Python performance workshop

Intro

⚙ Prerequisites

- Python programming basics (functions, `for` loops, `if` - `else` statements) and data structures (`list` , `set` , `dict` , `tuple`)
- Familiarity with well-known numeric libraries (for example, Numpy)

10 min	Installation
10 min	Introduction and motivation
10 min	Performance fundamentals
xx min	Benchmark
xx min	Profile
xx min	Optimize
xx min	Parallelize

Installation

This page contains instructions for installing the required dependencies on a local computer.

Local installation

If you already have a preferred way to manage Python versions and libraries, you can stick to that[1]. If not, we recommend that you install Python3 and all libraries using Miniforge, a free minimal installer for the package, dependency and environment manager conda.

Please follow the installation instructions on https://conda-forge.org/download/ to install Miniforge.

Make sure that both Python and conda are correctly installed:

```
$ python --version
$ # should give something like Python 3.12.5
$ conda --version
$ # should give something like conda 24.7.1
```

With conda (or mamba) installed, install the required dependencies by running:

```
$ conda env create -f https://raw.githubusercontent.com/ENCCS/python-perf/main/content/env/environment.yml
```

This will create a new environment `python-perf` which you need to activate by:

```
$ conda activate python-perf
```

Finally, open Jupyter-Lab in your browser:

```
$ jupyter-lab
```

---

[1] If you are not using conda, to install the right Python dependencies, download the `requirements.txt` file from this link. Then follow this guide to create a virtual environment and activate it. Finally inside the virtual environment run `python3 -m pip install -r requirements.txt`.

Introduction and motivation

Objectives

- Know what to expect from this course
- Build a general, programming-language agnostic notion of performance

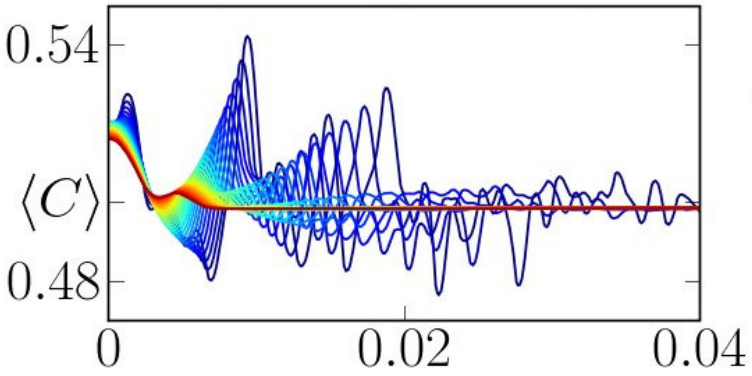
Instructor note

- 10 min teaching

Python and its defacto implementation CPython is now widely used for a spectrum of applications. It has now experienced practitioners doing web-development, analytics, research and data science. This is possible because of the following traits of the Python ecosystem:

- Batteries included
- High-level programming that abstracts away the technical details
- Mature well-maintained libraries which form a firm foundation, the scientific Python ecosystem, which includes:

Core numeric libraries | Advanced IDEs | Domain specific libraries



- Numpy: numerical computing with powerful numerical arrays objects, and routines to manipulate them.
- Scipy: high-level numerical routines. Optimization, regression, interpolation, etc.
- Matplotlib: 2-D visualization, “publication-ready” plots.

and many more…

Extensions: a technical detail hidden in plain sight

A common theme behind the Python standard library and its popular packages is that some parts of the code which are computationally intensive are actually modules or functions which are either:

- **interfaced extensions** with an external implementation in C, C++, Fortran, Rust…
- **source-to-source extensions** written in Python or Python-like code, which is compiled ahead-of-time or just-in-time

Extensions can be imported as normal Python functions or modules. There are many tools which help you in creating extensions:

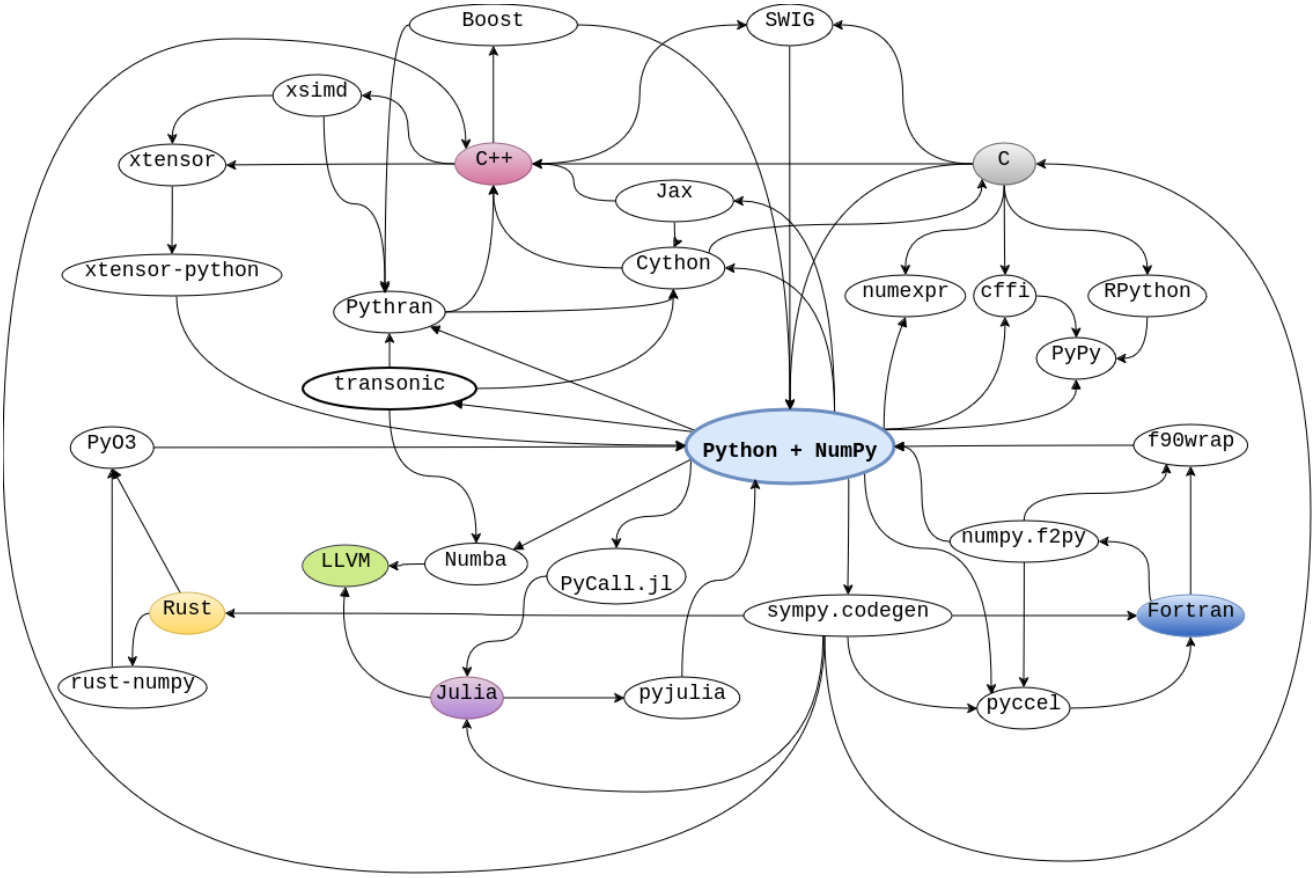


*Figure: The coloured bubbles represent **programming languages**. An outward arrow represents **exporting** a code into an extension using a runtime or a library. An inward arrow represents **importing** an extension (or linking to a code using an API, such as Python's C-API or via a foriegn function interface (FFI). The choices are many!*

💬 Discussion

What are the advantages and disadvantages of using Python code written using multiple programming languages under the hood, in terms of software development and maintenance?

✔ Solution

Pros 👍: This approach enables us to:

- build high-level, performant applications which tend to be readable
- focus on the problem at hand, without getting sidetracked with implementation details
- rapidly prototype the experimental parts of the code
- interfacing allows re-use of established codes

Cons 👎: Some known downsides are:

- interfaced codes require knowledge of multiple languages
- compiled codes are harder to debug and rapidly-prototype

Now we have a notion of how extensions work. We could use extensions to address performance issues. However building an extension is quite often the last resort. More on that will be discussed in the next episode.

Different kinds of performance bottlenecks

- **I/O bound**: the code idles often and is waiting for a disk or network read/write operation to finish. Such bottlenecks can be often remedied by caching, multi-threading or async-programming.
- **Memory bound**: the data to be processed does not fit in the RAM and the code needs to process data in batches instead. This is often a hardware limitation.
- **CPU bound**: the code consumes a lot of CPU cycles, often seen by monitoring the system showing 100% CPU usage in 1 core for serial applications, or in all cores for parallel applications. **This will be the focus of this workshop**.

Gems of wisdom

Before we dive further into the workshop it is important to remember some idioms, which is true in the case of most real-world applications.

Limitations of performance improvement

> The overall performance improvement gained by optimizing a single part of a system is **limited by the fraction of time that the improved part is actually used**.

– *Amdahl's law*[2] (see this demo)

Premature optimization is the root of all evil.

> The real problem is that programmers have spent far too much time **worrying about efficiency in the wrong places and at the wrong times**; *premature optimization is the root of all evil (or at least most of it) in programming*.
>
> Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a **strong negative impact when debugging and maintenance are considered**. We should forget about small efficiencies, say about *97% of the time: premature optimization is the root of all evil*. Yet we should not pass up our opportunities in that critical 3%.

– *Donald Knuth* (computer scientist, mathematician and the author of *The Art of Computer Programming*)

Measure, don't guess.

Pareto principle or the 80/20 rule

> 80 percent of the runtime is spent in 20 percent of the source code.
>
> – *Scott Meyers* (author of *Effective C++ Digital Collection: 140 Ways to Improve Your Programming*)

Keypoints

- Find a balance between *runtime efficency* and *cost of development*.
- Tests can help in maintain correctness before you change the code.
- CPU-bound or I/O-bound or memory bound?
- Do not optimize everything.
- Creating extensions are one way of improving performance

---

[2] Although Amdahl's law is about speedup due to parallelization, we can still associate it with speedup of serial programs. This is because the law is formulated in terms of execution-time.

Performance fundamentals

Objectives

- Learn Python specific performance aspects

Instructor note

- 10 min teaching/type-along

Understanding the Python interpreter

💬 Performance bottlenecks in Python

Have you ever written Python scripts that look something like this?

```
def read_xyz_from_text_file():
    f = open("mydata.dat", "r")
    for line in f.readlines():
        fields = line.split(",")
        x, y, z = fields[0], fields[1], fields[2]
        # some analysis with x, y and z
    f.close()
```

Compared to C/C++/Fortran, this for-loop will probably be orders of magnitude slower!

This happens because during the execution step CPython mostly interprets instructions. There is some level of optimization involved though. Here is a simplified schematic of how this is invoked:

flowchart TD a[Source code in .py files] --> tok[Tokenizer] tok --> ast[Abstract Syntax Tree] ast --> c[Byte-code: __pycache__] c --> d[Machine code in Python Virtual Machine] d --> c

Important

While doing so, the interpreter

- evaluates a result, expression-by-expression.
- every intermediate result is packed and unpacked as an instance of `object` (Python) / `PyObject` (CPython API) behind the scenes.

Type-Along

Try out the following code in Python Tutor

```
import io

def rms_from_text_file(f):
    """Compute root-mean-square of comma-separated values."""
    rms = 0
    n_samples = 0

    for line in f.readlines():
        fields = line.split(",")
        x, y, z = float(fields[0]), float(fields[1]), float(fields[2])
        # compute root-mean-square value
        rms += ((x**2 + y**2 + z**2) / 3) ** 0.5
        n_samples += 1

    return rms / n_samples


fake_file = io.StringIO("""\
0.27194615,0.85939776,0.76905204
0.51586611,0.59174447,0.06501842
0.23109192,0.8260391,0.08045166
""")

avg_rms = rms_from_text_file(fake_file)
```

Be aware that this is a simplified version of the execution. Since it does not go into the expression level. However you can get an idea of the intermediate objects being returned and the way the interpreter parses the code.

💬 Discussion

In the previous episode, we described I/O, Memory and CPU bound bottlenecks. For the above use case and **algorithm**, and **not necessarily the same code**, what kind of performance issue arise,

1. when the file becomes long, with several millions of lines?
2. when the file is stored in network filesystem which is slow to respond?
3. when instead of 3 fields, `x, y, z`, you have to read 10 million fields for every line of the text?

✔ Solution

We can only guess at this point, but we can expect the above code to be

1. **CPU bound**: the `for` loop becomes a *hotspot* and vanilla CPython without JIT does not optimize this.
2. **I/O bound**: if more time is spent in awaiting output of `f.readlines()` method
3. **Memory bound**: if a line of data does not fit in the memory the code needs to handle it in batches. The program will need to be rewritten with nested for-loop which depends on the memory availability.

We have to keep in mind that performance depends a lot on the kind of

- input data
- algorithm

Using a better container for the input data or a better algorithm with less computation complexity can often outperform technical solutions.

Structured approach towards optimization

The first priority is to look for an more efficient:

1. Data container, data structure, database etc.
2. Algorithm

If the above are not an option, then we move on to performance optimization.

1. First we evaluate the overall performance by **benchmarking**.
2. Then we measure the performance of at either function/method-level or line-level by **profiling**.
3. Finally we generate optimized code.

Any Python code can be replaced using optimized instructions. This is done by ahead of time (AOT) / just-in-time (JIT) compilation. The question which remains to be answered is at which level? One can optimize:

 : whole programs (Nuitka, Shed Skin)

 : interpreter compiling slowest loops (PyPy)

 : **modules (`cython`, `pythran`)**

 : **user-defined functions / methods (`numba`, `transonic`)**

 : **expressions (`numexpr`)**

 : call compiled functions (`numpy` / Python)

We will take a look at some of these approaches in the coming episodes.

Keypoints

- Develop a strategy on **how** to optimize.
- Go shopping.
  - Look for better ways of reading data or better algorithms
  - Look for tools and libraries to help you alleviate the performance bottlenecks.

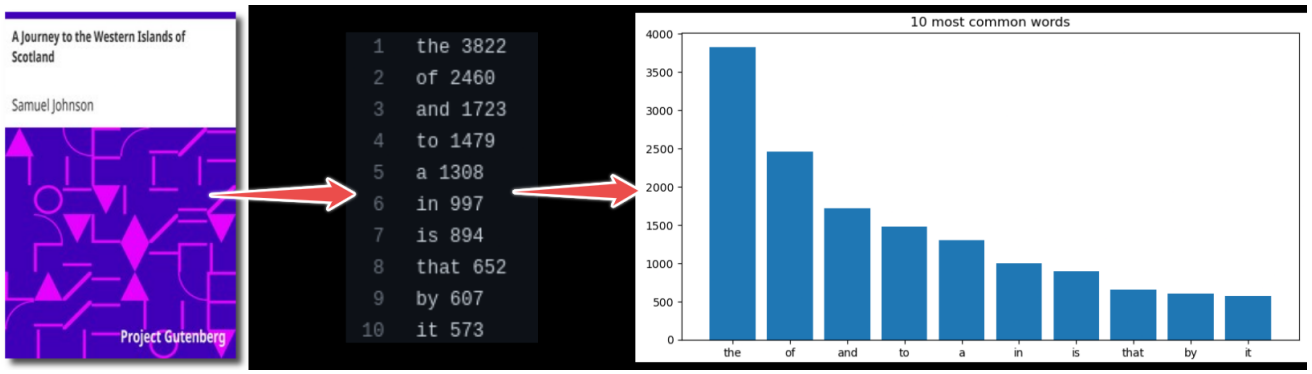
Benchmark

Objectives

- Introduce the
- Preparing the system for benchmarking
- Running benchmarks

Instructor note

The problem: word-count-hpda



In this episode, we will use an example project which finds most frequent words in books and plots the result from those statistics. The project contains a script `source/wordcount.py` which is executed to analyze word frequencies from some books. The books are saved in plain-text format in the data directory.

For example to run this code for one book, *pg99.txt*

```
$ git clone https://github.com/ENCCS/word-count-hpda.git
$ cd word-count-hpda
$ python source/wordcount.py data/pg99.txt processed_data/pg99.dat
$ python source/plotcount.py processed_data/pg99.dat results/pg99.png
```

Preparation: Use `pyperf` to tune your system

Most personal laptops would be running in a power-saver / balanced power management mode. This would include that the system has a scaling governor which can change the CPU clock frequency on demand, among other things. This can cause **jitter** which means that benchmarks are not reproducible enough and are less reliable.

In order to improve reliability of your benchmarks consider running the following

Warning

It requires admin / root privileges.

```
# python -m pyperf system tune
```

When you are done with the lesson, you can run `python -m pyperf system reset` or restart the computer to go back to your default CPU settings.

See also

- https://pyperf.readthedocs.io/en/latest/system.html#operations-and-checks-of-the-pyperf-system-command
- https://pyperf.readthedocs.io/en/latest/run_benchmark.html#how-to-get-reproducible-benchmark-results
- https://pyperformance.readthedocs.io/usage.html#how-to-get-stable-benchmarks

Benchmark using `time`

In order to observe the cost of computation, we need to choose a sufficiently large input data file and time the computation. We can do that by concatenating all the books into a single input file approximately 45 MB in size.

⌨ Type-Along

IPython / Jupyter | Unix Shell

Copy the following script.

```
import fileinput
from pathlib import Path

files = Path("data").glob("pg*.txt")
file_concat = Path("data", "concat.txt")

with (
    fileinput.input(files) as file_in,
    file_concat.open("w") as file_out
):
    for line in file_in:
        file_out.write(line)
```

Open an IPython console or Jupyterlab, with `word-count-hpda` as the current working directory (you can also use `%cd` inside IPython to change the directory).

```
%paste

%ls -lh data/concat.txt

import sys
sys.path.insert(0, "source")

import wordcount

%time wordcount.word_count("data/concat.txt", "processed_data/concat.dat", 1)
```

✔ Solution

IPython / Jupyter | Unix Shell

```
In [1]: %paste
import fileinput
from pathlib import Path

files = Path("data").glob("pg*.txt")
file_concat = Path("data", "concat.txt")

with (
    fileinput.input(files) as file_in,
    file_concat.open("w") as file_out
):
    for line in file_in:
        file_out.write(line)
## -- End pasted text --

In [2]: %ls -lh data/concat.txt
-rw-rw-r-- 1 ashwinmo ashwinmo 45M sep 24 14:54 data/concat.txt

In [3]: import sys
   ...: sys.path.insert(0, "source")

In [4]: import wordcount

In [5]: %time wordcount.word_count("data/concat.txt", "processed_data/concat.dat",
1)
CPU times: user 2.64 s, sys: 146 ms, total: 2.79 s
Wall time: 2.8 s
```

Note

What are the implications of this small benchmark test?

It takes a few seconds to analyze a 45 MB file. Imagine that you are working in a library and you are tasked with running this on several terabytes of data.

- 10 TB = 10 000 000 MB
- Current processing speed = 45 MB / 2.8 s ~ 16 MB/s
- Estimated time = 10 000 000 / 16 = 625 000 s = 7.2 days

Then the same script would take days to complete!

Benchmark using `timeit`

If you run the `%time` magic / `time` command again, you will notice that the results vary a bit. To get a **reliable** answer we should repeat the benchmark several times using `timeit`. `timeit` is part of the Python standard library and it can be imported in a Python script or used via a command-line interface.

If you're using IPython / Jupyter notebook, the best choice will be to use the `%timeit` magic.

As an example, here we benchmark the Numpy array:

```
import numpy as np

a = np.arange(1000)

%timeit a ** 2
# 1.4 µs ± 25.1 ns per loop
```

We could do the same for the `word_count` function.

IPython / Jupyter | Unix Shell

```
In [6]: %timeit wordcount.word_count("data/concat.txt", "processed_data/concat.dat", 1)
# 2.81 s ± 12.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Notice that the output reports the **arithmetic mean and standard deviation** of timings. This is a good choice, since it means that **outliers and temporary spikes in results are not automatically removed**, which could be as a result of:

- garbage collection
- JIT compilation
- CPU or memory resource limitations

Keypoints

- `pyperf` can be used to tune the system
- We understood the use of `time` and `timeit` to create benchmarks
- `time` is faster, since it is executed only once
- `timeit` is more reliable, since it collects statistics

Profile

Objectives

Instructor note

Optimize

Objectives

- Optimize the most expensive function from the word-count-hpda project's `wordcount.py` script.
- Show how changes to algorithm influences the performance.
- Introduce a few Python **accelerators**: `cython`, `numba`, `pythran`
- Mention the library `transonic`

Instructor note

Targeting the most expensive function

In the previous episode by profiling, we found out that `update_word_counts` consumes around half of the CPU wall time and is called repeatedly. Here is a snippet from profiling output.

```
...
         53473208 function calls in 8.410 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  1233410    4.151    0.000    7.204    0.000
source/wordcount.py:41(update_word_counts)
  ...
```

Option 1: changing the algorithm

If we look at the output from the line profiler, we can see that the following two lines are the most time-consuming.

```
def update_word_counts(line, counts):
    """
    Given a string, parse the string and update a dictionary of word
    counts (mapping words to counts of their frequencies). DELIMITERS are
    removed before the string is parsed. The function is case-insensitive
    and words in the dictionary are in lower-case.
    """
    for purge in DELIMITERS:
        line = line.replace(purge, " ")
    words = line.split()
    for word in words:
        word = word.lower().strip()
        if word in counts:
            counts[word] += 1
        else:
            counts[word] = 1
```

👀 Demo

Instead of a `for` loop and a `str.replace` we could use a single regular expression substitution. This change would look like this

```
import re
# WARNING: there is a bug in the regular expression below!
DELIMITERS = re.compile(r"[\.,;:?$@^<>#%`!\*-=\(\)\[\]\{\}/\\]")


def update_word_counts(line, counts):
    """
    Given a string, parse the string and update a dictionary of word
    counts (mapping words to counts of their frequencies). DELIMITERS are
    removed before the string is parsed. The function is case-insensitive
    and words in the dictionary are in lower-case.
    """
    line = DELIMITERS.sub(" ", line)
    words = line.split()
    for word in words:
        word = word.lower().strip()
        if word in counts:
            counts[word] += 1
        else:
            counts[word] = 1
```

If we run our benchmark with the original code (`v0.py`) and the regex version (`v0_1.py`), we get

```
$ time python v0.py data/concat.txt processed_data/concat.dat

real    0m2,934s
user    0m2,733s
sys     0m0,191s
$ time python v0_1.py data/concat.txt processed_data/concat.dat

real    0m2,472s
user    0m2,320s
sys     0m0,147s
```

💬 Summary

- There is a marginal gain of ~0.5 s which amounts to a 16% performance boost.
- Such changes are less maintainable, but sometime necessary.

Option 2: using an accelerator

Refactoring

One complication with optimizing `update_word_counts` is that it is an impure function. In other words, it has some side-effects since it:

1. accesses a global variable `DELIMITERS`, and
2. mutates an external dictionary `counts` which is a local variable inside the function `calculate_word_counts`.

Thus the function `update_word_counts` on its own can be complicated for an accelerator to compile since the types of the external variables are unknown.

```
def update_word_counts(line, counts):
    """
    Given a string, parse the string and update a dictionary of word
    counts (mapping words to counts of their frequencies). DELIMITERS are
    removed before the string is parsed. The function is case-insensitive
    and words in the dictionary are in lower-case.
    """
    for purge in DELIMITERS:
        line = line.replace(purge, " ")
    words = line.split()
    for word in words:
        word = word.lower().strip()
        if word in counts:
            counts[word] += 1
        else:
            counts[word] = 1
```

```
DELIMITERS = ". , ; : ? $ @ ^ < > # % ` ! * - = ( ) [ ] { } / \" '".split()
```

```
def calculate_word_counts(lines):
    """
    Given a list of strings, parse each string and create a dictionary of
    word counts (mapping words to counts of their frequencies). DELIMITERS
    are removed before the string is parsed. The function is
    case-insensitive and words in the dictionary are in lower-case.
    """
    counts = {}
    for line in lines:
        update_word_counts(line, counts)
    return counts
```

Accelerators

The following are the few well-known accelerators for Python-Numpy applications.

Accelerator	Compiles	Implemented in	Level	Supports	Advantage
Cython	Ahead of time	C	Module	All of Python, Numpy, and C	Generic and can also interface C,C++
Pythran	Ahead of time	C++	Module	Most Python and Numpy features	Escapes GIL always, can optimize vectorized code without loops. Can parallelize using OpenMP.
Numba	Just in time	LLVM	Function	Most Python and Numpy features	Specializes in Numeric codes. Has GPU support, can parallelize
Jax	Just in time	C++	Function or Expression	Most Python and Numpy features	Drop-in alternative for Numpy. Designed for creating ML libraries
Cupy	Pre-compiled / JIT	Cython / C / C++	Function or Expression	Numpy and Scipy	Drop-in alternative for Numpy. Supports CUDA and ROCm GPUs

In this example we shall demonstrate **Cython** via a package called **Transonic** . Transonic lets you switch between Cython, Numba, Pythran and to some extent Jax using very similar syntax.

Cython

To use Transonic we add decorators to functions we need to optimize. There are two decorators

- `@transonic.boost` to create ahead-of-time (AOT) compiled modules and it requires type annotations
- `@transonic.jit` to create just-in-time (JIT) compiled modules where type is inferred on runtime

The advantage of using transonic is that you can quickly find out which accelerator works best while preserving the Python code for debugging and future development. It also abstracts away the syntax variations that Cython, Pythran etc. have.

The accelerator backend can be chosen in 3 ways:

1. Using an environment variable, `export TRANSONIC_BACKEND=cython`
2. As a parameter to the decorator, `@boost(backend="cython")`
3. As a parameter to the Transonic CLI, `transonic -b cython /path/to/file.py`

We shall use the `@boost` decorator and the environment variable `TRANSONIC_BACKEND` for simplicity

👀 Demo

We make a few changes to the code:

- Pull `DELIMITERS` inside `update_word_counts` function
- Add `@boost` decorators
- Add type annotations as required by transonic.

Cython has an ability to create *inline functions* and this is also supported in Transonic. Therefore it is OK that `update_word_counts` is impure.

```python
from transonic import boost
from transonic.typing import List, Dict

@boost(inline=True)
def update_word_counts(line: str, counts: Dict[str, int]):
    """
    Given a string, parse the string and update a dictionary of word
    counts (mapping words to counts of their frequencies). DELIMITERS are
    removed before the string is parsed. The function is case-insensitive
    and words in the dictionary are in lower-case.
    """
    DELIMITERS = ". , ; : ? $ @ ^ < > # % ` ! * - = ( ) [ ] { } / \" '".split()

    for purge in DELIMITERS:
        line = line.replace(purge, " ")

    words = line.split()
    for word in words:
        word = word.lower().strip()
        if word in counts:
            counts[word] += 1
        else:
            counts[word] = 1


@boost
def calculate_word_counts(lines: List[str]):
    """
    Given a list of strings, parse each string and create a dictionary of
    word counts (mapping words to counts of their frequencies). DELIMITERS
    are removed before the string is parsed. The function is
    case-insensitive and words in the dictionary are in lower-case.
    """
    counts = {}

    for line in lines:
        update_word_counts(line, counts)

    return counts
```

Then compile the file ./wordcount/v1_1.py

```
$ export TRANSONIC_BACKEND=cython
$ transonic v1_1.py
...
1 files created or updated needs to be cythonized
$ ls -1 __cython__/
build
v1_1_ee8b793c43119b782190c854a1eb2ba7.cpython-312-x86_64-linux-gnu.so
v1_1.pxd
v1_1.py
```

This would auto-generate a module containing only the functions to be optimized and also compiles it. While running the application, Transonic takes care of swapping the Python function with the compiled counterpart.

We are ready to benchmark this.

```
$ time python v1_1.py data/concat.txt processed_data/concat.dat

real    0m4,071s
user    0m4,373s
sys     0m0,288s
```

💬 Summary

We see that the compiled function made the script slower! This could happen because of a few reasons

- Python's dictionary which uses hash-maps, is quite optimized and it is hard to beat it
- Cython interacts with Python a lot. This can be analyzed by running `cd __cython__; cythonize --annotate v1_1.py` which generates the following HTML page.

```
Generated by Cython 3.0.11

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: v1_1.c

+01: try:
 02:     import cython
 03: except ImportError:
 04:     from transonic_cl import cython
 05:
 06:
+07: def update_word_counts(line, counts):
 08:     """
 09:     Given a string, parse the string and update a dictionary of word
 10:     counts (mapping words to counts of their frequencies). DELIMITERS are
 11:     removed before the string is parsed. The function is case-insensitive
 12:     and words in the dictionary are in lower-case.
 13:     """
+14:     DELIMITERS = '. , ; : ? $ @ ^ < > # % ` ! * - = ( ) [ ] { } / " \''.split()
+15:     for purge in DELIMITERS:
+16:         line = line.replace(purge, ' ')
+17:     words = line.split()
+18:     for word in words:
+19:         word = word.lower().strip()
+20:         if word in counts:
+21:             counts[word] += 1
 22:         else:
+23:             counts[word] = 1
 24:
 25:
+26: def calculate_word_counts(lines):
 27:     """
 28:     Given a list of strings, parse each string and create a dictionary of
 29:     word counts (mapping words to counts of their frequencies). DELIMITERS
 30:     are removed before the string is parsed. The function is
 31:     case-insensitive and words in the dictionary are in lower-case.
 32:     """
+33:     counts = {}
+34:     for line in lines:
+35:         update_word_counts(line, counts)
+36:     return counts
 37:
 38:
+39: def __transonic__(): return "0.7.2"
```

- Pythran can be used to escape interaction the GIL, but it has a similar performance. Source code: ./wordcount/v1_2.py and ./wordcount/v1_2_pythran.py
- This is a very poor example, but when it involves contiguous data structures such as lists or arrays of numbers these accelerators can give amazing performance boosts. See here for a related example https://enccs.github.io/hpda-python/performance-boosting/

Keypoints

- Algortihmic optimizations are often better
- Accelerators work well with contiguous data structures

Parallelize

Objectives

Instructor note

Quick Reference

Instructor's guide

Why we teach this lesson

To deep dive in

Intended learning outcomes

By the end of a workshop covering this lesson, learners should be able to:

- gain a better understanding of common performance bottlenecks in Python
- profile, to measure the speed of distinct parts of the code
- benchmark, to quantify the overall performance of their code
- optimize, to alleviate bottlenecks
- parallelize, to scale up and reduce time to solution

Timing

Preparing exercises

e.g. what to do the day before to set up common repositories.

- Have a working Python installation.

Other practical aspects

Interesting questions you might get

Typical pitfalls

Who is the course for?

Software developers, researchers, students who use Python often and process a lot of data.

About the course

See also

Credits

The lesson is inspired and derived from the following:

Creative Commons CC-BY 4.0 licensed material

- https://github.com/ENCCS/hpda-python
- https://github.com/ENCCS/word-count-hpda
- https://github.com/coderefinery/word-count
- https://coderefinery.github.io/reproducible-research
- https://hpc-carpentry.github.io/hpc-python/
- Images and description by authors of lectures.scientific-python.org and by authors of deep-learning-intro
- PyCon Sweden 2019 talk on https://talks.fluid.quest/

Other open-source licenced material

- Images by authors of Project Jupyter is licensed under BSD 3-Clause "New" or "Revised" License
- Images from The Noun Project is licensed under CC-BY 3.0