = np.array([[1, 2], [3, 4]])

# Flatten the array using ravel()
flat_arr = arr.ravel()
```

```
# Print the original and flattened arrays
print("Original array:\n", arr)
print("Flattened array:\n", flat_arr)
```

3.7 Copying Arrays

In NumPy, arrays can be copied in different ways, depending on the desired behavior. Here are some ways to copy arrays in NumPy:

1- **Shallow copy**: In this type of copy, a new array object is created, but the data is not copied. Instead, a reference to the original data is used. Shallow copy can be made using the `view()` method.

2- **Deep copy**: In this type of copy, a completely new array and data are created. Deep copy can be made using the `copy()` method. Example:

```
import numpy as np

arr1 = np.array([1, 2, 3])
arr2 = arr1.view()

print(arr1)
print(arr2)

arr2[0] = 0
print("Shallow copy")
print(arr1)
print(arr2)

arr2=arr1.copy()
arr2[2] = 4
print("Deep copy")
print(arr1)
print(arr2)
```

3.8 Sorting Arrays

Sorting an array in numpy can be done using the `sort()` function. By default, it sorts the array in ascending order.

Here is an example:

```
import numpy as np

arr = np.array([3, 2, 1, 4, 5])
arr.sort()

print(arr)
#output
#[1 2 3 4 5]
```

If you want to sort a two-dimensional array, you can use the `sort()` function with the axis parameter. By default, it sorts each row independently.

Here is an example:

```
import numpy as np

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

arr.sort(axis=1)
#output
#[[1 2 3]
# [4 5 6]
# [7 8 9]]
print(arr)
```

3.9 Broadcasting:

- Broadcasting is a way to perform operations between arrays of different shapes in numpy, by implicitly creating temporary arrays that are compatible with the original arrays.
- Numpy broadcasts arrays by comparing their shapes element-wise, and applying the following rules:
  - If the shapes of the two arrays are equal, they are compatible.
  - If one of the arrays has a size of 1 for a particular dimension, and the other array has a size greater than 1 for that dimension, they are compatible.
  - If neither of the above conditions is true, a ValueError is raised. **It's important to note that in broadcasting, the comparison of indices is done from the right to the left**. ### Examples:

1. Broadcasting a scalar value to a matrix:

```
import numpy as np

# create a 3x3 matrix
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# broadcast the scalar value 2 to the matrix
result = matrix + 2

print(result)
#output array([[ 3,  4,  5],
#              [ 6,  7,  8],
#              [ 9, 10, 11]])
```

2. Broadcasting a vector to a matrix:

```
import numpy as np

# create a 3x3 matrix
```

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

# create a 1-dimensional vector
vector = np.array([10, 20, 30])

# broadcast the vector to the matrix
result = matrix + vector

print(result)
#output
#array([[11, 22, 33],
#        [14, 25, 36],
#        [17, 28, 39]])
```

3. Broadcasting a vector to a 2-dimensional array:

```
import numpy as np

# create a 2-dimensional array with shape (3, 2)
arr = np.array([[1, 2], [3, 4], [5, 6]])

# create a vector with shape (2,)
vec = np.array([10, 20])

# broadcast the vector to the array
result = arr + vec

print(result)
#Output:
#array([[11, 22],
#        [13, 24],
#        [15, 26]])
```

3.10 Indexing and Slicing in numpy:

- **Indexing** and **slicing** are two methods used in NumPy to access and manipulate elements and subarrays within arrays. The standard **square bracket** `[]` notation is used for both, similar to Python lists.
  - Indexing is the process of accessing individual elements of an array. To index a NumPy array, you specify the index position of the element you want to access along each axis (dimension) of the array. The index positions start from 0 and go up to one less than the length of the axis.
  - Slicing is the process of accessing a subarray of an array. To slice a NumPy array, you specify a range of index positions for each axis. The syntax for slicing is `[start:stop:step]`, where start is the index position to start the slice, stop is the index position to end the slice (exclusive), and step is the size of the stride between the indices. You can omit any of these parameters, and the defaults will be used (start=0, stop=length of axis, step=1). ### One-Dimensional Arrays

Here is the 1D-Array Indexing and Slicing possible expressions:

Expression	*Description*
$a[m]$	Select element at index m, where m is an integer (start counting form 0).
$a[-m]$	Select the nth element from the end of the list, where n is an integer. The last element in the list is addressed as -1, the second to last element as -2, and so on.
$a[m:n]$	Select elements with index starting at m and ending at n - 1 (m and n are integers).
$a[:]ora[0:-1]$	Select all elements in the given axis.
$a[:n]$	Select elements starting with index 0 and going up to index n - 1 (integer).
$a[m:]ora[m:-1]$	Select elements starting with index m (integer) and going up to the last element in the array.
$a[m:n:p]$	Select elements with index m through n (exclusive), with increment p.
$a[::-1]$	Select all the elements, in reverse order.

3.10.1 Two-Dimensional Arrays:

Here are the possible expressions for indexing and slicing multidimensional arrays in NumPy

Expression	*Description*
$a[i,j]$	Select element at row i and column j, where i and j are integers (start counting from 0).
$a[i][j]$	Another way to select element at row i and column j, but less efficient than the previous expression.
$a[i]$	Select row at index i.
$a[:,j]$	Select all rows at column j.
$a[i,:]$	Select all columns at row i.
$a[start\_row:end\_row,start\_col:end\_col]$	Select a subarray that includes the rows starting from start_row (inclusive) and ending at end_row (exclusive), and the columns starting from start_col (inclusive) and ending at end_col (exclusive). You can omit any of these parameters, and the defaults will be used (start=0, end=end of axis).
$a[start\_row:end\_row:step]$	Select a subarray with a step size between the indices.
$a[::-1,::-1]$	Select all elements, but in reverse order along both axes.

Note that you can apply these indexing and slicing expressions to higher-dimensional arrays as well, by simply adding more indices separated by commas.

Examples:

```
import numpy as np

# 2D array indexing
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr_2d[1][2]) # output: 6
print(arr_2d[1, 2]) # output: 6
```

```
# 2D array slicing
print(arr_2d[:2, 1:]) # output: [[2 3]
                      #          [5 6]]

# 3D array indexing
arr_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(arr_3d[0, 1, 0]) # output: 3

# 3D array slicing
print(arr_3d[:1, 1:, :]) # output: [[[3 4]]]
```

Exercise 1: Broadcasting Compatibility

- Determine if the following operations are possible using broadcasting:
  - Array A with shape (3, 4) and array B with shape (4,) are added together.
  - Array A with shape (3, 1) and array B with shape (3,) are added together.
  - Array A with shape (3, 4) and array B with shape (4, 3) are added together.
  - Array A with shape (2, 3, 4) and array B with shape (3, 4) are added together.
  - Array A with shape (2, 3, 4) and array B with shape (2, 1, 4) are added together.
- **Solutions:**
  - Possible. Array B will be broadcasted to shape (3, 4) and then added to array A.
  - Possible. Array B will be broadcasted to shape (3, 1) and then added to array A.
  - Not possible. The shapes of the two arrays are incompatible because their dimensions do not match.
  - Possible. Array B will be broadcasted to shape (2, 3, 4) and then added to array A.
  - Possible. Array B will be broadcasted to shape (2, 3, 4) and then added to array A.

Exercise 2: Indexing and Slicing

- Given the following NumPy array:

  ```
  import numpy as np

  A = np.array([[1, 2, 3, 4],
                [5, 6, 7, 8],
                [9, 10, 11, 12]])
  ```

  Perform the following indexing and slicing operations:
  - Select the element at row 0, column 2.
  - Select the last element in the last row.
  - Select the first two rows.
  - Select the last two columns.
  - Select the subarray from row 1 to row 2, and from column 1 to column 3.
- **Solution 2:**

```
import numpy as np

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

# 1. Select the element at row 0, column 2.
print(A[0, 2])

# 2. Select the last element in the last row.
print(A[-1, -1])

# 3. Select the first two rows.
print(A[:2, :])

# 4. Select the last two columns.
print(A[:, -2:])

# 5. Select the subarray from row 1 to row 2, and from column 1 to column 3.
print(A[1:3, 1:4])
```

3.10.2 Advanced Indexing in numpy

Advanced indexing in NumPy is a powerful feature that allows for more complex and flexible indexing of arrays. There are two main types of advanced indexing: boolean indexing and fancy indexing. #### 1. Boolean Indexing: Boolean indexing allows you to select elements of an array based on a boolean condition. The condition is usually based on some comparison of the array's values with a scalar or another array. The result of the comparison is a boolean array of the same shape as the original array, where each element is True or False depending on whether the condition is met or not. This boolean array can then be used as an index for selecting the elements of the original array.

For example, let's say we have an array `arr`:

```
arr = np.array([1, 2, 3, 4, 5])
```

We can create a boolean array based on a condition, such as:

```
mask = arr > 2
```

This will create a boolean array of the same shape as `arr`, where each element is True if the corresponding element in `arr` is greater than 2, and False otherwise:

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

We can then use this boolean array as an index for selecting elements of `arr`:

```
new_arr = arr[mask]
```

This will create a new array `new_arr` containing only the elements of arr where the corresponding element in `mask` is True:

```
array([3, 4, 5])
```

We can simply get the `new_arr` by:

```
arr=np.array([1,2,3,4,5])
new_arr=arr[arr>2]
print(new_arr)
```

2. Fancy Indexing

```
Fancy indexing: Fancy indexing allows you to select elements of an array using integer arrays as
This means that you can specify exactly which elements you want to select, and in what order, by
```

For example, let's say we have an array arr:

```
arr = np.array([1, 2, 3, 4, 5])
```

We can use fancy indexing to select specific elements of `arr`, by passing in a list of indices:

```
indices = [1, 3, 4]
new_arr = arr[indices]
```

This will create a new array new_arr containing only the elements of arr with indices 1, 3, and 4:

```
array([2, 4, 5])
```

We can also use fancy indexing to select specific elements of a multi-dimensional array. In this case, we need to pass in one or more arrays of indices, where each array corresponds to a different axis of the array. For example, let's say we have a 2D array arr:

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

We can use fancy indexing to select specific elements of `arr`, by passing in two arrays of indices:

```
row_indices = [0, 1, 2]
col_indices = [1, 2, 0]
new_arr = arr[row_indices, col_indices]
```

This will create a new array new_arr containing the elements of arr at the positions (0, 1), (1, 2), and (2, 0):

```
array([2, 6, 7])
```

Exercise: Extract from the array `np.array([3,4,6,10,24,89,45,43,46,99,100])` with Boolean masking all the number

- which are not divisible by 3.
- which are divisible by 5.
- which are divisible by 3 and 5.
- which are divisible by 3 and set them to 42.
  Solution
  ```
  import numpy as np
  A = np.array([3,4,6,10,24,89,45,43,46,99,100])
  div3 = A[A%3!=0]
  print("Elements of A not divisible by 3:")
  print(div3)
  div5 = A[A%5==0]
  ```

```
print("Elements of A divisible by 5:")
print(div5)
print("Elements of A, which are divisible by 3 and 5:")
print(A[(A%3==0) & (A%5==0)])
print("------------------")
#
A[A%3==0] = 42
print("""New values of A after setting the elements of A,
which are divisible by 3, to 42:""")
print(A)
```

3.11 Reading and Writing Data files in numpy

-

3.11.1 savetxt:

This function allows you to save an array to a text file. You can specify the file name, delimiter, and various other options. Here's an example:

```
import numpy as np

# create an array
a = np.array([[1, 2, 3], [4, 5, 6]])

# save the array to a text file
np.savetxt('myarray.txt', a, delimiter=',')
```

This will create a file called myarray.txt with the following contents:

```
1.000000000000000000e+00,2.000000000000000000e+00,3.000000000000000000e+00
4.000000000000000000e+00,5.000000000000000000e+00,6.000000000000000000e+00
```

-

3.11.2 loadtxt:

This function allows you to load data from a text file into an array. You can specify the file name, delimiter, and various other options. Here's an example: "'python import numpy as np

# load data from a text file a = np.loadtxt('myarray.txt', delimiter=',')

# print the array print(a) "' This will print:

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

-

3.11.3 tofile:

This function allows you to save an array to a binary file. You can specify the file name and various other options. Here's an example:

```
import numpy as np

# create an array
a = np.array([[1, 2, 3], [4, 5, 6]])

# save the array to a binary file
a.tofile('myarray.bin')
```

•

3.11.4 fromfile:

This function allows you to load data from a binary file into an array. You can specify the file name and various other options. Here's an example:

```
import numpy as np

# load data from a binary file
a = np.fromfile('myarray.bin', dtype=np.int32)

# reshape the array
a = a.reshape((2, 3))

# print the array
print(a)
```

This will print:

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

•

3.11.5 save:

This function allows you to save an array to a NumPy binary file. You can specify the file name and various other options. Here's an example:

```
import numpy as np

# create an array
a = np.array([[1, 2, 3], [4, 5, 6]])

# save the array to a NumPy binary file
np.save('myarray.npy', a)
```

•

3.11.6 load:

This function allows you to load data from a NumPy binary file into an array. You can specify the file name and various other options. Here's an example: “‘python import numpy as np

# load data from a NumPy binary file a = np.load(‘myarray.npy’)

# print the array print(a) “‘

This will print:

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