teach me about api handling in python

Got it 👍 Let's start from the ground up with **API handling in Python**.

---

1. What is an API?

- **API (Application Programming Interface)** is a way for two software systems to talk to each other.
- In Python, we usually interact with **Web APIs** (HTTP-based APIs) using the `requests` library.

Example:

- You → send a request (`GET`, `POST`, etc.).
- API → sends a response (usually JSON).

---

2. Install the Required Library

Most common library is `requests` ↗.

```
bash

pip install requests
```

---

3. Basic GET Request

```python
import requests

url = "https://jsonplaceholder.typicode.com/posts/1"
response = requests.get(url)

# Response content
print(response.status_code)  # 200 means OK
print(response.text)         # raw text
print(response.json())       # parse JSON into Python dict
```

👉 Here:

- `GET` fetches data.
- `.json()` converts JSON into a Python dictionary.

---

4. Passing Parameters in GET

```python
import requests

url = "https://jsonplaceholder.typicode.com/posts"
params = {"userId": 1}

response = requests.get(url, params=params)

print(response.json())
```

- `params` adds query parameters → URL becomes:

```arduino
https://jsonplaceholder.typicode.com/posts?userId=1
```

---

5. Sending Data with POST

```python
import requests

url = "https://jsonplaceholder.typicode.com/posts"
data = {
    "title": "Hello API",
    "body": "Learning API handling in Python",
    "userId": 1
}

response = requests.post(url, json=data)
print(response.json())
```

- `POST` is used to **send data**.
- Use `json=data` instead of `data=data` for JSON APIs.

---

6. Headers (Authentication / Content Type)

Some APIs need authentication tokens or custom headers.

```python
import requests

url = "https://api.github.com/user"
headers = {
    "Authorization": "token YOUR_GITHUB_TOKEN"
}

response = requests.get(url, headers=headers)
print(response.json())
```

---

7. Error Handling

Always handle cases where API fails:

```
python

import requests

url = "https://jsonplaceholder.typicode.com/invalid"

try:
    response = requests.get(url)
    response.raise_for_status()   # Raises HTTPError for 4xx/5xx
    data = response.json()
    print(data)
except requests.exceptions.RequestException as e:
    print("API call failed:", e)
```

---

8. Common HTTP Methods

- **`GET`** → Retrieve data
- **`POST`** → Create new data
- **`PUT`** → Update/replace data
- **`PATCH`** → Partially update data
- **`DELETE`** → Remove data

Example:

```
python

requests.put(url, json=data)
requests.patch(url, json=partial_data)
requests.delete(url)
```

---

9. Example: Using a Real API (Public Dog API 🐶)

```
python

import requests

url = "https://dog.ceo/api/breeds/image/random"
response = requests.get(url).json()

print("Random Dog Image:", response["message"])
```

---

10. Best Practices

✅ Use **`try/except`** for failures
✅ Respect API rate limits (don't spam requests)
✅ Read API documentation (endpoints, params, auth)
✅ Use **`session = requests.Session()`** for multiple requests (faster, reuses connection)

👉 So the flow is:
Understand API docs → choose right method (GET/POST/etc.) → pass params/headers → parse JSON → handle errors.

Do you want me to next show you how to **build a small wrapper class** in Python for API handling (so you don't repeat code), or keep it at just using `requests` directly?

web scraping basics

Perfect 👌 Let's go step by step into **web scraping basics in Python**.

Requests Get/Post & Web Scraping

- **Web scraping** = extracting data from websites.
- Instead of using an **API**, we fetch the **HTML page** and then parse the data we need.
- Libraries like `requests` (to fetch HTML) + `BeautifulSoup` (to parse HTML) are most common.

---

2. Install Required Libraries

```
bash

pip install requests beautifulsoup4
```

---

3. Fetch a Webpage

```
python

import requests

url = "https://quotes.toscrape.com/"
response = requests.get(url)

print(response.status_code)   # 200 = success
print(response.text[:500])    # show first 500 chars of HTML
```

- `requests.get()` fetches the webpage as raw HTML.

---

4. Parse HTML with BeautifulSoup

```
python

from bs4 import BeautifulSoup

html = response.text
```

```
soup = BeautifulSoup(html, "html.parser")

# Extract page title
print(soup.title.text)
```

---

5. Extract Data (Finding Tags)

```
python

# Get all quotes on the page
quotes = soup.find_all("span", class_="text")

for q in quotes:
    print(q.text)
```

👉 **`.find_all(tag, attrs)`** searches by HTML tag and attributes.

---

6. Extract Author Names

```
python

authors = soup.find_all("small", class_="author")

for a in authors:
    print(a.text)
```

---

7. Extract Links

```
python

# All tags with <a>
links = soup.find_all("a")

for link in links:
    print(link.get("href"))
```

---

8. Combining Everything

```
python

for quote, author in zip(quotes, authors):
    print(f"{quote.text} — {author.text}")
```

---

9. Pagination (Next Page Navigation)

Many sites have multiple pages. You can scrape them one by one.

```python
next_btn = soup.find("li", class_="next")
if next_btn:
    next_url = next_btn.a["href"]
    print("Next page URL:", next_url)
```

---

10. Best Practices ⚠️

- Always check a site's **robots.txt** (e.g., `example.com/robots.txt`) to see what's allowed.
- Don't overload the server → add delays (`time.sleep(1)`).
- Prefer official APIs if available.
- Use **headers** (like a browser) to avoid getting blocked:

```python
headers = {"User-Agent": "Mozilla/5.0"}
requests.get(url, headers=headers)
```

✅ Summary:

1. `requests` → fetch webpage
2. `BeautifulSoup` → parse HTML
3. `.find()` / `.find_all()` → locate elements
4. Loop + extract text/attributes
5. Handle pagination / automation

Do you want me to next show you how to **scrape tabular data into Pandas DataFrame** (so you can analyze/export it), or should I show you **selenium basics** (scraping dynamic JavaScript websites)?