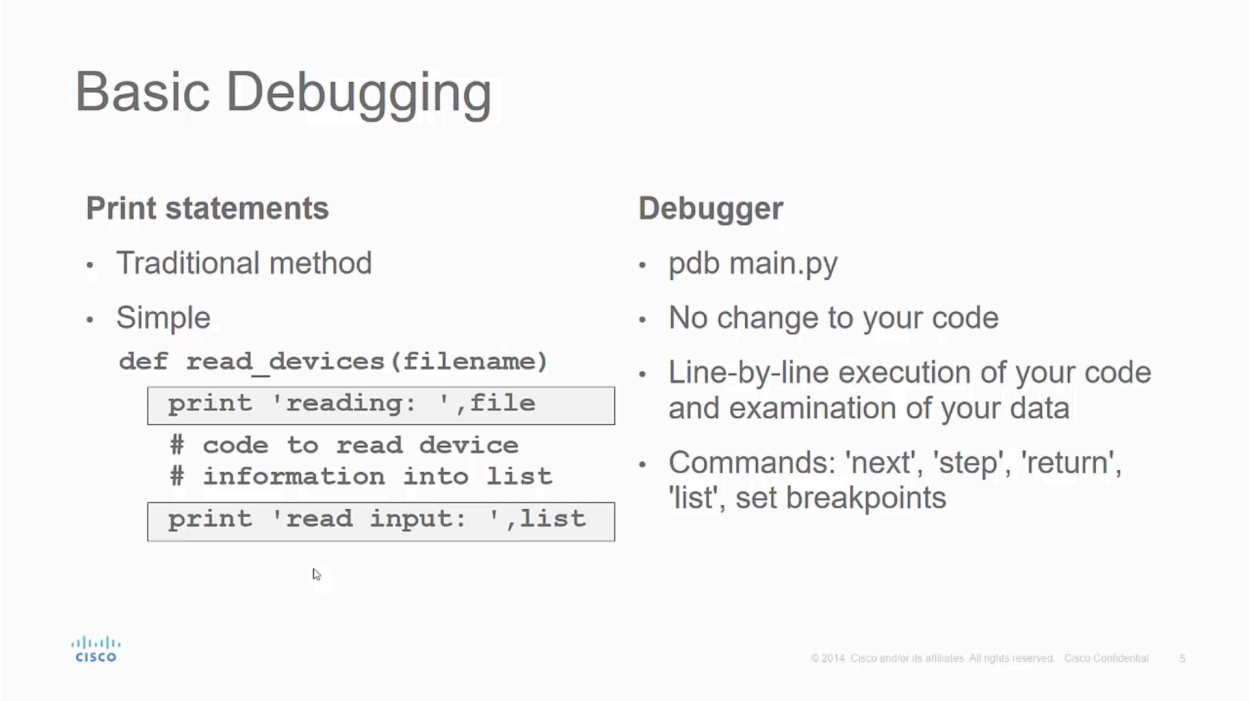


Debugging is done most frequently during the development of your application. Two of the most popular methods of debugging are:

- **Print statements:** When writing your application, often the easiest way to test is by inserting print statements liberally within the code.
  - Traditional Method
  - Simple

```
def read_devices(filen)
    print 'reading: ', file
    # code to read device
    # information into list
    print 'read input: ', list
```

- **Debugger:** Python, like other programming languages, provides a debugger, which allows you to examine the execution of your code without making any changes, such as adding print statements.
  - `pdb main.py`
  - No changes to your code
  - Line-by-line execution of your code and examination of your data
  - **Commands:** `next`, `step`, `return`, `list`, `set breakpoint`

Printing

When you create your application you may not have any temporary print statements however, during debugging of potential issues or just for error checking, you may find them helpful. Here are some guidelines for where and how you may use temporary print statements to help with debugging your application:

- Print important events: read input file, created device objects, connected to a device, got interface data from a device, set interface description for a device, and so on.
- Print important data associated with events: contents of device file just read, attributes of device object just created, connection credentials, and so on.
- Print result codes from calls to external functions: from connecting to a device, making a database request, and so on.
- Print entry and exit from your own functions, in addition to printing the values of all the parameters.

Printing may only take you so far, however, and further inspection of the behavior of your application may require the use of a debugger.

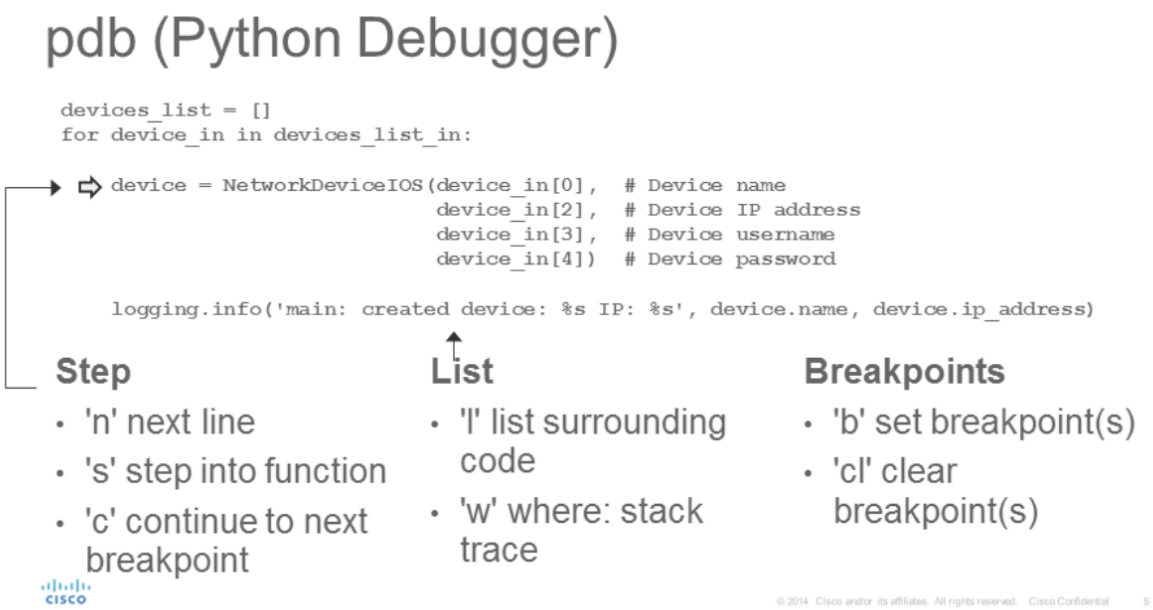
Debugger

The process of using a debugger involves starting your application in a special debug environment. Started in this way, Python allows you to stop execution of your program in order to examine data, and to step through your program in order to determine the order and execution path your code is taking. Are your variables set to the values you think they should be? Is your for loop executing as you were expecting? A debugger will help you answer these questions.

Debugging commands fall into the following general categories:

- **Step-wise execution**: For the execution of lines one at a time, from your current line.
- **Code listing**: For looking at the lines of code surrounding your current line, and looking at the function-call history (also known as a 'stack trace').
- **Breakpoints**: For 'breaking' (stopping execution) at specific lines in your code.
- **Variables**: Looking at the values of variables, while stopped at your breakpoint.

Pdb (Python Debugger)



The Python debugger `pdb` allows you to perform fine-grained debugging of your application.

To run the debugger, you simply invoke `pdb` rather than `python`:

```
$ pdb main.py
> /var/local/PyNE/labs/mod11/S11-2-logging/main.py(1)<module>()
-> import logging
```

It is also possible to run the debugger by specifying `pdb` as the module, and your application name as the script to use as input to the module:

```
$ python -m pdb main.py
> /var/local/PyNE/labs/mod11/S11-2-logging/main.py(1)<module>()
-> import logging
```

To step through code after starting the debugger, or after stopping at (also known as 'hitting') a breakpoint, you can use the following commands:

- `n`: Next – Execute the next line
- `s`: Step – Execute the next line, stepping *into* the function or method if applicable
- `c`: Continue – Continue on to the next breakpoint
- `r`: Return – Continue until the current function is about to return to the calling code.

You will likely not want to step through every line of your application, when attempting to debug a specific problem. In order to allow the execution of your application to continue up until a specific line of code, you will want to set a *breakpoint*. A breakpoint causes the debugger to halt execution of your application at a specific line, waiting for you to enter further instructions, such as to display variables, or to step through the code 1 line at a time.

Breakpoint commands are as follows:

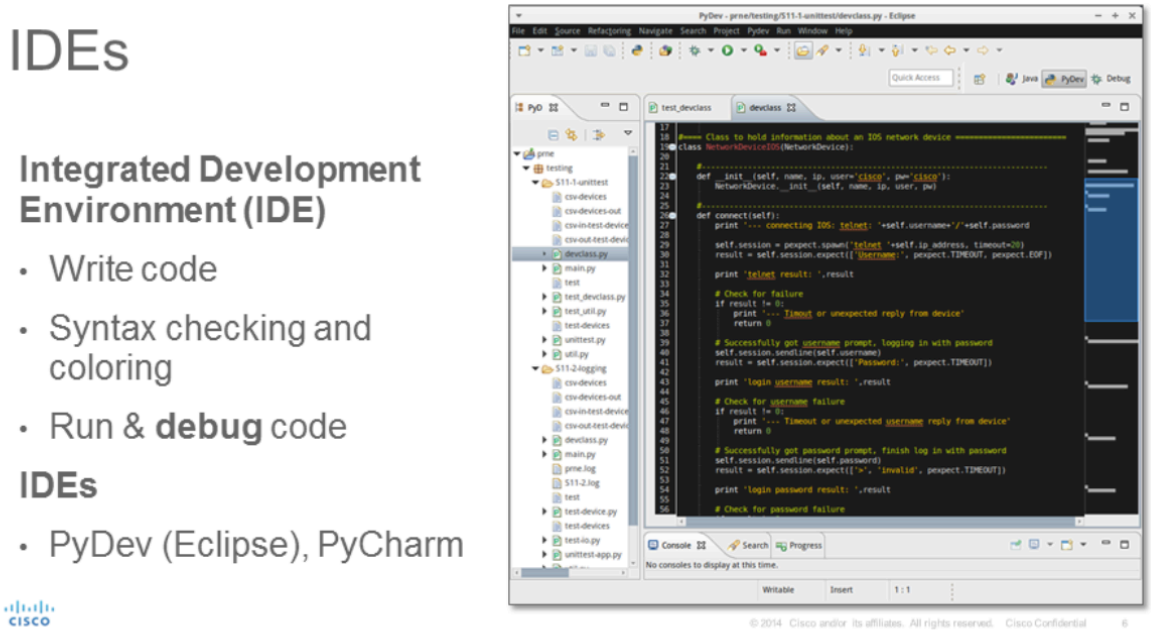
- `b`: Break– Set a breakpoint at the specific file and line number. It is also possible to set a *conditional* breakpoint, which breaks only if the provided condition is met. Specifying only the `b` with no arguments will list all breakpoints.
- `cl`: Clear – Clear specific breakpoints.

Variables

When your application stops at a breakpoint, you will want to display information about the current state of your application: the line where you stopped, the calling modules, and the variables that are relevant now for your code. The following commands are available for this purpose:

- `l`: List – List the lines of code surrounding the location where the application is currently stopped. Notice that the current line at which your application is stopped has an arrow ('->') pointing to it.
- `w`: Where – List the stack trace of calling modules and functions that were called to get to a specific line of code.
- `p`: Print `–` Print the value of the variable or expression. Note that since the Python debugger is capable of interpreting code, you can list the variable name, and it will be printed as if you were running inside the Python interpreter.

Integrated Development Environments (IDEs)



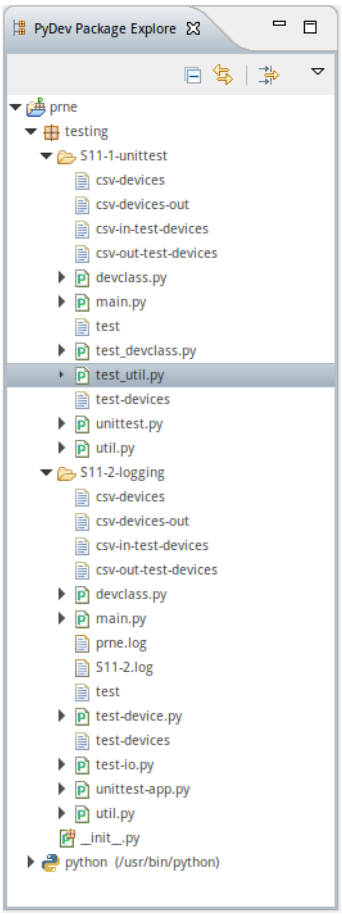
IDEs provide a rich user interface and a significant functionality improvement over using the command line and general-purpose editors. IDEs integrate the management of projects, editing of software, and the running and debugging of applications, all in a powerful user interface.

The following features of IDEs make the development of applications much easier:

Organizing

An IDE organizes your files and folders, your modules and packages, and presents them in an intuitive explorer-type format, for easy reference and access.

The example shows the Eclipse IDE with the 'PyDev' plugin, which provides Python-specific functionality. The figure is of the PyDev Package Explorer, showing the `prne` project, a package called 'testing', with Python modules and various other files.



Coding

An IDE provides editors that are language-aware, in that they are able to analyze the code you write, highlighting and coloring keywords, strings, comments, and so on; maintaining consistent and automatic indentation as you type, and checking for syntax errors.

The following example shows the PyDev Python editor within Eclipse. You can see the different coloring highlighting comments, strings, indentation lines, and numeric values. Line numbers, which can be handy during debugging, are shown on the left.

```
17
18 #==== Class to hold information about an IOS network device =======================
19 class NetworkDeviceIOS(NetworkDevice):
20
21     #-----------------------------------------------------------------------------
22     def __init__(self, name, ip, user='cisco', pw='cisco'):
23         NetworkDevice.__init__(self, name, ip, user, pw)
24
25     #-----------------------------------------------------------------------------
26     def connect(self):
27         print '--- connecting IOS: telnet: '+self.username+'/'+self.password
28
29         self.session = pexpect.spawn('telnet '+self.ip_address, timeout=20)
30         result = self.session.expect(['Username:', pexpect.TIMEOUT, pexpect.EOF])
31
32         print 'telnet result: ',result
33
34         # Check for failure
35         if result != 0:
36             print '--- Timout or unexpected reply from device'
37             return 0
38
39         # Successfully got username prompt, logging in with password
40         self.session.sendline(self.username)
41         result = self.session.expect(['Password:', pexpect.TIMEOUT])
42
43         print 'login username result: ',result
44
45         # Check for username failure
46         if result != 0:
47             print '--- Timeout or unexpected username reply from device'
48             return 0
49
50         # Successfully got password prompt, finish log in with password
```

Debugging

In the figure above, the very left-most column shows breakpoints – the green dot identifies the line as having a breakpoint, and the check mark beside it indicates that the breakpoint is set correctly and is active.

Overall, IDEs are very useful tools for writing any type of software, Python included, and if your project is going to be of a moderate size or bigger, it would be wise to consider using such a tool.