Profiling & Optimization in Python

Karl Kosack

CEA Paris-Saclay Astrophysics Department

Université de Paris (-Saclay, -Cité)

CTAO

S3 School 2026, Annecy

S3 School

Sustainable Scientific Software School

Context

Context

Where you are now:

- You have some code
- It **works!**
  - ➤ has good test coverage
  - ➤ the output is what you expect

Context

Where you are now:

- You have some code
- It **works!**
  - ➤ has good test coverage
  - ➤ the output is what you expect

The problem:

- You want to run the code many times / on larger datasets / using more complex algorithms...
- **Speed or memory use** is becoming an issue!

Context

Where you are now:

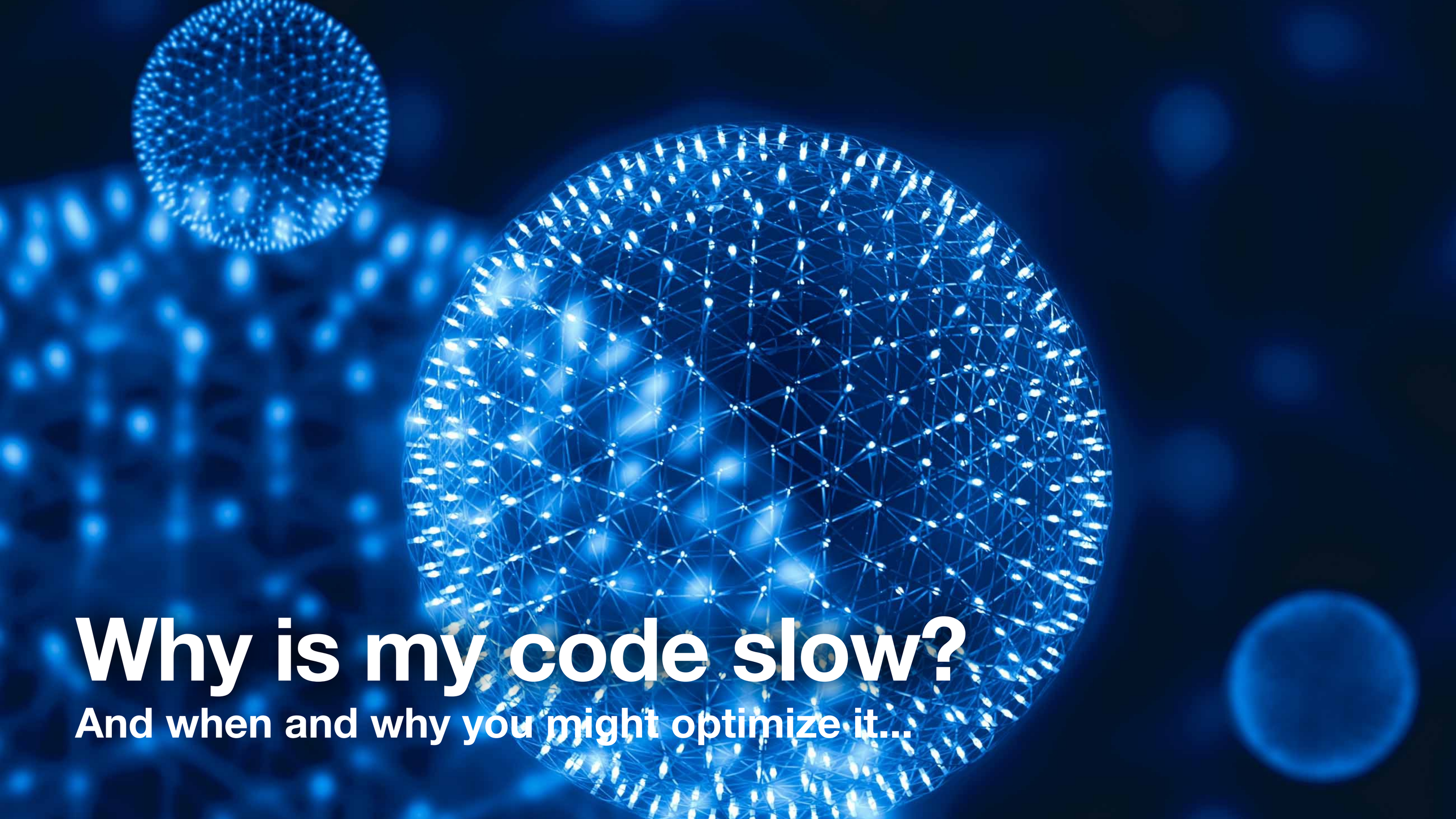
- You have some code
- It **works!**
  - ➤ has good test coverage
  - ➤ the output is what you expect

The problem:

- You want to run the code many times / on larger datasets / using more complex algorithms...
- **Speed or memory use** is becoming an issue!

What you will learn in this lecture:

- What causes performance to be poor?
- How to identify slow parts of your code?
- How to speed up slow code?
- How to identify high memory usage?
- How to fix high memory usage?



Why is my code slow?

And when and why you might optimize it...

Interpreted vs Compiled Code

runs on all code at once

runs per statement

```
def myfunc(x):
    return 2 + x**2
```

```
import dis

dis.dis(myfunc)
```

```
  3           RESUME                   0

  4           LOAD_CONST               1 (2)
              LOAD_FAST                0 (x)
              LOAD_CONST               1 (2)
              BINARY_OP                8 (**)
              BINARY_OP                0 (+)
              RETURN_VALUE
```

You can inspect the python byte code!

* actually, some optimization now happens in python 3.13+, and soon some just-in-time (JIT) compilation [See PEP-744].

Interpreted vs Compiled Code

Compiled: **Code** → [Compiler/Optimizer] → **Machine Code Executable**

runs on all code at once

runs per statement

```
def myfunc(x):
    return 2 + x**2
```

```
import dis

dis.dis(myfunc)
```

```
  3           RESUME                   0

  4           LOAD_CONST               1 (2)
              LOAD_FAST                0 (x)
              LOAD_CONST               1 (2)
              BINARY_OP                8 (**)
              BINARY_OP                0 (+)
              RETURN_VALUE
```

You can inspect the python byte code!

* actually, some optimization now happens in python 3.13+, and soon some just-in-time (JIT) compilation [See PEP-744].

Interpreted vs Compiled Code

Compiled: **Code** → [Compiler/Optimizer] → **Machine Code Executable**
runs on all code at once

Interpreted: **Code** → [Interpreter] → **Machine Code Instruction**
runs per statement

```
def myfunc(x):
    return 2 + x**2
```

```
import dis

dis.dis(myfunc)
```

```
  3           RESUME                   0

  4           LOAD_CONST               1 (2)
              LOAD_FAST                0 (x)
              LOAD_CONST               1 (2)
              BINARY_OP                8 (**)
              BINARY_OP                0 (+)
              RETURN_VALUE
```

You can inspect the python byte code!

* actually, some optimization now happens in python 3.13+, and soon some just-in-time (JIT) compilation [See PEP-744].

Interpreted vs Compiled Code

Compiled: **Code** → [Compiler/Optimizer] → **Machine Code Executable**
runs on all code at once

Interpreted: **Code** → [Interpreter] → **Machine Code Instruction**
runs per statement

```
def myfunc(x):
    return 2 + x**2
```

```
import dis

dis.dis(myfunc)
```

```
  3           RESUME                   0

  4           LOAD_CONST               1 (2)
              LOAD_FAST                0 (x)
              LOAD_CONST               1 (2)
              BINARY_OP                8 (**)
              BINARY_OP                0 (+)
              RETURN_VALUE
```

You can inspect the python byte code!

* actually, some optimization now happens in python 3.13+, and soon some just-in-time (JIT) compilation [See PEP-744].

Interpreted vs Compiled Code

Compiled: **Code** → [Compiler/Optimizer] → **Machine Code Executable**
runs on all code at once

Interpreted: **Code** → [Interpreter] → **Machine Code Instruction**
runs per statement

Python is Interpreted

- python pre-compiles all code before you run it
  - ➤ turns text into **python byte code**, not *machine code*
  - ➤ Byte code is *machine independent instructions*
  - ➤ Each instruction is executed as machine code by the interpreter
- (**Code** → [Compiler] → **Byte Code**) → [Interpreter] → **Machine Code Instruction**
  - ➤ Conceptually similar to Java, but executed per-statement.
  - ➤ Except: No global optimization, no further (JIT) compilation*

```
def myfunc(x):
    return 2 + x**2
```

```
import dis

dis.dis(myfunc)
```

```
  3           RESUME                   0

  4           LOAD_CONST               1 (2)
              LOAD_FAST                0 (x)
              LOAD_CONST               1 (2)
              BINARY_OP                8 (**)
              BINARY_OP                0 (+)
              RETURN_VALUE
```

You can inspect the python byte code!

* actually, some optimization now happens in python 3.13+, and soon some just-in-time (JIT) compilation [See PEP-744].

Interpreted vs Compiled Code

Compiled: **Code** → [Compiler/Optimizer] → **Machine Code Executable**
runs on all code at once

Interpreted: **Code** → [Interpreter] → **Machine Code Instruction**
runs per statement

Python is Interpreted

- python pre-compiles all code before you run it
  - ➤ turns text into **python byte code**, not *machine code*
  - ➤ Byte code is *machine independent instructions*
  - ➤ Each instruction is executed as machine code by the interpreter
- (**Code** → [Compiler] → **Byte Code**) → [Interpreter] → **Machine Code Instruction**
  - ➤ Conceptually similar to Java, but executed per-statement.
  - ➤ Except: No global optimization, no further (JIT) compilation*

Interpreted languages *can* be slow! Particularly loops.

```
def myfunc(x):
    return 2 + x**2
```

```
import dis

dis.dis(myfunc)
```

```
  3           RESUME                   0

  4           LOAD_CONST               1 (2)
              LOAD_FAST                0 (x)
              LOAD_CONST               1 (2)
              BINARY_OP                8 (**)
              BINARY_OP                0 (+)
              RETURN_VALUE
```

You can inspect the python byte code!

* actually, some optimization now happens in python 3.13+, and soon some just-in-time (JIT) compilation [See PEP-744].

Interpreted vs Compiled Code 2

Interpreted vs Compiled Code 2

Executing lots of interpreted statements is much slower than executing many machine-code instructions!

Interpreted vs Compiled Code 2

Executing lots of interpreted statements is much slower than executing many machine-code instructions!

Python is not *completely* interpreted!

- Many libraries contain fully compiled and optimized functions that can be executed as a single byte-code instruction!
  - ➤ numpy/scipy → contains C++, FORTRAN, Cython, …
- There are special libraries for speeding up python code (see later...)

Interpreted vs Compiled Code 2

Executing lots of interpreted statements is much slower than executing many machine-code instructions!

Python is not *completely* interpreted!

- Many libraries contain fully compiled and optimized functions that can be executed as a single byte-code instruction!
  - ➤ numpy/scipy → contains C++, FORTRAN, Cython, …
- There are special libraries for speeding up python code (see later...)

Don't always blame python on slowness!

- Algorithm design has a big impact!
- Bad design can lead to poor performance whether compiled or interpreted!

Loops in Python

Scientific code is full of loops!

- Explicit:

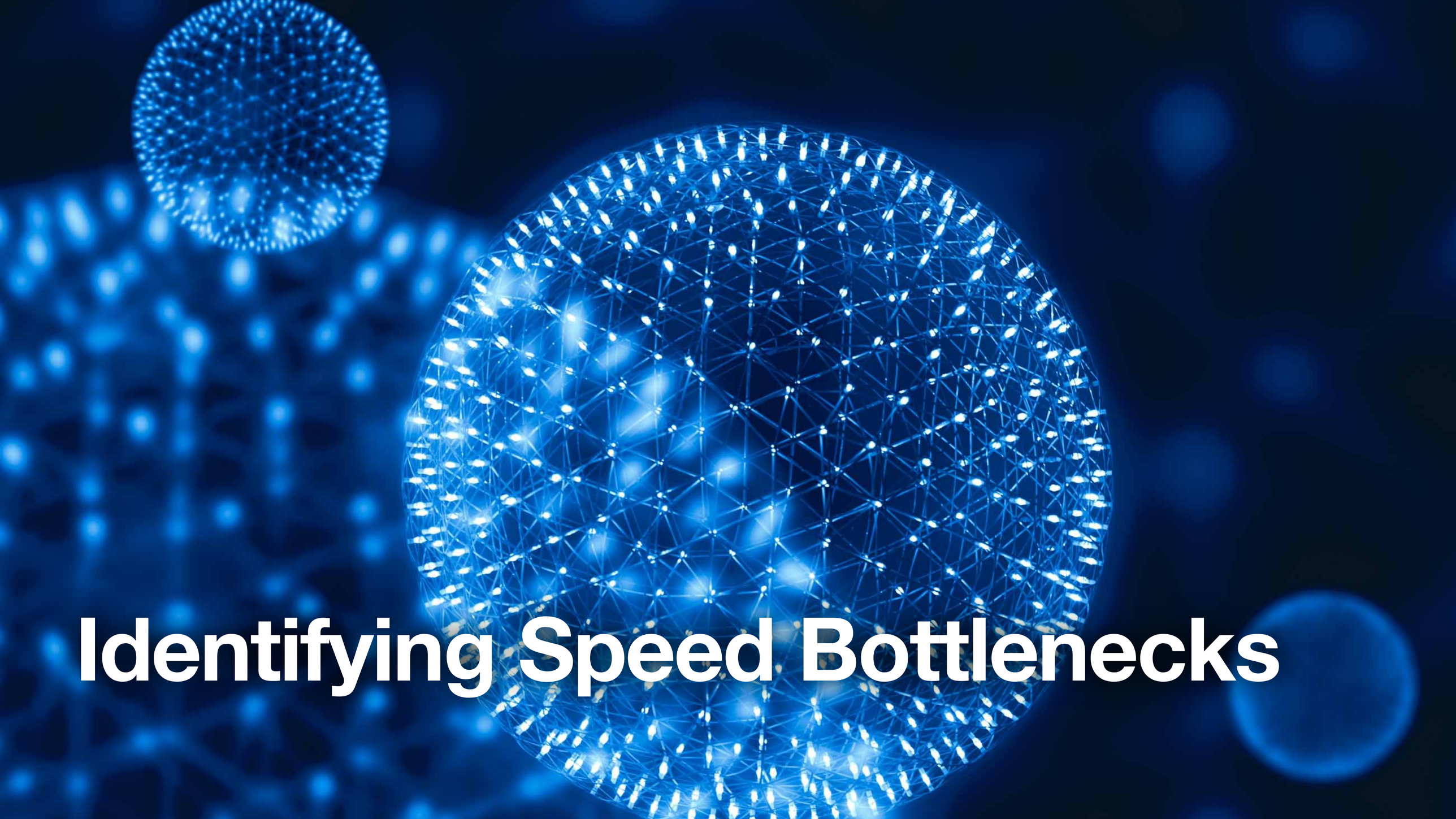
```
for item in some_list:
    do_some_computation(item)

values = [f(x) for x in some_list]
```

- Implicit

```
map(do_some_computation, some_list)
```

Python loops are *100 - 1000x slower* than those in:

- pre-compiled languages like **C, C++, FORTRAN, Rust**
- just-in-time compiled languages like **Julia** or **Java**



Identifying Speed Bottlenecks

What is profiling?

A way to identify where resources are used by a program:

- CPU resources (computation time)
- Memory resources

Identify problems in your code like hangs and *memory leaks*

Identify "hotspots" in your code that may be useful to optimize!

- ➤ always ask your question: *will it make a real difference*?
- ➤ If it's good enough, STOP

Speed: profiling with PyTest

You already saw this in the lecture on testing:

- Show the 3 slowest tests:

```
pytest --durations=3
```

➤ you can set the threshold in seconds

```
--durations-min=0.5
```

- Example, but not so exciting with our current code:

```
pytest --durations 3      (pkoffee) ? ✓  3.13.11
========================================= test session starts ==========================================
collected 39 items

tests/test_data.py ............                                                              [ 30%]
tests/test_fit_model.py ..........                                                          [ 56%]
tests/test_fit_model_io.py ....                                                             [ 66%]
tests/test_metrics.py ........                                                              [ 87%]
tests/test_parametric_function.py .....                                                     [100%]

========================================== slowest 3 durations ==========================================
0.01s call     tests/test_fit_model_io.py::test_save_models
0.01s setup    tests/test_data.py::test_load_csv_valid_file
0.01s call     tests/test_fit_model_io.py::test_save_models_toml
```

Speed profiling: in a *notebook*

What I often see...

```
from time import time

start = time.time()

[code]

stop = time.time()
print(stop - start)
```

*this **measures only wall-clock time**!*

*You want **CPU time**!*
(not dependent on other stuff you are running)

*You want **many trials**, for statistics!*

Better method: *%timeit*

- *interactive* **%timeit** *"magic" jupyter/ipython function*
- *Automatically runs a function many times and measures CPU time and standard deviation*
- *Usage:*

```
%timeit <python statement>
```

Notes:

- ➤ *to time an entire cell, use* ***%%time***
- ➤ *you can also import the* ***`timeit` module***
- ➤ *if you really only want one trial, use* ***%%time***

Speed profiling: in a *notebook*

What I often see...

```
from time import time

start = time.time()

[code]

stop = time.time()
print(stop - start)
```

*this **measures only wall-clock time**!*

*You want **CPU time**!*
(not dependent on other stuff you are running)

*You want **many trials**, for statistics!*

Better method: *%timeit*

- *interactive* **%timeit** *"magic" jupyter/ipython function*
- *Automatically runs a function many times and measures CPU time and* standard deviation
- *Usage:*

```
%timeit <python statement>
```

***Notes**:*

- ➤ *to time an entire cell, use* ***%%time***
- ➤ *you can also import the* ***`timeit` module***
- ➤ *if you really only want one trial, use* ***%%time***

DEMO

Profiling in a notebook

an advanced study:

- see using-timeit.ipynb
- **pixi run "jupyter lab"**

Using a profiler (Notebook version)

A profiler is fancier than %timeit: it measures all function calls

- use the magic %prun function

```
%prun <python statement>
```

- Generates a comprehensive report

```
In [27]: %prun create_array_loop(1000,1000)
```

```
         3001004 function calls in 0.845 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.477    0.477    0.835    0.835 <ipython-input-12-6d84b414c957>:1(create_array_loop)
  1000000    0.136    0.000    0.136    0.000 {built-in method math.cos}
  1000000    0.133    0.000    0.133    0.000 {built-in method math.sin}
  1001000    0.089    0.000    0.089    0.000 {method 'append' of 'list' objects}
        1    0.010    0.010    0.845    0.845 <string>:1(<module>)
        1    0.000    0.000    0.845    0.845 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Line-profiling in a Notebook

Sometimes you need finer grained info: per line, not per function call!

- unlike %timeit, need to load an extension first:

```
%load_ext line_profiler
```

- Then, if you have a function defined, you must "mark" it to be profiled by adding "-f <func>"

```
%lprun -f <function name> <python statement that uses function>
```

for example:

```
%lprun -f myfunc myfunc(100,100)
```

Note **you can mark more than one function to add to the report!**

```
In [51]: %lprun -f create_array_loop create_array_loop(1000,1000)
Timer unit: 1e-06 s

Total time: 1.31799 s
File: <ipython-input-12-6d84b414c957>
Function: create_array_loop at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def create_array_loop(N,M):
     2         1          2.0      2.0      0.0      arr = []
     3      1001        477.0      0.5      0.0      for y in range(M):
     4      1000       5244.0      5.2      0.4          row = []
     5   1001000     463343.0      0.5     35.2          for x in range(N):
     6   1000000     848316.0      0.8     64.4              row.append(sin(x)*cos(0.1*y))
     7      1000        606.0      0.6      0.0          arr.append(row)
     8         1          1.0      1.0      0.0      return arr
```

DEMO

Profiling in a notebook

Part 1

- see profiling-example.ipynb
- **pixi run "jupyter lab"**

For existing code: cProfile

Python provides several profilers, but the most common is *cProfile* (note: gprof for c++). It's what Jupyter uses by default

Profile an entire script:

- Run your script with the additional options:

```
python -m cProfile -o output.pstats  <script>
```

For a command-line script like **pkoffee**, it's better to run it as a module from the python interpreter:

```
python -m cProfile -o prof.pstats -m pkoffee.cli -- analyze -d coffee_productivity.csv -o out.pi
```

have to run the module, not "pkoffee" ↑ ↑ note the double dash: means after this, are the script's (not python's) arguments

- this generates a **binary data file (*output.pstats*)** that contains statistics on **how often** and **for how long** each function was called
- There is a built-in **pstats** module that can read it and print stats, but it's a bit difficult to use... but there are GUIs! (recommended)

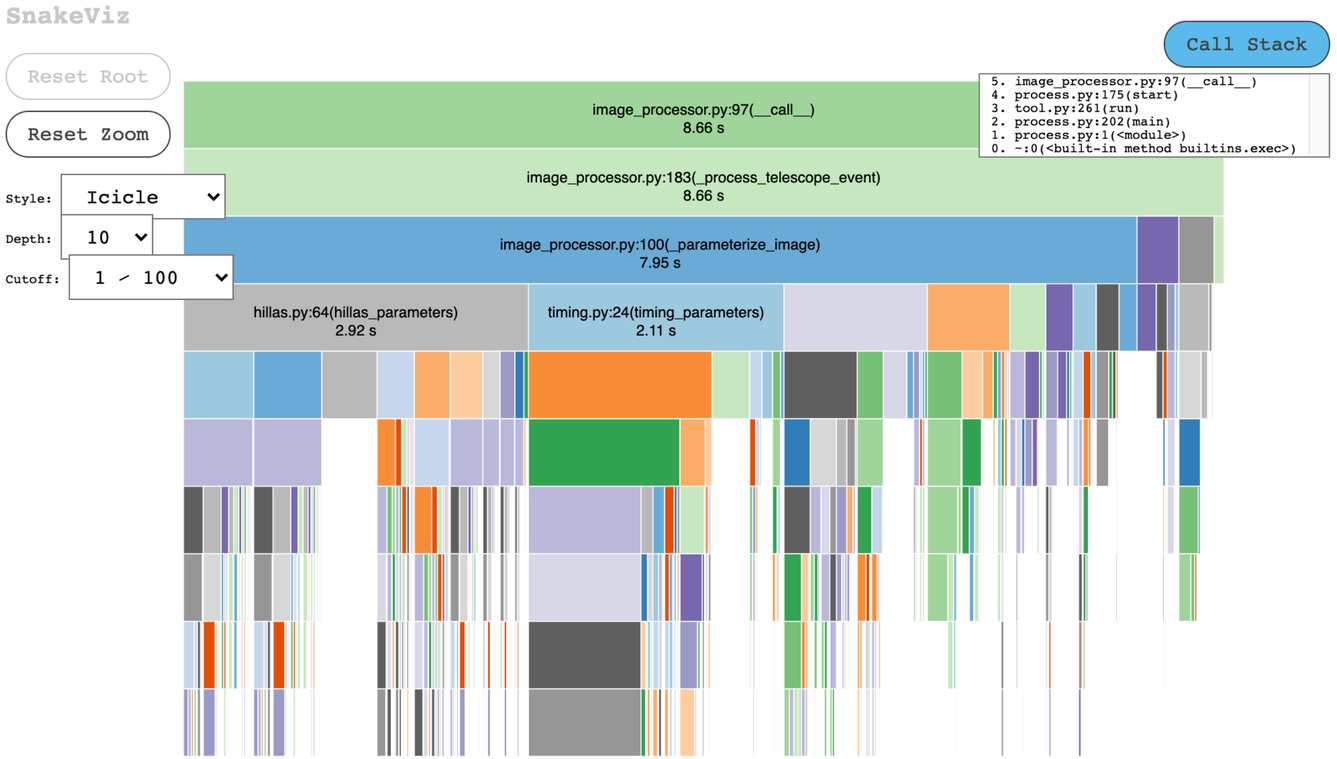
Tip: use a gui to view stats output

Viewing with *SnakeViz*

```
% pixi add snakeviz
% snakeviz output.pstats
```

- interactive call statistics viewer
- this is not the only one, but it's nice and simple and runs in your browser.
- Click and zoom to see the results

DEMO TIME



Search:

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
2226	4.629	0.002079	4.629	0.002079	extractor.py:195(neighbor_average_waveform)
9203931/4273437	4.232	9.903e-07	31.68	7.414e-06	~:0(<built-in method builtins.getattr>)
2701625	2.966	1.098e-06	3.257	1.205e-06	~:0(<built-in method numpy.array>)
1917670	2.925	1.525e-06	8.082	4.215e-06	quantity.py:289(__new__)
1726186	2.848	1.65e-06	5.015	2.905e-06	baseframe.py:850(get_representation_component_names)
2690488/1922760	2.069	1.076e-06	23.99	1.248e-05	attributes.py:95(get)

Aside: Line Profiling without Jupyter

What about time spent in each line of code?

The line_profiler module can help:

```
% conda install line_profiler
```

- mark code with @profile:

```
from line_profiler import profile

@profile
def slow_function(a, b, c):
    ...
```

- Then run:
  - % **kernprof** -l script_to_profile.py
- which generates a .lprof file that can be viewed with:
  - % **python -m line_profiler** script_to_profile.py.lprof

```
File: pystone.py
Function: Proc2 at line 149
Total time: 0.606656 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   149                                           @profile
   150                                           def Proc2(IntParIO):
   151     50000        82003      1.6     13.5      IntLoc = IntParIO
+ 10
   152     50000        63162      1.3     10.4      while 1:
   153     50000        69065      1.4     11.4          if Char1Glob
== 'A':
   154     50000        66354      1.3     10.9              IntLoc =
IntLoc - 1
   155     50000        67263      1.3     11.1              IntParIO
= IntLoc - IntGlob
   156     50000        65494      1.3     10.8              EnumLoc =
Ident1
   157     50000        68001      1.4     11.2          if EnumLoc ==
Ident1:
   158     50000        63739      1.3     10.5              break
   159     50000        61575      1.2     10.1      return IntParIO
```

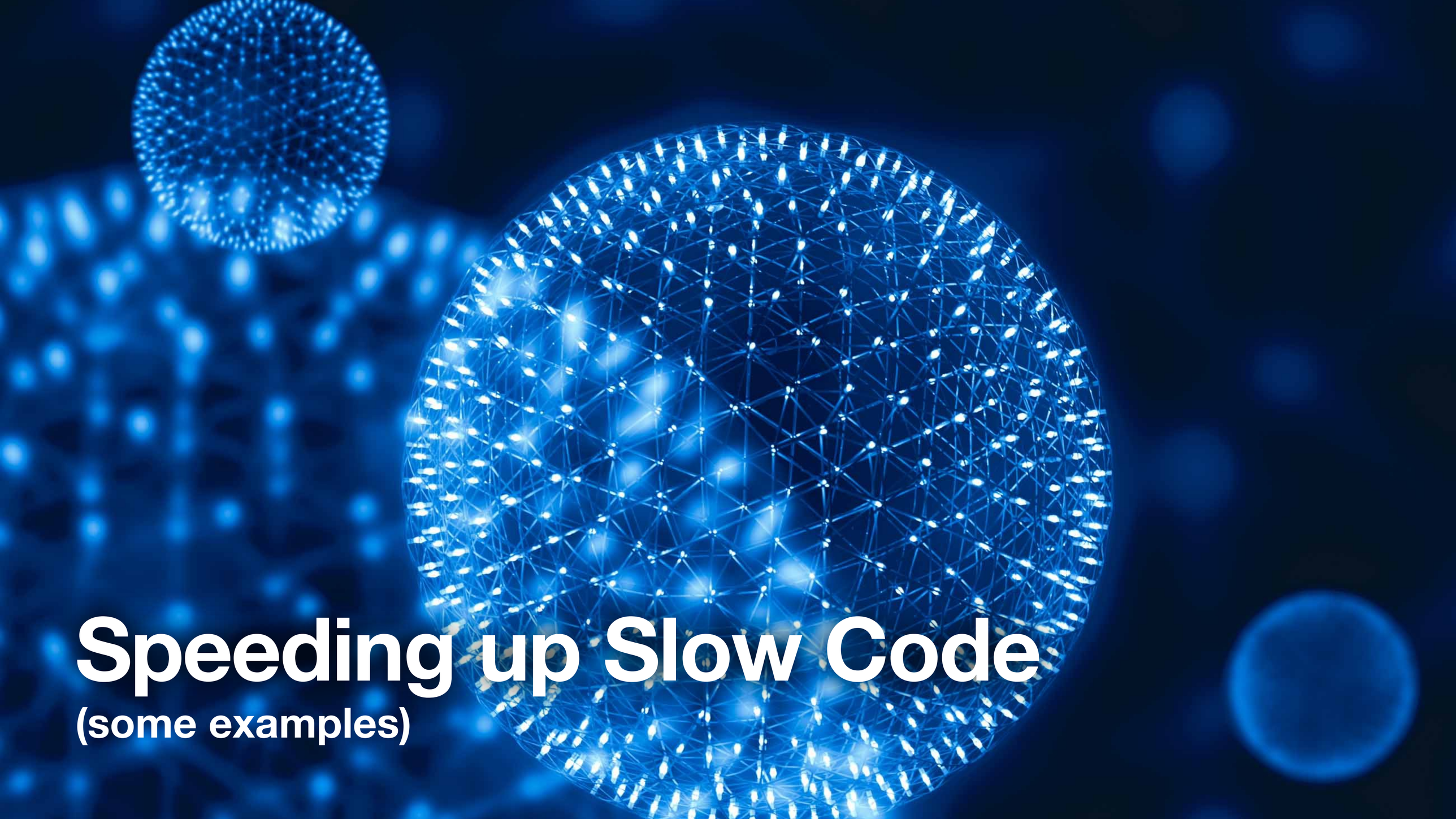
DEMO

Profiling everything using cProfile

Using the command-line to profile pkoffee

```
$ pixi shell

$ cd analysis

$ python -m cProfile -o prof.pstats -- pkoffee.cli  analyze --data-file coffee_productivity.csv --
output fitted_models.toml --show-rankings
```

Then try *snakeviz* to see the results (*pixi global install snakeviz*)



Speeding up Slow Code

(some examples)

“

We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**

- *Sir Tony Hoare?*
or Donald Knuth?

“

We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**

- *Sir Tony Hoare?*
or Donald Knuth?

From a 1974 article on why GOTO statements are good

Steps to optimization

1) Make sure code *works correctly* first

- DO NOT optimize code you are writing or debugging!

2) Identify use cases for optimization:

- how often is a function called? Is it useful to optimize it?
- If it is not called often and finishes with reasonable time/memory, stop!

3) *Profile* the code to identify bottlenecks in a scientific way

- **time spent** in each function, statement
- **memory use** per function, statement, time-step

4) Try to re-write as little as possible to achieve improvement

5) Think about overall design, if small changes are not sufficient

- some times the *design* is what is making the code slow... can it be improved? (e.g.: ***flat better than nested***!)
- Don't be afraid to re-write or refactor!

Scientific Python Universe

Some packages of Python's scientific stack. Source: VanderPlas 2017, slide 52.

Fast numerics
(avoid or speed-up loops):

- **Numpy**: fast N-dimensional arrays and array operations, vectors, tensors.
- **Numba**: compile python functions!

Other options (less recommended):

- **Cython**: python-like language to generate C-code
- Write C/C++/Rust and call it from python

Numpy: fast python numerics

https://numpy.org/

Replace python loops with array operations, slices, linear algebra

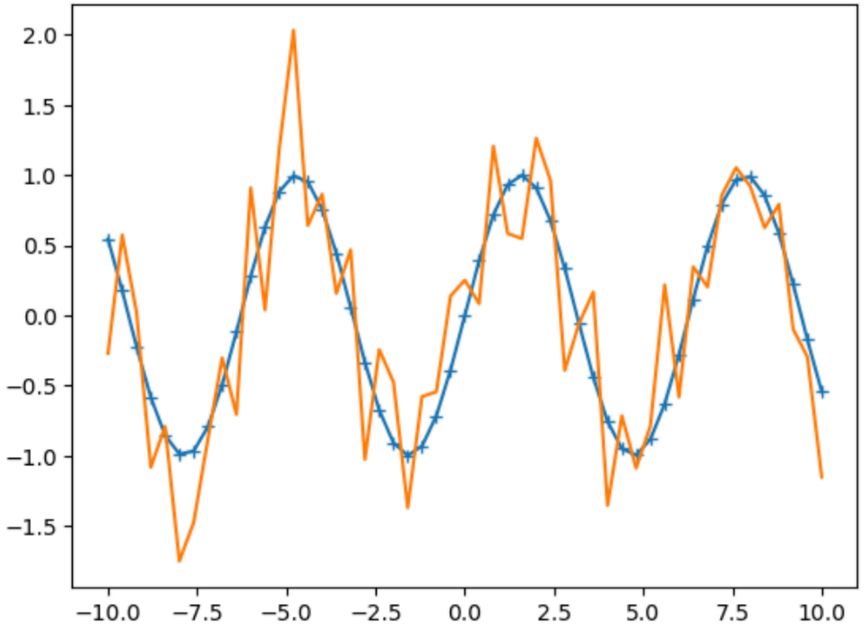
- Fully implemented in fast compiled languages
- supports N-dimensional arrays and fast transformations
- don't call a function on many small pieces of data when you can **call it on an array all at once**
- Nearly the speed of compiled languages

Have to think in arrays!

```
: import numpy as np
  import matplotlib.pyplot as plt
```

```
: X = np.linspace(-10, 10, 51)
  Y = np.sin(X)
  Y_noisy = Y + np.random.normal(scale=0.5, size=51)
  plt.plot(X, Y, "+-")
  plt.plot(X, Y_noisy)
```

```
: [<matplotlib.lines.Line2D at 0x113b28850>]
```



Numpy: fast python numerics

https://numpy.org/

Replace python loops with array operations, slices, linear algebra

- Fully implemented in fast compiled languages
- supports N-dimensional arrays and fast transformations
- don't call a function on many small pieces of data when you can **call it on an array all at once**
- Nearly the speed of compiled languages

Have to think in arrays!

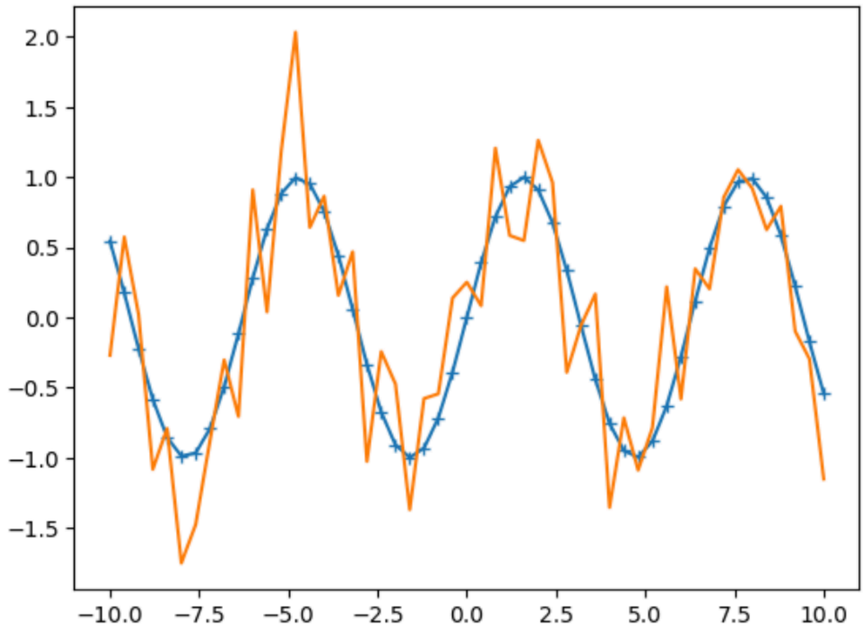
This requires practice, and feels very strange at first if you are coming from C programming!

Take some time to look through the *NumPy* and *SciPy* **API documentation** - there are tons of interesting functions to help you!

```
: import numpy as np
  import matplotlib.pyplot as plt
```

```
: X = np.linspace(-10, 10, 51)
  Y = np.sin(X)
  Y_noisy = Y + np.random.normal(scale=0.5, size=51)
  plt.plot(X, Y, "+-")
  plt.plot(X, Y_noisy)
```

```
: [<matplotlib.lines.Line2D at 0x113b28850>]
```



Numba Compile python just-in-time*

*JIT: compile code at runtime

https://numba.pydata.org/

Takes python code and *directly* uses introspection to compile it with LLVM

- **automatic**, *but only works on supported operations*
  - ➤ *most pure python functions, some numpy functions, but can't call anything else!*
  - ➤ ***fails** with complex objects like pandas DataFrames! (though there are ways to help)*
- Can generate **NumPy "ufuncs"** directly (function that works on scalars but is run on all elements of an array), which are too slow to write in python normally.
- Can even compile to **GPU** code for nVidia *CUDA* and AMD *ROC* GPUs!

```
from numba import jit
from numpy import arange

# jit decorator tells Numba to compile this function.
# The argument types will be inferred by Numba when function is called.
@jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
    return result

a = arange(9).reshape(3,3)
print(sum2d(a))
```

just add this decorator, and it's magic (nearly)

Numba with NumPy

Numba supports a large number of NumPy functions (and even some scipy)**:**

- It does not actually call NumPy code!
- it *re-implements* it in a way that is compilable with LLVM.

  https://numba.pydata.org/numba-doc/dev/reference/numpysupported.html

So what is the point? Isn't NumPy really optimized already?

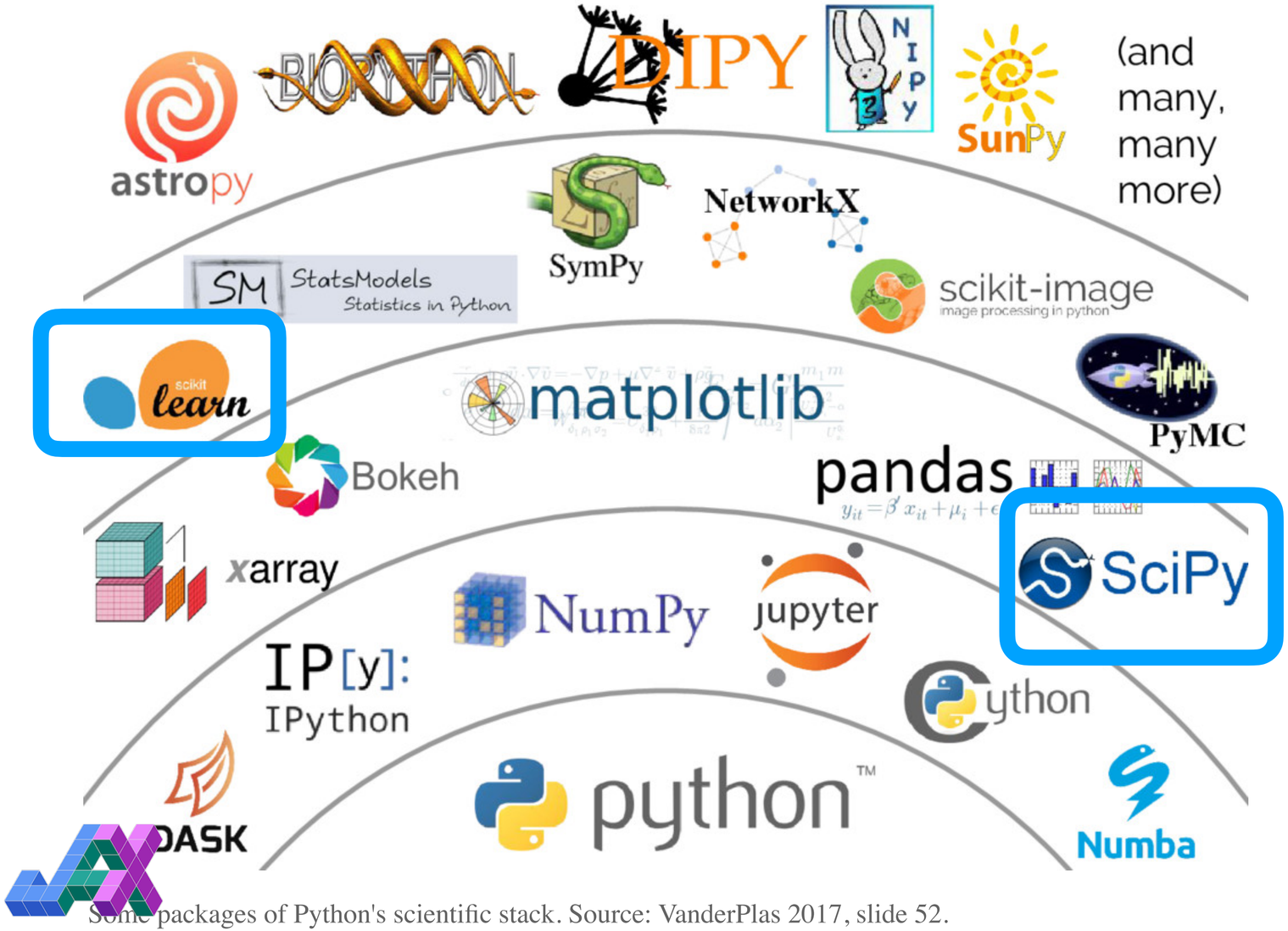
- Minimize intermediate results!
  - ➤ numpy operations often have to allocate memory for data that is not needed in the end:

```
x = np.arange(1000)
result = A * x**2 + B * x + C
```

in C, you might do this all in one loop, with no extra memory needed:

```
for (i=0; i<x.size; i++) {
    result[i] = A*x[i]*x[i] + B*x[i] + C;
}
```

Scientific Python Universe



Some packages of Python's scientific stack. Source: VanderPlas 2017, slide 52.

Fully implemented and well-designed algorithms built on numpy:

- **scipy**: interpolation, integration, minimization/optimization/fitting/signal processing/...
  - ➤ you won't implement something faster in general!
  - ➤ Written by experts, in FORTRAN, C, etc.
- **scikit-learn**: machine learning/model fitting
  - ➤ not covered in this lecture, but

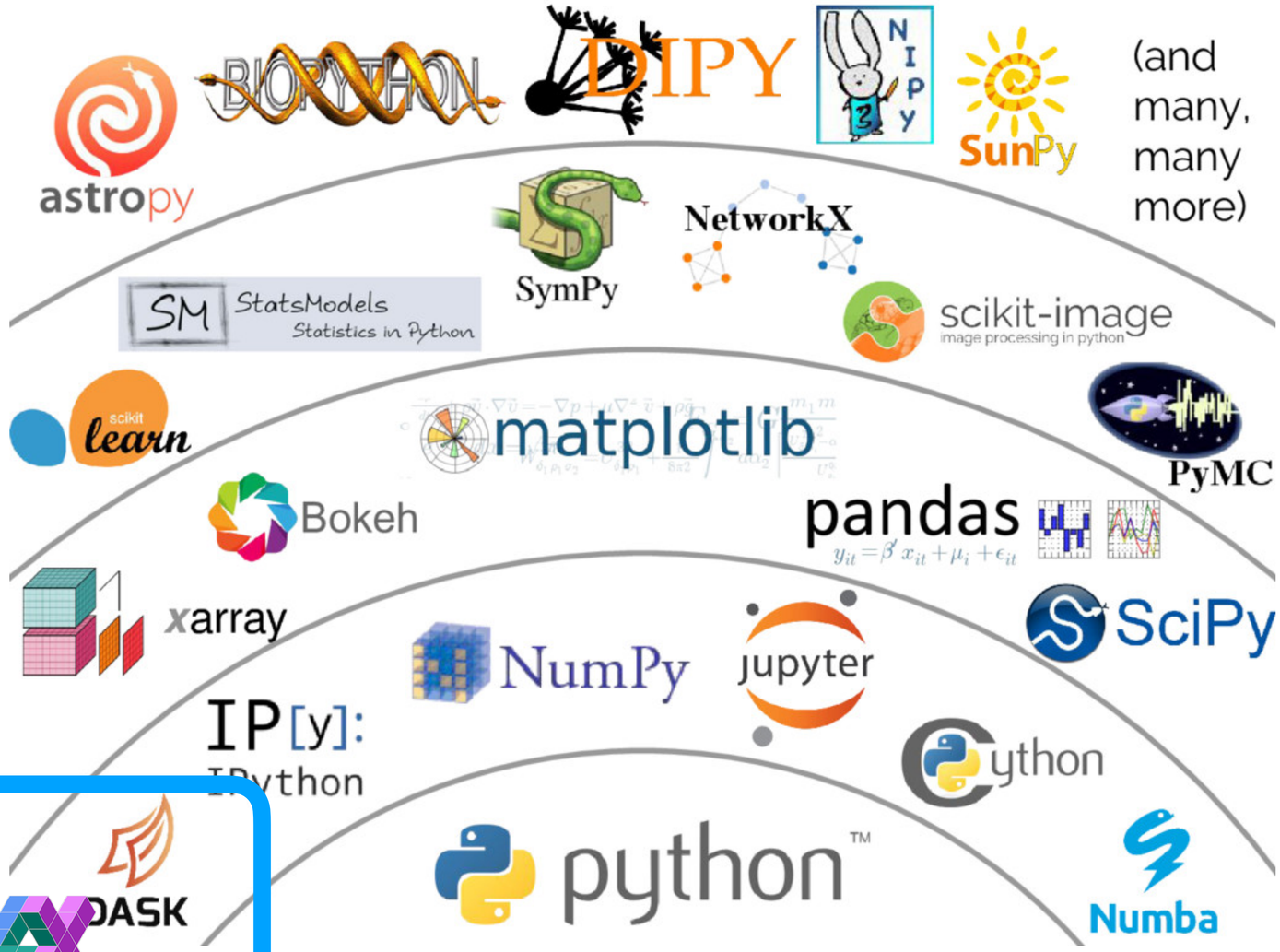
DEMO

Profiling in a notebook

an advanced study: back to our
Gaussian Blur examples...

- see profiling-example.ipynb
- **pixi run "jupyter lab"**

Scientific Python Universe



Some packages of Python's scientific stack. Source: VanderPlas 2017, slide 52.

Parallelization and HPC:

Nearly drop-in replacements for NumPy

- Dask:
  - ➤ Parallelize large NumPy operations across multiple machines
  - ➤ Supports pandas DataFrames
- JAX:
  - ➤ high-performance numerical computing and large-scale machine learning
  - ➤ Auto-differentatiation of array operations

Others... PyTorch, etc

Do nothing: Python keeps getting faster

Not an inherent problem with the *language!*

- The reference implementation of python (CPython) is written in C and is continuously improving!
  - ➤ Python 3.11: large speedup (>60%) with optimized interpreter
  - ➤ Python 3.12: faster dicts and sets, fast f-string
  - ➤ Python 3.13: better adaptive interpreter, experimental JIT support
  - ➤ Python 3.14: support for free-threading + multiple interpreters (faster parallel code)
- Future version have more more just-in-time compilation and more optimizers

Python ≠ CPython

- PyPy: alternative interpreter will full JIT compilation (not always faster if you use Numpy though!)

Do nothing: Python keeps getting faster

Not an inherent problem with the *language!*

- The reference implementation of python (CPython) is written in C and is continuously improving!
  - ➤ Python 3.11: large speedup (>60%) with optimized interpreter
  - ➤ Python 3.12: faster dicts and sets, fast f-string
  - ➤ Python 3.13: better adaptive interpreter, experimental JIT support
  - ➤ Python 3.14: support for free-threading + multiple interpreters (faster parallel code)
- Future version have more more just-in-time compilation and more optimizers

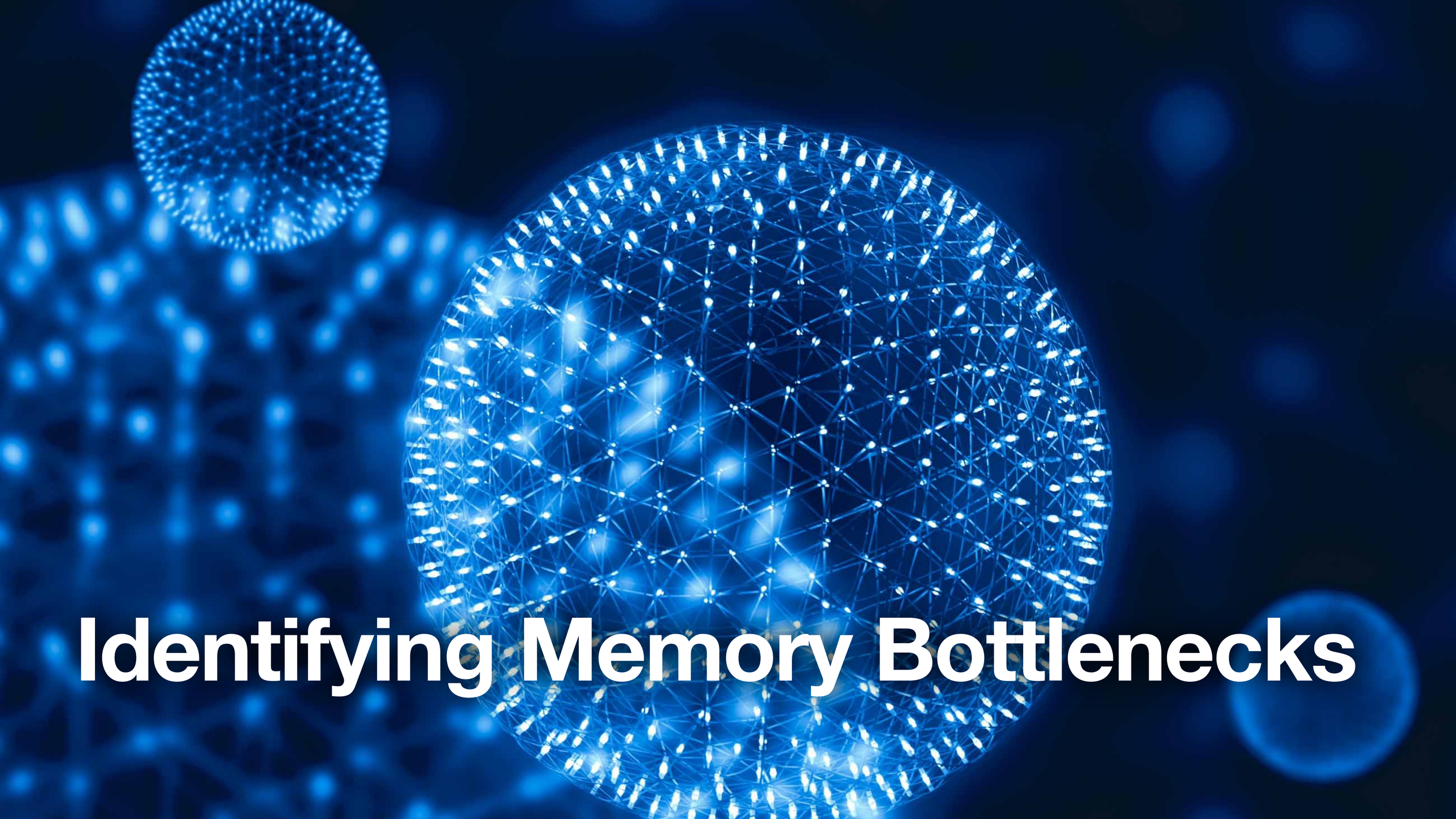
Python ≠ CPython

- PyPy: alternative interpreter will full JIT compilation (not always faster if you use Numpy though!)

So one option to optimization is:

Do nothing!

Wait for a faster implementation, or a new version of CPython to be released, or swap in a completely different implementation!



Identifying Memory Bottlenecks

CAVEAT!

- The standard package used for memory profiling ***memory_profiler*** is currently not (or poorly) maintained!
  - https://github.com/pythonprofilers/memory_profiler
  - **JupyterLab support broken** on Python 3.12+ due to minor dependence on distutils
    - there is a unmerged pull request from 2024 to fix this

- For now I have pinned python=3.11 in the examples (and in the exercise repo) to avoid this issue.
  - Works with later python versions in the **terminal or in scripts**, just **not in Jupyter Notebooks**.

- Alternatives?
  - **memray**: more modern, but Linux and MacOS only (no windows)

How memory is allocated in Python

Memory allocated implicitly when you create an object:

- type(x); sys.getsizeof(x)

Statement	Memory allocated (bytes)	Note
x = 12	1 integer (28)	Note unlike in C, there is overhead in a single variable
x = 101000**	1 integer (468)	Integers in python are not fixed bit! You can store arbitrarily large ones
x = 10.2	1 float (24)	Same overhead: 64-bit float with some extra info
x = [ ]	1 blank list container (56)	
x = [1,2,3,4,5]	list of 5 integers (104)	
x = np.arange(0)	empty numpy array (112)	Again, some overhead from "array"
x = np.arange(1000)	numpy array + 1000 int64s (8112)	
x = "this is a string"	string + 16 characters (49+16=65)	Python uses Unicode (one byte per character usually)

How memory is allocated in Python

You don't need to de-allocate memory in python, it's done automatically!

Garbage Collection

- Python keeps a count of references to each object in memory
- Periodically, the **garbage collector** runs in the background
  - ➤ If reference_count == 0, memory is de-allocated!
  - ➤ Even can find dangling circular references

You CAN explicitly delete references however!

```
del variable_name  # remove variable from scope
```

- if the memory pointed to by that variable is not pointed to elsewhere, it will be deallocated next time the garbage collector runs.
- Normally, you don't need to do this, however.

DEMO

Memory Profiling

see notebook :
memory_allocation_example.ipynb

Memory Leaks

Definition:

- memory which is no longer needed is not released.
  - ➤ In python this means a reference to that data is still in scope somewhere!
- data stored but cannot be accessed by the running code
  - ➤ generally not a problem with a garbage collector, but careful if you write C/C++ code!

Identification:

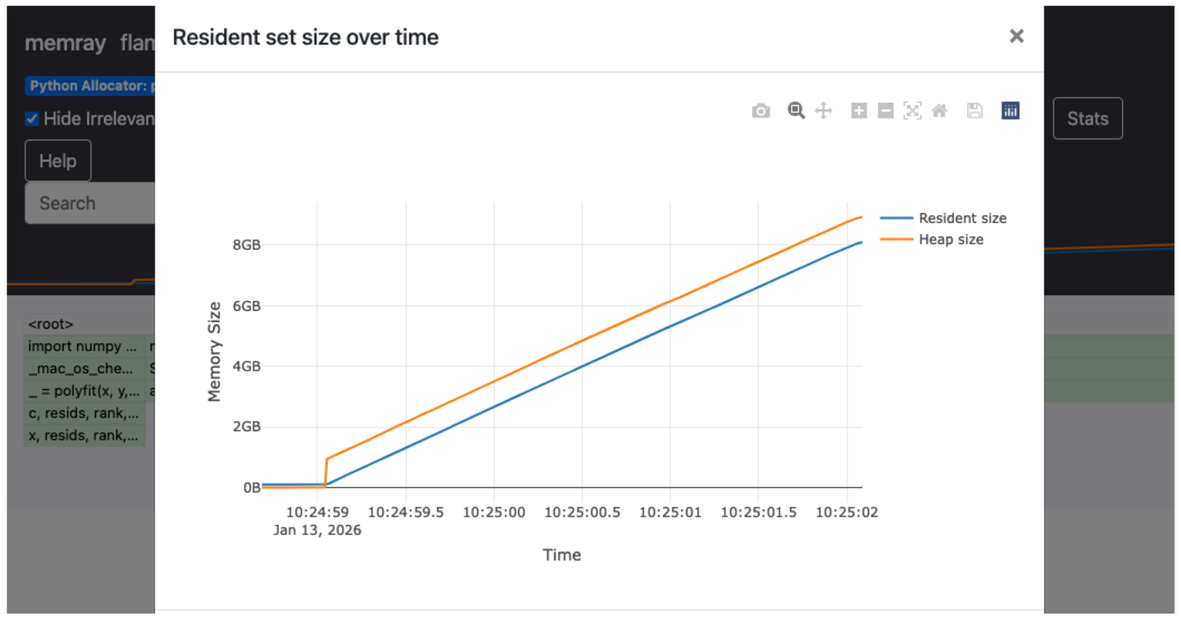
- Memory use grows over time during program execution.

A memory leak

```
[3]: %%memray_flamegraph
import numpy as np

SOME_GLOBAL_STATE = []

def my_leaky_function():
    SOME_GLOBAL_STATE.append(np.ones((1000,1000)))

for ii in range(1000):
    my_leaky_function()
```

```
/Users/kkosack/Projects/ScientificComputingSchools/Working/2026-S3School/optimize-example/.pixi/envs/default/lib/python3.11/site-packages/rich/live.py:256: UserWarning: install "ipywidgets" for Jupyter support
  warnings.warn('install "ipywidgets" for Jupyter support')
```

```
Results saved to memray-results/tmp648atpsf/flamegraph.html
```

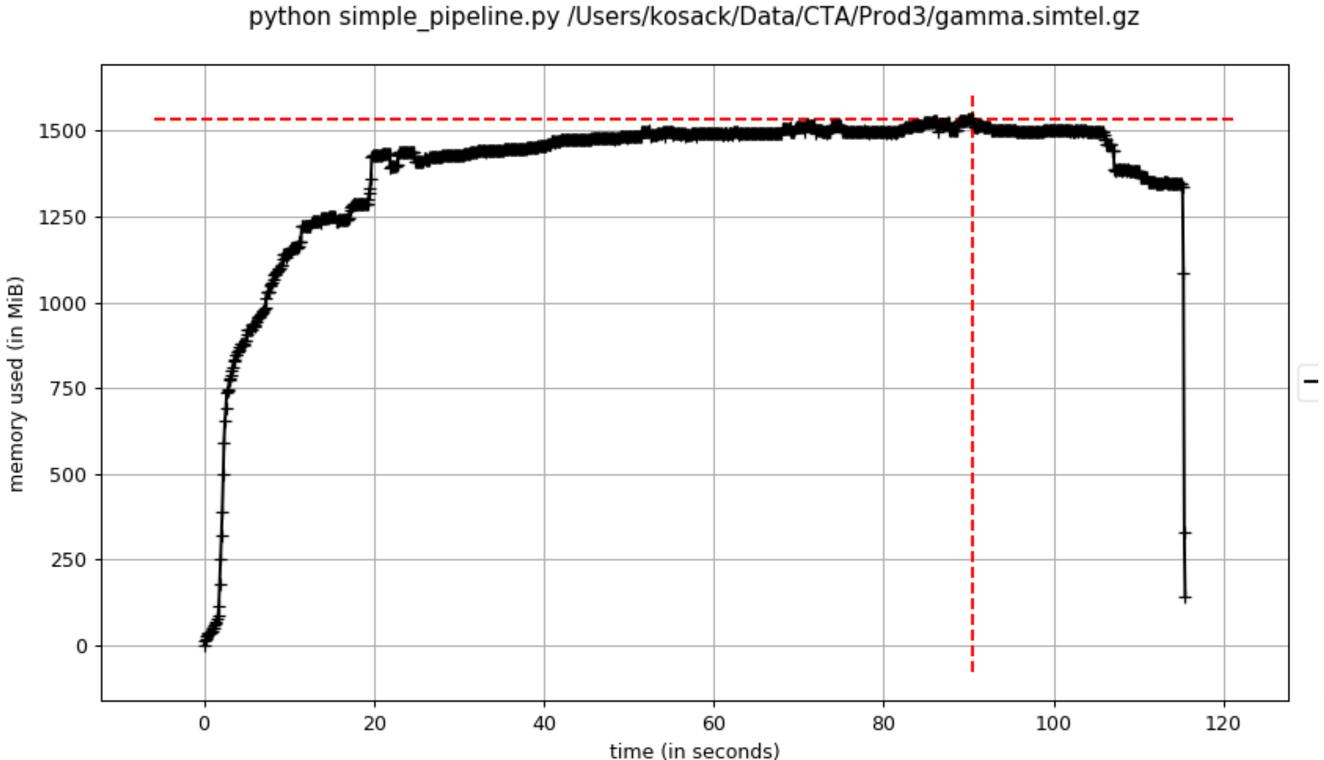


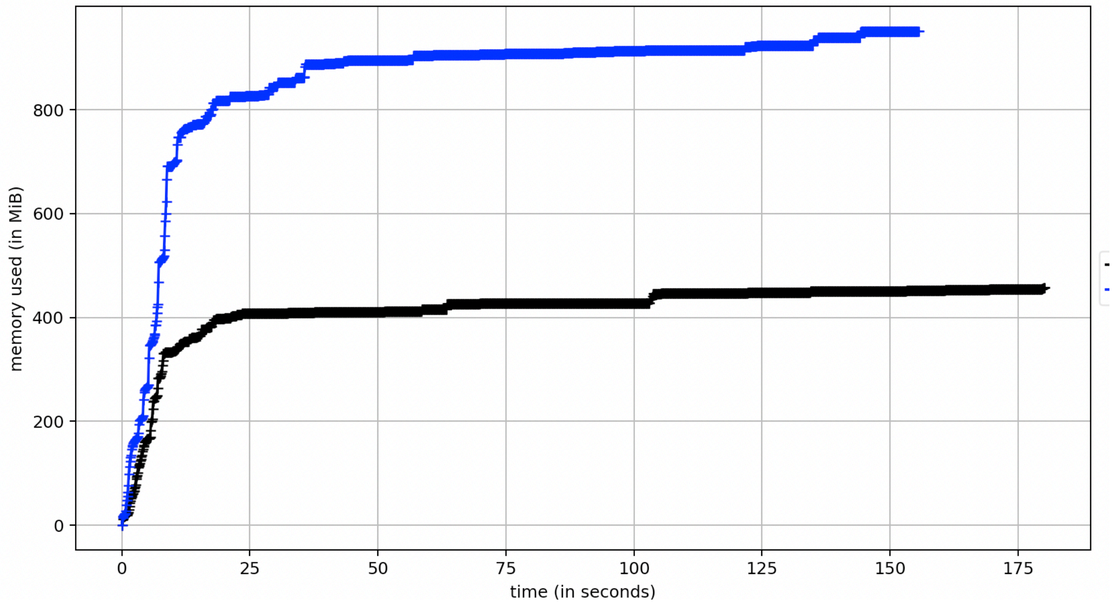
Memory Profiling

Use of CPU is not the only thing to worry about… what about RAM? Let's first check for memory leaks…

```
% pixi add memory_profiler
% pixi run "mprof run python <script>"
% pixi run "mprof plot"
```

if not already there





Memory Profiling in detail

Cumulative is nice, but we want to see the memory for a particular function or class…

- decorate the function you want to profile (line-wise) with memory_profiler.profile

```
% python -m memory_profiler <script>
```

Decorate what we want to measure (no import needed)

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    17                                           @profile
    18                                           def main():
    19         1          3.0      3.0      0.0      if len(sys.argv) >= 2:
    20                                                   filename = sys.argv[1]
    21                                               else:
    22         1        485.0    485.0      0.0          filename = get_dataset_path("gamma_test_large.simte
    24         1    3572651.0 3572651.0      9.8      with EventSource(filename, max_events=500) as source:
    26         1     438843.0 438843.0      1.2          calib = CameraCalibrator(subarray=source.subarray)
    27         2     249622.0 124811.0      0.7          process_images = ImageProcessor(
    28         1          2.0      2.0      0.0              subarray=source.subarray, is_simulation=source.
    29                                                   )
    30         1       1363.0   1363.0      0.0          process_shower = ShowerProcessor(subarray=source.su
    31         2     276938.0 138469.0      0.8          write = DataWriter(
    32         1          0.0      0.0      0.0              event_source=source, output_path="events.DL1.h5
    33                                                   )
    35       111   11506526.0 103662.4     31.5          for event in tqdm(source):
    36       110    1313386.0  11939.9      3.6              calib(event)
    37       110    2353948.0  21399.5      6.4              process_images(event)
    38       110   14044245.0 127675.0     38.4              process_shower(event)
    39       110    2814913.0  25590.1      7.7              write(event)
```

Output shows the time spent in the line or block (e.g. if , for)

Memory Profiling in a Notebook

Again, you can do memory profiling using magic commands in an iPython (Jupyter) notebook

- Enable the memory profiling notebook extension:

```
%load_ext memory_profiler
```

- Now you have access to several magic functions:

Like %timeit, but for memory usage:

```
%memit <python statement>
```

or a more full-featured report:

```
%mprun -f <function name> <statement>
```

```
In [40]: %memit range(100000)
         peak memory: 89.61 MiB, increment: 0.00 MiB

In [41]: %memit np.arange(100000)
         peak memory: 90.12 MiB, increment: 0.52 MiB
```

Caveats:

- the peak memory usage shown in the notebook may not relate to the function you are testing! It is the sum of all memory already allocated that has not yet been garbage collected. (so look at the "increment" instead).
- **%mprun** only works if your functions are **defined in a file** (not a notebook) and imported into the notebook

DEMO TIME

back to "profiling_example.ipynb"

Memory Profiling: jump to debugger

Automatic Debugger breakpoints:

- you can automatically start the debugging if the code tries to go above a memory limit, to see where the allocation is happening:

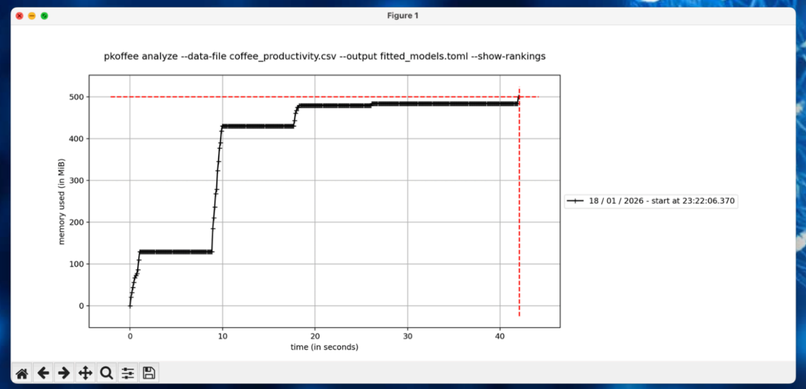
```
% python -m memory_profiler --pdb-mmem=100  <script>
```

will break and enter debugger after **100 MB** is allocated, on the line where the last allocation occurred

Usefull for when you don't know where to look.

> python **-m memory_profiler --pdb-mmem=100** pkoffee.cli analyze --data-file coffee_productivity.csv --output fitted_models.toml --show-rankings

Activity: speed and memory profiling



Find the speed and memory bottlenecks using profiler tools!

- Check out the branch: day_2_solution_slow

```
git fetch upstream day_2_solution_slow
git switch day_2_solution_slow
```

- I've (not so secretly) modified the code in yesterday's solution to add some bottlenecks!
- You might notice running your analysis is a bit slower than before...

Without looking at the code or git log*, use one or more profiling tools to find which function is slow and where some memory is wasted.

* Obviously that would be too easy, but in the real world it's YOU that added the slowness, and it was not on purpose!

HINT: you can run pkoffee's CLI directly with python using:
python pkoffee.cli analyze