Débogage et profilage

Konrad HINSEN
Centre de Biophysique Moléculaire (Orléans)
and
Synchrotron Soleil (St Aubin)

Debugging vs. profiling

Debugging: Identifying the origin of "undesirable" behavior:

- **Crash**
  1) Python exception
  2) OS-level crash (segmentation fault, memory allocation fault, ...)
- **Non-termination**

  Program stuck in a loop or recursion
- **Wrong results**

Profiling: Identifying the parts of a program that require a lot of CPU time and/or memory.

First debug, then profile!

Python level vs. C level

Bugs and performance problems can occur in plain Python code, but also in extension modules written in C/C++/Cython/Fortran/etc.

Python code is analyzed using Python debuggers and Python profilers. Extension modules are analyzed using C-level debuggers and profilers.

If you don't know at which level your problem is located, start with the Python tools, which are easier to handle!

This course concentrates on Python-level analysis. C-level tools are mentioned, but not explained in any detail.

Debugging

Debuggers

Common features of debugging tools:

- **Post-mortem analysis**

  Analysis of the program state when an exception is raised.
- **Breakpoints**

  Defining places in the program where execution is halted

  to permit an inspection of the state of the program.
- **Single-stepping**

  Executing one line/statement at a time.
- **Tracing**

  Showing the value of an expressions at predefined points during program execution (just like adding a print statement!)

Python debuggers

- Module pdb in the Python standard library.
- Winpdb (http://winpdb.org/), a GUI debugger based on wxWindows.
- PuDB (http://pypi.python.org/pypi/pudb), a console-based GUI for PDB.
- pydb / pydbgr (http://code.google.com/p/pydbgr/), a more gdb-compatible enhancement of pdb

Integrated Developement Environments with debuggers:

- PyDev (http://pydev.org/), an Eclipse plugin
- WingIDE (http://wingware.com/)
- Komodo IDE (http://www.activestate.com/komodo/features/)

Debuggers for C, C++, Fortran...

- gdb (http://www.gnu.org/software/gdb/)
- ddd (http://www.gnu.org/software/ddd/), a GUI for gdb and other debuggers
- Compiler-specific debuggers

Integrated Development Environments with debuggers:

- Emacs (http://www.gnu.org/software/emacs/)
- Eclipse (http://www.eclipse.org/)
- KDevelop (http://www.kdevelop.org/)
- OS-specific: XCode (Apple), VisualStudio (Microsoft)

Post-mortem analysis

Frequent situation: your program crashes because of an uncaught exception and you want to understand the cause.

Either...

- run your program under debugger control and wait for the exception

- or run your program interactively (python -i, or inside IDLE) and launch pdb after the exception:

```
import pdb
pdb.pm()
```

Running under debugger control with pdb:

```
python -m pdb my_script.py
```

Instant PDB

Create the file $HOME/.local/lib/python2.6/site-packages/sitecustomize.py with the following content:

```
def info(type, value, tb):
    import sys
    if hasattr(sys, 'ps1') or not sys.stderr.isatty():
        sys.__excepthook__(type, value, tb)
    else:
        import traceback, pdb
        traceback.print_exception(type, value, tb)
        print
        pdb.pm()

import sys
sys.excepthook = info
del info
del sys
```

This makes Python enter pdb whenever an exception is encountered.

Note: This doesn't work under Ubuntu !!!
You need to modify /usr/lib/python2.6/sitecustomize.py

Breakpoints and single-stepping

Frequent situation: your program doesn't crash, but produces wrong results.

- Start your program under debugger control
- Set a breakpoint (pdb: b) before the point where the error occurs
- Run to the breakpoint (pdb: c) and single-step from there on

Single-stepping modes:

- Step into (pdb: s) stops at the next possible location, usually at the beginning of a function being called.
- Step over (pdb: n) stops after the next statement, executing it with all its function calls.
- Step out (pdb: r) stops when the current function ends.

Conditional breakpoints

Frequent situation: a breakpoint needs to be passed hundreds of times before something interesting happens. But you don't want to type "c" hundreds of times!

Conditional breakpoints in pdb:

condition <number> <condition>

When the condition is not fulfilled, execution resumes immediately.

Passage counter:

ignore <number> <n>

The breakpoint is ignored n times before becoming active.

Analyzing the program state

Location in source code:

- (w)here prints a stack trace (as for an exception)

- (l)ist shows 11 lines around the current one

- (u)p and (d)own move up and down in the stack trace

Current variable values:

- (p)rint prints the value of an arbitrary Python expression

Tracing expressions at breakpoints:

- command <number>
  print <expression>
  end

Profiling

Principles of Profiling

Observe the behaviour of a program while it is running:

- Measure execution time per function
- Count how often a function is called
- Follow memory allocation and deallocation

Profiling steps

1) Run the program under profiler control
    - Execution statistics are collected
    - Program is slowed down!
2) Analyze the statistics
    - Identify the functions that use most of the CPU time
    - Check memory allocations
    - ...

Some popular profiling tools

1) **Python:** module **cProfile**

2) **gprof**

- Unix (including MacOS)
- works with GNU compilers and most others

2) **Valgrind**

- Linux (others in preparation)
- works with GNU compilers and others
- best known for memory profiling

3) **VTune** (Intel)

- selected Intel processors
- works with all compilers

4) **Shark** (Apple)

- MacOS X only
- very easy to use and provides excellent analysis

Using Python's cProfile module

- Basic use: `python -m cProfile my_script.py`

- Keeping the execution statistics in a file for later analysis:

```
python -m cProfile -o my_script.profile my_script.py
```

- Profiling part of a program:

```
import cProfile
cProfile.run("my_function()")
```

- Inspecting the statistics:

```
import pstats
p = pstats.Stats("my_script.profile")
p.print_stats("time")
```

How it works:

- Modifies the interpreter to call a bookkeeping routine when a function is called and when it returns.

- Use this to measure the execution time of each function.

Using gprof

1) Recompile program adding the option -pg (gcc, gfortran, ...)

2) Run the program normally.

3) Run "gprof" to analyze the execution statistics.

How it works:

- Recompilation with -pg inserts calls to gprof's profiling library.

- This library runs a second execution thread that observes the main thread's behaviour at regular intervals (statistical sampling).

- The execution statistics are written to the file gmon.out.

- gprof analyzes the data in gmon.out.

Using Shark

1) Run SHARK

2) Launch the program from SHARK.

3) Wait for the end of the program or press “STOP” at some time.

4) Look at the profiling data.

How it works:

- Uses special registers in the CPU designed for profiling.

- Uses statistical sampling of the program state.

Exercises

Trouvez les bogues !

Le script simulateur.py contient une version modifiée du simulateur du système solaire. Un grand méchant y a introduit quatre erreurs. Identifiez-les (et corrigez-les) en utilisant pdb !

Pour vérifier si votre simulateur fonctionne correctement, lancez-le avec l'option '-v' pour afficher une visualisation du mouvement des planètes. Si la terre revient à sa position d'origine au bout d'un an, tout va bien.

Profilage

Après avoir corrigé les bogues du simulateur du système solaire, analysez sa performance avec cProfile.