Breakpoints in the Cloud: Debugging Python Beyond Your IDE

PyWeb, October 2025

Yonatan Bitton
@bityob
linktr.ee/bityob

bit.ly/debug-slides
bit.ly/debug-git

Introduction

Debugging — one of the hardest, yet most powerful skills in a developer's toolbox

- Essential for developers and researchers
- Helps us understand *why* code fails when everything else looks fine

Before Debugging

- In production - we **prefer metrics, logs, and traces**
- Tools like Sentry help us understand exceptions
- Ideally, we fix the bug directly from data - no need to “step in”

Before Debugging

- In production - we **prefer metrics, logs, and traces**
- Tools like Sentry help us understand exceptions
- Ideally, we fix the bug directly from data - no need to “step in”

But sometimes... logs aren’t enough.

The stack trace is unclear. Something’s *weird*

The Debugging Mindset

“What on earth is going on in there?!”

- We want to **enter the code as it runs**
- See what variables do, line by line
- Understand *why* the logic fails

The Debugging Mindset

“What on earth is going on in there?!”

- We want to **enter the code as it runs**
- See what variables do, line by line
- Understand *why* the logic fails

Keep this mindset: **curious, methodical, and never afraid to dive deeper**

Three Debugging Zones

Three Debugging Zones

- **No Debugging Needed** - enough info from metrics, logs, or monitoring tools

Three Debugging Zones

- **No Debugging Needed** - enough info from metrics, logs, or monitoring tools
- **Interactive Debugging** - reproduce and debug in production/testing where the issue occurs, or preferably in a local controlled environment

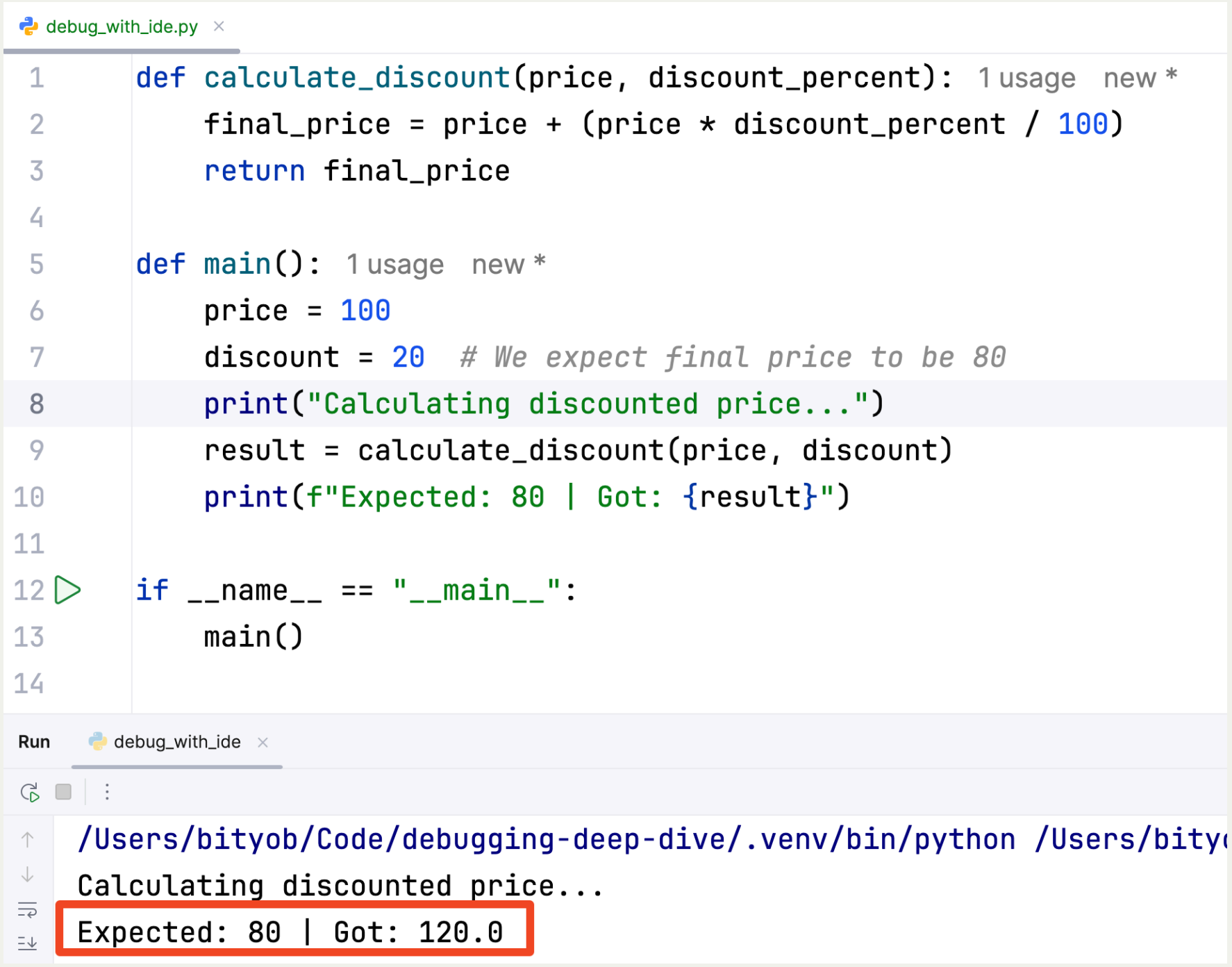
Three Debugging Zones

- **No Debugging Needed** - enough info from metrics, logs, or monitoring tools
- **Interactive Debugging** - reproduce and debug in production/testing where the issue occurs, or preferably in a local controlled environment
- **Non-interactive Debugging** — debug without attaching; use tracing, spying, or dump analysis

Let's begin!

From **local and simple**
→ to **remote, restricted, and deep**

IDE


debug_with_ide.py
1 def calculate_discount(price, discount_percent): 1 usage new *
2 final_price = price + (price * discount_percent / 100)
3 return final_price
4
5 def main(): 1 usage new *
6 price = 100
7 discount = 20 # We expect final price to be 80
8 print("Calculating discounted price...")
9 result = calculate_discount(price, discount)
10 print(f"Expected: 80 | Got: {result}")
11
12 if __name__ == "__main__":
13 main()
14
Run
debug_with_ide
/Users/bityob/Code/debugging-deep-dive/.venv/bin/python /Users/bityo
Calculating discounted price...
Expected: 80 | Got: 120.0


debug_with_ide.py
1 def calculate_discount(price, discount_percent): 1 usage new * price: 100 d
2 final_price = price + (price * discount_percent / 100) final_price: 120.0
3 return final_price
4
Debug
debug_with_ide
Threads & Variables
Console
MainThread
price - (price * discount_percent / 100)
result = {float} 80.0
discount_percent = {int} 20
final_price = {float} 120.0
price = {int} 100
calculate_discount, debug_with_ide.py
main, debug_with_ide.py:9
<module>, debug_with_ide.py:13

pdb (The Python Debugger)

When the IDE isn’t enough...

- Local debugging works great - until it doesn’t
- What if the code runs in production or testing, not inside your IDE?
- Time to drop into the terminal: `pdb`

When the IDE isn’t enough...

- Local debugging works great - until it doesn’t
- What if the code runs in production or testing, not inside your IDE?
- Time to drop into the terminal: `pdb`

Moving from *visual* debugging to *terminal* debugging

- Add `breakpoint()` to stop on this line
- Run the script as usual - `python <script_file_name>.py`

- `n` (next) - continue to next line
- `l` (list) - display source code around current line
- `s` (step) - step into function

debug_with_pdb.py

```python
def calculate_discount(price, discount_percent):  1 usage  new *
    breakpoint()
    final_price = price + (price * discount_percent / 100)
    return final_price
```

Terminal Local

```
bityob@Yonatans-MacBook-Pro debugging-deep-dive % uv run 02-pdb/debug_with_pdb.py
Calculating discounted price...
> /Users/bityob/Code/debugging-deep-dive/02-pdb/debug_with_pdb.py(2)calculate_discount()
-> breakpoint()
(Pdb) l
  1     def calculate_discount(price, discount_percent):
  2  ->     breakpoint()
  3          final_price = price + (price * discount_percent / 100)
  4          return final_price
  5
  6      def main():
  7          price = 100
  8          discount = 20  # We expect final price to be 80
  9          print("Calculating discounted price...")
 10          result = calculate_discount(price, discount)
 11          print(f"Expected: 80 | Got: {result}")
(Pdb) n
> /Users/bityob/Code/debugging-deep-dive/02-pdb/debug_with_pdb.py(3)calculate_discount()
-> final_price = price + (price * discount_percent / 100)
(Pdb)
> /Users/bityob/Code/debugging-deep-dive/02-pdb/debug_with_pdb.py(4)calculate_discount()
-> return final_price
(Pdb) final_price
120.0
(Pdb) price - (price * discount_percent / 100)
80.0
```

- Run script with `-m pdb` to start with pdb shell in the beginning
- Add breakpoints manually using `b <file_path or current file>:<line_number>`
- You can add a conditional breakpoint with adding `, <python logic condition>`

debug_with_pdb2.py

```python
def apply_discount(price, discount):  1 usage  new *
    final_result = price - discount
    return final_result

test_cases = [100, 75, 5]

for case in test_cases:
    result = apply_discount(case, discount: 20)
    print(f"Before: {case=}, After={result=}")
```

Terminal Local

```
(debugging-deep-dive) bityob@Yonatans-MBP debugging-deep-dive % uv run python -m pdb 02-pdb/debug_with_pdb2.py
> /Users/bityob/Code/debugging-deep-dive/02-pdb/debug_with_pdb2.py(1)<module>()
-> def apply_discount(price, discount):
(Pdb) b 3, final_result < 0
Breakpoint 1 at /Users/bityob/Code/debugging-deep-dive/02-pdb/debug_with_pdb2.py:3
(Pdb) c
Before: case=100, After=result=80
Before: case=75, After=result=55
> /Users/bityob/Code/debugging-deep-dive/02-pdb/debug_with_pdb2.py(3)apply_discount()
-> return final_result
(Pdb) price
5
(Pdb) discount
20
```

pdb++

Same power, better experience

- `pdb` is functional but plain
- `pdb++` builds on it with quality-of-life improvements:
  - Sticky mode (keeps code visible)
  - Syntax highlighting
- From now on, assume we're using `pdb++` under the hood
- Run `python -m pdb debug_with_pdbpp.py` just like normal pdb

```
bityob@Yonatans-MacBook-Pro debugging-deep-dive % uv run python -m pdb 03-pdbpp/debug_with_pdbpp.py
[2] > /Users/bityob/Code/debugging-deep-dive/03-pdbpp/debug_with_pdbpp.py(1)<module>()
-> def sanitize(quantity):
(Pdb++) sticky
```

```
[2] > /Users/bityob/Code/debugging-deep-dive/03-pdbpp/debug_with_pdbpp.py(1)<module>()

  1  -> def sanitize(quantity):
  2           # ❌ Bug: treats 0 as falsy and turns it into 1
  3           return quantity or 1
  4
  5       def total(price, quantity):
  6           q = sanitize(quantity)      # bug is hiding here
  7           return price * q
  8
  9       price = 100
 10       quantity = 0
 11       result = total(price, quantity)
 12       print(f"Total = {result}")  # Expected 0, got 100
(Pdb++)
```

```
[2] > /Users/bityob/Code/debugging-deep-dive/03-pdbpp/debug_with_pdbpp.py(11)<module>()

 1     def sanitize(quantity):
 2         # ❌ Bug: treats 0 as falsy and turns it into 1
 3         return quantity or 1
 4
 5     def total(price, quantity):
 6         q = sanitize(quantity)     # bug is hiding here
 7         return price * q
 8
 9     price = 100
10     quantity = 0
11  -> result = total(price, quantity)
12     print(f"Total = {result}")  # Expected 0, got 100
(Pdb++) step
```

```
[4] > /Users/bityob/Code/debugging-deep-dive/03-pdbpp/debug_with_pdbpp.py(3)sanitize()

1     def sanitize(quantity):
2         # ❌ Bug: treats 0 as falsy and turns it into 1
3  ->     return quantity or 1
(Pdb++)
```

- Run with `-m pdb` to stop on **unhandled exceptions**
- View the full stack trace using `w` (**where**)
- Navigate up and down the stack with `u` (**up**) and `d` (**down**) commands

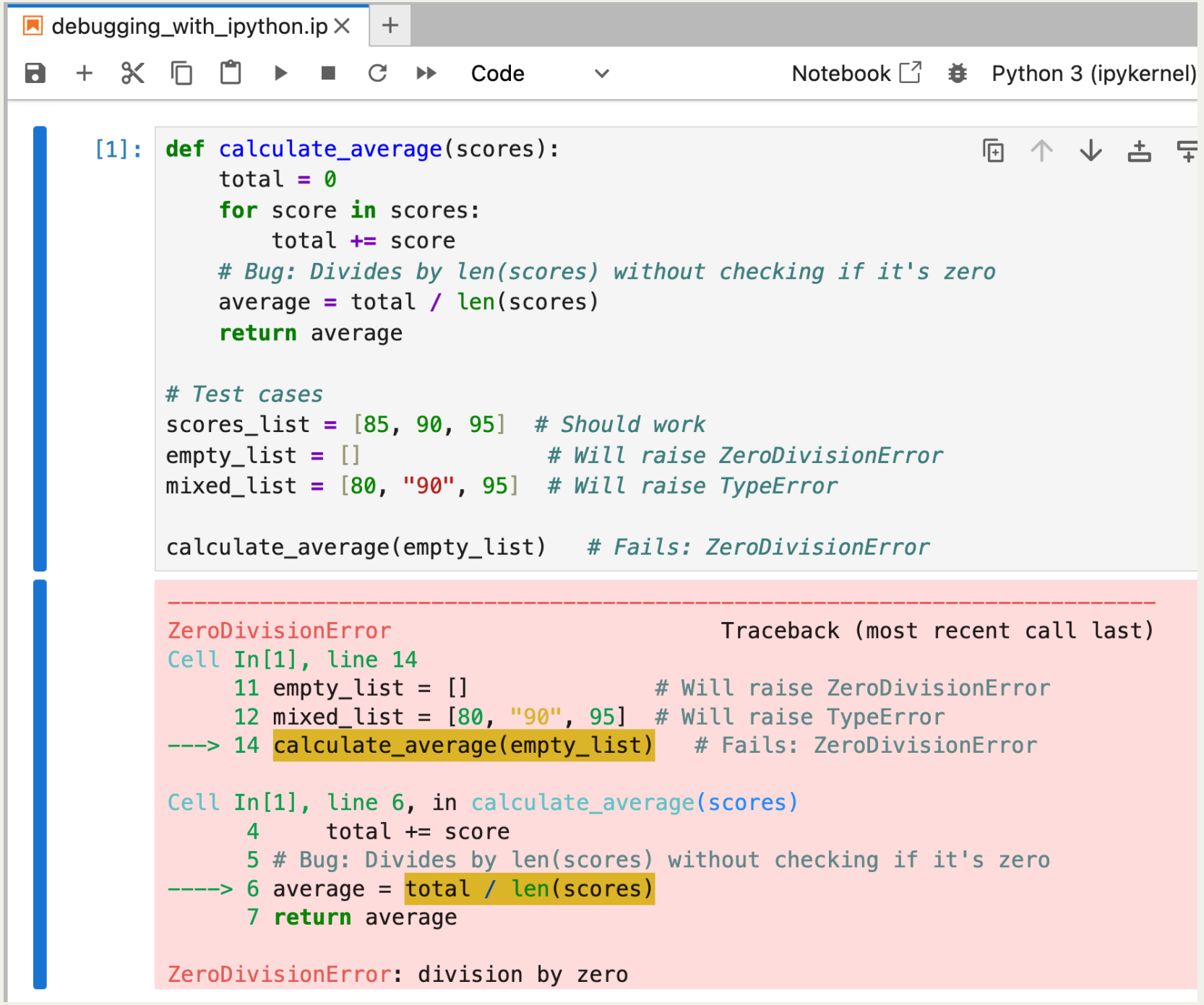
```
bityob@Yonatans-MacBook-Pro debugging-deep-dive % uv run -m pdb 03-pdbpp/debug_with_pdbpp2.py
[2] > /Users/bityob/Code/debugging-deep-dive/03-pdbpp/debug_with_pdbpp2.py(1)<module>()
-> def divide_numbers(a, b):
(Pdb++) c
100 / 10 = 10.0
100 / 5 = 20.0
Traceback (most recent call last):
  File "/Users/bityob/.local/share/uv/python/cpython-3.13.9-macos-aarch64-none/lib/python3.13/pdb.py", line 2504, in main
    pdb._run(target)
    ~~~~~~~~^^^^^^^^
  File "/Users/bityob/.local/share/uv/python/cpython-3.13.9-macos-aarch64-none/lib/python3.13/pdb.py", line 2244, in _run
    self.run(target.code)
    ~~~~~~~~^^^^^^^^^^^^^
  File "/Users/bityob/.local/share/uv/python/cpython-3.13.9-macos-aarch64-none/lib/python3.13/bdb.py", line 666, in run
    exec(cmd, globals, locals)
    ~~~~^^^^^^^^^^^^^^^^^^^^^^
  File "<string>", line 1, in <module>
  File "/Users/bityob/Code/debugging-deep-dive/03-pdbpp/debug_with_pdbpp2.py", line 7, in <module>
    result = divide_numbers(100, num)
  File "/Users/bityob/Code/debugging-deep-dive/03-pdbpp/debug_with_pdbpp2.py", line 2, in divide_numbers
    return a / b
           ~~^~~
ZeroDivisionError: division by zero
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
```

```
[5] > /Users/bityob/Code/debugging-deep-dive/03-pdbpp/debug_with_pdbpp2.py(2)divide_numbers()
-> return a / b
(Pdb++) w
  [0] /Users/bityob/.local/share/uv/python/cpython-3.13.9-macos-aarch64-none/lib/python3.13/pdb.py(2504)main()
      pdb._run(target)
  [1] /Users/bityob/.local/share/uv/python/cpython-3.13.9-macos-aarch64-none/lib/python3.13/pdb.py(2244)_run()
      self.run(target.code)
  [2] /Users/bityob/.local/share/uv/python/cpython-3.13.9-macos-aarch64-none/lib/python3.13/bdb.py(666)run()
      exec(cmd, globals, locals)
  [3] <string>(1)<module>()
  [4] /Users/bityob/Code/debugging-deep-dive/03-pdbpp/debug_with_pdbpp2.py(7)<module>()
      result = divide_numbers(100, num)
> [5] /Users/bityob/Code/debugging-deep-dive/03-pdbpp/debug_with_pdbpp2.py(2)divide_numbers()
      return a / b
(Pdb++) a
100
(Pdb++) b
0
(Pdb++) u
[4] > /Users/bityob/Code/debugging-deep-dive/03-pdbpp/debug_with_pdbpp2.py(7)<module>()
-> result = divide_numbers(100, num)
(Pdb++) num
0
```

IPython

Debugging small code blocks interactively

- Sometimes we don't need to run the full app
- We just want to test a few lines, functions, or reproduce a bug inline
- `IPython` gives a quick way to do that:
  - `%pdb on` to auto-enter post-mortem on exceptions
  - `%debug` to step through snippets
- Great for Jupyter or ad-hoc shell debugging


debugging_with_ipython.ip
Code
Notebook
Python 3 (ipykernel)
[1]:
def calculate_average(scores):
total = 0
for score in scores:
total += score
# Bug: Divides by len(scores) without checking if it's zero
average = total / len(scores)
return average
# Test cases
scores_list = [85, 90, 95] # Should work
empty_list = [] # Will raise ZeroDivisionError
mixed_list = [80, "90", 95] # Will raise TypeError
calculate_average(empty_list) # Fails: ZeroDivisionError
ZeroDivisionError Traceback (most recent call last)
Cell In[1], line 14
11 empty_list = [] # Will raise ZeroDivisionError
12 mixed_list = [80, "90", 95] # Will raise TypeError
---> 14 calculate_average(empty_list) # Fails: ZeroDivisionError
Cell In[1], line 6, in calculate_average(scores)
4 total += score
5 # Bug: Divides by len(scores) without checking if it's zero
----> 6 average = total / len(scores)
7 return average
ZeroDivisionError: division by zero

```
[2]: %debug calculate_average(empty_list)
```

```
NOTE: Enter 'c' at the ipdb++>  prompt to continue execution.
> <string>(1)<module>()

ipdb++>  s
--Call--
> /var/folders/v_/vx52ft7x1_g6cz_6tc56hxdm0000gn/T/ipykernel_53864/3917864610.py(1)
calculate_average()
----> 1 def calculate_average(scores):
      2     total = 0
      3     for score in scores:
      4         total += score
      5     # Bug: Divides by len(scores) without checking if it's zero

ipdb++>  q
> <string>(0)<module>()
```

[2]
```
%pdb
```

```
Automatic pdb calling has been turned ON
```

[3]:
```
calculate_average(empty_list)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[3], line 1
----> 1 calculate_average(empty_list)

Cell In[1], line 6, in calculate_average(scores)
      4     total += score
      5 # Bug: Divides by len(scores) without checking if it's zero
----> 6 average = total / len(scores)
      7 return average

ZeroDivisionError: division by zero
> /var/folders/v_/vx52ft7x1_g6cz_6tc56hxdm0000gn/T/ipykernel_54216/3917864610.py(6)calculate_avera
ge()
      4         total += score
      5     # Bug: Divides by len(scores) without checking if it's zero
----> 6     average = total / len(scores)
      7     return average
      8

   1 frame hidden (try 'help hidden_frames')
ipdb++>  w
  /var/folders/v_/vx52ft7x1_g6cz_6tc56hxdm0000gn/T/ipykernel_54216/2768597822.py(1)<module>()
----> 1 calculate_average(empty_list)
```

```
   1 frame hidden (try 'help hidden_frames')
ipdb++>  w
  /var/folders/v_/vx52ft7x1_g6cz_6tc56hxdm0000gn/T/ipykernel_54216/2768597822.py(1)<module>()
----> 1 calculate_average(empty_list)
> /var/folders/v_/vx52ft7x1_g6cz_6tc56hxdm0000gn/T/ipykernel_54216/3917864610.py(6)calculate_avera
ge()
      4         total += score
      5     # Bug: Divides by len(scores) without checking if it's zero
----> 6     average = total / len(scores)
      7     return average
      8

ipdb++>  total
0
ipdb++>  scores
[]
ipdb++>  q
```

Remote PDB

When the bug only happens remotely...

- Local debugging isn't always possible
- The code runs on a **server, container, or Kubernetes pod**
- No standard input/output, no IDE
- `remote-pdb` lets us open a socket and connect to it from outside
- Trigger the bug (e.g., via HTTP request) → connect via telnet → debug

```python
# Set a remote-pdb breakpoint
from remote_pdb import RemotePdb
# Opens socket for debugging
RemotePdb('0.0.0.0', 4444).set_trace()
```

First shell - Run the web server on docker container

```
docker build -t flask-debug .
docker run --name flask-debug flask-debug
```

```
bityob@Yonatans-MacBook-Pro debugging-deep-dive % (cd 05-remote-pdb && docker build -t flask-debug .)
docker run -p 5555:5555 -p 4444:4444 flask-debug

[+] Building 8.1s (9/9) FINISHED                                                        docker:colima
 => [internal] load build definition from Dockerfile                                              0.0s
 => => transferring dockerfile: 153B                                                              0.0s
 => [internal] load metadata for docker.io/library/python:3.13-slim                               0.0s
 => [internal] load .dockerignore                                                                 0.0s
 => => transferring context: 2B                                                                   0.0s
 => [1/4] FROM docker.io/library/python:3.13-slim@sha256:9a89bdad56e05645ab43d0ea50f811aae46b0090714557ad628a0d0426945500  0.0s
 => => resolve docker.io/library/python:3.13-slim@sha256:9a89bdad56e05645ab43d0ea50f811aae46b0090714557ad628a0d0426945500  0.0s
 => [internal] load build context                                                                 0.0s
 => => transferring context: 28B                                                                  0.0s
 => CACHED [2/4] WORKDIR /app                                                                     0.0s
 => [3/4] RUN pip install flask remote-pdb pdbpp                                                  7.0s
 => [4/4] COPY app.py .                                                                           0.0s
 => exporting to image                                                                            1.0s
 => => exporting layers                                                                           0.8s
 => => exporting manifest sha256:210fc3dab2e0b7e98595f51c8d71da557ab0a37f5a1a92dffb8d86f0f0ec0baa  0.0s
 => => exporting config sha256:907e356fb705a3c9824b475002a30e4f5e56ae85f4a7aa65d5c15726bdf0b181    0.0s
 => => exporting attestation manifest sha256:e086f8124726590af39fdd8e1924887fe00a497f3c7edaa99135a85ddae2ab3a  0.0s
 => => exporting manifest list sha256:ce2f76ff7e34db4aaee3b8aa710b810ca2f82273bfa9a6466c35b85b995e7091  0.0s
 => => naming to docker.io/library/flask-debug:latest                                             0.0s
 => => unpacking to docker.io/library/flask-debug:latest                                          0.2s
 * Serving Flask app 'app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5555
 * Running on http://172.17.0.2:5555
Press CTRL+C to quit
```

Second shell - Wait for breakpoint on server

```
docker exec -it flask-debug bash
until telnet 127.0.0.1 4444; do sleep 2; done
```

```
bityob@Yonatans-MacBook-Pro debugging-deep-dive % until telnet 127.0.0.1 4444; do sleep 2; done
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Connection closed by foreign host.
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Connection closed by foreign host.
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Connection closed by foreign host.
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Connection closed by foreign host.
```

Third shell - Send web request for hitting the breakpoint

```
docker exec -it flask-debug bash
curl -X POST \
    -H "Content-Type: application/json" \
    -d '{"data": ["1", "two"]}' \
    http://localhost:5555/process
```

```
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5555
 * Running on http://172.17.0.2:5555
Press CTRL+C to quit
RemotePdb session open at 0.0.0.0:4444, waiting for connection ...
RemotePdb session open at 0.0.0.0:4444, waiting for connection ...
RemotePdb accepted connection from ('172.17.0.1', 52940).
RemotePdb accepted connection from ('172.17.0.1', 52940).
```

```
[15] > /app/app.py(17)process()

12      @app.route('/process', methods=['POST'])
13      def process():
14          data = request.json.get('data', [])
15          # Set a remote-pdb breakpoint
16          from remote_pdb import RemotePdb
17  ->      RemotePdb('0.0.0.0', 4444).set_trace()  # Opens socket for debuggin
18          result = process_data(data)
19          return {"average": result}
(Pdb++) data
['1', 'two']
```

Web PDB

Making remote debugging visual

- Same idea as `remote-pdb`, but easier to control
- `web-pdb` opens a mini web UI you can control from your browser
- Useful when you **`do`** have network access to the container or server

```
import web_pdb
web_pdb.set_trace()
```

First shell - Run the web server on a docker container with open ports for the application and the debugger both

```
docker run -it -p 4444:4444 -p 5555:5555 webpdb
```

Second shell - Send web request for hitting the breakpoint

```
uvx --from httpie http POST :4444/process data:='["1", "two"]'
```

Now, go to http://localhost:5555/ in the browser...

Web-PDB Console on *localhost:5555*

Current file: web_pdb_app.py(6)

```
1  from flask import Flask, request
2
3  app = Flask(__name__)
4
5
6  def process_data(data):
7      # Bug: Assumes all data entries are integers; crashes on non-integer
8      total = sum(int(x) for x in data)
9      average = total / len(data)  # Potential ZeroDivisionError
10     return average
11
12
13 @app.route("/process", methods=["POST"])
14 def process():
```

Globals

```
Flask = <class 'flask.app.Flask'>
__pdb_convenience_variables = {'_fr
app = <Flask 'web_pdb_app'>
process = <function process at 0xe2
process_data = <function process_dat
request = <Request 'http://localhost
```

```
> /app/web_pdb_app.py(18)process()
-> import web_pdb; web_pdb.set_trace()
(Pdb) n
> /app/web_pdb_app.py(20)process()
-> result = process_data(data)
(Pdb) s
```

FIFO Streams

No network? No problem

- In restricted environments, we can't open network ports
- `FIFO` (named pipes) let us redirect stdin/stdout locally
- Works when you have shell access but **no network** access

```
import pdb
pdb.Pdb(stdin=open("fifo_stdin"), stdout=open("fifo_stdout", "w")).set_trace()
```

First shell - Create fifo files and read/write from/to those streams

```
mkfifo fifo_stdin
mkfifo fifo_stdout
cat fifo_stdout & cat > fifo_stdin
```

Second shell - Run the web server

```
uv run fifo_app.py
```

Third shell - Send web request for hitting the breakpoint

```
uvx --from httpie http POST :5555/process data:='["1", "two"]'
```

```
[15] > /Users/bityob/Code/debugging-deep-dive/07-fifo/fifo_app.py(20)process()

13      @app.route("/process", methods=["POST"])
14      def process():
15          data = request.json.get("data", [])
16
17          # Breakpoint using fifo streams
18          import pdb; pdb.Pdb(stdin=open("fifo_stdin"), stdout=open("fifo_stdout", "w")).set_trace()
19
20  ->      result = process_data(data)
21          return {"average": result}
(Pdb++)
```

Reverse Remote PDB

When you have no access at all...

- In **serverless** environments (e.g., AWS Lambda), we can't open sockets or SSH in
- Solution: reverse the connection
- The Lambda connects *to us* through a public tunnel (e.g., `ngrok`)
- Same debugging flow — just inverted

Real Case

Real Case

- Lambda failed to connect to Postgres → “No module named psycopg2”

Real Case

- Lambda failed to connect to Postgres → “No module named psycopg2”
- Worked perfectly in local Docker runtime

Real Case

- Lambda failed to connect to Postgres → “No module named psycopg2”
- Worked perfectly in local Docker runtime
- No SSH, no open sockets — true serverless environment

Real Case

- Lambda failed to connect to Postgres → “No module named psycopg2”
- Worked perfectly in local Docker runtime
- No SSH, no open sockets — true serverless environment
- Solution: reverse the connection

Real Case

- Lambda failed to connect to Postgres → “No module named psycopg2”
- Worked perfectly in local Docker runtime
- No SSH, no open sockets — true serverless environment
- Solution: reverse the connection
    - Lambda connects **out** to our `ngrok` tunnel

Real Case

- Lambda failed to connect to Postgres → “No module named psycopg2”
- Worked perfectly in local Docker runtime
- No SSH, no open sockets — true serverless environment
- Solution: reverse the connection
  - Lambda connects **out** to our `ngrok` tunnel
  - We debug locally through that socket

Real Case

- Lambda failed to connect to Postgres → “No module named psycopg2”
- Worked perfectly in local Docker runtime
- No SSH, no open sockets — true serverless environment
- Solution: reverse the connection
  - Lambda connects **out** to our `ngrok` tunnel
  - We debug locally through that socket
- Root cause was visible only **inside** the Lambda runtime

```
from remote_pdb import RemotePdb
RemotePdb("2.tcp.eu.ngrok.io", 18909, reverse=True).set_trace()
...
import psycopg
```

lambda_handler.py

```python
import os

def main(event, context):  Yonatan Bitton
    from remote_pdb import RemotePdb
    RemotePdb( host: "2.tcp.eu.ngrok.io",  port: 18909, reverse=True).set_trace()

    import psycopg

    conn = psycopg.connect(
        host=os.environ["PG_HOST"],
        dbname=os.environ["PG_DB"],
        user=os.environ["PG_USER"],
        password=os.environ["PG_PASSWORD"],
        port=os.environ["PG_PORT"],
    )
    with conn.cursor() as cur:
        cur.execute("SELECT now();")
        now = cur.fetchone()
    conn.close()

    return {"statusCode": 200, "body": f"PostgreSQL time: {now[0]}"}
```

First shell - Listen to incoming connections

```
nc -l 4444
```

Second shell - Tunnel my private machine to the internet using `ngrok`

```
ngrok tcp 4444
```

```
Session Status                online
Account                       Yonatan Bitton (Plan: Free)
Update                        update available (version 3.32.0, Ctrl-U to update)
Version                       3.31.0
Region                        Europe (eu)
Latency                       81ms
Web Interface                 http://127.0.0.1:4041
Forwarding                    tcp://0.tcp.eu.ngrok.io:12525 -> localhost:4444

Connections                   ttl     opn     rt1     rt5     p50     p90
                              0       0       0.00    0.00    0.00    0.00
```

Third shell - Trigger the lambda manually

```
sls invoke -f hello
```

```
[5] > /var/task/lambda_handler.py(7)main()

  3     def main(event, context):
  4         from remote_pdb import RemotePdb
  5         RemotePdb("0.tcp.eu.ngrok.io", 13395, reverse=True).set_trace()
  6
  7  ->     import psycopg
  8
  9         conn = psycopg.connect(
 10             host=os.environ["PG_HOST"],
 11             dbname=os.environ["PG_DB"],
 12             user=os.environ["PG_USER"],
 13             password=os.environ["PG_PASSWORD"],
 14             port=os.environ["PG_PORT"],
 15         )
 16         with conn.cursor() as cur:
 17             cur.execute("SELECT now();")
 18             now = cur.fetchone()
 19         conn.close()
 20
 21         return {"statusCode": 200, "body": f"PostgreSQL time: {now[0]}"}
(Pdb++) import psycopg
*** ImportError: no pq wrapper available.
```

sys.settrace

Beyond traditional debugging

- Sometimes we can't debug *live* anymore
- Instead, we can **trace** what happens
- `sys.settrace()` lets us catch exceptions or analyze code behavior line by line
- Useful for catching hidden exceptions or logic errors silently swallowed

Real Case

Real Case

- After upgrading Python 2 → 3, some tests failed silently

Real Case

- After upgrading Python 2 → 3, some tests failed silently
- No exceptions, only wrong results

Real Case

- After upgrading Python 2 → 3, some tests failed silently
- No exceptions, only wrong results
- Added `sys.settrace` to log all handled exceptions

Real Case

- After upgrading Python 2 → 3, some tests failed silently
- No exceptions, only wrong results
- Added `sys.settrace` to log all handled exceptions
- Found hidden `None` assignment caused by changed assumption

Real Case

- After upgrading Python 2 → 3, some tests failed silently
- No exceptions, only wrong results
- Added `sys.settrace` to log all handled exceptions
- Found hidden `None` assignment caused by changed assumption
- Tracing revealed the bug immediately — fixed and added a regression test

debugging_with_sys_settrace.py

```
28  servers = ["example.com", "nonexistent.domain", "python.org"]
29  server_info = aggregate_info(servers)
30
31  # subtle failure here: expecting ip to be a string
32  for info in server_info:
33      # Wrong str here...
34      ip = str(info["ip"])
35      # fails silently until this line
36      octets = ip.split(".")
37      print(f"{ip} first octet: {octets[0]}")
```

Terminal Local

```
(debugging-deep-dive) bityob@Yonatans-MacBook-Pro debugging-deep-dive % \
> uv run 09-sys-settrace/debugging_with_sys_settrace.py
23.215.0.138 first octet: 23
None first octet: None
151.101.64.223 first octet: 151
```

```python
import sys

def trace_exceptions(frame, event, arg):
    if event == "exception":
        exc_type, exc_value, exc_traceback = arg
        file_path = frame.f_code.co_filename

        if "debugging_with_sys_settrace" in file_path:
            print(f"Exception caught: {exc_type.__name__}: {exc_value}")
            print(f"  File: {frame.f_code.co_filename}")
            print(f"  Function: {frame.f_code.co_name}")
            print(f"  Line: {frame.f_lineno}")

    return trace_exceptions

sys.settrace(trace_exceptions)
```

```python
import sys

def trace_exceptions(frame, event, arg):
    if event == "exception":
        exc_type, exc_value, exc_traceback = arg
        file_path = frame.f_code.co_filename

        if "debugging_with_sys_settrace" in file_path:
            print(f"Exception caught: {exc_type.__name__}: {exc_value}")
            print(f"  File: {frame.f_code.co_filename}")
            print(f"  Function: {frame.f_code.co_name}")
            print(f"  Line: {frame.f_lineno}")

    return trace_exceptions

sys.settrace(trace_exceptions)
```

```
 2
 3 def trace_exceptions(frame, event, arg):
 4     if event == "exception":
 5         exc_type, exc_value, exc_traceback = arg
 6         file_path = frame.f_code.co_filename
 7
 8         if "debugging_with_sys_settrace" in file_path:
 9             print(f"Exception caught: {exc_type.__name__}: {exc_value}")
10             print(f"  File: {frame.f_code.co_filename}")
11             print(f"  Function: {frame.f_code.co_name}")
12             print(f"  Line: {frame.f_lineno}")
13
14     return trace_exceptions
15
16 sys.settrace(trace_exceptions)
```

```
 2
 3 def trace_exceptions(frame, event, arg):
 4     if event == "exception":
 5         exc_type, exc_value, exc_traceback = arg
 6         file_path = frame.f_code.co_filename
 7
 8         if "debugging_with_sys_settrace" in file_path:
 9             print(f"Exception caught: {exc_type.__name__}: {exc_value}")
10             print(f"  File: {frame.f_code.co_filename}")
11             print(f"  Function: {frame.f_code.co_name}")
12             print(f"  Line: {frame.f_lineno}")
13
14     return trace_exceptions
15
16 sys.settrace(trace_exceptions)
```

```
(debugging-deep-dive) bityob@Yonatans-MacBook-Pro debugging-deep-dive % \
uv run 09-sys-settrace/debugging_with_sys_settrace.py
Exception caught: gaierror: [Errno 8] nodename nor servname provided, or not known
  File: /Users/bityob/Code/debugging-deep-dive/09-sys-settrace/debugging_with_sys_settrace.py
  Function: resolve_hostname
  Line: 25
23.220.75.232 first octet: 23
None first octet: None
151.101.64.223 first octet: 151
```

py-spy

The code is running - but nothing happens...

- The process is stuck - no logs, no output
- We can’t attach a debugger
- `py-spy` lets us peek into the running Python process, see which thread or function is blocking
- Often used to inspect live issues safely

```
sudo uv run py-spy dump --pid 10758
```

dump_call_stack_with_py_spy.py

```
6   pool = queue.Queue(maxsize=1)
7   pool.put("CONN1")  # initially available connection
8
9   def worker(name):  2 usages
10      print(f"{name}: waiting for a connection from the pool...")
11      conn = pool.get()  # blocks if pool is empty
12      print(f"{name}: got connection {conn}")
13      time.sleep(3)  # simulate work
14      pool.put(conn)
15      print(f"{name}: returned connection {conn}")
16
17  def broken_returner():  1 usage
18      conn = pool.get()      # Takes connection but does NOT put it back
19      print(f"broken_returner: got {conn}, but does NOT put it back")   #
20
21  # Start threads
22  threading.Thread(target=broken_returner).start()
23  threading.Thread(target=worker, args=("worker-1",)).start()
24  threading.Thread(target=worker, args=("worker-2",)).start()
```

Terminal Local

```
(debugging-deep-dive) bityob@Yonatans-MacBook-Pro debugging-deep-dive %
broken_returner: got CONN1, but does NOT put it back
worker-1: waiting for a connection from the pool...
worker-2: waiting for a connection from the pool...
```

```
bityob@Yonatans-MacBook-Pro debugging-deep-dive % sudo uv run py-spy dump --pid 10758
Process 10758: /Users/bityob/Code/debugging-deep-dive/.venv/bin/python3 10-py-spy/dum
y
Python v3.12.0 (/Users/bityob/.local/share/uv/python/cpython-3.12.12-macos-aarch64-no

Thread 0x20900A0C0 (active): "MainThread"
    _shutdown (threading.py:1624)
Thread 0x171853000 (idle): "Thread-2 (worker)"
    wait (threading.py:355)
    get (queue.py:171)
    worker (dump_call_stack_with_py_spy.py:11)
    run (threading.py:1012)
    _bootstrap_inner (threading.py:1075)
    _bootstrap (threading.py:1032)
Thread 0x170847000 (idle): "Thread-3 (worker)"
    wait (threading.py:355)
    get (queue.py:171)
    worker (dump_call_stack_with_py_spy.py:11)
    run (threading.py:1012)
    _bootstrap_inner (threading.py:1075)
    _bootstrap (threading.py:1032)
```

gdb (GNU Debugger)

When even Python crashes

- Sometimes the crash happens **below** Python - in C extensions or the interpreter itself
- No traceback, no exception, just a *core dump*
- `gdb` lets us inspect that dump and see the native stack trace
- Essential for debugging native crashes (e.g., `libxml2`, `numpy`, `cython`, etc.)

Real Case

Real Case

- Crash happened during XML processing — no traceback, no logs

Real Case

- Crash happened during XML processing — no traceback, no logs
- Using `gdb` on the core dump revealed a native library issue

Real Case

- Crash happened during XML processing — no traceback, no logs
- Using `gdb` on the core dump revealed a native library issue
- Turns out, two dependencies linked against **different `libxml2` versions**

Real Case

- Crash happened during XML processing — no traceback, no logs
- Using `gdb` on the core dump revealed a native library issue
- Turns out, two dependencies linked against **different `libxml2` versions**
- The mismatch caused a memory access crash inside C code

Real Case

- Crash happened during XML processing — no traceback, no logs
- Using `gdb` on the core dump revealed a native library issue
- Turns out, two dependencies linked against **different `libxml2` versions**
- The mismatch caused a memory access crash inside C code
- `gdb` helped pinpoint the root cause quickly

debugging_with_gdb.py

```python
import io

import xmlsec
from lxml import etree

stream = io.BytesIO(
    b"""<?xml version="1.0" encoding="UTF-8"?>
<Envelope xmlns="urn:envelope" ID="ef115a20-cf73-11e5-aed1-3c15c2c2cc88">
  <Data>
    Hello, World!
  </Data>
</Envelope>
"""
)
root = etree.parse(stream).getroot()
xmlsec.tree.add_ids(root, ids: ["ID"])
print(etree.tostring(root, pretty_print=True).decode())
```

Terminal Local

```
root@00d1d6259714:/opt/app# .venv/bin/python debugging_with_gdb.py
Floating point exception (core dumped)
```

```
gdb .venv/bin/python <core-dump-file>
```

```
root@227ba9a606cd:/opt/app# gdb .venv/bin/python qemu_python_20251025-213743_13.core
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from .venv/bin/python...

warning: Loadable section ".dynstr" outside of ELF segments
  in /root/.local/share/uv/python/cpython-3.12.12-linux-x86_64-gnu/bin/python3.12
(No debugging symbols found in .venv/bin/python)

warning: core file may not match specified executable file.
[New LWP 13]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Core was generated by `.venv/bin/python debugging_with_gdb.py'.
Program terminated with signal SIGFPE, Arithmetic exception.
#0  0x0000004001082c51 in ?? () from /lib/x86_64-linux-gnu/libxml2.so.2
(gdb) q
```

```
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Core was generated by `.venv/bin/python debugging_with_gdb.py'.
Program terminated with signal SIGFPE, Arithmetic exception.
#0  0x0000004001082c51 in ?? () from /lib/x86_64-linux-gnu/libxml2.so.2
(gdb) bt
#0  0x0000004001082c51 in ?? () from /lib/x86_64-linux-gnu/libxml2.so.2
#1  0x0000004000f7dc50 in xmlHashLookup3 () from /lib/x86_64-linux-gnu/libxml2.so.2
#2  0x0000004000f8adc7 in xmlGetID () from /lib/x86_64-linux-gnu/libxml2.so.2
#3  0x0000004000ed8a49 in xmlSecAddIDs () from /lib/x86_64-linux-gnu/libxmlsec1.so.1
#4  0x0000004000e45dfb in PyXmlSec_TreeAddIds (self=<optimized out>, args=<optimized out>, kwargs=<optimized out>)
    at /root/.cache/uv/sdists-v9/pypi/xmlsec/1.3.13/JcbmZMbCx2Ml4pOCxpsEW/src/src/tree.c:192
#5  0x0000000001804a7c in cfunction_call ()
#6  0x0000000001815ce3 in _PyEval_EvalFrameDefault ()
#7  0x0000000001899422 in PyEval_EvalCode ()
#8  0x00000000018b6f42 in run_mod.llvm ()
#9  0x00000000019d67bf in pyrun_file ()
#10 0x00000000019d6e18 in _PyRun_SimpleFileObject ()
#11 0x00000000019d6cd0 in _PyRun_AnyFileObject ()
#12 0x00000000019157c4 in pymain_run_file_obj ()
#13 0x00000000019156d8 in pymain_run_file ()
#14 0x000000000191647b in Py_RunMain ()
#15 0x0000000001953c3a in pymain_main.llvm ()
#16 0x0000000001953a2d in main ()
```

Summary: The Debugging Journey

Summary: The Debugging Journey

- We began with simple, local debugging

Summary: The Debugging Journey

- We began with simple, local debugging
- Moved through containers, remote servers, and serverless environments

Summary: The Debugging Journey

- We began with simple, local debugging
- Moved through containers, remote servers, and serverless environments
- Ended with tracing, spying, and native crash analysis

Summary: The Debugging Journey

- We began with simple, local debugging
- Moved through containers, remote servers, and serverless environments
- Ended with tracing, spying, and native crash analysis
- The real skill isn't just using tools — it's knowing **which one fits each layer of control**

Summary: The Debugging Journey

- We began with simple, local debugging
- Moved through containers, remote servers, and serverless environments
- Ended with tracing, spying, and native crash analysis
- The real skill isn't just using tools — it's knowing **which one fits each layer of control**
- And above all, keep the **debugger's mindset**: curious, methodical, and never afraid to dive deeper

Any Questions?

bit.ly/debug-slides
bit.ly/debug-git

Yonatan Bitton
@bityob
linktr.ee/bityob