TASK

Hypothesis-driven debugging with the stack trace

Visit our website

Introduction

WELCOME TO THE TASK ON DEBUGGING YOUR CODE!

In this task, we will be learning about a hypothesis-driven process of debugging that can be implemented to make fixing code that does not behave as expected a more efficient experience .

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 Discord at **https://discord.com/invite/hyperdev** where our specialist team is ready to support you.

Our team is 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!

THE STACK TRACE

By now, you would have had the experience of writing some code and then when trying to execute that code, being faced with a bunch of red text in the terminal or in the IDLE shell stating some error message with a whole lot of information that seems that it might be difficult to understand. Do not panic, this error report is what is called the **stack trace**, or in Python specifically, it is called the Traceback! For the purpose of this task, I will just be referring to it as the stack trace for simplicity. When you understand 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, in order, that lead to a given point in a software program.

A stack trace is essentially a breadcrumb trail for your software.

Let us look at the following example where we are not using any real logic, but run through a series of function calls, to enable you to see what is represented in the stack trace when one of those functions raises an error/exception:

```
def calling_function():
    another_function()

def another_function():
    last_function()

def last_function():
    return 5/0

result = calling_function()
```

```
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. The first place to start is the very last line as this will indicate to us what type of error/exception has occurred.

```
ZeroDivisionError: division by zero
```

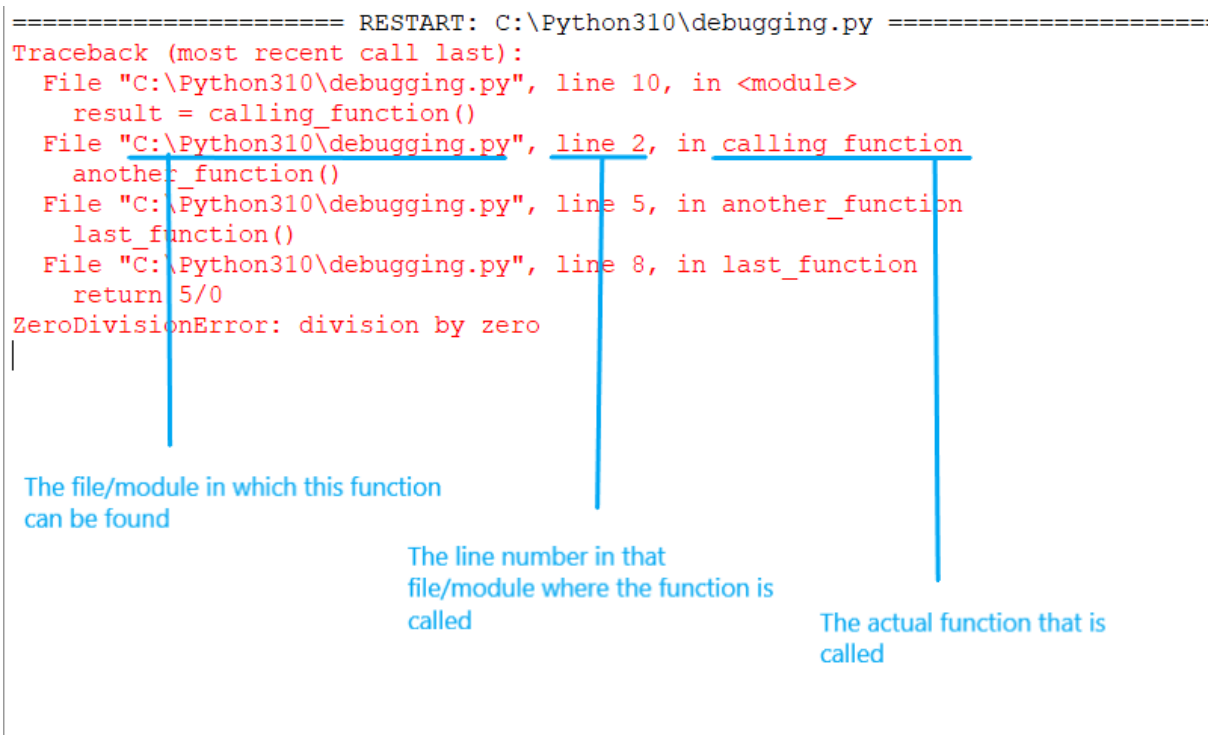
In this case, we have a specific error called a ZeroDivisionError which indicates that somewhere in our code we are trying to divide a number by zero which is not mathematically possible. This is just one of many errors that can occur. Some general errors that you can encounter are **Reference errors**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

We can always use the **Python Exception Documentation** to get a better idea of what type of problem we are facing so that we can narrow down what we are investigating.

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 that you have written 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'd 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:

Step 1: Make observations.

Try replicating the issue that is being experienced. Record the event that triggers the issue and what the expected behaviour should be as well as what the unexpected behaviour is that you are experiencing. Unexpected behaviour could be errors or even incorrect outputs.

Other observations that 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. Looking at the code by following the stack trace also gives you a good idea of how the code is expected to behave.

Step 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? There are other questions that you could also ask based on the particular problem depending on the issue at hand such as the user's role when the problem occurs, for example. Asking questions about the circumstances of the problem helps to be able to narrow down your search for the issue and form a better hypothesis.

Step 3: Hypothesis

Now that you are equipped with information, you can form a well-informed hypothesis as to where, when, and why the problem is occurring based on your understanding of the system as well.

Step 4: Make a prediction

Having formulated an idea of what might be the cause of the problem, make a prediction as to what will happen if you make specific changes and if those changes will or will not realistically fix the issue.

Step 5: Test your hypothesis

Now that you have hypothesised and made a prediction, you can make the necessary changes to test that hypothesis and prediction. If your hypothesis is incorrect, you can reevaluate and form a new hypothesis and related prediction based on any new information gathered, and employ deductive reasoning.

GAINING VISIBILITY INTO CODE

An important part of investigating code and forming a hypothesis is gaining visibility into exactly how lines of code are actually behaving vs how you may assume they behave. We can use the Debugger in IDLE or in VS Code to do this. We strongly recommend looking at these resources. You can bookmark the resources about debugging for quick reference.

Debugging with IDLE debugger

VS Code Debugger

VS Code Debugger Video

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

A quick example of how print statements help us get visibility into the code is as follows. Say we have the following code for a function that takes a string input with colours separated by commas and spaces. Our function should split up the string into an array, count the colours and then output the count.

```
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} Blues = {blue_count} Reds = {red_count}"

print(count_red_green_blue("green, red, green, green, blue, blue, blue"))
```

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

```
====================== RESTART: C:\Python310\debugging.py ======================
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/shell so that we can better understand how our code is behaving.

Let's see what happens when we output the values of the list "split_colors" after we create it using the split() function. We can do this with a print statement.

```
def count_red_green_blue(input_colors):
    red_count = 0
    green_count = 0
    blue_count = 0

    split_colors = input_colors.split(",")

    #adding a print statement to see what the values in the split_colors list are
    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} Blues = {blue_count} Reds = {red_count}"

print(count_red_green_blue("green, red, green, green, blue, blue, blue"))
```

What is the output in the terminal/shell?

```
====================== RESTART: C:\Python310\debugging.py ======================
['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 and end 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. changing 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 whitespace and our code behaves as expected.

```
def count_red_green_blue(input_colors):
    red_count = 0
    green_count = 0
    blue_count = 0

    split_colors = input_colors.split(", ")# <-- added a space so we split correctly

    #adding a print statement to see what the values in the split_colors list are
    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} Blues = {blue_count} Reds = {red_count}"

print(count_red_green_blue("green, red, green, green, blue, blue, blue"))
```

Lets run the code again and we see that we now have the correct output!

```
====================== RESTART: C:\Python310\debugging.py ======================
['green', 'red', 'green', 'green', 'blue', 'blue', 'blue']
Greens = 3 Blues = 3 Reds = 1
```

Compulsory Task 1

- Make a copy of the “Debugging” folder and rename it “CompulsoryTask”. You will be making changes to the **debugging.py** file in the “CompulsoryTask” folder and leaving the file in the “Debugging” folder as is.
- There are bugs in the code that need to be resolved. Debug the Python code and ensure that you get the suggested output as outlined in the comments at the bottom of the **debugging.py** file.
- For each bug that you have resolved, please leave a comment identifying the change that you made.

Rate us

Share your thoughts

HyperionDev 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.