Python for High Performance Data Analytics

— (1) Computation —

Qiyang Hu
UCLA Office of Advanced Research Computing

April 28, 2023

About this series

- The lectures will focus on
  - *High-level* **overviews**.
  - Introducing **libraries** that require *minimal* efforts to boost performance.
  - Short Jupyter Notebook **demos**
- What can/can't expected in the series?

✓ CAN	✗ CAN'T
• From an end users' perspective	• From a package developers' perspective
• A ***BIGGER***-picture review on the selected 3rd-party python libraries	• Native Python tricks (e.g. container, lazy eval, mem) • Line-by-line explanations on these library interfaces
• Demos on specific example problems	• Discussion on the performance of various algorithms

Outline for today

- What’s the performance concerns for big data analytics?
- Why learning Python in the ChatGPT era?
- Why Python is slow?
- How to speed Python up using JIT?
- How to speed Python up using AOT?
- Colab demos

Outline for today

- What's the performance concerns for big data analytics?
- Why learning Python in the ChatGPT era?
- Why Python is slow?
- How to speed Python up using JIT?
- How to speed Python up using AOT?
- Colab demos

COMPUTATION

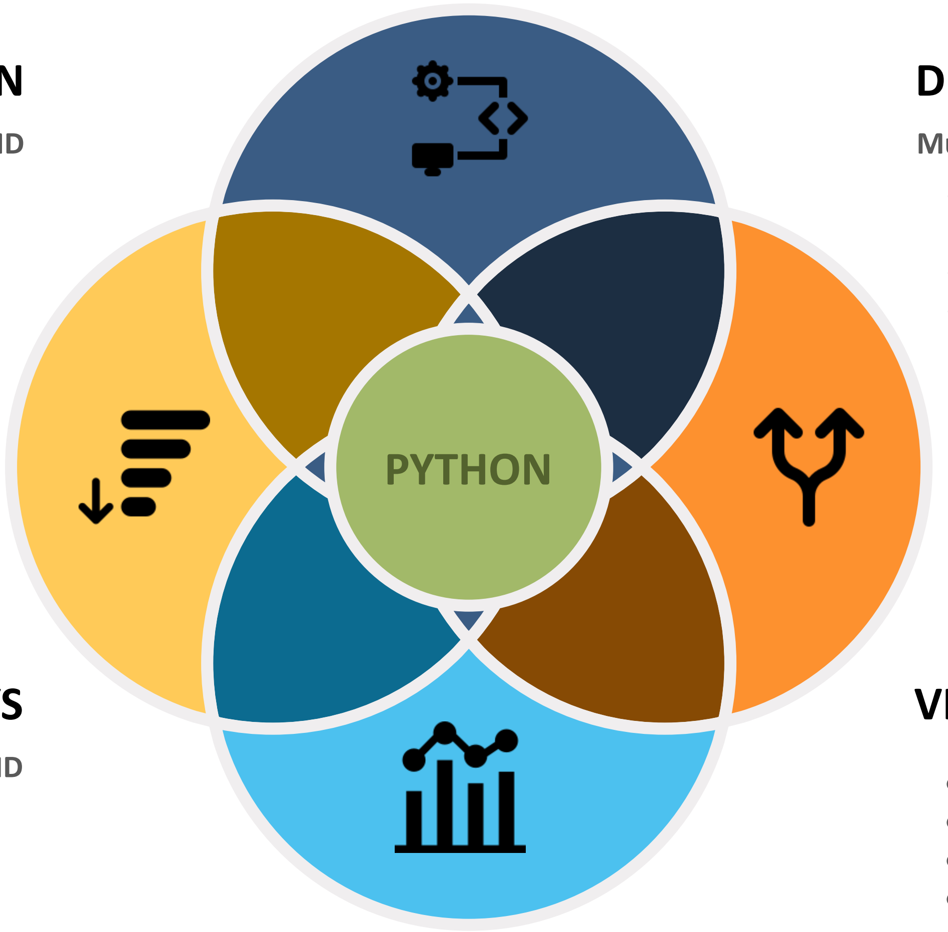
Single Node/GPU, SIMD

- Pypy, Numba, NumExpr
- Pythran, Cython
- F2py, ctypes

DATA ARRAYS

Single Node/GPU, SIMD

- Numpy
- Pandas, Polars
- Modin, Pandarallel, Swifter
- Dask DataFrame, Vaex



DISTRIBUTED

Multiple Nodes/Machines

- MapReduce-based: PySpark, PyFlink
- MPI-based: mpi4py, Horovod
- Joblib, Dask, Ray

VISUALIZATION

- Viz process for big data
- Matplotlib, Bokeh, Plotly
- Holoview and Datashader
- Traited VTK, Mayavi, Paraview

Two Big **Do-Not**'s

Don't optimize prematurely.

"The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times…"

-- Donald Knuth in "TAOCP"

- Easiest to understand and explain
- Quickest to write
- Easiest to test and maintain
- Most portable to migrate

Don't trust benchmarks.

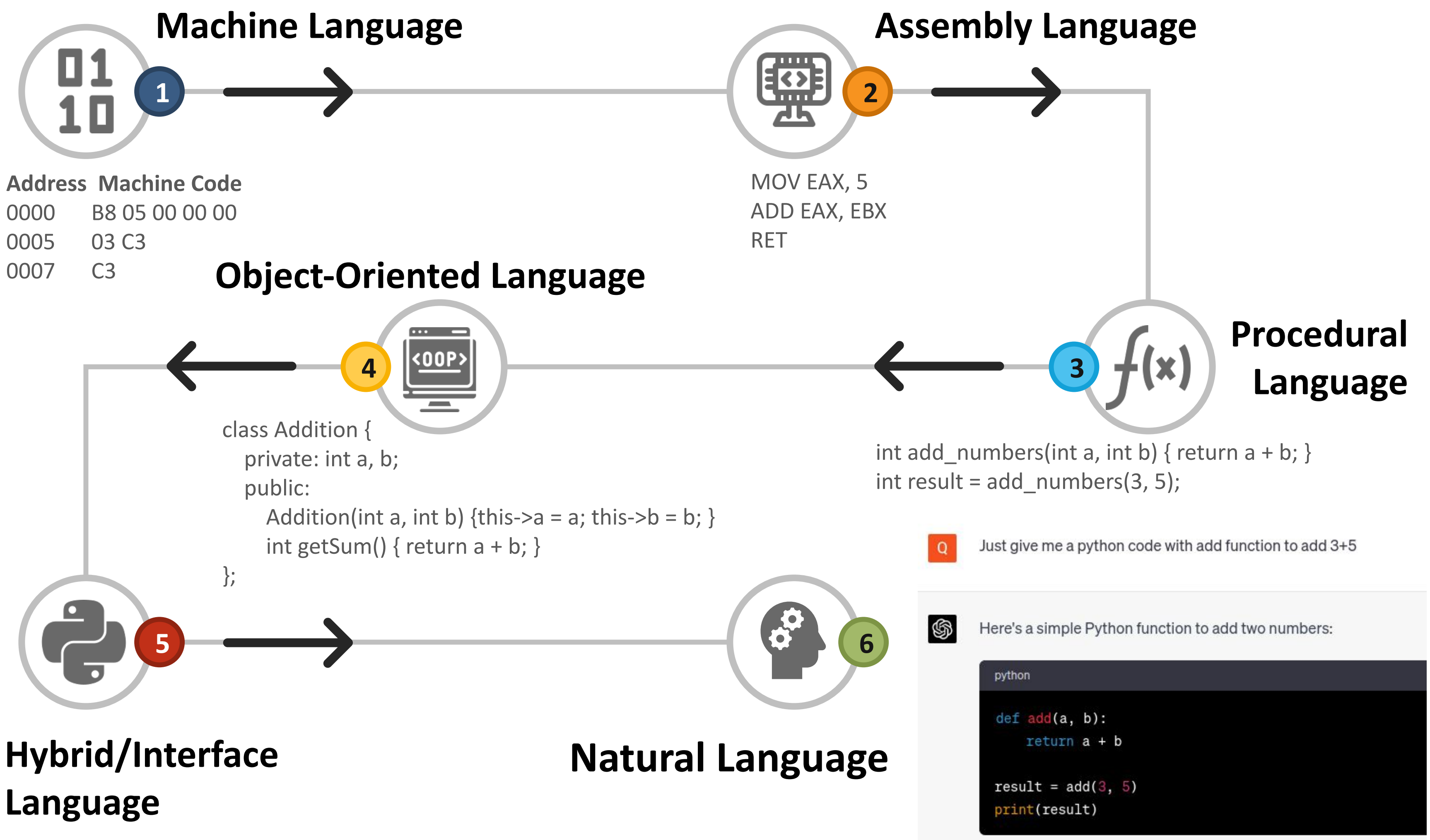
All benchmark numbers are "wrong".

- Specific hardware/OS/libraries
- In-situ running environments
- Different nature of datasets
- Sometimes very version-sensitive

- Understand the mechanisms
- Focus on the qualitative comparisons
- Need to do your own experiments.

Outline for today

- What’s the performance concerns for big data analytics?
- Why learning Python in the ChatGPT era?
- Why Python is slow?
- How to speed Python up using JIT?
- How to speed Python up using AOT?
- Colab demos



Machine Language
1
Address Machine Code
0000 B8 05 00 00 00
0005 03 C3
0007 C3
Assembly Language
2
MOV EAX, 5
ADD EAX, EBX
RET
Procedural Language
3
int add_numbers(int a, int b) { return a + b; }
int result = add_numbers(3, 5);
Object-Oriented Language
4
<OOP>
class Addition {
private: int a, b;
public:
Addition(int a, int b) {this->a = a; this->b = b; }
int getSum() { return a + b; }
};
Hybrid/Interface Language
5
Natural Language
6
Q
Just give me a python code with add function to add 3+5
Here's a simple Python function to add two numbers:
python
def add(a, b):
return a + b
result = add(3, 5)
print(result)

Why learning Python in the ChatGPT era?

Python can help us

- Understanding CS/HPC Concepts
- Providing chains of thoughts
- Nailing down the key problem quickly

Beyond the Zero/Few Shots

Better & Efficient Prompts

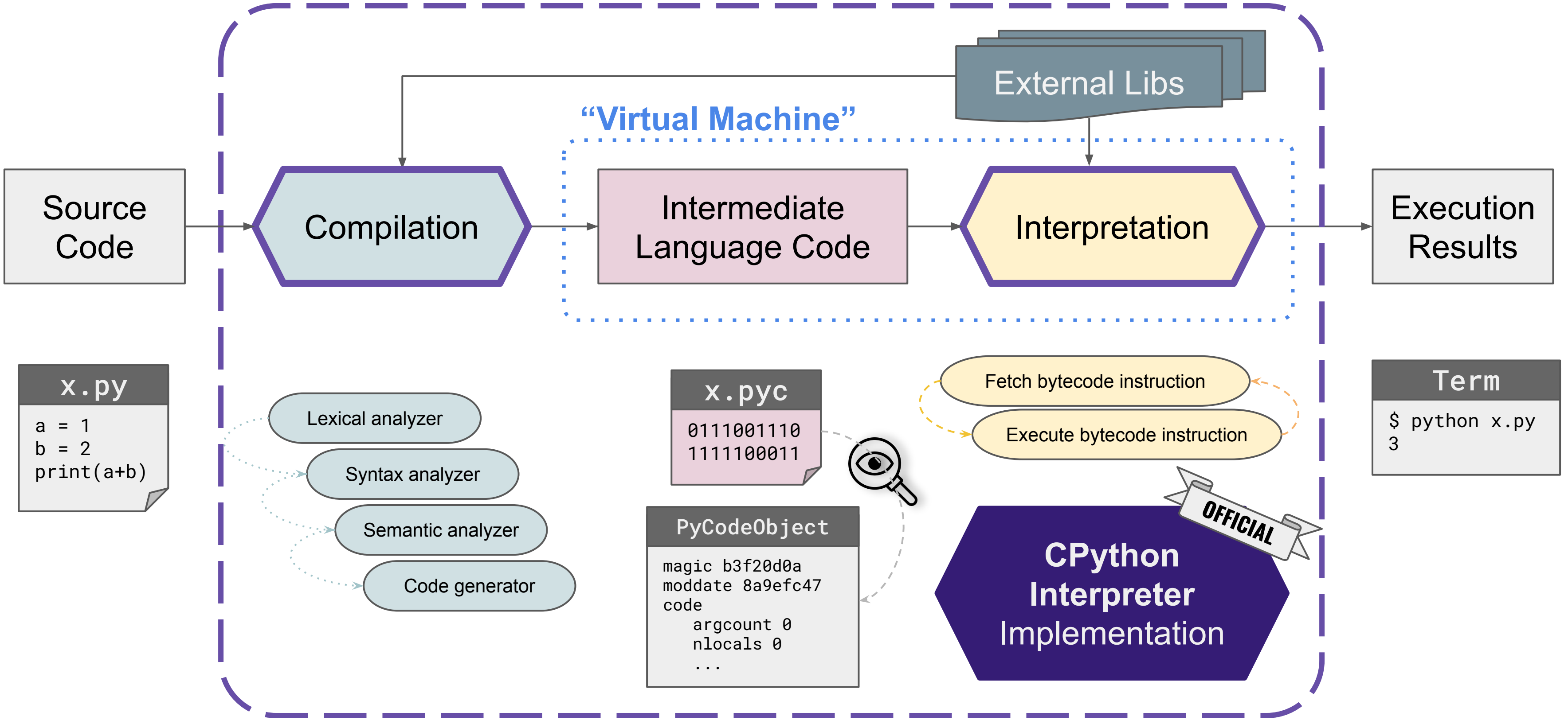
Current ChatGPT cannot do everything yet.

- Needs to fix the hallucination problem.
- Needs to distill the domain knowledge
- Needs to finetune the pre-trained model

Outline for today

- What's the performance concerns for big data analytics?
- Why learning Python in the ChatGPT era?
- Why Python is slow?
- How to speed Python up using JIT?
- How to speed Python up using AOT?
- Colab demos

Seriously, what is Python?



Why Python is slow?

Python is Dynamically Typed rather than Statically Typed.

```
/* C code */
int a = 1;
int b = 2;
int c = a + b;
```

```
# python code
a = 1
b = 2
c = a + b
```

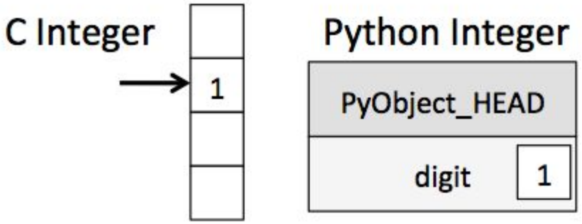
[Source]

C Addition

1. Assign `<int> 1` to `a`
2. Assign `<int> 2` to `b`
3. call `binary_add<int, int>(a, b)`
4. Assign the result to c

Python Addition

1. Assign `1` to `a`
    - **1a.** Set `a->PyObject_HEAD->typecode` to integer
    - **1b.** Set `a->val = 1`
2. Assign `2` to `b`
    - **2a.** Set `b->PyObject_HEAD->typecode` to integer
    - **2b.** Set `b->val = 2`

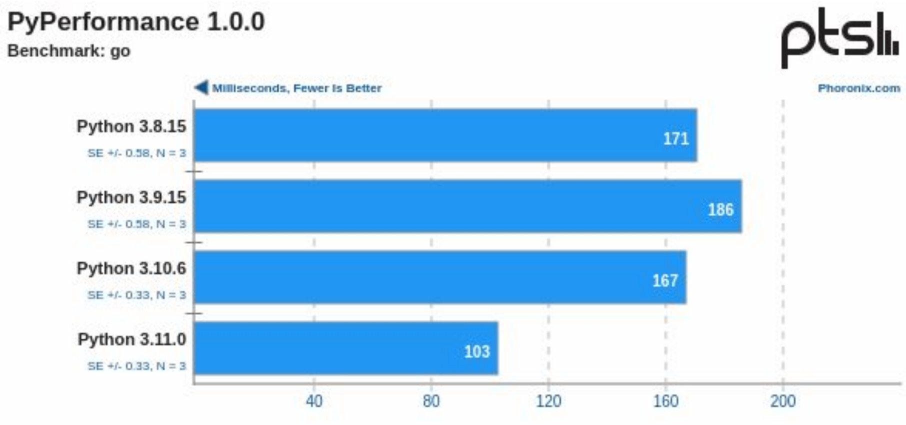


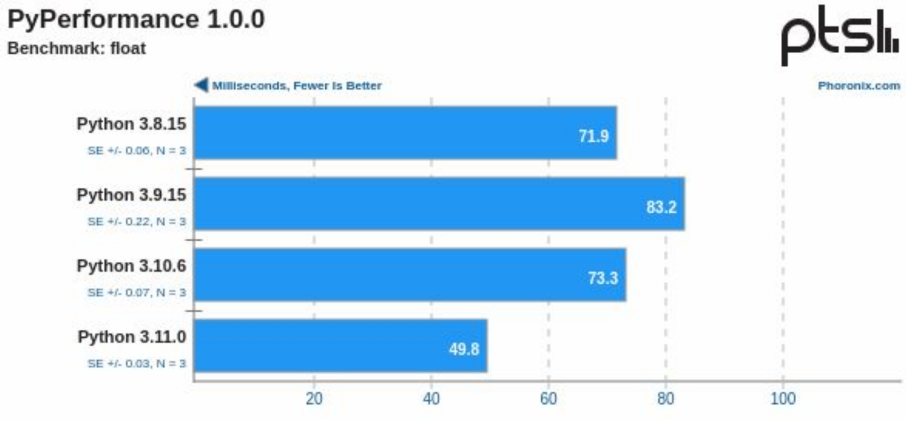
3. call `binary_add(a, b)`
    - **3a.** find typecode in `a->PyObject_HEAD`
    - **3b.** `a` is an integer; value is `a->val`
    - **3c.** find typecode in `b->PyObject_HEAD`
    - **3d.** `b` is an integer; value is `b->val`
    - **3e.** call `binary_add<int, int>(a->val, b->val)`
    - **3f.** result of this is `result`, and is an integer.
4. Create a Python object `c`
    - **4a.** set `c->PyObject_HEAD->typecode` to integer
    - **4b.** set `c->val` to `result`

The “Shannon Plan”

- A plan to make CPython faster
  - Originally proposed by Eric Snow, and Mark Shannon in 2020
  - Guido van Rossum joined and gave a talk in Python Language Summit (May 2021)
  - Based on the experience with “HotPy” and “HoyPy 2”
  - Promising 5x in 4 years, 1.5x per year
- Compatibility guarantees
  - Don’t break stable ABI compatibility
  - Don’t break limited API compatibility
  - Don’t break or slow down extreme cases
- Python 3.11 is 25% faster than 3.10
  - An adaptive, specializing bytecode interpreter and lots of optimization tweaks
  - Faster startup times and more efficient use of/communication with C
  - Will mostly benefit: CPU-intensive pure Python code

What will 3.12 look like?





Operation	Form	Specialization	Operation speedup (up to)	Contributor(s)
Binary operations	`x+x; x*x; x-x;`	Binary add, multiply and subtract for common types such as `int`, `float`, and `str` take custom fast paths for their underlying types.	10%	Mark Shannon, Dong-hee Na, Brandt Bucher, Dennis Sweeney
Subscript	`a[i]`	Subscripting container types such as `list`, `tuple` and `dict` directly index the underlying data structures. Subscripting custom `__getitem__` is also inlined similar to Inlined Python function calls.	10-25%	Irit Katriel, Mark Shannon
Store subscript	`a[i] = z`	Similar to subscripting specialization above.	10-25%	Dennis Sweeney
Calls	`f(arg)` `C(arg)`	Calls to common builtin (C) functions and types such as `len` and `str` directly call their underlying C version. This avoids going through the internal calling convention.	20%	Mark Shannon, Ken Jin
Load global variable	`print` `len`	The object's index in the globals/builtins namespace is cached. Loading globals and builtins require zero namespace lookups.	[1]	Mark Shannon
Load attribute	`o.attr`	Similar to loading global variables. The attribute's index inside the class/object's namespace is cached. In most cases, attribute loading will require zero namespace lookups.	[2]	Mark Shannon
Load methods for call	`o.meth()`	The actual address of the method is cached. Method loading now has no namespace lookups – even for classes with long inheritance chains.	10-20%	Ken Jin, Mark Shannon
Store attribute	`o.attr = z`	Similar to load attribute optimization.	2% in pyperformance	Mark Shannon
Unpack Sequence	`*seq`	Specialized for common containers such as `list` and `tuple`. Avoids internal calling convention.	8%	Brandt Bucher

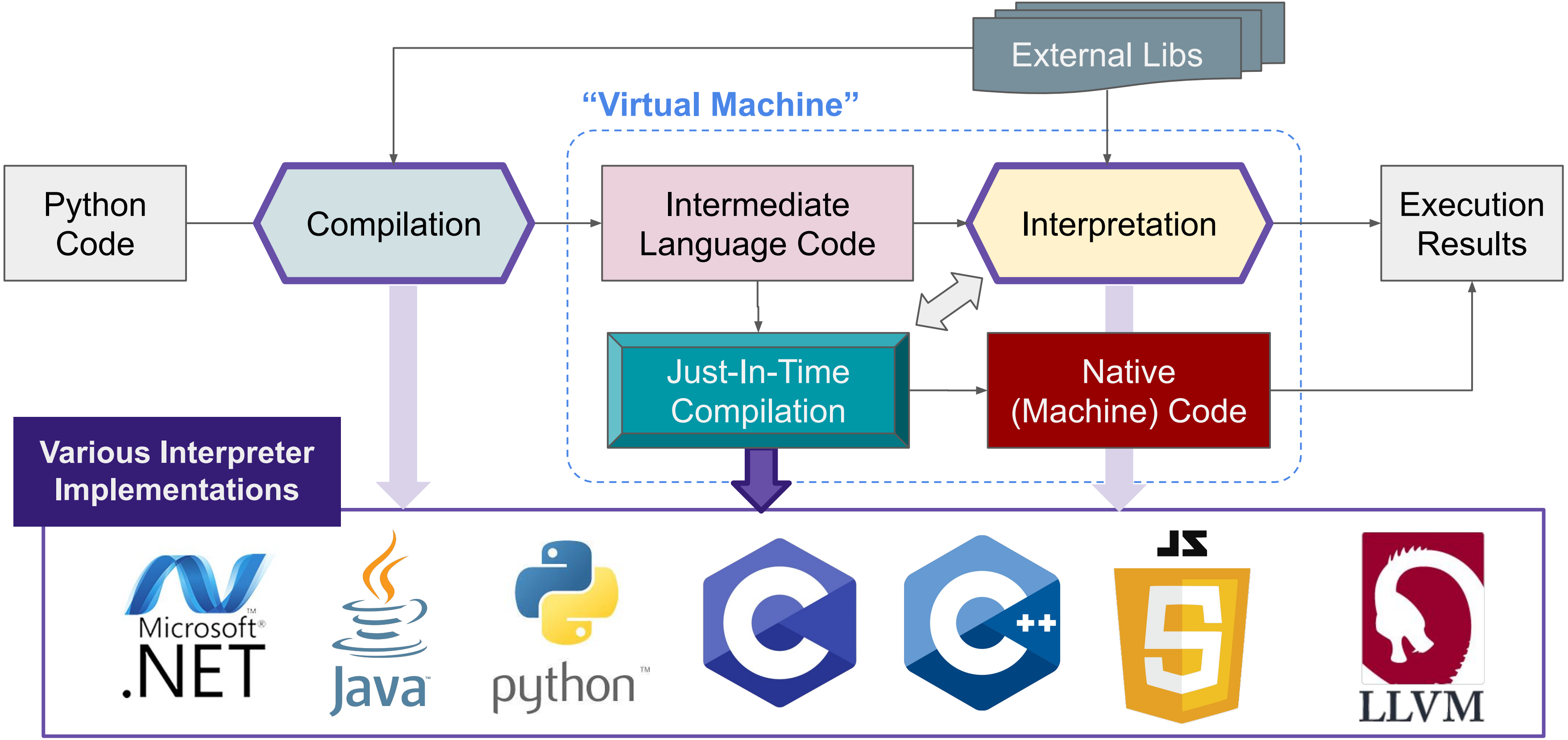
GIL: Guilty or Gilly?

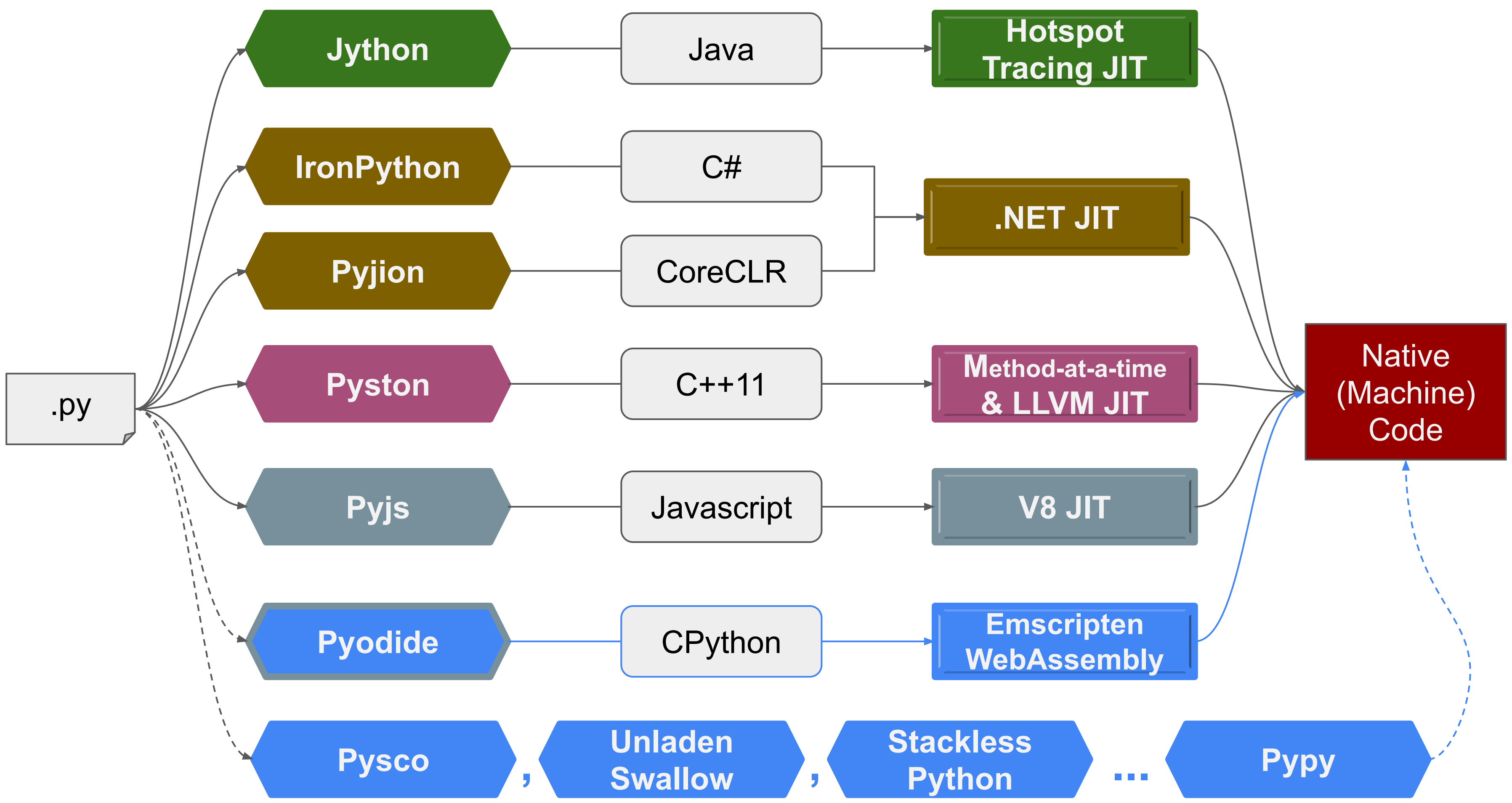
- GIL (Global Interpreter Lock)
  - A mutex (or a lock) that allows only one *thread* to hold the control of the Python interpreter.
- Why Python uses it?
  - GILs is added to the ref count variables to be kept protected from race conditions
  - GIL has performance benefits of GIL in single-threaded situation.
  - Historically Python has been around when OS did not have a concept of threads.
- Correct way to use it:
  - Multi-processing vs multi-threading:
    - Multi-threading: good for IO-intensive code, bad for CPU-intensive code
    - use multiple processes with “multiprocessing” module instead of threads
    - Consider to use Intel Distribution of Python
  - Attempts from Python community to remove the GIL from CPython:
    - Gilectomy (abandoned)
    - A new compiler flag: nogil (expected in Python 3.12)
  - Alternative Python interpreters, as GIL only with CPython
    - multiple interpreter implementations

Outline for today

- What’s the performance concerns for big data analytics?
- Why learning Python in the ChatGPT era?
- Why Python is slow?
- How to speed Python up using JIT?
- How to speed Python up using AOT?
- Colab demos

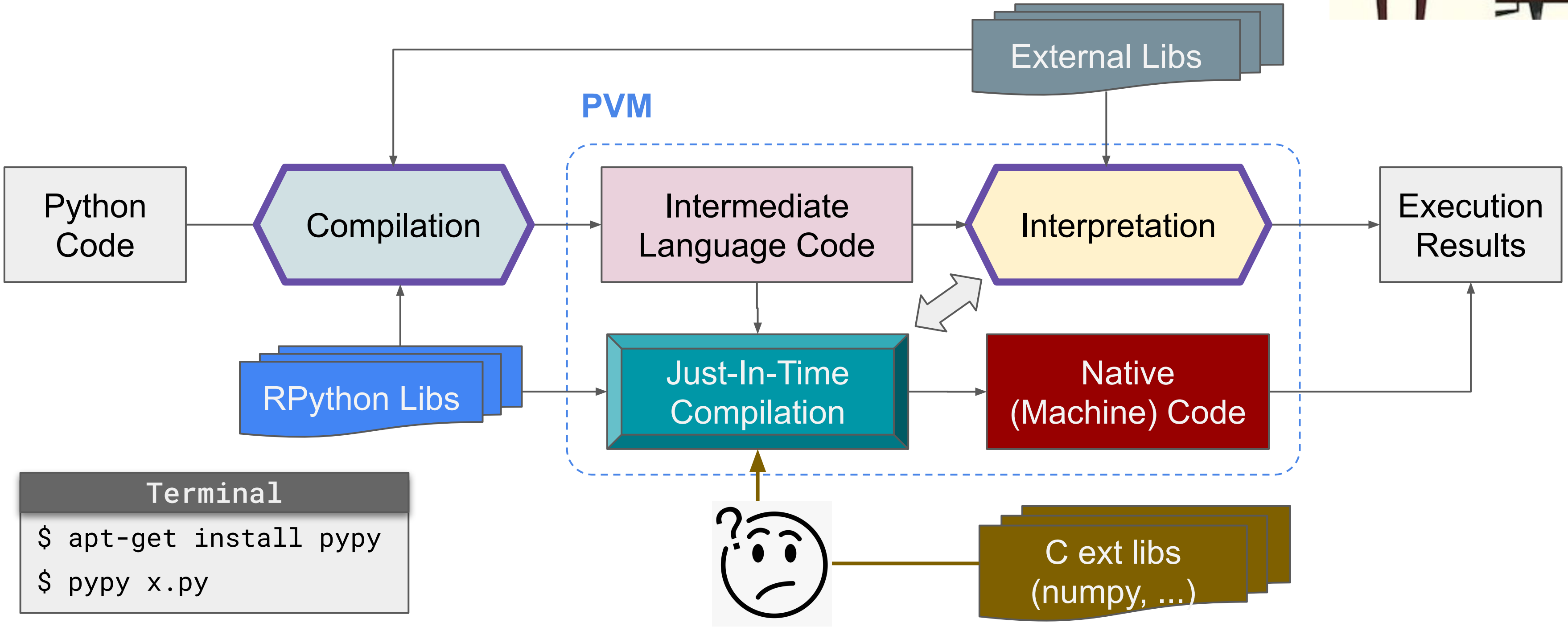
Boosting the speed by JIT




.py
Jython
IronPython
Pyjion
Pyston
Pyjs
Pyodide
Pysco
,
Unladen Swallow
,
Stackless Python
...
Pypy
Java
C#
CoreCLR
C++11
Javascript
CPython
Hotspot Tracing JIT
.NET JIT
Method-at-a-time & LLVM JIT
V8 JIT
Emscripten WebAssembly
Native (Machine) Code

Pypy: using Python to interpret Python

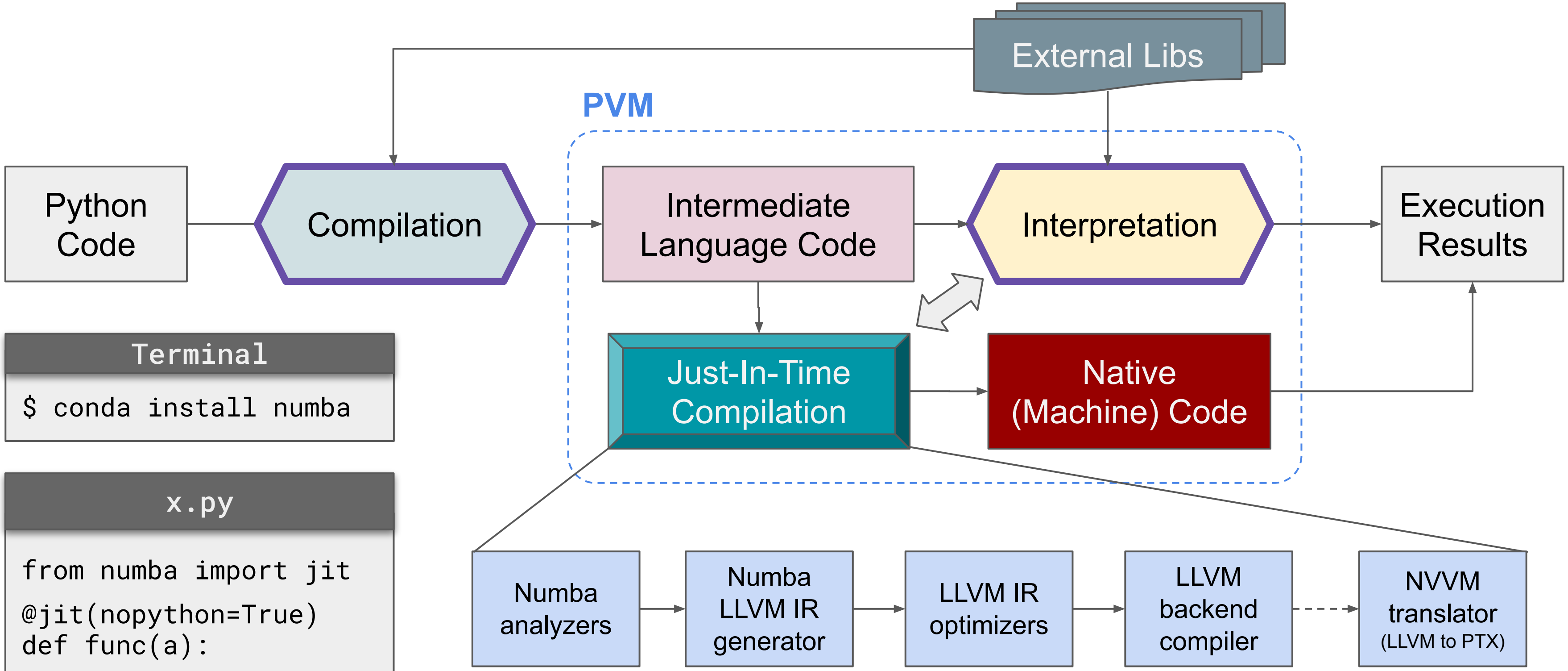
- RPython = Restricted/Reduced Python



Terminal

```
$ apt-get install pypy
$ pypy x.py
```

Numba: a high-performance python JIT compiler



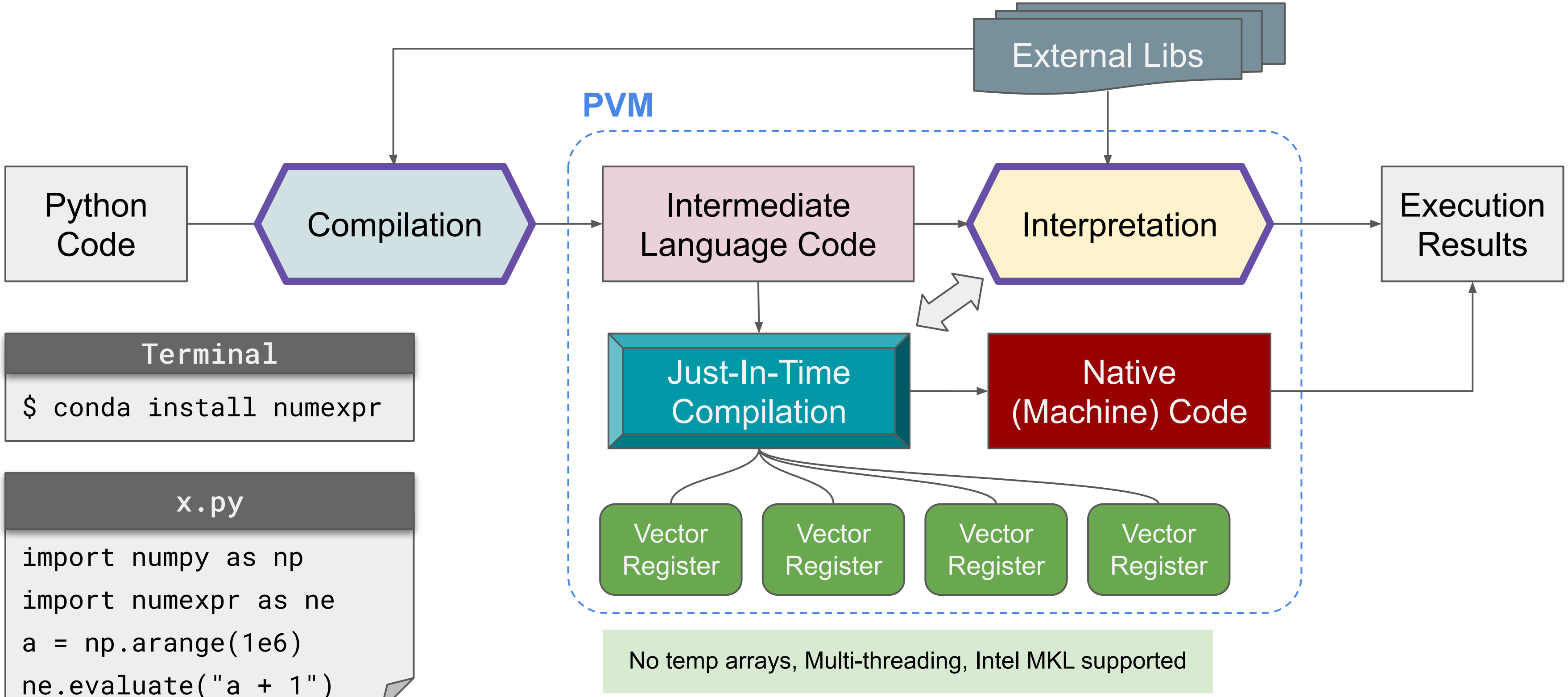
Terminal

```
$ conda install numba
```

x.py

```
from numba import jit
@jit(nopython=True)
def func(a):
    ...
```

NumExpr: C-based JIT booster for numpy large arrays



Terminal

```
$ conda install numexpr
```

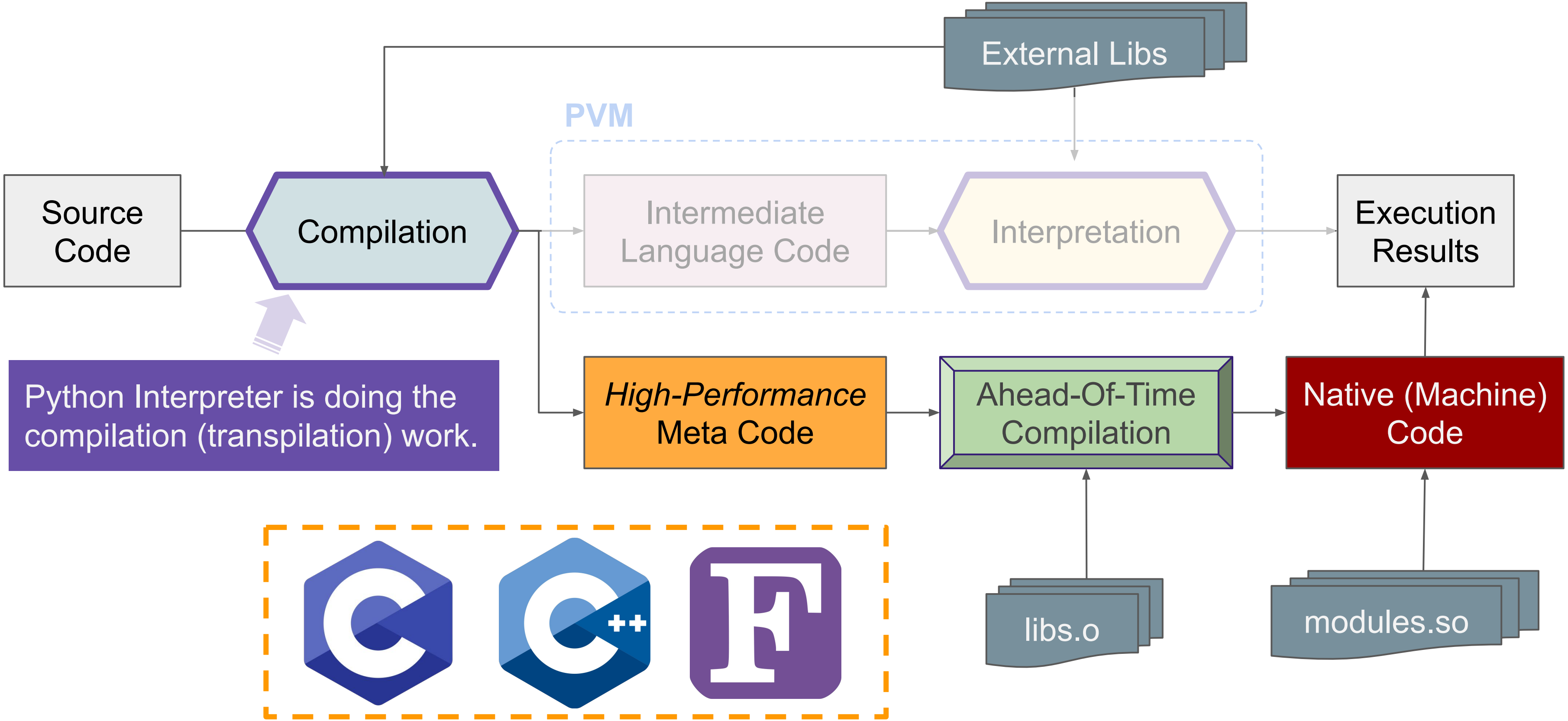
x.py

```
import numpy as np
import numexpr as ne
a = np.arange(1e6)
ne.evaluate("a + 1")
```

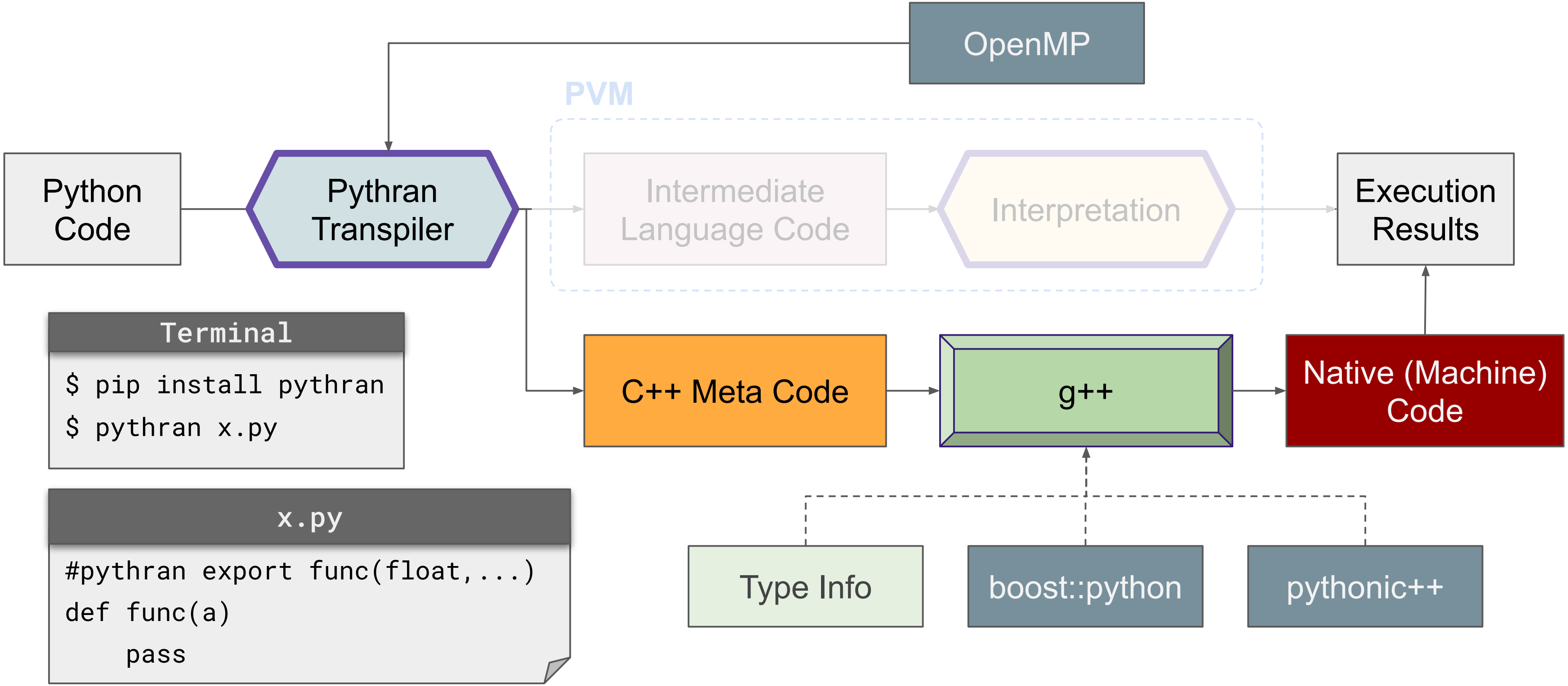
Outline for today

- What's the performance concerns for big data analytics?
- Why learning Python in the ChatGPT era?
- Why Python is slow?
- How to speed Python up using JIT?
- How to speed Python up using AOT?
- Colab demos

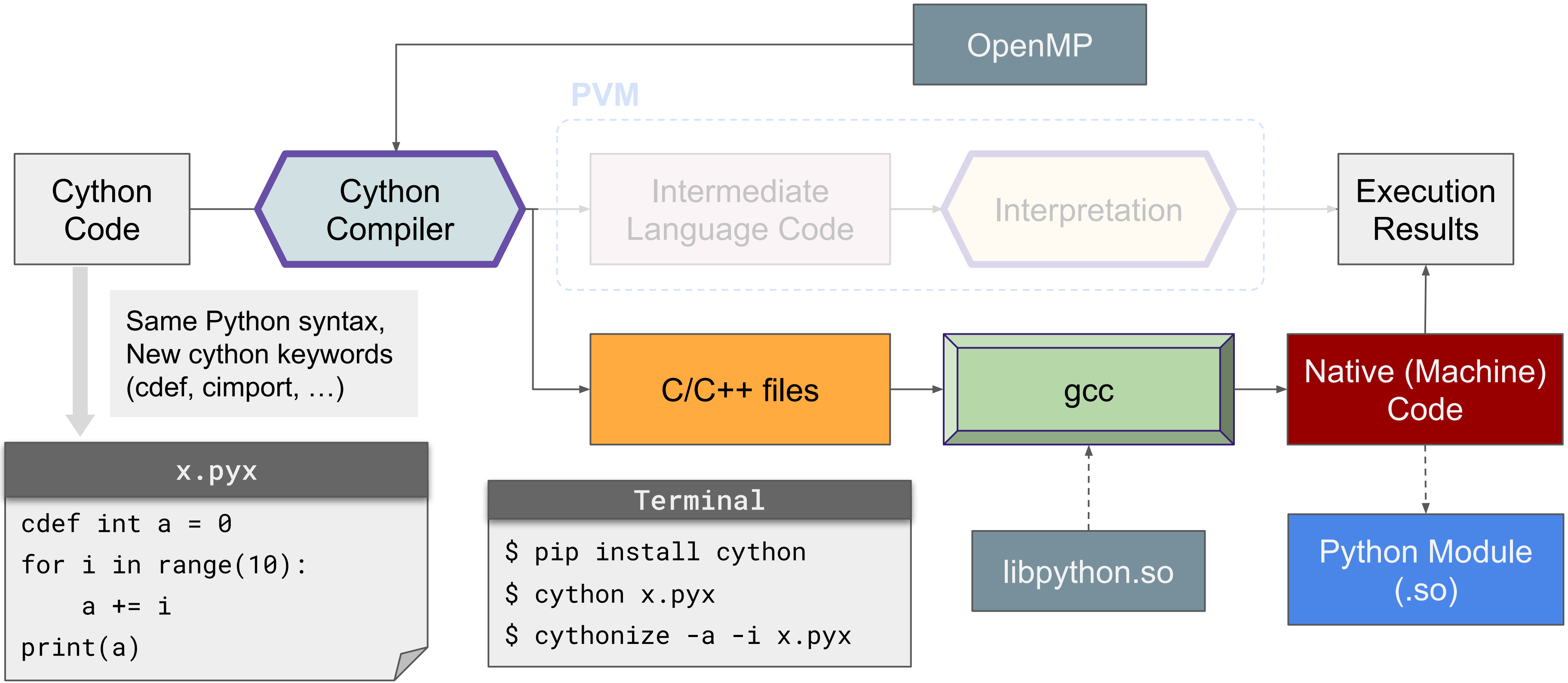
Boosting the speed by **AOT** Compiler



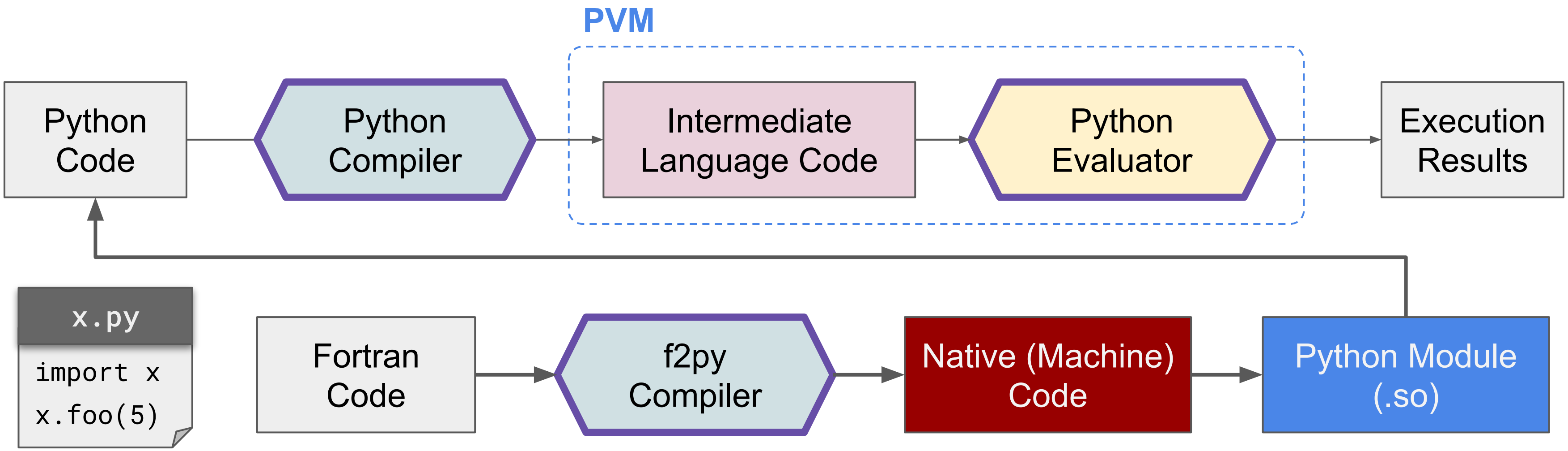
Pythran: an AOT compiler for a subset of the Python



Cython: Compiler to write C extensions for Python



f2py: wrap fortran code for use in Python



x.f

```
subroutine foo (n)
integer n
end
```

Terminal

```
$ pip install f2py
$ f2py -c -m x x.f
```

Similar idea for calling C/C++ code
- **SWIG**
- **Boost.python**
- **CFFI**
- **ctypes**

Outline for today

- What’s the performance concerns for big data analytics?
- Why learning Python in the ChatGPT era?
- Why Python is slow?
- How to speed Python up using JIT?
- How to speed Python up using AOT?
- Colab demos

Hands-on Demo

bit.ly/hpdspy_01

- Code example will be running in Google Colab.
    - `IPython` (interpreter implementation) as Python kernel in Jupyter Notebook
        - Based on `CPython`, enhancing interactive features.
        - Shell prompted as `In [#]:`
        - Interacting with external files/modules by `%`*magic* commands
        - Some comparisons were not made in the same baseline.
        - A "maybe" game changer: **PyScript**
    - Colab comes with some installed libraries, but not all.
    - Performance benchmark was done based on array operations
        - Started with 1000 points in 3 dimensions
        - Calculate the pairwise 1000x1000 distances
        - Arrays (containers, dataframes) will be our *main* subject to discuss in the next lecture.

Key Takeaways

- Before optimization:
  - Be sure everything is working properly in a simplest way.
  - Know what’s going on (data (dense/sparse?), code, algorithm, etc).
  - Profile your code and find the performance bottleneck.
- Optimization selection:
  - Consider the easiest way first (minimum changes), then the fastest.
  - Dive into the documentations
  - My personal preference order for performance optimization:
    - Scipy/Sklearn/numpy > Numba > Pythran > numexpr > Cython