Hyperiondev

TASK

Towards Defensive Programming I — Error Handling

Visit our website

Introduction

WELCOME TO THE TOWARDS DEFENSIVE PROGRAMMING TASK!

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 to create a running program. In this task, you'll 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 software development career!

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to **www.hyperiondev.com/portal** to start a chat with a code reviewer. You can also schedule a call or get support via email.

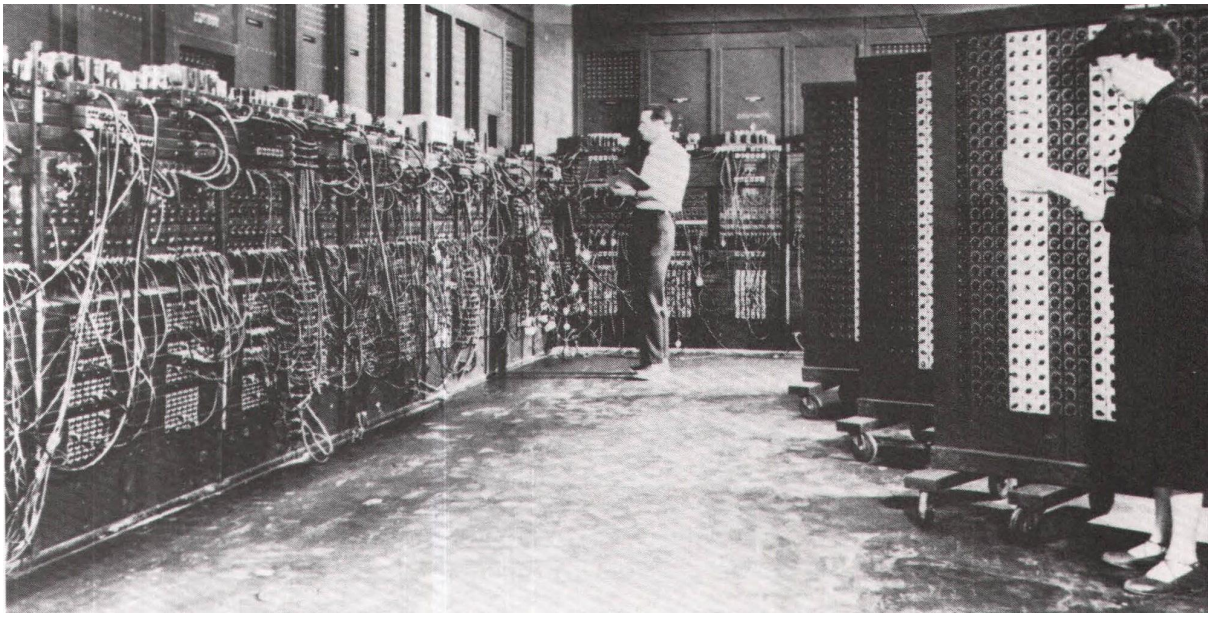
Our expert code reviewers are happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

DEBUGGING

Debugging is something that you, as a software engineer, will come to live, breathe and sleep. Debugging is when a programmer looks through their code to try and find a problem that is causing some error.

The word 'debugging' comes from the first computers — that is, when a computer was roughly the size of an entire room. It was called debugging because bugs (as in, 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 — debugging.

One such computer is the ENIAC. It was the first true electronic computer and was located in the University of Pennsylvania.



An image of the ENIAC.

Notice how today the computer we know is significantly smaller and also much more powerful than older computers like the ENIAC (despite their enormous size). It goes to show how, in just a few years, we can expand our computational power immensely. We owe that to the ever-changing and growing world of technology. So even though sometimes keeping up with the trends may be tiring, keep in mind that it's for the best!

You can do some more research about the ENIAC if you want on the University of Pennsylvania's website: http:// www.seas.upenn.edu/about-seas/eniac/.

Software developers live by the motto: “try, try again!” Testing and debugging your code repeatedly is essential for developing effective and efficient code.

DEALING WITH ERRORS

Everyone makes mistakes. Likely you've made some already. It's important to be able to DEBUG your code. You debug your code by removing errors from it.

Types of errors

There are three different categories of errors that can occur:

- **Syntax errors**: These 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**: Runtime errors occur during the execution of your program. Your program will compile, but the error occurs while the program is running. An error message will also pop up when trying to run it. Examples of what might cause a runtime error include dividing by zero or referencing an out of range list item.

- **Logical errors:** 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.

Syntax errors

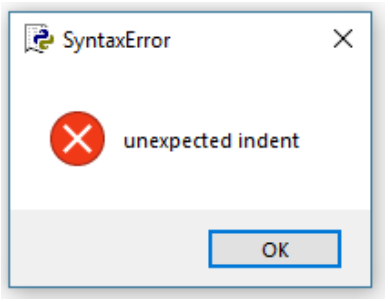
Syntax errors are comparable to making spelling or grammar mistakes when writing in English (or any other language), except that a computer requires perfect syntax to run correctly.

Common python syntax errors include:

- Leaving out a keyword or putting it in the wrong place
- Misspelling a keyword (for example a function or variable name)
- Leaving out an important symbol (such as a colon, comma or parentheses)

- Incorrect indentation
- Leaving an empty block (for example, an if statement containing 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 also be highlighted in red. The cursor may also automatically be put there so that error is easily found. However, 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.

Go to the line indicated by the error message and correct the error, then try to compile your code again. If you cannot see an error on that line, see if you have made any syntax errors on previous lines that could be the source of the problem.

Debugging is not always fun but it is an essential part of being a programmer!

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. If you recall from Task 5, we could indices from 0, and so the final index for "!" would be 5. That means that nothing exists for index 6, so we get a runtime error. It's important to read error messages carefully and think in a deductive way to solve runtime errors. 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

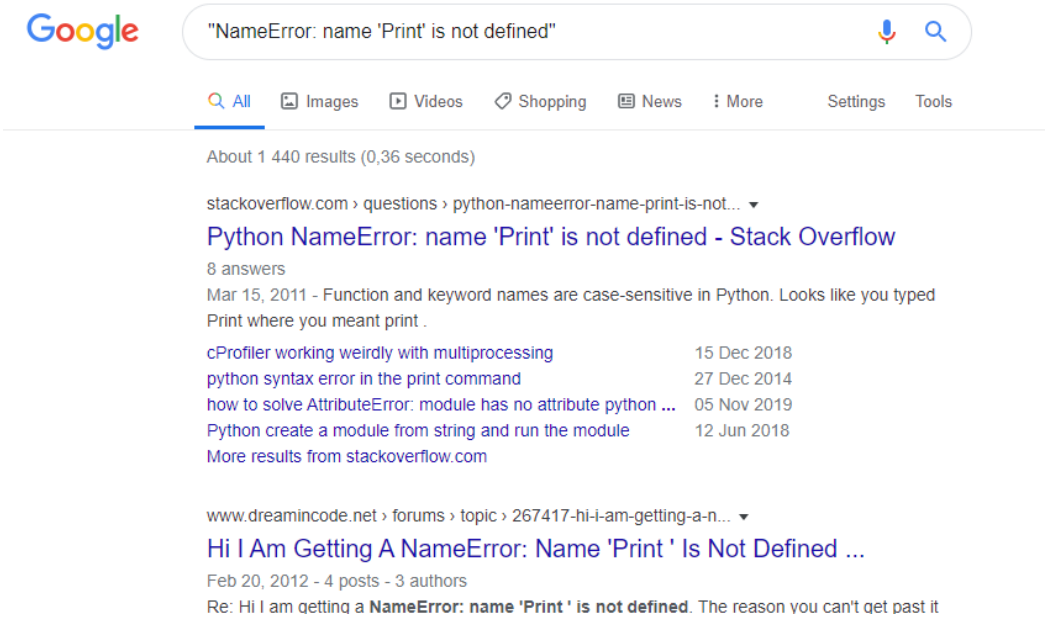
You are likely to encounter logical errors, especially when dealing with loops and control statements. Even after years of programming, it takes time to sit down and design an algorithm. Finding out why your program isn't working takes time and effort but becomes easier with practice and experience.

TIPS FOR DEBUGGING WITH IDLE

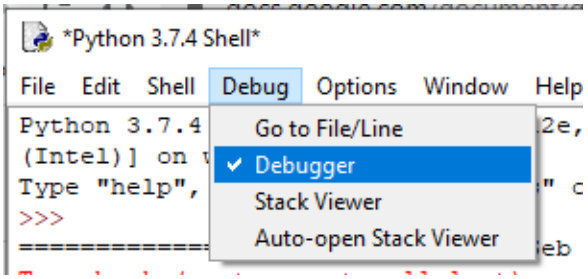
You have probably seen plenty of errors similar to the one shown in the image below in the Python shell.

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

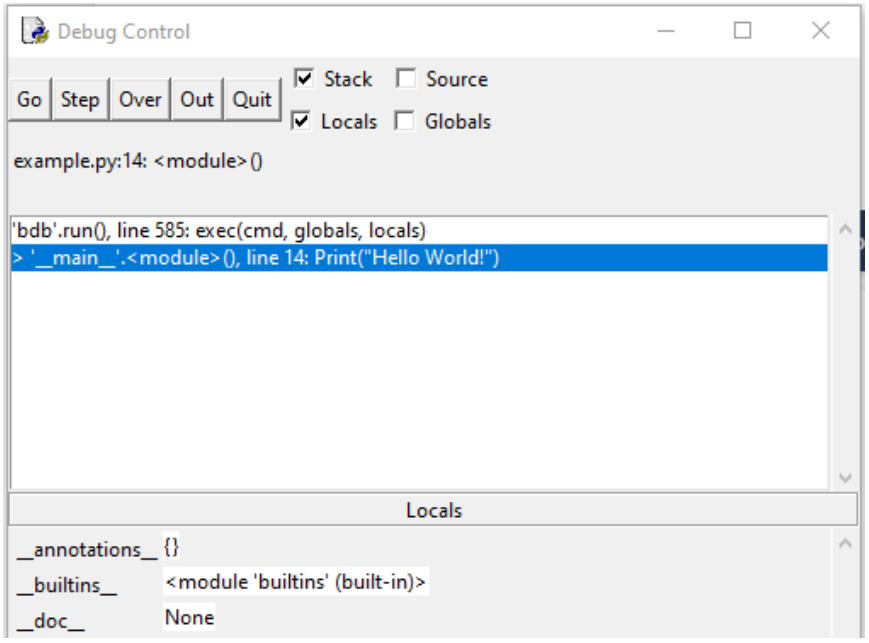
Many of the errors are quite easy to identify and correct. However, often you may find yourself thinking, "What does that mean?". One of the easiest, and often most effective, ways of dealing with error messages that you don't understand, is to simply copy and paste the error into Google. Many forums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



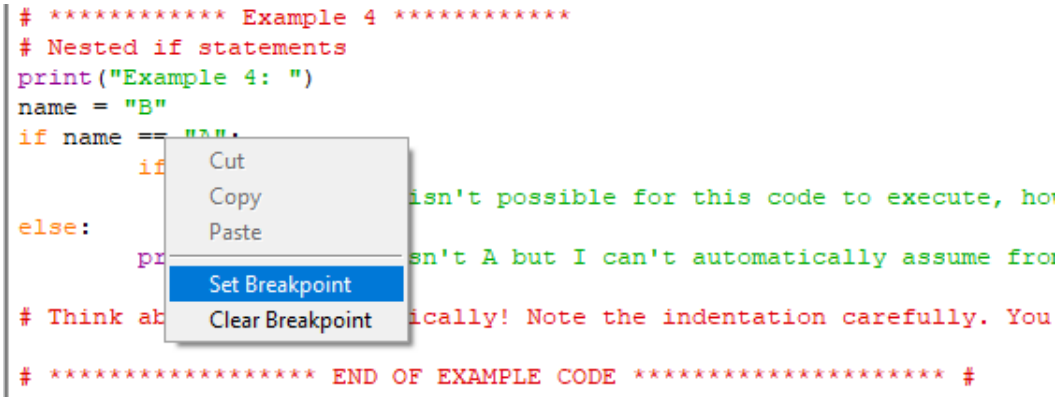
However, IDLE has some other built-in features that can also help you identify and correct more difficult-to-spot errors. IDLE includes a built-in debugger. To turn the debugger on, select Debug → Debugger from the Python IDLE menu bar.



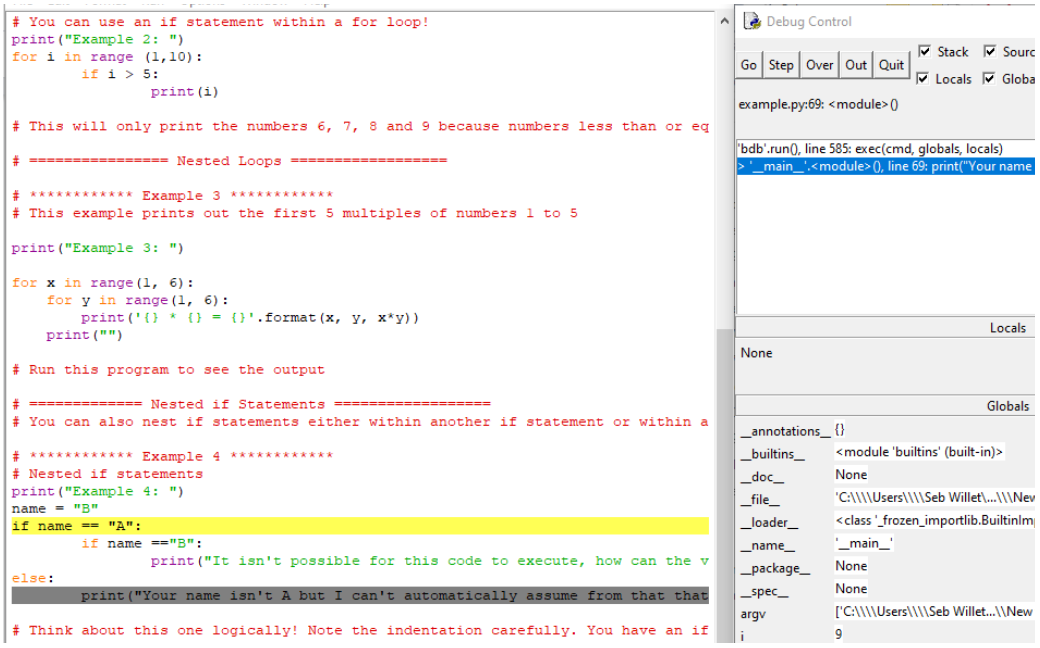
Then, if you run your code, more information about the error will be displayed in the debugger.



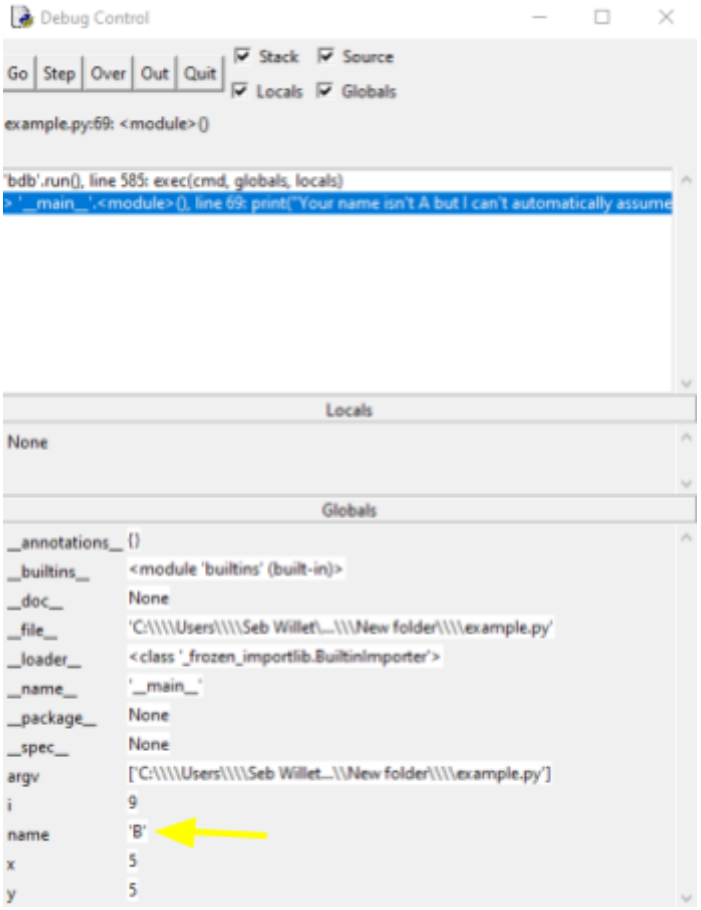
Notice that the debugger allows you to do things like "Go", "Step", etc. This part of the debugger is especially important if you use breakpoints in your code. A breakpoint is a point in your code where you tell it to pause the execution of the code so that you can check what is going on. For example, if you don't understand why an *if statement* isn't doing what you think it should, you could create a breakpoint at the *if statement*. You do this by right-clicking on the *if statement* and selecting "Set Breakpoint" (as shown in the image below).



Now, if you have the debugger on and you run the code, press "Go". The debugger will then run all the code until it reaches the breakpoint in your code.



The debugger stops execution at the breakpoint and allows you to check what values are stored in the variables you have declared, as shown in the image below:



When you press the “Next” button in the debugger, the next line of code in your program will be executed.

Instructions

First, read **example.py**. Open it using IDLE.

- **example.py** should help you understand some simple Python. Every task will have example code to help you get started. Make sure you read all of **example.py** and try your best to understand.
- You may run **example.py** to see the output. Feel free to write and run your own example code before doing the Task to become more comfortable with Python.

Compulsory Task 1

Follow these steps:

- Open **errors.py** in your task folder.
- Attempt to run the program. You will encounter various errors.
- Fix the errors and then 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.
- Save the corrected file.
- Do the same for **errors2.py**

Compulsory 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!).

Optional Bonus Task

Follow these steps:

- Create a new Python file in this folder called **optional_task.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.

Rate us

Share your thoughts

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.