W3.0.1 Additional Reading (Introduction to Debugging)

Learning Outcomes

This post-class activity will spend some time talking about what happens when errors occur as well as how to fix the errors that you will inevitably come across. This is called debugging, afterwards, we will also introduce you to some of the very first control structures you will learn when it comes to programming (i.e., conditional and while loop statements).

By the end of this lesson you should:

- understand good programming strategies to avoid errors
- understand common kinds of exceptions and their likely causes

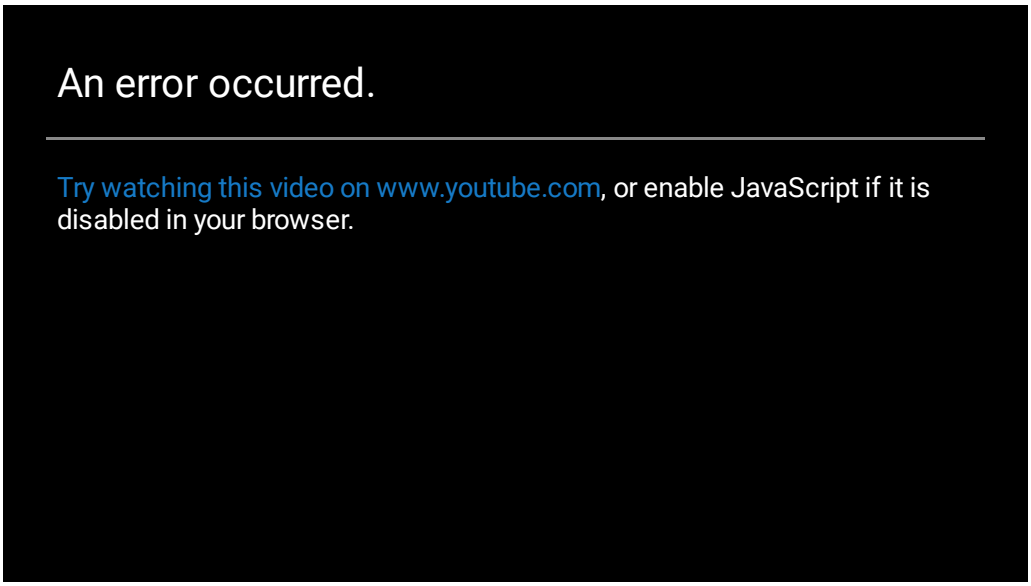
3.1 Introduction to Debugging

> "The art of debugging is figuring out what you really told your program to do rather than what you thought you told it to do." — Andrew Singer

Before computers became digital, debugging could mean looking for insects impeding the functioning of physical relays as in this somewhat apocryphal tale about Admiral Grace Hopper, a pioneer of computer programming.

Nowadays, debugging doesn't involve bug guts all over your computer but it can still be just as frustrating. To cope with this frustration, we will present some strategies to help you understand why the program you wrote does not behave as intended.

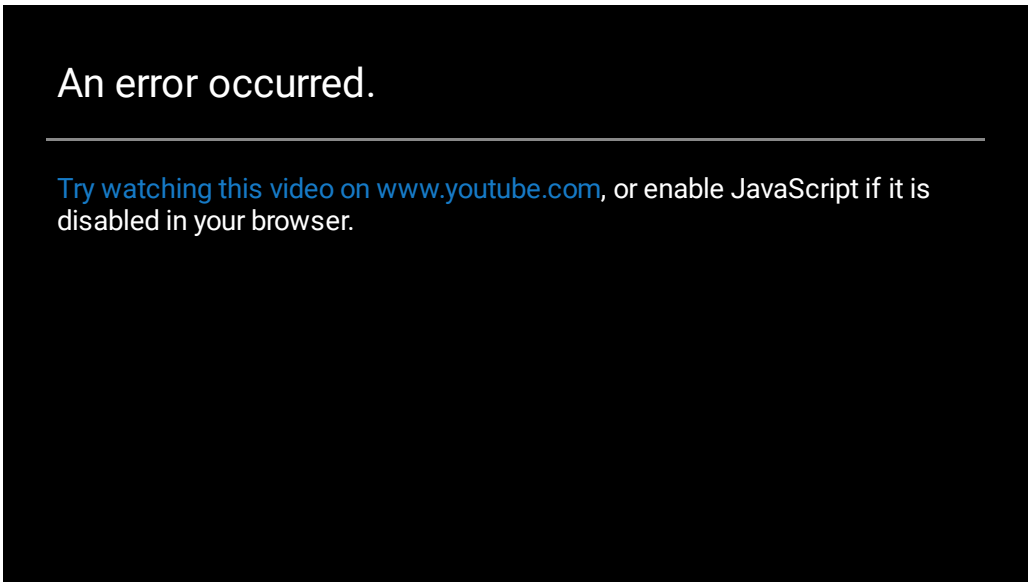
Many people think debugging is some kind of punishment for not being smart enough to write code correctly the first time. But nobody does that, failure in programming is part of the deal. Here's a fun video to keep in mind as you learn to program.



Source: [Youtube: TED]

3.2 Programming in the Real World

Before we dive into the nitty gritty details of debugging, here is a video to give you a flavor for what its like to be a programmer in the real world.



Source: [Youtube: Google Students]

Debugging

Programming is a complex process. Since it is done by human beings, errors may often occur. Programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**. Some claim that in 1945, a dead moth caused a problem on relay number 70, panel F, of one of the first computers at Harvard, and the term **bug** has remained in use since. For more about this historic event, see first bug.

One of the most important skills you need to acquire 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 interludes." 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 Way of the Programmer interludes 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 successess. 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.

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 start small and keep improving is the basis of **Agile** software development. This practice is used widely in the industry.

Ok, lets look at an example. Lets solve the problem posed below. 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.

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

```
current_time = input("what is the current time (in hours)?")
wait_time = input("How many hours do you want to wait")

print(current_time)
print(wait_time)
```

If you haven't yet, click Run: get in the habit of checking whether small things are working before you go on.

So far so good. Now lets take the next step. We need to figure out what the time will be after waiting `wait_time` number of hours. A good first approximation to that is to simply add `wait_time` to `current_time` and print out the result. So lets try that.

```
current_time = input("what is the current time (in hours 0--23)?")
wait_time = input("How many hours do you want to wait")

print(current_time)
print(wait_time)

final_time = current_time + wait_time
print(final_time)
```

When you run that example you see that something funny has happened. Python is doing string concatenation instead of 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 lets look at another iteration of the program that does that, and the conversion to integer.

```
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_str)

final_time_int = current_time_int + wait_time_int
print(final_time_int)
```

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 the current time and 5 for the wait time then 13 is correct. But if you entered 17 (5pm) for the hours and 9 for the wait time then the result of 26 is not correct. This illustrates an important

aspect of **testing**, which is that it is important to test your code on a range of inputs. It is especially important to test your code on **boundary conditions**. In this case you would want to test your program for hours including 0, 23, and some in between. You would want to test your wait times for 0, and some really large numbers. What about negative numbers? Negative numbers don't make sense, but 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 finally we need to account for those numbers that are bigger than 23. For this we will need one final step, using the modulo operator.

```
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_str)

final_time_int = current_time_int + wait_time_int

final_answer = final_time_int % 24

print("The time after waiting is: ", final_answer)
```

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 Quiz

Question *Submitted Mar 17th 2022 at 8:41:46 pm*

Debugging is:

- (●) tracking down programming errors and correcting them.
- ( ) removing all the bugs from your house.
- ( ) finding all the bugs in the program.
- ( ) fixing the bugs in the program.

3.4 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.

1. 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 (Computer Science 1) level problems millions of times by millions of other programmers. So, Python is probably not the problem.
2. Check your assumptions. At this point in your career you are still developing your mental model of how Python does its work. Its natural to think that your code is correct, but with debugging you need to make your code the primary suspect. Even if you think it is right, you should verify that it really is by liberally using print statements to verify that the values of variables really are what you think they should be. You'll be surprised how often they are not.
3. 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

Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

3.5 Syntax errors

Python can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message. **Syntax** refers to the structure of a program and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a **syntax error**. So does this one

In Python, rules of syntax include requirements like these: strings must be enclosed in quotes; statements must generally be written one per line; the print statement must enclose the value to be displayed in parenthesis; expressions must be correctly formed. The following lines contain syntax errors:

```
print(Hello, world!)
print "Hello, world!"
print(5 + )
```

For most readers of English, a few syntax errors are not a significant problem, which is why we can read the poetry of e. e. cummings without problems. Python is not so forgiving. When you run a Python program, the interpreter checks it for syntax errors before beginning to execute the first statement. If there is a single syntax error anywhere in your program, Python will display an error message and quit without executing *any* of the program.

To see a syntax error in action, look at the following program. Can you spot the error? After locating the error, run the program to see the error message.

```
print("Hello, World!")
print(5 + )
print("All finished!")
```

Notice the following:

1. The error message clearly indicates that the problem is a `SyntaxError`. This lets you know the problem is not one of the other two types of errors we'll discuss shortly.
2. The error is on line 2 of the program. However, even though there is nothing wrong with line 1, the print statement does not execute — **none** of the program successfully executes because of the presence of just one syntax error.
3. The error gives the line number where Python believes the error exists. In this case, the error message pinpoints the location correctly. But in other cases, the line number can be inaccurate or entirely missing.

One aspect of syntax you have to watch out for in Python involves indentation. Python requires you to begin all statements at the beginning of the line, unless you are using a flow control statement like a `for` or an `if` statement (we'll discuss these soon... stay tuned!). To see an example of this kind of problem, modify the program above by inserting a couple of spaces at the beginning of one of the

lines.

3.6 Quiz

Question 1 *Submitted Mar 17th 2022 at 8:41:53 pm*

Which of the following is a syntax error?

- ( ) Attempting to divide by 0.
- (•) Forgetting a colon at the end of a statement where one is required
- ( ) Forgetting to divide by 100 when printing a percentage amount.

Question 2 *Submitted Mar 17th 2022 at 8:41:58 pm*

Who or what typically finds syntax errors?

- ( ) The programmer.
- (•) The compiler / interpreter.
- ( ) The computer.
- ( ) The teacher / instructor.

3.7 Runtime Errors

The second type of error is a **runtime error**. A program with a runtime error is one that passed the interpreter's syntax checks, and started to execute. However, during the execution of one of the statements in the program, an error occurred that caused the interpreter to stop executing the program and display an error message. Runtime errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened.

Here are some examples of common runtime errors you are sure to encounter:

- Misspelled or incorrectly capitalized variable and function names
- Attempts to perform operations (such as math operations) on data of the wrong type (ex. attempting to subtract two variables that hold string values)
- Dividing by zero
- Attempts to use a type conversion function such as `int` on a value that can't be converted to an int

The following program contains various runtime errors. Can you spot any of them? After locating the error, run the program to see the error message.

```
subtotal = input("Enter total before tax:")
tax = .08 * subTotal
print("tax on", subtotal, "is:", tax)
```

Notice the following important differences between syntax errors and runtime errors that can help you as you try to diagnose and repair the problem:

- If the error message mentions `SyntaxError`, you know that the problem has to do with syntax: the structure of the code, the punctuation, etc.
- If the program runs partway and then crashes, you know the problem is a runtime error. Programs with syntax errors don't execute even one line.

3.8 Quiz

Question 1 *Submitted Mar 17th 2022 at 8:42:06 pm*

Which of the following is a run-time error?

- (●) Attempting to divide by 0.
- ( ) Forgetting a colon at the end of a statement where one is required.
- ( ) Forgetting to divide by 100 when printing a percentage amount.

Question 2 *Submitted Mar 17th 2022 at 8:42:12 pm*

Who or what typically finds runtime errors?

- ( ) The programmer.
- (●) The interpreter.
- ( ) The computer.
- ( ) The teacher / instructor.

3.9 Semantic Errors

The third type of error is the **semantic error**, also called a **logic error**. If there is a semantic error in your program, it will run successfully in the sense that the computer will not generate any error messages. However, your program will not do the right thing. It will do something else. Specifically, it will do what you **told** it to do, not what you **wanted** it to do.

The following program has a semantic error. Execute it to see what goes wrong:

```
num1 = input('Enter a number:')
num2 = input('Enter another number:')
sum = num1 + num2

print('The sum of', num1, 'and', num2, 'is', sum)
```

This program runs and produces a result. However, the result is not what the programmer intended. It contains a semantic error. The error is that the program performs concatenation instead of addition, because the programmer failed to write the code necessary to convert the inputs to integers.

With semantic errors, the problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. The computer is faithfully carrying out the instructions you wrote, and its results are correct, given the instructions that you provided. However, because your instructions have a flaw in their design, the program does not behave as desired.

Identifying semantic errors can be tricky because no error message appears to make it obvious that the results are incorrect. The only way you can detect semantic errors is if you *know in advance* what the program should do for a given set of input. Then, you run the program with that input data and compare the output of the program with what you expect. If there is a discrepancy between the actual output and the expected output, you can conclude that there is either 1) a semantic error or 2) an error in your expected results.

Once you've determined that you have a semantic error, locating it can be tricky because you must work backward by looking at the output of the program and trying to figure out what it is doing.

Test Cases

To detect a semantic error in your program, you need the help of something called a test case.

Test Case
A **test case** is a set of input values for the program, together with the output that you expect the program should produce when it is run with those particular inputs.

Here is an example of a test case for the program above:

```
Test Case
---------
Input: 2, 3
Expected Output: 5
```

If you give this test case to someone and ask them to test the program, they can type in the inputs, observe the output, check it against the expected output, and determine whether a semantic error exists based on whether the actual output matches the expected output or not. The tester doesn't even have to know what the program is supposed to do. For this reason, software companies often have separate quality assurance departments whose responsibility is to check that the programs written by the programmers perform as expected. The testers don't have to be programmers; they just have to be able to operate the program and compare its results with the test cases they're given.

In this case, the program is so simple that we don't need to write down a test case at all; we can compute the expected output in our heads with very little effort. More complicated programs require effort to create the test case (since you shouldn't use the program to compute the expected output; you have to do it with a calculator or by hand), but the effort pays off when the test case helps you to identify a semantic error that you didn't know existed.

Semantic errors are the most dangerous of the three types of errors, because in some cases they are not noticed by either the programmers or the users who use the program. Syntax errors cannot go undetected (the program won't run at all if they exist), and runtime errors are usually also obvious and typically detected by developers before a program is released for use (although it is possible for a runtime error to occur for some inputs and not for others, so these can sometimes remain undetected for a while). However, programs often go for years with undetected semantic errors; no one realizes that the program has been producing incorrect results. They just assume that because the results seem reasonable, they are correct. Sometimes, these errors are relatively harmless. But if they involve financial transactions or medical equipment, the results can be harmful or even deadly. For this reason, creating test cases is an important part of the work that programmers perform in order to help them produce programs that work correctly.

3.10 Quiz

Question 1 *Submitted Mar 17th 2022 at 8:42:22 pm*

Which of the following is a semantic error?

- ( ) Attempting to divide by 0.
- ( ) Forgetting a semi-colon at the end of a statement where one is required.
- (•) Forgetting to divide by 100 when printing a percentage amount.

Question 2 *Submitted Mar 17th 2022 at 8:42:30 pm*

Who or what typically finds semantic errors?

- (•) The programmer.
- ( ) The compiler / interpreter.
- ( ) The computer.
- ( ) The teacher / instructor.

3.11 Know your Error Messages

Many problems in your program will lead to an error message. For example, can you spot what type of error the following program below will through before pressing the "Run" button?

```
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)
```

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!

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 5 and you will see that `wait_time_int` is used on both the left and the right hand side of the assignment statement. `wait_time_int does` not already have a value in the preceeding lines so it cannot be used on the right hand side.

Below are some explanations of errors you might encounter when programming:

SyntaxError

Syntax 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 SyntaxErrors 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. Paretheses must be balanced, or else Python thinks that you are trying to include everything that follows as a parameter to some function.

TypeError

TypeErrors occur when you you try 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.

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.

ValueError

Value errors occur when you pass a parameter to a function and the function is expecting a certain limitations on the values, and the value passed is not compatible.

These are just some of the more common errors you will encounter when beginning to learn to program in Python, there are much more out there, so we suggest you consult the official Python documentation (or use your Google skills) to search up any other types of errors you may encounter.