CS107 / AC207

SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 23

Tuesday, November 23rd 2021

Fabian Wermelinger
Harvard University

RECAP OF LAST TIME

- Hands-on exercises using `sqlite3` and `pandas`
  - Table joins in SQL
  - SQL interface in `pandas`
  - SQL-like operations in `pandas`

OUTLINE

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

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 that are "free" of bugs are extremely rare.
  - Humans are a main source of bugs.
  - Open source projects have usually less bugs than commercial/proprietary software.
  - Bugs can make your program crash (these are obvious) or make your code run but behave strangely (these are the hard ones 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 instructions or 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 your code for the examination of local 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). E.g., 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 allow you interact with your program as it executes. Examples are `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 for example:

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

- *Online debugging*. Attach to a running process (could be in a production environment) to isolate a problem.
- *Isolation by bisection*. Narrowing down to smaller sections until you can see the bug. Examples are divide and conquer (e.g. `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. You are also 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.
- Yet other "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 cache memory architectures.

DEBUGGING PROCESS

- You have "to sense" that there is a bug in your program. This is easy when it crashes but can be hard when the bug is more silent and shows up randomly. You must have a belief for some expected behavior of your program and by debugging you prove that those beliefs are indeed true. If your program finds them to be false, you may find it surprising but you also found a clue for a bug.
- Once you know there is a bug, you must find a way to *reproduce* it. This is 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.

DEBUGGER

A *debugger* is a program that allows to step through your code. The GNU debugger `gdb` is the classic text based debugger. In `python` we use `pdb`.

Main debugger operations:

- ***Stepping through the source code:*** this can be done on a per line basis or per instruction basis. In order to stop at a particular point the notion of *breakpoint*s is used. 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 you moving to any stack frame that is currently on the stack. This allows to inspect variable values in the caller frame as well as generating a *backtrace* (`traceback` objects in `python`).

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" in your code can be source line numbers, a code address or function entry point.
- You can track breakpoints as well and get information such as how often the breakpoint has been hit or you could say I want to stop at this breakpoint only after it has been hit n times. Other conditions may be imposed on breakpoints additionally that depend on the debugger used.
- 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 through

python -m pdb your_script.py

Examples:

- ***Running from within the interpreter:*** You can run the debugger on specific code from a module:

```
>>> import pdb
>>> import your_module
>>> pdb.run('your_module.test()')
```

- ***Inserting a breakpoint in your code:*** the classic way:

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

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

```
# some code
breakpoint()
# more code
```

python **DEBUGGER**: pdb

- Simply running the following will drop us into the `pdb` shell:

```
$ python -m pdb factorial.py
< /home/fabs/CS107/lecture23/pdb_factorial/factorial.py(5)<module>()
-< import numpy as np
:(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.)

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 can be downloaded from the lecture website: https://harvard-iacs.github.io/2021-CS107/lectures/lecture23
- We can start the debugger with the following `python` command:

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

Alternatively, we can use `ipdb` with `ipython` for color highlighting:

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

- We want to investigate the following:
  - List the source code at the start of the debugging session.
  - Set a breakpoint at the statement `return 1` in the factorial function. When done list all active breakpoints.
  - Continue running until you reached the breakpoint.
  - Print a backtrace and study the recursion of `factorial()`. Go `up` and `down` the call stack.
  - 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 (most of them).
- The debugger is an extremely powerful tool and you should integrate it in your development workflow. We could only touch 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.
- ***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. In addition, you will not be able to tackle 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. Your approach for optimization is *top-down*: from simple (low time investment) to difficult (high time investment; you must assess if you really need these optimizations).

  Before you start with optimizations you must know where to start and the *profiler is the 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*.
- The bottlenecks you identify by profiling your code are your first targets for optimization. Removing them often leads 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. Due to associated timer overhead, you should prefer to use `cProfile`.

PROFILING: `cProfile` DEMO

- We are given 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:

```
$ python -m cProfile -s tottime my_module.py
         20 function calls in 5.507 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        8    5.508    0.688    5.508    0.688 {built-in method time.sleep}
        1    0.000    0.000    5.508    5.508 my_module.py:10(main)
        5    0.000    0.000    2.504    0.501 my_module.py:4(fast)
        3    0.000    0.000    3.004    1.001 my_module.py:7(slow)
        1    0.000    0.000    5.508    5.508 {built-in method builtins.exec}
        1    0.000    0.000    5.508    5.508 my_module.py:2(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

- 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`. By default sorting is done alphabetically based on the function name.

PROFILING: `cProfile` DEMO

Profile output:

```
$ python -m cProfile -s tottime my_module.py
         20 function calls in 5.507 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        8    5.508    0.688    5.508    0.688 {built-in method time.sleep}
        1    0.000    0.000    5.508    5.508 my_module.py:10(main)
        5    0.000    0.000    2.504    0.501 my_module.py:4(fast)
        3    0.000    0.000    3.004    1.001 my_module.py:7(slow)
        1    0.000    0.000    5.508    5.508 {built-in method builtins.exec}
        1    0.000    0.000    5.508    5.508 my_module.py:2(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

- 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:

```
$ python -m cProfile -s tottime my_module.py
         20 function calls in 5.507 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        8    5.508    0.688    5.508    0.688 {built-in method time.sleep}
        1    0.000    0.000    5.508    5.508 my_module.py:10(main)
        5    0.000    0.000    2.504    0.501 my_module.py:4(fast)
        3    0.000    0.000    3.004    1.001 my_module.py:7(slow)
        1    0.000    0.000    5.508    5.508 {built-in method builtins.exec}
        1    0.000    0.000    5.508    5.508 my_module.py:2(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

- `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 function call computed by `tottime/ncalls` .
- `filename:lineno(function)` : Provides data/information for the respective function.

PROFILING: `cProfile` DEMO

Profile output:

```
$ python -m cProfile -s tottime my_module.py
         20 function calls in 5.507 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        8    5.508    0.688    5.508    0.688 {built-in method time.sleep}
        1    0.000    0.000    5.508    5.508 my_module.py:10(main)
        5    0.000    0.000    2.504    0.501 my_module.py:4(fast)
        3    0.000    0.000    3.004    1.001 my_module.py:7(slow)
        1    0.000    0.000    5.508    5.508 {built-in method builtins.exec}
        1    0.000    0.000    5.508    5.508 my_module.py:2(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

- 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 `factorial.py` code from the debugging discussion before. See https://harvard-iacs.github.io/2021-CS107/lectures/lecture23.
- 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' of '_lsprof.Profiler' objects}
```

- A total of 103 function calls have been made (including recursive calls). 4 out of the 103 are *primitive* calls (not recursive).
- When there are recursive calls, the `ncalls` column displays the function calls by 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` STATS

- 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.
- You can achieve this by the following:

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

or when you profile your module from the command line:

```
$ python -m cProfile -o my_stats my_module.py # when you call the profiler from the command line
```

- ***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
2 from pstats import SortKey
3 p = pstats.Stats('my_stats')
4 p.sort_stats(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 what functions are taking the most time, the above `line 4` is what you would use.

- ***Interactive analysis:*** you can investigate your profiling data *interactively* using the `pstats` module (type `help` for possible analysis commands):

```
$ python -m pstats my_stats # enter the pstats interpreter for profiling analysis
```

PROFILING: `cProfile` HANDS-ON (15MIN)

- Download the `netwon.py` script from the lecture webpage: https://harvard-iacs.github.io/2021-CS107/lectures/lecture23 (***note:*** this is the code we have used in lecture 9).
- In the body of `if __name__ == "__main__":`, 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 "`fd_1.0e-1`".
  3. Profile the `main()` function using the FD approximation with `eps=1.0e-8` and save it in a file called "`fd_1.0e-8`".
- Analyse the three profiles using the interactive `python` tool `python -m pstats`. Type the command `help` and look for the commands `read`, `strip`, `stats` and `sort`.
- What do you observe? 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` translates our code into a low-level representation which is consumed by the interpreter.
- A bytecode instruction takes the following form: `opcode oparg`.
  - Both `opcode` and `oparg` are encoded by *one byte* each in the code object's raw byte string (think of paper punch cards in the early days).
  - Not all `opcode`'s require an argument. *Even it they do not need it*, the (superfluous) `oparg` is still encoded (and set to `0x00`).
  - 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 us think for a moment what does it mean when `opcode` and `oparg` are encoded by *one byte* each:
  - An *unsigned* byte can encode 255 bit permutations.
  - You can implement at most 255 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 255 elements. Is this true?
  - What does the latter bullet mean when you write code, for example a function?
- When you target micro-optimizations (these are the optimizations you do last, see the "***How***" bullet on the first slide about profiling) you must know how `python` bytecode and its instructions work.

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 loop structures:

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

```
 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
```

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

(Benchmarked using `python` 3.9.7 on an AMD Ryzen 7 Pro 4750U)

```
1  def counter(x):
2      k = 0
3      while k < x:
4          k += 1 # loop counter must be incremented
```

```
 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
```

python BYTECODE AND PERFORMANCE

Pythonic and C-style loops

- In Lecture 7 (python data model) we were looking at (proper) pythonic loops and C-style loops.
- Reconsider these two implementations (*both are iterator based*):

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

```
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
```

This code is about ***1.9x faster*** than the C-style version!

(Benchmarked using python 3.9.7 on an AMD Ryzen 7 Pro 4750U)

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

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

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 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         retval += math.sqrt(v) # global lookup
7     return retval
```

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

```
1  def cached(x):
2      sqrt = math.sqrt # cache the name locally
3      retval = 0.0
4      for v in x:
5          retval += sqrt(v) # cached lookup
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 name is about ***1.6x faster*** than dynamic lookup (`LOAD_FAST`)! This applies to any nested attributes in the form of `a.b.c.f()`.

(Benchmarked using `python` 3.9.7 on an AMD Ryzen 7 Pro 4750U)

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
```

python BYTECODE AND PERFORMANCE

Dynamic name lookup

- Consider lookup of an attribute inside the `math` package:

```python
import math

def dynamic(x):
    retval = 0.0
    for v in x:
        retval += math.sqrt(v) # global lookup
    return retval
```

```python
import math

def cached(x):
    sqrt = math.sqrt # cache the name locally
    retval = 0.0
    for v in x:
        retval += sqrt(v) # cached lookup
    return retval
```

- Of course by now you know that if your data allows it, the correct way to do this is:

```python
import numpy as np

def vectorized(x):
    return np.sum(np.sqrt(x))
```

RECAP

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