Using context managers

WRITING FUNCTIONS IN PYTHON

Shayne Miel

Director of Software Engineering @ American Efficient

datacamp

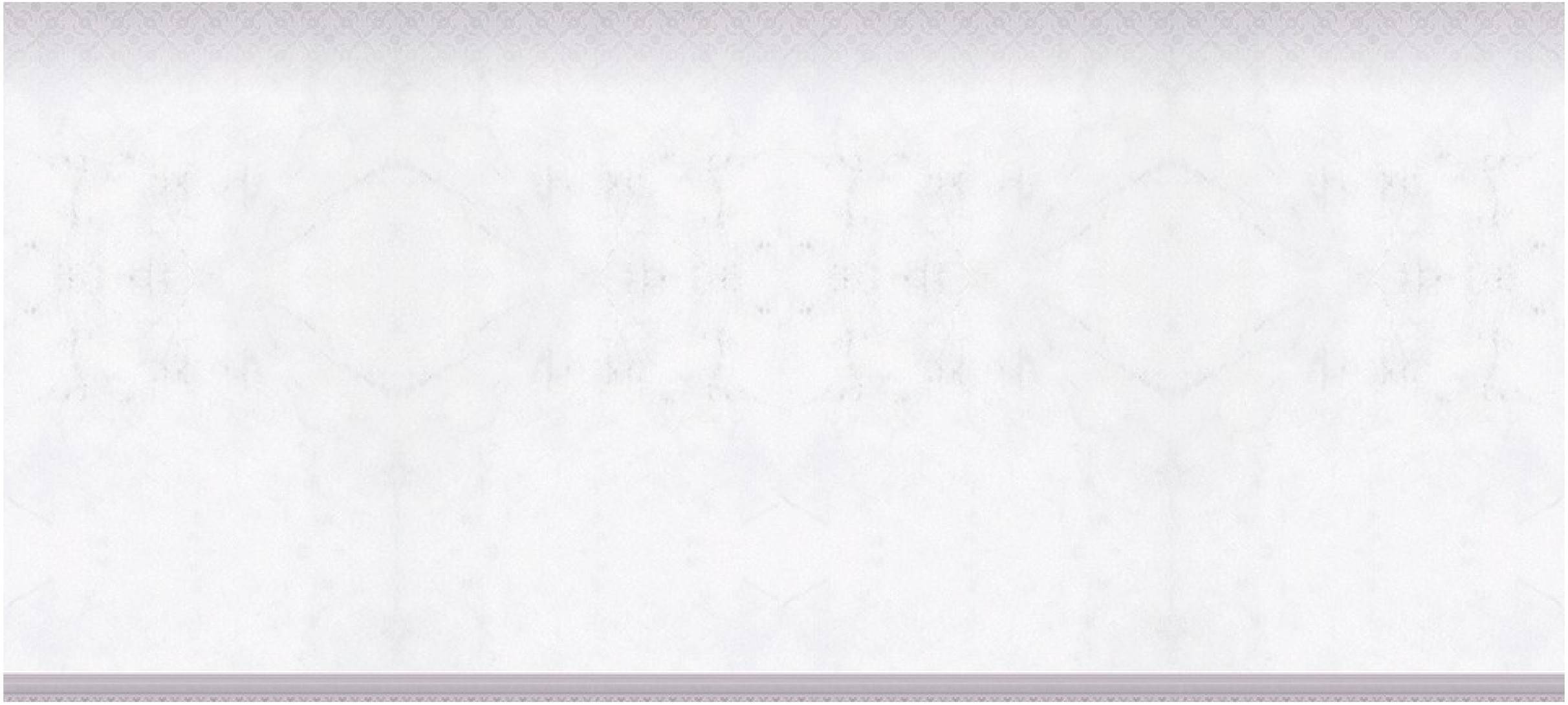
What is a context manager?

A context manager:

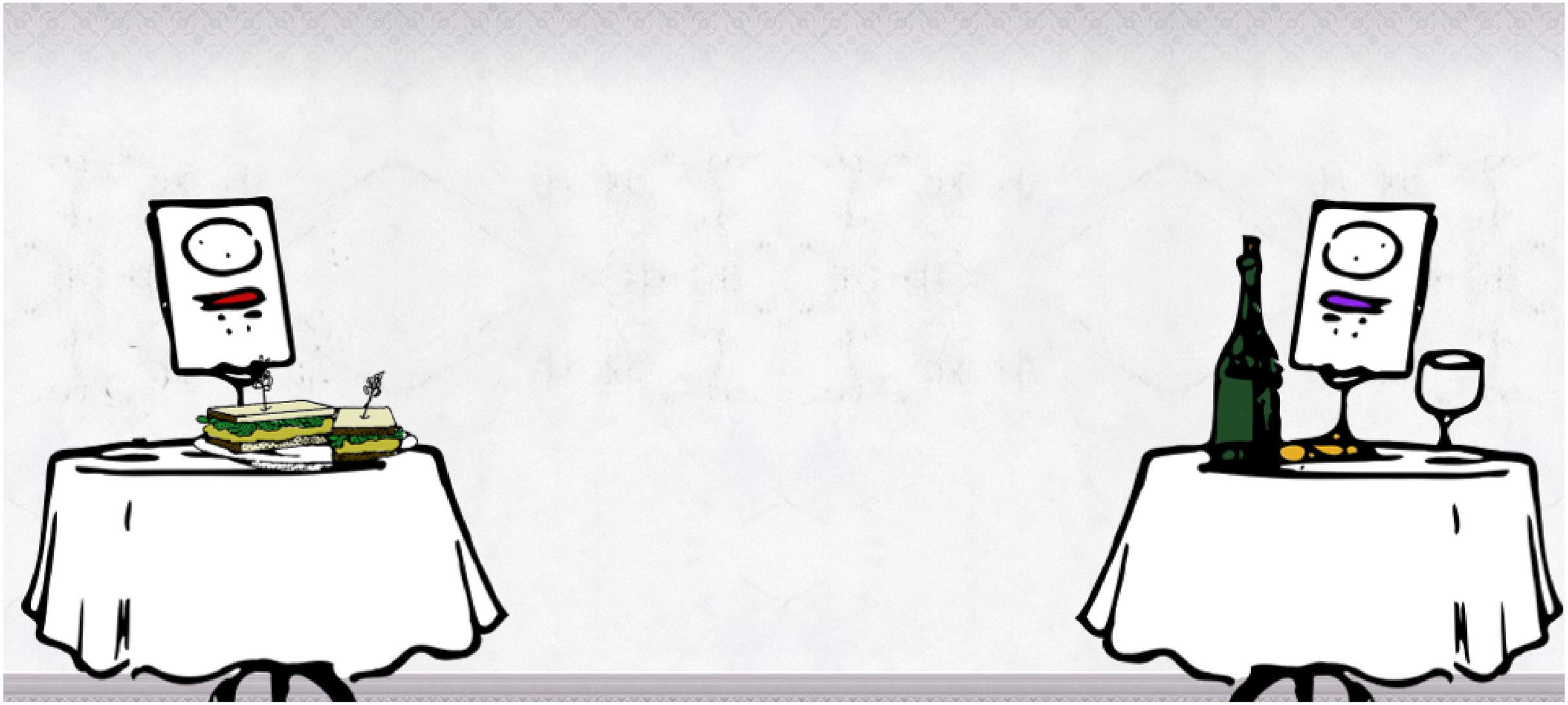
- Sets up a context
- Runs your code
- Removes the context

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

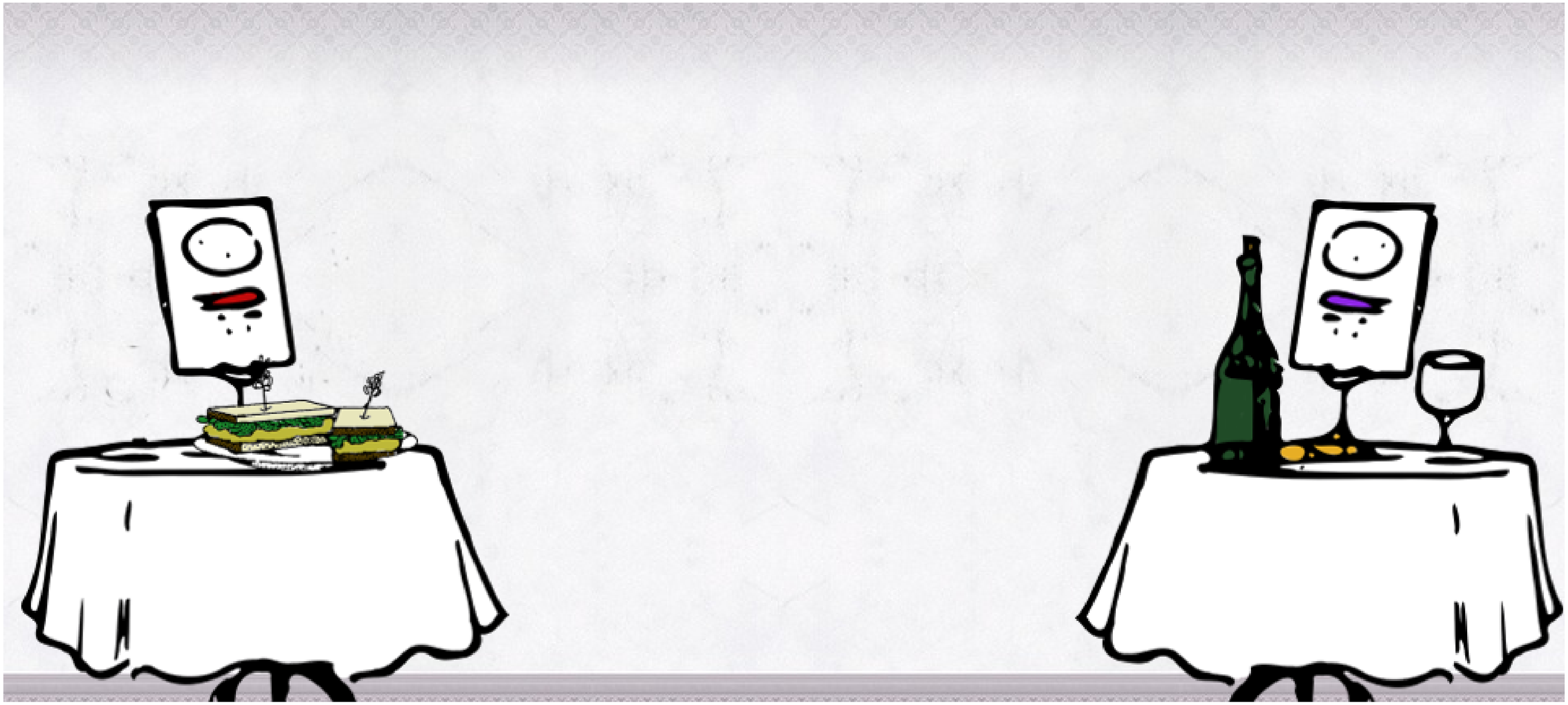


A catered party



A catered party

A catered party



A catered party

Catered party as context

Context managers:

- Set up a context
- Run your code
- Remove the context

Caterers:

- Set up the tables with food and drink
- Let you and your friends have a party
- Cleaned up and removed the tables

A real-world example

```
with open('my_file.txt') as my_file:
  text = my_file.read()
  length = len(text)

print('The file is {} characters long'.format(length))
```

`open()` does three things:

- Sets up a context by opening a file
- Lets you run any code you want on that file
- Removes the context by closing the file

9. 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

```
with
```

Using a context manager

```
with <context-manager>()
```

Using a context manager

```
with <context-manager>(<args>)
```

Using a context manager

```
with <context-manager>(<args>):
```

Using a context manager

```
with <context-manager>(<args>):
  # Run your code here
  # This code is running "inside the context"
```

Using a context manager

```
with <context-manager>(<args>):
  # Run your code here
  # This code is running "inside the context"

# This code runs after the context is removed
```

Using a context manager

```
with <context-manager>(<args>) as <variable-name>:
  # Run your code here
  # This code is running "inside the context"

# This code runs after the context is removed
```

```
with open('my_file.txt') as my_file:
  text = my_file.read()
  length = len(text)

print('The file is {} characters long'.format(length))
```

Let's practice!

WRITING FUNCTIONS IN PYTHON

Writing context managers

WRITING FUNCTIONS IN PYTHON

Shayne Miel
Director of Software Engineering @ American Efficient

datacamp

Two ways to define a context manager

- Class-based
- Function-based

Two ways to define a context manager

- Class-based
- **Function-based ***

3. 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

```
def my_context():
  # Add any set up code you need
  yield
  # Add any teardown code you need
```

4. 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.

1. Define a function.
2. (optional) Add any set up code your context needs.
3. Use the "yield" keyword.
4. (optional) Add any teardown code your context needs.

How to create a context manager

```
@contextlib.contextmanager
def my_context():
  # Add any set up code you need
  yield
  # Add any teardown code you need
```

5. How to create a context manager
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.

1. Define a function.
2. (optional) Add any set up code your context needs.
3. Use the "yield" keyword.
4. (optional) Add any teardown code your context needs.
5. Add the `@contextlib.contextmanager` decorator.

The "yield" keyword

```
@contextlib.contextmanager
def my_context():
  print('hello')
  yield 42
  print('goodbye')
```

```
with my_context() as foo:
  print('foo is {}'.format(foo))
```

6. The "yield" keyword
The "yield" keyword 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.

```
hello
foo is 42
goodbye
```

Setup and teardown

```
@contextlib.contextmanager
def database(url):
  # set up database connection
  db = postgres.connect(url)

  yield db

  # tear down database connection
  db.disconnect()
```

```
url = 'http://datacamp.com/data'
with database(url) as my_db:
  course_list = my_db.execute(
    'SELECT * FROM courses'
  )
```

7. 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.

8. Setup and teardown
ost 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.

Setup and teardown

```
@contextlib.contextmanager
def database(url):
  # set up database connection
  db = postgres.connect(url)

  yield db

  # tear down database connection
  db.disconnect()
```

```
url = 'http://datacamp.com/data'
with database(url) as my_db:
  course_list = my_db.execute(
    'SELECT * FROM courses'
  )
```

8. Setup and teardown
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.

Setup and teardown

```
@contextlib.contextmanager
def database(url):
  # set up database connection
  db = postgres.connect(url)

  yield db

  # tear down database connection
  db.disconnect()
```

```
url = 'http://datacamp.com/data'
with database(url) as my_db:
  course_list = my_db.execute(
    'SELECT * FROM courses'
  )
```

9. Setup and teardown
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.

Yielding a value or None

```
@contextlib.contextmanager
def database(url):
  # set up database connection
  db = postgres.connect(url)

  yield db

  # tear down database connection
  db.disconnect()
```

```
url = 'http://datacamp.com/data'
with database(url) as my_db:
  course_list = my_db.execute(
    'SELECT * FROM courses'
  )
```

```
@contextlib.contextmanager
def in_dir(path):
  # save current working directory
  old_dir = os.getcwd()

  # switch to new working directory
  os.chdir(path)

  yield

  # change back to previous
  # working directory
  os.chdir(old_dir)
```

```
with in_dir('/data/project_1/'):
  project_files = os.listdir()
```

10. Yielding a value or None
database() context manager that we've been looking at yields a specific value - 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.

Let's practice!

WRITING FUNCTIONS IN PYTHON

Advanced topics

WRITING FUNCTIONS IN PYTHON

Shayne Miel

Director of Software Engineering @ American Efficient

datacamp

Nested contexts

```
def copy(src, dst):
  """Copy the contents of one file to another.

  Args:
    src (str): File name of the file to be copied.
    dst (str): Where to write the new file.
  """
  # Open the source file and read in the contents
  with open(src) as f_src:
    contents = f_src.read()

  # Open the destination file and write out the contents
  with open(dst, 'w') as f_dst:
    f_dst.write(contents)
```

4. 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!

3. 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

```
with open('my_file.txt') as my_file:
  for line in my_file:
    # do something
```

Nested contexts

```
def copy(src, dst):
  """Copy the contents of one file to another.

  Args:
    src (str): File name of the file to be copied.
    dst (str): Where to write the new file.
  """
  # Open both files
  with open(src) as f_src:
    with open(dst, 'w') as f_dst:
      # Read and write each line, one at a time
      for line in f_src:
        f_dst.write(line)
```

Handling errors

```
def get_printer(ip):
  p = connect_to_printer(ip)

  yield

  # This MUST be called or no one else will
  # be able to connect to the printer
  p.disconnect()
  print('disconnected from printer')

doc = {'text': 'This is my text.'}

with get_printer('10.0.34.111') as printer:
  printer.print_page(doc['txt'])
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    printer.print_page(doc['txt'])
KeyError: 'txt'
```

5. 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.

Handling errors

```
try:
  # code that might raise an error
except:
  # do something about the error
finally:
  # this code runs no matter what
```

Handling errors

```
def get_printer(ip):
  p = connect_to_printer(ip)

  try:
    yield
  finally:
    p.disconnect()
    print('disconnected from printer')

doc = {'text': 'This is my text.'}

with get_printer('10.0.34.111') as printer:
  printer.print_page(doc['txt'])
```

```
disconnected from printer
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    printer.print_page(doc['txt'])
KeyError: 'txt'
```

6. Handling errors
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
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

Open	Close
Lock	Release
Change	Reset
Enter	Exit
Start	Stop
Setup	Teardown
Connect	Disconnect

8. 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.

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

Let's practice!

WRITING FUNCTIONS IN PYTHON