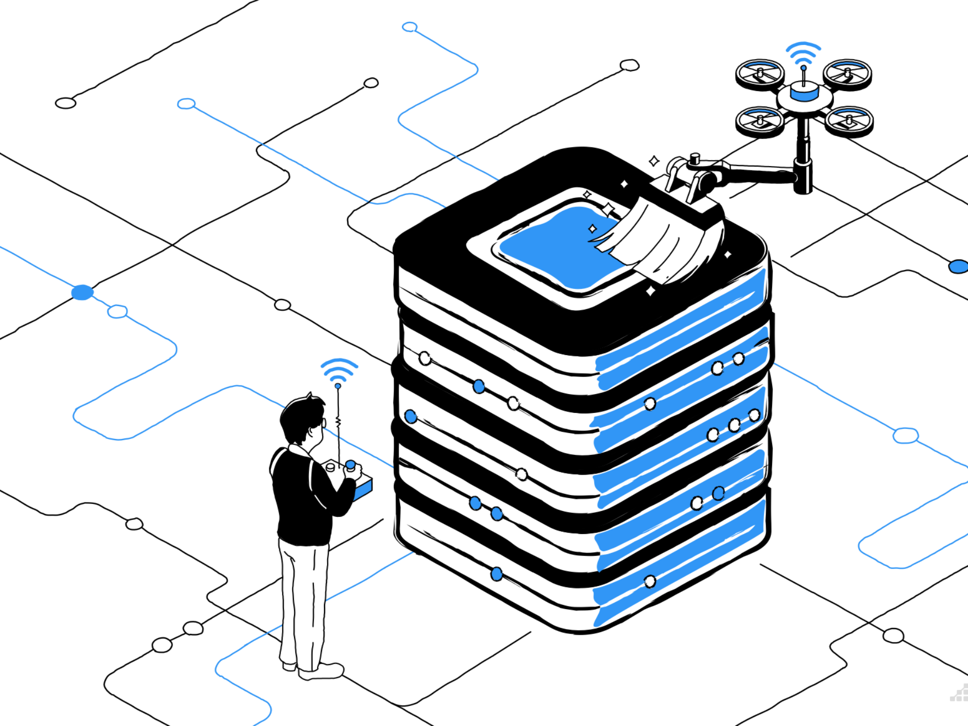
Data Science in Paleontology

Matilde Pós-de-Mina Pato, ISEL - IPL
Personal page ✉ matilde.pato[at]isel.pt

Welcome!

*The legendary engineer W. Edwards Deming put it “Without data, you’re just another person with an opinion.” Without insight into **data**, people make decisions based on **instinct**, **speculation**, or prevalent **theory**.*



Context

Motivation: Difficulty in integrating paleontological data from **different** sources (PBDB, museum collections, literature).

Premise: Unify and clean fossil occurrence data using *open-source* tools.

Target audience: Paleontology students and researchers working with large datasets.

Why Python for Paleobiological Data?

Available options:

- GUI tools (Tableau, Power BI, OpenRefine)
- Programming (Python, R)
- Enterprise solutions (Alteryx, IBM)

Our choice: Python

- Reproducible research
- Free and open-source
- Flexible for any workflow
- Transparent processing

Python

Python Ecosystem:

- `pandas`: Data manipulation and cleaning
- `requests`: API access to PBDB
- `numpy`: Numerical operations
- `matplotlib/seaborn`: Visualization

Additional Tools:

- `geopandas`: Spatial data handling
- `scipy`: Statistical analysis
- `jupyter`: Interactive notebooks

The Paleobiology Database (PBDB)

What: Global repository of **fossil occurrence data**

- >1.4 million fossil occurrences
- >400,000 taxonomic names
- Community-driven, peer-reviewed contributions

Access: `https://paleobiodb.org`

- Web interface (Navigator)
- API for programmatic access
- Download formats: CSV, JSON, TSV

Common Data Quality Issues in PBDB

Taxonomic Issues:

- Synonyms and homonyms
- Outdated classifications
- Misspellings

Temporal Issues:

- Mixed time scales
- Uncertain age ranges
- Missing epochs/stages

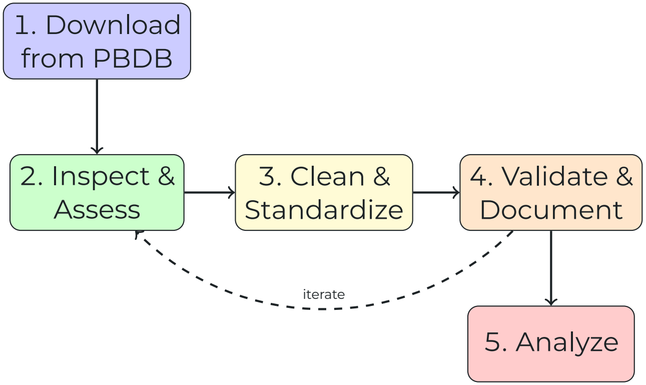
Geographic Issues:

- Missing coordinates
- Coordinate precision
- Incorrect datum/projection

Completeness Issues:

- Missing identifiers
- Incomplete references
- Environmental data gaps

Data Preparation Workflow



Today's Practical Exercise

1. Download fossil occurrence data for a specific class
2. Inspect data quality using summary statistics
3. Identify and handle missing/problematic records
4. Standardize taxonomic names and temporal assignments
5. Visualize cleaned data (geographic and temporal distribution)

Example: Trilobite occurrences from the Paleozoic

Best Practices: Documentation

Always document:

- Data source (PBDB) and download timestamp
- Query parameters used (taxon, interval, geographic bounds)
- Cleaning decisions and rationale
- Number of records removed at each step
- Software versions (Python, pandas, etc.)

PBDB API: Key Parameters

Common query parameters:

- `base_name`: Taxonomic name (e.g., 'Trilobita')
- `interval`: Time interval (e.g., 'Cambrian', 'Jurassic,Cretaceous')
- `min_ma, max_ma`: Age range in millions of years
- `lngmin, lngmax, latmin, latmax`: Geographic bounding box
- `show`: Additional fields to include

Useful 'show' options: coords, phylo, ident, loc, strat, time, ref

Step 1: Downloading Data from PBDB

Using Python with requests and pandas:

```python
import pandas as pd
import requests

# PBDB API endpoint
base_url = "https://paleobiodb.org/data1.2/occs/list.csv"

# Query parameters
params = {
    'base_name': 'Trilobita',
    'interval': 'Cambrian,Permian',
    'show': 'coords,phylo,ident'
}

# Download data
response = requests.get(base_url, params=params)
trilobites = pd.read_csv(pd.io.common.StringIO(response.text))

print(f"Downloaded {len(trilobites)} occurrences")
```

Documenting Your Download

```python
from datetime import datetime

# Record download metadata
metadata = {
    'source': 'Paleobiology Database',
    'url': base_url,
    'download_datetime': datetime.now().isoformat(),
    'query_params': params,
    'n_records': len(trilobites)
}

# Save metadata with your data
import json
with open('download_metadata.json', 'w') as f:
    json.dump(metadata, f, indent=2)

# Or add to your dataframe
print(f"Data downloaded: {metadata['download_datetime']}")
print(f"Query: {metadata['query_params']}")
```

Why timestamp matters:

- PBDB is continuously updated (new data, corrections, taxonomic revisions)
- Results from the same query can differ over time
- Essential for **reproducibility** and **citation**

Step 2: Inspect the Data

Check data completeness and structure:

```python
# Basic information
print(trilobites.shape)  # Rows and columns
print(trilobites.columns)  # Column names

# Missing values per column
print(trilobites.isnull().sum())

# Summary statistics for key variables
print(trilobites[['max_ma', 'min_ma', 'lng', 'lat']].describe())

# Unique taxonomic levels
print(f"Unique genera: {trilobites['genus'].nunique()}")
print(f"Unique families: {trilobites['family'].nunique()}")
```

Step 3: Assess Data Quality

Identify problematic records:

```python
# Records without coordinates
no_coords = trilobites[trilobites['lng'].isnull() |
                       trilobites['lat'].isnull()]
print(f"Missing coordinates: {len(no_coords)} records")

# Records with suspicious ages (max_ma < min_ma)
bad_ages = trilobites[trilobites['max_ma'] < trilobites['min_ma']]
print(f"Invalid age ranges: {len(bad_ages)} records")

# Records without genus identification
no_genus = trilobites[trilobites['genus'].isnull()]
print(f"Missing genus: {len(no_genus)} records")

# Age uncertainty (large ranges)
trilobites['age_range'] = trilobites['max_ma'] - trilobites['min_ma']
print(trilobites['age_range'].describe())
```

Step 4: Data Cleaning Strategies

Decision points: Remove or impute?

```python
# Strategy 1: Remove records without coordinates
# (if spatial analysis is critical)
clean_data = trilobites.dropna(subset=['lng', 'lat'])

# Strategy 2: Keep all, flag problematic records
trilobites['has_coords'] = ~(trilobites['lng'].isnull())
trilobites['valid_age'] = trilobites['max_ma'] >= trilobites['min_ma']

# Strategy 3: Calculate midpoint ages
trilobites['mid_ma'] = (trilobites['max_ma'] +
                        trilobites['min_ma']) / 2

# Document your decisions!
print(f"Original: {len(trilobites)} records")
print(f"After cleaning: {len(clean_data)} records")
print(f"Removed: {len(trilobites) - len(clean_data)} records")
```

Step 5: Handle Taxonomic Issues

```python
# Check for taxonomic inconsistencies
print(trilobites['accepted_name'].value_counts().head(10))

# Identify records with different accepted vs. identified names
changed = trilobites[trilobites['accepted_name'] !=
                     trilobites['identified_name']]
print(f"Taxonomic revisions: {len(changed)} records")

# Standardize to accepted names
trilobites['taxon_clean'] = trilobites['accepted_name'].fillna(
                            trilobites['identified_name'])

# Remove tentative identifications (e.g., "cf.", "?")
trilobites['uncertain'] = trilobites['taxon_clean'].str.contains(
                          'cf\.|aff\.|\\?', na=False)
print(f"Uncertain identifications: {trilobites['uncertain'].sum()}")
```

Step 6: Visualize Cleaned Data

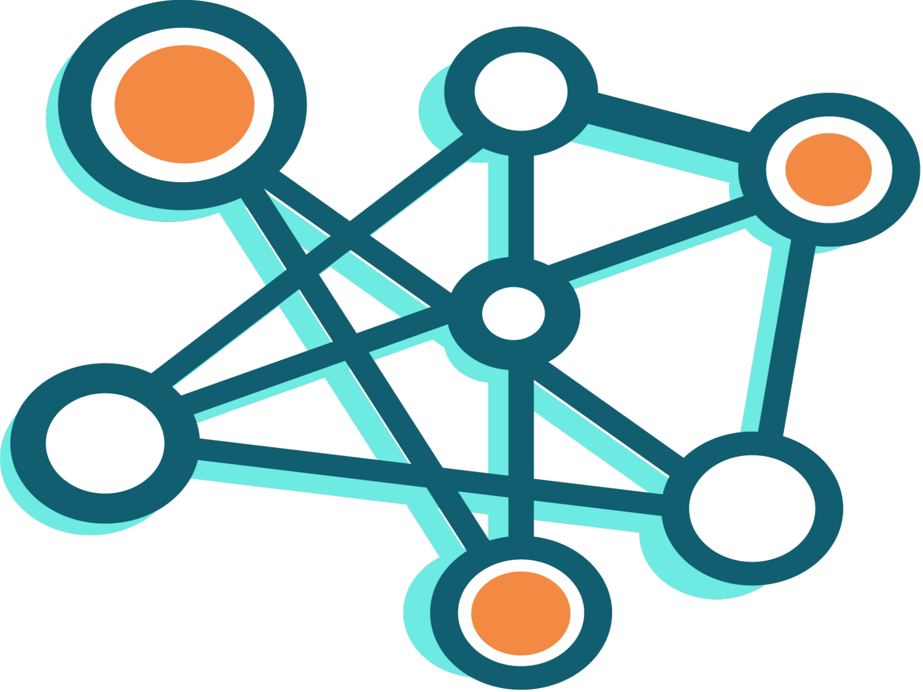
```python
import matplotlib.pyplot as plt
import seaborn as sns

# Temporal distribution
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

# Histogram of ages
clean_data['mid_ma'].hist(bins=50, ax=ax1)
ax1.set_xlabel('Age (Ma)')
ax1.set_ylabel('Number of occurrences')
ax1.set_title('Temporal Distribution')

# Geographic distribution
ax2.scatter(clean_data['lng'], clean_data['lat'],
           alpha=0.5, s=10)
ax2.set_xlabel('Longitude')
ax2.set_ylabel('Latitude')
ax2.set_title('Geographic Distribution')

plt.tight_layout()
plt.show()
```



Beyond Tabular Data: Knowledge Representation

Challenge: Tabular data (CSV, spreadsheets) stores information in rows and columns, but struggles to represent complex relationships.

Tabular format:

- Good for: Simple queries, statistics
- Limited: Relationships are implicit
- Example: “Family” is just a column

Knowledge graph:

- Good for: Complex relationships
- Explicit: Connections are first-class
- Example: “belongs_to” is a relation

Next: We’ll transform our cleaned PBDB data into a knowledge graph to unlock deeper insights.

What is a Knowledge Graph?

Formally, a KG is defined as a collection of triples of the form $\langle h, r, t \rangle$, where:

- h = **head entity** (source node)
- r = **relation** (edge)
- t = **tail entity** (target node)

Each triple represents a true known fact.

Paleontological example:

$$\langle \textit{Elrathia kingii}, \textit{belongs\_to}, \textit{Ptychopariidae} \rangle$$

This triple states the fact that *Elrathia kingii* belongs to the family Ptychopariidae.

KG: Paleontological Entities & Relations

Entities (Nodes):

- Taxonomic: Species, Genus, Family, Order, Class, Phylum
- Temporal: Geological periods, epochs, stages
- Spatial: Localities, formations, collections
- Occurrences: Individual fossil finds

Relations (Edges):

- `belongs_to`: Taxonomic hierarchy
- `found_in`: Geographic occurrence
- `lived_during`: Temporal range
- `collected_from`: Collection information
- `identified_as`: Taxonomic identification

From our cleaned PBDB data, we can extract these entities and relations to build a knowledge graph.

Example Triples from PBDB Data

Taxonomic relations:

$$\langle \textit{Elrathia kingii}, \textit{belongs\_to}, \textit{Elrathia} \rangle$$
$$\langle \textit{Elrathia}, \textit{belongs\_to}, \textit{Ptychopariidae} \rangle$$
$$\langle \textit{Ptychopariidae}, \textit{belongs\_to}, \textit{Ptychopariida} \rangle$$

Temporal relations:

$$\langle \textit{Occurrence\_12345}, \textit{lived\_during}, \textit{Middle Cambrian} \rangle$$

Spatial relations:

$$\langle \textit{Occurrence\_12345}, \textit{found\_in}, \textit{Wheeler Formation} \rangle$$

Identification relations:

$$\langle \textit{Occurrence\_12345}, \textit{identified\_as}, \textit{Elrathia kingii} \rangle$$

Why Use Knowledge Graphs in Paleontology?

Advantages over tabular data:

- Integration: Combine data from PBDB, museum collections, literature
- Querying: Ask complex relationship-based questions
  - "Find all genera in family X that lived during period Y"
  - "What localities contain fossils from the same family?"
- Reasoning: Infer new knowledge via transitivity
  - If $\langle A, belongs\_to, B\rangle$ and $\langle B, belongs\_to, C\rangle$
  - Then A is related to C through B
- Flexibility: Easy to add new entity types and relations
- Visualization: See connections and patterns clearly

When & Who Uses KGs?

When to use:

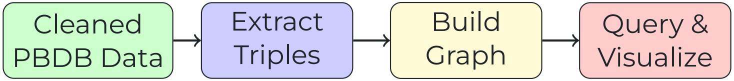
- Highly interconnected data
- Multiple data sources
- Complex relationship queries
- Need for inference/reasoning
- Evolving schemas

Who uses KGs:

- Google KG
- Biomedical ontologies
- Wikidata
- DBpedia
- Scientific domains

Paleontology use cases: Link fossil occurrences with taxonomy, stratigraphy, phylogeny, and paleoenvironments across multiple databases.

Our Workflow: Clean Data → Knowledge Graph



Steps:

1. Start with our cleaned trilobite data
2. Extract entities and relations (create triples)
3. Build graph structure using NetworkX
4. Query and visualize the knowledge graph

Extract Triples from Clean Data (1/4)

Starting with our cleaned data:

```python
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt

# Assume we have our cleaned trilobite data
# (from previous data cleaning steps)
print(f"Clean data: {len(clean_data)} occurrences")
print(f"Columns: {clean_data.columns.tolist()}")

# Initialize triple store
triples = []

# We’ll extract triples following the <h, r, t> pattern
# where h = head entity, r = relation, t = tail entity
```

We transform each row of tabular data into multiple knowledge triples.

Extract Triples from Clean Data (2/4)

Create triples for each occurrence:

```python
# Extract triples from each occurrence
for idx, row in clean_data.iterrows():
    # Create unique occurrence identifier
    occ_id = f"Occurrence_{row['occurrence_no']}"

    # Extract entities
    taxon = row['accepted_name']
    genus = row['genus'] if pd.notna(row['genus']) else None
    family = row['family'] if pd.notna(row['family']) else None
    order = row['order'] if pd.notna(row['order']) else None
    time_period = row['early_interval']
    locality = row['collection_name'] if pd.notna(row['collection_name']) \
               else f"Locality_{idx}"

    # Create identification and spatiotemporal triples
    triples.append((occ_id, 'identified_as', taxon))
    triples.append((occ_id, 'lived_during', time_period))
    triples.append((occ_id, 'found_in', locality))

    # (continued on next slide...)
```

Extract Triples from Clean Data (3/4)

Create taxonomic hierarchy triples:

```python
    # (continued from previous slide...)

    # Create taxonomic hierarchy triples
    if genus:
        triples.append((taxon, 'belongs_to', genus))
    if family:
        parent = genus if genus else taxon
        triples.append((parent, 'belongs_to', family))
    if order:
        parent = family if family else (genus if genus else taxon)
        triples.append((parent, 'belongs_to', order))

print(f"Extracted {len(triples)} triples from {len(clean_data)} occurrences")

# Display example triples
print("\nExample triples:")
for i, (h, r, t) in enumerate(triples[:10]):
    print(f"{i+1}. <{h}, {r}, {t}>")
```

Build Graph from Triples (4/4)

Create directed graph structure:

```python
# Build directed graph from triples
G = nx.DiGraph()

for head, relation, tail in triples:
    G.add_edge(head, tail, relation=relation)

print(f"\nKnowledge Graph Statistics:")
print(f"  Nodes: {G.number_of_nodes()}")
print(f"  Edges: {G.number_of_edges()}")

# Classify node types for visualization
node_types = {}
for node in G.nodes():
    if 'Occurrence' in str(node):
        node_types[node] = 'occurrence'
    elif any(p in str(node) for p in ['Cambrian', 'Ordovician', 'Silurian']):
        node_types[node] = 'time'
    elif 'Locality' in str(node) or 'Formation' in str(node):
        node_types[node] = 'locality'
    else:
        node_types[node] = 'taxon'

print(f"\nNode type distribution:")
for ntype, count in pd.Series(node_types).value_counts().items():
    print(f"  {ntype}: {count}")
```

Visualize the Knowledge Graph (1/2)

Setup visualization with colors by node type:

```python
# Color mapping for different entity types
color_map = {
    'occurrence': 'lightblue',
    'taxon': 'lightgreen',
    'time': 'yellow',
    'locality': 'pink'
}

node_colors = [color_map.get(node_types.get(n, 'taxon'), 'gray')
               for n in G.nodes()]

# Create layout
plt.figure(figsize=(16, 12))
pos = nx.spring_layout(G, k=0.8, iterations=50, seed=42)

# Draw nodes
nx.draw_networkx_nodes(G, pos, node_color=node_colors,
                       node_size=500, alpha=0.8)

# Draw edges
nx.draw_networkx_edges(G, pos, edge_color='gray',
                       arrows=True, arrowsize=10,
                       alpha=0.5, width=1.5)
```

Visualize the Knowledge Graph (2/2)

Add labels and save:

```python
# Draw node labels
labels = {n: n.split('_')[-1] if 'Occurrence' in str(n) else n
          for n in G.nodes()}
nx.draw_networkx_labels(G, pos, labels, font_size=6, font_weight='bold')

# Draw edge labels (relations)
edge_labels = nx.get_edge_attributes(G, 'relation')
nx.draw_networkx_edge_labels(G, pos, edge_labels, font_size=5)

# Add legend
from matplotlib.patches import Patch
legend_elements = [
    Patch(facecolor='lightblue', label='Occurrence'),
    Patch(facecolor='lightgreen', label='Taxon'),
    Patch(facecolor='yellow', label='Time Period'),
    Patch(facecolor='pink', label='Locality')
]
plt.legend(handles=legend_elements, loc='upper right')

plt.title("Knowledge Graph: Trilobite Occurrences from PBDB", fontsize=16)
plt.axis('off')
plt.tight_layout()
plt.savefig('trilobite_kg.png', dpi=300, bbox_inches='tight')
plt.show()
```

Querying: Triple Pattern Matching

Find triples matching specific patterns:

```python
def find_triples(triples, head=None, relation=None, tail=None):
    """
    Find triples matching the pattern <head, relation, tail>
    Use None as wildcard for any component
    """
    results = []
    for h, r, t in triples:
        if ((head is None or h == head) and
            (relation is None or r == relation) and
            (tail is None or t == tail)):
            results.append((h, r, t))
    return results

# Query 1: Find all entities belonging to 'Ptychopariidae'
q1 = find_triples(triples, head=None, relation='belongs_to',
                  tail='Ptychopariidae')
print(f"Found {len(q1)} entities in Ptychopariidae:")
for h, r, t in q1[:3]:
    print(f"  <{h}, {r}, {t}>")

# Query 2: Find all occurrences from 'Middle Cambrian'
q2 = find_triples(triples, head=None, relation='lived_during',
                  tail='Middle Cambrian')
print(f"\nFound {len(q2)} occurrences in Middle Cambrian")
```

Beyond Basic Knowledge Graphs

Advanced topics (beyond this lecture):

- Graph databases: Neo4j, Amazon Neptune for large-scale KGs
- Ontologies: Formal schemas (OWL, RDF) for knowledge representation
- SPARQL: Query language specifically for KGs
- KG embeddings: ML on graph structure
- Link prediction: Infer missing relationships
- Integration: Combine multiple data sources (PBDB + museum data + literature)

Resources:

- NetworkX documentation: `https://networkx.org`
- Neo4j (graph database): `https://neo4j.com`
- RDF and SPARQL: `https://www.w3.org/RDF/`

Summary

Step	**Action**	**Key Outputs**
1. Download	Retrieved PBDB data via API	Raw occurrence dataset
2. Inspect	Assessed quality dimensions	Identified missing coords, invalid ages
3. Clean	Removed/flagged bad records	Clean dataset, quality metrics
4. Standardize	Unified taxonomy, calculated ages	Consistent, analysis-ready data
5. Validate	Documented decisions	Metadata, removal statistics
6. Visualize	Created distribution plots	Temporal/spatial patterns
7. Extract	Converted tables to triples	$\langle h, r, t \rangle$ collection
8. Build KG	Constructed graph structure	NetworkX graph object
9. Visualize KG	Plotted entities & relations	Network diagram
10. Query	Pattern matching, reasoning	Complex relationship queries

Complete pipeline: Raw data $\rightarrow$ Quality assessment $\rightarrow$ Cleaning $\rightarrow$ Validation $\rightarrow$ Visualization $\rightarrow$ Knowledge representation $\rightarrow$ Advanced querying

QR-Code

`https://github.com/matpato/Paleo_DataScience.git`

Thanks!

High-quality data enables powerful knowledge representation and discovery.