



Mark as Completed



- [Getting Started: Printing a Variable's Value](#)
- [Printing Expressions](#)
- [Stepping Through Code](#)
 - [Listing Source Code](#)
- [Using Breakpoints](#)
- [Continuing Execution](#)
- [Displaying Expressions](#)
- [Python Caller ID](#)
- [Essential pdb Commands](#)
- [Python Debugging With pdb: Conclusion](#)

 Remove ads

[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Python Debugging With pdb](#)

Help

Regardless of the situation, debugging code is a necessity, so it's a good idea to be comfortable working in a debugger. In this tutorial, I'll show you the basics of using pdb, Python's interactive source code debugger.

I'll walk you through a few common uses of pdb. You may want to bookmark this tutorial for quick reference later when you might really need it. pdb, and other debuggers, are indispensable tools. When you need a debugger, there's no substitute. You really need it.

By the end of this tutorial, you'll know how to use the debugger to see the state of any [variable](#) in your application. You'll also be able to stop and resume your application's flow of execution at any moment, so you can see exactly how each line of code affects its internal state.

This is great for tracking down hard-to-find bugs and allows you to fix faulty code more quickly and reliably. Sometimes, stepping through code in pdb and seeing how values change can be a real eye-opener and lead to "aha" moments, along with the occasional "face palm".

pdb is part of Python's standard library, so it's always there and available for use. This can be a life saver if you need to debug code in an environment where you don't have access to the GUI debugger you're familiar with.

The example code in this tutorial uses Python 3.6. You can find the source code for these examples on [GitHub](#).

At the end of this tutorial, there is a quick reference for [Essential pdb Commands](#).

There's also a printable pdb Command Reference you can use as a cheat sheet while debugging:

Free Bonus: [Click here to get a printable "pdb Command Reference" \(PDF\)](#) that you can keep on your desk and refer to while debugging.

Getting Started: Printing a Variable's Value

In this first example, we'll look at using pdb in its simplest form: checking the value of a variable.

Insert the following code at the location where you want to break into the debugger:

Python

```
import pdb; pdb.set_trace()
```

When the line above is executed, Python stops and waits for you to tell it what to do next. You'll see a (Pdb) prompt. This means that you're now paused in the interactive debugger and can enter a command.

Starting in Python 3.7, [there's another way to enter the debugger](#). [PEP 553](#) describes the built-in function `breakpoint()`, which makes entering the debugger easy and consistent:

Python

```
breakpoint()
```

By default, `breakpoint()` will `import` pdb and call `pdb.set_trace()`, as shown above. However, using `breakpoint()` is more flexible and allows you to control debugging behavior via its API and use of the environment variable `PYTHONBREAKPOINT`. For example, setting `PYTHONBREAKPOINT=0` in your environment will completely disable `breakpoint()`, thus disabling debugging. If you're using Python 3.7 or later, I encourage you to use `breakpoint()` instead of `pdb.set_trace()`.

You can also break into the debugger, without modifying the source and using `pdb.set_trace()` or `breakpoint()`, by running Python directly from the command-line and passing the option `-m pdb`. If your application accepts [command-line arguments](#), pass them as you normally would after the filename. For example:

Shell

```
$ python3 -m pdb app.py arg1 arg2
```

There are a lot of pdb commands available. At the end of this tutorial, there is a list of [Essential pdb Commands](#). For now, let's use the `p` command to print a variable's value. Enter `p variable_name` at the (Pdb) prompt to print its value.

Let’s look at the example. Here’s the `example1.py` source:

Python

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

filename = __file__
import pdb; pdb.set_trace()
print(f'path = {filename}')
```

If you run this from your shell, you should get the following output:

Shell

```
$ ./example1.py
> /code/example1.py(5)<module>()
-> print(f'path = {filename}')
```

(Pdb)

If you’re having trouble getting the examples or your own code to run from the command line, read [How Do I Make My Own Command-Line Commands Using Python?](#) If you’re on Windows, check the [Python Windows FAQ](#).

Now enter `p filename`. You should see:

Shell

```
(Pdb) p filename
'./example1.py'
```

(Pdb)

Since you’re in a shell and using a CLI (command-line interface), pay attention to the characters and formatting. They’ll give you the context you need:

- `>` starts the 1st line and tells you which source file you’re in. After the filename, there is the current line number in parentheses. Next is the name of the function. In this example, since we’re not paused inside a function and at module level, we see `<module>()`.
- `->` starts the 2nd line and is the current source line where Python is paused. This line hasn’t been executed yet. In this example, this is line 5 in `example1.py`, from the `>` line above.
- `(Pdb)` is pdb’s prompt. It’s waiting for a command.

Use the command `q` to quit debugging and exit.



**Master Real-World Python Skills
With a Community of Experts**

Level Up With Unlimited Access to Our Vast Library
of Python Tutorials and Video Lessons

Watch Now »

 [Remove ads](#)

Printing Expressions

When using the print command `p`, you’re passing an expression to be evaluated by Python. If you pass a variable name, pdb prints its current value. However, you can do much more to investigate the state of your running application.

In this example, the function `get_path()` is called. To inspect what’s happening in this function, I’ve inserted a call to `pdb.set_trace()` to pause execution just before it returns:

Python

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

import os

def get_path(filename):
    """Return file's path or empty string if no path."""
    head, tail = os.path.split(filename)
    import pdb; pdb.set_trace()
    return head

filename = __file__
print(f'path = {get_path(filename)}')
```

If you run this from your shell, you should get the output:

Shell



```
$ ./example2.py
> /code/example2.py(10)get_path()
-> return head
(Pdb)
```

Where are we?

- >: We're in the source file `example2.py` on line 10 in the function `get_path()`. This is the frame of reference the `p` command will use to resolve variable names, i.e. the current scope or context.
- ->: Execution has paused at `return head`. This line hasn't been executed yet. This is line 10 in `example2.py` in the function `get_path()`, from the `>` line above.

Let's print some expressions to look at the current state of the application. I use the command `ll` (`longlist`) initially to list the function's source:

Shell



```
(Pdb) ll
6     def get_path(filename):
7         """Return file's path or empty string if no path."""
8         head, tail = os.path.split(filename)
9         import pdb; pdb.set_trace()
10    ->     return head
(Pdb) p filename
'./example2.py'
(Pdb) p head, tail
('.', 'example2.py')
(Pdb) p 'filename: ' + filename
'filename: ./example2.py'
(Pdb) p get_path
<function get_path at 0x100760e18>
(Pdb) p getattr(get_path, '__doc__')
"Return file's path or empty string if no path."
(Pdb) p [os.path.split(p)[1] for p in os.path.sys.path]
['pdb-basics', 'python36.zip', 'python3.6', 'lib-dynload', 'site-packages']
(Pdb)
```

You can pass any valid Python expression to `p` for evaluation.

This is especially helpful when you are debugging and want to test an alternative implementation directly in the application at runtime.

You can also use the command `pp` (`pretty-print`) to pretty-print expressions. This is helpful if you want to print a variable or expression with a large amount of output, e.g. [lists](#) and dictionaries. Pretty-printing keeps objects on a single line if it can or breaks them onto multiple lines if they don't fit within the allowed width.

Stepping Through Code

There are two commands you can use to step through code when debugging:

Command	Description
n (next)	Continue execution until the next line in the current function is reached or it returns.
s (step)	Execute the current line and stop at the first possible occasion (either in a function that is called or in the current function).

There’s a 3rd command named `unt` (until). It is related to `n` (next). We’ll look at it later in this tutorial in the section [Continuing Execution](#).

The difference between `n` (next) and `s` (step) is where `pdb` stops.

Use `n` (next) to continue execution until the next line and stay within the current function, i.e. not stop in a foreign function if one is called. Think of next as “staying local” or “step over”.

Use `s` (step) to execute the current line and stop in a foreign function if one is called. Think of step as “step into”. If execution is stopped in another function, `s` will print `--Call--`.

Both `n` and `s` will stop execution when the end of the current function is reached and print `--Return--` along with the return value at the end of the next line after `->`.

Let’s look at an example using both commands. Here’s the `example3.py` source:

Python

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

import os

def get_path(filename):
    """Return file's path or empty string if no path."""
    head, tail = os.path.split(filename)
    return head

filename = __file__
import pdb; pdb.set_trace()
filename_path = get_path(filename)
print(f'path = {filename_path}')
```

If you run this from your shell and enter `n`, you should get the output:

Shell

```
$ ./example3.py
> /code/example3.py(14)<module>()
-> filename_path = get_path(filename)
(Pdb) n
> /code/example3.py(15)<module>()
-> print(f'path = {filename_path}')
(Pdb)
```

With `n` (next), we stopped on line 15, the next line. We “stayed local” in `<module>()` and “stepped over” the call to `get_path()`. The function is `<module>()` since we’re currently at module level and not paused inside another function.

Let’s try `s`:

Shell

```
$ ./example3.py
> /code/example3.py(14)<module>()
-> filename_path = get_path(filename)
(Pdb) s
--Call--
> /code/example3.py(6)get_path()
-> def get_path(filename):
(Pdb)
```

With `s` (step), we stopped on line 6 in the function `get_path()` since it was called on line 14. Notice the line `--Call--` after the `s` command.

Conveniently, `pdb` remembers your last command. If you’re stepping through a lot of code, you can just press to repeat the last command.

Below is an example of using both `s` and `n` to step through the code. I enter `s` initially because I want to “step into” the function `get_path()` and stop. Then I enter `n` once to “stay local” or “step over” any other function calls and just press to repeat the `n` command until I get to the last source line.

Shell



```
$ ./example3.py
> /code/example3.py(14)<module>()
-> filename_path = get_path(filename)
(Pdb) s
--Call--
> /code/example3.py(6)get_path()
-> def get_path(filename):
(Pdb) n
> /code/example3.py(8)get_path()
-> head, tail = os.path.split(filename)
(Pdb)
> /code/example3.py(9)get_path()
-> return head
(Pdb)
--Return--
> /code/example3.py(9)get_path()->'.'
-> return head
(Pdb)
> /code/example3.py(15)<module>()
-> print(f'path = {filename_path}')
(Pdb)
path = .
--Return--
> /code/example3.py(15)<module>()->None
-> print(f'path = {filename_path}')
(Pdb)
```

Note the lines `--Call--` and `--Return--`. This is `pdb` letting you know why execution was stopped. `n` (next) and `s` (step) will stop before a function returns. That’s why you see the `--Return--` lines above.

Also note `->'.'` at the end of the line after the first `--Return--` above:

Shell



```
--Return--
> /code/example3.py(9)get_path()->'.'
-> return head
(Pdb)
```

When `pdb` stops at the end of a function before it returns, it also prints the return value for you. In this example it’s `'.'`.



Become a Python Expert »

[Remove ads](#)

Listing Source Code

Don't forget the command `ll` (longlist: list the whole source code for the current function or frame). It's really helpful when you're stepping through unfamiliar code or you just want to see the entire function for context.

Here's an example:

Shell



```
$ ./example3.py
> /code/example3.py(14)<module>()
-> filename_path = get_path(filename)
(Pdb) s
--Call--
> /code/example3.py(6)get_path()
-> def get_path(filename):
(Pdb) ll
6 -> def get_path(filename):
7     """Return file's path or empty string if no path."""
8     head, tail = os.path.split(filename)
9     return head
(Pdb)
```

To see a shorter snippet of code, use the command `l` (list). Without arguments, it will print 11 lines around the current line or continue the previous listing. Pass the argument `.` to always list 11 lines around the current line: `l .`

Shell



```
$ ./example3.py
> /code/example3.py(14)<module>()
-> filename_path = get_path(filename)
(Pdb) l
9     return head
10
11
12     filename = __file__
13     import pdb; pdb.set_trace()
14 -> filename_path = get_path(filename)
15     print(f'path = {filename_path}')
[EOF]
(Pdb) l
[EOF]
(Pdb) l .
9     return head
10
11
12     filename = __file__
13     import pdb; pdb.set_trace()
14 -> filename_path = get_path(filename)
15     print(f'path = {filename_path}')
[EOF]
(Pdb)
```

Using Breakpoints

Breakpoints are very convenient and can save you a lot of time. Instead of stepping through dozens of lines you're not interested in, simply create a breakpoint where you want to investigate. Optionally, you can also tell `pdb` to break only when a certain condition is true.

Use the command `b` (break) to set a breakpoint. You can specify a line number or a function name where execution is stopped.

The syntax for break is:

Shell



```
b(reak) [ ([filename:]lineno | function) [, condition] ]
```

If `filename:` is not specified before the line number `lineno`, then the current source file is used.

Note the optional 2nd argument to `b`: `condition`. This is very powerful. Imagine a situation where you wanted to break only if a certain condition existed. If you pass a Python expression as the 2nd argument, `pdb` will break when the expression evaluates to `true`. We'll do this in an example below.

In this example, there's a utility module `util.py`. Let's set a breakpoint to stop execution in the function `get_path()`.

Here's the source for the main script `example4.py`:

Python

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

import util

filename = __file__
import pdb; pdb.set_trace()
filename_path = util.get_path(filename)
print(f'path = {filename_path}')
```

Here's the source for the utility module `util.py`:

Python

```
def get_path(filename):
    """Return file's path or empty string if no path."""
    import os
    head, tail = os.path.split(filename)
    return head
```

First, let's set a breakpoint using the source filename and line number:

Shell



```
$ ./example4.py
> /code/example4.py(7)<module>()
-> filename_path = util.get_path(filename)
(Pdb) b util:5
Breakpoint 1 at /code/util.py:5
(Pdb) c
> /code/util.py(5)get_path()
-> return head
(Pdb) p filename, head, tail
('./example4.py', '.', 'example4.py')
(Pdb)
```

The command `c` (continue) continues execution until a breakpoint is found.

Next, let's set a breakpoint using the function name:

Shell



```
$ ./example4.py
> /code/example4.py(7)<module>()
-> filename_path = util.get_path(filename)
(Pdb) b util.get_path
Breakpoint 1 at /code/util.py:1
(Pdb) c
> /code/util.py(3)get_path()
-> import os
(Pdb) p filename
'./example4.py'
(Pdb)
```


Enter `b` with no arguments to see a list of all breakpoints:

Shell



```
(Pdb) b
Num Type      Disp Enb   Where
1  breakpoint keep yes    at /code/util.py:1
(Pdb)
```

You can disable and re-enable breakpoints using the command `disable bnumber` and `enable bnumber`. `bnumber` is the breakpoint number from the breakpoints list's 1st column `Num`. Notice the `Enb` column's value change:

Shell



```
(Pdb) disable 1
Disabled breakpoint 1 at /code/util.py:1
(Pdb) b
Num Type      Disp Enb   Where
1  breakpoint keep no     at /code/util.py:1
(Pdb) enable 1
Enabled breakpoint 1 at /code/util.py:1
(Pdb) b
Num Type      Disp Enb   Where
1  breakpoint keep yes    at /code/util.py:1
(Pdb)
```

To delete a breakpoint, use the command `c1` (clear):

Shell



```
c1(ear) filename:lineno
c1(ear) [bnumber [bnumber...]]
```

Now let's use a Python expression to set a breakpoint. Imagine a situation where you wanted to break only if your troubled function received a certain input.

In this example scenario, the `get_path()` function is failing when it receives a relative path, i.e. the file's path doesn't start with `/`. I'll create an expression that evaluates to true in this case and pass it to `b` as the 2nd argument:

Shell



```
$ ./example4.py
> /code/example4.py(7)<module>()
-> filename_path = util.get_path(filename)
(Pdb) b util.get_path, not filename.startswith('/')
Breakpoint 1 at /code/util.py:1
(Pdb) c
> /code/util.py(3)get_path()
-> import os
(Pdb) a
filename = './example4.py'
(Pdb)
```

After you create the breakpoint above and enter `c` to continue execution, `pdb` stops when the expression evaluates to true. The command `a` (args) prints the argument list of the current function.

In the example above, when you're setting the breakpoint with a function name rather than a line number, note that the expression should use only function arguments or [global variables](#) that are available at the time the function is entered. Otherwise, the breakpoint will stop execution in the function regardless of the expression's value.

If you need to break using an expression with a variable name located inside a function, i.e. a variable name not in the function's argument list, specify the line number:

Shell



```
$ ./example4.py
> /code/example4.py(7)<module>()
-> filename_path = util.get_path(filename)
(Pdb) b util:5, not head.startswith('/')
Breakpoint 1 at /code/util.py:5
(Pdb) c
> /code/util.py(5)get_path()
-> return head
(Pdb) p head
'.'
(Pdb) a
filename = './example4.py'
(Pdb)
```

You can also set a temporary breakpoint using the command `tbreak`. It’s removed automatically when it’s first hit. It uses the same arguments as `b`.



[Learn Python »](#)

 [Remove ads](#)

Continuing Execution

So far, we’ve looked at stepping through code with `n` (next) and `s` (step) and using breakpoints with `b` (break) and `c` (continue).

There’s also a related command: `unt` (until).

Use `unt` to continue execution like `c`, but stop at the next line greater than the current line. Sometimes `unt` is more convenient and quicker to use and is exactly what you want. I’ll demonstrate this with an example below.

Let’s first look at the syntax and description for `unt`:

Command	Syntax	Description
unt	unt(il) [lineno]	Without <code>lineno</code> , continue execution until the line with a number greater than the current one is reached. With <code>lineno</code> , continue execution until a line with a number greater or equal to that is reached. In both cases, also stop when the current frame returns.

Depending on whether or not you pass the line number argument `lineno`, `unt` can behave in two ways:

- Without `lineno`, continue execution until the line with a number greater than the current one is reached. This is similar to `n` (next). It’s an alternate way to execute and “step over” code. The difference between `n` and `unt` is that `unt` stops only when a line with a number greater than the current one is reached. `n` will stop at the next logically executed line.
- With `lineno`, continue execution until a line with a number greater or equal to that is reached. This is like `c` (continue) with a line number argument.

In both cases, `unt` stops when the current frame (function) returns, just like `n` (next) and `s` (step).

The primary behavior to note with `unt` is that it will stop when a line number **greater or equal** to the current or specified line is reached.

Use `unt` when you want to continue execution and stop farther down in the current source file. You can treat it like a hybrid of `n` (next) and `b` (break), depending on whether you pass a line number argument or not.

In the example below, there is a function with a loop. Here, you want to continue execution of the code and stop after the loop, without stepping through each iteration of the loop or setting a breakpoint:

Here’s the example source for `example4unt.py`:

Python

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

import os

def get_path(fname):
    """Return file's path or empty string if no path."""
    import pdb; pdb.set_trace()
    head, tail = os.path.split(fname)
    for char in tail:
        pass # Check filename char
    return head

filename = __file__
filename_path = get_path(filename)
print(f'path = {filename_path}')
```

And the console output using `unt`:

Shell



```
$ ./example4unt.py
> /code/example4unt.py(9)get_path()
-> head, tail = os.path.split(fname)
(Pdb) ll
   6      def get_path(fname):
   7          """Return file's path or empty string if no path."""
   8          import pdb; pdb.set_trace()
   9  ->      head, tail = os.path.split(fname)
  10          for char in tail:
  11              pass # Check filename char
  12          return head
(Pdb) unt
> /code/example4unt.py(10)get_path()
-> for char in tail:
(Pdb)
> /code/example4unt.py(11)get_path()
-> pass # Check filename char
(Pdb)
> /code/example4unt.py(12)get_path()
-> return head
(Pdb) p char, tail
('y', 'example4unt.py')
```

The `ll` command was used first to print the function's source, followed by `unt`. `pdb` remembers the last command entered, so I just pressed `Enter` to repeat the `unt` command. This continued execution through the code until a source line greater than the current line was reached.

Note in the console output above that `pdb` stopped only once on lines 10 and 11. Since `unt` was used, execution was stopped only in the 1st iteration of the loop. However, each iteration of the loop was executed. This can be verified in the last line of output. The `char` variable's value `'y'` is equal to the last character in `tail`'s value `'example4unt.py'`.

Displaying Expressions

Similar to printing expressions with `p` and `pp`, you can use the command `display [expression]` to tell `pdb` to automatically display the value of an expression, if it changed, when execution stops. Use the command `undisplay [expression]` to clear a display expression.

Here's the syntax and description for both commands:

Command	Syntax	Description
display	display [expression]	Display the value of expression if it changed, each time execution stops in the current frame. Without expression, list all display expressions for the current frame.
undisplay	undisplay [expression]	Do not display expression any more in the current frame. Without expression, clear all display expressions for the current frame.

Below is an example, `example4display.py`, demonstrating its use with a loop:

Shell

```
$ ./example4display.py
> /code/example4display.py(9)get_path()
-> head, tail = os.path.split(fname)
(Pdb) ll
6     def get_path(fname):
7         """Return file's path or empty string if no path."""
8         import pdb; pdb.set_trace()
9     ->     head, tail = os.path.split(fname)
10         for char in tail:
11             pass # Check filename char
12         return head
(Pdb) b 11
Breakpoint 1 at /code/example4display.py:11
(Pdb) c
> /code/example4display.py(11)get_path()
-> pass # Check filename char
(Pdb) display char
display char: 'e'
(Pdb) c
> /code/example4display.py(11)get_path()
-> pass # Check filename char
display char: 'x' [old: 'e']
(Pdb)
> /code/example4display.py(11)get_path()
-> pass # Check filename char
display char: 'a' [old: 'x']
(Pdb)
> /code/example4display.py(11)get_path()
-> pass # Check filename char
display char: 'm' [old: 'a']
```

In the output above, `pdb` automatically displayed the value of the `char` variable because each time the breakpoint was hit its value had changed. Sometimes this is helpful and exactly what you want, but there’s another way to use `display`.

You can enter `display` multiple times to build a watch list of expressions. This can be easier to use than `p`. After adding all of the expressions you’re interested in, simply enter `display` to see the current values:

Shell

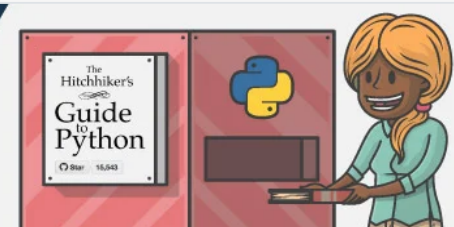
```

$ ./example4display.py
> /code/example4display.py(9)get_path()
-> head, tail = os.path.split(fname)
(Pdb) ll
 6     def get_path(fname):
 7         """Return file's path or empty string if no path."""
 8         import pdb; pdb.set_trace()
 9     ->     head, tail = os.path.split(fname)
10         for char in tail:
11             pass # Check filename char
12         return head
(Pdb) b 11
Breakpoint 1 at /code/example4display.py:11
(Pdb) c
> /code/example4display.py(11)get_path()
-> pass # Check filename char
(Pdb) display char
display char: 'e'
(Pdb) display fname
display fname: './example4display.py'
(Pdb) display head
display head: '.'
(Pdb) display tail
display tail: 'example4display.py'
(Pdb) c
> /code/example4display.py(11)get_path()
-> pass # Check filename char
display char: 'x' [old: 'e']
(Pdb) display
Currently displaying:
char: 'x'
fname: './example4display.py'
head: '.'
tail: 'example4display.py'

```

Your Guide to the Python Programming Language and a Best Practices Handbook

python-guide.org



[Remove ads](#)

Python Caller ID

In this last section, we'll build upon what we've learned so far and finish with a nice payoff. I use the name “caller ID” in reference to the phone system's caller identification feature. That is exactly what this example demonstrates, except it's applied to Python.

Here's the source for the main script `example5.py`:

Python

```

#!/usr/bin/env python3

import fileutil

def get_file_info(full_fname):
    file_path = fileutil.get_path(full_fname)
    return file_path

filename = __file__
filename_path = get_file_info(filename)
print(f'path = {filename_path}')

```

Here's the utility module `fileutil.py`:

Python

```
def get_path(fname):  
    """Return file's path or empty string if no path."""  
    import os  
    import pdb; pdb.set_trace()  
    head, tail = os.path.split(fname)  
    return head
```

In this scenario, imagine there's a large code base with a function in a utility module, `get_path()`, that's being called with invalid input. However, it's being called from many places in different packages.

How do you find who the caller is?

Use the command `w` (where) to print a stack trace, with the most recent frame at the bottom:

Shell



```
$ ./example5.py  
> /code/fileutil.py(5)get_path()  
-> head, tail = os.path.split(fname)  
(Pdb) w  
/code/example5.py(12)<module>()  
-> filename_path = get_file_info(filename)  
/code/example5.py(7)get_file_info()  
-> file_path = fileutil.get_path(full_fname)  
> /code/fileutil.py(5)get_path()  
-> head, tail = os.path.split(fname)  
(Pdb)
```

Don't worry if this looks confusing or if you're not sure what a stack trace or frame is. I'll explain those terms below. It's not as difficult as it might sound.

Since the most recent frame is at the bottom, start there and read from the bottom up. Look at the lines that start with `->`, but skip the 1st instance since that's where `pdb.set_trace()` was used to enter `pdb` in the function `get_path()`. In this example, the source line that called the function `get_path()` is:

Shell



```
-> file_path = fileutil.get_path(full_fname)
```

The line above each `->` contains the filename, line number (in parentheses), and function name the source line is in. So the caller is:

Shell



```
/code/example5.py(7)get_file_info()  
-> file_path = fileutil.get_path(full_fname)
```

That's no surprise in this small example for demonstration purposes, but imagine a large application where you've set a breakpoint with a condition to identify where a bad input value is originating.

Now we know how to find the caller.

But what about this stack trace and frame stuff?

A [stack trace](#) is just a list of all the frames that Python has created to keep track of function calls. A frame is a data structure Python creates when a function is called and deletes when it returns. The stack is simply an ordered list of frames or function calls at any point in time. The (function call) stack grows and shrinks throughout the life of an application as functions are called and then return.

When printed, this ordered list of frames, the stack, is called a [stack trace](#). You can see it at any time by entering the command `w`, as we did above to find the caller.

See this [call stack article on Wikipedia](#) for details.

To understand better and get more out of pdb, let’s look more closely at the help for w:

Shell

```
(Pdb) h w
w(here)
    Print a stack trace, with the most recent frame at the bottom.
    An arrow indicates the "current frame", which determines the
    context of most commands. 'bt' is an alias for this command.
```

What does pdb mean by “current frame”?

Think of the current frame as the current function where pdb has stopped execution. In other words, the current frame is where your application is currently paused and is used as the “frame” of reference for pdb commands like p (print).

p and other commands will use the current frame for context when needed. In the case of p, the current frame will be used for looking up and printing variable references.

When pdb prints a stack trace, an arrow > indicates the current frame.

How is this useful?

You can use the two commands u (up) and d (down) to change the current frame. Combined with p, this allows you to inspect variables and state in your application at any point along the call stack in any frame.

Here’s the syntax and description for both commands:

Command	Syntax	Description
u	u(p) [count]	Move the current frame count (default one) levels up in the stack trace (to an older frame).
d	d(own) [count]	Move the current frame count (default one) levels down in the stack trace (to a newer frame).

Let’s look at an example using the u and d commands. In this scenario, we want to inspect the variable full_fname that’s local to the function get_file_info() in example5.py. In order to do this, we have to change the current frame up one level using the command u:

Shell

```
$ ./example5.py
> /code/fileutil.py(5)get_path()
-> head, tail = os.path.split(fname)
(Pdb) w
/code/example5.py(12)<module>()
-> filename_path = get_file_info(filename)
/code/example5.py(7)get_file_info()
-> file_path = fileutil.get_path(full_fname)
> /code/fileutil.py(5)get_path()
-> head, tail = os.path.split(fname)
(Pdb) u
> /code/example5.py(7)get_file_info()
-> file_path = fileutil.get_path(full_fname)
(Pdb) p full_fname
'./example5.py'
(Pdb) d
> /code/fileutil.py(5)get_path()
-> head, tail = os.path.split(fname)
(Pdb) p fname
'./example5.py'
(Pdb)
```

The call to pdb.set_trace() is in fileutil.py in the function get_path(), so the current frame is initially set there. You can see it in the 1st line of output above:

Shell

```
> /code/fileutil.py(5)get_path()
```

To access and print the local variable `full_fname` in the function `get_file_info()` in `example5.py`, the command `u` was used to move up one level:

Shell

```
(Pdb) u
> /code/example5.py(7)get_file_info()
-> file_path = fileutil.get_path(full_fname)
```

Note in the output of `u` above that `pdb` printed the arrow `>` at the beginning of the 1st line. This is `pdb` letting you know the frame was changed and this source location is now the current frame. The variable `full_fname` is accessible now. Also, it's important to realize the source line starting with `->` on the 2nd line has been executed. Since the frame was moved up the stack, `fileutil.get_path()` has been called. Using `u`, we moved up the stack (in a sense, back in time) to the function `example5.get_file_info()` where `fileutil.get_path()` was called.

Continuing with the example, after `full_fname` was printed, the current frame was moved to its original location using `d`, and the local variable `fname` in `get_path()` was printed.

If we wanted to, we could have moved multiple frames at once by passing the `count` argument to `u` or `d`. For example, we could have moved to module level in `example5.py` by entering `u 2`:


Shell

```
$ ./example5.py
> /code/fileutil.py(5)get_path()
-> head, tail = os.path.split(fname)
(Pdb) u 2
> /code/example5.py(12)<module>()
-> filename_path = get_file_info(filename)
(Pdb) p filename
'./example5.py'
(Pdb)
```

It's easy to forget where you are when you're debugging and thinking of many different things. Just remember you can always use the aptly named command `w` (where) to see where execution is paused and what the current frame is.

A Python Best Practices Handbook

python-guide.org



 [Remove ads](#)

Essential pdb Commands

Once you've spent a little time with `pdb`, you'll realize a little knowledge goes a long way. Help is always available with the `h` command.

Just enter `h` or `help <topic>` to get a list of all commands or help for a specific command or topic.

For quick reference, here's a list of essential commands:

Command	Description
<code>p</code>	Print the value of an expression.
<code>pp</code>	Pretty-print the value of an expression.
<code>n</code>	Continue execution until the next line in the current function is reached or it returns.

Command	Description
s	Execute the current line and stop at the first possible occasion (either in a function that is called or in the current function).
c	Continue execution and only stop when a breakpoint is encountered.
unt	Continue execution until the line with a number greater than the current one is reached. With a line number argument, continue execution until a line with a number greater or equal to that is reached.
l	List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing.
ll	List the whole source code for the current function or frame.
b	With no arguments, list all breaks. With a line number argument, set a breakpoint at this line in the current file.
w	Print a stack trace, with the most recent frame at the bottom. An arrow indicates the current frame, which determines the context of most commands.
u	Move the current frame count (default one) levels up in the stack trace (to an older frame).
d	Move the current frame count (default one) levels down in the stack trace (to a newer frame).
h	See a list of available commands.
h <topic>	Show help for a command or topic.
h pdb	Show the full pdb documentation.
q	Quit the debugger and exit.

Python Debugging With pdb: Conclusion

In this tutorial, we covered a few basic and common uses of pdb:

- printing expressions
- stepping through code with n (next) and s (step)
- using breakpoints
- continuing execution with unt (until)
- displaying expressions
- finding the caller of a function

I hope it’s been helpful to you. If you’re curious about learning more, see:

- pdb’s full documentation at a pdb prompt near you: (Pdb) h pdb
- [Python’s pdb docs](#)

The source code used in the examples can be found on the associated [GitHub repository](#). Be sure to check out our printable pdb Command Reference, which you can use as a cheat sheet while debugging:

Free Bonus: [Click here to get a printable "pdb Command Reference" \(PDF\)](#) that you can keep on your desk and refer to while debugging.

Also, if you’d like to try a GUI-based Python debugger, read our [Python IDEs and Editors Guide](#) to see what options will work best for you. Happy Pythoning!