FINC-672 – Workshop in Finance: Empirical Research

Tabular Data

Prof. Matt Fleckenstein
University of Delaware

mflecken@udel.edu

Goals

- ☐ Tabular Data in Julia using `DataFrames`
- ☐ Load and Save Files
- ☐ Index Tabular Data
- ☐ Filter and Subset DataFrames
- ☐ Select Columns of DataFrames
- ☐ Missing Data and Data Types

Setting up DataFrames

- First, we need to step up the Julia `DataFrames` package.
- Start `Julia` and enter the following commands at the "REPL".

```
julia> using Pkg

julia> Pkg.add("DataFrames")

julia> using DataFrames
```

Tabular Data

- Data comes mostly in a tabular format.
- By tabular, we mean that the data consists of a table containing rows and columns.
- Columns are usually of the same data type, whereas rows have different types.
- The rows, in practice, denote observations while columns denote variables.
- For example, we can have a table of TV shows containing in which country it was produced and our personal rating.

Tabular Data (cont'd)

Name	**Country**	**Rating**
Game of Thrones	United States	8.2
The Crown	England	7.3
Friends	United States	7.8
...	...	...

- Here, the dots mean that this could be a very long table and we only show a few rows.

Tabular Data (cont'd)

- When we analyze data, often we come up with interesting questions about the data, also known as data **queries**.
- Examples of, so called **queries**, for this data could be:
  - Which TV show has the highest rating?
  - Which TV shows were produced in the United States?
  - Which TV shows were produced in the same country?
- To answer questions like "Which TV show has the highest rating?" we use **data transformation**.

Tabular Data (cont'd)

- Let's take the first three shows in the table and see how we model this using `DataFrames` in Julia.
- In a Julia `DataFrame`, we would set this up as follows

```
julia> tv_shows = DataFrame(
         name=["Game of Thrones", "The Crown", "Friends"],
         country=["United States", "England", "United States"],
         rating=[8.2, 7.3, 7.8]
    );
```

Tabular Data (cont'd)

```
julia> tv_shows
3×3 DataFrame
 Row │ name             country        rating
     │ String           String         Float64
─────┼──────────────────────────────────────────
   1 │ Game of Thrones  United States      8.2
   2 │ The Crown        England            7.3
   3 │ Friends          United States      7.8
```

Tabular Data (cont'd)

- As a second example, suppose we have data on bonds of four firms with (full) price and coupon rate (expressed in percentage points, and paid semi-annually).

firm	**price**	**coupon**
firmA	70.0	5.00
firmB	80.0	3.75
firmC	100.0	2.50
firmD	110.0	2.00

- Here, the column with the firm name (*firm*) has type `string`, price (*price*) and coupon rate (*coupon*) have type `float`.

Getting Data into a DataFrame

- With `DataFrames`, we can define a DataFrame to hold our tabular data.
- The following code gives us a variable `df` containing our data in table format.

```julia
julia> frm = ["firmA","firmB","firmC","firmD"];

julia> px = [70.0, 80.0, 100.0, 110.0];

julia> cpn = [5.00, 3.75, 2.50, 2.00];

julia> df = DataFrame(; firm=frm, price=px, coupon=cpn);
```

Getting Data into a DataFrame (cont'd)

- Let's display our DataFrame.

```
julia> df
4×3 DataFrame
 Row │ firm    price    coupon
     │ String  Float64  Float64
─────┼──────────────────────────
   1 │ firmA      70.0     5.0
   2 │ firmB      80.0     3.75
   3 │ firmC     100.0     2.5
   4 │ firmD     110.0     2.0
```

DataFrame Constructor

- We construct a `DataFrame` is simply to pass vectors as arguments into the **DataFrame constructor**.
- You can come up with any valid Julia vector and it will work as long as the vectors have the same length.
- Duplicates, Unicode symbols and any sort of numbers are fine.
- Another example:

```
julia> DataFrame(σ = ["a", "a", "a"], δ = [π, π/2, π/3])
3×2 DataFrame
 Row │ σ        δ
     │ String   Float64
─────┼──────────────────
   1 │ a        3.14159
   2 │ a        1.5708
   3 │ a        1.0472
```

Loadind and Saving Files

- We need to be able to store files and load files from disk.
- We focus on CSV and Excel file formats since those are the most common data storage formats for tabular data.
- **C**omma-**s**eparated **v**alues (CSV) files are are very effective way to store tables. CSV files have two advantages over other data storage files.
- First, it does exactly what the name indicates it does, namely storing values by separating them using commas `,`
- This acronym is also used as the file extension (you save your files using the ".csv" extension such as "myfile.csv").
- To demonstrate how a CSV file looks, we can install the `CSV.jl` package.

```
julia> using Pkg

julia> Pkg.add("CSV")

julia> using CSV
```

Saving to CSV Files

- We can now use our previous data on bonds and write it to CSV.

```
julia> path = "bonds.csv"
"bonds.csv"

julia>     CSV.write(path, df)
"bonds.csv"
```

Reading Data from CSV Files

- Next, let's read the data from the CSV file we have just created and put it into a DataFrame.
- Conveniently, `CSV.jl` will automatically infer column types for us.

```
julia> path = "bonds.csv"
"bonds.csv"

julia> CSV.File(path) |> DataFrame
4×3 DataFrame
 Row │ firm      price    coupon
     │ String7…  Float64  Float64
─────┼───────────────────────────
   1 │ firmA        70.0     5.0
   2 │ firmB        80.0     3.75
   3 │ firmC       100.0     2.5
   4 │ firmD       110.0     2.0
```

- Here we use the `|>` operator to "send" the CSV file into a DataFrame.

Writing Data to Excel Files

- To load an Excel file, we first need to add the `XLSX.jl` package.[1]

```julia
julia> using Pkg

julia> Pkg.add("XLSX")

julia> using XLSX
```

[1] Note that in the Julia codeblock the `using Pkg` is only needed once. That is if you have Julia opened and entered it before, you do not need to enter it again.

Writing Data to Excel Files (cont'd)

- Let's now write the bonds data to an Excel file.

```
julia> path==="bonds.xlsx";

julia> data===collect(eachcol(df));

julia> cols===names(df);

julia> XLSX.writetable(path,=data,=cols)
```

- Here, we need to provide the tabular data (`data`) and the column names (`cols`) individually to `writetable`.
- We get the data by *collecting* EACH column. This is what `collect(eachcol(df))` does.
- We get the column names by using `names(df)`.

Reading Data from Excel Files

- Let's now read the bond data in the Excel file we have just created back into a `DataFrame`.

```
julia> df = DataFrame(XLSX.readtable("bonds.xlsx","Sheet1")...)
4×3 DataFrame
 Row │ firm   price  coupon
     │ Any    Any    Any
─────┼─────────────────────
   1 │ firmA  70.0   5.0
   2 │ firmB  80.0   3.75
   3 │ firmC  100.0  2.5
   4 │ firmD  110.0  2.0
```

- Note that the `...` in the code above is again the splat operator we have encountered before. Here it basically unpacks the Excel worksheet so that we can put it into a DataFrame.

Indexing and Summarizing Data

- Let's continue to use our bond data as an example.
- Suppose we want to know all names of the firms in our dataset.
- To retrieve a vector for firm names, we can access the DataFrame with the `.` operator.

```
julia> df.firm
4-element Vector{Any}:
 "firmA"
 "firmB"
 "firmC"
 "firmD"
```

Indexing and Summarizing Data (cont'd)

- Alternatively, we can index a DataFrame much like an Array with symbols and special characters. The second index is the column indexing.

```
julia> df[!, :firm]
4-element Vector{Any}:
 "firmA"
 "firmB"
 "firmC"
 "firmD"
```

- Here, we use the `!` operator to indicate that we want to get all rows.

Indexing and Summarizing Data (cont'd)

- Let's suppose, you want to get the price and coupon rate for the second bond in our data.
- For any row, in our case the second row, we can use the first index as row indexing (in the codeblock below, this is the `2` before the comma).
- The colon `:` just means that we want to get all columns (in our case the firm name, bond price, and coupon rate).

```
julia> df[2, :]
DataFrameRow
 Row │ firm   price  coupon
     │ Any    Any    Any
─────┼──────────────────────
   2 │ firmB  80.0   3.75
```

Indexing and Summarizing Data (cont'd)

- How would we get the price and dividend for the third stock in our data?
- Simply use a `3` as the `row` index.

```
julia> df[3, :]
DataFrameRow
 Row │ firm   price  coupon
     │ Any    Any    Any
─────┼─────────────────────
   3 │ firmC  100.0  2.5
```

Indexing and Summarizing Data (cont'd)

- How about the firm name for the second *and* the third bond?

```
julia> df[1:2, :firm]
2-element Vector{Any}:
 "firmA"
 "firmB"
```

Indexing and Summarizing Data (cont'd)

- How can we get the price and coupon rate of the second and third bond?

```
julia> df[1:2, [:price,:coupon]]
2×2 DataFrame
 Row │ price   coupon
     │ Any     Any
─────┼───────────────
   1 │ 70.0    5.0
   2 │ 80.0    3.75
```

- Note that we write the column names with a colon `:` and put them between brackets (`[` and `]`) and separate the column names with a comma `,`

Filter and Subset DataFrames

- The DataFrame functions `filter` and subset `subset` allow us to "filter" out rows from a DataFrame, or, in other words, allow us to take a subset of a DataFrame.
- We can filter rows by using `filter(source => f::Function, df)`
- Let's illustrate this with an example using our bond data from before.

firm	**price**	**coupon**
firmA	70.0	5.00
firmB	80.0	3.75
firmC	100.0	2.50
firmD	110.0	2.00

Filter and Subset DataFrames (cont'd)

- Let's find the bond that is trading at par (i.e. its price is 100.0).

```
julia> filter(:price => (x->x==100.0), df)
1×3 DataFrame
 Row │ firm    price   coupon
     │ Any     Any     Any
─────┼───────────────────────
   1 │ firmC   100.0   2.5
```

- Let's figure out what is going on here.

Filter and Subset DataFrames (cont'd)

```
filter(:price => (x->x==100.0), df)
```

- We take the `price` column and use the `=>` operator to pass this column to a function.
- Why? Because we are looking for the bond with `price=100.0`.
- Then, we use a so-called **anonymous function** to check when the bond price is equal to 100.0
- This is the `(x->x==100.0)`
- The `filter` function then returns the row for which the condition `x==100.0` is `true`

Filter and Subset DataFrames (cont'd)

- We often want to subset data using multiple conditions.
- For instance, we would like to know which bond trades at a discount to par and has a coupon rate greater than four percent.
- In these cases, we do not use an anonymous function as in the previous example (`(x->x==100.0)`), but we define a function.
- To illustrate this, let's use a function in the previous example.

Filter and Subset DataFrames (cont'd)

```
julia> function isPar(x)
    if x==100.0
        return true
    else
        return false
    end
end;

julia>
df2 = filter(:price => (x->isPar(x)), df)
1×3 DataFrame
 Row │ firm   price  coupon
     │ Any    Any    Any
─────┼──────────────────────
   1 │ firmC  100.0  2.5
```

Filter and Subset DataFrames (cont'd)

- We can build a more complex filter
- Suppose we want to get the bonds that trade at a discount to par value and with coupon rate of at least four percent.
- Let's first build the function

```
julia> function getBond(price,coupon)
    if price<100.0 && coupon>=4.00
        return true
    else
        return false
    end
end;
```

Filter and Subset DataFrames (cont'd)

- Now, let's use our `getBond` function.

```
julia> df2 = filter([:price, :coupon] => ( (x,y)->getBond(x,y)), df)
1×3 DataFrame
 Row │ firm   price  coupon
     │ Any    Any    Any
─────┼─────────────────────
   1 │ firmA  70.0   5.0
```

- Let's figure out what is going on here.

Filter and Subset DataFrames (cont'd)

```
df2 = filter([:price, :coupon] => ( (x,y)->getBond(x,y)), df)
```

- Here, we need to check the price (`price`) and coupon rate (`coupon`) of the bonds.
- We get these two columns by using the colon `:` operator and by putting them between brackets (`[` and `]`), seperated by a comma `,` i.e. `[:price, :coupon]`
- We then use `=>` to "send" these two columns to our function.
- To call our function, we need two inputs: the price and the coupon rate.
- Thus, we use `x,y)`make sure to use the parentheses `(` and `)`).
- Then, we send these two input to our function `getBond(x,y)`.

Selecting Columns

- We select specific columns using the function `select`
- To illustrate, lets suppose we have the following bond dataset.
- Note that we have the same bonds as before, but we now know the year when the bonds were issued and the year of maturity of the bonds. We also have bid and ask prices.

firm	bidprice	askprice	coupon	issueyear	maturityyear
firmA	69.00	70.0	5.00	2018	2023
firmB	79.50	80.0	3.75	2020	2030
firmC	99.75	100.0	2.50	2021	2024
firmD	109.00	110.0	2.00	2015	2025

Selecting Columns

- First, let's create a `DataFrame`.

```
julia> frm   = ["firmA","firmB","firmC","firmD"];

julia> pxbid = [69.00, 79.50, 99.75, 109.00];

julia> pxask = [70.0, 80.0, 100.0, 110.0];

julia> cpn   = [5.00, 3.75, 2.50, 2.00];

julia> issyr = [2018, 2020, 2021, 2015];

julia> matyr = [2023, 2030, 2024, 2025];

julia>
df = DataFrame(firm=frm, bidprice=pxbid, askprice=pxask, coupon=cpn, issueyear=issyr, maturityyear=matyr);
```

Selecting Columns (cont'd)

- Let's display the DataFrame

```
julia> df
4×6 DataFrame
 Row │ firm    bidprice  askprice  coupon   issueyear  maturityyear
     │ String  Float64   Float64   Float64  Int64      Int64
─────┼─────────────────────────────────────────────────────────────
   1 │ firmA      69.0       70.0     5.0        2018          2023
   2 │ firmB      79.5       80.0     3.75       2020          2030
   3 │ firmC      99.75     100.0     2.5        2021          2024
   4 │ firmD     109.0      110.0     2.0        2015          2025
```

Selecting Columns (cont'd)

- First, we want to `select` the column with all firm names.

```
julia> df2 = select(df, :firm)
4×1 DataFrame
 Row │ firm
     │ String
─────┼────────
   1 │ firmA
   2 │ firmB
   3 │ firmC
   4 │ firmD
```

- Note that our DataFrame `df` comes first, i.e. `select(df`
- Also note that we could get the same result by using `df.firm`
- However, `select` is powerful when we select multiple columns

Selecting Columns (cont'd)

- Next, suppose we want to get back the original bond dataset that we started with (i.e. where we have the firm name, askprice, and the coupon rate).

```
julia> df2 = select(df, [:firm, :askprice, :coupon])
4×3 DataFrame
 Row │ firm    askprice  coupon
     │ String  Float64   Float64
─────┼───────────────────────────
   1 │ firmA       70.0     5.0
   2 │ firmB       80.0     3.75
   3 │ firmC      100.0     2.5
   4 │ firmD      110.0     2.0
```

- Let's discuss what is going on here.

Selecting Columns (cont'd)

```
df2 = select(df, [:firm, :askprice, :coupon])
```

- As before, we use the column names with a colon `:` and put them between brackets (`[` and `]`), separated by a comma `,`
- Then we simply use this as the second argument after `df` in the function call to `select`

Selecting Columns (cont'd)

- Suppose now that we want all columns, except the issue year.
- To exclude one (or more columns), we use `Not()` as shown below.

```
julia> df2 = select(df, Not(:issueyear))
4×5 DataFrame
 Row │ firm    bidprice  askprice  coupon   maturityyear
     │ String  Float64   Float64   Float64  Int64
─────┼──────────────────────────────────────────────────
   1 │ firmA       69.0      70.0     5.0           2023
   2 │ firmB       79.5      80.0     3.75          2030
   3 │ firmC       99.75    100.0     2.5           2024
   4 │ firmD      109.0     110.0     2.0           2025
```

Selecting Columns (cont'd)

- What if we want all columns except the issue year and the bid price?

```
julia> df2 = select(df, Not([:issueyear,:bidprice]))
4×4 DataFrame
 Row │ firm    askprice  coupon   maturityyear
     │ String  Float64   Float64  Int64
─────┼──────────────────────────────────────────
   1 │ firmA       70.0     5.0           2023
   2 │ firmB       80.0     3.75          2030
   3 │ firmC      100.0     2.5           2024
   4 │ firmD      110.0     2.0           2025
```

- Note that we need to put the two column names between brackets (`[` and `]`), separated by a comma `,`

Selecting Columns (cont'd)

- We can also "mix and match"
- Suppose we want the firm name, all other columns, but not the bid price.

```
julia> df2 = select(df, :firm, Not(:bidprice))
4×5 DataFrame
 Row │ firm     askprice  coupon   issueyear  maturityyear
     │ String   Float64   Float64  Int64      Int64
─────┼─────────────────────────────────────────────────────
   1 │ firmA        70.0     5.0        2018          2023
   2 │ firmB        80.0     3.75       2020          2030
   3 │ firmC       100.0     2.5        2021          2024
   4 │ firmD       110.0     2.0        2015          2025
```

Selecting Columns (cont'd)

- Can we **rename** columns using the `select` function?
- The answer is yes. Suppose we want to rename the **firm** column to **firmname**.

```
julia> df2 = select(df, :firm => :firmname, :)
4×7 DataFrame
 Row │ firmname  firm    bidprice  askprice  coupon   issueyear  maturityyear
     │ String    String  Float64   Float64   Float64  Int64      Int64
─────┼───────────────────────────────────────────────────────────────────────
   1 │ firmA     firmA       69.0      70.0     5.0        2018          2023
   2 │ firmB     firmB       79.5      80.0     3.75       2020          2030
   3 │ firmC     firmC       99.75    100.0     2.5        2021          2024
   4 │ firmD     firmD      109.0     110.0     2.0        2015          2025
```

- What is happening here?

Selecting Columns (cont'd)

```
df2 = select(df, :firm => :firmname, :)
```

- the first part `:firm => :firmname` means that we assign the new name "firmname" to the existing column `firm`.
- The colon `:` (which is separated by a comma `,`) means that we want to select all other columns as well (except the one we just renamed).

Missing Data and Data Types

- `CSV.jl` will typically work quite well in guessing what kind of types our data have as columns.
- However, this won't always work perfectly. Let's see how we fix wrong data types and what data types we should use.
- We work with the following bond dataset.

id	**firm**	**bidprice**	**askprice**	**coupon**	**issuedate**	**maturitydate**
1	firmA	69.00	70.0	5.00	31-01-2018	31-01-2023
2	firmB	79.50	80.0	3.75	31-03-2020	31-03-2030
3	firmC	99.75	100.0	2.50	30-09-2021	30-09-2024
4	firmD	109.00	110.0	2.00	31-10-2015	31-10-2025

Missing Data and Data Types (cont'd)

- Suppose someone created the DataFrame as shown below.

```
julia> idno  = ["1","2","3","4"];

julia> frm   = ["firmA","firmB","firmC","firmD"];

julia> pxbid = [69.00, 79.50, 99.75, 109.00];

julia> pxask = [70.0, 80.0, 100.0, 110.0];

julia> cpn   = [5.00, 3.75, 2.50, 2.00];

julia> issdt = ["31-01-2018","31-03-2020","30-09-2021","31-10-2015"];

julia> matdt = ["31-01-2023","31-03-2030","30-09-2024","31-10-2025"];

julia>
df = DataFrame(id=idno, firm=frm, bidprice=pxbid, askprice=pxask, coupon=cpn, issuedate=issdt, maturitydate=matdt
```

Missing Data and Data Types (cont'd)

- Let's display the DataFrame.

```
julia> df
4×7 DataFrame
 Row │ id      firm    bidprice  askprice  coupon   issuedate   maturitydate
     │ String  String  Float64   Float64   Float64  String      String
─────┼──────────────────────────────────────────────────────────────────────
   1 │ 1       firmA      69.0       70.0     5.0   31-01-2018  31-01-2023
   2 │ 2       firmB      79.5       80.0     3.75  31-03-2020  31-03-2030
   3 │ 3       firmC      99.75     100.0     2.5   30-09-2021  30-09-2024
   4 │ 4       firmD     109.0      110.0     2.0   31-10-2015  31-10-2025
```

- What could be wrong here?

Missing Data and Data Types (cont'd)

- Let's try to sort the DataFrame by issue date.
- We do this by using the function `sort` as follows

```
julia> sort(df, :issuedate)
4×7 DataFrame
 Row │ id      firm    bidprice  askprice  coupon   issuedate   maturitydate
     │ String  String  Float64   Float64   Float64  String      String
─────┼───────────────────────────────────────────────────────────────────────
   1 │ 3       firmC      99.75     100.0     2.5   30-09-2021  30-09-2024
   2 │ 1       firmA      69.0       70.0     5.0   31-01-2018  31-01-2023
   3 │ 2       firmB      79.5       80.0     3.75  31-03-2020  31-03-2030
   4 │ 4       firmD     109.0      110.0     2.0   31-10-2015  31-10-2025
```

- What went wrong?
- Because the issue date column has the wrong type, sorting does not work correctly.

Missing Data and Data Types (cont'd)

- To fix the sorting, we can use the `Date` module from Julias standard library.
- To illustrate convert a `String` to `Date`, consider the first date "31-01-2023".

```
julia> using Dates;

julia> date_str = "31-01-2023"
"31-01-2023"

julia> date = Dates.Date(date_str, "dd-mm-yyyy")
2023-01-31
```

Missing Data and Data Types (cont'd)

- Next, let's convert all issue date to Julia `Date` type.
- To do this, we first get all issue dates in a **`Vector`**.
- Then we **broadcast** the `Date` constructor.
- In the last step, we write the converted dates back to our DataFrame `df`

```
julia> issue_dates_str = df.issuedate
4-element Vector{String}:
 "31-01-2018"
 "31-03-2020"
 "30-09-2021"
 "31-10-2015"

julia> issue_dates_dt = Dates.Date.(issue_dates_str, "dd-mm-yyyy")
4-element Vector{Dates.Date}:
 2018-01-31
 2020-03-31
 2021-09-30
 2015-10-31
```

Missing Data and Data Types (cont'd)

- Likewise, we repeat the same operations for the maturity dates.
- Note, we will learn how to do this more quickly when we talk about **data transformation** function in `DataFrames`.

```
julia> mat_dates_str = df.maturitydate
4-element Vector{String}:
 "31-01-2023"
 "31-03-2030"
 "30-09-2024"
 "31-10-2025"

julia> mat_dates_dt = Dates.Date.(mat_dates_str, "dd-mm-yyyy")
4-element Vector{Dates.Date}:
 2023-01-31
 2030-03-31
 2024-09-30
 2025-10-31

julia> df.maturitydate = mat_dates_dt
```

Missing Data and Data Types (cont'd)

- We are not done yet. Notice that the `id` column is also recognized as a `String`.
- An id variable should be of **categorical** type.
- Julia helps us here since it implements functionality for **categorical** data.
- All we need to do is load `CategoricalArrays.jl`.

```
julia> using Pkg;

julia> Pkg.add("CategoricalArrays");

julia> using CategoricalArrays;
```

Missing Data and Data Types (cont'd)

- Now we are all set to convert the `id` column to `categorical`.

```
julia> categorical(df[!, :id])
4-element CategoricalArrays.CategoricalArray{String,1,UInt32}:
 "1"
 "2"
 "3"
 "4"
```

- Here we are using a shortcut by directly making the conversion on our `DataFrame`.
- Note: *We must use the* `!` *operator.*
- This ensures two things. First, recall that `!` gives us the entire `id` column. Second, by using `!` we change the contents of our DataFrame `df` directly (or *in place*).

Missing Data and Data Types (cont'd)

- Finally, let's sort our `DataFrame` by the `issuedate` column.

```
julia> sort(df, :issuedate)
4×7 DataFrame
 Row │ id      firm    bidprice  askprice  coupon   issuedate   maturitydate
     │ String  String  Float64   Float64   Float64  Date…       Date…
─────┼────────────────────────────────────────────────────────────────────────
   1 │ 4       firmD     109.0     110.0     2.0    2015-10-31  2025-10-31
   2 │ 1       firmA      69.0      70.0     5.0    2018-01-31  2023-01-31
   3 │ 2       firmB      79.5      80.0     3.75   2020-03-31  2030-03-31
   4 │ 3       firmC      99.75    100.0     2.5    2021-09-30  2024-09-30
```

Wrap-Up

- ☑ Tabular Data in Julia using `DataFrames`
- ☑ Load and Save Files
- ☑ Index Tabular Data
- ☑ Filter and Subset DataFrames
- ☑ Select Columns of DataFrames
- ☑ Missing Data and Data Types