HyperionDev

Debugging – The Stack Trace

Task

Visit our website

Introduction

In this lesson, you will learn about a hypothesis-driven method of debugging. This method makes fixing issues in your code a more efficient process.

Interpreting the stack trace

By now, you have had the experience of writing code and sometimes being faced with a bunch of error messages in the terminal or VS Code shell when executing code. This error report is called the **stack trace**, or in Python, the traceback! For the purpose of this task, we will be referring to it as the stack trace. The stack trace contains a lot of information that can be difficult to understand. Once you have learnt how to interpret the information in the stack trace, you will find it a great tool for helping you to debug your code.

A stack trace is a list of the functions that lead to a given point in a software program. It is a breadcrumb trail for finding errors in your software.

Let us look at the following example, where we run through a series of function calls to show what is represented in the stack trace when one of those functions raises an error or exception:

```
def calling_function():
    another_function()
def another_function():
    last_function()
def last_function():
    return 5/0
result = calling_function()
```

Output:

```
  Traceback (most recent call last):
  File "C:\Python310\debugging.py", line 10, in <module>
    result = calling_function()
  File "C:\Python310\debugging.py", line 2, in calling_function
    another_function()
  File "C:\Python310\debugging.py", line 5, in another_function
    last_function()
  File "C:\Python310\debugging.py", line 8, in last_function
    return 5/0
ZeroDivisionError: division by zero
```

When we look at the stack trace above, we can derive some meaning by carefully examining each line. To start, we look at the very last line, as this will indicate to us what type of error or exception has occurred.

```
ZeroDivisionError: division by zero
```

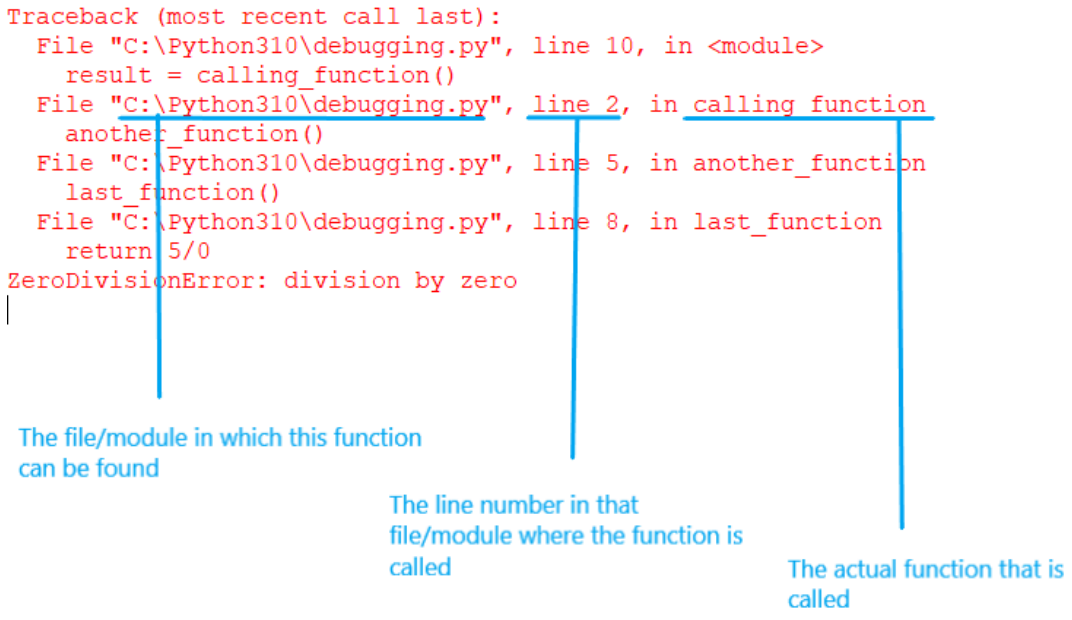
In this case, we have a specific error called a `ZeroDivisionError`. This error indicates that somewhere in our code, we are trying to divide a number by zero, which is not mathematically possible. This is just one of many errors that can occur. Some general errors you may encounter are reference errors, such as when a variable is not defined or syntax errors, such as when there is a missing colon or incorrect indentation.

We can use the **Python Exception Documentation** to get a better idea of what type of problem we're facing and narrow down what to investigate.

After we identify what type of error we are facing, we can move from the top of the stack trace to the bottom. The last function call will be at the bottom of the stack, and the first function call leading to that moment where the error occurred is at the top of the stack.

This means that `last_function` was called by `another_function` and `another_function` was called by `calling_function`. This is very helpful in tracing back through the code to determine what the code was meant to be used for, which narrows down even further what to look for as you are investigating.

When we look at each function call in the stack trace, we can also see which file and which line the functions were called in. In a very large code base with many modules, this is extremely useful when trying to find the problem. Imagine trying to figure out where an error is occurring without this information – you would just be working blindly through code!



Creating a hypothesis

Creating a hypothesis is the most important concept when debugging code. The more we know about the problem, the more targeted our bug-fix attempts will be.

The following steps are a guideline for forming a hypothesis:

1. **Make observations**

   Start by trying to replicate the issue. Record the event that triggers the problem, and the expected and unexpected behaviour of the code. Unexpected behaviour could be errors or incorrect outputs.

   Other observations you can record are where exactly the error occurs based on the stack trace, and which state your program was in when the error occurred. Following the stack trace also gives you a good idea of how the code is expected to behave.

2. **Question**

   At this stage, one of the questions that you could ask is when the problem started. Is it something that has existed for some time or has the problem only arisen after a recent change? You could also ask other questions based on the particular situation, such as the user's role when the problem occurs. Asking questions about the circumstances of the problem helps to narrow down your search for the issue and form a better hypothesis.

3. **Hypothesis**

   Equipped with information, you can develop a well-informed hypothesis as to where, when, and why the problem is occurring based on your understanding of the code.

4. **Make a prediction**

   Having formulated an idea of what might be the cause of the problem, predict what will happen if you make specific changes and if those changes will fix the issue.

5. **Test your hypothesis**

   Now that you have a hypothesis and prediction, you can make the necessary changes to test your hypothesis and prediction. If your hypothesis is incorrect, you can form a new hypothesis and prediction based on any further information gathered.

Gaining visibility into code

An essential part of investigating code and developing a hypothesis is gaining visibility into exactly how lines of code are behaving versus how you assume they are behaving. We can use the debugger in VS Code to do this. We strongly recommend using the resources listed below.

- **VS Code Debugger**
- **VS Code Debugger Video**

Bookmark these resources about the debugger for quick reference.

Another simple way to gain visibility into how code is behaving is to use well-placed print statements. This allows you to see the values of variables and whether those values correspond with what you assumed they should be.

The following example illustrates how print statements help us get visibility into the code. Consider the following code with a function that takes a string input with colours separated by commas and spaces. The function should split up the string into a list, count the colours, and then output the count for each colour.

```
# Function to count colours in a string that are separated by commas
def count_red_green_blue(input_colors):
    red_count = 0
    green_count = 0
    blue_count = 0
    split_colors = input_colors.split(",")
    for color in split_colors:
        if color == "red":
            red_count += 1
        elif color == "green":
            green_count += 1
        elif color == "blue":
            blue_count += 1

    return f"Greens = {green_count}\nBlues = {blue_count}\nReds = {red_count}"
# Call function to count colours in a string and print result
print(count_red_green_blue("green, red, green, green, blue, blue, blue"))
```

Unfortunately, when we run the code we get the incorrect output:

```
Greens = 1
Blues = 0
Reds = 0
```

Oh, no! Why is my output wrong? Only one green has been counted! Never fear; we can output the values of variables to the terminal or shell to better understand how our code is behaving.

Let's see what happens when we output the values of the `split_colors` list with a print statement.

```
# Function to count colours in a string that are separated by commas
def count_red_green_blue(input_colors):
    red_count = 0
    green_count = 0
    blue_count = 0
    split_colors = input_colors.split(",")
    # Add a print statement to see the values in split_colors
    print(split_colors)

    for color in split_colors:
        if color == "red":
            red_count += 1
        elif color == "green":
            green_count += 1
        elif color == "blue":
            blue_count += 1

    return f"Greens = {green_count}\nBlues = {blue_count}\nReds = {red_count}"
# Call function to count colours in a string and print result
print(count_red_green_blue("green, red, green, green, blue, blue, blue"))
```

What is the output in the terminal or shell?

```
['green', ' red', ' green', ' green', ' blue', ' blue', ' blue']
Greens = 1
Blues = 0
Reds = 0
```

By looking at the strings in the list, we can see that after splitting the input string, we still have whitespace at the beginning of some of the elements. We have not split correctly! Now that we know what the issue is, we can make a targeted change to the code, i.e. change the argument in the split method from **“,”** to **“, ”** thereby including the spaces in the splitting criteria and excluding them from the output. This removes the whitespaces and our code behaves as expected.

```
# Function to count colours in a string that are separated by commas
def count_red_green_blue(input_colors):
    red_count = 0
    green_count = 0
    blue_count = 0
    split_colors = input_colors.split(", ") # Add a space to split correctly
    # Add a print statement to see the values in split_colours
    print(split_colors)
    for color in split_colors:
        if color == "red":
            red_count += 1
        elif color == "green":
            green_count += 1
        elif color == "blue":
            blue_count += 1

    return f"Greens = {green_count}\nBlues = {blue_count}\nReds = {red_count}"
# Call function to count colours in a string and print result
print(count_red_green_blue("green, red, green, green, blue, blue, blue"))
```

Let's rerun the code. Now we have the correct output!

```
['green', 'red', 'green', 'green', 'blue', 'blue', 'blue']
Greens = 3
Blues = 3
Reds = 1
```

Instructions

Read and run the accompanying **example files** provided before doing the task in order to become more comfortable with the concepts covered in this task.

Take note

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

When you select “Request review”, the task is automatically complete, you do not need to wait for it to be reviewed by a mentor.

You will then 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, which you can use to self-assess your submission.

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

Auto-graded task

1. Make a copy of the debugging.py and rename it **debugging_task.py**. You will be making changes to the **debugging_task.py** file and leaving the **debugging.py** file as is.
2. Debug the Python code and ensure you get the suggested output outlined in the comment at the bottom of the Python file.
3. For each bug you resolve, please leave a comment identifying the change you made.

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

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.