A Python Tool

as a FASTX files pre-processor

Rodrigo Navarro García-Ochoa

Muhammad Umer Hussain Kousar

TABLE OF CONTENTS

Introduction

Sequence data in bioinformatics are commonly stored and exchanged using the FASTA and FASTQ file formats, providing essential representations of nucleotide or protein sequences, and, in the case of FASTQ, associated quality scores. The preprocessing of such files—including trimming, adaptor removal, reverse complementing, and validation—is critically important step in many analysis pipelines. Python scripts such as `RUfastx_pp.py` aim to streamline these preprocessing operations.

This report presents a comprehensive, function-level documentation and practical guide to `RUfastx_pp.py` .

The report is divided into two main sections:

- Section 1 provides detailed documentation for each core function, including its purpose, parameters, and return values.
- Section 2 offers a tutorial-style analysis that explains the script's workflow, evaluates its robustness, and demonstrates how it aligns with best practices in FASTA/FASTQ processing and bioinformatics scripting.

Together, these sections serve as both a reference manual and a step-by-step guide for effectively using and understanding `RUfastx_pp.py` .

Typical preprocessing operations include:

- **Reverse complementation** of DNA strands (to align complementary sequences).
- **Trimming** to remove low-quality or extraneous bases from sequence ends.
- **Adaptor removal** to eliminate synthetic oligonucleotide sequences introduced during library preparation.
- **File validation** to ensure that the input data conforms to standard specifications.

Function Documentation

This section presents a full reference for each function implemented in the script `RUfastx_pp.py`. Each entry describes the function's purpose, parameters, return values, and its biological or computational significance.

Function: usage_instructions()

- Description

The *usage_instructions()* function is responsible for displaying a clear and structured guide on how to correctly execute the script from the command line. It provides users with the expected command-line syntax, a list of mandatory and optional arguments, and practical usage examples for each main operation supported by the program (reverse complement, trimming, adaptor removal).

This function is automatically triggered when the user provides insufficient or invalid command-line arguments. Its main goal is to prevent runtime errors by ensuring that users understand the expected input/output structure before execution.

- Parameters

None. The function does not take any external parameters; it operates solely by printing instructions and terminating program execution.

- Returns

None. The function calls `sys.exit(1)` to terminate the program immediately after printing the usage message.

- Side effects

- Prints help and usage information to stderr (standard error output).

- Terminates program execution, preventing subsequent code from running.

- Communicates user-facing information about required arguments and correct syntax.

- Context and purpose

In bioinformatics, scripts that process sequence data (FASTA/FASTQ) are often executed in automated workflows or pipelines. Incorrectly specified arguments could lead to overwriting important data or misprocessing sequences.

The *usage_instructions()* function acts as a safety mechanism, ensuring that the user provides all required information (`--input`, `--output`, `--operation`) and understands the available options before any data manipulation occurs.

Function: check_arguments()

- Description

The *check_arguments()* function is responsible for parsing and validating command-line arguments provided by the user. It ensures that all required flags are present (`--input` , `--output` , `--operation`), that each flag is used only once, and that every flag is followed by a valid value. Additionally, it verifies that the input and output files share the same extension, which is critical for consistent format handling.

This function is essential for maintaining the integrity of the script's execution. By enforcing strict argument validation, it prevents misconfiguration, unexpected behavior, and potential data loss due to incorrect file handling or unsupported operations.

- Parameters

None. The function operates directly on `sys.argv` , which contains the list of command-line arguments passed to the script.

- Returns

A tuple containing six elements:

- `input_file` (str): Path to the input file.

- `output_file` (str): Path to the output file.

- `operation` (str): Type of operation to perform (rc, trim, or adaptor-removal).

- `ltrim_len` (int): Number of bases to trim from the left (only used in trim mode).

- `rtrim_len` (int): Number of bases to trim from the right (only used in trim mode).

- `adaptor` (str): Adaptor sequence to remove (only used in adaptor-removal mode).

- Side effects

- Prints error messages to the console if any validation fails.

- Calls *usage_instructions()* and terminates the program via `sys.exit(1)` when invalid arguments are detected.

- Ensures that the script does not proceed with incomplete or incorrect input.

- Context and purpose

The *check_arguments()* function acts as a gatekeeper, ensuring that the user provides all necessary information in the correct format before any processing begins.

By enforcing argument structure and validating file extensions, it guarantees that the script operates on compatible data and performs the intended operation safely.

Function: check_file(in_file, out_file)

- Description

The *check_file()* function is responsible for validating the input and output file paths provided by the user. It performs several critical checks to ensure that the files are accessible, correctly formatted, and safe to use. Specifically, it verifies:

- That the input file exists and is readable.

- That the output file does not overwrite the input file.

- That the output file is writable or prompts the user for confirmation if it already exists.

- That both files share the same extension (.fasta or .fastq), which is essential for consistent format handling.

- Additionally, the function determines the file format based on the extension and returns it to guide downstream processing.

- Parameters

- `in_file` (str): Path to the input file. Must end in .fasta or .fastq.

- `out_file` (str): Path to the output file. Must have the same extension as in_file.

- Returns

- `file_format` (str): A string indicating the format of the input file. Possible values are:

    - 'fasta'

    - 'fastq'

This return value is used to route the input through the appropriate processing function.

- Side Effects

- Prints error messages to the console if any validation fails.

- Prompts the user for confirmation if the output file already exists.

- Terminates the program via `sys.exit(1)` if:

- The input file does not exist.

- The input and output files have mismatched extensions.

- The input file is not readable, or the output file is not writable

- Context and purpose

In bioinformatics workflows, file integrity and format consistency are paramount. Processing a FASTA file as if it were FASTQ (or vice versa) can lead to corrupted data and invalid results. The *check_file()* function acts as a safeguard against such errors by enforcing strict validation rules.

Moreover, overwriting input files or writing to inaccessible paths can disrupt entire pipelines.

Function: dot_number(number)

- Description

The *dot_number()* function is a utility designed to improve the readability of large integers by formatting them with dot separators. This is particularly useful when displaying biological metrics such as the number of reads or bases processed, which often reach into the millions.

Instead of returning raw numeric values (e.g., 1000000), the function transforms them into a more human-friendly format (1.000.000). This enhances clarity in summary reports and helps users quickly interpret scale and magnitude.

- Parameters

- `number` (int): A positive integer representing a count or metric to be formatted. Typically used for read counts, base counts, or trimmed base totals.

- Returns

- (str): A string representation of the input number with dot separators inserted every three digits from the right. Example: dot_number(1234567) → '1.234.567'

- Side effects

None. The function is pure and does not modify external state or perform any I/O operations.

- Context and purpose

In bioinformatics reporting, clarity and precision are essential. When presenting results to users or researchers, especially in summary tables or terminal output, large numbers can become visually overwhelming. The *dot_number()* function addresses this by formatting numerical output in a way that aligns with European conventions, where dots are used as thousand separators.

This small but impactful enhancement improves user experience and ensures that key metrics are immediately understandable, even in high-throughput sequencing contexts where millions of reads or bases are common.

Function: base_percentages(base_counts, bases_processed)

- Description

The *base_percentages()* function calculates the relative frequency of each nucleotide base (A, C, G, T, N) in a given set of DNA or RNA sequences. It takes as input a dictionary of base counts and the total number of bases processed and returns the percentage representation of each base.

This function is typically used at the end of a processing operation (reverse complement, trimming, adaptor removal) to summarize the composition of the dataset. It helps users assess the quality and characteristics of the sequences, detect potential biases, and validate that the operation preserved biological integrity.

- Parameters

- `base_counts` (dict): A dictionary containing the count of each base. Expected keys: 'A', 'C', 'G', 'T', 'N'.

- `bases_processed` (int): The total number of bases analyzed across all reads.

- Returns

- (tuple of int): A five-element tuple representing the percentage of each base, rounded to the nearest integer: (`per_a` , `per_c` , `per_g` , `per_t` , `per_n`), where each value corresponds to the percentage of its respective base relative to bases_processed.

- Side effects

None. The function performs a pure calculation and does not modify external state or perform I/O operations.

- Context and purpose

In bioinformatics, base composition analysis is a standard quality control step. For example, an overrepresentation of 'N' bases may indicate low-quality reads or unresolved base calls.

Function: rc_fastq(in_file, out_file)

- Description

The *rc_fastq()* function performs a reverse complement operation on each sequence in a FASTQ file. In addition to reversing and complementing the nucleotide sequence, it also reverses the corresponding quality score line to maintain positional accuracy. This ensures that the quality scores remain aligned with the correct bases after transformation.

The function reads the input file line by line in blocks of four (identifier, sequence, separator, quality), applies the reverse complement transformation to the sequence, reverses the quality string, and writes the modified read to the output file. It also tracks base composition statistics for reporting purposes.

- Parameters

- `in_file` (str): Path to the input FASTQ file containing raw sequencing reads.

- `out_file` (str): Path to the output FASTQ file where the reverse complemented reads will be saved

- Returns

- (tuple): A three-element tuple containing:

    - `reads_processed` (int): Total number of reads processed.
    - `bases_processed` (int): Total number of bases across all reads.
    - `base_counts` (dict): Dictionary with counts of each base (A, C, G, T, N).

- Side effects

- Reads and writes to disk.

- Modifies sequence and quality lines.

- Updates base composition statistics.

- Context and purpose

Reverse complementing is a common operation in bioinformatics, especially when working with paired-end reads or aligning sequences to reference genomes. FASTQ files contain both sequence and quality information, so any transformation must preserve the integrity of both.

The function ensures that the reverse complement is applied correctly and that quality scores are accurately repositioned.

By integrating base counting and summary statistics, the function also provides useful feedback on the dataset’s composition, helping users verify that the operation was performed correctly and that the data remains biologically meaningful.

Function: rc_fasta(in_file, out_file)

- Description

The *rc_fasta()* function performs a reverse complement transformation on each sequence in a FASTA file. Unlike FASTQ, FASTA files do not contain quality scores, so the function focuses solely on reversing the nucleotide sequence and replacing each base with its complement (A↔T, C↔G, N↔N).

The function reads the input file line by line, identifies header lines (starting with >), and applies the reverse complement operation to the sequence lines. It writes the transformed sequences to the output file while preserving the original headers.

- Parameters

- `in_file` (str): Path to the input FASTA file containing raw nucleotide sequences.

- `out_file` (str): Path to the output FASTA file where the reverse complemented sequences will be saved.

- Returns

- (tuple): A three-element tuple containing:

    - `reads_processed` (int): Total number of sequences processed.
    - `bases_processed` (int): Total number of bases across all sequences.
    - `base_counts` (dict): Dictionary with counts of each base (A, C, G, T, N).

- Side effects

- Reads and writes to disk.

- Transforms sequence content.

- Updates base composition statistics.

- Context and purpose

Reverse complementing is a fundamental operation in sequence analysis, especially when aligning reads to reference genomes or analyzing antisense strands. In the context of FASTA files, which are often used for reference sequences or assembled contigs, this operation allows researchers to explore the complementary strand of DNA.

The *rc_fasta()* function ensures that the transformation is applied accurately and efficiently, preserving the structure of the FASTA format. It also collects base composition statistics, which are useful for validating the transformation and understanding the nucleotide distribution of the dataset.

Function: trim_fastq(in_file, out_file, left, right

- Description

The *trim_fastq()* function performs hard trimming on sequencing reads in a FASTQ file. It removes a specified number of bases from both the left and right ends of each read, and applies the same trimming to the corresponding quality score lines to maintain alignment.

This operation is commonly used to eliminate low-quality bases at the ends of reads, which can interfere with downstream analyses such as alignment, variant calling, or assembly. The function ensures that only reads with sufficient length after trimming are retained. It also tracks and reports statistics on base composition before trimming, as well as the total number of bases removed and their nucleotide distribution.

- Parameters

- `in_file` (str): Path to the input FASTQ file containing raw sequencing reads.

- `out_file` (str): Path to the output FASTQ file where trimmed reads will be saved.

- `left` (int): Number of bases to trim from the 5′ (left) end of each read.

- `right` (int): Number of bases to trim from the 3′ (right) end of each read.

- Returns

- (tuple): A five-element tuple containing:

    - `reads_processed` (int): Total number of reads processed.

    - `bases_processed` (int): Total number of bases after trimming.

    - `base_counts` (dict): Dictionary with counts of each base (A, C, G, T, N) after trimming.

    - `total_trimmed_bases` (int): Total number of bases removed across all reads.

    - `tbase_counts` (dict): Dictionary with counts of each base removed during trimming.

- Side Effects

- Reads and writes to disk.

- Modifies both sequence and quality lines.

- Skips reads that become empty or too short after trimming.

- Collects and returns detailed trimming statistics.

- Context and purpose

In high-throughput sequencing, the ends of reads often contain low-quality or adapter-contaminated bases. Trimming these regions improves the accuracy of downstream analyses and reduces noise in alignment and variant detection.

The *trim_fastq()* function provides a precise and reproducible way to remove unwanted bases from both ends of each read. By applying the same trimming to quality scores, it preserves the integrity of the FASTQ format. The function also generates detailed statistics, allowing users to assess the impact of trimming on their dataset and verify that the operation was performed as intended.

Function: trim_fasta(in_file, out_file, left, right)

- Description

The *trim_fasta()* function performs hard trimming on nucleotide sequences stored in a FASTA file. It removes a specified number of bases from both the 5′ (left) and 3′ (right) ends of each sequence. Unlike FASTQ files, FASTA files do not contain quality scores, so the trimming operation applies only to the sequence lines.

The function reads the input file line by line, identifies header lines (starting with >), and applies trimming to the corresponding sequence lines. It also tracks base composition statistics before and after trimming, as well as the total number of bases removed.

- Parameters

- `in_file` (str): Path to the input FASTA file containing raw sequencing reads.

- `out_file` (str): Path to the output FASTA file where trimmed reads will be saved.

- `left` (int): Number of bases to trim from the 5′ (left) end of each read.

- `right` (int): Number of bases to trim from the 3′ (right) end of each read.

- Returns

- (tuple): A five-element tuple containing:

  - `reads_processed` (int): Total number of reads processed.
  - `bases_processed` (int): Total number of bases after trimming.
  - `base_counts` (dict): Dictionary with counts of each base (A, C, G, T, N) after trimming.
  - `total_trimmed_bases` (int): Total number of bases removed across all reads.
  - `tbase_counts` (dict): Dictionary with counts of each base removed during trimming.

- Side effects

- Reads and writes to disk.

- Modifies sequence content.

- Skips sequences that become empty or too short after trimming.

- Collects and returns detailed trimming statistics.

- Context and purpose

Trimming is a common preprocessing step in bioinformatics pipelines, especially when working with assembled contigs or reference sequences in FASTA format. It helps remove low-quality or ambiguous regions from the ends of sequences, which can interfere with downstream analyses such as alignment, annotation, or motif discovery.

Function: adapt_removal_fastq(in_file, out_file, adaptor)

- Description

The *adapt_removal_fastq()* function removes a specified adaptor sequence from the beginning of each read in a FASTQ file. If the adaptor is detected at the start of a read, it is removed along with the corresponding number of characters from the quality score line to preserve positional alignment.

This operation is essential in preprocessing raw sequencing data, where adaptor contamination can interfere with downstream analyses such as alignment, assembly, or variant calling. The function ensures that only clean, biologically relevant sequences are retained. It also tracks the number of adaptors removed and updates base composition statistics for the cleaned dataset.

- Parameters

- `in_file` (str): Path to the input FASTQ file containing raw sequencing reads.

- `out_file` (str): Path to the output FASTQ file where adaptor-free reads will be saved.

- `adaptor` (str): The adaptor sequence to be removed from the beginning of each read.

- Returns

- (tuple): A four-element tuple containing:

  - `reads_processed` (int): Total number of reads processed.

  - `bases_processed` (int): Total number of bases after adaptor removal.

  - `base_counts` (dict): Dictionary with counts of each base (A, C, G, T, N) after cleaning.

  - `adaptors_removed` (int): Number of reads from which the adaptor was successfully removed.

- Side effects

- Reads and writes to disk.

- Modifies sequence and quality lines.

- Skips reads that become empty or too short after adaptor removal.

- Collects and returns adaptor removal statistics.

- Context and purpose

Adaptor sequences are synthetic oligonucleotides added during library preparation for sequencing. If not properly removed, they can cause misalignment, false variant calls, or poor assembly results. In FASTQ files, removing the adaptor also requires trimming the quality score line to maintain correct base-to-score mapping.

Function: adapt_removal_fasta(in_file, out_file, adaptor)

- Description

The *adapt_removal_fasta()* function removes a specified adaptor sequence from the beginning of each sequence in a FASTA file. If the adaptor is detected at the start of a sequence, it is removed. Unlike FASTQ files, FASTA files do not contain quality scores, so the operation focuses solely on the nucleotide sequence.

The function reads the input file line by line, identifies header lines (starting with >), and applies adaptor removal to the corresponding sequence lines. It also tracks how many adaptors were removed and updates base composition statistics for the cleaned dataset.

- Parameters

- `in_file` (str): Path to the input FASTQ file containing raw sequencing reads.

- `out_file` (str): Path to the output FASTQ file where adaptor-free reads will be saved.

- `adaptor` (str): The adaptor sequence to be removed from the beginning of each read.

- Returns

- (tuple): A four-element tuple containing:

    - `reads_processed` (int): Total number of reads processed.

    - `bases_processed` (int): Total number of bases after adaptor removal.

    - `base_counts` (dict): Dictionary with counts of each base (A, C, G, T, N) after cleaning.

    - `adaptors_removed` (int): Number of reads from which the adaptor was successfully removed.

- Side effects

- Reads and writes to disk.

- Modifies sequence content.

- Skips sequences that become empty or too short after adaptor removal.

- Collects and returns adaptor removal statistics.

- Context and purpose

Adaptor sequences are synthetic fragments added during library preparation for sequencing. If not properly removed, they can interfere with downstream analyses such as alignment, annotation, or motif discovery. In FASTA files, adaptor contamination can distort biological interpretation and reduce data quality.

Function: main()

- Description

The *main()* function serves as the central execution point of the script. It orchestrates the entire workflow by coordinating argument parsing, file validation, operation dispatching, and summary reporting. Upon execution, it performs the following steps:

1. Calls *check_arguments()* to validate and extract command-line parameters.
2. Calls *check_file()* to verify input/output file integrity and determine format (FASTA or FASTQ).
3. Based on the selected operation (rc, trim, or adaptor-removal), it routes the input through the appropriate processing function.
4. Collects and prints summary statistics, including read counts, base composition, and trimming/adaptor metrics.

- Parameters

None. The function does not accept external parameters. It operates entirely based on `sys.argv` and internal function calls.

- Returns

None. The function does not return a value. It performs all operations through side effects (file I/O, console output, and function calls).

- Side effects

- Reads and writes to disk.

- Prints summary statistics to the console.

- Terminates execution on error via `sys.exit(1)`.

- Dispatches processing functions based on user input.

- Context and purpose

In command-line bioinformatics tools, the *main()* function acts as the control hub. It ensures that all components of the script are executed in the correct order and that the user receives meaningful feedback.

How to use the tool

Purpose

This script preprocesses DNA sequences stored in FASTA or FASTQ files, performing three essential bioinformatics operations: reverse-complement calculation, sequence trimming, and adaptor removal. It automatically detects the input file format and maintains consistency throughout the processing pipeline.

Key Features

1. Automatic Format Detection: Recognizes and processes both FASTA and FASTQ file formats
2. Reverse-Complement Operation: Computes the reverse complement of DNA sequences (quality scores are reversed for FASTQ files)
3. Sequence Trimming: Removes specified numbers of bases from either end of sequences
4. Adaptor Removal: Identifies and removes adaptor sequences from the beginning of reads
5. Statistical Summary: Provides detailed statistics including total reads processed, base composition, and operation-specific metrics
6. Flexible Argument Order: Accepts command-line arguments in any order for user convenience

Requirements & Setup

- Prerequisites

- Python 3.10 or later version must be installed on your system

- Input File Format

The script accepts only two types of sequence files:

- *FASTA (.fasta)*

```
>Sequence_1   # header of read 1

ATCGATCGATCG  # sequence of read 1

>Sequence_2   # header of read 2

GCTAGCTAGCTA  # sequence of read 2
```

- *FASTQ (.fastq)*

```
@Sequence_1   # header of read 1

ATCGATCGATCG  # sequence of read 1
```

```
+             # “+” line
IIIIIIIIIIII  # quality of read 1
@Sequence_2   # header of read 2
GCTAGCTAGCTA  # sequence of read 2
+             # “+” line
JJJJJJJJJJJJ  # quality of read 2
```

- Important Notes:

- Input files must have either `.fasta` or `.fastq` extension

- The Output file must have the same extension as the input file

- Input Files are assumed to be properly formatted

- Multi-FASTA files are not supported and will produce incorrect results

Running the Script

- Basic Command Structure

All operations must follow this general syntax:

```
python3 RUfastx_pp.py --input <input_file> --output <output_file> --operation <operation> [additional options]
```

- Command Line Arguments

Required Arguments:

`--input <filename>` : Specifies the input FASTA/FASTQ file

`--output <filename>` : Specifies the output file name

`--operation <operation>` : Specifies the preprocessing operation

- Valid operations are: `rc` , `trim` , `adaptor-removal` .

Optional Arguments (operation-specific):

`rc` : Does not require any additional arguments

`--trim-left <number>` : Number of bases to trim from the left end (Required for `trim` operation)

`--trim-right <number>` : Number of bases to trim from the right end (Required for `trim` operation)

- Number must be a positive value (>1), and at least one optional argument (either left or right) must be provided

`--adaptor <sequence>` : Adaptor sequence to remove (Required for `adaptor-removal` operation)

- Must contain only A, T, C, G characters (i.e. DNA bases)

Flexible Argument Order:

Arguments can be provided in any order, but each must be immediately followed by its < value >:

```
python3 RUfastx_pp.py --input test.fastq --output result.fastq --operation rc
python3 RUfastx_pp.py --operation rc --output result.fastq --input test.fastq
python3 RUfastx_pp.py --output result.fastq --operation rc --input test.fastq
```

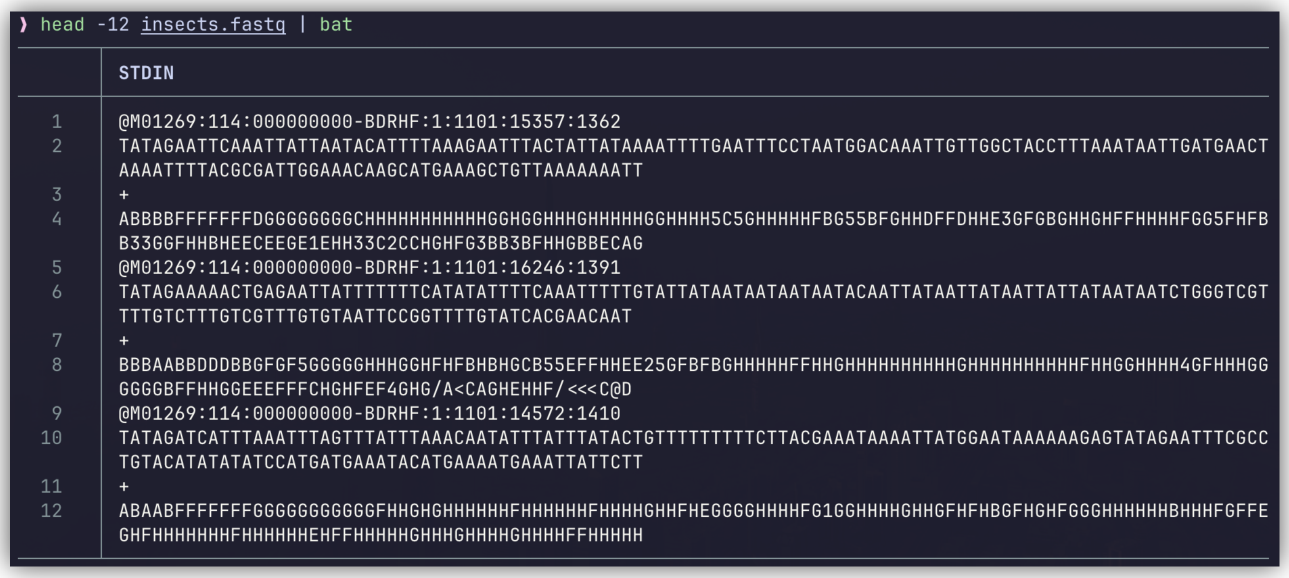
- Examples of usage

To demonstrate how the tool works, we'll present three examples — one for each operation:

Example 1. Reverse-Complement Operation

Scenario: you have a FASTQ file `insects.fastq` with sequencing reads that need to be reverse-complemented for downstream analysis. *Note: the same logic explained below would apply to FASTA files.*

The first 3 reads on the file look like this:

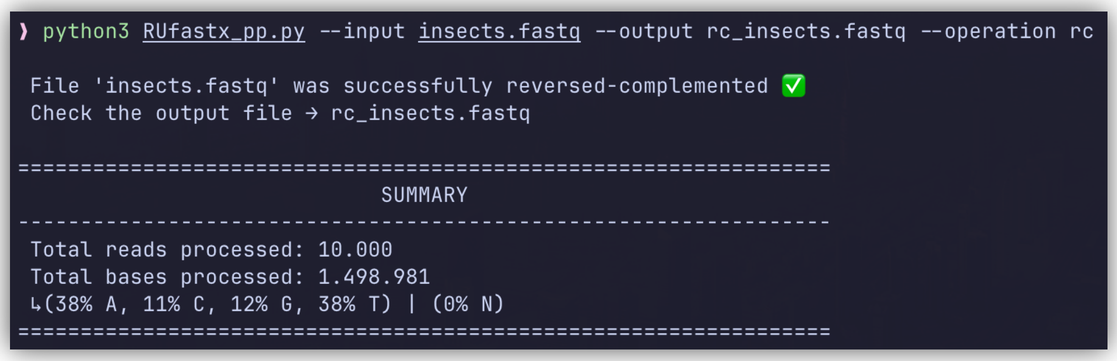


```
❯ head -12 insects.fastq | bat
STDIN
@M01269:114:000000000-BDRHF:1:1101:15357:1362
TATAGAATTCAAATTATTAATACATTTTAAAGAATTTACTATTATAAAATTTTGAATTTCCTAATGGACAAATTGTTGGCTACCTTTAAATAATTGATGAACT
AAAATTTTACGCGATTGGAAACAAGCATGAAAGCTGTTAAAAAAATT
+
ABBBBFFFFFFFDGGGGGGGGCHHHHHHHHHHHGGHGGHHHGHHHHHGGHHHH5C5GHHHHHFBG55BFGHHDFFDHHE3GFGBGHHGHFFHHHHFGG5FHFB
B33GGFHHBHEECEEGE1EHH33C2CCHGHFG3BB3BFHHGBBECAG
@M01269:114:000000000-BDRHF:1:1101:16246:1391
TATAGAAAAACTGAGAATTATTTTTTTCATATATTTTCAAATTTTTGTATTATAATAATAATAATACAATTATAATTATAATTATTATAATAATCTGGGTCGT
TTTGTCTTTGTCGTTTGTGTAATTCCGGTTTTGTATCACGAACAAT
+
BBBAABBDDDBBGFGF5GGGGGHHHGGHFHFBHBHGCB55EFFHHEE25GFBFBGHHHHHFFHHGHHHHHHHHHHGHHHHHHHHHHFHHGGHHHH4GFHHHGG
GGGGBFFHHGGEEEFFFCHGHFEF4GHG/A<CAGHEHHF/<<<C@D
@M01269:114:000000000-BDRHF:1:1101:14572:1410
TATAGATCATTTAAATTTAGTTTATTTAAACAATATTTATTTATACTGTTTTTTTTTCTTACGAAATAAAATTATGGAATAAAAAAGAGTATAGAATTTCGCC
TGTACATATATATCCATGATGAAATACATGAAAATGAAATTATTCTT
+
ABAABFFFFFFFGGGGGGGGGGGFHHGHGHHHHHHFHHHHHHFHHHHGHHFHEGGGGHHHHFG1GGHHHHGHHGFHFHBGFHGHFGGGHHHHHHBHHHFGFFE
GHFHHHHHHHHFHHHHHHHEHFFHHHHHGHHHGHHHHGHHHHFFHHHHH
```

We run the following command:

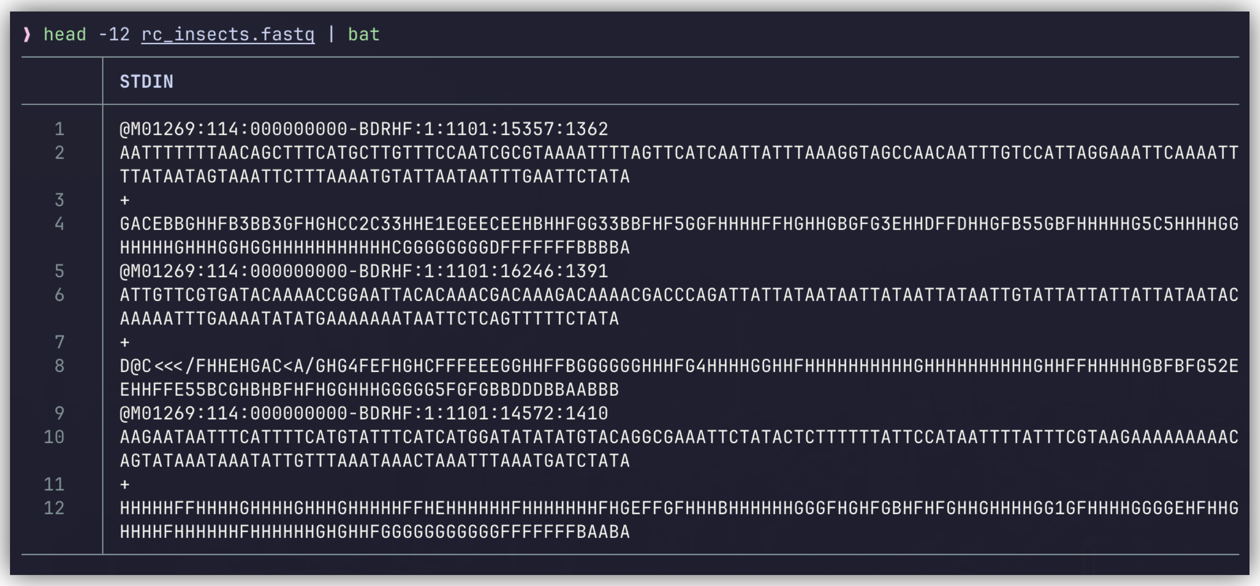
```
python3 RUfastx_pp.py --input insects.fastq --output rc_insects.fastq --operation rc
```

The terminal outputs the following summary:



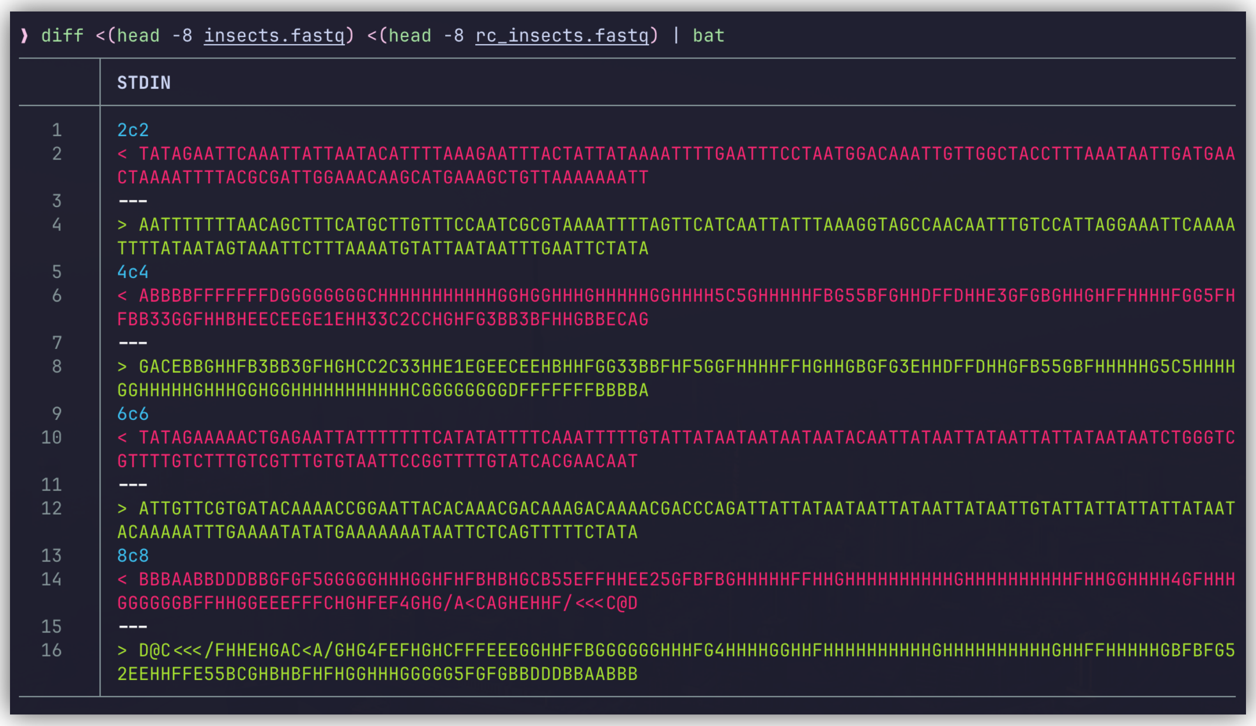
```
❯ python3 RUfastx_pp.py --input insects.fastq --output rc_insects.fastq --operation rc

 File 'insects.fastq' was successfully reversed-complemented ✅
 Check the output file → rc_insects.fastq

=================================================================
                             SUMMARY
-----------------------------------------------------------------
 Total reads processed: 10.000
 Total bases processed: 1.498.981
 ↳(38% A, 11% C, 12% G, 38% T) | (0% N)
=================================================================
```

If we now have a look at the first three reads in the output file `rc_insects.fastq` , we can see that the reads have been correctly reverse-complemented, and the quality scores have been reversed as well (i.e. each base retains its original quality score):



```
❯ head -12 rc_insects.fastq | bat
STDIN
@M01269:114:000000000-BDRHF:1:1101:15357:1362
AATTTTTTTAACAGCTTTCATGCTTGTTTCCAATCGCGTAAAATTTTAGTTCATCAATTATTTAAAGGTAGCCAACAATTTGTCCATTAGGAAATTCAAAATT
TTATAATAGTAAATTCTTTAAAATGTATTAATAATTTGAATTCTATA
+
GACEBBGHHFB3BB3GFHGHCC2C33HHE1EGEECEEHBHHFGG33BBFHF5GGFHHHHFFHGHHGBGFG3EHHDFFDHHGFB55GBFHHHHHG5C5HHHHGG
HHHHHGHHHGGHGGHHHHHHHHHHHCGGGGGGGGDFFFFFFFBBBBA
@M01269:114:000000000-BDRHF:1:1101:16246:1391
ATTGTTCGTGATACAAAACCGGAATTACACAAACGACAAAGACAAAACGACCCAGATTATTATAATAATTATAATTATAATTGTATTATTATTATTATAATAC
AAAAATTTGAAAATATATGAAAAAAATAATTCTCAGTTTTTCTATA
+
D@C<<</FHHEHGAC<A/GHG4FEFHGHCFFFEEEGGHHFFBGGGGGGHHHFG4HHHHGGHHFHHHHHHHHHHGHHHHHHHHHHGHHFFHHHHHGBFBFG52E
EHHFFE55BCGHBHBFHFHGGHHHGGGGG5FGFGBBDDDBBAABBB
@M01269:114:000000000-BDRHF:1:1101:14572:1410
AAGAATAATTTCATTTTCATGTATTTCATCATGGATATATATGTACAGGCGAAATTCTATACTCTTTTTTATTCCATAATTTTATTTCGTAAGAAAAAAAAAC
AGTATAAATAAATATTGTTTAAATAAACTAAATTTAAATGATCTATA
+
HHHHHFFHHHHGHHHHGHHHGHHHHHHFFHEHHHHHHFHHHHHHHFHGEFFGFHHHBHHHHHHGGGFHGHFGBHFHFGHHGHHHHGG1GFHHHHGGGGEHFHHG
HHHHFHHHHHHFHHHHHHGHGHHFGGGGGGGGGGGFFFFFFFBAABA
```

When we compare the first two reads side by side (red for the input file and green for the output file), this is very clear (*ignore the >,< signs*):

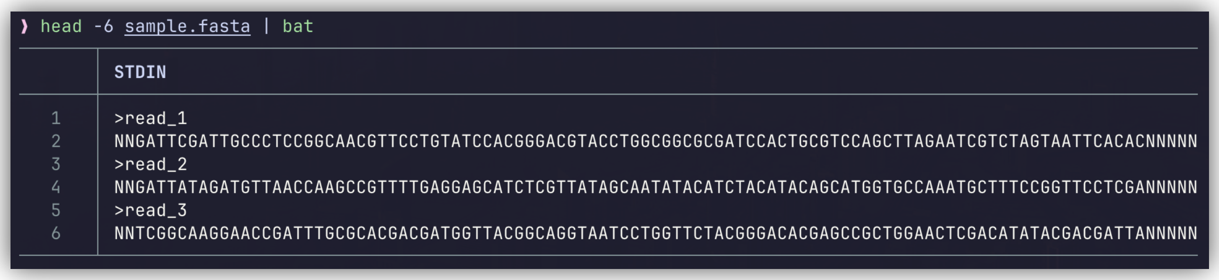


```
❯ diff <(head -8 insects.fastq) <(head -8 rc_insects.fastq) | bat
STDIN
2c2
< TATAGAATTCAAATTATTAATACATTTTAAAGAATTTACTATTATAAAATTTTGAATTTCCTAATGGACAAATTGTTGGCTACCTTTAAATAATTGATGAA
CTAAAATTTTACGCGATTGGAAACAAGCATGAAAGCTGTTAAAAAAATT
---
> AATTTTTTTAACAGCTTTCATGCTTGTTTCCAATCGCGTAAAATTTTAGTTCATCAATTATTTAAAGGTAGCCAACAATTTGTCCATTAGGAAATTCAAAA
TTTTATAATAGTAAATTCTTTAAAATGTATTAATAATTTGAATTCTATA
4c4
< ABBBBFFFFFFFDGGGGGGGGCHHHHHHHHHHHGGHGGHHHGHHHHHGGHHHH5C5GHHHHHFBG55BFGHHDFFDHHE3GFGBGHHGHFFHHHHFGG5FH
FBB33GGFHHBHEECEEGE1EHH33C2CCHGHFG3BB3BFHHGBBECAG
---
> GACEBBGHHFB3BB3GFHGHCC2C33HHE1EGEECEEHBHHFGG33BBFHF5GGFHHHHFFHGHHGBGFG3EHHDFFDHHGFB55GBFHHHHHG5C5HHHH
GGHHHHHGHHHGGHGGHHHHHHHHHHHCGGGGGGGGDFFFFFFFBBBBA
6c6
< TATAGAAAAACTGAGAATTATTTTTTTCATATATTTTCAAATTTTTGTATTATAATAATAATAATACAATTATAATTATAATTATTATAATAATCTGGGTC
GTTTTGTCTTTGTCGTTTGTGTAATTCCGGTTTTGTATCACGAACAAT
---
> ATTGTTCGTGATACAAAACCGGAATTACACAAACGACAAAGACAAAACGACCCAGATTATTATAATAATTATAATTATAATTGTATTATTATTATTATAAT
ACAAAAATTTGAAAATATATGAAAAAAATAATTCTCAGTTTTTCTATA
8c8
< BBBAABBDDDBBGFGF5GGGGGHHHGGHFHFBHBHGCB55EFFHHEE25GFBFBGHHHHHFFHHGHHHHHHHHHHGHHHHHHHHHFHHGGHHHH4GFHHH
GGGGGGBFFHHGGEEEFFFCHGHFEF4GHG/A<CAGHEHHF/<<<C@D
---
> D@C<<</FHHEHGAC<A/GHG4FEFHGHCFFFEEEGGHHFFBGGGGGGHHHFG4HHHHGGHHFHHHHHHHHHHGHHHHHHHHHHGHHFFHHHHHGBFBFG5
2EEHHFFE55BCGHBHBFHFHGGHHHGGGGG5FGFGBBDDDBBAABBB
```

Example 2. Trimming Operation

Scenario: your FASTA sequencing reads contain low-quality bases at both ends that need to be removed. *Note: the same logic explained below would apply to FASTQ files*.

The first 3 reads on the file look like this:

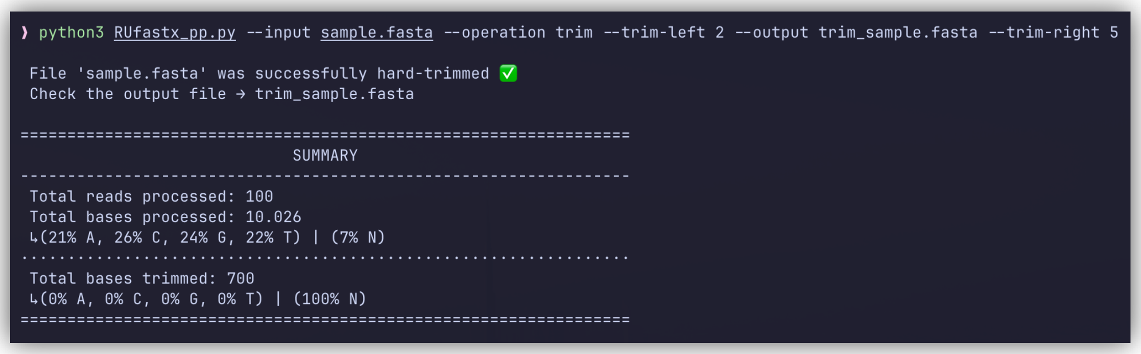


```
❱ head -6 sample.fasta | bat

STDIN

>read_1
NNGATTCGATTGCCCTCCGGCAACGTTCCTGTATCCACGGGACGTACCTGGCGGCGCGATCCACTGCGTCCAGCTTAGAATCGTCTAGTAATTCACACNNNNN
>read_2
NNGATTATAGATGTTAACCAAGCCGTTTTGAGGAGCATCTCGTTATAGCAATATACATCTACATACAGCATGGTGCCAAATGCTTTCCGGTTCCTCGANNNNN
>read_3
NNTCGGCAAGGAACCGATTTGCGCACGACGATGGTTACGGCAGGTAATCCTGGTTCTACGGGACACGAGCCGCTGGAACTCGACATATACGACGATTANNNNN
```

We run the following command to trim 2 bases from the left and 5 bases from the right:

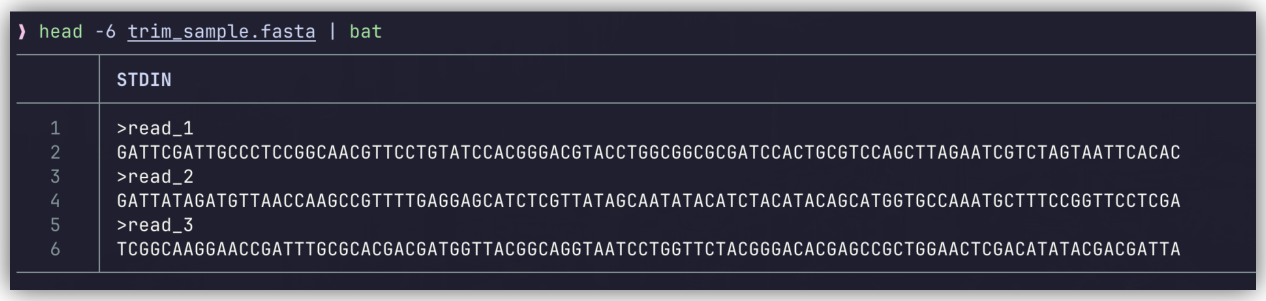
```
python3 RUfastx_pp.py --input sample.fasta --operation trim --trim-left 2 \
                      --output trim_sample.fasta --trim-right 5
```

The terminal outputs the following summary:



```
❱ python3 RUfastx_pp.py --input sample.fasta --operation trim --trim-left 2 --output trim_sample.fasta --trim-right 5

 File 'sample.fasta' was successfully hard-trimmed ✅
 Check the output file → trim_sample.fasta

=================================================================
                             SUMMARY
-----------------------------------------------------------------
 Total reads processed: 100
 Total bases processed: 10.026
 ↳(21% A, 26% C, 24% G, 22% T) | (7% N)
.................................................................
 Total bases trimmed: 700
 ↳(0% A, 0% C, 0% G, 0% T) | (100% N)
=================================================================
```

If we now have a look at the first three reads in the output file `trim_sample.fasta` , we can see that the reads have been correctly trimmed:

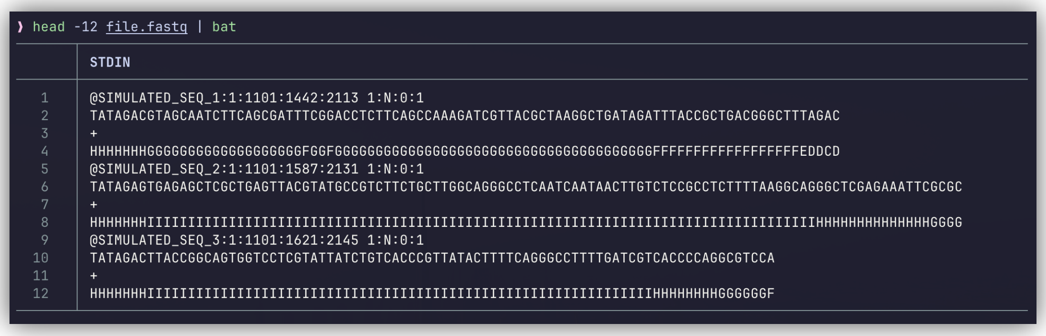


```
❱ head -6 trim_sample.fasta | bat

STDIN

>read_1
GATTCGATTGCCCTCCGGCAACGTTCCTGTATCCACGGGACGTACCTGGCGGCGCGATCCACTGCGTCCAGCTTAGAATCGTCTAGTAATTCACAC
>read_2
GATTATAGATGTTAACCAAGCCGTTTTGAGGAGCATCTCGTTATAGCAATATACATCTACATACAGCATGGTGCCAAATGCTTTCCGGTTCCTCGA
>read_3
TCGGCAAGGAACCGATTTGCGCACGACGATGGTTACGGCAGGTAATCCTGGTTCTACGGGACACGAGCCGCTGGAACTCGACATATACGACGATTA
```

Example 3. Adaptor Removal Operation

Scenario: you have a FASTQ file with sequencing reads that contain an adaptor sequence "TATAGA" at the beginning, which needs to be removed. *Note: the same logic explained below would apply to FASTA files*.

The first three reads on the file look like this:

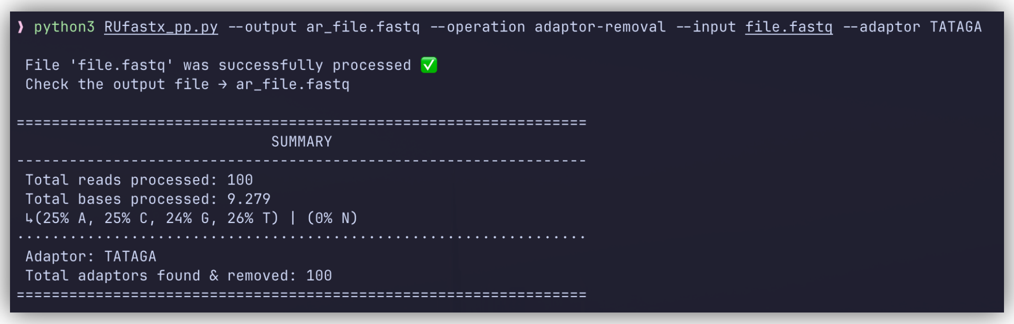


```
❱ head -12 file.fastq | bat

STDIN

@SIMULATED_SEQ_1:1:1101:1442:2113 1:N:0:1
TATAGACGTAGCAATCTTCAGCGATTTCGGACCTCTTCAGCCAAAGATCGTTACGCTAAGGCTGATAGATTTACCGCTGACGGGCTTTAGAC
+
HHHHHHHGGGGGGGGGGGGGGGGGGGGFGGFGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGFFFFFFFFFFFFFFFFFFEDDCD
@SIMULATED_SEQ_2:1:1101:1587:2131 1:N:0:1
TATAGAGTGAGAGCTCGCTGAGTTACGTATGCCGTCTTCTGCTTGGCAGGGCCTCAATCAATAACTTGTCTCCGCCTCTTTTAAGGCAGGGCTCGAGAAATTCGCGC
+
HHHHHHHIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIHHHHHHHHHHHHHHGGGG
@SIMULATED_SEQ_3:1:1101:1621:2145 1:N:0:1
TATAGACTTACCGGCAGTGGTCCTCGTATTATCTGTCACCCGTTATACTTTTCAGGGCCTTTTGATCGTCACCCCAGGCGTCCA
+
HHHHHHHIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIHHHHHHHHGGGGGGF
```

We run the following command:

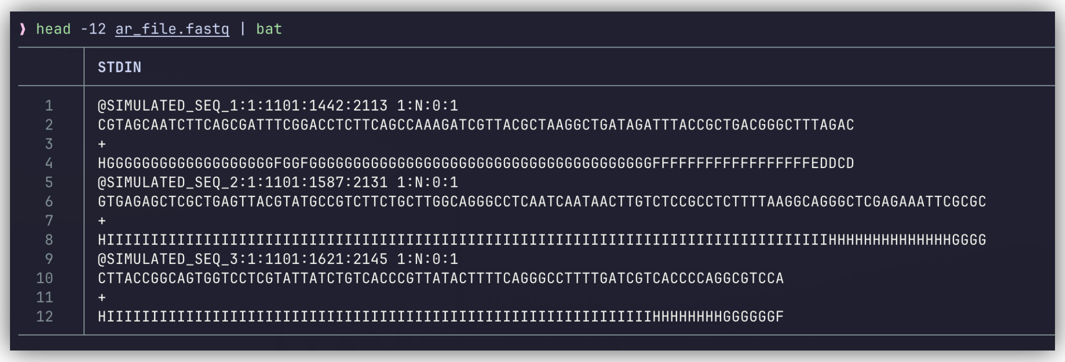
```
python3 RUfastx_pp.py --output ar_file.fastq --operation adaptor-removal \
                      --input file.fastq --adaptor TATAGA
```

The terminal outputs the following summary:



```
❱ python3 RUfastx_pp.py --output ar_file.fastq --operation adaptor-removal --input file.fastq --adaptor TATAGA

 File 'file.fastq' was successfully processed ✅
 Check the output file → ar_file.fastq

=================================================================
                             SUMMARY
-----------------------------------------------------------------
 Total reads processed: 100
 Total bases processed: 9.279
 ↳(25% A, 25% C, 24% G, 26% T) | (0% N)
.................................................................
 Adaptor: TATAGA
 Total adaptors found & removed: 100
=================================================================
```

If we now have a look at the first three reads in the output file `ar_file.fastq`, we can see that the adaptor has been correctly removed from the reads, as well as the quality scores for those bases:



```
❱ head -12 ar_file.fastq | bat

STDIN

@SIMULATED_SEQ_1:1:1101:1442:2113 1:N:0:1
CGTAGCAATCTTCAGCGATTTCGGACCTCTTCAGCCAAAGATCGTTACGCTAAGGCTGATAGATTTACCGCTGACGGGCTTTAGAC
+
HGGGGGGGGGGGGGGGGGGGGFGGFGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGFFFFFFFFFFFFFFFFFFEDDCD
@SIMULATED_SEQ_2:1:1101:1587:2131 1:N:0:1
GTGAGAGCTCGCTGAGTTACGTATGCCGTCTTCTGCTTGGCAGGGCCTCAATCAATAACTTGTCTCCGCCTCTTTTAAGGCAGGGCTCGAGAAATTCGCGC
+
HIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIHHHHHHHHHHHHHHGGGG
@SIMULATED_SEQ_3:1:1101:1621:2145 1:N:0:1
CTTACCGGCAGTGGTCCTCGTATTATCTGTCACCCGTTATACTTTTCAGGGCCTTTTGATCGTCACCCCAGGCGTCCA
+
HIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIHHHHHHHHGGGGGGF
```

Troubleshooting Common Issues

- Arguments provided are incorrect

Remember that the arguments can be given in any order, but they must follow this syntax:

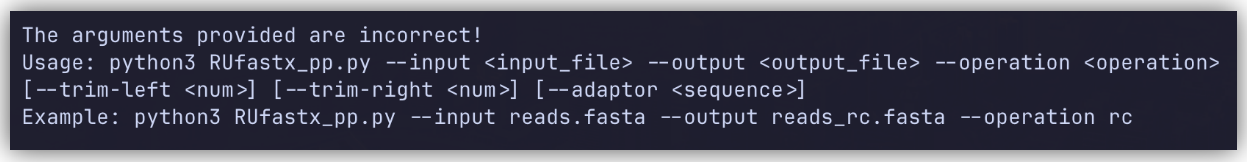
```
python3 RUfastx_pp.py --input <input_file> --output <output_file> --operation <operation> [additional options]
```

- For `rc` < operation >: Ensure you provide `--input` , `--output` , and `--operation rc`

- For `trim` < operation >: You must also include `--trim-left` and/or `--trim-right` with numeric values

- For `adaptor-removal` < operation >: You must also include `--adaptor` with a valid DNA sequence

If there are missing required arguments or they are incorrect, the following message will be shown:

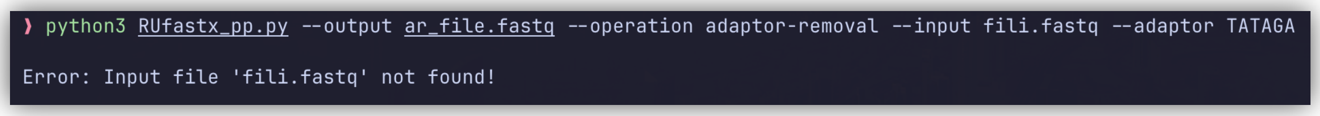


```
The arguments provided are incorrect!
Usage: python3 RUfastx_pp.py --input <input_file> --output <output_file> --operation <operation>
[--trim-left <num>] [--trim-right <num>] [--adaptor <sequence>]
Example: python3 RUfastx_pp.py --input reads.fasta --output reads_rc.fasta --operation rc
```

- Input file does not exist

Make sure the file name (including its extension) is correct and that it is in the same directory as your script, or use the full file path if it's located elsewhere

If the input file does not exist in the current directory, the following message will be shown:

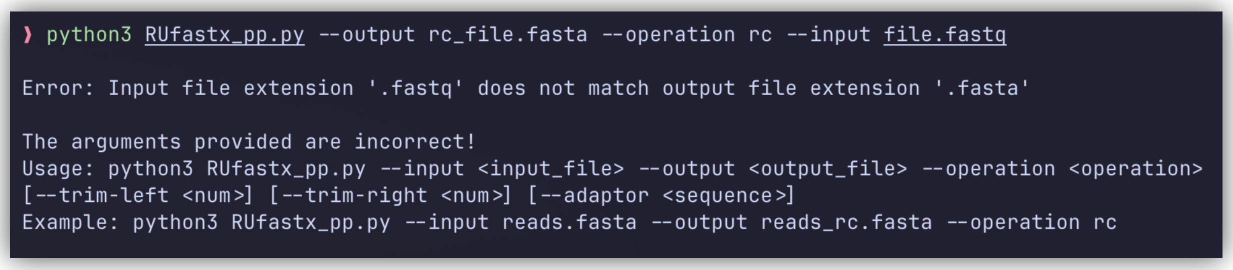


```
❯ python3 RUfastx_pp.py --output ar_file.fastq --operation adaptor-removal --input fili.fastq --adaptor TATAGA

Error: Input file 'fili.fastq' not found!
```

- Input & Output files extension do not match

The input and output file must have a `.fasta` or `.fastq` extension and the same extension.

If they have different extensions, the following message will be shown:

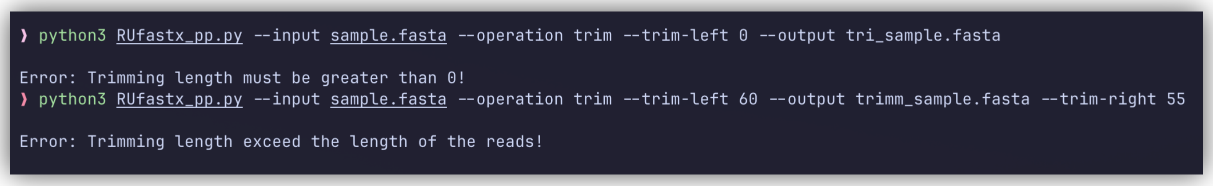


```
❯ python3 RUfastx_pp.py --output rc_file.fasta --operation rc --input file.fastq

Error: Input file extension '.fastq' does not match output file extension '.fasta'

The arguments provided are incorrect!
Usage: python3 RUfastx_pp.py --input <input_file> --output <output_file> --operation <operation>
[--trim-left <num>] [--trim-right <num>] [--adaptor <sequence>]
Example: python3 RUfastx_pp.py --input reads.fasta --output reads_rc.fasta --operation rc
```

- Trimming lengths of 0 or greater than the length of the read

At least one trim value must be greater than 0 and the total trimming length (left + right) must be less than the shortest read length.

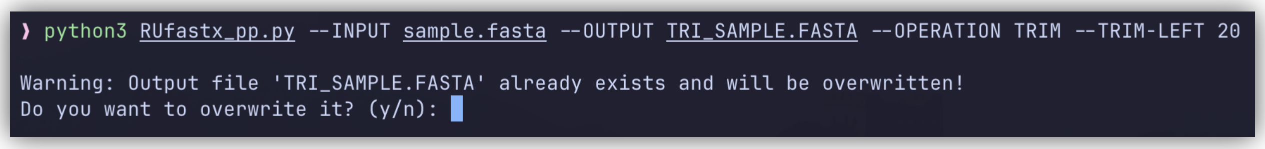
If the total trimming length is equal to 0 or greater than the sequence length, the following message will be shown:



```
❯ python3 RUfastx_pp.py --input sample.fasta --operation trim --trim-left 0 --output tri_sample.fasta

Error: Trimming length must be greater than 0!
❯ python3 RUfastx_pp.py --input sample.fasta --operation trim --trim-left 60 --output trimm_sample.fasta --trim-right 55

Error: Trimming length exceed the length of the reads!
```

- Output file already exists

If the file output already exists in the current directory, the tool will ask if you want to overwrite it or not before proceeding:



```
❯ python3 RUfastx_pp.py --INPUT sample.fasta --OUTPUT TRI_SAMPLE.FASTA --OPERATION TRIM --TRIM-LEFT 20

Warning: Output file 'TRI_SAMPLE.FASTA' already exists and will be overwritten!
Do you want to overwrite it? (y/n):
```

Error Prevention Checklist

Before running any operation, verify:

- Input file exists and is readable
- File extensions match (.fasta with .fasta, .fastq with .fastq)
- Trim values are reasonable (not larger than read length)
- Adaptor sequence is correctly spelled (check sequencing provider documentation)
- Sufficient disk space for output files (roughly same size as input)
- You have a backup of your raw data (never overwrite original files)

Additional Tips

1. Check your data first: Before processing large files, ensure the input file is properly formatted and there are no errors within.
2. All arguments are case insensitive: e.g., for the adaptor removal operation, and `--adaptor` `TATAGA` , `tataga` , and `TaTaGa` will all work the same way.
3. Always check your results: After processing, you can view your output files using any text editor or command-line tools like `head` or `cat` .