

SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 24

Fabian Wermelinger

Harvard University

CS107 / AC207

Tuesday, November 22nd 2022


LAST TIME

- Hands-on exercises using SQLite and Pandas in Python:
 - Table joins in SQL
 - SQL interface in Pandas
 - SQL-like operations in Pandas

TODAY

Main topics: *Debugging, Profiling code and some thoughts on optimization*

Details:

- Back to 
- **Debugging**: how to locate *bugs* in (Python) code.
- **Profiling**: how to locate *performance bottlenecks* in (Python) code.
- **Performance**: understand bytecode limitations to identify performance issues in your code.

AGENDA CHECK:

- Milestone 2 due today 11:59pm. You can find the milestone details at <https://harvard-iacs.github.io/2022-CS107/project/M2>.
- **Quiz 4** takes place next Tuesday. **You will have 25 minutes** for 12 questions. The quiz is available within a 12 hour time window (starting 9:00am). Question topics are posted in the forum: <https://edstem.org/us/courses/24296/discussion/2190445>

DEBUGGING

- **Debugging** refers to any technique to locate or prevent bugs (defective code) in a computer program.
- The term "debugging" (or "bug") is attributed to **Grace Hopper** while working on Mark II at Harvard. Supposedly a moth was stuck in the mechanical parts of the computer and her co-workers were debugging the machine.
- ***Codes "free" of bugs are extremely rare.***
 - Humans are a main source of bugs.
 - Open source projects have usually less bugs than commercial/proprietary software (more eyes see the code).
 - Bugs *can* make your program crash (*these are obvious*) or make your code run but behave strangely (*these are hard bugs to eliminate*).
- Good debugging skills will make you a more efficient and confident programmer. It requires knowledge of the debugging tools but also understanding of low-level concepts such as ***machine architecture and instructions*** (bytecode in Python).

DEBUGGING

- The main tool is called a **debugger**. It is a program that allows you to *analyze your code by stepping through the code while it runs*. In each step you can *examine values and memory states*.
- **Test driven development** is another passive debugging technique. Writing (good) tests helps to avoid bugs and side-effects.
- **Defensive programming:**
 - Use the `assert` statement in your code. You can avoid the additional overhead of `assert` by running your Python code with `python -O`.
 - Address boundary/edge cases in your tests. Also test for things you are certain they will not happen (**Murphy's Law**). *For example: non-physical results such as negative chemical concentrations, a negative age of a person and so on.*
 - **Apply the techniques discussed in class:** write unit and integration tests, use version control, write modular code (do not duplicate code), document carefully (this includes commit messages).

DEBUGGING TECHNIQUES

Some techniques that are being used to debug:

- **Interactive debugging:** This requires a *debugger* to interact with your program as it executes. **Examples:** *gdb* (GNU debugger for C, C++ and other languages), *pdb* (Python debugger) and various others in integrated environments.
- **Trace based debugging:** Manually creating a trace in your program by using print statements. **Example:** *(do you see any disadvantages with this approach?)*

```
1 def main():
2     x = 1
3     print(f'Value of x before function call: {x}')
4     f(x)
5     print(f'Value of x after function call: {x}')
```

- **Online debugging:** Attach to a running process to isolate a problem (*could be in a production environment*).
- **Isolation by bisection:** Narrowing down to smaller sections until you can see the bug. Examples are *divide and conquer* (**example:** *git bisect*).
- **Inspection after failure:** Refers to the analysis of a memory dump after a program has crashed. These are called *core dumps* historically because of the early *magnetic core memory* modules → *a debugger can load such files*.

SOME COMMON FORMS OF BUGS

- The most common form of bugs are **memory corruptions**. These may cause **segmentation faults** and crash your program. *If the program does not crash then its behavior will be strange and finding the root of the bug is usually hard. It is very easy to produce such bugs in the C programming language, harder in C++ and very hard in the Rust programming language. (It is less likely to run into them in pure Python.)*
- Other type of bugs involve illegal operations such as division by zero, **leakage of dynamic memory**, loops that run indefinitely or variables that have not been initialized (*uninitialized memory*).
- Yet other issues (*these are not necessarily bugs*) may not affect the correctness of your program but they can impact the **performance** of your program. Examples are wrong memory layouts or adverse memory access patterns for architectures with cache hierarchies (topic in CS205).

DEBUGGING PROCESS

- You have to *develop an intuition* that there may be a bug in your program. This is *easy when the program crashes* but can be *hard when the bug is more silent and shows up randomly*. You must expect a certain behavior of your program and by debugging you prove that this expectation is true or false (*if false → suspicion for a bug*).
- Once you suspect a bug, you must find a way to *reproduce* it → *not always easy*. You should consider the following steps:
 - Reduce/disable components in your code that seem irrelevant (divide and conquer).
 - Reduce the number of inputs if possible or restart your code from a snapshot before the anomalous behavior.
 - Add traces in your code (e.g. print statements) or use a *debugger* directly if possible to inspect memory values.

DEBUGGER

A *debugger* is a program that allows to pause and step through your code at runtime. The GNU debugger *gdb* is often used for C/C++ code. Python ships with its own *pdb* debugger.

Main debugger operations:

- ***Stepping through the source code:*** this can be done on a per line basis or per instruction basis (*pdb cannot step per instruction*). In order to stop at a particular point → you can set a *breakpoint* (*you can stop and resume execution at will*).
- ***Inspecting variables:*** when you pause, you can inspect the values of variables in the current frame and all other frames below the current one.
- ***Watching a variable of interest:*** breakpoints and variable inspection are combined to a *watchpoint*, which causes the debugger to stop whenever the value of a watched variable changes (*this concept is not supported in pdb*).
- ***Moving around the call stack:*** the debugger allows to move to any stack frame that is currently on the stack such that you can inspect variable values in the caller frame as well as generating a *backtrace*.

BREAKPOINTS

- *Breakpoints are like tripwires.* You can set them at *arbitrary* places in your code and the debugger will stop when it "trips" over one.
- *"Places"* can be source line numbers, a code address or function entry point.
- Breakpoints are tracked:
 - You can get statistics about how often a breakpoint has been visited.
 - You can configure breakpoints to stop only after the n -th hit.
- Once stopped at a breakpoint, you can inspect variables or remove/modify breakpoints and continue execution anytime.

PYTHON DEBUGGER: pdb

`pdb` is an interactive debugger for Python programs. You can import `pdb` in your code or run it from the command line with `python -m pdb your_script.py`.

Examples:

- *Running from within the interpreter:* (you can run the debugger on specific code defined in your module)

```
1 >>> import pdb
2 >>> import your_module
3 >>> pdb.run('your_module.test()')
```

- *Inserting a breakpoint in your code:*

```
1 # some code
2 import pdb; pdb.set_trace()
3 # more code
```

Since Python 3.7 you can use the `breakpoint()` built-in:

```
1 # some code
2 breakpoint()
3 # more code
```

PYTHON DEBUGGER: pdb

- Running the following will drop you into the pdb shell:

```
1 $ python -m pdb factorial.py
2 > /tmp/factorial.py(5)<module>()
3 -> def factorial(x)
4 :(Pdb)
```

Note: the default pdb debugger does not offer much color highlighting. If you prefer visual support through color you can run `ipdb` instead inside an `ipython` shell.

- The debugger pauses at the first line. The basic commands are similar to `gdb` (*you can also abbreviate them if there is no ambiguity*):
 - `run`: restart the debugging session.
 - `next` execute next line (skipping over functions)
 - `step` execute next line (stepping into functions)
 - `list` print lines of code around the current line.
 - `print x` print the value of name `x`.
 - `break [line_number]` set a breakpoint on the `line_number`.
 - `bt` Print the backtrace starting from the current frame.
 - `help` Print the help menu (*this command is important* → *example: `help break`*).

PYTHON DEBUGGER: pdb

Example: factorial(5)

- We now step through the factorial code we did for C++ and gdb before using the Python debugger. The `factorial.py` source is located in the `pdb_factorial` directory of the lecture code handout.
- The debugger is started with the following Python command:

```
1 $ python -m pdb factorial.py
```

Alternatively, you can use `ipdb` with iPython for color *highlighting*:

```
1 $ ipython -m ipdb factorial.py
```

- ***We want to investigate the following:***
 1. List the source code at the start of the debugging session.
 2. Set a breakpoint at the `factorial` function and stop only if `x == 1`. When done list all active breakpoints.
 3. Continue running until you reach the breakpoint.
 4. Print a backtrace and study the recursion of `factorial()`. Go up and down the call stack.
 5. Step through the return statements of the recursion and inspect the return values.
Using the built-in `locals()` is helpful to inspect the local names.

PYTHON DEBUGGER: pdb

Summary:

- The gdb and pdb (or ipdb) debuggers share the same command names for most of the commands → *because developers often use both tools.*
- The debugger is an **extremely powerful tool** and you should integrate it in your development workflow. We only touched the basics today → *continuous use will make you proficient (the set of commands in pdb is small).*
- The interactive inspection of variable values is very helpful for debugging, you do not need to manually insert `print()` statements in your code once you know how to use a debugger.
- Do not shy away from the debugger. Once you are familiar with it, the reward will be valuable! ***Useful references:***
 - [The Python debugger \(pdb\)](#)
 - [The iPython debugger \(ipdb\)](#)
 - [pdb cheatsheet \(pdf\)](#)

PROFILING

- **Debugging:** verify/investigate the *correctness* of code.
- **Profiling:** analyze *performance bottlenecks* of correct code.

Performance analysis and optimization:

- **Why:** *when you take your code to production you want it to perform.* Efficient code means you maximize your returns on investment. If you buy a \$10'000 GPU and run at 50% nominal peak, you waste \$5'000 plus the efforts to write the code. Also, you will not be able to target large computational problems at 50% efficiency!
- **When:** you start with *optimizations* once you have a running baseline that is *well tested and debugged*.
- **How:** there are many optimization techniques. The approach for performance optimization is *top-down*: from simple (*low time investment*) to difficult (*high time investment* → *are these optimizations really beneficial?*).

Before you start with optimizations you must **know where to start** and the *profiler* is the primary performance analysis tool for this task.

PROFILING

- A *profile* of your code lists statistics of the various function calls executed when running your code.
- It will show you *in which functions/subroutines you are spending the most time*, thus enabling you to *identify bottlenecks in your code*.
- These bottlenecks are the first targets for optimization → *removing them often leads to a significant improvement already.*

Commonly used profilers:

- **GNU gprof** for programs written in C, C++, Fortran or Pascal.
- **Nvidia nvprof** for GPU programs written in **CUDA**.
- The Python standard library contains the **cProfile** and **profile** packages for profiling Python code. **cProfile** is a C implementation while **profile** is implemented in pure Python.
→ due to associated timer overhead in **profile**, you should prefer to use **cProfile**.

PROFILING: cProfile DEMO

Assume the following small program:

```
1 import time
2 import cProfile
3
4 def fast():
5     time.sleep(0.5)
6
7 def slow():
8     time.sleep(1)
9
10 def main():
11     for i in range(5):
12         fast()
13     for i in range(3):
14         slow()
15
16 if __name__ == "__main__":
17     cProfile.run('main()')
```

- The `cProfile.run('main()')` statement runs the profiler on the `main()` function.
- We can also profile the whole module from the command line without using `cProfile` explicitly in our code (the `-s` option sorts the output relative to total time in this example):

```
1 $ python -m cProfile -s tottime my_module.py
```

[See this link for a table of possible sort flags.](#)

PROFILING: cProfile DEMO

Profile output:

```
1  $ python -m cProfile -s tottime my_module.py
2      20 function calls in 5.507 seconds
3
4  Ordered by: internal time
5
6  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
7      8    5.508    0.688    5.508    0.688 {built-in method time.sleep}
8      1     0.000    0.000    5.508    5.508 my_module.py:10(main)
9      5     0.000    0.000    2.504    0.501 my_module.py:4(fast)
10     3     0.000    0.000    3.004    1.001 my_module.py:7(slow)
11     1     0.000    0.000    5.508    5.508 {built-in method builtins.exec}
12     1     0.000    0.000    5.508    5.508 my_module.py:2(<module>)
13     1     0.000    0.000    0.000    0.000 {method 'disable'}
```

- 20 function calls have been carried out.
- The total runtime of the program is about 5.5 seconds (by design for this example).
- The **Ordered by:** tells you the sort order and depends on the sort flag **-s**. Sorting is done alphabetically based on the function name by default.

PROFILING: cProfile DEMO

Profile output:

```
1 $ python -m cProfile -s tottime my_module.py
2      20 function calls in 5.507 seconds
3
4      Ordered by: internal time
5
6      ncalls  tottime  percall  cumtime  percall filename:lineno(function)
7           8    5.508    0.688    5.508    0.688 {built-in method time.sleep}
8           1    0.000    0.000    5.508    5.508 my_module.py:10(main)
9           5    0.000    0.000    2.504    0.501 my_module.py:4(fast)
10          3    0.000    0.000    3.004    1.001 my_module.py:7(slow)
11          1    0.000    0.000    5.508    5.508 {built-in method builtins.exec}
12          1    0.000    0.000    5.508    5.508 my_module.py:2(<module>)
13          1    0.000    0.000    0.000    0.000 {method 'disable'}
```

- Displayed timings are in **seconds**. Some functions execute in very short time and show 0.000 seconds. Their execution time is not exactly zero but below the displayed resolution.
- Use the **timeit** module if you need a more accurate time measurement!

PROFILING: cProfile DEMO

Profile output:

```
1 $ python -m cProfile -s tottime my_module.py
2      20 function calls in 5.507 seconds
3
4      Ordered by: internal time
5
6      ncalls  tottime  percall  cumtime  percall filename:lineno(function)
7           8    5.508    0.688    5.508    0.688 {built-in method time.sleep}
8           1    0.000    0.000    5.508    5.508 my_module.py:10(main)
9           5    0.000    0.000    2.504    0.501 my_module.py:4(fast)
10          3    0.000    0.000    3.004    1.001 my_module.py:7(slow)
11          1    0.000    0.000    5.508    5.508 {built-in method builtins.exec}
12          1    0.000    0.000    5.508    5.508 my_module.py:2(<module>)
13          1    0.000    0.000    0.000    0.000 {method 'disable'}
```

- **ncalls**: this column indicates the number of times a function has been called.
- **tottime**: the total time spent *in* that function. The measurement *does not include calls to sub-functions*.
- **percall**: the third column corresponds to the average time per function call computed by `tottime/ncalls`.
- **filename:lineno(function)**: provides data/information for the respective function.

PROFILING: cProfile DEMO

Profile output:

```
1 $ python -m cProfile -s tottime my_module.py
2      20 function calls in 5.507 seconds
3
4      Ordered by: internal time
5
6      ncalls  tottime  percall  cumtime  percall filename:lineno(function)
7           8    5.508    0.688    5.508    0.688 {built-in method time.sleep}
8           1     0.000    0.000    5.508    5.508 my_module.py:10(main)
9           5     0.000    0.000    2.504    0.501 my_module.py:4(fast)
10          3     0.000    0.000    3.004    1.001 my_module.py:7(slow)
11          1     0.000    0.000    5.508    5.508 {built-in method builtins.exec}
12          1     0.000    0.000    5.508    5.508 my_module.py:2(<module>)
13          1     0.000    0.000    0.000    0.000 {method 'disable'}
```

- The profile above contains *primitive* function calls only.
- A primitive function call means that *it was not induced via recursion*.
- **cumtime**: this column displays the *cumulative* time (including time spent in sub-functions) from invocation to exit. This figure is accurate even for recursive functions.
- **percall**: the fifth column corresponds to the quotient of cumtime divided by the number of *primitive* function calls.

PROFILING: cProfile DEMO

- Consider the *recursive* pdb_factorial/factorial.py code from the debugging discussion before.
- The profile for recursive functions will look slightly different:

```
1      103 function calls (4 primitive calls) in 0.000 seconds
2
3 Ordered by: internal time
4
5 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
6  100/1    0.000    0.000    0.000    0.000 factorial.py:5(factorial)
7      1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
8      1    0.000    0.000    0.000    0.000 <string>:1(<module>)
9      1    0.000    0.000    0.000    0.000 {method 'disable'}
```

- A total of 103 function calls have been made (including recursive calls). 4 out of the 103 are *primitive* calls (not recursive) → 99 are recursive.
- When there are recursive calls, the *ncalls* column displays the function calls with two numbers *100/1*. The first number is the *total number of calls* and the second number indicates the number of *primitive calls*.

PROFILING: cProfile OUTPUT

- For more flexible post-processing of your profiling data, you can *save the profile to a file*. This is useful when you create profiles for codes that run a while → *just printing data to stdout is often not a good idea!*
- You can achieve this by the following:

```
1 # when you call the profiler in your code
2 cProfile.run('main()', filename='profile')
```

or when you profile your module from the command line:

```
1 $ python -m cProfile -o profile my_module.py
```

- **Post-processing script:** for reuse and consistency you can write small scripts that load the profiling data for analysis. An example could look like this:

```
1 import pstats # module for processing data
2 from pstats import SortKey
3 p = pstats.Stats('profile')
4 p.sort_stats(
5     SortKey.CUMULATIVE).print_stats(10)
```

This sorts the profile by cumulative time of function calls and only prints the *10 most significant calls*. If you want to understand which functions take the most time, something like line 4 would do the job.

Interactive analysis: instead of writing a script, you can analyse the profile interactively using the `pstats` module on the command line (type `help` for a list of commands):

```
1 $ python -m pstats profile
```

cProfile EXERCISE

- Open the `cProfile_newton/newton.py` script from the lecture code handouts and implement the following three tasks (*note*: this is the same code we have used in lecture 11).
- Look for the **TODO** comment and implement three profilers:
 1. Profile the `main()` function using the *exact* Jacobian and save the data in a file called `"exact"`.
 2. Profile the `main()` function using the FD approximation with `eps=1.0e-1` and save it in a file called `"approx_coarse"`.
 3. Profile the `main()` function using the FD approximation with `eps=1.0e-8` and save it in a file called `"approx_fine"`.
- Analyse the three profiles with `python -m pstats`. Type the command `help` and look for the commands `read`, `strip`, `stats` and `sort`.
- *Where do you spend the most time? How does the function call count differ?*

PYTHON BYTECODE INSTRUCTIONS

- **Recall:** in Lecture 19 we were looking at how the Python interpreter executes *bytecode instructions*.
- We were using the Python disassembler (`dis`) to understand how Python compiles our code into a *low-level representation* which is what is *consumed* by the interpreter.
- A bytecode instruction takes the following form: `opcode oparg`.
 - Both opcode and oparg are encoded by *one byte* each (these bytes are stored in the code object) → *think of paper punch cards in the early days*.
 - Not all opcode's require an argument. *Even if they do not need it*, the (superfluous) oparg is still encoded and set to zero → `0x00` (*not economic from a code size point of view*).
 - The oparg's are indices into the `co_names` or `co_consts` tuples of the underlying code object.
 - Why does Python encode instructions like that?

PYTHON BYTECODE AND PERFORMANCE

- Let's think for a moment *what does it mean* when opcode and oparg are encoded by *one byte* each:
 - An *unsigned* byte can encode 256 bit permutations.
 - You can implement at most 256 different instructions in the Python interpreter. Is this enough? (*RISC-V* has about 50.)
 - The `co_names` and `co_consts` tuples can have at most 256 elements. *Is this true?*
 - What does the bullet above mean when you write code in Python → for example a function?
- When you target micro-optimizations you must know how Python bytecode and its instructions work. *The following are optimizations you do last, see the "How" bullet on the first slide about profiling → top-down* (optimizations on following slides are at the very bottom).

PYTHON BYTECODE AND PERFORMANCE

Range based iteration (iterators) and loop-counters:

- Iterators are implemented directly in C code in the Python interpreter.
- *Using iterators is faster than using a counter variable.* Consider the following equivalent loop structures:

```
1 def iterator(x):
2     for i in range(x):
3         pass # no meaningful work done
```

```
1 def counter(x):
2     k = 0
3     while k < x:
4         k += 1 # counter increment
```

```
1 2          0 LOAD_GLOBAL      0 (range)
2          2 LOAD_FAST        0 (x)
3          4 CALL_FUNCTION      1
4          6 GET_ITER
5      >>    8 FOR_ITER          4 (to 14)
6          10 STORE_FAST       1 (i)
7
8 3          12 JUMP_ABSOLUTE    8
9      >>    14 LOAD_CONST       0 (None)
10         16 RETURN_VALUE
```

```
1 2          0 LOAD_CONST      1 (0)
2          2 STORE_FAST       1 (k)
3
4 3      >>    4 LOAD_FAST        1 (k)
5          6 LOAD_FAST        0 (x)
6          8 COMPARE_OP       0 (<)
7         10 POP_JUMP_IF_FALSE 22
8
9 4          12 LOAD_FAST        1 (k)
10         14 LOAD_CONST       2 (1)
11         16 INPLACE_ADD
12         18 STORE_FAST       1 (k)
13         20 JUMP_ABSOLUTE    4
14      >>    22 LOAD_CONST       0 (None)
15         24 RETURN_VALUE
```

This code is about **3x faster** than the loop counter version!

(Benchmarked using Python 3.10.8 on an AMD Ryzen 7 Pro 4750U)

PYTHON BYTECODE AND PERFORMANCE

Pythonic and C-style loops:

- In Lecture 8 (Python data model) we looked at proper Pythonic loops and C-style loops.
- Consider again these two equivalent loops (*both are iterator based!*):

```
1 def pythonic(x):
2     for v in x:
3         pass # no meaningful work done
```

```
1 def c_style(x):
2     for i in range(len(x)):
3         v = x[i] # reference array element
```

```
1 2          0 LOAD_FAST          0 (x)
2          2 GET_ITER
3      >>    4 FOR_ITER            4 (to 10)
4          6 STORE_FAST          1 (v)
5
6 3          8 JUMP_ABSOLUTE       4
7      >>   10 LOAD_CONST          0 (None)
8          12 RETURN_VALUE
```

```
1 2          0 LOAD_GLOBAL         0 (range)
2          2 LOAD_GLOBAL         1 (len)
3          4 LOAD_FAST          0 (x)
4          6 CALL_FUNCTION         1
5          8 CALL_FUNCTION         1
6
7      >>   10 GET_ITER
8          12 FOR_ITER           12 (to 26)
9          14 STORE_FAST          1 (i)
10
11
12 3         16 LOAD_FAST          0 (x)
13         18 LOAD_FAST          1 (i)
14         20 BINARY_SUBSCR
15         22 STORE_FAST          2 (v)
16      >>   24 JUMP_ABSOLUTE       12
17         26 LOAD_CONST          0 (None)
18         28 RETURN_VALUE
```

This code is about **2.2x faster** than the C-style version!

(Benchmarked using Python 3.10.8 on an AMD Ryzen 7 Pro 4750U)

PYTHON BYTECODE AND PERFORMANCE

Dynamic name lookup:

- *Dynamic name lookup of an attribute outside the local scope is more expensive* because it must be loaded every time it is referenced (*it may have changed between two consecutive lookups!*).
- Consider the lookup of an attribute inside the math package, for example the sqrt function:

```
1 import math
2
3 def dynamic(x):
4     retval = 0.0
5     for v in x:
6         # global lookup
7         retval += math.sqrt(v)
8     return retval
```

```
1 import math
2
3 def cached(x):
4     # cache the name locally
5     sqrt = math.sqrt
6     retval = 0.0
7     for v in x:
8         # cached lookup
9         retval += sqrt(v)
10    return retval
```

PYTHON BYTECODE AND PERFORMANCE

Dynamic name lookup:

- Consider lookup of an attribute inside the math package:

```
1 def dynamic(x):
2     retval = 0.0
3     for v in x:
4         retval += math.sqrt(v)
5     return retval
```

```
1 def cached(x):
2     sqrt = math.sqrt
3     retval = 0.0
4     for v in x:
5         retval += sqrt(v)
6     return retval
```

1	2	0	LOAD_CONST	1	(0.0)	
2		2	STORE_FAST	1	(retval)	
3						
4	3	4	LOAD_FAST	0	(x)	
5		6	GET_ITER			
6	>>	8	FOR_ITER	18	(to 28)	
7		10	STORE_FAST	2	(v)	
8						
9	4	12	LOAD_FAST	1	(retval)	
10		14	LOAD_GLOBAL	0	(math)	
11		16	LOAD_METHOD	1	(sqrt)	
12		18	LOAD_FAST	2	(v)	
13		20	CALL_METHOD	1		
14		22	INPLACE_ADD			
15		24	STORE_FAST	1	(retval)	
16		26	JUMP_ABSOLUTE	8		
17						
18	5	>>	28	LOAD_FAST	1	(retval)
19		30	RETURN_VALUE			

1	2	0	LOAD_GLOBAL	0	(math)	
2		2	LOAD_ATTR	1	(sqrt)	
3		4	STORE_FAST	1	(sqrt)	
4						
5	3	6	LOAD_CONST	1	(0.0)	
6		8	STORE_FAST	2	(retval)	
7						
8	4	10	LOAD_FAST	0	(x)	
9		12	GET_ITER			
10	>>	14	FOR_ITER	16	(to 32)	
11		16	STORE_FAST	3	(v)	
12						
13	5	18	LOAD_FAST	2	(retval)	
14		20	LOAD_FAST	1	(sqrt)	
15		22	LOAD_FAST	3	(v)	
16		24	CALL_FUNCTION	1		
17		26	INPLACE_ADD			
18		28	STORE_FAST	2	(retval)	
19		30	JUMP_ABSOLUTE	14		
20						
21	6	>>	32	LOAD_FAST	2	(retval)
22		34	RETURN_VALUE			

PYTHON BYTECODE AND PERFORMANCE

Dynamic name lookup:

- Consider lookup of an attribute inside the math package:

Dynamic lookup

```
1 2          0 LOAD_CONST          1 (0.0)
2          2 STORE_FAST           1 (retval)
3
4 3          4 LOAD_FAST           0 (x)
5          6 GET_ITER
6      >>    8 FOR_ITER             18 (to 28)
7          10 STORE_FAST          2 (v)
8
9 4          12 LOAD_FAST           1 (retval)
10         14 LOAD_GLOBAL          0 (math)
11         16 LOAD_METHOD          1 (sqrt)
12         18 LOAD_FAST           2 (v)
13         20 CALL_METHOD          1
14         22 INPLACE_ADD
15         24 STORE_FAST          1 (retval)
16         26 JUMP_ABSOLUTE        8
17
18 5      >>    28 LOAD_FAST           1 (retval)
19          30 RETURN_VALUE
```

Locally cached

```
1 2          0 LOAD_GLOBAL          0 (math)
2          2 LOAD_ATTR            1 (sqrt)
3          4 STORE_FAST           1 (sqrt)
4
5 3          6 LOAD_CONST          1 (0.0)
6          8 STORE_FAST           2 (retval)
7
8 4          10 LOAD_FAST          0 (x)
9          12 GET_ITER
10      >>    14 FOR_ITER             16 (to 32)
11          16 STORE_FAST          3 (v)
12
13 5          18 LOAD_FAST          2 (retval)
14          20 LOAD_FAST           1 (sqrt)
15          22 LOAD_FAST           3 (v)
16          24 CALL_FUNCTION        1
17          26 INPLACE_ADD
18          28 STORE_FAST          2 (retval)
19          30 JUMP_ABSOLUTE       14
20
21 6      >>    32 LOAD_FAST          2 (retval)
22          34 RETURN_VALUE
```

Locally cached name is about **1.6x faster** than dynamic lookup *because faster instructions can be used!* This applies to any nested attributes in the form of **a.b.c.f()** for example.

(Benchmarked using Python 3.10.8 on an AMD Ryzen 7 Pro 4750U)

PYTHON BYTECODE AND PERFORMANCE

Dynamic name lookup:

- Consider lookup of an attribute inside the math package:

```
1 import math
2
3 def dynamic(x):
4     retval = 0.0
5     for v in x:
6         # global lookup
7         retval += math.sqrt(v)
8     return retval
```

```
1 import math
2
3 def cached(x):
4     # cache the name locally
5     sqrt = math.sqrt
6     retval = 0.0
7     for v in x:
8         # cached lookup
9         retval += sqrt(v)
10    return retval
```

- By now you should know that processing a large number of elements is faster when they can be processed as a *vector* (process many elements at once), which is exactly what NumPy does:

```
1 import numpy as np
2
3 def vectorized(x):
4     return np.sum(np.sqrt(x)) # NumPy will still need a loop, but on the C-level!
```

RECAP

- **Debugging:** how to locate *bugs* in (Python) code.
- **Profiling:** how to locate *performance bottlenecks* in (Python) code.
- **Performance:** understand bytecode limitations to identify performance issues in your code.

Further reading:

- Python pdb debugger: <https://docs.python.org/3/library/pdb.html>
- [Python pdb cheatsheet \(pdf\)](#)
- pytest support for pdb: <https://docs.pytest.org/en/6.2.x/usage.html#dropping-to-pdb-python-debugger-on-failures>
- [The iPython debugger \(ipdb\)](#)
- Python profilers: <https://docs.python.org/3/library/profile.html>