

---

# Instrumentación de CPython con DTrace y SystemTap

*Versión 3.14.0rc2*

**Guido van Rossum and the Python development team**

septiembre 01, 2025

Python Software Foundation  
Email: [docs@python.org](mailto:docs@python.org)

## Índice general

<b>1</b>	<b>Habilitando los marcadores estáticos</b>	<b>2</b>
<b>2</b>	<b>Sondas estáticas DTrace</b>	<b>3</b>
<b>3</b>	<b>Marcadores estáticos SystemTap</b>	<b>4</b>
<b>4</b>	<b>Marcadores estáticos disponibles</b>	<b>5</b>
<b>5</b>	<b>SystemTap Tapsets</b>	<b>6</b>
<b>6</b>	<b>Ejemplos</b>	<b>7</b>

---

### **autor**

David Malcolm

### **autor**

Łukasz Langa

DTrace y SystemTap son herramientas de monitoreo, cada una de las cuales proporciona una forma de inspeccionar lo que están haciendo los procesos en un sistema informático. Ambos usan lenguajes específicos de dominio que permiten al usuario escribir scripts que:

- filtrar qué procesos deben observarse
- recopilar datos de los procesos de interés
- generar reportes sobre los datos

A partir de Python 3.6, CPython se puede construir con «marcadores» incrustados, también conocidos como «sondas», que se pueden observar mediante un script de DTrace o SystemTap, lo que facilita la supervisión de lo que hacen los procesos de CPython en un sistema.

Los marcadores de DTrace son detalles de implementación del intérprete CPython. No se ofrecen garantías sobre la compatibilidad de la sonda entre versiones de CPython. Los scripts de DTrace pueden dejar de funcionar o funcionar incorrectamente sin previo aviso al cambiar las versiones de CPython.

# 1 Habilitando los marcadores estáticos

macOS viene con soporte integrado para DTrace. En Linux, para construir CPython con los marcadores incrustados para SystemTap, se deben instalar las herramientas de desarrollo de SystemTap.

En una máquina Linux, esto se puede hacer a través de:

```
$ yum install systemtap-sdt-devel
```

o:

```
$ sudo apt-get install systemtap-sdt-dev
```

CPython debe ser configurado con la opción `--with-dtrace`:

```
checking for --with-dtrace... yes
```

En macOS, puede enumerar las sondas disponibles de DTrace ejecutando un proceso de Python en segundo plano y listando todas las sondas disponibles por el proveedor de Python:

```
$ python3.6 -q &
$ sudo dtrace -l -P python$! # or: dtrace -l -m python3.6
```

ID	PROVIDER	MODULE	FUNCTION NAME
29564	python18035	python3.6	_PyEval_EvalFrameDefault function-entry
29565	python18035	python3.6	dtrace_function_entry function-entry
29566	python18035	python3.6	_PyEval_EvalFrameDefault function-
↪return			
29567	python18035	python3.6	dtrace_function_return function-
↪return			
29568	python18035	python3.6	collect gc-done
29569	python18035	python3.6	collect gc-start
29570	python18035	python3.6	_PyEval_EvalFrameDefault line
29571	python18035	python3.6	maybe_dtrace_line line

En Linux, puede verificar si los marcadores estáticos SystemTap están presentes en el binario construido al ver si contiene una sección `«.note.stapsdt»`.

```
$ readelf -S ./python | grep .note.stapsdt
[30] .note.stapsdt          NOTE              0000000000000000 00308d78
```

Si ha creado Python como una biblioteca compartida (con la opción de configuración `--enable-shared`), debe buscar en la biblioteca compartida. Por ejemplo:

```
$ readelf -S libpython3.3dm.so.1.0 | grep .note.stapsdt
[29] .note.stapsdt          NOTE              0000000000000000 00365b68
```

Un lector de formato ELF suficientemente moderno puede imprimir los metadatos:

```
$ readelf -n ./python
```

Displaying notes found at file offset 0x00000254 with length 0x00000020:

Owner	Data size	Description
GNU	0x00000010	NT_GNU_ABI_TAG (ABI version tag)
OS: Linux, ABI: 2.6.32		

Displaying notes found at file offset 0x00000274 with length 0x00000024:

Owner	Data size	Description
GNU	0x00000014	NT_GNU_BUILD_ID (unique build ID)

(continúe en la próxima página)

```

↪bitstring)
    Build ID: df924a2b08a7e89f6e11251d4602022977af2670

Displaying notes found at file offset 0x002d6c30 with length 0x00000144:
    Owner          Data size          Description
    stapsdt         0x000000031          NT_STAPSDT (SystemTap probe
↪descriptors)
    Provider: python
    Name: gc__start
    Location: 0x00000000004371c3, Base: 0x0000000000630ce2, Semaphore:
↪0x000000000008d6bf6
    Arguments: -4@%ebx
    stapsdt         0x000000030          NT_STAPSDT (SystemTap probe
↪descriptors)
    Provider: python
    Name: gc__done
    Location: 0x00000000004374e1, Base: 0x0000000000630ce2, Semaphore:
↪0x000000000008d6bf8
    Arguments: -8@%rax
    stapsdt         0x000000045          NT_STAPSDT (SystemTap probe
↪descriptors)
    Provider: python
    Name: function__entry
    Location: 0x000000000053db6c, Base: 0x0000000000630ce2, Semaphore:
↪0x000000000008d6be8
    Arguments: 8@%rbp 8@%r12 -4@%eax
    stapsdt         0x000000046          NT_STAPSDT (SystemTap probe
↪descriptors)
    Provider: python
    Name: function__return
    Location: 0x000000000053dba8, Base: 0x0000000000630ce2, Semaphore:
↪0x000000000008d6bea
    Arguments: 8@%rbp 8@%r12 -4@%eax

```

The above metadata contains information for SystemTap describing how it can patch strategically placed machine code instructions to enable the tracing hooks used by a SystemTap script.

## 2 Sondas estáticas DTrace

El siguiente ejemplo de script DTrace se puede utilizar para mostrar la jerarquía de llamada/retorno de un script de Python, solo rastreando dentro de la invocación de una función llamada «start». En otras palabras, las llamadas a funciones durante la importación aparecerán en la lista:

```

self int indent;

python$target:::function-entry
/copyinstr(arg1) == "start"/
{
    self->trace = 1;
}

python$target:::function-entry
/self->trace/
{
    printf("%d\t%s:", timestamp, 15, probename);
    printf("%s", self->indent, "");
}

```

(continúe en la próxima página)

(proviene de la página anterior)

```
printf("%s:%s:%d\n", basename(copyinstr(arg0)), copyinstr(arg1), arg2);
self->indent++;
}

python$target::function-return
/self->trace/
{
    self->indent--;
    printf("%d\t%s:", timestamp, 15, probename);
    printf("%s", self->indent, "");
    printf("%s:%s:%d\n", basename(copyinstr(arg0)), copyinstr(arg1), arg2);
}

python$target::function-return
/copyinstr(arg1) == "start"/
{
    self->trace = 0;
}
```

Se puede invocar así:

```
$ sudo dtrace -q -s call_stack.d -c "python3.6 script.py"
```

La salida se verá así:

```
156641360502280 function-entry:call_stack.py:start:23
156641360518804 function-entry: call_stack.py:function_1:1
156641360532797 function-entry: call_stack.py:function_3:9
156641360546807 function-return: call_stack.py:function_3:10
156641360563367 function-return: call_stack.py:function_1:2
156641360578365 function-entry: call_stack.py:function_2:5
156641360591757 function-entry: call_stack.py:function_1:1
156641360605556 function-entry: call_stack.py:function_3:9
156641360617482 function-return: call_stack.py:function_3:10
156641360629814 function-return: call_stack.py:function_1:2
156641360642285 function-return: call_stack.py:function_2:6
156641360656770 function-entry: call_stack.py:function_3:9
156641360669707 function-return: call_stack.py:function_3:10
156641360687853 function-entry: call_stack.py:function_4:13
156641360700719 function-return: call_stack.py:function_4:14
156641360719640 function-entry: call_stack.py:function_5:18
156641360732567 function-return: call_stack.py:function_5:21
156641360747370 function-return:call_stack.py:start:28
```

### 3 Marcadores estáticos SystemTap

La forma de bajo nivel para utilizar la integración de SystemTap es utilizar los marcadores estáticos directamente. Esto requiere que indique explícitamente el archivo binario que los contiene.

Por ejemplo, este script SystemTap se puede utilizar para mostrar la jerarquía de llamada/retorno de un script de Python:

```
probe process("python").mark("function__entry") {
    filename = user_string($arg1);
    funcname = user_string($arg2);
    lineno = $arg3;
```

(continúe en la próxima página)

(proviene de la página anterior)

```
printf("%s => %s in %s:%d\\n",
        thread_indent(1), funcname, filename, lineno);
}

probe process("python").mark("function__return") {
    filename = user_string($arg1);
    funcname = user_string($arg2);
    lineno = $arg3;

    printf("%s <= %s in %s:%d\\n",
            thread_indent(-1), funcname, filename, lineno);
}
```

Se puede invocar así:

```
$ stap \
  show-call-hierarchy.stp \
  -c "./python test.py"
```

La salida se verá así:

```
11408 python(8274):      => __contains__ in Lib/_abcoll.py:362
11414 python(8274):      => __getitem__ in Lib/os.py:425
11418 python(8274):      => encode in Lib/os.py:490
11424 python(8274):      <= encode in Lib/os.py:493
11428 python(8274):      <= __getitem__ in Lib/os.py:426
11433 python(8274):      <= __contains__ in Lib/_abcoll.py:366
```

donde las columnas son:

- tiempo en microsegundos desde el inicio del script
- nombre del ejecutable
- PID de proceso

y el resto indica la jerarquía de llamada/retorno a medida que se ejecuta el script.

Para una compilación `--enable-shared` de CPython, los marcadores están contenidos dentro de la biblioteca compartida `libpython`, y la ruta de puntos de la sonda debe reflejar esto. Por ejemplo, esta línea del ejemplo anterior:

```
probe process("python").mark("function__entry") {
```

en su lugar debería leer:

```
probe process("python").library("libpython3.6dm.so.1.0").mark("function__entry") {
```

(asumiendo una compilación de depuración de CPython 3.6)

## 4 Marcadores estáticos disponibles

**function\_\_entry(str filename, str funcname, int lineno)**

Este marcador indica que ha comenzado la ejecución de una función de Python. Solo se activa para funciones de Python puro (código de bytes).

El nombre del archivo, el nombre de la función y el número de línea se retornan al script de rastreo como argumentos posicionales, a los que se debe acceder usando `$arg1`, `$arg2`, `$arg3`:

- `$arg1`: (const char \*) nombre del archivo, accesible usando `user_string($arg1)`

- `$arg2: (const char *)` nombre de la función, accesible usando `user_string($arg2)`
- `$arg3: int` número de línea

**`function__return(str filename, str funcname, int lineno)`**

This marker is the converse of `function__entry()`, and indicates that execution of a Python function has ended (either via `return`, or via an exception). It is only triggered for pure-Python (bytecode) functions.

The arguments are the same as for `function__entry()`

**`line(str filename, str funcname, int lineno)`**

Este marcador indica que una línea de Python está a punto de ejecutarse. Es el equivalente al rastreo línea por línea con un generador de perfiles de Python. No se activa con las funciones de C.

The arguments are the same as for `function__entry()`.

**`gc__start(int generation)`**

Fires when the Python interpreter starts a garbage collection cycle. `arg0` is the generation to scan, like `gc.collect()`.

**`gc__done(long collected)`**

Se activa cuando el intérprete de Python finaliza un ciclo de recolección de basura. `arg0` es el número de objetos recopilados.

**`import__find__load__start(str modulename)`**

Se activa antes `importlib` e intenta encontrar y cargar el módulo. `arg0` es el nombre del módulo.

Added in version 3.7.

**`import__find__load__done(str modulename, int found)`**

Se activa después de que la función `find_and_load` de `importlib` es llamada. `arg0` es el nombre del módulo, `arg1` indica si el módulo se cargó correctamente.

Added in version 3.7.

**`audit(str event, void *tuple)`**

Se activa cuando se llama `sys.audit()` o `PySys_Audit()`. `arg0` es el nombre del evento como cadena C, `arg1` es un puntero `PyObject` a un objeto tupla.

Added in version 3.8.

## 5 SystemTap Tapsets

La forma de nivel superior de utilizar la integración de SystemTap es utilizar un «tapset»: el equivalente de SystemTap a una biblioteca, que oculta algunos de los detalles de bajo nivel de los marcadores estáticos.

A continuación un archivo de tapset, basado en una compilación no compartida de CPython:

```
/*
   Provide a higher-level wrapping around the function__entry and
   function__return markers:
 */
probe python.function.entry = process("python").mark("function__entry")
{
    filename = user_string($arg1);
    funcname = user_string($arg2);
    lineno = $arg3;
    frameptr = $arg4
}
probe python.function.return = process("python").mark("function__return")
{
    filename = user_string($arg1);
```

(continúe en la próxima página)

(proviene de la página anterior)

```
funcname = user_string($arg2);
lineno = $arg3;
frameptr = $arg4
}
```

Si este archivo está instalado en el directorio de tapset de SystemTap (por ejemplo, `/usr/share/systemtap/tapset`), estos puntos de sonda adicionales estarán disponibles:

**python.function.entry(str filename, str funcname, int lineno, frameptr)**

Este punto de sonda indica que ha comenzado la ejecución de una función de Python. Solo se activa para funciones de Python puro (código de bytes).

**python.function.return(str filename, str funcname, int lineno, frameptr)**

Este punto de prueba es el inverso de `python.function.return`, e indica que la ejecución de una función de Python ha finalizado (ya sea mediante `return` o mediante una excepción). Solo se activa para funciones de Python puro (código de bytes).

## 6 Ejemplos

Este script de SystemTap utiliza el tapset anterior para implementar de manera más limpia el ejemplo de rastrear la jerarquía de llamadas a funciones de Python, sin necesidad de nombrar directamente los marcadores estáticos:

```
probe python.function.entry
{
    printf("%s => %s in %s:%d\n",
        thread_indent(1), funcname, filename, lineno);
}

probe python.function.return
{
    printf("%s <= %s in %s:%d\n",
        thread_indent(-1), funcname, filename, lineno);
}
```

The following script uses the tapset above to provide a top-like view of all running CPython code, showing the top 20 most frequently entered bytecode frames, each second, across the whole system:

```
global fn_calls;

probe python.function.entry
{
    fn_calls[pid(), filename, funcname, lineno] += 1;
}

probe timer.ms(1000) {
    printf("\033[2J\033[1;1H") /* clear screen */
    printf("%6s %80s %6s %30s %6s\n",
        "PID", "FILENAME", "LINE", "FUNCTION", "CALLS")
    foreach ([pid, filename, funcname, lineno] in fn_calls- limit 20) {
        printf("%6d %80s %6d %30s %6d\n",
            pid, filename, lineno, funcname,
            fn_calls[pid, filename, funcname, lineno]);
    }
    delete fn_calls;
}
```