**Using context managers**

In this lesson, I'll introduce the concept of context managers and show you how to use these special kinds of functions.

**2. What is a context manager?**

A context manager is a type of function that sets up a context for your code to run in, runs your code, and then removes the context. That's not a very helpful definition though, so let me explain with an analogy.

**A catered party**

Imagine that you are throwing a fancy party, and have hired some caterers to provide refreshments for your guests.

Before the party starts, the caterers set up tables with food and drinks.

Then you and your friends dance, eat, and have a good time.

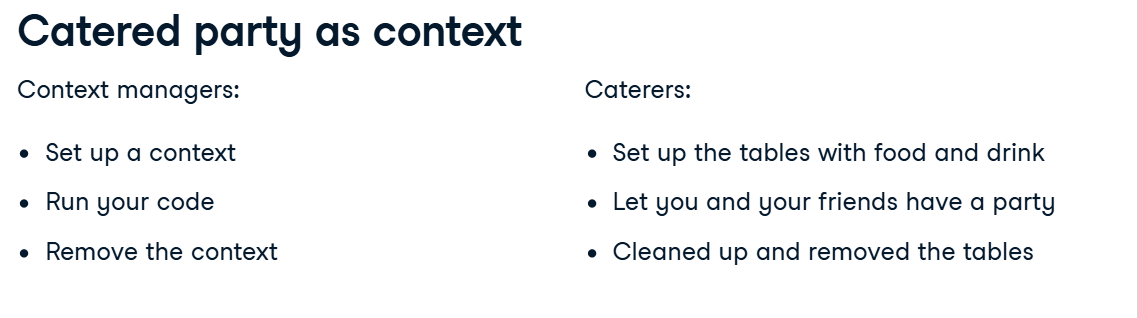
When the party is done, the caterers clean up the food and remove the tables.

**Catered party as context**

In this analogy, the caterers are like a context manager. First, they set up a context for your party, which was a room full of food and drinks. Then they let you and your friends do whatever you want. This is like you being able to run your code inside the context manager's context. Finally, when the party is over, the caterers clean up and remove the context that the party happened in.

**A real-world example**

You may have used code like this before. The "open()" function is a context manager. When you write "with open()", it opens a file that you can read from or write to. Then, it gives control back to your code so that you can perform operations on the file object. In this example, we read the text of the file, store the contents of the file in the variable "text", and store the length of the contents in the variable "length". When the code inside the indented block is done, the "open()" function makes sure that the file is closed before continuing on in the script. The print statement is outside of the context, so by the time it runs the file is closed.



**Using a context manager**

Any time you use a context manager, it will look like this. The keyword "with" lets Python know that you are trying to enter a context.

Then you call a function. You can call any function that is built to work as a context manager. In the next lesson, I'll show you how to write your own functions that work this way.

A context manager can take arguments like any normal function.

You end the "with" statement with a colon as if you were writing a for loop or an if statement.

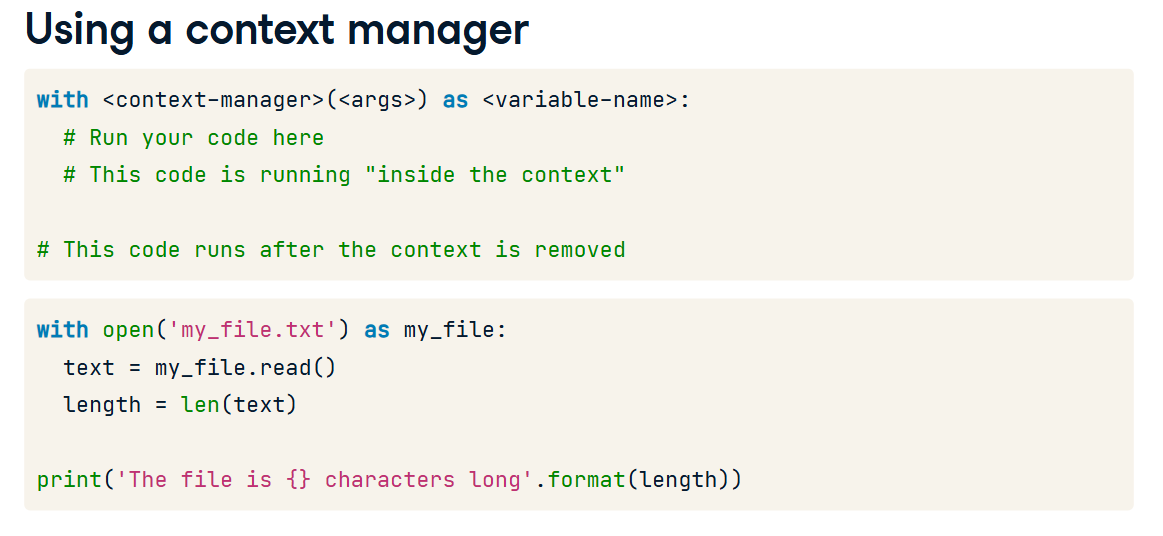


Statements in Python that have an indented block after them, like for loops, if/else statements, function definitions, etc. are called "compound statements". The "with" statement is another type of compound statement. Any code that you want to run inside the context that the context manager created needs to be indented.

When the indented block is done, the context manager gets a chance to clean up anything that it needs to, like when the "open()" context manager closed the file.

**Using a context manager**

Some context managers want to return a value that you can use inside the context. By adding "as" and a variable name at the end of the "with" statement, you can assign the returned value to the variable name. We used this ability when calling the "open()" context manager, which returns a file that we can read from or write to. By adding "as my\_file" to the "with" statement, we assigned the file to the variable "my\_file".



**Writing context managers**

Now that you know how to use context managers, I want to show you how to write a context manager for other people to use.

**Two ways to define a context manager**

There are two ways to define a context manager in Python: by using a class that has special \_\_enter\_\_() and \_\_exit\_\_() methods or by decorating a certain kind of function.

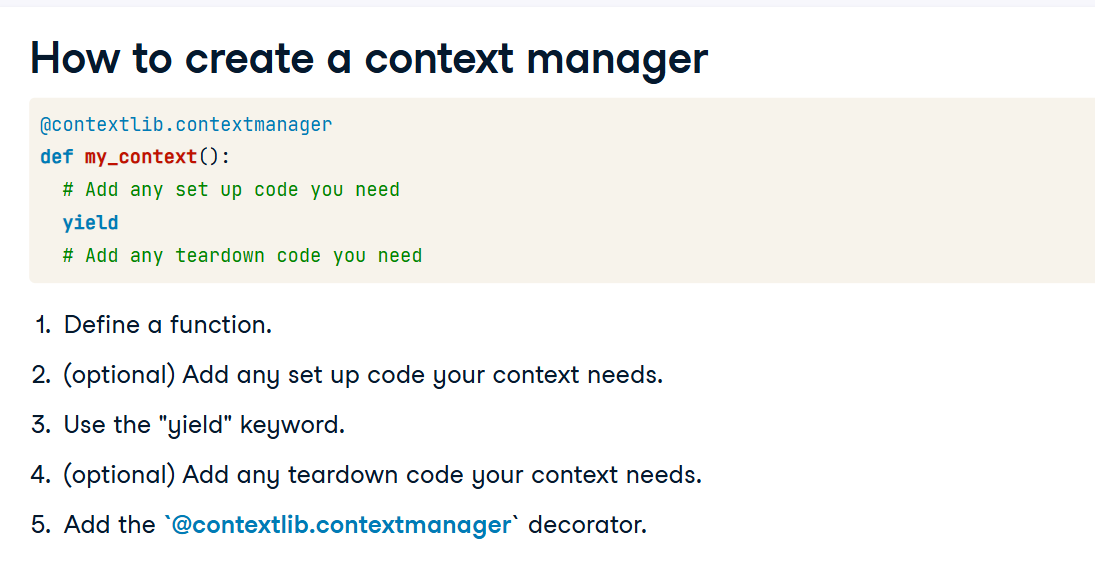
**Two ways to define a context manager**

This course is focused on writing functions, and some of you may not have been introduced to the concept of classes yet, so I will only present the function-based method here.

**How to create a context manager**

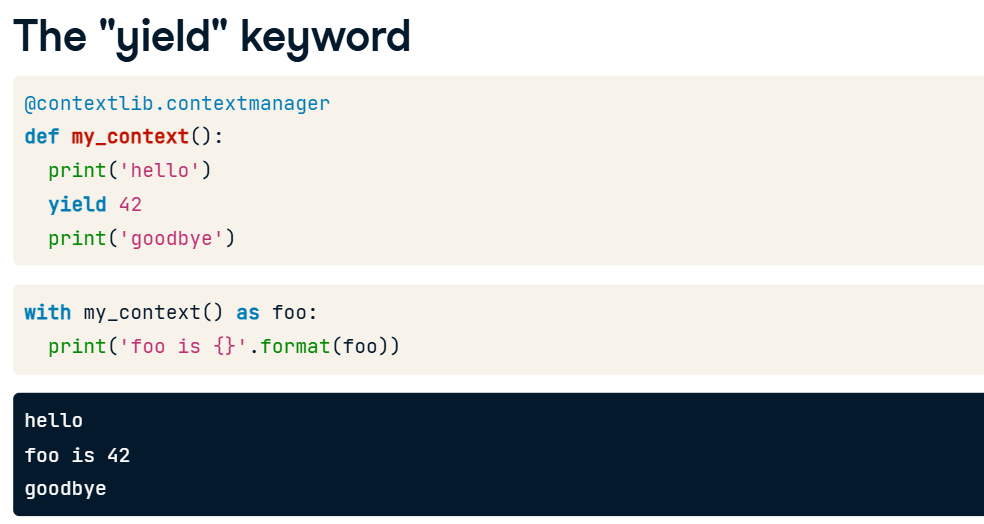
There are five parts to creating a context manager. First, you need to define a function. Next, you can add any setup code your context needs. This is not required though. Third, you must use the "yield" keyword to signal to Python that this is a special kind of function. I will explain what this keyword means in a moment. After the "yield" statement, you can add any teardown code that you need to clean up the context.

Finally, you must decorate the function with the "contextmanager" decorator from the "contextlib" module. You might not know what a decorator is, and that's ok. We will discuss decorators in-depth in the next chapter of this course. For now, the important thing to know is that you write the "at" symbol, followed by "contextlib.contextmanager" on the line immediately above your context manager function.



**The "yield" keyword**

The "yield" keyword may also be new to you. When you write this word, it means that you are going to return a value, but you expect to finish the rest of the function at some point in the future. The value that your context manager yields can be assigned to a variable in the "with" statement by adding "as <variable name>". Here, we've assigned the value 42 that my\_context() yields to the variable "foo". By running this code, you can see that after the context block is done executing, the rest of the my\_context() function gets run, printing "goodbye". Some of you may recognize the "yield" keyword as a thing that gets used when creating generators. In fact, a context manager function is technically a generator that yields a single value.

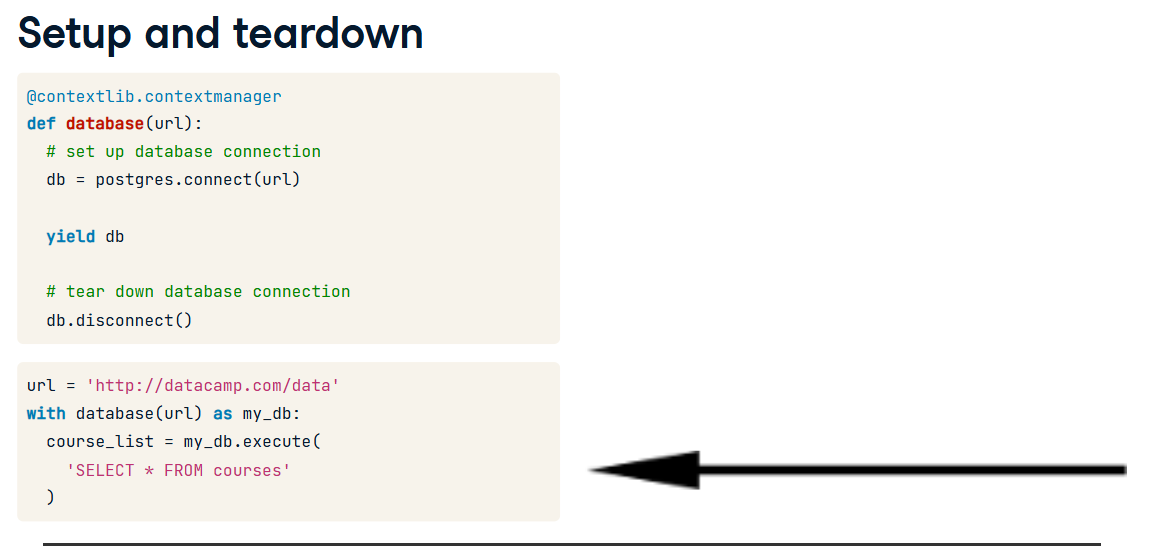


**Setup and teardown**

The ability for a function to yield control and know that it will get to finish running later is what makes context managers so useful. This context manager is an example of code that accesses a database. Like most context managers, it has some setup code that runs before the function yields. This context manager uses that setup code to connect to the database.

Most context managers also have some teardown or cleanup code when they get control back after yielding. This one uses the teardown section to disconnect from the database.

This setup/teardown behavior allows a context manager to hide things like connecting and disconnecting from a database so that a programmer using the context manager can just perform operations on the database without worrying about the underlying details.



**10. Yielding a value or None**

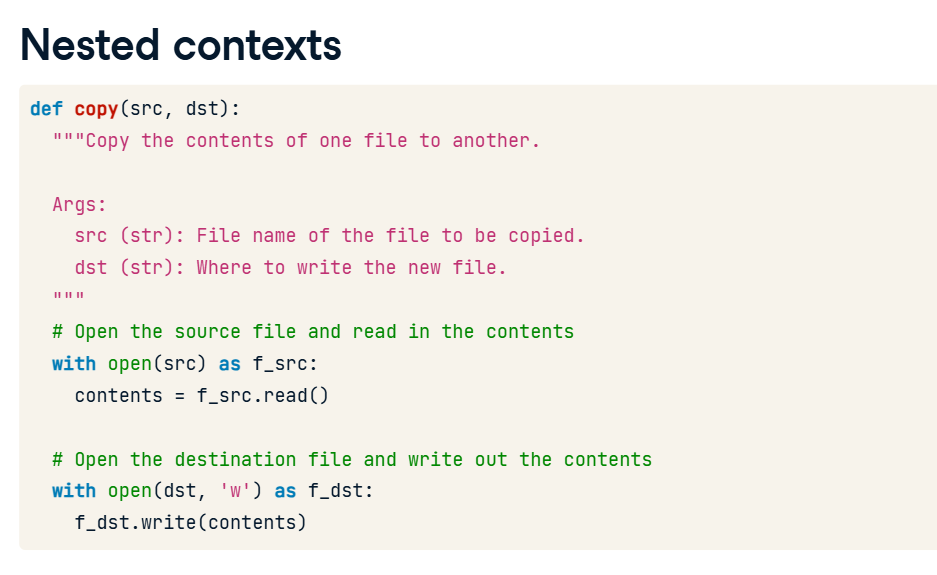
The database() context manager that we've been looking at yields a specific value - the database connection - that can be used in the context block. Some context managers don't yield an explicit value. in\_dir() is a context manager that changes the current working directory to a specific path and then changes it back after the context block is done. It does not need to return anything with its "yield" statement.

**Advanced topics**

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**

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.



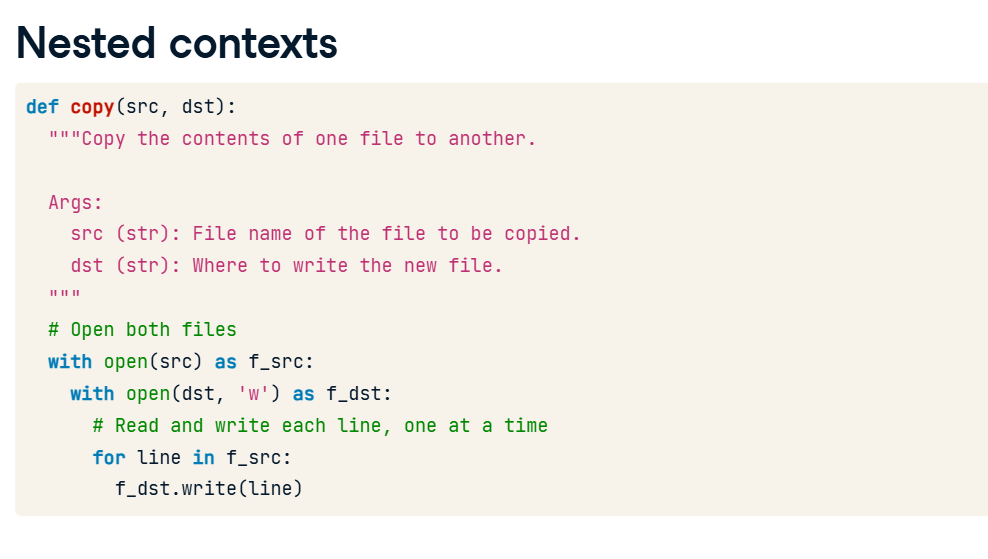
**Nested contexts**

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.



**Nested contexts**

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!



**Handling errors**

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**

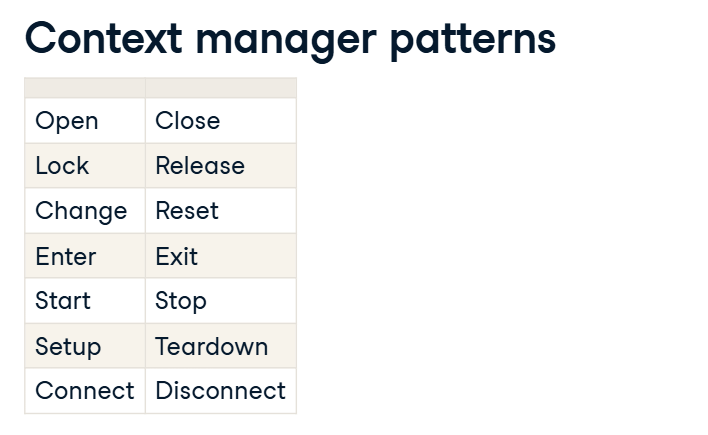
02:53 - 03:13

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.\



**Context manager patterns**

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.

****