Advanced Topics

1. Advanced topics

00:00 - 00:10

You've learned a lot about how to using and writing context managers. In this lesson, we'll cover nested contexts, handling errors, and how to know when to create a context manager.

2. Nested contexts

00:10 - 00:33

Imagine you are implementing this copy() function that copies the contents of one file to another file. One way you could write this function would be to open the source file, store the contents of the file in the "contents" variable, then open the destination file and write the contents to it. This approach works fine until you try to copy a file that is too large to fit in memory.

3. Nested contexts

00:33 - 00:56

What would be ideal is if we could open both files at once and copy over one line at a time. Fortunately for us, the file object that the "open()" context manager returns can be iterated over in a for loop. The statement "for line in my_file" here will read in the contents of my_file one line at a time until the end of the file.

4. Nested contexts

00:56 - 01:42

So, going back to our copy() function, if we could open both files at once, we could read in the source file line-by-line and write each line out to the destination as we go. This would let us copy the file without worrying about how big it is. In Python, nested "with" statements are perfectly legal. This code opens the source file and then opens the destination file inside the source file's context. That means code that runs inside the context created by opening the destination file has access to both the "f_src" and the "f_dst" file objects. So we are able to copy the file over one line at a time like we wanted to!

5. Handling errors

01:42 - 02:28

One thing you will want to think about when writing your context managers is: What happens if the programmer who uses your context manager writes code that causes an error? Imagine you've written this function that lets someone connect to the printer. The printer only allows one connection at a time, so it is imperative that "p.disconnect()" gets called, or else no one else will be able to print! Someone decides to use your get_printer() function to print the text of their document. However, they weren't paying attention and accidentally typed "txt" instead of

"text". This will raise a KeyError because "txt" is not in the "doc" dictionary. And that means "p.disconnect()" doesn't get called.

6. Handling errors

02:28 - 02:53

So what can we do? You may be familiar with the "try" statement. It allows you to write code that might raise an error inside the "try" block and catch that error inside the "except" block. You can choose to ignore the error or re-raise it. The "try" statement also allows you to add a "finally" block. This is code that runs no matter what, whether an exception occurred or not.

7. Handling errors

02:53 - 03:13

The solution then is to put a "try" statement before the "yield" statement in our get_printer() function and a "finally" statement before "p.disconnect()". When the sloppy programmer runs their code, they still get the KeyError, but "finally" ensures that "p.disconnect()" is called before the error is raised.

8. Context manager patterns

03:13 - 03:35

If you notice that your code is following any of these patterns, you might consider using a context manager. For instance, in this lesson we've talked about "open()", which uses the open/close pattern, and "get_printer()", which uses the connect/disconnect pattern. See if you can find other instances of these patterns in code you are familiar with.

1. [1] Adapted from Dave Brondsema's talk at PyCon 2012: https://youtu.be/cSbD5SKwak0?t=795

9. Let's practice!

03:35 - 03:41

Test your understanding with the next few exercises.