

Lecture 02: Data Wrangling with Lists + Dicts

Reminder: pandas and numpy arrive in Q3

Goals

- Turn raw CSV/JSON rows into clean records
- Group by artist/genre and compute summary stats
- Avoid repeated scans with simple indexes

Today's data (imaginary Spotify)

- `tracks.csv` : track_id, track_name, artist_id, genre, duration_ms
- `plays.csv` : play_id, track_id, user_id, timestamp, play_count
- `artists.json` : artist_id -> {name, country}

Demo files live in `lectures/data/` (or `data/` if you open this notebook inside `lectures/`).

We will build a read -> clean -> group -> report pipeline.

Segment 1: Clean rows for analysis

Goal: normalize IDs and parse numeric fields early.

- Strip and casifold IDs so joins are reliable
- Parse duration to int and fill missing genre
- Build a clean list of track dicts from CSV

Code example preview

- Define `parse_int` and `clean_track_row` helpers
- Load `tracks.csv` with `DictReader` and clean each row
- Inspect the first few cleaned records for sanity

```
In [ ]: from pathlib import Path
import csv

def parse_int(value, default=0):
    try:
        return int(value)
    except (TypeError, ValueError):
        return default

def clean_track_row(row):
    return {
        'track_id': row['track_id'].strip().casefold(),
        'track_name': row['track_name'].strip(),
        'artist_id': row['artist_id'].strip().casefold(),
        'genre': (row['genre'] or 'unknown').strip().casefold(),
        'duration_ms': parse_int(row['duration_ms']),
    }
```

```
In [ ]: data_dir = Path('lectures/data')
if not data_dir.exists():
    data_dir = Path('data')

with (data_dir / 'tracks.csv').open(newline='', encoding='utf-8') as f:
    reader = csv.DictReader(f)
    tracks = [clean_track_row(r) for r in reader]

tracks[:3]
```

Why clean early?

- Makes grouping keys stable (case + whitespace)
- Protects against bad numeric fields
- Keeps later stages simple

Segment 2: Filter + map with list comprehensions

Goal: build smaller, cleaner working sets.

- Filter to pop tracks with valid durations
- Use list comps to keep only needed fields
- Create lightweight pairs for later aggregation

Code example preview

- Filter tracks to a focused working set
- Project to `(track_id, duration_ms)` pairs
- Compare the reduced lists to the full data

In []: *# Keep only pop tracks with known duration*
pop_tracks = [t for t in tracks if t['genre'] == 'pop' and t['duration_r'][t['track_name']] for t in pop_tracks]

```
In [ ]: # Build a simple list of (track_id, duration_ms) pairs
track_durations = [(t['track_id'], t['duration_ms']) for t in tracks]
track_durations
```

Segment 3: Group plays by artist

Goal: collect stats in one pass.

- Clean play rows and standardize IDs
- Join plays to artists with a lookup dict
- Aggregate plays and unique tracks per artist

Code example preview

- Load and clean `plays.csv` records
- Build `track_id -> artist_id` for fast joins
- Use `defaultdict` to collect stats in one pass

```
In [ ]: from collections import defaultdict
        import json

def clean_play_row(row):
    return {
        'play_id': row['play_id'].strip().casifold(),
        'track_id': row['track_id'].strip().casifold(),
        'user_id': row['user_id'].strip().casifold(),
        'timestamp': row['timestamp'].strip(),
        'play_count': parse_int(row['play_count']),
    }

with (data_dir / 'plays.csv').open(newline='', encoding='utf-8') as f:
    reader = csv.DictReader(f)
    plays = [clean_play_row(r) for r in reader]

track_to_artist = {t['track_id']: t['artist_id'] for t in tracks}

artist_stats = defaultdict(lambda: {'plays': 0, 'tracks': set()})
for p in plays:
    artist_id = track_to_artist.get(p['track_id'])
    if not artist_id:
        continue
    artist_stats[artist_id]['plays'] += p['play_count']
    artist_stats[artist_id]['tracks'].add(p['track_id'])

artist_stats
```

```
In [ ]: with (data_dir / 'artists.json').open(encoding='utf-8') as f:  
    artists = json.load(f)  
  
list(artists.items())[:2]
```

Segment 1-3 summary

- Clean fields once, then reuse clean data everywhere
- List comps create focused working sets
- `defaultdict` helps aggregate in one pass

In-class exercise 1 (commit)

Build `artist_id -> stats` where stats includes `plays` and `avg_duration_ms`.

- Use a `track_id -> track` index
- Ignore tracks with missing durations
- Commit your notebook with a short note in a markdown cell

Break (3 minutes)

Segment 4: Group by genre with defaults

Goal: handle missing genres safely.

- Count tracks per genre with Counter
- Map plays to genres using a track index
- Default missing genres to keep totals consistent

Code example preview

- Compute simple genre counts from tracks
- Join plays to genres via `track_id`
- Accumulate play counts per genre safely

```
In [ ]: from collections import Counter  
  
genre_counts = Counter(t['genre'] for t in tracks)  
genre_counts
```

```
In [ ]: # Plays per genre
plays_by_genre = Counter()
track_to_genre = {t['track_id']: t['genre'] for t in tracks}
for p in plays:
    genre = track_to_genre.get(p['track_id'], 'unknown')
    plays_by_genre[genre] += p['play_count']

plays_by_genre
```

Segment 5: Compute summary stats

Goal: total plays and average durations.

- Accumulate total plays per artist
- Track duration sums and counts for averages
- Compute summary stats with a small helper

Code example preview

- Sum durations and counts from track data
- Add play totals from play records
- Return `(plays, avg_duration)` per artist

```
In [ ]: # Total plays and average track duration by artist
artist_totals = defaultdict(lambda: {'plays': 0, 'dur_sum': 0, 'dur_count': 0})
for t in tracks:
    artist_totals[t['artist_id']]['dur_sum'] += t['duration_ms']
    artist_totals[t['artist_id']]['dur_count'] += 1

for p in plays:
    artist_id = track_to_artist.get(p['track_id'])
    if artist_id:
        artist_totals[artist_id]['plays'] += p['play_count']

def avg_duration(stats):
    if stats['dur_count'] == 0:
        return 0
    return stats['dur_sum'] // stats['dur_count']

{aid: (s['plays'], avg_duration(s)) for aid, s in artist_totals.items()}
```

Segment 6: Avoid repeated scans

Goal: build indexes to reuse lookups.

- Build a `track_id` index once
- Enrich plays with artist and genre fields
- Avoid repeated scans across the dataset

Code example preview

- Create a dictionary index for fast lookup
- Enrich each play with joined track data
- Skip missing tracks instead of crashing

```
In [ ]: # Index for fast lookup by track_id
track_index = {t['track_id']: t for t in tracks}

def enrich_play(play):
    track = track_index.get(play['track_id'])
    if not track:
        return None
    return {
        'track_id': play['track_id'],
        'artist_id': track['artist_id'],
        'genre': track['genre'],
        'play_count': play['play_count'],
    }

[enrich_play(p) for p in plays]
```

Segment 4-6 summary

- `Counter` and `defaultdict` power fast aggregates
- Use indexes to avoid $O(n^2)$ scans
- Defaults keep pipelines resilient to missing data

In-class exercise 2 (commit)

Compute the **top 5 artists by total plays** and print a short report.

- Use an index for `track_id -> artist_id`
- Break ties alphabetically
- Commit your notebook output

Wrap-up

- Clean early, group once, index for speed
- Lists + dicts are enough for real data tasks
- Next time: contiguous tables and pipelines

