Mocking Classes x Mocking Functions

This is a **great comparison** because you actually demonstrated the two common "levels" of mocking:

- **Mocking functions** (e.g. `requests.get` in `main.py`)
- **Mocking classes/objects** (e.g. `APIClient` in `service.py`)

Let's break it down:

1. Mocking functions (your `main.py` / `test_main.py`)

Here you mock **direct calls** to an external dependency (the `requests.get` function).

- **Conceptually**: You're saying:

"I don't care about how the HTTP request is made, just pretend that `requests.get()` returns this response."

- **Practically**:

```
mock_get = mocker.patch("main.requests.get")
mock_get.return_value.status_code = 200
mock_get.return_value.json.return_value = {"temp": 20, "condition":
"Sunny"}
```

Every call to `requests.get()` inside `main.get_weather()` will now use the mock instead of doing a real HTTP request.

- ✅ **Pros**
  - Simple: you mock exactly what is used (`requests.get`).
  - Great when your code directly depends on a library function.
  - Fast and isolated: no network calls.
- ❌ **Cons**

- If later you change `main.py` to wrap requests in a helper (like `APIClient`), your test will break even if logic is fine → **tight coupling between test and implementation**.
- Harder to reuse the mock in multiple tests (because you patch per-function).

2. Mocking classes/objects (your `service.py` / `test_service.py`)

Here you mock **an entire collaborator** (`APIClient`) and inject it into another class (`UserService`).

- **Conceptually**: You're saying:

"I trust that `APIClient` works (or will be tested separately). I just want to test that `UserService` behaves correctly when `APIClient` returns X."

- **Practically**:

```
mock_api_client = mocker.Mock(spec=APIClient)
user_service = UserService(mock_api_client)
mock_api_client.get_user_data.return_value = {"id": 1, "name": "John Doe"}
```

Now `user_service` is isolated from real API calls but still behaves as if it had an API client.

- ✅ **Pros**
  - Looser coupling: your test does not care how `APIClient` works internally.
  - You can reuse the mock across multiple tests.
  - Encourages **dependency injection** (cleaner design).
  - Easier to test higher-level services in isolation.
- ❌ **Cons**
  - Requires more setup (you need to build and pass the mock object).
  - If you over-mock classes, you may not catch integration problems (e.g., `UserService` expects a dict but `APIClient` returns an object).

3. When to use which?

- **Mocking functions**
  - Use when your code directly calls external **functions** (e.g. `requests.get`, `os.remove`).
  - Good for lightweight modules or scripts.
- **Mocking classes/objects**
  - Use when you're testing services that depend on **other objects**.
  - Better for layered designs (e.g. controllers → services → repositories).
  - Keeps tests **focused on behavior**, not implementation.

4. Your examples side-by-side

- **main.py test** (mock function):

You directly mocked `requests.get` because that's what your function calls. Makes sense for a **small function**.

- **service.py test** (mock class):

You mocked `APIClient` because you wanted to test `UserService` **in isolation** from the actual API client. This is the more **scalable design** since it separates concerns.

✅ **Rule of thumb**:

- If you're testing a **leaf function** that calls a library → mock the function.
- If you're testing a **higher-level class/service** that collaborates with others → mock the collaborator object/class.