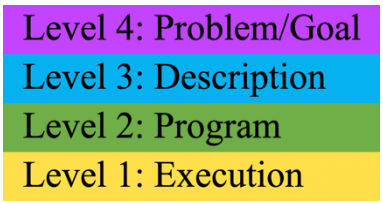
Debugging Toolkit

Here are some tools for your debugging toolkit. These tools are like a hammer, screwdriver, and measuring tape for a carpenter—you're going to need them for almost every project.

Tool #1: *Use the levels of problem solving.*

By the time you're writing your program (level 2) and running it (level 1), you should already have spent some time on level 4 making sense of the problem and level 3 outlining the solution. If you run into issues on the lower levels, *try "zooming out" to a higher level* of problem solving to help you make sense of what's going on in the program and execution.



Tool #2: *Take baby steps.*

As you write your program, run it frequently to test it. If something's not working right, try to fix it right away. This will save you having to search through your entire program later to try to find what's behaving oddly. As you're debugging, *make only one change at a time*, so that you know what works.

Tool #3: *Use comments!*

As you're writing your code, use comments to connect your program (level 2) to your detailed description (level 3). This will help you locate the source of errors more quickly.

Tool #4: *Try to break it.*

When you're testing your program, don't use the "right" inputs and data. *Use a variety of "wrong" inputs* to make sure your program can handle them. For example, if it's supposed to take text input, see how it reacts to numbers or symbols. (Friends are very helpful for this, because they don't have all the expectations you do about your program, so they tend to be more creative in breaking it!)

Tool #5: *Read the error messages.*

If Python finds an error while running your code (level 1), it will give you a nice, helpful message telling you what kind of error it is and where in your program it is. The backside of this sheet has an explanation of the most common error messages.

Tool #6: *Add* `print()` *statements.*

If you're not exactly sure where your program is going wrong, add some print statements around where you think the issue might be happening. Print the current values of variables, their types, or their lengths (`len(variable)`). This can help you figure out which variable is misbehaving, and where the error originates.

Tool #7: *Use* `try/except` *statements.*

These are similar to conditionals, but the `try` section holds your default code, and `except` holds alternate code to use if the `try` section produces an error. This is a great way to catch errors based on user input.

```
try:
  <default code>
except:
  <alternative code>
```

Tool #8: *Ask yourself these questions:*

What am I trying to do? What have I tried already? What else could I try? What would happen if…?

Created by Emmett Wald, July 2021

Decoding Python Error Messages

Error messages are really common, but that doesn't make them any less frustrating or annoying! Here are some tips for making sense of them, plus a guide to the most common error messages.

Tip #1: *Read the error message from bottom to top.*

Error messages sometimes print many lines, tracing how Python ran into the error message. If you're using functions or outside modules, this can get complicated fast. The last line of the error message will state **what type of error you got**, and the line before that will state **where in your program the error occurred**.

Tip #2: *Backtrack.*

If you can't find an issue with the specific line of code the error message pointed out, you can backtrack in two different ways. The first way is to look at the **previous line of code**. The second way is to look at the **code that's connected** to the error line, such as a function being defined or called, or a variable being defined or used.

Tip #3: *Get to know the most common types of errors.*

Some error messages are more common than others. These five are the ones you're most likely to encounter.

`SyntaxError`

Syntax is the rules about how a certain programming language works. A syntax error in Python usually means you forgot to close parentheses `()` or quotes `''` `""`, you forgot a colon `:`, or you put a symbol somewhere Python wasn't expecting it.

`IndentationError`

Indented blocks should be used for conditionals, loops, and defining functions. An indentation error means either you indented something that shouldn't have been indented, you didn't indent something that needed to be, or your indentations aren't quite lined up properly.

`IndexError`

An index error means you used an index value that doesn't exist (for example, your list has four items in it and you called on index 6).

`NameError`

A name error happens when Python can't find anything with the name you used. Either you tried to use a variable or function before you defined it (or didn't define it at all), or there's a typo in the name of a variable or function.

`TypeError`

A type error means that you tried to do something that can't be done to that type of variable. This most often happens with numbers, when a number needs to be converted from an integer to a string for what you're trying to do, or vice versa.

Created by Emmett Wald, July 2021