Python on HPC

Corey Adams, ACLF

Why this Talk?

Python is a crucial productivity language

Python is versatile and easy:

- Prototype many functionalities **quickly**
- Easy to glue together independent pieces of software with wrappers, bindings, and utilities
- Very large open source package community
- Easy to maintain
- Datascience and Machine learning language of choice

Python is firmly entrenched as THE productivity language. So, how do you use it correctly on HPC?

https://stackify.com/popular-programming-languages-2018/

Programming Language	Ratings	Change
Java	16.028%	-0.85%
C	15.154%	+0.19%
Python	10.020%	+3.03%
C++	6.057%	-1.41%
C#	3.842%	+0.30%
Visual Basic .NET	3.695%	-1.07%
JavaScript	2.258%	-0.15%
PHP	2.075%	-0.85%
Objective-C	1.690%	+0.33%
SQL	1.625%	-0.69%
Ruby	1.316%	+0.13%
MATLAB	1.274%	-0.09%
Groovy	1.225%	+1.04%
Delphi/Object Pascal	1.194%	-0.18%
Assembly language	1.114%	-0.30%
Visual Basic	1.025%	+0.10%
Go	0.973%	-0.02%
Swift	0.890%	-0.49%
Perl	0.860%	-0.31%
R	0.822%	-0.14%

Which Python? 2 or 3? Conda? …?

We have many pythons installed on theta:

intelpython27/2019.3.075
intelpython36/2019.3.075
Miniconda-* (2.7, 3.6, login or compute nodes …)
Cray-python/2.7, 3.6 ….

You are also able to bring your own python! Build from source, or use conda, or use pip, or … [your favorite install method here!]

Beware to get the right libraries for performance (intel numpy vs normal numpy, etc)

Virtualenv – use it!

https://virtualenv.pypa.io/en/latest/

“virtualenv is a tool to create isolated Python environments.” – from the docs

Virtual env is a way to isolate a python environment (either completely encapsulated, or based on another python interpreter but with encapsulated libraries). For example … (next slide)

Setting up a virtualenv: an example

```
cadams@thetalogin5 : ~
$ python -m virtualenv --system-site-packages test_env/
Using base prefix '/soft/interpreters/python/3.6/intel/2019.3.075'
New python executable in /gpfs/mira-home/cadams/test_env/bin/python
Installing setuptools, pip, wheel...
done.

cadams@thetalogin5 : ~
$ which python
/soft/interpreters/python/3.6/intel/2019.3.075/bin/python

cadams@thetalogin5 : ~
$ python
Python 3.6.8 |Intel Corporation| (default, Mar  1 2019, 00:10:45)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-15)] on linux
Type "help", "copyright", "credits" or "license" for more information.
Intel(R) Distribution for Python is brought to you by Intel Corporation.
Please check out: https://software.intel.com/en-us/python-distribution
>>> ^D
```

```
cadams@thetalogin5 : ~
$ source test_env/bin/activate

(test_env)
cadams@thetalogin5 : ~
$ which python
/gpfs/mira-home/cadams/test_env/bin/python

(test_env)
cadams@thetalogin5 : ~
$ python
Python 3.6.8 |Intel Corporation| (default, Mar  1 2019, 00:10:45)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-15)] on linux
Type "help", "copyright", "credits" or "license" for more information.
Intel(R) Distribution for Python is brought to you by Intel Corporation.
Please check out: https://software.intel.com/en-us/python-distribution
>>>
```

CPython: Under the hood

You don’t need to know all the details, but it helps to have a basic understanding

---

Python is a Read-Eval-Print Loop

- One line at a time
- No look-ahead for optimizations
- It is not thread safe, and therefore the **Global Interpreter Lock** enforces single thread utilization
- Everything is an object
- Everything is a reference

Everything is an object, everything is a reference

Each object in python is reference counted: every time you create a reference to it, it is counted, and when an object has no references to it, it is deleted.

Since everything is a reference, and everything is an object, python is very friendly with letting you change types on the fly as input to functions, etc.

- You may even implement type specific logic with simple if/else
- You may even completely shoot yourself in the foot by not know object types…
- Each object has to be queried for it's type on the fly, and this causes a large slow down. Maybe more than you realize!

Dynamic typing is one of python's greatest assets for ease, and greatest weaknesses for performance.

CPython

The core operation of python lines is via the interpreter, which itself is written in C.

- It doesn't have to be, though, it is simply that the most common (by far) implementation of python is writtin in C.
- As a C program itself, python has native compatibility to load C functions into your own python programs - more on this later.

There are other implemetations of python: jython, iron-python, PyPy

- All of these have their own reasons for existing. In particular Pypy has a Just-In-Time compiler for performance boosts.
- For HPC, CPython is perfectly acceptable and the different python versions almost certainly are not your bottleneck.

Global Interpreter Lock

Cpython's memory management is **not thread safe,** and the global interpreter lock is an enforcer to prevent problems arising from this.

- It's not all bad: it makes extensions easier to write, is generally faster on single threads.
- It's also almost entirely irrelevant in the HPC space where you have optimized C libraries for compute heavy tasks.

Don't fear the GIL. When you need concurrency in python, it's typically best to either:

- Use a C library or write one that does what you need, where you can manage memory yourself.
- Use a clean parallelization paradigm like MPI (which has excellent python bindings!)
- GIL is most impactful in compute heavy operations - optimized libraries exist!

Python Wrappers

The best python code is using python to glue together high performance libraries

---

"A **foreign function interface** (**FFI**) is a mechanism by which a program written in one programming language can call routines or make use of services written in another." – Wikipedia

For simple bindings, there is ctypes: https://docs.python.org/3/library/ctypes.html

For some projects, there is a C-API: https://docs.scipy.org/doc/numpy/reference/c-api.html

Cython can help expose in both directions: https://cython.org/

For complex projects, you can auto generate wrappers. Many options available:

https://wiki.python.org/moin/IntegratingPythonWithOtherLanguages

Boost python, sip, swig are popular and relatively well supported.

Doing python right

Use the right tool for the right task

- Use optimized libraries, and use the **right** optimized libraries:
  - Numpy from pip != numpy with mkl operations from intelpython.
  - When in doubt, use the modules installed on Theta or contact the datascience team.
- Don’t expect parallelism in python, except for very specific cases it’s better to try with other models directly.
  - MPI, OpenMP, etc.

Doing python wrong

It’s easy to mess up your python

- Use the right data structures for your data: don’t use a list of floats when you can use a numpy ndarray.
- **Do** loops with comprehensions instead of for:
  - See example.
- **Don’t** compute with python
  - Use numpy, tensorflow, torch, scipy, etc
- Be careful mixing and matching parallelism:
  - Tensorflow/torch/numpy/mkl/mkldnn all use OpenMP or other threading, and it is incredibly easy to oversubscribe with many threads.
- Mine own personal grudge: **don’t** ‘import perl’

Numpy

When you need performance in python on computation, think **numpy**.

- Supports N dimensional arrays with accessing and striding, sorting, indexing, etc. It is a tensor library.
- Built in operations (FFTs, matmult, tensor ops)
- Heavily optimized with support for many architectures, targeted by industry for optimizations for their products (intel has an optimized numpy)
- Is is the standard and reference for both performance and functionality:
  - Tensorflow, pytorch play nice with numpy
  - Numpy access patterns (strided access, masks, etc) are the functionality that other frameworks try to match.

Scipy

When you need performance in python on computation for science operation, think **scipy**.

- Well integrated with numpy, supports operations that numpy is missing
  - Optimization, integration, linear algebra, FFT, image processing
- Scikit-learn is an additional step beyond scipy worth looking in to

Cython

“Cython is an **optimising static compiler** for both the **Python** programming language and the extended Cython programming language (based on **Pyrex**). It makes writing C extensions for Python as easy as Python itself.”

- This is a half truth, it is not *quite as easy.*
- http://docs.cython.org/en/latest/index.html

Cython is a wrapper that looks at python code and turns it into C. You get some benefits:

- Static typing removes a lot of overhead and improves performance
- Look-ahead compilation allows for optimizations
- Cython turns your code into C, making it very easy to call out to C libraries.
- Native **numpy** support

Cython has some downsides:

- You end up re-implementing every API call with a wrapper function, by hand.
- You python code is not always clean
- The build system is a weird hybrid of python’s setup.py + C compilers

h5py

The h5py package is a Pythonic interface to the HDF5 binary data format.

- https://www.h5py.org/
- Compatible with hdf5 files directly from python, with pythonic interface and compatibility with numpy and fancy-indexing.
- Is a Cython implemented wrapper of the HDF5 C API
  - Development on github, large community of support
- Compatible with parallel IO using mpi4py – must have parallel hdf5 library underneath.
- Has a lot of knobs for performance (chunking, compression) and it's easy to get great IO performance directly from python
  - It's also easy to get bad performance – it's always worth it to study your IO performance!

Parallelism + Python

Parallelism in python is not necessarily hard, but the GIL limits your options

Threading is all within one python GIL, so suffers from serialization of calls in python. But, simple interface and can be useful – relatively low overhead if you can do parallel steps in IO or C libraries.

Multiprocessing sidesteps the GIL by using entirely separate subprocesses

- Easy interface like the **threading** module, but has overhead. Good for compute-bound operations.
- Best way to use multiprocessing is to send messages, not share resources.
  - At this point ... Just use MPI!

MPI + Python

MPI (Message Passing Interface) is the tool of choice for internode communication, but is also valuable for intranode communication

- Be wary of too much parallelism: mixing and matching too many frameworks can oversubscribe the resource and slow things down.
- Use “MPI + X” (MPI + OpenMP, MPI+tbb, MPI + Pthreads, etc…)

MPI has a tidy wrapper class in python using Cython.

- Collective operations supported
- Passing messages supported for numpy objects at near-C speed
- Passing messages supported to all python objects, but slower
  - Leverages “pickle” for serialization

mpi4py

Building mpi4py on you laptop is easy, but on Theta (or any HPC system) make sure to use the right MPI.

- For Theta, this usually means Cray's MPI

Mpi4py is compatible with horovod for machine learning, and compatible with h5py for parallel IO.

- If you run large scale distributed learning, odds are mpi4py is very useful to your workflow to coordinate and manage your worker processes.

Mpi4py is a valid intranode parallelism scheme: instead of threading or multiprocessing, you can use mpi4py to manage independent python processes.

- Additionally, this will easily scale beyond a single node.

Python Packages

197,710 projects 1,476,851 releases 2,176,956 files 375,362 users

From pypi.org: nearly 200k python projects are open source and available with pip install.

There are massively popular packages (think numpy) and incredibly specific packages.

Odds are, if you are doing something in python, someone has done it before and released it and it is useful to you!

You can use virtualenv to add packages onto available pythons! (intelpython36, cray-py, etc)

Python “Dependency Hell”

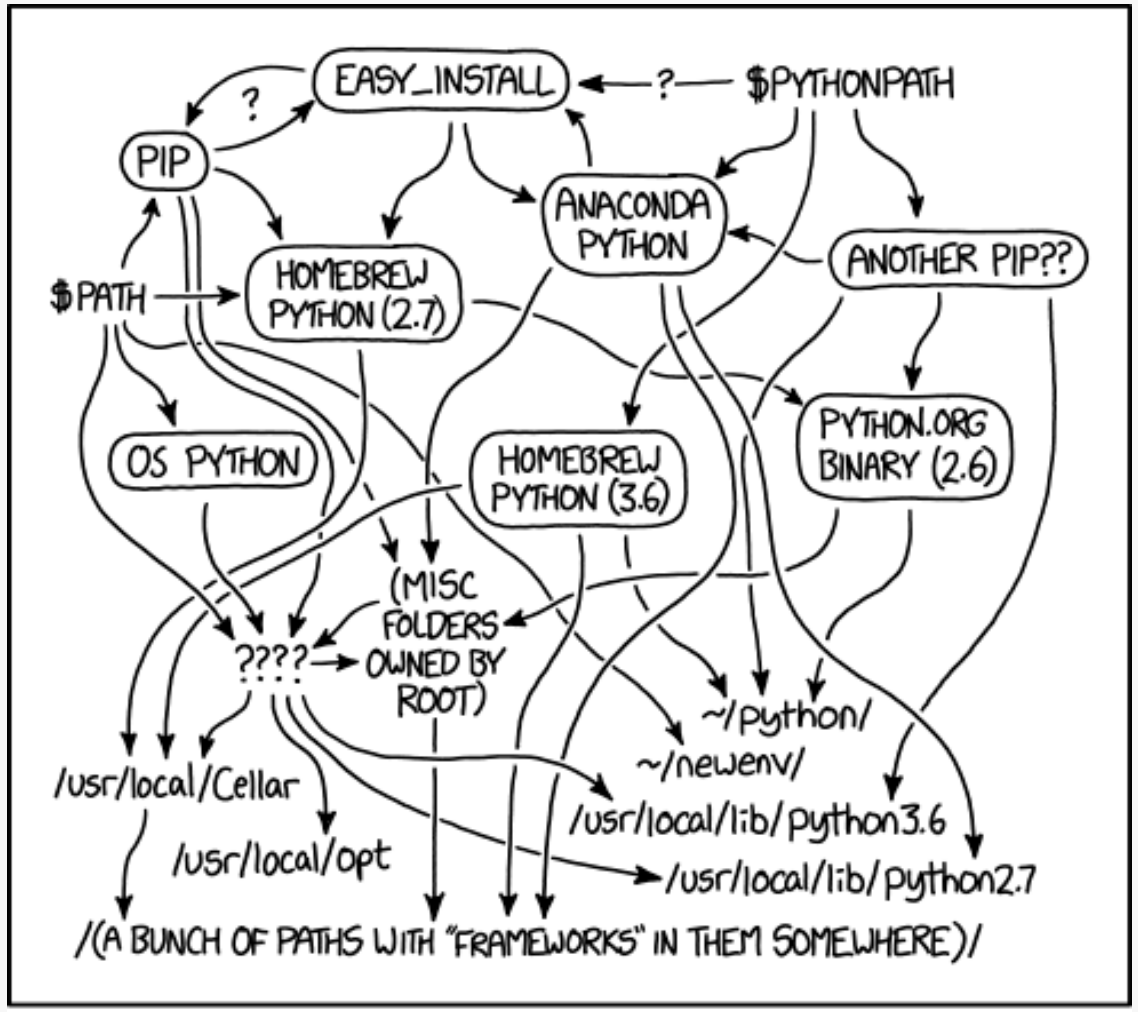
Python can load “modules” which is any set of python files with an __init__.py file.

- Sometimes, these pull in other modules or libraries
- Python checks sys.path to find modules, which include a home area, python’s installation area, and other locations – you can examine and modify it interactively too
- Package managers (pip, conda, even modules on theta) slip easily into “dependency hell”

From Misha Salim:

1. When in doubt, check `which python`
2. Use virtualenvs for every project; minimize global environment pollution
3. Avoid LD_LIBRARY_PATH at all costs
4. If it's not in sys.path, it's not going to be found by the importer

Python “Dependency Hell”



https://xkcd.com/1987/

My Advice:

1. Keep it simple. Don’t install package you don’t need.
2. Keep notes. Write down how you installed packages into an env, and how to set it up.
3. Use virtualenv when you have a python binary but need additional packages – mitigates conflicts.
4. Watch out for implicit dependencies in package managers. Conda tensorflow may install a new python and it’s own hdf5. Do you really want this?

The future of python

Python 2 or Python 3?

Python 3.

- Python 2 is end of life at the end of this year, and there is no assurance that it will be available on Theta after that.
- New packages are coming to python3, python2 support is limited.

Profiling Python

I hope it is clear that performance can be an issue with python.

It sounds redundant: The best way to use python properly is to check if you are using it properly!

There are many tools available:

- Using printouts of times (crude but useful!)
- cProfile – building python profiler gives function call stacks and function times
- Line_profiler requires adding small snippets to your code, and running in a special wrapper, but gives line-by-line measurements of the functions you specify. **It is extremely useful for pure python code.**
- Intel Vtune – great for mixed language workflows (which you should be using!) but challenging to use for python on Theta. **Does not run with many threads going underneath python (ie, tensorflow with OMP_NUM_THREADS == 64)**

Examples

There are several examples provided to show some python tools:

Matrix Multiplication

- pure python
- cython
- numpy