Python for High-Performance Computing

Stephen Szwiec

Center for Computationally Assisted Science and Technology

June 11, 2024

Outline

- Presentation
- Live Demo
- Open Discussion and Questions

Outline

How did we get here?

Introduction

Figure 1: Ken Thompson and Dennis Ritchie at Bell Labs, circa 1970. ©Peter Hamer, 2011.

Introduction

- Unix-like operating systems are the most common operating systems used in both academia and industry for scientific computing.
  - Unix was developed at Bell Labs in the 1970s by Ken Thompson, Dennis Ritchie, and others.
  - Efforts by the Free Software Foundation and Linus Torvalds led to the development of Linux in the 1990s.
- Linux is a Unix-like operating system, and shares common ideas with Unix (BSD, MacOS, iOS, etc.)
  - Linux is also the basis for Android.

Introduction

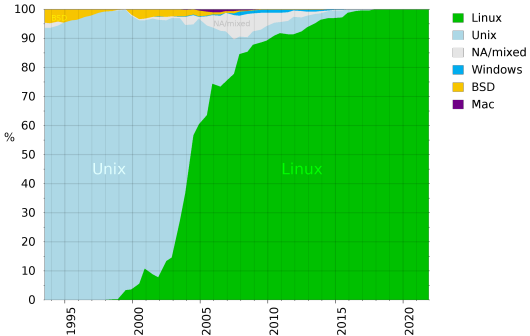


Figure 2: Top 500 Supercomputers, by operating system, 1993–2021 ©Benedikt Seidl, 2021.

- Together, Unix and Linux:
  - 99% of smartphones
  - 31% of personal computers
  - 80% of servers
  - 100% of TOP500 supercomputers
- Familiarity with these systems is essential for scientific computing, cloud computing, high-performance computing, and mobile development.

Introduction

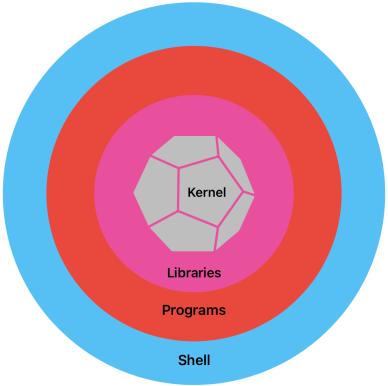


Figure 3: An idealized view of the Unix-like operating system.

- The Unix-like operating system is composed of three main parts:
  - **Kernel:** the core of the operating system, which manages the hardware.
  - **Libraries:** abstract interfaces to the kernel, which provide common functionality.
  - **Programs:** use the libraries to perform tasks.
  - **Shell:** a command-line interface to the operating system.
- The shell is the primary interface to the operating system.

The Shell and Shell Commands

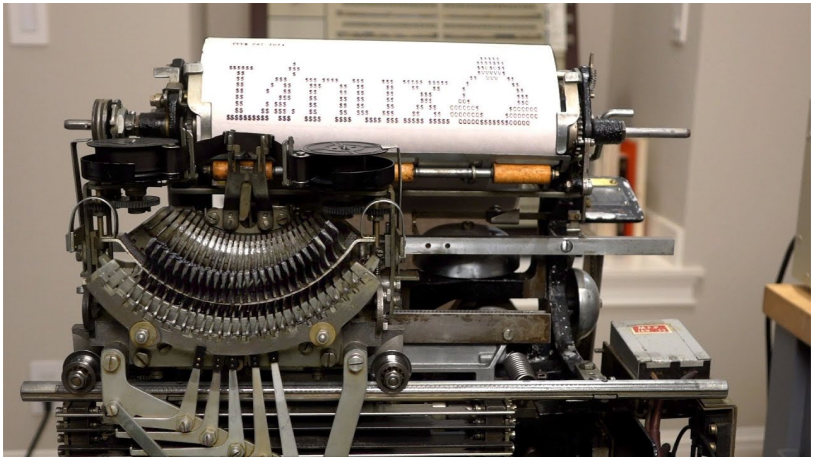


Figure 4: A 1930s Teletypewriter, connected to a modern Linux system via SSH. © CuriousMarc, 2020 [https://youtu.be/2XLZ4Z8LpEE]

The Shell and Shell Commands

- The shell is a command-line interface to the operating system.
  - The shell is a program that interprets commands and executes them.
  - The shell is also a Turing-complete scripting language.
- The shell is the primary interface to the operating system.
  - Implements a read-eval-print loop (REPL.)

The Shell and Shell Commands

- Shell scripts can be used to automate tasks, and are often used to run programs on high-performance computing clusters.
- Quick example: script to create a series of numbered directories:

```
$ for i in {1..10}; do mkdir $i; done
```

Environment Variables

- If the shell is just a type-and-run interface, how does the shell *know* what to do?
  - `$ which mkdir`
- The shell uses environment variables to store information about the system and the user:
  - Environment variables are key-value pairs that are used by the shell and programs to determine how to behave.
  - In the above example, we set a variable `i` to a value from 1 to 10, and called it with `$i`.

Environment Variables

- Common environment variables:
  - `$PATH`
  - `$LD_LIBRARY_PATH`
  - `$HOME`
- Worth noting for CCAST Portable Batch System (PBS) jobs:
  - `$SCRATCH`
  - `$PBS_NODEFILE`
  - `$PBS_O_WORKDIR`
- Environment variables can be set, unset, and used in the shell:
  - `$ export VARIABLE=value`
  - `$ unset VARIABLE`
  - `$ echo $VARIABLE`
  - `$ echo ${VARIABLE}`

Environment Variables

- On an HPC cluster, there are many different software packages installed.
  - Some software packages are incompatible with others.
- Environment modules are used to manage software packages on an HPC cluster:
  - set environment variables for a specific software package.
  - can be loaded and unloaded as needed.

```
$ module load cuda/12.3
```

- loads the CUDA 12.3 environment module on the cluster
- just sets `$PATH` and `$LD_LIBRARY_PATH`.

The shell is a scripting language, environment variables are part of the language.

History of HPC to Python

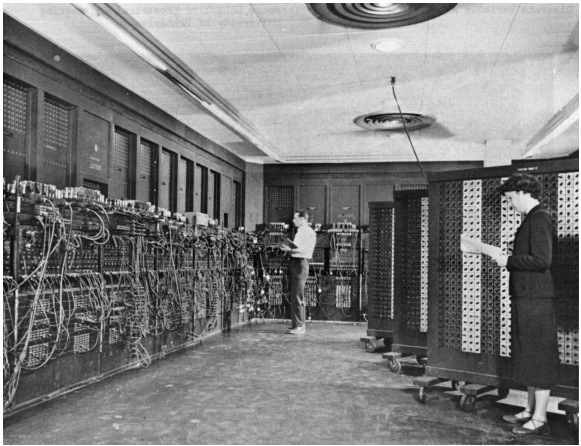


Figure 5: The ENIAC computer, ran between 1947 – 1955. ©Public Domain, US Army, 1947.

History of HPC to Python

- High-performance computing has a long history, dating back to the 1940s with the development of the ENIAC computer.
  - Problem fields: ballistics, weather forecasting, nuclear physics, fluid dynamics, etc.
  - All using linear algebraic discrete methods to solve partial differential equations: floating point approximations of continuous mathematics.
- Original 'programming' done by directly manipulating the hardware, setting registers, etc.
  - First programming language using instructions: Grace Hopper's 1952 A-0 System.

History of HPC to Python

Figure 6: Cray Y-MP Supercomputer, ran between 1988 – 1990 at NASA Center for Computational Sciences. ©Dave Pape

History of HPC to Python

- The development of high-level programming languages in the 1950s and 1960s allowed for more complex programs to be written.
  - **FORTRAN**, developed by IBM in 1957, was the first high-level programming language.
  - followed by **C** (and later, **C++**.)
- With these, the workflow of scientific computing changed:
  - Write code and compile it for the target system
  - Run the code on the computer using the shell or a batch scheduler
  - Analyze the results using a separate program
- These languages are still used today, especially where performance is critical:
  - Still the basis for scientific computing libraries: eg. **NumPy** (C), **SciPy** (Fortran,C++)

History of HPC to Python

Figure 7: The Borg cluster, a 52-node system run by the McGill University pulsar group. ©McGill University, 2014.

History of HPC to Python

- During the 1990s:
  - Hardware costs decreased and PC (x86) computers become plentiful, leading to the development of commodity clusters.
  - The development of the Message Passing Interface (MPI) standard for parallel computing.
  - GPUs are originally developed to make games run better: use matrix operations to render 3D graphics.
  - The development of low-performance languages for scripting and web development: Perl, PHP, Ruby, and...

Why Python?

- 1991 - Python 0.9.0 is released by Guido van Rossum.
  - Originally for scripting system administration tasks on the Amoeba distributed operating system.
  - Python is a high-level, interpreted, general-purpose programming language.
  - Guido wanted to create a language that was easy to read and write, and that was extensible.
    - "Python is a language that emphasizes readability and simplicity, with a focus on pragmatic use and ease of learning." – Van Rossum
- Python today is explosively popular:
  - 2nd most popular language on GitHub.
  - 1st most popular language for data science.
  - 1st most popular language for machine learning.

Python must be so much better than compiled languages like C, right?

Why Python?

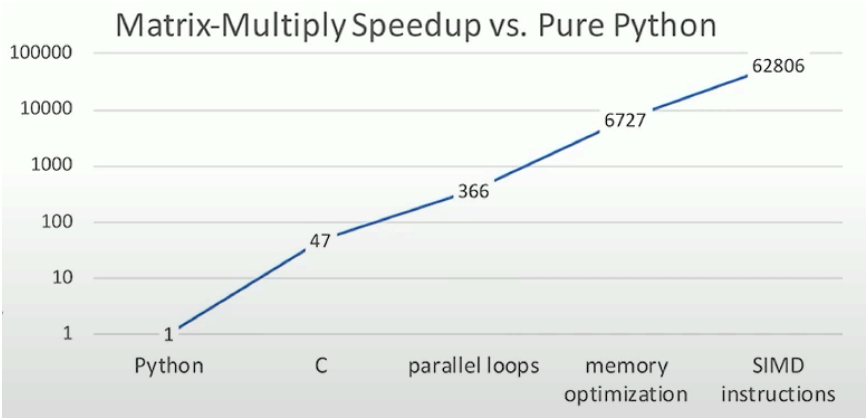


Figure 8: Matrix multiplication in Python vs. C. Source: Emery D. Berger, University of Massachusetts Amherst. *Triangulating Python Performance Issues with SCALENE*. **OSDI '23**.

Why Python?

- Each time Python executes a line of code, it must:
  - **Read** the line of code, parse it, and convert it to bytecode using the Python interpreter.
  - **Eval** the bytecode, which is executed by the Python Virtual Machine (PVM).
  - **Print** the result to the console or return it to the calling function.

Why Python?

- Curiously, Python was not designed for high-performance computing:
  - Pythonic loops are 40x slower than C loops, and 60,000x slower than SIMD parallelism.
  - Global Interpreter Lock (GIL) prevents true parallelism.
  - Python is dynamically typed, which can lead to runtime errors that would be caught at compile time in C.
  - Mystery-meat data structures:
    - C `sizeof(int) == 4` vs. Python `sys.getsizeof(1) == 28`
    - C `sizeof(list<int>) == 24` vs. Python `sys.getsizeof([]) == 56`
    - C `sizeof(map<int,int>) == 24` – Python `sys.getsizeof({}) == 64`

Why Python?

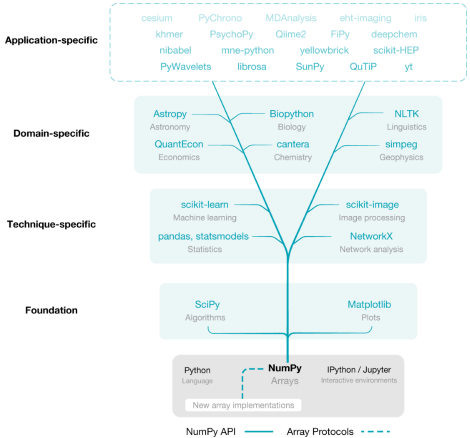


Figure 9: NumPy provides a high-performance multidimensional array object and is the basis for many projects. Source: Harris, C. et al. *Array programming with NumPy*. **Nature** 585, 357-362 (2020).

Why Python?

- Despite these drawbacks, Python is the most popular language for scientific computing.
  - Python is easy to read and write, and has a large standard library.
  - The Python Package Index (PyPI) has over 300,000 packages.
  - Data science and machine learning libraries: NumPy, SciPy, TensorFlow, PyTorch, etc.
    - None of which use Python for performance-critical code.
- Python is a glue language:
  - Python is used to call C, C++, Fortran, CUDA, and other languages
  - Python is used to orchestrate complex workflows.
  - Python is used to prototype and test algorithms.
- A Python developer's real job is to wrap a lower-level languages and generalize them.

Python is not a high performance language, but it *is* a high productivity language.

Python and Package Management

- Historically, Python package management was an afterthought, and was done manually with zip files or tarballs.
  - Guido ultimately wanted a simple and easy-to-use language, and didn't care about package management.
- In 2008, Ian Bicking (who also wrote `venv`) developed the Python Package Index (PyPI) to host Python packages and introduced `pyinstall` which later became `pip`.
- In 2012, Travis Oliphant and others developed the Anaconda distribution, which includes the `conda` package manager.
- So, there are two package managers for Python: `pip` and `conda`, neither of which are:
  - perfect
  - compatible
  - what the creator of Python intended

Using `pip`

- `pip` uses PyPI (https://pypi.org) or a custom index to install packages from source.
- `$ pip --user install package`
  - installs a `package` for the current user (into `$HOME/.local/lib/python3.x/site-packages`).
- `$ pip --user install -r requirements.txt`
- `$ pip install package==version`
- `$ pip install package>=version`

Using `pip`

- Virtual environments are used to isolate Python environments from each other.
- `venv` is the built-in Python virtual environment manager.
  - creates a new Python environment in a directory, and installs the Python interpreter and standard library into that directory.
- Managing virtual environments:
  - `$ python -m venv myenv`
  - `$ source myenv/bin/activate` - activates the environment.
  - `$ deactivate` - deactivates the environment.
- Use `pip` to install packages into the environment, no `--user` flag needed, as the environment is isolated.

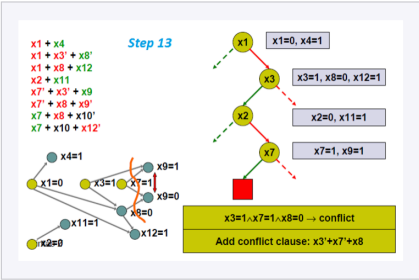
Using `pip`

- Dependencies are not managed:
  - `pip install package` installs `package` and its dependencies
  - but does not manage them
  - or check them against what else is installed.
- `pip` can quickly become a mess of conflicting dependencies.

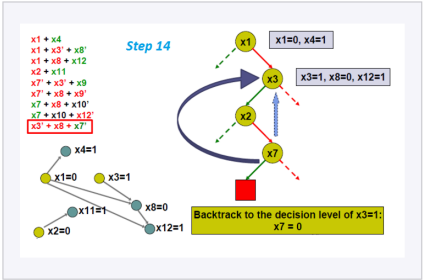
Using conda

- conda is a package manager which uses a dependency solver to manage packages with specifiable channels.
- Each conda environment is a directory that contains a specific collection of packages, and can be activated and deactivated.
  - `$ conda create -n condaenv -c default -c conda-forge python=3.9`
  - `$ conda activate condaenv`
  - `$ conda deactivate`
  - `$ conda install package`

Using conda



Add the conflict clause to the problem.



Non-chronological back jump to appropriate decision level, which in this case is the second highest decision level of the literals in the learned clause.

Figure 10: Graph based dependency solver. ©Tamkin04iut, 2013.

Using conda

- `$ conda env export > environment.yml`
- `$ conda env create -n newenv -f environment.yml`
- `$ conda list`
- conda can be used alongside `pip` to manage packages.
  - `$ conda -n my_tf -c conda-forge -c nvidia python=3.10`
  - `$ conda activate my_tf`
  - `$ pip install tensorflow[and-cuda]`

Using conda

- Caveats:
  - `conda` can be slow to resolve dependencies.
  - `conda` can possibly downgrade packages to resolve dependencies (use `--no-update-deps` if crucial).
  - `conda` comes in several versions: `conda`, `mamba`, `micromamba`, etc. leading to more sharp edges.

Successful Python on HPC

- We have a language that is not designed for high-performance computing.
- We have two package managers that are not perfect.
- We have a high-performance computing environment where performance is critical.

How do we make Python work for HPC?

Successful Python on HPC

- Consider the problem and the data.
- Develop a prototype you can run on small-scale data.
  - Develop a main method which encapsulates the prototype as a function.
  - Develop a configuration: package management, environment variables, etc.
- Use batch scheduler to run on large-scale data:
  - Use the scheduler to schedule the job with the correct environment variables.
  - Run the main method on the cluster.
- Revise the problem into smaller pieces and run in parallel:
  - Turn loops into functions which can be vectorized, turned into a **list comprehension**, or **map-reduced**.
  - Use Dask, joblib, multiprocessing, or mpi4py to run in parallel.
  - Offload mathematics to numpy or CUDA libraries.
- Analyze the results

Final Thoughts – Jupyter Notebooks

- Jupyter notebooks are a great way to:
  - develop and test code.
  - document code.
  - analyze or visualize data.
  - share code that looks great as a portfolio piece.
- Jupyter notebooks are not great for:
  - multi-file projects (use an IDE.)
  - parallel computing (unless you really know what you're doing with Dask/multiprocessing.)
  - managing dependencies (use a virtual environment.)
  - running on a cluster (use a batch scheduler.)
  - running in production (use a script.)

Conclusion

- Most day-to-day computing is done far away from the metal, but we need to get close to the metal to get the most out of a system.
- To get the maximum performance out of a system, we need to understand the underlying system:
  - A typewriter-like interface to the operating system.
- Python wraps lower-level languages and generalizes them, abstracting away the complexity.
  - Complexity is moved from the code to understanding the system.
- Development life-cycle: prototype, run on small-scale data, run on large-scale data in parallel, analyze results.

Live Demo – Python on CCAST

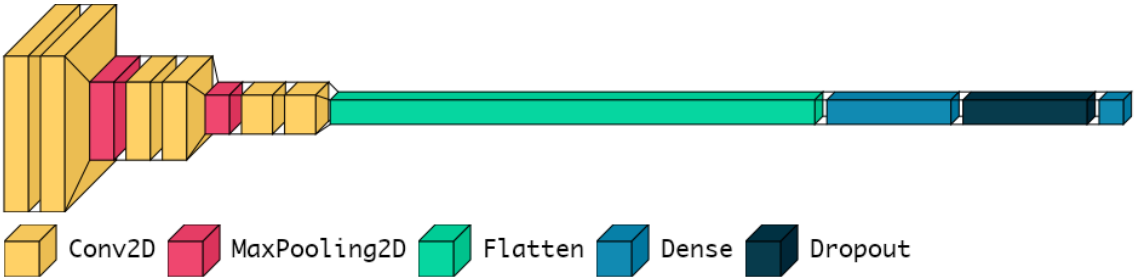


Figure 11: Sequential model of a simple convolutional neural network.

Questions?
Comments?
Concerns?