27.1. bdb — Debugger framework¶

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

The following exception is defined:

exception bdb.BdbQuit¶
Exception raised by the Bdb class for quitting the debugger.

The bdb module also defines two classes:

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

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

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

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

Breakpoint instances have the following methods:

deleteMe()¶
Delete the breakpoint from the list associated to a file/line. If it is the last breakpoint in that position, it also deletes the entry for the file/line.

enable()¶
Mark the breakpoint as enabled.

disable()¶
Mark the breakpoint as disabled.

pprint([*out*])¶

Print all the information about the breakpoint:

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

class bdb.Bdb(*skip=None*)¶

The Bdb class acts as a generic Python debugger base class.

This class takes care of the details of the trace facility; a derived class should implement user interaction. The standard debugger class (pdb.Pdb) is an example.

The *skip* argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns. Whether a frame is considered to originate in a certain module is determined by the __name__ in the frame globals.

New in version 2.7: The *skip* argument.

The following methods of Bdb normally don't need to be overridden.

canonic(*filename*)¶
Auxiliary method for getting a filename in a canonical form, that is, as a case-normalized (on case-insensitive filesystems) absolute path, stripped of surrounding angle brackets.

reset()¶
Set the botframe, stopframe, returnframe and quitting attributes with values ready to start debugging.

`trace_dispatch`(*frame*, *event*, *arg*)¶

This function is installed as the trace function of debugged frames. Its return value is the new trace function (in most cases, that is, itself).

The default implementation decides how to dispatch a frame, depending on the type of event (passed as a string) that is about to be executed. *event* can be one of the following:

- `"line"`: A new line of code is going to be executed.
- `"call"`: A function is about to be called, or another code block entered.
- `"return"`: A function or other code block is about to return.
- `"exception"`: An exception has occurred.
- `"c_call"`: A C function is about to be called.
- `"c_return"`: A C function has returned.
- `"c_exception"`: A C function has thrown an exception.

For the Python events, specialized functions (see below) are called. For the C events, no action is taken.

The *arg* parameter depends on the previous event.

See the documentation for `sys.settrace()` for more information on the trace function. For more information on code and frame objects, refer to *The standard type hierarchy*.

`dispatch_line`(*frame*)¶
If the debugger should stop on the current line, invoke the `user_line()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the `Bdb.quitting` flag is set (which can be set from `user_line()`). Return a reference to the `trace_dispatch()` method for further tracing in that scope.

`dispatch_call`(*frame*, *arg*)¶
If the debugger should stop on this function call, invoke the `user_call()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the `Bdb.quitting` flag is set (which can be set from `user_call()`). Return a reference to the `trace_dispatch()` method for further tracing in that scope.

`dispatch_return`(*frame*, *arg*)¶
If the debugger should stop on this function return, invoke the `user_return()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the `Bdb.quitting` flag is set (which can be set from `user_return()`). Return a reference to the `trace_dispatch()` method for further tracing in that scope.

`dispatch_exception`(*frame*, *arg*)¶
If the debugger should stop at this exception, invokes the `user_exception()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the `Bdb.quitting` flag is set (which can be set from `user_exception()`). Return a reference to the `trace_dispatch()` method for further tracing in that scope.

Normally derived classes don't override the following methods, but they may if they want to redefine the definition of stopping and breakpoints.

`stop_here`(*frame*)¶
This method checks if the *frame* is somewhere below `botframe` in the call stack. `botframe` is the frame in which debugging started.

`break_here`(*frame*)¶
This method checks if there is a breakpoint in the filename and line belonging to *frame* or, at least, in the current function. If the breakpoint is a temporary one, this method deletes it.

`break_anywhere`(*frame*)¶
This method checks if there is a breakpoint in the filename of the current frame.

Derived classes should override these methods to gain control over debugger operation.

`user_call`(*frame*, *argument_list*)¶
This method is called from `dispatch_call()` when there is the possibility that a break might be necessary anywhere inside the called function.

`user_line`(*frame*)¶
This method is called from `dispatch_line()` when either `stop_here()` or `break_here()` yields True.

`user_return`(*frame*, *return_value*)¶
This method is called from `dispatch_return()` when `stop_here()` yields True.

`user_exception`(*frame*, *exc_info*)¶
This method is called from `dispatch_exception()` when `stop_here()` yields True.

`do_clear`(*arg*)¶

Handle how a breakpoint must be removed when it is a temporary one.

This method must be implemented by derived classes.

Derived classes and clients can call the following methods to affect the stepping state.

`set_step`()¶
Stop after one line of code.

`set_next`(*frame*)¶
Stop on the next line in or below the given frame.

`set_return`(*frame*)¶
Stop when returning from the given frame.

`set_until`(*frame*)¶
Stop when the line with the line no greater than the current one is reached or when returning from current frame

`set_trace`([*frame*])¶
Start debugging from *frame*. If *frame* is not specified, debugging starts from caller's frame.

`set_continue`()¶
Stop only at breakpoints or when finished. If there are no breakpoints, set the system trace function to None.

`set_quit`()¶
Set the `quitting` attribute to True. This raises `BdbQuit` in the next call to one of the `dispatch_*()` methods.

Derived classes and clients can call the following methods to manipulate breakpoints. These methods return a string containing an error message if something went wrong, or `None` if all is well.

`set_break`(*filename*, *lineno*[, *temporary=0*[, *cond*[, *funcname*]]])¶
Set a new breakpoint. If the *lineno* line doesn't exist for the *filename* passed as argument, return an error message. The *filename* should be in canonical form, as described in the `canonic()` method.

`clear_break`(*filename*, *lineno*)¶
Delete the breakpoints in *filename* and *lineno*. If none were set, an error message is returned.

`clear_bpbynumber`(*arg*)¶
Delete the breakpoint which has the index *arg* in the `Breakpoint.bpbynumber`. If *arg* is not numeric or out of range, return an error message.

`clear_all_file_breaks`(*filename*)¶
Delete all breakpoints in *filename*. If none were set, an error message is returned.

`clear_all_breaks`()¶
Delete all existing breakpoints.

`get_break`(*filename*, *lineno*)¶
Check if there is a breakpoint for *lineno* of *filename*.

`get_breaks`(*filename*, *lineno*)¶
Return all breakpoints for *lineno* in *filename*, or an empty list if none are set.

`get_file_breaks`(*filename*)¶
Return all breakpoints in *filename*, or an empty list if none are set.

`get_all_breaks`()¶
Return all breakpoints that are set.

Derived classes and clients can call the following methods to get a data structure representing a stack trace.

`get_stack`(*f*, *t*)¶
Get a list of records for a frame and all higher (calling) and lower frames, and the size of the higher part.

`format_stack_entry`(*frame_lineno*[, *lprefix=': '*])¶

Return a string with information about a stack entry, identified by a `(frame, lineno)` tuple:

- The canonical form of the filename which contains the frame.
- The function name, or `"<lambda>"`.
- The input arguments.
- The return value.
- The line of code (if it exists).

The following two methods can be called by clients to use a debugger to debug a *statement*, given as a string.

`run`(*cmd*[, *globals*[, *locals*]])¶
Debug a statement executed via the `exec` statement. *globals* defaults to `__main__.__dict__`, *locals* defaults to *globals*.

`runeval`(*expr*[, *globals*[, *locals*]])¶
Debug an expression executed via the `eval()` function. *globals* and *locals* have the same meaning as in `run()`.

`runctx`(*cmd*, *globals*, *locals*)¶
For backwards compatibility. Calls the `run()` method.

`runcall`(*func*, **args*, ***kwds*)¶
Debug a single function call, and return its result.

Finally, the module defines the following functions:

`bdb.checkfuncname`(*b*, *frame*)¶

Check whether we should break here, depending on the way the breakpoint *b* was set.

If it was set via line number, it checks if `b.line` is the same as the one in the frame also passed as argument. If the breakpoint was set via function name, we have to check we are in the right frame (the right function) and if we are in its first executable line.

`bdb.effective`(*file*, *line*, *frame*)¶

Determine if there is an effective (active) breakpoint at this line of code. Return breakpoint number or 0 if none.

Called only if we know there is a breakpoint at this location. Returns the breakpoint that was triggered and a flag that indicates if it is ok to delete a temporary breakpoint.

`bdb.set_trace`()¶
Starts debugging with a `Bdb` instance from caller's frame.

Previous topic

27. Debugging and Profiling

Next topic

27.2. pdb — The Python Debugger

This Page

- Show Source

Navigation

- index
- modules |
- next |
- previous |
- Python v2.6.4 documentation »
- The Python Standard Library »
- 27. Debugging and Profiling »