Debugging in Python

Posted on

As a programmer, one of the first things that you need for serious program development is a debugger.

Python has a debugger, which is available as a module called pdb (for "Python DeBugger", naturally!). Unfortunately, most discussions of pdb are not very useful to a Python newbie — most are very terse and simply rehash the description of pdb in the Python library reference manual. The discussion that I have found most accessible is in the first four pages of Chapter 27 of the Python 2.1 Bible.

So here is my own personal gentle introduction to using pdb. It assumes that you are not using any IDE — that you're coding Python with a text editor and running your Python programs from the command line.

Some Other Debugger Resources

- For information on the IDLE interactive debugger, see the IDLE documentation
- For information on the Wing IDE debugger, see the Wing IDE documentation. There is a chapter on the Wing debugger, and another chapter on advanced debugging techniques which covers debugging code launched outside of Wing and/or on another host. *Thanks to Stephan Deibel of Wing for updating this information.*

Getting started — pdb.set_trace()

To start, I'll show you the very simplest way to use the Python debugger.

1. Let's start with a simple program, epdb1.py.

```
# epdb1.py -- experiment with the Python debugger, pdb
a = "aaa"
b = "bbb"
c = "ccc"
final = a + b + c
print final
```

2. Insert the following statement at the beginning of your Python program. This statement imports the Python debugger module, pdb.

```
import pdb
```

3. Now find a spot where you would like tracing to begin, and insert the following code:

```
pdb.set_trace()
```

So now your program looks like this.

```
# epdb1.py -- experiment with the Python debugger, pdb
import pdb
a = "aaa"
pdb.set_trace()
b = "bbb"
c = "ccc"
final = a + b + c
print final
```

4. Now run your program from the command line as you usually do, which will probably look something like this:

```
PROMPT> python epdb1.py
```

When your program encounters the line with pdb.set_trace() it will start tracing. That is, it will (1) stop, (2) display the “current statement” (that is, the line that will execute next) and (3) wait for your input. You will see the pdb prompt, which looks like this:

```
(Pdb)
```

Execute the next statement… with “n” (next)

At the (Pdb) prompt, press the lower-case letter “n” (for “next”) on your keyboard, and then press the ENTER key. This will tell pdb to execute the current statement. Keep doing this — pressing “n”, then ENTER.

Eventually you will come to the end of your program, and it will terminate and return you to the normal command prompt.

Congratulations! You’ve just done your first debugging run!

Repeating the last debugging command… with ENTER

This time, do the same thing as you did before. Start your program running. At the (Pdb) prompt, press the lower-case letter “n” (for “next”) on your keyboard, and then press the ENTER key.

But this time, after the first time that you press “n” and then ENTER, don’t do it any more. Instead, when you see the (Pdb) prompt, just press ENTER. You will notice that pdb continues, just as if you had pressed “n”. So this is Handy Tip #1:

> *If you press ENTER without entering anything, pdb will re-execute the last command that you gave it.*

In this case, the command was “n”, so you could just keep stepping through the program by pressing ENTER.

Notice that as you passed the last line (the line with the “print” statement), it was executed and you saw the output of the print statement (“aaabbbccc”) displayed on your screen.

Quitting it all… with “q” (quit)

The debugger can do all sorts of things, some of which you may find totally mystifying. So the most important thing to learn now — before you learn anything else — is how to quit debugging!

It is easy. When you see the (Pdb) prompt, just press “q” (for “quit”) and the ENTER key. Pdb will quit and you will be back at your command prompt. Try it, and see how it works.

Printing the value of variables… with “p” (print)

The most useful thing you can do at the (Pdb) prompt is to print the value of a variable. Here’s how to do it.

When you see the (Pdb) prompt, enter “p” (for “print”) followed by the name of the variable you want to print. And of course, you end by pressing the ENTER key.

Note that you can print multiple variables, by separating their names with commas (just as in a regular Python “print” statement). For example, you can print the value of the variables a, b, and c this way:

```
p a, b, c
```

When does pdb display a line?

Suppose you have progressed through the program until you see the line

```
final = a + b + c
```

and you give pdb the command

```
p final
```

You will get a NameError exception. This is because, although you are seeing the line, it has not yet executed. So the **final** variable has not yet been created.

Now press “n” and ENTER to continue and execute the line. Then try the “p final” command again. This time, when you give the command “p final”, pdb will print the value of **final**, which is “aaabbbccc”.

Turning off the (Pdb) prompt… with “c” (continue)

You probably noticed that the “q” command got you out of pdb in a very crude way — basically, by crashing the program.

If you wish simply to stop debugging, but to let the program continue running, then you want to use the “c” (for “continue”) command at the (Pdb) prompt. This will cause your program to continue running normally, without pausing for debugging. It may run to completion. Or, if the **pdb.set_trace()** statement was inside a loop, you may encounter it again, and the (Pdb) debugging prompt will appear once more.

Seeing where you are… with “l” (list)

As you are debugging, there is a lot of stuff being written to the screen, and it gets really hard to get a feeling for where you are in your program. That’s where the “l” (for “list”) command comes in. (Note that it is a lower-case “L”, not the numeral “one” or the capital letter “I”.)

“l” shows you, on the screen, the general area of your program’s souce code that you are executing. By default, it lists 11 (eleven) lines of code. The line of code that you are about to execute (the “current line”) is right in the middle, and there is a little arrow “–>” that points to it.

So a typical interaction with pdb might go like this

- The pdb.set_trace() statement is encountered, and you start tracing with the (Pdb) prompt
- You press “n” and then ENTER, to start stepping through your code.
- You just press ENTER to step again.
- You just press ENTER to step again.
- You just press ENTER to step again. etc. etc. etc.
- Eventually, you realize that you are a bit lost. You’re not exactly sure where you are in your program any more. So…

- You press “l” and then ENTER. This lists the area of your program that is currently being executed.
- You inspect the display, get your bearings, and are ready to start again. So….
- You press “n” and then ENTER, to start stepping through your code.
- You just press ENTER to step again.
- You just press ENTER to step again. etc. etc. etc.

Stepping into subroutines… with “s” (step into)

Eventually, you will need to debug larger programs — programs that use subroutines. And sometimes, the problem that you’re trying to find will lie buried in a subroutine. Consider the following program.

```
# epdb2.py -- experiment with the Python debugger, pdb
import pdb

def combine(s1,s2):      # define subroutine combine, which...
    s3 = s1 + s2 + s1    # sandwiches s2 between copies of s1, ...
    s3 = '"' + s3 +'"'   # encloses it in double quotes,...
    return s3            # and returns it.
```

```
a = "aaa"
pdb.set_trace()
b = "bbb"
c = "ccc"
final = combine(a,b)
print final
```

As you move through your programs by using the "n" command at the (Pdb) prompt, you will find that when you encounter a statement that invokes a subroutine — the final = combine(a,b) statement, for example — pdb treats it no differently than any other statement. That is, the statement is executed and you move on to the next statement — in this case, to print final.

But suppose you suspect that there is a problem in a subroutine. In our case, suppose you suspect that there is a problem in the combine subroutine. What you want — when you encounter the final = combine(a,b) statement — is some way to step into the combine subroutine, and to continue your debugging inside it.

Well, you can do that too. Do it with the "s" (for "step into") command.

When you execute statements that do not involve function calls, "n" and "s" do the same thing — move on to the next statement. But when you execute statements that invoke functions, "s", unlike "n", will step into the subroutine. In our case, if you executed the

```
final = combine(a,b)
```

statement using "s", then the next statement that pdb would show you would be the first statement in the combine subroutine:

```
def combine(s1,s2):
```

and you will continue debugging from there.

Continuing… but just to the end of the current subroutine… with "r" (return)

When you use "s" to step into subroutines, you will often find yourself trapped in a subroutine. You have examined the code that you're interested in, but now you have to step through a lot of uninteresting code in the subroutine.

In this situation, what you'd like to be able to do is just to skip ahead to the end of the subroutine. That is, you want to do something like the "c" ("continue") command does, but you want just to continue to the end of the subroutine, and then resume your stepping through the code.

You can do it. The command to do it is "r" (for "return" or, better, "continue until return"). If you are in a subroutine and you enter the "r" command at the (Pdb) prompt, pdb will continue executing until the end of the subroutine. At that point — the point when it is ready to return to the calling routine — it will stop and show the (Pdb) prompt again, and you can resume stepping through your code.

You can do anything at all at the (Pdb) prompt …

Sometimes you will be in the following situation — You think you've discovered the problem. The statement that was assigning a value of, say, "aaa" to variable var1 was wrong, and was causing your program to blow up. It should have been assigning the value "bbb" to var1.

… at least, you're pretty sure that was the problem…

What you'd really like to be able to do, now that you've located the problem, is to assign "bbb" to var1, and see if your program now runs to completion without bombing.

It can be done!

One of the nice things about the (Pdb) prompt is that you can do anything at it — you can enter any command that you like at the (Pdb) prompt. So you can, for instance, enter this command at the (Pdb) prompt.

```
(Pdb) var1 = "bbb"
```

You can then continue to step through the program. Or you could be adventurous — use "c" to turn off debugging, and see if your program will end without bombing!

… but be a little careful!

[Thanks to Dick Morris for the information in this section.]

Since you can do anything at all at the (Pdb) prompt, you might decide to try setting the variable b to a new value, say "BBB", this way:

```
(Pdb) b = "BBB"
```

If you do, pdb produces a strange error message about being unable to find an object named '= "BBB" '. Why???

What happens is that pdb attempts to execute the pdb **b** command for setting and listing breakpoints (a command that we haven't discussed). It interprets the rest of the line as an argument to the **b** command, and can't find the object that (it thinks) is being referred to. So it produces an error message.

So how can we assign a new value to **b**? The trick is to start the command with an exclamation point (!).

```
(Pdb)!b = "BBB"
```

An exclamation point tells pdb that what follows is a Python statement, not a pdb command.

The End

Well, that's all for now. There are a number of topics that I haven't mentioned, such as help, aliases, and breakpoints. For information about them, try the online reference for pdb commands on the Python documentation web site. In addition, I recommend Jeremy Jones' article Interactive Debugging in Python in O'Reilly's Python DevCenter.

I hope that this introduction to pdb has been enough to get you up and running fairly quickly and painlessly. Good luck!

—Steve Ferg