# Python for HPC

**Dr. Simone Bacchio**

Computational Scientist

CaSToRC, The Cyprus Institute

National Competence Center in HPC

THE CYPRUS INSTITUTE

RESEARCH • TECHNOLOGY • INNOVATION

# Today's program

- ➢ 11:30-12:30 Performance in Python and Numpy
- ➢ 12:30-13:30 Lunch Break
- ➢ 13:30-14:30 Performance Optimization and Numba

**Requirements:**

- ➢ Some basic knowledge of Python
- ➢ Some basic knowledge of Numpy

*What is your knowledge of Python??*

**Goal:**

- ➢ Understand performance issues of Python and how to use it for HPC

# Programming languages & Performance

- ➢ Not all programming languages are designed with performance in mind

**Abstracted Programming Languages** *VS* **HPC Programming Languages**

- ➢ Python
- ➢ Matlab
- ➢ R, etc..

- ➢ Julia
- ➢ Java

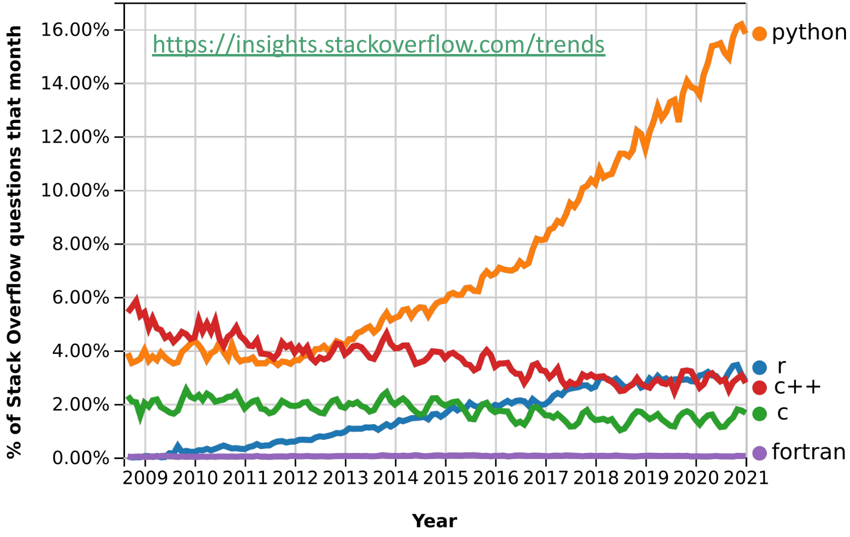
- ➢ Fortran
- ➢ C, C++

*Slow but easy-to-use* ⟷ *Fast but difficult*

**“Pure”** Python is <u>slow, very slow</u>, but it can be made very fast… Very important to learn how!

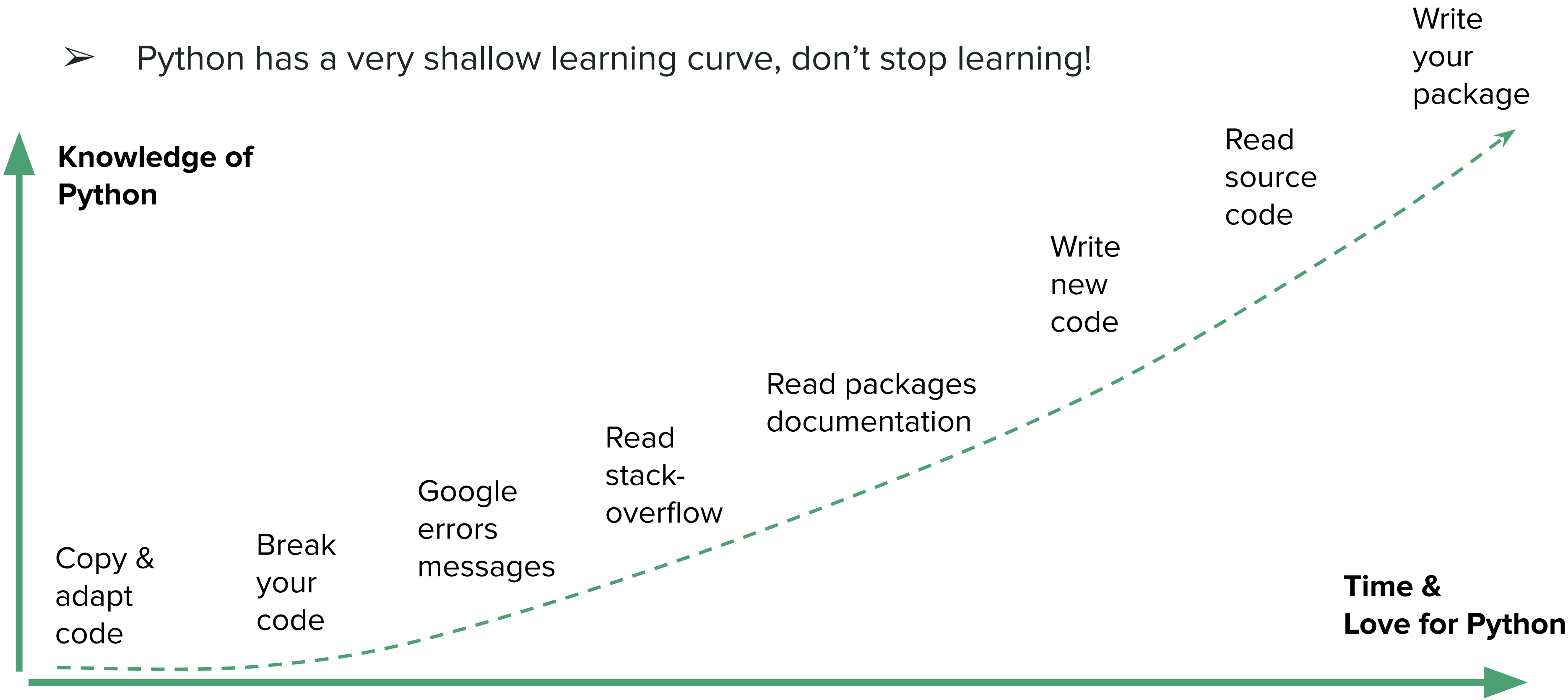
# Why Python?

- **Most used programming language in data science**
- **Interpreted** and object oriented programming language
- Science- and data-oriented
- Easy to Learn and Use
- Huge community
- Hundreds of Python Libraries and Frameworks
- First choice for Big Data and **Machine learning**
- User-friendly and great **APIs**
- Easy deployment of software (PyPI)
- Build with a scientific approach (PEPs)
- **Performance issues?** They can be overcome



# How to learn Python?

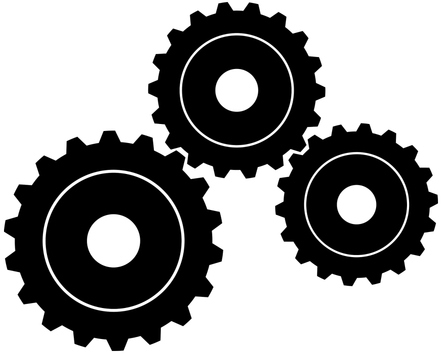
➢ Python has a very shallow learning curve, don't stop learning!



# How to use Python?

## Pure Python and APIs

- Build up the logic and abstraction
- Make it effective and user-friendly
- Limit its use in computationally intensive parts



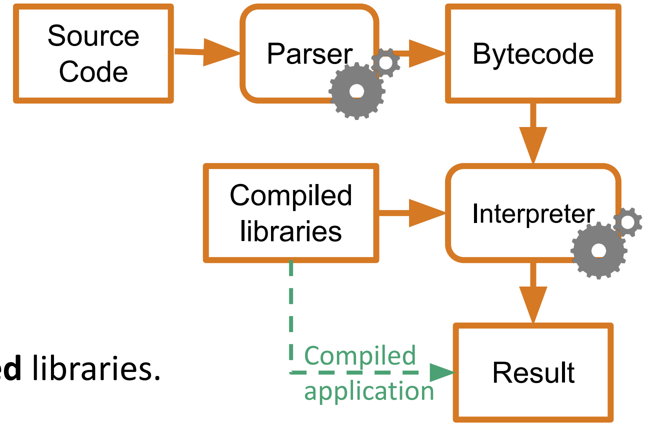
## Compiled code & backends

- Many packages come with compile code
- Make it efficient and very fast (C performance)
- Use as much as possible in computations

# Why is Python slow?

Python is a very powerful and flexible programming language, but…

- interpreted = bad (computational) performance
- it is important to know the strengths and the weaknesses!
- By its own it is not mean for High-Performance computing.




Built-in functions and HPC modules are based on **compiled** and **optimized** libraries.

Use as much as possible:

- built-in functions
- numerical modules (Numpy, Scipy, Pandas, …)
- compile your kernels (Cython, Pythran, Numba, …)

**NEVER** do for-loops on data!

# Numpy

➢ Numpy nowadays is the Python standard for numeric array calculations

➢ It is largely used and many packages are based on its **API**

- **Scipy:** uses Numpy for implementing numerical algorithms
- **Cupy:** a Numpy-compatible implementation for GPUs
- **Numba:** JIT compiler for Python code using Numpy
- **Pytorch:** its API is largely based on Numpy (not fully compatible tough)
- …

➢ A very good knowledge of Numpy is fundamental

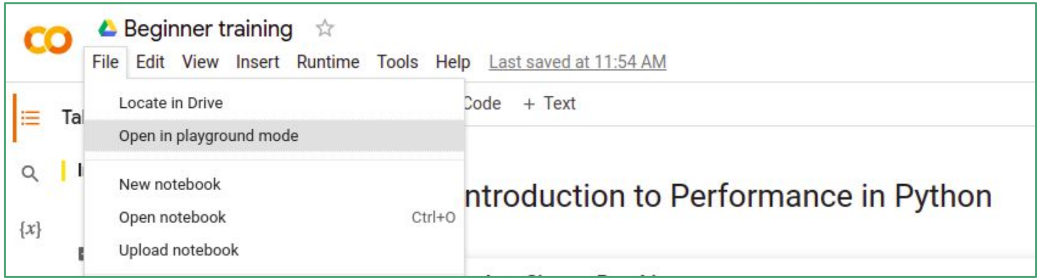
➢ Documentation: https://numpy.org/doc/stable/

➢ Remaining of the training on Numpy

# Let’s get started

➢ For the training we will use Jupyter Notebooks in Google Colaboratory
https://colab.research.google.com/drive/1B9_gVPwIXohe2MqOJ5Il_NI20sfQUIdR

➢ Open the link and start a new notebook or open in playground mode

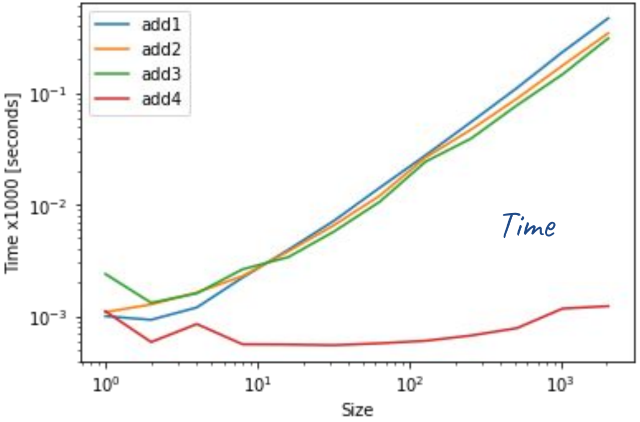


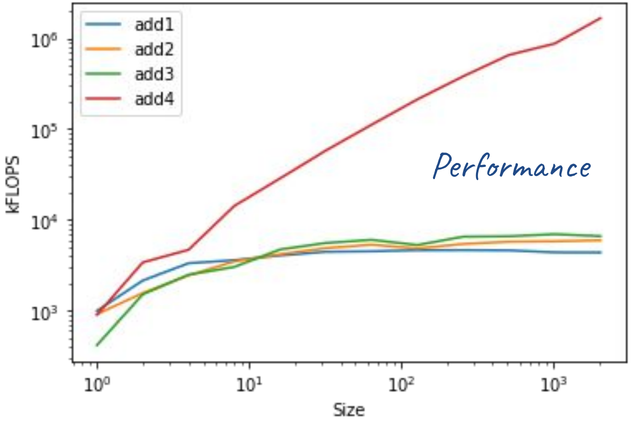

➢ Notebook and presentation also available on Github
https://github.com/CaSToRC-CyI/NCC-Beginner-Training-2022

# Performance

➢ For basic operations, Numpy achieves close-to-optimal performance and it is 1000x times faster than pure Python

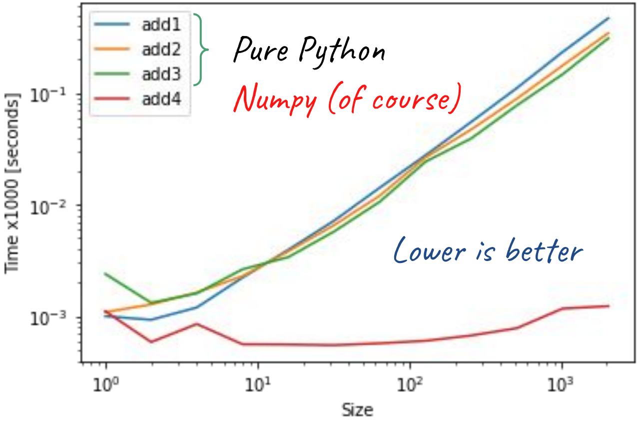
Which one is Numpy??

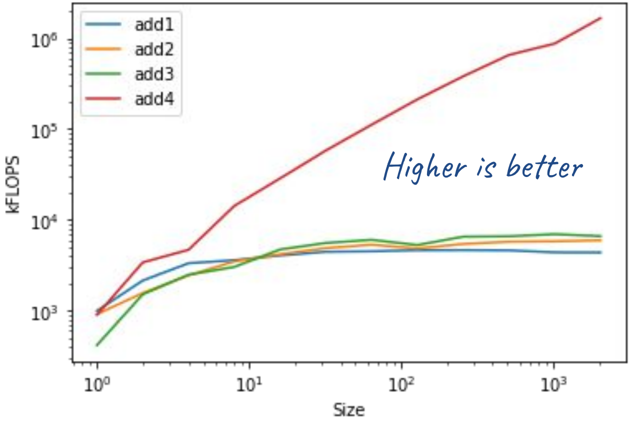


# Performance

➢ For basic operations, Numpy achieves close-to-optimal performance and it is 1000x times faster than pure Python



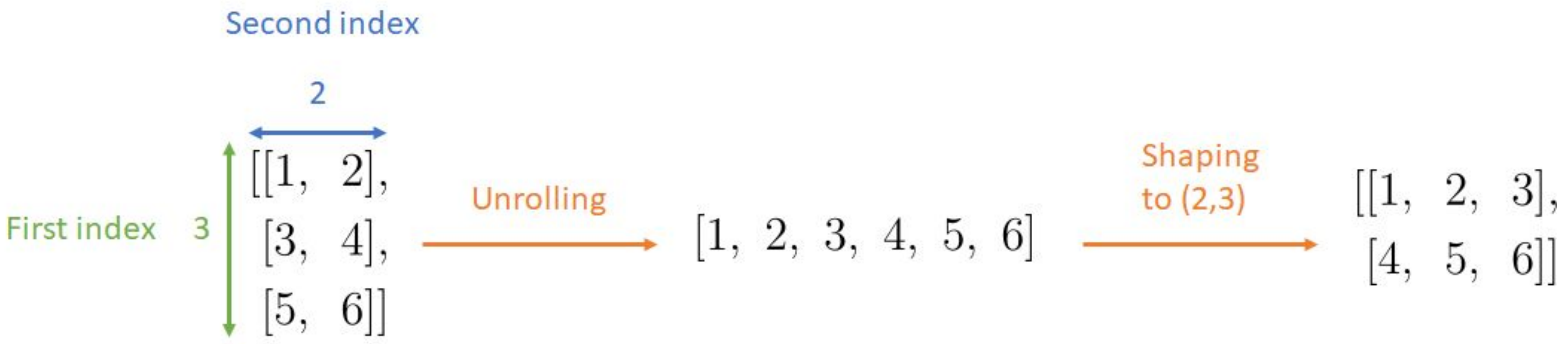

➢ **Remarks:**

- For small arrays Python overheads dominate
- Operations are done serially and between a step and another a new array is created

# Introduction to Numpy

- ➢ The core of Numpy is `ndarray` (n-dimensional array)
- ➢ An `ndarray` is defined by
  - **`shape:`** the size of the array along each dimension
  - **`dtype:`** the data type of the array and its size (`arr.dtype.itemsize`)
  - **`ordering`:** the data ordering in memory (C or F-contiguous)
- ➢ Any operation on the array is done via compiled code with high performance
- ➢ Implementation-wise `ndarray` is a **view** of a 1-dimensional array (unrolled data)
  - See **Python Buffer Protocol**, https://docs.python.org/3/c-api/buffer.html
  - See **Array Interface Protocol**, https://numpy.org/doc/stable/reference/arrays.interface.html
  - See e.g. `arr.__array_interface__`

# How does it work?




- ➢ N-dimensional arrays are *views* of **unrolled data**
- ➢ The shape is an artifact on the Python side but implementation-wise numpy always process unrolled data
- ➢ **NOTE:** for performance purposes, often many operation return different view of the same pointer. Therefore be careful when modifying arrays in-place!

# Item access, modification and slicing

- Arrays elements can be accessed and modified as for lists
  - Elements per dimensions can be either extracted serially or at once
    - E.g. `arr[0,1,2,3] = arr[0][1][2][3]`
    - The first, of course, is optimal because avoids creation of intermediate arrays
- Slices, ranges or lists can me used for accessing multiple elements at once
  - Slices are open ranges
    - E.g. `:10 == 0:10`
  - **Note:** tuples cannot be used!
- Dimensions can be skipped using ellipses (`...`)
- Broadcasting also applies for element assignment
- Assignment and assigning operations (+=) might change the original array!

# Universal functions

- See https://numpy.org/doc/stable/reference/ufuncs.html
- **Element-wise operations**
  - **Binary operations:** `+(add), -(sub), *(mul), /(div), %(mod), ==(eq), **(pow),` …
  - **Math functions:** `exp, log, sin, cos, tan,` …
  - Custom functions can be implemented via `np.vectorize`
- **Reductions**
  - Equal to: `for i in range(len(A)): r = op(r, A[i])`
  - Examples: `sum, mean, std, max, min`
  - They can be performed axis-wise (via argument `axis`)
  - Custom reductions can be implemented via `ufunc.reduce`
    - E.g. `sum = add.reduce`

# Performance limitations

```
y = x ** 2 + 2 * x + 1
```

*VS*

```
for(int i=0; i<N; i++) {
    y[i] = x[i] ** 2 + 2 * x[i] + 1
}
```

*What is the difference?*

# Performance limitations

$$y = \underbrace{\underbrace{\underbrace{x ** 2}_{a1} + \underbrace{2 * x}_{a2}}_{a3} + 1}_{y}$$

VS

```
for(int i=0; i<N; i++) {
    y[i] = x[i] ** 2 + 2 * x[i] + 1
}
```

**Left:** 4 loop over data, 5 array access, 3 extra arrays allocated
**Right:** 1 loop over data, 1 array access, no extra array allocated

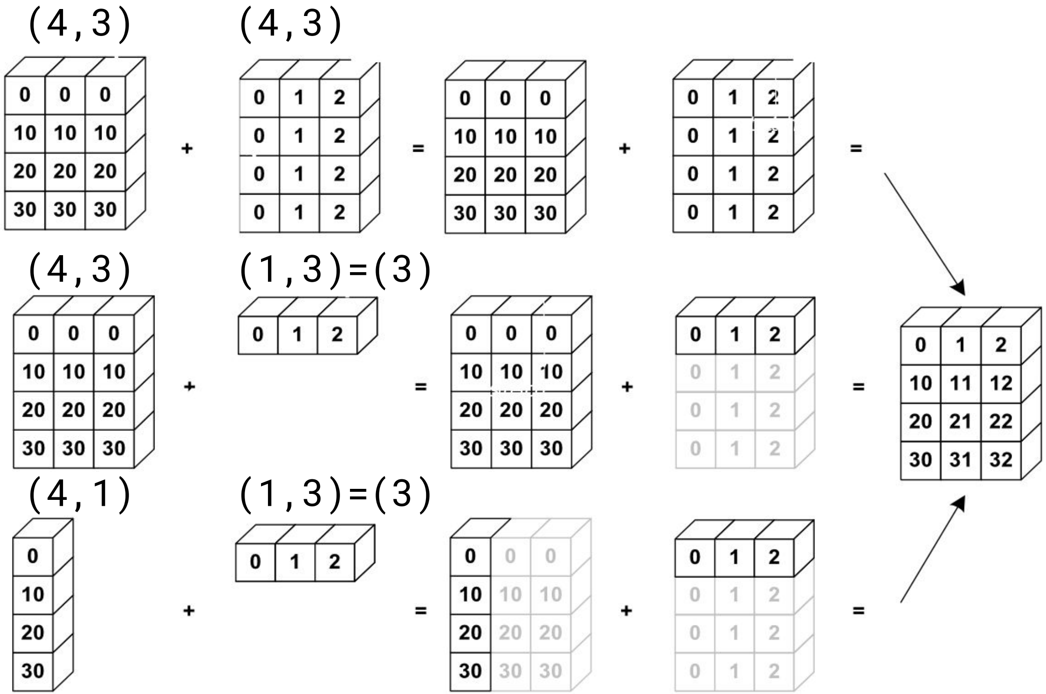
- ➢ Any operation on the arrays creates intermediate results and therefore new arrays
- ➢ This is quite a performance drawback because many allocations and loops are done
- ➢ Additionally a compiled loop can be optimized and use “special” operations
- ➢ This issue can be solved using `np.vectorize`

# Broadcasting

➢ Arrays of different dimensions can be operated together

**Requirements:**

- Sizes must be <u>either 1 or equal comparing from right to left</u>
- **If same size:**
  they are combined element-wise
- **If one-sized:**
  same value used for all axis
- **If missing dimensions:**
  automatically one-sized from left



# Additional operations

- With numpy you can almost do everything, without having to write a for-loop in Python
- For this you need a good knowledge of the API and can be achieved only practicing!
- E.g. how to do “`x[i+1] - x[i]`”? `y = np.roll(x,-1) - x`
  - See e.g. https://numpy.org/doc/stable/reference/routines.array-manipulation.html
- Many examples available online or on stack overflow… just search!

You didn’t find what you are looking for?

- **Try Numba!**

# Numba: a JIT compiler for Python

Numba is an open source JIT compiler that translates a subset of Python and NumPy code into fast machine code.

- ➢ Documentation: https://numba.pydata.org
- ➢ Installation: `pip install numba`
- ➢ CPU compiler: `from numba import jit`
- ➢ GPU API: `from numba import cuda`

# Easy compilation and parallelization

- Numba easily compiles, vectorize and parallelize Python code!

**Advantages?**

- The code gets compiled reaching C-performance
- The code can run in parallel using multi-threading

**Issue?**

- You need to explicitly write for-loops in Python!

So if you do not have any other way than writing explicitly a for-loop...

**Then do it and use Numba to speed it up!**

```python
from numba import njit, prange

@njit(parallel=True)
def difference(arr):
    N = arr.shape[0]
    out = np.empty_like(arr)
    for i in prange(N):
        out[i] = arr[(i+1)%N] - arr[i]
    return out
```

# Conclusions

- ➢ Never do for-loop on data in Python
- ➢ Numpy comes first at rescue with its very user-friendly API
  - **NOTE:** Other packages are available, e.g. Pandas dataframe (on Wedsneday) but a very good knowledge of numpy is fundamental
- ➢ Use Numba to speed-up Python code
  - We just had time to scratch the surface. Give it a try it is very useful!
  - More will be covered in the intermediate training including GPU programming

*Questions??*