

Matthew Morris
Professor Kontothanassis
DS210
11 April 2025

DS210 HW8 Journal

Question 1

```
use std::collections::HashMap;

#[derive(Debug, Clone)]
enum ColumnVal {
    One(String),
    Two(bool),
    Three(f64),
    Four(i64),
}
```

- This enum defines the possible types of data stored in the dataframe. I assigned one to string, two to booleans, three to f64 and four to i64. This is as described in the instructions.

```
#[derive(Debug)]
struct DataFrame {
    columns: HashMap<String, Vec<ColumnVal>>,
    labels: Vec<String>,
}
```

- This defines the main dataframe structure, defining specifically the columns as a hashmap of strings and column values, as well as labels, which is a vector of strings storing column names.

```
impl DataFrame {
    fn new() -> Self {
        DataFrame {
            columns: HashMap::new(),
            labels: Vec::new(),
        }
    }
}
```

- Implements methods for the dataframe.
- New() initializes a new empty dataframe
- Dataframe contains the column information in a hashmap, labels stored as vector

```
fn read_csv(&mut self, path: &str, types: &Vec<u32>) {
    let mut rdr = csv::ReaderBuilder::new()
        .delimiter(b',')
        .has_headers(true)
        .flexible(true)
```

```
.from_path(path)
.unwrap();
```

This function, aptly titled `read_csv` takes and reads the csv file, saving it as a mutable “rdr.” I obtained this function from a combination of chatgpt and the notes.

- Uses csv crate to parse file
- Takes a path and a vector of types (1-4 assigned previously) indicating the data type for each column.

```
let headers = rdr.headers().unwrap();
self.labels = headers.iter().map(|s| s.to_string()).collect();

for result in rdr.records() {
    let r = result.unwrap();

    for (i, elem) in r.iter().enumerate() {
        let label: &String = &self.labels[i];
        let val = match types[i] {
            1 => ColumnVal::One(elem.to_string()),
            2 => {
                let bool_val = match elem.trim().to_lowercase().as_str() {
                    "true" => true,
                    "false" => false,
                    _ => false,
                };
                ColumnVal::Two(bool_val)
            }
            3 => ColumnVal::Three(elem.parse::<f64>().unwrap()),
            4 => ColumnVal::Four(elem.parse::<i64>().unwrap()),
            _ => ColumnVal::One(elem.to_string()),
        };
    }
};
```

This is the data processing as a part of reading the csv file. For this, it the function unwraps the header, reading the headers and storing them as labels (stored in the enum previously assigned). Then it processes each row of data using the `enumerate` function. Then, it converts each value to the appropriate type based on the “types” vector. Stores the converted values in the data frame’s columns. For ex. If it is a boolean, it will be stored in the respective dataframe column. Additionally, for booleans, it turns the string value of a boolean into a true boolean, and with the edge case assigning to false -> this is for functionality, but realistically it should be NaN.

```
fn print(&self) {
    let mut col_widths: Vec<usize> = self.labels.iter().map(|label|
label.len()).collect();
```

Implements a method to print the dataframe. Calculates column widths for proper formatting.

```
fn print(&self) {
    let mut col_widths: Vec<usize> = self.labels.iter().map(|label|
label.len()).collect();

    for (i, label) in self.labels.iter().enumerate() {
        if let Some(col) = self.columns.get(label) {
            for val in col {
                let val_len = match val {
                    ColumnVal::One(s) => s.len(),
                    ColumnVal::Two(b) => b.to_string().len(),
                    ColumnVal::Three(f) => format!("{:.1}", f).len(),
                    ColumnVal::Four(i) => i.to_string().len(),
                };
                col_widths[i] = col_widths[i].max(val_len);
            }
        }
    }

    for (i, label) in self.labels.iter().enumerate() {
        print!("{:<width$} ", label, width = col_widths[i]);
    }
    println!();

    if let Some(first_col) = self.columns.get(&self.labels[0]) {
        for i in 0..first_col.len() {
            for (j, label) in self.labels.iter().enumerate() {
                if let Some(col) = self.columns.get(label) {
                    match &col[i] {
                        ColumnVal::One(s) => print!("{:<width$} ", s, width =
col_widths[j]),
                        ColumnVal::Two(b) => print!("{:<width$} ", b, width =
col_widths[j]),
                        ColumnVal::Three(f) => {
                            print!("{:<width$.1} ", f, width = col_widths[j])
                        }
                        ColumnVal::Four(i) => print!("{:<width$} ", i, width =
col_widths[j]),
                    }
                }
            }
        }
    }
}
```

```

println!();
    }
}
}

```

It calculates the column widths for proper formatting. I also used chatgpt for this. Previously I just used spaces as buffers, however since the column labels were longer than the data stored beneath in the print statement, it would appear with the data misaligned, so I wanted to align it. It prints each row of data with consistent formatting, then handles different data types appropriately.

```

fn add_column(&mut self, label: String, data: Vec<ColumnVal>) -> DataFrame {
    let mut new_df = DataFrame::new();
    new_df.labels = self.labels.clone();
    new_df.columns = self.columns.clone();

    new_df.labels.push(label.clone());
    new_df.columns.insert(label, data);

    new_df
}

```

The `add_column()` function does exactly what it says, it adds a column. From here it initializes a new dataframe, cloning a label and data, and then adds the new column with its data. From there it returns the new dataframe.

```

fn merge_frame(&self, other: &DataFrame) -> DataFrame {
    let mut new_df = DataFrame::new();
    new_df.labels = self.labels.clone();

    for label in &self.labels {
        let mut combined = self.columns[label].clone();
        combined.extend(other.columns[label].clone());
        new_df.columns.insert(label.clone(), combined);
    }

    new_df
}

```

Combines two dataframes vertically. From there, it creates a new dataframe. It takes itself and the other dataframe that was previously created. From there it iterates through each label, combining with the previous dataframe to make this new combined dataframe.

```

fn find_columns(&self, labels: &[String]) -> DataFrame {

```

```

    let mut new_df = DataFrame::new();
    for label in labels {
        new_df.labels.push(label.clone());
        new_df
            .columns
            .insert(label.clone(), self.columns[label].clone());
    }
    new_df
}

```

This function creates a new dataframe with only the specified columns, similar to the pandas functionality in python. It copies the selected columns/labels and their corresponding data. First it iterates through all labels, then it pushes to the new data frame, cloning the data, and inserting the columns.

```

fn restrict_columns(&self, labels: &[String]) -> DataFrame {
    let mut new_df = DataFrame::new();
    new_df.labels = labels.to_vec();

    for label in labels {
        new_df
            .columns
            .insert(label.clone(), self.columns[label].clone());
    }
    new_df
}

```

Very similar to find_columns, but takes ownership of the labels. It creates a new dataframe with only specified columns.

```

fn filter(&self, label: &str, operation: fn(&ColumnVal) -> bool) -> DataFrame {
    let mut new_df = DataFrame::new();
    new_df.labels = self.labels.clone();

    let indices: Vec<usize> = self.columns[label]
        .iter()
        .enumerate()
        .filter(|(&, val)| operation(val))
        .map(|(i, _)| i)
        .collect();

    for label in &self.labels {
        let filtered_data: Vec<ColumnVal> = indices

```

```

        .iter()

        .map(|&i| self.columns[label][i].clone())

        .collect();

    new_df.columns.insert(label.clone(), filtered_data);
}

new_df
}

```

This function takes itself, label, and operation (which is what it filters), returning a dataframe. From there it initializes a dataframe, and then clones labels, initializing them as `new_df.labels`. It assigns indices to a vector. It uses a function to determine which rows to keep, and preserves all columns for the filtered rows. Within the indices, it iterates through (`iter`, `enumerate`) then filters based on the function and operation (`.filter[(_, val) | operation(val)]`). It then maps it to the specific point in the table and collects the values. From there it iterates through the labels.

Question 2

```
fn column_op(
    &self,
    labels: &[String],
    op: fn(&[Vec<ColumnVal>]) -> Vec<ColumnVal>,
) -> Vec<ColumnVal> {
    let columns: Vec<Vec<ColumnVal>> = labels
        .iter()
        .map(|label| self.columns[label].clone())
        .collect();
    op(&columns)
}
```

For `column_op()`, The goal is to perform operations on specific columns. It takes the function that operates on vectors of `columnval`, returning the result of the operation. The code for the assigning is similar to previous with `“.iter()`

```
.map(|label| self.columns[label].clone())
.collect();"
```

```
fn median(&self, label: &str) -> f64 {
    let result = self.column_op(&[label.to_string()], |columns| {
        let mut values: Vec<f64> = columns[0]
            .iter()
            .filter_map(|val| {
                if let ColumnVal::Three(f) = val {
                    Some(*f)
                } else {
```

```

        None
    }
})
.collect();

values.sort_by(|a, b| a.partial_cmp(b).unwrap());
let len = values.len();
let median = if len % 2 == 0 {
    (values[len / 2 - 1] + values[len / 2]) / 2.0
} else {
    values[len / 2]
};
vec![ColumnVal::Three(median)]
});

if let ColumnVal::Three(median) = result[0] {
    median
} else {
    0.0
}
}

```

This function calculates the median of a numeric column, outputting an f64. First, it uses `column_op` to get the column we want to calculate the median for. Then, it extracts the numeric values from the column. Then it only keeps the values that are f64's (only ones you can take median for). Then it sorts the values to find median. Then it calculates median based on number of values. For even number of values it averages of the middle two. If it is odd, then it takes middle value. Then returns the result wrapped in `columnval::three`. It then extracts the value from the result.

```

fn sub_columns(&self, label1: &str, label2: &str) -> Vec<i64> {
    let result = self.column_op(&[label1.to_string(), label2.to_string()],
|columns| {
        let mut differences = Vec::new();
        for i in 0..columns[0].len() {
            if let (ColumnVal::Four(val1), ColumnVal::Four(val2)) =
                (&columns[0][i], &columns[1][i])
            {
                differences.push(ColumnVal::Four(val1 - val2));
            }
        }
        differences
    })
}

```

```

    });

    result
        .into_iter()
        .filter_map(|val| {
            if let ColumnVal::Four(i) = val {
                Some(i)
            } else {
                None
            }
        })
        .collect()
    }
}

```

First, it uses `column_op` to get both columns we want to subtract. Then it iterates through each row. Then it checks if both values are integers. Then it subtracts the values and stores the result. It then converts the results back to `i64` values.

Main function:

```

fn main() {
    let types = vec![1, 4, 3, 4, 4, 2];

    let mut df1 = DataFrame::new();
    df1.read_csv("src/data.csv", &types);

    println!("\nOriginal DataFrame:");
    df1.print();

    let new_data = vec![
        ColumnVal::Two(false), // Kareem
        ColumnVal::Two(false), // Karl
        ColumnVal::Two(true),  // LeBron
        ColumnVal::Two(false), // Kobe
        ColumnVal::Two(false), // Michael
    ];

    let df2 = df1.add_column("IsAllStar".to_string(), new_data);
    println!("\nDataFrame with new column:");
    df2.print();
}

```


Initializes the types as vectors (1, 4, 3, 4, 4, 2) which are assigned initially in the dataframe. They are as follows:

- 1: String (Name)
- 4: i64 (Number)
- 3: f64 (PPG - Points Per Game)
- 4: i64 (YearBorn)
- 4: i64 (TotalPoints)
- 2: bool (LikesPizza)

It then creates a new dataframe and loads data from data.csv

Then I added a all-star column which assigns each of players if they are allstars or not (I assigned lebron as an allstar (sorry!)). It creates a new boolean column, the other players other than lebron are marked false for all-star. Then it creates a new dataframe with this column.

```
let mut df3 = DataFrame::new();
df3.read_csv("data.csv", &types);
let merged_df = df1.merge_frame(&df3);
println!("\nMerged DataFrame:");
merged_df.print();
```

I then Merge dataframes. I create a second dataframe from the same csv, merging the two dataframe vertically. This doubles the # of rows. All columns are preserved in the merge.

```
let selected_columns = df1.find_columns(&["Name".to_string(), "PPG".to_string()]);
println!("\nSelected columns:");
selected_columns.print();
let restricted_df = df1.restrict_columns(&["Name".to_string(),
"TotalPoints".to_string()]);
println!("\nRestricted columns:");
restricted_df.print();
```

I shows two ways to select columns, firstly, find_columns: Creates new DataFrame with just Name and PPG, then restrict_columns: Creates new DataFrame with just Name and TotalPoints. Both methods produce similar results but with different ownership semantics.

```
let high_scorers = df1.filter("PPG", |val| {
    if let ColumnVal::Three(ppg) = val {
        *ppg > 20.0
    } else {
        false
    }
});
println!("\nPlayers with PPG > 20:");
high_scorers.print();
```

I then filter the dataframe to show only players with ppg >20, using closure to define the filtering condition. I then only keep rows where the ppg value is > 20.

```
let median_ppg = df1.median("PPG");
println!("\nMedian PPG: {:.2}", median_ppg);
```

Then I calculate the median ppg for all players. I return a singular f64 value representing the median.

```
let differences = df1.sub_columns("TotalPoints", "YearBorn");
println!("\nTotalPoints - YearBorn differences:");
for (i, diff) in differences.iter().enumerate() {
    if let ColumnVal::One(name) = &df1.columns["Name"][i] {
        println!("{}", name, diff);
    }
}
```

I then calculate the difference between total points and yearborn for each player. I print each player's name with their calculated difference. Then I show how to perform operations between columns.

Tests

I then created tests.

```
#[test]
fn test_read_csv() {
    let mut df = DataFrame::new();
    let types = vec![1, 4, 3, 4, 4, 2]; // Name, Number, PPG, YearBorn,
    TotalPoints, LikesPizza
    df.read_csv("src/data.csv", &types);

    assert_eq!(df.labels.len(), 6);
    assert!(df.labels.contains(&"Name".to_string()));
    assert!(df.labels.contains(&"PPG".to_string()));

    if let Some(name_col) = df.columns.get("Name") {
        assert!(matches!(name_col[0], ColumnVal::One(_)));
    }
    if let Some(ppg_col) = df.columns.get("PPG") {
        assert!(matches!(ppg_col[0], ColumnVal::Three(_)));
    }
}
```

Test 1: tests if the csv functionality works. Firstly it creates a new dataframe, then reads the data from a csv file. It verifies the number of columns, and the specific names "name" and "ppg" which I manually input from the csv file to check. Then it verifies the correctness of the data types in the respective columns.

```
#[test]
fn test_filter_ppg() {
    let mut df = DataFrame::new();
    let types = vec![1, 4, 3, 4, 4, 2];
    df.read_csv("src/data.csv", &types);

    let filtered = df.filter("PPG", |val| {
        if let ColumnVal::Three(ppg) = val {
            *ppg > 20.0
        } else {
            false
        }
    });

    if let Some(ppg_col) = filtered.columns.get("PPG") {
        for val in ppg_col {
            if let ColumnVal::Three(ppg) = val {
                assert!(*ppg > 20.0);
            }
        }
    }
}
```

Test 2: This creates and loads a dataframe and applies a filter with ppg>20. I then verified that all the remaining players in the filtered dataframe have ppg>20. It uses pattern matching to safely extract and check numeric values.

```
#[test]
fn test_median_calculation() {
    let mut df = DataFrame::new();
    let types = vec![1, 4, 3, 4, 4, 2];
    df.read_csv("src/data.csv", &types);

    let median_ppg = df.median("PPG");
    assert!(median_ppg > 0.0);
}
```

```
    assert!(median_ppg >= 0.0 && median_ppg <= 50.0);  
  }  
}
```

Test 3: The third test creates and loads a dataframe, then calculates the median ppg. It verifies the following: if the median >0, if it is within reasonable bounds (<50).

Output:

With all of this said, this is my output based on my functions and main function. I wanted to showcase each of the functions to make sure that they worked in practice.

```
(base) matthewmorris@crcc-dotix-nat-10-239-102-246 question1 % cargo run
Compiling question1 v0.1.0 (/Users/matthewmorris/Downloads/BU/Sem 2/DS210HW8/question1)
Finished dev profile [unoptimized + debuginfo] target(s) in 0.44s
Running target/debug/question1

Original DataFrame:
  Name  Number  PPG  YearBorn  TotalPoints  LikesPizza
Kareem  33      24.6  1947      48387         true
Karl    32      25.1  1963      46928         false
LeBron  23      27.0  1984      46381         false
Kobe    24      25.0  1978      43643         true
Michael 23      30.1  1963      42292         false

DataFrame with new column:
  Name  Number  PPG  YearBorn  TotalPoints  LikesPizza  IsAllStar
Kareem  33      24.6  1947      48387         true        true
Karl    32      25.1  1963      46928         false        true
LeBron  23      27.0  1984      46381         false        true
Kobe    24      25.0  1978      43643         true         true
Michael 23      30.1  1963      42292         false        true

Merged DataFrame:
  Name  Number  PPG  YearBorn  TotalPoints  LikesPizza
Kareem  33      24.6  1947      48387         true
Karl    32      25.1  1963      46928         false
LeBron  23      27.0  1984      46381         false
Kobe    24      25.0  1978      43643         true
Michael 23      30.1  1963      42292         false
Kareem  33      24.6  1947      48387         true
Karl    32      25.1  1963      46928         false
LeBron  23      27.0  1984      46381         false
Kobe    24      25.0  1978      43643         true
Michael 23      30.1  1963      42292         false

Selected columns:
  Name  PPG
Kareem  24.6
Karl    25.1
LeBron  27.0
Kobe    25.0
Michael 30.1

Restricted columns:
  Name  TotalPoints
Kareem  48387
Karl    46928
```

```
Restricted columns:
  Name  TotalPoints
Kareem  48387
Karl    46928
LeBron  46381
Kobe    43643
Michael 42292

Players with PPG > 20:
  Name  Number  PPG  YearBorn  TotalPoints  LikesPizza
Kareem  33      24.6  1947      48387         true
Karl    32      25.1  1963      46928         false
LeBron  23      27.0  1984      46381         false
Kobe    24      25.0  1978      43643         true
Michael 23      30.1  1963      42292         false

Median PPG: 25.10

TotalPoints - YearBorn differences:
Kareem: 46440
Karl: 44965
LeBron: 44397
Kobe: 41665
Michael: 40329
```

Chatgpt Queries:

how can i obtain and read csv files in rust?



To obtain and read CSV files in Rust, you'll typically use the [csv](#) crate along with [serde](#) for deserialization. Here's a minimal, practical guide:

✓ 1. Add dependencies to Cargo.toml

toml

Copy

```
[dependencies]
csv = "1.3"
serde = { version = "1.0", features = ["derive"] }
```

2. Read a CSV file into memory

A. Read row-by-row as raw strings:

rust

Copy

```
use std::error::Error;
use csv::Reader;

fn main() -> Result<(), Box<dyn Error>> {
    let mut rdr = Reader::from_path("data.csv")?;


    for result in rdr.records() {
        let record = result?;
        println!("{:?}", record);
    }

    Ok(())
}
```



B. Deserialize rows into a struct:

rust

 Copy

```
use std::error::Error;
use serde::Deserialize;
use csv::Reader;

#[derive(Debug, Deserialize)]
struct Person {
    name: String,
    age: u8,
    city: String,
}

fn main() -> Result<(), Box<dyn Error>> {
    let mut rdr = Reader::from_path("people.csv")?;

    for result in rdr.deserialize() {
        let person: Person = result?;
        println!("{:?}", person);
    }

    Ok(())
}
```

Ensure the CSV header matches the struct field names (name, age, city for the above example).

Optional: Read from a string or stdin

rust

 Copy

```
let data = "name,age\nAlice,30\nBob,25";
let mut rdr = csv::Reader::from_reader(data.as_bytes());
```


I have a problem where i am trying to create a dataframe in rust, and the labels misalign with the columns because the test for the labels is longer than the data how do i fix

Ah, yes—this is a common pitfall when working with CSVs or trying to simulate a DataFrame-like structure in Rust (since Rust doesn't have a built-in pandas-style DataFrame). The issue you're hitting sounds like:

Your column headers (labels) are more numerous than the fields in the rows (or vice versa), causing misalignment.

What's Going Wrong

If your header has more columns than your row data:


- The CSV reader will try to match fields by position.
- Missing fields in rows will be `None` (if using `Option<T>`) or cause deserialization errors (if using strict types like `String` or `u32`).

Fix It

Option 1: Fix the input data

Ensure all rows have the same number of fields as the header. This is the cleanest fix:

csv

 Copy

```
name,age,city
Alice,30,San Francisco
Bob,25,New York
```



If some fields might be missing, use `Option` (17):

rust

 Copy

```
#[derive(Debug, Deserialize)]
struct Person {
    name: Option<String>,
    age: Option<u8>,
    city: Option<String>,
}
```

This lets Rust safely handle missing fields instead of crashing.

Option 3: Read raw records and pad manually

If you're manually building a data structure like a "DataFrame" using `csv::Reader`, you can pad short rows:

rust

 Copy

```
use csv::Reader;
use std::error::Error;

fn main() -> Result<(), Box<dyn Error>> {
    let mut rdr = Reader::from_path("data.csv")?;
    let headers = rdr.headers()?.clone();
    let column_count = headers.len();

    for result in rdr.records() {
        let mut record = result?;
        // Pad the record if it's shorter than the header
        while record.len() < column_count {
            record.push_field("");
        }
        println!("{:?}", record);
    }

    Ok(())
}
```




Bonus: Want DataFrame-like functionality?

Consider using:


- [polars](#) – Fast DataFrame library in Rust
- [ndarray](#) – For numerical matrix-style data

toml

 Copy

```
# Cargo.toml
[dependencies]
polars = { version = "0.36", features = ["csv-file", "lazy"] }
```

rust

 Copy

```
use polars::prelude::*;

fn main() -> PolarsResult<()> {
    let df = CsvReader::from_path("data.csv")?
        .has_header(true)
        .finish()?;

    println!("{}", df);
    Ok(())
}
```

Let me know what kind of “DataFrame” structure you’re building—can tailor it better if you’re aiming for matrix math, table joins, or data exploration.