3. Debugging

3.1. How to be a Successful Programmer

3.2. How to Avoid Debugging

3.3. Beginning tips for Debugging

3.4. Know Your Error Messages

3.4.1. ParseError

3.4.2. TypeError

3.4.3. NameError

3.4.4. ValueError

3.5. Summary

3.6. Exercises

3.1. How to be a Successful Programmer

One of the most important skills you need to aquire to complete this book successfully is the ability to debug your programs. Debugging might be the most under-appreciated, and under-taught, skill in introductory computer science. For that reason we are introducing a series of “debugging segments” Debugging is a skill that you need to master over time, and some of the tips and tricks are specific to different aspects of Python programming. So look for additional debugging segments throughout the rest of this book.

Programming is an odd thing in a way. Here is why. As programmers we spend 99% of our time trying to get our program to work. We struggle, we stress, we spend hours deep in frustration trying to get our program to execute correctly. Then when we do get it going we celebrate, hand it in, and move on to the next homework assignment or programming task. But here is the secret, when you are successful, you are happy, your brain releases a bit of chemical that makes you feel good. You need to organize your programming so that you have lots of little successes. It turns out your brain doesn’t care all that much if you have successfully written hello world, or a fast fourier transform (trust me its hard) you still get that little release that makes you happy. When you are happy you want to go on and solve the next little problem. Essentially I’m telling you once again, start small, get something small working, and then add to it.

3.2. How to Avoid Debugging

Perhaps the most important lesson in debugging is that it is largely avoidable – if you work carefully.

Understand the Problem You must have a firm grasp on what you are trying to accomplish but not necessarily how to do it. You do not need to understand the entire problem. But you must understand at least a portion of it and what the program should do in a specific circumstance – what output should be produced for some given input. This will allow you to test your progress. You can then identify if a solution is correct or whether there remains work to do or bugs to fix. This is probably the single biggest piece of advice for programmers at every level.

Start Small It is tempting to sit down and crank out an entire program at once. But, when the program – inevitably – does not work, you have a myriad of options for things that might be wrong. Where to start? Where to look first? How to figure out what went wrong? I’ll get to that in the next section. So, start with something really small. Maybe just two lines and then make sure that runs. Hitting the run button is quick and easy. It gives you immediate feedback about whether what you have just done works or not. Another immediate benefit of having something small working is that you have something to turn in. Turning in a small, incomplete program, is almost always better than nothing.

Keep Improving It Once you have a small part of your program working, the next step is to figure out something small to add to it – how can you move closer to a correct solution. As you add to your program, you gain greater insight into the underlying problem you are trying to solve.

If you keep adding small pieces of the program one at a time, it is much easier to figure out what went wrong. (This of course means you must be able to recognize if there is an error. And that is done through testing.)

As long as you always test each new bit of code, it is most likely that any error is in the new code you have just added. Less new code means its easier to figure out where the problem is.

This notion of Get something working and keep improving it is a mantra that you can repeat throughout your career as a programmer. It’s a great way to avoid the frustrations mentioned above. Think of it this way. Every time you have a little success, your brain releases a tiny bit of chemical that makes you happy. So, you can keep yourself happy and make programming more enjoyable by creating lots of small victories for yourself.

Note

The technique of starting small and keep improving is the basis of Agile software development. This practice is used widely in the industry.

Ok, let’s look at an example. Let’s solve the problem posed in question 3 at the end of the Simple Python Data chapter. Ask the user for the time now (in hours 0 - 23), and ask for the number of hours to wait. Your program should output what the time will be on the clock when the alarm goes off. For example, if current\_time is 8 and wait\_time is 5, final\_time should be 13 (1 pm).

So, where to start? The problem requires two pieces of input from the user, so let’s start there and make sure we can get the data we need.

1 current\_time = input("what is the current time (in hours)?")

2 wait\_time = input("How many hours do you want to wait")

3

​4 print(current\_time)

5 print(wait\_time)

6

​

So far so good. Now let’s take the next step. We need to figure out what the time will be after waiting wait\_time number of hours. A reasonable solution is to simply add wait\_time to current\_time and print out the result. So lets try that.

1 current\_time = input("What is the current time (in hours 0 - 23)?")

2 wait\_time = input("How many hours do you want to wait")

3

​4 print(current\_time)

5 print(wait\_time)

6

​7 final\_time = current\_time + wait\_time

8 print(final\_time)

9

​

Hmm, when you run this example you see that something unexpected has happened. You would not realize this was an error unless you first knew what the program was supposed to do.

debug-2-3: Which of the following best describes what is wrong with the previous example?

A. Python is stupid and does not know how to add properly.

B. There is nothing wrong here.

C. Python is doing string concatenation, not integer addition.

This error was probably pretty simple to spot, because we printed out the value of final\_time and it is easy to see that the numbers were just concatenated together rather than added.

So what do we do about the problem? We will need to convert both current\_time and wait\_time to int. At this stage of your programming development, it can be a good idea to include the type of the variable in the variable name itself. So let’s look at another iteration of the program that does that, and the conversion to integer.

1 current\_time\_str = input("What is the current time (in hours 0-23)?")

2 wait\_time\_str = input("How many hours do you want to wait")

3

​4 current\_time\_int = int(current\_time\_str)

5 wait\_time\_int = int(wait\_time\_str)

6

​7 final\_time\_int = current\_time\_int + wait\_time\_int

8 print(final\_time\_int)

9

Now, that’s a lot better, and in fact depending on the hours you chose, it may be exactly right. If you entered 8 for current\_time and 5 for wait\_time then 13 is correct. But if you entered 17 (5 pm) for current\_time and 9 for wait\_time then the result of 26 is not correct.

This illustrates an important aspect of testing: it is important to test your code on a range of inputs. It is especially important to test your code on boundary conditions. For this particular problem, you should test your program with current\_time of 0, 23, and some values in between. You should test your wait\_time for 0, and some larger values. What about negative numbers? Negative numbers don’t make sense, and since we don’t really have the tools to deal with telling the user when something is wrong we will not worry about that just yet.

So to account for those numbers that are bigger than 23, we need one final step: using the modulus operator.

1 current\_time\_str = input("What is the current time (in hours 0-23)?")

2 wait\_time\_str = input("How many hours do you want to wait")

3

​4 current\_time\_int = int(current\_time\_str)

5 wait\_time\_int = int(wait\_time\_str)

6

​7 final\_time\_int = current\_time\_int + wait\_time\_int

8

​9 final\_answer = final\_time\_int % 24

10

​11 print("The time after waiting is: ", final\_answer)

12

Of course even in this simple progression, there are other ways you could have gone astray. We’ll look at some of those and how you track them down in the next section.

3.3. Beginning tips for Debugging

Debugging a program is a different way of thinking than writing a program. The process of debugging is much more like being a detective. Here are a few rules to get you thinking about debugging.

Everyone is a suspect (Except Python)! It’s common for beginner programmers to blame Python, but that should be your last resort. Remember that Python has been used to solve CS1 level problems millions of times by millions of other programmers. So, Python is probably not the problem.

Find clues. This is the biggest job of the detective and right now there are two important kinds of clues for you to understand.

Error Messages

Print Statements

3.4. Know Your Error Messages

Many problems in your program will lead to an error message. For example as I was writing and testing this chapter of the book I wrote the following version of the example program in the previous section.

current\_time\_str = input("What is the current time (in hours 0-23)?")

wait\_time\_str = input("How many hours do you want to wait")

current\_time\_int = int(current\_time\_str)

wait\_time\_int = int(wait\_time\_int)

final\_time\_int = current\_time\_int + wait\_time\_int

print(final\_time\_int)

Can you see what is wrong, just by looking at the code? Maybe, maybe not. Our brain tends to see what we think is there, so sometimes it is very hard to find the problem just by looking at the code. Especially when it is our own code and we are sure that we have done everything right!

Let’s try the program again

1 current\_time\_str = input("What is the current time (in hours 0-23)?")

2 wait\_time\_str = input("How many hours do you want to wait")

3

​4 current\_time\_int = int(current\_time\_str)

5 wait\_time\_int = int(wait\_time\_int)

6

​7 final\_time\_int = current\_time\_int + wait\_time\_int

8 print(final\_time\_int)

9

​

Aha! Now we have an error message that might be useful. The name error tells us that wait\_time\_int is not defined. It also tells us that the error is on line 5. That’s really useful information. Now look at line five and you will see that wait\_time\_int is used on both the left and the right hand side of the assignment statement.

debug-4-2: Which of the following explains why wait\_time\_int = int(wait\_time\_int) is an error?

A. You cannot use a variable on both the left and right hand sides of an assignment statement.

B. wait\_time\_int does not have a value so it cannot be used on the right hand side.

C. This is not really an error, Python is broken.

In writing and using this book over the last few years we have collected a lot of statistics about the programs in this book. Here are some statistics about error messages for the exercise we have been looking at.

ParseError:

4999

54.74%

TypeError:

1305

14.29%

NameError:

1009

11.05%

ValueError:

893

9.78%

URIError:

334

3.66%

TokenError:

244

2.67%

SyntaxError:

227

2.49%

TimeLimitError:

44

0.48%

IndentationError:

28

0.31%

AttributeError:

27

0.30%

ImportError:

16

0.18%

IndexError:

6

0.07%

Nearly 90% of the error messages encountered for this problem are ParseError, TypeError, NameError, or ValueError. We will look at these errors in three stages:

First we will define what these four error messages mean.

Then, we will look at some examples that cause these errors to occur.

Finally we will look at ways to help uncover the root cause of these messages.

3.4.1. ParseError

Parse errors happen when you make an error in the syntax of your program. Syntax errors are like making grammatical errors in writing. If you don’t use periods and commas in your writing then you are making it hard for other readers to figure out what you are trying to say. Similarly Python has certain grammatical rules that must be followed or else Python can’t figure out what you are trying to say.

Usually ParseErrors can be traced back to missing punctuation characters, such as parentheses, quotation marks, or commas. Remember that in Python commas are used to separate parameters to functions. Parentheses must be balanced, or else Python thinks that you are trying to include everything that follows as a parameter to some function.

Here are a couple examples of Parse errors in the example program we have been using. See if you can figure out what caused them.

Find and fix the error in the following code.

1 current\_time\_str = input("What is the current time (in hours 0-23)?")

2 wait\_time\_str = input("How many hours do you want to wait"

3

​

4 current\_time\_int = int(current\_time\_str)

5 wait\_time\_int = int(wait\_time\_str)

6

​

7 final\_time\_int = current\_time\_int + wait\_time\_int

8 print(final\_time\_int)

9

Finding Clues How can you help yourself find these problems? One trick that can be very valuable in this situation is to simply start by commenting out the line number that is flagged as having the error. If you comment out line four, the error message now changes to point to line 5. Now you ask yourself, am I really that bad that I have two lines in a row that have errors on them? Maybe, so taken to the extreme, you could comment out all of the remaining lines in the program. Now the error message changes to TokenError: EOF in multi-line statement This is a very technical way of saying that Python got to the end of file (EOF) while it was still looking for something. In this case a right parenthesis.

Find and fix the error in the following code.

1 current\_time\_str = input("What is the "current time" (in hours 0-23)?")

2 wait\_time\_str = input("How many hours do you want to wait")

3

​

4 current\_time\_int = int(current\_time\_str)

5 wait\_time\_int = int(wait\_time\_str)

6

​

7 final\_time\_int = current\_time\_int + wait\_time\_int

8 print(final\_time\_int)

9

Finding Clues If you follow the same advice as for the last problem, comment out line one, you will immediately get a different error message. Here’s where you need to be very careful and not panic. The error message you get now is: NameError: name 'current\_time\_str' is not defined on line 4. You might be very tempted to think that this is somehow related to the earlier problem and immediately conclude that there is something wrong with the variable name current\_time\_str but if you reflect for a minute you will see that by commenting out line one you have caused a new and unrelated error. That is you have commented out the creation of the name current\_time\_str. So of course when you want to convert it to an int you will get the NameError. Yes, this can be confusing, but it will become much easier with experience. It’s also important to keep calm, and evaluate each new clue carefully so you don’t waste time chasing problems that are not really there.

Uncomment line 1 and you are back to the ParseError. Another track is to eliminate a possible source of error. Rather than commenting out the entire line you might just try to assign current\_time\_str to a constant value. For example you might make line one look like this: current\_time\_str = "10" #input("What is the "current time" (in hours 0-23)?"). Now you have assigned current\_time\_str to the string 10, and commented out the input statement. And now the program works! So you conclude that the problem must have something to do with the input function.

3.4.2. TypeError

TypeErrors occur when you ry to combine two objects that are not compatible. For example you try to add together an integer and a string. Usually type errors can be isolated to lines that are using mathematical operators, and usually the line number given by the error message is an accurate indication of the line.

Here’s an example of a type error created by a Polish learner. See if you can find and fix the error.

1 a = input('wpisz godzine')

2 x = input('wpisz liczbe godzin')

3 int(x)

4 int(a)

5 h = x // 24

6 s = x % 24

7 print (h, s)

8 a = a + s

9 print ('godzina teraz', a)

10

​

Finding Clues One thing that can help you in this situation is to print out the values and the types of the variables involved in the statement that is causing the error. You might try adding a print statement after line 4 print(x, type(x)) You will see that at least we have confirmed that x is of type string. Now you need to start to work backward through the program. You need to ask yourself, where is x used in the program? x is used on lines 2, 3, and of course 5 and 6 (where we are getting an error). So maybe you move the print statement to be after line 2 and again after 3. Line 3 is where you expect the value of x to be changed to an integer. Could line 4 be mysteriously changing x back to a string? Not very likely. So the value and type of x is just what you would expect it to be after line 2, but not after line 3. This helps you isolate the problem to line 3. In fact if you employ one of our earlier techniques of commenting out line 3 you will see that this has no impact on the error, and is a big clue that line 3 as it is currently written is useless.

3.4.3. NameError

Name errors almost always mean that you have used a variable before it has a value. Often NameErrors are simply caused by typos in your code. They can be hard to spot if you don’t have a good eye for catching spelling mistakes. Other times you may simply mis-remember the name of a variable or even a function you want to call. You have seen one example of a NameError at the beginning of this section. Here is another one. See if you can get this program to run successfully:

1 str\_time = input("What time is it now?")

2 str\_wait\_time = input("What is the number of hours to wait?")

3 time = int(str\_time)

4 wai\_time = int(str\_wait\_time)

5

​6 time\_when\_alarm\_go\_off = time + wait\_time

7 print(time\_when\_alarm\_go\_off)

8

Finding Clues With name errors one of the best things you can do is use the editor, or browser search function. Quite often if you search for the exact word in the error message one of two things will happen:

The word you are searching for will appear only once in your code, it’s also likely that it will be on the right hand side of an assignment statement, or as a parameter to a function. That should confirm for you that you have a typo somewhere. If the name in question is what you thought it should be then you probably have a typo on the left hand side of an assignment statement on a line before your error message occurs. Start looking backward at your assignment statements. In some cases it’s really nice to leave all the highlighted strings from the search function visible as they will help you very quickly find a line where you might have expected your variable to be highlighted.

The second thing that may happen is that you will be looking directly at a line where you expected the search to find the string in question, but it will not be highlighted. Most often that will be the typo right there.

Here is another one for you to try:

1 n = input("What time is it now (in hours)?")

2 n = imt(n)

3 m = input("How many hours do you want to wait?")

4 m = int(m)

5 q = m % 12

6 print("The time is now", q)

7

And one last bit of code to fix.

1 present\_time = input("Enter the present timein hours:")

2 set\_alarm = input("Set the hours for alarm:")

3 int (present\_time, set\_time, alarm\_time)

4 alarm\_time = present\_time + set\_alarm

5 print(alarm\_time)

6

​

Activity: 3.4.3.3 ActiveCode (db\_ex3\_11)

3.4.4. ValueError

Value errors occur when you pass a parameter to a function and the function is expecting a certain limitation on the values, and the value passed is not compatible. We can illustrate that with this particular program in two different ways.

1 current\_time\_str = input("What is the current time (in hours 0-23)?")

2 current\_time\_int = int(current\_time\_str)

3

​4 wait\_time\_str = input("How many hours do you want to wait")

5 wait\_time\_int = int(wait\_time\_int)

6

​7 final\_time\_int = current\_time\_int + wait\_time\_int

8 print(final\_time\_int)

9

Run the program but instead of typing in anything to the dialog box just click OK. You should see the following error message: ValueError: invalid literal for int() with base 10: '' on line: 4 This error is not because you have made a mistake in your program. Although sometimes we do want to check the user input to make sure it’s valid, however, we don’t have all the tools we need for that yet. The error happens because the user did not give us something we can convert to an integer, instead we gave it an empty string. Try running the program again. Now this time enter “ten” instead of the number 10. You will get a similar error message.

ValueErrors are not always caused by user input error, but in this program that is the case. We’ll look again at ValueErrors again when we get to more complicated programs. For now it is worth repeating that you need to keep track of the restrictions needed for your variables, and understand what your function is expecting. You can do this by writing comments in your code, or by naming your variables in a way that reminds you of their proper form.

3.5. Summary

Make sure you take the time to understand error messages. They can help you a lot.

print statements are your friends. Use them to help you uncover what is really happening in your code.

Work backward from the error. Many times an error message is caused by something that has happened before it in the program. Always remember that python evaluates a program top to bottom.