

Python performance workshop

This mini-workshop is intended for Python developers who are interested in writing code with better performance and potentially running it on supercomputers.

⚙️ 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)

5 min	Installation
5 min	Introduction and motivation
10 min	Performance fundamentals
15 min	Benchmark
15 min	Profile
5-10 min	Break
15 min	Optimize
5 min	Parallelize

Installation

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

Instructor note

- 5 min to initialize the setup and `conda` / `pip` command
- Let `conda env create` / `pip install` run in the background

Local installation

If you already have a preferred way to manage Python versions and libraries, you can stick to that¹. 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

- Inform the format of this talk
 - Mix of talk, type-along and demos

- 5 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:

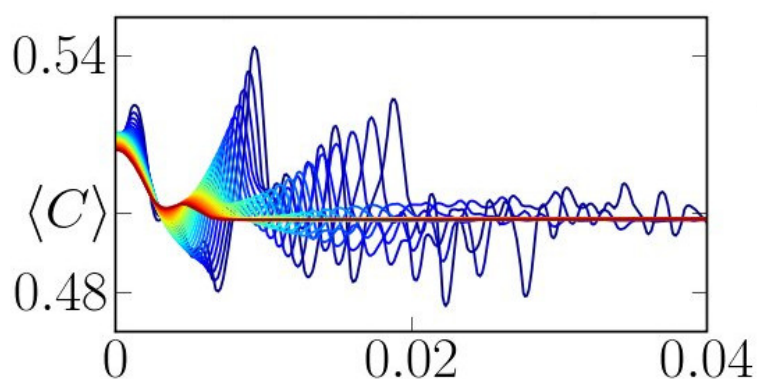
- Batteries included
- High-level programming that abstracts away the technical details
- Mature well-maintained libraries which form a firm foundation, the scientific Python ecosystem, which includes:

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:

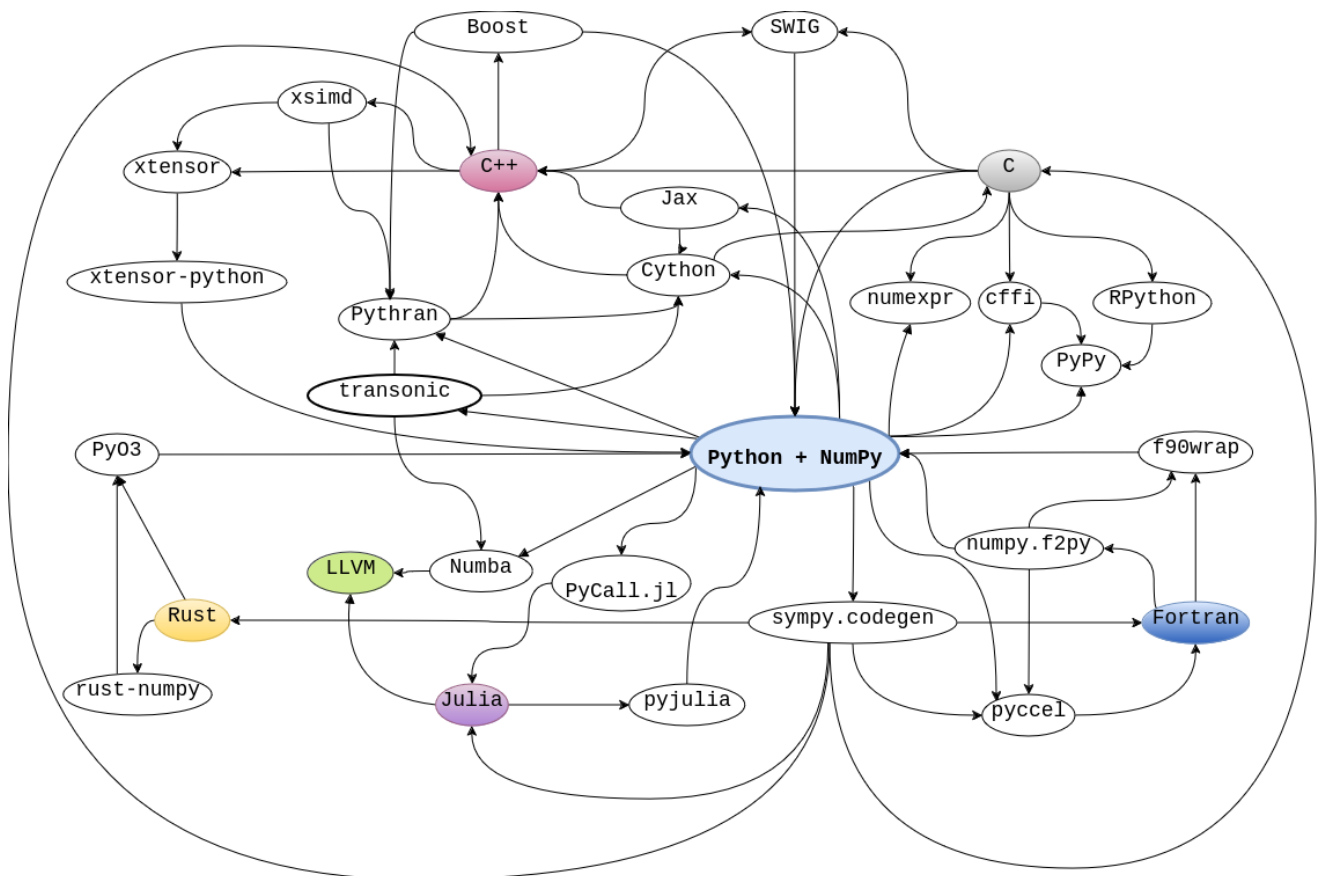


Figure: The coloured bubbles represent **programming languages**. An outward arrow represents **exporting** a code into an extension using a runtime or a library. An inward arrow represents **importing** an extension (or linking to a code using an API, such as Python's C-API or via a foreign function 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 law² (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 efficiency* 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

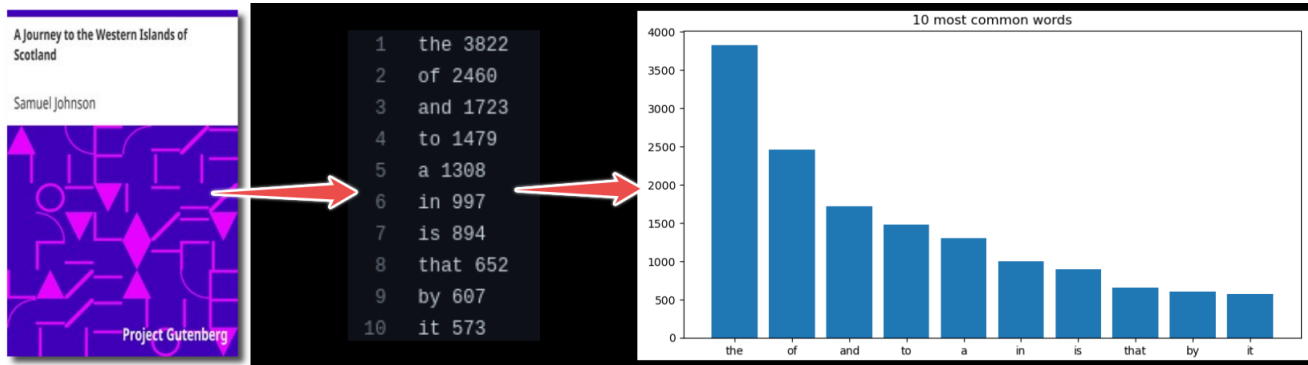
! Objectives

- Introduce the example problem
- Preparing the system for benchmarking using `pyperf`
- Learn how to run benchmarks using `time` , `timeit` and `pyperf timeit`

Instructor note

- 15 min teaching/type-along

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

```
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

- Learn how to profile Python code using `cProfile`
- Learn how to visualise cProfile results using `SnakeViz`
- Examine the most most expensive function call via `line_profiler`

Instructor note

- 15 min teaching/type-along

Using `cProfile` to investigate performance

While `%timeit` can provide good benchmarking information on single lines or single functions, larger codebases have more complex function hierarchies which require more sophisticated tools to traverse properly. Python comes with two [built-in tools](#) to profile code, which implement the same interface: `cProfile` and `profile`. These tools can help to identify performance bottlenecks in the code.

In this lesson, we will use `cProfile` due to its smaller overhead (`profile`, on the other hand, is more extensible). The standard syntax to call it is:

```
$ python -m cProfile [-o <outputFile>] <python_module>
```

By default, `cProfile` writes the results to `stdout`, but the optional `-o` flag redirects the output to file instead. A report can be generated using the `pstats` command.

Type-Along

Let's profile the `wordcount` script and write the results to a file.

Warning

Use the shell variant. The profiling output from Jupyter, although it seems to work, is hard to decipher.

IPython / Jupyter

Unix Shell

The `%run` magic supports profiling out-of-the-box using the `-p` flag. The script can be run as:

```
In [1]: %run -p -D wordcount.prof source/wordcount.py data/concat.txt
processed_data/concat.dat
```

```
*** Profile stats marshalled to file 'wordcount.prof'.
```

Discussion

Profiling introduces a non-negligible overhead on the code being executed. Thus, the absolute values for time being spent in each function should be taken with a grain of salt. The real objective lies in understanding the *relative* amount of time spent in each function call.

Using SnakeViz to visualise performance reports

SnakeViz is a browser-based visualiser of performance reports generated by `cProfile`. It is already included among the dependencies installed in this virtual/Conda environment.

☰ Type-Along

IPython / Jupyter

Unix Shell

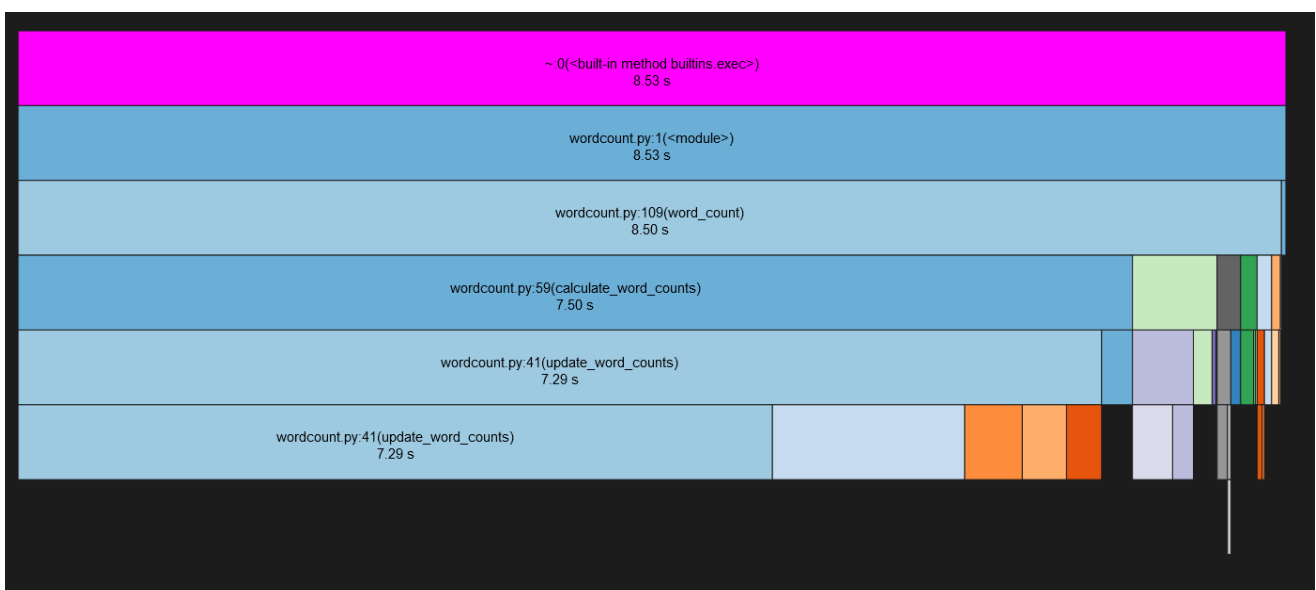
SnakeViz has a IPython magic to profile and open a browser directly. To use it, we just need to load the relevant extension and run it:

```
In [4]: %load_ext snakeviz
In [5]: %snakeviz wordcount.word_count("data/concat.txt",
"processed_data/concat.dat", 1)
```

ⓘ Warning

This will run only if the source IPython instance has access to a local web browser. This also means that, e.g., if you are on Windows and following the tutorial in WSL, this will `not` work.

The output will contain a clickable link containing the visualisation.



Based on the output, we can clearly see that the `update_word_counts()` function is where most of the runtime of the script is spent.

Using `line_profiler` to inspect the expensive function

Once the main performance-intensive function is identified, we can further examine it to find bottlenecks. This can be done using the `line_profiler` tool, which returns a line-by-line breakdown of where time is spent.

≡ Type-Along

Let's profile the `wordcount` script and write the results to a file.

IPython / Jupyter

Unix Shell

The `line_profiler` package provides a magic to be used in IPython. First, the magic needs to be loaded:

```
In [1]: %load_ext line_profiler
```

The script can be run with the `%lprun` magic, whose syntax is very close to the `%run` introduced above. Notice that we have to explicitly mention which functions we want to step through line by line:

```
In [5]: %lprun -f wordcount.update_word_counts
wordcount.word_count("data/concat.txt", "processed_data/concat.dat", 1)
```


Wrote profile results to wordcount.py.lprof

Timer unit: 1e-06 s

Total time: 12.2802 s

File: source/wordcount.py

Function: update_word_counts at line 40

Line #	Hits	Time	Per Hit	% Time	Line Contents
40					@profile
41					def update_word_counts(line,
counts):					
42					"""
43					Given a string, parse the
string and update a dictionary of word					
44					counts (mapping words to
counts of their frequencies). DELIMITERS are					
45					removed before the string is
parsed. The function is case-insensitive					
46					and words in the dictionary
are in lower-case.					"""
47					
48	33302070	2574252.9	0.1	21.0	for purge in DELIMITERS:
49	32068660	4405499.9	0.1	35.9	line = line.replace(purge,
" ")					
50	1233410	392268.8	0.3	3.2	words = line.split()
51	8980773	819407.4	0.1	6.7	for word in words:
52	7747363	1457841.0	0.2	11.9	word =
word.lower().strip()					
53	7747363	1355462.5	0.2	11.0	if word in counts:
54	7364810	1211000.7	0.2	9.9	counts[word] += 1
55					else:
56	382553	64505.7	0.2	0.5	counts[word] = 1

Based on the output, we can conclude that most of the time is spent replacing delimiters.

Keypoints

- The `cProfile` module can provide information on how costly each function call is.
- Profile reports can be inspected using the `pstats` tool in tabular form or with SnakeViz for a graphical visualisation
- The `line_profiler` tool can be used to inspect line-by-line performance overhead.

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

- 15 min teaching/demo
- No type-along intended

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

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

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
```

Cython

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

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.

```

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 -l __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](#)

When do we use accelerators?

An example: Astrophysics N-body problem

To simulate an **N-body problem** using a naive algorithm involves $\mathcal{O}(N^2)$ operations for each time-step. .



Naive Python version

It uses a list of Numpy arrays!

```

import sys
import numpy as np
from itertools import combinations
from time import perf_counter
from datetime import timedelta

class Particle:
    """
    A Particle has mass, position, velocity and acceleration.
    """

    def __init__(self, mass, x, y, z, vx, vy, vz):
        self.mass = mass
        self.position = np.array([x, y, z])
        self.velocity = np.array([vx, vy, vz])
        self.acceleration = np.array([[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]])

    @property
    def ke(self):
        return 0.5 * self.mass * sum(v ** 2 for v in self.velocity)

class Cluster(list):
    """
    A Cluster is just a list of particles with methods to accelerate and
    advance them.
    """

    @property
    def ke(self):
        return sum(particle.ke for particle in self)

    @property
    def energy(self):
        return self.ke + self.pe

    def step(self, dt):
        self.__accelerate()
        self.__advance_positions(dt)
        self.__accelerate()
        self.__advance_velocities(dt)

    def __accelerate(self):
        for particle in self:
            particle.acceleration[1] = particle.acceleration[0]
            particle.acceleration[0] = 0.0
            self.pe = 0.0
        for p1, p2 in combinations(self, 2):
            vector = np.subtract(p1.position, p2.position)
            distance = np.sqrt(np.sum(vector ** 2))
            p1.acceleration[0] = (
                p1.acceleration[0] - (p2.mass / distance ** 3) * vector
            )
            p2.acceleration[0] = (
                p2.acceleration[0] + (p1.mass / distance ** 3) * vector
            )
            self.pe -= (p1.mass * p2.mass) / distance

    def __advance_positions(self, dt):
        for p in self:
            p.position = (
                p.position + p.velocity * dt + 0.5 * dt ** 2 * p.acceleration[0]
            )

```

```

def __advance_velocities(self, dt):
    for p in self:
        p.velocity = (
            p.velocity + 0.5 * (p.acceleration[0] + p.acceleration[1]) * dt
        )

if __name__ == "__main__":
    t_start = perf_counter()
    tend, dt = 10.0, 0.001 # end time, timestep
    cluster = Cluster()
    with open(sys.argv[1]) as input_file:
        for line in input_file:
            # try/except is a blunt instrument to clean up input
            try:
                cluster.append(Particle(*[float(x) for x in line.split()[1:])))
            except:
                pass
    old_energy = -0.25
    for step in range(1, int(tend / dt + 1)):
        cluster.step(dt)
        if not step % 100:
            print(
                f"t = {dt * step:.2f}, E = {cluster.energy:.10f}, "
                f"dE/E = {(cluster.energy - old_energy) / old_energy:.10f}"
            )
            old_energy = cluster.energy
    print(f"Final dE/E = {(cluster.energy + 0.25) / -0.25:.6e}")
    print(f"run in {timedelta(seconds=perf_counter()-t_start)}")

```

Numpy vectorized version

```

from math import sqrt
from time import perf_counter
from datetime import timedelta

import numpy as np
import pandas as pd

def load_input_data(path):
    df = pd.read_csv(
        path, names=["mass", "x", "y", "z", "vx", "vy", "vz"], delimiter=r"\s+"
    )
    # warning: copy() is for Pythran...
    masses = df["mass"].values.copy()
    positions = df.loc[:, ["x", "y", "z"]].values.copy()
    velocities = df.loc[:, ["vx", "vy", "vz"]].values.copy()
    return masses, positions, velocities

def advance_positions(positions, velocities, accelerations, time_step):
    positions += time_step * velocities + 0.5 * time_step ** 2 * accelerations

def advance_velocities(velocities, accelerations, accelerations1, time_step):
    velocities += 0.5 * time_step * (accelerations + accelerations1)

def compute_accelerations(accelerations, masses, positions):
    nb_particules = masses.size
    for index_p0 in range(nb_particules - 1):
        position0 = positions[index_p0]
        mass0 = masses[index_p0]
        for index_p1 in range(index_p0 + 1, nb_particules):
            mass1 = masses[index_p1]
            vector = position0 - positions[index_p1]
            distance = sqrt(sum(vector ** 2))
            coef = 1.0 / distance ** 3
            accelerations[index_p0] -= coef * mass1 * vector
            accelerations[index_p1] += coef * mass0 * vector

def loop(time_step, nb_steps, masses, positions, velocities):
    accelerations = np.zeros_like(positions)
    accelerations1 = np.zeros_like(positions)

    compute_accelerations(accelerations, masses, positions)

    time = 0.0
    energy0, _, _ = compute_energies(masses, positions, velocities)
    energy_previous = energy0

    for step in range(nb_steps):
        advance_positions(positions, velocities, accelerations, time_step)
        # swap acceleration arrays
        accelerations, accelerations1 = accelerations1, accelerations
        accelerations.fill(0)
        compute_accelerations(accelerations, masses, positions)
        advance_velocities(velocities, accelerations, accelerations1, time_step)
        time += time_step

        if not step % 100:
            energy, _, _ = compute_energies(masses, positions, velocities)
            print(
                f"t = {time_step * step:5.2f}, E = {energy:.10f}, "

```

```

        f"dE/E = {(energy - energy_previous) / energy_previous:.10f}"
    )
    energy_previous = energy

    return energy, energy0

def compute_kinetic_energy(masses, velocities):
    return 0.5 * np.sum(masses * np.sum(velocities ** 2, 1))

def compute_potential_energy(masses, positions):
    nb_particules = masses.size
    pe = 0.0
    for index_p0 in range(nb_particules - 1):
        for index_p1 in range(index_p0 + 1, nb_particules):
            mass0 = masses[index_p0]
            mass1 = masses[index_p1]
            vector = positions[index_p0] - positions[index_p1]
            distance = sqrt(sum(vector ** 2))
            pe -= (mass0 * mass1) / distance
    return pe

def compute_energies(masses, positions, velocities):
    energy_kin = compute_kinetic_energy(masses, velocities)
    energy_pot = compute_potential_energy(masses, positions)
    return energy_kin + energy_pot, energy_kin, energy_pot

if __name__ == "__main__":

    import sys

    t_start = perf_counter()
    try:
        time_end = float(sys.argv[2])
    except IndexError:
        time_end = 10.

    time_step = 0.001
    nb_steps = int(time_end / time_step) + 1



    path_input = sys.argv[1]
    masses, positions, velocities = load_input_data(path_input)

    energy, energy0 = loop(time_step, nb_steps, masses, positions, velocities)
    print(f"Final dE/E = {(energy - energy0) / energy0:.6e}")
    print(
        f"{nb_steps} time steps run in {timedelta(seconds=perf_counter()-t_start)}"
    )

```

Demo

Data for 16-bodies:  [./nbabel/input16](#)

- Naive Python version:  [./nbabel/bench0.py](#)
- Numpy vectorized version:  [./nbabel/bench_numpy_highlevel.py](#)

- Numpy vectorized version + JIT compilation using Transonic and Pythran:

📄 [./nbabel/bench_numpy_highlevel_jit.py](#)

```
$ time python bench0.py input16
$ time bench_numpy_highlevel.py input16
$ export TRANSONIC_BACKEND=pythran
$ time bench_numpy_highlevel_jit.py input16 # Rerun after Pythran module is
compiled
```

✓ Solution

```
$ time python bench0.py input16
...
run in 0:00:12.637249

$ time bench_numpy_highlevel.py input16
...
10001 time steps run in 0:00:04.297485

$ export TRANSONIC_BACKEND=pythran
$ time bench_numpy_highlevel_jit.py input16 # Rerun after Pythran module is
compiled
...
10001 time steps run in 0:00:00.042925
```

~300x speedup by using Pythran!

📌 Keypoints

- Algorithmic optimizations are often better
- Accelerators work well with contiguous data structures
- The word-count problem is a poor candidate, but when it involves contiguous data structures such as arrays of numbers these accelerators can give amazing performance boosts. See here:
 - <https://enccs.github.io/hpda-python/performance-boosting/>
 - <https://github.com/paugier/nbabel/>

Parallelize

📌 Objectives

- Learn what approaches exist to parallelize applications
- Showcase Dask

Different ways to do parallelization

- **Data parallelization:**
 - split an array / data into chunks
 - do computation in separate processes / threads
 - combine it
 - **Libraries:** `mpi4py`, `dask`, `cython`, `numba`, `cupy`, `pythran`
- **Task parallelization:**
 - construct a task or a function
 - feed different sets input data
 - collect the results in a queue and write it
 - **Libraries:**
 - Standard library: `multiprocessing`, `threading`, `queue`, `asyncio`, `concurrent.futures`
 - Third party: `trio`, `dask`, `ray`

Irrespective of the approach, one common consideration is that a parallelized part of the code should be free from race-conditions.

In this episode we will showcase parallelizing using Dask.

Dask

Dask is composed of two parts:

- Dynamic task scheduling optimized for computation. Similar to other workflow management systems, but optimized for interactive computational workloads.
- “Big Data” collections like parallel arrays, dataframes, and lists that extend common interfaces like NumPy, Pandas, or Python iterators to larger-than-memory or distributed environments. These parallel collections run on top of dynamic task schedulers.

Collections

(create task graphs)

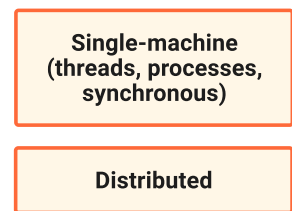
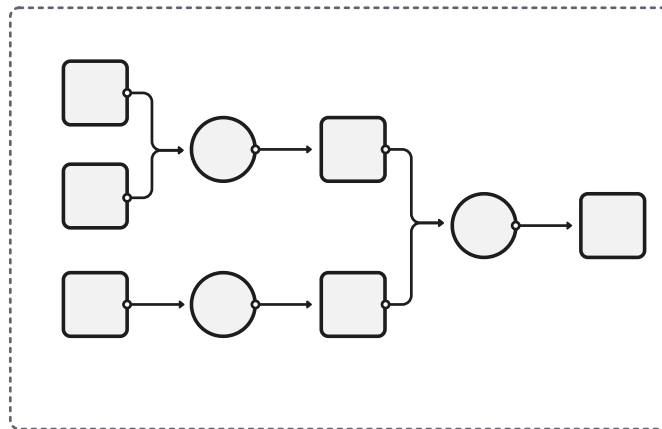
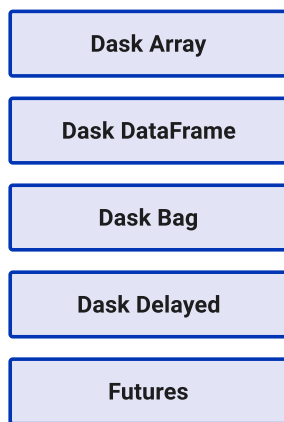


Task Graph



Schedulers

(execute task graphs)



High level collections are used to generate task graphs which can be executed by schedulers on a single machine or a cluster. From the [Dask documentation](#)

Dask distributed and dashboard

Dask has a plugin package known as [distributed](#) which brings in the capability to tap into a variety of computing setups: ranging from local machines to HPC/Supercomputers and Kubernetes clusters. It also has an integrated web application called dashboard to monitor the application.

Dask Bag

A Dask bag enables processing data that can be represented as a sequence of arbitrary inputs ("messy data"), like in a Python list. Dask Bags are often used to for preprocessing log files, JSON records, or other user defined Python objects.

We will content ourselves with implementing a dask version of the word-count problem, specifically the step where we count words in a text.

👁️ Demo: Dask version of word-count

First navigate to the `word-count-hpda` directory. The serial version (wrapped in multiple functions in the `source/wordcount.py` code) looks like this:

```

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

with open(filename, "r") as input_fd:
    lines = input_fd.read().splitlines()

counts = {}
for line in lines:
    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

sorted_counts = sorted(
    list(counts.items()),
    key=lambda key_value: key_value[1],
    reverse=True
)

sorted_counts[:10]

```

A very compact `dask.bag` version of this code is as follows:

```

import dask.bag as db

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

text = db.read_text(filename, blocksize='1MiB')
sorted_counts = (
    text
    .filter(lambda word: word not in DELIMITERS)
    .str.lower()
    .str.strip()
    .str.split()
    .flatten()
    .frequencies().topk(10, key=1)
    .compute()
)

sorted_counts

```

The last two steps of the pipeline could also have been done with a Dask dataframe (which is the Dask equivalent of a Pandas dataframe):

```
text = db.read_text(filename, blocksize='1MiB')
filtered = (
    text
    .filter(lambda word: word not in DELIMITERS)
    .str.lower()
    .str.strip()
    .str.split()
    .flatten()
)
ddf = filtered.to_dataframe(columns=['words'])
ddf['words'].value_counts().compute()[:10]
```

Dashboard

Try adding the following snippet and visualize the run in a dashboard

```
from dask.distributed import Client
client = Client() # start distributed scheduler locally.
client
```

! When to use Dask

There is no benefit from using Dask on small datasets. But imagine we were analysing a very large text file (all tweets in a year? a genome?). Dask provides both parallelisation and the ability to utilize RAM on multiple machines.

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.

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](https://lectures.scientific-python.org) and by [authors of deep-learning-intro](https://deep-learning-intro)
- PyCon Sweden 2019 talk on <https://talks.fluid.quest/>

Other open-source licenced material

- Images by [authors of Project Jupyter](https://projectjupyter.org) is licensed under [BSD 3-Clause “New” or “Revised” License](https://www.bsdlicense.com/)
- Images from [The Noun Project](https://the-noun-project.com) is licensed under [CC-BY 3.0](https://creativecommons.org/licenses/by/3.0/)
- Code from <https://github.com/paugier/nbabel> is licensed under [GPLv2](https://www.gnu.org/licenses/gpl-3.0.html)
- Video from https://en.wikipedia.org/wiki/File:Galaxy_collision.ogg licensed under [CC-BY 3.0](https://creativecommons.org/licenses/by/3.0/)