pdb Python debugger

START HERE

`breakpoint()`
Stop here when this line runs

`p <expr>`
Print a value

`python -m pdb <script>`
Run a script under pdb

`w`
Where am I: the call stack

`n / s`
Over / into the next line

`ll`
Source of the current function

`c`
Run to the next breakpoint

`q`
Quit and abort the program

Start breakpoint(), -m pdb → **(Pdb) prompt** frame, locals → **Inspect** p, pp, w, ll → **Resume** n, s, r, c

CONCEPTS

Command names are written like s(tep): type the short part or the whole word. An empty line repeats the last command (except list, which continues).

Anything that is not a pdb command runs as Python in the current frame. Prefix ! when a name clashes with a command: !n = 1, !c.

Tab completes commands, names and attributes where readline is available.

START DEBUGGING

`breakpoint()`
Drop into pdb at this line
Built-in since Python 3.7; no import needed

`import pdb; pdb.set_trace()`
Same, the pre-3.7 way

`python -m pdb <script> <args>`
Run a script under pdb from its first line
Crash: post-mortem; normal end: restarts, keeping breakpoints

`python -m pdb -m <module>`
Debug a module, like python -m

`python -m pdb -c continue <script>`
Run freely, stop only on a crash

`python -m pdb -c 'until <line>' <script>`
Run up to a line, then stop

`python -m pdb -p <pid>`
Attach to a running process (3.14+)

`PYTHONBREAKPOINT=0`
Make every breakpoint() a no-op

`PYTHONBREAKPOINT=<mod>.<func>`
Send breakpoint() to another debugger
e.g. IPython.core.debugger.set_trace; unset or empty means pdb

AFTER A CRASH

`python -i <script>`
Keep the REPL open after an exception

`import pdb; pdb.pm()`
Post-mortem on the last uncaught exception

AFTER A CRASH (cont.)

`pdb.post_mortem()`
Debug the exception handled in an except block
Accepts an exception object since 3.13

`pytest --pdb`
Open pdb on every failing test

`pytest -x --pdb`
Debug the first failure, then stop the run

`pytest --trace`
Stop at the start of every test

`exceptions [<n>]`
List chained exceptions / switch to one (3.13+)

STEP AND RUN

`n(ext)`
Run the line, stepping over calls

`s(tep)`
Run the line, stepping into calls

`r(eturn)`
Run until the current function returns

`c(ontinue)`
Run until the next breakpoint

`unt(il)`
Run to a higher line: leaves a loop

`unt(il) <line>`
Run to a line in this frame

`j(ump) <line>`
Set the next line to run (skip or rerun code)
Bottom frame only; cannot jump into a for loop or out of finally

`run [<args>]`
Restart the program, keeping breakpoints
Only under python -m pdb; restart is an alias

`q(uit)`
Leave pdb and abort the program

STACK

`w(here)`
Show the stack, newest frame at the bottom
An arrow marks the selected frame; bt is an alias

`u(p) / d(own)`
Select the caller / callee frame

`u <n> / d <n>`
Move n frames at once

`a(rgs)`
Arguments of the current function

BREAKPOINTS

`b(reak) <line>`
Break at a line of the current file

`b <file>:<line>`
Break at a line of another file
The file is looked up on sys.path; .py may be left out

`b <func>`
Break at the first line of a function

`b <line>, <cond>`
Break only when the condition is true

`b`
List breakpoints with hit counts

`tbreak <line>`
One-shot breakpoint, deleted on first hit

`cl(ear) <n> [<n> ...]`
Delete breakpoints by number

`cl`
Delete all breakpoints (asks first)

`disable <n> / enable <n>`
Turn a breakpoint off / on

`condition <n> [<cond>]`
Change a condition; none removes it

`ignore <n> <count>`
Skip the next count hits

`commands [<n>]`
Commands to run on each hit, end with 'end'
Put silent first to hide the stop message

INSPECT

`p <expr>`
Print an expression

`pp <expr>`
Pretty-print it (dicts, nested data)

`whatis <expr>`
Show the type

`display <expr>`
Show it whenever it changes at a stop
In-place changes are missed: display lst[:] instead of lst

`undisplay [<expr>]`
Stop displaying one, or all

`retval`
Value the function is returning (rv)

`l(ist)`
11 lines around here, again continues

`l .`
List around the current line again

INSPECT (cont.)

`l <first>, <last>`
List a line range

`ll`
Whole source of the current function

`source <obj>`
Source code of a function or class

`p $_exception / $_retval`
Exception raised / value returned here (3.12+)

CHANGE AND RUN CODE

`!<stmt>`
Run a Python statement in this frame

`<var> = <value>`
Change a variable, then go on

`interact`
Full Python REPL with the frame's names
Ctrl+D returns to the (Pdb) prompt

`debug <code>`
Step into a call in a nested debugger

`h(elp) [<cmd>]`
List commands or explain one

CONFIG

`~/.pdbrc / ./.pdbrc`
Commands run at every start, home file first

`alias <name> <cmd>`
Define a command; %1, %* take arguments

RECIPES

Crash post-mortem

```
python -m pdb -c continue
<script>
# ... exception: pdb stops
where it was raised
(Pdb) w
(Pdb) u
(Pdb) pp locals()
```

Stop in a loop on a condition

```
(Pdb) b <file>:<line>, i == 99
(Pdb) c
(Pdb) display total
(Pdb) n
```

Log and go on

```
(Pdb) b <func>
(Pdb) commands
(com) silent
(com) p <arg>
(com) c
```

~/.pdbrc

```
# print an object's attributes:
pi <obj>
alias pi for k, v in
vars(%1).items(): print(k, v)
alias ps pi self
```