Writing programs is hard work, and invariably "mistakes are made." When trying to understand how a computer might have given you an unexpected result, it's helpful to be able to consider the error according to one of several categories.

Syntax Errors

Syntax errors are the equivalent of spelling or grammar mistakes: you have mistyped something, or constructed a program statement in such a way that Python doesn't understand what you're trying to do.

```
print("Hello, world!);
```

The statement above is a syntax error because the closing quotation mark is missing, and there's a semi-colon at the end of the line that shouldn't be there.

Typical syntax errors include:

- misspelling a variable name
- incorrectly indenting a line
- incorrect use of parentheses
- incorrect use of a Python instruction

In the case of most syntax errors, your program won't even run, and Python will try to give you an error message that may help you figure out what you did wrong. Note: If you get a syntax error on line 5 and everything looks okay there, check line 4 of your program. That's the next most likely place that the error may be found.

Run-time errors

Run-time errors are problems with your program that crop up when the program is running: you ask the user to enter two numbers to be added together, for example, and they type in the *words* "two" and "three" when your program was expecting to see the *values* 2 and 3. Or the user enters two numbers with the second one to be divided into the first one, and they do this:

```
Enter a number: 25
Enter another number to divide into the first: 0
Traceback (most recent call last):
File "test.py", line 47, in <module>
print(n1/n2)
ZeroDivisionError: division by zero
```

Writing robust programs that help to prevent this kind of problem is an important part of programming.

Logical errors

Logical errors have to do with unexpected behavior of the program that often doesn't give any indication of an error. For example, a program designed to add up the numbers 1 to 10 produces the following output:

```
The sum of the integers 1 to 10 is 45
```

This is a logical error because the sum of the integers 1 to 10 is actually 55, not 45. The only way to avoid these types of errors is with testing. During the development of the program, also printing out the expected result can help diagnose issues with code:

```
The sum of the integers 1 to 10 is 45
Expected result: 55
```

In the logical error example here:

```
age = 21
if age < 35:
    print("You're not old enough to be President of the U.S.")
print("You are old enough to be President of the U.S.")
```

... the program is syntactically correct, and it runs—it just doesn't correctly indicate whether someone who is 21 years old is old enough to be President of the U.S. Because the programmer hasn't included an `else` statement, its output is:

```
You're not old enough to be President of the U.S.
You are old enough to be President of the U.S.
```

The logical flow of the program doesn't work the way that the programmer intended.

Overflow errors

An *overflow* error occurs when the programmer attempts to store a number in a memory space that is too small. The size of the number exceeds the memory available for it, so the value that is stored in memory is incorrect.

Some languages like Python dynamically adjust to accommodate numbers of varying size, so most Python programmers don't have to think about this kind of problem much.

Python

```
n = 1000000       # one million
print(n * n)      # produces the correct result 1000000000000
```

In most languages however, like Java, data is stored in variables that have a **type** with a strictly defined memory size.

Java

```
int n = 1000000         // one million
print(n * n)            // produces the incorrect result -727379968
```

The solution to this problem is to use the `long` data type, which allows for integers with a greater number of decimal places.

An overflow error happens without any indication that you have a problem, so you have to anticipate the problem and fix it yourself without relying on the computer to give you an error message.

Rounding errors

A *rounding* error occurs as a result of the imperfect way that decimal numbers are represented in a binary computer's memory.

Just as the fraction $\frac{1}{3}$ can be written as 0.333333... which is an imperfect decimal representation of that value, computers have to represent decimal values in binary, where there is no exact representation for 0.1, for example.

Python

```
print(0.1 + 0.2)        # produces the incorrect result 0.30000000000000004
```

Another example:

Java

```
double f = 4.35
print(f * 100)       // produces the incorrect result 434.99999999999994
```

To solve an issue with rounding, we acknowledge that decimal values from the computer might be a little off, so we in cases where we think we might be off, we use an EPSILON value that will allow us to be within a certain degree of precision:

Python

```
two = math.sqrt(2) * math.sqrt(2)
print(two)                      # output is 2.0000000000000004   rounding error!

EPSILON = 1e-10
if abs(two - 2) < EPSILON:      # ie. if two is within 0.0000000001 of 2
    print("Two = 2")            # output is Two = 2
```