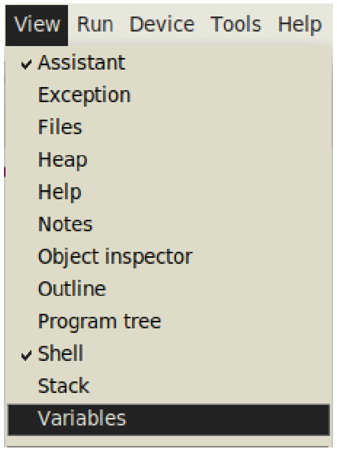
ICS3U: Using the Debugger (Thonny)



Trace tables are useful for monitoring the values of variables as they change during program execution, but this usefulness is limited: for large programs, creating a trace table can be time comsuming and prone to error (e.g. calculating values manually). For this reason, it may be preferrable to use a debugger, which acts in a similar manner but tracks all changes for you. Thonny includes a debugger, as do most modern IDEs.

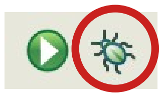
Monitoring Variables

Before launching the debugger, we will need to configure the IDE to display the values of all variables. To do this, click **View** → **Variables** from the main menu. This will open a sidebar containing a table where each variable's name and value will be shown. You can adjust the size of the table, or the width of the columns, as needed.

Debugging Options

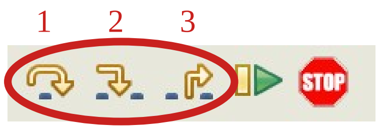
There are two debugging modes in Thonny: "nicer" and "faster". By default, the debugger uses "nicer" mode which displays more information than "faster". This can be changed under **Tools** → **Options**. There is a third option, bird's eye, which is a graphical debugging tool. We will not discuss this option here. You should be fine with either of the two main options.

Launching the Debugger



To start the debugger, click the "bug" icon from the main toolbar, or select **Debug current script** (either "nicer" or "faster") from the **Run** menu. You will see a highlighted line (usually the first) showing the current location in your program.

"Stepping" Through Code



There are three main actions that allow us to "step" through each command in a program: **step over** (1), **step into** (2) and **step out** (3). These are explained below:

- Step over: execute a command, or block of commands, *without* examining the internal workings of the command/block.
- Step into: execute a command, or block or commands, while examining the internal workings of the command/block.
- Step out: if inside of a block of commands or a function, execute the remaining code inside of it and exit the block/function.

The difference between stepping over and stepping into a command largely depend on whether you are using the "nicer" or "faster" debugging option. Using the "nicer" mode, Thonny will display far more information (e.g. evaluations, comparisons, etc.) when using step into than when using step over. In "faster" mode, step into will behave similarly to step over unless a programmer-defined function is called. We will explore each method below.

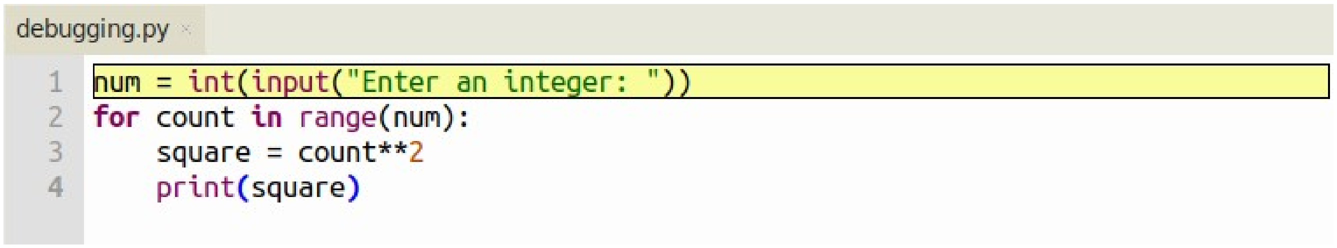
Debugging In “Nicer” Mode

When first learning how to code in Python, “nicer” mode can help us understand how things work under-the-hood. The debugger will show us lots of details about the expressions that we are evaluating, and how values are compared to others. Even if a program is working fine, we may want to better understand how the sequence of instructions produces correct results.

Consider the following code that we want to examine.

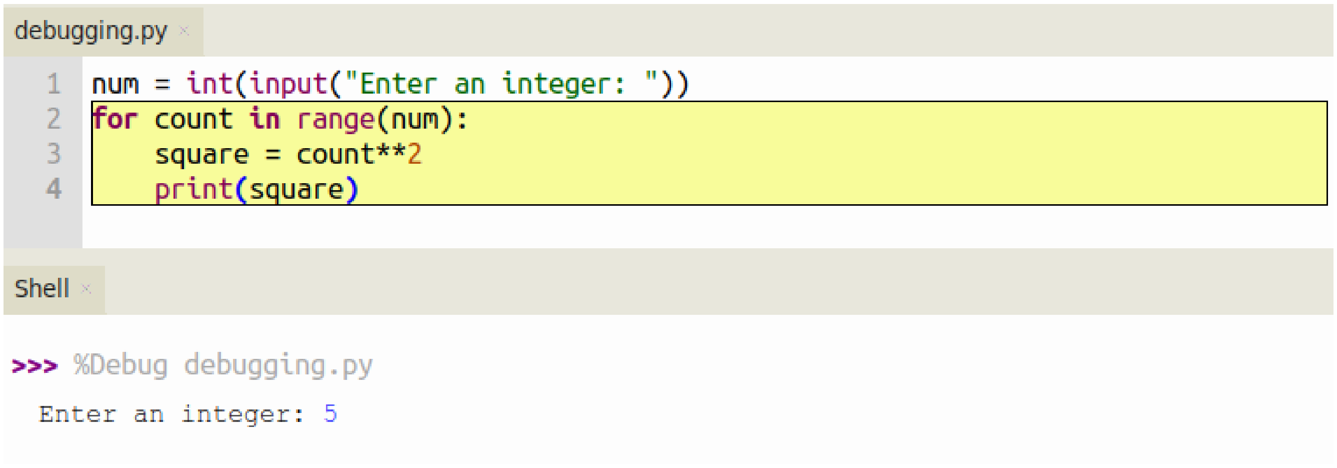
```
num = int(input("Enter an integer: "))
for count in range(num):
    square = count**2
    print(square)
```

Clicking the “bug” icon starts the debugger, highlighting the first line of code as shown.



```
num = int(input("Enter an integer: "))
for count in range(num):
    square = count**2
    print(square)
```

If we do not wish to trace through each evaluation in the line, we can click “step over” to execute the command as normal. This will bring us to the shell, where we can enter a value.

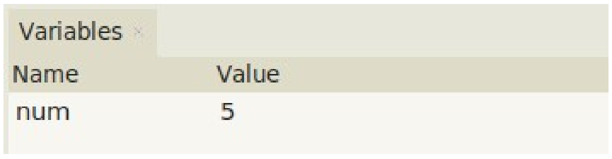


```
num = int(input("Enter an integer: "))
for count in range(num):
    square = count**2
    print(square)
```

```
>>> %Debug debugging.py
  Enter an integer: 5
```

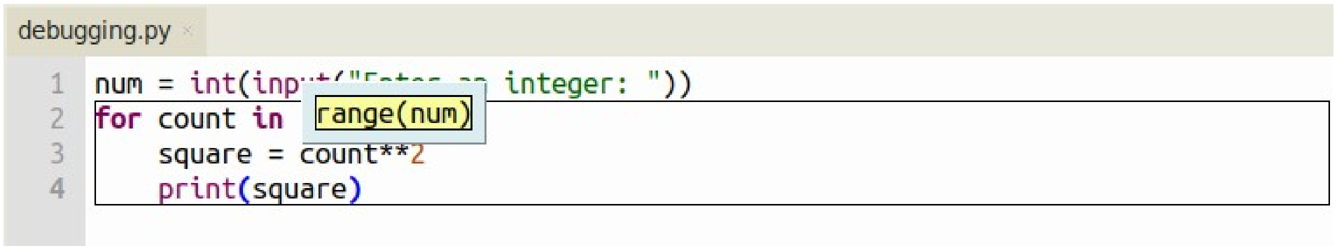
The debugger moves on to the next command, which happens to be a block of code (a `for` loop). The entire block is highligted, showing us that the three lines of code are associated with each other.

Also note that once we have entered a value for `num`, it appears in the Variables table in the sidebar. Each variable will have an entry created in the table when it is assigned a value, and as the values change, they will be updated automatically. This allows us to quickly see what values are being used at any time.



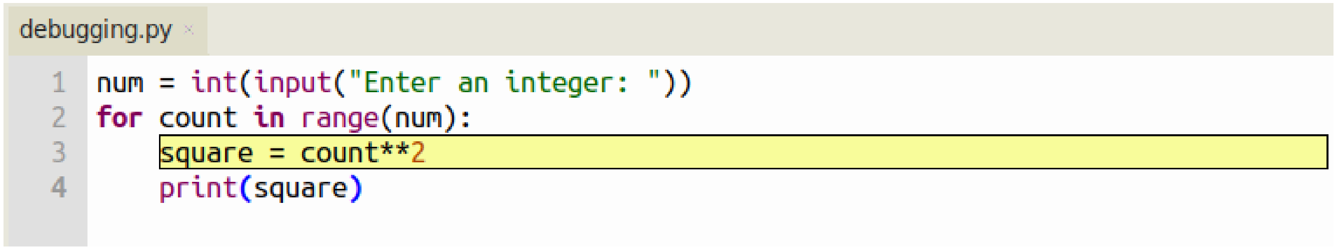
Name	Value
num	5

If we click "step over" again, the entire block will execute and the program will end. To walk through each line one-at-a-time, click "step into" instead.


debugging.py

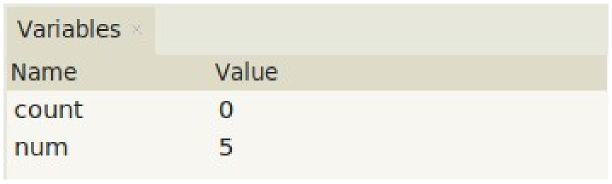
```
num = int(input("Enter an integer: "))
for count in range(num)
    square = count**2
    print(square)
```

The debugger enters the `for` loop, and we can see it performing an examination of our call to `range`. If we want to continue seeing information about commands one-by-one, we can keep selecting "step into" from the debugging toolbar; however, if we want to evaluate each line with less detail, we can go back to clicking "step over". Once inside of a block of code, it is not necessary to click "step into" all of the time unless we want to. Clicking "step over" results in the screenshot below.


debugging.py

```
num = int(input("Enter an integer: "))
for count in range(num):
    square = count**2
    print(square)
```

The debugger is ready to execute the command on line 3, and an entry for count has been created in the Variables table. From here, we can continue to step over or into our code as required.

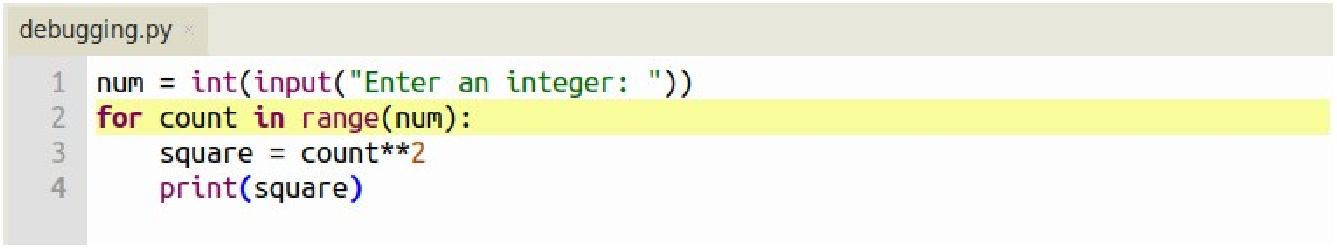

Variables

Name	Value
count	0
num	5

Debugging In "Faster" Mode

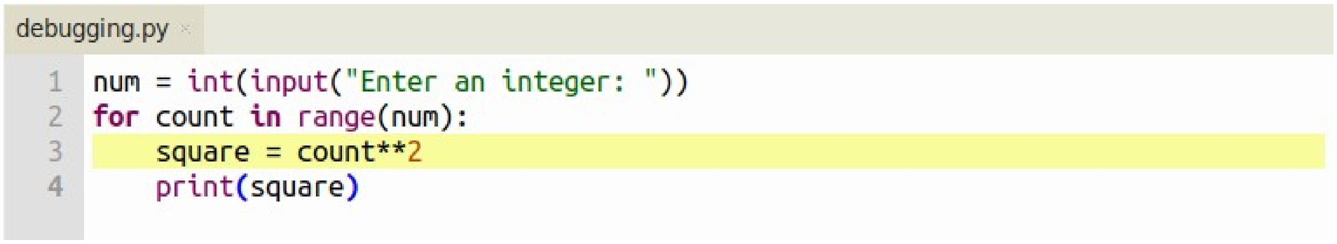
Using "faster" mode is, as its name suggests, *faster* than using "nicer" mode. This is because most of the evaluations, comparisons and substitutions are hidden. Variable values will continue to show up in the Variables table, however, which is what we really want to monitor anyways. Using the debugger controls is similar to "nicer" mode, but there are a few differences, particularly when encountering blocks of code.

Consider the same program as earlier. When we start the debugger in "faster" mode and click "step into", the program immediately prompts us to input a value for `num`. After we do this, the debugger moves to the next line.


debugging.py

```
num = int(input("Enter an integer: "))
for count in range(num):
    square = count**2
    print(square)
```

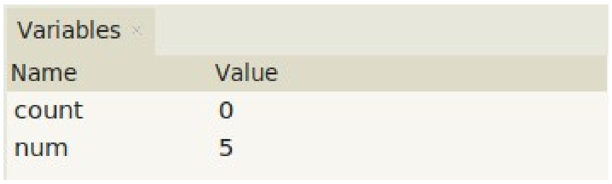
Notice that the entire `for` loop is *not* highlighted like it was in "nicer" mode. Furthermore, clicking "step into" simply moves the debugger to the next line.



debugging.py

```
num = int(input("Enter an integer: "))
for count in range(num):
    square = count**2
    print(square)
```

It may appear that "step over" and "step into" act in the same way, but this is not the case when it comes to functions that you have created (more on this later). Note that the variables num and count do show up in the Variable table, as before.

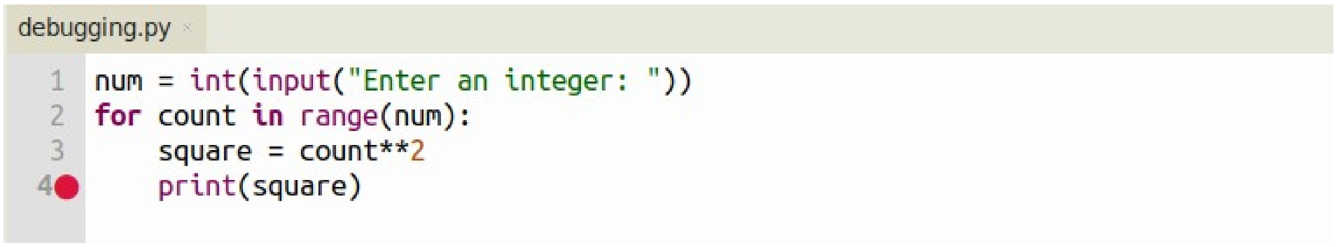


Variables

Name	Value
count	0
num	5

Breakpoints

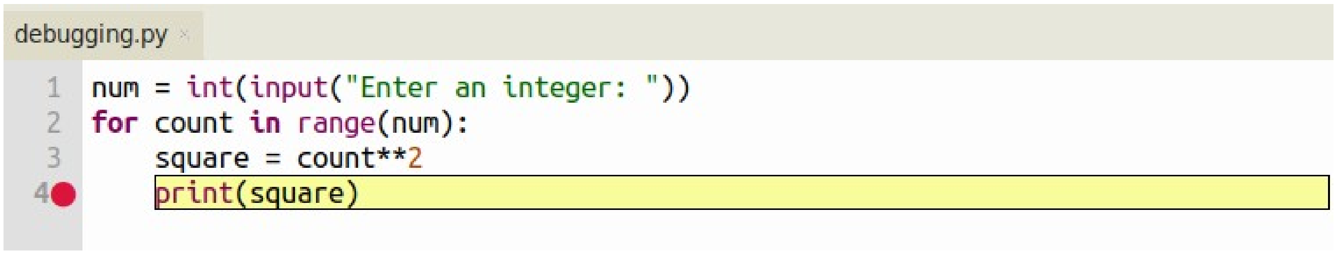
A breakpoint is a location where you want your program to pause. To create a breakpoint, you must first turn on line numbers if you have not already (go to **Tools** → **Options** → **Editor** and click the checkbox next to **Show line numbers**). Double-clicking the line number will mark the line with a red circle, indicating that a breakpoint is set at that line. Double-clicking again will remove it.



debugging.py

```
1  num = int(input("Enter an integer: "))
2  for count in range(num):
3      square = count**2
4●     print(square)
```

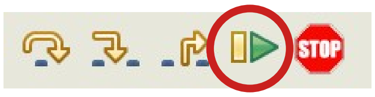
If you set a breakpoint, then running the debugger will execute all commands up until the breakpoint is reached, regardless of whether you are using "nicer" or "faster" mode. You will not be prompted to step through the lines that come before it, although any requests for user input will be executed in the shell. This is useful if you have a large amount of code that you want to skip over, and want to focus on just a specific portion of a program. Clicking the debug icon results in the screenshot below.



debugging.py

```
1  num = int(input("Enter an integer: "))
2  for count in range(num):
3      square = count**2
4●     print(square)
```

Once the program has reached the breakpoint, you can resume stepping through the code using either "step over" or "step into". If you want to execute the program until it hits the breakpoint again (as it might in a loop), you can also click the Resume button on the debug toolbar.

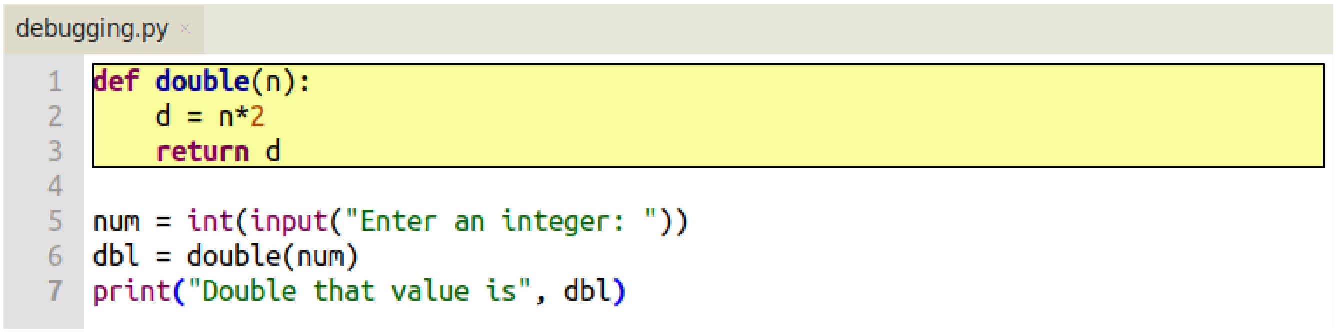


The Debugger and Programmer-Defined Functions

Consider the following code, which uses a function created by the programmer to double the value of a user-entered integer.

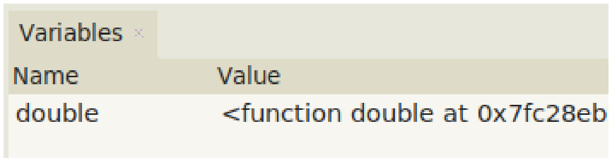
```
def double(n):
    d = n*2
    return d

num = int(input("Enter an integer: "))
dbl = double(num)
print("Double that value is", dbl)
```

If we are debugging in "nicer" mode, the debugger begins by highlighting the programmer-defined function, `double`, as shown.



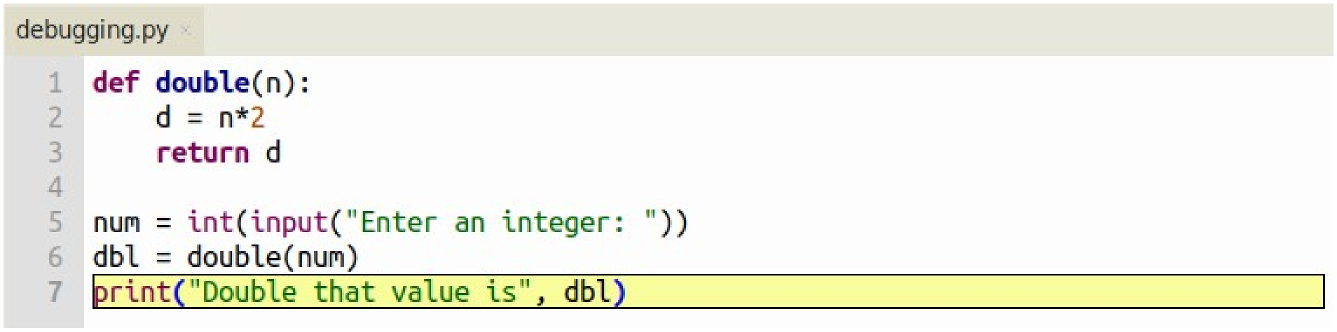
```
1  def double(n):
2      d = n*2
3      return d
4
5  num = int(input("Enter an integer: "))
6  dbl = double(num)
7  print("Double that value is", dbl)
```

No code in the function will be executed until it is called, and so the debugger will move to line 5 whether we click on "step into" or "step over". In the Variables table, there is an entry for the function's address in memory. This isn't particularly useful to us, but it tells us that the function has been loaded.



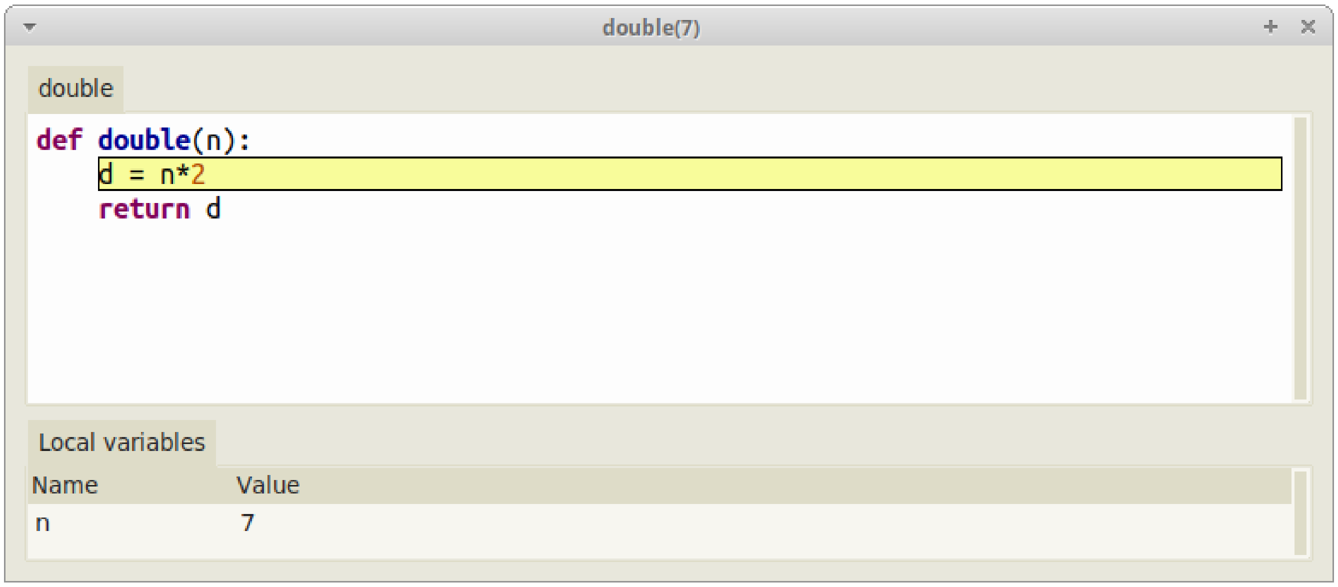
Name	Value
double	<function double at 0x7fc28eb

When we reach line 6 and call `double`, we can choose to examine the function or not. Selecting "step over" will skip this and assign a value to `dbl`, then move on to line 7.



```
1  def double(n):
2      d = n*2
3      return d
4
5  num = int(input("Enter an integer: "))
6  dbl = double(num)
7  print("Double that value is", dbl)
```

On the other hand, selecting "step into" (regardless of whether we are using "nicer" or "faster" debugging) will open a new window with the function code inside.

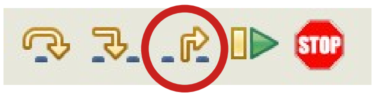


```
def double(n):
    d = n*2
    return d
```

Local variables

Name	Value
n	7

Below the code is a Local variables table, where values of variables declared inside of the function are displayed. You may need to expand the window to see it. It behaves exactly like the Variables table in the main program. Later in the course when we talk about **variable scope**, we will discuss how variables created inside of a function are separate from those created inside of the main program.



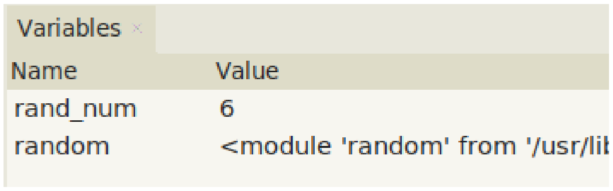
To control the program flow, click the icons in the debugging toolbar in the main Thonny window. When the function has completed (either by executing all commands or encountering a `return` statement), the window will close and the debugger will resume execution on the next line. If you wish to exit the function without stepping through the remaining lines of code, click the "step out" button.

Using the Debugger With Randomness

The code below plays a simple game where the user must guess an integer between 1 and 10.

```
import random
rand_num = random.randint(1, 10)
guess = int(input("Guess a number between 1 and 10: "))
if guess == rand_num:
    print("You guessed it!")
else:
    print("Sorry, the number was ", rand_num, ".", sep="")
```

Due to its random nature, it might take some time before the user correctly guesses the integer, making testing the `if` statement problematic. We *could* insert a `print` statement after the random number is generated to show us what the value is, but this would require both adding the line of code and also removing it later (so the user cannot see it). In a much larger program, there could be multiple values that we want to test, scattered throughout the program. Modifying the code may not be so easy.



Variables

Name	Value
rand_num	6
random	<module 'random' from '/usr/lit

Alternatively, we can use the debugger on its own to see what the value of the random integer is, and then enter that value to test the `if` statement. As shown to the right, `rand_num` has a value of 6 during this run of the program.

Entering 6, we obtain the following output, suggesting that the code is working as intended.

```
Guess a number between 1 and 10: 6
You guessed it!
```

Using the Debugger To Correct Errors

Another use of the debugger is to locate errors in calculations or other variable assignments. Consider the code below, which determines the slope of a line using the familiar formula, $\text{slope}=\frac{y2-y1}{x2-x1}$.

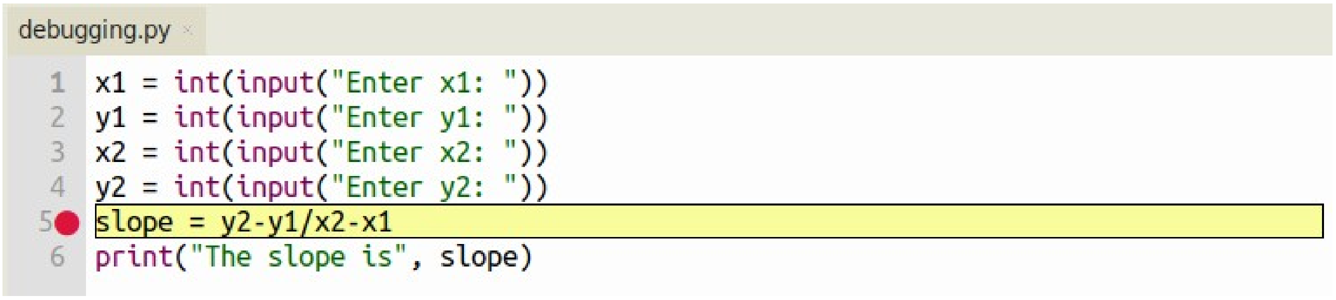
```
x1 = int(input("Enter x1: "))
y1 = int(input("Enter y1: "))
x2 = int(input("Enter x2: "))
y2 = int(input("Enter y2: "))
slope = y2-y1/x2-x1
print("The slope is", slope)
```

When the program is run, the following output is produced.

```
Enter x1: 3
Enter y1: 8
Enter x2: 1
Enter y2: 2
The slope is -9.0
```

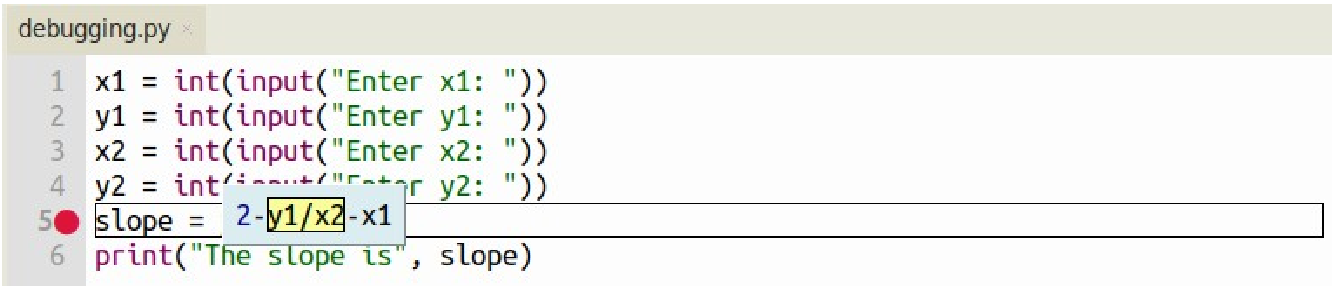
The slope is incorrect, since simple calculations by hand show that $\text{slope}=\frac{8-2}{3-1}=\frac{6}{2}=3$.

To determine the cause of the error, we can use the debugger in "nicer" mode. First, we set a breakpoint on line 5 because we are fairly confident that the error is not happening on the previous four lines and we do not wish to have to step through all of them. This pauses the program on line 5.



```
debugging.py
1  x1 = int(input("Enter x1: "))
2  y1 = int(input("Enter y1: "))
3  x2 = int(input("Enter x2: "))
4  y2 = int(input("Enter y2: "))
5● slope = y2-y1/x2-x1
6  print("The slope is", slope)
```

At this point, we begin stepping into the code on line 5 and notice the following evaluation:



```
debugging.py
1  x1 = int(input("Enter x1: "))
2  y1 = int(input("Enter y1: "))
3  x2 = int(input("Enter x2: "))
4  y2 = int(input("Enter y2: "))
5● slope = 2-y1/x2-x1
6  print("The slope is", slope)
```

The formula is not being evaluated in the correct order, because we are missing parentheses for both the numerator and denominator. That is, line 5 should read

```
slope = (y2-y1)/(x2-x1)
```

in order to force the division after the subtractions. Making this change fixes the program and produces the expected output.

Using the Debugger To Determine Program Flow

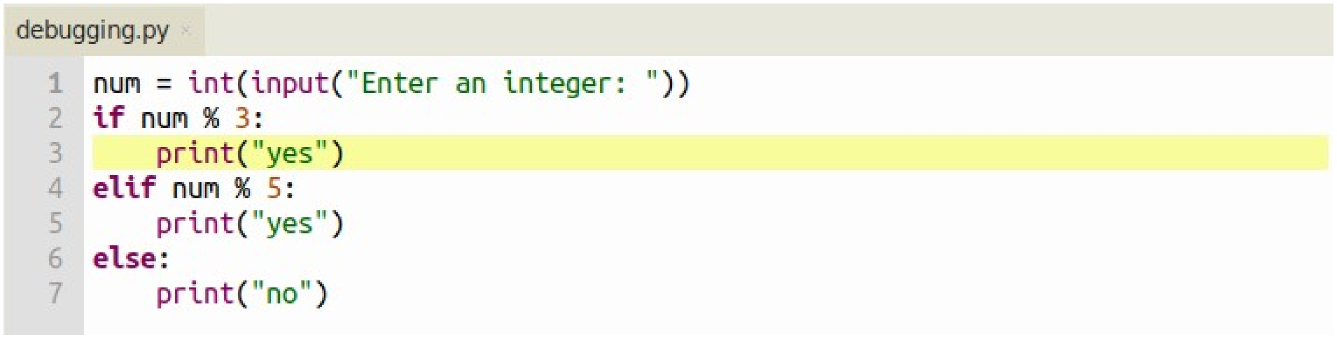
One final use of the debugger is to identify the paths that the interpreter follows when executing a program. This can be particularly handy for locating sources or logical errors. Unfortunately, the hard part (fixing them) remains our responsibility. The code below reads an integer from the user and determines whether or not it is a multiple of either 3 or 5.

```
num = int(input("Enter an integer: "))
if num % 3:
    print("yes")
elif num % 5:
    print("yes")
else:
    print("no")
```

Unfortunately, *all* integers are reported as being multiples of 3 or 5.

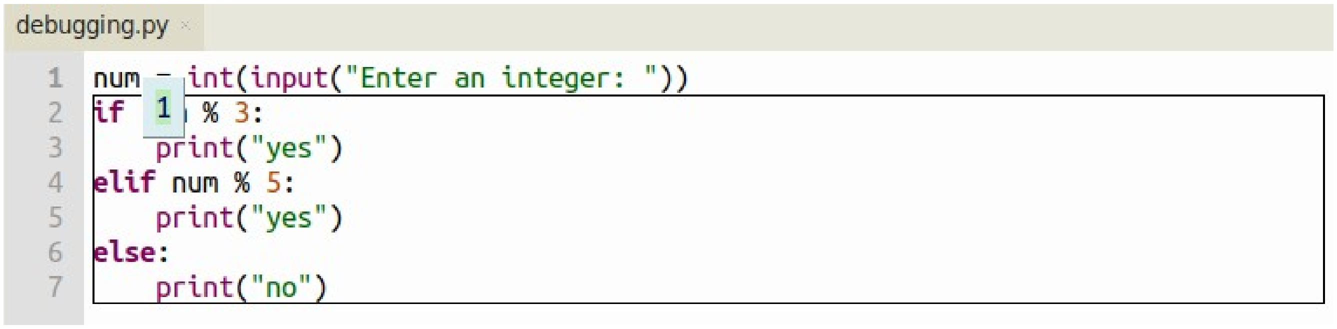
```
Enter an integer: 10
yes

Enter an integer: 7
yes

Enter an integer: 1
yes
```

It is clear that the `else` statement is *never* being executed, but which other condition (the `if` or the `elif`) is causing the problem? Using the debugger in "faster" mode shows us that, no matter what value is entered, the initial `if` condition is *always* executed. The screenshot below shows what happens when the user enters a value of 7 for `num`.


debugging.py

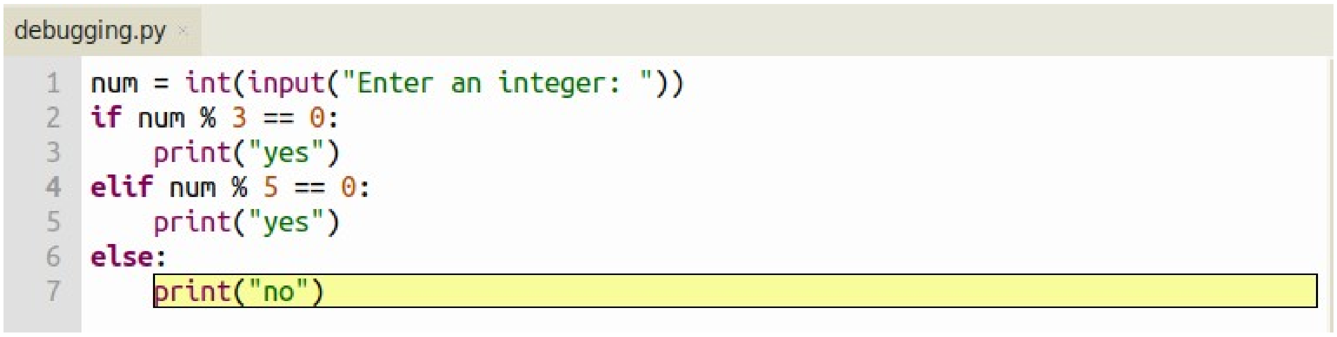
```
num = int(input("Enter an integer: "))
if num % 3:
    print("yes")
elif num % 5:
    print("yes")
else:
    print("no")
```

This means that the statement `if num % 3` evaluates to `True` each time the program is run. For more information, we can try debugging the code in "nicer" mode.


debugging.py

```
1  num = int(input("Enter an integer: "))
2  if  1  % 3:
3      print("yes")
4  elif num % 5:
5      print("yes")
6  else:
7      print("no")
```

As we can see, the expression `num % 3` has a value of 1. Since we know that Python implicitly treats non-zero integers as `True`, the expression `if 1` is `True` and the code on line 3 is executed. It seems that we forgot to test if the remainder after division is zero in our condition. The code below, which has been corrected, shows that the code inside of the `else` statement is executed when it should be.


debugging.py

```
1  num = int(input("Enter an integer: "))
2  if num % 3 == 0:
3      print("yes")
4  elif num % 5 == 0:
5      print("yes")
6  else:
7      print("no")
```