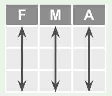
Data Wrangling with DataFrames.jl Cheat Sheet

(for version 1.x)

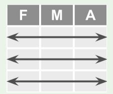
Tidy Data - the foundation of data wrangling

In a tidy data set:



Each **variable** is saved in its own **column**.

&



Each **observation** is saved in its own **row**.

Tidy data makes data analysis ***easier*** and ***more intuitive***. DataFrames.jl can help you tidy up your data.

Create DataFrame

DataFrame(x = [1,2,3], y = 4:6, z = 9)
Create data frame with column data from vector, range, or constant.

DataFrame([(x=1, y=2), (x=3, y=4)])
Create data frame from a vector of named tuples.

DataFrame("x" => [1,2], "y" => [3,4])
Create data frame from pairs of column name and data.

DataFrame(rand(5, 3), [:x, :y, :z])
DataFrame(rand(5, 3), :auto)
Create data frame from a matrix.

DataFrame()
Create an empty data frame without any columns.

DataFrame(x = Int[], y = Float64[])
Create an empty data frame with typed columns.

DataFrame(mytable)
Create data frame from any data source that supports Tables.jl interface.

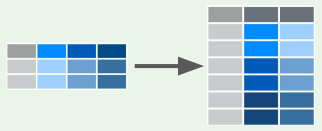
Describe DataFrame

describe(df)
Summary stats for all columns.

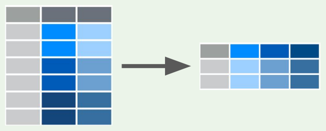
describe(df, :mean, :std)
Specific stats for all columns.

describe(df, extrema => :extrema)
Apply custom function to all columns.

Reshape Data - changing layout



stack(df, [:sibsp, :parch])
Stack columns data as rows with new **variable** and **value** columns



unstack(df, :variable, :value)
Unstack rows into columns using **variable** and **value** columns

Sort Data

Mutation**: use **sort!

sort(df, :age)
Sort by age

sort(df, :age, rev = true)
Sort by age in reverse order

sort(df, [:age, order(:sibsp, rev = true)])
Sort by in ascending age and descending *sibsp* order

Select Observations (rows)

Function syntax

first(df, 5) or **last(df, 5)**
First 5 rows or last 5 rows

unique(df)
unique(df, [:pclass, :survived])
Return data frame with unique rows.

filter(:sex => ==("male"), df)
filter(row -> row.sex == "male", df)
Return rows having *sex* equals “male”.
Note: the first syntax performs better.

subset(df, :survived)
subset(df, :sex => x -> x .== "male")
Return rows for which value is true.
Note: the “survived” column is Bool type

Indexing syntax

df[6:10, :]
Return rows 6 to 10

df[df.sex .== "male", :]
Return rows having *sex* equals “male”.

df[findfirst(==(30), df.age), :]
Return first row having *age* equals 30.

df[findall(==(1), df.pclass), :]
Return all rows having *pclass* equals 1.

Mutation**: use **unique!, filter!,** or **subset!

Select Variables (columns)

Function syntax

select(df, :sex)
select(df, "sex")
select(df, [:sex, :age])
Select desired column(s).

select(df, 2:5)
Select columns by index.

select(df, r"^s")
Select columns by regex.

select(df, Not(:age))
Select all columns except the *age* column.

select(df, Between(:name, :age))
Select all columns between *name* and *age* columns.

Indexing syntax

df[:, [:sex, :age]]
Select a copy of columns.

df[!, [:sex, :age]]
Select original column vectors.

P.S. Indexing syntax can select observations and variables at the same time!

Mutation**: use **select!

View Metadata

names(df)
propertynames(df)
Column names.

nrow(df)
ncol(df)
Number of rows and columns.

columnindex(df, "sex")
Index number of a column.

Handle Missing Data

dropmissing(df)
dropmissing(df, [:age, :sex])
Return rows without any missing data.

allowmissing(df)
allowmissing(df, :sibsp)
Allow missing data in column(s).

disallowmissing(df)
disallowmissing(df, :sibsp)
Do not allow missing data in column(s).

completecases(df)
completecases(df, [:age, :sex])
Return Bool array with *true* entries for rows without any missing data.

Mutation**: use **dropmissing!**, **allowmissing!**, or **disallowmissing!

Cumulative and Moving Stats

Cumulative Stats

```
select(df, :x => cumsum)
select(df, :x => cumprod)
```
Cumulative sum and product of column *x*.

```
select(df, :x => v -> accumulate(min, v))
select(df, :x => v -> accumulate(max, v))
```
Cumulative minimum/maximum of column *x*.

```
select(df, :x => v -> cumsum(v) ./ (1:length(v)))
```
Cumulative mean of column *x*.

Moving Stats (a.k.a Rolling Stats)

```
select(df, :x => (v -> runmean(v, n)))
select(df, :x => (v -> runmedian(v, n)))
select(df, :x => (v -> runmin(v, n)))
select(df, :x => (v -> runmax(v, n)))
```
Moving mean, medium, minimu, and maximum for column *x* with window size *n*

*The **run*** functions (and more) are available from RollingFunctions.jl package.*

Ranking and Lead/Lag Functions

```
select(df, :x => ordinalrank)        # 1234
select(df, :x => competerank)        # 1224
select(df, :x => denserank)          # 1223
select(df, :x => tiedrank)           # 1 2.5 2.5 4
```

*The ***rank** functions come from StatsBase.jl package.*

```
select(df, :x => lead)               # shift up
select(df, :x => lag)                # shift down
```

*The **lead** and **lag** functions come from ShiftedArrays.jl package.*

Build Data Pipeline

```
@pipe df |>
    filter(:sex => ==("male"), _) |>
    groupby(_, :pclass) |>
    combine(_, :age => mean)
```

The @pipe macro comes from Pipe.jl package. Underscores are automatically replaced by return value from the previous operation before the |> operator.

Summarize Data

Aggregating variables

```
combine(df, :survived => sum)
combine(df, :survived => sum => :survived)
```
Apply a function to a column; optionally assign colum name.

```
combine(df, :age => (x -> mean(skipmissing(x))))
```
Apply an anonymous function to a column.

```
combine(df, [:parch, :sibsp] .=> maximum)
```
Apply a function to multiple columns using broadcasting syntax.

Adding variables with aggregation results

```
transform(df, :fare => mean => :average_fare)
```
Add a new column that is populated with the aggregated value.

```
select(df, :name, :fare, :fare => mean => :average_fare)
```
Select any columns and add new ones with the aggregated value.

Adding variables by row

```
transform(df, [:parch, :sibsp] => ByRow(+) => :relatives)
```
Add new column by applying a function over existing column(s).

```
transform(df, :name => ByRow(x -> split(x, ",")) => [:lname, :fname])
```
Add new columns by applying a function that returns multiple values.

***Tips:** Use **skipmissing** function to remove missing values.*

Group Data Sets

```
gdf = groupby(df, :pclass)
gdf = groupby(df, [:pclass, :sex])
```
Group data frame by one or more columns.

```
keys(gdf)
```
Get the keys for looking up SubDataFrame's in the group.

```
gdf[(1,)]
```
Look up a specific group using a tuple of key values.

```
combine(gdf, :survived => sum)
```
Apply a function over a column for every group. Returns a single data frame.

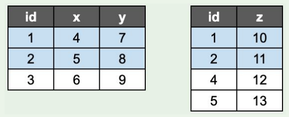
```
combine(gdf) do sdf
    DataFrame(survived = sum(sdf.survived))
end
```
Apply a function to each SubDataFrame in the group and combine results.

```
combine(gdf, AsTable(:) => t -> sum(t.parch .+ t.sibsp))
```
Apply a function to each SubDataFrame in the group and combine results.

Tips:
You can also use these functions to add summarized data to all rows:
- ***select***
- ***select!***
- ***transform***
- ***transform!***

Combine Data Sets

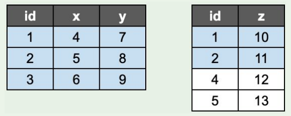
innerjoin(df1, df2, on = :id)



id	x	y
1	4	7
2	5	8
3	6	9

id	z
1	10
2	11
4	12
5	13

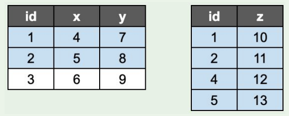
leftjoin(df1, df2, on = :id)



id	x	y
1	4	7
2	5	8
3	6	9

id	z
1	10
2	11
4	12
5	13

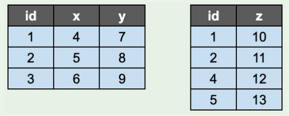
rightjoin(df1, df2, on = :id)



id	x	y
1	4	7
2	5	8
3	6	9

id	z
1	10
2	11
4	12
5	13

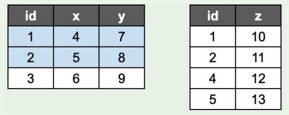
outerjoin(df1, df2, on = :id)



id	x	y
1	4	7
2	5	8
3	6	9

id	z
1	10
2	11
4	12
5	13

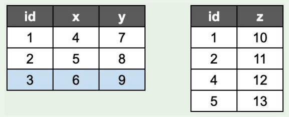
semijoin(df1, df2, on = :id)



id	x	y
1	4	7
2	5	8
3	6	9

id	z
1	10
2	11
4	12
5	13

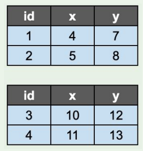
antijoin(df1, df2, on = :id)



id	x	y
1	4	7
2	5	8
3	6	9

id	z
1	10
2	11
4	12
5	13

vcat(df1, df2)

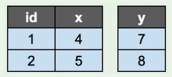


id	x	y
1	4	7
2	5	8

id	x	y
3	10	12
4	11	13

Data frames can be combined vertically or horizontally.

hcat(df1, df2)



id	x
1	4
2	5

y
7
8