# Context Managers: Takeaways

## Syntax

### CONTEXT MANAGERS

- Use a context manager. Assign the value that a context manager yields to a variable in the with statement by adding "as":

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

- Create a context manager:

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

- How to write a try statement:

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

## Concepts

- 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.

- There are five parts to creating a context manager:
  - Define a function.
  - (optional) Add any setup code your context needs.
  - Use the `yield` keyword to signal to Python that this is a special kind of function.
  - (optional) Add any teardown code needed to clean up the context.
  - Add the `@contextlib.contextmanager` decorator.
- The **yield** keyword means that we are going to return a value, but we expect to finish the rest of the function at some point in the future. 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.
- The `try` statement allows you to write code that might raise an error inside the try block and catch that error inside the except block. It also allows you to add a `finally` block. This is code that runs no matter what, whether an exception occured or not.
- If you notice that your code is following any of these patterns, consider using a context manager:
  - OPEN/CLOSE
  - LOCK/RELEASE
  - CHANGE/RESET
  - ENTER/EXIT
  - START/STOP
  - SETUP/TEARDOWN
  - CONNECT/DISCONNECT

## Resources

- The `contextlib` module