

---

# Soporte de Python para el perfilador perf de Linux

Versión 3.12.4

Guido van Rossum and the Python development team

julio 31, 2024

Python Software Foundation  
Email: docs@python.org

## Índice general

1	Cómo habilitar el soporte de creación de perfiles <code>perf</code>	4
2	Cómo obtener los mejores resultados	4
	Índice	5

---

### autor

Pablo Galindo

El perfilador `perf` de Linux es una herramienta muy poderosa que le permite crear perfiles y obtener información sobre el rendimiento de su aplicación. `perf` también tiene un ecosistema muy vibrante de herramientas que ayudan con el análisis de los datos que produce.

El principal problema con el uso del perfilador `perf` con aplicaciones Python es que `perf` sólo obtiene información sobre símbolos nativos, es decir, los nombres de funciones y procedimientos escritos en C. Esto significa que los nombres y nombres de archivos de las funciones de Python en su código no aparecerán en la salida de `perf`.

Desde Python 3.12, el intérprete puede ejecutarse en un modo especial que permite que las funciones de Python aparezcan en la salida del perfilador `perf`. Cuando este modo está habilitado, el intérprete interpondrá un pequeño fragmento de código compilado sobre la marcha antes de la ejecución de cada función de Python y enseñará a `perf` la relación entre este fragmento de código y la función de Python asociada usando `perf map` files.

### Nota

Actualmente, el soporte para el perfilador `perf` solo está disponible para Linux en arquitecturas seleccionadas. Verifique el resultado del paso de compilación `configure` o verifique el resultado de `python -m sysconfig | grep HAVE_PERF_TRAMPOLINE` para ver si su sistema es compatible.

Por ejemplo, considere el siguiente script:

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

(continúe en la próxima página)

(proviene de la página anterior)

```

    return result

def bar(n):
    foo(n)

def baz(n):
    bar(n)

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

```

Podemos ejecutar `perf` para obtener un registro de los seguimientos de la pila de CPU a 9999 hercios:

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

Luego podemos usar `perf report` para analizar los datos:

```
$ perf report --stdio -n -g
```

#	Children	Self	Samples	Command	Shared Object	Symbol
#	.....	.....	.....	.....	.....	.....
↪	.....					
#	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						
						PyObject_
↪ Vectorcall						
						_PyEval_Vector
						_PyEval_
↪ EvalFrameDefault						
						PyObject_
↪ Vectorcall						
						_PyEval_Vector
						_PyEval_
↪ EvalFrameDefault						
						PyObject_
↪ Vectorcall						
						_PyEval_Vector
						--51.67%--
↪ PyEval_EvalFrameDefault						
↪ 11.52%--						_PyLong_Add
↪						

(continúe en la próxima página)

```

↪ | --2.97% --PyObject_Malloc
...

```

As you can see, the Python functions are not shown in the output, only `_PyEval_EvalFrameDefault` (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.

En cambio, si ejecutamos el mismo experimento con el soporte `perf` habilitado obtenemos:

```

$ perf report --stdio -n -g
# Children      Self          Samples  Command      Shared Object      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_
↪EvalFrameDefault
                |   |   |   |   |   PyObject_
↪Vectorcall
                |   |   |   |   |   _PyEval_Vector
                |   |   |   |   |   py::bar:/src/
↪script.py
                |   |   |   |   |   _PyEval_
↪EvalFrameDefault
                |   |   |   |   |   PyObject_
↪Vectorcall
                |   |   |   |   |   _PyEval_Vector
                |   |   |   |   |   py::foo:/src/
↪script.py
                |   |   |   |   |   |
↪PyEval_EvalFrameDefault
                |   |   |   |   |   |   |
                |   |   |   |   |   |   |--
↪13.77%--_PyLong_Add

```

(continúe en la próxima página)

(proviene de la página anterior)

```
--3.26%--PyObject_Malloc
```

## 1 Cómo habilitar el soporte de creación de perfiles perf

El soporte de creación de perfiles `perf` se puede habilitar desde el principio usando la variable de entorno `PYTHONPERFSUPPORT` o la opción `-X perf`, o dinámicamente usando `sys.activate_stack_trampoline()` y `sys.deactivate_stack_trampoline()`.

Las funciones `sys` tienen prioridad sobre la opción `-X`, la opción `-X` tiene prioridad sobre la variable de entorno.

Ejemplo, usando la variable de entorno:

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

Ejemplo, usando la opción -X:

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

Ejemplo, usando las API `sys` en el archivo `example.py`:

```
import sys

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

non_profiled_stuff()
```

...entonces:

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

## 2 Cómo obtener los mejores resultados

Para obtener mejores resultados, Python debe compilarse con `CFLAGS="-fno-omit-frame-pointer-mno-omit-leaf-frame-pointer"` ya que esto permite a los perfiladores desenrollarse usando solo el puntero del marco y no en la información de depuración de DWARF. Esto se debe a que como el código que se interpone para permitir el soporte `perf` se genera dinámicamente, no tiene ninguna información de depuración DWARF disponible.

Puede verificar si su sistema ha sido compilado con este indicador ejecutando:

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

Si no ve ningún resultado, significa que su intérprete no ha sido compilado con punteros de marco y, por lo tanto, es posible que no pueda mostrar funciones de Python en el resultado de `perf`.

## Índice

### P

PYTHONPERFSUPPORT, [4](#)

### V

variables de entorno  
PYTHONPERFSUPPORT, [4](#)