Debugging Your Python Code: For Dummies

Tyler J. Metivier
University of Connecticut
Dept. of Physics

May 4, 2018

1 What's the problem?

It doesn't matter if you've written 1 script or programmed a space shuttle launch- everyone's code breaks at some point. What's **more** important is that you know how to interpret the error you've encountered and recover. Tears won't suddenly make your Numpy array callable.
Generally, there are three different types of errors in Python:

1. Syntax errors
2. Logic errors
3. Exceptions

From personal experience, I can say that there are myriad ways to end up with one of the errors listed above- but there are more fundamental issues that could plague your code.
Despite efforts to standardize the newest release of Python (3), many programmers and businesses have stuck with Python 2. The "end-of-life" date (*the predetermined date in which the product becomes obsolete*) for Python 2 has even been extended 5 years until 2020 because of how many people currently rely on it's support.

So why don't people just update their code?

Python 2 and Python 3 are incredibly similar, but there are some large differences in how they accomplish tasks. Python 2 contains many built-in functions and dependencies that simply don't exist in Python 3. This can be a major problem if you want to work in Python 3 but your code is based on a function that doesn't exist anymore.
So what if you run into this issue? Perhaps you were browsing Github and downloaded some interesting-looking code to you. You try to run it and...

```
  File "test.py", line 4
    print"test"
               ^
SyntaxError: invalid syntax
```

The terminal returns to us: the file name, the line that triggered the error, and the type of error, respectively.
So we see that our file contains a syntax error in line 4 (more on syntax errors in section 2). By inspection, we can see that the line, print"test", does **NOT** have parentheses around the argument: print("test"). We received this error because we tried running a code written in Python 2 on our system running Python 3[1]. The line above is completely valid in earlier releases of Python, but now it isn't. This is one example of how Python 3 requires more "order" than its predecessors. Indentation and syntax matter in the long run when codes become complicated. Luckily, Python 3 comes with a very helpful script that you can call in any directory: 2to3.
Now I could open my code and go through adding () after every "print" I see, but what if there are thousands of lines or more errors hiding? Accomplishing tasks systematically is generally much better practice than doing it manually. Looking at our original problem:

```
2  test = "Hey how are you"
3
4  print"test"
```

We see the missing parentheses on line 4, just like we saw before. Now we can enter the following into the terminal (the file here is called test.py):

```
TYLERs-MacBook-Pro:~ tylermetivier$ 2to3 test.py
```

[1]To find out which version of Python you have, type: "python -v" into the terminal and it will return the installation directory and version.

Python's 2to3 script uses something called a "fixer" to interpret code, identify what's outdated, and change it into working code. Fixers work by reading through code and identifying Python 2-specific blocks of text. When a piece of outdated code is recognized, 2to3 will scan through its database to identify what the Python 3-equivalent is. Simply running this program (as shown above) will rewrite your original file with the implemented fixes. However, 2to3 always saves a backup[2]. If we run the same line again but with a -w after 2to3, this will restore the source file to its original form.

```
RefactoringTool: Skipping optional fixer: buffer
RefactoringTool: Skipping optional fixer: idioms
RefactoringTool: Skipping optional fixer: set_literal
RefactoringTool: Skipping optional fixer: ws_comma
RefactoringTool: Refactored test.py
--- test.py     (original)
+++ test.py     (refactored)
@@ -1,4 +1,4 @@

 test = "Hey how are you"

-print"test"
+print("test")
RefactoringTool: Files that need to be modified:
RefactoringTool: test.py
```

As seen to the right, 2to3 subtracted the line that was causing trouble and then put it back with parentheses. This can be a big time-saver depending on what you're working with.

Now if we reopen the file, we see that parentheses have indeed been added.

```
2  test = "Hey how are you"
3
4  print("test")
```

But of course, most errors aren't because of conversion issues. The generalized type of error we saw here was a **syntax error**. Syntax errors are "fatal"- meaning that when one is encountered, the code will fail to execute and the script will stop being read at that line.

Above, we were kicked out of the code at line 4 when Python didn't understand the print argument. When Python encounters a line of code it can't read, it is a syntax error. Above is an example of an incorrect argument. Syntax errors are also commonly due to typos or incorrect indentation (mixed use of spaces and tabs).

[2]Adding a -n to your original call will generate no backup file- this is NOT recommended.

A more complicated situation is the **logic error**. Logic errors could crash your code or seemingly do nothing at all. They often go completely undiscovered because your code may appear to run perfectly fine. However, that doesn't mean that it's running correctly. Even though your code may appear to be fine, you may have used the wrong variable name in a function and your mathematical result is 5 orders of magnitude different than expected.
Lastly, we have the **exceptions**. This occurs when Python correctly interprets the code you wrote, attempts to execute it, and for whatever reason cannot. For example, if my code relied on a web-based repository and I attempt to run it while on vacation at the Hubble Space Telescope- it may throw an exception due to me not having the HST wifi-password (because I'm in space... get it?). Python knows what I want to do, it just can't do it because there's no internet connection.

2 How do I fix it?

After identifying the type of error, we can begin to try and figure out how to solve the problem.

Dealing with syntax errors is generally not a very difficult task. Because they are mostly comprised of typos, inconsistent indentation, and incorrect usage of arguments- we can start fixing the problem when we know where it is. Just like when we found the syntax error before, the terminal tells us the file that caused it and the exact place where the code breaks down.
If the problem seems clear to you (made a typo, forgot parentheses, etc...)- simply attempt to implement the fix and rerun your code. If you can't identify the issue, try browsing a website like StackExchange to see if other people have encountered the same problem[3].
Learning Python means learning how other people code and deal with their problems, as people often find "better" and easier ways to accomplish such things.
The most frustrating part about logical errors is that they are occasionally invisible. If you've encountered a logic error, the first thing you can do is look over your code entirely. The issue may be easy to identify- you may

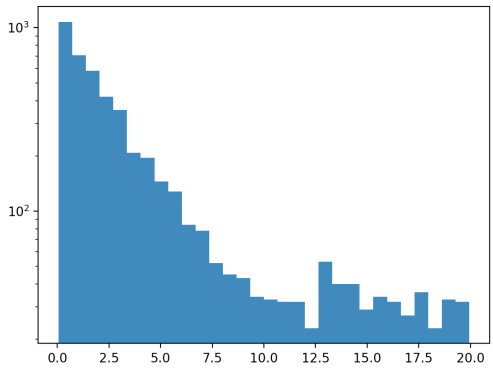
[3]Link: https://stackoverflow.com/questions/tagged/python

have meant "x1" and accidentally put "x!". Or your code may be comprised of many convoluted mathematical functions that would take an incredible amount of time to pour through.
If you **really** want to rid your work of logic errors- the recommended course of action is to install a debugging software. Python comes with a debugging module "pdb" installed, so this would be the best one to try initially.

```
TYLERs-MacBook-Pro:~ tylermetivier$ python -m pdb test.py
```

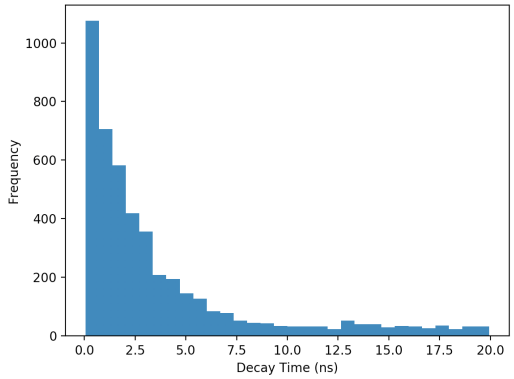
We can debug a code of our choice with the terminal entry above. (Hopefully) the debugger will identify issues in your code and implement fixes.
If all else fails, you can go line by line and try to make sense of what could be going wrong. Python will do exactly what you tell it to do- as long as you know what you're **actually** telling it to do. This is why debuggers can be helpful when the source of the error is unknown. In a similar fashion to 2to3, debugging software looks for identifiable pieces of code and attempts to implement corrections. Naturally, this method is not always perfect.

In science, being mindful of your expected results is vital. If you run your program and receive a value, take time to consider if it's reasonable or makes sense given the situation. If you are expecting to receive a histogram with an exponential trend but instead receive:



-you may initially want to run your fancy debugging software to fix the issue. But that won't help you whatsoever in this case. Nothing is *actually* wrong with your code. Instead, if we look at the generated plot and notice

that something's funky about the scale on the vertical axis- we are halfway through solving our problem.
We may discover that a lonesome "log=True" was the culprit all along. Removing that argument then generates:



We now have our expected exponential trend. But still there was nothing ever *wrong* with the code- we were just accidentally telling it to do something and it listened.

Exceptions are the least frightening to deal with, as this means that your code is (probably) not broken. You are requesting something from Python and it understands your request- it just can't help you. That means it's all on you here. Once again, we must look and see where the terminal is telling us the error is. Once we've found the location, we can implement fixes.
If we find the exception is thrown because there isn't internet and we're asking Python to go to a website- we either need to get a connection or work around the issue.
Creating an "exception handling block" can allow us to bypass exception-related errors. This is most commonly done with a "try-except block." An example of a try-except block is:

```
try:
    catalog = np.loadtxt('directory_catalog_135.txt',
                dtype={'names': ('subdirs', 'galaxy_numbers', 'galaxy_masses'),
                'formats': ('S3', 'U10', 'f8')})
except:
    os.system("wget " + dl_base + "/files/directory_catalog_135.txt")
    catalog = np.loadtxt('directory_catalog_135.txt',
                dtype={'names': ('subdirs', 'galaxy_numbers', 'galaxy_masses'),
                'formats': ('S3', 'U10', 'f8')})
```

Under both try and except, we see we are ultimate attempting to call a text file and call it "catalog." We first **try** to simply call the file from our system. If the file exists within your working directory, the catalog should be read with no problem. If this file doesn't exist, we attempt to call the file in an alternative method with **except**.
As shown above, we first attempt to download the text file with os.system(...). Here, we are telling our computer to "wget"[4] (install from the internet) our file from a web page, "dl_base." In this specific situation, we can define dl_base as some url to the website we want to get our file from. There is also "/files/directory_catalog_135.txt" appended at the end of the line. This just specifies the exact file that we want to download from this website.

If your computer doesn't have the file **AND** doesn't have internet... it may be time to call a local internet service provider.

Despite the fact that Python errors come in many different forms, it's good to have a methodology when working through them.

1. Read your code thoroughly before attempting to run it.
2. Identify the type of error you may be receiving.
3. Based on the type of error, use the provided suggestions or research how to bypass your issues.
4. Never be afraid to use the internet as a guide- chances are that someone in the world has already encountered the issue you're facing.

[4]Wget is a very common program that allows you to download content from the internet through the terminal/command line. (https://www.gnu.org/software/wget/)