Python Library Reference for the Extended Python Debugger

Release 2.4.2pydb

Revised by Rocky Bernstein

July 29, 2006

Abstract

Python is an extensible, interpreted, object-oriented programming language. It supports a wide range of applications, from simple text processing scripts to interactive Web browsers.

We describe here only the Extended Python Debugger. The rest of the The `Python Reference Manual` should be consulted for other standard Python modules including the original `Python Debugger` `(pdb.py)`.

CONTENTS

The Extended Python Debugger

The extended Python debugger builds on work done in the standard `Python Debugger` (pdb.py).

On *not* inventing yet another interface:
In extending the command set and functionality, we've used the GNU debugger (gdb) as a guide for many reasons.

First the command set is likely to be complete, given its longevity. Also, because of its longevity and pervasiveness, it is likely to be more helpful to users—especially those that know gdb or my GNU Bash debugger bashdb or to some extent my GNU Make debugger. All of these follow the same command interface and the learning curve is reduced for people familiar with one of these; they are less likely to get confused when switching between debuggers.

But it also has been helpful for other programs. I digress for a little history.

When I first thought about adding the bash debugger into the GUI interface ddd, basically all I had to do was tell ddd that handling this construct (e.g., stepping) is like gdb. There were a few places where I told ddd not to follow gdb but Perl instead because the paradigm was more like a scripting language than a compiled language. But in the end, adding support for the bash debugger inside ddd was much more straightforward and required much less thought than if I had invented my own debugger command set.

After this was done and I fired up ddd, I noticed that, when my cursor was hovering over some of the buttons, short descriptions for the command were given. Furthermore there was a button called "customize bash" added setting variables inside. But I hadn't added a box widget for customization or modified any code for using tool tips. How did ddd do this?

Because I had copied the output format of gdb's info, set and show commands, ddd ran these commands on its own and parsed the output; it then used that output to form tool tips and create customization boxes.

In responses to a preliminary posting to comp.lang.python asking why the Python debugger was different from other debuggers, a number of people indicated that it didn't matter since they did not use the standard Python debugger, or did not use it much. To some extent, I wonder if this is not a chicken-and-egg problem: is the debugger lacking in usefulness because people don't use it much or do people not use the debugger because it is lacking in usefulness?

I'm not sure, but if the standard Python debugger is little used, keeping compatibility is not important.

So, in cases where the standard Python debugger was incompatible with gdb, the gdb commands have been used.

Least action: a design principle for debuggers in general:
By necessity, debuggers change the operation of the program they are debugging. And this can lead to unexpected and unwanted differences. It has happened so often that the term http://en.wikipedia.org/wiki/Heisenbug was coined to describe the situation where the adding the use of a the debugger (among other possibilities) changes behavior of the program so that the bug doesn't manifest itself anymore.

Of course a debugger, and this one in particular, tries hard to make itself transparent when not asked to do its thing. But there can be unavoidable differences. One such noticeable difference, mentioned in conjunction with set_trace() (see 1.3.5), is the speed at which the debugger runs. Another difference involves name-space issues since some of the debugger may live inside the debugged program.

Consequently, a general principle as a debugger writer (and a principle used in this debugger) is: *bring in services only when needed or requested.* For example, it is possible that we will add debugging threads and out-of-process debugging, including debugging from outside of the running computer. For thread debugging, most certainly the debugged program will, by necessity, add a thread to manage the others. For out of process debugging, most certainly some sort of communication mechanism will need to get added, e.g. a socket. However there are many programs that one might want to debug that might not use either threads or sockets and might not need debugging outside of the debugged program. So when debugging those, a debugger should not add either a thread or a socket, unless the features that require them are requested. And even then, not *until* they are requested.

1.1 Invoking the Debugger

First it should be noted that in various situations and for various reasons you might not be able to call the debugger directly from a command line or at the outset of the program. That's okay. See section 1.3.1 for how to invoke the debugger from inside your running Python program. Also, see section 1.3.2 for how to invoke after an error is encontered and you have a traceback.

Of the many additions to the standard Python debugger, pdb, three will be mentioned here.

First, this debugger should install itself somewhere in your command path, usually as `pydb` so you don't have to invoke it as an argument to the `python` command.

Second the extended debugger supports command switches courtesy of `optparse`. In particular `pydb` has the two very common options '`--help`' to show what options are available and '`--version`' to report the version that is installed.

Third, you need not supply a script name to debug at the outset. Usually though you will want to give the name of a script to debug and after that you may want to pass options for this script.

Thus the general form of invoking the debugger is:

`pydb` $\big[$*debugger-options…* $\big]$ $\big[$*python-script* $\big[$*script-options…* $\big]\big]$

python-script should be the Python source (usually has extension `.py` if any), and not a compiled or optimized Python program.

In contrast to running a program from a shell (or using the `gdb` debugger), no path searching is performed on *python-script*. Therefore *python-script* should be explicit enough (include relative or absolute file paths) so that the debugger can read it as a file name.

Similarly, the location of the Python interpreter used for the script will not necessarily be the one specified in the magic field (the first line of the file), but will be the Python interpreter that the debugger specifies. (In most cases they'll be the same and/or it won't matter.)

A detailed list of options is given next.

1.1.1 Debugger Command Options (`--trace, --output, --command, --nx, ...`)

Many options have both a short and a long version. For example, '`-x`' is the short version while '`--command`' is the long version.

—**basename** Report file locations as only the base filename, and omit the directory name. This is useful in running regression tests.

—**batch** Normally the debugger is entered before the debugged script is executed. The user sets breakpoints or starts interactively stepping through the program. However, if you want to start the script running without any interactive behavior from the debugger, use this option for example, if you know that a script will terminate with

an exception which causes the debugger to be entered at that point. The `--trace` option implicitly sets this option.

—cd *directory* Run `pydb` using *directory* as its working directory, instead of the current directory.

—command=*filename* **| –x** *filename* Run debugger script *filename*. This script is run after the user's `.pydbrc` file.

—exec=*command-list* **| –e** *command-list* Run debugger commands *command-list*. Commands should be separated by "`;;`"—the same as you would do inside the debugger. You may need to quote this option to prevent command shell interpretation, e.g. `--exec "break 20;; continue"`.

—nx | Before execution is started, a debugger configuration file `.pydbrc` is run. In some situations, for example regression testing the debugger, you want to make sure that such configuration files are not run and this option will do that.

—output=*filename* Write the normal output ('`stdout`') to the file *filename*. Useful when running a Python script without access to a terminal.

—error=*filename* Write the error output ('`stderr`') to file *filename*. Useful in running a Python script without access to a terminal.

—trace POSIX-style line tracing is available. In POSIX shells the short option for this is `-x`; however since we follow `gdb` conventions `-x` is used as a short option for `--command`. When line tracing is turned on, each location (file name and linenumber) is printed before the command is executed. This option can be used in conjunction with the `--output` and `--error` options described above when a terminal is not available or when not running interactively. The corresponding debugger command is '`set linetrace on`'. See 1.2.1 for more information.

1.1.2 Startup files (`.pydbrc`)

If a file `.pydbrc` exists in the user's home directory or in the current directory, it is read in and executed as if it had been typed at the debugger prompt. This is particularly useful for aliases. If two files exist, the one in the home directory is read first and aliases defined there can be overridden by the local file. Finally you can specify a command file to be read when invoking `pydb` and this is run last. See 1.1.1 for information on how to run a command file.

Sometimes you may not want to run startup files. For example, you may have a special installation script that uses the debugger and want to make sure a user's profile doesn't get in the way. See the '`--nx`' command option, 1.1.1.

For tracking down problems with command files, see the '`set cmdtrace on`' debugger command, 1.2.1. To run a debugger command script inside the debugger see the '`source`' command, 1.2.10.

1.2 Debugger Commands

In this section we describe debugger commands which can be used when the debugger is run as a standalone program.

Most commands can be abbreviated to one or two letters; e.g., '`h(elp)`' means that either '`h`' or '`help`' can be used to enter the help command (but not '`he`', '`hel`', '`H`', '`Help`', or '`HELP`'). Arguments to commands must be separated by whitespace (spaces or tabs). Optional arguments are enclosed in square brackets ('`[]`') in the command syntax; the square brackets must not be typed. Alternatives in the command syntax are separated by a vertical bar ('`|`').

Entering a blank line repeats the last command entered. Exception: if the last command was a '`list`' command, the next 11 lines are listed.

Commands that the debugger doesn't recognize are assumed to be Python statements and are executed in the context of the program being debugged. Python statements can also be prefixed with an exclamation point ('`!`'). This may be

---

a good way to inspect the program being debugged; it is even possible to change a variable or call a function. When an exception occurs in such a statement, the exception name is printed but the debugger's state is not changed.

The debugger supports aliases. Aliases can have parameters which allow a certain level of adaptability to the context under examination. See 1.2.10.

Debugger Prompt:
By default the debugger's prompt string is '`(Pydb)` ' with a trailing blank. Recursive invocations using the '`debug`' command strip off the trailing blanks, add a layer of parenthesis around the string, and add a trailing blank. For example, for the default prompt the first debug invocation will be '`((Pydb))` '.

There's currently a bug in the code where specified trailing blanks are chopped. Furthermore the prompt may change in the future to add a history number. It is generally not advisable to change the prompt.

If you do need to change the prompt see 1.2.1.

Multiple commands may be entered on a single line, separated by '`;;`'. (A single '`;`' is not used because it is the separator for multiple commands in a line that is passed to the Python parser.) No intelligence is applied to separating the commands; the input is split at the first '`;;`' pair, even if it is in the middle of a quoted string.

1.2.1 Status and Debugger Settings (`info`, `set`, `show`)

An `info` command shows things about the program being debugged. A `set` command modifies parts of the debugger environment. You can see these environment settings with the `show` command.

The subobptions to `info`, or `set` (or `show`) don't can be abbreviated to any prefix that uniquely specifies them. For example `info li` is a valid abbrevation for `info line` while `info l` is not since there is another subcommand (`locals`) which also starts with an 'l'.

In all of the set options that take "on" or "off" parameters, you can also use 1 for "on" and 0 for "off."

Each command has a corresponding `show` command to show the current value. See 1.2.1 for these counterparts.

If a readline module is available, `pydb` can keep track of the commands you type during your debugging sessions, so that you can be certain of precisely what happened. The `set history` commands to manage the command history facility.

POSIX-style line tracing is available and the `set linetrace` commands can be used to control that.

You may want to save the output of `pydb` commands to a file. See the `set logging` commands to control `pydb`'s logging.

Info (`info`)

Running this command without parameters will print the list of available info commands. Below is a description of the individual commands.

info args Show function/method parameters. See 1.2.4.

info break Show the list of breakpoints.

info globals Show the global variables. See 1.2.5.

info line Show the current line number in source file. If a function name is given, the starting line of the function is reported.

info locals Show the local variables. See 1.2.4.

info program Show the execution status of the program. The possible status is that the program is not running (e.g. in post-mortem dump), or the program is "stoppped" and if stopped at a breakpoint that is shown as well.

info source Information about the current Python file.

Set (`set`)

As with subobptions to `info`, or `set`, `show` subcommands can be abbreviated to any prefix that uniquely specifies them. For example `set lis 5` is a valid abbrevation for `info listsize 5` while `set li` is not since there is another subcommand (`linetrace`) which also starts with 'li'.

set basename on | off When showing filenames print only the basename. This option is useful in regression testing where the base file names are the same on different installations even though the directory path may be different. You may want to use this in other situtations as well, like showing a debugger session in a manual such as this one.

set cmdtrace on | off Show lines as they are read from the debugger command file (or '`source`' debugger command). This is useful in running regression tests, but it may be helpful in tracking down a problem in your `.pydbrc` file.

set history filename *filename* Set the filename in which to record the command history. (the list of previous commands of which a record is kept). The default file is `˜/.pydbhist`.

set history save Set saving of the history record on exit. Use "on" to enable the saving, and "off" to disable it. Without an argument, saving is enabled.

set history size Set the size of the command history, ie. the number of previous commands to keep a record of. The default is 256.

set linetrace on | off If this is set on, the position (file and linenumber) is shown before executing a statement. By default this is off. Using the command-line option '`--trace`' when invoking `pydb` implicitly sets this on. For information on '`--trace`', see 1.1.1.

```
$ pydb --basename --trace hanoi.py 2
(hanoi.py:2):
+ """Towers of Hanoi"""
(hanoi.py:3):
+ import sys
(hanoi.py:5):
+ def hanoi(n,a,b,c):
(hanoi.py:12):
+ if __name__=='__main__':
(hanoi.py:13):
+     i_args=len(sys.argv)
(hanoi.py:14):
+     if i_args != 1 and i_args != 2:
(hanoi.py:18):
+     n=3
(hanoi.py:20):
+     if i_args > 1:
(hanoi.py:21):
+       try:
(hanoi.py:22):
+         n = int(sys.argv[1])
(hanoi.py:27):
+     if n < 1 or n > 100:
(hanoi.py:31):
+     hanoi(n, "a", "b", "c")
--Call--
(hanoi.py:5):  hanoi
+ def hanoi(n,a,b,c):
(hanoi.py:6):  hanoi
+     if n-1 > 0:
(hanoi.py:7):  hanoi
+         hanoi(n-1, a, c, b)
--Call--
(hanoi.py:5):  hanoi
+ def hanoi(n,a,b,c):
(hanoi.py:6):  hanoi
+     if n-1 > 0:
(hanoi.py:8):  hanoi
+     print "Move disk %s to %s" % (a, b)
Move disk a to c
(hanoi.py:9):  hanoi
+     if n-1 > 0:
--Return--
(hanoi.py:9):  hanoi
+     if n-1 > 0:
(hanoi.py:8):  hanoi
+     print "Move disk %s to %s" % (a, b)
Move disk a to b
(hanoi.py:9):  hanoi
+     if n-1 > 0:
(hanoi.py:10):  hanoi
+         hanoi(n-1, c, b, a)
--Call--
(hanoi.py:5):  hanoi
+ def hanoi(n,a,b,c):
(hanoi.py:6):  hanoi
+     if n-1 > 0:
(hanoi.py:8):  hanoi
+     print "Move disk %s to %s" % (a, b)
Move disk c to b
```

```
(hanoi.py:9):  hanoi
```

```
+     if n-1 > 0:
--Return--
(hanoi.py:9):  hanoi
+     if n-1 > 0:
```

Adding linetracing output will slow down your program. Unless single stepping through a program, normally the debugger is called only at breakpoints or at the call and return of a function or method. However when line tracing is turned on, the debugger is called on execution of every statement.

That said, execution may still be pretty fast. If you want to slow down execution further, see the following option.

set linetrace delay *time* One of the useful things you can do with this debugger if you run it via a front-end GUI is watch your program as it executes. To do this, use '`set linetrace on`' which prints the location before each Python statement is run. Many front-end GUIs like the one in GNU Emacs and `ddd` will read the location and update the display accordingly.

There is however one catch—Python runs too fast. So by using this option you can set a delay after each statement is run in order for GNU and your eyes to catch up with Python. Specify a floating point indicating the number of seconds to wait. For example:

```
set linetrace delay 0.5 # 1/2 a second
```

In my experience half a second is about right.

set listsize *lines* Sets how many lines are shown by the `list` command. See 1.2.9.

set logging Prints `set logging` usage.

set logging on | off Enable or disable logging.

set logging file *filename* By default, `pydb` output will go to both the terminal and the logfile. Set `redirect` if you want output to go only to the log file.

set logging overwrite on | off By default, `pydb` will append to the logfile. Set `overwrite` if you want `set logging on` to overwrite the logfile instead.

set logging redirect on | off By default, `pydb` output will go to both the terminal and the logfile. Set `redirect` if you want output to go only to the log file.

set prompt *prompt-string* Set debugger's prompt string. By default it is '`(Pydb) `' with a trailing space. For information on how the prompt changes, see 1.2.

There's currently a bug in the code where specified trailing blanks specified. Furthermore the prompt may change in the future to add a history number. It is generally not advisable to change the prompt.

Show (`show`)

All of the "show" commands report some sort of status and all have a corresponding "set" command to change the value. See 1.2.1 for the "set" counterparts.

show args Show the argument list that was given the program being debugged or it is restarted

show basename Show short or long filenames

show cmdtrace Show the debugger commands before running

show commands Show the history of commands you typed. You can supply a command number to start with, or a '+' to start after the previous command number shown. A negative number starts from the end.

This command is available only if a readline module is available and supports the history saving.

show history Generic command for showing command history parameters. The command history filename, saving of history on exit and size of history file are shown.

show linetrace Show the line tracing status.

show linetrace delay Show the delay after tracing each line.

show listsize Show the number of source lines `pydb` will list by default.

show logging Show summary information of logging variables which can be set via `set logging`.

show logging file Show the current logging file.

show logging overwrite Show whether logging overwrites or appends to the log file.

show prompt Show the current debugger prompt.

show version Show the debugger version number.

1.2.2 Breakpoints (`break`, `tbreak`, `clear`, `commands`, `delete`, `disable`, `ignore`)

A breakpoint makes your program stop at that point. You can add conditions for each breakpoint. You can set breakpoints with the `break` command and its variants. You can specify the place where your program should stop by file and line number or by function name.

The debugger assigns a number to each breakpoint when you create it; these numbers are successive integers starting with 1. In many of the commands for controlling various features of breakpoints you use this number. Each breakpoint may be enabled or disabled; if disabled, it has no effect on your program until you enable it again.

The debugger allows you to set any number of breakpoints at the same place in your program. There is nothing unusual about this because different breakpoints can have different conditions associated with them.

b(reak) $\big[\big[$*filename***:** $\big]$*lineno* **|** *function* $\big[$**,** *condition* $\big]\big]$ With a *lineno* argument, set a break at that line number in the current file. With a *function* argument, set a break at the first executable statement within that function. The line number may be prefixed with a filename and a colon to specify a breakpoint in another file (probably one that hasn't been loaded yet). The file is searched on `sys.path`. Note that each breakpoint is assigned a number to which all the other breakpoint commands refer.

If a second argument is present, it is an expression which must evaluate to true before the breakpoint is honored.

Without an argument, list all breaks, including for each breakpoint: the number of times that breakpoint has been hit, the current ignore count, and the associated condition if any.

tbreak $\big[\big[$*filename***:** $\big]$*lineno* **|** *function* $\big[$**,** *condition* $\big]\big]$ Temporary breakpoint, which is removed automatically when it is first hit. The arguments are the same as those for `break`.

cl(ear) $\big[\big[$*filename***:** $\big]$*lineno* **|** *function* $\big]$ Clear breakpoint at specified line or function. Argument may be line number, function name, or '*' and an address. If a line number is specified, all breakpoints in that line are cleared. If a function is specified, the breakpoints at the beginning of the function are cleared. If an address is specified, breakpoints at that address are cleared.

With no argument, clears all breakpoints in the line where the selected frame is executing.

See also the `delete` command below which clears breakpoints by number.

commands $\big[\big[$*bpnumber* $\big]\big]$ Set commands to be executed when a breakpoint is hit. Give breakpoint number as the argument after "commands". With no bpnumber argument, commands refers to the last one set. The commands themselves follow starting on the next line. Type a line containing "end" to terminate the commands.

To remove all commands from a breakpoint, type commands and follow it immediately with `end`; that is, give no commands.

Specifying any command resuming execution (currently `continue`, `step`, `next`, `return`, `jump`, and `quit`) terminates the command list as if that command was immediately followed by `end`. This is because any

time you resume execution (even with a simple next or step), you may encounter another breakpoint—which could have its own command list, leading to ambiguities about which list to execute.

If you use the `silent` command in the command list, the usual message about stopping at a breakpoint is not printed. This may be desirable for breakpoints that are to print a specific message and then continue. If none of the other commands print anything, you see no sign that the breakpoint was reached.

delete $\big[$*bpnumber* $\big[$*bpnumber* ... $\big]\big]$ With a space-separated list of breakpoint numbers, clear those breakpoints. Without argument, clear all breaks (but first ask confirmation).

disable $\big[$*bpnumber* $\big[$*bpnumber* ... $\big]\big]$ Disable the breakpoints given as a space-separated list of breakpoint numbers. Disabling a breakpoint means it cannot cause the program to stop execution, but unlike clearing a breakpoint, it remains in the list of breakpoints and can be (re-)enabled.

enable $\big[$*bpnumber* $\big[$*bpnumber* ... $\big]\big]$ Enable the breakpoints specified.

ignore *bpnumber* $\big[$*count*$\big]$ Set the ignore count for the given breakpoint number. If count is omitted, the ignore count is set to 0. A breakpoint becomes active when the ignore count is zero. When non-zero, the count is decremented each time the breakpoint is reached, the breakpoint is not disabled, and any associated condition evaluates to true.

condition *bpnumber* $\big[$*condition*$\big]$ Condition is an expression which must evaluate to true before the breakpoint is honored. If condition is absent, any existing condition is removed; i.e., the breakpoint is made unconditional.

1.2.3 Resuming Execution (`step`, `next`, `finish`, `return`, `continue`, `jump`)

"Continuing" means resuming program execution until the program completes normally. In contrast, "stepping" means executing just one statement of the program. When continuing or stepping, the program may stop even sooner, due to a breakpoint or an exception.

s(tep) $\big[$*count*$\big]$ Execute the current line, stop at the first possible occasion (either in a function that is called or on the next line in the current function).

n(ext) $\big[$*count*$\big]$ Continue execution until the next line in the current function is reached or the function returns. The difference between 'next' and 'step' is that 'step' stops inside a called function, while 'next' executes called functions at (nearly) full speed, stopping only at the next line in the current function.

finish Continue execution until the current function returns. that point the 'retval' command can be used to show the return value. The short command name is rv.

See also 1.2.4 and 1.2.3.

return Make selected stack frame return to its caller. Control remains in the debugger, but when you continue execution will resume at the return statement found inside the subroutine or method. At present we are only able to perform this if we are in a subroutine that has a `return` statement in it. See also 1.2.4 and 1.2.3

c(ontinue) Continue execution, stop only when a breakpoint is encountered.

jump *lineno* Set the next line that will be executed. available only in the bottom-most frame. This lets you jump back and execute code again, or jump forward to skip code that you don't want to run.

Not all jumps are allowed—for instance it is not possible to jump into the middle of a `for` loop or out of a `finally` clause.

One common use for the `jump` statement is to get out of a loop. Sometimes the bounds of loops are computed in advance so you can't leave a loop early by say setting the value of the loop variable

Here's an example demonstrating this:

```
pydb ptest.py
(ptest.py:2):
(Pydb) list
  1   #!/bin/python
  2   ->for i in range(1,10):
  3        print i
  4   print "tired of this"
[EOF]
(Pydb) step
(ptest.py:3):
(Pydb) i=1000
(Pydb) step
1000
(ptest.py:2):
(Pydb) jump 4
(ptest.py:4):
(Pydb) step
tired of this
--Return--
--Return--
The program finished and will be restarted
(ptest.py:2):
(Pydb)
```

Not that the assignment of 1,000 to `i` took effect, although it had no effect on termintating the `for` loop; `jump` was needed to get out of the loop early.

1.2.4   Examining Call Frames (`info args`, `info locals`, `down`, `frame`, `up`)

Each line in the backtrace shows the frame number and the function name, if it exists and the place in a file where the statement is located.

Here is an example of a backtrace from a sample Towers of Hanoi program that is used in regression testing:

```
## 0 hanoi() called from file '/tmp/pydb/test/hanoi.py' at line 5
-> 1 hanoi() called from file '/tmp/pydb/test/hanoi.py' at line 6
## 2 in file '/tmp/pydb/test/hanoi.py' at line 29
## 3 in file '<string>' at line 1
## 4 run() called from file '/usr/lib/python2.4/bdb.py' at line 366
```

The `->` arrow indicates the focus. In the example, I issued an 'up' command which is why the focus is on 1 rather than 0 as it would normally be after a stop.

There are two "hanoi" frames listed because this is a hanoi called itself recursively. In frame 2 and 3 we don't have a function name listed. That's because there is none. Furthermore in frame 3 there is a funny "in file '<string>' at line 1." That's because there isn't even a file assocated with the command. The command issued:

```
exec cmd in globals, locals
```

This statement can be seen in frame 4. This is a bug which I hope to fix with a more informative message.

Finally, note that frames 2 and 3 really are not part of the program to be debugged but are part of the internal workings of the debugger. It's possible to hide this, but in the open spirit of Python for now it hasn't been hidden.

---

info args Show the method or function parameters and their values.

Here is an example of the output for the backtrace of the hanoi program shown at the beginning of this section:

```
(Pydb) info args
n= 3
a= a
b= b
c= c
(Pydb)
```

info locals Show all local variables for the given stack frame. This will include the variables that would be shown by `info args`.

where | T | bt $\big[$*count*$\big]$ Print a backtrace, with the most recent frame at the top. An arrow indicates the current frame, which determines the context of most commands.

With a positive number *count*, print at most many entries.

An example of a backtrace is given at the beginning of this section.

retval | rv Show the value that will be returned by the current function. This command is meaningful only just before a return (such as you'd get using the `finish` or `return` commands) or stepping after a return statement.

To change the value, make an assignment to the variable `__return__`.

See also 1.2.3.

down $\big[$*count*$\big]$ Move the current frame one level down in the stack trace (to a newer frame). With a count, which can be positive or negative, move that many positions.

up $\big[$*count*$\big]$ Move the current frame one level up in the stack trace (to an older frame). With a count, which can be positive or negative, move that many positions.

frame $\big[$*position*$\big]$ Move the current frame to the specified frame number. A negative number indicates position from the end, so `frame -1` moves to the newest frame, and `frame 0` moves to the oldest frame.

1.2.5 Examining Data (`print`, `pprint`, `examine`, `info globals`)

display $\big[$*format*$\big]$ *expression* Print value of expression *expression* each time the program stops. *format* may be used before *expression* as in the "print" command. *format* "i" or "s" or including a size-letter is allowed, and then *expression* is used to get the address to examine.

With no argument, display all currently requested auto-display expressions. Use "undisplay" to cancel display requests previously made.

undisplay $\big[$*format*$\big]$ *expression* Evaluate the *expression* in the current context and print its value. **Note:** `print` can also be used, but is not a debugger command—it executes the Python `print` statement.

p *expression* Evaluate the *expression* in the current context and print its value. One can also often have an expression printed by just typing the expression. If the first token doesn't conflict with a debugger built-in command Python will, by default, print the result same as if you did this inside a Python interpreter shell. To make things even more confused, a special case of running an arbitrary Python command is the `print` command. But note that the debugger command is just `p`.

So what's the difference? The debugger's print command encloses everything in a `repr()`, to ensure the resulting output is not too long. **Note:** *Should add info as to how to customize what "too long" means.* So if you want abbreviated output, or are not sure if the expression may have an arbitrarily long (or infinite) representation, then use `p`. If you want the output as Python would print it, just give the expression or possibly use python's `print` command.

pp *expression* Like the 'p' command, except the value of the expression is pretty-printed using the `pprint` module.

examine *expression* Print the type of the expression and pretty-print its value. For functions, methods, classes, and modules print out the documentation string if any. For functions also show the argument list.

> The examine debugger command in Perl is the model here, however much more work is needed. Note that 'x' is not a short name for "expression" (as it is in Perl's debugger), although you could easily make it be via an alias.

info globals Show all global variables. These variables are not just the variables that a programs sees via a `global` statement, but all of them that can be accessible.

1.2.6 Running Arbitrary Python Commands (`debug`, `!`)

[`!`]*statement* Execute the (one-line) *statement* in the context of the current stack frame. The exclamation point can be omitted unless the first word of the statement resembles a debugger command. To set a global variable, you can prefix the assignment command with a 'global' command on the same line, e.g.:

```
(Pydb) global list_options; list_options = ['-l']
(Pydb)
```

debug *statement* Enter a recursive debugger that steps through the code argument (which is an arbitrary expression or statement to be executed in the current environment). The prompt is changed to indicate nested behavior. See 1.2

1.2.7 Restarting a Python Script (`restart`, `run`)

restart *args...* Restart debugger and program via an `exec` call. All state is lost, and new copy of the debugger is used.

> Sometimes in debugging it is necessary to modify module code when one finds a bugs in them. Python will not notice dynamically that a module has changed and thus not reimport it (which also means that module initialization code is not rerun either). So in such a situation one must use `restart` rather than `run`.[1]

run *args...* Run or "soft" restart the debugged Python program. If a string is supplied that becomes the new command arguments. History, breakpoints, actions and debugger options are preserved. `R` is a short command alias for `run`.

> You may notice that the sometimes you can `step` into modules included via an `import` statement, but after a `run` this stepping skips over the import rather than goes into it. A similar situation is that you may have a breakpoint set inside class `__init__` code, but after issiuing `run` this doesn't seem to get called—and in fact it isn't run again!

> That's because in Python the `import` occurs only once. In fact, if the module was imported *before* invoking the program, you might not be able to step inside an `import` the first time as well.

> In such a situation or other situations where `run` doesn't seem to have the effect of getting module initialization code executed, you might try using `restart` rather than `run`.

1.2.8 Interfacing to the OS (`cd`, `pwd`, `shell`)

cd *directory* Set working directory to *directory* for debugger and program being debugged.

pwd Print working directory.

shell *statement* Execute the rest of the line as a shell command.

---

[1]It may be possible to unimport by removing a the module from a namespace, but if there are shared dynamically loaded objects those don't get unloaded.

1.2.9 Listing Program Code (`list`, `disassemble`)

disassemble $\big[\,arg\,\big]$ With no argument, disassemble at the current frame location. With a numeric argument, disassemble at the frame location at that line number. With a class, method, function, code or string argument, disassemble that.

l(ist) $\big[\,\text{-} \,$—$\, first \big[\,, last\,\big]\,\big]$ List source code. Without arguments, list *n* lines centered around the current line or continue the previous listing, where *n* is the value set by '`set listsize`' or shown by '`show listsize`'. The default value is 10.

'`list -`' lists *n* lines before a previous listing. With one argument other than '-', list *n* lines centered around the specified position. With two arguments, list the given range; if the second argument is less than the first, it is a count. *first* and *last* can be either a function name, a line number, or *filename:line-number*.

1.2.10 Interfacing to the debugger (`alias`, `complete`, `help`, `quit`, `source`, `unalias`)

alias $\big[\,name\,\big[\,\textbf{command}\,\big]\,\big]$ Create an alias called *name* that executes *command*. The command must not be enclosed in quotes. Replaceable parameters can be indicated by '`%1`', '`%2`', and so on, while '`%*`' is replaced by all the parameters. If no command is given, the current alias for *name* is shown. If no arguments are given, all aliases are listed.

Aliases may be nested and can contain anything that can be legally typed at the `pydb` prompt. Note that internal `pydb` commands can be overridden by aliases. Such a command is then hidden until the alias is removed. Aliasing is recursively applied to the first word of the command line; all other words in the line are left alone.

As an example, here are two useful aliases (especially when placed in the `.pydbrc` file):

```
#Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print "%1.",k,"=",%1.__dict__[k]
#Print instance variables in self
alias ps pi self
```

complete *command-prefix* If readline or one of readline-compatible interfaces such as pyreadline are available on your OS, the `complete` command will print a list of command names that start with *command-prefix*.

`complete` will also work on `info`, `set`, and `show` sub-command.

In addition the command-completion key (usually the tab key) can be used to complete command names, or `info`, `set`, and `show` subcommands.

h(elp) $\big[\,command\,\big[\,\textbf{subcommand}\,\big]\,\big]$ Without argument, print the list of available commands. With *command* as argument, print help about that command. '`help pydb`' displays the full documentation file; if the environment variable PAGER is defined, the file is piped through that command. Since the *command* argument must be an identifier, '`help exec`' must be entered to get help on the '`!`' command.

Some commands, `info`, `set`, and `show` can accept an additional subcommand to give help just about that particular subcommand. For example `help info line` give help about the `info line` command.

q(uit) Quit the debugger. The program being executed is aborted. For now, `kill` is a synonym for quit.

source *filename* Read commands from a file named *filename*. Note that the file `.pydbrc` is read automatically this way when `pydb` is started.

An error in any command terminates execution of the command and control is returned to the console.

For tracking down problems with command files, see the '`set cmdtrace on`' debugger command, 1.2.1.

unalias *name* Delete the specified alias.

1.3 The Debugger Module (`pydb`) and Class (`Pdb`)

The module `pydb` defines an interactive source code debugger for Python programs.

'pydb.py' can be invoked as a script to debug other scripts. For example:

```
python -m pydb myscript.py
```

As with `pydb` invocation (see 1.1), *myscript.py* must be fully qualified as a file, no path-searching is done to find it. Also it must be Python source, not compiled or optimized versions. Finally you may have to adjust `PYTHONPATH` if the module `pydb` is not found.

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

The debugger is extensible—it is defined as the class `Pdb`. When creating a new `Pdb` object, command completion is available if `readline` or `pyreadline` is available. In such circumstance, one can optionally specifying the key for command completion. The parameter name here is `completekey` an its default value is the tab key.

The `Pdb` extension interface of `Gdb` a gdb-like debugger command interface. This in turn inherts from `Cmd` and `Bdb`; `Cmd` handles command-line aspects (e.g. keyboard input, completion, help system, top-level command-invocation), while `Bdb` handles debugger aspects (e.g. breakpoints, stepping, call stack formatting). However `Cmd` and `Bdb` go through two more modules `pycmd` and `pybdb` which are extensions of `cmd` and `bdb` respectively. In general these extensions are used to paper over deficiencies or differences in operation that are needed.

All of the methods intended for outside use are documented. In fact the help documentation for commands comes from the document strings of the corresponding methods.

One group of methods that may be useful to subclass would be the output methods `errmsg`, `msg`, and `msg_nocr`. In early regression development, I subclassed these so that I could capture debugger output. It turned out, however, that I needed far too many tests and working this way would not scale.

1.3.1 Calling the Debugger from Inside your Program

When you issue `pydb myscript.py` you are running `pydb` first which then invokes your script `myscript.py` via Python's `exec` command. The debugger, `pydb`, tries hard to make itself transparent and thus strips its own program options, resets `sys.argv` and sets `__file__` to the values you would get by calling the debugged program directly.

There may be however some subtle differences. For example the Python interpreter used would be the one specified by `pydb` rather than possibly that specified inside `myscript.py`. Also `pydb` does not search your command path, as would be done if issued from a shell.

Of course you can arrange to get the same interpeter by putting `python` (with the right path) first before `pydb`; and you can give an explicit file path in the script name to debug.

But there are times when even this won't work right.

There is another approach which obviates this complexity and the attendant drawbacks. However in this approach though you need to modify your program to add calls to the debugger at special places. For this, the `pydb` function `set_trace()` (see 1.3.5) can be used. I've even this method to debug the debugger itself and to debug regression tests.

When the `pydb.set_trace()` function is called, the program stops before the next statement. To continue running the program, issue a debugger `next`, `step` or a `continue` command.

The exit the program use the `quit` or a termanal EOF.

To make this more clear, let's go through an example. Save this in a file called `hardtodebug.py`:

```
import pydb
# some code here
def test():
    # Force a call to the debugger in running code here
    pydb.set_trace()
    # ...
# ...
test()
x=5
```

Now here's a sample run of the program:

```
python hardtodebug.py a b c
--Return--
--Return--
(/tmp/hardtodebug.py:9):
(Pydb) list
   4             # Force a call to the debugger in running code here
   5             pydb.set_trace()
   6             # ...
   7     # ...
   8     test()
   9  -> x=5
(/tmp/hardtodebug.py:9):
(Pydb) restart
Re exec'ing
['pydb', 'hardtodebug.py', 'a', 'b', 'c']
(/tmp/hardtodebug.py:1):
(Pydb)
```

The first `--Return--` line printed is a result of the `set_trace()` exiting. But note that we stopped *after* the return from `test()`, which explains the *second* `--Return--` line. Because the next executable statement is after the implicit return. If you want to stop inside `test()` put a statement after the `set_trace()`, such as a `return` statement. Furthermore, if the program had ended at line 8, then *no* stopping would have occured becuase the end of the program is reached first.[2]

Also note that we can issue a `restart` which involves some hackery. But at least in this case it gets things right.

There is one final advantage of using `set_trace()`. When one is stepping code or has put a breakpoint in code, the interpreter has to be involved and calls debugger-checking code written in Python for every statement that he debugged program runs. And this has a noticeable effect. With `set_trace()` there is absolutely no overhead (provided there are no other breakpoints set in the program).

Line tracing

Rather than trace the entire program, if there is a specific portion that you want traced that can be done too, passing a list of debugger commands to `set_trace`.

---

[2]Perhaps this is a deficiency of the debugger. I'm not sure what the right way to address though. One could have another routine to stop at a return which would then skip intervening statements. Another possibility is to add a routine to have debugger stop inside the call, e.g. `set_trace` above. This is not hard to code and I've done so, but I'm not sure that doesn't just confuse things more.

```
import pydb
# some code here ...
pydb.set_trace(["set linetrace on", "continue"])
# This and subsequent lines will be traced
# more code ...
pydb.set_trace(["set linetrace off", "continue"])
```

1.3.2   Entering the Debugger after a Crash (Post-Mortem Debugging)

It is also to possible enter the debugger after a crash or traceback even though the program was not started via `pydb`. This is called *post-mortem debugging*.

Basically all you do is `import pydb` (if that hasn't been done already), and call `pydb.pm()`.

To make this more concrete we will give an example. We have the following text `mymodule.py`

```
def test():
    print spam
```

Now here's a sample session

```
>>> import mymodule
>>> mymodule.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "mymodule.py", line 2, in test
    print spam
NameError: global name 'spam' is not defined
>>> import pydb
>>> pydb.pm()
(/home/src/external-cvs/pydb/test/mymodule.py:2):  test
(Pydb) where
-> 0 test() called from file '/tmp/mymodule.py' at line 2
## 1 in file '<stdin>' at line 1
(Pydb) list
  1     def test():
  2 ->      print spam
[EOF]
(Pydb) quit
>>>
```

At present if you are using `ipython`, that captures the exception and `sys.last_traceback` will not be defined.

If you have a traceback stored say in variable `t`, instead of `pydb.pm()` above, use `pydb.post_mortem(t)`.

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

1.3.3 Yet Another Method of Invocation

With the caveat mentioned in 1.3.1 about problems with using `pydb` initially to run a script, I think most people will probably use the `pydb` command described in 1.1. However for completeness here we give another alternative.

Here's another way run a program under control of the debugger:

```
>>> import pydb
>>> import mymodule
>>> pydb.run('mymodule.test()')
(<string>:1):
(/usr/lib/python2.4/bdb.py:366):  run
(Pydb) continue
```

One difference betwen this kind of invocation that of Section 1.1 is that statements may get executed in the import of `mymodule` will not be debugged. For example if `mymodule` implements a standalone program and/or has a `__name__ = '__main__'` clause, you probably don't want to use this method.

1.3.4 Inheritance from class `Cmd`

Because `Pdb` inherts from `Cmd`, the following conventions are used.

All of the debugger commands listed in **??** are methods of a `Pdb` object. The method names are the command name prefixed by `do_`. For example the method handling the '`step`' command is `do_step`, and the command handling the '`frame`' command is `do_frame`.

If you have a `Pdb` object, it is possible to call any of the commands listed in 1.2 directly. Parameters needed by the methods (for example breakpoint numbers for the `enable` command), are passed as a single string argument. The string contains the values of the parameters and multiple parameters are separated by spaces. Each method parses out parameters and performs needed conversions. For example, a `Pdb` object, p, can enable breakpoint numbers 3, 5, and 10 like this: `p.do_enable("3 5 10")`, and this has the same effect as if `"enable 3 5 10"` were issued as a debugger command. String parameters should not have additional quotes in the string. For example to set the filename where history commands are to be saved (1.2.1), the method call would be `p.do_set("history filename /tmp/myhistfile")` without any quotes around `/tmp/myhistfile`.

Also inherited `Cmd`, is the help mechanism, although some customization has been made to allow for subcommand help. See the Python cmd module for more information.

If readline support is available on your OS, `Cmd` will use that and both command history and command completion will be available. In addition, the debugger `complete` command is defined (see 1.2.10). The `complete` command for example is used internally by GNU Emacs debugger `gud`.

1.3.5 Debugger Entry Functions

The `pydb` module defines the following functions, and each enters the debugger in a slightly different way:

pm (⌈*dbg_cmds=None*⌉)
>   Enter post-mortem debugging of the traceback found in `sys.last_traceback`. Note you may need to have `sys` imported priort to having the error raised to have sys.last_traceback set.
>
>   You can run debugger commands by passing this as a list as parameter `dbg_cmds`. For example if you want the display listsize to be 20 by default on entry pass `["set listsize 20",]`.

post_mortem (*traceback*⌈*dbg_cmds=None, cmdfile=None*⌉)
>   Enter post-mortem debugging of the given *traceback* object.

You can run debugger commands by passing this as a list as parameter `dbg_cmds`. For example if you want the display listsize to be 20 by default on entry pass `["set listsize 20",]`.

run (*statement* [, *globals* [, *locals*]])

Execute the *statement* (given as a string) under debugger control starting with the statement subsequent to the place that the this appears in your program.

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`' See 1.2.3 and 1.2.3 for explanations of these commands. The optional *globals* and *locals* arguments specify the environment in which the code is executed; by default the dictionary of the module `__main__` is used. (See the explanation of the `exec` statement or the `eval()` built-in function.)

Note that this is not at all like pydb's (or GDB's) `run` (see 1.2.7) debugger command. This function should be called from inside the program you are trying to debug.

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

Call the *function* (a function or method object, not a string) with the given arguments starting with the statement subsequent to the place that the this appears in your program..

When `runcall()` returns, it returns whatever the function call returned. The debugger prompt appears as soon as the function is entered.

runeval (*expression* [, *globals* [, *locals*]])

Evaluate the *expression* (given as a string) under debugger control starting with the statement subsequent to the place that the this appears in your program.

When `runeval()` returns, it returns the value of the expression. Otherwise this function is similar to `run()`.

set_trace ([*dbg_cmds=None, add_exception_hook=True*])

Enter the debugger at the statement which follows (in execution) the `set_trace()` statement. This hard-codes a call to the debugger at a given point in a program, even if the code is not otherwise being debugged. For example you might want to do this when an assertion fails.

It is useful in a couple of other situations. First, there may be some problem in getting the debugger to stop at this particular place for whatever reason (like flakiness in the debugger). Alternatively, using the debugger and setting a breakpoint can slow down a program a bit. But if you use this instead, the code will run as though the debugger is not present.

When the debugger is quitting, this causes the program to be terminated. If you want the program to continue instead, use the `debugger` function.

You can run debugger commands by passing this as a list in parameter `dbg_cmds`. Unless `add_-exception_hook` is set to `False`, we install an exception hook to enter the debugger on any otherwise unhandled exception.

1.4 How the Debugger Works

Some changes were made to the interpreter:

- `sys.settrace` (*func*) sets the global trace function

- there can also be a local trace function (see below)

Trace functions have three arguments: *frame*, *event*, and *arg*. *frame* is the current stack frame. *event* is a string: `'call'`, `'line'`, `'return'`, `'exception'`, `'c_call'`, `'c_return'`, or `'c_exception'`. *arg* depends on the event type.

The global trace function is invoked (with *event* set to `'call'`) whenever a new local scope is entered; it returns a reference to the local trace function to be used in that scope, or `None` if the scope couldn't be traced.

The local trace function returns a reference to itself (or to another function for further tracing in that scope), or `None` to turn off tracing in that scope.

Instance methods are accepted (and very useful) as trace functions.

The events have the following meanings:

`'call'` A function is called (or some other code block entered). The global trace function is called; *arg* is `None`; the return value specifies the local trace function.

`'line'` The interpreter is about to execute a new line of code (sometimes multiple line events on one line). The local trace function is called; *arg* is `None`; the return value specifies the new local trace function.

`'return'` A function (or other code block) is about to return. The local trace function is called; *arg* is the value that will be returned. The trace function's return value is ignored.

`'exception'` An exception has occurred. The local trace function is called; *arg* is a triple (*exception, value, traceback*); the return value specifies the new local trace function.

`'c_call'` A C function is about to be called. This may be an extension function or a builtin. *arg* is the C function object.

`'c_return'` A C function has returned. *arg* is `None`.

`'c_exception'` A C function has thrown an exception. *arg* is `None`.

Since an exception is propagated down the chain of callers, an `'exception'` event is generated at each level.

For more information on code and frame objects, see to the Python Reference Manual.

1.5   Files making up the Debugger

Here are the files making up the debugger and what is in them. Currently we use GNU automake and makes substitutings in files and Python programs which then get installed. A file with a `.in` suffix is a file that is the input for the substitution. The corresponding output file has the `.in` stipped off. For example `pydb.py.in` becomes `pydb.py` after a Python interpreter and package name (pydb) is substituted inside it.

pydb.py.in Python debugger. Contains user-callable routines, e.g. `run`, `set_trace`, `runeval`.

display.py Classes to support gdb-like display/undisplay for pydb, the Extended Python debugger. Class Display and DisplayNode are defined.

fns.py Functions to support the Debugger.

gdb.py.in Handles gdb-like command processing.

info.py Info subcommands. (Well, most of them).

pydbbdb.py Routines here have to do with the subclassing of bdb. Defines Python debugger Basic Debugger (Bdb) class. This file could/should probably get merged into bdb.py

pydbcmd.py Routines here have to do with parsing or processing commands, generally (but not always) they are not specific to pydb. They are sort of more oriented towards any gdb-like debugger. Also routines that need to be changed from cmd are here.

set.py Set subcommands. (Well, most of them).

show.py Show subcommands. (Well, most of them).

subcmd.py Implements a sub-command class for handling `info`, `set`, and `show` subcommand processing.

1.6 Installation

The Python code is available from the pydb files section on sourceforge.net. *If you have made an OS package and would like you're URL listed here, let me know*.

1.6.1 Installation options

The program is not configured using `setup.py` yet, but `configure`. Some configuration options that may be of interest.

—with—python Normally the `configure` uses your execution search-path variable (`PATH`) to find the python interpreter. However it is possible that you might now have python or the right python in your `PATH`. Soome installations may have several versions of python may be installed or installed in an unusual place. Use this configuration in such cases.

—with—lispdir This option overrides where to put files which can be used by GNU Emacs. It should be a place that GNU Emacs users will have listed in the `load-path` inside GNU Emacs.

—with—site-packages This option overrides where to put files the `pydb` package. It should be a place that python searches when modules are `import`'ed.

—enable—pyreadline If you don't have readline but have `pyreadline` or prefer to use that, use this option.

INDEX

Symbols

.pydbrc
> file, 3

B

breakpoints, 8

C

configuration
> file, debugger, 3

D

debugger
> configuration file, 3

debugging, 14

E

environment variables
> PAGER, 13

F

file
> .pydbrc, 3
> debugger configuration, 3

P

PAGER, 13

Pdb (class in pydb), 14

pm() (in module), 17

post_mortem() (in module), 17

pydb (standard module), **1**

R

run() (in module), 18

runcall() (in module), 18

runeval() (in module), 18

S

set_trace() (in module), 18