trepan Documentation

Release 1.0

Rocky Bernstein

Aug 17, 2017

Contents

CHAPTER 1

Abstract

This is a gdb-like debugger for Python. It is a rewrite of *pdb* from the ground up.

A command-line interface (CLI) is provided as well as an remote access interface over TCP/IP.

See the Tutorial for how to use. See ipython-trepan for using this in *ipython* or an *ipython notebook*.

This package is for Python 2.6 and 2.7. See trepan3k for the same code modified to work with Python 3. For Python before 2.6, use pydbgr .

CHAPTER 2

Features

Since this debugger is similar to other trepanning debuggers and *gdb* in general, knowledge gained by learning this is transferable to those debuggers and vice versa.

There's a lot of cool stuff here that's not in the stock Python debugger *pdb*.

Exact location information

Python reports line information on the granularity of a line. To get more precise information, we can (de)parse into Python the byte code around a bytecode offset such as the place you are stopped at.

So far as I know, there is no other debugger that can do this.

Debugging Python bytecode (no source available)

You can pass the debugger the name of Pytnon bytecode and many times, the debugger will merrily proceed. This debugger tries very hard find the source code. Either by using the current executable search path (e.g. *PATH*) or for some by looking inside the bytecode for a filename in the main code object (*co_filename*) and applying that with a search path which takes into account directory where the bytecode lives.

Failing to find source code this way, and in other situations where source code can't be found, the debugger will decompile the bytecode and use that for showing source test.

But if you happen to know where the source code is located, you can associate a file source code with the current name listed in the bytecode. See the set_substitute command for details here.

Source-code Syntax Colorization

Starting with release 0.2.0, terminal source code is colorized via pygments . And with that you can set the pygments color style, e.g. "colorful", "paraiso-dark". See set_style . Furthermore, we make use of terminal bold and emphasized

text in debugger output and help text. Of course, you can also turn this off. Starting with release 0.6.0, you can use your own pygments_style, provided you have a terminal that supports 256 colors. If your terminal supports the basic ANSI color sequences only, we support that too in both dark and light themes.

Command Completion

Starting with release 2.8, readline command completion has been added. Command completion is not just a simple static list, but varies depending on the context. For example, for frame-changing commands which take optional numbers, on the list of *valid numbers* is considered.

Terminal Handling

We can adjust debugger output depending on the line width of your terminal. If it changes, or you want to adjust it, see set_width .

Smart Eval

Starting with release 0.2.0, if you want to evaluate the current source line before it is run in the code, use `eval`. To evaluate text of a common fragment of line, such as the expression part of an *if* statement, you can do that with `eval?`. See eval for more information.

More Stepping Control

Sometimes you want small steps, and sometimes large stepping.

This fundamental issue is handled in a couple ways:

Step Granularity

There are now `step` *event* and `next` *event* commands with aliases to `s+`, `s>` and so on. The plus-suffixed commands force a different line on a subsequent stop, the dash-suffixed commands don't. Suffixes `>`, `<`, and `!` specify `call`, `return` and `exception` events respectively. And without a suffix you get the default; this is set by the *set different* command.

Event Filtering and Tracing

By default the debugger stops at every event: `call`, `return`, `line`, `exception`, `c-call`, `c-exception`. If you just want to stop at `line` events (which is largely what you happens in *pdb*) you can. If however you just want to stop at calls and returns, that's possible too. Or pick some combination.

In conjunction with handling *all* events by default, the event status is shown when stopped. The reason for stopping is also available via *info program*.

Event Tracing of Calls and Returns

I'm not sure why this was not done before. Probably because of the lack of the ability to set and move by different granularities, tracing calls and returns lead to too many uninteresting stops (such as at the same place you just were at). Also, stopping on function definitions probably also added to this tedium.

Because we're really handling return events, we can show you the return value. (*pdb* has an "undocumented" *retval* command that doesn't seem to work.)

Debugger Macros via Python Lambda expressions

Starting with release 0.2.3, there are debugger macros. In *gdb*, there is a *macro* debugger command to extend debugger commands.

However Python has its own rich programming language so it seems silly to recreate the macro language that is in *gdb*. Simpler and more powerful is just to use Python here. A debugger macro here is just a lambda expression which returns a string or a list of strings. Each string returned should be a debugger command.

We also have *aliases* for the extremely simple situation where you want to give an alias to an existing debugger command. But beware: some commands, like step inspect command suffixes and change their behavior accordingly.

We also envision a number of other ways to allow extension of this debugger either through additional modules, or user-supplied debugger command directories.

If what you were looking for in macros was more front-end control over the debugger, then consider using the experimental (and not finished) Bullwinkle protocol.

Byte-code Instruction Introspection

We do more in the way of looking at the byte codes to give better information. Through this we can provide:

- a *skip* command. It is like the *jump* command, but you don't have to deal with line numbers.
- disassembly of code fragments. You can now disassemble relative to the stack frames you are currently stopped at.
- Better interpretation of where you are when inside *execfile* or *exec*. (But really though this is probably a Python compiler misfeature.)
- Check that breakpoints are set only where they make sense.
- A more accurate determination of if you are at a function-defining *def* statement (because the caller instruction contains `MAKE_FUNCTION`.)

Even without "deparsing" mentioned above, the abilty to disassemble by line number range or byte-offset range lets you tell exactly where you are and code is getting run.

Debugger Command Arguments can be Variables and Expressions

Commands that take integer arguments like *up*, *list* or *disassemble* allow you to use a Python expression which may include local or global variables that evaluates to an integer. This eliminates the need in *gdb* for special "dollar" debugger variables. (Note however because of *shlex* parsing ,expressions can't have embedded blanks.)

Out-of-Process Debugging

You can now debug your program in a different process or even a different computer on a different network!

Egg, Wheel, and Tarballs

Can be installed via the usual *pip* or *easy_install*. There is a source tarball. How To Install has full instructions and installing from git and by other means.

Modularity

The Debugger plays nice with other trace hooks. You can have several debugger objects.

Many of the things listed below doesn't directly effect end-users, but it does eventually by way of more robust and featureful code. And keeping developers happy is a good thing.(TM)

- Commands and subcommands are individual classes now, not methods in a class. This means they now have properties like the context in which they can be run, minimum abbreviation name or alias names. To add a new command you basically add a file in a directory.
- I/O is it's own layer. This simplifies interactive readline behavior from reading commands over a TCP socket.
- An interface is it's own layer. Local debugging, remote debugging, running debugger commands from a file (*source*) are different interfaces. This means, for example, that we are able to give better error reporting if a debugger command file has an error.
- There is an experimental Python-friendly interface for front-ends
- more testable. Much more unit and functional tests. More of *pydb*'s integration test will eventually be added.

Documentation

Documentation: http://python2-trepan.readthedocs.org

CHAPTER 3

How to install

Using pip

If you are using pyenv or don't need special root access to install:

```
$ pip install trepan # or trepan3k for Python 3.x
```

If you need root access you may insert *sudo* in front or become root:

```
$ sudo pip install trepan
```

or:

```
$ su root
# pip install trepan
```

Using easy_install

Basically the same as using *pip*, but change "pip install" to "easy_install":

```
$ easy_install trepan  # or trepan3k
```

```
$ git clone https://github.com/rocky/python2-trepan.git
$ cd python-trepan
$ make check-short # to run tests
$ make install # if pythonbrew or you don't need root access
$ sudo make install # if pythonbrew or you do need root access
```

Above I used GNU "make" to run and install. However this just calls *python setup.py* to do the right thing. So if you are more familiar with *setup.py* you can use that directly. For example:

```
$ ./setup.py test
$ ./setup.py install
```

CHAPTER 4

Entering the Trepan Debugger

Contents

Invoking the Debugger Initially

The simplest way to debug your program is to call run *trepan2* (or *trepan3k* for Python 3). Give the name of your program and its options and any debugger options:

```
$ cat test.py
print('Hello, World!')
```

```
$ trepan2 test.py # or trepan3k test.py
```

For help on trepan2's or trepan3k's options, use the `--help` option.

```
$ trepan2 --help
Usage: trepan2 [debugger-options] [python-script [script-options...]]
...
```

To separate options to the program you want to debug from trepan2's options put – after the debugger's options:

```
$ trepan2 --trace -- test.py --test-option1 b c
```

If you have previously set up remote debugging using `trepan2 --server`, you'll want to run the client version of *trepan2* which is a separate program `trepan2c`.

Calling the debugger from IPython

Installing the IPython extension

Use the trepan IPython extension.

To install execute the the following code snippet in an IPython shell or IPython notebook cell:

or put *trepanmagic.py* in *$HOME/.python/profile_default/startup*:

```
cd `$HOME/.python/profile_default/startup`:
wget https://raw.github.com/rocky/ipython-trepan/master/trepanmagic.py
```

Trepan IPython Magic Functions

After installing the trepan extension, the following IPython magic functions are added:

- *%trepan_eval* evaluate a Python statement under the debugger
- *%trepan* run the debugger on a Python program
- *%trepan_pm* do post-mortem debugging

Example

See also the examples directory.

Calling the debugger from an Interactive Python Shell

Note: by "interactive python shell" I mean running "python" or "python -i" and this is distinct from going into IPython which was covered in the last section.

Put these lines in a file:

```
import inspect
from trepan.api import run_eval
def debug(str):
  frame = inspect.currentframe()
  return run_eval(str, globals_=frame.f_globals, locals_=frame.f_locals)
print(".pythonrc.py loaded") # customize or remove this
```

A copy of the above can be found here. I usually put these line in *$HOME/.pythonrc.py*. Set the environment variable *PYTHONSTARTUP* to *$HOME/.pythonrc.py*.

After doing this, when you run *python -i* you should see on entry the *print* message from the file. For example:

```
$ python -i
Python ...
Type "help", "copyright", "credits" or "license" for more information.
.pythonrc.py loaded
>>>
```

If you see the above ".pythonrc.py" message, great! If not, it might be that *PYTHONSTARTUP* is not defined. Here run:

and you should see the ".pythonrc.py" message as shown above.

Once that code is loaded, the *debug()* function is defined. To debug some python code, you can call that function. Here is an example:

```
>>> import os.path
>>> debug('os.path.join("a", "b")')
(/tmp/eval_stringBMzXCQ.py:1 remapped <string>): <module>
-> 1 os.path.join("a", "b")
(trepan2) step
(/home/rocky/.pyenv/versions/2.7.8/lib/python2.7/posixpath.py:68): join
-> 68 def join(a, *p):
(trepan2) continue
'a/b'
>>>
```

Note in the above, we pass to the *debug()* function a *string*. That is, we pass *'os.path.join("a", "b")'*, not *os.path.join("a", "b")* which would have the effect of running the code to be evaluated first *before* calling *debug()*. This is not an error, but debugging evaluating a string, is probably not what you want to do.

To do: add and document run_call()

Calling the debugger from your program

Sometimes it is not feasible to invoke the program from the debugger. Although the debugger tries to set things up to make it look like your program is called, sometimes the differences matter. Also the debugger adds overhead and slows down your program.

Another possibility then is to add statements into your program to call the debugger at the spot in the program you want. To do this, `import trepan.api` and make a call to *trepan.api.debug()*. For example:

```
# Code run here trepan2/trepan3k doesn't even see at all.
# ...
from trepan.api import debug
# trepan is accessible but inactive.
```

```
# work, work, work...
debug() # Get thee to thyne debugger!
```

Since *debug()* is a function, call it can be nested inside some sort of conditional statement allowing one to be very precise about the conditions you want to debug under. And until first call to *debug()*, there is no debugger overhead.

debug() causes the statement after the call to be stopped at. Sometimes though there is no after statement. In this case, adding the named parameter `step_ignore=0` will cause the debugger to be entered inside the *debug()* call:

```
# ...
def foo():
   # some code
   debug(step_ignore=0) # Stop before even returning from the debug() call
foo()  # Note there's no statement following foo()
```

If you want your startup profile to get run, perhaps you want to set your pygments style, add *start_opts={'startup-profile': True}*. For example:

```
debug(start_opts={'startup-profile': True})
```

Calling the debugger from pytest

Install pytest-trepan:

```
pip install pytest-trepan
```

After installing, to set a breakpoint to enter the trepan debugger:

```
import pytest
def test_function():
    ...
    pytest.trepan()     # get thee to thyne debugger!
    x = 1
    ...
```

The above will look like it is stopped at the *pytest.trepan()* call. This is most useful when this is the last statement of a scope. If you want to stop instead before `x = 1` pass `immediate=False` or just `False`:

```
import pytest
def test_function():
    ...
    pytest.trepan(immediate=False)
    # same as py.trepan(False)
    x = 1
    ...
```

You can also pass as keyword arguments any parameter accepted by *trepan.api.debug()*.

To have the debugger entered on error, use the `--trepan` option:

```
$ py.test --trepan ...
```

Set up an exception handler to enter the debugger on a signal

This is really just a variation of one of the other methods. To install and call the debugger on signal *USR1*:

```
import signal
def signal_handler(num, f):
  from trepan.api import debug; debug()
  return
signal.signal(signal.SIGUSR1, signal_handler)
# Go about your business...
```

However, if you have entered the debugger either by running intially or previously via a debug() call, trepan has already set up such default handlers for many of the popular signals, like *SIGINT*. To see what *trepan2* has installed use the `info signals` command:

```
(trepan2) info signals INT
 Signal        Stop   Print   Stack   Pass    Description
 SIGINT        Yes    Yes     No      No      Interrupt
(trepan2) info signals
Signal        Stop    Print   Stack   Pass    Description

SIGHUP        Yes     Yes     No      No      Hangup
SIGSYS        Yes     Yes     No      No      Bad system call
...
```

Commonly occuring signals like *CHILD* and unmaskable signals like *KILL* are not intercepted.

Set up an exception handler allow remote connections

The extends the example before to set to allow remote debugging when the process gets a *USR1* signal

```
import signal

def signal_handler(num, f):
    from trepan.interfaces import server as Mserver
    from trepan.api import debug
    connection_opts={'IO': 'TCP', 'PORT': 1955}
    intf = Mserver.ServerInterface(connection_opts=connection_opts)
    dbg_opts = {'interface': intf}
    print('Starting TCP server listening on port 1955.')
    debug(dbg_opts=dbg_opts)
    return

signal.signal(signal.SIGUSR1, signal_handler)
# Go about your business...

import time
import os
print(os.getpid())
for i in range(10000):
    time.sleep(0.2)
```

Now run that:

```
$ python /tmp/foo.py
8530
```

From above output we helpfully listed the pid of the Python process we want to debug.

Now in a shell we send the signal to go into the debugger listening for commands on port 1955. You will have to adjust the process id.

```
$ kill -USR1 8530   # Adjust the pid to what you see above
```

And in the shell where we ran */tmp/foo.py* you should now see the new output:

```
$ python /tmp/foo.py
8530
Starting TCP server listening on port 1955. # This is new
```

Back to the shell where we issued the *kill -USR1* we can now attach to the debugger on port 1955:

```
$ trepan2 --client --port 1955
Connected.
(/tmp/foo.py:11 @101): signal_handler
-- 11     return
(trepan2*) backtrace
  6              connection_opts={'IO': 'TCP', 'PORT': 1955}
  7              intf = Mserver.ServerInterface(connection_opts=connection_opts)
  8              dbg_opts = {'interface': intf}
  9              print('Starting TCP server listening on port 1955.')
 10              debug(dbg_opts=dbg_opts)
 11  ->          return
 12
 13          signal.signal(signal.SIGUSR1, signal_handler)
 14          # Go about your business...
(trepan2*) list
 ->   0 signal_handler(num=10, f=<frame object at 0x7f9036796050>)
      called from file '/tmp/foo.py' at line 11
 ##   1 <module> file '/tmp/foo.py' at line 20
```

Startup Profile

A startup profile is a text file that contains debugger commands. For example it might look like this:

```
$ cat ~/.config/profile
set autolist
set different on
set autoeval on
set style colorful
print("My trepan startup file loaded")
$
```

CHAPTER 5

Command Syntax

Debugger Command Syntax

Command names and arguments are separated with spaces like POSIX shell syntax. Parenthesis around the arguments and commas between them are not used. If the first non-blank character of a line starts with #, the command is ignored.

Commands are split at whereever *;;* appears. This process disregards any quotes or other symbols that have meaning in Python. The strings after the leading command string are put back on a command queue, and there should be white space around ';;'.

Within a single command, tokens are then white-space split. Again, this process disregards quotes or symbols that have meaning in Python. Some commands like *eval*, *macro*, and *break* have access to the untokenized string entered and make use of that rather than the tokenized list.

Resolving a command name involves possibly 4 steps. Some steps may be omitted depending on early success or some debugger settings:

1. The leading token is first looked up in the macro table. If it is in the table, the expansion is replaces the current command and possibly other commands pushed onto a command queue. Run *help macros* for help on how to define macros, and *info macro* for current macro definitions.

2. The leading token is next looked up in the debugger alias table and the name may be substituted there. See "help alias" for how to define aliases, and "show alias" for the current list of aliases.

3. After the above, The leading token is looked up a table of debugger commands. If an exact match is found, the command name and arguments are dispatched to that command.

4. If after all of the above, we still don't find a command, the line may be evaluated as a Python statement in the current context of the program at the point it is stoppped. However this is done only if "auto evaluation" is on. It is on by default.

If *auto eval* is not set on, or if running the Python statement produces an error, we display an error message that the entered string is "undefined".

If you want python-, ipython- or bpython-like shell command-processing, it's possible to go into an python shell with the corresponding command. It is also possible to arrange going into an python shell every time you enter the debugger.

See also:

help syntax suffixes

Command suffixes which have special meaning

Some commands like *step*, or *list* do different things when an alias to the command ends in a particular suffix like >.

Here are a list of commands and the special suffixes:

command	suffix
list	>
step	+, -, <, >
next	+, -, <, >
quit	*!*
kill	*!*
eval	*?*

See the help on the specific commands listed above for the specific meaning of the suffix.

CHAPTER 6

Trepan Command Reference

Following *gdb*, we classify commands into categories. Note though that some commands, like *quit*, and *restart*, are in different categories and some categories are new, like *set*, *show*, and *info*.

Breakpoints

Making the program stop at certain points

A *breakpoint* can make your program stop at that point. 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.

The simplest sort of breakpoint breaks every time your program reaches a specified place. You can also specify a condition for a breakpoint. A condition is just a Boolean expression in your programming language. A breakpoint with a condition evaluates the expression each time your program reaches it, and your program stops only if the condition is true.

This is the converse of using assertions for program validation; in that situation, you want to stop when the assertion is violated-that is, when the condition is false.

Break conditions can have side effects, and may even call functions in your program. This can be useful, for example, to activate functions that log program progress, or to use your own print functions to format special data structures. The effects are completely predictable unless there is another enabled breakpoint at the same address. (In that case, pydb might see the other breakpoint first and stop your program without checking the condition of this one.) Note that breakpoint commands are usually more convenient and flexible than break conditions for the purpose of performing side effects when a breakpoint is reached.

Break conditions can be specified when a breakpoint is set, by adding a comma in the arguments to the break command. They can also be changed at any time with the *condition* command.

Break (set a breakpoint)

break [*location*] [if *condition*]]

With a line number argument, set a break there in the current file. With a function name, set a break at first executable line of that function. Without argument, set a breakpoint at current location. If a second argument is *if*, subsequent arguments given an expression which must evaluate to true before the breakpoint is honored.

The location line number may be prefixed with a filename or module name and a colon. Files is searched for using *sys.path*, and the *.py* suffix may be omitted in the file name.

Examples:

```
break               # Break where we are current stopped at
break if i < j      # Break at current line if i < j
break 10            # Break on line 10 of the file we are
                    # currently stopped at
break os.path.join # Break in function os.path.join
break os.path:45    # Break on line 45 of os.path
break myfile:5 if i < j # Same as above but only if i < j
break myfile.py:45 # Break on line 45 of myfile.py
break myfile:45     # Same as above.
```

See also:

tbreak, and *condition*.

Clear (Remove all breakpoints on a line)

clear [*linenumber*]

Clear some breakpoints by line number.

See also:

delete

Condition (add condition to breakpoint)

condition *bp_number condition*

bp_number is a breakpoint number. *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.

Examples:

```
condition 5 x > 10  # Breakpoint 5 now has condition x > 10
condition 5         # Remove above condition
```

See also:

break, *tbreak*.

Delete (remove breakpoints)

delete [*bpnumber* [*bpnumber*...]]

Delete some breakpoints.

Arguments are breakpoint numbers with spaces in between. To delete all breakpoints, give no argument. Without arguments, clear all breaks (but first ask confirmation).

See also:

clear

Disable (disable breakpoints)

disable *bpnumber* [*bpnumber* ...]

Disables the breakpoints given as a space separated list of breakpoint numbers. See also *info break* to get a list.

See also:

enable

Enable (enable breakpoints)

enable *bpnumber* [*bpnumber* ...]

Enables the breakpoints given as a space separated list of breakpoint numbers. See also *info break* to get a list.

See also:

disable, *tbreak*

Tbreak (temporary breakpoint)

tbreak [*location*] [**if** *condition*]

With a line number argument, set a break there in the current file. With a function name, set a break at first executable line of that function. Without argument, set a breakpoint at current location. If a second argument is *if*, subequent arguments given an expression which must evaluate to true before the breakpoint is honored.

The location line number may be prefixed with a filename or module name and a colon. Files is searched for using *sys.path*, and the *.py* suffix may be omitted in the file name.

Examples:

```
tbreak     # Break where we are current stopped at
tbreak 10  # Break on line 10 of the file we are currently stopped at
tbreak os.path.join # Break in function os.path.join
tbreak os.path:45   # Break on line 45 of os.path
tbreak myfile.py:45 # Break on line 45 of myfile.py
tbreak myfile:45    # Same as above.
```

See also:

break.

Data

Examining data.

Deparse (CPython bytecode deparser)

deparse [options] [.]

Options are:

```
-p | --parent         show parent node
-P | --pretty         show pretty output
-A | --tree | --AST   show abstract syntax tree (AST)
-o | --offset [num]   show deparse of offset NUM
-h | --help           give this help
```

deparse around where the program is currently stopped. If no offset is given, we use the current frame offset. If *-p* is given, include parent information.

If an '.' argument is given, deparse the entire function or main program you are in. The *-P* parameter determines whether to show the prettified as you would find in source code, or in a form that more closely matches a literal reading of the bytecode with hidden (often extraneous) instructions added. In some cases this may even result in invalid Python code.

Output is colorized the same as source listing. Use *set highlight plain* to turn that off.

Examples:

```
deparse              # deparse current location
deparse --parent     # deparse current location enclosing context
deparse .            # deparse current function or main
deparse --offset 6   # deparse starting at offset 6
deparse --offsets    # show all exect deparsing offsets
deparse --AST        # deparse and show AST
```

See also:

disassemble, *list*, and *set highlight*

Disassemble (CPython disassembly)

disassemble [*thing*] [*start-line* [*end-line*]]

With no argument, disassemble the current frame. With an integer start-line, the disassembly is narrowed to show lines starting at that line number or later; with an end-line number, disassembly stops when the next line would be greater than that or the end of the code is hit.

If *start-line* or *end-line is* `.`, +, or −, the current line number is used. If instead it starts with a plus or minus prefix to a number, then the line number is relative to the current frame number.

With a class, method, function, pyc-file, code or string argument disassemble that.

Examples:

```
disassemble    # Possibly lots of stuff dissassembled
disassemble .  # Disassemble lines starting at current stopping point.
disassemble +                 # Same as above
disassemble +0                # Same as above
disassemble os.path           # Disassemble all of os.path
disassemble os.path.normcase  # Disaassemble just method os.path.normcase
disassemble -3  # Disassemble subtracting 3 from the current line number
disassemble +3  # Disassemble adding 3 from the current line number
disassemble 3                 # Disassemble starting from line 3
disassemble 3 10              # Disassemble lines 3 to 10
disassemble myprog.pyc        # Disassemble file myprog.pyc
```

Display (set display expression)

display [*format*] *expression*

Print value of expression *expression* each time the program stops. *format* may be used before *expression* and may be one of */c* for char, */x* for hex, */o* for octal, */f* for float or */s* for string.

For now, display expressions are only evaluated when in the same code as the frame that was in effect when the display expression was set. This is a departure from gdb and we may allow for more flexibility in the future to specify whether this should be the case or not.

With no argument, evaluate and display all currently requested auto-display expressions.

See also:

ref:*undisplay* <*undisplay*> to cancel display requests previously made.

Eval (evaluate Python code)

eval *python-statement*

Run *python-statement* in the context of the current frame.

If no string is given, we run the string from the current source code about to be run. If the command ends *?* (via an alias) and no string is given, the following translations occur:

```
assert = <expr>       => <expr>
{if|elif} <expr> :    => <expr>
while <expr> :        => <expr>
return <expr>         => <expr>
for <var> in <expr> : => <expr>
<var> = <expr>        => <expr>
```

The above is done via regular expression matching. No fancy parsing is done, say, to look to see if *expr* is split across a line or whether var an assignment might have multiple variables on the left-hand side.

Examples:

```
eval 1+2  # 3
eval      # Run current source-code line
eval?     # but strips off leading 'if', 'while', ..
          # from command
```

See also:

set autoeval, *pr*, *pp* and *examine*.

Examine

examine *expr1* [*expr2* ...]

Examine value, type and object attributes of an expression.

In contrast to normal Python expressions, expressions should not have blanks which would cause shlex to see them as different tokens.

Examples:

```
examine x+1   # ok
examine x + 1 # not ok
```

See also:

pr, *pp*, and *whatis*.

Pdef

pdef *obj*

Print the definition header for a callable object *obj*. If the object is a class, print the constructor information.

See also:

pydocX, *pp*

Pp (pretty print expression)

pp *expression*

Pretty-print the value of the expression.

Simple arrays are shown columnized horizontally. Other values are printed via *pprint.pformat()*.

See also:

pr and *examine* for commands which do more in the way of formatting.

Pr (print expression)

pr *expression*

Print the value of the expression. Variables accessible are those of the environment of the selected stack frame, plus globals.

The expression may be preceded with */fmt* where *fmt* is one of the format letters 'c', 'x', 'o', 'f', or 's' for chr, hex, oct, float or str respectively.

If the length output string large, the first part of the value is shown and ... indicates it has been truncated.

See also:

pp and *examine* for commands which do more in the way of formatting; *pydocx*

Pydocx (show pydoc)

pydocx *name ...*

Show pydoc documentation on something. *name* may be the name of a Python keyword, topic, function, module, or package, or a dotted reference to a class or function within a module or module in a package. If *name* contains a '/', it is used as the path to a Python source file to document. If name is *keywords*, *topics*, or *modules*, a listing of these things is displayed.

See also:

whatis, *undisplay*

Undisplay (cancel a display expression)

undisplay *display-number...*

Cancel some expressions to be displayed when program stops. Arguments are the code numbers of the expressions to stop displaying.

No argument cancels all automatic-display expressions and is the same as *delete display*.

See also:

info display to see current list of code numbers. *whatis*

Whatis

whatis *arg*

Prints the information argument which can be a Python expression.

When possible, we give information about:

- type of argument
- doc string for the argument (if a module, class, or function)
- comments around the definition of the argument (module)
- the module it was defined in
- where the argument was defined

We get this most of this information via the *inspect* module.

See also:

pydocx, the `inspect` module.

Files

Specifying and examining files.

Edit

edit *position*

Edit specified file or module. With no argument, edits file containing most recent line listed.

See also:

list

List (show me the code!)

list [*module] [*first* [*num*]]

list *location* [*num*]

List source code.

Without arguments, print lines centered around the current line. If *num* is given that number of lines is shown.

If this is the first *list* command issued since the debugger command loop was entered, then the current line is the current frame. If a subsequent list command was issued with no intervening frame changing, then that is start the line after we last one previously shown.

A *location* is either:

- a number, e.g. 5,
- a function, e.g. *join* or *os.path.join*
- a module, e.g. os or os.path
- a filename, colon, and a number, e.g. *foo.py:5*,
- a module name and a number, e.g,. *os.path:5*.
- a "." for the current line number
- a "-" for the lines before the current linenumber

If the location form is used with a subsequent parameter, the parameter is the starting line number is used. When there two numbers are given, the last number value is treated as a stopping line unless it is less than the start line, in which case it is taken to mean the number of lines to list instead.

Wherever a number is expected, it does not need to be a constant – just something that evaluates to a positive integer.

Examples:

```
list 5            # List starting from line 5
list 4+1          # Same as above.
list foo.py:5     # List starting from line 5 of foo.py
list os.path:5    # List starting from line 5 of os.path
list os.path 5    # Same as above.
list os.path 5 6  # list lines 5 and 6 of os.path
list os.path 5 2  # Same as above, since 2 < 5.
list foo.py:5 2   # List two lines starting from line 5 of foo.py
list os.path.join # List lines around the os.join.path function.
list .            # List lines centered from where we currently are stopped
list -            # List lines previous to those just shown
```

See also:

set listize, or *show listsize* to see or set the number of source-code lines to list.

Info

info [*info-subcommand*]

Get information on the program being debugged.

You can give unique prefix of the name of a subcommand to get information about just that subcommand.

Type *info* for a list of info subcommands and what they do. Type help *info* * for just a list of info subcommands.

Info Args

info args

Show parameters of the current stack frame.

See also:

info locals, *info globals*, *info frame*

Info Break

info breakpoints [*bp-number...*]

Show breakpoints.

See also:

break, *delete*

Info Builtins

info builtins

Show the builtin-functions for the current stack frame.

Info Display

info display

Show the display expression evaluated when the program stops.

See also:

display, *undisplay*

Info Files

info files [*filename* [**all** | **brkpts** | **lines** | **sha1** | **size**]]

Show information about the current file. If no filename is given and the program is running then the current file associated with the current stack entry is used. Sub options which can be shown about a file are:

brkpts Line numbers where there are statement boundaries. These lines can be used in breakpoint commands.

sha1 A SHA1 hash of the source text. This may be useful in comparing source code

size The number of lines in the file.

all All of the above information.

info frame [-v] [*frame-number* | *frame-object*]

Info Frame

Show the detailed information for *frame-number* or the current frame if *frame-number* is not specified. You can also give a frame object instead of a frame number

Specific information includes:

- the frame number (if not an object)
- the source-code line number that this frame is stopped in
- the last instruction executed; -1 if the program are before the first instruction
- a function that tracing this frame or *None*
- Whether the frame is in restricted execution
- Exception type and value if there is one

If *-v* is given we show builtin and global names the frame sees.

See also:

info locals, *info globals*, *info args*

Info Globals

info globals

Show the global variables of the current stack frame.

See also:

info locals, *info args*, *info frame*

Info Line

info line

Show information about the current line

See also:

info program, *info frame*

Info Locals

info locals [*var1 ...*] **info locals ***

Show the local variables of the current stack frame.

See also:

info globals, *info args*, *info frame*

Info Macro

info macro
info macro *
info macro *macro1* [*macro2 ..*]

In the first form a list of the existing macro names are shown in column format.

In the second form, all macro names and their definitions are shown.

In the last form the only definitions of the given macro names is shown.

See also:

show aliases

Info PC

info pc

List the current program counter or bytecode offset, and disassemble the instructions around that.

See also:

info line, *info program*

Info Program

info program

Execution status of the program. Listed are:

- Program name
- Instruction PC
- Reason the program is stopped.

See also:

info line, *info args*, *info frame*

Info Return

info return

Show the value that is to be returned from a function. This command is useful after a running a debugger *finish* command or stepping just after a 'return' statement.

Info Signals

info signals [*signal-name*]

info signals *

Show information about how debugger treats signals to the program. Here are the boolean actions we can take:

- Stop: enter the debugger when the signal is sent to the debugged program
- Print: print that the signal was received
- Stack: show a call stack
- Pass: pass the signal onto the program

If *signal-name* is not given, we the above show information for all signals. If '*' is given we just give a list of signals.

Info Source

info source

Information about the current Python file.

Info Threads

info threads [*thread-name* | *thread-number*] [**terse** | **verbose**]

List all currently-known thread name(s).

If no thread name is given, we list info for all threads. Unless a terse listing, for each thread we give:

- the class, thread name, and status as *Class(Thread-n, status)*
- the top-most call-stack information for that thread.

Generally the top-most calls into the debugger and dispatcher are omitted unless set dbg_trepan is *True*.

If 'verbose' appended to the end of the command, then the entire stack trace is given for each frame. If 'terse' is appended we just list the thread name and thread id.

To get the full stack trace for a specific thread pass in the thread name.

Running

Running, restarting, or stopping the program.

When a program is stopped there are several possibilities for further program execution. You can:

- terminate the program inside the debugger
- restart the program

- continue its execution until it would normally terminate or until a breakpoint is hit
- step exection which is runs for a limited amount of code before stopping

Continue

continue [[*file* :] *lineno* | *function*]

Leave the debugger read-eval print loop and continue execution. Subsequent entry to the debugger however may occur via breakpoints or explicit calls, or exceptions.

If a line position or function is given, a temporary breakpoint is set at that position before continuing.

Examples:

```
continue           # Continue execution
continue 5         # Continue with a one-time breakpoint at line 5
continue basename # Go to os.path.basename if we have basename imported
continue /usr/lib/python2.7/posixpath.py:110 # Possibly the same as
                                             # the above using file
                                             # and line number
```

See also:

step jump, *next*, and *finish* provide other ways to progress execution.

Exit

exit [*exitcode*]

Hard exit of the debugged program.

The program being debugged is exited via *sys.exit()*. If a return code is given, that is the return code passed to *sys.exit()*, the return code that will be passed back to the OS.

See also:

quit and *kill*

Finish (step out)

finish [*level*]

Continue execution until leaving the current function. When *level* is specified, that many frame levels need to be popped. Note that *yield* and exceptions raised my reduce the number of stack frames. Also, if a thread is switched, we stop ignoring levels.

See the *break* command if you want to stop at a particular point in a program.

See also:

step skip, *jump*, *continue*, and *finish* provide other ways to progress

Jump

jump *lineno*

Set the next line that will be executed. The line must be within the stopped or bottom-most execution frame.

See also:

step skip, *next*, *continue*, and *finish* provide other ways to progress

Kill

kill [*signal-number*] [unconditional]

Send this process a POSIX signal ('9' for 'SIGKILL' or 'kill -SIGKILL')

9 is a non-maskable interrupt that terminates the program. If program is threaded it may be expedient to use this command to terminate the program.

However other signals, such as those that allow for the debugged to handle them can be sent.

Giving a negative number is the same as using its positive value.

Examples:

```
kill                # non-interuptable, nonmaskable kill
kill 9              # same as above
kill -9             # same as above
kill!               # same as above, but no confirmation
kill unconditional  # same as above
kill 15             # nicer, maskable TERM signal
kill! 15            # same as above, but no confirmation
```

See also:

quit for less a forceful termination command; *exit* for another way to force termination. *run* and *restart* are ways to restart the debugged program.

Next (step over)

next [**+** | **-**] [*count*]

Step one statement ignoring steps into function calls at this level.

With an integer argument, perform *next* that many times. However if an exception occurs at this level, or we *return*, *yield* or the thread changes, we stop regardless of count.

A suffix of + on the command or an alias to the command forces to move to another line, while a suffix of - does the opposite and disables the requiring a move to a new line. If no suffix is given, the debugger setting 'different-line' determines this behavior.

See also:

skip, *jump*, *continue*, and *finish* provide other ways to progress execution.

Quit

quit [**unconditionally**]

Gently terminate the debugged program.

The program being debugged is aborted via a *DebuggerQuit* exception.

When the debugger from the outside (e.g. via a *trepan* command), the debugged program is contained inside a try block which handles the *DebuggerQuit* exception. However if you called the debugger was started in the middle of a program, there might not be such an exception handler; the debugged program still terminates but generally with a traceback showing that exception.

If the debugged program is threaded, we raise an exception in each of the threads ending with our own. However this might not quit the program.

See also:

kill or *kill* for more forceful termination commands. *run* and *restart* are other ways to restart the debugged program.

Run

run

Soft restart debugger and program via a *DebuggerRestart* exception.

See also:

restart for another way to restart the debugged program. *quit*, or *kill* for termination commands.

Restart

restart

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

See also:

run for another way to restart the debugged program. *quit*, or *kill* for termination commands.

Skip

skip [*count*]

Set the next line that will be executed. The line must be within the stopped or bottom-most execution frame.

See also:

step jump, *continue*, and *finish* provide other ways to progress execution.

Step (step into)

step [**+** | **-** | **<** | **>** | **!**] [*event...*] [*count*]

Execute the current line, stopping at the next event.

With an integer argument, step that many times.

event is list of an event name which is one of: *call*, *return*, *line*, *exception c-call*, *c-return* or *c-exception*. If specified, only those stepping events will be considered. If no list of event names is given, then any event triggers a stop when the count is 0.

There is however another way to specify a *single* event, by suffixing one of the symbols <, >, or *!* after the command or on an alias of that. A suffix of + on a command or an alias forces a move to another line, while a suffix of - disables this requirement. A suffix of > will continue until the next call. (*finish* will run run until the return for that call.)

If no suffix is given, the debugger setting *different-line* determines this behavior.

Examples:

```
step        # step 1 event, *any* event
step 1      # same as above
step 5/5+0  # same as above
step line   # step only line events
step call   # step only call events
step>       # same as above
step call line # Step line *and* call events
```

See also:

next command. *skip*, *jump* (there's no *hop* yet), *continue*, and *finish* provide other ways to progress execution.

set [*set-subcommand*]

Modifies parts of the debugger environment.

You can give unique prefix of the name of a subcommand to get information about just that subcommand.

Type *set* for a list of set subcommands and what they do. Type help set * for just the list of set subcommands.

All of the "set" commands have a corresponding *show* command.

Set

Modifies parts of the debugger environment. You can see these environment settings with the *show* command.

Set Auto Eval

set autoeval [on | off]

Evaluate unrecognized debugger commands.

Often inside the debugger, one would like to be able to run arbitrary Python commands without having to preface Python expressions with `print` or `eval`. Setting *autoeval* on will cause unrecognized debugger commands to be *eval*'d as a Python expression.

Note that if this is set, on error the message shown on type a bad debugger command changes from:

```
Undefined command: "fdafds". Try "help".
```

to something more Python-eval-specific such as:

```
NameError: name 'fdafds' is not defined
```

One other thing that trips people up is when setting autoeval is that there are some short debugger commands that sometimes one wants to use as a variable, such as in an assignment statement. For example:

```
s = 5
```

which produces when *autoeval* is on:

```
Command 'step' can take at most 1 argument(s); got 2.
```

because by default, `s` is an alias for the debugger `step` command. It is possible to remove that alias if this causes constant problem.

See also:

show autoeval

Set Auto List

set autolist [on | off]

Run the *list* command every time you stop in the debugger.

With this, you will get output like:

```
-> 1 from subprocess import Popen, PIPE
(trepan2) next
(/users/fbicknel/Projects/disk_setup/sqlplus.py:2): <module>
** 2 import os
  1     from subprocess import Popen, PIPE
  2  -> import os
  3     import re
  4
  5     class SqlPlusExecutor(object):
  6         def __init__(self, connection_string='/ as sysdba', sid=None):
  7             self.__connection_string = connection_string
  8             self.session = None
  9             self.stdout = None
 10             self.stderr = None
(trepan2) next
(/users/fbicknel/Projects/disk_setup/sqlplus.py:3): <module>
** 3 import re
  1     from subprocess import Popen, PIPE
  2     import os
  3  -> import re
  4
  5     class SqlPlusExecutor(object):
  6         def __init__(self, connection_string='/ as sysdba', sid=None):
  7             self.__connection_string = connection_string
  8             self.session = None
  9             self.stdout = None
 10             self.stderr = None
(trepan2)
```

You may also want to put this this in your debugger startup file. See Startup Profile

See also:

show autolist

Set Autopython

set autopython [**on** | **off**]

Go into a Python shell on debugger entry.

See also:

python

Set Basename

set basename [**on** | **off**]

Set short filenames in debugger output.

Setting this causes the debugger output to give just the basename for filenames. This is useful in debugger testing or possibly showing examples where you don't want to hide specific filesystem and installation information.

See also:

show basename

Set Cmdtrace

set cmdtrace [**on** | **off**]

Set echoing lines read from debugger command files

See also:

show cmdtrace

Set Confirm

set confirm [**on** | **off**]

Set confirmation of potentially dangerous operations.

Some operations are a bit disruptive like terminating the program. To guard against running this accidentally, by default we ask for confirmation. Commands can also be exempted from confirmation by suffixing them with an exclamation mark (!).

See also:

show confirm

Set Dbg_trepan

set dbg_trepan [**on** | **off**]

Set the ability to debug the debugger.

Setting this allows visibility and access to some of the debugger's internals. Specifically variable "frame" contains the current frame and variable "debugger" contains the top-level debugger object.

See also:

show dbg_trepan

Set Different

set different [**on** | **off**]

Set consecutive stops must be on different file/line positions.

By default, the debugger traces all events possible including line, exceptions, call and return events. Just this alone may mean that for any given source line several consecutive stops at a given line may occur. Independent of this, Python allows one to put several commands in a single source line of code. When a programmer does this, it might be because the programmer thinks of the line as one unit.

One of the challenges of debugging is getting the granualarity of stepping comfortable. Because of the above, stepping all events can often be too fine-grained and annoying. By setting different on you can set a more coarse-level of stepping which often still is small enough that you won't miss anything important.

Note that the *step* and *next* debugger commands have '+' and '-' suffixes if you wan to override this setting on a per-command basis.

See also:

set trace to change what events you want to filter. *show trace*.

Set Events

set events [*event* ...]

Sets the events that the debugger will stop on. Event names are:

- *c_call*
- *c_exception*
- *c_return*
- *call*
- *exception*
- *line*
- *return*

all can be used as an abbreviation for listing all event names.

Changing trace event filters works independently of turning on or off tracing-event printing.

Examples:

```
set events line        # Set trace filter for line events only.
set events call return # Trace calls and returns only
set events all         # Set trace filter to all events.
```

See also:

set trace, *show trace*, and *show events*

Set Flush

set flush [**on** | **off**]

Set flushing output after each write

See also:

show flush

Set Highlight

set highlight [**reset**] {**plain** | **light** | **dark** | **off**}

Set whether we use terminal highlighting for ANSI 8-color terminals. Permissible values are:

plain no terminal highlighting

off same as plain

light terminal background is light (the default)

dark terminal background is dark

If the first argument is *reset*, we clear any existing color formatting and recolor all source code output.

A related setting is *style* which sets the Pygments style for terminal that support, 256 colors. But even here, it is useful to set the highlight to tell the debugger for bold and emphasized text what values to use.

Examples:

```
set highlight off   # no highlight
set highlight plain # same as above
set highlight       # same as above
set highlight dark  # terminal has dark background
set highlight light # terminal has light background
set highlight reset light # clear source-code cache and
                          # set for light background
set highlight reset # clear source-code cache
```

See also:

show highlight and *set style*

Set Listsize

set listsize *number-of-lines*

Set the number lines printed in a *list* command by default

See also:

show listsize

Set Maxstring

set maxstring *number*

Set the number of characters allowed in showing string values

See also:

show maxstring

Set Skip

Set stopping before *def* or *class* (function or class) statements.

Classes may have many methods and stand-alone programs may have many functions. Often there isn't much value to stopping before defining a new function or class into Python's symbol table. (More to the point, it can be an annoyance.) However if you do want this, for example perhaps you want to debug methods is over-writing one another, then set this off.

See also:

show skip

Set Style

set style [*pygments-style*]

Set the pygments style in to use in formatting text for a 256-color terminal. Note: if your terminal doesn't support 256 colors, you may be better off using *–highlight=plain* or *–highlight=dark* instead. To turn off styles use *set style none*.

To list the available pygments styles inside the debugger, omit the style name.

Examples:

```
set style            # give a list of the style names
set style colorful   # Pygments 'colorful' style
set style fdasfda    # Probably display available styles
set style none       # Turn off style, still use highlight though
```

See also:

show style and *set highlight*

Set Substitute

set substitute *from-name to-path*

Add a substitution rule replacing *from-name* into *to-path* in source file names. If a substitution rule was previously set for *from-name*, the old rule is replaced by the new one.

Spaces in "filenames" like <*frozen importlib._bootstrap*> messes up our normal shell tokenization, so we have added a hack to ignore <*frozen .. >*.

So, for frozen files like <*frozen importlib._bootstrap*>, use *importlib._bootstrap*

Examples:

```
set substitute importlib._bootstrap /usr/lib/python3.4/importlib/_bootstrap.py
set substitute ./gcd.py /tmp/gcd.py
```

Set Trace

set trace [**on** | **off**]

Set event tracing.

See also:

set events, *set trace*, and *show trace*

Set Width

set width *number*

Set the number of characters the debugger thinks are in a line.

See also:

show width

Stack

Examining the call stack.

The call stack is made up of stack frames. The debugger assigns numbers to stack frames counting from zero for the innermost (currently executing) frame.

At any time the debugger identifies one frame as the "selected" frame. Variable lookups are done with respect to the selected frame. When the program being debugged stops, the debugger selects the innermost frame. The commands below can be used to select other frames by number or address.

Backtrace (show call-stack)

backtrace [*count*]

Print a stack trace, with the most recent frame at the top. With a positive number, print at most many entries. With a negative number print the top entries minus that number.

An arrow indicates the 'current frame'. The current frame determines the context used for many debugger commands such as expression evaluation or source-line listing.

Examples:

```
backtrace    # Print a full stack trace
backtrace 2  # Print only the top two entries
backtrace -1 # Print a stack trace except the initial (least recent) call.
```

Frame (absolute frame positioning)

frame [*thread-Name*|**thread-number*] [*frame-number*]

Change the current frame to frame *frame-number* if specified, or the current frame, 0, if no frame number specified.

If a thread name or thread number is given, change the current frame to a frame in that thread. Dot (.) can be used to indicate the name of the current frame the debugger is stopped in.

A negative number indicates the position from the other or least-recently-entered end. So *frame -1* moves to the oldest frame, and *frame 0* moves to the newest frame. Any variable or expression that evaluates to a number can be used as a position, however due to parsing limitations, the position expression has to be seen as a single blank-delimited parameter. That is, the expression *(5*3)-1* is okay while *(5 * 3) - 1)* isn't.

Examples:

```
frame     # Set current frame at the current stopping point
frame 0   # Same as above
frame 5-5 # Same as above. Note: no spaces allowed in expression 5-5
frame .   # Same as above. "current thread" is explicit.
frame . 0 # Same as above.
frame 1   # Move to frame 1. Same as: frame 0; up
frame -1  # The least-recent frame
frame MainThread 0 # Switch to frame 0 of thread MainThread
frame MainThread   # Same as above
frame -2434343 0   # Use a thread number instead of name
```

See also:

down, *up*, *backtrace*, and *info threads*.

Up (relative frame motion towards a less-recent frame)

up [*count*]

Move the current frame up in the stack trace (to an older frame). 0 is the most recent frame. If no count is given, move up 1.

See also:

down and *frame*.

Down (relative frame motion towards a more-recent frame)

down [*count*]

Move the current frame down in the stack trace (to a newer frame). 0 is the most recent frame. If no count is given, move down 1.

See also:

up and *frame*.

show [*subcommand*]

A command for showing things about the debugger. You can give unique prefix of the name of a subcommand to get information about just that subcommand.

Type *show* for a list of show subcommands and what they do. Type help *show* * for just a list of show subcommands. Many of the "show" commands have a corresponding *set* command.

Show

Show Aliases (show debugger command aliases)

show aliases [*alias* ...| *****]

Show command aliases. If parameters are given a list of all aliases and the command they run are printed. Alternatively one can list specific alias names for the commands those specific aliases are attached to. If instead of an alias "*" appears anywhere as an alias then just a list of aliases is printed, not what commands they are attached to.

See also:

alias

Show Args (show arguments when program is started)

show args

Show the argument list to give debugged program when it is started

Show Autoeval

show autoeval

Show Python evaluation of unrecognized debugger commands.

See also:

set autoeval

Show Autolist

show autolist

Run a debugger ref:*list* <*list*> command automatically on debugger entry.

See also:

set autolist

Show Autopython

show autopython

Show whether we go into a Python shell when automatically when the debugger is entered.

See also:

set autopython

Show Basename

show basename

Show Python evaluation of unrecognized debugger commands.

See also:

set basename

Show Cmdtrace

show cmdtrace

Show debugger commands before running them

See also:

set cmdtrace

Show Confirm

show confirm

Show confirmation of potentially dangerous operations

See also:

show confirm

Show Dbg_trepan

Show debugging the debugger

See also:

set dbg_trepan

Show Different

Show consecutive stops on different file/line positions

See also:

set different

Show Events

show events

Show the kinds of events the debugger will stop on.

See also:

set events

Show Highlight

show highlight

Show whether we use terminal highlighting.

See also:

set highlight

Show Listsize

show listsize

Show the number lines printed in a *list* command by default

See also:

set listsize

Show Maxstring

show maxstring

Show maximum string length to use in string-oriented output

See also:

set maxstring

Show Skip

show skip

Show whether debugger steps over lines which define functions and classes

See also:

set skip

Show Style

show style *pygments-style*

Show the pygments style used in formatting 256-color terminal text.

See also:

set style and *show highlight*

Show Trace

show trace

Show event tracing.

See also:

set trace, *show events*

Show Width

show width

Show the number of characters the debugger thinks are in a line.

See also:

set width

Support

Alias (add debugger command alias)

alias *alias-name debugger-command*

Add alias *alias-name* for a debugger command *debugger-comand*.

Add an alias when you want to use a command abbreviation for a command that would otherwise be ambigous. For example, by default we make `s` be an alias of `step` to force it to be used. Without the alias, `s` might be `step`, `show`, or `set` among others

Example:

```
alias cat list   # "cat myprog.py" is the same as "list myprog.py"
alias s   step   # "s" is now an alias for "step".
                 # The above example is done by default.
```

See also:

unalias and *show alias*.

BPython (go into a bpython shell)

bpython [*-d*]

Note: this command is available only if bpython is installed

Run Python as a command subshell. The *sys.ps1* prompt will be set to `trepan2 >>>`.

If *-d* is passed, you can access debugger state via local variable *debugger*.

To issue a debugger command use function *dbgr()*. For example:

```
dbgr('info program')
```

See also:

python, and *ipython*.

Debug (recursively debug an expression)

debug *python-expression*

Enter a nested debugger that steps through the *python-expression* argument which is an arbitrary expression to be executed the current environment.

Help (Won't you please help me if you can)

help [*command* [*subcommand*]|*expression*]

Without argument, print the list of available debugger commands.

When an argument is given, it is first checked to see if it is command name.

If the argument is an expression or object name, you get the same help that you would get inside a Python shell running the built-in *help()* command.

If the environment variable *$PAGER* is defined, the file is piped through that command. You'll notice this only for long help output.

Some commands like *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.

See also:

examine and *whatis*.

IPython (go into an IPython shell)

ipython [*-d*]

Note: this command is available only if ipython is installed

Run Python as a command subshell. The *sys.ps1* prompt will be set to `trepan2 >>>`.

If *-d* is passed, you can access debugger state via local variable *debugger*.

To issue a debugger command use function *dbgr()*. For example:

```
dbgr('info program')
```

See also:

python, and *bpython*.

Macro (add a debugger macro)

macro *macro-name lambda-object*

Define *macro-name* as a debugger macro. Debugger macros get a list of arguments which you supply without parenthesis or commas. See below for an example.

The macro (really a Python lambda) should return either a String or an List of Strings. The string in both cases is a debugger command. Each string gets tokenized by a simple split() . Note that macro processing is done right after splitting on `;;`. As a result, if the macro returns a string containing `;;` this will not be interpreted as separating debugger commands.

If a list of strings is returned, then the first string is shifted from the list and executed. The remaining strings are pushed onto the command queue. In contrast to the first string, subsequent strings can contain other macros. `;;` in those strings will be split into separate commands.

Here is an trivial example. The below creates a macro called `l=` which is the same thing as `list .`:

```
macro l= lambda: 'list .'
```

A simple text to text substitution of one command was all that was needed here. But usually you will want to run several commands. So those have to be wrapped up into a list.

The below creates a macro called `fin+` which issues two commands `finish` followed by `step`:

```
macro fin+ lambda: ['finish','step']
```

If you wanted to parameterize the argument of the `finish` command you could do that this way:

```
macro fin+ lambda levels: ['finish %s' % levels ,'step']
```

Invoking with:

```
fin+ 3
```

would expand to: `['finish 3', 'step']`

If you were to add another parameter for `step`, the note that the invocation might be:

```
fin+ 3 2
```

rather than `fin+(3,2)` or `fin+ 3, 2`.

See also:

alias, and *info macro*.

Python (go into a Python shell)

python [*-d*]

Run Python as a command subshell. The *sys.ps1* prompt will be set to `trepan2 >>>`.

If *-d* is passed, you can access debugger state via local variable *debugger*.

To issue a debugger command use function *dbgr()*. For example:

```
dbgr('info program')
```

See also:

ipython, and *bpython*.

Source (Read and run debugger commands from a file)

source [**-v**][**-Y****|-N**][**-c**] *file*

Read debugger commands from a file named *file*. Optional *-v* switch (before the filename) causes each command in *file* to be echoed as it is executed. Option *-Y* sets the default value in any confirmation command to be "yes" and *-N* sets the default value to "no".

Note that the command startup file *.trepanc* is read automatically via a *source* command the debugger is started.

An error in any command terminates execution of the command file unless option *-c* is given.

Unalias (remove debugger command alias)

unalias *alias-name*

Remove alias *alias-name*.

See also:

alias.

CHAPTER 7

Manual Pages

trepan2 (Python2 debugger)

Synopsis

trepan2 [*debugger-options*] [–] [*python-script* [*script-options* ...]]

Description

Run the Python2 trepan debugger from the outset.

Options

-h, –help Show the help message and exit

-x, –trace Show lines before executing them.

-F, –fntrace Show functions before executing them.

–basename Filenames strip off basename, (e.g. for regression tests)

–client Connect to an existing debugger process started with the *–server* option

-x *FILE*, –command= *FILE* Execute commands from *FILE*

–cd= *DIR* Change current directory to *DIR*

-confirm Confirm potentially dangerous operations

–dbg_trepan Debug the debugger

–different Consecutive stops should have different positions

-e *EXECUTE-CMDS*, –exec= *EXECUTE-CMDS* list of debugger commands to execute. Separate the commands with *;;*

–highlight={light|dark|plain} Use syntax and terminal highlight output. “plain” is no highlight

–private Don’t register this as a global debugger

–post-mortem Enter debugger on an uncaught (fatal) exception

-n, –nx Don’t execute commands found in any initialization files

-o *FILE*, –output= *FILE* Write debugger’s output (stdout) to *FILE*

-p *PORT*,–port= *PORT* Use TCP port number *NUMBER* for out-of-process connections.

–server Out-of-process server connection mode

–sigcheck Set to watch for signal handler changes

-t *TARGET*, –target= *TARGET* Specify a target to connect to. Arguments should be of form, *protocol*:*address*

–from_ipython Called from inside ipython

– Use this to separate debugger options from any options your Python script has

See also

trepan3k (Python3 debugger) (1), *trepan2c (Python2 client to connect to remote trepan session)* (1), *trepan3kc (Python3 client to connect to remote trepan session)*

Full Documentation is available at http://python2-trepan.readthedocs.org

trepan2c (Python2 client to connect to remote trepan session)

Synopsis

trepan2c [*debugger-options*] [–] [*python-script* [*script-options* ...]]

Description

Run the Python2 trepan debugger client to connect to an existing out-of-process Python *trepan* session

Options

-h, –help Show the help message and exit

-x, –trace Show lines before executing them.

-H *IP-OR-HOST*, –host= *IP-OR-HOST* connect to *IP* or *HOST*

-P *NUMBER, –port= *NUMBER* Use TCP port number NUMBER for out-of-process connections.

–pid=*NUMBER* Use PID to get FIFO names for out-of-process connections.

See also

trepan3k (Python3 debugger) (1), *trepan2c (Python2 client to connect to remote trepan session)* (1), *trepan3kc (Python3 client to connect to remote trepan session)*

Full Documentation is available at http://python2-trepan.readthedocs.org

trepan3k (Python3 debugger)

Synopsis

trepan3k [*debugger-options*] [–] [*python-script* [*script-options* ...]]

Description

Run the Python3 trepan debugger from the outset.

Options

-h, –help Show the help message and exit

-x, –trace Show lines before executing them.

-F, –fntrace Show functions before executing them.

–basename Filenames strip off basename, (e.g. for regression tests)

–client Connect to an existing debugger process started with the *–server* option

-x *FILE*, –command= *FILE* Execute commands from *FILE*

–cd= *DIR* Change current directory to *DIR*

-confirm Confirm potentially dangerous operations

–dbg_trepan Debug the debugger

–different Consecutive stops should have different positions

-e *EXECUTE-CMDS*, –exec= *EXECUTE-CMDS* list of debugger commands to execute. Separate the commands with *;;*

–highlight={light|dark|plain} Use syntax and terminal highlight output. "plain" is no highlight

–private Don't register this as a global debugger

–post-mortem Enter debugger on an uncaught (fatal) exception

-n, –nx Don't execute commands found in any initialization files

-o *FILE*, –output= *FILE* Write debugger's output (stdout) to *FILE*

-p *PORT*,–port= *PORT* Use TCP port number *NUMBER* for out-of-process connections.

–server Out-of-process server connection mode

–sigcheck Set to watch for signal handler changes

-t *TARGET*, –target= *TARGET* Specify a target to connect to. Arguments should be of form, *protocol*:*address*

–from_ipython Called from inside ipython

– Use this to separate debugger options from any options your Python script has

See also

trepan2 (Python2 debugger) (1), *trepan3kc (Python3 client to connect to remote trepan session)* (1), *trepan2c (Python2 client to connect to remote trepan session)*

Full Documentation is available at http://python2-trepan.readthedocs.org

trepan3kc (Python3 client to connect to remote trepan session)

Synopsis

trepan3kc [*debugger-options*] [–] [*python-script* [*script-options* ...]]

Description

Run the Python3 trepan debugger client to connect to an existing out-of-process Python *trepan* session

Options

-h, –help Show the help message and exit

-x, –trace Show lines before executing them.

-H *IP-OR-HOST*, –host= *IP-OR-HOST* connect to *IP* or *HOST*

-P *NUMBER*, *–port= *NUMBER* Use TCP port number NUMBER for out-of-process connections.

–pid=*NUMBER* Use PID to get FIFO names for out-of-process connections.

See also

trepan3k (Python3 debugger) (1), *trepan2 (Python2 debugger)* (1), *trepan2c (Python2 client to connect to remote trepan session)*

Full Documentation is available at http://python2-trepan.readthedocs.org