HyperionDev

Defensive Programming – Error Handling

Task

Visit our website

Introduction

You should now be quite comfortable with basic variable identification, declaration, and implementation. You should also be familiar with the process of writing basic code which adheres to correct Python formatting in order to create a running program. In this lesson, you will be exposed to error handling and basic debugging to fix issues in your code, as well as the code of others – a skill that is extremely useful in a development or programming career!

Defensive programming

Defensive programming is an approach to writing code where the programmer tries to anticipate problems that could affect the program and then takes steps to defend the program against these problems. **Many** problems could cause a program to run unexpectedly.

Debugging

Debugging is something that you, as a software developer/programmer, will come to eat, sleep, and breathe. Debugging is when a programmer looks through their code to try to identify a problem that is causing some error.

Software developers live by the motto: “try, try, try again!” Testing and debugging your code repeatedly is essential for developing effective and efficient code, and achieving mastery as a developer/programmer.

Extra resource

The word “debugging” comes from the **first computers**, back when a computer was roughly the size of an entire room. Bugs (living insects) would get into the computer and cause the device to function incorrectly. The programmers would need to go in and remove the insects, hence the name – debugging.

Error handling

Everyone makes mistakes. Likely you have made some already, and that is not a bad thing. As you have probably already realised, however, just making the mistakes does not facilitate any learning – it is important to be able to **debug** your code. You debug your code by identifying various types of errors and correcting them. What type of errors might you encounter? Let's take a look.

Types of errors

There are three different categories of errors that can occur:

- **Syntax errors** are due to incorrect syntax used in the program (e.g. wrong spelling, incorrect indentation, missing parentheses, etc.). The compiler can detect such errors. If syntax errors exist when the program is compiled, the compilation of the program fails and the program is terminated. Syntax errors are also known as compilation errors. They are the most common type of error.

- **Runtime errors** occur during the execution of your program. The program will compile as normal, but the error occurs while the program is running. An error message will likely also pop up when trying to run the buggy program. Examples of what might cause a runtime error include dividing by zero, and referencing an out-of-range list item (if you are unfamiliar with lists and impatient to learn more, you can read more on **Geeks for Geeks**).

- **Logical errors** occur when your program's syntax is correct, and it runs and compiles, but the output is not what you are expecting. This means that the logic you applied to the problem contains an error. Logical errors can be the most difficult errors to spot and resolve because you need to identify where you have gone wrong in your thought process. Building **metacognitive skills** can help you gain proficiency in detecting logical errors. A useful resource for building metacognitive skills is through HyperionDev's **Learning Accelerator**.

Now that you know the three types of errors you might encounter, let's consider each in more detail.

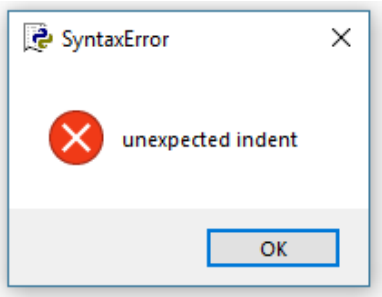
Syntax errors

Syntax errors are comparable to making spelling or grammar mistakes when writing in English (or any other language), except that a computer requires perfect syntax to run correctly.

Common Python syntax errors include:

- leaving out a keyword or putting it in the wrong place,
- misspelling a keyword (such as function or variable name),
- leaving out an important symbol (such as a colon, comma, or parentheses),
- incorrect indentation, and
- leaving an empty block (such as an **if** statement with no indented statements).

Below is a typical example of a compilation error message. Compilation errors are the most common type of error in Python. They can get a little complicated to fix when dealing with loops and if statements.



When a syntax error occurs, the line in which the error is found will often be highlighted, although exactly how depends on your development environment (IDE). The cursor may also automatically be placed at the point where the error occurred to make the error easy to identify. However, watch out – this can be misleading! For example, if you leave out a symbol, such as a closing bracket or quotation mark, the error message could refer to a place later on in the code, which is not the actual source of the problem.

Nonetheless, when you get a syntax error, go to the line indicated by the error message and correct the error if possible. If it is not in that line, work backwards through the code until you identify the source of the problem and then correct it. Then try to compile your code again. Debugging is an essential part of being a programmer, and although sometimes frustrating, it can be quite fun to problem solve the causes of your errors.

Runtime errors

Using your knowledge of strings, take a look at the code below and see if you can identify the runtime error:

```
greeting = "Hello!"
print(greeting[6])
```

This would be the error you get, but why?

```
Exception has occurred: IndexError
string index out of range
```

Look carefully at the description of the error. It must have something to do with the string index. String characters are indexed from zero, and so the final index for "!" would be five, not six. That means that nothing exists for `greeting[6]`, so we get a runtime error. It is important to read error messages carefully and think in a deductive way to solve them. It may, at times, be useful to copy and paste the error message into Google to figure out how to fix the problem.

Logical errors

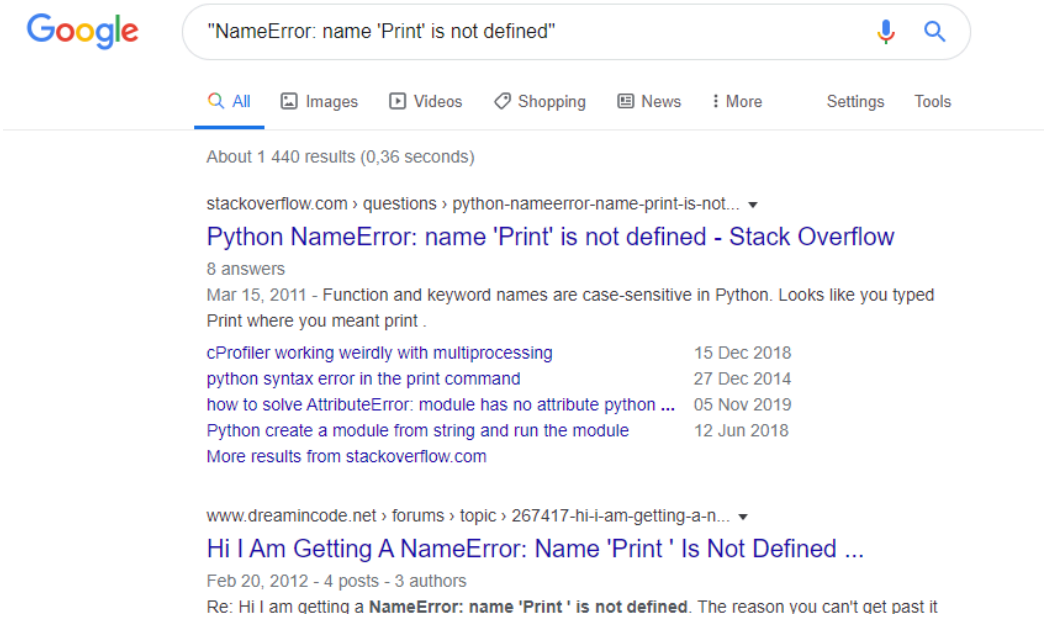
You are likely to encounter logical errors, especially when dealing with loops and control statements. Even after years of programming, logical errors still occur, and more so if you dive into coding solutions to problems without planning the high-level approach using pseudocode. This pseudocode planning, so often skipped by students eager to start coding, is a vital component of logical planning for problem solving. It takes time to sit down and design an algorithm, but it makes later debugging much easier.

Tips for debugging

Debugging becomes easier with practice and experience. You have probably seen plenty of errors in the Python shell similar to the following error:

```
Print ("Hello World!")
NameError: name 'Print' is not defined
```

Many of the errors are quite easy to identify and correct. However, often you may find yourself thinking, 'What does that mean?'. One of the easiest, and often most effective, ways of dealing with error messages that you do not understand is to simply copy and paste the error into Google.



Many forums and websites are available that can help you to identify and correct errors easily. **Stack Overflow** is an excellent resource that software developers around the world use to get help.

GenAI, like OpenAI's ChatGPT or Google's Gemini, can also be really helpful in debugging tricky errors.

In addition, your IDE may have some other built-in features to help you identify and correct subtle errors. Find out if it has a debugger and how to turn it on if it is not on by default (**here are the instructions for VS Code)**.

Many debuggers will allow you to do things like "Go", "Stop", etc. This part of debugger functionality is especially important if you use breakpoints in your code. A breakpoint is a point in your code where you tell it to pause the execution of the code so that you can check what is going on. For example, if you don't understand why an **if** statement is not doing what you think it should, you could create a breakpoint at the **if** statement. You can follow this **video** to learn how to set breakpoints in VS Code.

Instructions

First, read and run the **example files** provided. Feel free to write and run your own example code before doing the tasks, in order to become more comfortable with the concepts covered within.

Take note

The task(s) below is/are **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give the task your best attempt and submit it when you are ready.

After you click "Request review" on your student dashboard, you will receive a 100% pass grade if you've submitted the task.

When you submit the task, you will receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer. Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey for this task, which you can use as a self-assessment tool. Please take a moment to complete the survey.

Once you've done that, feel free to progress to the next task.

Auto-graded task 1

For both the **errors.py** and **errors2.py** task files in your folder, open the files and follow these steps:

- Attempt to run the program. You will encounter various errors.
- Fix the errors and then re-run the program.
- Each time you fix an error, add a **comment** in the line where you fixed it and indicate which of the **three types of errors** it was, with a brief explanation of why that is.
- Save the corrected file.

Auto-graded task 2

Follow these steps:

- Create a new Python file called **logic.py.**
- Write a program that displays a logical error (be as creative as possible!).

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Challenge

Use this opportunity to extend yourself by completing an optional challenge activity.

Follow these steps:

- Create a new Python file called **optional_challenge.py.**
- Write a program with two compilation errors, a runtime error, and a logical error.
- Next to each error, add a comment that explains what type of error it is and why it occurs.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.