Ruby Programming Exercises

Introduction

This document contains a series of exercises designed to enhance your understanding and skills in Ruby programming. The exercises are categorized into three levels: Intermediate, Advanced and Extra. Each exercise is thoroughly explained to ensure a clear understanding of the concepts involved.

Table of Contents

Intermediate Exercises in Ruby

Exercise 1: Filtering Even Numbers

Description

Write a Ruby program that filters out the even numbers from an array of integers.

Implementation

- **Function**: `filter_even(arr)`
  - **Parameter**: `arr` - An array of integers.
  - **Return**: An array containing only the even numbers from the input array.
- **Example Usage**:
  - `filter_even([1, 2, 3, 4, 5, 6])` should return `[2, 4, 6]`.

Exercise 2: Person Class with Adult Check

Description

Create a `Person` class in Ruby with `name` and `age` attributes and a method to check if the person is an adult (18 years or older).

Implementation

- **Class**: `Person`
  - **Attributes**: `name`, `age`.
  - **Method**: `is_adult?` - Returns `true` if age is 18 or older, `false` otherwise.
- **Example Usage**:
  - `person = Person.new("John", 20)`
  - `person.is_adult?` should return `true`.

Exercise 3: Word Count in a File

Description

Write a Ruby program that reads a text file and counts the occurrences of each word, displaying the word count for each word.

Implementation

- **Class**: `WordCounter`
  - **Method**: `count_words` - Reads a file and returns a hash with word counts.
- **Example Usage**:
  - `counter = WordCounter.new("file.txt")`
  - `word_counts = counter.count_words`
  - Iterate over `word_counts` to display each word and its count.

Exercise 4: Hotel Reservation System Simulation

Description

Implement a program in Ruby that simulates a hotel reservation system, processing reservations in parallel using threads.

Implementation

- **Class**: `Hotel`
  - **Method**: `processar_reserva(nome_cliente, quarto)` - Simulates reservation processing for a client and room.
- **Parallel Processing**:
  - Read reservations from a file, creating a thread for each reservation.
  - Each thread should print start and end messages for reservation processing.
  - Use synchronization objects as necessary.
  - Wait for all threads to finish before terminating the program.
- **Example Output**:
  - Messages indicating the start and end of reservation processing for each client and room.

These intermediate exercises provide a comprehensive understanding of array manipulation, object-oriented programming, file handling, and multithreading in Ruby. They are ideal for students looking to deepen their Ruby programming skills.

Advanced Exercises

Exercise 1: Simple Blog Engine in Ruby

Objective of the Exercise

Develop a straightforward blog engine in Ruby. This task involves creating classes for authors, blog posts, and the blog itself. Each class should have specific attributes and capabilities as outlined below.

Class Specifications

1. **Author Class**:
   - **Attributes**:
     - `name` : Represents the author's name.
     - `biography` : A short biography of the author.
   - **Methods**:
     - `initialize` : Constructor that initializes the author's name and biography.
2. **Post Class**:
   - **Attributes**:
     - `title` : The title of the blog post.
     - `content` : The main content/body of the blog post.
     - `author` : An instance of the Author class representing the post's author.
     - `publication_date` : The date when the post was published (use Ruby's `Date` class).
   - **Methods**:
     - `initialize` : Constructor to initialize title, content, author, and publication date.
3. **Blog Class**:
   - **Attributes**:
     - `posts` : An array to store instances of the Post class.
   - **Methods**:
     - `initialize` : Constructor that initializes the blog with an empty posts array.
     - `add_post` : Adds a new post to the blog.
     - `list_all_posts` : Lists all posts in the blog.
     - `list_posts_by_author` : Filters and lists posts written by a specific author.

Task

- Implement the `Author`, `Post`, and `Blog` classes as per the specifications.
- Ensure the `Post` class references the `Author` class for its author attribute.
- In the `Blog` class, implement methods to add new posts, list all posts, and filter posts by a specific author.

Exercise 2: Log Analysis in Ruby

Objective of the Exercise: Develop a Ruby-based log analysis tool. This involves creating `LogEntry` and `LogAnalyzer` classes to parse, filter, and summarize log files.

Part 1: Defining the LogEntry Class

- **Attributes**:
  - `date`: The date of the log entry.
  - `time`: The time of the log entry.
  - `operation`: The type of operation.
  - `message`: The message of the log entry.
- **Methods**:
  - `initialize`: Constructor that takes date, time, operation, and message.
  - `to_s`: Returns a string representation of the log entry.
  - `is_type?`: Checks if the operation type of the log entry matches the specified type.

Part 2: Developing the LogAnalyzer Class

- **Attributes**:
  - `entries`: An array of LogEntry instances.
- **Methods**:
  - `initialize`: Reads a log file and creates instances of LogEntry for each valid line.
  - `read_log_file`: Reads a log file and creates LogEntry instances.
  - `parse_log_line`: Parses a log line and returns an instance of LogEntry.
  - `filter_by_operation`: Filters entries by type of operation.
  - `create_filter_methods`: Dynamically creates methods for each type of operation.
  - `summarize_operations`: Generates a summarized report of the operations.
  - `sort_by_datetime`: Sorts the entries by date and time.

Example Usage

- Create an instance of LogAnalyzer, using a log file.
- Utilize the instance methods to analyze the log file.
- Display sorted entries by datetime.
- Show filtered logs based on operation type (e.g., Error, Warning).
- Present a summary of operations.

Expected Outcome

- The ability to parse and analyze log files efficiently using OOP principles in Ruby.
- Dynamically created methods for filtering based on operation types.
- Summarized reports of operations, aiding in log file analysis.

Extra Exercises

Exercise: Chat Server with Metaprogramming Features

Objective of the Exercise: Develop a simple chat server in Ruby using socket programming for network communication and metaprogramming to dynamically add commands to the server.

Part 1: Socket Programming - Chat Server

1. **Create a Basic Chat Server**:
   - Utilise Ruby's `TCPServer` class to create a basic chat server that can accept client connections.
   - The server should be able to receive messages from one client and relay them to all other connected clients.

Part 2: Metaprogramming - Dynamic Commands

2. **Add Commands Dynamically**:
   - Implement a feature allowing the creation of new commands on the chat server during runtime.
   - Use metaprogramming to define methods based on received commands. For example, a `/shout` command could be added to send a message in uppercase to all clients.

Part 3: General Syntax of Ruby and Code Structure

3. **Use Ruby Syntax and Data Structures**:
    - Organise the code idiomatically, using Ruby's conventions and data structures like arrays, hashes, blocks, and iterators.
    - Ensure the code follows Ruby's best style practices, like using snake_case for variable and method names.

Additional Challenges

- Implement a logging system that records messages and actions on the server.
- Add chat room functionality, where users can create or join separate rooms.

Expected Outcome

- A functional chat server where users can connect, send messages, and use special commands.
- The ability to add new commands to the server during runtime, demonstrating the use of metaprogramming.

This exercise addresses socket programming for network communication, metaprogramming for adding flexibility and dynamism to the code, and requires a good understanding of Ruby's syntax and conventions. It's a challenging but highly rewarding project for those wishing to delve deeper into these Ruby areas.