5.2 The Shell Debugger

A *debugger* is a software tool that helps programmers find bugs in complex programs. This and the next section look at two simple debuggers that are standard parts of Python. A word of caution before we start: you should not let yourself become too dependent on a debugger. When you are starting out as a programmer, it is much more important to learn to read your code carefully than to become proficient with debugging tools. Good programmers write code that is easy to read; once you know how the various statements of Python work, errors should jump out at you. There is no program in all of these notes for which a debugger is a necessary part of the program development. Still debuggers are handy tools that can come to your rescue in those rare times when you just don't see what the problem is with your code. So use a debugger sparingly, but don't hesitate to use it when it will help.

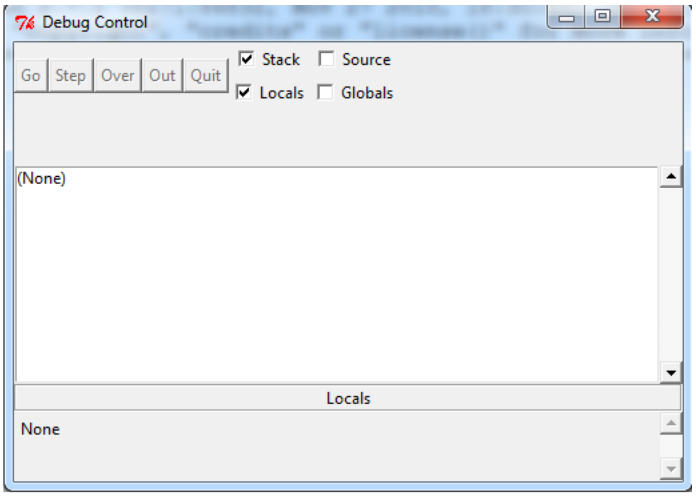
There are two standard debuggers for the Python. One of these is built into IDLE and the Python shell. The other is a module called "Pdb". The shell debugger is simpler to use and has fewer features, so we will look at it first. The next section discusses Pdb.

Suppose you are working with the following program:

```
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 5.2.1: Finding Prime Numbers

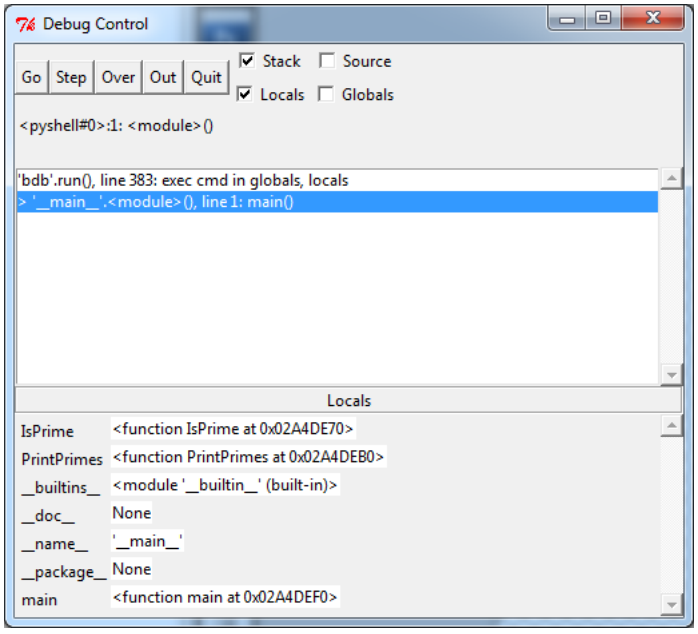
To use the shell debugger your program must be loaded into the system's memory. The easiest way to achieve this is to run the program from IDLE: just select Run Module from the Run menu. You need to run the program through to completion, so if your program expects input from the user, give it input until the program stops executing. Then in the Debug menu of the shell select the Debugger option. A *Debug Control* window will pop up:



To start debugging you need to restart your program. The program starts wtih a call to main() so at the debugger prompt in the shell we type

```
>>> main()
```

The Debug Controls window changes to include some information about the program.



There are five buttons at the top of this window that control the actions of the debugger. They are

Go This causes execution of the program to resume and run until the program finishes.

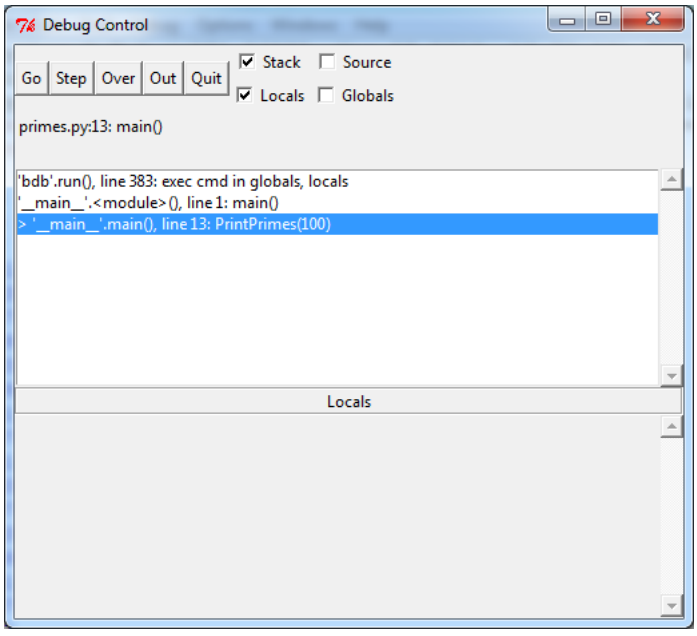
Step This causes the next instruction in the code to be executed. If that instruction is a function call, control changes to the first line of that function.

Over This is similar to Step, only function calls are evaluated rather than stepped into.

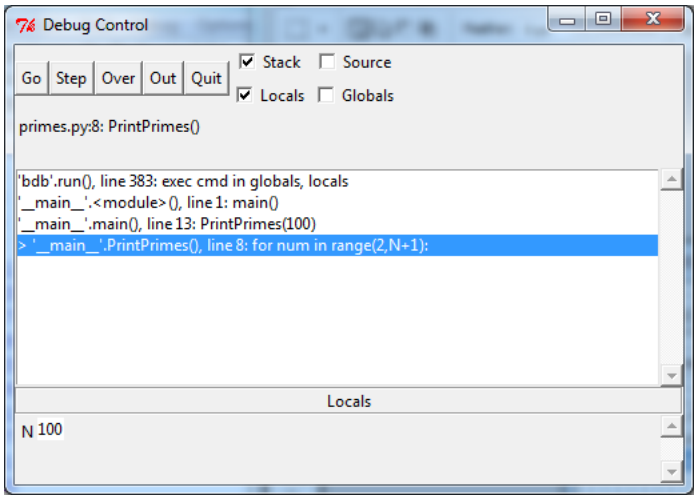
Out This causes the rest of the current function to be executed. The program halts as soon as it returns from the current function.

Quit This stops the execution of the program and halts the debugger.

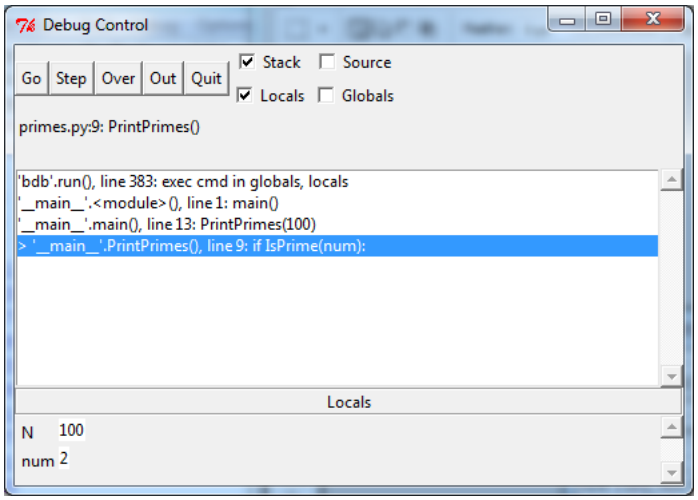
We will now go through a long series of steps examining our prime-generating program in the debugger. When we Step into main() we see:



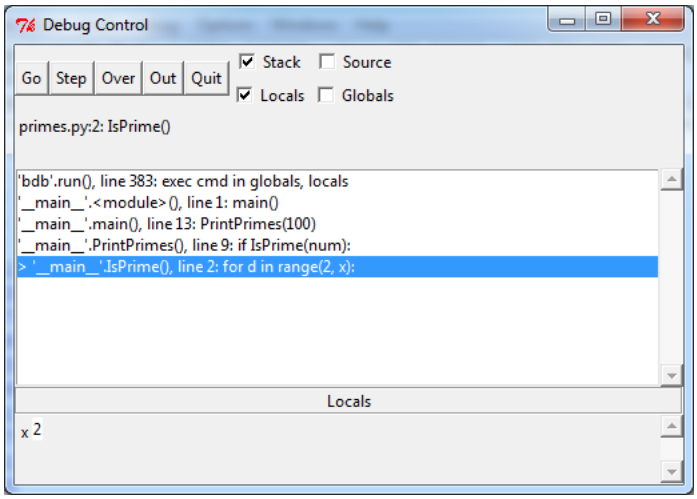
We now Step into PrintPrimes with argument 100:



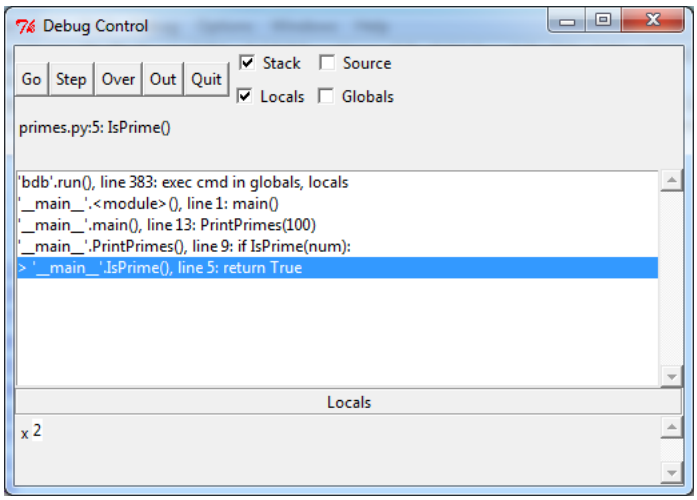
You can see the value of N at the bottom of the screen. The first line of PrintPrimes is a **for**-loop. Initially the loop variable num has no value. After we Step again:

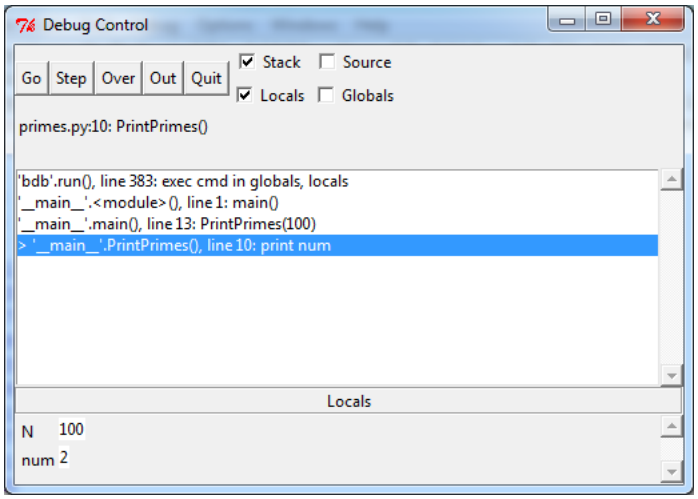


we see that num has the value 2 and we call IsPrime(num). The next Step takes us inside funciton IsPrime

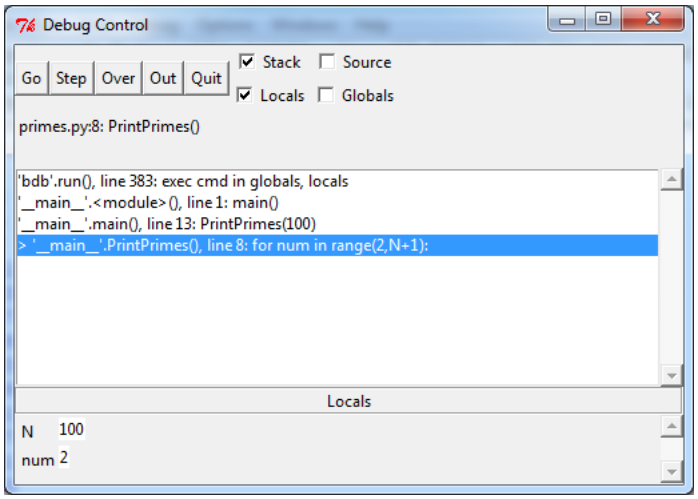


The **for**-loop is vacuous since the **range**(2,2) is the empty list, so IsPrime returns True and we go back to function PrintPrimes:

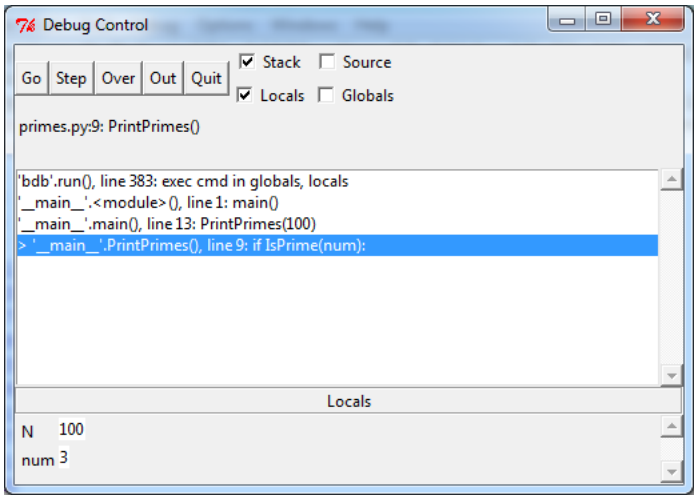




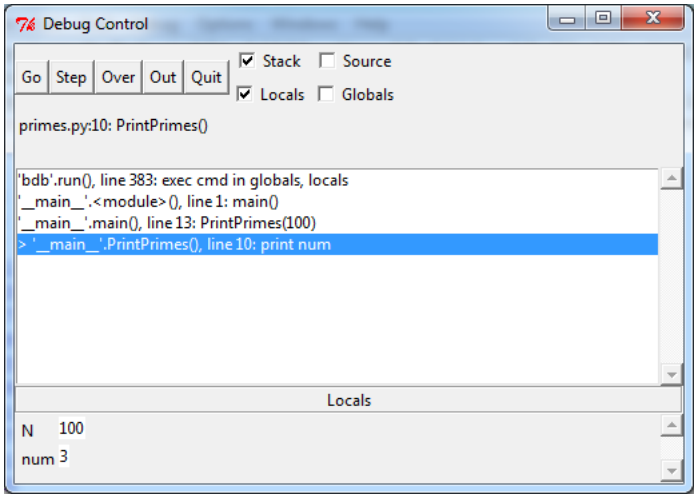
When we execute the **print** instruction the value of num, 2, is printed in the shell and we go to the next instruction:



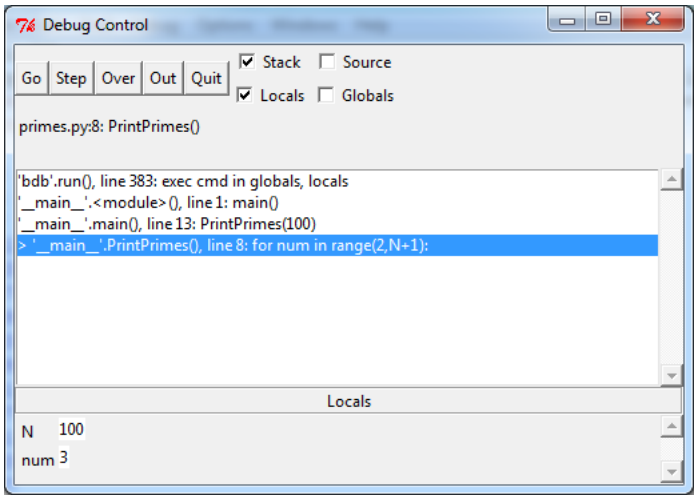
The next value of num is 3:



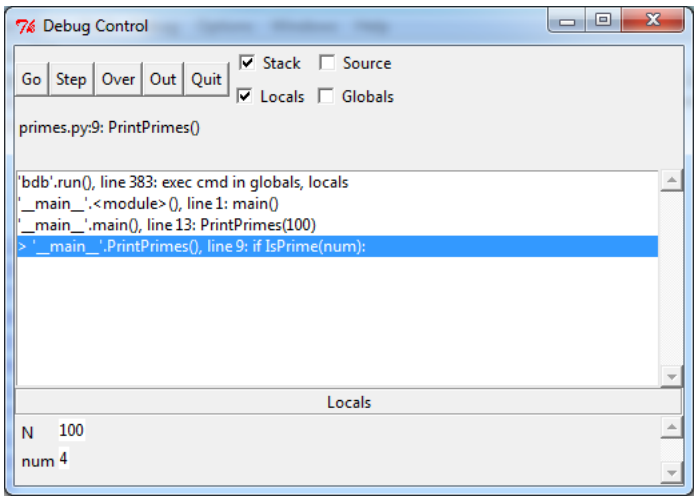
This time we will step Over the call to IsPrime(num). This returns True since 3 is prime and we come to:

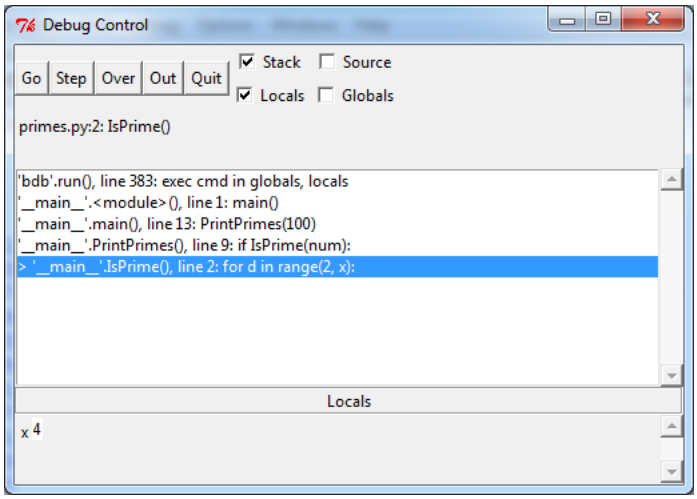


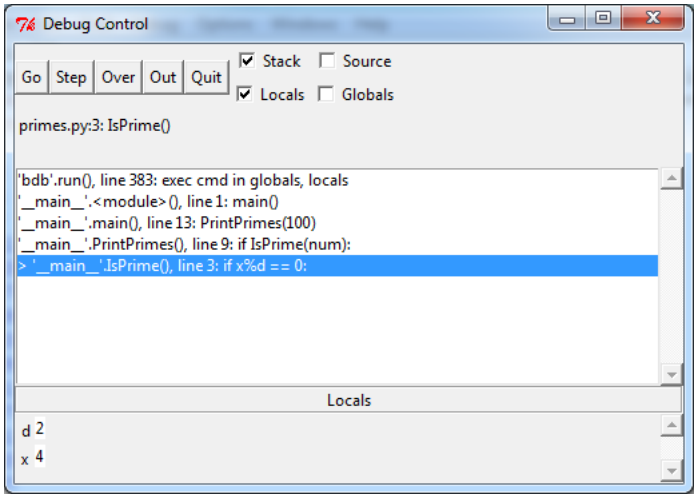
When we Step again num is printed, and we are back to our main **for**-loop:



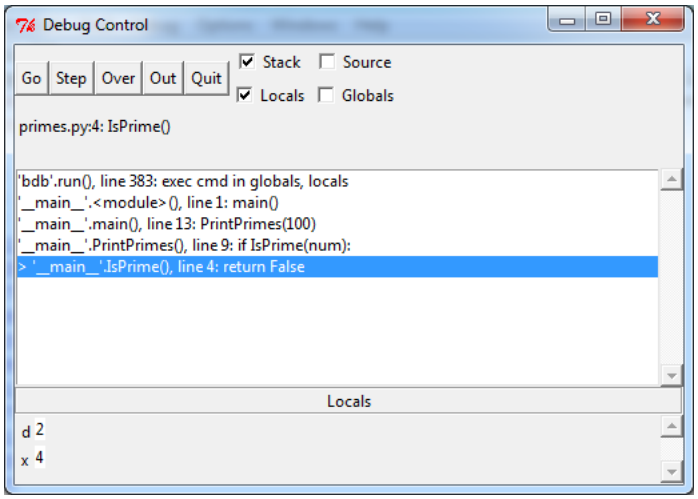
We now have a sequence of Steps with the value of num at 4:



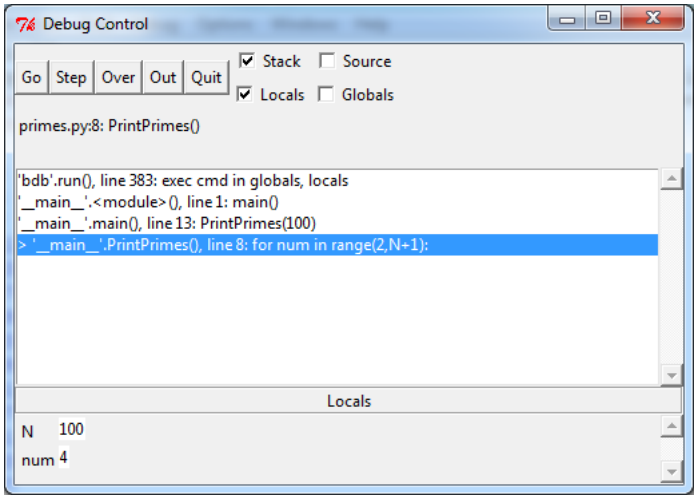




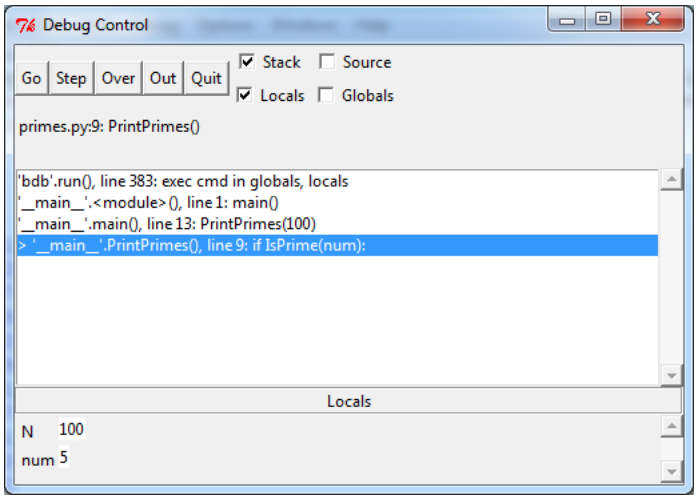
When x is 4 and d is 2, x%d==0 is True, so the function returns False

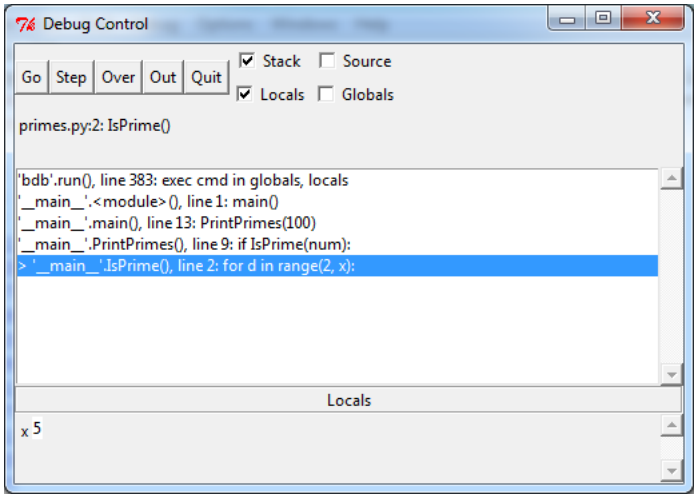


and we are back to the main **for**-loop of function PrintPrimes:

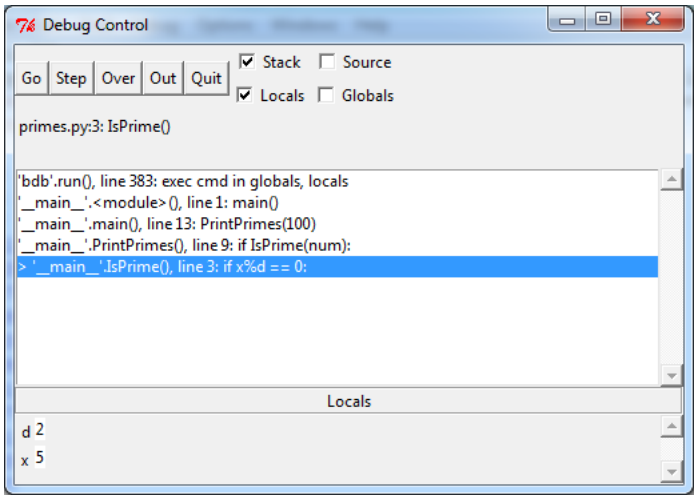


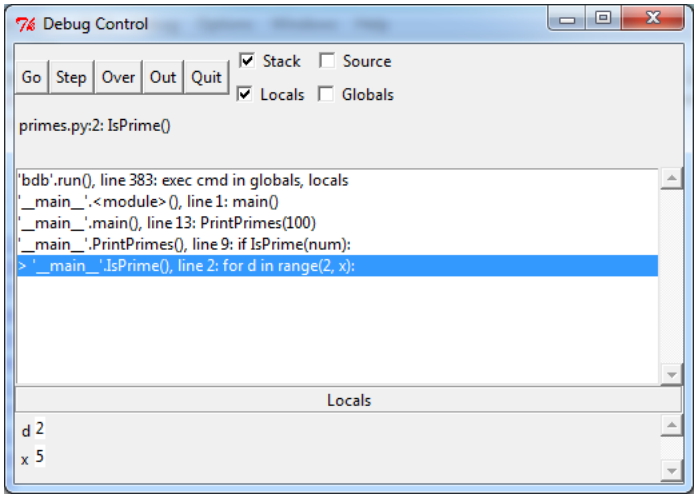
Now we have another sequence of steps to determine if 5 is prime:



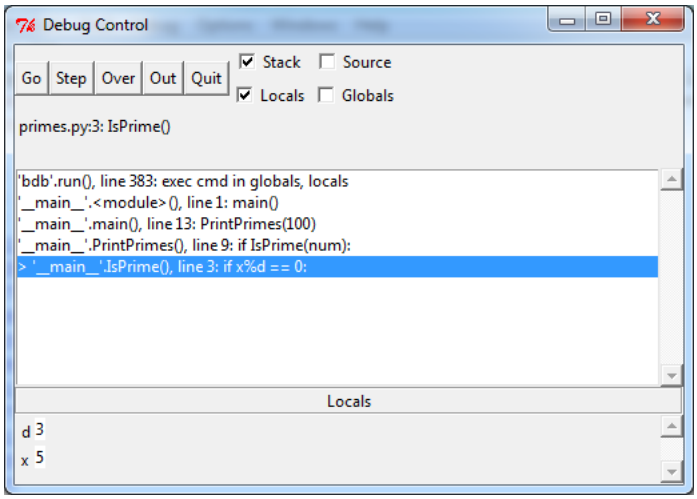


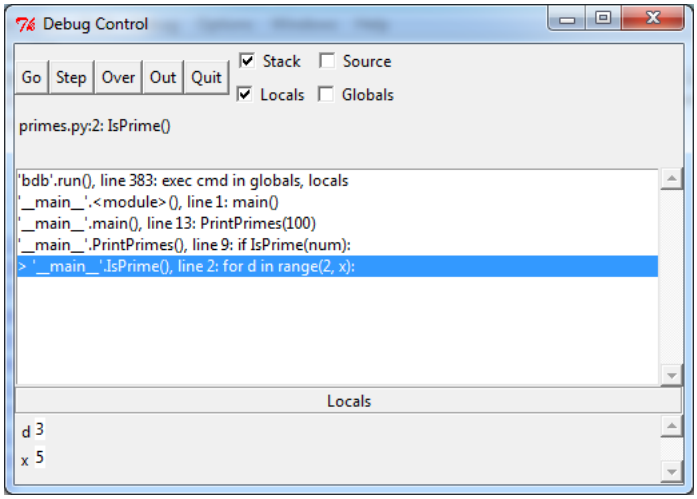
First we try to divide 2 into 5:


Debug Control
Go
Step
Over
Out
Quit
Stack
Source
Locals
Globals
primes.py:3: IsPrime()
'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
'__main__'.PrintPrimes(), line 9: if IsPrime(num):
> '__main__'.IsPrime(), line 3: if x%d == 0:
Locals
d 2
x 5

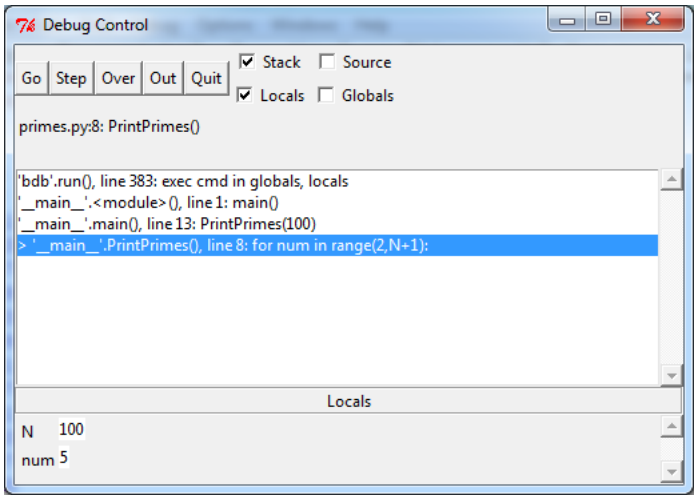

Debug Control
Go
Step
Over
Out
Quit
Stack
Source
Locals
Globals
primes.py:2: IsPrime()
'bdb'.run(), line 383: exec cmd in globals, locals
'__main__'.<module>(), line 1: main()
'__main__'.main(), line 13: PrintPrimes(100)
'__main__'.PrintPrimes(), line 9: if IsPrime(num):
> '__main__'.IsPrime(), line 2: for d in range(2, x):
Locals
d 2
x 5

Then we try 3:





We could continue in this way, but we know that 5 is prime and all of these divisions will fail. Let's step Out of the call to IsPrime This function completes its execution and returns True, so we are back in function PrintPrimes:



At this point we have seen enough. We click on the Go button to continue the rest of the program without interruption, and the remaining prime numbers are printed to the shell. We could run the program again in the debugger by typing

```
>>> main()
```

at the shell prompt. To exit from the debugger, either uncheck Debugger in the Debug menu or click on the close-box in the upper right corner of the Debug Control window.