

Forum:[Hack Forums](#)

Thread: [Debugging Python with GDB](#)

Sigma - April 26, 2021, 6:48 pm

Debugging Python code with GDB is not a trivial task, but since GDB 7 it has gotten significantly easier. That version introduced a built in Python interpreter into GDB itself, which enabled the support for GDB specific Python commands such as py-bt, py-list etc.

Setup:

Install GDB using your distributions package manager.Ubuntu: gdb/focal-updates

CentOS: gdb.x86_64 : A GNU source-level debugger for C, C++, Fortran, Go and other languages

Install libpython.py from the appropriate GDB source!Identify which Python version is build into your GDB. There is an important distinction here! The GDB's Python version has nothing to do with other Python versions installed on the system or the executable we wish to examine the core file with. We only need to know it, so that the appropriate libpython.py extension can be downloaded.

```
$ gdb
```

```
(gdb) python
```

```
>import sys
```

```
>print(sys.version)
```

```
>end
```

```
3.6.9 (default, Apr 18 2020, 01:56:04)
```

```
[GCC 8.4.0]
```

Go to GitHub CPython repository: <https://github.com/python/cpython> Choose the appropriate branch for your Python version. In this example 3.6.

Either clone the repository and get the file that way or just download it using wget. The file is located under: cpython\Tools\gdb\libpython.py

Before downloading it using wget don't forget to ask github to give you the Raw output (somewhere next to the blame button):

```
~$ wget https://raw.githubusercontent.com/python/cpython/3.6/Tools/gdb/libpython.py
```

Put this file wherever it is convenient to access from the location you will be starting your GDB debugging session. It will need to be sourced into GDB for the python specific commands. There are 2 ways of doing this: Source it inside GDB itself: (gdb) source <your libpython.py location>

Add the source command to a .gdbinit file

(Optional) Build your C++/C source with debugging symbols enabled and rerun your application for generating the core files or to debug actually running. Usually there is enough information in the core without debug information to have a good sense about what has exactly gone wrong.

Notes:

To have a successful session with GDB one must use the exact Python version that was used for generating the core file. As we will see this isn't usually an issue as GDB will clearly tell us which executable created the coredump. In this case the executable will always be a Python interpreter. If an incompatible interpreter is chosen for debugging then GDB will clearly state it and produce lines and lines of gibberish:

```
Code root@3c4eba22324c:~/data# gdb python coredump/core.2947
```

```
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
```

```
Copyright (C) 2018 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:
```

```
<http://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"...
```

```
"/root/.pyenv/shims/python": not in executable format: File format not recognized
```

```
[New LWP 2947]
```

```
[New LWP 3120]
```

```
Core was generated by `'/root/.pyenv/versions/3.8.3/bin/python collect_dev.py collect_dataset --model_t'.
```

```
Program terminated with signal SIGSEGV, Segmentation fault.
```

```
#0 0x00007f0aa12fb0ff in ?? ()
```

```
[Current thread is 1 (LWP 2947)]
```

```
(gdb) bt
```

```
#0 0x00007f0aa12fb0ff in ?? ()
```

```
#1 0x00007f0c1c12ebe8 in ?? ()
```

```
#2 0x0000000017ed5910 in ?? ()
```

```
#3 0x00007fff1fa2c2b0 in ?? ()
```

```
#4 0x00007fff1fa2c350 in ?? ()
```

```
#5 0x0000000000000000 in ?? ()
```

Well alright, it didn't state it clearly that the executable version was incorrect but it does tell us that Core was generated by ... and we can see the question mark galore signaling to us that GDB has no idea what happened in the given frames.

Example core dump inspection:

Run GDB with the correct executable followed by the core file:

Code `gdb <executable> <corefile>`

`gdb /root/.pyenv/versions/3.8.3/bin/python coredump/core.2947`

Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Core was generated by `/root/.pyenv/versions/3.8.3/bin/python collect_dev.py collect_dataset --model_t'.

Program terminated with signal SIGSEGV, Segmentation fault.

#0 CCynomixData::FreeData (this=0x17ed5910) at Cynomix.cpp:2621

2621 Cynomix.cpp: No such file or directory.

[Current thread is 1 (Thread 0x7f0c32ef93c0 (LWP 2947))]

Now we can type `bt` (backtrace) which will print the whole stack:

Code #0 CCynomixData::FreeData (this=0x17ed5910) at Cynomix.cpp:2621

#1 0x00007f0aa12fb139 in CCynomixData::~CCynomixData (this=<optimized out>, __in_chrg=<optimized out>) at Cynomix.cpp:2400

#2 0x00007f0aa13013e8 in pyCynDestroy (ptr=0x17ed5910) at main.cpp:13

#3 0x00007f0c2ff96dae in ffi_call_unix64 () from /usr/lib/x86_64-linux-gnu/libffi.so.6

#4 0x00007f0c2ff9671f in ffi_call () from /usr/lib/x86_64-linux-gnu/libffi.so.6

To observe what the Python interpreter was doing we need to load the extension provided by gdb called `libpython.py`.

Code (gdb) `source libpython.py`

Now we can use the python specific debugging commands like `py-bt`, which similarly to the `bt` command will print the whole stack, but the this time for Python.

Code (gdb) `py-bt`

Traceback (most recent call first):

File "/root/.pyenv/versions/3.8.3/lib/python3.8/site-packages/dsml_model_pe/pe.py", line 96, in `__del__`

`CYN_DESTROY(self.cyn)`

Garbage-collecting