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())
```


When you left 17 hours ago, you worked on Writing tests with unittest, chapter 4 of the course Introduction to Testing in Python. Here is what you covered in your last lesson:

You learned about the concept of integration testing, a crucial software testing method that checks the interactions between different system modules to ensure they work together as expected. Integration testing is vital because it helps identify issues that can arise when modules interact, such as data loss, slow interactions, or version conflicts.

Key points covered include:

- **Integration Testing Definition**: It verifies the interactions between modules, ensuring they perform as intended when combined.
- **Real-World Examples**: Examples like power cables, internet connections, file reading drivers, database connections, and APIs illustrate the concept of integration in everyday technology.
- **Importance of Integration Testing**: You discovered why testing integrations is essential, highlighting potential problems like connection losses, slow interactions, and interface mismatches.
- **Practical Example**: A hands-on example demonstrated how to test the integration between Python and a file system using a "setup file" fixture function in pytest to create and remove a test file, ensuring Python can interact with the file system correctly.

```
import pandas as pd
import pytest

@pytest.fixture
def get_df():
    return pd.read_csv('https://example.com/dataset.csv')

def test_get_df(get_df):
    assert type(get_df) == pd.DataFrame
    assert get_df.shape[0] > 0
```

This code snippet exemplifies how to implement integration tests with pytest, checking the integration between Python's pandas library and a CSV file to ensure data is correctly read into a DataFrame. Through these exercises, you've gained a foundational understanding of integration testing, preparing you to create and run your own tests in real-world projects.

The goal of the next lesson is to learn how to write and run your own unit tests using pytest, focusing on developing thorough test cases for comprehensive coverage.

```
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
```

```
import unittest
import pandas as pd
DF_PATH = 'https://assets.datacamp.com/production/repositories/6253/datasets/
f015ac99df614ada3ef5e011c168054ca369d23b/energy_truncated.csv'

def get_data():
return pd.read_csv(DF_PATH)

def min_country(df):
return df['VALUE'].idxmin()

class TestDF(unittest.TestCase):
def setUp(self):
self.df = get_data()
self.df.drop('previousYearToDate', axis=1, inplace=True)
self.df = self.df.groupby('COUNTRY')\
.agg({'VALUE': 'sum'})

def test_NAs(self):
# Check the number of nulls
self.assertEqual(self.df.isna().sum().sum(), 0)
```

```
def test_argmax(self):
# Check that min_country returns a string
self.assertIsInstance(min_country(self.df), str)

def tearDown(self):
self.df.drop(self.df.index, inplace=True)
```