PDB - PYTHON DEBUGGER

11/26/2018 2:36 PM

help [command]	Print help about `command` or, if not specified, print a list of available commands.
quit	Quit from the debugger. The program being executed is aborted.
break [[filename:]line# -OR- function] [,condition]	Set a breakpoint at specified line# or function, or list breakpoints. • `break` → List all breakpoints. • `break [filename:]line#` → Set breakpoint at the specified line# in current (or specified) file. • `break function` → Set breakpoint at the first executable statement in the specified function. If `condition` is specified (with line# or function), execution pauses only when `condition` is true.
tbreak […]	Set a temporary breakpoint (removed automatically when it is first hit). Arguments are the same as for `break`.
clear [breakpoint# […]]	Clear specified breakpoint(s). Separate multiple breakpoints with a space. If none are specified, all are cleared.
disable [breakpoint# …]	Disables the specified breakpoint(s). Unlike clear, it remains in the list of breakpoints and can be (re-)enabled.
enable [breakpoint# …]	Enable the specified breakpoint(s).
ignore breakpoint# [count]	Sets the ignore count for the given breakpoint to `count`, or 0 if `count` is omitted. This lets you ignore a breakpoint some number of time. A breakpoint becomes active when its ignore count is zero. When the ignore count is non-zero, the count is decremented each time execution stops for that breakpoint.
condition breakpoint# [condition]	Set or clear the condition for the specified breakpoint. Execution will pause only if `condition` is true. If `condition` is absent, any existing condition is removed (i.e., breakpoint is made unconditional).
list [first[, last]]	List lines of code. • `list` → List 11 lines around the current line, or continue the previous listing. • `list first` → List 11 lines around `first`. • `list first, last` → List code from `first` to `last`. If `last` is less than `first`, then treat `last` as a count.
ll	List all source code for the current function or frame. Interesting lines are marked as for list. Alias: `longlist`
args	Print the argument list of the current function.
where	Print a stack trace, with the most recent frame at the bottom. An arrow indicates the current frame.
up	Move the current frame one level up in the stack trace (to an older frame). Execution begins at the top.
down	Move the current frame one level down in the stack trace (to a newer frame). Crashes occur at the bottom.
[!] statement	Execute `statement` in the context of the current stack frame. Details: Only one statement is allowed. The ! can be omitted unless the statement looks like a debugger command.
p expression	Print the value of `expression`.
pp expression	Pretty-print the value of `expression` (using the `pprint` module).
whatis expression	Print the type of the expression
display [expression]	Display the value of the expression if it changed, each time execution stops in the current frame. Without `expression`, list all display expressions for the current frame. *(New in Python version 3.2)*
undisplay [expression]	Do not display the expression any more in the current frame. Without expression, clear all display expressions for the current frame. *(New in Python version 3.2)*
next	Continue execution until the next statement *in this function*.
step	Continue execution until the next statement encountered in *this or any other function (e.g., caller or callee)*.
until	Continue execution until a line with higher line number is reached, or upon returning from current frame.
return	Continue execution until the current function returns.
continue	Continue execution. Stop only if a breakpoint is encountered.
jump line_number	Set the next line to be executed. This lets you jump back and execute code again, or jump forward to skip code you don't want to run. Details: This is only available in the bottom-most frame. Also, 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.
run [args...]	Restart the debugged program. History, breakpoints, actions and debugger options are preserved. If `args` is supplied, the program is run with `args` as command line arguments. Details: `args` is split with `shlex`. The result is used as the new `sys.argv`. Alias: `restart [args …]`
interact	Start an interactive interpreter. All global and local names from the current scope will be available. *(New in Python version 3.2)*
source	Try to get source code for the given object and display it. *(New in Python version 3.2)*

Adapted from Python documentation (partially verbatim). .pdbp aliases by Ned Batchelder. pdb complete trick by Esa-Matti Suuronen. Compiled by Alexander J. Quinn

Starting a debugging session (8 ways)

Command line
```
python -m pdb myscript.py
```

Manual break in code
```
pdb.set_trace()
```

Post mortem on current exception
```
try:
    …
except:
    pdb.post_mortem()
```

Call method
```
import pdb, mymodule
pdb.runcall(myfunction, a, b, c,…)
```

Run code snippet
```
import pdb, mymodule
pdb.run("mymodule.test()")
```

Evaluate string
```
import pdb, mymodule
pdb.runeval("mymodule.getX()")
```

Post mortem on last traceback
```
> myscript.myfunction()
Traceback (most recent call last):
…
> pdb.pm()
```

Post mortem on stored traceback
```
…
tb = sys.exc_info()[2]
…
pdb.post_mortem(tb)
```

Extending PDB

alias [name [command]]	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 pdb prompt. Note that internal pdb 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.
unalias name	Delete the specified alias.

commands [breakpoint#]	Specify a list of commands for breakpoint number `bpnumber`. The commands themselves appear on the following lines. Type a line containing just `end` to terminate the commands. An example:
	`(Pdb) commands 1`
	`(com) print some_variable`
	`(com) end`
	`(Pdb)`
	To remove all commands from a breakpoint, type commands and follow it immediately with end; that is, give no commands. With no `bpnumber` argument, `commands` refers to the last breakpoint set. You can use breakpoint commands to start your program up again. Simply use the command, or `step`, or any other command that resumes execution.
	Specifying any command resuming execution (currently `continue`, `step`, `next`, `return`, `jump`, `quit` and their abbreviations) 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.

.pdbrc

When pdb starts, any pdb commands in `~/.pdbrc` and/or `./.pdbrc` are executed (in that order), as if they had been typed into pdb.

```
# Print a dictionary, sorted. %1 is the dict, %2 is the prefix for the names.  (Credit: Ned Batchelder)
alias p_ for k in sorted(%1.keys()): print "%s%-15s= %-80.80s" % ("%2",k,repr(%1[k]))
alias pi p_ %1.__dict__ %1.   # Print the member variables of a thing
alias ps pi self.             # Print the member variables of self
alias pl p_ locals() local:   # Print the locals
alias nl n;;l                 # Next and list
alias sl s;;l                 # Step and list.

# Autocomplete from locals  (Credit: Esa-Matti Suuronen)
import pdb, rlcompleter
pdb.Pdb.complete = rlcompleter.Completer(locals()).complete

# Fix terminal if something funky happens that kills the echo (Unix only)
if sys.platform != "win32": import termios
if sys.platform != "win32": term_fd = sys.stdin.fileno()
if sys.platform != "win32": term_echo = termios.tcgetattr(term_fd)
if sys.platform != "win32": term_echo[3] = term_echo[3] | termios.ECHO
if sys.platform != "win32": term_result = termios.tcsetattr(term_fd, termios.TCSADRAIN, term_echo)
```