In this exercise, we will identify the three types of errors in programming using Python 3. We will be using the lab2a.py file, which will be supplied with this video. If you do not have access to this file, don’t worry; it’s not required for this exercise.

In programming, there are three main types of errors: Syntax Errors, Runtime Errors, and Logical Errors. All of these errors can cause your program to perform in unexpected ways, most of which will prevent your program from even compiling at all. We will cover all three of these errors and try to extract their meaning, as well as their differences between one another.

The first type of error is known as a Syntax Error. Syntax Errors occurs whenever the programmer fails to abide by the rules of the grammar of the language that they are working in. Common examples of Syntax Errors include using the wrong casing when typing keywords, and failing to putting punctuation where it is supposed to be.

The best way to identify Syntax Errors is to read through the script line by line, making sure that our code follows the grammar rules. In the case of lab2a.py, we will be following the grammar rules of Python 3. Giving this program a quick test run will show that there are indeed several errors littered throughout the file. At a glance, we can see that line 3 of our file has an extra bracket attached to the print function, which is not allowed. By Python’s grammar ruleset, any left-hand brackets must be accompanied by a closing right-hand bracket, no more, no less. Thus, we can remove the second right-hand bracket, thereby correcting the Syntax Error on this line.

We can now run the program, although there are still several errors to be found. Running the program at this point will reveal another Syntax Error, this time on line 6.

The use of the colon in this line suggests that we are trying to start a loop, a programming technique that allows the same lines of code to be run multiple times. This is a Syntax Error because the computer thinks we are trying to start a loop at this instance, however, the colon is merely a typo. Erasing it will remove the Syntax Error here.

We could continue going through the program checking for errors without having to execute the code over and over again. Keep in mind that the compiler will always catch Syntax Errors and Runtime Errors for you upon execution. For now, let’s keep trying to run our program as we resolve each issue.

Another Syntax Error can be found on line 9, where we assign the value of x1\*2 to the variable 5\_x. 5\_x is not a valid name for a variable in Python since the first character is a number. Thus, we much change it to a more suitable name, such as x5. This corrects the Syntax Error on this line.

That should fix every Syntax Error in this program. Now we can try executing out program again.

The next type of error we will cover is known as a Runtime Error. Runtime Errors occur whenever the program instructs the computer to do something or perform a certain instruction that it is either incapable of or unwilling to do. Some example of Runtime Errors include dividing a number by 0, misspelling function names, or utilizing a variable before it has been initialized. Whenever a Runtime Error is encountered, the compiler should always output an error message, with an identifier hinting at what type of error is causing our program to fail. According to the error message displayed here, there is a problem on line 5, with the NameError identifier. This message usually means that there is a problem pertaining to the naming of a variable or type in our program.

If we check line 5, we can see that we attempt to call a print statement on variable x2 before x2 has been initialized. X2 is actually initialized on line 6, so to resolve this Runtime Error, we must swap lines 5 and 6 with each other. This way, x2 will be initialized on line 5, and then its value will be printed on line 6. A variable must always be initialized first before it can be used.

Again, we try to run our program and encounter another Runtime Error, this time on line 7 under the TypeError identifier. The error message tells us specifically that there is a problem with the summation we are trying to do on this line.

On line 7, the value of x1 + s1 is being assigned to variable x3. While the syntax is valid, this statement will throw an exception since adding a numeric value and a string value is not allowed. In this case, the value 19, which is stored in x1, will be added to the value ‘Hello’, which is stored in s1. Adding 19 to ‘Hello’ is not a valid statement, and so we have a Runtime Error. To resolve this, we can change the variable s1 to x2. Since x2 contains a numeric value, this statement will be valid since adding two integers together is allowed.

One more run will show us that there is one more NameError on line 15. Like our earlier error, the variable x is undefined in our program, yet on line 15 we are attempting to use a variable of this name. Changing the name to x1 will resolve this since x1 is already defined.

We can run our program again and it seems that there are no more Syntax or Runtime Errors in the program since our output has no error messages. We are not done yet however, for there is one last type of error we haven’t covered yet: the Logical Error.

Unlike the previous errors, Logical Errors do not become as obvious once we get past program compilation. This is because unlike Syntax Errors and Runtime Errors, Logical Errors will not be thrown as an exception or display any kind of error message on screen. The program will execute just fine, however, it may not produce the correct output. This can happen due to a failed understanding of how the program works, or if the programmer overlooks something in the logic.

In our example, look at lines 10 and 11. On the surface, there’s nothing wrong with these lines at all. They will be executed like normal without any problems from the compiler. What if however, we wished to access one of these variables later on in the program? For example, let’s say we wanted to take the value of float1 on line 11 and add it to a numeric variable. We would run into a Runtime Error like before because float1 actually contains a string value, which is misleading because the name of the variable indicates that it is housing a float value. As a general note, we should always try to name our variables appropriately. Based on the values that are being stored in int1 and float1, we should change their names to float1 and string1 respectively.

The last Logical Error can be found on line 13. Based on the comment on line 12, the quotient value should contain an integer quotient after x1 has been divided by 4. Using the single divider operator however, will return a float value instead. Recall, this line is still valid and will compile properly. We want it to output an integer quotient though, which is a rule that we have set for ourselves as the programmer. Thus, this line contains a Logical Error, since it does do what we are expecting it to do. Using an integer divider resolves this.

We can now run the program without any errors. As you can see, our output is slightly different from before, as one of our outputs now does not contain decimals. This is correct in terms of the problem we are trying to solve, since according to the comment on line 12, line 13 is supposed to output an integer value through integer division. Remember, this is an error in regards to the problem itself and how our solution is developed. Logical Errors are not as technical as Syntax Errors or Runtime Errors since they will not be explicitly stated upon compilation. They are errors to be discovered by the programmer once they have started testing their program.

To summarize, there are three types of errors that can be encountered while programming. Syntax Errors pertain to the grammar and rules of the language being used. The program will never compile if at least one Syntax Error exists, and the compiler will always catch them for you when you attempt to run your program. Similarly, Runtime Errors will also be caught by the compiler, and occur whenever your code tells the computer to do something that is unwilling or incapable of doing. Dividing by zero is a prime example of this. Runtime Errors also come with identifier names to help pinpoint where the issue is in the code. Logical Errors will usually allow the program to run, however, they can be hard to detect since they are errors that pertain to the solution of the problem itself. If a programmer writes their code expecting a certain output and get something completely different, then chances are they have a Logical Error hidden somewhere in the code. To find Logical Errors, tracing through the program line by line, or using a Debugger tool can be used to help locate the issue.