UNIT 10 DEBUGGING AND PROFILING

Structure

10.1 INTRODUCTION

This session introduces you to the important modules that handle the basic debugger functions such as analysing stack frames and set breakpoints etc. It also gives a brief description of what profilers are and how profiling is performed on a Python applications.

10.2 OBJECTIVES

Upon completion of this unit you will be able to:

- *explain* the importance of performing Python profiling
- *identify* the modules that handles the basic debugger functions
- *use* these modules appropriately in the Python program
- *perform* profiling on an existing application

10.3 TERMINOLOGIES

Debugging: The process of finding and removing programming errors

Framework An abstraction in software providing a generic functionality that can be modified.

10.4 FINDING AND REMOVING PROGRAMMING ERRORS

Finding and removing programming errors is called debugging in shorten form. It is an important skill that all programmers should acquire since it is an integral part of programming.

In Python, there are two important modules that play a major role in debugging. They are bdb, the debugger framework and pdp, the Python debugger.

The following sections give a brief description on both the modules.

The bdp module handles basic debugger functions, like setting breakpoints or managing execution via the debugger.

The following syntax is used to define the exception, which is raised by the bdb class for quitting the debugger.

exception bdb.BdbQuit

The following class implements temporary breakpoints, ignore counts, disabling and (re-)enabling, and conditionals.

class bdb.Breakpoint(self, file, line, temporary=0, cond=None, funcname=None)

Breakpoints are indexed by number through a list called bpbynumber and by (file, line) pairs through bplist. The former points to a single instance of class Breakpoint. The latter points to a list of such instances since there may be more than one breakpoint per line.

When creating a breakpoint, its associated filename should be in canonical form. If a funcname is defined, a breakpoint hit will be counted when the first line of that function is executed. A conditional breakpoint always counts a hit.

Breakpoint instances have the following methods:

- deleteMe()
  Delete the breakpoint from the list associated to a file/line. If it is the last breakpoint in that position, it also deletes the entry for the file/line.
- enable()
  Mark the breakpoint as enabled.
- disable()
  Mark the breakpoint as disabled.
- pprint([out])

Print all the information about the breakpoint:

- The breakpoint number.
- If it is temporary or not.
- Its file, line position.
- The condition that causes a break.
- If it must be ignored the next N times.
- The breakpoint hit count.

- The other module that is important for debugging is called Python Debugger also known as (pdb). Use of this debugger to track down exceptions will allow you to examine the state of the program just before the error.
- The module pdb defines an interactive source code debugger for Python programs. It supports setting (conditional) breakpoints and single stepping at the source line level, inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame. It also supports post-mortem debugging and can be called under program control.
- The debugger is extensible and it is defined as the class pdb. The extension interface uses the modules bdb and cmd.

Activity 10.1:

What is a debugger framework (bdb)? State each function it handles with examples.

The debugger's prompt is (Pdb). Typical usage to run a program under control of the debugger is shown in Example 10.1

Example 10.1

```
>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

pdb.pycan also be invoked as a script to debug other scripts. For example:

Python -m pdb myscript.py

When invoked as a script, pdb will automatically enter post-mortem debugging if the program being debugged exits abnormally. After normal exit of the program, pdb will restart the program. Automatic restarting preserves pdb's state (such as breakpoints) and in most cases is more useful than quitting the debugger upon program's exit.

The typical usage to break into the debugger from a running program is to insert,

import pdb; pdb.set_trace()

at the location you want to break into the debugger. You can then step through the code following this statement, and continue running without the debugger using the c command.

The typical usage to inspect a crashed program is shown in Example 10.2.

Example 10.2:

```
>>> import pdb
>>> import mymodule
>>> mymodule.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./mymodule.py", line 4, in test
    test2()
  File "./mymodule.py", line 3, in test2
    print spam
NameError: spam
>>> pdb.pm()
> ./mymodule.py(3)test2()
-> print spam
(Pdb)
```

The module defines the following functions; each enters the debugger in a slightly different way:

pdb.run(statement[, globals[, locals]])

executes the statement (given as a string) under debugger control. The debugger prompt appears before any code is executed; you can set breakpoints and type continue, or you can step through the statement using step or next (all these commands are explained below). The optional global and local arguments specify the environment in which the code is executed; by default the dictionary of the module__main__ is used.

pdb.runeval(expression[, globals[, locals]])

Evaluate the expression (given as a string) under debugger control. When **runeval()** returns, it returns the value of the expression. Otherwise this function is similar to **run()**.

pdb.runcall(function[, argument, ...])

Call the function (a function or method object, not a string) with the given arguments. When **runcall()** returns, it returns whatever the function call returned. The debugger prompt appears as soon as the function is entered.

pdb.**set_trace()**

Enter the debugger at the calling stack frame. This is useful to hard- code a breakpoint at a given point in a program, even if the code is not otherwise being debugged (e.g. when an assertion fails).

pdb.post_mortem([traceback])

Enter post-mortem debugging of the given traceback object. If no traceback is given, it uses one of the exception that is currently being handled (an exception must be being handled if the default is to be used).

pdb.pm()

Enter post-mortem debugging of the trace back found in sys.last_traceback.

The run* functions and **set_trace()** are aliases for instantiating the **Pdb** class and calling the method of the same name. If you want to access further features, you have to do this yourself:

*class pdb.**Pdb**(completekey='tab', stdin=None, stdout=None, skip= None)*

Pdb is the debugger class.

The *completekey*, *stdin* and *stdout* arguments are passed to the underlying cmd.Cmd class;

The *skip* argument, if given, must be an iterable of glob-style module name patterns. (Glob module finds all path names matching a specified pattern) The debugger will not step into frames that originate in a module that matches one of these patterns.

Example call to enable tracing with *skip*:

import pdb; pdb.Pdb(skip=['django.*']).set_trace()

10.5 INTRODUCTION TO THE PROFILERS

A *profile* is a set of statistics that describes how often and for how long various parts of the program are executed. These statistics can be formatted into reports via the pstats module.

cProfile and profile provide deterministic profiling of Python programs.

The Python standard library provides three different implementations of the same profiling interface:

1) **cProfile** is recommended for most users; it's a C extension with reasonable overhead that makes it suitable for profiling long-running programs. Based on ls**prof**, contributed by Brett Rosen and Ted Czotter.
2) Available in Python version 2.5.
3) **profile**, a Python only module whose interface is imitated by **cProfile**, but which adds significant overhead to profiled programs. If you're trying to extend the profiler in some way, the task might be easier with this module. Originally designed and written by Jim Roskind.

   Changed in Python version 2.4: Now also reports the time spent in calls to built-in functions and methods.
4) **hotshot** was an experimental C module that focuses on minimising the overhead of profiling, at the expense of longer data post-processing times.

It is no longer maintained and may be dropped in a future version of Python.

The **profile** and **cProfile** modules export the same interface, so they are mostly interchangeable; **cProfile** has a much lower overhead but is newer and might not be available on all systems. **cProfile** is really a compatibility layer on top of the internal _lsprof module. The hotshot module is reserved for specialized usage.

Note: The profiler modules are designed to provide an execution profile for a given program, not for benchmarking purposes This particularly applies to benchmarking Python code against C code: the profilers introduce overhead for Python code, but not for C-level functions, and so the C code would seem faster than any Python code.

Activity 10.2:

What is a Python debugger (pdb) and what are the debugging functionalities it supports?

How to Perform Profiling

To profile a function that takes a single argument, you can do:

```
import cProfile
import re
cProfile.run('re.compile("test|bar")')
```

Use **profile** instead of **cProfile** if the latter is not available on your system.

The above action would run **re.compile()** and print profile results like the following figure:

```
     197 function calls (192 primitive calls) in 0.002 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.001    0.001 <string>:1(<module>)
     1    0.000    0.000    0.001    0.001 re.py:212(compile)
     1    0.000    0.000    0.001    0.001 re.py:268(_compile)
     1    0.000    0.000    0.000    0.000 sre_compile.py:172(_compile_charset)
     1    0.000    0.000    0.000    0.000 sre_compile.py:201(_optimize_charset)
     4    0.000    0.000    0.000    0.000 sre_compile.py:25(_identityfunction)
   3/1    0.000    0.000    0.000    0.000 sre_compile.py:33(_compile)
```

Figure 10.1: Profile Results

The first line indicates that 197 calls were monitored. Of those calls, 192 were *primitive*, meaning that the call was not induced via recursion. The next

line: 'Orderedby: standard name', indicates that the text string in the far right column was used to sort the output. The column headings include:

- **ncalls**

  for the number of calls,

- **tottime**

  for the total time spent in the given function (and excluding time made in calls to sub-functions)

- **percall**

  is the quotient of tottime divided by ncalls

- **cumtime**

  is the cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate even for recursive functions.

- **percall**

  is the quotient of cumtime divided by primitive calls

- **filename:lineno(function)**

  provides the respective data for each function

When there are two numbers in the first column (for example 3/1), it means that the function recursed. The second value is the number of primitive calls and the former is the total number of calls. Note that when the function does not recurse, these two values are the same, and only the single figure is printed.

Instead of printing the output at the end of the profile run, you can save the results to a file by specifying a filename to the run() function:

The pstats.Stats class reads profile results from a file and formats them in various ways.

The file cProfile can also be invoked as a script to profile another script. For example:

```
Python -m cProfile [-o output_file] [-s sort_order]
myscript.py
```

-owrites the profile results to a file instead of to stdout

-s specifies one of the sort_stats() sort values to sort the output by. This only applies when -ois not supplied.

The **pstats** module's **Stats** class has a variety of methods for manipulating and printing the data saved into a profile results file:

```
import pstats
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

The **strip_dirs()** method removed the extraneous path from all the module names. The **sort_stats()** method sorted all the entries according to the standard module/line/name string that is printed. The **print_stats()** method printed out all the statistics. You might try the following sort calls:

```
p.sort_stats('name')
p.print_stats()
```

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:

```
p.sort_stats('cumulative').print_stats(10)
```

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand which algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

```
p.sort_stats('time').print_stats(10)
```

to sort according to time spent within each function, and then print the statistics for the top ten functions.

You might also try:

```
p.sort_stats('file').print_stats(' init   ')
```

This will sort all the statistics by file name, and then print out statistics for only the class init methods (since they are spelled with__init_ in them). As one final example, you could try:

```
p.sort_stats('time', 'cum').print_stats(.5, 'init')
```

This line sorts statistics with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re: .5) of its original size, then only lines containing init are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (p is still sorted according to the last criteria) do:

```
p.print_callers(.5, 'init')
```

and you would get a list of callers for each of the listed functions.

If more functionality is needed, you will have to read the manual, or guess what the following functions do:

```
p.print_callees()
p.add('restats')
```

Activity 10.3:

What is profiling?

Refering to the examples above, profile a function that takes in a single argument.

Check Your Progress

Q-1 What is debugging? Discuss functions of bdb and pdb debugging modules of Python.

Q-2 Compare profilers and debuggers, give example of each.

Q-3 Discuss the implementation of profiling interface provided by Python standard library.

10.6 SUMMARY

The objective of this unit is to introduce you to important modules that handle the basic debugger functions and it also provides few examples in order to use these debugger functions appropriately. Further it provides an introduction on how to perform profiling in Python along with examples that will help you achieve it.

10.7 REFERENCES AND FURTHER READING

1) pdp – The Python Debugger

https://docs.python.org/2/library/pdb.html

Profiling –

https://developer.android.com/studio/profile/battery-historian

10.8 ANSWER TO CHECK YOUR PROGRESS

Ans-1 Refer section 10.4

Ans-2 Refer section10.5

Ans-3 Refer section 10.5