CodeMite's guide through the adventures – with **NumPy** –

A beginner friendly tutorial book
– with easy examples –

– made by Najam –

NumPy — Complete Guide

NumPy

🥳 **NumPy** brings the computational power of languages like **C** and **Fortran** to **Python**, a language much easier to learn and use.

To use `numpy` in your code, `import` it as `np`.

```python
import numpy as np
```

NumPy Arrays

🤩 **NumPy** provides a way to create *arrays of fixed size* and of the *same data type*. This allows very fast processing on the data as compared to *Python lists*.

☝️ NumPy arrays have the following characteristics:

- 👉 Fixed sized
- 👉 Same datatype
- 👉 Can be 1 or n-dimensional

NumPy Arrays

⚡ To create a **NumPy array** from existing data, you can give **any iterable** to its `array` function. It will return a *NumPy array*.

⚠️ ***But the data in the iterable must be of the same type, otherwise it will raise an exception.***

☝️ The following are a few ways of doing that:

👉 `array1 = np.array([1,2,3,4])`

👉 `array2 = np.array(range(10))`

👉 `array3 = np.array(existing_pylist)`

NumPy Arrays

☝️ A **NumPy array** has the following properties:

- 👉 `ndim` – Number of dimensions of the array
- 👉 `size` – Size (number of elements) of the array
- 👉 `dtype` – Datatype of the array
- 👉 `shape` – Dimensions as a tuple

NumPy Arrays: Creation

🐍 **NumPy** gives lots of ways to create arrays. We can opt from them according to our requirements.

⚡ In the following slides, we are going to be looking at them in detail.

NumPy Arrays: Creation

🐍 **NumPy** has a function called `arange`. This does what the **Python's range** function does but *NumPy* suggests that we use *its* variant whenever possible.

Here's how it would go:

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

```
array = np.arange(10, 20, 2)
# array contains:
#    [10 12 14 16 18]
```

NumPy Arrays: Creation

🐍 **np.zeros**(size | shape, dtype=‘float64’)

```
array = np.zeros(5)
# array contains:
#     [0. 0. 0. 0. 0.]
```

```
array = np.zeros(5, dtype='int32’)
# array contains:
#     [0 0 0 0 0]
```

```
array = np.zeros([2,3])
# array contains:
#     [[0. 0. 0.]
#      [0. 0. 0.]]
```

NumPy Arrays: Creation

🐍 **np.ones**(size | shape, dtype=‘float64’)

```
array = np.ones(5)
# array contains:
#    [1. 1. 1. 1. 1.]
```

```
array = np.ones(5, dtype='int32’)
# array contains:
#    [1 1 1 1 1]
```

```
array = np.ones([2,3])
# array contains:
#    [[1. 1. 1.]
#     [1. 1. 1.]]
```

NumPy Arrays: Creation

🐍 **np.full**(size | shape, fill_value**)**

```
array = np.full(5, fill_value=4)
# array contains:
#     [4 4 4 4 4]

array = np.full(10, fill_value=3, dtype='uint32’)
# array contains:
#     [3 3 3 3 3 3 3 3 3 3]
```

NumPy Arrays: Creation

🐍 **np.linspace**(start, stop, num=50, dtype=‘float64’)

```
array = np.linspace(0, 10, 5)
# array contains:
#     [ 0.   2.5  5.   7.5 10. ]

array = np.linspace(46, 50, 10, dtype='int16’)
# array contains:
#     [46 46 46 47 47 48 48 49 49 50]
```

NumPy Arrays: Creation

🐍 **np.random.random**(size | shape)

```python
array = np.random.random(5)
# array contains (random):
#     [0.71347375 0.37824523
#        0.59149209 0.20544824 0.118407  ]
array = np.random.random([2, 3])
# array contains (random):
#     [[0.81139571 0.97613414 0.29534939]
#      [0.25330936 0.40710077 0.65493009]]
```

NumPy Arrays: Creation

🐍 **np.random.randint**(low=0, high, size=size | shape)

```
array = np.random.randint(5, size=4)
# array contains (random):
#    [0 2 3 1]

array = np.random.randint(50, 100, size=[2,3])
# array contains (random):
#    [[61 73 77]
#     [64 83 82]]
```

NumPy Arrays: Creation

🐍 **np.eye**(N, k=0) # k is the index of the diagonal

```
array = np.eye(2)
# array contains (identity matrix):
#    [[1. 0.]
#     [0. 1.]]
array = np.eye(3, k=1)
# array contains (identity matrix):
#    [[0. 1. 0.]
#     [0. 0. 1.]
#     [0. 0. 0.]]
```

NumPy Arrays: Creation

🐍 **np.empty**(shape=size | shape, dtype=None)

```
array = np.empty(6)
# array contains (six uninitialized elements):
#    (whatever was there in the memory before)

array = np.empty([2,3])
# array contains (2x3 uninitialized elements):
#    (whatever was there in the memory before)
```

NumPy: Recap

Let's recap everything we have learned so far.

- `np.arange` – *Creates an array of a given range.*
- `np.zeros` – *Creates an array of given size or shape (dimensions) and fills all the values as zero (default is* **`float`** *but we can change it by specifying the* **`dtype`** *argument.*
- `np.ones` – *Creates an array just like* `np.zeros()` *but it initializes each element by one instead of a zero.*
- `np.full` – *Creates an array of given size or shape and it takes one more argument which it would use to initialize each element of the array.*

NumPy: Recap

- **`np.linspace`** – *Creates an array of a given range (first two arguments) and spreads the values evenly in that range by the given number of times (third argument).*
- **`np.eye`** – *Creates an array as an identity matrix of any given size. By default, the diagonal starts from the first index but you can change that by using the **k** argument.*
- **`np.empty`** – *Creates an uninitialized array of a given size or shape.*
- **`np.random.randint`** – *Creates an array of integers in a given range and of any given size or shape.*
- **`np.random.random`** – *Creates an array of floats of any given size or shape.*

NumPy Arrays: Indexing

🤩 **NumPy** arrays can be *indexed* for *accessing* single elements just like the native **lists/arrays of Python**. Let's bring our array object and see this in action!

```
array = np.arange(10)

array[0] # ————will give back <0> (value at index 0)
array[8] # ————will give back <8> (value at index 8)
array[4] = 44 # will overwrite the previous value at index 4
```

NumPy Arrays: Indexing

🤩 **NumPy** arrays are indexed a bit differently for n-dimensions. Let's see!

```
array = np.arange(10, 20).reshape(2, 5) # 2D array (2x5)

array[1, 3] # -----will give back array[1][3]
array[0, 0] # -----will give back array[0][0]
array[1, 2] = 55 # will overwrite array[1][2]
```

NumPy Arrays: Slicing

🤩 **NumPy** arrays can be *sliced* just like the native **lists/arrays of Python** but there's some extra syntactic + performance features.

Unlike the native slicing, **NumPy slicing doesn't create a copy** but instead gives a ***view into the original memory***.

That means two things:

- ➢ Faster slicing as no copying is needed.
- ➢ The sliced array, if modified, would affect the 'original' array as well because they're basically the same memory.

NumPy Arrays: Slicing

😎 Just like indexing, NumPy arrays are sliced a bit differently. Let's see that!

```
array = np.arange(10, 20)

array[:2] #  view into [0, 1] indexes
array[::2] # view into every other index

array = array.reshape(2, 5)

array[:2, :3] # view into first 2 rows and 3 columns
```

NumPy Arrays: Copying

🤔We just saw that slices do not return a copy but instead a reference to the same memory. ***But what if we did want a copy?*** NumPy has a solution for that as well. We can use its `copy` method to make a copy of an entire array or a slice even.

```
array = np.empty([2,5])

copy = array.copy() # --------copy of the entire array
copy = array[0].copy() # -----copy of the first row
copy = array[:1, :3].copy() # copy of the 0th row and
                            ## first 3 columns
```

NumPy Arrays: Concat

🥳 NumPy arrays can be concatenated with each other and with great flexibility. We can use its `concatenate` function to join arrays of same shape. To join arrays with different shapes, there's an option of `vstack` and `hstack` as well.

```
x = np.array([1,2,3,4])
y = np.array([5,6,7,8])
z = np.concatenate([x, y])
# z = [1 2 3 4 5 6 7 8]
```

NumPy Arrays: Concat

```
x = np.array([[1,2,3,4],[1,2,3,4]])
y = np.array([5,6,7,8])
z = np.concatenate([x, y]) # ERROR - shapes are not same
```

The solution to this problem are `vstack` and `hstack` functions of the **NumPy** library.

- ⚡ **`np.vstack`** – Concatenates the arrays vertically (number of columns must be the same).
- ⚡ **`np.hstack`** – Concatenates the arrays horizontally (number of rows must be the same).

NumPy Arrays: Concat

```
x = np.array([[1,2,3,4],[1,2,3,4]])
y = np.array([5,6,7,8])
z = np.vstack([x, y]) # OK - both arrays have 4 columns (same)

# z is now equal to
# this array:
#    [[1 2 3 4]
#     [1 2 3 4]
#     [5 6 7 8]]
```

NumPy Arrays: Concat

```
x = np.array([[1,2,3,4],[1,2,3,4]])
y = np.array([[5,6],[7,8]])
z = np.hstack([x, y]) # OK - both arrays have 2 rows (same)

# z is now equal to
# this array:
#    [[1 2 3 4 5 6]
#     [1 2 3 4 7 8]]
```

NumPy Arrays: Reshape

🤓 **NumPy arrays** can be reshaped! We already know that a *NumPy array has a shape*. This shape can be modified to something else is what the `np.reshape` function does.

⚠️ ***BUT! The new shape must match the original size!***

One of the best uses of the reshape function is to add a dimension to an array. Let's suppose, we have:

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

NumPy Arrays: Reshape

This array, as of now, has the `shape=(4,)` which means it's, obviously, a 1D array. If we wanted to make it a 2D array, we could do:

```
array = array.reshape([1,4])
```

The 4 came from the previous `size` and the new `shape` has now become `(1,4)` which hasn't changed the `size` which would still be 4. But doing this has made some changes:

```
# array = [[1 2 3 4]] - a new dimension has been added
```

NumPy Arrays: Reshape

That's not all though. Let's see some more examples to further clear the concept:

```
array = np.arange(10).reshape([2,5])
# array = [[0 1 2 3 4]
#          [5 6 7 8 9]]
```

☝️ Notice that the new shape adds up to be the original size (i.e., 2x5=10).

Similarly, we could do it like:

```
array = np.arange(10).reshape([5,2]) # OK
array = np.arange(10).reshape([1,5,2]) # OK
```

NumPy Arrays: Transpose

NumPy gives the ability to take transpose of any `np.array`. If you know basic linear algebra, you're already familiar with the concept. To take a transpose, you have to use the **T** attribute with any NumPy array.

Simply put, taking a transpose of an `np.array` reverses the shape of the array. Let's see some examples 🤩

```
array = np.array([[1,2,3,4],[1,2,3,4]]) # shape=(2,3)
array = array.T # shape=(3,2)
```

NumPy Arrays: Transpose

🔥 This works with any shape. Let's some more examples:

```
array = np.arange(30).reshape([3,2,5]) # shape=(3,2,5)
array = array.T # shape=(5,2,3)
```

And just like that, you can use this amazing feature of the NumPy library. This is very helpful when working with *datasets* that may be of different shapes.

⚡ Previously we saw the concatenation of arrays but there were some conditions to concatenate them. *Transpose could solve that problem as well.*

NumPy Arrays: Arithmetic

NumPy provides all the arithmetic operations on its arrays natively. Every arithmetic operation that you know of in native Python is also in NumPy and then some.

This makes performing arithmetics on arrays so easy. All of these operations are applied to the *entire arrays* and a new array is returned. Let's see it with examples to better understand it.

NumPy Arrays: Arithmetic

```python
numbers = np.arange(10)
array = numbers + 3
# This will add 3 to each element
# and return a new array
```

All of these operators can also be used by calling their functions directly. There's so many of them but the most common ones are the following:

NumPy Arrays: Arithmetic

Function	Operator	Purpose
`np.add`	`+`	Addition
`np.subtract`	`-`	Subtraction
`np.multiply`	`*`	Multiplication
`np.divide`	`/`	Division
`np.floor_divide`	`//`	Integer Division
`np.pow`	`**`	Power
`np.mod`	`%`	Modulo/Remainder

NumPy Arrays: Arithmetic

All of these functions take an argument called `out=`. We can specify our output object where we want to put the results of the operation in.

The previous process could've been done like this as well:

```
np.add(numbers, 5, out=array)
```

It's very helpful because it works with slices as well:

```
numbers = np.arange(10)
array = np.full(10, 0)
output = np.add(numbers[:5], 5, out=array[::2])
```

NumPy Arrays: Arithmetic

🧑‍💻 Sometimes we want the ***result of an arithmetic operation reduced to a single value***. That's when we have the ability to use the `reduce` function. We can chain it to any NumPy arithmetic function.

To find the sum of an array, we could do:

```python
array = np.arange(10)
sum = np.add.reduce(array)

# Similarly, we could find the product
prod = np.multiply.reduce(array)
```

NumPy Arrays: Arithmetic

🧮 Another one of the amazing features of **NumPy** is ***aggregation*** functions.

There's quite a few but the most common ones are the following:

```
all = np.all(array) # all true
any = np.any(array) # at least one true
sum = np.nansum(array) # NaN safe sum
std = np.nanstd(array) # NaN safe standard deviation
max = np.nanmax(array) # NaN safe max value in the array
min = np.nanmin(array) # NaN safe min value in the array
mean = np.nanmean(array) # NaN safe mean
median = np.nanmedian(array) # NaN safe median
```

NumPy

💯 And just like that, this **NumPy** introduction has come to an end. But there's *so, so much more* in the library and so many amazing things that it can do.

👩‍💻👨🏻‍💻 You must explore the rest on your own. This intro was to give you a head start. The official numpy documentation is one of the best ways to do that.

🔗 **Click here to visit the official NumPy Documentation**

🥰 **Visit my YouTube Channel**