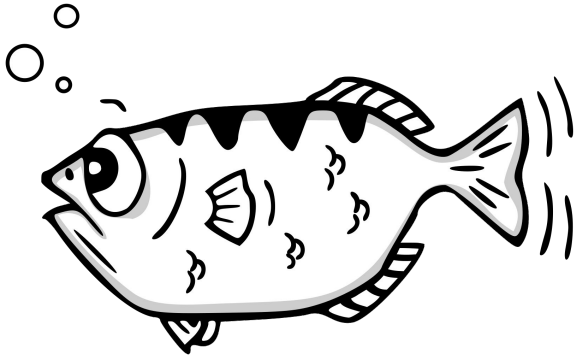


GDB for CPython

“For a fish, the archer fish is known to shoot down bugs from low hanging plants by spitting water at them.” - Jamie Guinan, <GDB's mascot?> Wed, 18 Jul 2001

graymon🏳️‍🌈

PYCHINA
Python 中国社区

1: 97% of 212k reviews are 👍
2: my favorite turn-based tactics
3: GOTY 2023

gdb -p $(pidof python)

pdb -p $(pidof python)?

PEP-768

```
env:
1. OS: 6.14.0-29-generic 24.04.1-Ubuntu
2. gdb --version: 15.0.50.20240403-git
3. python3.13-dbg --version: 3.13.7
4. docker run -d python:3.14.0rc2 python --version: 3.14.0rc2
```

```
gdb -p $(pidof python)
```

In case you haven’t used gdb 🙂

```c
#include <stdio.h>
int h(int a,int b){ return a/b; }
int g(int x){ int y=0; return h(10,y); }
int f(int n){ return g(n)+1; }
int main(void){ printf("%d\n", f(42)); }
```

```
$ ./a.out
Floating point exception (core dumped)

$ gdb ./a.out core.125794

(gdb) bt
#0  0x000056df795f115b in h (a=10, b=0) at main.c:2
#1  0x000056df795f1185 in g (x=42) at main.c:3
#2  0x000056df795f11a0 in f (n=42) at main.c:4
#3  0x000056df795f11b7 in main () at main.c:6

(gdb) l
2    int h(int a,int b){ return a/b; }

(gdb) p b
$1 = 0
```

Bedrock1: “gdb” python module

```
$ gdb --configuration | grep python                                  [1]
        --with-python=/usr (relocatable)
        --with-python-libdir=/usr/lib (relocatable)

$ gdb -q
(gdb) python print("hello from the dart monkey")
hello from the dart monkey

(gdb) python print("0x{:x}".format(gdb.newest_frame().pc()))        [2]
0x792fab71b494

(gdb) p $pc
$1 = (void (*)()) 0x792fab71b494 <__GI___poll+20>
```

[1] Most gdbs are compiled with python support, feel free to use it in official python images and alpine images.

[2] “gdb” python module is auto-imported in python interpreter inside gdb session. It’s maintained as a part of gdb project: https://sourceware.org/git/?p=archer.git;a=tree;h=refs/heads/tromey/python;hb=refs/heads/tromey/python (Last commit was in 2017😢)

Bedrock2: gdb/libpython.py

```
// gdb native commands

(gdb) bt                    [1]
(gdb) list                  [2]
(gdb) up                    [3]
(gdb) down                  [4]
(gdb) print                 [5]
```



```
// libpython.py-defined commands

(gdb) py-bt/py-bt-full
(gdb) py-list
(gdb) py-up
(gdb) py-down
(gdb) py-print
```

[1] “bt” prints backtrace of all stack frames.
[2] “list” lists source code of specified function or line.
[3] “up” selects and print stack frame that called this one.
[4] “down” selects and print stack frame called by this one.
[5] “print” prints value of expression EXP.

They are maintained in cpython, based on “gdb” python module:
https://github.com/python/cpython/blob/main/Tools/gdb/libpython.py

Gdb detects target language, auto-import libpython.py for python process.

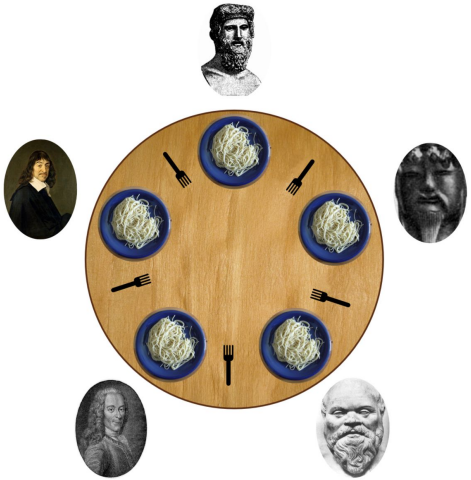
Showcase1: Dining philosophers problem

```
import threading

n = 5
forks = [threading.Lock() for _ in range(n)]


def phil(i):
    L, R = forks[i], forks[(i + 1) % n]
    while True:
        with L:
            with R:
                print(i, "eats")


ts = [threading.Thread(target=phil, args=(i,)) for i in range(n)]
[t.start() for t in ts]
[t.join() for t in ts]
```



Showcase1: Dining philosophers problem

```
$ sudo gdb -p $(pidof python3.13-dbg)

(gdb) bt                                                                                    [1]
#0  0x000076edd3098d71 in __futex_abstimed_wait_common64 (private=<optimized out>,
    cancel=true, abstime=0x0, op=393, expected=0, futex_word=0x7fff06e8bc50)
    at ./nptl/futex-internal.c:57
#1  __futex_abstimed_wait_common (cancel=true, private=<optimized out>, abstime=0x0,
    clockid=0, expected=0, futex_word=0x7fff06e8bc50) at ./nptl/futex-internal.c:87
#2  __GI___futex_abstimed_wait_cancelable64 (futex_word=futex_word@entry=0x7fff06e8bc50,
    expected=expected@entry=0, clockid=clockid@entry=0, abstime=abstime@entry=0x0,
    private=<optimized out>) at ./nptl/futex-internal.c:139
#3  0x000076edd30a4f1f in do_futex_wait (sem=sem@entry=0x7fff06e8bc50, abstime=0x0, clockid=0)
    at ./nptl/sem_waitcommon.c:111
#4  0x000076edd30a4fb8 in __new_sem_wait_slow64 (sem=0x7fff06e8bc50, abstime=0x0, clockid=0)
    at ./nptl/sem_waitcommon.c:183
#5  0x000076edd30a503d in __new_sem_wait (sem=<optimized out>) at ./nptl/sem_wait.c:42
#6  0x000000000062f36d in _PySemaphore_PlatformWait (sema=sema@entry=0x7fff06e8bc50,
    timeout=timeout@entry=-1) at ../Python/parking_lot.c:149
#7  0x000000000062f3f7 in _PySemaphore_Wait (sema=sema@entry=0x7fff06e8bc50,                [2]
    timeout=timeout@entry=-1, detach=detach@entry=1) at ../Python/parking_lot.c:220
```

[1] C level backtrace, not really helpful.

[2] Semaphore deadlock!

Showcase1: Dining philosophers problem

```
$ sudo gdb -p $(pidof python3.13-dbg)

(gdb) py-bt
Traceback (most recent call first):
  File "/usr/lib/python3.13/threading.py", line 1094, in join
    self._handle.join(timeout)
  File "/home/gray/src/github.com/jschwinger233/pycon2025/phi.py", line 17, in <module>
    [t.join() for t in ts]                                                    [1]

(gdb) thread apply all py-bt                                                  [2]

Thread 6 (Thread 0x77f2722ff6c0 (LWP 25327) "python3.13-dbg"):
Traceback (most recent call first):
  <built-in method __enter__ of _thread.lock object at remote 0x77f2723232b0>
  File "/home/gray/src/github.com/jschwinger233/pycon2025/phi.py", line 11, in phil
    with R:
  File "/usr/lib/python3.13/threading.py", line 994, in run
    self._target(*self._args, **self._kwargs)
  File "/usr/lib/python3.13/threading.py", line 1043, in _bootstrap_inner
    self.run()
  File "/usr/lib/python3.13/threading.py", line 1014, in _bootstrap
    self._bootstrap_inner()                                                   [3]
```

[1] Python VM level backtrace. By default it only handles main thread.
[2] “thread apply all [EXP]” is a gdb native command, which can be simplified as “t a a [e]”.
[3] There should have been 6 bt blocks for thread 1 (main) + thread 2-6 (phil), I picked thread 6 only.

Showcase2: Coredump

```
$ cat -n must_coredump.py
     1  import ctypes
     2
     3  crash = ctypes.CFUNCTYPE(None)(0)
     4  crash()                                                    [1]

$ python3.13-dbg must_coredump.py
Segmentation fault (core dumped)

$ gdb $(which python3.13-dbg) core.99215

(gdb) py-bt
Traceback (most recent call first):
  File "/home/gray/src/github.com/jschwinger233/pycon2025/must_coredump.py", line 4, in
<module>
    crash()
```

[1] In real world, a coredump crash is likely to be triggered by 3rd party libs with Cython/ctypes/C extensions.

How?

1. Each Python function call creates a new frame, and that frame is executed by `_PyEval_EvalFrameDefault`.

```
py_func1()  ──────►  _PyEval_EvalFrameDefault(tstate=..., frame=..., exc=...)
```

2. A Python `frame` object exposes the current function name and line number in the *Python* source code.

```c
void print_pyframe(struct _PyInterpreterFrame *frame)
{
    PyCodeObject *code = (PyCodeObject *)frame->f_executable;
    const char *py_filename = PyUnicode_AsUTF8(code->co_filename);
    const char *py_funcname = PyUnicode_AsUTF8(code->co_name);
    printf("file=%s func=%s\n", py_filename, py_funcname);
}
```

3. In CPython, each frame object links to upper frame.

```c
struct _PyInterpreterFrame *upper_frame(struct _PyInterpreterFrame *frame)
{
    return frame->previous;
}
```

Re-invent py-bt

Pseudo code:

```
def py-bt:
    for cframe in cstack:
        if cframe.called_by(“_PyEval_EvalFrameDefault”):
            pyframe = arg(“frame”)
            while pyframe:
                print_pyframe(pyframe)
                pyframe = upper_pyframe(pyframe)
```

Gdb script:

```
define py-bt
    frame 0
    set $i = 0
    while $i < 50
        if $_caller_is("_PyEval_EvalFrameDefault", $i)
            frame $i
            set $pyframe = frame
            walk_pyframe
            loop_break
        end
        set $i = $i + 1
    end
end
```

```
define walk_pyframe
    while !Py_IsNone($pyframe->f_executable)
        print_pyframe
        set $pyframe = $pyframe->previous
    end
end
```

```
define print_pyframe
        set $code = (PyCodeObject *)$pyframe->f_executable
        set $file = PyUnicode_AsUTF8($code->co_filename)
        set $func = PyUnicode_AsUTF8($code->co_name)
        printf " %s() at %s\n", $func, $file
end
```

Showcase3: cpython bug gh-134163 (@yihong0618)

```
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))

try:
    import _testcapi
    _testcapi.set_nomemory(0)
    a = list(range(1000, 2000))
except Exception as e:
    import traceback
    traceback.print_exc()
```

Left: exit with MemoryError

```
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
a = list(range(1000, 2000))
try:
    import _testcapi
    _testcapi.set_nomemory(0)
    a = list(range(1000, 2000))
except Exception as e:
    import traceback
    traceback.print_exc()
```

Right: hang forever😊

Showcase3: cpython bug gh-134163 (@yihong0618)

Why the right one gets stuck?

```
[...]
a = list(range(1000, 2000))
try:
    import _testcapi
    _testcapi.set_nomemory(0)
    a = list(range(1000, 2000))
except Exception as e:
    import traceback                                                    [1]
    traceback.print_exc()
```

```
// Python/ceval.c
685 _PyEval_EvalFrameDefault(PyThreadState *tstate, _PyInterpreterFrame *frame, int throwflag)
...
884 exception_unwind:                                                  <----┐
…                                                                          |
911             if (lasti) {                                               |
912                 int frame_lasti = _PyInterpreterFrame_LASTI(frame);    |
913                 PyObject *lasti = PyLong_FromLong(frame_lasti);        |
914                 if (lasti == NULL) {                                   |
915                     goto exception_unwind;                          ---┘   [2]
```

[1] Python stuck at “import traceback”

[2] “goto exception_unwind” leads to infinite loop in the _PyEval_EvalFrameDefault.

Showcase3: cpython bug gh-134163 (@yihong0618)

Why one less “list(range())” avoids ∞ ?

```
// Python/ceval.c
685 _PyEval_EvalFrameDefault(...)
913                 PyObject *lasti = PyLong_FromLong(frame_lasti);
914                 if (lasti == NULL) {                                    <-
915                     goto exception_unwind;
```

```
(gdb) b Python/ceval.c:914                                                  [1]
Breakpoint 1 at 0x5dab79: file ../Python/ceval.c, line 914.
```

[1] The breakpoint hits for every Python function call.

```
(gdb) b Python/ceval.c:914 if $py("'import traceback' in gdb.execute('py-bt', to_string=True)")   [2]
Breakpoint 1 at 0x5dab79: file ../Python/ceval.c, line 914.
```

[2] Now breakpoint hits only if it’s executing “import traceback”.

```
// Left: exit
Breakpoint 1, at ../Python/ceval.c:914
(gdb) p frame_lasti
$1 = 243                                [3a]
```

```
// Right: hang
Breakpoint 1, at ../Python/ceval.c:914
(gdb) p frame_lasti
$1 = 258                                [3b]
```

[3] #define _PY_NSMALLPOSINTS 257

Summary

1. Many thanks to gdb-python ecosystem🙏: gdb module, libpython.py
2. Case1: Python process hangs.
3. Case2: Python process crashes with coredump.
4. Case3: Debug “how CPython runs C code for a specific Python code”.

Spoiler: fancy PEP-768 **CANNOT** help any scenario above.


```
pdb -p $(pidof python)?
```

In case you haven’t used pdb 🙂 🙂

```
import pdb; pdb.set_trace()
class A:
    def __init__(self, x):
        self.x = x
    def double(self):
        return self.x * 2
print(A(10).double())
```

Debug everything in Python VM context!

```
(Pdb) b 6
Breakpoint 1 at /home/gray/src/github.com/jschwinger233/pycon2025/b.py:6

(Pdb) c
> /home/gray/src/github.com/jschwinger233/pycon2025/b.py(6)double()
-> return self.x * 2
(Pdb) p self.__dict__
{'x': 10}

(Pdb) globals()
{'__name__': '__main__', 'A': <class '__main__.A'>, '__pdb_convenience_variables': {...}...}

(Pdb) bt
  /home/gray/src/github.com/jschwinger233/pycon2025/b.py(7)<module>()
-> print(A(10).double())
> /home/gray/src/github.com/jschwinger233/pycon2025/b.py(6)double()
-> return self.x * 2

(Pdb) import os
(Pdb) print(os.environ)
```

```
gdb -p $(pidof python)
```



```
pdb -p $(pidof python)
```

PyDev adventures

Posting about venturing (and creating) PyDev.

LINKS: PyDev.org Blog RSS Twitter RSS

Fabio Zadrozny

Thursday, September 25, 2014

Attaching debugger to running process (PyDev 3.8.0) [1]

The latest PyDev 3.8.0 has just been released... along with a bunch of bugfixes, the major feature added is the possibility of attaching the debugger to a running process.

So, I thought about explaining a bit how to use it (and later a bit on how it was done).

The first thing is that the Debug perspective must be activated (as the attach to process menu is only shown by default in the Debug Perspective). Then, when on the debug perspective, select PyDev > Attach to Process (as the image below shows).

[1] JetBrains/PyCharm sponsors this work:
https://pydev.blogspot.com/2014/08/pydev-370-pydevpycharm-debugger-merge.html

How?

1. (gdb) call *expr*

```
$ gdb -p $(pidof python3.13-dbg)

(gdb) call write(1, "hello from sectoid\n", 19)
$1 = 19
```

```
$ python3.13-dbg -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
hello from sectoid
```

2. (cpython) PyRun_SimpleString

```
// github.com/python/cpython/Programs/_testembed.c
static void
dump_config(void)
{
    (void) PyRun_SimpleStringFlags(
        "import _testinternalcapi, json; "
        "print(json.dumps(_testinternalcapi.get_configs()))",
        0);
}
```

gdb call + PyRun_SimpleString()

```
$ gdb -p $(pidof python) --batch \                                                  [1]
    -eval-command='call PyGILState_Ensure()' \                                      [2]
    -eval-command='call PyRun_SimpleString("print(\"hello from sectoid\")")' \      [3]
    -eval-command='call PyGILState_Release($1)'                                     [4]
```

[1] “--batch --eval-command” has no functional differences from interactive gdb session.
[2] “call PyGILState_Ensure()” acquires GIL.
[3] “PyRun_SimpleString()” executes any Python code!
[4] “call PyGILState_Release($1)” releases GIL.

```
$ python3.13-dbg -m http.server

Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) …
127.0.0.1 - - [09/Sep/2025 21:32:59] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [09/Sep/2025 21:32:59] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [09/Sep/2025 21:32:59] "GET / HTTP/1.1" 200 -
hello from sectoid
127.0.0.1 - - [09/Sep/2025 21:33:00] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [09/Sep/2025 21:33:00] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [09/Sep/2025 21:33:00] "GET / HTTP/1.1" 200 -
```

remote_pdb

```
$ pip install remote-pdb
```
[1]

[1] “Remote vanilla PDB (over TCP sockets) *done right*: no extras, proper handling around connection failures and CI. Based on pdbx.” —— https://pypi.org/project/remote-pdb/

```
from remote_pdb import RemotePdb; RemotePdb('127.0.0.1', 4444).set_trace()
```
[2]

[2] By design it’s an intrusive debugging tool, but since we have the the archer fish...

remote_pdb + gdb + PyRun_SimpleString

```
$ gdb -p $(pidof python) --batch \
    -eval-command='call PyGILState_Ensure()' \
    -eval-command='call PyRun_SimpleString("from remote_pdb import RemotePdb; RemotePdb(\"127.0.0.1\",
4444).set_trace()")' \
    -eval-command='call PyGILState_Release($1)'
```

```
$ python3.13-dbg -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
RemotePdb session open at 127.0.0.1:4444, waiting for connection ...
```

```
$ nc localhost 4444
> <string>(1)<module>()

(Pdb) bt
  /usr/lib/python3.13/runpy.py(198)_run_module_as_main()
-> return _run_code(code, main_globals, None,
  /usr/lib/python3.13/runpy.py(88)_run_code()
-> exec(code, run_globals)
  /usr/lib/python3.13/http/server.py(1323)<module>()
-> test(
  /usr/lib/python3.13/http/server.py(1278)test()
-> httpd.serve_forever()
  /usr/lib/python3.13/socketserver.py(235)serve_forever()
-> ready = selector.select(poll_interval)
  /usr/lib/python3.13/selectors.py(398)select()
-> fd_event_list = self._selector.poll(timeout)
> <string>(1)<module>()
```

Two Clouds

1. PyGILState_Ensure(): GIL contention?

2. PyRun_SimpleString(): Guess the probability of crashing the Python process? ☺️

This approach is fundamentally unsafe because the injected code can execute at any point during the interpreter's execution cycle - even during critical operations like memory allocation, garbage collection, or thread state management. When this happens, the results are catastrophic: **attempting to allocate memory while already inside malloc() causes crashes, modifying objects during garbage collection corrupts the interpreter's state, and touching thread state at the wrong time leads to deadlocks.**
— https://peps.python.org/pep-0768/#motivation

Conclusion: Don't do this.

PEP-768

Safe external debugger interface for CPython

python

Sign Up | Log In

Please read our Community Guidelines that are tailored to this space in addition to the Python Software Foundation Code of Conduct that
bound by.

Reach out to @moderators with any questions.

PEP 768 – Safe external debugger interface for CPython

PEPs

Pablo Galindo Salgado pablogsal Steering Council Member — Dec 2024

Hi everyone 👋,

We are very exited to share with you PEP 768 133 , which proposes adding a safe external debugger interface to CPython. We think this is a really exciting development that would allow debuggers and profilers to safely attach to running Python processes without stopping or restarting them.

The key highlight is that it would enable tools like pdb to attach to live processes by PID (similar to `gdb -p`), letting developers inspect and debug Python applications in real-time. This capability can be also leveraged by other tools such as memory profilers, performance profilers and other state-inspection tools.

The proposal has already been successfully implemented in PyPy 37 (Thanks @cfbolz ❤️). The target version is Python 3.14.

You can read the full PEP here: PEP 768 – Safe external debugger interface for CPython | peps.python.org 133

31

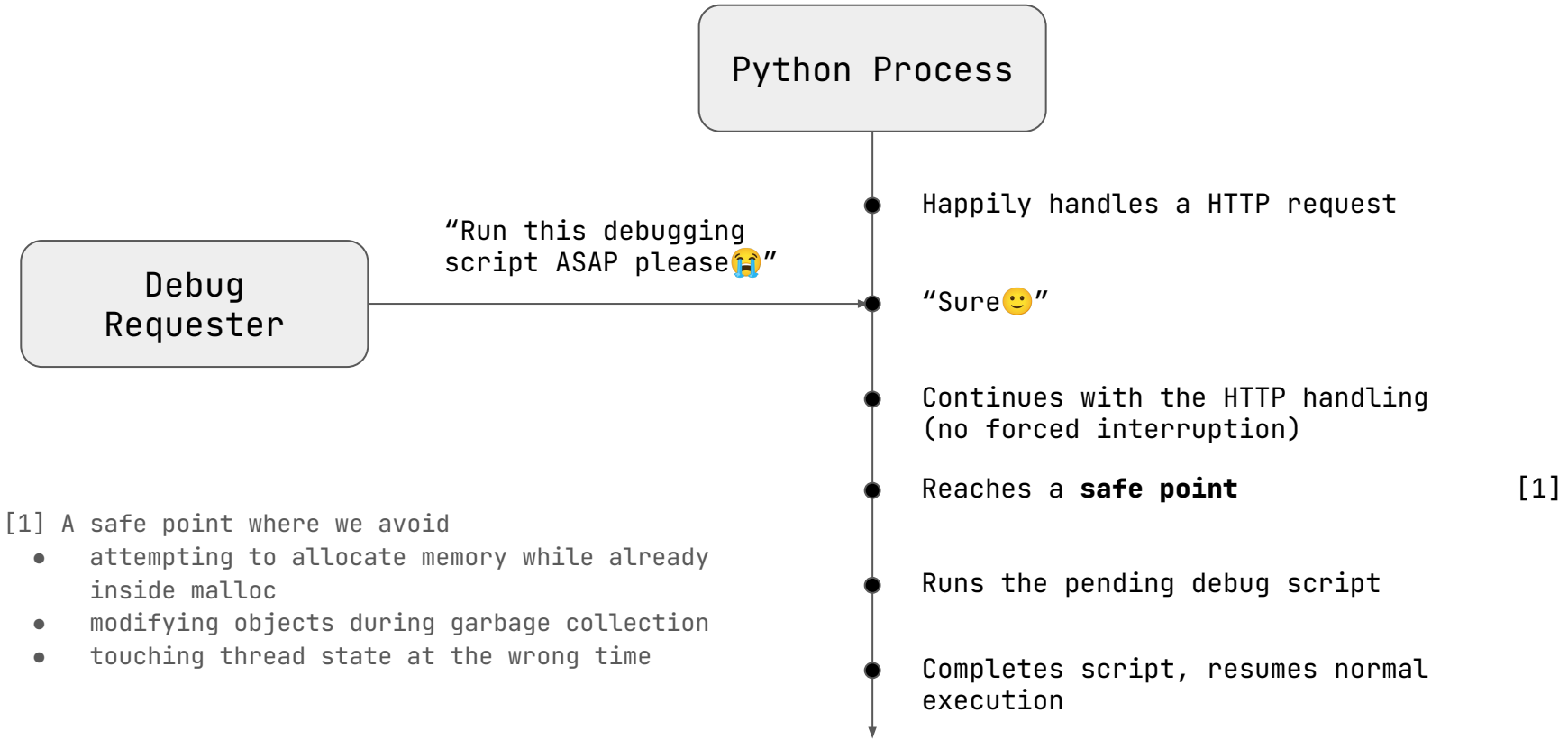
Dec 2024

1 / 59
Dec 2024

Jun 25

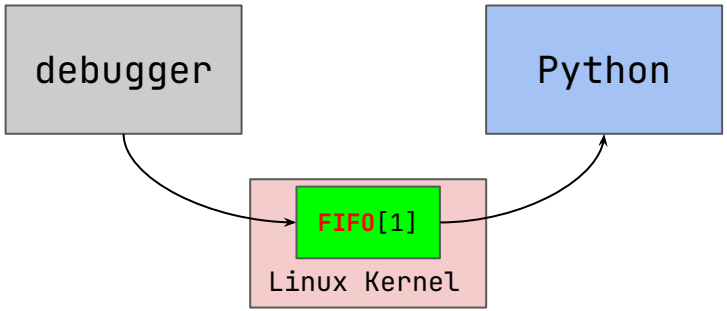
Pablo Galindo Salgado

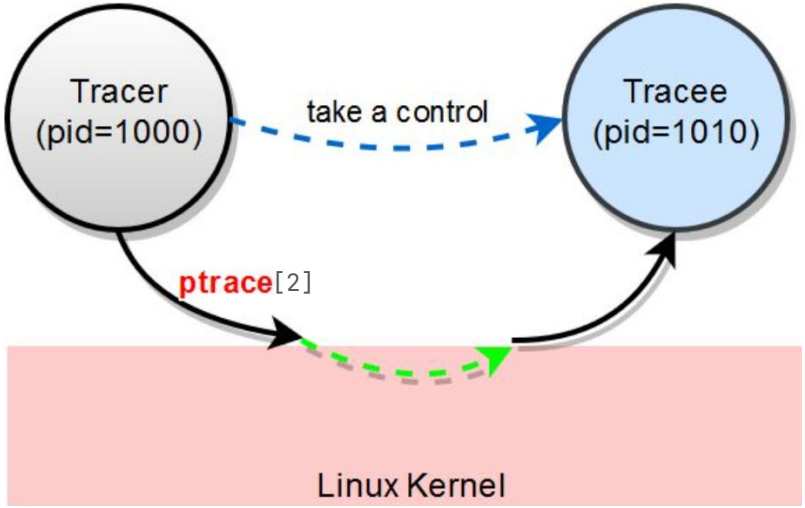
NUDGE, DON’T HIJACK



[1] A safe point where we avoid
- attempting to allocate memory while already inside malloc
- modifying objects during garbage collection
- touching thread state at the wrong time

Debugger's IPC (Interprocess Communication Mechanisms)





[1] Or unix socket, real-time signal, etc. The concerns are security: we mustn't allow a random process to tamper with Python.

[2] Only authorized processes can call ptrace syscall to deliver debugging request.

Haven't heard ptrace? That's the syscall behind GDB!

+_PyRuntime

```
$ objdump -T /usr/local/bin/../lib/libpython3.14.so.1.0 | grep _PyRuntime
000000000050ade0 g    DO .PyRuntime   000000000004d478  Base        _PyRuntime          [1]

$ gdb python3.14
(gdb) p &_PyRuntime
$1 = (_PyRuntimeState *) 0x71191f359de0 <_PyRuntime>
```

[1] CPython reserves an ELF symbol that debuggers can resolve at runtime.

```
(gdb) ptype _PyRuntime.main_tstate.remote_debugger_support                    [2]
type = struct {
    int32_t debugger_pending_call;
    char debugger_script_path[512];
}
```

[2] From this symbol, we find a “remote_debugger_support” where we can put a “debugger_script_path” and “debugger_pending_call”.

Where PEP-768 lands

```
$ cat hello.py
print('hello from The Absolute')

$ gdb -p $(pidof python) --batch
    -eval-command='set _PyRuntime.main_tstate.remote_debugger_support.debugger_pending_call=1'
    -eval-command='set _PyRuntime.main_tstate.remote_debugger_support.debugger_script_path="/hello.py"'

$ cat rpdb.py
from remote_pdb import RemotePdb; RemotePdb("127.0.0.1", 4444).set_trace()

$ gdb -p $(pidof python) --batch
    -eval-command='set _PyRuntime.main_tstate.remote_debugger_support.debugger_pending_call=1'
    -eval-command='set _PyRuntime.main_tstate.remote_debugger_support.debugger_script_path="/rpdb.py"'
```

In fact, a new API “sys.remote_exec” saves the hassle of gdb.

```
>>> import sys
>>> sys.remote_exec(pid=1, script='/hello.py')
>>> quit
```

Happy Ending?

```
import time; time.sleep(2333333)                                    [1]
```

[1] Process in S (Interruptable Sleep) state.

```
t = threading.Thread(target=worker)
t.start()
t.join()                                                            [2]
```

[2] Main thread goes off CPU.

```
async def foo():
    await asyncio.sleep(2333)
    print("foo done")

async def bar():
    await asyncio.sleep(233)
    print("bar done")

async def main():
    await asyncio.gather(foo(), bar())

asyncio.run(main())                                                 [3]
```

[3] Async run blocks on epoll_wait, still off-CPU.

Make main thread remote-debugger friendly

1. Sleep with yields.

```
def sleep_with_yields(total, interval=1):
    end = time.monotonic() + total
    while True:
        now = time.monotonic()
        if now >= end: break
        time.sleep(min(interval, end - now))
```

2. Join with timeout.

```
while True:
    t.join(timeout=1)
```

3. Wake event loop every 1 second.

```
async def heartbeat(period=1):
    while True:
        await asyncio.sleep(period)

async def main():
    hb = asyncio.create_task(heartbeat(0.5))
    await asyncio.gather(foo(), bar())

asyncio.run(main())
```

Takeaways

1. **GDB is still the most versatile tool** for inspecting a live or crashed CPython process.

2. Fancy PEP-768 adds a “safe remote debug” API, but they don’t replace GDB in tough scenarios (hangs, crashes, or C/Python stack correlation).

3. Always understand your scenario and choose the right tool.

4. Above all, have fun debugging!