CSI-160 Python Programming
Errors

Vikas Thammanna Gowda

07/10/2025

Contents

1 Introduction **2**

2 Syntax Errors **2**
2.1 Common causes 2
2.2 Illustration 2

3 Runtime Errors **4**
3.1 Common causes 4
3.2 Illustration 4

4 Logical Errors **5**
4.1 Characteristics 5
4.2 Illustration 5

1 Introduction

Programming errors (often called "bugs") are mistakes in your code that prevent it from working as intended. No one writes perfect code on the first try – in fact, making errors is a normal part of learning to program. Each error you encounter is an opportunity to learn and improve your coding skills. It's important to understand different types of errors so you can identify problems quickly and fix your programs. In Python (and most languages), errors generally fall into three main categories:

- Syntax Errors
- Runtime Errors
- Logical Errors

In the sections below, we'll explain each type in a friendly way, with examples, the exact error messages or outputs you'd see, and tips on how to fix the errors.

2 Syntax Errors

Syntax errors are like grammar mistakes in a sentence. Every programming language has rules (syntax) about how code must be written. If you break these rules – for example, by misspelling a keyword or forgetting a symbol – Python doesn't understand the code and will stop running it. In other words, a syntax error means the code cannot even be executed until the mistake is corrected. Python usually detects these errors before running the program and will point out where the problem is. Just as a bad sentence can't be understood, bad syntax prevents Python from understanding your instructions.

2.1 Common causes

1. **Missing punctuation:** Forgetting a required symbol, such as a colon : at the end of an if or for statement, or a quote or parenthesis in a pair.
2. **Indentation mistakes:** In Python, indentation (spaces or tabs at the beginning of a line) matters. If you indent code incorrectly or inconsistently, it can cause a syntax error.
3. **Typos or misspelled keywords:** For example, writing prnt("Hello") instead of print("Hello") will cause a syntax error because prnt is not a valid keyword.
4. **Unmatched pairs:** Not closing a string with a matching quote, or not closing parentheses/brackets. For instance, print("Hello World! (missing the ending quote) is a syntax error.

2.2 Illustration

Example with Syntax Error:

```
length = 5                                  # line 1
width = 4                                   # line 2
area_rectangle = length * width             # line 3
print("The area is" area_rectangle)         # line 4
```

In the if statement above, we forgot to put a comma **,** in **line 4** after closing the double quotes. Running this code will produce a syntax error. Python will highlight the location of the mistake and show an error message.

[27]:

```
length = 5                              # line 1
width = 4                               # line 2
area_rectangle = length * width         # line 3
print("The area is" area_rectangle)   # line 4
```

```
  Cell In[27], line 4
    print("The area is" area_rectangle)   # line 4
          ^
SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

Figure 1: Syntax Error

To fix the error, we simply add the missing **,** in **line 4** after closing the double quotes and the code will execute without a syntax error.

Example with Syntax Error:

```
length = 5                                  # line 1
width = 4                                   # line 2
area_rectangle = length * width             # line 3
print("The area is", area_rectangle)        # line 4
```

Output:

```
The area is 20
```

3 Runtime Errors

Runtime errors (also known as exceptions) are problems that occur while the program is running. Unlike syntax errors, the code passes the initial syntax check and starts executing, but something unexpected happens during execution that causes the program to crash or stop. In simpler terms, a runtime error means **"Python understood your code, but couldn't complete the execution due to an issue that arose as it was running."** These errors will produce an error message explaining what went wrong. Runtime errors can be frustrating, but the messages Python gives are very helpful for debugging.

3.1 Common causes

1. Using a variable that was never defined (this causes a NameError – Python doesn't know what that name refers to).
2. Dividing by zero (this causes a ZeroDivisionError, because mathematically division by zero is undefined).
3. Trying to access a list index that doesn't exist (this causes an IndexError – for example, asking for the 10th item in a list that has only 5 items).
4. Performing an operation on incompatible types (this causes a TypeError. For instance, trying to add a number and a string together will error out).

3.2 Illustration

Example with Runtime Error (ZeroDivisionError):

```
num_1 = 1
num_2 = 23
num_1 -= 1
result = num_2 / num_1
```

In the above code, we try to divide 23 by 0. Dividing by zero is not allowed, so when Python executes `23/0`, it will throw an error at runtime. The program will stop and output an error message.

```
[28]:
1 num_1 = 1
2 num_2 = 23
3 num_1 -= 1
4 result = num_2 / num_1
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[28], line 4
      2 num_2 = 23
      3 num_1 -= 1
----> 4 result = num_2 / num_1

ZeroDivisionError: division by zero
```

Figure 2: Runtime Error

The message `ZeroDivisionError: division by zero` is telling us exactly what went wrong: we attempted to divide by zero. For instance, we could add a check codition in the code to avoid dividing by zero, or use a try/except block to catch the error and handle it gracefully.
Note: We will dive deep into working around these errors when we handle selection and loops.

4 Logical Errors

Logical errors are the trickiest type of errors to deal with. Unlike syntax errors or most runtime errors, a logic error does not crash your program or produce any obvious error message. The code will run to completion, but the result will be wrong – not what you intended. In other words, the program's logic is flawed. The computer is doing exactly what you told it to do, but because of a mistake in your reasoning or algorithm, it's not doing what you wanted it to do. These errors can be subtle and hard to spot because Python won't point to a specific line – you have to notice the incorrect output yourself and deduce where the problem is.

4.1 Characteristics

1. No syntax error: The code runs without any syntax or runtime complaints. Python thinks everything is fine because from a language perspective, your code is valid.

2. Unexpected or incorrect output: The program's result is not what the programmer intended or expected. The output might look plausible, but it's wrong for the given problem.

3. Difficult to detect: Since there's no crash or loud error message, you might not realize something is wrong until you notice the results. You often have to test your program with different inputs or manually check results to catch logical errors.

4. Cause: faulty logic or assumptions: Logical errors come from mistakes in the algorithm or reasoning – for example, using the wrong formula, wrong conditions, or misunderstanding the problem. They are essentially bugs in your thinking that translate into bugs in the code.

4.2 Illustration

Imagine you want to find the area of a rectangle. We write the code as:

Example

```
length = 5                                  # line 1
width = 4                                   # line 2
area_rectangle = length + width             # line 3
print("The area is", area_rectangle)         # line 4
```

If we run this code, it will execute without any syntax or runtime errors, but the output will be incorrect.

Output:

```
The area is 9
```

The code ran completely, but the result 9 is wrong (the correct answer should be 20). This is a logical error. Why did it happen? The mistake was that in **line 3**, we use addition operator (`+`) instead of multiplication operator `*`.

Generally, fixing a logic error involves revisiting your thought process: you might add print statements or use a debugger to trace what the program is doing, then adjust the code to do the right thing.

Note: Python didn't give any error message for the above bug – the only indication of a problem was the unexpected output. This is why testing your code and knowing the expected result is crucial for catching logical errors.