

Appendix C

What's wrong with my code?

“We have met the enemy and he is us.”

Pogo by Walt Kelly.

C.1 It's your fault!

A computer does what you tell it to do, not what you intended it to do. If the program *you wrote* is not working as you intended, it's your fault. Who else could be at fault after all! The real question though is, how do I find the problem and fix it? That is the goal of this chapter.

Programmers make mistakes when writing programs. These mistakes come in many different forms: assumptions about the data, misunderstanding of the algorithm, incorrect calculations, poorly designed data structures, and—best of all—just plain blunders. Testing is a way to protect us from ourselves. If we make a mistake, it would be most useful to be informed *before* we release code for use. Without some form of testing, how will we ever really know how well our program works? In addition, testing done well indicates where the error which is the first step to fixing it.

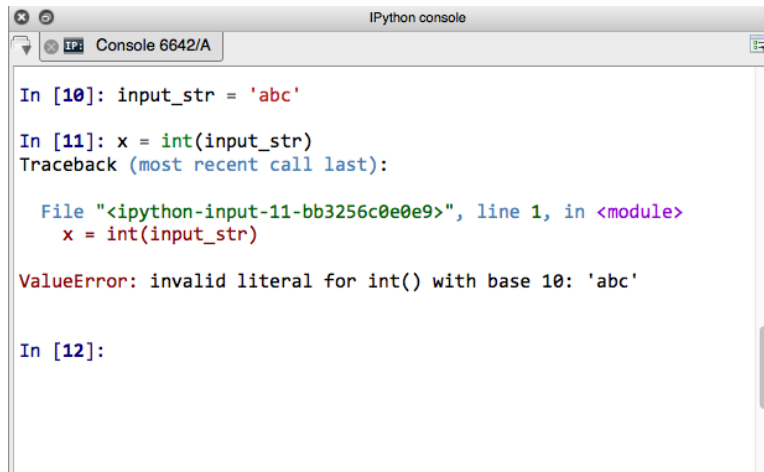
C.1.1 Kinds of Errors

There are at least three classes of errors:

- Syntactic errors
- Runtime errors
- Design errors

Syntactic errors are the easiest to detect and are detected for us by the interpreter. They are errors in the use of the programming language. Examples of syntactic errors include forgetting a colon (:) at the end of a header, a misplaced parenthesis, and a misspelling of a Python command—the list is long. When a program is run that contains a syntax error, Python reports the error, and (as best it can) identifies where the error occurred. Figure C.1 illustrates a `ValueError` generated by trying to convert a string of characters to an integer.

Runtime errors are errors of intent. They are legal language statements; they are syntactically correct, but they ask Python to do something that cannot be done. Their incorrectness cannot be detected by the Python interpreter syntactically, but occur when Python attempts to run code that cannot be completed. The classic example is to make a list of length 10 (indices 0–9) and then at some point during processing ask for the list element at index 10. There is no syntactic error; requesting a value at some index is a reasonable



```
IPython console
Console 6642/A

In [10]: input_str = 'abc'

In [11]: x = int(input_str)
Traceback (most recent call last):

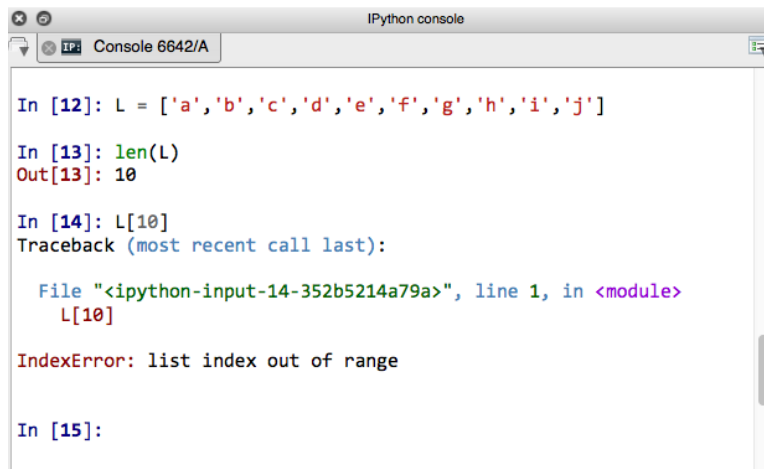
  File "<ipython-input-11-bb3256c0e0e9>", line 1, in <module>
    x = int(input_str)

ValueError: invalid literal for int() with base 10: 'abc'

In [12]:
```

Figure C.1: Syntax error.

request. Furthermore, requests of values at index 0–9 provide results. Only when the tenth index value is requested does Python encounter the problem. Python cannot provide that value at index 10 because it doesn’t exist; hence the runtime error shown in Figure C.2. Runtime errors can be hard to find, even with testing. They may occur intermittently: we may only occasionally ask for the index 10 element, depending on our calculations. Runtime errors may occur only under special circumstances. For example, if the name of the file we open has a suffix “.txt” and an error is triggered, any other suffix might be fine. Whatever the cause, just because you haven’t seen the error doesn’t mean it isn’t there!



```
IPython console
Console 6642/A

In [12]: L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

In [13]: len(L)
Out[13]: 10

In [14]: L[10]
Traceback (most recent call last):

  File "<ipython-input-14-352b5214a79a>", line 1, in <module>
    L[10]

IndexError: list index out of range

In [15]:
```

Figure C.2: Run-time error.

Design errors are a broad class of errors that basically cover everything else that is neither a syntactic nor a runtime error. The program is syntactically correct and we do not ask Python to do something that it inherently cannot do. However, the results we get are simply wrong. Design errors are just that: errors. We didn’t write our averaging equation correctly, we skipped a value in the list, we multiplied instead of dividing, we didn’t reset a default value in our function call, and so on. We made a mistake and, through no fault of the language, we got the wrong answer.

Testing for Runtime and Semantic Errors

Testing is about minimizing our errors as we program. We will make errors—that's a given. Can we recover from them, find them early, and minimize their effect? These are important questions.

C.1.2 “Bugs” and Debugging

Programmers often talk about bugs in their programs. The source of the term “bug” is something much discussed in computer science. The source of the term is not clear, but it is clear that it pre-dates computers. There are a number of documented letters of Thomas Edison using the term “bug” to indicate defects in his inventions. (http://en.wikipedia.org/wiki/Software_bug). There is some evidence that the term dates back to telegraphy equipment and the use of semi-automatic telegraph keys that were called “bugs” and, though efficient, were difficult to work with.¹ One of the most famous computer examples is traced to Admiral Grace Hopper. See Figure C.3.

In 1946, when Hopper was released from active duty, she joined the Harvard Faculty at the Computation Laboratory where she continued her work on the Mark II and Mark III. Operators traced an error in the Mark II to a moth trapped in a relay, coining the term *bug*. This bug was carefully removed and taped to the log book. Stemming from the first bug, today we call errors or glitch's [sic] in a program a *bug*.^a

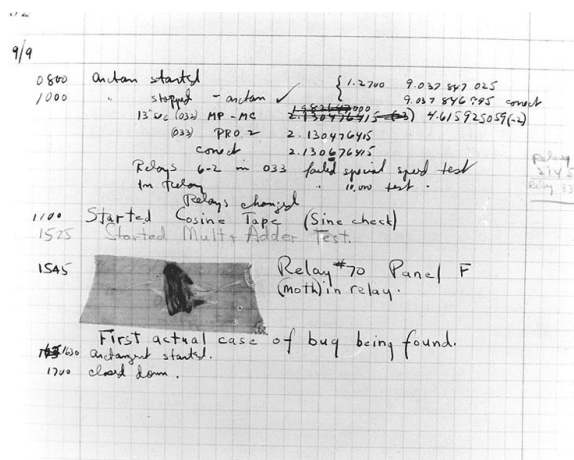


Figure C.3: A real hardware bug!

^a<http://ei.cs.vt.edu/~history/Hopper.Danis.html>

Removing bugs from a program has been called “debugging,” likely from this incident.

To call these errors “bugs” is a bit self-forgiving of us. It implies that these errors were not our fault. Rather, they slipped into our program without our knowing. We may not have known it at the time, but they are our errors. Edsger Dijkstra, a famous professor of computer science, said the following about the word “bug”:

We could, for instance, begin with cleaning up our language by no longer calling a bug “a bug” but by calling it an error. It is much more honest because it squarely puts the blame where it belongs, viz., with the programmer who made the error. The animistic metaphor of the bug that maliciously sneaked in while the programmer was not looking is intellectually dishonest as it is a disguise that the error is the programmer’s own creation. The nice thing of this simple change of vocabulary is that it has such a profound effect. While, before, a program with only one bug used to be “almost correct,” afterwards a program with an error is just “wrong.”¹

C.2 Debugging

There are three steps to finding errors in your code:

- 1 Test: run tests to determine if the program works correctly.

¹<http://id.answers.yahoo.com/question/index?qid=1006052411732>

```
# sum the numbers from 1 to 100
while number = 100:
    the_sum = the_sum + number
print("Sum:", the_sum)
```

2 Probe: insert probes to determine where the error occurs.

3 Fix: fix the error.

C.2.1 Testing for Correctness

Part of the specification of most problems in an introductory course often includes input with corresponding expected output. However, such specifications are usually incomplete so you need to develop your own tests.

When developing your code it is often useful to begin with small test cases that are easy to verify. If the sample file has 1000 lines, create a test file with the first 5 lines from the sample file.

Once you think your code is working take an adversarial approach: try to break your code. What input might cause a problem? The specifications included a few sample inputs so consider variations on that input. If integer input is expected, test what happens when non-integers are input.

We will return later to testing.

C.2.2 Probes

Once you know that an error exists the next step is to figure out where in the code the error occurs. Simply reading your code is a good place to start, and explaining your code to someone else is surprisingly effective (this is known as *code review*). Sometimes what made sense late last night seems quite stupid today when you are trying to explain it to someone else. They ask: why did you do that? You reply: I have no idea!

If simply reading (inspecting) the code doesn't reveal the error, insert *probes*. Probes allow you to view the values that variable actually have as opposed to what you think they have. There are two ways to probe your code:

- Insert *print()* probes into the code.
- Use a debugger (such as is available in Spyder).

Let's take a look at two programs with errors and see how to use the debugger to help find and fix them. The first program uses topics only from Chapters 1 and 2. The second program uses later topics so it can be used to illustrate more debugging features.

C.2.3 Debugging with Spyder Example 1

This first example will sum the numbers from 1 to 100 inclusive. We have already seen this example, but will revisit it to learn about debugging.

Code Listing B.1 shows our first version of the code – with errors (can you find them?).

If you copy the code into Spyder the first thing you will notice is the tiny warning flag with an embedded exclamation point with the line numbers. If you hover your mouse over it, you will see the message: invalid syntax (Figure C.4). Often the syntax error is on the flagged line, but frequently the error is on the previous line – the interpreter got confused when it got to this line. Unfortunately, the error message is not very helpful. If you run the program, you will see the same error message in the shell, but the interpreter will also try to point to where in the line it thinks the error is. In Figure C.5 you can see that the "=" sign is

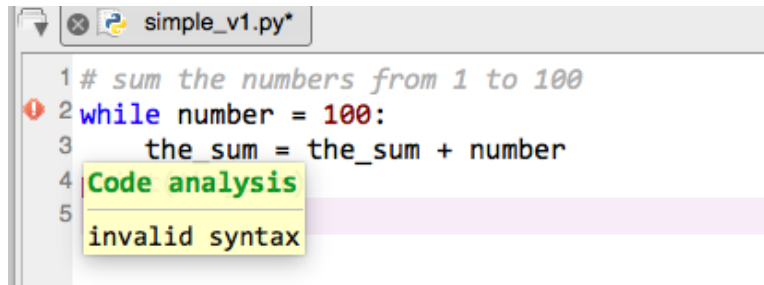


Figure C.4: Flagged syntax error.

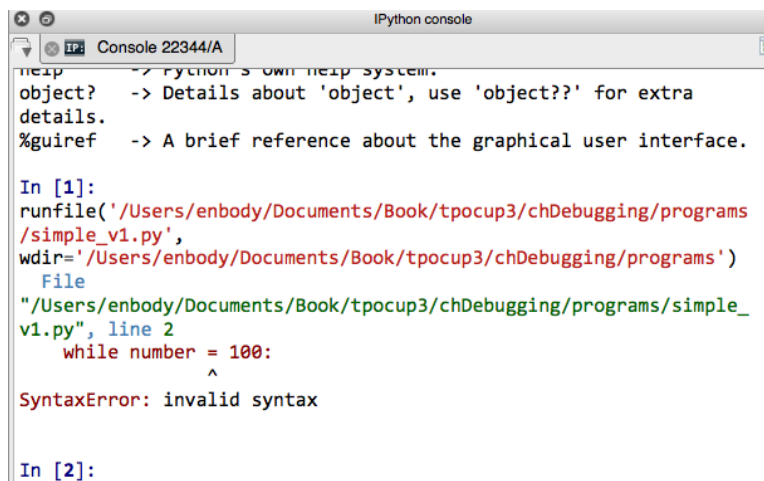


Figure C.5: Syntax error flagged in the shell.

```

# sum the numbers from 1 to 100
the_sum = 0
number = 1
while number <= 100:
    the_sum = the_sum + number
5 print("Sum:", the_sum)

```

indicated. Oops, we have an assignment rather than the “less than or equal” test we wanted: change “=” to “<=”.

When we make that fix the error flag disappears but a new one appears as shown in Figure C.6. In fact two new error flags appear indicating two “undefined name” errors. Neither `number` nor `the_sum` has been given an initial value. That is, neither has appeared on the left side of an assignment statement. Since we are collecting values into a sum, the additive identity zero is the appropriate initial value. Since we are adding numbers starting at 1, that is an appropriate initial value for `number`.

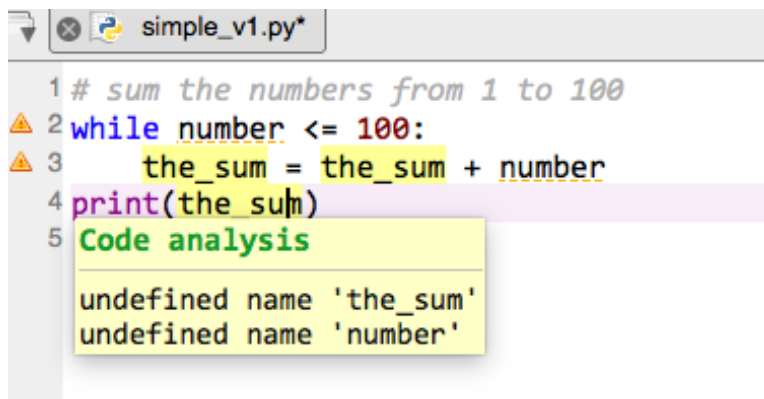


Figure C.6: Undefined name error.

Code Listing B.2 shows our second version of the code—still with errors (can you find them?).

We have now removed syntax errors that the interpreter could find. Let’s move on to other errors such as occur when the program runs. When we execute this code nothing happens—usually that means that there is an infinite loop. We could find it by inspection (can you see it?), but how can I use the debugger to help? First, let’s set a breakpoint – a spot where we will stop the program so we can examine the namespace for the values of variables. We can have an unlimited number of breakpoints, but fewer is easier to manage. There is only one loop so let’s put a breakpoint there. The best place is the first line of the loop, line 5 (in this case there is only one line). We can set a breakpoint there by double-clicking on the number 5. A small red circle marks a set breakpoint. The mark can be seen along the left side in Figure C.13.

With a breakpoint set we need to fire up the debugger and run the program to the breakpoint. The debugger symbols are in blue at the top of Spyder, Figure C.7. You can start the debugger from the drop down under “Debug” or select the start debugging symbol, a right-facing blue arrow with two vertical bars. That sets up the debugger. To run the debugger to the breakpoint, execute to the next breakpoint by selecting the double, blue, right-facing arrow symbol. (To stop debugging select the blue square.) With the Variable Explorer selected in the upper right of Spyder you can see the active variables in the program’s namespace.

Here are the steps:

- 1 set the breakpoint (double-click the line number for line 5)

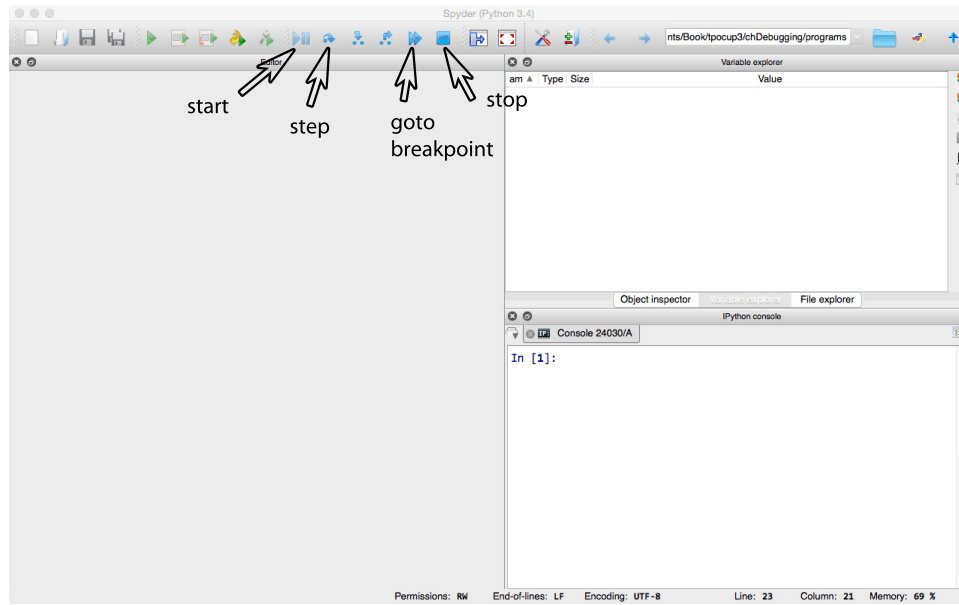


Figure C.7: Debugging command icons.

```
# sum the numbers from 1 to 100
the_sum = 0
number = 1
while number <= 100:
    the_sum = the_sum + number
    number+= 1
print("Sum:", the_sum)
```

2 check that the Variable Explorer is selected in the upper-right window

3 start the debugger (the blue right-arrow and two bars symbol)



4 execute to the breakpoint (the blue, double right-arrow symbol)



5 step one line at a time (blue curved arrow with one dot)



6 when you are done hit the “stop debugging” button



Figure C.7 indicates the stepping symbol. The Variable Explorer window shows the variables in the program’s namespace. Every time you click the step symbol one line of code is executed. That line is highlighted in the editor window and is indicated in the shell. As you step through the code observe variable changes in the Variable Explorer window in the upper right. What you notice as you step through this code is that the variable `number`, the loop control variable, never changes so that is the reason for the infinite loop. It needs to be incremented within the loop suite at the end.

With that error fixed we have the Code Listing B.3.

```
# sum the numbers from 1 to 100
the_sum = 0
number = 1
while number <= 100:
    print("number:", number, "; the_sum:", the_sum)
    the_sum = the_sum + number
print("Sum:", the_sum)
```

Running that code produces the sum 5050. Is that correct? The formula for the sum of a sequence of n integers starting at 1 is $n(n-1)/2$ which for $n = 100$ also produces the sum 5050 so the code is correct.

C.2.4 Debugging Example 1 using print()

How might I find the error of Example 1 without using the debugger? You might not have access to a debugger or might find it inconvenient for the information you need to gather. The idea here is to get to information about the state of the program while it is running. The technique is to carefully place `print()` statements to display the values of variables—much like we examined values in the Variable Explorer. The rule is to place `print()` statements where we set breakpoints. For example, place a `print()` statement at the beginning of a loop to discover the state in each iteration. A useful protocol is to label your output. Code Listing B.4 shows Example 1 with a labeled print statement at the beginning of the loop.

When we run this version of the code the infinite loop is apparent. Figure C.8 shows the output displayed in the shell. In the time it took to take a snapshot of the screen the program had generated over half-a-million print statements. The only way to halt Spyder is to Restart the kernel by selecting the icon shown in Figure C.8. Restarting loses the printed information, but it is easy to observe that `number` never changes, as we noted earlier. We need to increment `number` at the bottom of the loop suite as shown above.

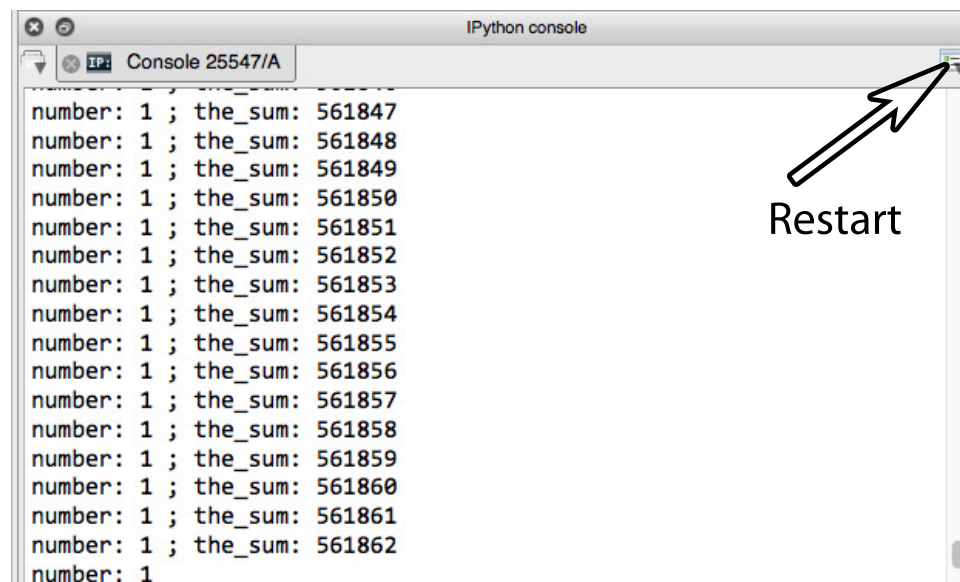
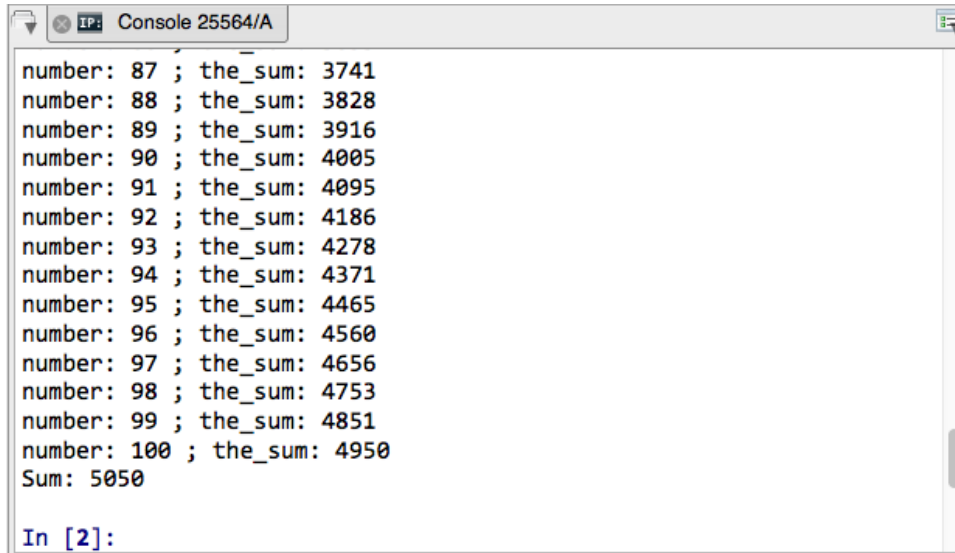


Figure C.8: Infinite loop printing.

One nice feature of using print statements is that you watch the values change in “real” time. For example, when this program is corrected you can see the sum accumulate as shown in Figure C.9.



```
number: 87 ; the_sum: 3741
number: 88 ; the_sum: 3828
number: 89 ; the_sum: 3916
number: 90 ; the_sum: 4005
number: 91 ; the_sum: 4095
number: 92 ; the_sum: 4186
number: 93 ; the_sum: 4278
number: 94 ; the_sum: 4371
number: 95 ; the_sum: 4465
number: 96 ; the_sum: 4560
number: 97 ; the_sum: 4656
number: 98 ; the_sum: 4753
number: 99 ; the_sum: 4851
number: 100 ; the_sum: 4950
Sum: 5050

In [2]:
```

Figure C.9: Loop printing.

```
def median(L):
    '''return median of a list of values'''
    L_length = len(L)
    L_sorted = sorted(L) # Let's not change L
    if L_length%2 == 1: # even length
        return L_sorted[L_length/2]
    else: # odd length
        return (L_sorted[L_length/2] + L_sorted[L_length/2 + 1])/2
```

C.2.5 Debugging with Spyder Example 2

To explore the debugger further let's find the median of a list of values. For example, given the sorted list [1, 3, 5] the median is 3, the middle value. That list had an odd number of values. If there are an even number of values, take the average of the two middle values. That is, the even-length sorted list [1, 3, 5, 7] has median 4.0 which was calculated using $(3 + 5)/2$.

Our algorithm is:

- Find the length of the list.
- if the length is odd, divide the length by two to get the index of the middle
- else: find the indices of the two middle values and find their average

Code Listing B.4 shows our first version of the code – with errors (can you find them?).

We first have to eliminate syntax errors such as is indicated in Figure C.10. As before, the error in the shell may be more informative. We have an assignment rather than the equality check we wanted: change “=” to “==”. When we make that fix the error flag disappears (there is a delay whose length you can control in Spyder preferences).

With the syntax error fixed we can run the program. Since median is a function we must call it in the shell with some list, e.g. `median([1,5,3])` as shown in Figure C.11. We now see a new error: *TypeError*:

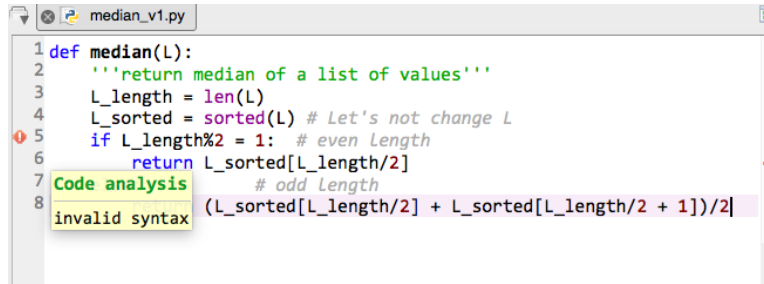


Figure C.10: Flagged syntax error.

list indices must be integers, not float. The error message indicates that we have an error in the index to a list. The indicated line indexes the `L_sorted` list as `L_sorted[L_length/2]`. Since the division `"/"` in our index `L_length/2` returns a float that is the source of the error. We can fix that with integer division (`"//"`), but we made the same mistake in line 8 so let's fix those, too.

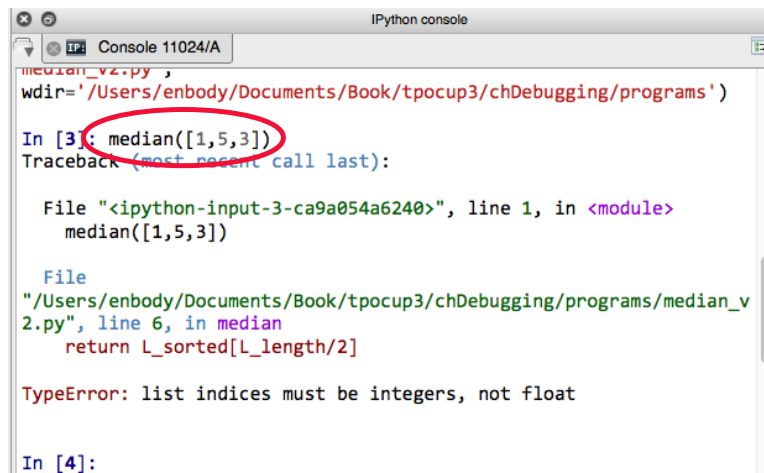


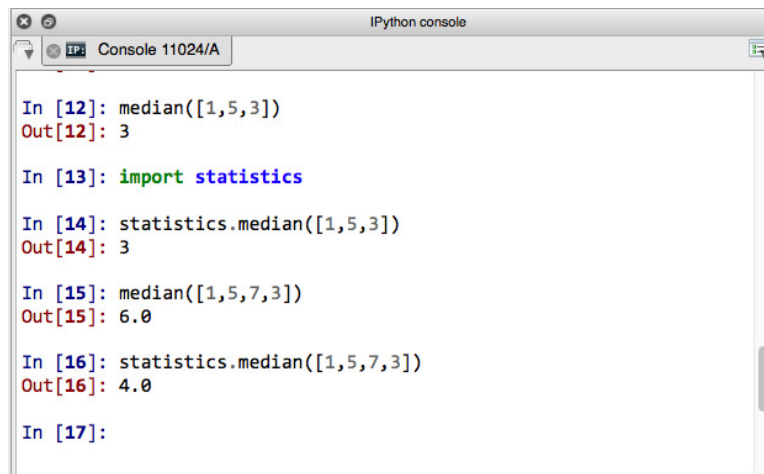
Figure C.11: Syntax error in shell.

With those errors fixed we can test that `median([1,5,3])` returns a result, but is it correct? What is the right answer? In this case, we can call on a statistics module so let's `import statistics` and find out what the correct answer is. See Figure C.12. Calling `median` in the statistics module yields the same result. Note that we have only tested an odd-length list so let's test an even-length list, `median([1,5,7,3])`, and compare it to the result from the statistics module. Again, we find a problem as shown in Figure C.12.

By this time you may be tiring of typing and retyping the testing code so let's build it into our program as shown in Code Listing B.5.

With the call to `median` embedded in the program we can now call up the debugger and set a breakpoint. Upon inspecting our code we have only one line related to odd length lists and it is likely that our problem is again with the indices. That is line 10 so we can set a breakpoint there by double-clicking on the number 10. A small red circle marks a set breakpoint. The mark can be seen along the left side in Figure C.13.

With a breakpoint set we need to fire up the debugger and run the program to the breakpoint. As shown earlier in Figure C.7

An IPython console window titled "IPython console" with a sub-tab "Console 11024/A". It displays a series of Python commands and their outputs. The commands involve calling a custom median function and the statistics module's median function on different lists. The outputs show that the custom function returns integers for odd-length lists and floats for even-length lists, while the statistics module always returns floats.

```
In [12]: median([1,5,3])
Out[12]: 3

In [13]: import statistics

In [14]: statistics.median([1,5,3])
Out[14]: 3

In [15]: median([1,5,7,3])
Out[15]: 6.0

In [16]: statistics.median([1,5,7,3])
Out[16]: 4.0

In [17]:
```

Figure C.12: No error, but still not correct.

```
import statistics

def median(L):
    '''return median of a list of values'''
    L_length = len(L)
    L_sorted = sorted(L) # Let's not change L
    if L_length%2 == 1: # even length
        return L_sorted[L_length//2]
    else: # odd length
        return (L_sorted[L_length//2] + L_sorted[L_length//2 + 1])/2

L = [1,5,7,3]
print("Median of",L,"is",median(L))
print("Statistics median of",L,"is",statistics.median(L))
```

- 1 set the breakpoint (double-click the line number for line 10)
- 2 check that the Variable Explorer is selected in the upper-right window
- 3 start the debugger (the blue right-arrow and two bars symbol)
- 4 execute to the breakpoint (the blue, double right-arrow symbol)
- 5 when you are done hit the “stop debugging” button

In the Variable Explorer you can see the active variables in the function’s namespace. What is new in the example is that we can double-click on the name of a collection, such as a list, in the Variable Explorer pops up a window with more information.

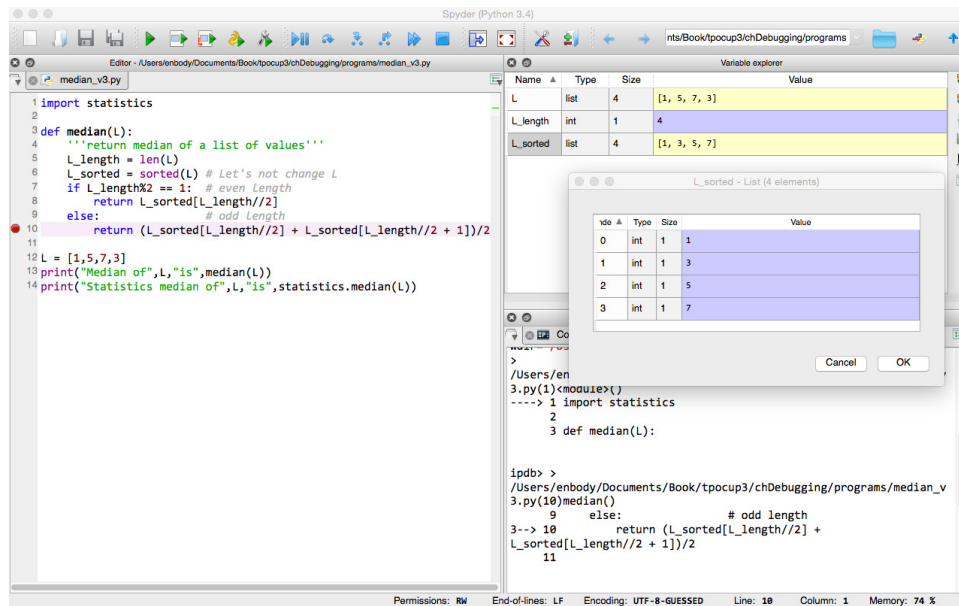


Figure C.13: Collection Pop-up in Variable Explorer.

In our example, you can see `L_sorted` as `[1,3,5,7]` as well as `L_length` whose value is 4. Our index calculations are `L_length//2` and `L_length//2+1` yielding indices of 2 and 3 respectively. Those indices select values 5 and 7 in `L_sorted` rather than the 3 and 5 values we want (indices 1 and 2). Therefore, we are “off by one”, a common problem. We can fix it by subtracting 1 from each index as shown in Code Listing B.9. Since the second index had a “+1” now subtracting 1 cancels that increment.

We now get the correct answer for this list, but we should not stop testing. Next test the edge cases—in this case what happens if we pass an empty list to the median function? It generates an *IndexError: list index out of range* because any index we generate will be out of range. One fix would be to return `None`, if the length of the list is zero. An alternative would be `if not L`.

We are still not done. We need to test more. Let’s generate random lists and use that to explore the debugger further. We’ll begin with lists of length 10 containing 10 random integers, but we can randomize the length, too. Code Listing B.10 adds a `gen_list` function, but with errors so we can explore the debugger.

```

import statistics

def median(L):
    '''return median of a list of values'''
    L_length = len(L)
    L_sorted = sorted(L) # Let's not change L
    if L_length%2 == 1: # even length
        return L_sorted[L_length//2]
    else: # odd length
        return (L_sorted[L_length//2-1] + L_sorted[L_length//2])/2

L = [1,5,7,3]
print("Median of",L,"is",median(L))
print("Statistics median of",L,"is",statistics.median(L))

```

```

import statistics, random

def gen_list(count,max):
    '''return a list of length count of random integers from 0 to max '''
5   L = []
    i = 1
    while i < count:
        L.append(random.randint(0,max))
    return L

10  def median(L):
    '''return median of a list of values'''
    if len(L) == 0:
        return None
15  L_length = len(L)
    L_sorted = sorted(L) # Let's not change L
    if L_length%2 == 1: # even length
        return L_sorted[L_length//2]
    else: # odd length
20  return (L_sorted[L_length//2-1] + L_sorted[L_length//2])/2

L = gen_list(10, random.randint(0,100))
print("L=",L)
print("My median of L is",median(L))
25 print("Statistics median of L is",statistics.median(L))

```

When we run the code in Code Listing B.10 nothing happens—there must be an infinite loop. We could find it by inspection (can you see it?). As before we can set a breakpoint and step through the function one line at a time. There is only one loop and that is in the `gen_list` function so let's put a breakpoint there. The best place is the first line of the loop, line 8 (in this case there is only one line).

Let's review our steps (Refer to Figure C.7:

- 1 set the breakpoint (double-click the line number for line 8)
- 2 check that the Variable Explorer is selected in the upper-right window
- 3 start the debugger (the blue right-arrow and two bars symbol)
- 4 execute to the breakpoint (the blue, double right-arrow symbol)
- 5 step one line at a time (blue curved arrow with one dot)
- 6 when you are done hit the “stop debugging” button

Figure C.14 was grabbed after four items were added to the list. Every time you click the step symbol one line of code is executed. That line is highlighted in the editor window and is indicated in the shell. As you step through the code observe variable changes in the Variable Explorer window in the upper right. What you notice as you step through this code is that the variable `i`, the loop control variable, never changes so that is the reason for the infinite loop. It needs to be incremented within the loop suite at the end.

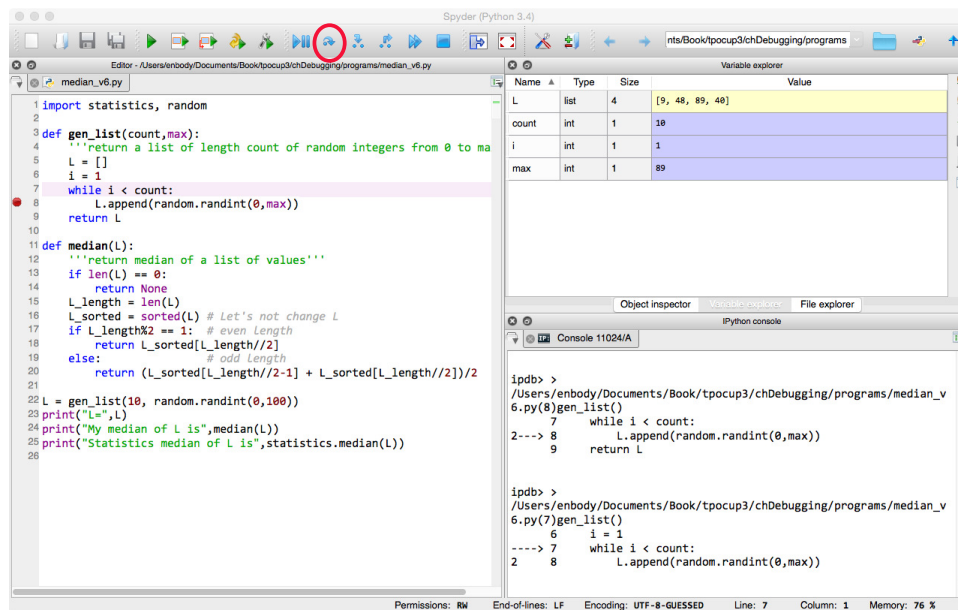


Figure C.14: Stepping one line at a time in the debugger.

Now we generate correct medians, but there is still a problem. We specified lists of length 10, but end up with lists of length 9. Why? (Can you spot it?) The `count` variable is suspect since it controls the loop that builds the lists so let's use the debugger to find this error. This time set *two* breakpoints: one at the top of the loop suite and the other right after the loop as shown in Figure C.15. Everything looked fine when stepping through the loop until it exited too early. Figure C.15 shows what the Variable Explorer looks like after the loop. We see that the value of `i` is 10 so what happened? Our loop-control Boolean is wrong. If with start with `i=1`, the Boolean should be `i <= count`. Alternatively we could start with `i=0` and use `i < count` as the Boolean.

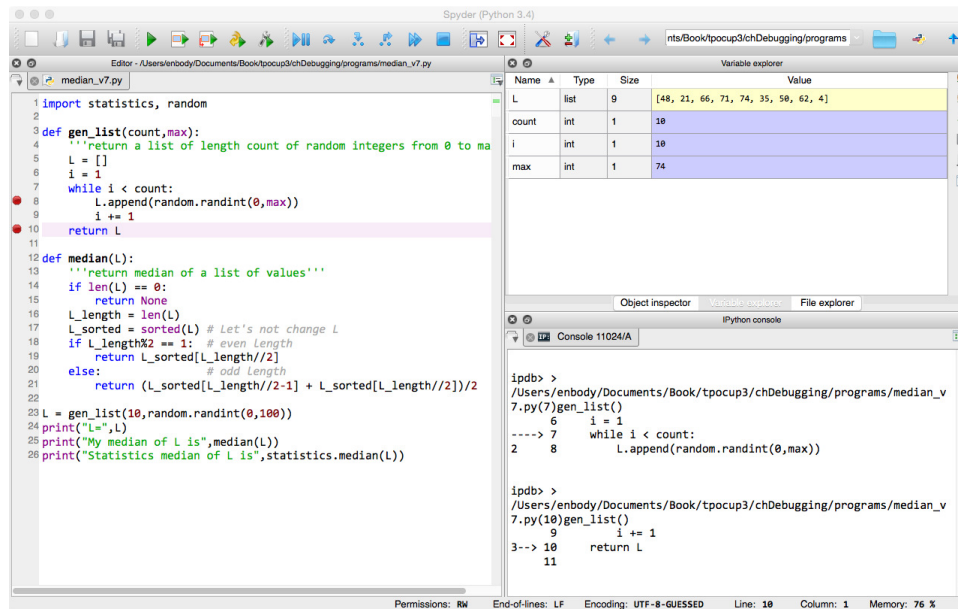


Figure C.15: Two breakpoints.

```

def gen_list(count,max):
    '''return a list of length count of random integers from 0 to max '''
    L = []
    i = 0
    while i < count:
        L.append(random.randint(0,max))
        i += 1
    return L

```

A corrected `gen_list` function is shown in Code Listing B.11.

However, since the loop iterates a fixed number of times a “for” loop would be more appropriate and wouldn’t have that “off by one” error. One could take it a step further to a terser version using list comprehension as is shown in Code Listing B.12.

With `gen_list` working correctly many tests can be run easily to thoroughly test the median function. By including calls to `randint` we can easily generate a variety of test cases. By including calls to `statistics.median` each test case gets compared to a correct solution. In addition, the call to median can be put in a loop to easily generate many test cases.


```


def gen_list(count,max):
    '''return a list of length count of random integers from 0 to max '''
    return [random.randint(0,max) for i in range(count)]

```

C.2.6 More Debugging Tips

Reset How can I reset the Variable Explorer? Look at the shell in the lower right corner and in its upper right corner there is a small symbol that when you click on it gives you a drop-down box with the option to “Interrupt Kernel” or “Restart Kernel”. See Figure A.5.

Step Into  There are three step commands. We have seen the basic “step” command. The “step in” command takes you *into* a function. It can be handy, but can have difficult-to-understand results because sometimes you end up stepping into system or module functions. For example, if there is a simple print statement you will be taken into the messy details of the print operation. If you end up there, stop debugging and restart. It is usually easiest to set a breakpoint where you want to go.

Step Over  The third step command is the “step over” command that takes you past a function. It’s stated operation is “Run until the current function or method returns.” However, like the “step in” command it can take you into the messy details of functions or methods.

C.3 More about Testing

Testing is a very broad category; it would be difficult to cover it in sufficient depth in an introductory programming book. However, it is an interesting topic and one often ignored in introductory books. We think it is important enough to take a quick look. Let’s get started.

There are various philosophies on how, when and why testing should be done. Some of these include:

- Planning versus code testing. This might be better phrased as *static* versus *dynamic* testing. Static testing is work done without running the code. You might think of it as the various stages of understanding the requirements of the code and then planning to achieve those goals. Dynamic testing is the kind of testing done with the code, during execution of the code. It should be noted that in good software development, developing static tests is an important part of the overall software development process. However, this chapter will focus on dynamic testing, that is, testing with code.
- When to test: *post-coding* versus *pre-coding* testing. Traditionally, code is developed and then handed off to an independent testing group. The testing group develops the tests and, when errors are found, passes that information back to the developers, all in a cycle. However, more modern testing approaches, including *agile* and *extreme* programming, focus on developing the tests to be applied to code *before* the code is written. As the code is written and updated, these tests are used as the standard for gauging progress and success.
- Levels of testing. *Unit* testing is focused on the individual pieces (modules, functions, classes) of code as they are developed. *Integration* testing focuses on bringing tested modules together to see whether they work as intended. *System* testing focuses on the entire system to see whether it meets the goals of the project.
- What to test? This is probably a category all unto itself, as there are many goals one might focus on for testing. They would include:
 - **Correctness.** Are the results produced correct? What counts as correct? Do we know all the cases?
 - **Completeness.** Are all the potential cases covered: all user entries, all file formats, all data types, and so on?
 - **Security.** Is the program safe from being broken into by another user, through the Internet, or otherwise?

¹<http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1036.PDF>

- **Interface.** Is the interface usable? Are its features complete? Does the user get what he or she expects when using the program?
- **Load.** Does the program respond well if it is presented with a heavy load: many users, large data sets, heavy Internet traffic, and so on?
- **Resources.** Does the program use appropriate amounts of memory, CPU, network?
- **Responsiveness.** Does the program respond in a reasonable amount of time? If there are time limits, does it meet those limits?

And on and on. Testing is a complicated business!

C.3.1 Testing is Hard!

One of things we want to convey is that testing is hard—at least as hard as writing the program in the first place. That is one of the reasons we have left testing to the end. You have had enough on your hands, learning to design programs, implement them in code, and then fix them.

Correctness

How hard is testing? Let's just pick one of the examples mentioned earlier: correctness. How does one determine whether a program is correct? One type of correctness would be a *proof*. There is a branch of software engineering that focuses on that very topic: taking a program, turning it into a mathematical expression, and proving its correctness. However, that is a very difficult task. In fact, at this time it is not possible to take an arbitrary program and prove it correct. So correctness is often practically measured against a specification of behavior provided by the user. However, even that measurement can be difficult.

In 1993, Intel released its first PentiumTM desktop CPU. It was a new architecture and required many years of development, including new software to make the chip run. In 1994, it was discovered that the floating-point division instruction had a small flaw, due to an error in tables that the algorithm used for its calculations (these tables were stored on the chip). It resulted in a rare error in some divisions. *Byte* magazine in 1994 estimated that about 1 in 9×10^9 divisions of random numbers would show the defect. How many cases should Intel have tested to see whether it was correct? Yet a professor at Lynchburg college performing some mathematical calculations *did* hit the error in his work, rare or not.³ To further illustrate the insidiousness of this example, Intel *had* developed both the correct algorithm and table, but when the table was loaded into the chip, a portion was inadvertently set to zeros!

C.3.2 Importance of Testing

Testing is important. In fact, many modern software developers put testing as the first problem to be solved! As we mentioned earlier, *agile* software development focuses on writing the tests before the code is written. In so doing, a developer accomplishes two very important tasks:

- Focus on what the code *should* do before actually coding the implementation. It relates to our old maxim: think before you program. By writing the tests first, you focus on what the code should do.
- Once tests have been set up, every new piece of code or change to existing code has a test-base in place to assure the developer that their code is correct, at least as far as the measures provided.

Even beginning programmers should try to integrate testing into their development process. Python embeds tools to make this process easier. We will introduce some examples here and hope that you will be exposed to more in your programming career.

C.3.3 Other Kinds of Testing

Python provides a number of support modules for testing. The `unittest` module provides a much greater level of control to run testing. The module `nose`⁴ also provides a greater level of control, and both can do

³See more information at the wikipedia page on the FDIV error, http://en.wikipedia.org/wiki/Pentium_FDIV_bug.

⁴<http://code.google.com/p/python-nose>.

some system testing. In-depth coverage of that module is beyond the scope of an introductory book and is left to the reader.

C.4 What's wrong with my code?

Here we have collected in one place all the suggestions that were at the end of each chapter.

C.4.1 Chapter 1: Beginnings

Here are suggestions for finding problems with Python beginnings.

The most important thing to do is: “Read the Error message!” Learning to read error message helps tremendously when deciphering errors.

Error messages can appear in two places:

- **In the Spyder editor:** an error flag appears to the left of line numbers. You can mouse over error marker to see the error. If not a syntax error, the error is likely a kind of `NameError` (see below).
- **In the shell:** Two types of errors

Static Errors that Spyder can also point out in the editor. These include syntax errors, missing modules and misnamed variables.

Run-time Errors that can only be found when running the program.

Types of Error messages that are likely to occur from topics in this chapter.

- **ValueError** These are often a conversion error, e.g. `float('abc')`.
`ValueError: could not convert string to float: 'abc'`
- **TypeError** These often result from trying to do an operation on two different types which are not permitted in Python. For example, mismatched types, e.g. `'abc' - 2` `TypeError: unsupported operand type(s) for -: 'str' and 'int'` The Variable Explorer shows type so it is a good place to check for helpful information.

Also remember our naming conventions in Section 1.4.6. If you append type information to your variables it is easier to track down this kind of error.

- **NameError** This error usually means that the name is not in the namespace. This often means that you never gave the identifier (name) an initial value. Since Variable Explorer shows your namespace you will see that the name isn't there. There are generally two reasons:

- *Identifier not initialized*, e.g. `x = y #` but `y` never initialized
`NameError: name 'y' is not defined`
- *Forgot import*, e.g. you forgot to import `math` or forgot to prepend `math`
`x = sqrt(y)`
`NameError: name 'sqrt' is not defined`

- **Syntax Error** For example,

```
x === 3
```

```
SyntaxError: invalid syntax
```

In the shell, the message will point to the error.

- **ZeroDivisionError:** self explanatory

There are other errors which either do not generate an error or the error generated is not helpful. Watch for these!

- **Mismatched Parentheses** can occur for many reasons and can be hard to find. If you have many parentheses or wonder if you have them correct, the IDE such as Spyder can help by matching corresponding pairs. One trick is to delete parenthesis at the end of a line or expression and watch the IDE highlight the corresponding one as you add parentheses back in one at a time. Also, Spyder automatically fills in all matching parenthesis as you type. An important thing to keep in mind is that errors with parenthesis can be generated by Python a few lines *after* where the error actually occurred—Python got confused and finally gave up and generated an unhelpful error message.

Note, error messages indicate a line number for the error, but sometimes, especially with a `SyntaxError`, the error actually happens on an earlier line. This is because the interpreter tries hard to understand what was given to it, but gives up at some point, usually down the program from where the error initially occurred.

Finally, be careful that you do not select a variable name that is already in use by Python as a builtin or module name. For example, you might run the following code `str = ``abc```, assigning a value to the variable `str`. The only problem is the Python is using `str` as the name of the string type. However, Python is happy to oblige. During the remainder of your program, `str` is a variable and not a type. Pay attention to **colors** when creating variable names. Spyder colors Python keywords. If a color pops up on your variable, perhaps you should look and see if it is a Python keyword. Finally, if you fix the problem (change the variable name) then, when you start Python up again, the problem will go away.

C.4.2 Chapter 2: Control

Here are suggestions for finding problems with control.

- **Conditional:**

- *Boolean expression error*: test your Boolean expression in the shell.
- *Value errors*: insert a breakpoint on the header line to examine the values in the Boolean expression.
- *Incorrect indentation* is rare: inspection is the only solution.

- **Repetition:**

- *Started wrong*—the initialization expression is wrong: insert a breakpoint (or print the value) right before your Boolean expression first checks it.
- *Stopped wrong*—the Boolean expression is wrong: insert a breakpoint on the loop’s header line and examine the loop-control variable (or print the loop-control variable as the first instruction in the suite). It may be useful to put a breakpoint immediately **after** the loop to examine the final value of the loop-control variable (or print it).
- *Incremented wrong*: the change expression is wrong or in the wrong place. A breakpoint (or print) at the first instruction in the suite is helpful.
- **Problem with the loop variable**: Generally the way to isolate these problems is to observe the loop variable. When you see the values changing that is often a valuable clue to the problem. There are three things you can try:
 - * *Print* the loop variable and watch it change.
 - * *Set a Breakpoint*: An alternative to printing is to break at the beginning of the loop and examine the loop value in the Variable Explorer.
 - * *Print range*: Printing the range values with your expression can help understand what your code is actually iterating over. However, you need to wrap `list(...)` around range to see the values. For example, `print(list(range(5)))`.

Here are some common causes:

- * *Modified loop variable in for*: modifying the loop variable in a *for* loop can cause unexpected problems. That is, if you have `for i in range(5):` do not modify `i` within the loop suite.
- * *Range value is wrong*: a common problem is to forget that `range` generates a half-open interval in that it doesn't include the final value. That is, `range(5)` generates 0, 1, 2, 3, and 4, but **not** 5.
- * *Range value is zero*: sometimes `range(x)` will be a problem because `x` is either zero or `None`.

C.4.3 Chapter 4: Strings

With strings the error message itself is more useful than the type of error message generated. Two types of error messages dominate string errors: `IndexError` and `TypeError`. Many string errors manifest themselves as `TypeError` so it is the error message itself that is most useful.

The two most common error messages are:

- **IndexError: string index out of range** clearly describes the problem. e.g. `s='hello'` and you try `s[15]` whose index is beyond the end of the string `s` (indices 0:4).
- **TypeError: 'str' object does not support item assignment** occurs when indexing appears on the LHS of an assignment statement, e.g. `s[0]='x'`. Since a string is immutable, individual characters cannot be modified and this is the resulting error message.

Less common errors are these; both are `TypeError`s.

- **TypeError: replace() takes at least 2 arguments (1 given)** or something similar occurs when the wrong number of arguments are used for the method. In this case, `replace` expects at least two arguments, but only one was provided: `s.replace('e')`. A similar error can occur for other string methods, or for a function with the wrong number of arguments such as `len()`, i.e. providing no arguments when one is expected.
- **TypeError: unorderable types: str() < int()** occurs when two different types are compared, but such a comparison is not permitted in Python, e.g. `'x'<5`. The problem is obvious in that example, but less obvious when comparing two identifiers, e.g. `x<y` unless you know the types. Observing the types in Spyder's variable explorer can help, as can the naming practices we have described (appending type info to the variable name, see Section 1.4.6).

Other errors which are less likely.

- **String arithmetic errors**: string arithmetic allows some operations, but not others. For example `'hello'*2` is fine; it results in `'hellohello'`. However, the following cause problems:
 - `'hello'*'a'` generates the error: `TypeError: can't multiply sequence by non-int of type 'str'`. While multiplication by an integer is allowed, multiplying a string by another string is not permitted by Python.
 - `'hello'+2` generates the error: `TypeError: Can't convert 'int' object to str implicitly`. While multiplication is allowed, there is no corresponding operation for adding a string and a number. In this case, the interpreter thinks you might be trying to add two strings together (something it understands) and that the second argument, a number, should be converted to a string. However, as the error message indicates, there isn't a rule for automatically converting a number to a string. If your intent is to convert the number to a string, then you must do so explicitly: `'hello'+str(2)`.
- **String formatting errors** come from a mismatch of either types or number of items to be formatted.

- **Type mismatch** occurs when the format specifier type doesn't match the provided item, e.g. specifying an integer format ("`{:d}`") while providing a string argument, i.e. `"{:d}".format("abc")`. The resulting error message is: `ValueError: Unknown format code 'd' for object of type 'str'` which effectively says that the 'd' format isn't defined for use with a string (it is for use with an integer).
- **Wrong number of arguments** occurs when you specify say two items to format, but there is a mismatch in the number of format elements and arguments e.g. `"{}{}".format("a")`. The resulting error message seems unrelated: `IndexError: tuple index out of range`. We don't describe 'tuples' until a later chapter, but the argument to the format method is implicitly a tuple and this message is saying that two items were expected, but only one was found, hence "out of range."
- **Missing Parenthesis in Method:** Some string methods such as `lower()` and `upper()` take no arguments so it is easy to forget the parenthesis. If you type such an expression into the interpreter you don't get an error message. Instead you get unexpected output such as `<function str.upper>` or `<built-in method upper of str object at 0x106e85998>`. Effectively you not asking Python to **run** the method, which would require the parentheses, but are instead asking Python to **report** what "lower" is, which Python dutifully reports.

Even worse would be the following code: `f = `abc`.lower`. Again, no error occurs but what is assigned to the variable `f` is the **function** "lower", not the result of **running** lower. Probably not what you wanted

C.4.4 Chapter 5: Functions

Here are suggestions for finding problems with functions.

- **Test Functions Independently:** A particularly useful trick is the ability to run individual functions within Spyder. That allows us to test functions independently. Within the Spyder IDE highlight the function and under the "Run" drop-down select "Run Selection or current line" to run the highlighted function. After it has been "run" you can now call the function from within the shell. Run the function with sample arguments and see if it produces the results you expect. **Testing functions independently is one of the most important approaches to programming you can learn!**
- **Examine the Function's Namespace:** Setting a breakpoint within a function allows you to examine the function's namespace within the Variable Explorer window of Spyder (upper right of IDE). Knowing the value of variables, particularly the provided arguments, is useful for finding problems.
- **Stepping through a Function:** Set a breakpoint at the beginning of the function and then step through the function, one instruction at a time. This is useful for observing what is the function is actually doing. Pay particular attention to which parts of control (loops or selection) that you step through. Often there is a problem with a Boolean expression or the values of variables within the expression.
- **Mismatched arguments and parameters:** This kind of error message is self explanatory and helpful with its information. Here is an example.
`TypeError: square() missing 1 required positional argument: 'x'`
- **None:** When a variable has an unexpected value such as `None` of a function definition such as `<function str.lower>` it is often due to that variable being incorrectly assigned a value from a function call in one of two ways: First, mistakenly assigning the result of a function call that **has not return** i.e., `result = [1,3,2].sort()`. Such non-return functions are usually associated with mutable data structures such as lists. The other is forgetting the parenthesis in the function call i.e., `f = `ABC.lower``. The function was never **run** and you don't get the result you expected.

When you see such an error, walk the code back to find the function that provided a value to that variable. It likely is missing a return statement.

- **TypeError:** Sometimes you get a **TypeError** when there is mismatch of types between the provided argument and how that argument is used in the function. For example, maybe you passed a string to a function that is performing integer arithmetic that is not defined for strings. Again, either the Variable Explorer or proper naming can help sort out this problem.

C.4.5 Chapter 6: Files and Exceptions

Here are suggestions for finding problems with files.

Here are three common errors or problems you will find when opening and writing to files:

- **FileNotFoundError:** The error message describes the problem, but not what caused it. Here are the two most common causes:
 - **Misspelled file name:** If you misspelled the file name, Python cannot find it.
 - **File not in the same folder as the program:** By default a program looks for a file in the same directory as the program. If it isn't there, Python cannot find it. If the file is in another directory by choice, you must give a fully qualified path to that file, i.e. `"/Users/bill/Documents/file.txt"`
- Remember that the ipython console can help with the previous two errors in two ways:
 - In the console itself, when you **Run** your edited file a line shows up in the console of the form: `runfile('/Users/user/python/file.py', wdir='/Users/user/python')` where the `wdir` is the working directory, the assumed location of any files you are trying to open. Is that the correction working directory?
 - You can use the ipython history commands (the up and down arrow on your keyboard) to get the previous command back and fix any misspellings of the filename.
- **Empty or partially written file:** If you forget to `close` a file, some of what you wrote to the file might be missing. Python tries to deal with this, but it is not always successful. *Always remember to close files!*
- **Accidentally overwriting a file:** Python does not warn you if you are about to overwrite a file. For example, if you open a file with `"w"` as the modifier, the contents of the file, if it already exists, will be lost.

Here are some errors you might run into when reading the contents of a file, especially when iterating through the file's contents line by line.

- **TypeError:** Remember that what is returned by iterating through a file is a string. Even if that file's contents contains a number on that particular line, that number is returned as a **string**, not an integer (or a float, or anything else). You must explicitly convert the numeric-string to its appropriate arithmetic value before you can use it in an arithmetic calculation.
- **Extra Line:** Not a Python error per se, but if you print out your lines of input and get an extra line (like double spacing), what happened? Did you remember to `strip` your line when you got it? Otherwise it is carrying a extra ```n''`.
- **IndexError:** Remember that some of the file lines may only have a ```n''`, basically a blank line in the file. When you `strip` such a line you are left with an empty string. You cannot index any character in an empty string as it has none, thus leading to the **IndexError**.

C.4.6 Chapter 7: Lists and Tuples

Here are suggestions for finding problems with lists.

As a sequence, Lists have similar issues with ranges as we saw in strings. Tuples, being immutable, cannot have their elements changed, as we also saw in strings. Lists do not have as many methods (11) as strings (44) so there are fewer errors associated with method use. However, there are some errors that tend to happen with lists that are really control problems.

- **None:** a common way to end up with `None` with lists is to combine assignment with append, e.g. `x=my_list.append(y)` . Since append has a side effect of updating the end of the string and **no return**, the value of x will be `None` . A similar thing happens by mixing up the `sorted` function (which returns a sorted list) vs. the `sort` method (with has a side effect of sorting the list and no return). The problem manifests itself with `x=my_list.sort(y)` which results in x having a value of `None` .
- **[]:** an unexpected empty list is usually a control error—your `append` statement was never executed. Usually that is a problem with the Boolean expression or its values. Set a breakpoint and check Variable Explorer for errant values.
- **IndexError:** as with other collections such as strings an `IndexError` comes from two sources:
 - The list has a different length than expected. Often the list is empty, but another common problem is “off by one”, i.e. the list is one element smaller or larger than expected. Remember, indices in lists begin with 0 and the last index is at length - 1. This is often a control problem so set a breakpoint and examine your Boolean expression.
 - The index variable is wrong—again, “off by one” is common.
- **Tuples are immutable** so item assignment is not allowed. This problem is similar to what happens with strings, the other immutable collection. For example, `my_tuple[2]=5` will generate this error: `TypeError: 'tuple' object does not support item assignment.`
- **IndexError:** Finally, there is a way to generate an `IndexError` other than “off by one” error or the like. Unlike other sequences, Lists are mutable. Thus, as you iterate through a list it is possible to modify the list as you iterate. However, **never modify a mutable during iteration**. We mentioned this as a programming tip in Section 7.2.4 but `IndexError` is one way this manifests itself.

C.4.7 Chapter 8: More Functions

The problems we encounter here all center on the use of mutable parameter objects such as lists.

- If you pass a mutable, such as a list, to a function, then a change in that mutable is preserved across the call. That is, the mutable **argument** in the calling program will be changed by any change to that mutable **parameter** in the function. You can check for this by setting a breakpoint before and after the function call in Spyder and watching how the mutable value is updated.
- Another problem discussed in some detail in Section 8.2.1 is using a mutable, such as a list, as a default parameter value. The short answer is **don't**. A default value is created once, and only once, when the function is defined. If the default is modified in the function, then as a mutable that change persists and you get changing default values. As mentioned, if you need a mutable default value, assign a value of `None` as the default and check for it in the function code.

C.4.8 Chapter 9: Dictionaries

Here are suggestions for finding problems with dictionaries, sets and scope.

- **Dictionaries**

- **TypeError: ‘builtin_function_or_method’ object is not iterable:** happens when you forget the parenthesis in `for k,v in D.items():` . Similarly for the `keys()` or `values()` methods. You must invoke the method (use parentheses) to generate the results you require.
- **KeyError** occurs when the key doesn’t exist in the dictionary, similar to “out of range” for lists and strings. The error message prints the errant key.
- **Empty dictionary:** When you unexpectedly find that your dictionary has no values, that is a *control* error. Suggestions include setting a breakpoint and examining your control Boolean expression and values.
- **Unexpected value:** If you find unexpected values in a dictionary, you may have overwritten an existing key,value pair by reusing a key with a new value. Remember, dictionaries are mutable and the key can be updated with a new value.
- **TypeError: unhashable type: ‘list’:** happens when you use a mutable for a key. Remember, you cannot use mutable items as a key in a dictionary. As in this example error, you cannot use a List as a key.

- **Sets**

- **AttributeError: ‘dict’ object has no attribute ‘add’:** happens when you initialized a set with `s={}` instead of `s=set()` , i.e. you created a dictionary when you wanted a set. You can observe the type in the Variable Explorer.

- **Scope**

- **Global:** The most common “error” with respect to scope is using global variable references. A global variable is set in the outermost scope of the program and available to all other program elements. While not specifically an error, resolving why a global variable is being properly set can be difficult to debug. It is better to pass arguments to functions specifically to indicate the values being operated on by the function, making debugging easier. Setting a breakpoint at the end of a function lets you observe a function’s namespace in Variable Explorer.
More difficult is the **shadowing problem** discussed in Section 9.6. You can have the same name in different namespaces, and the **LEGB** rule is used to resolve the value. Thus you may try to refer to a variable’s value and not get the expected result because **LEGB** resolved the value in an unexpected way. In particular, if you assign a variable inside of a function, that variable is now **local to the function namespace**. Again, passing arguments explicitly to functions can help make debugging the problem easier.
- **NameError** for a name you know exists usually occurs because the name is in a different namespace. Variable Explorer is helpful for seeing what names are in the current namespace.
- **UnboundLocalError** can occur when you unintentionally reference a global variable within a function. In this case you get a helpful error message: *UnboundLocalError: local variable ‘i’ referenced before assignment*. How did this happen? Consider the following code. In the `test()` function the `i + 1` on the RHS refers to `i` which is local to the function because `i` first occurs on the LHS, but the local name has not been initialized.

```
def test():  
    i = i + 1  
  
i = 0  
test()
```


C.4.9 Chapter11: Classes I

Here are suggestions for finding problems with classes.

- **Forgot self:** If you forget `self` as the first parameter in a method you will get a mismatch in the number of parameters. Remember, **every** class method has `self` as its first parameter, even if the method takes no arguments when it is called. Python **automatically** associates the calling object and passes it as the first method parameter. The error message looks like this (for a method named `my_method`): **TypeError: my_method() takes 0 positional arguments but 1 was given**
- **Method `__str__` must return a string:** If you forget to return a string from the `__str__` method, you will get this message: **TypeError: __str__ returned non-string (type int)**
- **AttributeError** often happens because you have mistyped the name of a method or an attribute is private.
 - If you mistyped the method, you can use your iPython history to bring it back and retype it.
 - If you can't remember the proper spelling, you can make an instance of the class, put a dot behind it and type the tab key. iPython will list all the potential methods.
 - If that doesn't work, remember the privacy rule covered in Section 11.9.1. If you prepend double underscores to an attribute name, special naming rules apply. For example, if the attribute is named `__my_attribute` in the class `MyClass`, the following are the rules:
 - * **Inside MyClass:** The name remains `__my_attribute`, including in the context of any other `MyClass` methods.
 - * **Outside MyClass:** The name outside of the `MyClass` namespace, you refer to the attribute as `_MyClass__my_attribute`.
- **NameError** often happens with you do not specify a class attribute correctly. This often occurs when you forget to prepend `self` to the name of a method in the class.
- **Forgot `__init__`:** If you forget to define an `__init__` method, you may get either unexpected results or an error such as an **AttributeError**. A class definition is not required to have an `__init__` method defined (a default one exists), in which case class instances are created with no attributes. However, by having `__init__` in your class, you can define, by assignment, attributes that you want to exist in every instance. Set a breakpoint after an instance is created to make sure it contains the attributes you expect. If not, check your `__init__`, or create one if you don't have one.

C.4.10 Chapter 12: Classes II

Here are suggestions for finding problems with classes.

- **Return class type with arithmetic methods:** The most common mistake with arithmetic methods such as `__add__` is to forget to return objects of the class type. For example, if you add two Rational numbers, you expect the result to be a Rational number. The solution is `return Rational(num,denom)` where you create a Rational object before you return it.

Summary

In this chapter, we introduced the built-in debugging and testing capabilities of Python. These techniques assist in developing better code.