Mocker

1. `mocker` as a parameter

- `mocker` is **not a Python keyword**.
- It comes from the **`pytest-mock` plugin**, which provides a fixture named `mocker`.
- When you declare it as a parameter in your test function:

```
def test_get_weather(mocker):
```

pytest automatically **injects** the `mocker` fixture (similar to how `tmp_path` or `capfd` work).

🔑 **What it gives you:**

- `mocker` is essentially a wrapper around the **`unittest.mock` library**.
- It has methods like `.patch()`, `.spy()`, `.MagicMock()`, etc.

2. `mocker.patch('main.requests.get')`

- `.patch()` replaces the **real object** with a **mocked one** during the test.
- Here:

```
mock_get = mocker.patch('main.requests.get')
```

    - `requests.get` inside `main.py` is replaced by a mock object.
    - This ensures your test **does not make a real HTTP request** (fast, reliable, no external dependency).

🔑 **Why?**

Unit tests should isolate your function logic. If you hit the real API, the test would be slow, flaky, and fail without internet.

3. `.return_value`

- When you call a mock, it produces a **mock object as its return value**.
- With:

```
mock_get.return_value.status_code = 200
```

→ You're configuring what the fake `response` object will look like.

Essentially:

```
fake_response = mock_get.return_value
fake_response.status_code = 200
```

So when your code does:

```
response = requests.get(...)
```

it actually gets `mock_get.return_value`.

4. `.status_code` and `.json`

- These are attributes/methods of the **real `requests.Response` object**.
- In the real world:

```
response = requests.get(...)
response.status_code   # integer (e.g., 200, 404)
response.json()        # dict parsed from JSON
```

- In the test, you **simulate** them:

```
mock_get.return_value.status_code = 200
mock_get.return_value.json.return_value = {"temp": 20, "condition":
```

```
"Sunny"}
```

So when your code under test does:

```
if response.status_code == 200:
    return response.json()
```

it behaves exactly like with a real API, but **using your fake values**.

5. `.assert_called_once_with(...)`

- This is a method of the `Mock` object (`mock_get`).
- It verifies that your code called `requests.get` **exactly once** with the expected arguments.

Here:

```
mock_get.assert_called_once_with("http://api.weatherapi.com/v1/London"
)
```

ensures your function:

- Built the correct URL string.
- Didn't call `requests.get` multiple times.

🔑 Summary of Objects

Item	Comes From	Purpose
`mocker`	`pytest-mock` fixture	Provides easy access to mocking utilities (`.patch()`).
`.patch()`	`unittest.mock.patch`	Replaces a real object with a mock.

`.return_value`	`Mock` object	Defines what calling the mock should return.
`.status_code` (fake)	Simulated `Response` attr	Emulates HTTP status (e.g., 200 OK).
`.json.return_value` (fake)	Simulated `Response.json()`	Returns fake JSON dict instead of real API response.
`.assert_called_once_with()`	`Mock` object method	Verifies the mock was called once with exact args.

✅ **Why all this matters:**

- You're **isolating your function** from external APIs.
- You test only your logic (building URL, handling response).
- You ensure reproducibility (test passes anywhere, anytime).