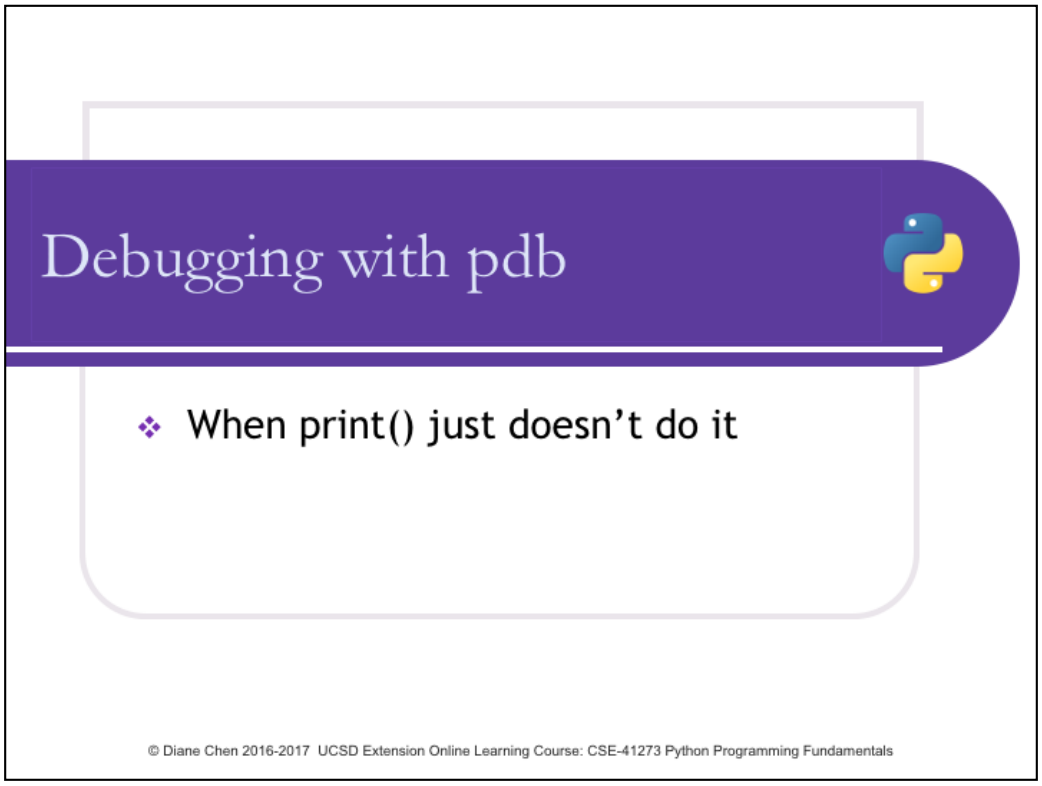
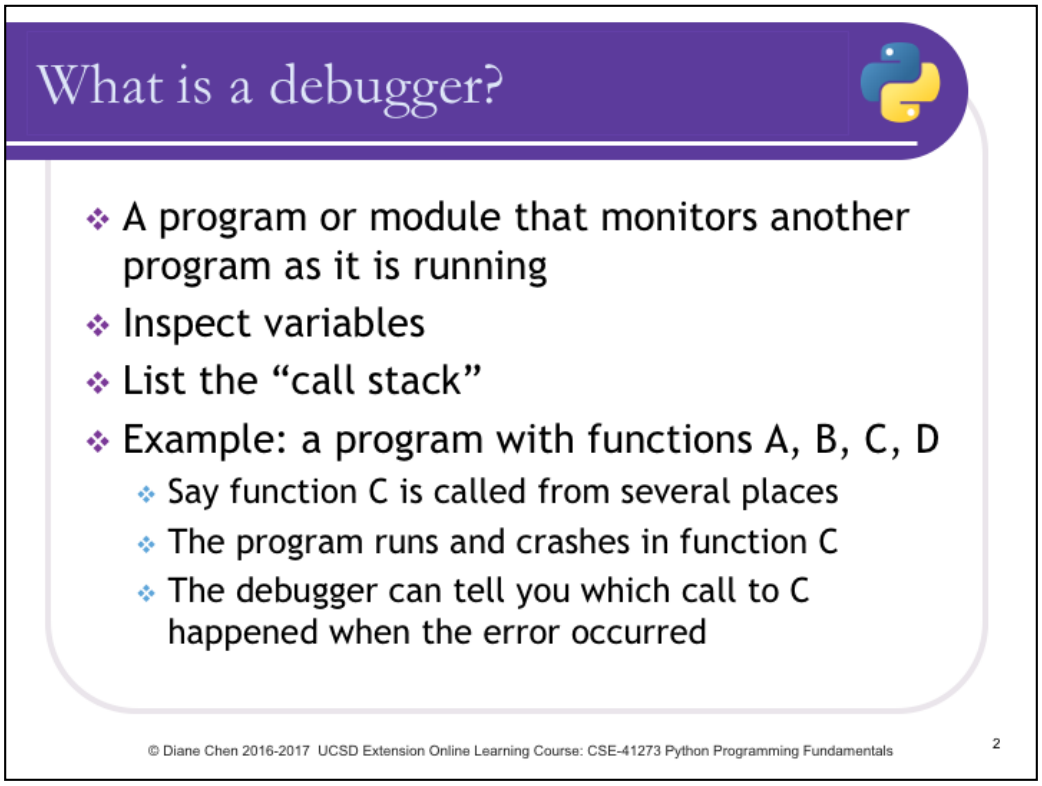


Now we'll go over the python debugger, pdb. With the simple programs that we are using in this class, most of the time if there is an error, you can add a print() statement or 2 and find the error immediately. But in real life, your programs will often be larger and more complex, involving multiple modules and many functions. In addition, the program might have its own output, and debug print statements can easily get lost in the normal output of the program. And then when you find the error, you have to go back through and remove your debug print statements, which is tedious and therefore error-prone. Using a debugger to investigate the error is often the best option.

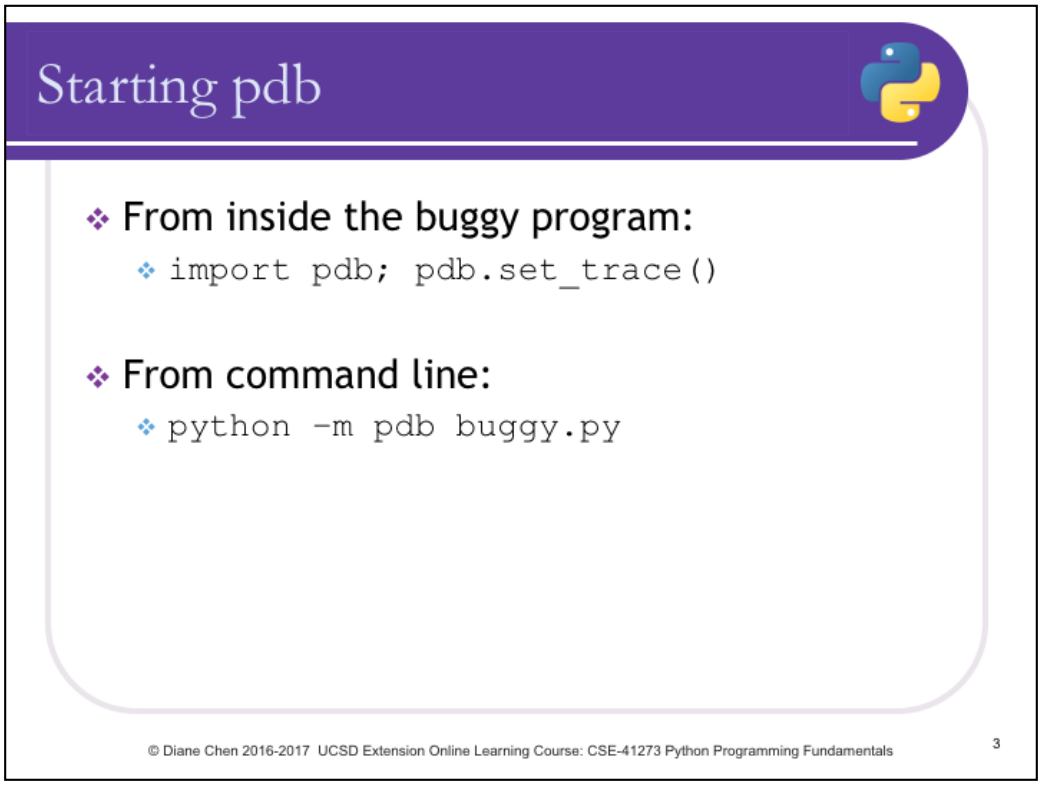


What is a debugger? A debugger is a program – or in Python's case, a module – that, in essence, takes over for the computer in running another program. As the program to be debugged is running – we often say it is "running inside the debugger", you can monitor what is happening in it.

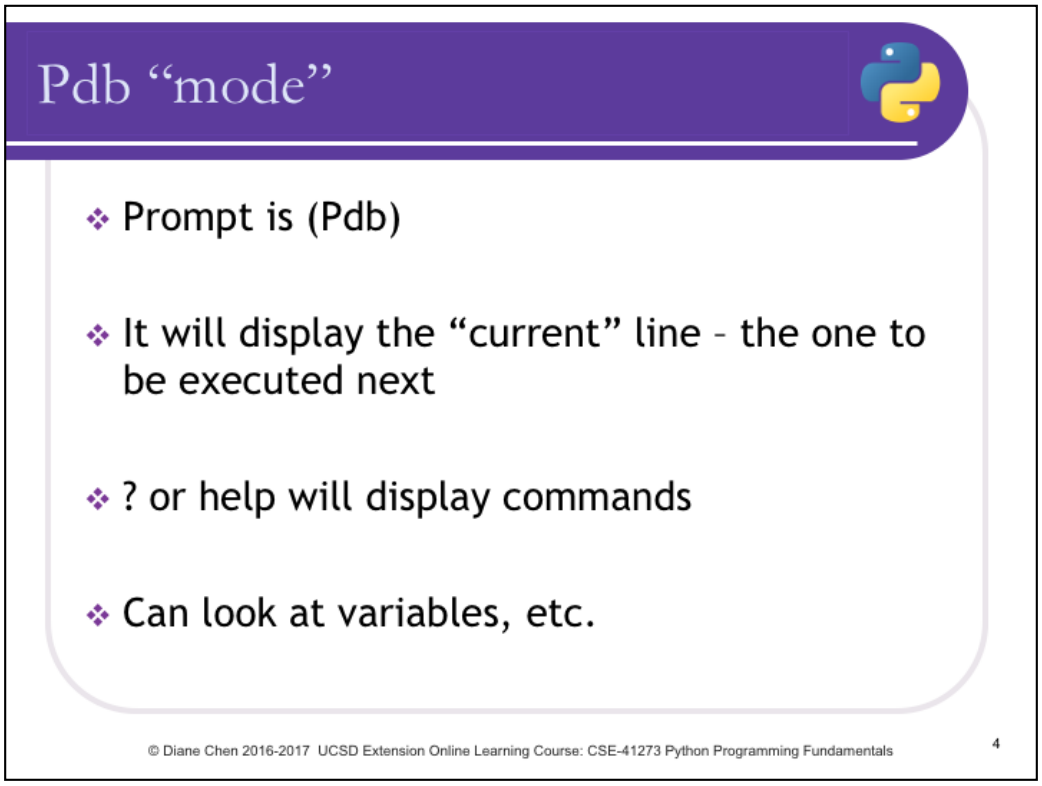
Using the debugger, you can execute one line of code at a time and inspect many things about the program, like variable names or a list of all the functions called to get to a particular location. This is referred to as the "call stack", because it is a last-in-first-out data structure.

For example, say we had a program with functions A, B, C and D. In it, function C is called at least once from each of the other functions. The program runs and gets an error in function C. With the debugger, we can find out what the call stack is and who called function C when it got the error. For example, maybe A called B, which called D, which then called C. You can examine the value of variables and determine why D called C. Perhaps the error is actually in function B, which called D with some bad data that was then passed to C, causing the crash. Or maybe there is a logic error in D which resulted in the call to C, when it shouldn't even have called C in this situation.

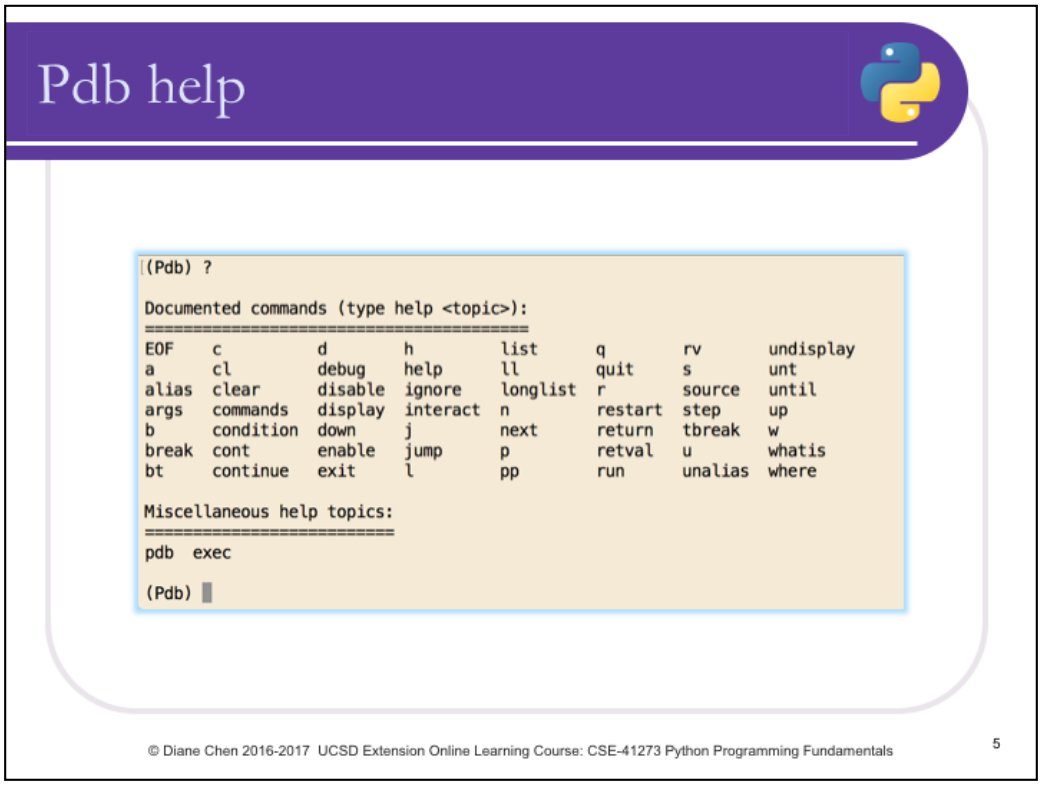
With the power of the debugger, we can look at all this information and find the cause of the error.



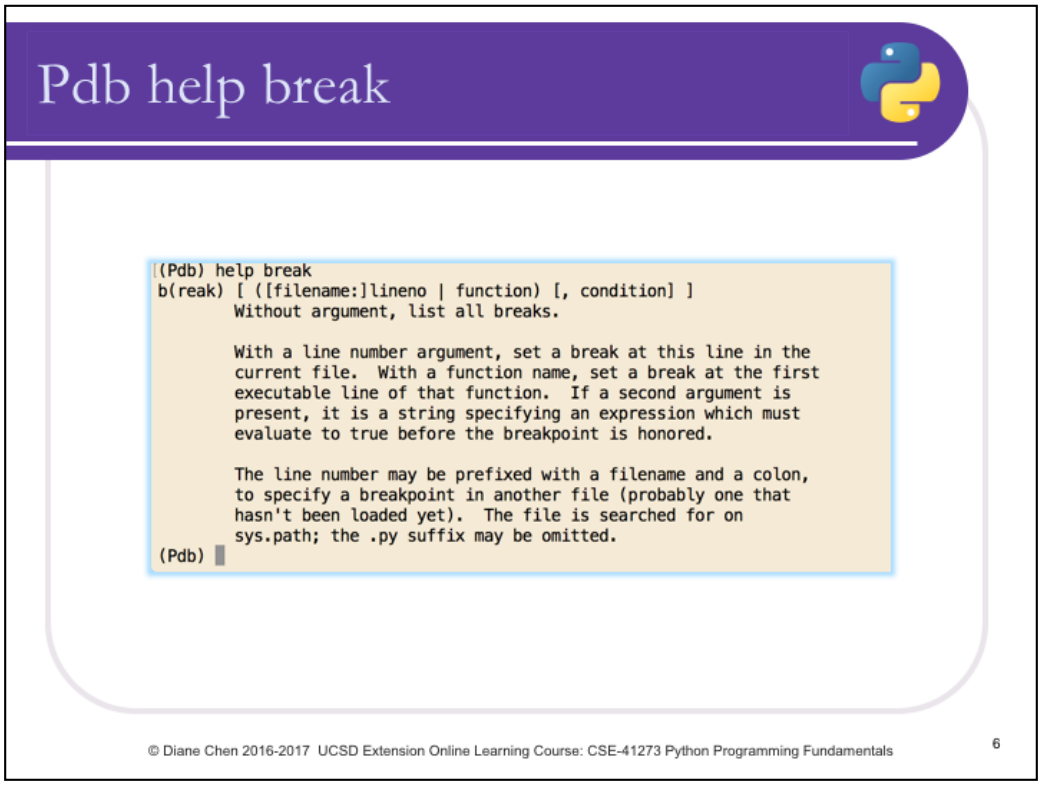
There are 2 ways to start pdb. The first way is to add the line shown here containing the statements import pbd and pdb.set_trace() in the program at the point where you want pdb to start. Perhaps you have an idea of the location where things have started to go wrong. This is the most common way that pdb is used. It can also be run from the command line, and pdb starts up right at the beginning of the program. I will go through an example of using pdb in a simple program as we go along in this lesson.



When you are in "pdb mode", it is just like being in an interactive shell. The prompt is pdb in parentheses. It will display the current line, where "current" means the line that is about to be executed next. You can always type in a question mark or "help" to get a list of the basic commands. You can also look at variables, list lines of the code and other commands we will go into here.

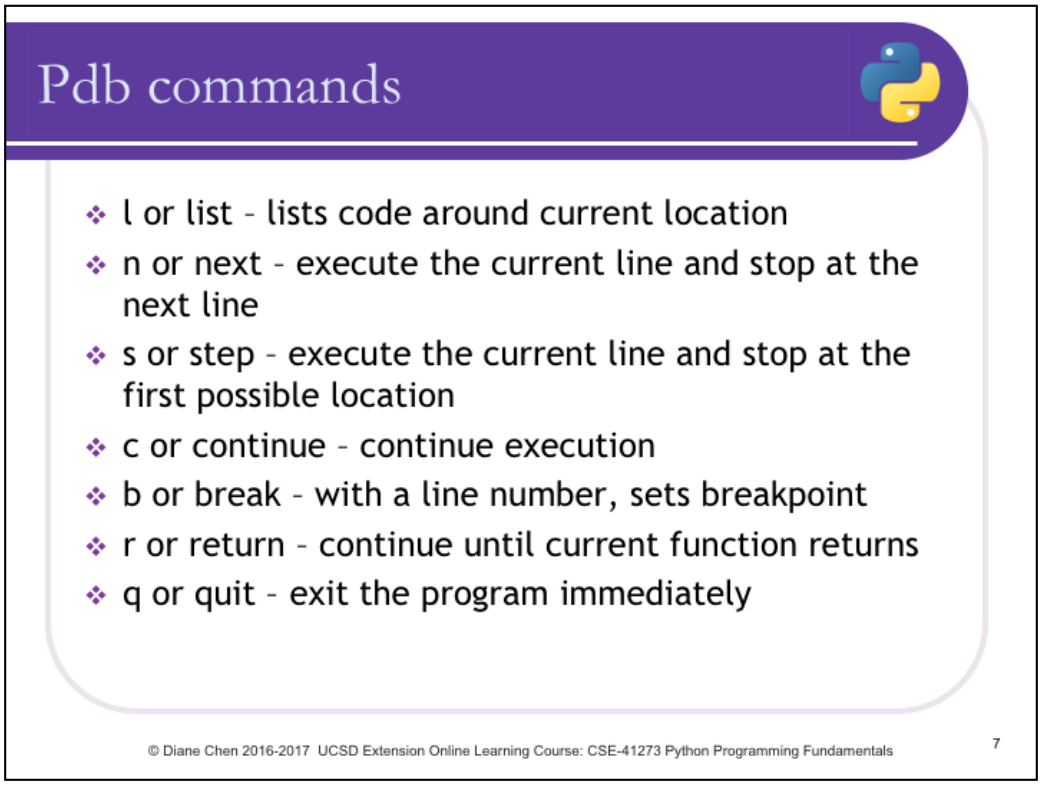


Here is a list of commands as displayed by pdb when you enter either the question mark or help. Some are duplicates, as many of the single letter commands are abbreviations of longer commands. For example, the letter c by itself means continue, but you can also type "cont" or the whole word continue.



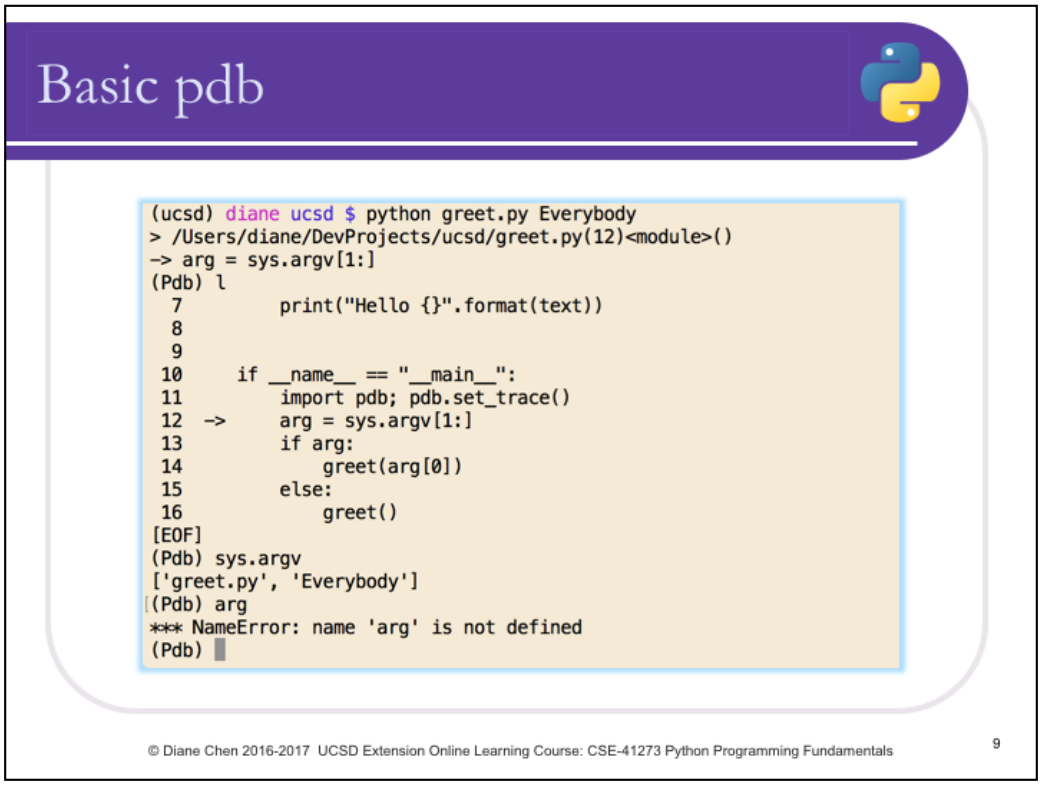
As an example, here is the output from “help break”, letting us know how to use the ‘b’ or ‘break’ command. It is used to set, list or modify breakpoints in the code. A breakpoint is a place in the code where pdb stops the program – well really it’s just paused – and puts you in pdb mode where you can inspect what is happening at that moment in the program.

So, when you are in pdb mode, you can set a breakpoint at a particular location in the code, and then enter 'c' to continue. The program will run until it reaches the breakpoint location, when it will stop and enter into pdb mode again so you can see what is happening at that point in the code.



Here are some useful Pdb commands. L or list will list lines of code around the current location. It prints 11 lines total; 5 lines before and after the current line. N or next will execute the current line and stop at the next line in the current function, or if the current line is a return, stop at the first opportunity in the calling function. S or step executes the current line, just like next, but it stops at the first opportunity – this means if the current line is a function call, it will "step into" the function and the current pointer will only advance to the first executable line in the function that is called. C or continue will cause execution to continue normally until finished or a breakpoint is reached. When you list lines of code, it shows the line numbers with the code. You can set another breakpoint with b or break followed by a line number. R or return causes execution to continue until the function returns and stops at the next line in the calling code. Q or quit will exit the program immediately

I'll just show how you can look at a program using our old friend greet.py in an early, simple version. Note that I have added the line "import pdb; pdb.set_trace()". This is one of those rare times where I put more than one Python statement on one line, which you can do by separating the statements with a semicolon. Because it is a line that is not going to be in production, it is OK to do that, especially since that makes it easier to remove later. We will just go through this simple program to show how pdb works.



Here I run greet.py with the argument "Everybody". The program stops immediately after it reaches the set_trace() call and goes into pdb mode.

The first line it displays is information on the module and the line number - 12 in this case - of the current execution pointer. This can be very helpful information if there are multiple module files used by the program. The code statement at the current execution pointer is shown on the next line, with the arrow on the left pointing to it to indicate this is the current line. I then typed 'l' for list and it displayed lines of code around this location. The line at the current execution pointer is indicated here also with an arrow.

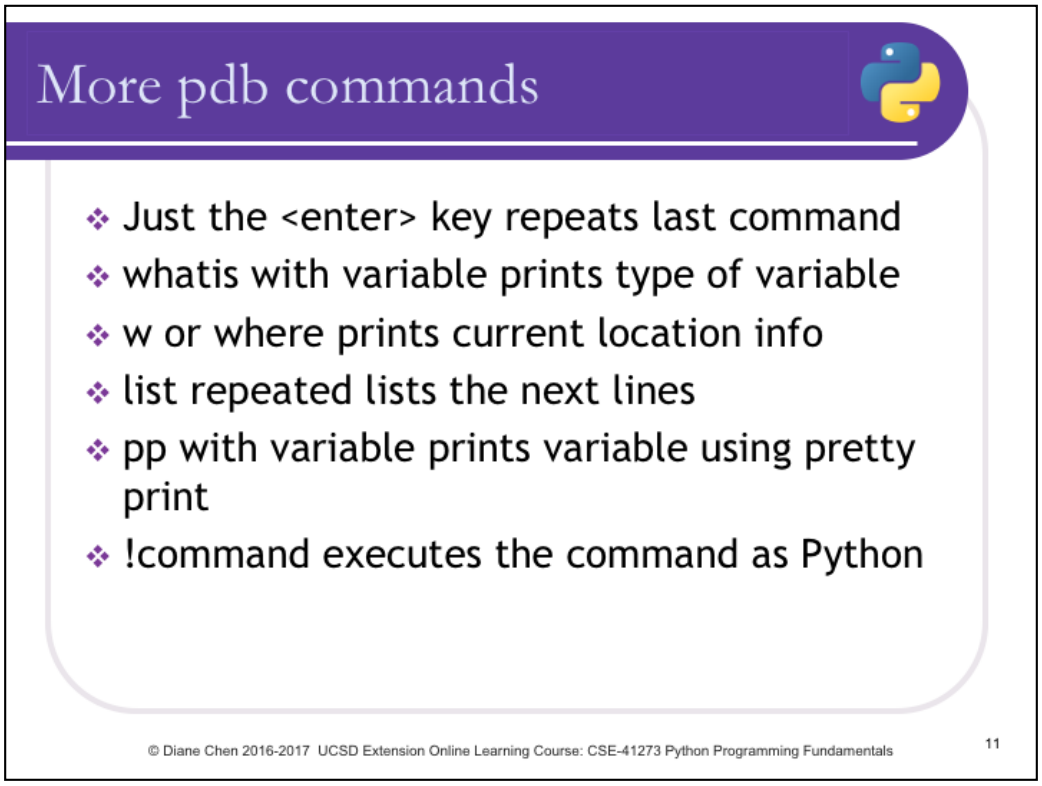
So you can imagine why it's important to keep your code readable with lines that are not too long, as it would be just that much more difficult to use the debugger if all the lines were really long and complicated. Or if there are multiple statements on a line, it is not possible to execute them separately in the debugger.

You can look at the value of variables by just typing the variable's name. But if you have a variable whose name conflicts with a pdb command, for example the single letter 'c', then you would need to type "p space c" to print it. Here we see the value of sys.argv; it contains 'greet.py' as the first element and 'Everybody' as the second element.

At this point, we can confirm that the statement at the current pointer has not been executed yet, because trying to print the value of the variable 'arg' gets an error because it does not exist yet.

Here we're still in pdb mode, I keep going in the program by typing n for next. Each time, it displays the module information and the line at the current execution pointer. Because I used 'n' for next instead of 's' for step, we don't see anything of what happens in the function greet.

So, the program prints out 'Hello Everybody' and then we have a line indicating there is a return. This is the return from the greet program back to the system when the program ends. It is telling us that this is what is going to happen next. A final 'n' for next finishes the return and the program is ended.



Here are some more pdb commands. Just typing the enter key by itself repeats the last command. This is handy as you are going through the code line by line using 'next' or 'step'.

The command 'whatis' when given with a variable name, tells you the type of the variable. 'w' or 'where' prints the current location information including the call stack.

If you repeat the l or list command, it will continue to list more lines of code. But whenever a statement is executed, the list command resets itself, so after any code is executed, 'list' will again list the lines of code around the new current execution pointer.

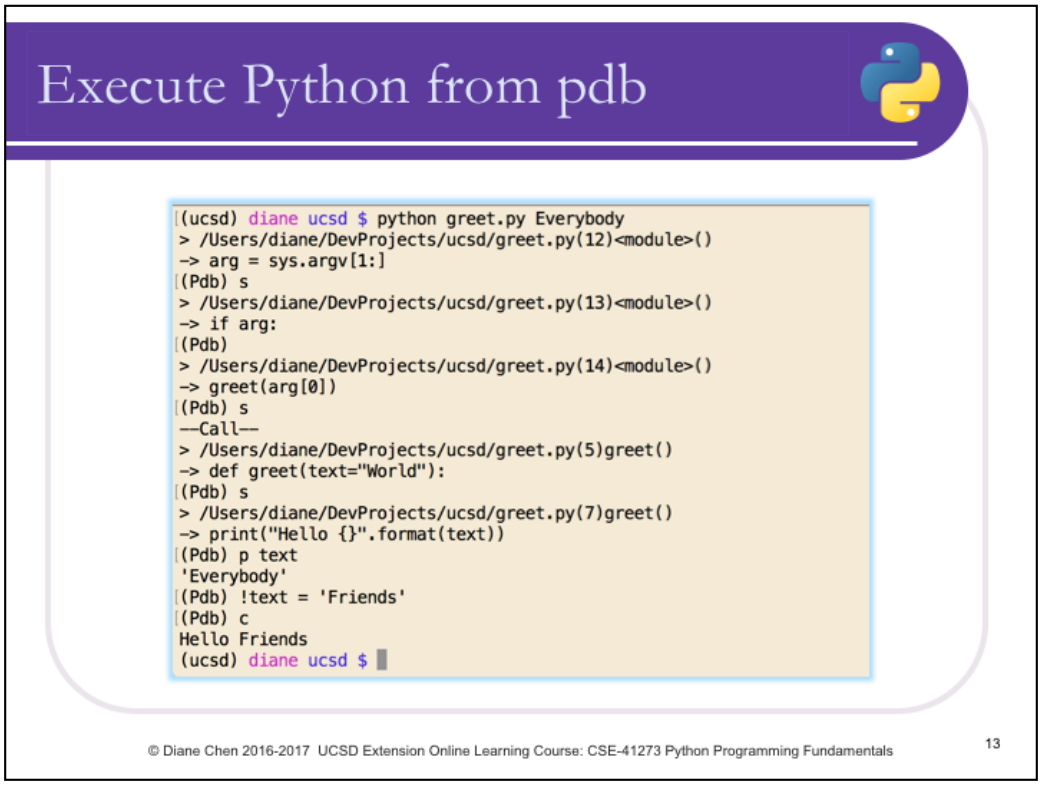
The command 'pp' with a variable will print the variable using pretty print. This is most useful when you have a dictionary or complicated object.

You can also execute a Python command from pdb by prefacing it with an exclamation mark.

Here I run the program just as I did before. The green '1' indicates the beginning, and at this point, everything is just like the last time. I enter 's' for step, and then the 'enter' key twice to repeat the 'step' command two times.

At the point of the green '2', this is where it is different from before. Because I am stepping into the function greet, pdb displays the line '--Call--' to tell me that a function call is occurring. Now we are at the start of the function greet, and 'list' shows the lines of code around the current execution pointer. 'step' goes to the next line, then 'enter' repeats it and it prints out 'Hello Everybody'.

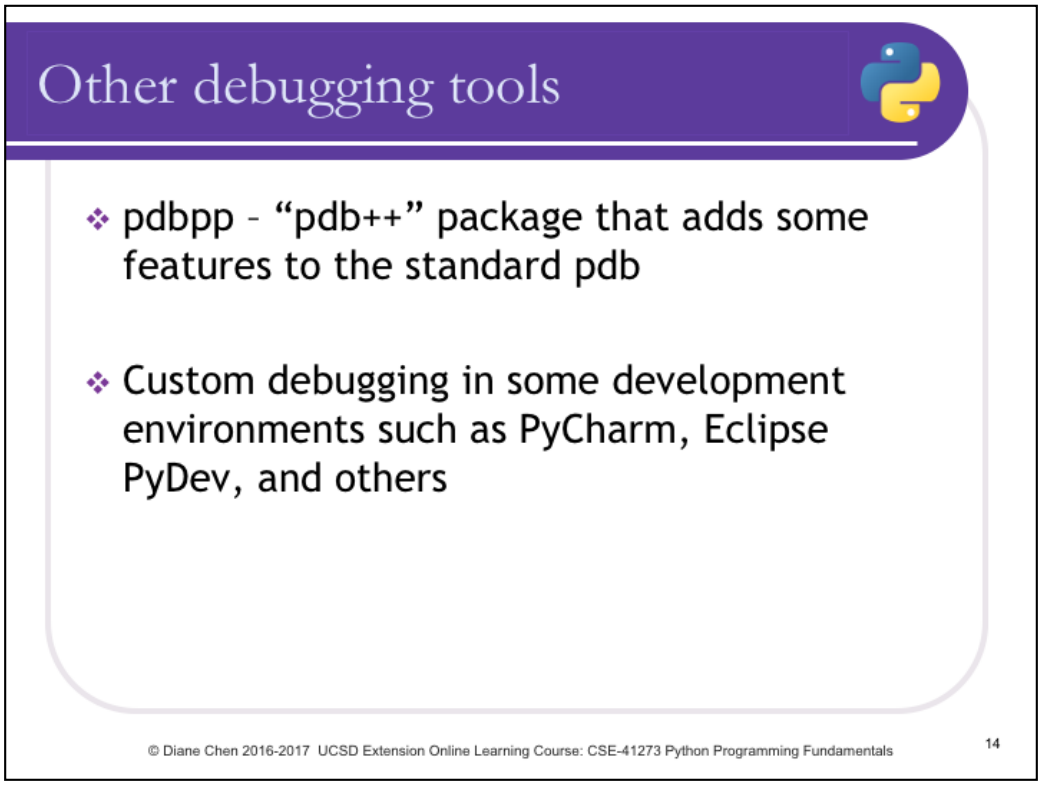
Then where the green '3' is, pdb displays the '--Return--' as the next thing to happen is the return from the function greet. I think it's a little unfortunate that the display makes it look like the next thing to be executed is the print statement again; you just have to remember that the '--Return--' indicates that the statement is finished and the function is returning. It can be confusing sometimes the way it displays things in this situation.



In this example, greet.py is run again with the argument 'Everybody'. Stepping through until the code is inside the greet function, I print out the value of 'text' and see that it is 'Everybody' as we would expect.

The next line sets the valule of 'text' to be 'Friends'. The exclamation point tells Python that the rest of the line is a statement to execute as Python instead of a command to pdb. Entering 'c' to continue execution results in the printing of 'Hello Friends' instead of 'Hello Everybody'. This is very useful when you think your problem is caused by a variable having a bad value. You can change the variable in pdb mode and see if that fixes the problem. If that does fix things, then you know you just need to work backwards to fix the cause of the variable getting bad data.

As long as your variable names do not conflict with pdb commands, you can leave off the exclamation point and pdb will know to execute the code like Python. But it's probably better to get in the habit of using the exclamation point to avoid confusion.



There is a package called pdb++ that adds some features to the standard pdb module. It can be installed for a project if needed. In addition, some development environments have their own custom debugging features.