

# KodLang Programming Language Documentation

## Table of Contents

1. Introduction
  2. Basic Syntax
  3. Data Types
  4. Variables
  5. Operators
  6. Expressions
  7. Built-in Functions
  8. Pipeline Operations
  9. Data Management
  10. System Commands
- 

## Introduction

KodLang is a domain-specific language designed for data processing and manipulation. It provides a rich set of mathematical, statistical, and string operations, along with powerful data pipeline capabilities.

---

## Basic Syntax

### Statement Termination

All statements in KodLang must end with a semicolon (;).

```
$x <- 10;  
$y <- 20;
```

### Comments

Currently not explicitly documented in the lexer, but the language focuses on executable statements.

### Identifiers

- **Variable identifiers:** Begin with \$ followed by letters, digits, or underscores
    - Example: \$myVar, \$data\_1, \$result
  - **Function/Named identifiers:** Begin with a letter, contain only letters
    - Example: download, upload, fetch
-

## Data Types

### Primitive Types

**Numbers** Integer or floating-point numbers.

```
$integer <- 42;  
$decimal <- 3.14;  
$negative <- -10;
```

**Strings** Text enclosed in single (' ) or double (" ) quotes.

```
$name <- "John Doe";  
$greeting <- 'Hello World';
```

**Booleans** Logical values: true or false.

```
$isActive <- true;  
$isComplete <- false;
```

**Null** Represents an absence of value.

```
$empty <- null;
```

### Complex Types

**Arrays/Lists** Collections of values enclosed in square brackets.

```
$numbers <- [1, 2, 3, 4, 5];  
$mixed <- [1, "text", true, null];  
$nested <- [[1, 2], [3, 4]];
```

**DataFrames** Specialized column-based data structures accessed through subscript notation.

```
$df <- fetch(CSV, "data.csv");  
$column <- $df[columnName];  
$columns <- $df[[col1, col2, col3]];
```

---

## Variables

### Assignment

Use the assignment operator <- to assign values to variables.

```
$x <- 5;  
$name <- "Alice";  
$list <- [1, 2, 3];
```

## Array Destructuring

Assign multiple values from a list to multiple variables.

```
$data <- [10, 20, 30];
[$a, $b, $c] <- $data;
```

## Subscript Assignment

Modify elements within arrays or dataframes.

```
$list <- [1, 2, 3];
$list[0] <- 100;

$df <- fetch(CSV, "data.csv");
$df[columnName] <- newValue;
```

---

# Operators

## Arithmetic Operators (Direct Use Only)

These operators work on regular variables but NOT in pipelines.

- + Addition
- - Subtraction
- \* Multiplication
- / Division
- \*\* Power/Exponentiation

```
$sum <- 5 + 3;
$product <- 4 * 7;
$power <- 2 ** 8;
```

**Note:** For DataFrame operations, use pipeline functions like `abs`, `sqrt`, `pow`, etc. instead of arithmetic operators.

## Comparison Operators

- = Equal to
- <> Not equal to
- < Less than
- <= Less than or equal to
- > Greater than
- >= Greater than or equal to

```
$isEqual <- 5 == 5;
$isGreater <- 10 > 5;
```

## Logical Operators

- `and` Logical AND
- `or` Logical OR

```
$result <- true and false;  
$condition <- (5 > 3) or (2 < 1);
```

## Special Operators

- `<-` Assignment
  - `,` Comma (separator)
  - `.` Dot (decimal point)
  - `:` Colon (used in pipelines)
- 

## Expressions

### Unary Expressions

Negation operator applied to numbers.

```
$negative <- -42;
```

### Binary Expressions

Combine two values with an operator.

```
$result <- ($a + $b) * $c;
```

### Subscript Access

Access elements from arrays or dataframes.

```
$firstElement <- $list[0];  
$character <- $text[3];  
$column <- $dataframe[columnName];
```

### Function Calls

Invoke data management functions with arguments.

```
$data <- fetch "CSV", "data.csv";  
export $results, "CSV", "output.csv";  
$connection <- connect "mysql", "mydb", "localhost", 3306, "user", "pass";
```

**Important Notes:** - Function calls do NOT use parentheses. Arguments are space-separated. - Mathematical, statistical, and string operations (like `abs`, `sqrt`, `upper`, etc.) can ONLY be used in pipelines with DataFrames, not directly on variables.

---

## Built-in Functions

KodLang functions are divided into two categories:

### 1. Pipeline Operations (DataFrame Only)

These operations work **only on DataFrames** through pipelines. They cannot be used directly on regular variables.

### 2. Data Management Functions

These functions work with regular data types and can be called directly.

---

## Pipeline Operations (DataFrame Only)

**Important:** All mathematical, statistical, string manipulation, and data transformation operations listed below can ONLY be used within pipelines and applied to DataFrames. They do NOT work on regular variables.

### Mathematical Operations

#### Basic Math

- `abs` - Absolute value
- `sqrt` - Square root
- `pow exponent` - Power
- `exp` - Exponential ( $e^x$ )
- `log` - Natural logarithm
- `log10` - Base-10 logarithm

#### Usage in Pipeline:

```
$pipe <- pipeline {  
  abs "columnName":  
  sqrt "columnName":  
  pow "columnName", 2:  
};  
$result <- apply $pipe, $dataframe;
```

### Trigonometric Operations

- `sin` - Sine
- `cos` - Cosine
- `tan` - Tangent
- `asin` - Arcsine

- **acos** - Arccosine
- **atan** - Arctangent

#### **Usage in Pipeline:**

```
$pipe <- pipeline {
  sin "angle":
  cos "angle":
};
```

#### **Hyperbolic Operations**

- **sinh** - Hyperbolic sine
- **cosh** - Hyperbolic cosine
- **tanh** - Hyperbolic tangent

#### **Rounding Operations**

- **ceil** - Round up
- **floor** - Round down
- **round** - Round to nearest

#### **Usage in Pipeline:**

```
$pipe <- pipeline {
  ceil "price":
  floor "quantity":
  round "average":
};
```

#### **Other Math Operations**

- **clamp min, max** - Restrict value to range
- **sign** - Sign of number (-1, 0, 1)
- **mod divisor** - Modulo operation
- **deg** - Convert radians to degrees
- **rad** - Convert degrees to radians
- **factorial** - Factorial
- **root n** - Nth root

#### **Usage in Pipeline:**

```
$pipe <- pipeline {
  clamp "score", 0, 100:
  mod "value", 5:
  deg "radians":
};
```

## Statistical Operations

- `sum` - Sum of values
- `mean` - Average
- `median` - Median value
- `mode` - Most frequent value
- `count` - Count elements
- `product` - Product of values

### Usage in Pipeline:

```
$pipe <- pipeline {  
  sum "sales":  
  mean "temperature":  
  median "ages":  
};
```

## String Operations

- `length` - String length
- `upper` - Convert to uppercase
- `lower` - Convert to lowercase
- `trim` - Remove whitespace
- `concat str` - Concatenate strings
- `substring start, end` - Extract substring
- `replace old, new` - Replace text
- `indexof search` - Find position
- `startswith prefix` - Check prefix
- `endswith suffix` - Check suffix
- `split` - Split into array
- `join separator` - Join array elements
- `reverse` - Reverse string
- `contains search` - Check if contains
- `repeat count` - Repeat string
- `toString` - Convert to string

### Usage in Pipeline:

```
$pipe <- pipeline {  
  upper "name":  
  trim "description":  
  substring "text", 0, 10:  
  replace "oldValue", "newValue":  
};
```

## Logical Operations

- `equals value` - Check equality
- `not` - Logical negation

#### **Usage in Pipeline:**

```
$pipe <- pipeline {  
  equals "status", "active":  
  not "isDeleted":  
};
```

#### **Type Checking Operations**

- `type` - Get type name
- `isnumber` - Check if number
- `isstring` - Check if string
- `islist` - Check if list
- `isbool` - Check if boolean

#### **Usage in Pipeline:**

```
$pipe <- pipeline {  
  type "value":  
  isnumber "amount":  
};
```

#### **Collection Operations**

- `distinct` - Remove duplicates
- `take n` - Take first n elements
- `skip n` - Skip first n elements
- `fill value, defaultValue` - Fill missing values

#### **Usage in Pipeline:**

```
$pipe <- pipeline {  
  distinct "category":  
  take "results", 10:  
  skip "data", 5:  
};
```

#### **Utility Operations**

- `print` - Output value to console
- `coalesce default` - Return first non-null value

#### **Usage in Pipeline:**

```
$pipe <- pipeline {  
  print "debug_column":  
  coalesce "value", 0:  
};
```

---

## Data Management Functions

These functions work with regular data types and are NOT restricted to pipelines.

---

## Pipeline Operations

Pipelines allow chaining operations in a sequential manner using the `pipeline` keyword. Functions in pipelines can take column names as string arguments.

### Syntax

```
$result <- pipeline {  
  functionName "columnName":  
  functionName "columnName":  
  functionName "columnName":  
};
```

**Important Notes:** - Functions in pipelines do NOT use parentheses - Arguments (like column names) are written as space-separated strings - Each operation ends with a colon : - The pipeline is enclosed in curly braces {}

### Example

```
$processed <- pipeline {  
  abs "amount":  
  sqrt "amount":  
  round "amount":  
};  
  
$dataResult <- apply $processed, $dataframe;
```

### Complex Pipeline Example

```
$df <- fetch "CSV", "Iris.csv";  
  
$pipe <- pipeline {  
  cos "PetalLengthCm":  
  sin "SepalWidthCm":  
  sinh "SepalLengthCm":  
  cos "SepalWidthCm":  
  sinh "SepalLengthCm":  
  cosh "SepalWidthCm":  
  exp "SepalLengthCm":  
  exp "SepalWidthCm":  
  log "SepalLengthCm":
```

```

    exp "SepalWidthCm":
    sinh "SepalLengthCm":
    sin "SepalWidthCm":
};

$res <- apply $pipe, $df;

```

### Using Pipelines with Apply

The `apply` function executes a pipeline on a dataframe.

```

$pipeline <- pipeline {
  trim "name":
  upper "name":
  length "name":
};

$result <- apply $pipeline, $df;

```

## Data Management Functions

These functions work with regular data types and are NOT restricted to pipelines.

### Data Source Functions

#### Connect to Database

```
$connection <- connect databaseType, databaseName, host, port, userOrDatacentre, password;
```

Supported database types: - Standard databases: requires `user` and `password`  
 - Cassandra: requires `dataCentre` instead

**Fetch Data** Load data from various sources.

```
$data <- fetch dataType, dataSource;
$dataWithQuery <- fetch dataType, dataSource, query;
```

Supported data types: - CSV - EXCEL - JSON - Database connections

Examples:

```
$csvData <- fetch "CSV", "data.csv";
$dbData <- fetch "DATABASE", $connection, "SELECT * FROM users";
```

**Export Data** Export dataframes to files.

```
$exported <- export $dataframe, dataType, fileName;
```

Example:

```
export $results, "CSV", "output.csv";
```

**Download Files** Download generated files.

```
download fileName;
```

**Remove Data** Remove variables or dataframes from session.

```
remove dataName;
```

### Apply Operations

Execute pipeline operations on dataframes. This is the ONLY way to use mathematical, statistical, and string operations.

```
$result <- apply $pipeline, $dataframe;
```

**Example:**

```
# Create a pipeline with operations
$transformPipeline <- pipeline {
  abs "value": 
  sqrt "value": 
  round "value": 
};
```

---

```
# Apply to dataframe (this is required - operations don't work on regular variables)
$result <- apply $transformPipeline, $dataframe;
```

## System Commands

**User Management**

**Create User**

```
user "create", name, maxSession, maxProcessPerSession, maxMemoryPerProcess, maxStorageUsage,
```

**Remove User**

```
user "remove", userId;
```

**Task Management**

**Cancel Task**

```
task "cancelTask", taskId, userId;
```

**List Tasks**

```
task "taskList", sessionId;
```

**Session Management****Terminate Session**

```
session "terminateSession", sessionId;
```

**Server Control****Stop Server**

```
stop;
```

---

**Real-World Example: Iris Dataset Processing**

```
# Load the famous Iris dataset
$df <- fetch "CSV", "Iris.csv";

# Create a complex mathematical transformation pipeline
$pipe <- pipeline {
  cos "PetalLengthCm": 
  sin "SepalWidthCm": 
  sinh "SepalLengthCm": 
  cos "SepalWidthCm": 
  sinh "SepalLengthCm": 
  cosh "SepalWidthCm": 
  exp "SepalLengthCm": 
  exp "SepalWidthCm": 
  log "SepalLengthCm": 
  exp "SepalWidthCm": 
  sinh "SepalLengthCm": 
  sin "SepalWidthCm": 
};

# Apply the transformations
$res <- apply $pipe, $df;

# Export the transformed data
export $res, "CSV", "iris_transformed.csv";
```

---

## Asynchronous Operations

KodLang supports asynchronous operations through `CompletableFuture`. When assigning async results:

```
$asyncResult <- someAsyncOperation();
$x <- $asyncResult; # Will resolve when complete
```

The language automatically handles async resolution, allowing you to work with futures transparently.

---

## Error Handling

The language uses exceptions for error conditions:

- Division by zero returns `null`
- Index out of bounds throws exceptions
- Type mismatches throw exceptions
- Unknown tokens halt lexical analysis

---

## Best Practices

1. Always terminate statements with semicolons
  2. Use descriptive variable names with the `$` prefix
  3. Leverage pipelines for sequential data transformations
  4. Use type checking functions before operations on dynamic data
  5. Handle null values with `coalesce` function
  6. Use array destructuring for clean multi-assignment
  7. Group related operations in pipelines for readability
- 

## Language Limitations

- No user-defined functions (only built-in functions)
- No control flow statements (`if/else`, loops)
- No custom types or classes
- Limited to single-file execution
- Statements must end with semicolons
- **Mathematical, statistical, and string operations work ONLY on DataFrames through pipelines** - they cannot be used on regular variables
- Regular variables support only basic arithmetic operators (`+, -, *, /, **`) and comparison operators
- For data transformations, you must: load data into DataFrame → create pipeline → apply to DataFrame