UAB
Universitat Autònoma
de Barcelona

Packages and Biopython Exercises

Alejandro Cobos

MV - HPC and Big Data Analysis

07 January 2026

Table of Contents

1. Fasta Sequence Counter with OS Module

[See Python Modules and BioPython - Slide 46]

[E2] Implement a program that counts the total amount of sequences contained in all the FASTA files located in a given directory tree.

1.1. Implementation

```python
import os


def count_sequences_in_fasta(filepath: str) -> int:
    """Count the number of sequences in a FASTA file by counting lines starting
with '>'"""
    count = 0
    with open(filepath, "r") as f:
        for line in f:
            if line.startswith(">"):
                count += 1
    return count


def count_all_sequences(root_dir: str) -> int:
    """Count total sequences in all FASTA files within a directory tree"""
    total_sequences = 0

    for dirpath, dirnames, filenames in os.walk(root_dir):
        for filename in filenames:
            if filename.endswith(".fasta") or filename.endswith(".fa"):
                filepath = os.path.join(dirpath, filename)

                seq_count = count_sequences_in_fasta(filepath)
                total_sequences += seq_count

                print(f"{filepath}: {seq_count} sequences")

    return total_sequences


if __name__ == "__main__":
    main_directory = os.getcwd()
    total: int = count_all_sequences(main_directory)
    print(f"\nTotal sequences found: {total}")
```

1.2. Notes

The core of this implementation relies on the `os` module. Specifically, `os.walk` is used to recursively traverse the directory tree. This approach ensures that all FASTA files are discovered regardless of their depth in the folder structure, avoiding the need for hardcoded paths.

To keep the script lightweight and focused on standard library modules, no external bioinformatics libraries were used. The sequence count is determined by identifying lines that start with the `>` character, which is the standard header indicator in FASTA format.

1.3. Results

To validate the solution, the following directory structure was created:

```
MainDir
├── fastaFinder.py
├── sample_01.fasta         (5 sequences)
├── sample_02.fasta         (4 sequences)
├── subdir_A
│   ├── sample_03.fasta     (10 sequences)
│   ├── sample_04.fasta     (1 sequence)
│   └── subdir_B
│       └── sample_05.fasta (4 sequences)
```

The script was executed from the main directory:

```
$ python fastaFinder.py
```

The output confirms the recursive search successfully identified sequences in all subfolders:

```
sample_01.fasta: 5 sequences
sample_02.fasta: 4 sequences
subdir_A/sample_03.fasta: 10 sequences
subdir_A/sample_04.fasta: 1 sequences
subdir_A/subdir_B/sample_05.fasta: 4 sequences

Total sequences found: 24
```

2. SLURM Workflow Submission with Python

[See Python Modules and BioPython - Slide 47]

Write a Python script that generates and submits to SLURM a workflow composed of multiple jobs with dependencies.

Hint: use `subprocess` methods to invoke `sbatch` commands and to obtain the SLURM job IDs

2.1. Implementation

The following script automates the submission of SLURM jobs while enforcing a specific execution order as defined in the workflow diagram (Fig. 1).

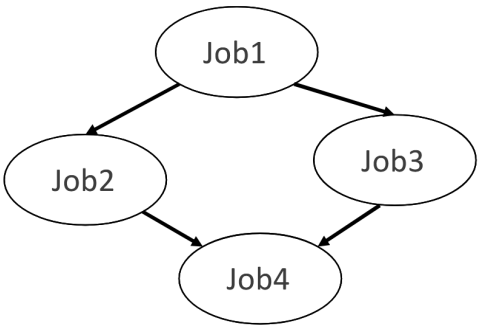

Figure 1: Workflow diagram showing job dependencies.

```python
#!/usr/bin/env python3

import subprocess
import re


def submit_job(script_name: str, dependency=None) -> str:
    """Submit SLURM job and return its job ID."""
    cmd = ["sbatch"]

    if dependency:
        cmd.append(f"--dependency=afterok:{dependency}")

    cmd.append(script_name)

    result = subprocess.run(cmd, capture_output=True, text=True, check=True)
    job_id = re.search(r"(\d+)", result.stdout).group(1)
    return job_id


def main():
    job1_id = submit_job("job1.sh")
    print(f"Job1: {job1_id}")

    job2_id = submit_job("job2.sh", dependency=job1_id)
    print(f"Job2: {job2_id}")
```

```
    job3_id = submit_job("job3.sh", dependency=job1_id)
    print(f"Job3: {job3_id}")

    job4_id = submit_job("job4.sh", dependency=f"{job2_id}:{job3_id}")
    print(f"Job4: {job4_id}")


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

2.2. Notes

This solution utilizes the `subprocess` module to interact with the system's `sbatch` command. Each command is passed as a list of strings, which is the recommended practice for security and to ensure arguments like `--dependency=afterok:12345` are parsed correctly by the shell.

The `submit_job` function captures the standard output of the command. Because `sbatch` returns a string (e.g., "Submitted batch job 12345"), a regular expression `re.search(r"(\d+)", ...)` is used to isolate the numeric Job ID.

The dependency logic is handled as follows:

- **Branching**: Job2 and Job3 both depend on Job1. Once Job1 succeeds, SLURM can run Job2 and Job3 in parallel.
- **Merging**: Job4 depends on both Job2 and Job3. It will remain in the queue until both preceding jobs complete successfully.

2.3. Results

For testing purposes, identical job scripts were created (represented here by `job1.sh`):

```
#!/bin/bash
#SBATCH --partition=nodo.q
#SBATCH --job-name=Job1

echo "Starting ${SLURM_JOB_NAME} (${SLURM_JOB_ID}) at $(date)"
sleep 10
echo "Finished ${SLURM_JOB_NAME} (${SLURM_JOB_ID}) at $(date)"
```

Upon execution, the Python script captured the following Job IDs:

```
Job1: 79950
Job2: 79951
Job3: 79952
Job4: 79953
```

Monitoring the queue with `squeue` immediately after submission showed the dependency system in action:

```
JOBID  PARTITION  NAME  USER    ST  TIME  NODES  NODELIST(REASON)
79953  nodo.q     Job4  biom-2  PD  0:00  1      (Dependency)
```

```
79952  nodo.q     Job3  biom-2  PD  0:00  1      (Dependency)
79951  nodo.q     Job2  biom-2  PD  0:00  1      (Dependency)
79950  nodo.q     Job1  biom-2   R  0:02  1      clus09
```

Finally, the output logs confirmed that the execution order strictly followed the defined workflow:

```
$ cat slurm-7995*.out
Starting Job1 (79950)
Finished Job1 (79950)
Starting Job2 (79951)
Starting Job3 (79952)
Finished Job2 (79951)
Finished Job3 (79952)
Starting Job4 (79953)
Finished Job4 (79953)
```

3. Restriction Enzyme Digestion (A)

[See Packages and Biopython Exercises]

Write a pure Python program that accepts a FASTA file and an enzyme recognition sequence, and computes how many restriction cuts the enzyme would produce across all sequences in the file.

3.1. Implementation

The solution is implemented entirely in Python without external bioinformatics libraries. FASTA sequences are read sequentially, concatenating multi-line sequence blocks while discarding header information.

```
import argparse


def read_fasta(filename: str) -> list[str]:
    """Read a FASTA file and return all sequences concatenated."""
    sequences = []
    current_seq = []

    with open(filename, "r") as f:
        for line in f:
            line = line.strip()
            if line.startswith(">"):
                if current_seq:
                    sequences.append("".join(current_seq))
                    current_seq = []
            else:
                current_seq.append(line)

        if current_seq:
            sequences.append("".join(current_seq))

    return sequences


def count_cuts(sequences, enzyme):
    """Count total occurrences of enzyme in all sequences."""
    total_cuts = 0
    enzyme_upper = enzyme.upper()

    for seq in sequences:
        seq_upper = seq.upper()
        start = 0
        while True:
            pos = seq_upper.find(enzyme_upper, start)
            if pos == -1:
                break
            total_cuts += 1
            start = pos + len(enzyme_upper)

    return total_cuts
```

```
def main():
    parser = argparse.ArgumentParser(
        description="Count enzyme cuts in FASTA sequences"
    )
    parser.add_argument("fasta_file", help="Input FASTA file")
    parser.add_argument("enzyme", help="Enzyme recognition sequence")

    args = parser.parse_args()

    sequences = read_fasta(args.fasta_file)
    cuts = count_cuts(sequences, args.enzyme)

    print(f"Total cuts by enzyme {args.enzyme}: {cuts}")


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

3.2. Notes

To handle command-line arguments, I used the `argparse` module. Although I am aware of `sys.argv`, I prefer `argparse` for these tasks as it provides a much more robust and cleaner interface, and it is a tool I have become comfortable with through previous projects.

Since the task only requires counting enzyme recognition sites and not tracking their position or associated sequence headers, header information is discarded during parsing. Occurrences of the enzyme recognition sequence are counted in a non-overlapping manner to ensure accuracy.

3.3. Results

The script was tested using the file `PM_50.fasta` as the input dataset:

```
$ time python digest_A.py ./PM_50.fasta GAATTC
Total cuts by enzyme GAATTC: 19

real    0m0.165s
user    0m0.031s
sys     0m0.030s
```

4. Restriction Enzyme Digestion (B)

[See Packages and Biopython Exercises]

Modify the previous program so that FASTA file parsing is performed exclusively using Biopython modules.

4.1. Implementation

In this version, the custom FASTA parser is replaced with Biopython's `SeqIO` module. This allows for a direct comparison between a minimal custom parser and a standardized library.

```
import argparse
from Bio import SeqIO


def read_fasta_biopython(filename: str) -> list[str]:
    """Read a FASTA file and return a list of sequences."""
    sequences = []
    for record in SeqIO.parse(filename, "fasta"):
        sequences.append(str(record.seq))
    return sequences


def count_cuts(sequences, enzyme):
    """Count total occurrences of enzyme in all sequences (non-overlapping)."""
    total_cuts = 0
    enzyme_upper = enzyme.upper()

    for seq in sequences:
        seq_upper = seq.upper()
        start = 0
        while True:
            pos = seq_upper.find(enzyme_upper, start)
            if pos == -1:
                break
            total_cuts += 1
            start = pos + len(enzyme_upper)

    return total_cuts


def main():
    parser = argparse.ArgumentParser(
        description="Count enzyme cuts in FASTA sequences"
    )
    parser.add_argument("fasta_file", help="Input FASTA file")
    parser.add_argument("enzyme", help="Enzyme recognition sequence")

    args = parser.parse_args()

    sequences = read_fasta_biopython(args.fasta_file)
    cuts = count_cuts(sequences, args.enzyme)
```

```
    print(f"Total cuts by enzyme {args.enzyme}: {cuts}")


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

4.2. Notes

Biopython's `SeqIO.parse` function is used to iterate over the FASTA records. Each sequence is extracted from the `SeqRecord` object and converted to a string.

4.2.1. Environment Setup Recommendation

To ensure the script runs correctly, Biopython must be installed. Although not strictly required by the exercise statement, I highly recommend using a Python virtual environment to isolate dependencies from the system-wide installation:

```
$ python -m venv .venv
$ source .venv/bin/activate
$ pip install biopython
```

4.3. Results

The script produced the same count as Part A:

```
$ time python digest_B.py PM_50.fasta GAATTC
Total cuts by enzyme GAATTC: 19

real    0m0.438s
user    0m0.015s
sys     0m0.060s
```

4.3.1. Performance Discussion

The Biopython implementation is approximately 0.3 seconds slower than the custom parser. This is expected, as `SeqIO.parse` performs validation, object creation, and metadata management, whereas the manual parser in Part A performs only basic string operations.