Debugging

Ask a random sample of students enrolled in a computer science course what their favorite aspect of programming is, and very few will respond with debugging. Debugging programs in computer science can be daunting, since there are a variety of ways programs can fail. However, armed with the right tools and mindset, debugging can be easy and even fun[1]! In this document, we will go over one of the most powerful tools in your arsenal in fighting errors, the stack trace. We have also provided a list of common Java and Python exceptions you may encounter in your work in CS16. This list is at the end of the document.

The Stack Trace

Throughout this course and the previous course, you have learned and analyzed the stack structure used to store and produce data[2]. The underlying principle of a stack in computer science is the *last-in first-out* (LIFO) order of objects, where the last object "pushed" onto the stack is the first object "popped". For example, imagine we have an empty stack onto which we will push and pop integers. We might use the stack as follows:

Operation	Stack [bottom → top]
Instantiate	[ ]
Push "1"	[1]
Push "2"	[1, 2]
Push "3"	[1, 2, 3]
Pop	[1, 2]
Push "4"	[1, 2, 4]
Pop	[1, 2]

Stacks are especially important because the implications of LIFO ordering extend beyond manipulating data. In fact, you can interpret running a program as pushing and popping methods from a stack! Think of this stack of methods as the *call stack*. In this frame of mind, calling methods is equivalent to pushing them onto the stack, and completing a method is the same as popping it from the stack. As long as a program exists, the method at the top of the stack is run until completion. When no methods are left in the call stack, the program has finished running.

For example, imagine we have a program with the methods `main()`, `a()`, `b()`, and `c()`, where `main()` calls `a()` and `b()` in order, with `b()` calling `c()` as part of its definition. Launching this program with `main()`, we might see the state of the call stack over the course of the program as follows:

[1] Really!

[2] Or you will soon, depending on when you read this.

Method	Call Stack [bottom → top]
Call `main`	`main()`
`main` calls `a`	`main(), a()`
`a` finishes	`main()`
`main` calls `b`	`main(), b()`
`b` calls `c`	`main(), b(), c()`
`c` finishes	`main(), b()`
`b` finishes	`main()`
`main` finishes	<end>

Imagine now that while testing your program, one of your methods runs into an error or exception that is unhandled. To help debug this, a Java or Python program will produce a *stack trace* in your terminal. A stack trace is simply a printout of information about the call stack at the time the error was observed. Depending on the error, your program may continue or stop executing and crash.

Learning how to read and understand a stack trace can be enormously useful in quickly fixing bugs. Let us analyze some code, and the stack trace that results when we try to run this code, to see how we can approach stack traces in general.

Look the following Python code, written in a file called *error.py*. As the file name suggests, can you see an error in the code?

```
 1 def main():
 2     a()
 3     b()
 4
 5
 6 def a():
 7     return
 8
 9
10 def b():
11     c()
12
13
14 def c():
15     num = num - 1
16
17
18 if __name__ == "__main__":
19     main()
```

If run this code, our program prints out the following in the terminal:

```
Traceback (most recent call last):
  File "error.py", line 19, in <module>
    main()
  File "error.py", line 3, in main
    b()
  File "error.py", line 11, in b
    c()
  File "error.py", line 15, in c
    num = num - 1
UnboundLocalError: local variable 'num' referenced before assignment
```

According to the stack trace, an unhandled error was detected in line 15, in `c()`. We can also see that in reaching this unhandled error, `c()` was initially called by `b()` in line 11, which itself was earlier called by `main()` in line 3. The program was itself started by a call to `main()` in line 19, with <module> referring to the mainline in this case. The program also states that the methods in the call stack came from the file *error.py*, which matches with the file name provided above. So at some point in the life of this program, it called the method `main()` in *error.py*, which called `b()` in *error.py*, which called `c()` in *error.py*, which ran into an `UnboundLocalError` on line 15 of *error.py*.

The stack trace tells us plenty of information about the fatal error we countered. We know, for example, the sequence of methods that resulted in the error being thrown. We also know the line in question that encountered the error, as well as the error itself. Based on the information in this stack trace, we can see that our error was caused by referencing the variable `num` before assigning a value to it. We can then use the stack trace to determine where we should assign a value to `num` so that line 15 of *error.py* does not run into an `UnboundLocalError`. For example, we can fix this error by asserting that `num` is not `None` before attempting to execute `num = num - 1`.

Make sure you are comfortable with how we just used the information provided by the stack trace to debug our error. Knowing the exact circumstances that led to our program receiving an error, and being able to leverage that information to look backwards in time through the call stack to diagnose the problem, are both useful skills in debugging. Reading the stack trace becomes even more important as bugs become more nuanced.

Often in a stack trace, the error described may not necessarily be fixed by changing the most recent call, as was the case in the previous example. For example, look at the following Python code in the file *trickyerror.py*. This error might be a little harder to find, but take a brief moment here and try and see if you can find it!

```
 1 def main():
 2     x = 4;
 3     y = None;
 4     a(x)
 5     a(y)
 6
 7
 8 def a(param):
 9     """
10     a(num)
11     Consumes: a number
12     Produces: nothing
13     Calls b() on the input parameter
14     """
15     b(param)
16
17
18 def b(num):
19     """
20     b(num)
21     Consumes: a number
22     Produces: that number minus 1
23     Subtracts 1 from the input number
24     and returns it
25     """
26     return num - 1
27
28
29 if __name__ == "__main__":
30     main()
```

As before, if we run this code, we get a catastrophic error and a nice stack trace to help us fix it:

```
Traceback (most recent call last):
  File "trickyerror.py", line 30, in <module>
    main()
  File "trickyerror.py", line 5, in main
    a(y)
  File "trickyerror.py", line 15, in a
    b(param)
  File "trickyerror.py", line 26, in b
    return num - 1
TypeError: unsupported operand type(s) for -: 'NoneType' and 'int'
```

According to this stack trace, an unhandled error was detected in line 26 of *trickyerror.py* in the method `b()`. However, after a closer look at this line, nothing appears to be wrong!

We are simply subtracting 1 from a number and returning it, which is a perfectly acceptable thing to do - unless the argument isn't a number. By traveling up the stack trace to deeper method calls, and referencing those method calls to our code, we can see that our problem originated from setting a value of `y = None` in line 3 of main, and then passing the bad value of `y` through method calls to `b()` in line 5 of main. The value of `None` lines up with the `TypeError` explanation that was part of the stack trace, since Python does not permit any arithmetic manipulation of `None`. Two reasonable fixes we can now implement are to remove the call `a(y)` in line 5, or set `y` to be a number in line 3.

This highlights an important distinction between where an error's cause may originate, and where an error may be discovered or caught. Stack traces will always highlight the line where an error was first discovered or caught at the top of the stack, though that line may not necessarily be the cause of the error. When debugging using a stack trace, it is recommendable to first identify what, in the line at the top of the trace, may have received the error named by the stack trace. From there, identify what may have caused any bad values to appear in this top line, and work down from the top of the call stack into earlier method calls to determine an appropriate fix.

The two examples provided above are nowhere near as cool as the code you will be writing in CS16. Consequently, any errors you may find in your work may be more difficult to diagnose, and the accompanying stack traces may be more complicated to understand. For example, stack traces can stretch across different files and even extend through a dozen methods.

Though call stacks and stack traces can be much more complicated than what was covered above in this guide, the basic strategy we recommend to approaching and using stack traces in debugging remains the same. Start by examining the error that was caught in the top method in the call stack, and use the stack trace to backtrack through your program's frozen state to figure out how to fix the problem. The lines and variables highlighted by the stack trace can help build a picture of the state of the program when the error was encountered, which in turn can help in devising an appropriate solution.

Common Exceptions and Errors

There are a wide variety of exceptions and errors in Java and Python that you may encounter in this course. The Javadocs[3] and Python docs[4] provide more information than you may ever need, and will be very helpful in understanding these exceptions. We have included a few of the more common ones here, with both a brief description of the error as well as likely causes.

Java

- `NullPointerException`: A method was performed on a null object, or a method was executed by a null object. Try using a debugger or print lines to print the state (null or not) of every object in the topmost line of the resulting stack trace. Once the null object has been identified, work backwards to determine why it was null, and from there, what changes can be made to handle null values.
- `IndexOutOfBoundsException`: A data structure was accessed at an index out of bounds. Try using a debugger or print lines to follow the execution of code before the line at the top of the stack trace is called. Double check your math behind your indexing, and make sure the size of your data structure is appropriate.
- `OutOfMemoryException`: Java ran out of memory to allocate to your program. In the context of CS16, this most likely means that there is an infinitely recurring loop somewhere in the flow of the program execution.
- `ConcurrentModificationException`: A data structure was modified while iterating through it. Most data structures in Java permit you to both iterate and modify the underlying data; however, using fail-fast iterators to accomplish this can often lead to this exception. Fail-fast iterators are generated by using a for-each loop. Consider converting the for-each loop highlighted by the stack trace into a for loop or while loop, or redesign the logical flow of your code to instead iterate through the data structure and separately modify it.
- `IllegalArgumentException`: A method was passed an illegal argument. In the top line of the stack trace, examine each argument to make sure you are passing arguments appropriately.
- `ArithmeticException`: An exceptional arithmetic operation" was attempted. These include divide-by-zeros. Try using a debugger or print lines to identify the values of variables in the top line of the stack trace, and work backwards from there. No project or homework assignment in CS16 will ask you to divide by zero or compute any arithmetic that would create this exception.

[3] `http://docs.oracle.com/javase/7/docs/api/java/lang/RuntimeException.html`
[4] `https://docs.python.org/2/library/exceptions.html`

Python

- `TypeError`: A method or operation was applied to an object or was executed by an object of the incorrect type. Make sure that the type of all objects in the most recent line of the stack trace is what you expect, and think of all ways where the types may not be as desired. A common situation where this arises is when arithmetic operations, such as addition, subtraction, multiplication, and division, are applied to strings.

- `UnboundLocalError`: A reference is made to a variable that has had no value bound to it yet. For each variable in the top line of the stack trace, assert that your program assigns a value to the variable before referring to it. Additionally, make sure that variable names are spelled correctly, and that variable names are different from method names.

- `IndexError`: A sequence subscript is out of range. Try using a debugger or print lines to follow the execution of code before the most recent line in the stack trace. Confirm that the size of your data structure is appropriate for the logic you are using in indexing it.

- `AttributeError`: Raised when an attribute reference or assignment fails. If your error is of the form "`AttributeError: NoneType has no attribute x`", then the object on which an attribute reference or assignment is attempted has a value of `None`. That is, if the stack trace reports an error of "`AttributeError: NoneType has no attribute x`" to a line that consists of `node.x = 4`, then `node` has a value of `None`.

- `NameError`: Often arises when a local or global name is referenced but not found. Make sure you have spelled everything correctly, and confirm that methods and variables have different names.

- `ZeroDivisionError`: Dividing by zero was attempted. Try using a debugger or print lines to identify the values of variables in the top line of the stack trace, and work backwards from there. No project or homework assignment in CS16 will ask you to divide by zero or compute any arithmetic that would create this error.