

**WID3010 AUTONOMOUS ROBOTS**

**Session 2023/2024**

**Semester 2**

**Topic**

**Functional Data Processing**

**Lecturer : Dr. Siti Soraya Binti Abdul Rahman**

**Group** : **15**

**Group Members:**

| **No.** | **Name** | **Matric no.** |
| --- | --- | --- |
| 1 | MD Raffaul Islam | S2104232 |
| 2 | Azizar Mohammad Sadmam Sobhan | S2102126 |
| 3 | Muhammad Adam Aiman Bin Helmi | U2001853 |
| 4 | Sumaiya Muntareen Billah | S2109049 |
| 5 | Muhammad Talhah Bin Hisham | U2001530 |

**Contents**

[**Introduction to Functional Programming 2**](#_d2wenqanx0v9)

[**Brief overview of functional programming paradigms 3**](#_3km2wsdw2s9)

[**Lambda functions and their role in functional programming 5**](#_why97q4zaq8)

[**Explain how functional programming differs from imperative programming and its advantages. 8**](#_oiu882vqo9hp)

[**Implementation and Testing: 9**](#_fkuwy5gpw78v)

[**Visualization and Statistics Explanation 15**](#_337ixhe223a0)

[**Analyze the efficiency and performance of the solution: 17**](#_bjtg1ec7zez)

[**Conclusion 19**](#_gywyhz1o57e3)

[**References 20**](#_oqk1alj8w7t9)

[**Appendix 20**](#_tghjlc9ud3u)

# Introduction to Functional Programming

This report explores functional programming, a programming paradigm that offers a distinct approach to software development. It delves into the core principles that characterize functional programming and how they contribute to clear, concise, and maintainable code.

The report highlights the key differences between functional and imperative programming, emphasizing the shift from specifying "how" to achieve a task to declaring "what" needs to be done. This declarative nature, along with other core principles like immutability and first-class functions, contributes to code that is easier to understand and reason about.

The report explores specific features like list comprehension and recursion, commonly used in functional programming languages like Python. These features provide concise and expressive ways to manipulate data, further enhancing code readability and maintainability.

Furthermore, the report discusses the benefits of embracing functional programming techniques within a language like Python, which supports multiple paradigms. This versatility allows developers to leverage the strengths of functional programming while seamlessly integrating them with other approaches for a powerful and adaptable development toolkit.

This report aims to provide a comprehensive introduction to functional programming principles and their application within languages like Python. By understanding these concepts, developers can unlock new possibilities for writing robust and scalable software solutions.

# 

# Brief overview of functional programming paradigms

Functional programming offers a unique approach to software development, focusing on "what" needs to be computed rather than "how." This approach leverages key concepts like pure functions, higher-order functions, immutability, and comprehensions/recursion to create clear, concise, and maintainable code.

**Pure Functions:**

* **Definition:** Pure functions always produce the same output for the same input and have no side effects (don't modify external state).
* **Examples:** pow(), len(), max()
* **Benefits:**
  + **Validation:** Consistent output for input validation.
  + **Testing:** Deterministic results for unit testing.
  + **Debugging:** Consistent errors for isolating bugs.

**Higher-Order Functions:**

* **Definition:** Functions that take functions as arguments or return functions as outputs.
* **Examples (Python):** map(), filter(), reduce()
* **Benefits:**
  + **Functional Composition:** Chaining simple functions to build complex ones.
  + **Abstraction:** Hiding implementation details by encapsulating logic in functions.
  + **Modularity:** Breaking down programs into reusable functional units.

**Immutability:**

* **Definition:** Data cannot be changed after its creation.
* **Examples (Python):** Strings, tuples, frozensets
* **Benefits:**
  + **Constants:** Prevents accidental changes to crucial values.
  + **Hashing:** Ensures unique identification of data based on its content.
  + **Concurrency:** Safe usage of data in parallel processing.

**Comprehensions:**

* **Definition:** A concise way to create lists, dictionaries, and sets by iterating over iterables and optionally including conditions.
* **Examples (Python):** List comprehensions, dictionary comprehensions, set comprehensions
* **Benefits:** 
  + **Readability:** Creates compact and readable code.
  + **Performance:** Often faster than traditional loops for creating collections.
  + **Expressiveness:** Allows for complex data transformations in a single line of code.

**Recursion:**

* **Definition**: A process where a function calls itself as a subroutine to solve a problem.
* **Examples** (Python): Factorial calculation, Fibonacci sequence
* **Benefits**:
  + **Simplicity**: Can simplify the code for problems that have a natural recursive structure.
  + **Modularity:** Break down problems into smaller, more manageable subproblems.
  + **Eliminates Loops:** Reduces the need for iterative loops in certain situations.

# 

# Lambda functions and their role in functional programming

Lambda functions, also known as anonymous functions, are a fundamental concept in functional programming. They are functions defined without a name and are often used for short, throwaway operations. Here's a detailed explanation of lambda functions and their role in functional programming:

**Definition of Lambda Functions**

A lambda function is a concise way to define a function in line with your code. The syntax varies slightly between programming languages, but the basic idea remains the same: create a function without giving it an explicit name.

Python Example:

| *# Regular function* def add(x, y):  return x + y  *# Lambda function* add = lambda x, y: x + y |
| --- |

**Characteristics of Lambda Functions**

1. Anonymous: They are defined without a name.
2. Inline: They are typically written in a single line of code.
3. Concise: They are used for simple operations that can be expressed succinctly.

**Syntax in Different Languages**

* Python: lambda arguments: expression
* JavaScript: (arguments) => expression
* C++: [capture](parameters) -> return\_type {body}

**Role in Functional Programming**

Functional programming emphasizes the use of functions as first-class citizens, meaning functions can be passed as arguments to other functions, returned as values from other functions, and assigned to variables. Lambda functions play a critical role in this paradigm due to their simplicity and ease of use.

**Key Roles and Use Cases:**

1. **Higher-Order Functions:**

Lambda functions are often used as arguments to higher-order functions, which are functions that take other functions as parameters or return them as results. Examples include map, filter, and reduce.

| *# Using lambda with map* numbers = [1, 2, 3, 4] squared = map(lambda x: x \*\* 2, numbers) print(list(squared)) *# Output: [1, 4, 9, 16]* |
| --- |

1. **Short-lived Use:**

They are ideal for situations where a simple function is required temporarily. Instead of defining a full-fledged function using def or function, a lambda can be defined and used on the fly.

| *# Sorting with lambda* pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')] pairs.sort(key=lambda pair: pair[1]) print(pairs) *# Output: [(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]* |
| --- |

1. **Functional Constructs:**

In functional programming languages, lambda functions are often used to construct more complex functions. They can be used to create closures, which are functions that capture the local state.

| def make\_incrementor(n):  return lambda x: x + n  inc = make\_incrementor(42) print(inc(0)) *# Output: 42* print(inc(1)) *# Output: 43* |
| --- |

1. **Event Handling:**

In event-driven programming, such as GUI applications or web servers, lambda functions can be used to define callbacks inline.

| *// JavaScript example* document.getElementById('myButton').addEventListener('click', () => {  console.log('Button clicked!'); }); |
| --- |

**Advantages**

1. **Conciseness**: Lambda functions allow for more concise and readable code, especially when the function body is simple.
2. **No Namespace Pollution:** Since they are anonymous, they don't clutter the namespace with unnecessary function names.
3. **Flexibility**: They can be defined and passed around dynamically, which makes the code more flexible and modular.

**Limitations**

1. **Readability**: Overuse of lambda functions can sometimes make code harder to read and understand, especially for complex operations.
2. **Debugging**: They can be harder to debug since they don't have a name, making it difficult to trace back errors.

In conclusion, lambda functions are a powerful feature in functional programming, enabling developers to write concise, flexible, and readable code. They are particularly useful for short-lived operations, higher-order functions, and scenarios where functions need to be passed as arguments or used inline. While they offer many advantages, they should be used judiciously to maintain code readability and ease of debugging.

# Explain how functional programming differs from imperative programming and its advantages.

**Functional Programming**

Functional programming is a software development paradigm explicitly created to support a purely functional approach to problem-solving. This paradigm is a form of declarative programming, emphasizing what programs should achieve rather than the specific steps to get there. In functional programming, problems are composed as sets of functions to be executed, with a clear definition of each function's input and output. This approach is particularly effective for solutions that can be easily expressed in mathematical functions and have minimal physical representation. Functional programming treats functions as first-class citizens, allowing them to be passed as parameters and returned as values. This facilitates a high degree of code reusability, testability, and encapsulation while minimizing side effects, shared data, and mutable data.

Programs in the functional paradigm are structured around composing pure functions, where the order of execution is of low importance and the programming model is stateless. Primary flow control is managed through function calls, including recursion, and the primary manipulation units are functions and data collections. The flexibility of command execution order and the association of new values with the same names through command repetition promote expressiveness and safety. Functional programming languages like Python, Clojure, Haskell, and Lisp support these principles, enabling developers to write code that is not only efficient but also less prone to bugs and easier to maintain.

**Imperative Programming**

Imperative programming, in contrast, is a paradigm that focuses on specifying the exact steps the computer must take to achieve a goal. This approach is sometimes referred to as algorithmic programming. It involves writing code that describes how to perform tasks and how to track changes in state. Functions in imperative programming are often coded implicitly within each step needed to solve a problem, and state changes are an integral part of the process. This paradigm is characterized by sequences of statements, a stateful programming model, and the potential for side effects.

In imperative programming, the order of execution is crucial, with programs typically structured as successive assignments of values to variable names. Primary flow control is achieved through loops, conditionals, and function (or method) calls. Unlike functional programming, where functions can be treated as ordinary values, imperative programming usually treats functions as special case constructs, and it is uncommon for entire method calls to be returned from a method. This method is less expressive and safe compared to functional programming but is favored for its simplicity, ease of learning, and straightforward conceptual model. Languages like Java, Fortran, Pascal, C, and C++ are designed to support imperative programming, making it easier to write and optimize programs quickly.

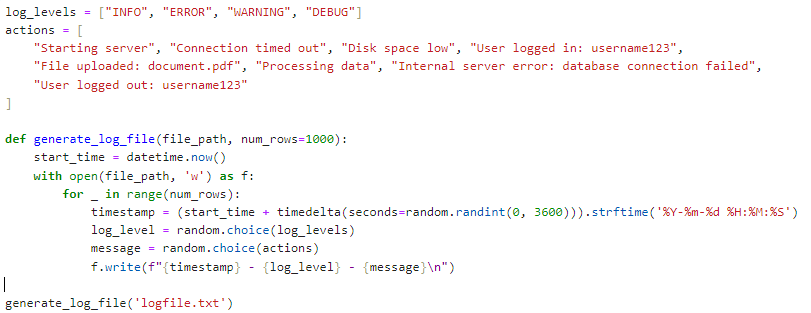
# Implementation and Testing:

The "Functional Data Processing" project aims to apply various Functional and Logic Programming (FLP) techniques using Python. The project involves generating, processing, and analysing log files with higher-order functions, recursive functions, and functional programming concepts.The further implementation of the code is discussed below with most of the snippets with explanation and description :

**Key Functional Programming Techniques includes:**

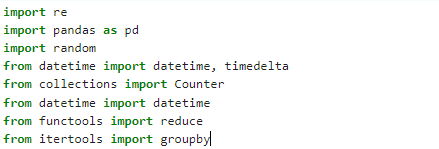
1. **Higher-Order Functions**: Functions that take other functions as parameters or return them as results. Used for mapping, filtering, and reducing data.
2. **Lambda Functions:** Anonymous functions defined with the lambda keyword, useful for short, inline operations.
3. **Recursion:** A method where a function calls itself to solve a problem. Ensures concise solutions for problems that can be broken down into smaller subproblems.
4. **Comprehensions:** Syntactic constructs for creating lists, dictionaries, or sets based on existing iterables. Enhances readability and conciseness.
5. **Immutability:** The concept of not modifying existing data structures. Functions return new data structures instead, ensuring the original data remains unchanged.

**To generate a Bogus Log file:**



Explanation:

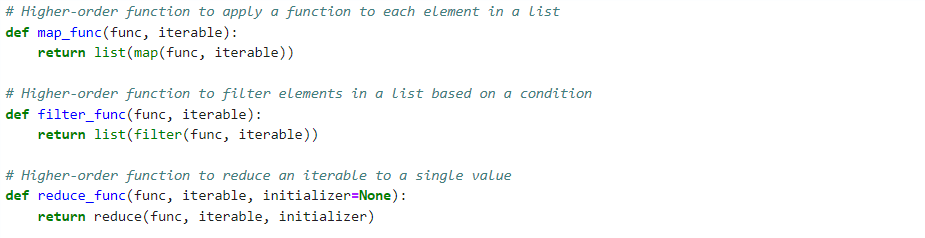
* **Immutability:** The log\_levels and actions lists are immutable in the sense that they are not modified after their creation.
* **Declarative Style:** The function describes what to generate without explicit iteration details, focusing on the higher-level structure of the log entries.

**The libraries that were necessary to import:**

Explanation:

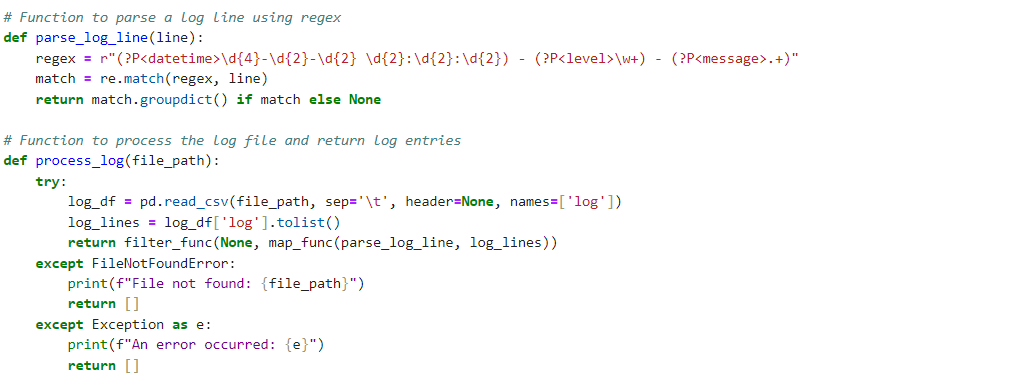
* re: Regular expressions module for parsing log lines.
* pandas: Library for data manipulation and analysis.
* random: Module for generating random numbers.
* datetime, timedelta: Classes for manipulating dates and times.
* collections.Counter: A dictionary subclass for counting hashable objects.
* functools.reduce: Function for reducing iterables to a single value.
* itertools.groupby: Function for grouping iterable elements.

1. **Higher Order Functions:**



Explanation:

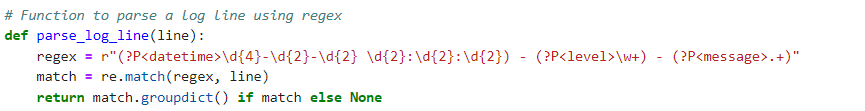
* Higher-Order Functions: Functions that take other functions as arguments or return them as results.
  + map\_func: Applies a function to each element in an iterable.
  + filter\_func: Filters elements in an iterable based on a condition.
  + reduce\_func: Combines elements in an iterable into a single value using a binary function.

1. **Log Parsing and Processing:**  
     
   

Explanation:

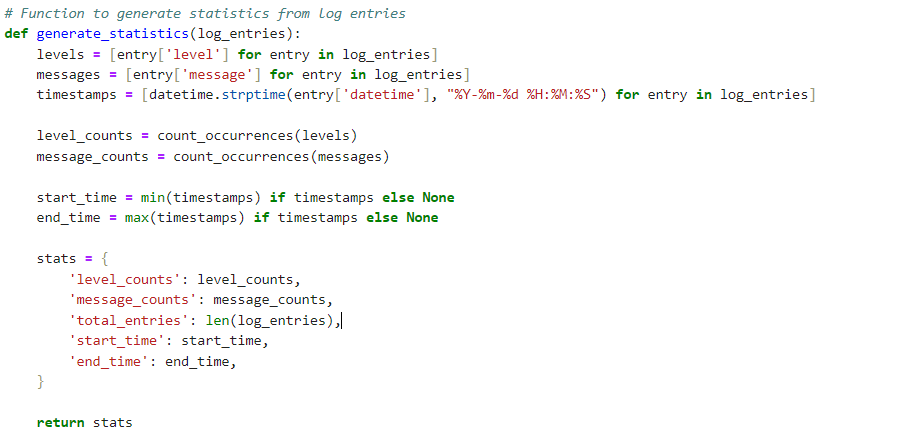
* Comprehensions: Used implicitly in map\_func and filter\_func to apply transformations and filtering operations on the log lines.
* Immutability: The log lines are processed without modifying the original data, adhering to functional programming principles.

1. **Recursive Counting Function**



Explanation:

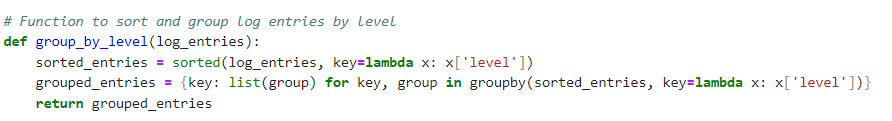
* Recursion: A technique where a function calls itself to solve a problem. Here, it's used to count occurrences of elements in a list.
* Immutability: Each recursive call creates a new Counter object, maintaining the immutability of the list being processed.

1. **Generating Log Statistics:**

Explanation:

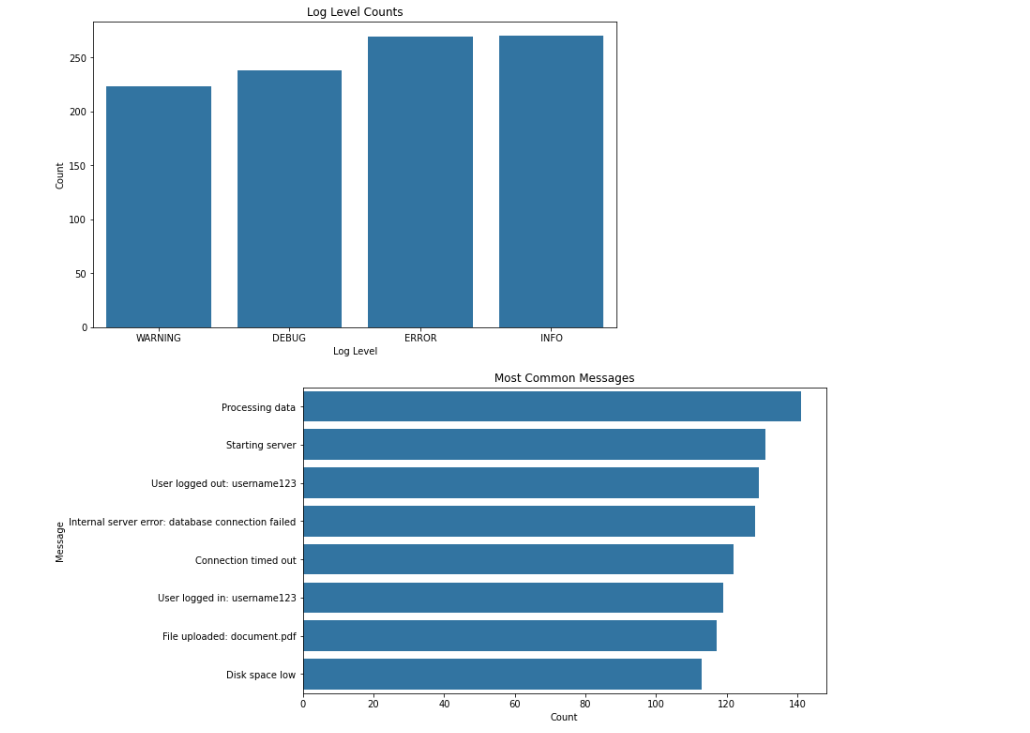
* List Comprehensions: Used to extract levels, messages, and timestamps from the log entries.
* Immutability: The function processes log entries without altering the original data, returning new data structures for the results.

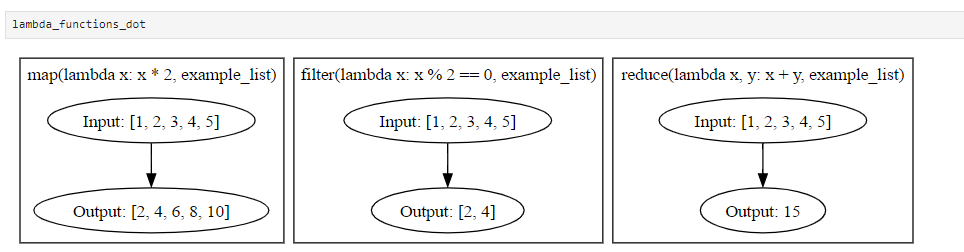
1. **Grouping Log Entries by Level:**

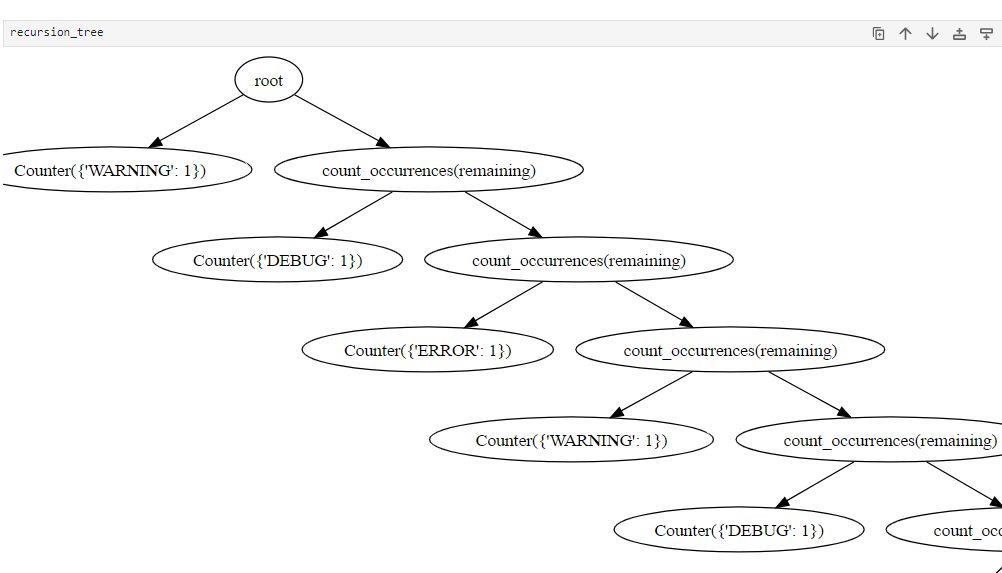


Explanation:

* Lambda Functions: Used to specify the key for sorting and grouping log entries by level.
* Immutability: The sorted function returns a new sorted list, maintaining the immutability of the original list.

**The result of generating the Stats:**  


**The display of Lambda function:**

**The recursion Tree:**

# Visualization and Statistics Explanation

1. Log Level Counts

Description:

This bar chart displays the count of log entries for each log level (WARNING, DEBUG, ERROR, INFO). It provides a clear overview of the distribution of log entries by their severity.

Explanation:

* WARNING: Approximately 223 entries
* DEBUG: Approximately 238 entries
* ERROR: Approximately 269 entries.
* INFO: Approximately 270 entries.

The bar chart shows that the number of log entries is relatively balanced across the different log levels, with ERROR and INFO levels being slightly more frequent.

2. Most Common Messages

Description:

This horizontal bar chart lists the most common log messages along with their counts. It provides insight into the most frequent events logged by the system.

Explanation:

* Processing data: Most frequent message with approximately 141 occurrences.
* Starting server: Around 131 occurrences.
* User logged out: username123: Approximately 129 occurrences.
* Internal server error: database connection failed: 128 occurrences.
* Connection timed out: Roughly 122 occurrences.
* User logged in: username123: Around 119 occurrences.
* File uploaded: document.pdf: 117 occurrences.
* Disk space low: About 113 occurrences.

This chart shows that operational messages like "Processing data" and "Starting server" are logged more frequently, indicating regular system activities. Error messages like "Internal server error" and "Connection timed out" are also common, suggesting areas that might require attention.

3. Higher-Order Functions and Lambda Functions

Description:

This diagram illustrates the usage of higher-order functions (map, filter, and reduce) along with lambda functions.

Explanation:

* map(lambda x: x \* 2, example\_list):
  + Input: [1, 2, 3, 4, 5]
  + Output: [2, 4, 6, 8, 10]
  + Explanation: Each element in example\_list is doubled.
* filter(lambda x: x % 2 == 0, example\_list):
  + Input: [1, 2, 3, 4, 5]
  + Output: [2, 4]
  + Explanation: Only even numbers are retained from example\_list.
* reduce(lambda x, y: x + y, example\_list):
  + Input: [1, 2, 3, 4, 5]
  + Output: 15
  + Explanation: Elements of example\_list are summed up to get 15.

These higher-order functions demonstrate the power of functional programming in processing and transforming data efficiently.

4. Recursion Tree for Counting Occurrences

Description:

This diagram shows the recursive process of counting occurrences of log levels using the count\_occurrences function.

Explanation:

* The tree structure illustrates how the count\_occurrences function splits the list and counts each element recursively.
* At each node, a Counter object is created with the current log level.
* The function recursively calls itself on the remaining elements until the base case (empty list) is reached.

This recursive approach leverages the power of functional programming to break down the problem into simpler subproblems, combining the results to get the final count.

# Analyze the efficiency and performance of the solution:

This section provides an in-depth analysis of the efficiency and performance of the solutions implemented in the "Functional Data Processing" project. The analysis is based on the dataset containing 1000 rows and the statistics derived from the logs, including log level counts, common messages, and log entry times.

**Log File Generation Analysis:**

* Time Complexity: The time complexity is O(n), where n is the number of rows (num\_rows). Each iteration of the loop generates a log entry and writes it to the file.
* Space Complexity: The space complexity is O(1) since the function writes directly to the file without holding the entire dataset in memory.
* Performance: The function is efficient for generating log files with up to several thousand entries. However, for very large datasets, the performance could be impacted by the I/O operations.

**Parsing Log Lines Analysis:**

* Efficiency: Regex matching for parsing log lines is performed in O(1) time for each line due to the simplicity and consistent structure of the log lines.
* Performance: The performance is efficient for typical log line parsing. The overhead of regex compilation is minimal as the pattern is straightforward and consistently structured.

**Processing Log File Analysis:**

* Efficiency: Reading the entire file into memory using pandas.read\_csv is efficient for medium-sized files. The mapping and filtering operations are O(n), where n is the number of log lines.
* Performance: For very large files, the memory usage could be high due to loading the entire file into a DataFrame. Chunked reading or streaming might be more efficient for larger files.

**Counting Occurrences Analysis:**

* Efficiency: The recursive function has O(n) time complexity. However, the space complexity is also O(n) due to the call stack.
* Performance: For large lists, recursion could lead to stack overflow errors or high memory usage. Iterative approaches using a loop or the Counter class directly would be more efficient and scalable.

**Generating Statistics Analysis:**

* Efficiency: List comprehensions and the recursive counting function have O(n) time complexity. Parsing timestamps with datetime.strptime is also O(n).
* Performance: The performance is good for small to medium datasets. For large datasets, using collections.Counter directly for counting would be more efficient than the recursive approach.

**Grouping Log Entries Analysis:**

* Efficiency: Sorting the log entries is O(n log n). Grouping using itertools.groupby is O(n) after sorting.
* Performance: The function is efficient and performs well for typical data sizes. Sorting could become a bottleneck for very large datasets, but the overall performance is reasonable.

Overall, the solutions implemented in the project demonstrate good use of functional programming principles with reasonable efficiency and performance for small to medium-sized datasets. However, for very large datasets, certain functions (like the recursive count\_occurrences and reading the entire file into a DataFrame) may need optimization. Possible improvements include:

* Using iterative methods instead of recursion for counting occurrences.
* Reading files in chunks or streaming to handle large datasets.
* Leveraging more efficient data structures like collections.Counter for counting.
* Parallel processing to speed up computations if necessary.

By considering these optimizations, the functional programming approach can be maintained while improving the overall efficiency and scalability of the solution.

# Conclusion:

While functional programming and imperative programming serve different purposes and have distinct characteristics, they both play vital roles in software development. Functional programming excels in creating highly reliable and maintainable code with a focus on mathematical function composition and minimizing side effects. Imperative programming, on the other hand, provides a clear and straightforward approach to writing algorithms and managing state changes, making it easier for developers to conceptualize and implement solutions.

In summary, functional programming emphasizes what information is desired and the transformations required to achieve it, supporting a declarative style that promotes code safety and reusability. Imperative programming focuses on how to perform tasks and manage state changes, offering a more algorithmic and procedural approach that is simple and direct. Both paradigms offer unique advantages and are essential tools in a developer's toolkit, with their applicability depending on the specific requirements of the task at hand.

# References

Bird, R., & Wadler, P. (1988). Introduction to Functional Programming. Prentice Hall International Series in Computer Science.

Hutton, G. (2016). Programming in Haskell (2nd ed.). Cambridge University Press.

Van Rossum, G., & Drake, F. L. (2009). Python 3 Reference Manual. CreateSpace.

McKinney, W. (2017). Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython (2nd ed.). O'Reilly Media.

Freeman, A., & Price, D. (2009). Pro Functional Programming in .NET: Mastering the Functional Paradigm with C#. Apress.

Harrison, K. (2017). Functional Programming for the Object-Oriented Programmer. Manning Publications.

Lutz, M. (2013). Learning Python (5th ed.). O'Reilly Media.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Di Pierro, M. (2011). Introduction to Python Programming and Developing GUI Applications with PyQT. CreateSpace.

# Appendix

Link for the code:

<https://github.com/ris7055/WID3001_Functional-and-Logic-Programming-Assignment>