Tutorial II.III - DataFrames in Julia

Optimization with Julia

Introduction

Imagine a DataFrame as a digital spreadsheet. It's a way to organize and work with data in rows and columns. Each column can hold different types of information, like names, ages, or salaries. In this tutorial, we'll learn how to create DataFrames, add and change data, and perform simple operations on our data.

> **Note**
>
> Before we start, make sure you have the DataFrames package installed. If you're not sure how to do this, check the previous tutorial on package management!

Let's begin by importing the DataFrames package:

```
# Import the DataFrames package
using DataFrames
```

```
Precompiling DataFrames...
  18780.1 ms    DataFrames
  1 dependency successfully precompiled in 19 seconds. 32 already precompiled.
Precompiling QuartoNotebookWorkerDataFramesTablesExt...
   1358.3 ms    QuartoNotebookWorker → QuartoNotebookWorkerDataFramesTablesExt
  1 dependency successfully precompiled in 1 seconds. 54 already precompiled.
```

Section 1 - Creating DataFrames

A DataFrame in Julia is akin to a table in SQL or a spreadsheet - each column can have its own type, making it highly versatile. A DataFrame can be created using the DataFrame constructor and passing key-value pairs where the key is the column name and the value is an array of data. For more help, use `?` in the REPL and type `DataFrame`. Example:

```
students = DataFrame(
    Name = ["Elio", "Bob", "Yola"],
    Age = [18, 25,29],
)
```

	Name	Age
	String	Int64
1	Elio	18
2	Bob	25
3	Yola	29

Exercise 1.1 - Create a DataFrame

Create and Test a DataFrame. Create a DataFrame named `employees` with the columns `Name`, `Age`, and `Salary`, and populate it with the specified data: John is `28` years old and earns `50000`, Mike is `23` years old and earns `62000`. Frank is `37` years old and earns `90000`.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert employees == DataFrame(
    Name = ["John", "Mike", "Frank"],
    Age = [28, 23, 37],
    Salary = [50000, 62000, 90000]
)
println("DataFrame created successfully!")
println(employees)
```

> **Tip**
>
> Remember, for more help, use `?` in the REPL and type `DataFrame`.

Section 2 - Accessing and Modifying Data

Accessing columns in a DataFrame can be done using the dot syntax, while rows can be accessed via indexing. Modification of data is straightforward; just assign a new value to the desired cell. To access the column 'name' in our DataFrame with `employees`, we could do:

```
employees.Name
```

```
3-element Vector{String}:
 "John"
 "Mike"
 "Frank"
```

To access the third name specifically, we could do:

```
employees.Name[3]
```

```
"Frank"
```

Exercise 2.1 - Access the `Age` Column

Access the `Age` column from the DataFrame and save it in a new variable `ages`.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert ages == [28, 23, 37]
println("Correct, the Ages column is: ", ages)
```

Exercise 2.2 - Update John's Salary

Update John's salary to `59000`.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert employees.Salary[1] == 59000
println("Modified DataFrame: ")
println(employees)
```

Section 3 - Filtering Data

Logical indexing can be used to filter rows in a DataFrame based on conditions. To filter the DataFrame to include only employees names “Frank” we could do:

```
allFranks = employees[employees.Name .== "Frank", :]
```

	Name	Age	Salary
	String	Int64	Int64
1	Frank	37	90000

Alternatively, the filter function provides a powerful tool to extract subsets of data based on a condition:

```
allFranks = filter(row -> row.Name == "Frank", employees)
```

	Name	Age	Salary
	String	Int64	Int64
1	Frank	37	90000

Exercise 3.1 - Filter the DataFrame

Filter the DataFrame to include only employees with salaries above `60000`. Save the resulting employees in the DataFrame `high_earners`.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert nrow(high_earners) == 2
println("High earners: ")
println(high_earners)
```

Section 4 - Basic Data Manipulation

Julia provides functions for basic data manipulation tasks, including sorting, grouping, and joining DataFrames. The `sort` function can be used to order the rows in a DataFrame based on the values in one or more columns. To see how to use the function, type `?` into the REPL (terminal) and type `sort`.

Exercise 4.1 - Sort the DataFrame

Sort the DataFrame based on the `Age` column and save it as `sorted_df`.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert sorted_df.Age[1] == 23
println("DataFrame sorted by age: ")
println(sorted_df)
```

> **Tip**
>
> If you have more complicated data structures, take a look at JSON files which can be used to work with all kind of differently structured data sets.

Section 5 - Loop over DataFrames

Sometimes, you might need to iterate over the rows of a DataFrame to perform operations on each row individually. Julia provides a convenient way to do this using the `eachrow` function. For example, if we want to check for each employee if they have a salary above 60000, we can do the following:

```
for row in eachrow(employees)
    if row.Salary > 60000
        println("$(row.Name) earns more than 60000")
    end
end
```

```
Mike earns more than 60000
Frank earns more than 60000
```

Here, the `row` holds all the values of the row as a `NamedTuple`. We can access the values of a column then by using the dot syntax. To create a new column, we can use the `push!` function. For example, to create a new column called `VacationDays` in the `employees` DataFrame, we can do one of the following:

```
employees.VacationDays = [0 for row in eachrow(employees)]
employees.VacationDays .= 0
```

```
3-element Vector{Int64}:
 0
 0
 0
```

Exercise 5.1 - Loop over DataFrame

Create a new column called `Bonus` in the `employees` DataFrame. The bonus should be calculated as 10% of the salary for employees over 30, and 5% for those 30 and under. Use a loop to iterate over the rows and calculate the bonus.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert filter(
    row -> row.Bonus == 2950,
    employees
).Name == ["John"] "John should have a bonus of 2950"
@assert filter(
    row -> row.Bonus == 3100,
    employees
).Name == ["Mike"] "Mike should have a bonus of 3100"
@assert filter(
```

```
    row -> row.Bonus == 9000,
    employees
).Name == ["Frank"] "Frank should have a bonus of 9000"
println("Great job! All the bonuses are correct!")
```

Section 6 - Filling a new DataFrame with values

Do you remember the `push!` function? We can use it to fill a new DataFrame with values. For example, we can create a new DataFrame called `WorkingHours` and fill it with the values from the `employees` DataFrame. Imagine that the company has a policy, where employees above 30 work 30 hours a week, and employees under 30 work 40 hours a week:

```
# Create a new DataFrame
WorkingHours = DataFrame(
    Name = String[],
    Hours = Int[]
)
# Loop over the rows of the employees DataFrame
for row in eachrow(employees)
    if row.Age < 30
        push!(WorkingHours, (
            Name = row.Name,
            Hours = 40
        ))
    else
        push!(WorkingHours, (
            Name = row.Name,
            Hours = 30
        ))
    end
end
println(WorkingHours)
```

```
3×2 DataFrame
 Row   Name    Hours
       String  Int64

   1   John        40
   2   Mike        40
   3   Frank       30
```

Conclusion

Fantastic work! You've completed the tutorial on DataFrames in Julia. You've seen how to create DataFrames and access, modify and filter data. Continue to the next file to learn more.

Solutions

You will find the solutions to all exercises online here in the `solutions` folders for each part. However, I strongly encourage you to work on these exercises independently without searching explicitly for the exact answers to the exercises. Understanding someone else's solution is very different from developing your own. Use the lecture notes and try to solve the exercises on your own. This approach will significantly enhance your learning and problem-solving skills.

Remember, the goal is not just to complete the exercises, but to understand the concepts and improve your programming abilities. If you encounter difficulties, review the lecture materials, experiment with different approaches, and don't hesitate to ask for clarification during class discussions.