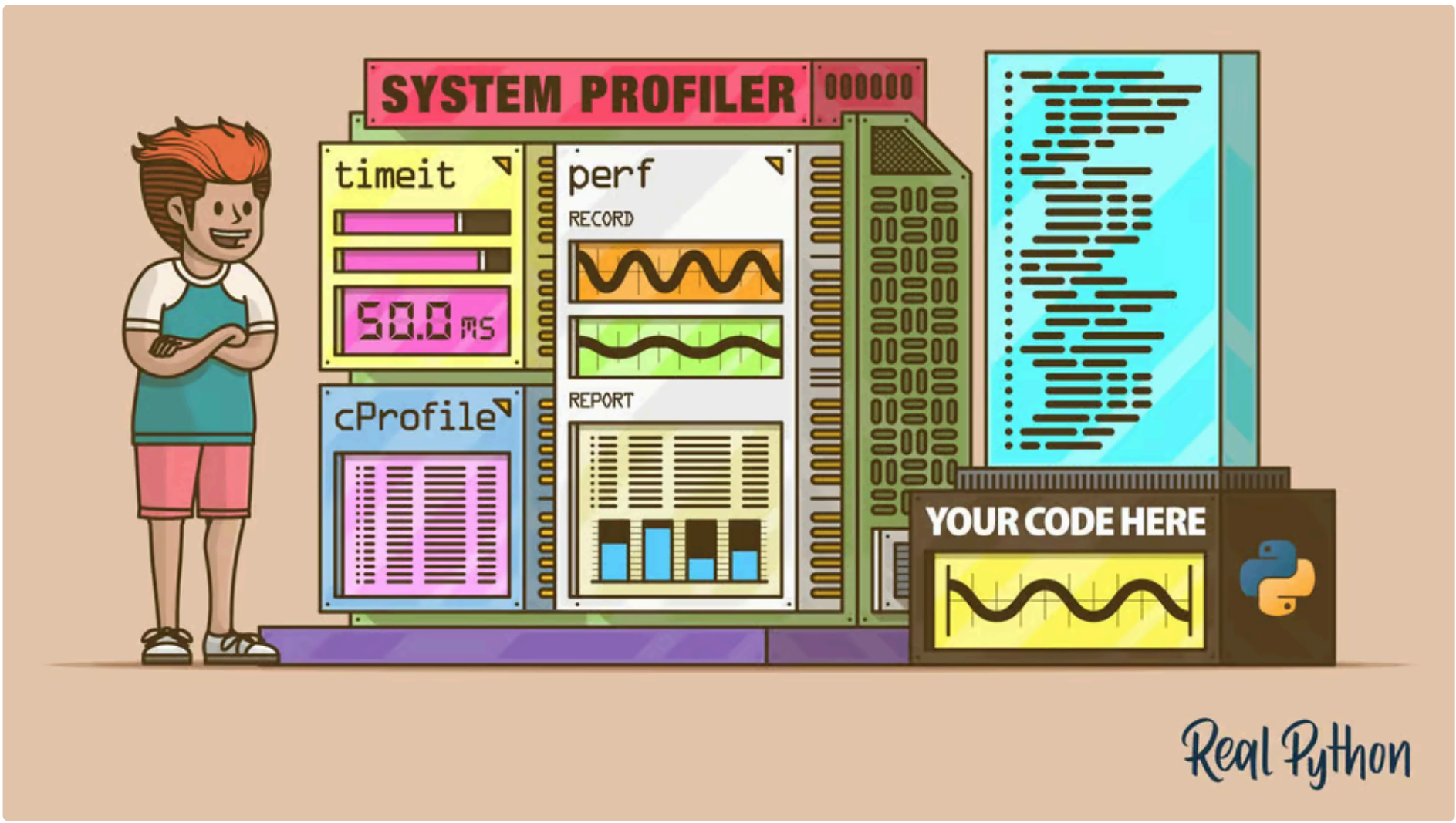

Profiling in Python: How to Find Performance Bottlenecks

by Bartosz Zaczyński Jul 10, 2023 0 Comments intermediate tools

Mark as Completed Share

Table of Contents

Do you want to optimize the performance of your Python program to make it run faster or consume less memory? Before diving into any performance tuning, you should strongly consider using a technique called **software profiling**. It may help you answer whether optimizing the code is necessary and, if so, which parts of the code you should focus on.

Sometimes, the return on investment in performance optimizations just isn't worth the effort. If you only run your code once or twice, or if it takes longer to improve the code than execute it, then what's the point?

When it comes to improving the quality of your code, you'll probably optimize for performance as a final step, if you d
Often, your code will become speedier and more memory efficient thanks to other changes that you make. When in do
through this short checklist to figure out whether to work on performance:

1. Testing: Have you tested your code to prove that it works as expected and without errors?
2. Refactoring: Does your code need some cleanup to become more maintainable and Pythonic?
3. Profiling: Have you identified the most inefficient parts of your code?

Only when all the above items check out should you consider optimizing for performance. It's usually more important that your code runs correctly according to the business requirements and that other team members can understand it than that it's the most efficient solution.

The actual time-saver might be elsewhere. For example, having the ability to quickly extend your code with new features before your competitors will make a real impact. That's especially true when the performance bottleneck lies not in the underlying code's execution time but in network communication. Making Python run faster won't win you anything in that case, but it'll likely increase the code's complexity.

Finally, your code will often become faster as a result of fixing the bugs and refactoring. One of the creators of Erlang once said:

> Make it work, then make it beautiful, then if you really, really have to, make it fast. 90 percent of the time, if you make it beautiful, it will already be fast. So really, just make it beautiful! (Source)
>
> — *Joe Armstrong*

As a rule of thumb, anytime you're considering optimization, you should profile your code first to identify which bottlenecks to address. Otherwise, you may find yourself chasing the wrong rabbit. Because of the Pareto principle or the 80/20 rule, which applies to a surprisingly wide range of areas in life, optimizing just 20 percent of your code will often yield 80 percent of the benefits!

But without having factual data from a profiler tool, you won't know for sure which parts of the code are worth improving. It's too easy to make false assumptions.

So, what's software profiling, and how do you profile programs written in Python?

How to Find Performance Bottlenecks in Your Python Code Through Profiling

Software profiling is the process of collecting and analyzing various metrics of a running program to identify performance bottlenecks known as **hot spots**. These hot spots can happen due to a number of reasons, including excessive memory use, inefficient CPU utilization, or a suboptimal data layout, which will result in frequent cache misses that increase latency.

> **Note:** A performance profiler is a valuable tool for identifying hot spots in existing code, but it won't tell you how to write efficient code from the start.
>
> It's often the choice of the underlying algorithm or data structure that can make the biggest difference. Even when you throw the most advanced hardware available on the market at some computational problem, an algorithm with a poor time or space complexity may never finish in a reasonable time.

When profiling, it's important that you perform **dynamic analysis** by executing your code and collecting real-world data rather than relying on static code review. Because dynamic analysis often entails running a slow piece of software over and over again, you should start by feeding small amounts of input data to your algorithm if possible. This will limit the amount of time that you spend waiting for results on each iteration.

Once you have your code running, you can use one of the many Python profilers available. There are many kinds of profilers out there, which can make your head spin. Ultimately, you should know how to pick the right tool for the job. Over the next few sections, you'll get a quick tour of the most popular Python profiling tools and concepts:

- **Timers** like the `time` and `timeit` standard library modules, or the `codetiming` third-party package
- **Deterministic profilers** like `profile`, `cProfile`, and line_profiler
- **Statistical profilers** like Pyinstrument and the Linux `perf` profiler

Fasten your seatbelt because you're about to get a crash course in Python's performance profiling!

`time`: Measure the Execution Time

In Python, the most basic form of profiling involves measuring the code execution time by calling one of the timer functions from the `time` module:

Python

```
>>> import time

>>> def sleeper():
...     time.sleep(1.75)
...

>>> def spinlock():
...     for _ in range(100_000_000):
...         pass
...

>>> for function in sleeper, spinlock:
...     t1 = time.perf_counter(), time.process_time()
...     function()
...     t2 = time.perf_counter(), time.process_time()
...     print(f"{function.__name__}()")
...     print(f" Real time: {t2[0] - t1[0]:.2f} seconds")
...     print(f" CPU time: {t2[1] - t1[1]:.2f} seconds")
...     print()
...
sleeper()
 Real time: 1.75 seconds
 CPU time: 0.00 seconds

spinlock()
 Real time: 1.77 seconds
 CPU time: 1.77 seconds
```

You first define two test functions, `sleeper()` and `spinlock()`. The first function asks your operating system's task scheduler to suspend the current thread of execution for about 1.75 seconds. During this time, the function remains dormant without occupying your computer's CPU, allowing other threads or programs to run. In contrast, the second function performs a form of busy waiting by wasting CPU cycles without doing any useful work.

Later, you call both of your test functions. Before and after each invocation, you check the current time with `time.perf_counter()` to obtain the elapsed real time, or wall-clock time, and `time.process_time()` to get the CPU time. These will tell you how long your functions took to execute and how much of that time they spent on the processor. If a function waits for another thread or an I/O operation to finish, then it won't use any CPU time.

> **Note:** The performance of a computer program is typically limited by the available processing power, memory amount, input and output operations, and program latency.
>
> If a given task predominantly does a lot of computation, then the processor's speed will determine how long it'll take to finish. Such a task is CPU-bound. You can sometimes run such tasks in parallel on multiple CPU cores simultaneously to reduce the overall computation time.
>
> On the other hand, an I/O-bound task spends most of its time waiting for data to arrive from a disk, a database, or a network. Such tasks can benefit from using faster I/O channels or running them concurrently as well as asynchronously.

Calling the timer functions directly to profile your Python code can be cumbersome, especially when you need to time many code snippets. To make working with them more convenient, you may try a third-party package like `codetiming`, which wraps those functions in classes, context managers, and decorators.

The `time` module is versatile and quick to set up, making it suitable for temporary checks. It'll give you a faithful impression of runtime in real-world conditions, taking into account factors like the current system load. However, if you're more interested in getting lab-condition timing for your code snippets, with less external influence, then `time` may not be the best choice.

In cases like this, you should ideally disable Python's garbage collector and repeat the exercise several times to minimize the influence of external factors, such as the interpreter's start-up time or the system noise. Next up, you'll learn how to automate these steps with the built-in `timeit` module.

`timeit`: Benchmark Short Code Snippets

To keep you from falling into common pitfalls, Python comes with a handy utility module called `timeit`, which takes care of most profiling complexities. This means accounting for factors such as system load, garbage collection, or other processes running concurrently that might skew your timing results. The `timeit` module helps to mitigate these factors, providing a more accurate measure of code execution time.

You can use it either programmatically or through the command-line interface, whichever you prefer. Here's an example of timing a recursive function that calculates the nth element of the Fibonacci sequence:

Python

```
>>> from timeit import timeit

>>> def fib(n):
...     return n if n < 2 else fib(n - 2) + fib(n - 1)
...

>>> iterations = 100
>>> total_time = timeit("fib(30)", number=iterations, globals=globals())

>>> f"Average time is {total_time / iterations:.2f} seconds"
'Average time is 0.15 seconds'
```

You ask `timeit` to measure the total execution time of `fib(30)` repeated one hundred times in a loop. Then, you compute the average time by dividing the result by the number of iterations.

This repetition minimizes the effects of system noise on the timing. By repeating the same function call multiple times, you can average out random fluctuations in execution time that may come from other processes running on your computer. You can start with the default five iterations by leaving out the `number` parameter. If your function is very fast or slow, then adjust that number as needed to get an accurate measure.

When you run `timeit` in the command line or use the `%timeit` magic command in a Jupyter Notebook, then it'll show you the best runtime of the code snippet that you've given it:

Shell

```
$ SETUP_CODE="def fib(n): return n if n < 2 else fib(n - 2) + fib(n - 1)"
$ python3 -m timeit -s "$SETUP_CODE" -r 100 "fib(30)"
2 loops, best of 100: 158 msec per loop
```

If your code under test requires a one-time setup, then you can optionally instruct `timeit` to run it once before entering the loop. Often, you'll only be interested in seeing the **best result**, which is closest to the truth, while the longer runs indicate disturbance from random noise.

While `timeit` allows you to benchmark a particular code snippet by measuring the execution time, it falls short when you want to collect more detailed metrics to find bottlenecks. Fortunately, Python ships with a more sophisticated profiler tool that you'll explore next.

`cProfile`: Collect Detailed Runtime Statistics

The standard library in Python includes a pure-Python `profile` module and an equivalent extension implemented in C with the same interface, called `cProfile`. You're generally advised to use the much more performant `cProfile`, which has significantly less overhead.

Conversely, you should only fall back to using `profile` when `cProfile` isn't available on your system. Also, you may sometimes prefer `profile` when you want to extend it using Python.

Both modules provide a deterministic profiler, which can help you answer questions like how many times a particular function was called or how much total time was spent inside that function. A deterministic profiler can give you reproducible results under the same conditions because it traces *all* function calls in your program.

> **Note:** Internally, `cProfile` takes advantage of `sys.setprofile()` to register an event listener that will be notified whenever one of your functions gets called. For even more fine-grain monitoring, you can call `sys.settrace()`, which lets you track a few types of events at the level of individual lines of code. Alternatively, you may find a third-party tool like line_profiler more convenient.

You can use `cProfile` against your whole program in the command line or profile a narrow code fragment programmatically, like in the following example:

Python

```
>>> from cProfile import Profile
>>> from pstats import SortKey, Stats

>>> def fib(n):
...     return n if n < 2 else fib(n - 2) + fib(n - 1)
...

>>> with Profile() as profile:
...     print(f"{fib(35) = }")
...     (
...         Stats(profile)
...         .strip_dirs()
...         .sort_stats(SortKey.CALLS)
...         .print_stats()
...     )
...
fib(35) = 9227465
         29860712 function calls (10 primitive calls) in 9.624 seconds

   Ordered by: call count

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
29860703/1    9.624    0.000    9.624    9.624 <stdin>:1(fib)
        1    0.000    0.000    0.000    0.000 pstats.py:118(init)
        1    0.000    0.000    0.000    0.000 pstats.py:137(load_stats)
        1    0.000    0.000    0.000    0.000 pstats.py:108(__init__)
        1    0.000    0.000    0.000    0.000 cProfile.py:50(create_stats)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.hasattr}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.print}


<pstats.Stats object at 0x7fbbd6d47610>
```

The output is quite verbose, but it tells you that your program took over nine and a half seconds to finish while making exactly 29,860,712 function calls. Only ten of them were **primitive** or non-recursive calls, including just one non-recursive call to `fib()`. All the others were calls from `fib()` to itself.

> **Note:** You use a separate module called `pstats` to format, sort, and print the collected runtime statistics. This module becomes even more useful when you dump the statistics into a binary file. In that case, you can run `pstats` as an interactive program to browse the file's content in the terminal.

Calling `fib()` with a relatively small input value of `35` results in nearly thirty million recursive calls! The profiler reports that this high number of function calls coincides with an area of code where the program spends the majority of its time. When you investigate further, you'll find that most of these recursive calls are redundant because they keep calculating the same values over and over again.

As a quick optimization, you may use memoization to cache the intermediate results. That way, you'll calculate each Fibonacci number once and reuse the cached result for subsequent calls to `fib()`.

You wouldn't necessarily know this without profiling your code first. However, profiling a program adds noticeable **runtime overhead** because of the extra instrumentation code that needs to register and keep track of certain events. In some cases, this may prohibit you from using a profiler tool, especially in a production environment already suffering from poor performance.

Now, you'll take a look at a popular technique that helps combat the challenges of a profiler's overhead.

Pyinstrument: Take Snapshots of the Call Stack

To lower the profiler's overhead, you can use statistical profiling and only collect metrics once in a while. This works by taking a snapshot of the running program's state at specified intervals. Each time, the profiler records a **sample** consisting of the entire call stack from the currently executing function all the way up to the top ancestor in the call hierarchy.

While a statistical profiler won't provide the same level of detail as a deterministic one, it frees you from some shortcomings.

Because a **deterministic profiler** monitors all the function calls across your application, it has considerable overhead and produces a lot of noise in the report. Moreover, this overhead isn't uniform because it depends on the number of actual function calls, leading to inaccurate and distorted results.

In contrast, a **statistical profiler** will filter out insignificant calls that don't affect the overall performance, and its overhead is uniform and adjustable. Depending on your sampling rate, functions that return quickly may not even show up in the report.

To use a statistical profiler in Python, you'll need to install a third-party tool like Pyinstrument or py-spy. Some of them are better than others depending on the use case. For example, Pyinstrument can't handle code that runs in multiple threads or calls functions implemented in C extension modules, such as NumPy or pandas.

To experience the power of Pyinstrument, it's best to try an example comprising more than one function. Here's a straightforward implementation of a Monte Carlo method for estimating the value of pi by means of simulation and geometric probability:

Python

```
>>> from random import uniform

>>> def estimate_pi(n):
...     return 4 * sum(hits(point()) for _ in range(n)) / n
...

>>> def hits(point):
...     return abs(point) <= 1
...

>>> def point():
...     return complex(uniform(0, 1), uniform(0, 1))
...

>>> for exponent in range(1, 8):
...     n = 10 ** exponent
...     estimates = [estimate_pi(n) for _ in range(5)]
...     print(f"{n = :<10,} {estimates}")
...
n = 10         [2.8, 2.8, 3.6, 4.0, 3.6]
n = 100        [3.04, 3.04, 3.2, 2.96, 3.28]
n = 1,000      [3.136, 3.144, 3.128, 3.14, 3.12]
n = 10,000     [3.1448, 3.1408, 3.1448, 3.1456, 3.1664]
n = 100,000    [3.14872, 3.13736, 3.14532, 3.14668, 3.13988]
n = 1,000,000  [3.140528, 3.14078, 3.14054, 3.140972, 3.141344]
n = 10,000,000 [3.1414564, 3.1427292, 3.1402788, 3.1420736, 3.1407568]
```

The more iterations in a simulation, the better the approximation of pi. Here, you use complex numbers as a convenient way to represent two-dimensional points chosen randomly using a uniform probability distribution. Each `point()` lies within a unit square. By counting the points that hit a quadrant of a circle enclosed in that square, you can estimate the ratio of their areas, which is π/4.

Like other tools, Pyinstrument lets you either run your entire Python script in the command line or profile a specific code block:

Python

```
>>> from pyinstrument import Profiler
>>> with Profiler(interval=0.1) as profiler:
...     estimate_pi(n=10_000_000)
...

3.142216
>>> profiler.print()

  _     ._   __/__   _ _  _  _ _/_   Recorded: 11:17:13  Samples:  201
 /_//_/// /_\ / //_// / //_'/ //     Duration: 20.150    CPU time: 20.149
/   _/                      v4.5.0

Program:

20.100 <module>  <stdin>:1
└─ 20.100 estimate_pi  <stdin>:1
      [12 frames hidden]  <stdin>, random, <built-in>
         19.200 <genexpr>  <stdin>:2
         ├─ 12.900 point  <stdin>:1
         │  ├─ 8.000 Random.uniform  random.py:520
         │  │  ├─ 6.400 [self]  None
         │  │  └─ 4.900 [self]  None
         ├─ 4.800 [self]  None

>>> profiler.open_in_browser()
```

In this case, by setting the `interval` parameter, you tell Pyinstrument to take a snapshot every one-tenth of a second or one hundred milliseconds. Then, you estimate the value of pi using a Monte Carlo method with ten million iterations.

A frequency of `0.1` seconds is pretty low, so it'll have relatively little overhead at runtime, but you'll end up with coarse data in the report. By fine-tuning the sampling interval, you can change the amount of detail that will appear in the report. The more frequent the sampling rate, the more data the profiler will collect, at the cost of higher overhead.

Next, you print the report depicting the functions' call hierarchy. Right off the bat, this tree view is more useful than the default report from `cProfile` because it can show you the context of a function call. After all, the same function may be called from many places and for different purposes.

However, some of the stack frames were hidden to make the report more readable. If you wanted to reveal them, then you could open an interactive report in your default web browser by calling `profiler.open_in_browser()`, as you did in the last line.

The report tells you that `estimate_pi()` spends most of its time in the generator expression. When you drill down, you notice that the `point()` function is the bottleneck. Unfortunately, it seems there's not much you can do about it because it merely calls the library function `random.uniform()`, which takes a significant amount of time to execute.

But, when you look closer at the documentation of `random.uniform()` or its implementation, then you'll find that it's a pure-Python function. Such functions can be orders of magnitude slower than built-in functions implemented in C.

In this case, you can safely replace the call to `uniform(0, 1)` with `random()` because both functions are mathematically equivalent for these specific input values. When you do, you'll observe an improvement in computation time—by as much as a whopping 40 percent!

For the ultimate performance analysis experience, though, you'll want to use the Linux `perf` tool. It has marginal overhead while allowing you to see a much bigger picture.

`perf`: Count Hardware and System Events on Linux

Starting with the Python 3.12 release, the interpreter now supports the Linux `perf` profiler, which can access hardware performance counters on some computer architectures. It's one of the most advanced and powerful profilers in existence. It can provide detailed information about the entire stack, including hardware events, system calls, library code, and more. Additionally, its overhead is small and adjustable.

> **Note:** The `perf` profiler is only available on Linux, as it's baked into the operating system's kernel. However, you'll usually need to install an extra system package to bring the accompanying command-line utility tool for accessing the underlying kernel subsystem.

Bear in mind that using `perf` requires a basic understanding of how the hardware and operating system fit together. You'll also need to be able to build and install Python from source code using special compiler flags for best results.

> **Note:** Check out the dedicated Python 3.12 preview tutorial to get step-by-step instructions on getting `perf` ready to profile your Python code.

Once everything is set up, meaning that you're on a Linux distribution with the `perf` tool installed and Python 3.12 compiled from source, you can start collecting and analyzing performance data. But before you can do that, you must write a short benchmark script that you'll run through `perf`:

Python

```python
# profile_perf.py

from concurrent.futures import ThreadPoolExecutor

def find_divisors(n):
    return [i for i in range(1, n + 1) if n % i == 0]

def slow_function():
    print("Slow thread started")
    try:
        return find_divisors(100_000_000)
    finally:
        print("Slow thread ended")

def fast_function():
    print("Fast thread started")
    try:
        return find_divisors(50_000_000)
    finally:
        print("Fast thread ended")

def main():
    with ThreadPoolExecutor(max_workers=2) as pool:
        pool.submit(slow_function)
        pool.submit(fast_function)

    print("Main thread ended")

if __name__ == "__main__":
    main()
```

Here, you use a thread pool from the `concurrent.futures` module to execute two functions concurrently. Both tasks simulate heavy computations by finding the integer divisors of a big number. While the first function is designed to be intentionally slow, the other one should be noticeably faster, which you can show by running your script:

Shell

```
$ python profile_perf.py
Slow thread started
Fast thread started
Fast thread ended
Slow thread ended
Main thread ended
```

Even though you scheduled the slow function before the fast one, the second one ends up finishing first.

With your script in place, you can use `perf` to make a performance profile. The most common way of using this tool consists of two steps:

1. Recording stack trace samples
2. Printing or visualizing the report

If you followed the Python 3.12 preview tutorial mentioned earlier, and you have a custom Python build installed in the `python-custom-build/` folder in your home directory, then you can issue the following command:

Shell

```
$ sudo perf record -g -F 999 \
                   "$HOME/python-custom-build/bin/python3" -X perf \
                   profile_perf.py
```

This will run your script through the custom Python build with a special stack trampoline mode enabled (`-X perf`). The `perf` tool will interrupt your script roughly 999 times per second (`-F 999`) to take a snapshot of the function call stack (`-g`). Note that you need to access superuser privileges by running the command with `sudo` to collect sensitive data from your CPU counters.