

---

# Instrumenter CPython avec DTrace et SystemTap

Version 3.12.4

Guido van Rossum and the Python development team

juillet 31, 2024

Python Software Foundation  
Email : docs@python.org

## Table des matières

1	Activer les marqueurs statiques	2
2	Sondes DTrace statiques	3
3	Marqueurs statiques <i>SystemTap</i>	4
4	Marqueurs statiques disponibles	5
5	<i>Tapsets</i> de <i>SystemTap</i>	6
6	Exemples	7

---

### auteur

David Malcolm

### auteur

Lukasz Langa

*DTrace* et *SystemTap* sont des outils de surveillance, chacun fournissant un moyen de d'inspecter ce que font les processus d'un système informatique. Ils utilisent tous les deux des langages dédiés permettant à un utilisateur d'écrire des scripts qui permettent de :

- Filtrer les processus à observer.
- Recueillir des données sur le processus choisi.
- Générer des rapports sur les données.

À partir de Python 3.6, CPython peut être compilé avec des « marqueurs » intégrés, aussi appelés « sondes », qui peuvent être observés par un script *DTrace* ou *SystemTap*, ce qui facilite le suivi des processus CPython.

**Particularité de l'implémentation CPython :** Les marqueurs DTrace sont des détails d'implémentation de l'interpréteur CPython. Aucune garantie n'est donnée quant à la compatibilité des sondes entre les versions de CPython. Les scripts DTrace peuvent s'arrêter de fonctionner ou fonctionner incorrectement sans avertissement lors du changement de version de CPython.

# 1 Activer les marqueurs statiques

macOS est livré avec un support intégré pour *DTrace*. Sous Linux, pour construire CPython avec les marqueurs embarqués pour *SystemTap*, les outils de développement *SystemTap* doivent être installés.

Sur une machine Linux, cela se fait via :

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

ou :

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

CPython doit être configuré avec l'option `--with-dtrace` :

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

Sous macOS, vous pouvez lister les sondes *DTrace* disponibles en exécutant un processus Python en arrière-plan et en listant toutes les sondes mises à disposition par le fournisseur 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

Sous Linux, pour vérifier que les marqueurs statiques *SystemTap* sont présents dans le binaire compilé, il suffit de regarder s'il contient une section `.note.stapsdt`.

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

Si vous avez compilé Python en tant que bibliothèque partagée (avec l'option `--enable-shared` du script configure), vous devez plutôt regarder dans la bibliothèque partagée. Par exemple :

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

Une version suffisamment moderne de *readelf* peut afficher les métadonnées :

```
$ 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_
↪bitstring)		
Build ID: df924a2b08a7e89f6e11251d4602022977af2670		

(suite sur la page suivante)

```

Displaying notes found at file offset 0x002d6c30 with length 0x00000144:
  Owner          Data size      Description
  stapsdt         0x00000031      NT_STAPSDT (SystemTap probe_
↪descriptors)
    Provider: python
    Name: gc__start
    Location: 0x00000000004371c3, Base: 0x0000000000630ce2, Semaphore:_
↪0x00000000008d6bf6
    Arguments: -4@%ebx
  stapsdt         0x00000030      NT_STAPSDT (SystemTap probe_
↪descriptors)
    Provider: python
    Name: gc__done
    Location: 0x00000000004374e1, Base: 0x0000000000630ce2, Semaphore:_
↪0x00000000008d6bf8
    Arguments: -8@%rax
  stapsdt         0x00000045      NT_STAPSDT (SystemTap probe_
↪descriptors)
    Provider: python
    Name: function__entry
    Location: 0x000000000053db6c, Base: 0x0000000000630ce2, Semaphore:_
↪0x00000000008d6be8
    Arguments: 8@%rbp 8@%r12 -4@%eax
  stapsdt         0x00000046      NT_STAPSDT (SystemTap probe_
↪descriptors)
    Provider: python
    Name: function__return
    Location: 0x000000000053dba8, Base: 0x0000000000630ce2, Semaphore:_
↪0x00000000008d6bea
    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 Sondes DTrace statiques

L'exemple suivant de script *DTrace* montre la hiérarchie d'appel/retour d'un script Python, en ne traçant que l'invocation d'une fonction *start*. En d'autres termes, les appels de fonctions lors de la phase d'import ne seront pas répertoriés :

```

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

python$target:::function-return

```

(suite sur la page suivante)

```

/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;
}

```

Il peut être utilisé de cette manière :

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

La sortie ressemble à ceci :

```

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 Marqueurs statiques *SystemTap*

La façon la plus simple d'utiliser l'intégration *SystemTap* est d'utiliser directement les marqueurs statiques. Pour cela vous devez pointer explicitement le fichier binaire qui les contient.

Par exemple, ce script *SystemTap* peut être utilisé pour afficher la hiérarchie d'appel/retour d'un script Python :

```

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

    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;
}

```

(suite sur la page suivante)

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

Il peut être utilisé de cette manière :

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

La sortie ressemble à ceci :

```
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
```

où les colonnes sont :

- temps en microsecondes depuis le début du script
- nom de l'exécutable
- PID du processus

et le reste indique la hiérarchie d'appel/retour lorsque le script s'exécute.

Pour une compilation `--enable-shared` de CPython, les marqueurs sont contenus dans la bibliothèque partagée *libpython*, et le chemin du module de la sonde doit le refléter. Par exemple, la ligne de l'exemple ci-dessus :

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

doit plutôt se lire comme :

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

(en supposant une version de CPython 3.6 compilée avec le débogage activé)

## 4 Marqueurs statiques disponibles

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

Ce marqueur indique que l'exécution d'une fonction Python a commencé. Il n'est déclenché que pour les fonctions en Python pur (code intermédiaire).

Le nom de fichier, le nom de la fonction et le numéro de ligne sont renvoyés au script de traçage sous forme d'arguments positionnels, auxquels il faut accéder en utilisant `$arg1`, `$arg2`, `$arg3` :

- `$arg1` : (const char \*) nom de fichier, accessible via `user_string($arg1)`
- `$arg2` : (const char \*) nom de la fonction, accessible via `user_string($arg2)`
- `$arg3` : numéro de ligne int

### **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)**

Ce marqueur indique qu'une ligne Python est sur le point d'être exécutée. C'est l'équivalent du traçage ligne par ligne avec un profileur Python. Il n'est pas déclenché dans les fonctions C.

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

### **gc\_start(int generation)**

Fonction appelée lorsque l'interpréteur Python lance un cycle de collecte du ramasse-miettes. `arg0` est la génération à scanner, comme `gc.collect()`.

### **gc\_done(long collected)**

Fonction appelée lorsque l'interpréteur Python termine un cycle de collecte du ramasse-miettes. `Arg0` est le nombre d'objets collectés.

### **import\_\_find\_\_load\_\_start(str modulename)**

Fonction appelée avant que `importlib` essaye de trouver et de charger le module. `arg0` est le nom du module.

Ajouté dans la version 3.7.

### **import\_\_find\_\_load\_\_done(str modulename, int found)**

Fonction appelée après que la fonction `find_and_load` du module `importlib` soit appelée. `arg0` est le nom du module, `arg1` indique si le module a été chargé avec succès.

Ajouté dans la version 3.7.

### **audit(str event, void \*tuple)**

Fonction appelée quand les fonctions `sys.audit()` ou `PySys_Audit()` sont appelées. `arg0` est le nom de l'évènement de type chaîne de caractère C. `arg1` est un pointeur sur un  $n$ -uplet d'objet de type `PyObject`.

Ajouté dans la version 3.8.

## **5 Tapsets de SystemTap**

La façon la plus simple d'utiliser l'intégration *SystemTap* est d'utiliser un « *tapset* ». L'équivalent pour *SystemTap* d'une bibliothèque, qui permet de masquer les détails de niveau inférieur des marqueurs statiques.

Voici un fichier *tapset*, basé sur une version non partagée compilée 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);
    funcname = user_string($arg2);
    lineno = $arg3;
    frameptr = $arg4
}
```

Si ce fichier est installé dans le répertoire *tapset* de *SystemTap* (par exemple `/usr/share/systemtap/tapset`), alors ces sondes supplémentaires deviennent disponibles :

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

Cette sonde indique que l'exécution d'une fonction Python a commencé. Elle n'est déclenchée que pour les fonctions en Python pur (code intermédiaire).

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

Cette sonde est l'inverse de `python.function.entry`, et indique que l'exécution d'une fonction Python est terminée (soit via `return`, soit via une exception). Elle est uniquement déclenchée pour les fonctions en Python pur (code intermédiaire ou *bytecode*).

## 6 Exemples

Ce script *SystemTap* utilise le *tapset* ci-dessus pour implémenter plus proprement l'exemple précédent de traçage de la hiérarchie des appels de fonctions Python, sans avoir besoin de nommer directement les marqueurs statiques :

```
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;
}
```