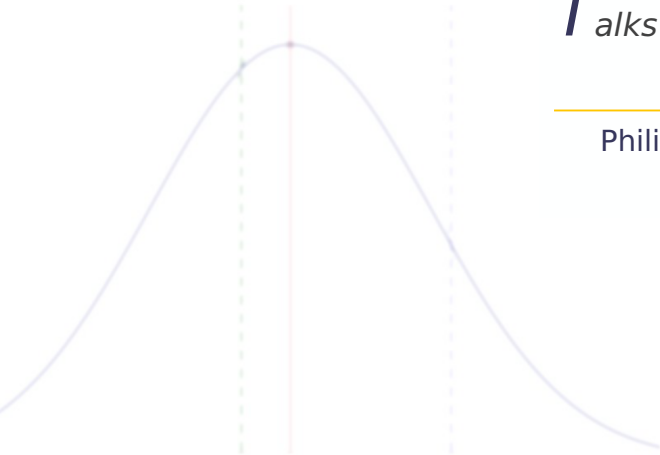


# $T_{alks}$

---

Philipp Eisenhauer



Material available on



Visit us!

# High-performance computing using Python

Philipp Eisenhauer

I draw on the material presented in:

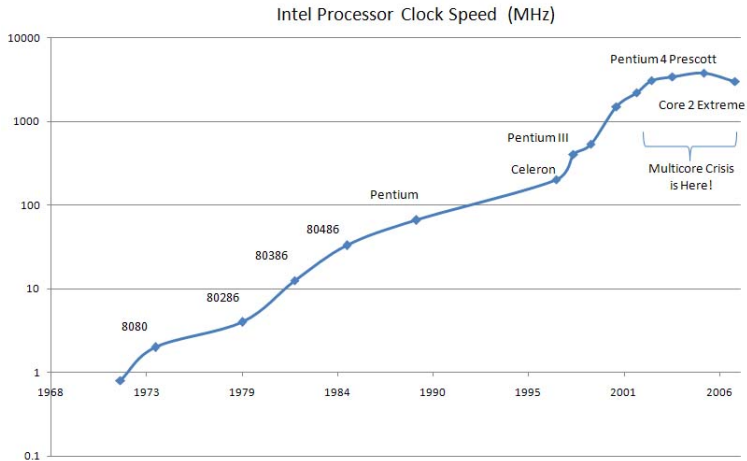
- ▶ Gorelick, M., & Ozsvald, I. (2014). *High performance python*.
- ▶ Lanaro, G. (2017). *Python high performance*.

## Basic architecture

- ▶ computing units, e.g. CPU and GPU
- ▶ memory units, e.g. RAM and hard disk
- ▶ connections

## **Main properties of computing unit**

- ▶ number of operations in one cycle, e.g. vectorization
- ▶ how many cycles in one second



## Global interpreter lock

*The GIL makes sure that a Python process can only run one instruction at a time, regardless of the number of cores it is currently using. This means that even though some Python code has access to multiple cores at a time, only one core is running a Python instruction at any given time.*

## Profiling

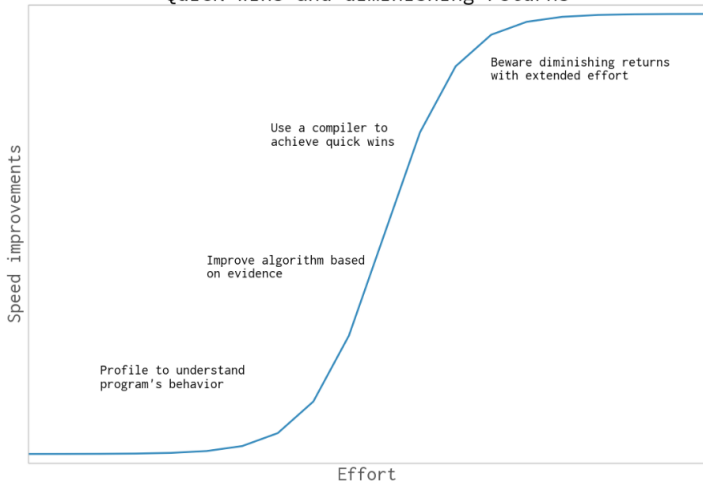
- ▶ *Premature optimization is the root of all evil.*
- ▶ focus on readability
- ▶ set up development environment
- ▶ flesh out testing harness
- ▶ tackle performance bottlenecks



## Points of attack

- ▶ pure Python
- ▶ high-performance libraries, SciPy Stack
- ▶ compilation to faster language
- ▶ parallel computing
- ▶ distributed computing

## Quick wins and diminishing returns

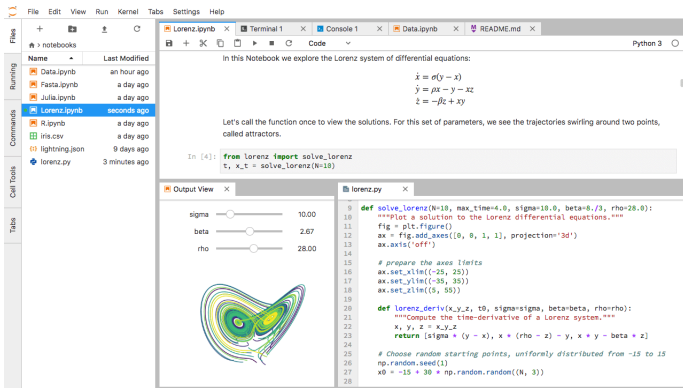


# **High-performance libraries**

## Vectorization

- ▶ Vectorization is when a CPU is provided with multiple pieces of data at a time and is able to operate on all of them at once. This sort of CPU instruction is known as SIMD (Single Instruction, Multiple Data).

# Practical illustration



# **Compilation to faster language**

## Alternatives

- ▶ ahead-of-time, e.g. f2py, Cython
- ▶ just-in-time, e.g. numba

# Practical illustration

File Edit View Run Kernel Tabs Settings Help

Files

- notebooks
- Data.ipynb an hour ago
- Fasta.ipynb a day ago
- Julia.ipynb a day ago
- Lorenz.ipynb seconds ago**
- R.ipynb a day ago
- Iris.csv a day ago
- lightning.json 9 days ago
- lorenz.py 3 minutes ago

Running

Commands

Cell Tools

Tab

Lorenz.ipynb x Terminal 1 x Console 1 x Data.ipynb x README.md x Python 3

Code

In this Notebook we explore the Lorenz system of differential equations:

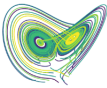
$$\begin{aligned}\dot{x} &= \sigma(y - x) \\ \dot{y} &= \rho x - y - xz \\ \dot{z} &= -\beta z + xy\end{aligned}$$

Let's call the function once to view the solutions. For this set of parameters, we see the trajectories swirling around two points, called attractors.

In [4]: `from lorenz import solve_lorenz  
t, x_t = solve_lorenz(N=18)`

Output View x lorenz.py x

sigma 10.00  
beta 2.67  
rho 28.00



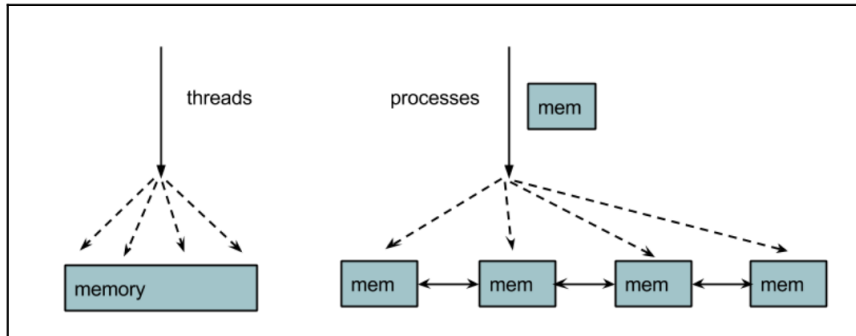
```
9 def solve_lorenz(N=18, max_time=4.0, sigma=10.0, beta=8./3, rho=28.0):  
10     """Plot a solution to the Lorenz differential equations."""  
11     fig = plt.figure()  
12     ax = fig.add_axes([0, 0, 1, 1], projection='3d')  
13     ax.axis('off')  
14  
15     # prepare the axes limits  
16     ax.set_xlim([-25, 25])  
17     ax.set_ylim([-35, 35])  
18     ax.set_zlim([0, 55])  
19  
20  
21     def lorenz_deriv(x,y,z, t0, sigma=sigma, beta=beta, rho=rho):  
22         """Compute the time-derivative of a Lorenz system."""  
23         x_dot, y_dot, z_dot = x.y.z  
24         return [sigma * (y - x), x * (rho - z) - y, x * y - beta * z]  
25  
26     # Choose random starting points, uniformly distributed from -15 to 15  
27     np.random.seed(1)  
28     x0 = -15 + 30 * np.random.random((N, 3))
```



# Parallel computing

## **Memory access**

- ▶ shared memory
- ▶ distributed memory



# Practical illustration

File Edit View Run Kernel Tabs Settings Help

Files

- notebooks
- Data.ipynb an hour ago
- Fasta.ipynb a day ago
- Julia.ipynb a day ago
- Lorenz.ipynb seconds ago**
- R.ipynb a day ago
- Iris.csv a day ago
- lightning.json 9 days ago
- lorenz.py 3 minutes ago

Running

Commands

Cell Tools

Tab

Lorenz.ipynb x Terminal 1 x Console 1 x Data.ipynb x README.md x Python 3

Code

In this Notebook we explore the Lorenz system of differential equations:

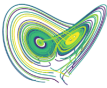
$$\begin{aligned}\dot{x} &= \sigma(y - x) \\ \dot{y} &= \rho x - y - xz \\ \dot{z} &= -\beta z + xy\end{aligned}$$

Let's call the function once to view the solutions. For this set of parameters, we see the trajectories swirling around two points, called attractors.

In [4]: `from lorenz import solve_lorenz  
t, x_t = solve_lorenz(N=18)`

Output View x lorenz.py x

sigma 10.00  
beta 2.67  
rho 28.00



```
9 def solve_lorenz(N=18, max_time=4.0, sigma=10.0, beta=8./3, rho=28.0):  
10     """Plot a solution to the Lorenz differential equations."""  
11     fig = plt.figure()  
12     ax = fig.add_axes([0, 0, 1, 1], projection='3d')  
13     ax.axis('off')  
14  
15     # prepare the axes limits  
16     ax.set_xlim(-25, 25)  
17     ax.set_ylim(-35, 35)  
18     ax.set_zlim(0, 55)  
19  
20     def lorenz_deriv(x,y,z, t0, sigma=sigma, beta=beta, rho=rho):  
21         """Compute the time-derivative of a Lorenz system."""  
22         x_dot, y_dot, z_dot = x.y.z  
23         return [sigma * (y - x), x * (rho - z) - y, x * y - beta * z]  
24  
25     # Choose random starting points, uniformly distributed from -15 to 15  
26     np.random.seed(1)  
27     x0 = -15 + 30 * np.random.random((N, 3))  
28
```

# **Distributed computing**

## Challenges to parallelization

- ▶ algorithm limitations
- ▶ bottlenecks
- ▶ startup overhead
- ▶ communication

## **Communications**

- ▶ point-to-point communications
- ▶ collective communications
- ▶ dynamic process management

# Resources



## Textbooks

- ▶ Lanaro, G. (2017). *Python high performance*.
- ▶ Gorelick, M., & Ozsvald, I. (2014). *High performance python*.

# Appendix

# *References*

Gorelick, M., & Ozsvald, I. (2014). *High performance python*.

Lanaro, G. (2017). *Python high performance*.