pdb — Interactive Debugger

Purpose: Python's Interactive Debugger

pdb implements an interactive debugging environment for Python programs. It includes features to pause a program, look at the values of variables, and watch program execution step-by-step, so you can understand what the program actually does and find bugs in the logic.

Starting the Debugger

The first step to using pdb is causing the interpreter to enter the debugger at the right time. There are a few different ways to do that, depending on the starting conditions and what is being debugged.

From the Command Line

The most straightforward way to use the debugger is to run it from the command line, giving it the program as input so it knows what to run.

```
# pdb_script.py

#!/usr/bin/env python3
# encoding: utf-8
#
# Copyright (c) 2010 Doug Hellmann.  All rights reserved.
#


class MyObj:

    def __init__(self, num_loops):
        self.count = num_loops

    def go(self):
        for i in range(self.count):
            print(i)
        return

if __name__ == '__main__':
    MyObj(5).go()
```

Running the debugger from the command line causes it to load the source file and stop execution on the first statement it finds. In this case, it stops before evaluating the definition of the class `MyObj` on line 8.

```
$ python3 -m pdb pdb_script.py

> .../pdb_script.py(8)<module>()
-> class MyObj(object):
(Pdb)
```

Note

Normally pdb includes the full path to each module in the output when printing a filename. In order to maintain clear examples, the path in the sample output in this section has been replaced with ellipsis (`...`).

Within the Interpreter

Many Python developers work with the interactive interpreter while developing early versions of modules because it lets them experiment more iteratively without the save/run/repeat cycle needed when creating standalone scripts. To run the debugger from within an interactive interpreter, use `run()` or `runeval()`.

```
$ python3

Python 3.5.1 (v3.5.1:37a07cee5969, Dec  5 2015, 21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import pdb_script
>>> import pdb
>>> pdb.run('pdb_script.MyObj(5).go()')
> <string>(1)<module>()
(Pdb)
```

The argument to `run()` is a string expression that can be evaluated by the Python interpreter. The debugger will parse it, then pause execution just before the first expression evaluates. The debugger commands described here can be used to navigate and control the execution.

From Within a Program

Both of the previous examples start the debugger at the beginning of a program. For a long-running process where the problem appears much later in the program execution, it will be more convenient to start the debugger from inside the program using `set_trace()`.

```
# pdb_set_trace.py
```

```
1  #!/usr/bin/env python3
2  # encoding: utf-8
3  #
4  # Copyright (c) 2010 Doug Hellmann.  All rights reserved.
5  #
6
7  import pdb
8
9
10 class MyObj:
11
12     def __init__(self, num_loops):
13         self.count = num_loops
14
15     def go(self):
16         for i in range(self.count):
17             pdb.set_trace()
18             print(i)
19         return
20
21 if __name__ == '__main__':
22     MyObj(5).go()
```

Line 17 of the sample script triggers the debugger at that point in execution, pausing it on line 18.

```
$ python3 ./pdb_set_trace.py

> .../pdb_set_trace.py(18)go()
-> print(i)
(Pdb)
```

`set_trace()` is just a Python function, so it can be called at any point in a program. This makes it possible to enter the debugger based on conditions inside the program, including from an exception handler or via a specific branch of a control statement.

After a Failure

Debugging a failure after a program terminates is called *post-mortem* debugging. pdb supports post-mortem debugging through the `pm()` and `post_mortem()` functions.

```
# pdb_post_mortem.py
```

```
1  #!/usr/bin/env python3
2  # encoding: utf-8
3  #
```

```
4  # Copyright (c) 2010 Doug Hellmann.  All rights reserved.
5  #
6
7
8  class MyObj:
9
10     def __init__(self, num_loops):
11         self.count = num_loops
12
13     def go(self):
14         for i in range(self.num_loops):
15             print(i)
16         return
```

Here the incorrect attribute name on line 14 triggers an `AttributeError` exception, causing execution to stop. `pm()` looks for the active traceback and starts the debugger at the point in the call stack where the exception occurred.

```
$ python3
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  5 2015, 21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from pdb_post_mortem import MyObj
>>> MyObj(5).go()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../pdb_post_mortem.py", line 14, in go
    for i in range(self.num_loops):
AttributeError: 'MyObj' object has no attribute 'num_loops'
>>> import pdb
>>> pdb.pm()
> .../pdb/pdb_post_mortem.py(14)go()
-> for i in range(self.num_loops):
(Pdb)
```

Controlling the Debugger

The interface for the debugger is a small command language that lets you move around the call stack, examine and change the values of variables, and control how the debugger executes the program. The interactive debugger uses `readline` to accept commands, and supports tab completion for commands, filenames, and function names. Entering a blank line re-runs the previous command again, unless it was a `list` operation.

Navigating the Execution Stack

At any point while the debugger is running use `where` (abbreviated `w`) to find out exactly what line is being executed and where on the call stack the program is. In this case, the module `pdb_set_trace.py` line 18 in the `go()` method.

```
$ python3 pdb_set_trace.py
> .../pdb_set_trace.py(18)go()
-> print(i)
(Pdb) where
  .../pdb_set_trace.py(22)<module>()
-> MyObj(5).go()
> .../pdb_set_trace.py(18)go()
-> print(i)
(Pdb)
```

To add more context around the current location, use `list` (`l`).

```
(Pdb) l
 13          self.count = num_loops
 14
 15      def go(self):
 16          for i in range(self.count):
 17              pdb.set_trace()
 18  ->          print(i)
 19          return
 20
 21  if __name__ == '__main__':
```

```
 22      MyObj(5).go()
[EOF]
(Pdb)
```

The default is to list 11 lines around the current line (five before and five after). Using `list` with a single numerical argument lists 11 lines around that line instead of the current line.

```
(Pdb) list 14
  9
 10  class MyObj(object):
 11
 12      def __init__(self, num_loops):
 13          self.count = num_loops
 14
 15      def go(self):
 16          for i in range(self.count):
 17              pdb.set_trace()
 18  ->          print(i)
 19          return
```

If `list` receives two arguments, it interprets them as the first and last lines to include in its output.

```
(Pdb) list 7, 19
  7  import pdb
  8
  9
 10  class MyObj(object):
 11
 12      def __init__(self, num_loops):
 13          self.count = num_loops
 14
 15      def go(self):
 16          for i in range(self.count):
 17              pdb.set_trace()
 18  ->          print(i)
 19          return
```

The `longlist` (`ll`) command prints the source for the current function or frame, without having to determine the line numbers in advance. The command is “longlist” because for long functions it may produce considerably more output than the default for `list`.

```
(Pdb) longlist
 15      def go(self):
 16          for i in range(self.count):
 17              pdb.set_trace()
 18  ->          print(i)
 19          return
```

The `source` command loads and prints the full source for an arbitrary class, function, or module.

```
(Pdb) source MyObj
 10  class MyObj:
 11
 12      def __init__(self, num_loops):
 13          self.count = num_loops
 14
 15      def go(self):
 16          for i in range(self.count):
 17              pdb.set_trace()
 18              print(i)
 19          return
```

Move between frames within the current call stack using `up` and `down`. `up` (abbreviated `u`) moves towards older frames on the stack. `down` (`d`) moves towards newer frames. Each time you move up or down the stack, the debugger prints the current location in the same format as produced by `where`.

```
(Pdb) up
> .../pdb_set_trace.py(22)<module>()
-> MyObj(5).go()
```

```
(Pdb) down
> .../pdb_set_trace.py(18)go()
-> print(i)
```

Pass a numerical argument to either up or down to move that many steps up or down the stack at one time.

Examining Variables on the Stack

Each frame on the stack maintains a set of variables, including values local to the function being executed and global state information. pdb provides several ways to examine the contents of those variables.

```
# pdb_function_arguments.py
```

```
#!/usr/bin/env python3
# encoding: utf-8
#
# Copyright (c) 2010 Doug Hellmann.  All rights reserved.
#

import pdb


def recursive_function(n=5, output='to be printed'):
    if n > 0:
        recursive_function(n - 1)
    else:
        pdb.set_trace()
        print(output)
    return

if __name__ == '__main__':
    recursive_function()
```

The `args` command (abbreviated `a`) prints all of the arguments to the function active in the current frame. This example also uses a recursive function to show what a deeper stack looks like when printed by `where`.

```
$ python3 pdb_function_arguments.py
> .../pdb_function_arguments.py(15)recursive_function()
-> print(output)
(Pdb) where
  .../pdb_function_arguments.py(19)<module>()
-> recursive_function()
  .../pdb_function_arguments.py(12)recursive_function()
-> recursive_function(n - 1)
  .../pdb_function_arguments.py(12)recursive_function()
-> recursive_function(n - 1)
  .../pdb_function_arguments.py(12)recursive_function()
-> recursive_function(n - 1)
  .../pdb_function_arguments.py(12)recursive_function()
-> recursive_function(n - 1)
  .../pdb_function_arguments.py(12)recursive_function()
-> recursive_function(n - 1)
> .../pdb_function_arguments.py(15)recursive_function()
-> print(output)

(Pdb) args
n = 0
output = to be printed

(Pdb) up
> .../pdb_function_arguments.py(12)recursive_function()
-> recursive_function(n - 1)

(Pdb) args
n = 1
output = to be printed
```

The `p` command evaluates an expression given as argument and prints the result. Python's `print()` function is also available,

but it is passed through to the interpreter to be executed rather than running as a command in the debugger.

```
(Pdb) p n
1

(Pdb) print(n)
1
```

Similarly, prefixing an expression with `!` passes it to the Python interpreter to be evaluated. This feature can be used to execute arbitrary Python statements, including modifying variables. This example changes the value of `output` before letting the debugger continue running the program. The next statement after the call to `set_trace()` prints the value of `output`, showing the modified value.

```
$ python3 pdb_function_arguments.py

> .../pdb_function_arguments.py(14)recursive_function()
-> print(output)

(Pdb) !output
'to be printed'

(Pdb) !output='changed value'

(Pdb) continue
changed value
```

For more complicated values such as nested or large data structures, use `pp` to "pretty print" them. This program reads several lines of text from a file.

```
# pdb_pp.py
```

```
#!/usr/bin/env python3
# encoding: utf-8
#
# Copyright (c) 2010 Doug Hellmann.  All rights reserved.
#

import pdb

with open('lorem.txt', 'rt') as f:
    lines = f.readlines()

pdb.set_trace()
```

Printing the variable `lines` with `p` results in output that is difficult to read because it may wrap awkwardly. `pp` uses `pprint` to format the value for clean printing.

```
$ python3 pdb_pp.py

> .../pdb_pp.py(12)<module>()->None
-> pdb.set_trace()
(Pdb) p lines
['Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
\n', 'Donec egestas, enim et consecte tuer ullamcorper, lect
us \n', 'ligula rutrum leo, a elementum el it tortor eu quam
.\n']

(Pdb) pp lines
['Lorem ipsum dolor sit amet, consectetuer adipiscing elit. \n',
 'Donec egestas, enim et consectetuer ullamcorper, lectus \n',
 'ligula rutrum leo, a elementum elit tortor eu quam.\n']

(Pdb)
```

For interactive exploration and experimentation it is possible to drop from the debugger into a standard Python interactive prompt with the globals and locals from the current frame already populated.

```
$ python3 -m pdb pdb_interact.py
```

```
> .../pdb_interact.py(7)<module>()
-> import pdb
(Pdb) break 14
Breakpoint 1 at .../pdb_interact.py:14

(Pdb) continue
> .../pdb_interact.py(14)f()
-> print(l, m, n)

(Pdb) p l
['a', 'b']

(Pdb) p m
9

(Pdb) p n
5

(Pdb) interact
*interactive*

>>> l
['a', 'b']

>>> m
9

>>> n
5
```

Mutable objects such as lists can be changed from the interactive interpreter. Immutable objects cannot, and names cannot be rebound to new values.

```
>>> l.append('c')
>>> m += 7
>>> n = 3

>>> l
['a', 'b', 'c']

>>> m
16

>>> n
3
```

Use the end-of-file sequence Ctrl-D to exit the interactive prompt and return to the debugger. In this example, the list `l` has been changed but the values of `m` and `n` are not.

```
>>> ^D

(Pdb) p l
['a', 'b', 'c']

(Pdb) p m
9

(Pdb) p n
5

(Pdb)
```

Stepping Through a Program

In addition to navigating up and down the call stack when the program is paused, it is also possible to step through execution of the program past the point where it enters the debugger.

```
# pdb_step.py

#!/usr/bin/env python3
```

```
#!/usr/bin/env python3
# encoding: utf-8
#
# Copyright (c) 2010 Doug Hellmann.  All rights reserved.
#

import pdb


def f(n):
    for i in range(n):
        j = i * n
        print(i, j)
    return

if __name__ == '__main__':
    pdb.set_trace()
    f(5)
```

Use `step` (abbreviated `s`) to execute the current line and then stop at the next execution point – either the first statement inside a function being called or the next line of the current function.

```
$ python3 pdb_step.py

> .../pdb_step.py(18)<module>()
-> f(5)
```

The interpreter pauses after the call to `set_trace()` and gives control to the debugger. The first `step` causes the execution to enter `f()`.

```
(Pdb) step
--Call--
> .../pdb_step.py(10)f()
-> def f(n):
```

One more `step` moves execution to the first line of `f()` and starts the loop.

```
(Pdb) step
> .../pdb_step.py(11)f()
-> for i in range(n):
```

Stepping again moves to the first line inside the loop where `j` is defined.

```
(Pdb) step
> .../pdb_step.py(12)f()
-> j = i * n

(Pdb) p i
0
```

The value of `i` is `0`, so after one more step the value of `j` should also be `0`.

```
(Pdb) step
> .../pdb_step.py(13)f()
-> print(i, j)

(Pdb) p j
0

(Pdb)
```

Stepping one line at a time in this way can become tedious if there is a lot of code to cover before the point where the error occurs, or if the same function is called repeatedly.

```
# pdb_next.py

#!/usr/bin/env python3
# encoding: utf-8
```

```
#
# Copyright (c) 2010 Doug Hellmann.  All rights reserved.
#

import pdb


def calc(i, n):
    j = i * n
    return j


def f(n):
    for i in range(n):
        j = calc(i, n)
        print(i, j)
    return

if __name__ == '__main__':
    pdb.set_trace()
    f(5)
```

In this example, there is nothing wrong with `calc()`, so stepping through it each time it is called in the loop in `f()` obscures the useful output by showing all of the lines of `calc()` as they are executed.

```
$ python3 pdb_next.py

> .../pdb_next.py(23)<module>()
-> f(5)
(Pdb) step
--Call--
> .../pdb_next.py(15)f()
-> def f(n):

(Pdb) step
> .../pdb_next.py(16)f()
-> for i in range(n):

(Pdb) step
> .../pdb_next.py(17)f()
-> j = calc(i, n)

(Pdb) step
--Call--
> .../pdb_next.py(10)calc()
-> def calc(i, n):

(Pdb) step
> .../pdb_next.py(11)calc()
-> j = i * n

(Pdb) step
> .../pdb_next.py(12)calc()
-> return j

(Pdb) step
--Return--
> .../pdb_next.py(12)calc()->0
-> return j

(Pdb) step
> .../pdb_next.py(18)f()
-> print(i, j)

(Pdb) step
0 0

> .../pdb_next.py(16)f()
-> for i in range(n):
(Pdb)
```

The `next` command (abbreviated `n`) is like `step`, but does not enter functions called from the statement being executed. In effect, it steps all the way through the function call to the next statement in the current function in a single operation.

```
> .../pdb_next.py(16)f()
-> for i in range(n):
(Pdb) step
> .../pdb_next.py(17)f()
-> j = calc(i, n)

(Pdb) next
> .../pdb_next.py(18)f()
-> print(i, j)

(Pdb)
```

The `until` command is like `next`, except it explicitly continues until execution reaches a line in the same function with a line number higher than the current value. That means, for example, that `until` can be used to step past the end of a loop.

```
$ python3 pdb_next.py

> .../pdb_next.py(23)<module>()
-> f(5)
(Pdb) step
--Call--
> .../pdb_next.py(15)f()
-> def f(n):

(Pdb) step
> .../pdb_next.py(16)f()
-> for i in range(n):

(Pdb) step
> .../pdb_next.py(17)f()
-> j = calc(i, n)

(Pdb) next
> .../pdb_next.py(18)f()
-> print(i, j)

(Pdb) until
0 0
1 5
2 10
3 15
4 20
> .../pdb_next.py(19)f()
-> return

(Pdb)
```

Before the `until` command was run, the current line was 18, the last line of the loop. After `until` ran, execution was on line 19, and the loop had been exhausted.

To let execution run until a specific line, pass the line number to the `until` command. Unlike when setting a breakpoint, the line number passed to `until` must be higher than the current line number, so it is most useful for navigating within a function for skipping over long blocks.

```
$ python3 pdb_next.py
> .../pdb_next.py(23)<module>()
-> f(5)
(Pdb) list
 18          print(i, j)
 19      return
 20
 21  if __name__ == '__main__':
 22      pdb.set_trace()
 23  ->    f(5)
[EOF]
```

```
(Pdb) until 18
*** "until" line number is smaller than current line number

(Pdb) step
--Call--
> .../pdb_next.py(15)f()
-> def f(n):

(Pdb) step
> .../pdb_next.py(16)f()
-> for i in range(n):

(Pdb) list
 11      j = i * n
 12      return j
 13
 14
 15  def f(n):
 16  ->    for i in range(n):
 17          j = calc(i, n)
 18          print(i, j)
 19      return
 20
 21  if __name__ == '__main__':

(Pdb) until 19
0 0
1 5
2 10
3 15
4 20
> .../pdb_next.py(19)f()
-> return

(Pdb)
```

The `return` command is another short-cut for bypassing parts of a function. It continues executing until the function is about to execute a `return` statement, and then it pauses, providing time to look at the return value before the function returns.

```
$ python3 pdb_next.py

> .../pdb_next.py(23)<module>()
-> f(5)
(Pdb) step
--Call--
> .../pdb_next.py(15)f()
-> def f(n):

(Pdb) step
> .../pdb_next.py(16)f()
-> for i in range(n):

(Pdb) return
0 0
1 5
2 10
3 15
4 20
--Return--
> .../pdb_next.py(19)f()->None
-> return

(Pdb)
```

Breakpoints

As programs grow longer, even using `next` and `until` will become slow and cumbersome. Instead of stepping through the program by hand, a better solution is to let it run normally until it reaches a point where the debugger should interrupt it. `set_trace()` can start the debugger, but that only works if there is a single point in the program where it should pause. It is more convenient to run the program through the debugger, but tell the debugger where to stop in advance using *breakpoints*.

The debugger monitors the program, and when it reaches the location described by a breakpoint the program is paused before the line is executed.

```
# pdb_break.py
```

```
#!/usr/bin/env python3
# encoding: utf-8
#
# Copyright (c) 2010 Doug Hellmann.  All rights reserved.
#


def calc(i, n):
    j = i * n
    print('j =', j)
    if j > 0:
        print('Positive!')
    return j


def f(n):
    for i in range(n):
        print('i =', i)
        j = calc(i, n)  # noqa
    return

if __name__ == '__main__':
    f(5)
```

There are several options to the `break` command (abbreviated `b`) used for setting break points, including the line number, file, and function where processing should pause. To set a breakpoint on a specific line of the current file, use `break lineno`.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break 12
Breakpoint 1 at .../pdb_break.py:12

(Pdb) continue
i = 0
j = 0
i = 1
j = 5
> .../pdb_break.py(12)calc()
-> print('Positive!')

(Pdb)
```

The command `continue` (abbreviated `c`) tells the debugger to keep running the program until the next breakpoint. In this case, it runs through the first iteration of the `for` loop in `f()` and stops inside `calc()` during the second iteration.

Breakpoints can also be set to the first line of a function by specifying the function name instead of a line number. This example shows what happens if a breakpoint is added for the `calc()` function.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break calc
Breakpoint 1 at .../pdb_break.py:8

(Pdb) continue
i = 0
> .../pdb_break.py(9)calc()
-> j = i * n

(Pdb) where
  .../pdb_break.py(23)<module>()
```

```
-> f(5)
  .../pdb_break.py(19)f()
-> j = calc(i, n)
> .../pdb_break.py(9)calc()
-> j = i * n

(Pdb)
```

To specify a breakpoint in another file, prefix the line or function argument with a filename.

```
# pdb_break_remote.py
```

```
#!/usr/bin/env python3
# encoding: utf-8

from pdb_break import f

f(5)
```

Here a breakpoint is set for line 12 of `pdb_break.py` after starting the main program `pdb_break_remote.py`.

```
$ python3 -m pdb pdb_break_remote.py

> .../pdb_break_remote.py(4)<module>()
-> from pdb_break import f
(Pdb) break pdb_break.py:12
Breakpoint 1 at .../pdb_break.py:12

(Pdb) continue
i = 0
j = 0
i = 1
j = 5
> .../pdb_break.py(12)calc()
-> print('Positive!')

(Pdb)
```

The filename can be a full path to the source file, or a relative path to a file available on `sys.path`.

To list the breakpoints currently set, use `break` without any arguments. The output includes the file and line number of each break point, as well as information about how many times it has been encountered.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break 12
Breakpoint 1 at .../pdb_break.py:12

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:12

(Pdb) continue
i = 0
j = 0
i = 1
j = 5
> .../pdb/pdb_break.py(12)calc()
-> print('Positive!')

(Pdb) continue
Positive!
i = 2
j = 10
> .../pdb_break.py(12)calc()
-> print('Positive!')

(Pdb) break
```

```
(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:12
        breakpoint already hit 2 times

(Pdb)
```

Managing Breakpoints

As each new breakpoint is added, it is assigned a numerical identifier. These ID numbers are used to enable, disable, and remove the breakpoints interactively. Turning off a breakpoint with `disable` tells the debugger not to stop when that line is reached. The breakpoint is remembered, but ignored.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break calc
Breakpoint 1 at .../pdb_break.py:8

(Pdb) break 12
Breakpoint 2 at .../pdb_break.py:12

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:8
2   breakpoint   keep yes   at .../pdb_break.py:12

(Pdb) disable 1

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep no    at .../pdb_break.py:8
2   breakpoint   keep yes   at .../pdb_break.py:12

(Pdb) continue
i = 0
j = 0
i = 1
j = 5
> .../pdb_break.py(12)calc()
-> print('Positive!')

(Pdb)
```

The next debugging session sets two breakpoints in the program, then disables one. The program is run until the remaining breakpoint is encountered, and then the other breakpoint is turned back on with `enable` before execution continues.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break calc
Breakpoint 1 at .../pdb_break.py:8

(Pdb) break 18
Breakpoint 2 at .../pdb_break.py:18

(Pdb) disable 1

(Pdb) continue
> .../pdb_break.py(18)f()
-> print('i =', i)

(Pdb) list
 13      return j
 14
 15
 16  def f(n):
 17      for i in range(n):
 18 B >      print('i =', i)
```

```
 18 B->      print('i =', i)
 19          j = calc(i, n)
 20      return
 21
 22  if __name__ == '__main__':
 23      f(5)

(Pdb) continue
i = 0
j = 0
> .../pdb_break.py(18)f()
-> print('i =', i)

(Pdb) list
 13      return j
 14
 15
 16  def f(n):
 17      for i in range(n):
 18 B->      print('i =', i)
 19          j = calc(i, n)
 20      return
 21
 22  if __name__ == '__main__':
 23      f(5)

(Pdb) p i
 1

(Pdb) enable 1
Enabled breakpoint 1 at .../pdb_break.py:8

(Pdb) continue
i = 1
> .../pdb_break.py(9)calc()
-> j = i * n

(Pdb) list
  4  # Copyright (c) 2010 Doug Hellmann.  All rights reserved.
  5  #
  6
  7
  8 B   def calc(i, n):
  9  ->    j = i * n
 10        print('j =', j)
 11        if j > 0:
 12            print('Positive!')
 13        return j
 14

(Pdb)
```

The lines prefixed with `B` in the output from `list` show where the breakpoints are set in the program (lines 8 and 18).

Use `clear` to delete a breakpoint entirely.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break calc
Breakpoint 1 at .../pdb_break.py:8

(Pdb) break 12
Breakpoint 2 at .../pdb_break.py:12

(Pdb) break 18
Breakpoint 3 at .../pdb_break.py:18

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:8
```

```
2   breakpoint   keep yes   at .../pdb_break.py:12
3   breakpoint   keep yes   at .../pdb_break.py:18

(Pdb) clear 2
Deleted breakpoint 2

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:8
3   breakpoint   keep yes   at .../pdb_break.py:18

(Pdb)
```

The other breakpoints retain their original identifiers and are not renumbered.

Temporary Breakpoints

A temporary breakpoint is automatically cleared the first time program execution hits it. Using a temporary breakpoint makes it easy to reach a particular spot in the program flow quickly, just as with a regular breakpoint, but since it is cleared immediately it does not interfere with subsequent progress if that part of the program is run repeatedly.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) tbreak 12
Breakpoint 1 at .../pdb_break.py:12

(Pdb) continue
i = 0
j = 0
i = 1
j = 5
Deleted breakpoint 1 at .../pdb_break.py:12
> .../pdb_break.py(12)calc()
-> print('Positive!')

(Pdb) break

(Pdb) continue
Positive!
i = 2
j = 10
Positive!
i = 3
j = 15
Positive!
i = 4
j = 20
Positive!
The program finished and will be restarted
> .../pdb_break.py(8)<module>()
-> def calc(i, n):

(Pdb)
```

After the program reaches line 12 the first time, the breakpoint is removed and execution does not stop again until the program finishes.

Conditional Breakpoints

Rules can be applied to breakpoints so that execution only stops when the conditions are met. Using conditional breakpoints gives finer control over how the debugger pauses the program than enabling and disabling breakpoints by hand. Conditional breakpoints can be set in two ways. The first is to specify the condition when the breakpoint is set using `break`.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break 10, j>0
```

```
Breakpoint 1 at .../pdb_break.py:10

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:10
        stop only if j>0

(Pdb) continue
i = 0
j = 0
i = 1
> .../pdb_break.py(10)calc()
-> print('j =', j)

(Pdb)
```

The condition argument must be an expression using values visible in the stack frame where the breakpoint is defined. If the expression evaluates as true, execution stops at the breakpoint.

A condition can also be applied to an existing breakpoint using the `condition` command. The arguments are the breakpoint id and the expression.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break 10
Breakpoint 1 at .../pdb_break.py:10

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:10

(Pdb) condition 1 j>0

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:10
        stop only if j>0

(Pdb)
```

Ignoring Breakpoints

Programs that loop or use a large number of recursive calls to the same function are often easier to debug by skipping ahead in the execution, instead of watching every call or breakpoint. The `ignore` command tells the debugger to pass over a breakpoint without stopping. Each time processing encounters the breakpoint, it decrements the ignore counter. When the counter is zero, the breakpoint is re-activated.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break 19
Breakpoint 1 at .../pdb_break.py:19

(Pdb) continue
i = 0
> .../pdb_break.py(19)f()
-> j = calc(i, n)

(Pdb) next
j = 0
> .../pdb_break.py(17)f()
-> for i in range(n):

(Pdb) ignore 1 2
Will ignore next 2 crossings of breakpoint 1.

(Pdb) break
```

```
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:19
        ignore next 2 hits
        breakpoint already hit 1 time

(Pdb) continue
i = 1
j = 5
Positive!
i = 2
j = 10
Positive!
i = 3
> .../pdb_break.py(19)f()
-> j = calc(i, n)

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:19
        breakpoint already hit 4 times
```

Explicitly resetting the ignore count to zero re-enables the breakpoint immediately.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break 19
Breakpoint 1 at .../pdb_break.py:19

(Pdb) ignore 1 2
Will ignore next 2 crossings of breakpoint 1.

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:19
        ignore next 2 hits

(Pdb) ignore 1 0
Will stop next time breakpoint 1 is reached.

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_break.py:19
```

Triggering Actions on a Breakpoint

In addition to the purely interactive mode, pdb supports basic scripting. Using `commands`, a series of interpreter commands, including Python statements, can be executed when a specific breakpoint is encountered. After running `commands` with the breakpoint number as argument, the debugger prompt changes to `(com)`. Enter commands one a time, and finish the list with `end` to save the script and return to the main debugger prompt.

```
$ python3 -m pdb pdb_break.py

> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break 10
Breakpoint 1 at .../pdb_break.py:10

(Pdb) commands 1
(com) print('debug i =', i)
(com) print('debug j =', j)
(com) print('debug n =', n)
(com) end

(Pdb) continue
i = 0
debug i = 0
debug j = 0
debug n = 5
> .../pdb_break.py(10)calc()
```

```
> .../pdb_break.py(10)calc()
-> print('j =', j)

(Pdb) continue
j = 0
i = 1
debug i = 1
debug j = 5
debug n = 5
> .../pdb_break.py(10)calc()
-> print 'j =', j

(Pdb)
```

This feature is especially useful for debugging code that uses a lot of data structures or variables, since the debugger can be made to print out all of the values automatically, instead of doing it manually each time the breakpoint is encountered.

Watching Data Change

It is also possible to watch as values change during the course of program execution without scripting explicit `print` commands by using the `display` command.

```
$ python3 -m pdb pdb_break.py
> .../pdb_break.py(8)<module>()
-> def calc(i, n):
(Pdb) break 18
Breakpoint 1 at .../pdb_break.py:18

(Pdb) continue
> .../pdb_break.py(18)f()
-> print('i =', i)

(Pdb) display j
display j: ** raised NameError: name 'j' is not defined **

(Pdb) next
i = 0
> .../pdb_break.py(19)f()
-> j = calc(i, n)  # noqa

(Pdb) next
j = 0
> .../pdb_break.py(17)f()
-> for i in range(n):
display j: 0  [old: ** raised NameError: name 'j' is not defined **]

(Pdb)
```

Each time execution stops in the frame, the expression is evaluated and if it changes then the result is printed along with the old value. The `display` command with no argument prints a list of the displays active for the current frame.

```
(Pdb) display
Currently displaying:
j: 0

(Pdb) up
> .../pdb_break.py(23)<module>()
-> f(5)

(Pdb) display
Currently displaying:

(Pdb)
```

Remove a display expression with `undisplay`.

```
(Pdb) display
Currently displaying:
j: 0
```

```
(Pdb) undisplay j

(Pdb) display
Currently displaying:

(Pdb)
```

Changing Execution Flow

The `jump` command alters the flow of the program at runtime, without modifying the code. It can skip forward to avoid running some code, or backward to run it again. This sample program generates a list of numbers.

```
# pdb_jump.py
```

```
#!/usr/bin/env python3
# encoding: utf-8
#
# Copyright (c) 2010 Doug Hellmann.  All rights reserved.
#


def f(n):
    result = []
    j = 0
    for i in range(n):
        j = i * n + j
        j += n
        result.append(j)
    return result

if __name__ == '__main__':
    print(f(5))
```

When run without interference the output is a sequence of increasing numbers divisible by 5.

```
$ python3 pdb_jump.py

[5, 15, 30, 50, 75]
```

Jump Ahead

Jumping ahead moves the point of execution past the current location without evaluating any of the statements in between. By skipping over line 13 in the example, the value of `j` is not incremented and all of the subsequent values that depend on it are a little smaller.

```
$ python3 -m pdb pdb_jump.py

> .../pdb_jump.py(8)<module>()
-> def f(n):
(Pdb) break 13
Breakpoint 1 at .../pdb_jump.py:13

(Pdb) continue
> .../pdb_jump.py(13)f()
-> j += n

(Pdb) p j
0

(Pdb) step
> .../pdb_jump.py(14)f()
-> result.append(j)

(Pdb) p j
5

(Pdb) continue
> .../pdb_jump.py(13)f()
```

```
-> j += n

(Pdb) jump 14
> .../pdb_jump.py(14)f()
-> result.append(j)

(Pdb) p j
10

(Pdb) disable 1

(Pdb) continue
[5, 10, 25, 45, 70]

The program finished and will be restarted
> .../pdb_jump.py(8)<module>()
-> def f(n):
(Pdb)
```

Jump Back

Jumps can also move the program execution to a statement that has already been executed, to run it again. Here, the value of `j` is incremented an extra time, so the numbers in the result sequence are all larger than they would otherwise be.

```
$ python3 -m pdb pdb_jump.py

> .../pdb_jump.py(8)<module>()
-> def f(n):
(Pdb) break 14
Breakpoint 1 at .../pdb_jump.py:14

(Pdb) continue
> .../pdb_jump.py(14)f()
-> result.append(j)

(Pdb) p j
5

(Pdb) jump 13
> .../pdb_jump.py(13)f()
-> j += n

(Pdb) continue
> .../pdb_jump.py(14)f()
-> result.append(j)

(Pdb) p j
10

(Pdb) disable 1

(Pdb) continue
[10, 20, 35, 55, 80]

The program finished and will be restarted
> .../pdb_jump.py(8)<module>()
-> def f(n):
(Pdb)
```

Illegal Jumps

Jumping in and out of certain flow control statements is dangerous or undefined and therefore not allowed by the debugger.

```
# pdb_no_jump.py
```

```
#!/usr/bin/env python3
# encoding: utf-8
#
# Copyright (c) 2010 Doug Hellmann.  All rights reserved.
#
```

```
#


def f(n):
    if n < 0:
        raise ValueError('Invalid n: {}'.format(n))
    result = []
    j = 0
    for i in range(n):
        j = i * n + j
        j += n
        result.append(j)
    return result


if __name__ == '__main__':
    try:
        print(f(5))
    finally:
        print('Always printed')

    try:
        print(f(-5))
    except:
        print('There was an error')
    else:
        print('There was no error')

    print('Last statement')
```

`jump` can be used to enter a function, but the arguments are not defined and the code is unlikely to work.

```
$ python3 -m pdb pdb_no_jump.py

> .../pdb_no_jump.py(8)<module>()
-> def f(n):
(Pdb) break 22
Breakpoint 1 at .../pdb_no_jump.py:22

(Pdb) jump 9
> .../pdb_no_jump.py(9)<module>()
-> if n < 0:

(Pdb) p n
*** NameError: NameError("name 'n' is not defined",)

(Pdb) args

(Pdb)
```

`jump` will not enter the middle of a block such as a `for` loop or `try:except` statement.

```
$ python3 -m pdb pdb_no_jump.py

> .../pdb_no_jump.py(8)<module>()
-> def f(n):
(Pdb) break 22
Breakpoint 1 at .../pdb_no_jump.py:22

(Pdb) continue
> .../pdb_no_jump.py(22)<module>()
-> print(f(5))

(Pdb) jump 27
*** Jump failed: can't jump into the middle of a block

(Pdb)
```

The code in a `finally` block must all be executed, so `jump` will not leave the block.

```
$ python3 -m pdb pdb_no_jump.py

> .../pdb_no_jump.py(8)<module>()
-> def f(n):
(Pdb) break 24
Breakpoint 1 at .../pdb_no_jump.py:24

(Pdb) continue
[5, 15, 30, 50, 75]
> .../pdb_no_jump.py(24)<module>()
-> print 'Always printed'

(Pdb) jump 26
*** Jump failed: can't jump into or out of a 'finally' block

(Pdb)
```

And the most basic restriction is that jumping is constrained to the bottom frame on the call stack. After moving up the stack to examine variables, the execution flow cannot be changed at that point.

```
$ python3 -m pdb pdb_no_jump.py

> .../pdb_no_jump.py(8)<module>()
-> def f(n):
(Pdb) break 12
Breakpoint 1 at .../pdb_no_jump.py:12

(Pdb) continue
> .../pdb_no_jump.py(12)f()
-> j = 0

(Pdb) where
  .../lib/python3.5/bdb.py(
431)run()
-> exec cmd in globals, locals
  <string>(1)<module>()
  .../pdb_no_jump.py(22)<module>()
-> print(f(5))
> .../pdb_no_jump.py(12)f()
-> j = 0

(Pdb) up
> .../pdb_no_jump.py(22)<module>()
-> print(f(5))

(Pdb) jump 25
*** You can only jump within the bottom frame

(Pdb)
```

Restarting a Program

When the debugger reaches the end of the program, it automatically starts it over, but it can also be restarted explicitly without leaving the debugger and losing the current breakpoints or other settings.

```
# pdb_run.py
```

```
#!/usr/bin/env python3
# encoding: utf-8
#
# Copyright (c) 2010 Doug Hellmann.  All rights reserved.
#

import sys


def f():
    print('Command-line args:', sys.argv)
    return

```

```
14 if __name__ == '__main__':
15     f()
```

Running this program to completion within the debugger prints the name of the script file, since no other arguments were given on the command line.

```
$ python3 -m pdb pdb_run.py

> .../pdb_run.py(7)<module>()
-> import sys
(Pdb) continue

Command line args: ['pdb_run.py']
The program finished and will be restarted
> .../pdb_run.py(7)<module>()
-> import sys

(Pdb)
```

The program can be restarted using `run`. Arguments passed to `run` are parsed with `shlex` and passed to the program as though they were command line arguments, so the program can be restarted with different settings.

```
(Pdb) run a b c "this is a long value"
Restarting pdb_run.py with arguments:
        a b c this is a long value
> .../pdb_run.py(7)<module>()
-> import sys

(Pdb) continue
Command line args: ['pdb_run.py', 'a', 'b', 'c',
'this is a long value']
The program finished and will be restarted
> .../pdb_run.py(7)<module>()
-> import sys

(Pdb)
```

`run` can also be used at any other point in processing to restart the program.

```
$ python3 -m pdb pdb_run.py

> .../pdb_run.py(7)<module>()
-> import sys
(Pdb) break 11
Breakpoint 1 at .../pdb_run.py:11

(Pdb) continue
> .../pdb_run.py(11)f()
-> print('Command line args:', sys.argv)

(Pdb) run one two three
Restarting pdb_run.py with arguments:
        one two three
> .../pdb_run.py(7)<module>()
-> import sys

(Pdb)
```

Customizing the Debugger with Aliases

Avoid typing complex commands repeatedly by using `alias` to define a shortcut. Alias expansion is applied to the first word of each command. The body of the alias can consist of any command that is legal to type at the debugger prompt, including other debugger commands and pure Python expressions. Recursion is allowed in alias definitions, so one alias can even invoke another.

```
$ python3 -m pdb pdb_function_arguments.py

> .../pdb_function_arguments.py(7)<module>()
```

```
> .../pdb_function_arguments.py(7)<module>()
-> import pdb
(Pdb) break 11
Breakpoint 1 at .../pdb_function_arguments.py:11

(Pdb) continue
> .../pdb_function_arguments.py(11)recursive_function()
-> if n > 0:

(Pdb) pp locals().keys()
dict_keys(['output', 'n'])

(Pdb) alias pl pp locals().keys()

(Pdb) pl
dict_keys(['output', 'n'])
```

Running `alias` without any arguments shows the list of defined aliases. A single argument is assumed to be the name of an alias, and its definition is printed.

```
(Pdb) alias
pl = pp locals().keys()

(Pdb) alias pl
pl = pp locals().keys()

(Pdb)
```

Arguments to the alias are referenced using `%n` where `n` is replaced with a number indicating the position of the argument, starting with 1. To consume all of the arguments, use `%*`.

```
$ python3 -m pdb pdb_function_arguments.py

> .../pdb_function_arguments.py(7)<module>()
-> import pdb
(Pdb) alias ph !help(%1)

(Pdb) ph locals
Help on built-in function locals in module builtins:

locals()
    Return a dictionary containing the current scope's local
    variables.

    NOTE: Whether or not updates to this dictionary will affect
    name lookups in the local scope and vice-versa is
    *implementation dependent* and not covered by any backwards
    compatibility guarantees.
```

Clear the definition of an alias with `unalias`.

```
(Pdb) unalias ph

(Pdb) ph locals
*** SyntaxError: invalid syntax (<stdin>, line 1)

(Pdb)
```

Saving Configuration Settings

Debugging a program involves a lot of repetition: running the code, observing the output, adjusting the code or inputs, and running it again. pdb attempts to cut down on the amount of repetition needed to control the debugging experience, to let you concentrate on the code instead of the debugger. To help reduce the number of times you issue the same commands to the debugger, pdb can read a saved configuration from text files interpreted as it starts.

The file `~/.pdbrc` is read first, allowing global personal preferences for all debugging sessions. Then `./.pdbrc` is read from the current working directory, to set local preferences for a particular project.

```
$ cat ~/.pdbrc
```

```
# Show python help
alias ph !help(%1)
# Overridden alias
alias redefined p 'home definition'

$ cat .pdbrc

# Breakpoints
break 11
# Overridden alias
alias redefined p 'local definition'

$ python3 -m pdb pdb_function_arguments.py

Breakpoint 1 at .../pdb_function_arguments.py:11
> .../pdb_function_arguments.py(7)<module>()
-> import pdb
(Pdb) alias
ph = !help(%1)
redefined = p 'local definition'

(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at .../pdb_function_arguments.py:11

(Pdb)
```

Any configuration commands that can be typed at the debugger prompt can be saved in one of the start-up files. Some commands that control the execution (`continue`, `next`, etc.) can as well.

```
$ cat .pdbrc
break 11
continue
list

$ python3 -m pdb pdb_function_arguments.py
Breakpoint 1 at .../pdb_function_arguments.py:11
  6
  7  import pdb
  8
  9
  10  def recursive_function(n=5, output='to be printed'):
  11 B->    if n > 0:
  12            recursive_function(n - 1)
  13        else:
  14            pdb.set_trace()
  15            print(output)
  16        return
> .../pdb_function_arguments.py(11)recursive_function()
-> if n > 0:
(Pdb)
```

Especially useful is `run`, which means the command line arguments for a debugging session can be set in `./.pdbrc` so they are consistent across several runs.

```
$ cat .pdbrc
run a b c "long argument"

$ python3 -m pdb pdb_run.py
Restarting pdb_run.py with arguments:
      a b c "long argument"
> .../pdb_run.py(7)<module>()
-> import sys

(Pdb) continue
Command-line args: ['pdb_run.py', 'a', 'b', 'c',
'long argument']
The program finished and will be restarted
> .../pdb_run.py(7)<module>()
-> import sys
```

```
(Pdb)
```

See also

- Standard library documentation for pdb
- `readline` – Interactive prompt editing library.
- `cmd` – Build interactive programs.
- `shlex` – Shell command line parsing.
- Python issue 26053 – If the output of run does not match the values presented here, refer to this bug for details about a regression in pdb output between 2.7 and 3.5.

cgitb — Detailed Traceback Reports

profile and pstats — Performance Analysis

Quick Links

This page was last updated 2016-12-31.

Navigation

cgitb — Detailed Traceback Reports
profile and pstats — Performance Analysis

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?