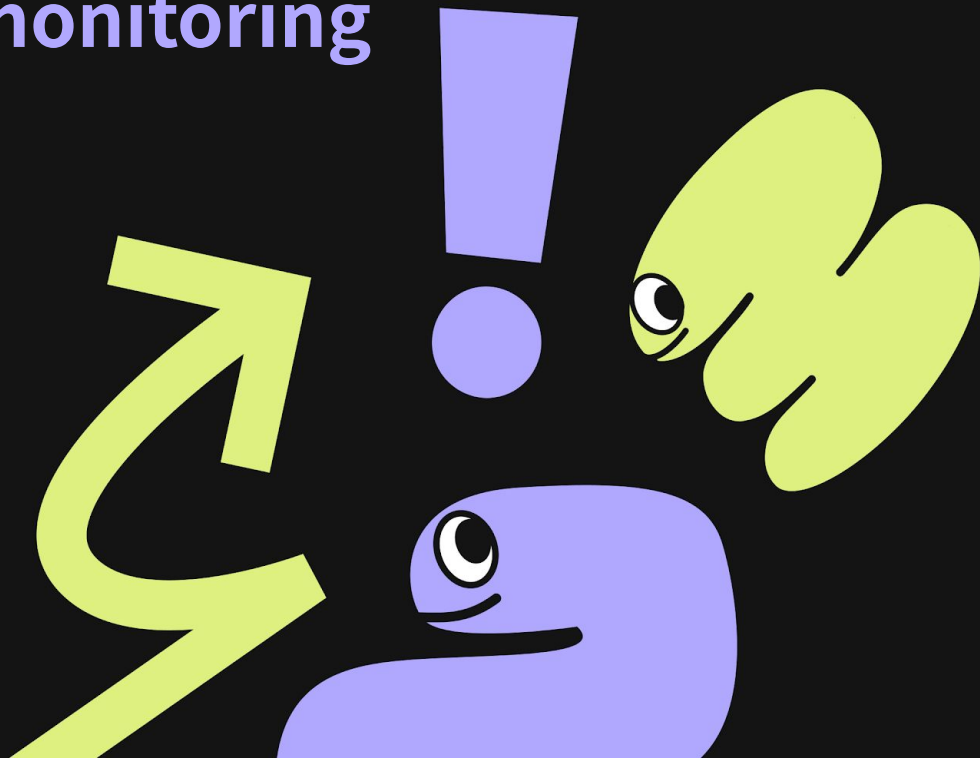


PyConKR 2024

# 함수 추적 도구를 만들며 알아보는 파이썬 고성능 event-monitoring

강민철



# 목차

1. 파이썬 코드 실행 추적
2. `sys.settrace()`의 한계
3. `sys.monitoring()`: 고성능 이벤트 모니터링 API
4. 성능 향상
5. `pyftrace`: 간단한 함수 추적 도구 제작
6. Q&A

# 목차

1. 파이썬 코드 실행 추적
2. `sys.settrace()`의 한계
3. `sys.monitoring()`: 고성능 이벤트 모니터링 API
4. 성능 향상
5. `pyftrace`: 간단한 함수 추적 도구 제작
6. Q&A

# 목차

1. 파이썬 코드 실행 추적
2. `sys.settrace()`의 한계
3. `sys.monitoring()`: 고성능 이벤트 모니터링 API
4. 성능 향상
5. `pyftrace`: 간단한 함수 추적 도구 제작
6. Q&A

# 목차

1. 파이썬 코드 실행 추적
2. `sys.settrace()`의 한계
3. **`sys.monitoring()`: 고성능 이벤트 모니터링 API**
4. 성능 향상
5. `pyftrace`: 간단한 함수 추적 도구 제작
6. Q&A

# 목차

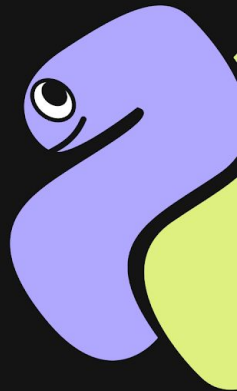
1. 파이썬 코드 실행 추적
2. `sys.settrace()`의 한계
3. `sys.monitoring()`: 고성능 이벤트 모니터링 API
4. 성능 향상
5. `pyftrace`: 간단한 함수 추적 도구 제작
6. Q&A

# 목차

1. 파이썬 코드 실행 추적
2. `sys.settrace()`의 한계
3. `sys.monitoring()`: 고성능 이벤트 모니터링 API
4. 성능 향상
5. **pyftrace: 간단한 함수 추적 도구 제작**
6. Q&A

PART 1

# 이벤트 모니터링 기반 파이썬 코드 실행 추적





# 코드 실행 추적

## 이벤트 기반 코드 추적

코드 실행 과정에서 발생하는 다양한 **이벤트**를 근거로  
어떤 코드가 어떻게 실행되었는지 **관찰**

# 코드 실행 추적

## 이벤트 기반 코드 추적

```
1 def divide(a, b):  
2     return a / b  
3  
4 def main():  
5     num1 = 10  
6     num2 = 2  
7     if num1 > num2:  
8         divide(num1, num2)  
9     else:  
10        divide(num2, num1)  
11  
12 if __name__ == "__main__":  
13     main()
```

# 코드 실행 추적

## 이벤트 기반 코드 추적

```
def divide(a, b):  
    return a / b  
  
def main():  
    num1 = 10  
    num2 = 2  
    if num1 > num2:  
        divide(num1, num2)  
    else:  
        divide(num2, num1)  
  
if __name__ == "__main__":  
    main()
```

# 코드 실행 추적

## 이벤트 기반 코드 추적

```
def divide(a, b):  
    return a / b  
  
def main():  
    num1 = 10  
    num2 = 2  
    if num1 > num2:  
        divide(num1, num2)  
    else:  
        divide(num2, num1)  
  
if __name__ == "__main__":  
    main()
```

# 코드 실행 추적

## 이벤트 기반 코드 추적

```
def divide(a, b):  
    return a / b # ZeroDivisionError  
  
def main():  
    num1 = 10  
    num2 = 0  
    if num1 > num2:  
        divide(num1, num2)  
    else:  
        divide(num2, num1)  
  
if __name__ == "__main__":  
    main()
```

# 코드 실행 추적

이벤트 기반 코드 추적 활용 사례

디버깅

프로파일링

커버리지

...

# 코드 실행 추적, 어디서 활용될까?

## Debugging (e.g. pdb)

```
$ python -m pdb some_script.py
> /home/minchul/workspace/some_script.py(1)<module>( )
-> def divide(a, b):
(Pdb) b divide
Breakpoint 1 at /home/minchul/workspace/some_script.py:1
(Pdb) c
> /home/minchul/workspace/some_script.py(2)divide( )
-> return a / b
```

# 코드 실행 추적, 어디서 활용될까?

## Profiling (e.g. cProfile/profile)

```
$ python -m cProfile -s time some_script.py  
4 function calls in 0.000 seconds
```

Ordered by: internal time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.000	0.000	some_script.py:1(divide)
1	0.000	0.000	0.000	0.000	some_script.py:1(<module>)
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof
1	0.000	0.000	0.000	0.000	some_script.py:4(main)



# 코드 실행 추적, 어디서 활용될까?

Test coverage (e.g. coverage.py)

```
$ coverage run some_script.py  
$ coverage report
```

Name	Stmts	Miss	Cover
some_script.py	10	1	90%
TOTAL	10	1	90%

# 코드 실행 추적, 어디서 활용될까?

## Test coverage (e.g. coverage.py)

Coverage report: 90%				
filter...				
<input type="checkbox"/> hide covered				
Files Functions Classes				
coverage.py v7.6.1, created at 2024-09-04 16:29 +0900				
File ▲	statements	missing	excluded	coverage
some_script.py	10	1	0	90%
<b>Total</b>	<b>10</b>	<b>1</b>	<b>0</b>	<b>90%</b>

# 코드 실행 추적,

## Test coverage (e.g

### Coverage report:

Files

Functions

coverage.py v7.6.1, creat

File ▲

some\_script.py

**Total**Coverage for **some\_script.py**: 90%

10 statements

9 run

1 missing

0 excluded

« prev

^ index

» next

coverage.py v7.6.1, created at 2024-09-04 16:29 +0900

```
1 def divide(a, b):
2     return a / b
3
4 def main():
5     num1 = 10
6     num2 = 2
7     if num1 > num2:
8         divide(num1, num2)
9     else:
10         divide(num2, num1)
11
12 if __name__ == "__main__":
13     main()
14
```

# ‘이벤트 모니터링 기반’ 코드 실행 추적

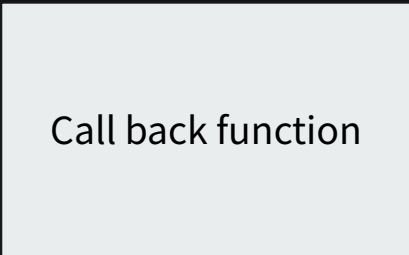
## 이벤트 모니터링

이벤트 모니터링 기본 양상

# ‘이벤트 모니터링 기반’ 코드 실행 추적

## 이벤트 모니터링

이벤트 모니터링 기본 양상

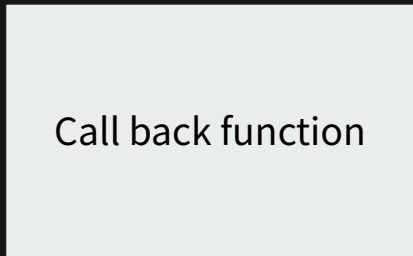


Call back function

# ‘이벤트 모니터링 기반’ 코드 실행 추적

## 이벤트 모니터링

이벤트 모니터링 기본 양상

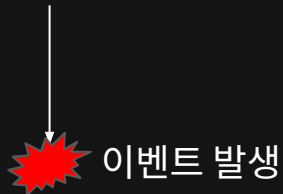


# ‘이벤트 모니터링 기반’ 코드 실행 추적

## 이벤트 모니터링

이벤트 모니터링 기본 양상

Call back function



# ‘이벤트 모니터링 기반’ 코드 실행 추적

## 이벤트 모니터링

이벤트 모니터링 기본 양상

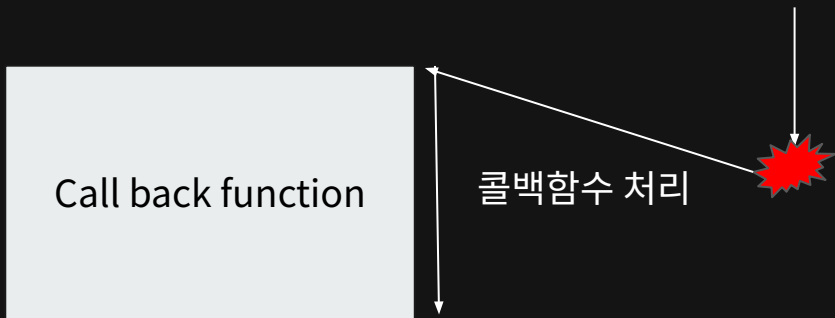




# ‘이벤트 모니터링 기반’ 코드 실행 추적

## 이벤트 모니터링

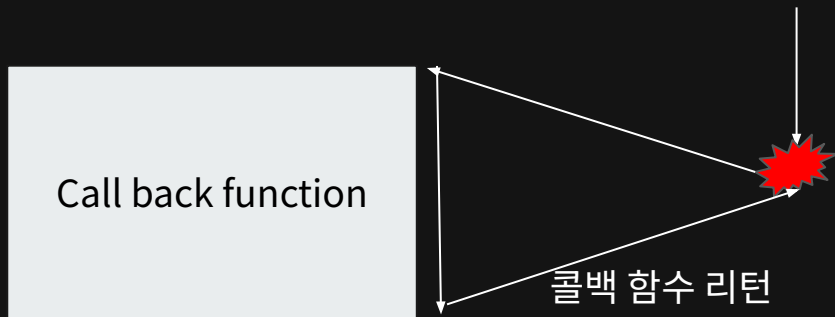
이벤트 모니터링 기본 양상



# ‘이벤트 모니터링 기반’ 코드 실행 추적

## 이벤트 모니터링

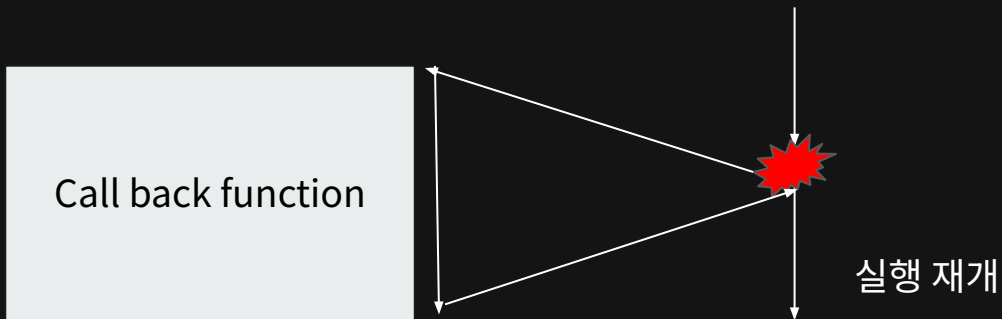
이벤트 모니터링 기본 양상



# ‘이벤트 모니터링 기반’ 코드 실행 추적

## 이벤트 모니터링

이벤트 모니터링 기본 양상



# 코드 실행 추적, 무엇으로 구현되었을까?

`sys.settrace(tracefunc)`



**Callback**; Global trace function

# 코드 실행 추적, 무엇으로 구현되었을까?

`sys.settrace(tracefunc)`



**Callback**; Global trace function

\* `sys.settrace(None)` to unset trace func

# 코드 실행 추적, 무엇으로 구현되었을까?

`sys.settrace(tracefunc)`



`tracefunc(frame, event, arg)`

# 코드 실행 추적, 무엇으로 구현되었을까?

`sys.settrace(tracefunc)`



`tracefunc(frame, event, arg)`

= Current stack frame

f_back	이전(caller) 스택 프레임
f_code	실행 중인 코드 객체
f_locals	로컬 변수 딕셔너리
f_globals	전역 변수 딕셔너리
f_builtins	내장 함수 딕셔너리
f_trace	이벤트 발생 시 호출되는 함수
f_lineno	실행 중인 라인 번호

# 코드 실행 추적, 무엇으로 구현되었을까?

`sys.settrace(tracefunc)`

<code>code.co_name</code>	함수 이름
<code>code.co_nlocals</code>	로컬 변수 수
<code>code.co_varnames</code>	로컬 변수 이름
<code>code.co_filename</code>	파일 이름
<code>code.co_firstlineno</code>	함수 라인 번호

<code>f_back</code>	이전(caller) 스택 프레임
<code>f_code</code>	실행 중인 <b>코드 객체</b>
<code>f_locals</code>	로컬 변수 딕셔너리
<code>f_globals</code>	전역 변수 딕셔너리
<code>f_builtins</code>	내장 함수 딕셔너리
<code>f_trace</code>	이벤트 발생 시 호출되는 함수
<code>f_lineno</code>	실행 중인 라인 번호



# 코드 실행 추적, 무엇으로 구현되었을까?

`sys.settrace(tracefunc)`



`tracefunc(frame, event, arg)`

`= Event (string)`

'call'	함수 또는 코드 블록 호출
'line'	새로운 코드 라인 실행
'return'	함수가 반환 직전
'exception'	예외 발생
'opcode'	새로운 명령어 실행 직전

# 코드 실행 추적, 무엇으로 구현되었을까?

`sys.settrace(tracefunc)`



`tracefunc(frame, event, arg)`

= argument(s) for event

# 코드 실행 추적, 무엇으로 구현되었을까?

```
sys.settrace(tracefunc)
```



```
tracefunc(frame, event, arg)
```

```
...
```

```
return something
```

# 코드 실행 추적, 무엇으로 구현되었을까?

```
sys.settrace(tracefunc)
```



```
tracefunc(frame, event, arg)
```

...

return something

< local trace function  
None

# 코드 실행 추적, 무엇으로 구현되었을까?

```
sys.settrace(tracefunc)
```



```
tracefunc(frame, event, arg)
```

...

```
return something
```



if 관심있는 함수 → local trace function

if 관심없는 함수 → None

# 코드 실행 추적, 무엇으로 구현되었을까?

`sys.settrace(tracefunc)`

```
def bar():  
    pass
```

```
def foo():  
    bar()
```

```
def main():  
    foo()  
    bar()
```

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

# 코드 실행 추적, 무엇으로 구현되었을까?

`sys.settrace(tracefunc)`

```
def bar():  
    pass
```

```
def foo():  
    bar()
```

```
def main():  
    foo()  
    bar()
```

```
if __name__ == "__main__":  
    sys.settrace(trace_func)  
    main()  
    sys.settrace(None)
```

# 코드 실행 추적, 무엇으로 구현되었을까?

`sys.settrace(tracefunc)`

```
def trace_func(frame, event, arg):  
    if event == "call":  
        print(f"[Call] {frame.f_code.co_name}")  
    elif event == "return":  
        print(f"[Return] {frame.f_code.co_name}")  
    return trace_func
```



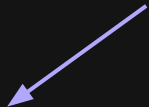
# 코드 실행 추적, 무엇으로 구현되었을까?

`sys.settrace(tracefunc)`

```
$ python3 settrace_example.py  
[Call] main  
[Call] foo  
[Call] bar  
[Return] bar  
[Return] foo  
[Call] bar  
[Return] bar  
[Return] main
```

# 코드 실행 추적, 무엇으로 구현되었을까?

참고) `sys.setprofile(profilefunc)`



`profilefunc(frame, event, arg)`

`∈ {'call', 'return', 'c_call', 'c_return'}`

# 코드 실행 추적, 어떻게 구현되었을까?

## 디버거 내부에서 활용된 사례

`sys.settrace(self.trace_dispatch)`

```
def do_debug(self, arg):
    """debug code

    Enter a recursive debugger that steps through the code
    argument (which is an arbitrary expression or statement to be
    executed in the current environment).
    """
    sys.settrace(None)
    globals = self.curframe.f_globals
    locals = self.curframe_locals
    p = Pdb(self.completekey, self.stdin, self.stdout)
    p.prompt = "(%s) " % self.prompt.strip()
    self.message("ENTERING RECURSIVE DEBUGGER")
    try:
        sys.call_tracing(p.run, (arg, globals, locals))
    except Exception:
        self._error_exc()
    self.message("LEAVING RECURSIVE DEBUGGER")
    sys.settrace(self.trace_dispatch)
    self.lastcmd = p.lastcmd
```

# 코드 실행 추적, 어떻게 구현되었을까?

## 커버리지 도구 내부에서 활용된 사례

`sys.settrace(self._trace)`

```
def start(self):
    """Start this Tracer.

    Return a Python function suitable for use with sys.settrace().

    """
    self.stopped = False
    if self.threading:
        if self.thread is None:
            self.thread = self.threading.currentThread()
        else:
            if self.thread.ident != self.threading.currentThread().ident:
                # Re-starting from a different thread!? Don't set the trace
                # function, but we are marked as running again, so maybe it
                # will be ok?
                #self.log("~", "starting on different threads")
            return self._trace
    sys.settrace(self._trace)
    return self._trace
```

# 코드 실행 추적, 어떻게 구현되었을까?

프로파일러 내부에서 활용된 사례

`sys.setprofile(self.dispatcher)`

```
def run(self, cmd):
    import __main__
    dict = __main__.__dict__
    return self.runctx(cmd, dict, dict)

def runctx(self, cmd, globals, locals):
    self.set_cmd(cmd)
    sys.setprofile(self.dispatcher)
    try:
        exec(cmd, globals, locals)
    finally:
        sys.setprofile(None)
    return self
```

PART 2

## sys.settrace()의 한계



# 성능에 최선일까?

## 불필요한 이벤트 처리 함수의 반복 호출

- 특정 이벤트 비활성화 불가
- 이벤트 추적 함수의 불필요한 반복 호출 야기

# 성능에 최선일까?

불필요한 이벤트 처리 함수의 반복 호출


```
def global_trace_function():  
  
    # do something  
  
    return local_trace_function
```



# 성능에 최선일까?

불필요한 이벤트 처리 함수의 반복 호출

```
def global_trace_function():  
    # do something  
    return local_trace_function  
  
def local_trace_function():  
    # do something
```




모든 event

# 성능에 최선일까?

불필요한 이벤트 처리 함수의 반복 호출

```
def global_trace_function():  
    # do something  
    return local_trace_function  
  
def local_trace_function():  
    # do something
```



추적 scope  $\propto$  호출 비용

# 여러 도구가 사용될 경우

## 도구의 충돌



디버거

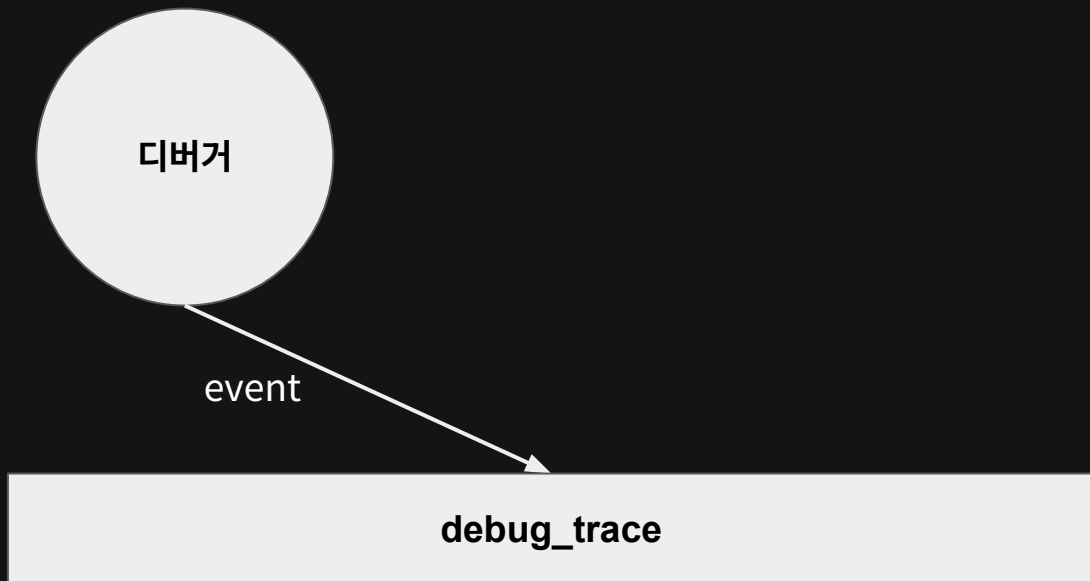
프로파일러

커버리지  
도구

...

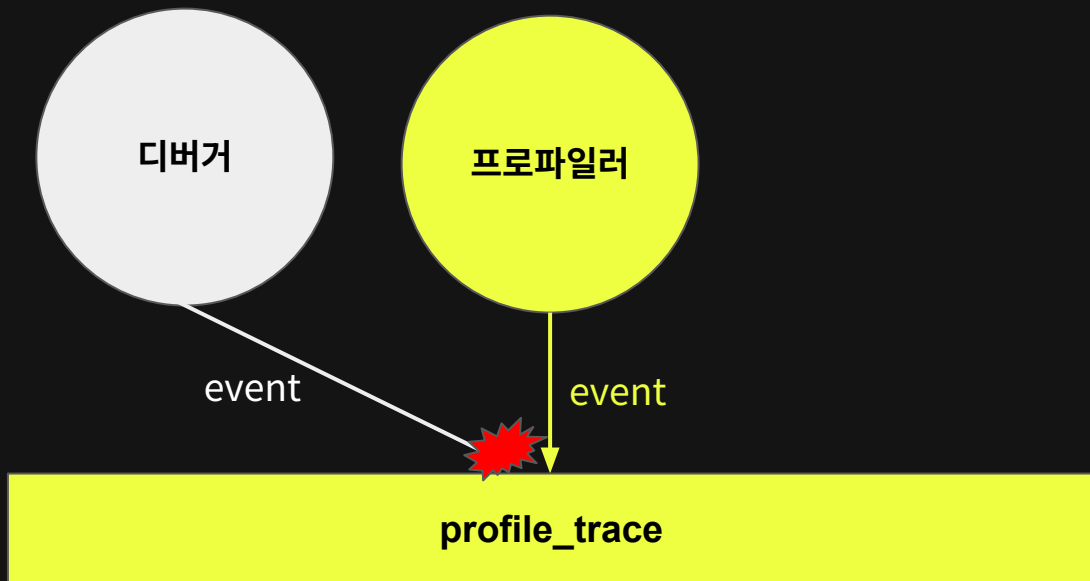
# 여러 도구가 사용될 경우

## 도구의 충돌



# 여러 도구가 사용될 경우

## 도구의 충돌



# 여러 도구가 사용될 경우

## 도구의 충돌

```
# 첫 번째 도구: 디버거
def debugger(frame, event, arg):
    if event == 'line':
        print(f"[Debugger] {event}")
    return debugger
```

# 여러 도구가 사용될 경우

## 도구의 충돌

```
# 두 번째 도구: 프로파일러
def profiler(frame, event, arg):
    if event == 'call':
        print(f"[Profiler] {event}")
    return profiler
```

# 여러 도구가 사용될 경우

## 도구의 충돌

```
# 세 번째 도구: 테스트 커버리지
def coverage(frame, event, arg):
    if event == 'return':
        print(f"[Coverage] {event}")
    return coverage
```



# 여러 도구가 사용될 경우

## 도구의 충돌

```
if __name__ == '__main__':  
    sys.settrace(debugger)  
    sys.settrace(profiler)  
    sys.settrace(coverage)  
  
    foo()  
  
    sys.settrace(None)
```

```
def bar():  
    return  
  
def foo():  
    bar()  
    return
```

# 여러 도구가 사용될 경우

## 도구의 충돌

debugger, profiler 무시

```
$ python3 multiple_tools_problem.py  
[Coverage] return  
[Coverage] return
```

PART 3

# `sys.monitoring():` 고성능 이벤트 모니터링 API



# sys.monitoring() since Python 3.12

## 고성능 이벤트 모니터링 API, sys.monitoring

### PEP 669 – Low Impact Monitoring for CPython

**Author:** Mark Shannon <mark at hotpy.org>

**Discussions-To:** [Discourse thread](#)

**Status:** Final

**Type:** Standards Track

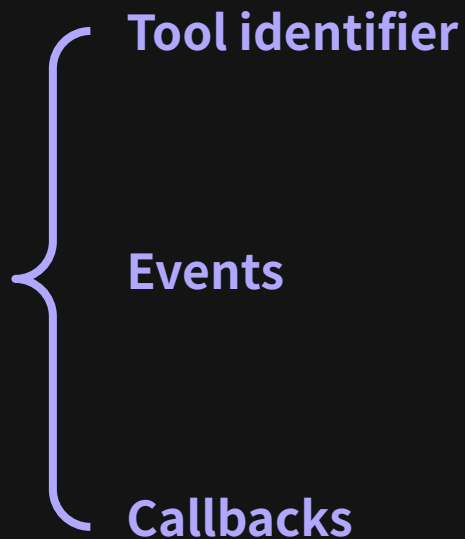
**Created:** 18-Aug-2021

**Python-Version:** 3.12

**Post-History:** [07-Dec-2021](#), [10-Jan-2022](#)

**Resolution:** [Discourse message](#)

# **sys.monitoring()** since Python 3.12



# `sys.monitoring()` since Python 3.12

**Tool identifier:** (int) 0~5, 모니터링 도구 충돌 방지

**Events**

**Callbacks**

## `sys.monitoring()` since Python 3.12

```
sys.monitoring.DEBUGGER_ID = 0  
sys.monitoring.COVERAGE_ID = 1  
sys.monitoring.PROFILER_ID = 2  
sys.monitoring.OPTIMIZER_ID = 5
```

## `sys.monitoring()` since Python 3.12

- tool id 사용 등록

`sys.monitoring.use_tool_id(tool_id, name)`

- tool id 사용 해제

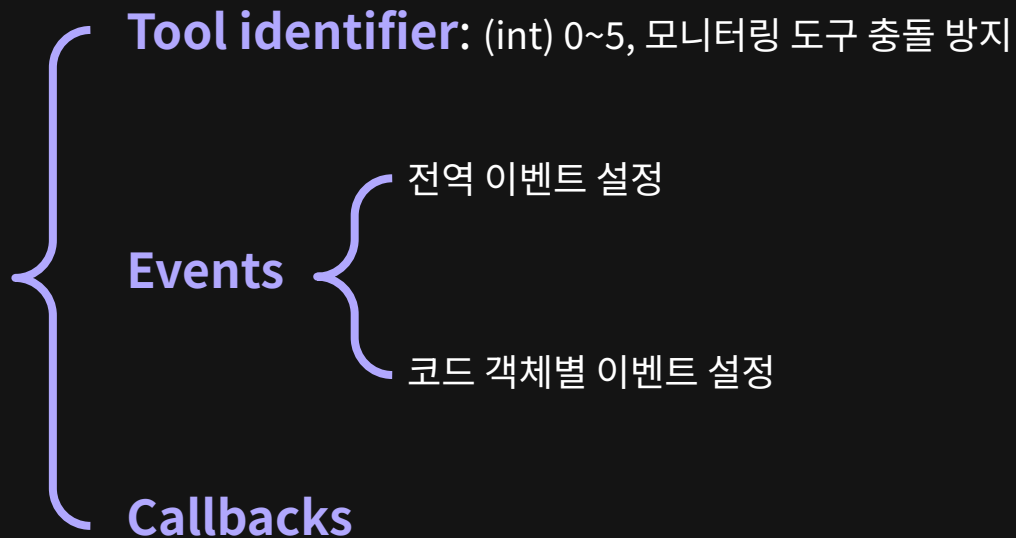
`sys.monitoring.free_tool_id(tool_id)`

- tool id 조회

`sys.monitoring.get_tool(tool_id)`



# sys.monitoring() since Python 3.12



## `sys.monitoring()` since Python 3.12

- 활성화된 모든 이벤트 반환

`sys.monitoring.get_events(tool_id)`

- 이벤트(`event_set`) 활성화

`sys.monitoring.set_events(tool_id, event_set)`

## `sys.monitoring()` since Python 3.12

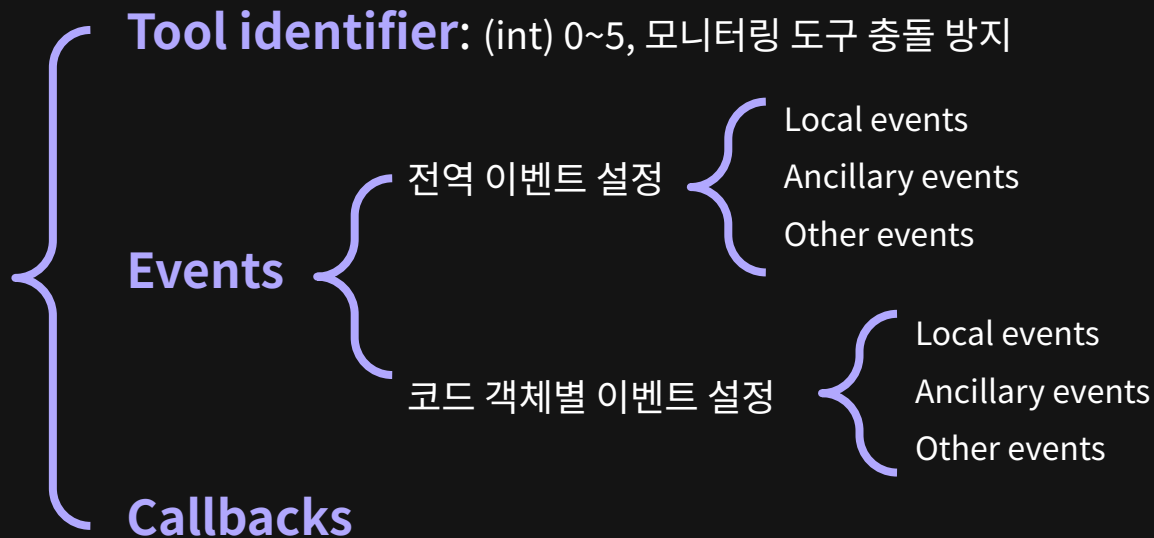
- 활성화된 모든 이벤트 반환

`sys.monitoring.get_local_events(tool_id, code)`

- 이벤트(`event_set`) 활성화

`sys.monitoring.set_local_events(tool_id, code, event_set)`

# sys.monitoring() since Python 3.12



Local events	PY_START	함수 시작(호출 이후 이벤트 발생)
	PY_RESUME	함수 재개
	PY_RETURN	함수 리턴
	CALL	함수 호출(호출 이전 이벤트 발생)
	LINE	다른 줄 번호 실행
	BRANCH	조건에 따른 분기 실행
	STOP_ITERATION	반복 종료

Ancillary events	C_RAISE	C 함수에서 예외 발생
	C_RETURN	C 함수 리턴

Other events	PY_THROW	Python 함수 throw
	RAISE	예외 발생
	EXCEPTION_HANDLED	예외 처리 발생

# `sys.monitoring()` since Python 3.12

Add BRANCH\_TAKEN and BRANCH\_NOT\_TAKEN events to `sys.monitoring` #122548

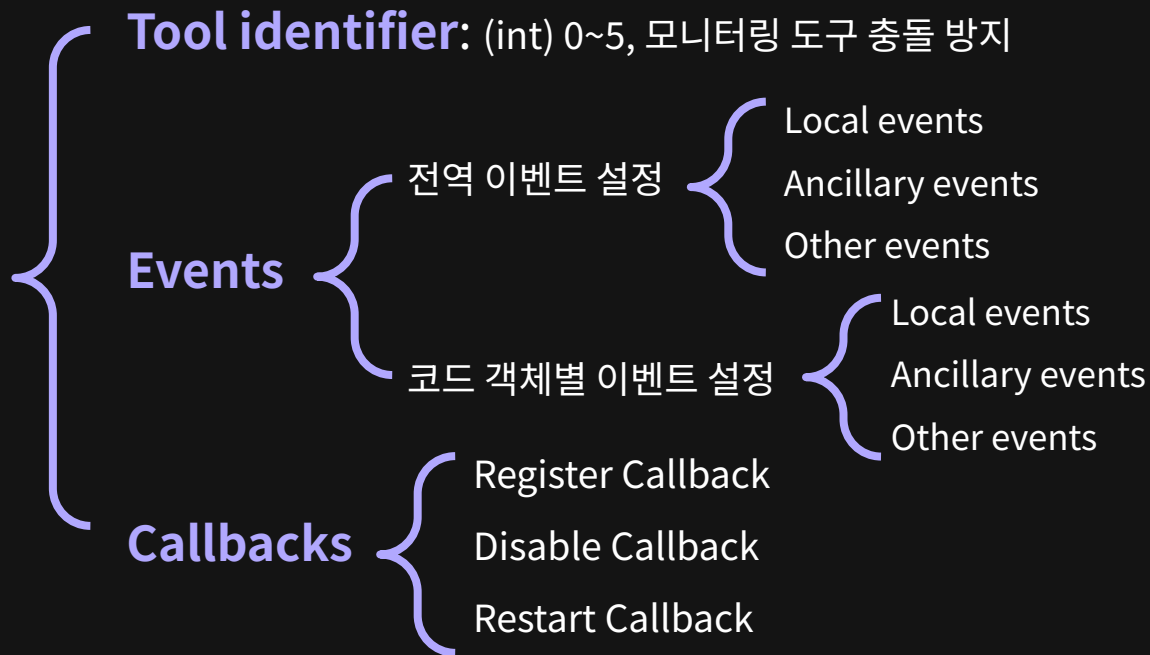
New issue

Open

1 of 3 tasks

markshannon opened this issue on Aug 1 · 6 comments

# sys.monitoring() since Python 3.12





## `sys.monitoring()` since Python 3.12

- tool id에 대한 콜백 함수 func 등록  
`sys.monitoring.register_callback(tool_id, event, func)`
- 현재 코드에 대한 이벤트 비활성화  
`return sys.monitoring.DISABLE`
- 비활성화된 모든 이벤트 활성화  
`sys.monitoring.restart_events()`

# `sys.monitoring()` since Python 3.12

## `sys.monitoring` 예제

```
def bar():  
    return
```

```
def foo():  
    bar()  
    return
```

```
foo()
```

# sys.monitoring() since Python 3.12

## sys.monitoring 예제

```
# tool_id = 0
tool_id = sys.monitoring.DEBUGGER_ID

sys.monitoring.use_tool_id(tool_id, "SimpleMonitor")
```

# sys.monitoring() since Python 3.12

## sys.monitoring 예제

```
sys.monitoring.register_callback(  
    tool_id,  
    sys.monitoring.events.CALL,  
    call_func  
)
```

```
sys.monitoring.register_callback(  
    tool_id,  
    sys.monitoring.events.PY_RETURN,  
    return_func  
)
```

# sys.monitoring() since Python 3.12

## sys.monitoring 예제

```
sys.monitoring.register_callback(  
    tool_id,  
    sys.monitoring.events.CALL,  
    call_func  
)
```

```
sys.monitoring.register_callback(  
    tool_id,  
    sys.monitoring.events.PY_RETURN,  
    return_func  
)
```

# sys.monitoring() since Python 3.12

## sys.monitoring 예제

```
def call_func(code, offset, func=None, arg0=None):  
    print(f"Function {func.__name__} called")
```

```
def return_func(code, offset, retval=None):  
    print(f"Function {code.co_name} returned")
```

# `sys.monitoring()` since Python 3.12

## `sys.monitoring` 예제

```
def bar():  
    return
```

```
def foo():  
    bar()  
    return
```

```
foo()
```

# sys.monitoring() since Python 3.12

## sys.monitoring 예제

```
$ python3 foo.py  
Function foo called  
Function bar called  
Function bar returned  
Function foo returned  
...
```



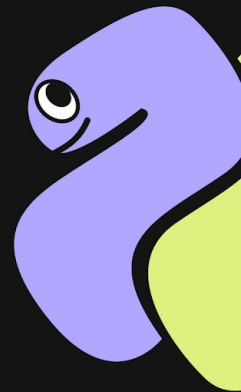
# `sys.monitoring()` since Python 3.12

## `sys.monitoring` 예제

```
sys.monitoring.free_tool_id(tool_id)
```

PART 4

## 성능 향상



# 얼마나 빠른가?

## 실제로 비교해 보자: keras - Simple MNIST convnet\*

실험 환경 (example 코드 있는 그대로 실행)

- 데이터셋: MNIST
- 학습 데이터 수/테스트 데이터 수: 60,000/10,000
- 배치 크기/**반복 수**: 128/**10**
- 손실 함수: Categorical Crossentropy
- 최적화 알고리즘: Adam
- 평가 지표: 정확도 (Accuracy)

\*[https://github.com/keras-team/keras-io/blob/ebe14fb985afd908e10dba8cbca179e5491e6d58/examples/vision/mnist\\_convnet.py](https://github.com/keras-team/keras-io/blob/ebe14fb985afd908e10dba8cbca179e5491e6d58/examples/vision/mnist_convnet.py)

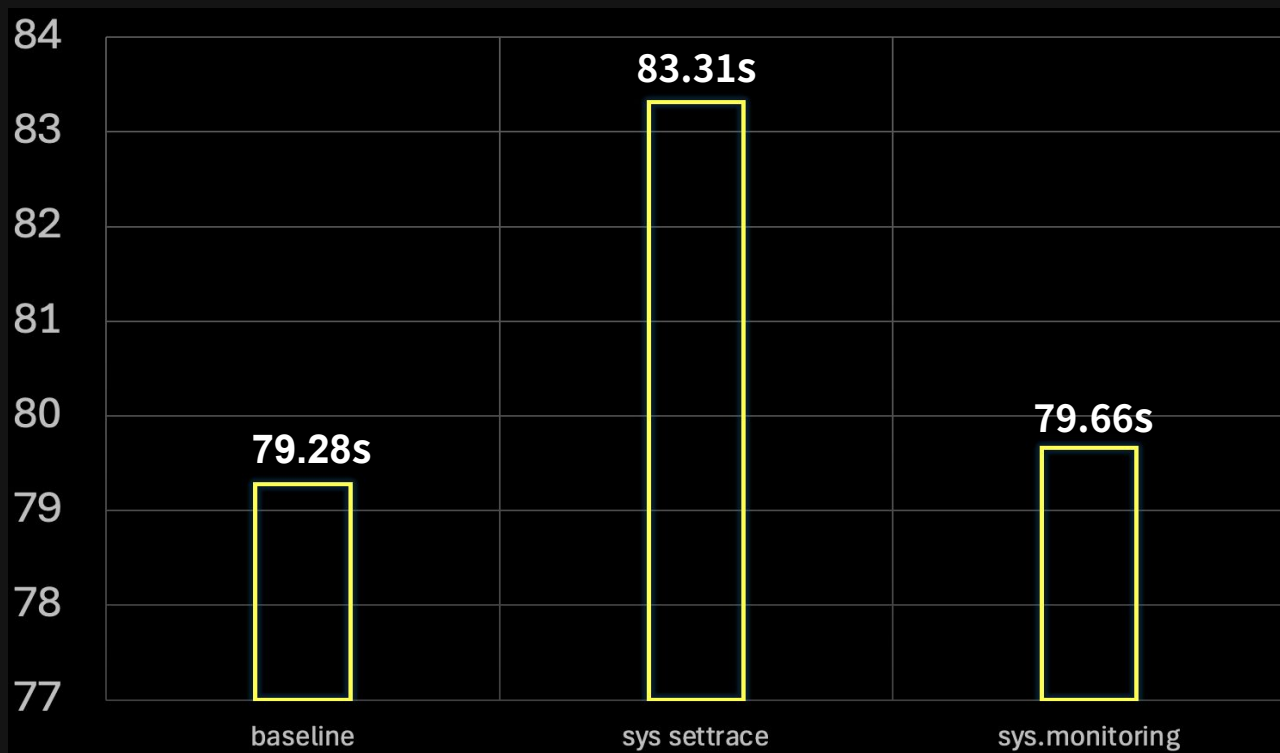
# 얼마나 빠른가?

## 실제로 비교해 보자: keras - Simple MNIST convnet

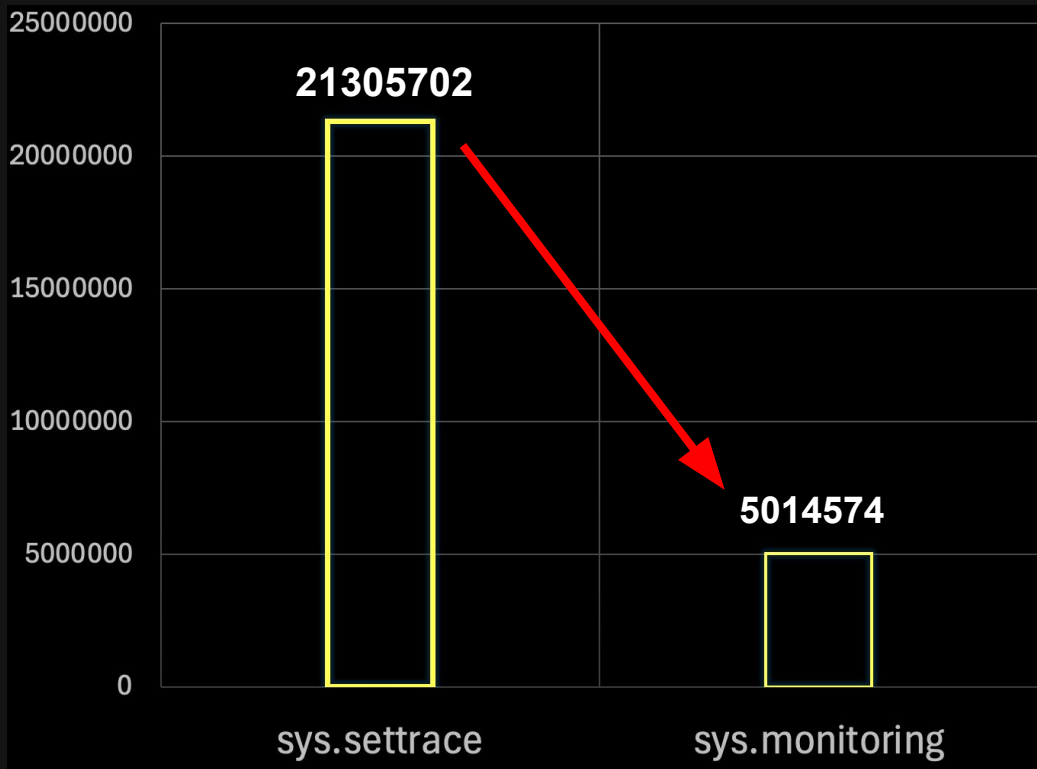
실험 환경	세팅 1: macOS	세팅 2: Linux
Python	Python 3.12.0	
OS	Sonoma 14.6.1	Ubuntu 20.04.6 LTS
CPU	Apple M1	AMD EPYC 7313P 16-Core
Mem.	16G	125G
Disk	256G	1.8T

[https://github.com/kangtegong/pycon2024/blob/main/trace\\_callback\\_keras\\_less\\_callbacks{N}.py](https://github.com/kangtegong/pycon2024/blob/main/trace_callback_keras_less_callbacks{N}.py)

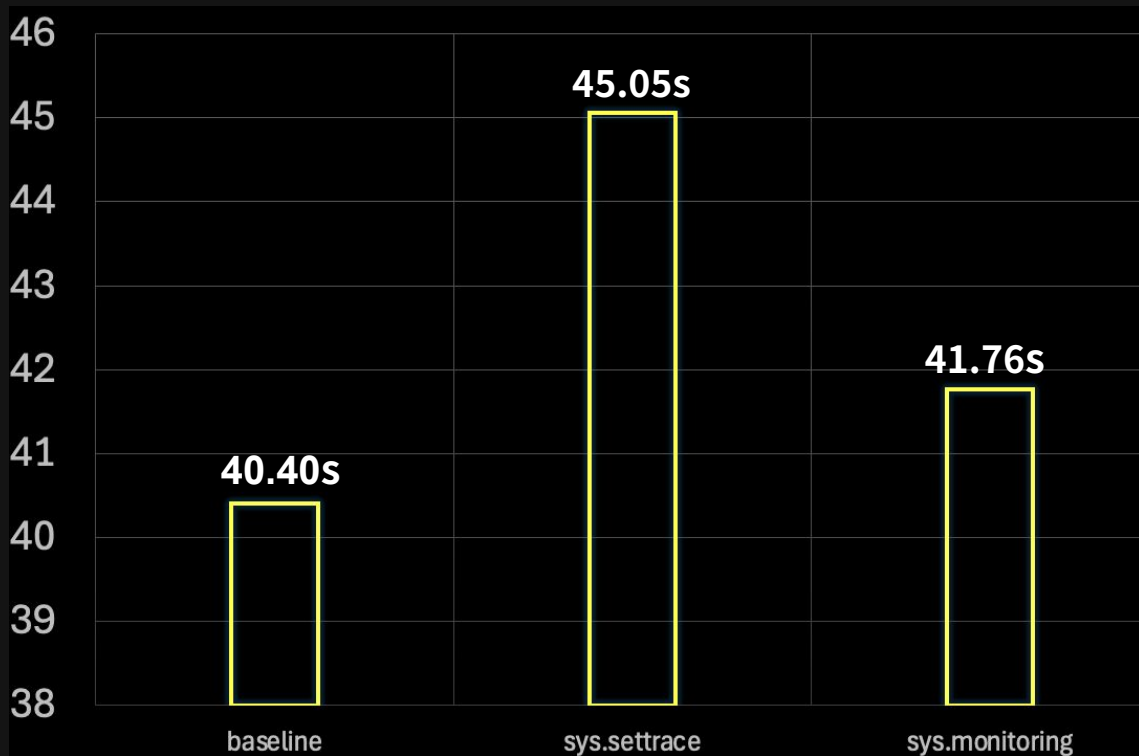
# 얼마나 빠른가?: 실행 시간 비교 @ macos



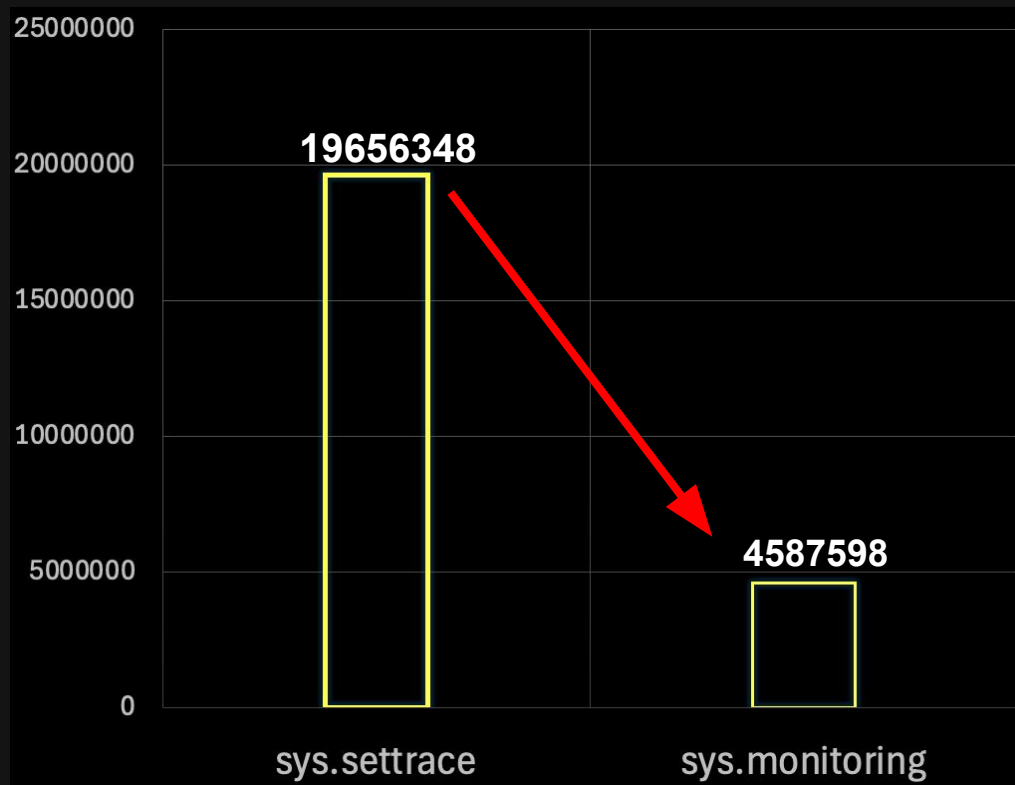
# 얼마나 빠른가?: 호출 횟수 비교 @ macos



# 얼마나 빠른가?: 실행 시간 비교 @ linux



# 얼마나 빠른가?: 호출 횟수 비교 @ linux





# 변경의 움직임

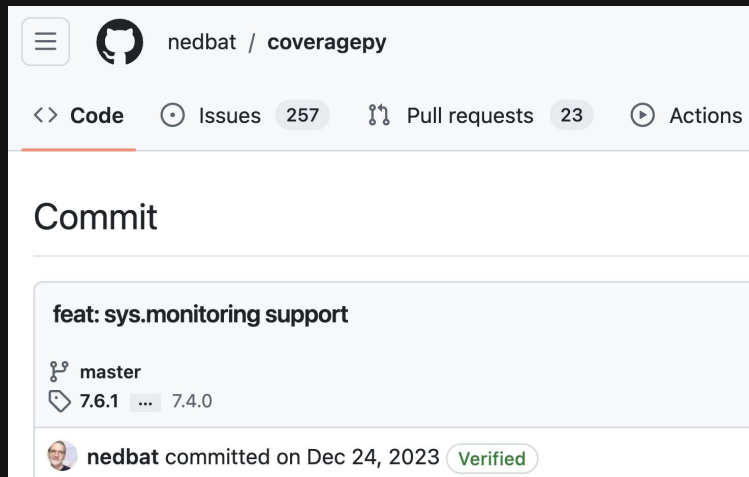
## coverage.py

### Coverage.py with sys.monitoring

Wednesday 27 December 2023

New in Python 3.12 is [sys.monitoring](#), a lighter-weight way to monitor the execution of Python programs. [Coverage.py 7.4.0](#) now can optionally use sys.monitoring instead of [sys.settrace](#), the facility that has underpinned coverage.py for nearly two decades. This is a big change, both in Python and in coverage.py. It would be great if you could try it out and provide some feedback.

Using sys.monitoring should reduce the overhead of coverage measurement, often lower than 5%, but of course your timings might be different. One of the things I would like to know is what your real-world speed improvements are like.



[https://nedbatchelder.com/blog/202312/coveragepy\\_with\\_sysmonitoring.html](https://nedbatchelder.com/blog/202312/coveragepy_with_sysmonitoring.html)

<https://github.com/nedbat/coveragepy/commit/bc1dbb2bee9647e321ca5f1ee95cdf9bf5da7c2a>

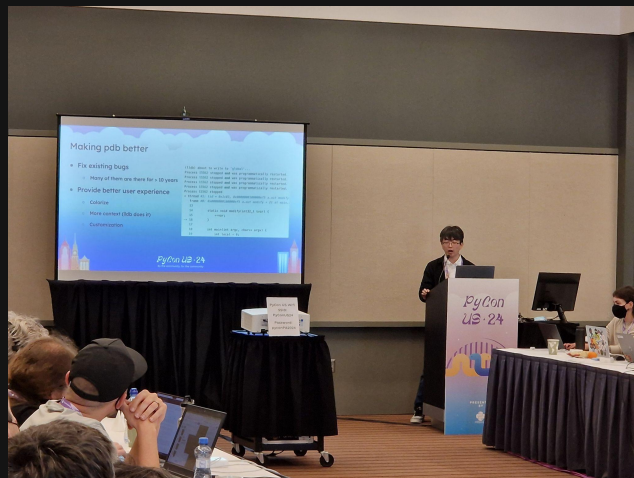
# 변경의 움직임

## pdb

“pdb의 주요 문제는 성능”

“sys.monitoring으로의 변경을 검토해야”

“현재 breakpoint는 최대 100배의 속도 저하가 발생하지만,  
sys.monitoring은 오버헤드가 거의 없다”



PART 5

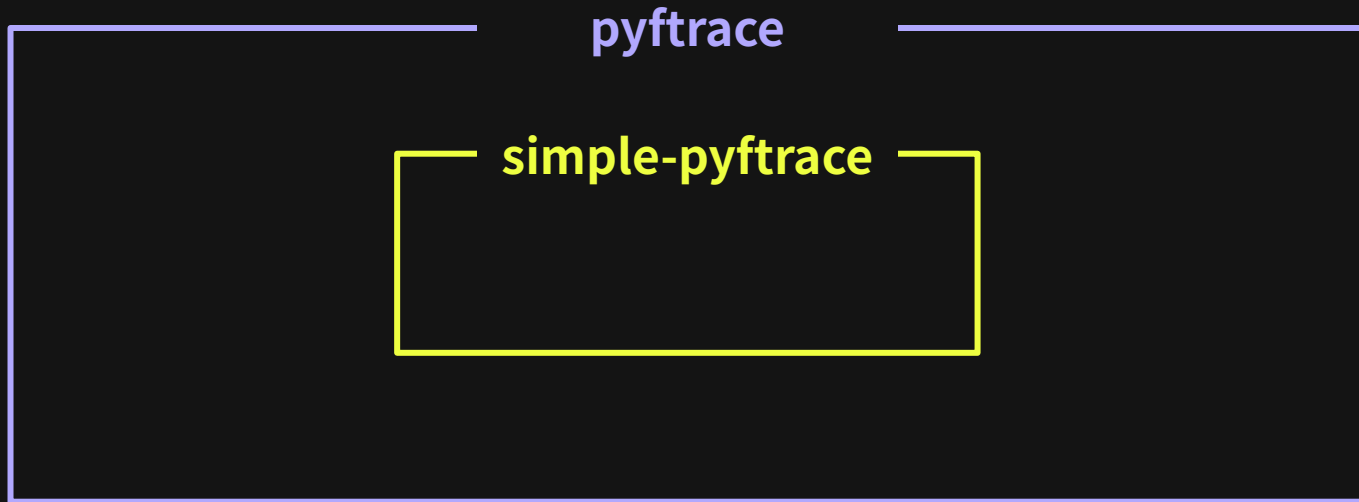
pyftrace: 간단한 함수 추적 도구 제작



<https://github.com/kangtegong/pyftrace>



# pyftrace: PEP 669 based function tracing tool



# pyftrace: PEP 669 based function tracing tool

## Commands & Options:

```
$ python3 simple-pyftrace.py path/to/python/script.py
```

e.g.)

```
$ python3.12 simple-pyftrace.py examples/foobar.py
```

# pyftrace: PEP 669 based function tracing tool

## Commands & Options:

\$ python3 simple-pyftrace.py **path/to/python/script.py**

```
1 def foo():  
2     bar()  
3     return  
4  
5 def bar():  
6     return 20  
7  
8 foo()  
9
```

```
Running script: foobar.py  
Called foo:1 from line 8  
    Called bar:5 from line 2  
        Returning bar-> 20  
    Returning foo-> None
```

# pyftrace: PEP 669 based function tracing tool

## Commands & Options:

\$ python3 simple-pyftrace.py **path/to/python/script.py**

```
1 def foo():  
2     bar()  
3     return  
4  
5 def bar():  
6     return 20  
7  
8 foo()  
9
```

```
Running script: foobar.py  
Called foo:1 from line 8  
    Called bar:5 from line 2  
        Returning bar-> 20  
    Returning foo-> None
```

# pyftrace: PEP 669 based function tracing tool

## Commands & Options:

\$ python3 simple-pyftrace.py **path/to/python/script.py**

```
1 def foo():  
2     bar()  
3     return  
4  
5 def bar():  
6     return 20  
7  
8 foo()  
9
```

```
Running script: foobar.py  
Called foo:1 from line 8  
    Called bar:5 from line 2  
        Returning bar-> 20  
    Returning foo-> None
```



# pyftrace: PEP 669 based function tracing tool

## Commands & Options:

\$ python3 simple-pyftrace.py **path/to/python/script.py**

```
1 def foo():  
2     bar()  
3     return  
4  
5 def bar():  
6     return 20  
7  
8 foo()  
9
```

```
Running script: foobar.py  
Called foo:1 from line 8  
    Called bar:5 from line 2  
        Returning bar-> 20  
    Returning foo-> None
```

# pyftrace: PEP 669 based function tracing tool

## Commands & Options:

\$ python3 simple-pyftrace.py **path/to/python/script.py**

```
1 def foo():  
2     bar()  
3     return  
4  
5 def bar():  
6     return 20  
7  
8 foo()  
9
```

```
Running script: foobar.py  
Called foo:1 from line 8  
    Called bar:5 from line 2  
        Returning bar-> 20  
Returning foo-> None
```

# pyftrace: PEP 669 based function tracing tool

```
def foo():  
    bar()  
    return 10
```

```
def bar():  
    return 20
```

```
foo()
```



# pyftrace: PEP 669 based function tracing tool

```
def foo():  
    bar()  
    return 10
```

```
def bar():  
    return 20
```

```
foo()
```



# pyftrace: PEP 669 based function tracing tool

```
sys.monitoring.register_callback(  
    self.tool_id,  
    sys.monitoring.events.CALL,  
    self.monitor_call  
)
```

```
sys.monitoring.register_callback(  
    self.tool_id,  
    sys.monitoring.events.PY_RETURN,  
    self.monitor_return  
)
```

# pyftrace: PEP 669 based function tracing tool

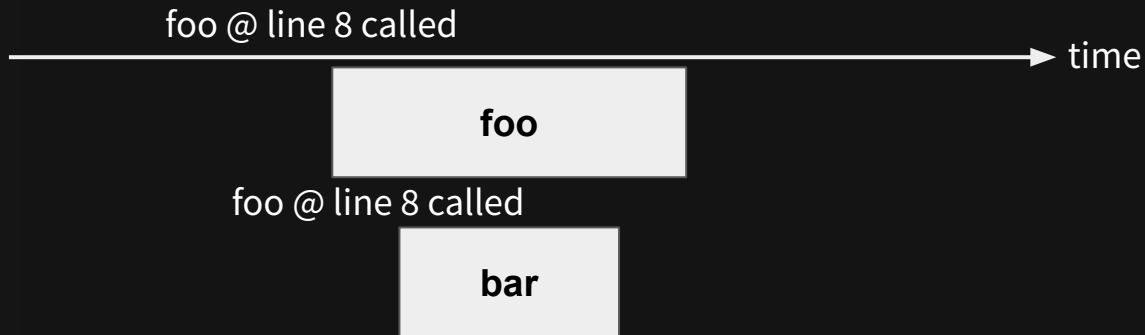
```
sys.monitoring.register_callback(  
    self.tool_id,  
    sys.monitoring.events.CALL,  
    self.monitor_call  
)  
  
sys.monitoring.register_callback(  
    self.tool_id,  
    sys.monitoring.events.PY_RETURN,  
    self.monitor_return  
)
```

# pyftrace: PEP 669 based function tracing tool

```
def foo():  
    bar()  
    return 10
```

```
def bar():  
    return 20
```

```
foo()
```



# pyftrace: PEP 669 based function tracing tool

```
def monitor_call(self, code, instruction_offset, callable_obj, arg0):  
    call_lineno = self.get_line_number(code, instruction_offset)  
    func_name = callable_obj.__name__  
    ...  
    print(f"{indent}Called {func_info} from line {call_lineno}")
```



# pyftrace: PEP 669 based function tracing tool

```
def foo():  
    bar()  
    return 10
```

```
def bar():  
    return 20
```

```
foo()
```



# pyftrace: PEP 669 based function tracing tool

## 주요 코드 해설

```
def monitor_return(self, code, instruction_offset, retval):  
    ...  
    print(f"{indent}Returning {code.co_name}-> {retval}")
```

# pyftrace: PEP 669 based function tracing tool

```
$ python3 simple-pyftrace.py --report path/to/python/script.py
```

```
$ python3.12 simple-pyftrace.py --report examples/foobar.py  
Running script: foobar.py
```

Function Name	Total Execution Time	Call Count
foo	0.000029 seconds	1
bar	0.000008 seconds	1

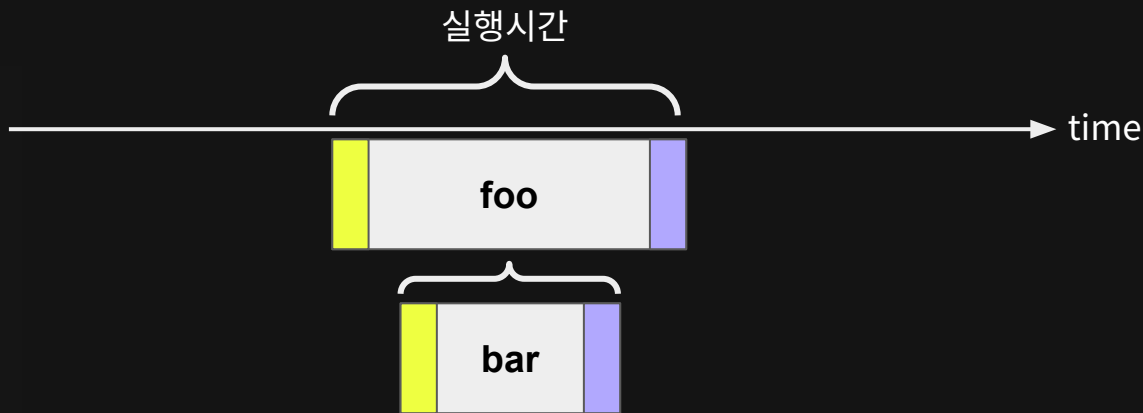
# pyftrace: PEP 669 based function tracing tool

## Commands & Options:

```
def foo():  
    bar()  
    return 10
```

```
def bar():  
    return 20
```

```
foo()
```



{RETURN} timestamp - {CALL} timestamp

{Some Function:count++}

# pyftrace: PEP 669 based function tracing tool

## 주요 코드 해설

```
execution_report = {}

...

if self.report_mode and func_name in self.execution_report:
    start_time, total_time, call_count = self.execution_report[func_name]
    exec_time = time.time() - start_time
    self.execution_report[func_name] = (
        start_time,
        total_time + exec_time,
        call_count + 1
    )
```

# pyftrace: PEP 669 based function tracing tool

## 주요 코드 해설

```
execution_report = {}
```

```
...
```

```
if self.report_mode and func_name in self.execution_report:
    start_time, total_time, call_count = self.execution_report[func_name]
    exec_time = time.time() - start_time
    self.execution_report[func_name] = (
        start_time,
        total_time + exec_time,
        call_count + 1
    )
```

# pyftrace: PEP 669 based function tracing tool

## 주요 코드 해설

```
execution_report = {}
```

```
...
```

```
if self.report_mode and func_name in self.execution_report:
    start_time, total_time, call_count = self.execution_report[func_name]
    exec_time = time.time() - start_time
    self.execution_report[func_name] = (
        start_time,
        total_time + exec_time,
        call_count + 1
    )
```

# pyftrace: PEP 669 based function tracing tool

## 주요 코드 해설

```
execution_report = {}
```

```
...
```

```
if self.report_mode and func_name in self.execution_report:
    start_time, total_time, call_count = self.execution_report[func_name]
    exec_time = time.time() - start_time
    self.execution_report[func_name] = (
        start_time,
        total_time + exec_time,
        call_count + 1
    )
```



# pyftrace: PEP 669 based function tracing tool

## 주요 코드 해설

```
execution_report = {}
```

```
...
```

```
if self.report_mode and func_name in self.execution_report:
    start_time, total_time, call_count = self.execution_report[func_name]
    exec_time = time.time() - start_time
    self.execution_report[func_name] = (
        start_time,
        total_time + exec_time,
        call_count + 1
    )
```

# pyftrace: PEP 669 based function tracing tool

## 주요 코드 해설

```
execution_report = {}
```

```
...
```

```
if self.report_mode and func_name in self.execution_report:
    start_time, total_time, call_count = self.execution_report[func_name]
    exec_time = time.time() - start_time
    self.execution_report[func_name] = (
        start_time,
        total_time + exec_time,
        call_count + 1
    )
```

Thank You!

Q&A



<https://github.com/kangtegong/pyftrace>

[tegongkang@gmail.com](mailto:tegongkang@gmail.com)  
<https://minchul.net>