High-Performance Data Science in Python

—— (1) Interpreter War ——

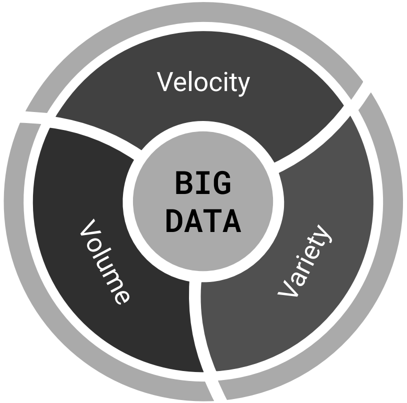
Qiyang Hu

UCLA IDRE/OARC Workshop

May 5th, 2021

About this series

- Performance concerns in big-data scenario:
  - Speed-up computation and processing
  - Handling big volume of data
- The lectures will focus on
  - *High-level* overviews.
  - Selectively introducing libraries that require least efforts to boost performance.



May 05, 2021	**Interpreter War** **(single CPU / GPU)**	• Pypy, Numba, NumExpr • Pythran, Cython • F2py, ctypes
May 12, 2021	**DataFrame Game** **(single node w/ many CPUs & GPU)**	• Numpy & Pandas • Modin, Pandarallel, Swifter • Dask DataFrame, Vaex
TBD	**Parallel Universe** **(Distributed multiple machines)**	• Joblib, Dask, Ray • PySpark, Elephas • Horovod, SINGA, Analytics-Zoo

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

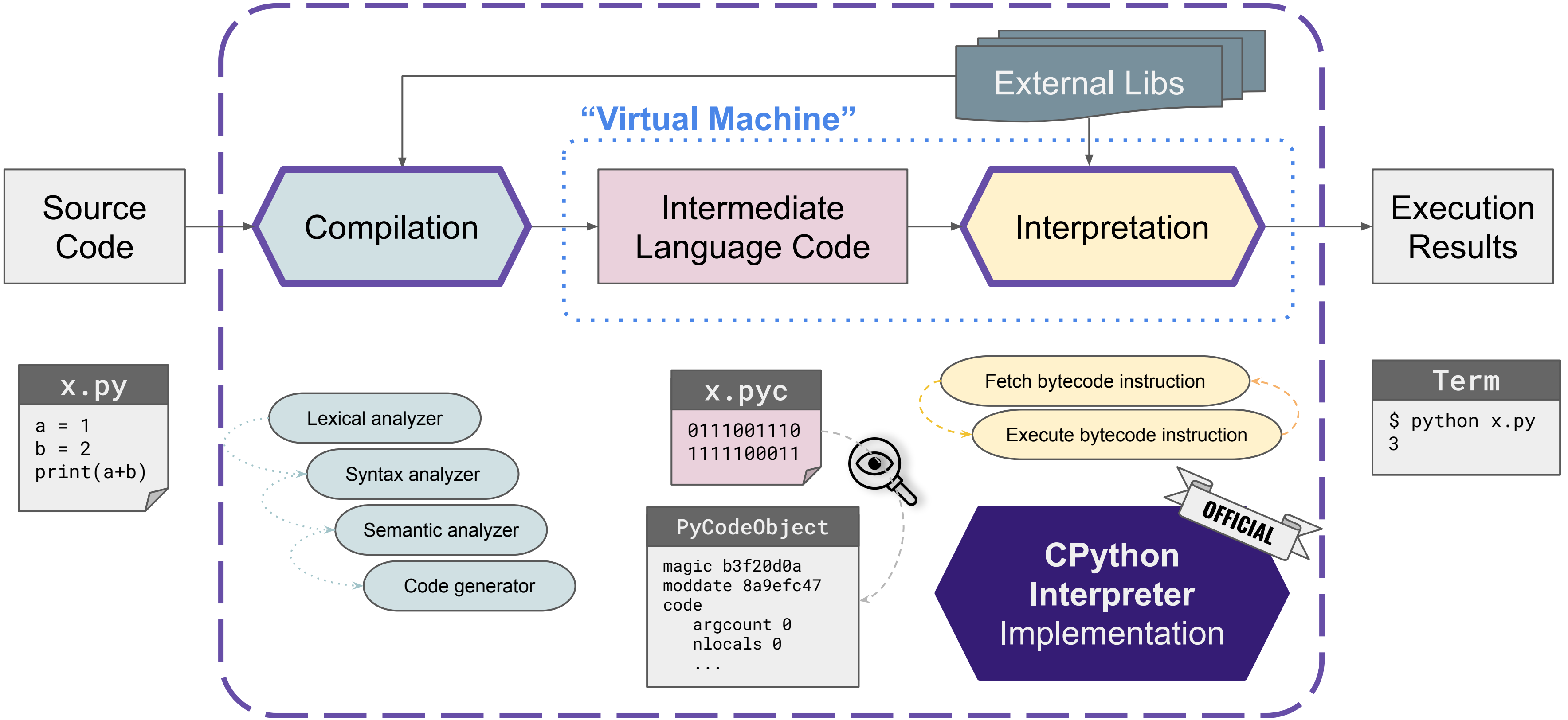
Don’t trust benchmarks.

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.

Seriously, what is Python?



Why Python is slow?

Python is Dynamically Typed rather than Statically Typed.

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

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

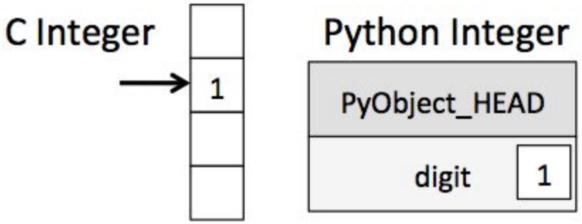
[Source]

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

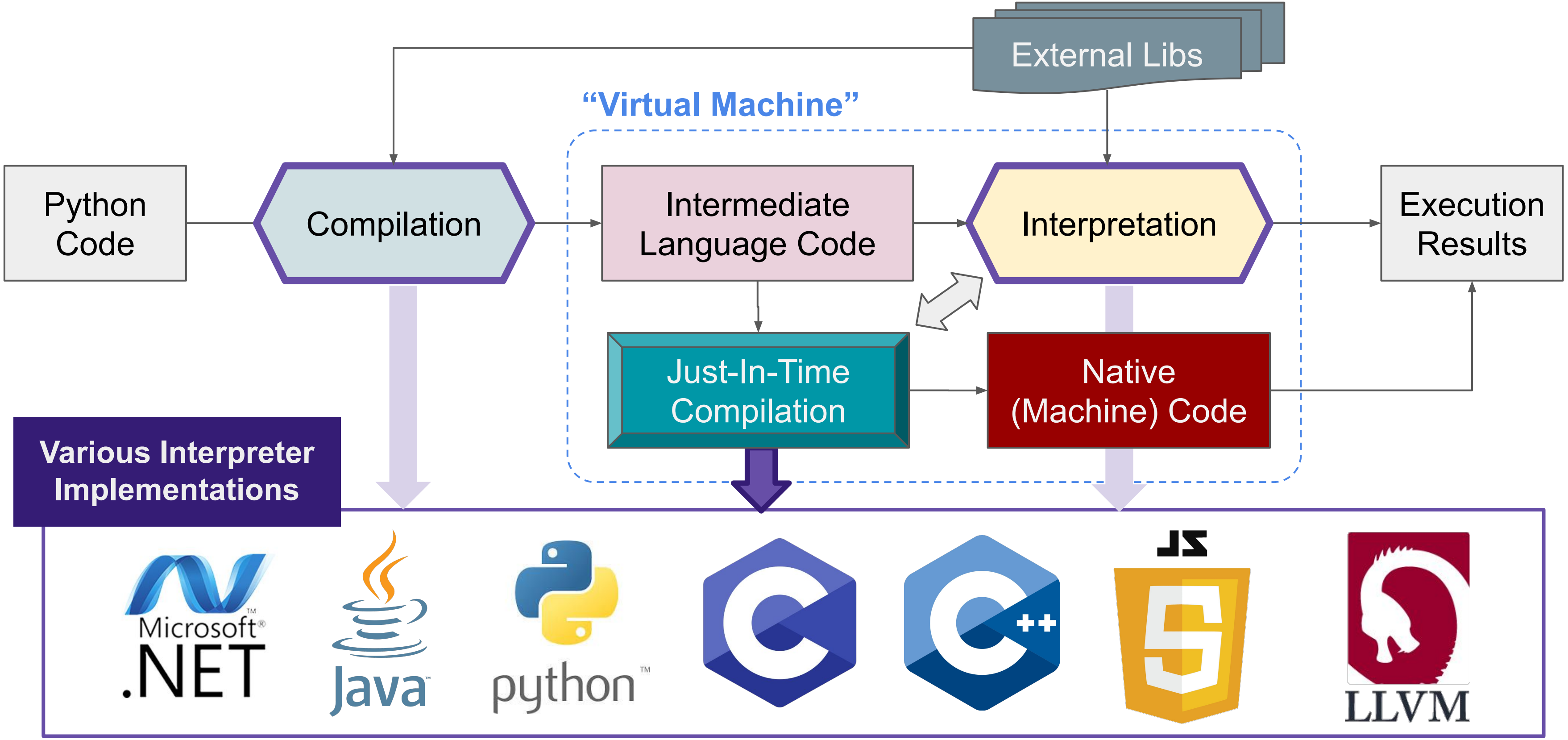
Python Addition

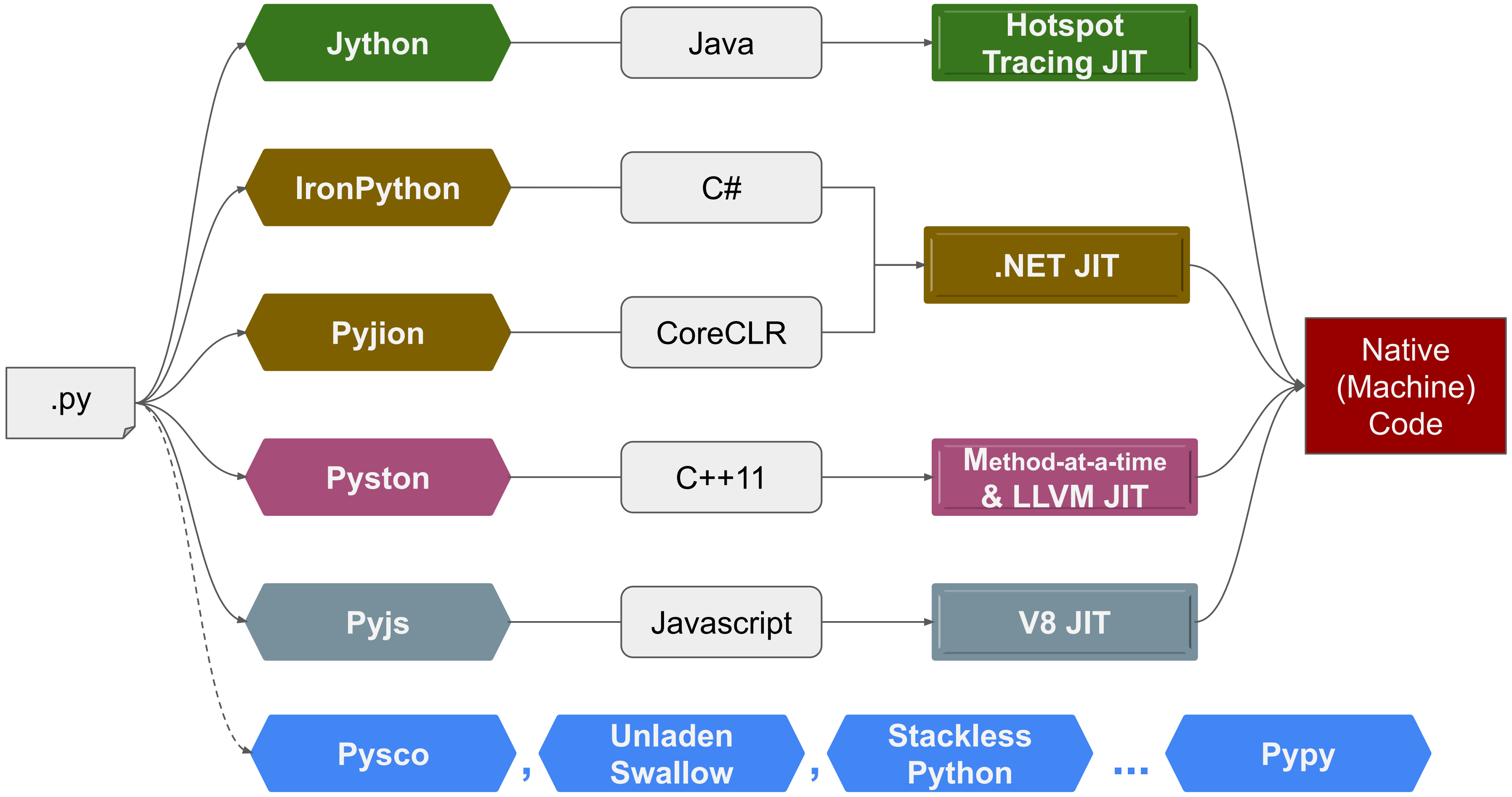
1. Assign `1` to `a`
   - **1a.** Set `a->PyObject_HEAD->typecode` to integer
   - **1b.** Set `a->val = 1`
2. Assign `2` to `b`
   - **2a.** Set `b->PyObject_HEAD->typecode` to integer
   - **2b.** Set `b->val = 2`



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`

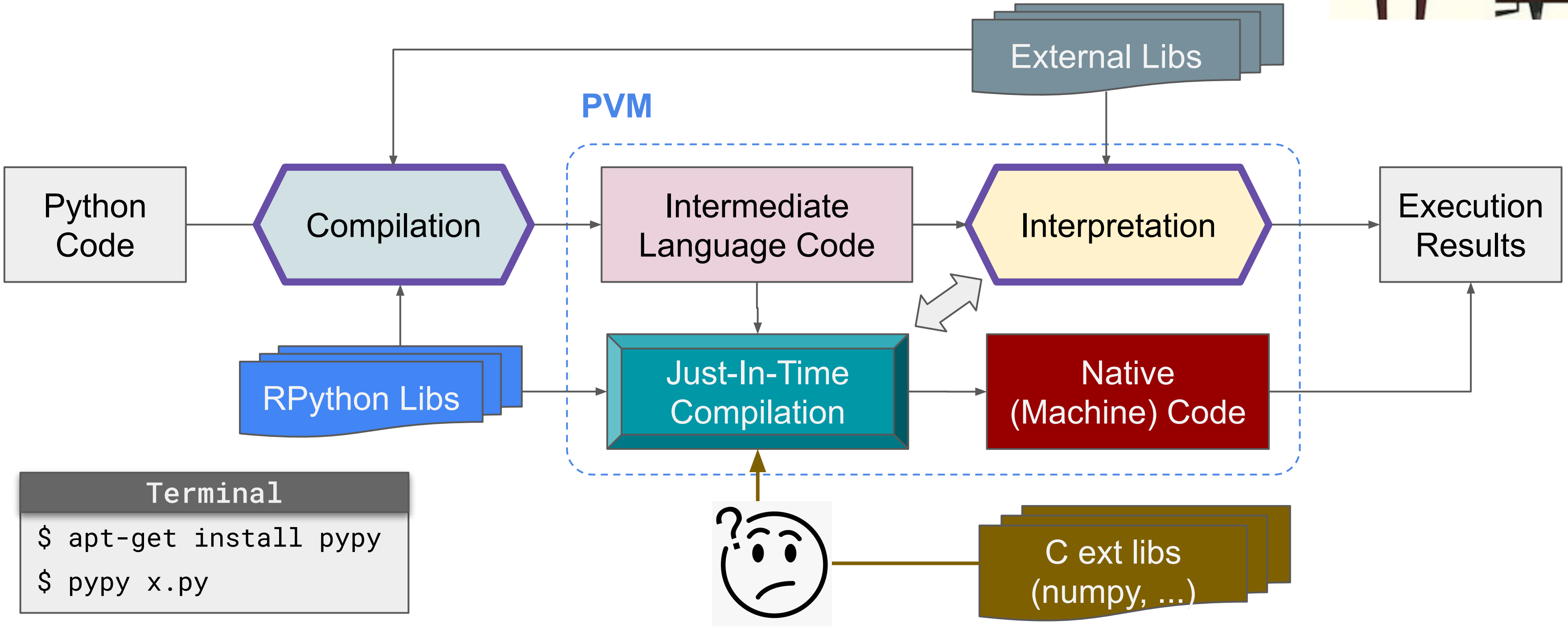
Boosting the speed by JIT




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

Pypy: using Python to interpret Python

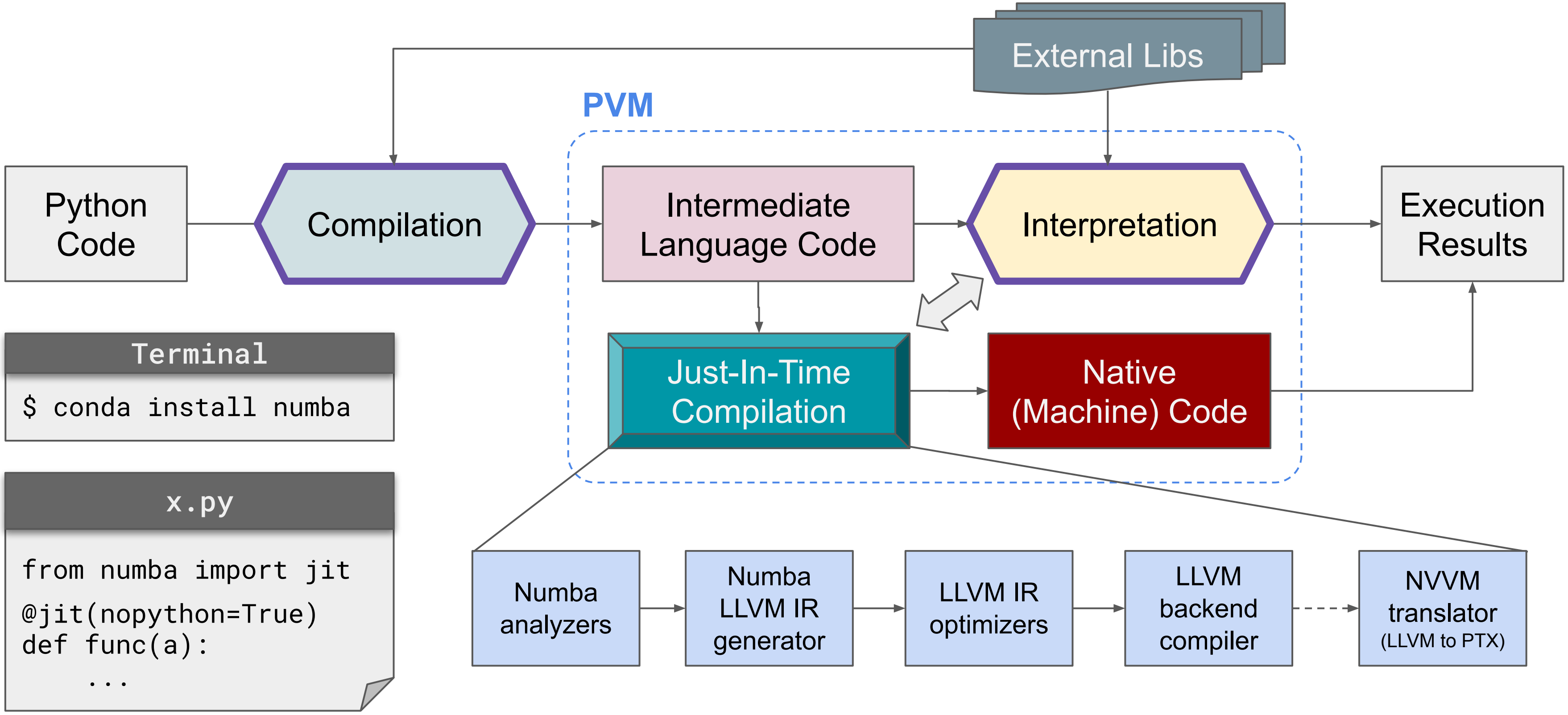
- RPython = Restricted/Reduced Python



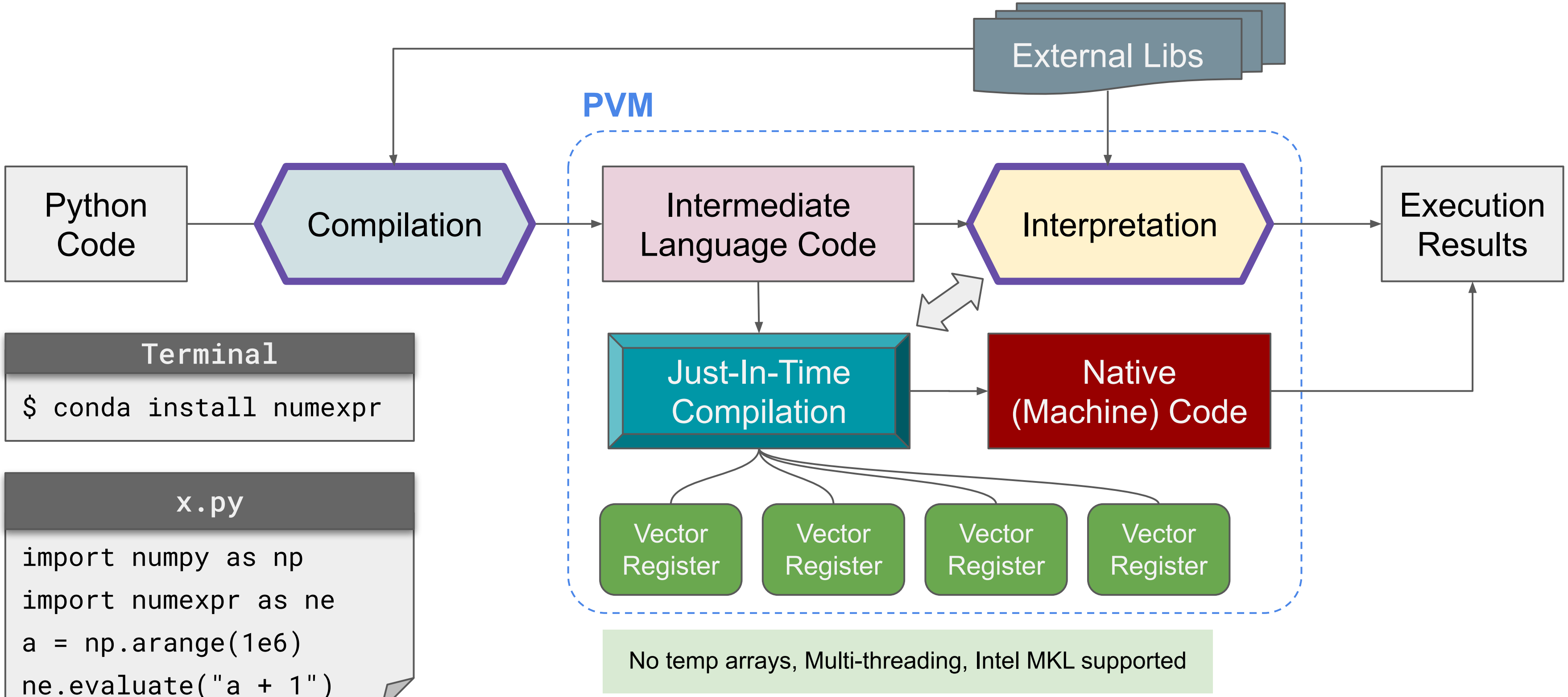
Terminal

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

Numba: a high-performance python JIT compiler



NumExpr: C-based JIT booster for numpy large arrays



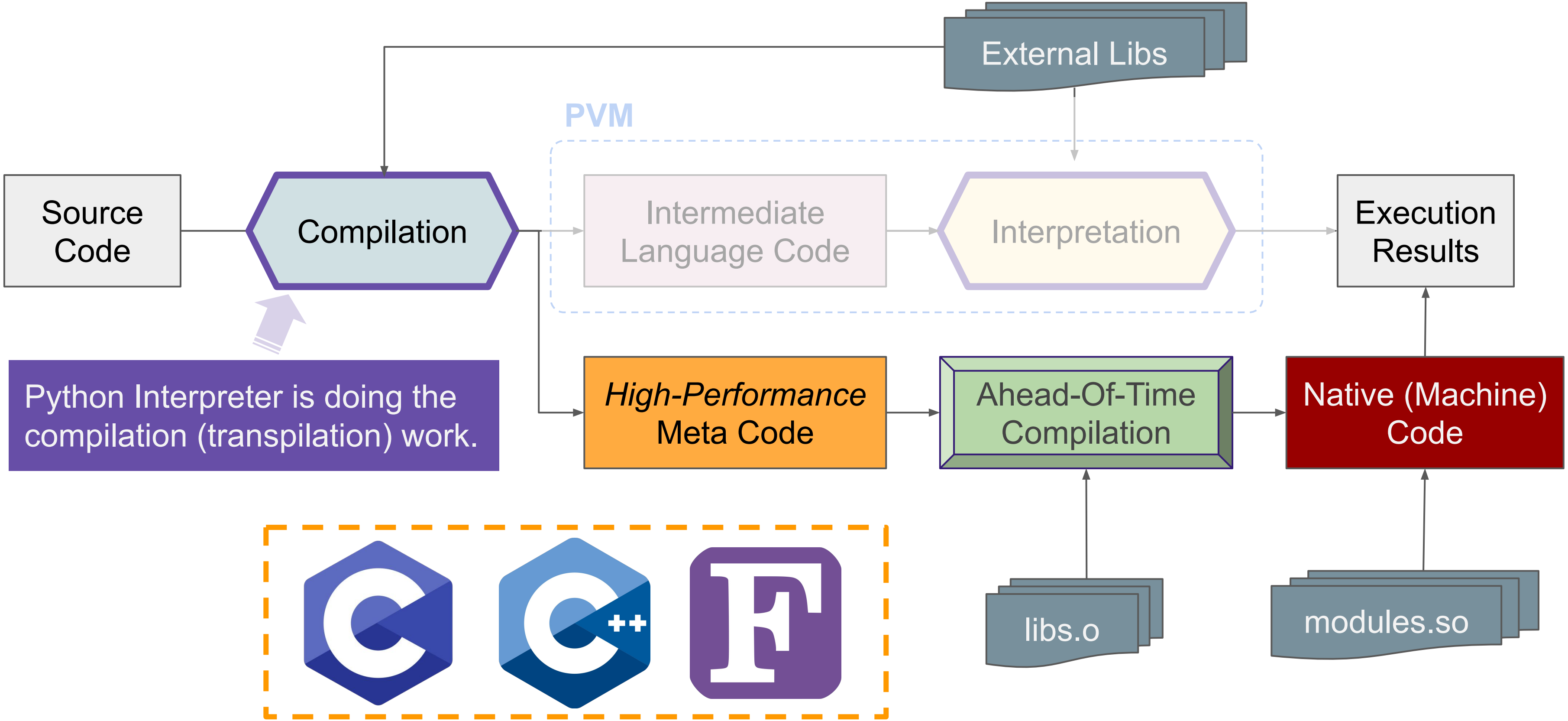
Terminal

```
$ conda install numexpr
```

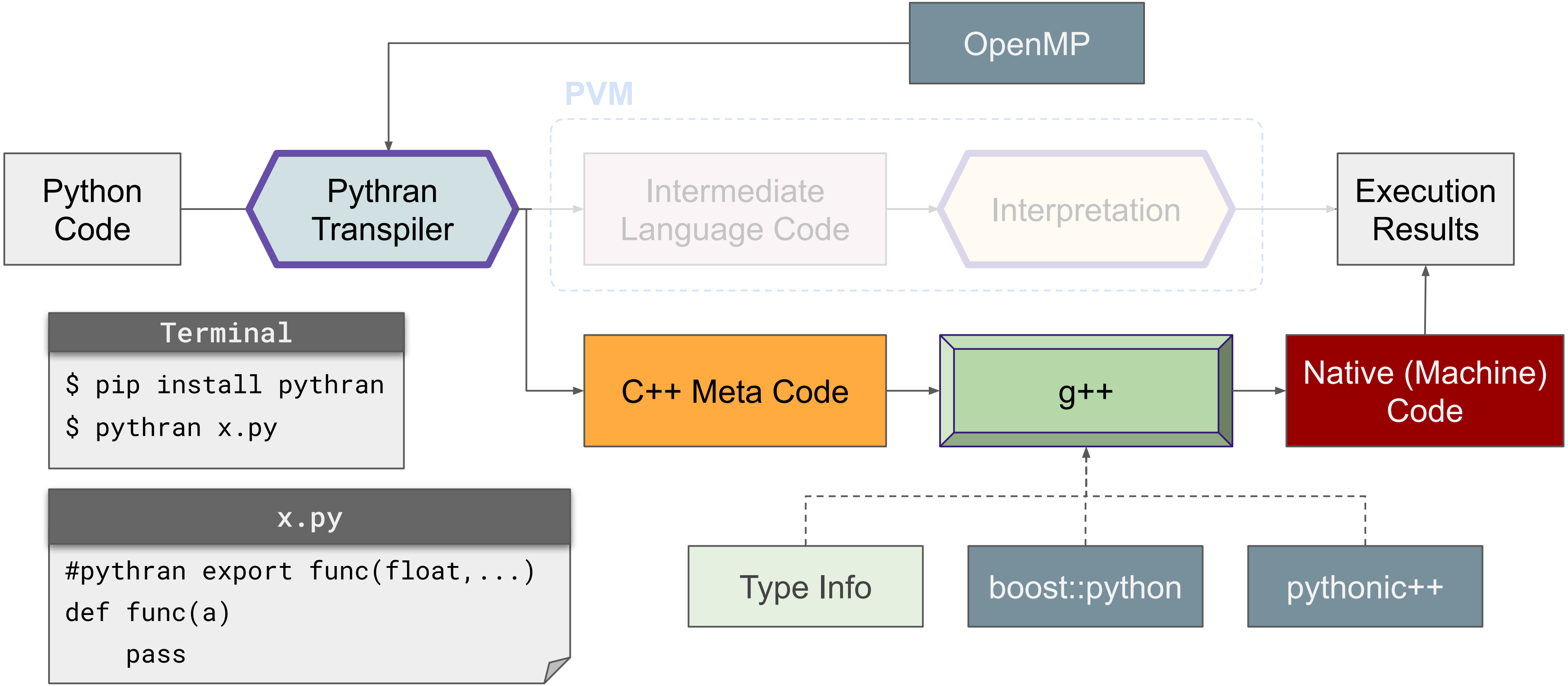
x.py

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

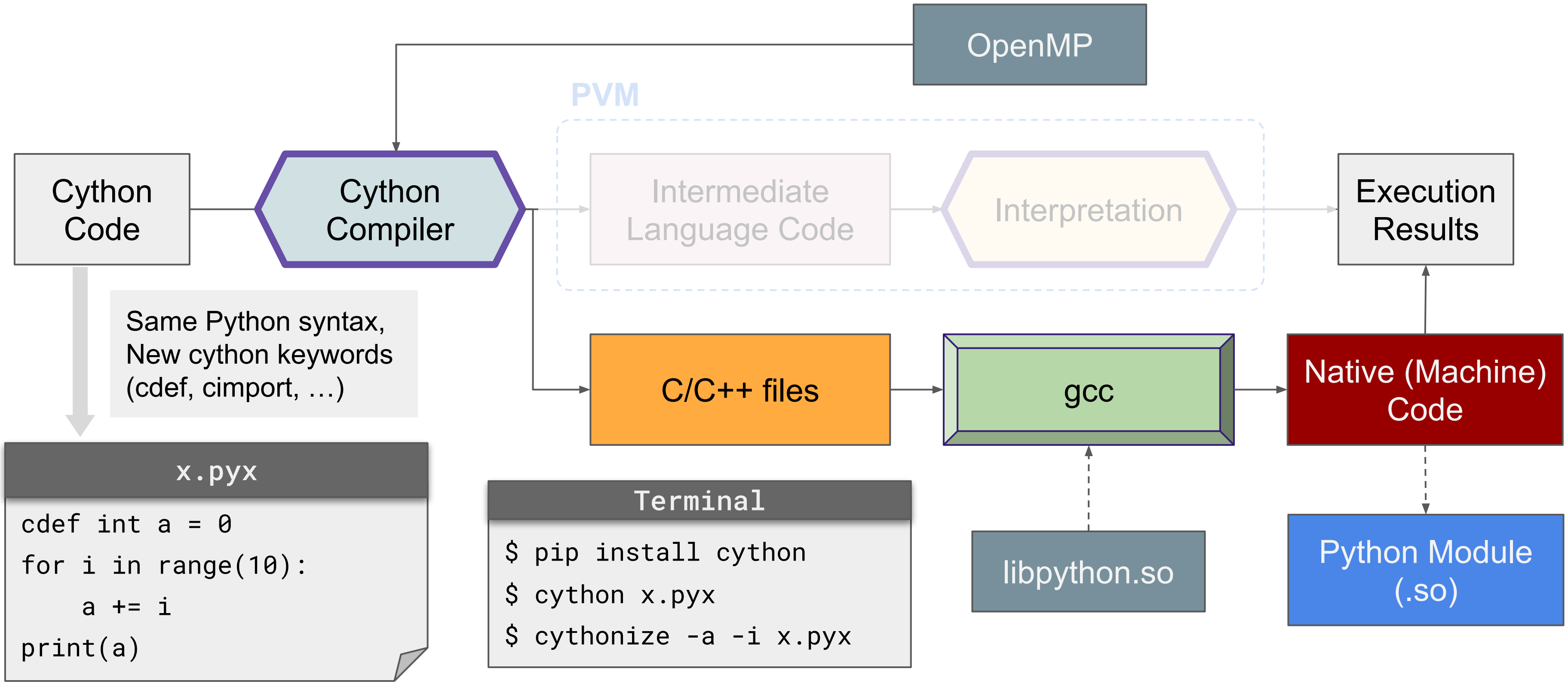
Boosting the speed by **AOT** Compiler



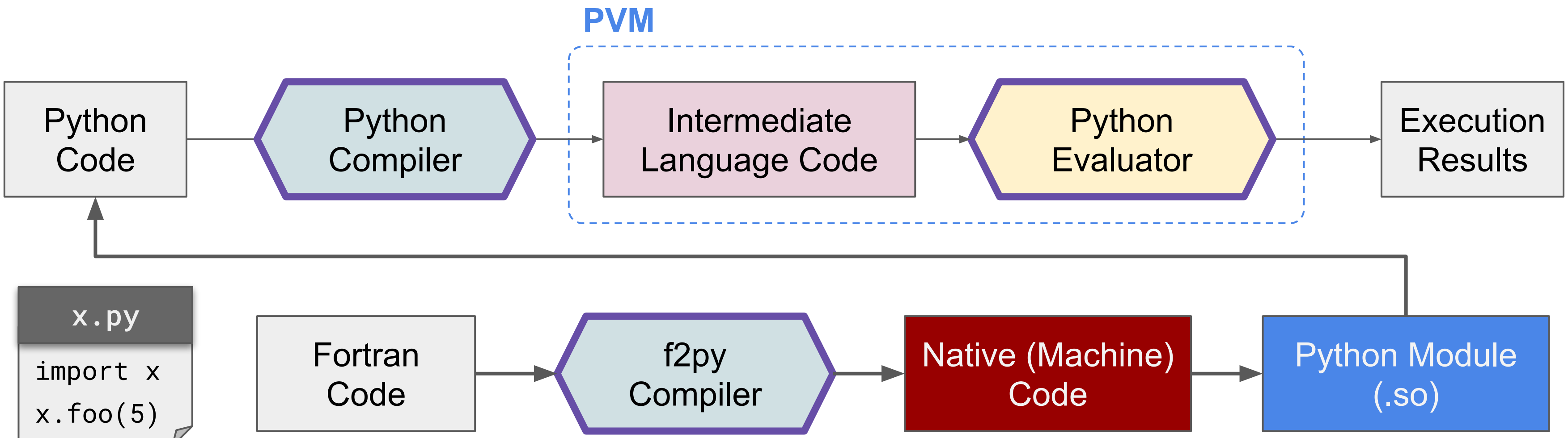
Pythran: an AOT compiler for a subset of the Python



Cython: Compiler to write C extensions for Python



f2py: wrap fortran code for use in Python



x.py

```
import x
x.foo(5)
```

x.f

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

Terminal

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

ctypes has the similar idea for calling C code.

“Guilt” of GIL

- 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,
  - Alternative Python interpreters:
    - GIL only with CPython
    - multiple interpreter implementations
  - Some people are working on attempts to remove the GIL from CPython: like Gilectomy

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.
    - 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 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