# Lesson 24: Building and Debugging Code

## 1 Building Code

Write your code in small steps that you can test along the way. For example, consider a code that simulates a simple guessing game. The user is asked to guess an integer from 1 to 1000, and the program responds with "too high" or "too low" or "you're right!". The user continues to guess until he/she gets the right answer.

Let's create this code. To begin, the code needs to pick a random number from 1 to 1000. A quick check of numpy's random packages shows that the command `random.randint(low,high)` should generate a random integer between `low` (inclusive) and `high` (exclusive). Should we use `random.randint(1,1000)`, or `random.randint(1,1001)`, or `random.randint(0,1000)`? Let's check by writing

```
1 import numpy as n
2 thenum = n.random.randint(1,5)
3 print thenum
```

Running this code repeatedly shows that `thenum` is always one of the integers 1, 2, 3, or 4, and never 5. So we will need to use `random.randint(1,1001)` to get a random integer from 1 to 1000, inclusive. We can now delete the print statement in line 3.

The next step will be to prompt the user for a guess, and then add a bit of code to check if the guess is too high or too low:

```
1 import numpy as n
2 thenum = n.random.randint(1,1001)
3 guess = raw_input('guess a number from 1 to 1000: ')
4 if (guess < thenum):
5     print "too low"
6 else:
7     print "too high"
```

The code does not yet take into account the possibility that `guess` is correct. Also, it only lets the user make one guess. Before adding these features, let's test the code by adding a line to print out `thenum` and `guess`:

```
1 import numpy as n
2 thenum = n.random.randint(1,1001)
3 guess = raw_input('guess a number from 1 to 1000: ')
```

```
4 if (guess < thenum):
5     print "too low"
6 else:
7     print "too high"
8 print thenum, guess
```

When we run this code with the input `19`, the output is (for example)

```
too high
681 19
```

The guess was actually too low. What went wrong? After taking a closer look at the code, and perhaps using a debugger to trace the flow step–by–step, we realize that `guess` is a string, while `thenum` is an integer. The comparison `guess < thenum` always produces `False` and the code prints `too high`. The corrected code is:

```
1 import numpy as n
2 thenum = n.random.randint(1,1001)
3 guess = int(raw_input('guess a number from 1 to 1000: '))
4 if (guess < thenum):
5     print "too low"
6 else:
7     print "too high"
8 print thenum, guess
```

Several tests show that this works as expected.

Now extend the code to account for the possibility that `guess` is correct:

```
1 import numpy as n
2 thenum = n.random.randint(1,1001)
3 guess = int(raw_input('guess a number from 1 to 1000: '))
4 if (guess < thenum):
5     print "too low"
6 elif (guess > thenum):
7     print "too high"
8 else:
9     print "you're right!"
```

How can we test this? Let's insert a print statement before line 3, telling the value of `thenum`. That way we can put in the correct value for `guess`:

```
1 import numpy as n
2 thenum = n.random.randint(1,1001)
3 print thenum
4 guess = int(raw_input('guess a number from 1 to 1000: '))
5 if (guess < thenum):
6     print "too low"
7 elif (guess > thenum):
```

```
8     print "too high"
9 else:
10    print "you're right!"
```

Running this code several times, with values of `guess` that are too low, too high, and equal to `thenum`, shows that it works as expected.

Finally, we need to allow the user to make multiple guesses. If the number of guesses is unlimited, we can use a while loop that is always `True`. We can break from the loop when the guess is correct:

```
1 import numpy as n
2 thenum = n.random.randint(1,1001)
3 print thenum
4 while "True":
5     guess = int(raw_input('guess a number from 1 to 1000: '))
6     if (guess < thenum):
7         print "too low"
8     elif (guess > thenum):
9         print "too high"
10    else:
11        print "you're right!"
12        break
```

The `print` statement in line 3 allows us to continue testing the code with correct and incorrect guesses. Several tests show that this code works as desired—it continues to run until the right answer is guessed. Now remove the `print` command in line 3, leaving the final code:

```
1 import numpy as n
2 thenum = n.random.randint(1,1001)
3 while "True":
4     guess = int(raw_input('guess a number from 1 to 1000: '))
5     if (guess < thenum):
6         print "too low"
7     elif (guess > thenum):
8         print "too high"
9     else:
10        print "you're right!"
11        break
```

# 2 Debugging Code

So you've written a code, or part of a code, and you're ready to test it. Maybe the code doesn't run at all, or maybe the code runs but you need to verify that it's working as intended. The "old school" way to test or debug a code is to insert `print` statements to monitor the values of certain variables.

Consider a code that computes the Kempner series. The Kempner series is the sum of $1/n$ for all $n$'s from 1 to infinity, excluding those $n$'s that contain a 9. Of course we can't keep summing to infinity, so we'll cut the sum off after $n = 1000000$. To begin, let's not worry about excluding the $n$'s that contain a 9. Here's the code so far:

```
1 import numpy as n
2 Nmax = 1000000
3 K = 0.0
4 for n in range(1,Nmax+1):
5     K = K + 1/n
6 print K
```

This code runs, but gives the unexpected answer $K = 1$. We can investigate the problem by inserting a print statement within the loop to monitor the values of $n$ and $K$. Unfortunately, that will give us a million lines of output, which is not very practical. So let's also change `Nmax` to something more managable, like $Nmax = 5$:

```
1 import numpy as n
2 Nmax = 5
3 K = 0.0
4 for n in range(1,Nmax+1):
5     K = K + 1/n
6     print n, K
7 print K
```

The output from this code is

```
1 1
2 1
3 1
4 1
5 1
1
```

Although $n$ is running through the expected values, $K$ is stuck at 1. If we still don't see where the problem lies, we can monitor the individual parts of the calculation by adding $1/n$ to the print statement in line 6. The resulting output

```
1 1 1
2 1 0
3 1 0
4 1 0
5 1 0
1
```

shows that $1/n$ is being computed as 0, for all $n \geq 2$. Now we see what's wrong! By the rules of integer division, $1/n$ is rounded down to the nearest integer. We can correct this problem by replacing the 1 in $1/n$ with 1.0:

```
1 import numpy as n
2 Nmax = 5
3 K = 0.0
4 for n in range(1,Nmax+1):
5     K = 1.0/n
6     print n, K, 1.0/n
7 print K
```

Now the output is

```
1 1.0 1.0
2 1.5 0.5
3 1.83333333333 0.333333333333
4 2.08333333333 0.25
5 2.28333333333 0.2
2.28333333333
```

This looks correct. Remember, we still need to modify the code to exclude terms in the sum for all $n$'s that contain a 9.

A debugger is a powerful alternative to using print statements to monitor the flow of a code. Debugger software allows you to step through the code and monitor the values of various variables. Canopy has a built–in debugger that you can access by logging in to Canopy[1] from the "Welcome to Canopy" window. Having logged in, your editor window should reveal some new features, shown in Fig. 1. In this figure, the first debugger icon is

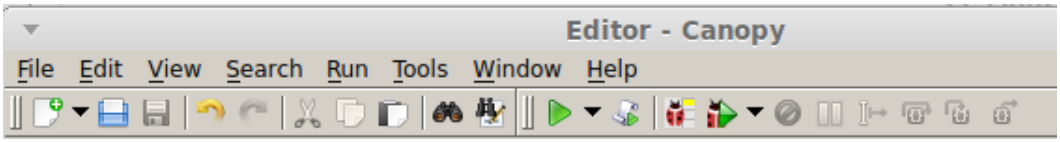


Figure 1: The Canopy menu bar with debugger icons.

directly below the "dash" in the title "Editor - Canopy". It is the "Show Debug Panes" icon. The remaining icons (from left to right) are "Debug Current File", "Stop debugging", "Pause debugged script", "Continue execution", "Step over next line of code", "Step into the next statement", and "Return from current function". You can see these descriptions by letting your mouse hover over the icon.

If we click on the "Show Debug Panes" icon, the Editor will reveal three new window panes, shown in Fig. 2. The new window panes are "Python Variable Browser", "Stack Browser", and "Breakpoint Viewer".

We can use the debugger to monitor the values of variables. We first set a "breakpoint" by double clicking to the left of a line, say, line 6. A red dot appears to the left of line 6. Also, line 6 is now listed in the "Breakpoint Viewer". A breakpoint is a place in the code where we want the execution to stop.

Now click the "Debug Current File" icon. The code will execute up to the breakpoint; that is, up to but not including line 6. Note that the values of the variables `K` and `Nmax`

[1]You must use your NCSU email address to create an academic account.

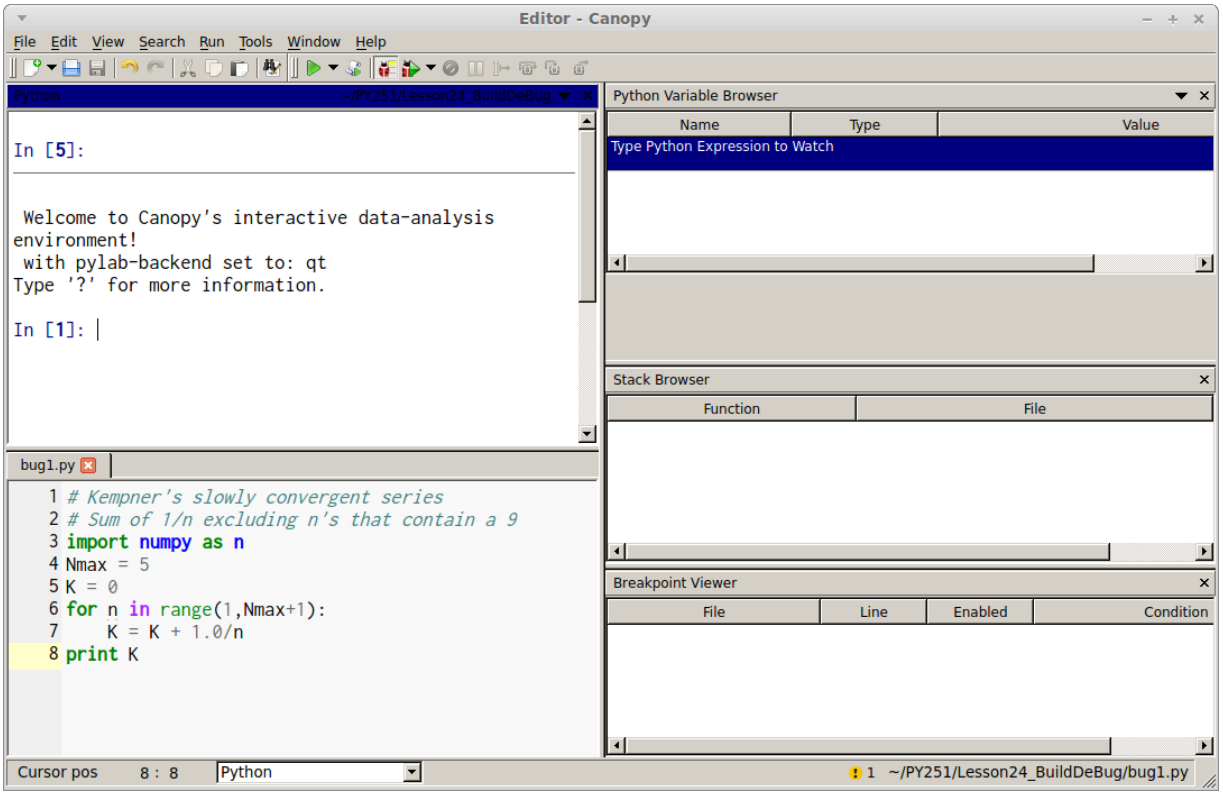


Figure 2: Canopy Editor showing the debug panes. (The code is not complete since $n$'s with a 9 are included in the sum.)

are now shown in the "Debug Variable Browser" (which was formerly called the "Python Variable Browser"). We can click "Continue execution" if we want the code to continue running until the end, or until it reaches another breakpoint. Alternatively, we can click the "Step Over next line of code" icon. This will execute the current line of code (line 6 in our case) and move the cursor to line 7. See Fig. 3. Now notice that the values of `K`, `Nmax`, and `n` are all displayed in the "Debug Variable Browser".

We can continue to click on the "Step Over next line of code" icon, which will execute one line of code for each click. We can see how the values of `K` and `n` change as we step through the code.

Exercise: Complete the Kempner series code by excluding from the sum all $n$'s that contain a 9. Have your code compute the series up to `Nmax = 1000000`. Use the debugger to step through the key parts of your code and make sure it is working correctly. (It can be shown that the Kempner series converges, very slowly, to about 22.9.)

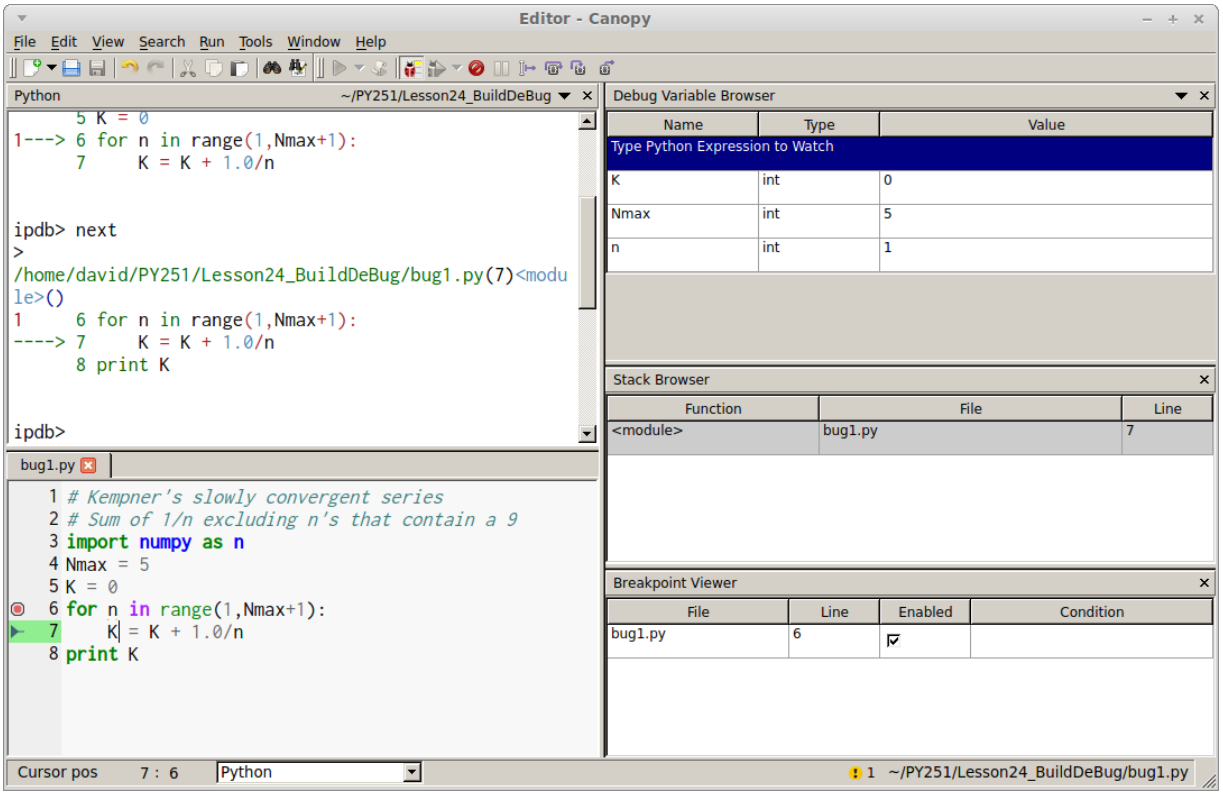


Figure 3: The breakpoint has been set at line 6. The "Step Over next line of code" icon was clicked, causing line 6 to execute and moving the cursor to line 7.

Homework: Pentagonal numbers are generated by the formula $P_n = n(3n-1)/2$ for $n = 1, 2, \ldots$. Write a code that finds pairs of pentaganol numbers $P_a$ and $P_b$ such that $P_a + P_b$ is also a pentagonal number. (Consider pentagonal numbers up to a maximum of, say, $P_{50}$.) Build your code in small steps. Use the debugger along the way to correct errors, and to make sure your code is working properly.