Lecture 01: Python Standard Library for Data Work

Course: EECS 291

Scenario: We are working with a small, imaginary Spotify-style dataset: `tracks.csv`, `plays.csv`, `artists.json`.

Goals

- Load CSV/JSON with stdlib tools
- Normalize messy strings and ids
- Aggregate with `collections` and basic stats
- Build a read -> clean -> aggregate -> report pipeline

Outline (Applied)

Python Skills: Comprehensions + Exception Handling
Half 1: Files + Parsing + Basic Aggregation
Half 2: Grouping + Summaries + Time fields

We will commit twice: once per half.

Python Skill: Comprehensions

- Compact way to build lists/dicts from existing data.
- We will use them to filter rows and transform fields quickly.
- If this is new, focus on the pattern: [output for item in items if condition].

```
In [ ]: rows = [
            {'track_id': 't1', 'genre': 'pop'},
            {'track_id': 't2', 'genre': 'indie'},
            {'track_id': 't3', 'genre': 'pop'},
        ]

        pop_ids = [r['track_id'] for r in rows if r['genre'] == 'pop']
        genre_map = {r['track_id']: r['genre'] for r in rows}
        pop_ids, genre_map
```

Python Skill: Exception Handling

- Protects your pipeline from messy or missing values.
- We will use try/except to convert strings to integers safely.
- If conversion fails, we keep a default and continue.

In [ ]:
```python
def parse_int(value, default=0):
    try:
        return int(value)
    except (TypeError, ValueError):
        return default

[parse_int('42'), parse_int('bad'), parse_int(None, default=-1)]
```

Half 1: Files + Parsing + Basic Aggregation

Goal: Ingest raw data and compute a first metric in pure stdlib.

Segment 1: Locate data files with `pathlib`

Goal: Find CSV/JSON inputs and build paths safely.

- `pathlib` is the standard library for filesystem paths as objects.
- Use `/` to join path parts instead of manual string concatenation.
- Check inputs with `.exists()` or `.is_file()` before reading.
- Use `.glob('*.csv')` to discover data files in a folder.

In [ ]:

```python
from pathlib import Path

data_dir = Path("data")

# List files in the dataset folder
for path in sorted(data_dir.glob("*")):
    print(path.name)

tracks_csv = data_dir / "tracks.csv"
plays_csv = data_dir / "plays.csv"
artists_json = data_dir / "artists.json"

print(tracks_csv.exists(), plays_csv.exists(), artists_json.exists())
```

Segment 2: Reading data from files

Goal: Open file using `pathlib`

- Use `with` to open files
- Python automatically closes files when using the `with` keyword (context manager)

In [ ]:
```
with (tracks_csv).open(newline='', encoding='utf-8') as file:
    # here we have access to the `file` object

    # readline reads 1 line of the file and returns it as a string.
    print(file.readline())

    # you can also loop over the lines with a for loop
    for line in file:
        print(line)

    # notice that we did not read the header twice. We can only read ea

# file is closed once we exit the `with` block
```

Segment 3: Parse CSV into rows

Goal: Read CSV safely and keep memory use predictable.

- `csv` parses comma-separated files into rows and dictionaries.
    - `DictReader` maps headers to values (row-by-row).
- `io` provides file-like objects for text and bytes.
    - `StringIO` lets you treat a string like a file (useful for demos/tests).
- `itertools` offers fast iterator tools for slicing and grouping streams.
    - `islice` takes the first N rows without loading everything.

```
In [ ]: import csv
        from io import StringIO
        from itertools import islice

        raw = StringIO("""track_id,track_name,artist_id,genre,duration_ms
        t01,Ocean Breeze,a10,Lo-Fi,183000
        t02,Midnight City,a20,Synthpop,243000
        """)

        reader = csv.DictReader(raw)
        preview = list(islice(reader, 2))
        print(preview)
```

Segment 4: Clean fields + first aggregation

Goal: Normalize messy strings and convert numbers early.

- Convert to integer using exceptions instead of manual parsing
- String conversion issues
    - Empty strings
    - Unwanted whitespace (leading and/or trailing)
    - Letter case

In [ ]:
```python
def parse_int(value, default=0):
    try:
        return int(value)
    except (TypeError, ValueError):
        return default


def normalize_genre(text):
    return (text or "").strip().casefold()


clean_tracks = []
for row in preview:
    clean_tracks.append(
        {
            "track_id": row["track_id"].strip(),
            "track_name": row["track_name"].strip(),
            "artist_id": row["artist_id"].strip(),
            "genre": normalize_genre(row["genre"]),
            "duration_ms": parse_int(row["duration_ms"]),
        }
    )

print(clean_tracks[0])
```

Segment 4 Follow-up: Counter + defaultdict

- `from collections import Counter, defaultdict`
- `Counter` counts repeated keys quickly (like genres).
- `defaultdict` automatically creates a default value when a missing key is accessed
    - This avoids explicit checks for key existence and simplifies logic by handling default insertion for you

In [ ]:
```python
from collections import Counter, defaultdict

genre_counts = Counter(normalize_genre(r['genre']) for r in clean_tracks

tracks_by_artist = defaultdict(list)
for r in clean_tracks:
    tracks_by_artist[r['artist_id'].strip()].append(r['track_name'].str

genre_counts, dict(list(tracks_by_artist.items())[:2])
```

Half 1 Summary

- Use `pathlib` to find inputs
- `csv.DictReader` + `itertools.islice` to inspect rows
- Clean and convert types before aggregating

In-class Exercise 1 (Commit Required)

Task: Load `tracks.csv`, normalize `genre`, and compute top-3 genres.

Commit: `lecture01-half1`

- Include your code or notebook output
- Note one cleaning rule you used

In [ ]:

```
# Exercise 1
#  - Load 'tracks.csv' from 'data' directory (data columns: Track ID, A
#  - Normalize genre (use 'unknown' if none provided)
#  - Compute top-3 genres
```

Break (3 minutes)

Stretch, refill, and be back at :__

Half 2: Grouping + Summaries + Time Fields

Goal: Build richer summaries and prepare a report.

Segment 5: Grouping with `collections`

Goal: Count and group rows efficiently; compare list-of-dicts vs a dict index.

In [ ]:

```python
from collections import Counter, defaultdict

plays = [
    {"play_id": "p1", "track_id": "t01", "play_count": 3},
    {"play_id": "p2", "track_id": "t02", "play_count": 1},
    {"play_id": "p3", "track_id": "t01", "play_count": 2},
]

track_index = {track["track_id"]: track for track in clean_tracks}

plays_by_artist = defaultdict(int)
for play in plays:
    # gets the track information if track_id exists, otherwise default
    track_info = track_index.get(play["track_id"], {})

    artist_id = track_info.get("artist_id", "unknown")
    plays_by_artist[artist_id] += play["play_count"]

print(plays_by_artist)

track_counts = Counter(play["track_id"] for play in plays)
print(track_counts.most_common(1))
```

Segment 6: Summaries with `statistics`

Goal: Compute quick descriptive stats for a column.

- `statistics` provides simple descriptive stats for small/medium datasets.
- `mean` returns the arithmetic average.
    - Calculation: sum(values) / count(values).
    - Syntax: `statistics.mean(values)` -> float.
- `median` returns the middle value after sorting.
    - If there are two middle values, it returns their average.
    - Syntax: `statistics.median(values)` -> float.
- `pstdev` returns population standard deviation.
    - Measures typical spread around the mean for the full population.
    - Syntax: `statistics.pstdev(values)` -> float.

In [ ]:

```python
from statistics import mean, median, pstdev

durations = [track["duration_ms"] for track in clean_tracks]
print(mean(durations), median(durations), round(pstdev(durations), 2))
```

Segment 7: Time fields with `datetime`

Goal: Parse play timestamps and derive simple features.

- `datetime` library parses, stores, and formats dates/times from strings.
- A `datetime` object represents a specific moment in time.
- `datetime.fromisoformat(text)` parses ISO strings like `2026-01-12T11:22:00`.
- Common members: `year`, `month`, `day`, `hour`, `minute`, `second`, `date()`, `weekday()`.

```
In [ ]: from datetime import datetime

        raw_ts = "2024-10-05T12:34:56"
        played_at = datetime.fromisoformat(raw_ts)
        print(played_at.date(), played_at.hour)
```

Segment 8: Export results with `json`

JSON: **J**ava**S**cript **O**bject **N**otation

Goal: Save summaries for reuse in later steps.

- JSON is a lightweight *text* format for structured data (lists + dicts).
- It is useful for sharing results between scripts and storing summaries.
- The `json` library converts Python objects to/from JSON.
- Common usage: `json.load(f)` / `json.dump(obj, f, indent=2)` for files.
- Common usage: `json.loads(str)` / `json.dumps(obj, f, indent=2)` for strings.
- Common object types: dict, list, str, int, float, bool, None.

In [ ]:

```
import json
from datetime import datetime
from io import StringIO

report = {
    "top_artists": [{"artist_id": "a10", "plays": 5}],
    "generated_at": datetime.now().isoformat(timespec="seconds"),
}
print(json.dumps(report, indent=2))

raw_artists = StringIO('{"a10": {"name": "Luna Waves", "country": "US"}
artists = json.load(raw_artists)
print(artists["a10"]["name"])
```

Half 2 Summary

- `collections` handles counting and grouping
- `statistics` gives quick data summaries
- `datetime` and `json` help with features and outputs

In-class Exercise 2 (Commit Required)

Task: Compute top-5 artists by total plays and average duration.

Commit: `lecture01-half2`

- Include your report output
- Note one tradeoff between list scans vs dict indexing

Wrap + Cleanup (5 minutes)

- Push your commits
- Note any blockers
- Preview: Lecture 02 = stdlib wrangling patterns