✅ Pytest – Crisp One-Liner Answers

1. What is the requests library in Python?
A simple HTTP client used to send API requests (GET/POST/PUT/DELETE) and handle headers, auth, and responses for API testing.

2. How do you make a GET request using requests?
By calling requests.get(url) and validating the response status code and body.

3. How can you send data using POST request?
Using requests.post() with payload passed via json or data parameters.

4. How do you handle headers in a request?
By passing headers as a dictionary using the headers parameter.

5. What if the request fails or times out?
Handle with timeout settings, exception handling, logging, retries, and assertions.

6. How do you send JSON in a POST request?
Using the json parameter which auto-serializes data and sets Content-Type.

7. How to read response content?
Use response.text, response.json(), or response.content based on type.

8. How to send query parameters?
Pass them as a dictionary using the params argument.

9. Can you upload files using requests?
Yes, using the files parameter in a POST request.

10. Real-world example use-case?
Testing login APIs by sending credentials and validating auth tokens for secured endpoints.

✅ Pytest Basics

11. How does "yield" work in pytest frameworks?
yield splits fixture setup and teardown in resource management.

12. What is Pytest?
A scalable Python testing framework for writing simple and maintainable tests.

13. Why use Pytest for API testing?
Because of simple syntax, fixtures, parameterization, plugins, and reporting.

14. How do you install Pytest?
Using pip install pytest.

15. How do you run Pytest tests?
Using the pytest command.

16. What is an assertion in Pytest?
A statement used to validate expected test results.

17. What is a fixture in Pytest?
Reusable setup/teardown logic shared across tests.

18. How do you use a fixture?
By passing it as a parameter to a test function.

19. What is scope in fixtures?
It controls fixture lifetime (function/class/module/session).

20. What is parameterization?
Running the same test with multiple input datasets.

21. How do you skip tests?
Using @pytest.mark.skip.

✅ API Testing with Requests + Pytest

22. How do you send a GET request?
Using requests.get() and validating the response.

23. How do you validate JSON response?
By parsing with response.json() and asserting values.

24. How do you send POST request?
Using requests.post() with JSON payload.

25. How do you send headers?
Pass headers dictionary in the request.

26. How do you handle authentication?
Using auth parameters or tokens in headers.

27. How do you validate response time?
By asserting response.elapsed time.

28. How do you validate schema?
Using the jsonschema library.

29. How do you handle dynamic tokens?
Store them in fixtures and reuse across tests.

30. How do you chain API calls?
Extract response data and pass it to the next request.

31. How do you test negative scenarios?
Send invalid inputs and validate error responses.

✅ Advanced Pytest Features

32. What is conftest.py?
A shared file for global fixtures.

33. What is pytest.ini?
A configuration file for pytest settings.

34. How do you mark tests?
Using @pytest.mark decorators.

35. What is test discovery?
Pytest auto-detects files starting with test_.

36. How do you generate reports?
Using pytest reporting plugins like HTML reports.

37. What is xfail?
Marks expected failing tests.

38. How do you run tests in parallel?
Using pytest-xdist.

39. What is a plugin in Pytest?
An extension that adds extra functionality.

40. How do you capture logs?
Using pytest logging options.

41. What is setup/teardown in Pytest?
Managed using fixtures with yield.

✅ API Framework & Best Practices

42. How do you structure API framework?
Using modular folders for tests, endpoints, payloads, utils, and configs.

43. What is a base URL fixture?
A centralized reusable API endpoint.

44. How do you use environment configs?
Using config files or environment variables.

45. How do you read test data from JSON?
Using Python’s json module.

46. How do you read CSV test data?
Using Python’s csv module.

47. How do you validate status codes?
Using assertions on response status.

48. How do you reuse API methods?
Through helper or client functions.

49. How do you handle retries?
Using pytest retry plugins.

50. How do you integrate Pytest with CI/CD?
By running tests in pipeline tools like Jenkins or GitHub Actions.

✅ Real-World Scenario Questions

51. How do you test CRUD operations?
Validate create, read, update, and delete workflows.

52. How do you test pagination?
Verify page size and navigation behavior.

53. How do you test rate limits?
Send rapid requests and validate throttling errors.

54. How do you validate error messages?
Assert expected error responses.

55. How do you handle flaky tests?
Stabilize environment and use retries.

56. How do you mock APIs?
Using mocking libraries like pytest-mock.

57. How do you test file uploads?
Using multipart file requests.

58. How do you test API security?
Validate authentication and permissions.

59. How do you generate test data?
Using libraries like Faker.

60. What are best practices in Pytest API testing?
Use modular design, fixtures, data-driven tests, clean assertions, and CI integration.

✅ Recommended Pytest API Automation Folder Structure

```
api-automation/
 ├── tests/
 ├── endpoints/
 ├── payloads/
 ├── testdata/
 ├── credentials/
 ├── utils/
 ├── conftest.py
 ├── pytest.ini
 ├── requirements.txt
 └── README.md
```

✅ 1. tests/ Folder (Main Test Scripts)

Contains pytest test files organized by feature (e.g., login, users, orders).

Purpose: Organizes tests clearly, improves maintainability, and supports parallel execution.

✅ 2. endpoints/ Folder

Stores reusable API request functions (GET/POST/etc.) in one central place.

Purpose: Avoids duplicate code and simplifies endpoint updates.

✅ 3. payloads/ Folder

Contains reusable payload templates defining API request structure.

Purpose: Keeps request structure clean and avoids hardcoding inside tests.

✅ 4. testdata/ Folder

Stores external test data (JSON/CSV/Excel) for data-driven testing.

Purpose: Separates test logic from data and makes updates easy.

✅ 5. credentials/ Folder 🔐

Stores environment configs and secrets (e.g., base URLs, tokens).

Purpose: Secure credential management and multi-environment support.

✅ 6. utils/ Folder

Contains helper utilities like API clients, logging, and reusable functions.

Purpose: Reduces repetition and keeps test scripts clean.

✅ 7. conftest.py

Holds shared pytest fixtures (setup, teardown, authentication).

Purpose: Provides global reusable test configuration.

✅ 8. pytest.ini

Pytest configuration file for test paths and reporting settings.

Purpose: Controls global pytest behavior.

✅ Real Workflow

When pytest runs:
tests → endpoints → payloads → testdata → credentials

All layers work together in a modular flow.

✅ Benefits of This Structure

✓ Modular and scalable
✓ Easy to maintain
✓ Secure credential handling
✓ Supports CI/CD pipelines
✓ Industry-standard design

✅ Interview Answer (Best Version)

A modular pytest framework separates tests, endpoints, payloads, test data, credentials, and utilities, with shared fixtures in conftest.py, improving maintainability, scalability, and CI/CD integration.

✅ Payload vs Test Data (Short Interview Answer)

Payload defines the API request structure, while test data provides variable input values for different test scenarios.

✅ Simple Explanation

👉 Payload = template/container
👉 Test data = values inserted into it

✅ Payload

The structured request body (usually JSON) sent to an API; it defines required fields and format and is mostly static.

✅ Test Data

Dynamic input values used to test positive, negative, and edge scenarios.

✅ How They Work Together

Payload defines structure; test data injects values to create executable API requests.

✅ Key Differences

Aspect	**Payload**	**Test Data**
Meaning	Request structure	Input values
Purpose	Defines format	Defines scenarios
Nature	Static template	Dynamic data
Location	payloads folder	testdata folder

✅ Real-World Analogy

Payload is the form layout; test data is the information entered into the form.

✅ When They Overlap

Small projects may combine them, but professional frameworks separate them for scalability and clean design.

✅ Interview Trap Answers

Can payload exist without test data?
Yes, but it cannot run meaningful tests.

Can test data exist without payload?
No, it must be injected into a payload.

✅ Final Best Interview Answer

Payload defines API request structure, while test data supplies variable inputs for scenario-based testing, and separating them improves scalability and maintainability.

✅ Requests Library – Practical Code Examples

🔹 Basic GET request with validation

```
import requests


def test_get_users():
    url = "https://reqres.in/api/users/2"
    response = requests.get(url, timeout=5)


    assert response.status_code == 200
    data = response.json()
    assert data["data"]["id"] == 2
```

� POST request with JSON payload

```
def test_create_user():
    payload = {
        "name": "John",
        "job": "Tester"
    }

    response = requests.post(
        "https://reqres.in/api/users",
        json=payload
    )

    assert response.status_code == 201
```

� Sending headers + authentication token

```
def test_auth_api():
    headers = {
        "Authorization": "Bearer my_token",
        "Content-Type": "application/json"
    }

    response = requests.get(
        "https://api.example.com/profile",
        headers=headers
    )

    assert response.status_code == 200
```

🔹 Query parameters

```
def test_query_params():
    params = {"page": 2}

    response = requests.get(
        "https://reqres.in/api/users",
        params=params
    )

    assert response.status_code == 200
```

🔹 File upload example

```
def test_file_upload():
    with open("test.txt", "rb") as f:
        response = requests.post(
            "https://httpbin.org/post",
            files={"file": f}
        )

    assert response.status_code == 200
```

✅ Pytest Fixtures – Setup & Teardown with yield

🔹 Fixture example

```
import pytest
import requests


@pytest.fixture(scope="session")
def base_url():
    return "https://reqres.in/api"


@pytest.fixture
def api_client(base_url):
    print("Setup")
    yield requests.Session()
    print("Teardown")


def test_users(api_client, base_url):
    response = api_client.get(f"{base_url}/users")
    assert response.status_code == 200
```

👉 yield runs setup **before** the test and teardown **after** the test.

✅ Parameterization Example

```
import pytest
import requests


@pytest.mark.parametrize("user_id", [1, 2, 3])
def test_multiple_users(user_id):
    response = requests.get(f"https://reqres.in/api/users/{user_id}")
    assert response.status_code == 200
```

✅ Schema Validation Example

```
from jsonschema import validate


schema = {
    "type": "object",
    "properties": {
        "id": {"type": "integer"}
    },
    "required": ["id"]
}


def test_schema():
    response = requests.get("https://reqres.in/api/users/2")
    data = response.json()["data"]

    validate(instance=data, schema=schema)
```

✓ Chaining API Calls Example

```
def test_chain_api():
    # Create user
    create = requests.post(
        "https://reqres.in/api/users",
        json={"name": "Alice"}
    )

    user_id = create.json().get("id")

    # Fetch user using ID
    get_user = requests.get(
        f"https://reqres.in/api/users/{user_id}"
    )

    assert get_user.status_code in [200, 404]
```

✓ conftest.py Example (Framework Structure)

```
project/
│
├── tests/
│   └── test_users.py
├── utils/
│   └── api_client.py
├── data/
│   └── test_data.json
└── conftest.py
```

conftest.py

```
import pytest


@pytest.fixture(scope="session")
def config():
    return {"base_url": "https://reqres.in/api"}
```

✅ Reading JSON Test Data

```
import json


def load_test_data():
    with open("data/test_data.json") as f:
        return json.load(f)
```

✅ Running Tests in Parallel

```
pytest -n 4
(Requires pip install pytest-xdist)
```

✅ HTML Report Generation

```
pytest --html=report.html

(Requires pip install pytest-html)
```

✅ Retry Failed Tests

```
pytest --reruns 2

(Requires pip install pytest-rerunfailures)
```

✅ CI/CD Example (GitHub Actions)

```
name: Pytest API Tests


on: [push]


jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - run: pip install pytest requests
      - run: pytest
```

✅ Best Practice – API Client Wrapper

Instead of calling requests everywhere:

```
class APIClient:
    def __init__(self, base_url):
        self.base_url = base_url


    def get(self, endpoint):
        return requests.get(f"{self.base_url}/{endpoint}")
```

Usage:

```
def test_users():
    client = APIClient("https://reqres.in/api")
    response = client.get("users")
    assert response.status_code == 200
```