

# MRJob Python Library Crash Course

---

## What is MRJob?

MRJob is a Python library that lets you write MapReduce jobs in pure Python and run them on:

- **Your local machine** (for testing)
- **Hadoop clusters**
- **Amazon EMR** (Elastic MapReduce)
- **Google Cloud Dataproc**

It was created by Yelp and makes MapReduce development much simpler than writing raw Hadoop Java code.

## Installation

```
pip install mrjob
```

## Basic Structure

Every MRJob program follows this structure:

```
from mrjob.job import MRJob

class MyJobName(MRJob):

    def mapper(self, _, line):
        # Process each line of input
        # yield (key, value)
        pass

    def reducer(self, key, values):
        # Process all values for a key
        # yield (key, result)
        pass

if __name__ == '__main__':
    MyJobName.run()
```

## Example 1: Word Count

Word count is the "Hello World" of MapReduce - it's simple but demonstrates the core concepts perfectly. This example shows how to break down text into individual words (mapper) and count how many times each word appears across all documents (reducer). It's the foundation for understanding more complex text analysis tasks like building search indexes or analyzing sentiment in social media posts.

**What we're doing:** Taking text input, splitting it into words, and counting the frequency of each word.

**Real-world use:** Log analysis, document indexing, finding trending topics in tweets.

```
from mrjob.job import MRJob

class WordCount(MRJob):

    def mapper(self, _, line):
        # Split line into words and emit each with count 1
        for word in line.split():
            yield (word.lower(), 1)

    def reducer(self, word, counts):
        # Sum all counts for this word
        yield (word, sum(counts))

if __name__ == '__main__':
    WordCount.run()
```

**Running it:**

```
# Run locally
python word_count.py input.txt

# With output file
python word_count.py input.txt > output.txt

# Multiple input files
python word_count.py file1.txt file2.txt file3.txt
```

## Example 2: Finding Most Common Words

Now let's level up! This example introduces **multi-step jobs** - one of MRJob's most powerful features. We'll chain two MapReduce operations together: first counting all words, then finding the top 10 most common ones. This pattern is incredibly useful when you need to sort or rank results from a previous computation.

**What we're doing:** Count all words, then identify which ones appear most frequently.

**Real-world use:** Finding trending hashtags, identifying most accessed URLs in logs, discovering popular products.

**Key concept:** Notice how the second reducer gets **None** as the key? That's a clever trick to force all data to a single reducer for the final ranking step.

```
from mrjob.job import MRJob
from mrjob.step import MRStep

class TopWords(MRJob):
```

```

def steps(self):
    return [
        MRStep(mapper=self.mapper_get_words,
                reducer=self.reducer_count_words),
        MRStep(reducer=self.reducer_find_top)
    ]

def mapper_get_words(self, _, line):
    for word in line.split():
        yield (word.lower(), 1)

def reducer_count_words(self, word, counts):
    yield (None, (sum(counts), word))

def reducer_find_top(self, _, word_count_pairs):
    # Get top 10 words
    top_words = sorted(word_count_pairs, reverse=True)[:10]
    for count, word in top_words:
        yield (word, count)

if __name__ == '__main__':
    TopWords.run()

```

## Multi-Step Jobs

Before we dive into more examples, let's understand multi-step jobs. Sometimes one MapReduce pass isn't enough - you might need to process data, then aggregate those results, then maybe rank them. MRJob makes this easy by letting you chain multiple Map and Reduce phases together. Each step can have its own mapper and reducer, and the output of one step automatically becomes the input to the next.

**When you need this:** Sorting final results, computing percentiles, iterative algorithms, complex aggregations.

```

from mrjob.job import MRJob
from mrjob.step import MRStep

class MultiStepJob(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper1,
                    reducer=self.reducer1),
            MRStep(mapper=self.mapper2,
                    reducer=self.reducer2),
            MRStep(reducer=self.reducer3)
        ]

    def mapper1(self, _, line):
        # First map phase
        pass

    def reducer1(self, key, values):

```

```
# First reduce phase
pass

# ... and so on
```

## Example 3: Log Analysis

Web server logs are one of the most common big data sources, and this example shows practical log parsing with regex. We're extracting meaningful metrics from Apache/Nginx logs - counting HTTP status codes and measuring bandwidth usage per URL. This is the kind of analysis that runs in production at companies every day.

**What we're doing:** Parse structured log lines, extract relevant fields, and aggregate metrics.

**Real-world use:** Monitor error rates (count 404s and 500s), identify high-traffic pages, detect bandwidth hogs, security analysis.

**Key concept:** One mapper can emit multiple different keys! We're counting status codes AND summing bytes per path in the same job.

```
from mrjob.job import MRJob
import re

class LogAnalyzer(MRJob):

    PATTERN = re.compile(
        r'(\S+) - - \[(.*?)\] "(\S+) (\S+) (\S+)" (\d+) (\d+)'
    )

    def mapper(self, _, line):
        match = self.PATTERN.match(line)
        if match:
            ip, timestamp, method, path, protocol, status, size = match.groups()
            yield (status, 1) # Count by status code
            yield (path, int(size)) # Sum bytes by path

    def reducer(self, key, values):
        yield (key, sum(values))

if __name__ == '__main__':
    LogAnalyzer.run()
```

## Example 4: Average Calculation

Computing averages in distributed systems is trickier than it looks! You can't just average the averages from different mappers. This example shows the correct approach: collect all values for each key, then compute the average in the reducer. We're also outputting structured JSON data instead of simple values.

**What we're doing:** Calculate the average rating and total review count for each product.

**Real-world use:** Product ratings, student grade averages, sensor data analysis, performance metrics.

**Important note:** We have to collect all values in a list first because the `values` iterator can only be traversed once. This is a common gotcha in MapReduce!

```
from mrjob.job import MRJob
from mrjob.step import MRStep

class AverageRatings(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper_parse,
                  reducer=self.reducer_average)
        ]

    def mapper_parse(self, _, line):
        # Input format: product_id,rating
        product_id, rating = line.split(',')
        yield (product_id, float(rating))

    def reducer_average(self, product_id, ratings):
        ratings_list = list(ratings)
        avg = sum(ratings_list) / len(ratings_list)
        yield (product_id, {
            'average': avg,
            'count': len(ratings_list)
        })

if __name__ == '__main__':
    AverageRatings.run()
```

## Combiners

Here's a performance optimization that can make your jobs dramatically faster! Combiners are like "mini-reducers" that run on each mapper node before data is sent across the network. For word count, instead of sending thousands of `("hello", 1)` pairs across the network, we can pre-sum them locally to `("hello", 437)` and send just one pair. This can reduce network traffic by orders of magnitude!

**When to use:** Any time your reduce operation is associative and commutative (like sum, max, min, count).

**When NOT to use:** Computing averages (can't average averages), operations that need all data together.

```
from mrjob.job import MRJob

class WordCountWithCombiner(MRJob):

    def mapper(self, _, line):
        for word in line.split():
            yield (word.lower(), 1)
```

```
def combiner(self, word, counts):
    # Local aggregation before sending to reducer
    yield (word, sum(counts))

def reducer(self, word, counts):
    yield (word, sum(counts))

if __name__ == '__main__':
    WordCountWithCombiner.run()
```

## Custom Protocols

By default, MRJob expects plain text input and outputs tab-separated key-value pairs. But what if your data is JSON? Or CSV? Or some custom format? Protocols let you specify how to serialize and deserialize your data. This example shows working with JSON, which is incredibly common in modern data pipelines.

**What we're doing:** Reading JSON records, processing them as Python dictionaries, and outputting JSON results.

**Real-world use:** Processing API responses, working with MongoDB exports, integrating with JSON-based systems.

**Bonus:** You can create custom protocols for CSV, XML, Avro, or any format you need!

```
from mrjob.job import MRJob
from mrjob.protocol import JSONProtocol, RawValueProtocol

class JSONProcessor(MRJob):

    # Read JSON input
    INPUT_PROTOCOL = JSONProtocol
    # Write JSON output
    OUTPUT_PROTOCOL = JSONProtocol

    def mapper(self, _, record):
        # record is already parsed JSON
        yield (record['user_id'], record['purchase_amount'])

    def reducer(self, user_id, amounts):
        yield (user_id, {'total': sum(amounts)})

if __name__ == '__main__':
    JSONProcessor.run()
```

## Running on Different Platforms

### Local Mode (Default)

```
python my_job.py input.txt
```

## Inline Mode (Single Process)

```
python my_job.py -r inline input.txt
```

## Hadoop

```
python my_job.py -r hadoop hdfs:///input/path/ -o hdfs:///output/path/
```

## Amazon EMR

```
python my_job.py -r emr s3://my-bucket/input/ -o s3://my-bucket/output/
```

## Google Cloud Dataproc

```
python my_job.py -r dataproc gs://my-bucket/input/ -o gs://my-bucket/output/
```

# Configuration

## Using mrjob.conf

Create a configuration file `mrjob.conf`:

```
runners:
  emr:
    aws_region: us-east-1
    instance_type: m5.xlarge
    num_core_instances: 4

  hadoop:
    hadoop_home: /usr/local/hadoop
```

## Command Line Options

```
# Specify runner
python job.py -r emr input.txt
```

```
# Set number of reducers
python job.py --num-reducers 10 input.txt

# Configure output format
python job.py --output-dir /output/path input.txt

# Cleanup temp files
python job.py --cleanup ALL input.txt
```

## Advanced Features

### Custom Options

```
from mrjob.job import MRJob

class CustomOptionsJob(MRJob):

    def configure_args(self):
        super(CustomOptionsJob, self).configure_args()
        self.add_passthru_arg(
            '--min-count', type=int, default=5,
            help='Minimum count threshold'
        )

    def reducer(self, word, counts):
        total = sum(counts)
        if total >= self.options.min_count:
            yield (word, total)

if __name__ == '__main__':
    CustomOptionsJob.run()
```

### Usage:

```
python job.py --min-count 10 input.txt
```

### Secondary Sort

Sometimes you need the values in your reducer to arrive in a specific order. Secondary sort lets you use a composite key (a tuple) where part of the key determines which reducer gets the data, and part determines the sort order. This is useful for time-series data or when you need to process events chronologically.

**What we're doing:** Group user actions by user, but ensure they arrive sorted by timestamp.

**Real-world use:** Reconstructing user sessions, processing time-series data, analyzing sequential events.

**Technique:** We use a tuple (`user`, `timestamp`) as the key. All records with the same user go to the same reducer, sorted by timestamp.

```
from mrjob.job import MRJob

class SecondarySortJob(MRJob):

    def mapper(self, _, line):
        user, timestamp, action = line.split(',')
        # Composite key: (user, timestamp)
        yield ((user, timestamp), action)

    def reducer(self, key, values):
        user, timestamp = key
        actions = list(values)
        yield (user, {'timestamp': timestamp, 'actions': actions})

if __name__ == '__main__':
    SecondarySortJob.run()
```

## Counters

Want to track statistics about your job while it runs? Counters are your friend! They let you count things like "how many malformed records did we skip?" or "how many rare events did we find?" without cluttering your actual output. Counters are aggregated across all mappers and reducers, giving you global statistics about your job execution.

**What we're doing:** Track how many lines and words we process, plus how many "common" words we find.

**Real-world use:** Data quality monitoring, debugging, tracking rare events, measuring job progress.

**Bonus:** Hadoop displays these counters in the job monitoring UI, making them great for operations teams!

```
from mrjob.job import MRJob

class CounterExample(MRJob):

    def mapper(self, _, line):
        words = line.split()
        self.increment_counter('stats', 'lines', 1)
        self.increment_counter('stats', 'words', len(words))

        for word in words:
            yield (word.lower(), 1)

    def reducer(self, word, counts):
        total = sum(counts)
        if total > 100:
            self.increment_counter('stats', 'common_words', 1)
        yield (word, total)

if __name__ == '__main__':
    CounterExample.run()
```

## Testing MRJob Code

Never deploy untested code to a cluster! MRJob makes testing easy by letting you run jobs with fake input data and verify the output programmatically. This example shows how to use Python's unittest framework with MRJob. Write tests locally, iterate quickly, and only deploy to expensive cluster resources when you're confident your logic is correct.

**Why this matters:** Cluster time is expensive! A bug that requires 10 iterations to fix could cost hundreds of dollars in EMR charges. Test locally first.

```
from io import BytesIO
from mrjob.job import MRJob

def test_word_count():
    job = WordCount()

    # Create fake input
    input_bytes = b'hello world\nhello mrjob\n'
    job.sandbox(stdin=BytesIO(input_bytes))

    # Run the job
    with job.make_runner() as runner:
        runner.run()

    # Check output
    results = {}
    for key, value in job.parse_output(runner.cat_output()):
        results[key] = value

    assert results['hello'] == 2
    assert results['world'] == 1
    assert results['mrjob'] == 1
```

## Real-World Example: Click Stream Analysis

Let's put it all together with a realistic example! This is the kind of analysis that e-commerce sites and content platforms run constantly. We're analyzing user behavior by processing clickstream logs - tracking what pages users visit, what actions they take, and deriving meaningful metrics about user engagement. This uses multiple steps and shows how to work with structured data.

**The business question:** "How engaged are our users? How many pages does an average user visit per session?"

**What we're doing:**

1. Parse raw clickstream logs into structured events
2. Group events by user and count their activity
3. Compute aggregate statistics across all users

**Real-world use:** User engagement analysis, conversion funnel tracking, A/B test analysis, identifying power users.

```
from mrjob.job import MRJob
from mrjob.step import MRStep
from datetime import datetime

class ClickStreamAnalysis(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper_parse_logs,
                    reducer=self.reducer_user_sessions),
            MRStep(reducer=self.reducer_session_stats)
        ]

    def mapper_parse_logs(self, _, line):
        # Parse: timestamp,user_id,page_url,action
        parts = line.strip().split(',')
        if len(parts) == 4:
            timestamp, user_id, page_url, action = parts
            yield (user_id, {
                'timestamp': timestamp,
                'page': page_url,
                'action': action
            })

    def reducer_user_sessions(self, user_id, events):
        # Group events by user
        events_list = sorted(events, key=lambda x: x['timestamp'])
        session_count = len(events_list)
        pages_visited = len(set(e['page'] for e in events_list))

        yield (None, {
            'user_id': user_id,
            'events': session_count,
            'unique_pages': pages_visited
        })

    def reducer_session_stats(self, _, user_stats):
        stats_list = list(user_stats)
        total_users = len(stats_list)
        avg_events = sum(s['events'] for s in stats_list) / total_users
        avg_pages = sum(s['unique_pages'] for s in stats_list) / total_users

        yield ('summary', {
            'total_users': total_users,
            'avg_events_per_user': avg_events,
            'avg_unique_pages': avg_pages
        })
```

```
if __name__ == '__main__':  
    ClickStreamAnalysis.run()
```

## Best Practices

1. **Test locally first:** Always test with small data locally before running on a cluster
2. **Use combiners:** Reduce network traffic when possible
3. **Keep state minimal:** Mappers and reducers should be stateless
4. **Handle errors gracefully:** Use try-except blocks for parsing
5. **Use appropriate protocols:** JSON for structured data, raw for simple text
6. **Set appropriate number of reducers:** Balance parallelism with overhead
7. **Clean up:** Use `--cleanup` option to remove temporary files
8. **Monitor counters:** Track job progress and data quality

## Common Pitfalls

### ✗ Don't:

- Store large objects in memory
- Make external API calls in mappers (too slow)
- Assume ordering of values in reducer
- Forget to handle malformed input

### ☑ Do:

- Process data incrementally
- Use generators for large value lists
- Handle edge cases and errors
- Test with realistic data samples

## Performance Tips

1. **Optimize mapper output:** Reduce data sent to reducers
2. **Use compression:** Enable for intermediate data
3. **Partition wisely:** Ensure even distribution across reducers
4. **Minimize memory usage:** Stream data instead of loading all at once
5. **Profile your code:** Identify bottlenecks before scaling up

## Quick Command Reference

```
# Run locally  
python job.py input.txt  
  
# Run inline (single process, easier debugging)  
python job.py -r inline input.txt  
  
# Multiple inputs  
python job.py input1.txt input2.txt input3.txt
```

```
# Save to file
python job.py input.txt > output.txt

# Set number of tasks
python job.py --jobconf mapred.reduce.tasks=10 input.txt

# Verbose output
python job.py -v input.txt

# Clean up files
python job.py --cleanup ALL input.txt
```

## Conclusion

MRJob makes MapReduce accessible to Python developers by:

- Abstracting away Hadoop complexity
- Providing a simple, Pythonic API
- Enabling local testing before cluster deployment
- Supporting multiple cloud platforms

It's perfect for batch processing large datasets when you need the power of MapReduce without the Java complexity.