Python support for the Linux perf profiler

Release 3.12.8

Guido van Rossum and the Python development team

January 17, 2025

Python Software Foundation Email: docs@python.org

Contents

1	How to enable perf profiling support	4
2	How to obtain the best results	4
Inc	dex	6

author

Pablo Galindo

The Linux perf profiler is a very powerful tool that allows you to profile and obtain information about the performance of your application. perf also has a very vibrant ecosystem of tools that aid with the analysis of the data that it produces.

The main problem with using the perf profiler with Python applications is that perf only gets information about native symbols, that is, the names of functions and procedures written in C. This means that the names and file names of Python functions in your code will not appear in the output of perf.

Since Python 3.12, the interpreter can run in a special mode that allows Python functions to appear in the output of the perf profiler. When this mode is enabled, the interpreter will interpose a small piece of code compiled on the fly before the execution of every Python function and it will teach perf the relationship between this piece of code and the associated Python function using perf map files.

1 Note

Support for the perf profiler is currently only available for Linux on select architectures. Check the output of the configure build step or check the output of python -m sysconfig | grep HAVE_PERF_TRAMPOLINE to see if your system is supported.

For example, consider the following script:

```
def foo(n):
    result = 0
    for _ in range(n):
        result += 1
```

(continues on next page)

(continued from previous page)

```
return result

def bar(n):
    foo(n)

def baz(n):
    bar(n)

if __name__ == "__main__":
    baz(1000000)
```

We can run perf to sample CPU stack traces at 9999 hertz:

```
$ perf record -F 9999 -g -o perf.data python my_script.py
```

Then we can use perf report to analyze the data:

```
$ perf report --stdio -n -g
           Self Samples Command Shared Object Symbol
# Children
91.08% 0.00% 0 python.exe python.exe [.] _start
         ---_start
            --90.71%--__libc_start_main
                  Py_BytesMain
                  |--56.88%--pymain_run_python.constprop.0
                           |--56.13%--_PyRun_AnyFileObject
                                   _PyRun_SimpleFileObject
                                   |--55.02%--run_mod
                                  --54.65%--PyEval_EvalCode
                                                    _PyEval_
→EvalFrameDefault
                          1
                                                    PyObject_
Vectorcall
                                                    _PyEval_Vector
                                                    _PyEval_
→EvalFrameDefault
                          PyObject_
→Vectorcall
                                                    _PyEval_Vector
                                                    _PyEval_
→EvalFrameDefault
                                                    PyObject_
→Vectorcall
                                                    _PyEval_Vector
                                                    |--51.67%--_
→PyEval_EvalFrameDefault
                                                      (continues on next page)
```

(continued from previous page)

As you can see, the Python functions are not shown in the output, only <code>PyEval_EvalFrameDefault</code> (the function that evaluates the Python bytecode) shows up. Unfortunately that's not very useful because all Python functions use the same C function to evaluate bytecode so we cannot know which Python function corresponds to which bytecode-evaluating function.

Instead, if we run the same experiment with perf support enabled we get:

```
$ perf report --stdio -n -g
                    Samples Command Shared Object
# Children
            Self
                                                         Symbol
# ...... ... ... ... .... .... ......
90.58%
           0.36%
                    1 python.exe python.exe [.] _start
          ---_start
             --89.86%--__libc_start_main
                    Py_BytesMain
                    |--55.43%--pymain_run_python.constprop.0
                             |--54.71%--_PyRun_AnyFileObject
                                      _PyRun_SimpleFileObject
                                      |--53.62%--run_mod
                                                --53.26%--PyEval_EvalCode
                                                         py::<module>:/
→src/script.py
                                                         _PyEval_
→EvalFrameDefault
                                                         PyObject_
→Vectorcall
                                                         _PyEval_Vector
                                                         py::baz:/src/
→script.py
                                                         _PyEval_
\hookrightarrowEvalFrameDefault
                                                         PyObject_
→Vectorcall
                                                         _PyEval_Vector
                                                         py::bar:/src/
⇔script.py
                                                         _PyEval_
→EvalFrameDefault
                                                         PyObject_
→Vectorcall
                                                         _PyEval_Vector
```

(continues on next page)

(continued from previous page)

1 How to enable perf profiling support

perf profiling support can be enabled either from the start using the environment variable PYTHONPERFSUPPORT or the -X perf option, or dynamically using sys.activate_stack_trampoline() and sys.deactivate_stack_trampoline().

The sys functions take precedence over the -X option, the -X option takes precedence over the environment variable.

Example, using the environment variable:

```
$ PYTHONPERFSUPPORT=1 python script.py
$ perf report -g -i perf.data
```

Example, using the -x option:

```
$ python -X perf script.py
$ perf report -g -i perf.data
```

Example, using the sys APIs in file example.py:

```
import sys

sys.activate_stack_trampoline("perf")
do_profiled_stuff()
sys.deactivate_stack_trampoline()

non_profiled_stuff()
```

...then:

```
$ python ./example.py
$ perf report -g -i perf.data
```

2 How to obtain the best results

For best results, Python should be compiled with CFLAGS="-fno-omit-frame-pointer -mno-omit-leaf-frame-pointer" as this allows profilers to unwind using only the frame pointer and not on DWARF debug information. This is because as the code that is interposed to allow perf support is dynamically generated it doesn't have any DWARF debugging information available.

You can check if your system has been compiled with this flag by running:

```
$ python -m sysconfig | grep 'no-omit-frame-pointer'
```

If you don't see any output it means that your interpreter has not been compiled with frame pointers and therefore it may not be able to show Python functions in the output of perf.

Index

Ε

environment variable PYTHONPERFSUPPORT, 4

Р

PYTHONPERFSUPPORT, 4