Contents

1 Extracting Taxon Objects from *.txt.ann Files

This document explains how to use the `taxon.py`, `finder.py`, `paragraph.py`, and `file.py` modules to extract Taxon objects from annotated text files (*.txt.ann).

1.1 Overview

The extraction process follows a pipeline architecture:

```
*.txt.ann files → Lines → Paragraphs → Taxon objects
```

For CouchDB-backed data:

`CouchDB attachments → Line objects (with metadata) → Paragraphs → Taxon objects`

The modules work together as follows:

1. **file.py**: Reads text files and produces `Line` objects
2. **couchdb_file.py**: Reads CouchDB attachments and produces `Line` objects with populated CouchDB metadata fields (doc_id, attachment_name, db_name)
3. **paragraph.py**: Groups `Line` objects into `Paragraph` objects
4. **finder.py**: Orchestrates the pipeline and provides parsing functions
5. **taxon.py**: Groups `Paragraph` objects into `Taxon` objects (Nomenclature + Description)

1.2 Annotated File Format

Annotated files (*.txt.ann) use YEDDA annotation format:

`[@text content here#Label*]`

Where: - `[@` marks the start of an annotated paragraph - `#Label*]` marks the end with the label - Common labels: `Nomenclature`, `Description`, `Misc-exposition`, `Bibliography`, etc.

1.3 Module Details

1.3.1 1. file.py - File Reading

The `File` class reads text files line by line and tracks metadata.

Key Components: - `File(filename)`: Opens and reads a file - `read_line()`: Iterator yielding `Line` objects - `read_files(files)`: Process multiple files

Example:

```
from file import read_files

# Read multiple files
files = ['path/to/file1.txt.ann', 'path/to/file2.txt.ann']
lines = read_files(files)

for line in lines:
    print(line.filename, line.line_number, line.line)
```

Line Object Attributes: - `line`: The text content - `filename`: Source file path - `line_number`: Line number within file - `page_number`: Page number (if present) - `empirical_page_number`: Page number as

printed in document - `contains_start()`: Returns True if line starts with `[@` - `end_label()`: Returns label value if line ends with `#Label*]`

1.3.2 2. paragraph.py - Paragraph Parsing

The `Paragraph` class groups related lines together and maintains label information.

Key Components: - `Paragraph()`: Container for lines and labels - `append()`: Add a line to the paragraph - `top_label()`: Get the paragraph's label - `as_annotated()`: Format as annotated text - `as_dict()`: Convert to dictionary

Paragraph Attributes: - `filename`: Source filename - `paragraph_number`: Sequential paragraph number - `page_number`: Page location - `labels`: List of Label objects - `body`: Full text content via `str(paragraph)`

1.3.3 3. finder.py - Parsing Pipeline

The `finder.py` module provides two main parsing functions for converting lines into paragraphs.

1.3.3.1 parse_annotated()

Use this function for annotated files where paragraph boundaries are explicitly marked with `[@...#Label*]` tags.

```
from file import read_files
from finder import parse_annotated

# Read annotated files
files = ['path/to/file.txt.ann']
lines = read_files(files)

# Parse into paragraphs using annotation boundaries
paragraphs = parse_annotated(lines)

for pp in paragraphs:
    print(f"Label: {pp.top_label()}")
    print(f"Text: {pp}")
```

When to use: For *.txt.ann files where annotations define paragraph boundaries.

1.3.3.2 parse_paragraphs()

Use this function for raw text files where heuristic rules determine paragraph boundaries.

```python
from file import read_files
from finder import parse_paragraphs

# Read raw text files
files = ['path/to/file.txt']
lines = read_files(files)

# Parse into paragraphs using heuristics
paragraphs = parse_paragraphs(lines)

for pp in paragraphs:
    print(f"Paragraph: {pp}")
```

Heuristic Rules: - Nomenclature patterns (genus + species + year) - Blank lines - Short lines - Tab indentation - Page breaks - Tables and figures

When to use: For raw *.txt files without annotation markup.

1.3.4 4. taxon.py - Taxon Grouping

The Taxon class represents a taxonomic entity with its nomenclature and description paragraphs.

Key Components: - `Taxon()`: Container for nomenclature and description paragraphs - `add_nomenclature(paragraph)`: Add a nomenclature paragraph - `add_description(paragraph)`: Add a description paragraph - `has_nomenclature()`: Check if taxon has nomenclature - `has_description()`: Check if taxon has description - `dictionaries()`: Iterate over all paragraphs as dictionaries - `as_row()`: Get a single-row dictionary representation

group_paragraphs() Function:

The main function for extracting Taxon objects:

```python
from file import read_files
from finder import parse_annotated
from taxon import group_paragraphs

# Read and parse files
files = ['path/to/file.txt.ann']
lines = read_files(files)
paragraphs = parse_annotated(lines)

# Group into Taxon objects
for taxon in group_paragraphs(paragraphs):
    print(f"Taxon {taxon._serial}:")
```

```
    for paragraph_dict in taxon.dictionaries():
        print(f"  {paragraph_dict['label']}: {paragraph_dict['body'][:100]}...")
```

Grouping Algorithm:

The function uses a state machine with two states:

1. **“Look for Nomenclatures”**: Collect consecutive Nomenclature paragraphs
2. **“Look for Descriptions”**: Collect Description paragraphs that follow

A new Taxon is yielded when: - A new Nomenclature appears after collecting Descriptions - Too many paragraphs (>6) pass without finding relevant content - End of file is reached

Only complete taxa (with both Nomenclature and Description) are yielded.

1.4 Complete Extraction Example

Here’s a complete example showing the full pipeline:

```
import csv
import sys
from file import read_files
from finder import parse_annotated
from taxon import Taxon, group_paragraphs

def extract_taxa_from_files(file_paths):
    """
    Extract Taxon objects from annotated files.

    Args:
        file_paths: List of paths to *.txt.ann files

    Returns:
        Iterator of Taxon objects
    """
    # Step 1: Read files into Line objects
    lines = read_files(file_paths)

    # Step 2: Parse Lines into Paragraph objects using annotations
    paragraphs = parse_annotated(lines)

    # Step 3: Group Paragraphs into Taxon objects
    taxa = group_paragraphs(paragraphs)
```

```
    return taxa

def main():
    # Example: Extract taxa and write to CSV
    files = [
        'data/annotated/journals/Mycotaxon/Vol117/s10.txt.ann',
        'data/annotated/journals/Mycotaxon/Vol117/s11.txt.ann'
    ]

    # Extract taxa
    taxa = extract_taxa_from_files(files)

    # Write to CSV
    writer = csv.DictWriter(sys.stdout, fieldnames=Taxon.FIELDNAMES)
    writer.writeheader()

    for taxon in taxa:
        # Each taxon can produce multiple rows (one per paragraph)
        for paragraph_dict in taxon.dictionaries():
            writer.writerow(paragraph_dict)

if __name__ == '__main__':
    main()
```

1.4.1 Output Format

The CSV output has these columns (defined in `Taxon.FIELDNAMES`):

- `serial_number`: Unique taxon identifier
- `filename`: Source file
- `label`: Paragraph label (Nomenclature or Description)
- `paragraph_number`: Paragraph sequence number
- `page_number`: Page number in file
- `empirical_page_number`: Printed page number
- `body`: Full paragraph text

1.4.2 Alternative Output: Single Row per Taxon

To get one row per taxon with combined nomenclature and description:

```
for taxon in group_paragraphs(paragraphs):
    row = taxon.as_row()
    print(f"Nomenclature: {row['taxon']}")
    print(f"Description: {row['description']}")
    print(f"Page: {row['page_number']}")
```

1.5 Command-Line Usage

The `finder.py` module can be run directly as a script with the `--group_paragraphs` flag:

```
python finder.py --group_paragraphs --annotated_paragraphs \
    data/annotated/**/*.txt.ann
```

Useful flags:

- `--annotated_paragraphs`: Use annotated boundaries (for *.txt.ann files)
- `--group_paragraphs`: Group paragraphs into Taxon objects
- `--output_label Nomenclature Description`: Filter output by label
- `--dump_input`: Show the parsed paragraphs before grouping

1.6 Advanced Filtering

1.6.1 Remove Interstitial Paragraphs

```
from finder import remove_interstitials

# Filter out figures, tables, and blank paragraphs
paragraphs = parse_annotated(lines)
filtered = remove_interstitials(paragraphs)
taxa = group_paragraphs(filtered)
```

1.6.2 Filter by Label

```
from finder import target_classes
from label import Label

# Keep only specific labels, mark others as "Misc-exposition"
paragraphs = parse_annotated(lines)
filtered = target_classes(
    paragraphs,
    default=Label('Misc-exposition'),
    keep=[Label('Nomenclature'), Label('Description')]
)
taxa = group_paragraphs(filtered)
```

1.7 Taxon Object Structure

A Taxon object contains:

Taxon

```
├── _serial (int): Unique identifier
├── _nomenclatures (List[Paragraph]): Nomenclature paragraphs
└── _descriptions (List[Paragraph]): Description paragraphs
```

Each Paragraph contains:

```
Paragraph
├── _lines (List[Line]): Text lines
├── _labels (List[Label]): Label stack
├── _paragraph_number (int): Sequence number
├── filename (str): Source file
├── page_number (int): Page location
└── empirical_page_number (str): Printed page number
```

1.8 Common Patterns

1.8.1 Pattern 1: Extract All Taxa from Directory

```python
import glob
from file import read_files
from finder import parse_annotated
from taxon import group_paragraphs

# Find all annotated files
files = glob.glob('data/annotated/**/*.txt.ann', recursive=True)

# Extract all taxa
lines = read_files(files)
paragraphs = parse_annotated(lines)
all_taxa = list(group_paragraphs(paragraphs))

print(f"Found {len(all_taxa)} taxa across {len(files)} files")
```

1.8.2 Pattern 2: Filter by Filename

```python
# Extract taxa only from specific journal
files = glob.glob('data/annotated/journals/Mycologia/**/*.txt.ann', recursive=Tr
lines = read_files(files)
paragraphs = parse_annotated(lines)

for taxon in group_paragraphs(paragraphs):
    # All taxa from Mycologia journal
    process_taxon(taxon)
```

1.8.3 Pattern 3: Count Taxa per File

```
from collections import defaultdict

files = ['file1.txt.ann', 'file2.txt.ann']
lines = read_files(files)
paragraphs = parse_annotated(lines)

taxa_by_file = defaultdict(list)
for taxon in group_paragraphs(paragraphs):
    # Get filename from first nomenclature paragraph
    filename = list(taxon.dictionaries())[0]['filename']
    taxa_by_file[filename].append(taxon)

for filename, taxa in taxa_by_file.items():
    print(f"{filename}: {len(taxa)} taxa")
```

1.9 Error Handling

The parser is robust but watch for these issues:

1. **Malformed annotations**: Missing `[@` or `#Label*]` markers
2. **Unmatched labels**: Opening `[@` without closing `*]`
3. **Empty files**: No paragraphs or labels found
4. **Encoding issues**: Use UTF-8 encoding for all files

1.10 Extracting from CouchDB (PySpark Distributed)

For large-scale processing of annotated files stored in CouchDB, use the distributed PySpark approach.

1.10.1 Using couchdb_file.py

The `couchdb_file.py` module provides a UDF alternative to `read_files()` that:

- Reads attachments from CouchDB in PySpark partitions
- Preserves database metadata (doc_id, attachment_name, db_name)
- Works with `CouchDBConnection` for efficient distributed I/O
- Produces `Line` objects compatible with existing parsers

Key Classes:

- `CouchDBFile`: File-like object for CouchDB attachment content

- `Line`: Line class with optional CouchDB metadata fields (doc_id, attachment_name, db_name)
- `read_couchdb_partition()`: Process CouchDB rows in a partition, returns Line objects with metadata

Example: Distributed Processing

```python
from pyspark.sql import SparkSession
from skol_classifier.couchdb_io import CouchDBConnection
from couchdb_file import read_couchdb_partition
from finder import parse_annotated, remove_interstitials
from taxon import group_paragraphs

def process_partition_to_taxa(partition, db_name):
    """Process partition of CouchDB rows into taxa."""
    # Read CouchDB rows into Line objects
    lines = read_couchdb_partition(partition, db_name)

    # Parse into paragraphs
    paragraphs = parse_annotated(lines)

    # Filter and group
    filtered = remove_interstitials(paragraphs)
    taxa = group_paragraphs(filtered)

    # Convert to dictionaries
    for taxon in taxa:
        for para_dict in taxon.dictionaries():
            # Extract CouchDB metadata from composite filename
            parts = para_dict['filename'].split('/', 2)
            if len(parts) == 3:
                para_dict['db_name'] = parts[0]
                para_dict['doc_id'] = parts[1]
                para_dict['attachment_name'] = parts[2]
            yield para_dict

# Create Spark session
spark = SparkSession.builder.appName("TaxonExtractor").getOrCreate()

# Load from CouchDB
conn = CouchDBConnection("http://localhost:5984", "mycobank", "user", "pass")
df = conn.load_distributed(spark, "*.txt.ann")

# Process in parallel
taxa_rdd = df.rdd.mapPartitions(
    lambda part: process_partition_to_taxa(part, "mycobank")
)
```

```python
# Convert to DataFrame and save
from pyspark.sql.types import StructType, StructField, StringType

schema = StructType([
    StructField("serial_number", StringType(), False),
    StructField("db_name", StringType(), True),
    StructField("doc_id", StringType(), True),
    StructField("attachment_name", StringType(), True),
    StructField("label", StringType(), False),
    StructField("body", StringType(), False),
    # ... other fields
])

taxa_df = taxa_rdd.toDF(schema)
taxa_df.write.parquet("output/taxa.parquet")
```

Example: Local Processing from CouchDB

```python
from couchdb_file import read_couchdb_files_from_connection
from skol_classifier.couchdb_io import CouchDBConnection

# Connect to CouchDB
conn = CouchDBConnection("http://localhost:5984", "mycobank")

# Read files with metadata
lines = read_couchdb_files_from_connection(
    conn, spark, db_name="mycobank", pattern="*.txt.ann"
)

# Parse and extract
paragraphs = parse_annotated(lines)
taxa = list(group_paragraphs(paragraphs))

# Access CouchDB metadata from lines
for taxon in taxa:
    for para_dict in taxon.dictionaries():
        # filename format: "db_name/doc_id/attachment_name"
        print(f"From: {para_dict['filename']}")
```

1.10.2 Line Metadata for CouchDB

When `Line` objects are created from `CouchDBFile`, they include these optional CouchDB properties:

- `doc_id`: CouchDB document ID (Optional[str])

- `attachment_name`: Attachment filename, e.g., "article.txt.ann" (Optional[str])
- `db_name`: Database name - ingest_db_name for tracking data source (Optional[str])
- `filename`: Composite identifier in format "db_name/doc_id/attachment_name"

This metadata is preserved through the entire pipeline and appears in the final Taxon output. For regular file-based Line objects, these fields are None.

1.10.3 Complete CouchDB Example

See examples/extract_taxa_from_couchdb.py for a complete working example with:

- Distributed Spark processing
- Local processing for small datasets
- Filtering and analysis examples
- Command-line interface

Usage:

```
# Distributed processing
python examples/extract_taxa_from_couchdb.py \
    --mode distributed \
    --database mycobank_annotations \
    --db-name mycobank \
    --username admin \
    --password secret

# Local processing (for testing)
python examples/extract_taxa_from_couchdb.py \
    --mode local \
    --database mycobank_annotations \
    --db-name mycobank
```

1.10.4 Why Use CouchDB Integration?

Benefits of the CouchDB-aware approach:

1. **Scalability**: Process millions of documents in parallel using Spark
2. **Metadata Preservation**: Track which database/document each taxon came from
3. **Efficient I/O**: One connection per partition, not per document
4. **Traceability**: Full lineage from database to extracted taxa
5. **Flexibility**: Works with existing `parse_annotated()` and `group_paragraphs()` functions

1.11 Summary

To extract Taxon objects from *.txt.ann files:

1.11.1 From Local Files:

1. Import the necessary modules
2. Use `read_files()` to load the files into `Line` objects
3. Use `parse_annotated()` to group lines into `Paragraph` objects
4. Use `group_paragraphs()` to group paragraphs into `Taxon` objects
5. Process each `Taxon` using its methods:
    - `dictionaries()` for row-per-paragraph output
    - `as_row()` for single-row output with combined text

1.11.2 From CouchDB (Distributed):

1. Create `CouchDBConnection` instance
2. Use `conn.load_distributed()` to load attachments into DataFrame
3. Use `read_couchdb_partition()` in `mapPartitions()` to create `Line` objects with CouchDB metadata
4. Use `parse_annotated()` to group lines into `Paragraph` objects
5. Use `group_paragraphs()` to group paragraphs into `Taxon` objects
6. Extract CouchDB metadata from composite filenames in output

The pipeline is designed to be flexible and composable, allowing you to insert filtering or transformation steps at any stage.