












< Previous	 								Next >
------------	---	---	---	--	---	---	---	---	--------

Part 1: Logging

 [Bookmark this page](#)

Logging and Debugging

Logging

We'll be talking about both logging and debugging in this lesson. In the introduction, I said that logging and debugging are a great step up from the print statements that beginning programmers often like to use to debug their code.

Your assignment for this week will be the culmination of the logging exercises that we complete in this lesson. I encourage you to actually create and modify each of the exercise scripts that I demonstrate below.

Print Statement Debugging

Let's start off by writing a simple Python script with an obvious problem. Begin by creating simple.py:

```
# simple.py
def my_fun(n):
    for i in range(0, n):
        100 / (50 - i)

if __name__ == "__main__":
    my_fun(100)
```

You can probably already see the problem that we'll encounter when we run simple.py. Let's run it in python:

```
$ python simple.py
Traceback (most recent call last):
  File "simple.py", line 7, in <module>
    my_fun(100)
  File "simple.py", line 4, in my_fun
    100 / (50 - i)
ZeroDivisionError: division by zero
```

As you might have expected, we get a ZeroDivisionError! At some point, for some value of *i*, the instruction `100 * / (50 - i)*` causes our program to attempt to divide by zero.

I'm sure that you can see what value of *i* would cause this problem, but let's pretend that we didn't know, and we were trying to figure it out. If you had no better tools, you might try to investigate this problem by adding print statements to the loop. You could print out the value of *i* just before the problem-fraught division statement. Make the following modification to simple.py:

```
for i in range(0, n):
    print(i)          # <-- Add this line
    100 / (50 - i)
```

Now running simple.py will give us some clues about the faulting value of *i*:

```
$ python simple.py
0
1
2
...
48
49
50
Traceback (most recent call last):
  File "simple.py", line 2, in <module>
    100 / (50 - i)
```

If we didn't know it already, now we do! The value of *i* *just before the ZeroDivisionError* is 50. *This is the faulting value of *i*.*

This “print statement debugging” is how a lot of new programmers begin trying to understand problems in their code. And many advanced programmers will still use a print statement when they’re writing simple scripts.

But what are the problems with print statement debugging?

Here are a few issues with using a print statement to debug your code:

- You have to go back in and take them out, otherwise they produce distracting output when you’re running your program.

Previous

Next

- If you have more than a couple of print statements, it becomes hard to keep track of where they all are and what each one specifically is reporting on.
- Print statements don’t help you when your code is being run in production: you can only use print statements when you’re running the code on your own machine from your console.

© All Rights Reserved

To fix all of these problems, we’re going to use *logging*. Logging is a practice that’s used in similar ways across a lot of different languages: you’ll be able to apply these lessons about logging to your entire programming career.

We’ll practice logging statements that:

- You can choose to hide or show with each run of your code.
- You can automatically add extra information to, like the line number and file that they’re invoked in.
- You can send from any Internet connected device to a centralized server, to monitor your code as it works in production.

If you like using print statements to debug your code, you’ll enjoy logging: message logging is a direct step up from print statements.

