5.3 Pdb

The Pdb module gives a richer debugging environment than the shell debugger that we discussed in section 5.2. Again, you select Run Module from the Run menu of IDLE to load your program into the shell's memory. Then at the shell prompt you need two lines:

```
>>> import pdb
>>> pdb.run("main()")
```

Don't forget the parentheses after main; you want to call function main, not just evaluate it. The system responds with

```
> <string>(1)<module>()->None
(Pdb)
```

At this point you have access to a number of pdb commands. You type these commands into the shell at the (Pdb) prompt; the system responds in the shell. Here is a list of the most useful pdb commands; most can be abbreviated to their first letter:

s(tep) Execute the current line, stepping into function calls. This is like the Step button in the shell debugger.

n(ext) Execute the current line, stepping over function calls. This is like the Over button in the shell debugger.

r(eturn) Execute the rest of the current function call, continuing until it returns. This is like the Out button in the shell debugger.

c(ontinue) Continue to execute the rest of the program, not stopping unless a breakpoint is encountered. This is like the Go button in the shell debugger.

l(ist) List the portion of the source code around the current instruction.

list first, last List the portion of the source code between lines first and last.

p expression The expression is evaluated and its value is printed. Note that the command is *p*, not *print*. The command `p (x, y,z)` is the usual way to get the current values of variables x, y and z.

b(reak) function Set a *breakpoint* at the first line of the named function.

b(reak) n Set a *breakpoint* at line number n.

b(reak) function, condition Set a conditional breakpoint at the start of the named function.

b(reak) n, condition Set a conditional breakpoint at line number n.

disable n Disable breakpoint number n (not the breakpoint at line number n).

enable n Re-enable the disabled breakpoint number n

condition n Removes any condition on breakpoint n

condition n c Sets the condition on breakpoint n to c.

cl(ear) Removes all breakpoints.

cl(ear) n Removes breakpoint number n

Return-key On a Return the system re-runs the last pdb command it executed. So to step through a program you need to give one s or step command, followed by a series of <Return> keys.

The only thing really new here is the notion of a *breakpoint*. If you are executing more than one line of code with **next**, **return**, or **continue** commands, execution will always stop when a breakpoint is reached. Listings of the code will indicate breakpoints with a B. Note that the breakpoints are numbered from 1 as you create them. Some of the commands require you to know the number of a breakpoint; you can always look back in the shell to find the number that was given when you created it.

As in section 1.5, we will run the following program through the pdb debugger:

```
def IsPrime(x):
    for d in range(2, x):
        if x%d == 0:
            return False
    return True

def PrintPrimes(N):
    for num in range(2,N+1):
        if IsPrime(num):
            print( num )

def main():
    PrintPrimes(100)

main()
```

Program 5.3.1: Finding Prime Numbers

After running the program in the shell, we type

```
>>> import pdb
>>> pdb.run("main()")
```

and the system responds:

```
> <string>(1)<module>()
(Pdb)
```

You need to actually step into the program before doing anything with it. So we type

```
(Pdb) step
```

and the system responds with

```
--Call--
> c:\users\bob geitz\...\primes.py(12)main()
-> def main():
(Pdb)
```

The program has 15 lines (to see this, put the cursor on the last line of the program in IDLE; the line number appears at the bottom right of the IDLE window). To get a listingof it in the shell, so we can avoid bouncing back and forth between the shell and IDLE, we ask the first 15 lines. We type:

```
(Pdb) list 1, 15
```

and the system responds:

```
  1          def IsPrime(x):
  2              for d in range(2, x):
  3                  if x%d == 0:
  4                      return False
  5              return True
  6
  7          def PrintPrimes(N):
  8              for num in range(2,N+1):
  9                  if IsPrime(num):
 10                      print num
 11
 12  ->        def main():
 13              PrintPrimes(100)
 14
 15          main()
(Pdb)
```

Note the arrow at line 12, which indicates the current instruction, in this case the start of function main. Let's do a few more steps. We type

```
(Pdb) s
```

and the system responds

```
> c:\users\bob geitz\...\primes.py(13)main()
-> PrintPrimes(100)
(Pdb)
```

We could step again by just pressing the <Return>-key, but the output is easier to read if we at least use the `s` of `step`:

```
(Pdb) s
```

The system responds:

```
--Call--
> c:\users\bob geitz\...\primes.py(7)PrintPrimes()
-> def PrintPrimes(N):
(Pdb)
```

One more time: we type

```
(Pdb) s
```

and the system responds

```
> c:\users\bob geitz\...\primes.py(8)PrintPrimes()
-> for num in range(2,N+1):
(Pdb)
```

To see the current value of N we type

```
(Pdb) p N
```

and the system responds

```
100
```

If we try to find the value of variable num from PrintPrimes:

```
(Pdb) p num
```

we get an error message from the debugger because the **for**-loop in which num is defined has not yet started executing.

We could continue stepping through the program, but let's take advantage of the fact that pdb has breakpoints. To watch the program determine if the number 65 is prime, we can set a breakpoint just inside the **for**-loop in PrintPrimes when the value of num becomes 65. We type:

```
(Pdb) break 9, num==65
```

and the system responds:

```
Breakpoint 1 at c:\users\bob geitz\...\primes.py:9
(Pdb)
```

We can run the execution of the program to this point with

```
(Pdb) continue
```

The program runs, printing out prime numbers as it finds them and stops at the breakpoint when our condition is met:

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
> c:\users\bob geitz\...\primes.py(9)PrintPrimes()
-> if IsPrime(num):
(Pdb)
```

We can see the current values of N and num with

```
(Pdb) p (N, num)
```

and the system prints them:

```
(100, 65)
(Pdb)
```

We will now step 4 times and watch the action of the code. To the first step the system responds:

```
--Call--
> c:\users\bob geitz\...\primes.py(1)IsPrime()
-> def IsPrime(x):
(Pdb)
```

On the next step we go inside IsPrime():

```
> c:\users\bob geitz\...\primes.py(2)IsPrime()
-> for d in range(2, x):
(Pdb)
```

Our first value of d is 2, so when we step

```
> c:\users\bob geitz\...\primes.py(3)IsPrime()
-> if x%d == 0:
(Pdb)
```

the result is False and we go back and increment d.

```
> c:\users\bob geitz\...\primes.py(2)IsPrime()
-> for d in range(2, x):
(Pdb)
```

We know that when d gets to 5 it will divide evenly into x and IsPrime will return False. To see if this happens, we tell use the Return command, which tells the system to continue until the current function returns. We type:

```
(Pdb) return
```

and the system responds:

```
--Return--
> c:\users\bob geitz\...\primes.py(4)IsPrime()->False
-> return False
(Pdb)
```

The first line of this says it is executing a Return command. The next line says that the current function, IsPrime, returns False. The third line is the next instruction to execute, the actual return instruction from line 4 of the code. If we step again we are back in function PrintPrimes:

```
> c:\users\bob geitz\...\primes.py(8)PrintPrimes()
-> for num in range(2,N+1):
(Pdb)
```

We now finish the exeuction of the program with a *Continue* statement. We type:

```
(Pdb) continue
```

and the system responds with the remaining prime numbers less than 100, and stops the program execution:

```
67
71
73
79
83
89
97
>>>
```

Note that the final prompt we get from the system is the usual shell prompt, not the (Pdb) prompt.

Using Pdb we can quickly get to any portion of a long program. With some patience, stepping through instructions and examining the values of variables, you can use this tool to find some very subtle errors in your code. This only works if you know exactly what your code is supposed to be doing; you must compare actual values to your expectations for those values. Debuggers are no substitute for good programming practices, but in some cases they can very helpful in the development of long programs.