# Benchmarking Efficiency of Programming Languages

**Rashi Shetty**

**MSc BDA Part I**

The objective was to compare the efficiency of various programming languages using code execution times for a singular task. Jupyter Notebook was chosen as the optimal interface for code execution and profiling. The results obtained through benchmarking and profiling are instrumental in enabling informed decisions when selecting a language to optimize performance for similar computational tasks.

**Languages used:** Python, Cython and Julia

**Methodology:**

1. Python
2. Python using loops
3. Python using NumPy
4. Cython
5. Julia

**Requirements:**

1. Jupyter Notebook interface.
2. Python, Cython, and Julia kernels
3. Microsoft Visual C++ Build Tools

**Problem:**

Print numbers from 0 to 10,000

While a more comprehensive and precise comparison could be achieved by evaluating larger and more complex codes, limitations arose with constraints such as limited expertise in the languages and a slower system.

**Code:** Jupyter Notebook

**Findings:**

Execution speed [Python]: `0.002049 seconds`
Execution speed [Python using for loop]: `0.022551 seconds`
Execution speed [Python using NumPy]: `0.000297 seconds`
Execution speed [Cython]: `0.025426 seconds`
Execution speed [Julia]: `0.682496 seconds`

The findings indicate the following efficiency sequence:

Python using NumPy > Python > Python using loop > Cython > Julia

**Limitations:**

The benchmarks are limited to specific tasks and does not represent overall language performance. The hardware and system load during execution influences the results.

**Conclusion:**

1. In general, loops and NumPy in Python can be faster and more efficient compared to plain Python for certain computational tasks, especially when dealing with large datasets and numerical operations. NumPy, in particular, is well-known for its high-performance array operations, which can significantly speed up numerical computations compared to using plain Python lists and loops.
2. Cython enhances Python's efficiency by allowing compiled C-like extensions. It bridges the performance gap, as it enables direct C integration and static typing, leading to faster code execution.
3. Julia exhibits high efficiency, particularly in numerical tasks, owing to its just-in-time (JIT) compilation and multiple dispatch mechanisms. Its dynamic typing enables faster execution of complex operations.
4. The initial slower performance of Julia and Cython might be attributed to the first-time installation overhead and code complexity. With optimization and proficiency, both languages can outperform Python in computation-intensive scenarios, making them valuable choices for scientific computing and numerical simulations.