You worked on Basic Testing Types, chapter 3 of the course Introduction to Testing in Python. Here is what you covered in your last lesson:

You learned about the concept of fixture teardowns in pytest, which are crucial for cleaning up resources after tests to prevent issues like memory leaks and performance degradation. Fixture teardowns are implemented using the `yield` keyword in a pytest fixture, marking the separation between setup and teardown phases. This approach ensures that resources are properly released after a test, maintaining the integrity and performance of the testing environment. Key points include:

- Understanding the importance of teardowns in testing environments to clean up resources and prevent memory leaks, performance issues, and invalid test results.
- Learning how to use the `yield` keyword in pytest fixtures to implement teardowns, with `yield` marking the transition from setup to teardown.
- Recognizing when teardowns are particularly important, such as when dealing with large objects or multiple tests that could interfere with each other without proper cleanup.

Here's an example of implementing a teardown in a pytest fixture:

```
import pytest

@pytest.fixture
def prepare_data():
    data = [i for i in range(10)]
    yield data  # Setup phase ends and teardown begins after this line
    data.clear()  # Clear the data list
    del data  # Delete the data variable
```

This lesson emphasized the necessity of teardowns in any testing scenario involving more than a trivial amount of resources or tests, highlighting how proper cleanup ensures reliable, efficient testing processes.

The goal of the next lesson is to explore advanced features of pytest fixtures, enhancing test efficiency and organization.

```
import pytest


@pytest.fixture
def prepare_data():
return [i for i in range(10)]

def test_elements(prepare_data):
assert 9 in prepare_data
assert 10 not in prepare_data
```

```
def factorial(n):
    if n == 0: return 1
    elif (type(n) == int):
        return n * factorial(n-1)
    else: return -1

# Test case: input of a wrong type
def test_str():
    assert factorial('5') == -1
    print('Test passed')

test_str()
```

```
import pytest
import pandas as pd
import numpy as np

def test_aggregated_is_series(aggregated):
    assert isinstance(aggregated, pd.Series), "aggregated should
be a pandas Series"

def test_aggregated_not_empty(aggregated):
    assert len(aggregated) > 0, "aggregated should have more than
0 rows"

def test_aggregated_dtype(aggregated):
    assert np.issubdtype(aggregated.dtype, np.number), "aggregated
should have a numeric dtype (int or float)"
```

```
def create_list():
  return [i for i in range(1000)]
def create_set():
  return set([i for i in range(1000)])
def find(it, el=50):
  return el in it

# Write the performance test for a list
def test_list(benchmark):
  benchmark(find, it=create_list())

# Write the performance test for a set
def test_set(benchmark):
  benchmark(find, it=create_set())
```

```
-------------------------------------------------------------------
-------------------------------------------------------------------
```

```
def test_list(benchmark):
  # Add decorator here
  @benchmark
  def iterate_list():
    # Complete the loop here
    for el in [i for i in range(1000)]:
      pass

def test_set(benchmark):
  # Add decorator here
  @benchmark
  def iterate_set():
    # Complete the loop here
    for el in {i for i in range(1000)}:
      pass
```