Code Analysis

Code analysis is a vital skill to master when it comes to coding. Simple programs may work on the first attempt or after some minor edits to logic or to fix accidental typos, but when it comes to more complex projects or reading other people's code, this skill becomes indispensable.

Knowledge, which is the term used in Zybooks, is exactly what is means: the collection of facts, information, and skills pertaining to a certain topic or concept. The more you learn about Python (or really anything), the more knowledge you have of it. The deeper the knowledge, the more information you can use to analyze code to figure out what it does or what has gone wrong. A simple example is if I asked you what the code in Example 8a-1 does:

In [ ]:
```
# Example 8a-1: Simple code snippet

for i in range(10):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

Although I provide the output, it should clear that this `for` loop outputs the numbers 0 through 9 on separate lines. Although there are only two lines of code, there's actually a lot of details you can pull out of these lines based on prior knowledge:

- `for` loop is used so the loop runs a limited amount of times
  - `range(10)` means the loop will run exactly 10 times
    - only `10` is provided as an argument so all the following are true:
      - 0 is the starting number
      - 9 is the ending number
      - Every loop iteration, the number will increase by one
  - the name right after `for` is `i`, so the values from `range` will be stored in `i`
    - the value of `i` will be updated every iteration
    - `i` will hold an `int` value
- `print()` without changing the ending will always print a newline at the very end
  - next output will be on the next line
- `i` is being printed out without any changes, so each line with just be the current value of `i`

Some of these things are trivial, but as you can see, there's a lot of information in just these two lines of code. Realistically, you'll be analyzing chunks of code in class or even whole segments and programs and finding out what everything does. Although this course covers the foundation of coding, each concept covered has a lot of depth detail-wise, especially when combining concepts together. Knowing every single detail about every single concept so far is not a necessity: you just need to remember every single detail that we tell you to remember.

Bug and Error Hunting

Generally speaking, bugs are problems with your code that lead to unexpected results, including errors. Some bugs do not cause an error message, they just cause a weird result: these are quite hard to deal with since Python won't really tell you where the problem is. Some bugs do give you an error message, which gives you valuable information of where the problem has been found (but not necessarily started at).

With whatever information you get, whether it's the weird outputs or the error message, along with whatever information you have in your brain, you must figure out which part of your code is *not written properly*. You may commonly hear people say that their code does not work. By this, they mean that their code *does not work as intended*. Remember that coding languages will execute code exactly as you write it: they will follow your instructions perfectly, but any mistake on your end will lead to weird results. Do not go into your code assuming all your code is correctly written but rather assume everything could be wrong and verify that each chunk or section of your code is written as intended and for the right reason. This is arguably the core of debugging.

Example 8a-2 shows a common bug with students first dealing with `while` loops and conditionals put together:

In [ ]:
```
# Example 8a-2: While + if/else bug

x = 0
while x < 10:
    x += 1
    if x % 2 == 1:
```

```
        print(x, "Odd")
else:
    print(x, "Even")
```

```
1 Odd
3 Odd
5 Odd
7 Odd
9 Odd
10 Even
```

The code at first glance may look as if it should alternate between `"Odd"` and `"Even"` for the output. However, it only outputs `"Odd"` for a few times and then `"Even"` at the very end. The bug here is caused by the `else` block being indented improperly: it is connected to the `while` instead of the `if` statement. This is a small bug and may be simple enough to notice at first glance, but the reason I wanted to use this as an example is because of the weird placement of the `else`.

Generally, when conditionals are taught, the `else` is always connected to the `if` statement, and when the `while` loop is taught, the code consists of just the `while` statement and some code inside of it. This for the grand majority of cases is perfectly fine. However, for Python's `while` loop, you can actually use an `else` statement to make the code do something immediately after exiting the `while` loop. Without an error message (as Example 8a-2 is legal code), new coders can be a bit confused of where to start if they do not realize the indentation error.

Most of the times, code will be much longer than Example 2, which means debugging can be a lot more complicated due to the presence of more, diverse elements. In some cases, knowledge alone will not be enough to identify the bug. The code will look super correctly written, and it is beyond your knowledge of what could be wrong (and a lot of times, the bug comes from a certain interaction or behavior that you don't know about). At this point, it's time for some testing.

Hypothesis Testing

When debugging, there are two types of hypotheses that are generally made:

- What could go wrong (finding out where a bug can exist)
- Is it correct (verifying that the code is done correctly)

There are several ways to test, but the three most common ways of testing are:

- Unit testing (testing several values/scenarios to ensure the correct output is done)
- Running with a debugger (checking each step of the code to ensure the correct behavior and to find irregularities)
- Using a lot of print statements (pretty much the easier version of using a debugger, allows flexibility of how you want to see certain values)

Unit testing is just testing your program with various values and scenarios (it's also how your code is checked). You also want to test some "normal" (not special) values to make sure at least the code works in basic scenarios. Then you want to start testing values that are likely to be implemented incorrectly, which tend to be edge cases (special values/scenarios that lead to special outputs) and boundary values (ensuring your code works at either extremes). This is the formal way of saying plug and chug essentially.

The limitation to unit testing is that you only see the output given the input. You don't get to see *how* the input is processed. By using a debugger, which should be built-in with your IDE, you can execute the code step by step to see how all the variables and values change at each point of execution. This is useful if you are confused on how a variable ends up with a certain value, and there are a lot of steps involved.

However, a debugger might be overkill if you have a general idea of what you want to check. Maybe you just want to see a variable change at multiple spots to better locate where the confusion can be before using a debugger, which can be quite slow to use. So you can just place `print()` statements and output the variables you wanna check at whatever valid location you wanna check. The downside to this is that again, you can only see the value itself and not the process, but these values are seen during the process, so this method is a good in between of unit testing and using a debugger.

If my code from Module Five was ever wrong, I can employ a combination of these three to ensure I find and fix the bugs.