**How To Use the Python Debugger**

* **By**[**Lisa Tagliaferri**](https://www.digitalocean.com/community/users/ltagliaferri)

**PostedApril 25, 2017 161.5kviews**

**Working Interactively with the Python Debugger**

The Python debugger comes as part of the standard Python distribution as a module called pdb. The debugger is also extensible, and is defined as the class Pdb. You can read the [official documentation of pdb](https://docs.python.org/3/library/pdb.html) to learn more.

We’ll begin by working with a short program that has two global [variables](https://www.digitalocean.com/community/tutorials/how-to-use-variables-in-python-3), a [function](https://www.digitalocean.com/community/tutorials/how-to-define-functions-in-python-3) that creates a nested [loop](https://www.digitalocean.com/community/tutorials/how-to-construct-for-loops-in-python-3), and the if \_\_name\_\_ == '\_\_main\_\_': construction that will call the nested\_loop() function.

looping.py

num\_list = [500, 600, 700]

alpha\_list = ['x', 'y', 'z']

def nested\_loop():

for number in num\_list:

print(number)

for letter in alpha\_list:

print(letter)

if \_\_name\_\_ == '\_\_main\_\_':

nested\_loop()

Copy

We can now run this program through the Python debugger by using the following command:

* python -m pdb looping.py

The -m command-line flag will import a module for you and run it as a script. In this case we are importing and running the pdb module.

Upon running this command, you’ll receive the following output:

Output

> /Users/sammy/looping.py(1)<module>()

-> num\_list = [500, 600, 700]

(Pdb)

In the output, the first line contains the current module name (as indicated with <module>) with a directory path, and the printed line number that follows (in this case it’s 1, but if there is a comment or other non-executable line it could be a higher number). The second line shows the current line of source code that is executed here, as pdb provides an interactive console for debugging. You can use the command help to learn its commands, and help command to learn more about a specific command. Note that the pdb console is different than the Python interactive shell.

The Python debugger will automatically start over when it reaches the end of your program. Whenever you want to leave the pdb console, type the command quit or exit. If you would like to explicitly restart a program at any place within the program, you can do so with the command run.

**Using the Debugger to Move through a Program**

When working with programs in the Python debugger, you’re likely to use the list, step, and next commands to move through your code. We’ll go over these commands in this section.

Within the shell, we can type the command list in order to get context around the current line. From the first line of the program looping.py that we displayed above — num\_list = [500, 600, 700] — that will look like this:

(Pdb) list

1 -> num\_list = [500, 600, 700]

2 alpha\_list = ['x', 'y', 'z']

3

4

5 def nested\_loop():

6 for number in num\_list:

7 print(number)

8 for letter in alpha\_list:

9 print(letter)

10

11 if \_\_name\_\_ == '\_\_main\_\_':

(Pdb)

The current line is indicated with the characters ->, which in our case is the first line of the program file.

Since this is a relatively short program, we receive nearly all of the program back with the list command. Without providing arguments, the list command provides 11 lines around the current line, but you can also specify which lines to include, like so:

(Pdb) list 3, 7

3

4

5 def nested\_loop():

6 for number in num\_list:

7 print(number)

(Pdb)

Here, we requested that the lines 3-7 be displayed by using the command list 3, 7.

To move through the program line by line, we can use step or next:

(Pdb) step

> /Users/sammy/looping.py(2)<module>()

-> alpha\_list = ['x', 'y', 'z']

(Pdb)

(Pdb) next

> /Users/sammy/looping.py(2)<module>()

-> alpha\_list = ['x', 'y', 'z']

(Pdb)

The difference between step and next is that step will stop within a called function, while next executes called functions to only stop at the next line of the current function. We can see this difference when we work with the function.

The step command will iterate through the loops once it gets to the running of the function, showing exactly what the loop is doing, as it will first print a number with print(number) then go through to print the letters with print(letter), return to the number, etc:

(Pdb) step

> /Users/sammy/looping.py(5)<module>()

-> def nested\_loop():

(Pdb) step

> /Users/sammy/looping.py(11)<module>()

-> if \_\_name\_\_ == '\_\_main\_\_':

(Pdb) step

> /Users/sammy/looping.py(12)<module>()

-> nested\_loop()

(Pdb) step

--Call--

> /Users/sammy/looping.py(5)nested\_loop()

-> def nested\_loop():

(Pdb) step

> /Users/sammy/looping.py(6)nested\_loop()

-> for number in num\_list:

(Pdb) step

> /Users/sammy/looping.py(7)nested\_loop()

-> print(number)

(Pdb) step

500

> /Users/sammy/looping.py(8)nested\_loop()

-> for letter in alpha\_list:

(Pdb) step

> /Users/sammy/looping.py(9)nested\_loop()

-> print(letter)

(Pdb) step

x

> /Users/sammy/looping.py(8)nested\_loop()

-> for letter in alpha\_list:

(Pdb) step

> /Users/sammy/looping.py(9)nested\_loop()

-> print(letter)

(Pdb) step

y

> /Users/sammy/looping.py(8)nested\_loop()

-> for letter in alpha\_list:

(Pdb)

The next command, instead, will execute the entire function without showing the step-by-step process. Let’s quit the current session with the exit command and then begin the debugger again:

* python -m pdb looping.py

Now we can work with the next command:

(Pdb) next

> /Users/sammy/looping.py(5)<module>()

-> def nested\_loop():

(Pdb) next

> /Users/sammy/looping.py(11)<module>()

-> if \_\_name\_\_ == '\_\_main\_\_':

(Pdb) next

> /Users/sammy/looping.py(12)<module>()

-> nested\_loop()

(Pdb) next

500

x

y

z

600

x

y

z

700

x

y

z

--Return--

> /Users/sammy/looping.py(12)<module>()->None

-> nested\_loop()

(Pdb)

While going through your code, you may want to examine the value passed to a variable, which you can do with the pp command, which will pretty-print the value of the expression using the [pprint module](https://docs.python.org/3/library/pprint.html" \l "module-pprint):

(Pdb) pp num\_list

[500, 600, 700]

(Pdb)

Most commands in pdbhave shorter aliases. For step that short form is s, and for next it is n. The help command will list available aliases. You can also call the last command you called by pressing the ENTER key at the prompt.

**Breakpoints**

You typically will be working with larger programs than the example above, so you’ll likely be wanting to look at particular functions or lines rather than going through an entire program. By using the break command to set breakpoints, you’ll run the program up until the specified breakpoint.

When you insert a breakpoint, the debugger assigns a number to it. The numbers assigned to breakpoints are successive integers that begin with the number 1, which you can refer to when working with breakpoints.

Breakpoints can be placed at certain line numbers by following the syntax of <program\_file>:<line\_number> as shown below:

(Pdb) break looping.py:5

Breakpoint 1 at /Users/sammy/looping.py:5

(Pdb)

Type clear and then y to remove all current breakpoints. You can then place a breakpoint where a function is defined:

(Pdb) break looping.nested\_loop

Breakpoint 1 at /Users/sammy/looping.py:5

(Pdb)

To remove current breakpoints, type clear and then y. You can also set up a condition:

(Pdb) break looping.py:7, number > 500

Breakpoint 1 at /Users/sammy/looping.py:7

(Pdb)

Now, if we issue the continue command, the program will break when the number x is evaluated to being greater than 500 (that is, when it is set equal to 600 in the second iteration of the outer loop):

(Pdb) continue

500

x

y

z

> /Users/sammy/looping.py(7)nested\_loop()

-> print(number)

(Pdb)

To see a list of breakpoints that are currently set to run, use the command break without any arguments. You’ll receive information about the particularities of the breakpoint(s) you’ve set:

(Pdb) break

Num Type Disp Enb Where

1 breakpoint keep yes at /Users/sammy/looping.py:7

stop only if number > 500

breakpoint already hit 2 times

(Pdb)

We can also disable a breakpoint with the command disable and the number of the breakpoint. In this session, we add another breakpoint and then disable the first one:

(Pdb) break looping.py:11

Breakpoint 2 at /Users/sammy/looping.py:11

(Pdb) disable 1

Disabled breakpoint 1 at /Users/sammy/looping.py:7

(Pdb) break

Num Type Disp Enb Where

1 breakpoint keep no at /Users/sammy/looping.py:7

stop only if number > 500

breakpoint already hit 2 times

2 breakpoint keep yes at /Users/sammy/looping.py:11

(Pdb)

To enable a breakpoint, use the enable command, and to remove a breakpoint entirely, use the clear command:

(Pdb) enable 1

Enabled breakpoint 1 at /Users/sammy/looping.py:7

(Pdb) clear 2

Deleted breakpoint 2 at /Users/sammy/looping.py:11

(Pdb)

Breakpoints in pdb provide you with a lot of control. Some additional functionalities include ignoring breakpoints during the current iteration of the program with the ignore command (as in ignore 1), triggering actions to occur at a breakpoint with the commands command (as in command 1), and creating temporary breakpoints that are automatically cleared the first time program execution hits the point with the command tbreak (for a temporary break at line 3, for example, you could type tbreak 3).

**Integrating pdb into Programs**

You can trigger a debugging session by importing the pdb module and adding the pdb function pdb.set\_trace() above the line where you would like the session to begin.

In our sample program above, we’ll add the import statement and the function where we would like to enter into the debugger. For our example, let’s add it before the nested loop.

num\_list = [500, 600, 700]

alpha\_list = ['x', 'y', 'z']

def nested\_loop():

for number in num\_list:

print(number)

# Trigger debugger at this line

import pdb

pdb.set\_trace()

for letter in alpha\_list:

print(letter)

if \_\_name\_\_ == '\_\_main\_\_':

nested\_loop()

Copy

By adding the debugger into your code you do not need to launch your program in a special way or remember to set breakpoints.

Importing the pdb module and running the pdb.set\_trace() function lets you begin your program as usual and run the debugger through its execution.

**Modifying Program Execution Flow**

The Python debugger lets you change the flow of your program at runtime with the jump command. This lets you skip forward to prevent some code from running, or can let you go backwards to run the code again.

We’ll be working with a small program that creates a list of the letters contained in the string sammy = "sammy":

letter\_list.py

def print\_sammy():

sammy\_list = []

sammy = "sammy"

for letter in sammy:

sammy\_list.append(letter)

print(sammy\_list)

if \_\_name\_\_ == "\_\_main\_\_":

print\_sammy()

Copy

If we run the program as usual with the python letter\_list.py command, we’ll receive the following output:

Output

['s']

['s', 'a']

['s', 'a', 'm']

['s', 'a', 'm', 'm']

['s', 'a', 'm', 'm', 'y']

With the Python debugger, let’s show how we can change the execution by first **jumping ahead** after the first cycle. When we do this, we’ll notice that there is a disruption of the [for loop](https://www.digitalocean.com/community/tutorials/how-to-construct-for-loops-in-python-3):

* python -m pdb letter\_list.py

> /Users/sammy/letter\_list.py(1)<module>()

-> def print\_sammy():

(Pdb) list

1 -> def print\_sammy():

2 sammy\_list = []

3 sammy = "sammy"

4 for letter in sammy:

5 sammy\_list.append(letter)

6 print(sammy\_list)

7

8 if \_\_name\_\_ == "\_\_main\_\_":

9 print\_sammy()

10

11

(Pdb) break 5

Breakpoint 1 at /Users/sammy/letter\_list.py:5

(Pdb) continue

> /Users/sammy/letter\_list.py(5)print\_sammy()

-> sammy\_list.append(letter)

(Pdb) pp letter

's'

(Pdb) continue

['s']

> /Users/sammy/letter\_list.py(5)print\_sammy()

-> sammy\_list.append(letter)

(Pdb) jump 6

> /Users/sammy/letter\_list.py(6)print\_sammy()

-> print(sammy\_list)

(Pdb) pp letter

'a'

(Pdb) disable 1

Disabled breakpoint 1 at /Users/sammy/letter\_list.py:5

(Pdb) continue

['s']

['s', 'm']

['s', 'm', 'm']

['s', 'm', 'm', 'y']

The above debugging session puts a break at line 5 to prevent code from continuing, then continues through code (along with pretty-printing some values of letter to show what is happening). Next, we use the jump command to skip to line 6. At this point, the variable letter is set equal to the string 'a', but we jump the code that adds that to the list sammy\_list. We then disable the breakpoint to proceed with the execution as usual with the continue command, so 'a' is never appended to sammy\_list.

Next, we can quit this first session and restart the debugger to **jump back** within the program to re-run a statement that has already been executed. This time, we’ll run the first iteration of the for loop again in the debugger:

> /Users/sammy/letter\_list.py(1)<module>()

-> def print\_sammy():

(Pdb) list

1 -> def print\_sammy():

2 sammy\_list = []

3 sammy = "sammy"

4 for letter in sammy:

5 sammy\_list.append(letter)

6 print(sammy\_list)

7

8 if \_\_name\_\_ == "\_\_main\_\_":

9 print\_sammy()

10

11

(Pdb) break 6

Breakpoint 1 at /Users/sammy/letter\_list.py:6

(Pdb) continue

> /Users/sammy/letter\_list.py(6)print\_sammy()

-> print(sammy\_list)

(Pdb) pp letter

's'

(Pdb) jump 5

> /Users/sammy/letter\_list.py(5)print\_sammy()

-> sammy\_list.append(letter)

(Pdb) continue

> /Users/sammy/letter\_list.py(6)print\_sammy()

-> print(sammy\_list)

(Pdb) pp letter

's'

(Pdb) disable 1

Disabled breakpoint 1 at /Users/sammy/letter\_list.py:6

(Pdb) continue

['s', 's']

['s', 's', 'a']

['s', 's', 'a', 'm']

['s', 's', 'a', 'm', 'm']

['s', 's', 'a', 'm', 'm', 'y']

In the debugging session above, we added a break at line 6, and then jumped back to line 5 after continuing. We pretty-printed along the way to show that the string 's' was being appended to the list sammy\_list twice. We then disabled the break at line 6 and continued running the program. The output shows two values of 's' appended to sammy\_list.

Some jumps are prevented by the debugger, especially when jumping in and out of certain flow control statements that are undefined. For example, you cannot jump into functions before arguments are defined, and you cannot jump into the middle of a try:except statement. You also cannot jump out of a finally block.

The jump statement with the Python debugger allows you to change the execution flow while debugging a program to see whether flow control can be modified to different purposes or to better understand what issues are arising in your code.

**Table of Common pdb Commands**

Here is a table of useful pdb commands along with their short forms to keep in mind while working with the Python debugger.

| **Command** | **Short form** | **What it does** |
| --- | --- | --- |
| args | a | Print the argument list of the current function |
| break | b | Creates a breakpoint (requires parameters) in the program execution |
| continue | c or cont | Continues program execution |
| help | h | Provides list of commands or help for a specified command |
| jump | j | Set the next line to be executed |
| list | l | Print the source code around the current line |
| next | n | Continue execution until the next line in the current function is reached or returns |
| step | s | Execute the current line, stopping at first possible occasion |
| pp | pp | Pretty-prints the value of the expression |
| quit or exit | q | Aborts the program |
| return | r | Continue execution until the current function returns |

You can read more about the commands and working with the debugger from the [Python debugger documentation](https://docs.python.org/3/library/pdb.html).

**Conclusion**

Debugging is an important step of any software development project. The Python debugger pdb implements an interactive debugging environment that you can use with any of your programs written in Python.

With features that let you pause your program, look at what values your variables are set to, and go through program execution in a discrete step-by-step manner, you can more fully understand what your program is doing and find bugs that exist in the logic or troubleshoot known issues.