MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Pipeline-oriented scripting language for Data processing
Project report

Mentor: prof., Leon Brînzan
Students: Clepa Rodion, FAF-221
Gavriliuc Tudor, FAF-221
Pascal Adrian, FAF-221
Sungur Emre-Batuhan, FAF-221

Chișinău, 2024

Introduction

In the fast-changing world of modern computing, the demand for efficient data processing and automation tools continues to grow. Organizations across industries are constantly looking for innovative solutions to streamline workflows, speed up processes, and extract actionable insights from massive amounts of data. In the search for optimization, pipeline-based scripting languages have emerged as a powerful paradigm that offers a universal approach to organizing complex data workflows.

The purpose of this report is to provide a comprehensive analysis of pipeline-oriented scripting languages, with a focus on their use in revolutionizing data workflows. We delve into the fundamental concepts, design principles, and practical implications of these languages, exploring their implications in modern computing environments. Through detailed exploration of domain specifics, use cases, and best practices, we aim to elucidate the transformative potential of pipeline-oriented scripting languages in the fields of data science and automation.

The exponential growth of data in recent years has created both opportunities and challenges for organizations around the world. The sheer volume, velocity and variety of data generated from different sources has required the development of innovative approaches to manage, process and extract value from this wealth of information. Traditional programming paradigms often struggle to cope with the complexities of modern data workflows, leading to inefficiencies, bottlenecks, and scalability issues.

Pipeline-oriented scripting languages offer a paradigm shift in how data workflows are conceptualized, designed, and executed. Based on the concept of pipelines—a series of interconnected processes in which the output of one stage serves as input to the next—these languages provide a streamlined framework for automating tasks, coordinating data transformations, and facilitating seamless integration with existing systems and tools. By breaking complex operations into modular, building blocks, pipeline-oriented scripting languages enable users to develop flexible, scalable, and efficient data workflows tailored to their specific requirements.

Abstract

This article explores the field of pipeline-oriented scripting languages, focusing on the development and use of a Python-based pipeline language. Pipeline-oriented scripting languages offer an approach that will optimize workflows through interconnections between processes. Benefits include optimized workflow, modularity—which refers to the ease of breaking down a system into interconnected modules, clean and simple syntax, efficient use of resources, and full compatibility. The study outlines the key requirements for building large-scale data pipelines and describes existing solutions that meet them.

The article addresses common issues in data science, automation, integration, maintainability, and scalability, and highlights the benefits of Python's pipeline language. The designed tool is used to simplify complex data processing tasks. Design considerations for a Python pipeline language include domain-specific abstractions, support for pipeline composition, declarative syntax, integration with an existing ecosystem. The proposal is to develop a special Python pipeline language to improve data processing and analysis.

Keywords: Pipeline, data, processing, language

Content

1 Midterm 1

1.1 Domain Analysis

Pipeline-oriented scripting languages can be especially useful in the field of automation and scripting, where time is of the importance and efficiency is critical. These languages provide a distinctive method for managing data, carrying out commands, and smoothly automating operations.

At its core, a pipeline-oriented scripting language revolves around the concept of pipelines - a series of connected processes where the output of one process serves as the input to the next.

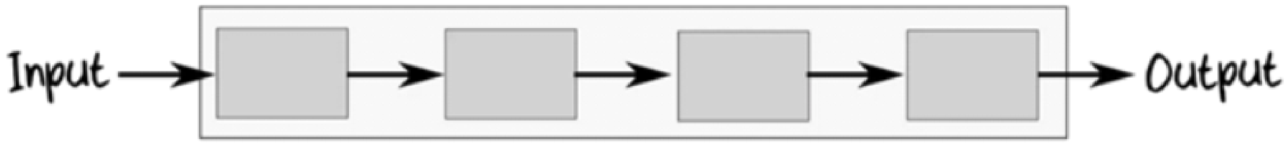


Figure 1.1.1 - Context-Free Grammar Notations

The significance of pipeline-oriented scripting languages lies in their ability to simplify and expedite automation tasks. By breaking down complex operations into smaller, composable units, these languages empower users to design modular and reusable scripts. The advantages they offer are manifold:

- Streamlined Workflow
- Modularity and Reusability
- Concise and Expressive Syntax
- Efficient Resource Utilization
- Interoperability and Integration

Streamlined Workflow:

- In a pipeline-oriented scripting language, data and commands flow seamlessly through interconnected processes, forming a pipeline. This streamlined workflow eliminates the need for manual intervention between each step, allowing users to automate complex operations effortlessly.
- By combining commands and functions in a pipeline, users can organize complex sequences of actions without having to write extensive scripts by hand.
- This streamlined workflow enhances productivity, reduces human error, and accelerates the execution of tasks, particularly in scenarios involving data processing, system administration, and automation.

Modularity and Reusability:

- Pipelines enable tasks to be decomposed into modular, independent components, each performing a specific function within the larger workflow.
- These modular components can be reused across different pipelines and projects, promoting code

reusability and minimizing duplication of effort.

- By breaking down complex tasks into smaller, manageable units, developers can build libraries of reusable components, fostering collaboration and accelerating development cycles.

Concise and Expressive Syntax:

- Pipeline-oriented scripting languages often feature concise and expressive syntax, enabling users to express complex operations using minimal code.
- Built-in constructs such as pipelines, filters, and transformations allow developers to write clear, readable scripts that are easy to understand and maintain.
- The expressive nature of these languages facilitates rapid prototyping and iteration, empowering users to translate ideas into functional scripts with minimal friction.

Efficient Resource Utilization:

- Pipelines support parallel execution of tasks, allowing multiple processes to run concurrently and utilize system resources efficiently.
- By distributing workloads across multiple threads or processes, pipelines maximize CPU utilization and minimize idle time, leading to improved performance and responsiveness.
- Efficient resource utilization is particularly advantageous in environments with high throughput requirements or limited computing resources, where optimizing execution speed and resource usage is paramount.

Interoperability and Integration:

- Pipeline-oriented scripting languages seamlessly integrate with existing tools, libraries, and systems, facilitating interoperability and data exchange.
- Users can leverage built-in mechanisms for interfacing with external programs, APIs, and data sources, enabling seamless integration with third-party services and platforms.
- This interoperability enhances the versatility and extensibility of pipeline-oriented scripting languages, allowing users to leverage a rich ecosystem of libraries, modules, and plugins to augment their workflows and extend functionality.

Pipeline-oriented is particularly useful in domains such as:

System Administration: Automating tasks related to managing servers, networks, and infrastructure.

Data Processing: Manipulating and transforming data from various sources, such as logs, databases, or APIs.

Text Processing: Analyzing and modifying text data, such as parsing log files, extracting information, or formatting output.

Automation: Creating scripts to automate repetitive tasks, ranging from file management to software deployment.

DevOps: Streamlining processes related to software development, testing, and deployment. [1]

Understanding Machine Learning Pipelines

In recent years, the field of machine learning has experienced remarkable advancements. Thanks to the widespread availability of GPUs and breakthrough concepts like Transformers and GANs, the realm of AI projects has exploded. Startups in the AI space are popping up left and right, while organizations in various sectors are eagerly adopting these cutting-edge techniques to tackle real-world problems.

There is an urgent need for standardized machine learning pipelines. These pipelines, like CI/CD processes in software development, are aimed at formalizing and optimizing work processes.

Unlike the cumbersome procedures of the past, modern web app deployment has become swift and efficient, thanks to tools and concepts like CI/CD. Similarly, standardizing machine learning pipelines can enhance our efficiency and reliability. By automating repetitive tasks and ensuring reproducibility, these pipelines empower us to focus on innovation rather than mundane maintenance.

Automated pipelines offer several advantages. Firstly, they free from the drudgery of maintaining existing models. Moreover, they minimize the occurrence of bugs by ensuring that models are always aligned with the latest data and preprocessing steps, thus preventing the deployment of flawed models.

Additionally, these pipelines generate a comprehensive paper trail, documenting model changes and facilitating compliance with regulations such as GDPR. Standardization further streamlines our operations, expediting onboarding processes and reducing setup time for new projects. Ultimately, the implementation of automated machine learning pipelines yields cost savings, enhances model accuracy, and promotes regulatory compliance.

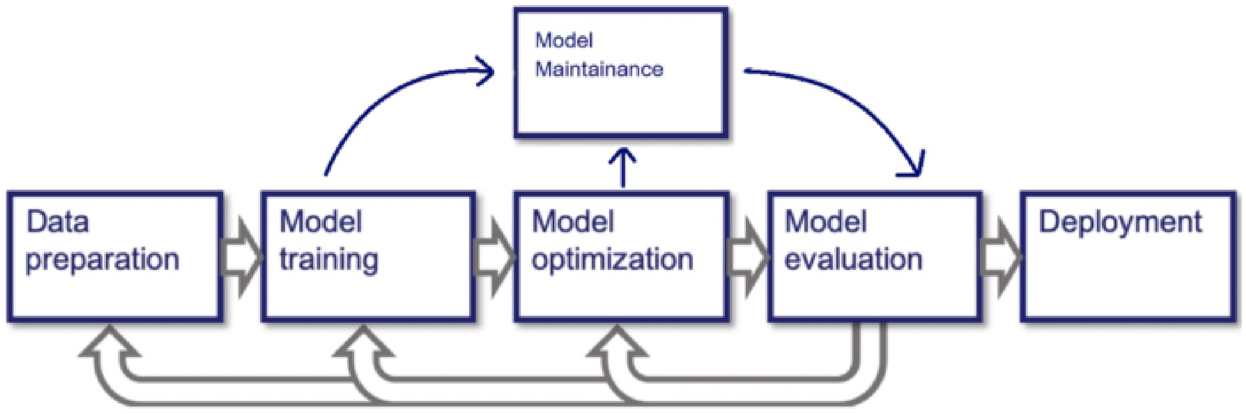


Figure 1.1.2 - Machine Learning Pipeline Key Components

Designing and Implementing Machine Learning Pipelines

Designing efficient ML pipelines requires adherence to best practices such as ensuring modularity and flexibility. Tools like TensorFlow Extended (TFX), Apache Airflow, and Kubeflow facilitate the building and management of pipelines. These frameworks offer robust capabilities for automating and scaling ML workflows, supporting everything from simple to complex pipeline configurations.

Challenges in Machine Learning Pipelines

Despite their benefits, ML pipelines face challenges, including handling large data volumes, maintaining data privacy, and adapting to dynamic data distributions. Addressing these challenges involves adopting advanced data management strategies, implementing robust security measures, and continuously monitoring data and model performance to make necessary

How Python Pipeline Library Benefits Machine Learning AI Workflows

In our project for the domain pipeline-oriented scripting language, the Python Pipeline Library demonstrates significant benefits for machine learning AI workflows. By employing a structured and standardized approach, this library enhances efficiency and productivity throughout the development life-cycle.

Firstly, the library streamlines the development process by providing a cohesive framework for defining pipelines and variables. This standardized approach ensures consistency and reproducibility across projects, reducing the risk of errors and accelerating time-to-deployment.

Furthermore, the Python Pipeline Library automates repetitive tasks such as data preprocessing, model training, and evaluation. By automating these processes, it not only saves time but also improves the reliability and robustness of machine learning models.

Additionally, the library facilitates collaboration among team members by providing a common platform and language for communication. With standardized workflows and documentation, team members can easily understand and contribute to each other's projects, fostering a culture of knowledge sharing and innovation.

Moreover, the library offers scalability and flexibility, allowing users to adapt to changing requirements and environments. Whether working with small datasets or large-scale deployments, the Python Pipeline Library provides the tools and resources needed to scale workflows efficiently.

Overall, the Python Pipeline Library is a valuable asset for machine learning AI workflows, empowering teams to focus on innovation and problem-solving. By embracing this library, organizations can unlock the full potential of their machine learning initiatives and drive impactful results.

Design Considerations for a Python Pipeline Library

Designing a Python pipeline library tailored for data science warrants careful deliberation:

Pipeline Blocks: Implement a syntax for defining blocks of pipeline operations using the | >pipe block production rule. This involves specifying individual operations within the pipeline block, along with any associated arguments.

Operation Syntax: Define a syntax for specifying individual pipeline operations using the | >pipe production rule. This should include the use of pipe symbols ($| >, || >, ||| >$) to denote the type of operation, the function name, and any function arguments.

Whitespace Handling: Implement rules for handling whitespace within pipeline definitions and operations to ensure readability and consistency.

Error Handling: Include mechanisms for handling syntax errors and other issues that may arise during parsing to provide informative error messages to users.

Integration with Python: Design the library to seamlessly integrate with Python code, allowing users to define and execute pipelines within their Python programs.

Python Tools and Frameworks for Data Pipeline;

Apache Airflow: Airflow is a platform to programmatically author, schedule, and monitor workflows. It allows you to define complex data pipelines as code using Python, providing features like scheduling, dependency management, and task monitoring.

Luigi: Luigi is a Python package that helps you build complex pipelines of batch jobs. It provides a simple API for defining tasks and dependencies between them, making it easy to create and manage data workflows.

Pandas: While not a pipeline framework per se, Pandas is a powerful library for data manipulation and analysis in Python. It provides versatile data structures and functions for reading, transforming, and writing data, making it a valuable tool in data pipeline development.

Dask: Dask is a flexible parallel computing library for Python that scales from single machines to clusters. It provides high-level interfaces for parallelizing and distributing data processing tasks, making it suitable for building scalable data pipelines.

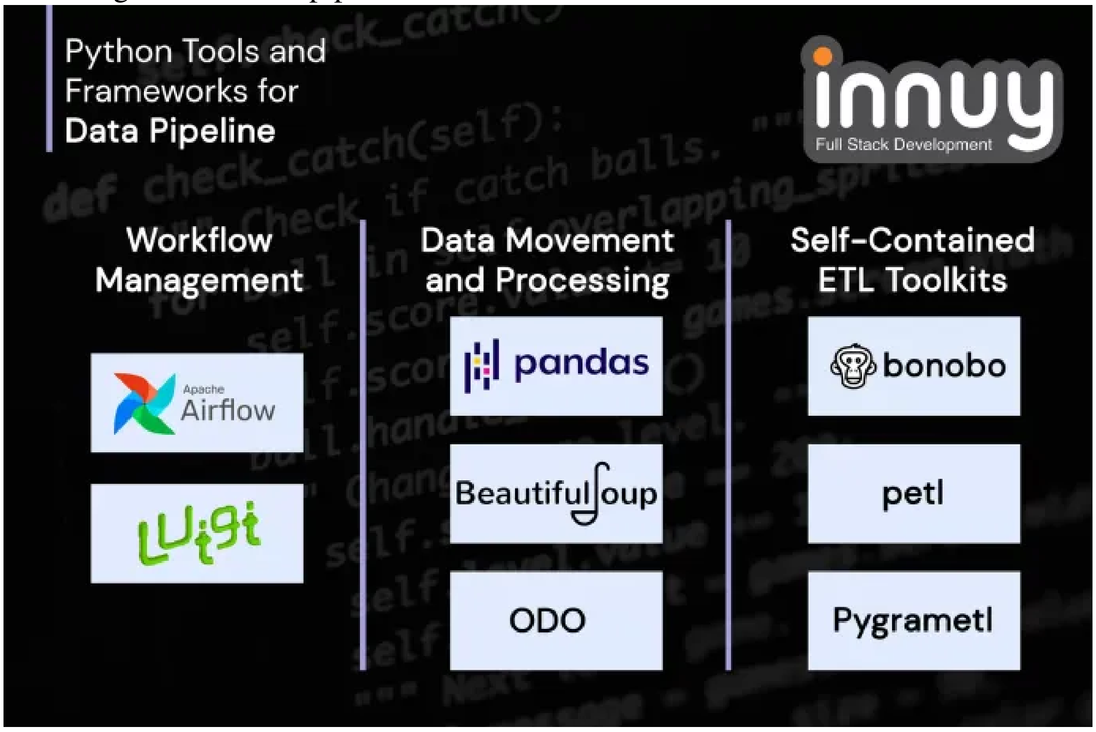


Figure 1.1.3 - Python Tools and Frameworks for Data Pipeline

The Case for a Dedicated Pipeline Library

The absence of native pipeline support in Python presents several challenges and opportunities for improvement. Firstly, the current landscape forces developers to cobble together solutions using disparate tools and libraries, leading to code that is harder to understand, maintain, and debug. Secondly, a dedicated pipeline library could provide a unified interface with built-in support for common pipeline operations such as data transformation, filtering, aggregation, and parallel execution. Such a library would simplify code and accelerate development, particularly in domains like data science and automation where pipelines are prevalent

Potential users

The field of data processing and pipeline tasks is vast and diverse, encompassing professionals from various backgrounds, including data engineers, data analysts, data scientists, software developers, DevOps engineers, and business analysts. These individuals all share a common need: the ability to efficiently manipulate and transform data from disparate sources to derive actionable insights and support decision-making processes within their organizations. However, despite Python's popularity and extensive library ecosystem, it lacks native support for pipeline operations, resulting in fragmented workflows and suboptimal solutions.

1. Data scientists:

Data scientists will profit from the initiative by using the library to optimize their workflow, automate tedious operations, and increase the reproducibility of their results. By offering a standardized framework for developing and running data pipelines, the project allows data scientists to focus on data analysis and interpretation rather than pipeline creation mechanics.

2. Machine Learning Engineers: Machine learning engineers can use the library to build and deploy large-scale machine learning models. By incorporating the library into their workflow, businesses can automate model training, evaluation, and deployment, minimizing manual labor and increasing productivity. The library's flexibility and scalability allow machine learning engineers to quickly experiment with various model topologies and hyperparameters.

3. Data engineers:

Data engineers can use the library to create reliable and scalable data pipelines for ingesting, analyzing, and converting enormous amounts of data. The library's capabilities for parallel processing and distributed computing makes it ideal for managing large data sets. The initiative will help data engineers by making it easier to create and maintain data pipelines, lowering time-to-delivery, and enhancing overall system reliability.

4. Software Developers:

Software developers can use the library to automate data-related operations and workflows in their applications. By offering a Python-native interface for designing and executing pipelines, the library allows developers to quickly include data processing capabilities into their applications. This allows developers to focus on developing core application logic while utilizing the library's capability for data manipulation and analysis.

5. Business analysts:

Business analysts can utilize the library to automate data preparation and analysis operations, allowing them to generate insights faster and more correctly. The library's support for data transformation and visualization enables exploratory data analysis and reporting, allowing analysts to confidently make data-driven decisions. By optimizing the data pipeline from raw data to actionable insights, the project enables business analysts to discover useful insights and drive corporate growth.

Facilitating Better Communication Among Stakeholders

One of the most significant benefits of a well-designed DSL is its ability to enhance communication among various stakeholders. By offering a common vocabulary that is precise within the domain context, a DSL reduces ambiguities and misunderstandings, enabling clearer articulation of requirements, challenges, and solutions. This shared language fosters a collaborative environment where developers, domain experts, and end-users can contribute more effectively to the project's success.

For instance, in a healthcare application, a DSL can enable clinicians, software developers, and regulatory experts to collaboratively design and implement software solutions that meet clinical needs while complying with healthcare regulations.

1.2 Description of the DSL

A well-designed DSL can also enhance communication among stakeholders, including developers, domain experts, and end-users, by providing a common vocabulary that is both precise and meaningful within the context of the domain. This can lead to better collaboration and a clearer understanding of requirements and solutions.

The role of Domain-Specific Languages (DSLs) is pivotal in modern software development, offering a focused and efficient way to address the unique challenges of various domains. By design, DSLs offer a high level of abstraction tailored to a specific field, whether it be web development, data science, machine learning, or another specialized area. This targeted approach allows developers and domain experts to communicate more effectively, leveraging a language that embodies the concepts, operations, and workflows intrinsic to their domain.

DSLs can be broadly categorized into two types: external DSLs, which are standalone languages with their own syntax and compiler or interpreter, and internal DSLs, which are embedded within a host general-purpose language, leveraging its syntax while providing domain-specific functionalities. Both types aim to simplify complex tasks, but they do so in ways that best suit their intended use cases and environments.

A DSL designed with deep domain insights can inherently guide users toward adopting best practices. In the context of machine learning, this might mean integrating data validation checks directly into the language, or offering simplified abstractions for complex model evaluation metrics. By encoding such practices into the language's syntax and libraries, a DSL not only educates its users but also helps prevent common errors, thereby enhancing the overall quality and effectiveness of the work produced.

The specificity of a DSL allows it to address challenges unique to its domain with unmatched precision. For instance, in the realm of real-time systems, timing constraints, and concurrency are critical issues. A DSL tailored for this domain can provide first-class language constructs for expressing time-bound operations or concurrent execution flows, making it significantly easier for developers to build robust real-time applications. This level of specificity in tackling domain challenges is difficult, if not impossible, to achieve with general-purpose languages without extensive boilerplate code or external libraries.

The Pipeline-Oriented Scripting Language (DSL) is a specialized programming language designed to facilitate the creation of automated workflows by organizing tasks into interconnected pipelines. At its core, DSL revolves around the concept of pipelines, where data flows seamlessly through a series of processing stages, each performing a specific operation on the input data. The language provides a concise and expressive syntax for defining these pipelines and orchestrating complex data processing tasks.

Key Features:

Pipeline Composition: DSL allows users to construct pipelines by chaining together individual processing stages. Each stage represents a discrete operation or transformation applied to the input data. Pipelines can be easily composed and modified to accommodate different processing requirements.

Modularity and Reusability: DSL promotes modularity by breaking down complex tasks into smaller, reusable components. Users can encapsulate common processing logic into standalone stages, making it easier to maintain and reuse code across different pipelines and projects.

Streamlined Workflow: By organizing tasks into pipelines, DSL streamlines the workflow of data processing and automation tasks. Pipelines enable sequential execution of processing stages, eliminating the need for manual intervention between each step and accelerating the execution of tasks.

Concise and Expressive Syntax: DSL offers a concise and expressive syntax for defining pipelines, making it easy to express complex data processing logic using minimal code. Built-in constructs such as pipe operators facilitate the chaining of processing stages, enhancing readability and maintainability of scripts.

Efficient Resource Utilization: DSL supports parallel execution of processing stages, allowing multiple tasks to run concurrently and utilize system resources efficiently. This parallelism maximizes CPU utilization and minimizes idle time, leading to improved performance and responsiveness of pipelines.

Interoperability and Integration: DSL seamlessly integrates with existing tools, libraries, and systems, enabling interoperability and data exchange. Users can leverage built-in mechanisms for interfacing with external programs, APIs, and data sources, facilitating seamless integration with third-party services and platforms.

1.3 Grammar

ANTLR, short for ANother Tool for Language Recognition, is a robust parser generator used to construct parsers, interpreters, compilers, and translators for various programming languages and domain-specific languages. It operates by taking a formal grammar of the language as input and producing a parser for that language in target languages such as Java, C, Python, and others. Its advantages include its language-agnostic nature, support for LL parsing allowing for more expressive grammars, automatic generation of Abstract Syntax Trees for parsed input, detailed error reporting facilitating easier debugging, seamless integration with IDEs and development tools, and a large and active community providing support and resources for users.

Lexical Considerations

All Decaf keywords are lowercase. Keywords and identifiers are case-sensitive. For example, ’if’ is a keyword, but ’IF’ is a variable name; ’name’ and ’Name’ are two different names referring to two distinct variables.

Comments are started by ’#’ and are terminated by the end of the line.

White space may appear between any lexical tokens. White space is defined as one or more spaces, tabs, page and line-breaking characters, and comments.

Keywords and identifiers must be separated by white space or a token that is neither a keyword nor an identifier. For example, ’thisfortrue’ is a single identifier, not three distinct keywords.

If a sequence begins with an alphabetic character or an underscore, then it, and the longest sequence of characters following it forms a token.

String literals are composed of <char>s enclosed in double quotes.

A character literal consists of a <char>enclosed in single quotes.

Numbers in Decaf are 32-bit signed. That is, decimal values between -2147483648 and 2147483647.

If a sequence begins with ’0x’, then these first two characters and the longest sequence of characters drawn from [0-9a-fA-F] form a hexadecimal integer literal.

If a sequence begins with a decimal digit (but not ’0x’), then the longest prefix of decimal digits forms a decimal integer literal.

A ’char’ is any printable ASCII character (ASCII values between decimal value 32 and 126, or octal 40 and 176) other than quote (”), single quote (’), or backslash (\), plus the 2-character sequences ’\”’ to denote quote, '\' to denote single quote, '\\' to denote backslash, '\t' to denote a literal tab, or '\n' to denote newline.

Reference Grammar

Notation	Meaning
`<foo>`	(in bold font) foo is a non-terminal
foo	foo is a terminal
`[x]`	zero or one occurrence of x
`x*`	zero or more occurrences of x
x^+	a comma-separated list of one or more x's
{}	large braces for grouping
|	separates alternatives

Table 1.3.1 - Context-Free Grammar Notations

```
< prog > := < pipeline_def > < var_def > < pipe_block > EOF;
< pipeline_flag > := 'pipe';
< arg > := INT | CHAR | STRING | FLOAT | VAR_NAME;
< args > := < arg > ( < comma > < space > * < arg > )*;
< function_args > := < left_par > < args > < right_par > ;
< function_name > := VAR_NAME;
< pipeline_def > := < pipeline_flag > < space > < function_name > < left_par > < right_par > < two_points >
                | < pipeline_flag > < space > < function_name > < function_args > < two_points > ;
< single_pipe_symbol > := '|>' ;
< double_pipe_symbol > := '||>' ;
< triple_pipe_symbol > := '|||>' ;
< pipe > := < tab > < single_pipe_symbol > < space > < function_name > < left_par > < right_par >
                | < tab > < double_pipe_symbol > < space > < function_name > < left_par > < right_par >
                | < tab > < tripple_pipe_symbol > < space > < function_name > < left_par > < right_par >
                | < tab > < single_pipe_symbol > < space > < function_name > < function_args >
                | < tab > < double_pipe_symbol > < space > < function_name > < function_args >
                | < tab > < tripple_pipe_symbol > < space > < function_name > < function_args > ;
< pipe_block > := < pipe > ( < pipe > )*;
< tab > := ' ';
< space > := ' ';
< left_par > := '(';
< right_par > := ')';
< comma > := ',';
< two_points > := ':';
< var_name > := VAR_NAME | CHAR;
< var_def > := < tab > < var_name > ;
NEWLINE : [ \r \n]+ - > skip;
INT : [0-9]+ ;
CHAR : [a-zA-Z] ;
STRING : ['][a-zA-Z]+['] ;
FLOAT : [0-9]+[.][0-9]+ ;
VAR_NAME : [a-zA-Z][a-zA-Z0-9]* ;
COMMENT : '#'  ( ' \r' | ' \n' )* - > skip;
```

Parsing example

```
[language=Python]
    pipe pipeLine(x):
    x
    |> firstOperation()
    ||> secondOperation()
```

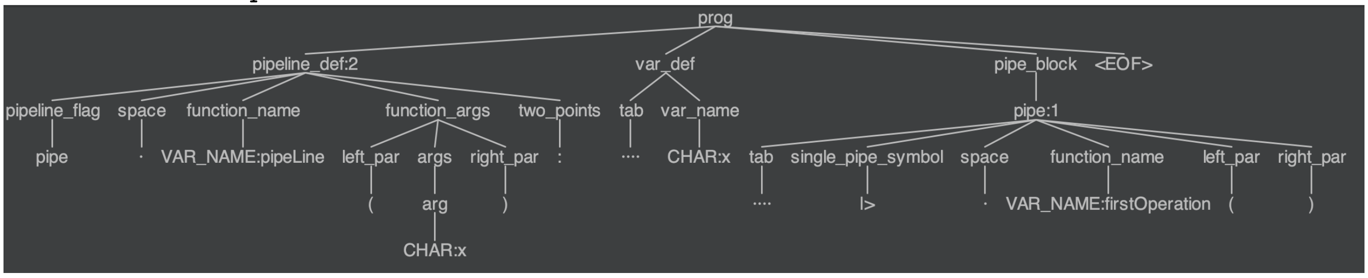


Figure 1.3.1 - Context-Free Grammar Notations

ANTLR Grammar definiton

ANTLR Grammar definiton: Parsing example Some code + tree screen

Semantics

The pipe operation is a mechanism for sequentially applying a series of functions to a given value. In the provided example, the pipe function facilitates this operation. The functions to be applied are specified as arguments to the pipe function, and they are executed in the order they are provided. The result of each function becomes the input for the next one, creating a sequential transformation of the original value.

Scope Rules

Declaration before Use: All identifiers, such as variables and functions, must be declared before they are used.

Method Invocation: A method can only be invoked or called by code that appears after its header. This rule ensures that methods are defined and their functionality is known before any attempts to use them in the program.

No Redefinition in the Same Scope: No identifier may be defined more than once in the same scope. This rule ensures that there is clarity and uniqueness in the naming of fields, methods, local variables, and formal parameters.

Locations

In terms of data types, can include both scalar and array types. Scalar locations might hold individual quotes or author names, typically represented as strings. On the other hand, array locations could accommodate collections of quotes or authors, potentially organized by theme, category, or other criteria. This flexibility allows for the effective management of diverse data structures, enhancing the capabilities of the DSL.

1.4 Lexer and Parser

Parser:

rogContext class: This class represents a context for the main program. It inherits from ParserRuleContext, which is a context for parsing rules. It contains methods for parsing pipeline definitions, variable definitions, pipeline blocks, and end of file (EOF) tokens.

prog() function: This function is responsible for parsing the main program. It initializes a ProgContext object, enters the parsing rule for a program (self.RULE_prog), and then tries to parse pipeline definitions, variable definitions, pipeline blocks, and EOF tokens.

Pipeline_flagContext class: This class represents a context for pipeline flags. It inherits from ParserRuleContext as well. However, in the provided code, it seems empty. There's no actual parsing logic defined inside it.

pipeline_flag() function: This function is supposed to parse pipeline flags, but currently, it only matches T__0 token, which appears to be a placeholder. It doesn't contain any meaningful parsing logic.

ArgContext class: This class represents a context for parsing individual arguments. It defines methods for parsing different types of arguments such as integers, characters, strings, floats, and variable names.

arg() function: This function is responsible for parsing individual arguments. It tries to consume a token corresponding to one of the allowed argument types. The _la = self._input.LA(1) line is used to peek ahead at the next token without consuming it to decide which type of argument to parse.

ArgsContext class: This class represents a context for parsing a sequence of arguments. It contains methods for parsing multiple arguments separated by commas and spaces.

args() function: This function parses a sequence of arguments. It starts by parsing the first argument using the arg() method. Then, it enters a loop to parse additional arguments if there are any. Inside the loop, it first checks if there's a comma token, then optionally consumes space tokens, and finally parses another argument using the arg() method. The loop continues until there are no more arguments left to parse.

Function_argsContext class: This class represents a context for parsing arguments within a function call. It contains methods for parsing the left parenthesis, arguments, and right parenthesis of a function call.

function_args() function: This function parses the arguments within a function call. It enters a rule for parsing the left parenthesis, then parses the arguments using the args() method, and finally parses the right parenthesis.

Function_nameContext class: This class represents a context for parsing the name of a function. It contains a method for parsing the variable name of the function.

function_name() function: This function parses the name of a function. It simply matches a variable name token.

Pipeline_defContext class: This class represents a context for parsing the definition of a pipeline. It contains methods for parsing various elements of a pipeline definition, such as pipeline flags, function names, function arguments, etc.

pipeline_def() function: This function parses the definition of a pipeline. It starts by parsing a pipeline flag using the pipeline_flag() method. Then, it expects a space, followed by a function name, left parenthesis, function arguments, right parenthesis, and a colon (two points).

pipeline_def() function: This function defines parsing rules for pipeline definitions. It first attempts to predict the alternative using adaptive prediction. If the prediction indicates the first alternative, it parses a pipeline flag, followed by a space, a function name, left parenthesis, right parenthesis, and a colon. If the prediction indicates the second alternative, it parses a pipeline flag, space, function name, function arguments, and a colon.

Single_pipe_symbolContext class: This class represents a context for parsing a single pipe symbol (|>).

single_pipe_symbol() function: This function parses a single pipe symbol.

Double_pipe_symbolContext class: This class represents a context for parsing a double pipe symbol (||>).

double_pipe_symbol() function: This function parses a double pipe symbol.

Tripple_pipe_symbolContext class: This class represents a context for parsing a triple pipe symbol (|||>).

tripple_pipe_symbol() function: This function parses a triple pipe symbol.

PipeContext class: This class represents a context for parsing a pipe within the pipeline definition. It contains methods for parsing various elements of a pipe, such as tabs, pipe symbols, function names, parentheses, and function arguments.

pipe() function: This function defines parsing rules for individual pipes within the pipeline block. It first attempts to predict the alternative using adaptive prediction. Then, it parses different combinations of tabs, pipe symbols, spaces, function names, parentheses, and function arguments based on the predicted alternative.

Pipe_blockContext class: This class represents a context for parsing a block of pipes within the pipeline. It contains methods for parsing multiple pipes.

pipe_block() function: This function parses a block of pipes within the pipeline. It starts by parsing the first pipe using the pipe() method and then enters a loop to parse additional pipes if there are any. The loop continues until there are no more pipes left to parse.

TabContext class: This class represents a context for parsing a tab character.

tab() function: This function parses a tab character.

SpaceContext class: This class represents a context for parsing a space character.

space() function: This function parses a space character.

The Var_nameContext class: defines a context for parsing variable names within the grammar. It inherits from ParserRuleContext and provides methods to initialize the context and handle exit actions. The var_name() function implements the parsing rule for variable names, consuming either VAR_NAME or CHAR tokens as specified in the grammar.

In the Var_defContext class: a context is established for parsing variable definitions according to the grammar rules. It inherits from ParserRuleContext and contains methods to initialize the context and manage exit actions. The var_def() function, within this context, implements the parsing rule for variable definitions, incorporating the tab() method to handle indentation and the var_name() method to parse the variable nam

Lexer:

Tokens: Tokens represent the smallest units of language syntax recognized by the lexer. Each token is assigned a numeric ID and may have a literal name and symbolic name.

Literal Names: These are literal symbols in the language, such as keywords and punctuation marks. 'pipe', ' |> ', '||>', ' |> ', ' ', '(', ')', ',', ':'

Symbolic Names: These are abstract names assigned to token types. NEWLINE, INT, CHAR, STRING, FLOAT, VAR_NAME, COMMENT

Rule Names: These define the lexer rules for recognizing patterns in the input text. Each rule corresponds to a regular expression or set of conditions for identifying tokens. Rules like NEWLINE, INT, CHAR, STRING, FLOAT, VAR_NAME, and COMMENT define patterns for matching specific types of tokens.

Channel Names: ANTLR allows tokens to be assigned to different channels for processing. This can be useful for separating tokens used for different purposes, such as separating comments or whitespace from significant language constructs.

Mode Names: Lexer modes allow for different sets of rules to be applied based on the lexer's current state. However, this lexer only has the default mode.

Constructor: The __init__ method initializes the lexer, setting up its input and output streams, checking the ANTLR version, and initializing the lexer's internal components.

Midterm 2

General Theory

Introduction to Decorators: Decorators in Python are a powerful feature that allows the modification or enhancement of functions or classes dynamically. They provide a convenient syntax for adding functionality to existing code without modifying its structure.

Role of Decorators in the Implementation: In the provided implementation of pipes, decorators play a crucial role in orchestrating the transformation process. The pipes decorator is applied to functions or classes that are intended to be used within pipelines, enabling the chaining of operations using the left and right shift operators.

Implementation

The revised implementation of pipes in Python introduces a novel approach using Abstract Syntax Trees (AST) manipulation. By leveraging the AST module, the implementation transforms function calls within pipelines into a more expressive and functional style, enhancing readability and flexibility.

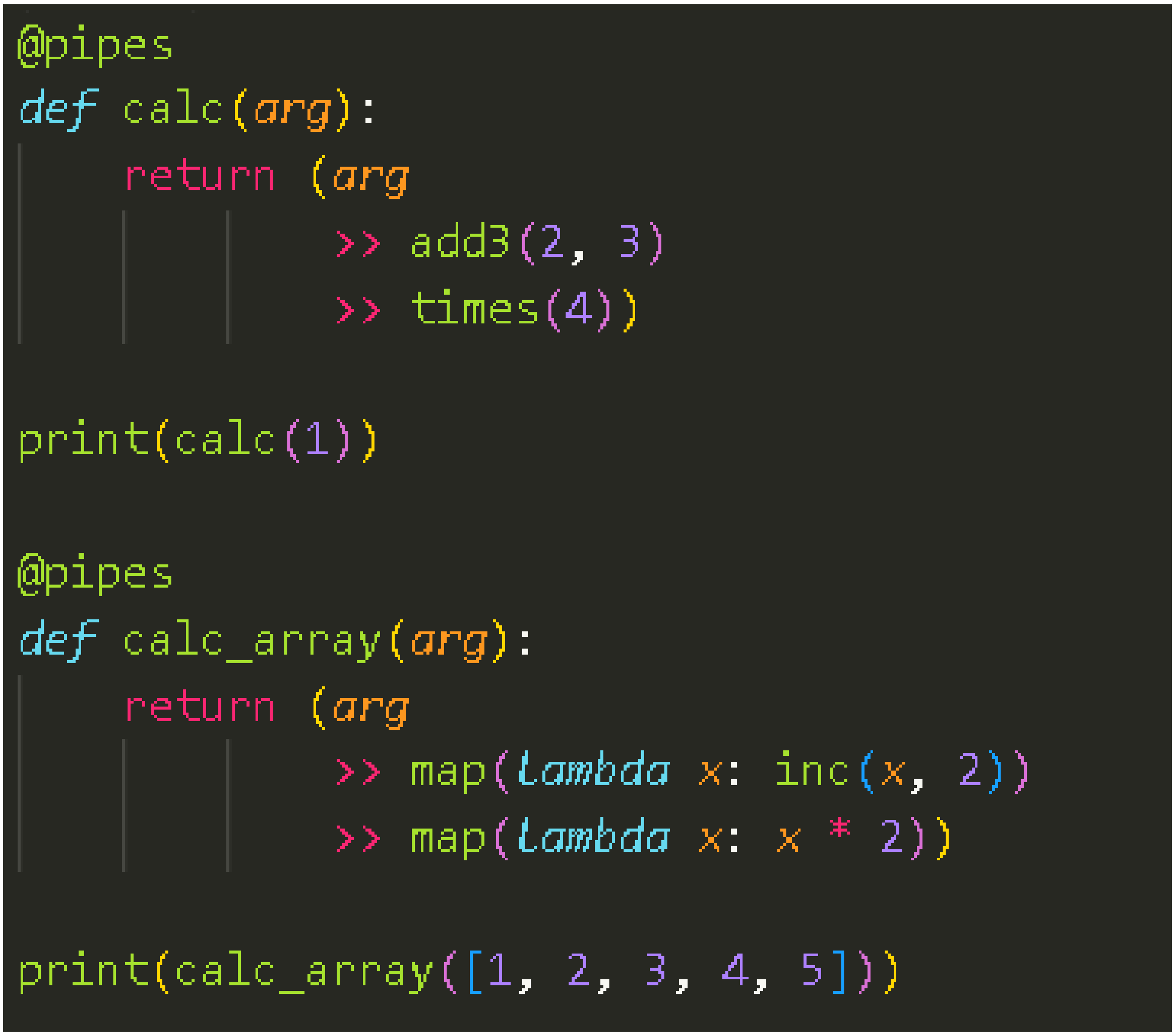
Key Components of the Implementation:

AST Transformation: The _PipeTransformer class serves as a NodeTransformer, responsible for visiting and transforming nodes in the AST. It specifically targets binary operations involving left and right shift operators (<< and >>) and converts them into function calls.

Decorator Logic: The pipes decorator, applied to functions or classes, orchestrates the transformation process. It identifies the relevant source code, parses it into an AST, applies transformations, and recompiles the modified code for execution.

Functional Composition: The implementation promotes functional composition by allowing functions to be chained together using the left and right shift operators. This enables concise and expressive pipelines for data processing.

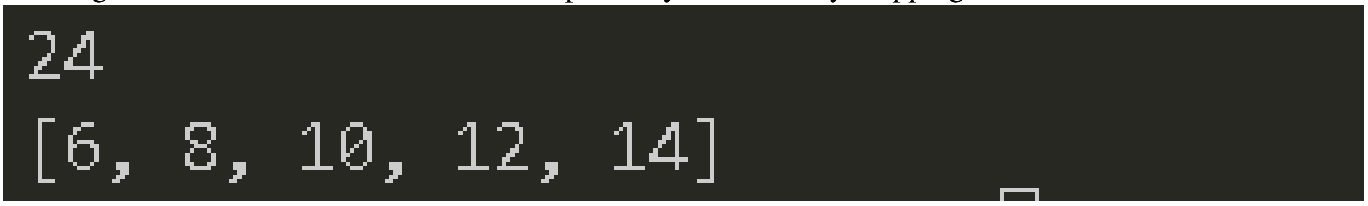
Integration with Existing Functions: The implementation seamlessly integrates with existing functions, enabling them to be used within pipelines without modification. Functions such as add, times, and map can be chained together using the pipe operators to perform complex data transformations.



```
@pipes
def calc(arg):
    return (arg
            >> add3(2, 3)
            >> times(4))

print(calc(1))

@pipes
def calc_array(arg):
    return (arg
            >> map(lambda x: inc(x, 2))
            >> map(lambda x: x * 2))

print(calc_array([1, 2, 3, 4, 5]))
```

Figure 1.4.1 - Example Usage

In this example, the calc function applies the add3 function with arguments 2 and 3, followed by the times function with argument 4, to the input argument arg. The calc_array function maps the inc function with argument 2 over each element of the input array, followed by mapping each element to its double.



```
24
[6, 8, 10, 12, 14]
```

Figure 1.4.2 - Example Result

Conclusions

In conclusion, the analysis presented in this report highlights the transformative potential of pipeline-oriented scripting languages to simplify data science processes. Through a comprehensive exploration of fundamental concepts, practical applications, and design features, it is clear that these languages offer a versatile and efficient approach to managing, processing, and automating data-intensive tasks.

The proposed Python pipeline language offers a clean and intuitive syntax, facilitating the construction of intricate data processing pipelines. The key benefit is its ability to streamline complex data workflows by organizing tasks into interconnected pipelines. By breaking operations down into modular building blocks, these languages enable users to develop flexible, scalable, and maintainable solutions tailored to their specific requirements. It is just one of many instruments which is developed to simplify complex tasks. It makes complex tasks more simple for a wider range of users.

However, it is important to recognize that implementing pipeline-oriented scripting languages is not without its challenges. Users may encounter issues related to performance optimization, resource utilization. Moreover, as data volumes continue to grow and new technologies emerge, pipeline-oriented scripting languages must evolve to meet the changing needs and requirements of the industry.

Bibliography

[1] What is a data pipeline — IBM https://www.ibm.com/topics/data-pipeline#:~:text=A%20data%20pipeline%20is%20a,usually%20undergoes%20some%20data%20processing.

[2] Hapke, H., & Nelson, C. (2020). Building machine learning pipelines. O'Reilly Media.

https://books.google.ie/books?id=H6_wDwAAQBAJ&printsec=frontcover&dq=O%27REILLY+Building+Machine+Learning+Pipelines+Automating+Model+Life+Cycles+with+TensorFlow+Hannes+Hapke+%26+Catherine+Nelson+Foreword+By+Aur%C3%A9lien+G%C3%A9ron&hl=&cd=1&source=gbs_api#v=onepage&q=O'REILLY%20Building%20Machine%20Learning%20Pipelines%20Automating%20Model%20Life%20Cycles%20with%20TensorFlow%20Hannes%20Hapke%20%26%20Catherine%20Nelson%20Foreword%20By%20Aur%C3%A9lien%20G%C3%A9ron&f=false

[3] Doris Xin, Hui Miao, Aditya Parameswaran, and Neoklis Polyzotis. 2021. Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities. In Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21). Association for Computing Machinery, New York, NY, USA, 2639–2652. https://doi.org/10.1145/3448016.3457566

[4] Leo Goodstadt, Ruffus: a lightweight Python library for computational pipelines, Bioinformatics, Volume 26, Issue 21, November 2010, Pages 2778–2779, https://doi.org/10.1093/bioinformatics/btq524

[5] Github of the team https://github.com/RodionClepa/PBL_TEAM5_Pipe