# Tools for Testing and Debugging

Note: the code here is not terribly Pythonic, because we're trying to illustrate particular points, so we must know, for instance, exactly what our type is. The examples are also intended to transfer without error between Py 2 and Py3, which leaves some things slightly an-idiomatic

Actual commands are in **bold**, {} mark arguments you should fill in with relevant values. See e.g. https://sourceware.org/gdb/onlinedocs/gdb/ for command details or use (help {command})

For these first exercises, try the debugger BEFORE looking at the source code. The answers are quite easy to spot, but in real code they're less obvious, so you want to practice using the tool. ALWAYS start at the first error!!

All python examples have the same basic structure. Run them by typing

**import {programname}**

**{programname}.main()**

- PDB_eg1

  - Demonstrates: array bounds violation

  **Info:**

  This example will generate an exception, try debugging it using both the default Python exception information and using PDB, the python debugger

  **Try:**

  Import PDB_eg1 and run its “main” function. You will get an exception.

  Run it through pdb using

  **import pdb**
  **pdb.run(“PDB_eg1.main()”)**

This will leave you at the pdb command line. Type

**break PDB_eg1.py:4**

to put a breakpoint at line 4 of PDB_eg1.py. Then type

**continue**

to start the code. The code will stop **BEFORE** it executes line 4. You can print values from within pdb using "**p**" (or "**pp**" to pretty print). Try printing i and a.

**p i**

**p a**

You will see that "**i"** has an expected value, but "**a**" is all -1. You can print a specific element of **a** using

**p a[1]**

To tell pdb to continue running the program, use "**continue**" again. Type "**continue**" once, and then print a again. What do you see now?

Keep typing **continue** until the code finishes. You should see the usual error message.

- GDB_eg2

- Demonstrates: uninitialised variable error

**Info:**

In a compiled language, where an array has to be given a type at compile time, but can be given a length at runtime it is possible to try to access an array that either isn't long enough (see example 1) or is given a type, but is never given a length. The closest equivalent in Python is to just not initialise the variable at all.

**Try:**

Try running example 2 through pdb in the same way. Put a break point at line 6.

See what

**p a**

produces as output.

- GDB_eg3

  - Demonstrates: named breakpoints

  **Info:**

In a real program, we would use the debugger to gain info to help us to work out WHY our input is going negative. In the example we can simply examine inside get_x

**Try:**

run the code, note that we get -ve arguments hence NaN

run the code through pdb, but this time put in a breakpoint on my_sqrt

**break GDB_eg3.my_sqrt**

continue running the code until it fails. The breakpoint will trigger several times, and "**p input**" at each of them. At the last one before the code fails, you should find that input is negative.

Now try a conditional breakpoint where you specify to only stop when input is negative

**break GDB_eg3.my_sqrt, input<0**

You should now see that the breakpoint only triggers just before the code fails.

Put a "**try**" "**except**" block around the call to "**math.sqrt**" so that the code doesn't fail when it encounters a negative value. Using the conditional breakpoint see how often the code encounters this negative input effect.

- GDB_eg4

  - Demonstrates: a simple real program with bugs

  **Info:**

The program is a simple binary search in ordered list of integers. It uses fixed size arrays, randomly filled with increasing values. There are two global options: "**norand**" which uses fixed rather than random data, and "**debug**" which fixes the random seed so that the same numbers occur every time and adds extra prints. Examine these first, by setting GDB_eg4.debug=True or GDB_eg4.norand=True

**Try:**

run the code a few times. Note that we get a random array, although always increasing. By default the target value is size/2. Supply an integer parameter to the call to "**main**" to search for that parameter (>> ./eg4 6) In pdb supply the number in the string passed to "**pdb.run**"

A list of the intentional bugs is at the end of this document

**Since Python doesn't have dynamic user controlled memory management, there is no equivalent of valgrind for internal Python functions.**

- Prof_eg

  - Demonstrates: the basics of profiling

  **Info:**

  This is a very boring program that just calls a few levels of functions. Use cProfile to see where it spends most of its time. The syntax is very similar to pdb,

  **import cProfile**
  **cProfile.run("Prof_eg.main()")**

  **Try:**

  Try swapping the division operation (line 46) for the multiply. Does this help? Hinder?

  Try adding file output (write_array function): how does this compare to the compute?

  Try running the function (other than main) that takes the longest through "**dis**"

  **import dis**

  **dis.dis({function_name})**

  This will tell you about the internal bytecode representation that Python uses. If you want to optimise a single routine, this will be needed to tell you where to optimise

BUGS in PDB_eg4:

Array definition has size-1 not size (line 78)

Loop on line 48 is fine unless array_size is 1, in which case we get no iterations

No type checking anywhere. You could specify a non integer type

Won't find last element (once first bug is fixed and the array size is as expected)

If you specify "**norand = True**" the code will fail to run because I'm attempting to set "**array_shape[i]**" on line 19 rather than "**array[i]**"

**"If target"** on line 91 will fail if target has value 0. Should just check for None