Python 3.12's new monitoring and profiling API

Johannes Bechberger
mostlynerdless.de

(033) PRO 2 2.130476415

correct 2.130676415

Relays 6-2 in 033 failed special speed test

In relay " 10,000 test.

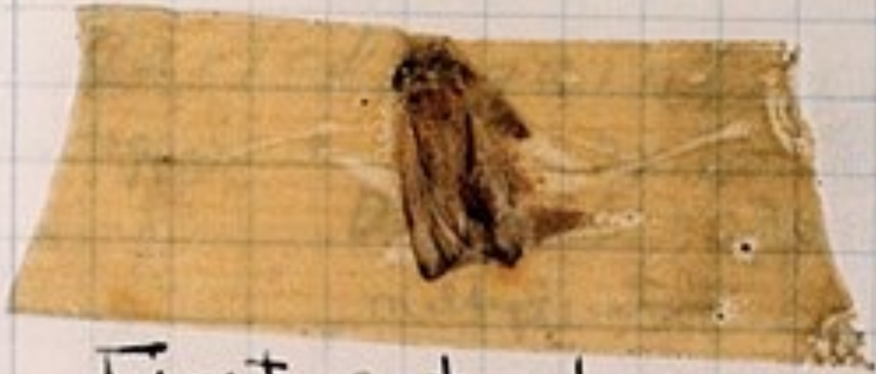
Relays changed

Relay 2145

Relay 3370

1100 Started Cosine Tape (Sine check)

1525 Started Mult + Adder Test.

1545 

Relay #70 Panel F

(moth) in relay.

First actual case of bug being found.

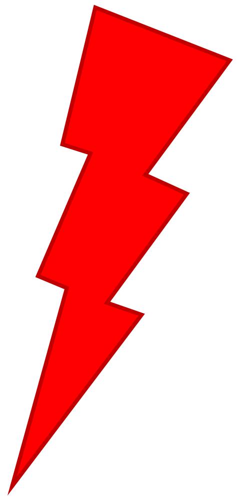
~~143~~ 1630 Antangent started.

1700 closed down.

“ If debugging is the process of removing software bugs, then programming must be the process of putting them in.

— Edsger Dijkstra

```
➜ python3 counter.py \
          lines counter.py

0
```



```
➜ python3 counter.py \
         lines counter.py

26
```

Let’s look at the code

```python
def main():

    match cmd := sys.argv[1]:
        case "lines":
            count = count_code_lines(Path(sys.argv[2]))
            print(count)
        case "help":
            print_help()
        case _:
            raise ValueError(f"Unknown operation {cmd}")
```

```python
def is_code_line(line: str) -> bool:
    return line.isspace() and line.strip().startswith("#")


def count_code_lines(file: Path) -> int:
    count = 0
    with file.open('r') as f:
        for line in f:
            if is_code_line(line):
                count += 1
    return count
```

Any ideas?

Debuggers are your friend

jar profiler

Sap
Machine

Why do we need a monitoring API?

Java has built-in debugging support…

But Python?

Does the interpreter "know" breakpoints?

No.

Any ideas?

```python
def is_code_line(line: str) -> bool:
   dbg();return line.isspace() and line.strip().startswith("#


def count_code_lines(file: Path) -> int:
   dbg();count = 0
   dbg();with file.open('r') as f:
           dbg();for line in f:
                    dbg();if is_code_line(line):
                               dbg();count += 1
   dbg();return count
```

dbg(); line

```python
def dbg():
    if at_breakpoint(file, line):
        dbg_shell()
```

?

`sys._getframe`

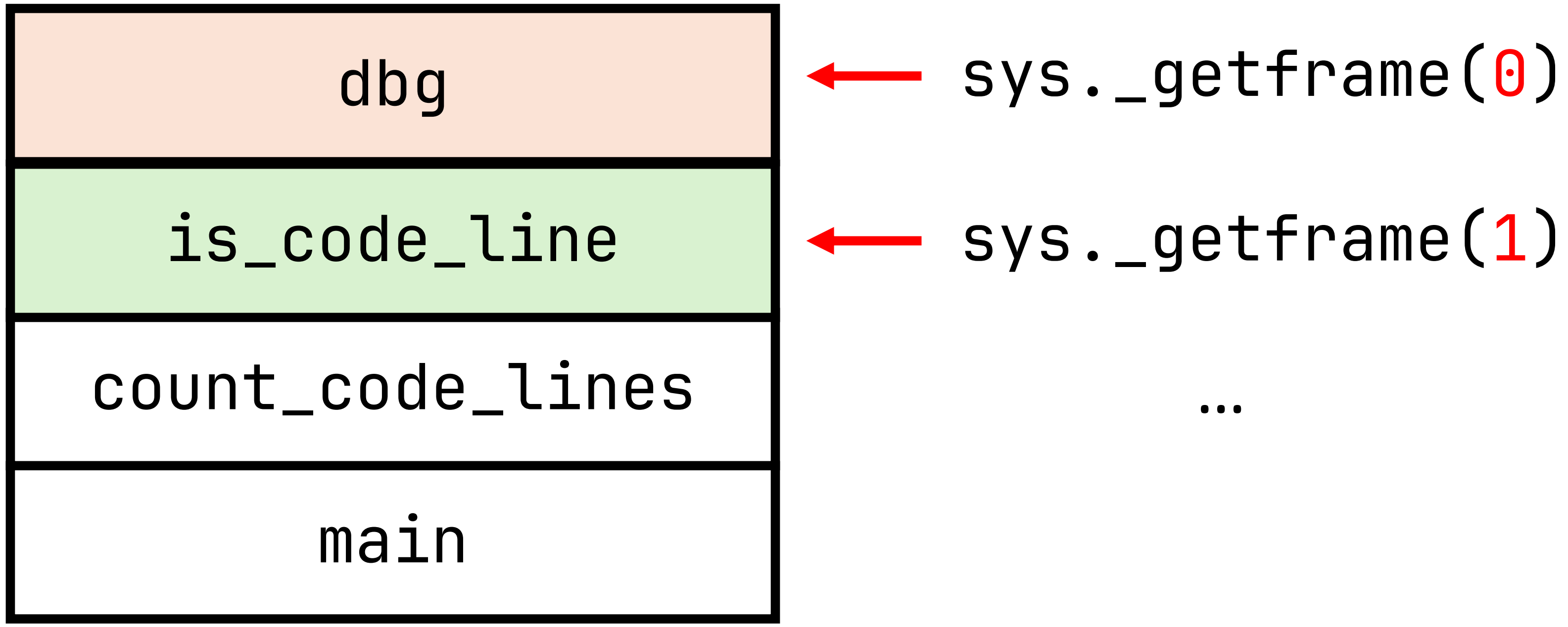
```
sys._getframe
```

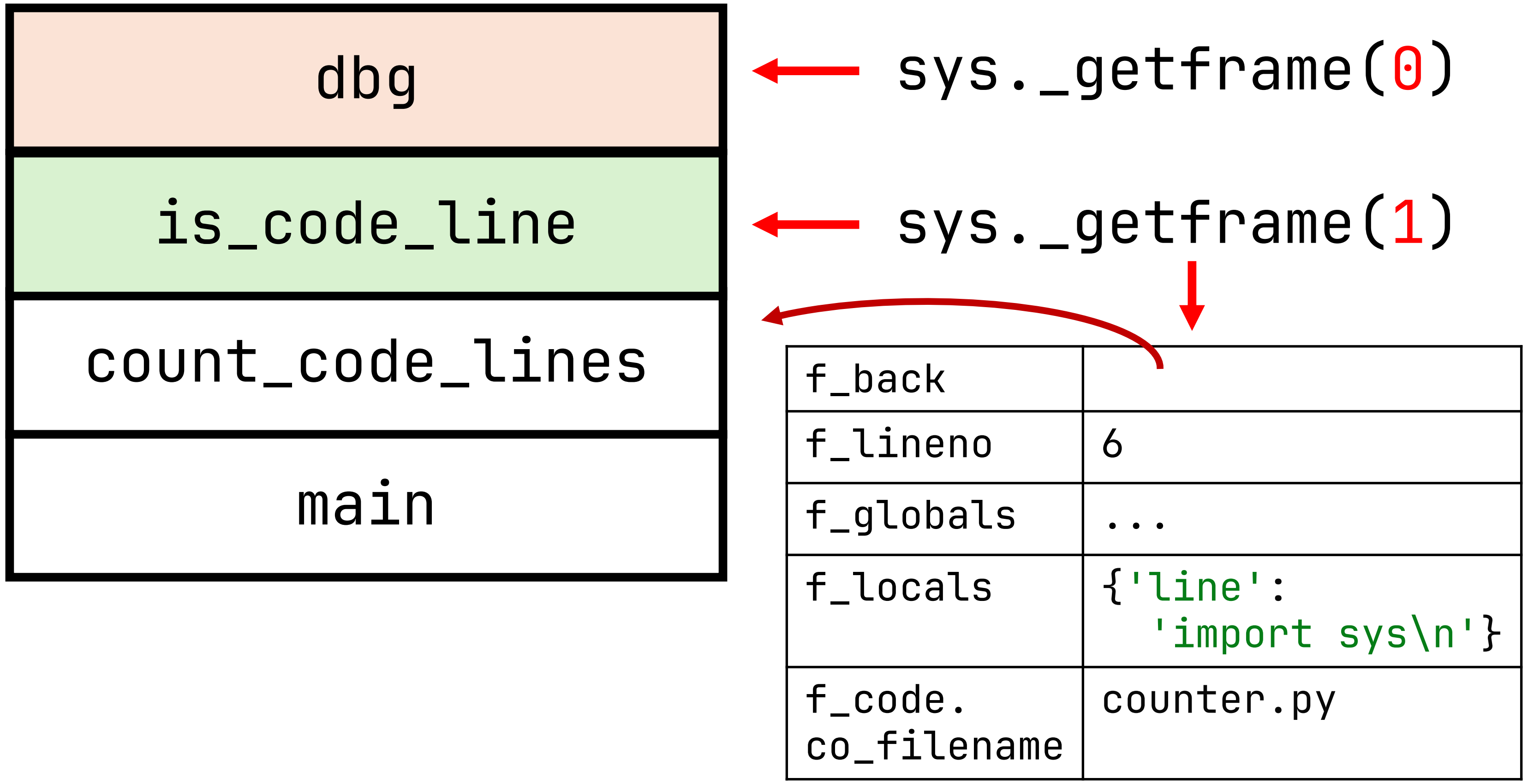
sys._getframe

CPython implementation detail

“ locals(), globals(), sys._getframe(), sys.exc_info(), and sys.settrace **work in PyPy, but they incur a performance penalty that can be huge by disabling the JIT over the enclosing JIT scope.**

– https://www.pypy.org/performance.html


dbg
is_code_line
count_code_lines
main
sys._getframe(0)
sys._getframe(1)
...



f_back	
f_lineno	6
f_globals	...
f_locals	{'line': 'import sys\n'}
f_code.co_filename	counter.py

dbg(); line

```
def dbg():



    if at_breakpoint(file, line):
        dbg_shell()
```

dbg(); line

```python
def dbg():
    frame = sys._getframe(1)
    line = frame.f_lineno
    file = Path(frame.f_code.co_filename).stem
    if at_breakpoint(file, line):
        dbg_shell(frame)
```

dbg(); line

```python
def dbg():
    frame = sys._getframe(1)
    line = frame.f_lineno
    file = Path(frame.f_code.co_filename).stem
    if at_breakpoint(file, line):
        dbg_shell(frame)
```

dbg(); line

```python
def dbg():
    frame = sys._getframe(1)
    line = frame.f_lineno
    file = Path(frame.f_code.co_filename).stem
    if at_breakpoint(file, line):
        dbg_shell(frame)


def at_breakpoint(file: str, line: int) -> bool:
    return file == "counter" and line == 6
```

But how do we automate this?

The pre-3.12 way

sys.settrace

sys.settrace(handler)

```python
Event = Union['call', 'line', 'return', 'exception', 'opcode']


def handler(frame: FrameType, event: Event, arg):
    pass
```

```
                                        handler(frame, 'call', None)

def is_code_line(line: str) -> bool:
    return line.isspace() and line.strip().startswith("#")



def count_code_lines(file: Path) -> int:
    count = 0
    with file.open('r') as f:                handler(frame, 'call', None)
        for line in f:
            if is_code_line(line):
                count += 1
    return count
```

sys.settrace(handler)

```python
def handler(frame: FrameType, event: Event, arg) \
        -> Optional[Callable[[FrameType, Event, Any], None]]:
    return inner_handler
```

sys.settrace(handler)

```python
def inner_handler(frame: FrameType, event: Event, arg):
    pass


def handler(frame: FrameType, event: Event, arg) \
        -> Optional[Callable[[FrameType, Event, Any], None]]:
    return inner_handler
```

dbg(); line

```python
def dbg():

    frame = sys._getframe(1)
    line = frame.f_lineno
    file = Path(frame.f_code.co_filename).stem
    if at_breakpoint(file, line):
        dbg_shell(frame)


def at_breakpoint(file: str, line: int) -> bool:
    return file == "counter" and line == 6
```

dbg(); line

```
def inner_handler(frame: FrameType, event: str, arg):


    line = frame.f_lineno
    file = Path(frame.f_code.co_filename).stem
    if at_breakpoint(file, line):
        dbg_shell(frame)


def at_breakpoint(file: str, line: int) -> bool:
    return file == "counter" and line == 6
```

dbg(); line

```python
def inner_handler(frame: FrameType, event: str, arg):
    if event != 'line':
        return
    line = frame.f_lineno
    file = Path(frame.f_code.co_filename).stem
    if at_breakpoint(file, line):
        dbg_shell(frame)


def at_breakpoint(file: str, line: int) -> bool:
    return file == "counter" and line == 6
```

Do we get line events for every function?

```
handler(frame, 'call', None)
```

```python
def is_code_line(line: str) -> bool:
    return line.isspace() and line.strip().startswith("#")
```

add breakpoint

```python
def count_code_lines(file: Path) -> int:
    count = 0
    with file.open('r') as f:
        for line in f:
            if is_code_line(line):
                count += 1
    return count
```

```
handler(frame, …, None)
```

This is slow, so…

Add

Add a new API

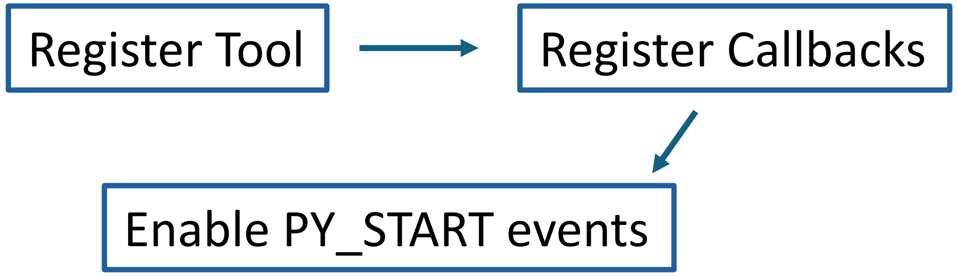
Python 3.12
and PEP 669

PEP 669 – Low Impact Monitoring for CPython

Author:	Mark Shannon <mark at hotpy.org>
Discussions-To:	Discourse thread
Status:	Accepted
Type:	Standards Track
Created:	18-Aug-2021
Python-Version:	3.12
Post-History:	07-Dec-2021, 10-Jan-2022
Resolution:	Discourse message

Register Tool

```python
# some aliases and constants
mon = sys.monitoring
E = mon.events
TOOL_ID = mon.DEBUGGER_ID

# register the tool
mon.use_tool_id(TOOL_ID, "dbg")
```



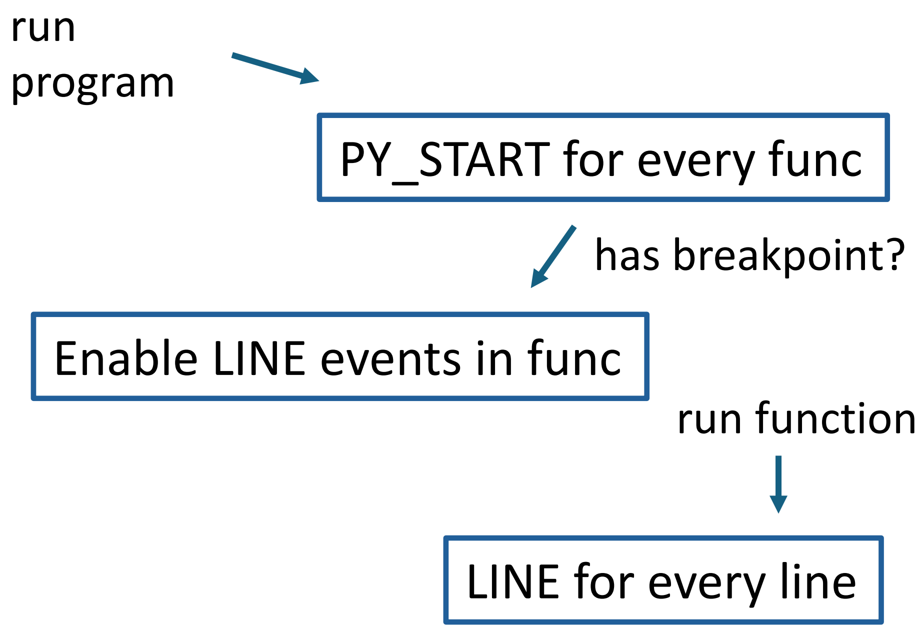
```python
# some aliases and constants
mon = sys.monitoring
E = mon.events
TOOL_ID = mon.DEBUGGER_ID

# register the tool
mon.use_tool_id(TOOL_ID, "dbg")

# register callbacks for the events we are interested in
mon.register_callback(TOOL_ID, E.LINE, line_handler)
mon.register_callback(TOOL_ID, E.PY_START, start_handler)

def start_handler(code: CodeType, offset: int):
    pass

def line_handler(code: CodeType, line: int) -> DISABLE|Any:
    pass
```

disable till
mon.restart_eve

```python
# some aliases and constants
mon = sys.monitoring
E = mon.events
TOOL_ID = mon.DEBUGGER_ID

# register the tool
mon.use_tool_id(TOOL_ID, "dbg")

# register callbacks for the events we are interested in
mon.register_callback(TOOL_ID, E.LINE, line_handler)
mon.register_callback(TOOL_ID, E.PY_START, start_handler)
```

disable till
`mon.restart_events()`

```python
def start_handler(code: CodeType, offset: int):
    pass

def line_handler(code: CodeType, line: int) -> DISABLE|Any:
    pass
```



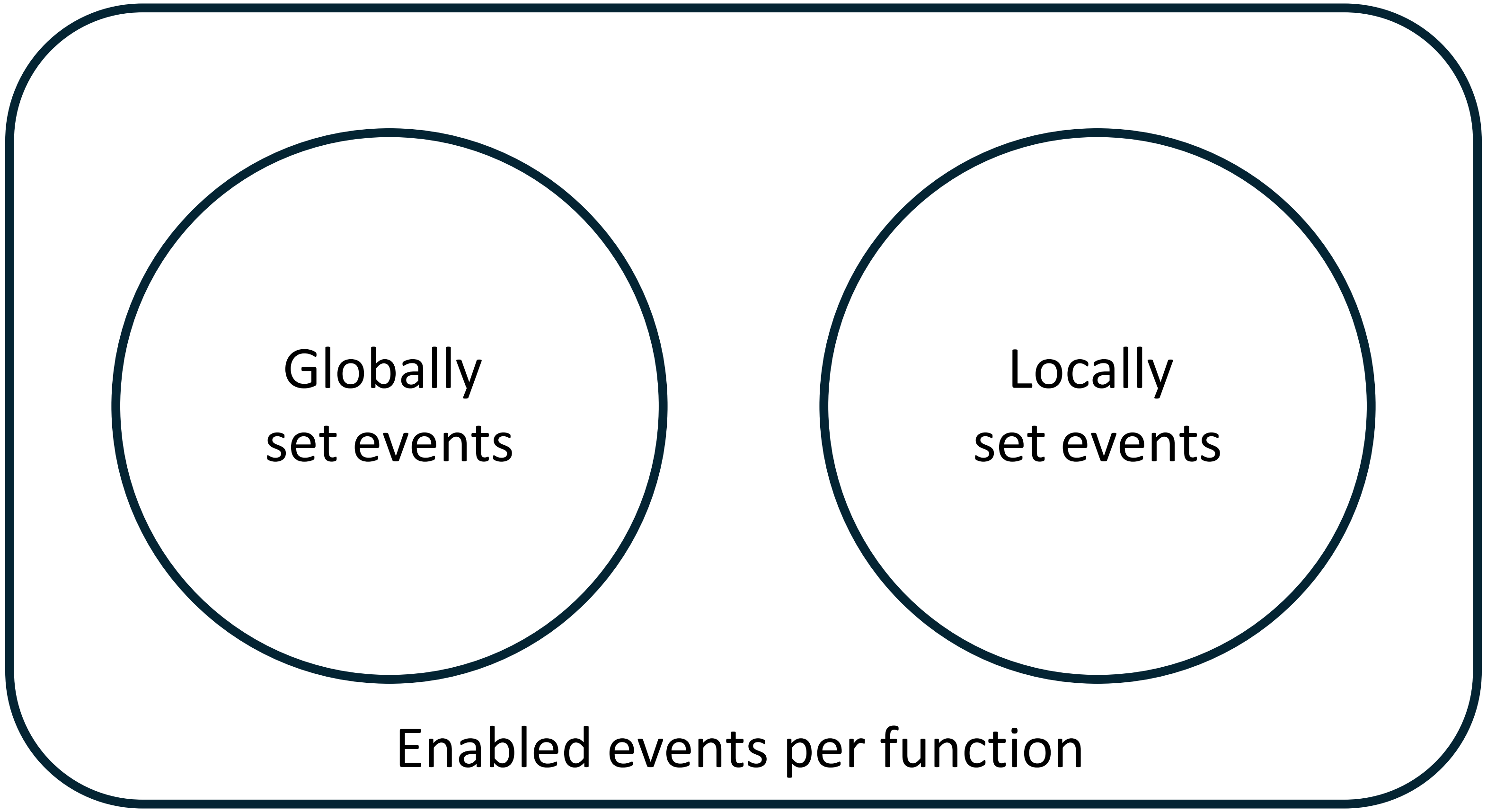
```python
# some aliases and constants
mon = sys.monitoring
E = mon.events
TOOL_ID = mon.DEBUGGER_ID

# register the tool
mon.use_tool_id(TOOL_ID, "dbg")

# register callbacks for the events we are interested in
mon.register_callback(TOOL_ID, E.LINE, line_handler)
mon.register_callback(TOOL_ID, E.PY_START, start_handler)

# enable PY_START event globally
mon.set_events(TOOL_ID, E.PY_START)

# Later
mon.set_local_events(TOOL_ID, code, E.LINE)
```

emitted per thread,
not per interpreter

“ The biggest opportunity of PEP 669 isn't even the speed, it's the fact that a debugger built on top of it will **automatically support all threads**.

— Łukasz Langa

https://github.com/python/cpython/issues/103103#issuecomment-1488312628

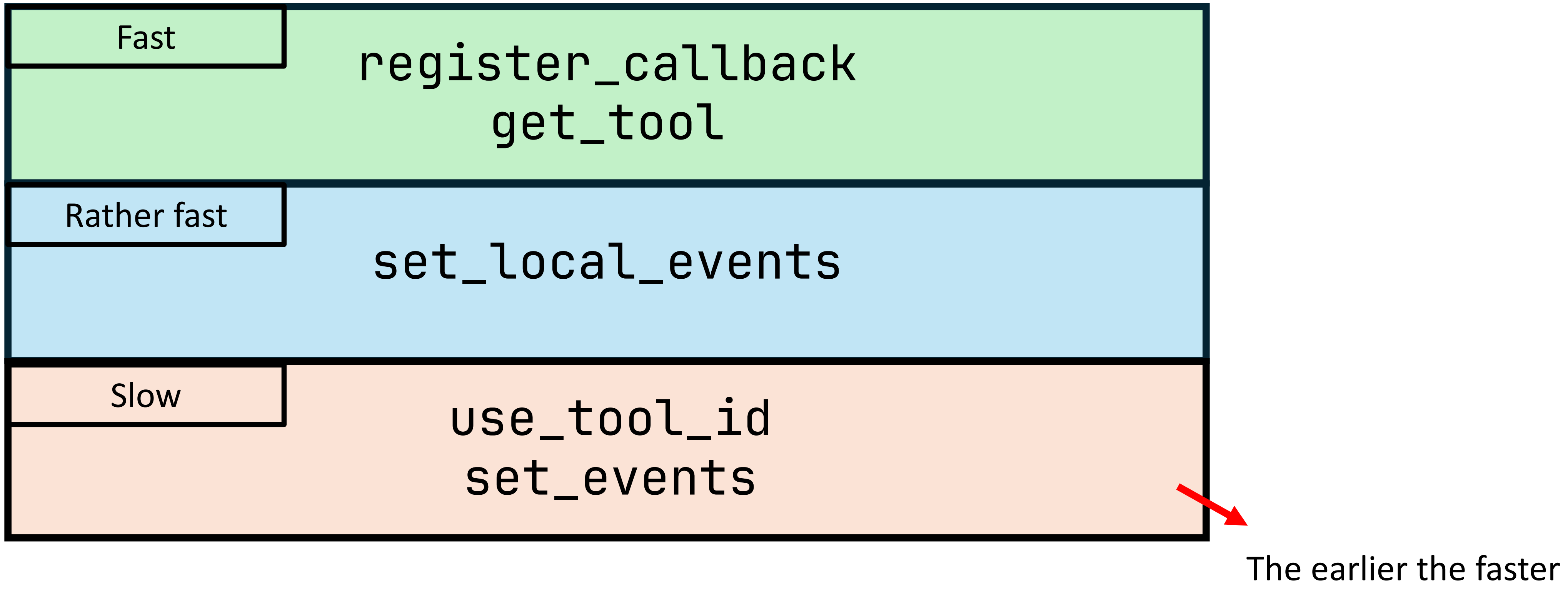

Globally set events
Locally set events
Enabled events per function

The power is in the fine-grained configuration

You can set events in f for f

```python
def line_handler(code: CodeType, line_number: int):
    print(f"  {code.co_name}: {line_number}")

mon.register_callback(tool_id, E.LINE, line_handler)

def f():
    print("hello")
    mon.set_local_events(tool_id, f.__code__, E.LINE)
    print("inner")
    mon.set_local_events(tool_id, f.__code__, 0)
    print("end")

f()

# Output
hello
  f: 18
inner
  f: 19
end
```

What's fast?


Fast
register_callback
get_tool
Rather fast
set_local_events
Slow
use_tool_id
set_events
The earlier the faster

Back to the debugger

```python
def start_handler(code: CodeType, _: int):
    # ... handle first call

    file = Path(code.co_filename).stem
    if has_breakpoint(file, code.co_firstlineno,
                      len(list(code.co_lines()))):
        print(f"enable line events for {code.co_name}")
        enable_line_events(code)
    print(f"start {code.co_name}")
```

```python
def line_handler(code: CodeType, line: int):
    print(f"line {line} in {code.co_name}")
    if at_breakpoint(code.co_name, line):
        print(f"in break point at line {line}")
        dbg_shell(sys._getframe(1))
```

Event kinds

Local Events

```
PY_START
PY_RESUME
PY_RETURN
PY_YIELD
CALL
LINE
INSTRUCTION
JUMP
BRANCH
STOP_ITERATION
```

controls

Ancillary Events

```
C_RAISE
C_RETURN
```

Other Events

```
PY_RAISE
PY_UNWIND
PY_THROW
EXCEPTION_
 HANDLED
```

not tied to specific location

Performance

Hacking pyperformance
for fun and profit...

```python
def inner_handler(*args):
    pass


def handler(*args):
    return inner_handler

sys.settrace(handler)
```

VS

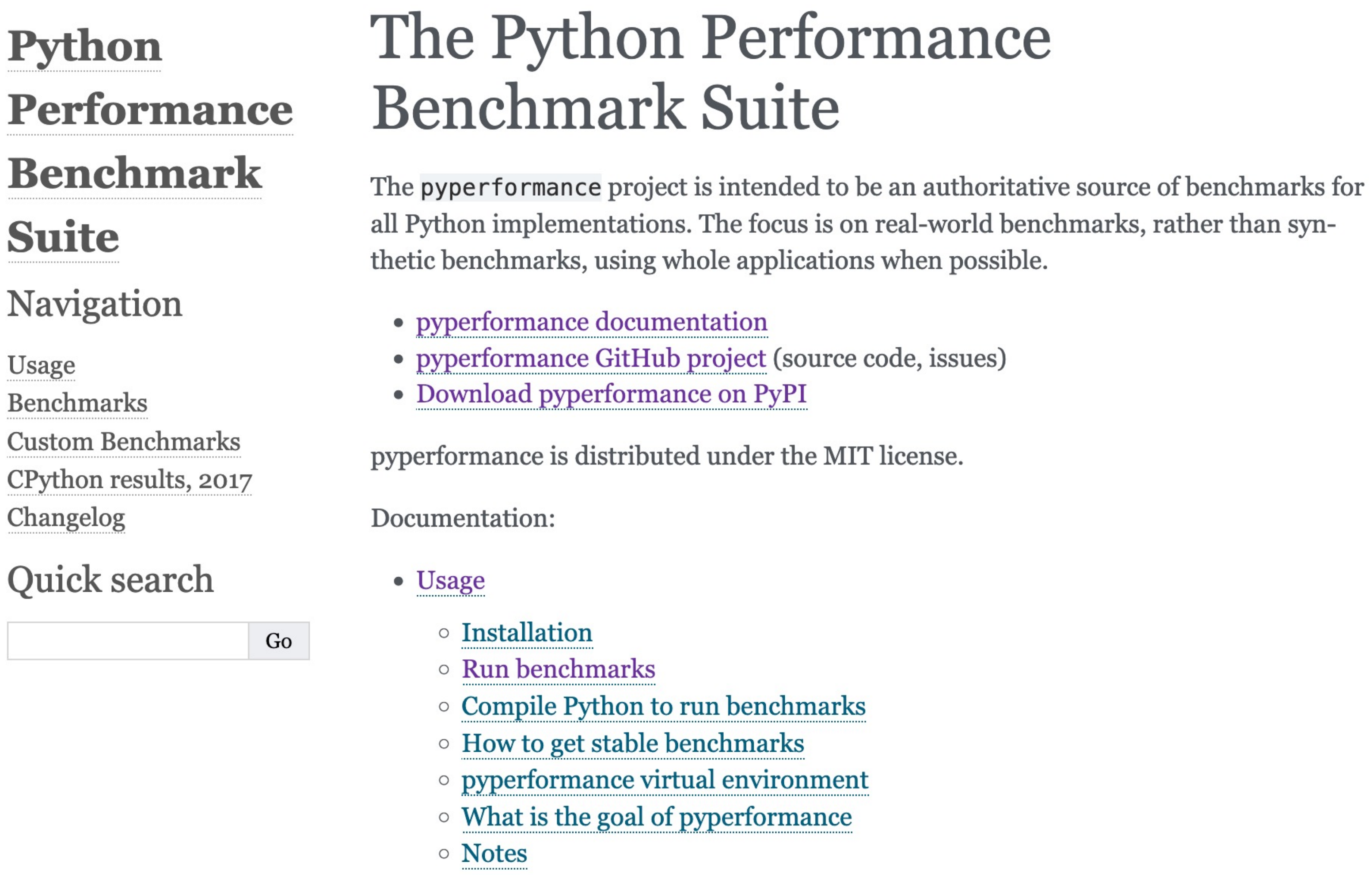
```python
def line_handler(*args):
    pass

def start_handler(*args):
    pass


mon.use_tool_id(TOOL_ID, "dbg")
mon.register_callback(...)
mon.set_events(TOOL_ID,
  E.PY_START)
```

VS

```python
mon.set_events(TOOL_ID,
  E.PY_START | E.LINE)
```

sys.settrace

monitoring



Python Performance Benchmark Suite

Navigation

- Usage
- Benchmarks
- Custom Benchmarks
- CPython results, 2017
- Changelog

Quick search

Go

The Python Performance Benchmark Suite

The `pyperformance` project is intended to be an authoritative source of benchmarks for all Python implementations. The focus is on real-world benchmarks, rather than synthetic benchmarks, using whole applications when possible.

- pyperformance documentation
- pyperformance GitHub project (source code, issues)
- Download pyperformance on PyPI

pyperformance is distributed under the MIT license.

Documentation:

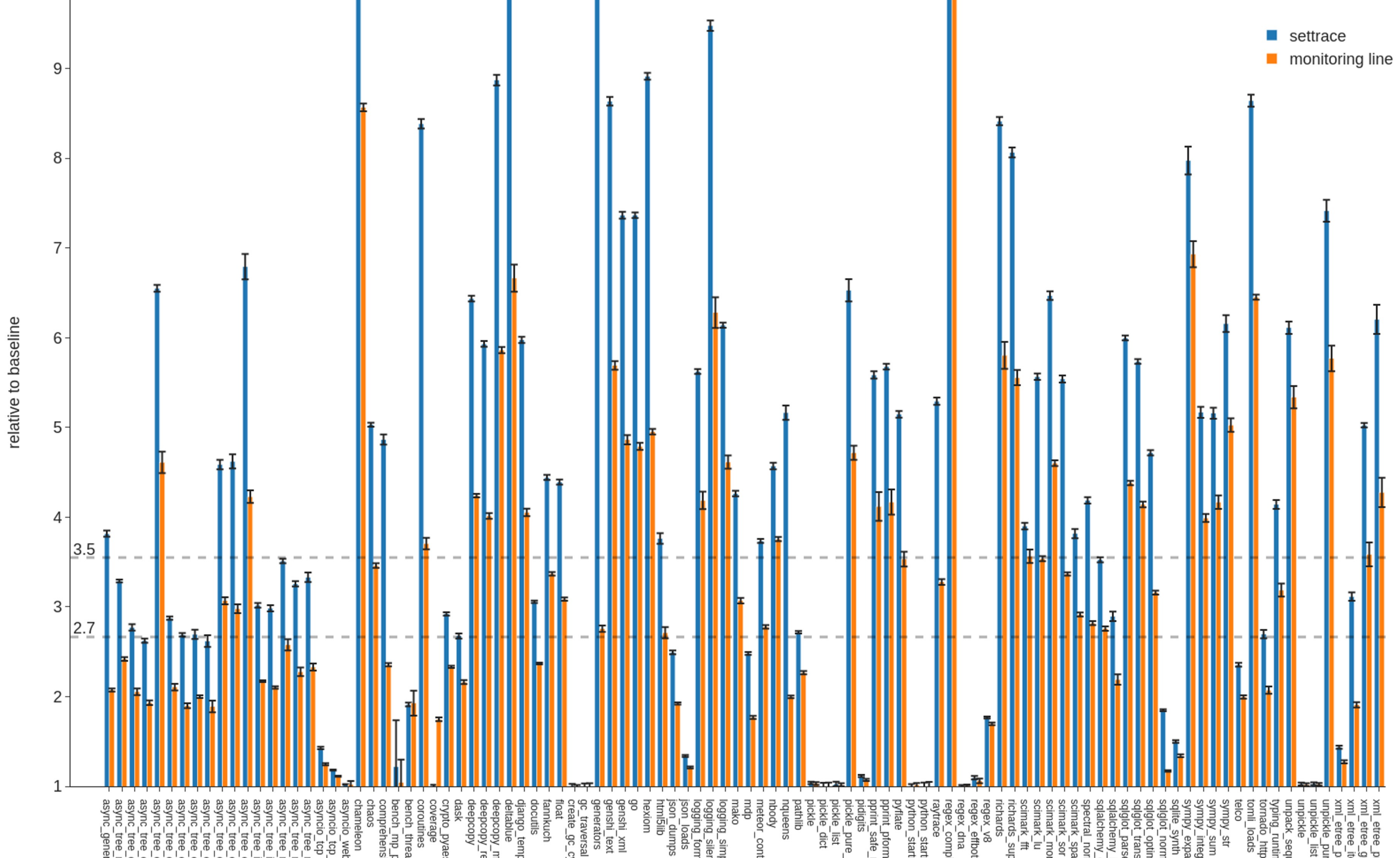
- Usage
  - Installation
  - Run benchmarks
  - Compile Python to run benchmarks
  - How to get stable benchmarks
  - pyperformance virtual environment
  - What is the goal of pyperformance
  - Notes

modified version: https://github.com/parttimenerd/pyperformance/tree/dbg

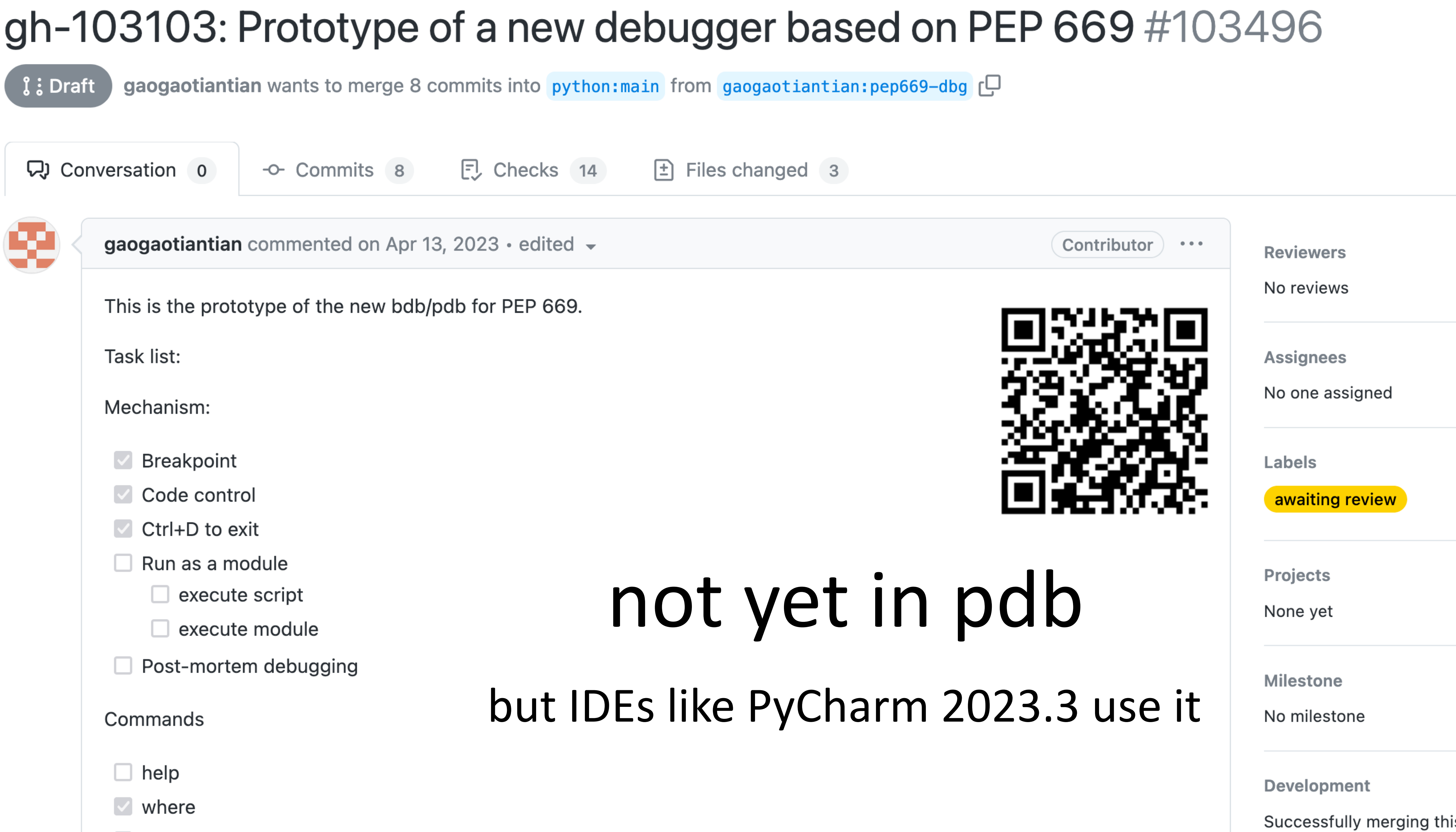
3.5x runtime **VS** 1.2x runtime

VS 2.7x runtime

sys.settrace monitoring


settrace
monitoring line
relative to baseline
1
2
3
4
5
6
7
8
9
2.7
3.5
async_generators
chameleon
chaos
coroutines
coverage
crypto_pyaes
dask
deepcopy
deltablue
docutils
fannkuch
float
gc_traversal
generators
genshi_text
genshi_xml
go
hexiom
html5lib
json_dumps
json_loads
mako
mdp
nbody
nqueens
pathlib
pickle
pickle_dict
pickle_list
pidigits
pyflate
raytrace
regex_dna
regex_v8
richards
scimark_fft
scimark_lu
sympy_str
telco
tomli_loads
unpickle
unpickle_list
xml_etree_p

Is it used?



gh-103103: Prototype of a new debugger based on PEP 669 #103496

Draft gaogaotiantian wants to merge 8 commits into `python:main` from `gaogaotiantian:pep669-dbg`

Conversation 0 · Commits 8 · Checks 14 · Files changed 3

gaogaotiantian commented on Apr 13, 2023 • edited · Contributor

This is the prototype of the new bdb/pdb for PEP 669.

Task list:

Mechanism:

- [x] Breakpoint
- [x] Code control
- [x] Ctrl+D to exit
- [ ] Run as a module
  - [ ] execute script
  - [ ] execute module
- [ ] Post-mortem debugging

Commands

- [ ] help
- [x] where

Reviewers
No reviews

Assignees
No one assigned

Labels
awaiting review

Projects
None yet

Milestone
No milestone

Development
Successfully merging thi

not yet in pdb

but IDEs like PyCharm 2023.3 use it

“

After #103082, we will have the chance to “build a much faster debugger. For breakpoints, we do not need to trigger trace function all the time and checking for the line number. [...]

The bad news is - it's almost impossible to do a completely backward compatible transition because the mechanism is quite different.

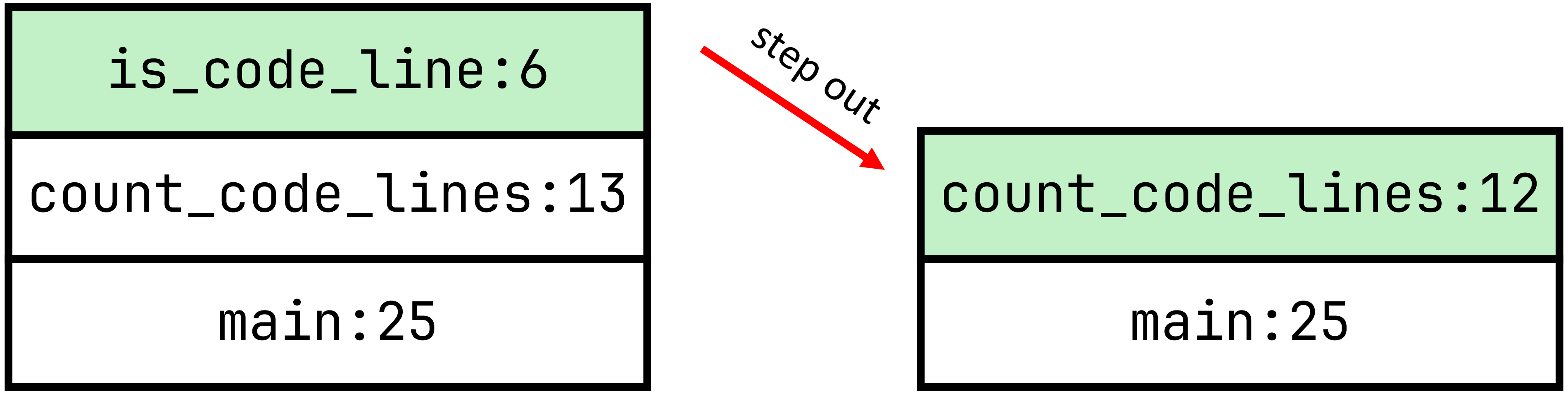
— Tian Gao

https://github.com/python/cpython/issues/103103#issue-1644836791

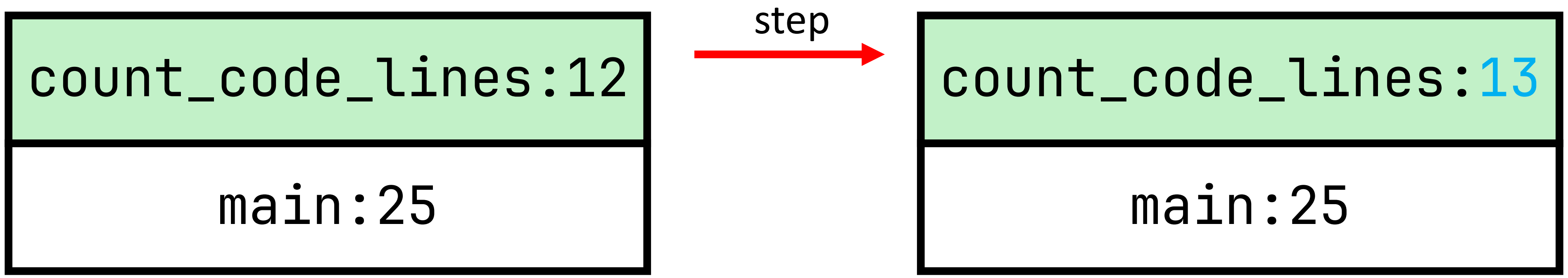
Addendum

Single Stepping

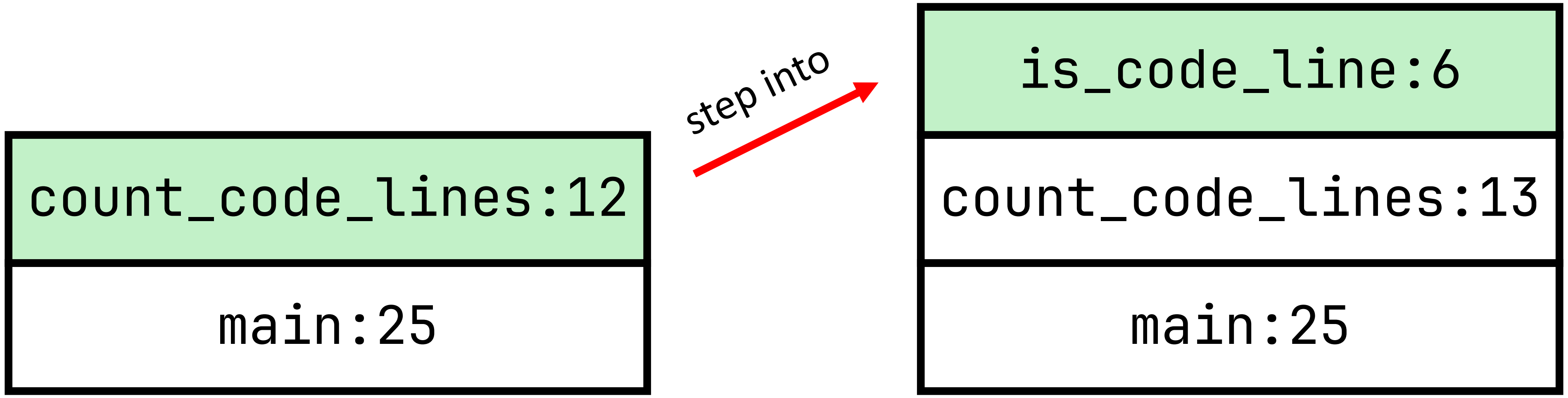
Just extend `at_breakpoint`



Just extend `at_breakpoint`



Just extend `at_breakpoint`



@parttimen3rd on Twitter
parttimenerd on GitHub
mostlynerdless.de

@SweetSapMachine
sapmachine.io