Data Sourcing — Topper-Style Notes

- **Goal**: Obtain data via download, query, or scrape.

1) Download

- **Examples**: Kaggle datasets, IMDb dumps, public CSVs.
- **Tips**: Prefer bulk downloadable archives; verify licenses; check update cadence.
- **Links**:
  - The Movies Dataset (Kaggle)
  - IMDb Datasets and direct download endpoints
  - Explore the Internet Movie Database
  - What does the world search for?
  - HowStat - Cricket statistics
  - Cricket Strike Rates

Best practices:

- Verify data licenses and usage terms
- Check data update frequency and freshness
- Prefer bulk downloadable archives for large datasets
- Store original files for audit trails
- Document data sources and timestamps

2) Query (APIs, Databases, SDKs)

- **REST essentials**:
  - Methods: GET (read), POST (create), PUT/PATCH (update), DELETE (remove)
  - Status: 2xx success, 4xx client errors, 5xx server errors
  - Filtering/pagination: `/items?status=active&page=2&limit=50`

```
 # HTTPie or curl
curl -H 'Authorization: Bearer $TOKEN' \
```

```
  'https://api.example.com/items?limit=100'
```

```
 # Minimal FastAPI server to practice against (run with uv)
# /// script
# dependencies = ["fastapi", "uvicorn"]
# ///
from fastapi import FastAPI, HTTPException
app = FastAPI()
items = []
@app.get('/items')
def get_items():
    return items
@app.post('/items')
def create_item(item: dict):
    item = {"id": len(items)+1, **item}
    items.append(item)
    return item
```

- **Tools**: Postman, Swagger/OpenAPI, HTTPie, JSON Schema.

3) Scrape (HTML, JS-rendered pages, PDFs)

- **Respect**: robots.txt, rate limits, terms; cache raw data.

Crawling CLI (static sites):

```
wget --recursive --level=3 --no-parent --convert-links \
  --adjust-extension --compression=auto --accept html,htm \
  --directory-prefix=./site https://example.com
```

wget2 (improved version):

```
wget2 --recursive --level=3 --no-parent --convert-links \
  --adjust-extension --compression=auto --accept html,htm \
  --directory-prefix=./site https://example.com
```

Alternative tools:

- **wpull**: WARC output, on-disk resumption, PhantomJS integration
- **httrack**: Rich filtering and link-conversion options

Robots.txt ethics:

- Default: honor; override only with permission or valid reasons
- Use `-e robots=off` (wget), `-s0` (httrack), `--no-robots` (wpull) to bypass
- Only bypass when you have explicit permission or legitimate reasons

JS-rendered pages (Playwright):

```
 # /// script
# dependencies = ["playwright"]
# ///
from playwright.sync_api import sync_playwright
with sync_playwright() as p:
    browser = p.chromium.launch(headless=True)
    page = browser.new_page()
    page.goto('https://quotes.toscrape.com/js/')
    quotes = [q.inner_text() for q in page.query_selector_all('.quote .text
    print(quotes)
```

PDF tables (Tabula):

- Find PDF links → download → extract tables → CSV.
- Control page/area for accurate extraction; save outputs for audit.

PDF→Markdown (for LLM-ready text):

```
 # PyMuPDF4LLM (high-quality MD)
PYTHONUTF8=1 uv run --with pymupdf4llm \
  python -c "import pymupdf4llm,sys;open('out.md','w').write(pymupdf4llm.to

# Markitdown (multi-format)
PYTHONUTF8=1 uvx markitdown file.pdf > out.md
```

PDF tools comparison:

Tool	Strengths	Weaknesses	Best For
PyMuPDF4LLM	Structure preservation, LLM optimization	Requires PyTorch	AI training data, semantic structure
Markitdown	Multi-format support, simple usage	Less precise layout handling	Batch processing, mixed documents

Tool	Strengths	Weaknesses	Best For
Unstructured	Wide format support, active development	Can be resource-intensive	Production pipelines, integration
GROBID	Reference extraction excellence	Narrower use case	Academic papers, citations
Tabula	Table extraction from PDFs	Limited to tabular data	Financial reports, data tables

Spreadsheet-powered scraping

- **Purpose**: Pull web data without code using Excel and Google Sheets
- **Why**: Quick, shareable, schedulable, no programming required

Excel scraping:

- Use Data → From Web to import tables from websites
- Navigate query editor to view and manage web data tables
- Apply transformations and refresh data for updates
- Example: Chennai Weather Forecast

Google Sheets scraping:

```
 =IMPORTHTML(URL, "query", index)
=IMPORTXML(URL, xpath_query)
=IMPORTFEED(feed_url, query, headers, num_items)
=IMPORTRANGE(spreadsheet_url, range_string)
=IMPORTDATA(url)
```

Best practices:

- Use stable selectors and URLs
- Implement throttling to respect rate limits
- Store snapshots for audit trails
- Handle layout changes gracefully
- Refresh data regularly for updates

Robots.txt ethics

- Default: honor; override only with permission or valid reasons; throttle requests.

Comparison Table

Method	Strengths	Weaknesses	Best for
Download	Simple, bulk	Stale, slow updates	Public dumps, archives
Query/API	Fresh, selective	Rate limits, auth	Live services, microservices
Scrape	Any visible data	Fragile, legal/ethical	Sites without APIs/dumps

Video takeaways

- Source ideas: IMDb, search trends, sports stats portals.
- Build a trusted sources list; prefer official mirrors.

TA reminders

- Cache GETs; back off on 429 with jitter; log `Retry-After`.
- Prefer cursor pagination for changing datasets.
- Scrape politely: throttle, rotate, honor robots.txt unless you have written permission.
- For PDFs, try multiple extractors; keep an audit trail of raw files and transformations.

Advanced theory and tricky exam asks

- **API auth**: API keys (simple, per-project), OAuth2 (scoped tokens, refresh), HMAC (signed requests); rotate and scope secrets.
- **Pagination**: Offset/limit (easy but unstable), cursor-based (stable); always store `next` cursors when provided.
- **Rate limiting**: Respect `Retry-After`; exponential backoff and jitter; cache GETs.

- **robots.txt semantics**: It’s advisory (not auth); legal terms still apply; sitemaps can guide discovery.
- **Anti-bot defenses**: Headless detection, CAPTCHAs, dynamic tokens; Playwright: use real user agent, wait for network idle, handle SPA route changes.
- **PDF limits**: Scanned vs text PDFs; OCR needs (ocrmypdf); table structure ambiguity; compare tools and postprocess.

Likely exam asks:

- Design a robust pagination strategy for a volatile dataset.
- Outline an ethical scraping plan for a site without an API.
- Explain why two PDF tools extract different tables and how to reconcile.

Deep dive details

OAuth2 flows (simplified):

Flow	Who uses	Pros	Cons
Client Credentials	Server-to-server	Simple	No user context
Authorization Code + PKCE	SPAs/mobile	Secure for public clients	Redirect complexity
Device Code	TVs/CLI	No browser on device	Polling delay

Cursor pagination example:

```
GET /items?limit=100&cursor=eyJpZCI6IDEyMzQ1fQ==
```

Response includes:

```
{"items": [...], "next_cursor": "eyJpZCI6IDEzMzM2fQ=="}
```

Backoff with jitter:

```
 import random, time
for i in range(5):
```

```
    try:
        resp = call()
        break
    except RateLimited:
        time.sleep(min(60, (2**i) + random.random()))
```

robots.txt semantics:

```
 User-agent: *
Disallow: /private/
Sitemap: https://example.com/sitemap.xml
```

Playwright anti-bot tactics:

- Realistic UA strings; set viewport; enable `stealth` patterns where possible.
- Wait for network idle or specific selectors; randomize delays; handle SPA routes.

Spreadsheet-powered scraping

- Purpose: pull web data without code.
- Why: quick, shareable, schedulable.
- Core: IMPORTXML/IMPORTHTML/IMPORTJSON equivalents; XPath/CSS selectors; refresh cadence.
- Pitfalls: layout changes; rate limits; brittle selectors.
- Checklist: stable selectors; throttle; store snapshots.

JS site scraping

- Purpose: extract from dynamic sites.
- Why: static fetch misses rendered content.
- Core: headless browser, wait for selectors/network idle, DOM parsing.

```
const res = await page.$$eval('.item', els => els.map(e => e.textContent.t
```

- Pitfalls: auth, anti-bot, infinite scroll.
- Checklist: login flows; paging strategy; backoff.

Market research portals

- Purpose: collect competitive intel.
- Why: data locked behind portals.
- Core: authenticated sessions, CSV exports, API endpoints if available.
- Pitfalls: TOS violations; fragile scrapers.
- Checklist: prefer official exports; legal review; modest crawl rates.

Weather and public APIs

- Purpose: structured data via HTTP.
- Why: reliable and documented sources.
- Core: query params, JSON parsing, caching by key/time window.

BBC Weather API example:

```
 import requests
import json

def get_location_id(city_name):
    """Get BBC Weather location ID for a city."""
    url = "https://locator-service.api.bbci.co.uk/locations"
    params = {
        "api_key": "AGbFAKx58hyjQScCXIYrxuEwJh2W2cmv",
        "stack": "aws",
        "locale": "en",
        "filter": "international",
        "place-types": "settlement,airport,district",
        "order": "importance",
        "s": city_name,
        "a": "true",
        "format": "json"
    }
    response = requests.get(url, params=params)
    return response.json()

def get_weather_data(location_id):
    """Get weather data for a location ID."""
    url = f"https://weather-broker-cdn.api.bbci.co.uk/en/forecast/aggregate
    response = requests.get(url)
    return response.json()
```

General API example:

```
curl '.../weather?city=Paris&units=metric' | jq '.'
```

- Pitfalls: rate limits; time zone confusion.
- Checklist: cache GETs; standardize timezones; retry on 429 with jitter.

Wikipedia data

- Purpose: encyclopedic, structured content.
- Why: open API with good docs.
- Core: page queries, category members, pagination via continue tokens.

Python wikipedia library example:

```
 import wikipedia as wk

# Search for pages
results = wk.search("Python programming", results=5)

# Get page summary
summary = wk.summary("Python (programming language)", sentences=3)

# Get full page content
page = wk.page("Python (programming language)")
print(f"Title: {page.title}")
print(f"URL: {page.url}")
print(f"Content length: {len(page.content)}")
print(f"References: {len(page.references)}")
print(f"Images: {len(page.images)}")

# Extract tables using pandas
import pandas as pd
tables = pd.read_html(page.html())
```

API endpoints:

- Page content: `https://en.wikipedia.org/api/rest_v1/page/summary/{title}`
- Search: `https://en.wikipedia.org/api/rest_v1/page/search/{query}`
- Categories: `https://en.wikipedia.org/api/rest_v1/page/categories/{title}`

- Pitfalls: ambiguous pages; vandalism; rate limits.
- Checklist: include polite UA; verify content; handle continues.

Geocoding (Nominatim)

- Purpose: addresses ↔ coordinates.
- Why: maps and spatial analysis.
- Core: address strings, bounding boxes, per-IP limits, contact header.

Python geopy example:

```
 from geopy.geocoders import Nominatim

# Create geocoder with user agent
geolocator = Nominatim(user_agent="myGeocoder")

# Geocode an address
location = geolocator.geocode("IIT Madras, Chennai, India")
print(f"Latitude: {location.latitude}")
print(f"Longitude: {location.longitude}")
print(f"Address: {location.address}")
print(f"Type: {location.raw.get('type')}")

# Reverse geocoding
location = geolocator.reverse("13.0827, 80.2707")
print(f"Address: {location.address}")
```

API endpoints:

- Search: `https://nominatim.openstreetmap.org/search?q={query}&format=json`
- Reverse: `https://nominatim.openstreetmap.org/reverse?lat={lat}&lon={lon}&format=json`

Best practices:

- Always set a user agent (email or name)
- Add delays between requests (1 second minimum)
- Cache results to avoid repeated requests

- Handle rate limiting gracefully
- Pitfalls: rate bans; inconsistent formats.
- Checklist: add email; sleep between requests; cache results.

Headless automation

- Purpose: robust scripted browsing.
- Why: handle JS-heavy flows.
- Core: auto-wait locators, stable selectors, random delays.

Playwright example:

```
 from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    browser = p.chromium.launch(headless=True)
    page = browser.new_page()

    # Navigate to page
    page.goto('https://quotes.toscrape.com/js/')

    # Wait for content to load
    page.wait_for_selector('.quote')

    # Extract data
    quotes = page.query_selector_all('.quote .text')
    data = [quote.inner_text() for quote in quotes]

    browser.close()
```

JavaScript scraping example:

```
 // Extract IMDb Top 250 movies
const movies = Array.from(document.querySelectorAll('.ipc-metadata-list-sum
  .map(item => {
    const title = item.querySelector('.ipc-title__text')?.textContent;
    const year = item.querySelector('.cli-title-metadata span')?.textConten
    const rating = item.querySelector('.ipc-rating-star')?.textContent;
    return { title, year, rating };
  })
  .filter(movie => movie.title && movie.year && movie.rating);
```

```
// Copy to clipboard
copy(JSON.stringify(movies, null, 2));
```

Anti-bot tactics:

- Use realistic user agent strings
- Set viewport and enable stealth patterns
- Wait for network idle or specific selectors
- Randomize delays between actions
- Handle SPA route changes
- Pitfalls: flaky timing; brittle selectors.
- Checklist: prefer role/aria labels; wait for specific states; record raw HTML.

HTML to Markdown conversion

- Purpose: Convert web content to clean, structured text
- Why: Content analysis, data mining, offline reading, content migration

Tools comparison:

Tool	Speed	Format Quality	Preservation	Best For
defuddle-cli	Slow	High	Good structure and links	Content migration, publishing
pandoc	Slow	Very High	Almost everything	Academic papers, documentation
lynx	Fast	Low	Basic structure only	Quick extraction, large batches
w3m	Very Slow	Medium-Low	Basic structure with better tables	Improved readability over lynx

Batch processing example:

```
 # Using pandoc for high-quality conversion
find . -name "*.html" -exec pandoc -f html -t markdown_strict -o {}.md {} \

# Using lynx for fast text extraction
find . -type f -name '*.html' -exec sh -c 'for f; do lynx -dump -nolist "$f
```

Combined crawling and conversion:

```
 # Crawl4AI for AI training data
uv venv
source .venv/bin/activate
uv pip install crawl4ai
crawl4ai-setup

# markdown-crawler for bulk processing
uv pip install markdown-crawler
markdown-crawler -t 5 -d 3 -b ./markdown https://example.com/
```

LLM Video Screen-Scraping

- Purpose: Extract structured data from screen recordings using LLMs
- Why: Bypass authentication, anti-scraping measures, work with any visible content

Key benefits:

- No setup cost or authentication handling
- Works with any visible screen content
- Full control over data exposure
- Extremely cost-effective (< $0.001 per short video)
- Bypasses anti-scraping measures
- Handles varying formats and layouts

Workflow:

1. **Record the Screen**: Use QuickTime (Mac), Windows Game Bar, Screen2Gif, or OBS Studio
2. **Process with Gemini**: Upload to Google AI Studio, select Gemini 1.5 Flash
3. **Prompt for structured output**: Request JSON/CSV format

Example prompt:

```
 Turn this video into a JSON array where each item has:
{
  "date": "yyyy-mm-dd",
  "amount": float
}
```

Cost calculation:

- Gemini 1.5 Flash: $0.075 per million tokens
- Cost per frame: ~250 tokens
- 24 hours of video at 1 frame/second: ~$1.62

Scheduled Scraping with GitHub Actions

- Purpose: Automate data collection from websites on a schedule
- Why: Regular data updates, no manual intervention, version control

Key concepts:

- **Scheduling**: Use cron syntax to run scrapers at specific times
- **Dependencies**: Install required packages like `httpx`, `lxml`
- **Data Storage**: Save scraped data to files and commit back to repository
- **Error Handling**: Implement robust error handling for network issues
- **Rate Limiting**: Respect website terms of service and implement delays

Example scraper (IMDb Top 250):

```
 import json
import httpx
from datetime import datetime, UTC
from lxml import html

def scrape_imdb():
    """Scrape IMDb Top 250 movies using httpx and lxml."""
    headers = {"User-Agent": "Mozilla/5.0 (compatible; IMDbBot/1.0)"}
    response = httpx.get("https://www.imdb.com/chart/top/", headers=headers
    response.raise_for_status()

    tree = html.fromstring(response.text)
    movies = []
```

```
    for item in tree.cssselect(".ipc-metadata-list-summary-item"):
        title = item.cssselect(".ipc-title__text")[0].text_content()
        year = item.cssselect(".cli-title-metadata span")[0].text_content()
        rating = item.cssselect(".ipc-rating-star")[0].text_content()
        movies.append({"title": title, "year": year, "rating": rating})

    return movies

# Save with timestamp
now = datetime.now(UTC)
with open(f'imdb-top250-{now.strftime("%Y-%m-%d")}.json', "a") as f:
    f.write(json.dumps({"timestamp": now.isoformat(), "movies": scrape_imdb
```

GitHub Actions workflow:

```
 name: Scrape IMDb Top 250

on:
  schedule:
    - cron: "0 0 * * *"  # Daily at midnight UTC
  workflow_dispatch:  # Allow manual triggers

jobs:
  scrape-imdb:
    runs-on: ubuntu-latest
    permissions:
      contents: write

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Install uv
        uses: astral-sh/setup-uv@v5

      - name: Run scraper
        run: uv run --with httpx,lxml,cssselect python scrape.py

      - name: Commit and push changes
        run: |
          git config --local user.email "github-actions[bot]@users.noreply.
          git config --local user.name "github-actions[bot]"
          git add *.json
          git commit -m "Update IMDb Top 250 data [skip ci]" || exit 0
          git push
```

PDFs → Markdown

- Purpose: reusable, searchable text.
- Why: turn documents into clean text for downstream tasks.
- Core: choose extractors (structure preservation vs speed), OCR scanned docs first, post-process.

```
 ocrmypdf --force-ocr in.pdf ocr.pdf
uvx markitdown ocr.pdf > out.md
```

- Pitfalls: tables merged incorrectly; headers/footers noise.
- Checklist: keep originals; note tool/version; spot-check samples.