

# Lecture VII - NumPy for Scientific Computing

## Programming with Python

Dr. Tobias Vlček

Kühne Logistics University Hamburg - Fall 2025

### Quick Recap of the last Lecture

#### Modules

- Modules are `.py` files containing Python code
- They are used to organize and reuse code
- They can define functions, classes, and variables
- Can be imported into other scripts

...

#### Tip

We can import entire modules or individual functions, classes or variables.

#### Standard Libraries

- Python includes many built-in modules like:
  - `random` provides functions for random numbers
  - `os` allows interaction with the operating system
  - `csv` is used for reading and writing CSV files
  - `re` is used for working with regular expressions

#### Packages

- Packages are collections of modules
- Often available from the Python Package Index (PyPI)
- Install using `uv add <package_name>`
- Virtual environments help manage dependencies

...

#### Tip

Virtual environments are not that important for you right now, as they are mostly used if you work on several projects with different dependencies at once.

# NumPy Module

## What is NumPy?

- NumPy is a package for scientific computing in Python
- Provides large, multi-dimensional arrays and matrices
- Wide range of functions to operate on these
- Python lists can be slow - Numpy arrays are much faster

...

### Note

The name of the package comes from Numerical Python.

## Why is NumPy so fast?

- Arrays are stored in a contiguous block of memory
- This allows for efficient memory access patterns
- Operations are implemented in the languages C and C++

...

Question: Have you heard of C and C++?

## How to get started

1. Install NumPy using `uv add numpy`
2. Import NumPy in a script using `import numpy as np`

...

```
import numpy as np
x = np.array([1, 2, 3, 4, 5]); type(x)
```

```
numpy.ndarray
```

...

### Note

You don't have to use `as np`. But it is a common practice to do so.

## Creating Arrays

- The backbone of Numpy is the so called `ndarray`
- Can be initialized from different data structures:

```
import numpy as np
```

```
array_from_list = np.array([1, 1, 1, 1])
print(array_from_list)
```

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

```
import numpy as np

array_from_tuple = np.array((2, 2, 2, 2))
print(array_from_tuple)
```

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

## Heterogenous Data Types

- It is possible to store different data types in a `ndarray`

```
import numpy as np

array_different_types = np.array(["s", 2, 2.0, "i"])
print(array_different_types)
```

```
['s' '2' '2.0' 'i']
```

...

### Note

But it is mostly not recommended, as it can lead to performance issues. If possible, try to keep the types homogenous.

## Prefilled Arrays

Improve performance by allocating memory upfront

- `np.zeros(shape)`: to create an array of zeros
- `np.random.rand(shape)`: array of random values
- `np.arange(start, stop, step)`: evenly spaced
- `np.linspace(start, stop, num)`: evenly spaced

...

### Note

The shape refers to the size of the array. It can have one or multiple dimensions.

## Dimensions

- The shape is specified as tuple in these arrays
- `(2)` or `2` creates a 1-dimensional array (vector)
- `(2,2)` creates a 2-dimensional array (matrix)
- `(2,2,2)` 3-dimensional array (3rd order tensor)
- `(2,2,2,2)` 4-dimensional array (4th order tensor)
- ...

## Computations

- We can apply operations to the entire array at once
- This is much faster than applying them element-wise

...

```
import numpy as np
x = np.array([1, 2, 3, 4, 5])
x + 1
```

```
array([2, 3, 4, 5, 6])
```

## Arrays in Action

Task: Practice working with Numpy:

```
# TODO: Create a 3-dimensional tensor with filled with zeros
# Choose the shape of the tensor, but it should have 200 elements
# Add the number 5 to all values of the tensor

# Your code here
assert sum(tensor) == 1000

# TODO: Print the shape of the tensor using the method shape()
# TODO: Print the dtype of the tensor using the method dtype()
# TODO: Print the size of the tensor using the method size()
```

## Indexing and Slicing

- Accessing and slicing `ndarray` works as before
- Higher dimension element access with multiple indices

...

Question: What do you expect will be printed?

```
import numpy as np
x = np.random.randint(0, 10, size=(3, 3))
print(x); print("---")
print(x[0:2,0:2])
```

```
[[4 0 6]
 [6 7 2]
 [5 0 3]]
---
[[4 0]
 [6 7]]
```

## Data Types

- Numpy provides data types as characters
- `i`: integer
- `b`: boolean
- `f`: float
- `S`: string
- `U`: unicode

...

```
string_array = np.array(["Hello", "World"]); string_array.dtype
```

```
dtype('<U5')
```

## Enforcing Data Types

- We can also provide the type when creating arrays

...

```
x = np.array([1, 2, 3, 4, 5], dtype = 'f'); print(x.dtype)
```

```
float32
```

...

- Or we can change them for existing arrays

```
x = np.array([1, 2, 3, 4, 5], dtype = 'f'); print(x.astype('i').dtype)
```

```
int32
```

...

### Note

Note, how the types are specified as `int32` and `float32`.

## Sidenote: Bits

Question: Do you have an idea what 32 stands for?

...

- It's the number of bits used to represent a number
  - `int16` is a 16-bit integer
  - `float32` is a 32-bit floating point number
  - `int64` is a 64-bit integer
  - `float128` is a 128-bit floating point number

## Why do Bits Matter?

- They matter, because they can affect:
  - the performance of your code
  - the precision of your results

...

- That's why numbers can have a limited precision!
  - An `int8` has to be in the range of -128 to 127
  - An `int16` has to be in the range of -32768 to 32767

...

Question: Size difference between `int16` and `int64`?

## Joining Arrays

- You can use `concatenate` to join arrays
- With `axis` you can specify the dimension
- In 2-dimensions `hstack()` and `vstack()` are easier

...

Question: What do you expect will be printed?

```
import numpy as np
ones = np.array((1,1,1,1))
twos = np.array((1,1,1,1)) * 2
print(np.vstack((ones,twos))); print(np.hstack((ones,twos)))
```

```
[[1 1 1 1]
 [2 2 2 2]]
[1 1 1 1 2 2 2 2]
```

## Common Methods

- `sort()`: sort the array from low to high
- `reshape()`: reshape the array into a new shape
- `flatten()`: flatten the array into a 1D array
- `squeeze()`: squeeze the array to remove 1D entries
- `transpose()`: transpose the array

...

#### 💡 Tip

Try experiment with these methods, they can make your work much easier.

## Speed Differences in Action

Task: Complete the following task to practice with Numpy:

```
# TODO: Create a 2-dimensional matrix with filled with ones of size 1000 x 1000.
# Afterward, flatten the matrix to a vector and loop over the vector.
# In each loop iteration, add a random number between 1 and 10000.
# TODO: Now, do the same with a list of the same size and fill it with random numbers.
# Then, sort the list as you have done with the Numpy vector before.
# You can use the 'time' module to compare the runtime of both approaches.
import time
start = time.time()
# Your code here
end = time.time()
print(end - start) # time in seconds
```

5.9604644775390625e-06

## That's it for today!

#### i Note

And that's it for today's lecture!  
You now have the basic knowledge to start working with scientific computing.

## Literature

### Interesting Books

- Downey, A. B. (2024). Think Python: How to think like a computer scientist (Third edition). O'Reilly. [Link to free online version](#)
- Elter, S. (2021). Schrödinger programmiert Python: Das etwas andere Fachbuch (1. Auflage). Rheinwerk Verlag.

...

For more interesting literature to learn more about Python, take a look at the [literature list](#) of this course.