# Introducing Julia/DataFrames

Search this book

## DataFrames

The Julia DataFrames package is an alternative to Python's Pandas package, but can be used with Pandas using the Pandas.jl wrapper package. Julia-only packages Query.jl and DataFramesMeta.jl can also be used with DataFrames.

A *DataFrame* is a data structure like a table or spreadsheet. You can use it for storing and exploring a set of related data values. Think of it as a smarter array for holding tabular data.

To explore the use of DataFrames.jl, we'll start by examining a well-known statistics dataset called Anscombe's Quartet.

Start by downloading the DataFrames and CSV packages, if you've not used them before:

```
julia> ]
(v1.2) pkg> add DataFrames
...messages
(v1.2) pkg>
```

You have to do this just once. The version should be at least v0.22.0.

Also, you can add CSV.jl while you're there:

```
(v1.2) pkg> add CSV
```

To use DataFrames:

```
julia> using DataFrames
```

For this document, we're using the 0.21 version of DataFrames.jl. Earlier versions had slightly different syntax for accessing columns, so it's worth updating if you're on an earlier version.

### Loading data into DataFrames

There a few different ways to create new DataFrames. For this introduction, the quickest way to load the Anscombe dataset and assign it to a variable `anscombe` is to copy/paste several rows of data, convert them to an array, and then rename the column names, like this:

```
julia>  anscombe = DataFrame(
 [10  10  10  8   8.04   9.14  7.46   6.58;
  8   8   8   8   6.95   8.14  6.77   5.76;
  13  13  13  8   7.58   8.74  12.74  7.71;
```

```
  9   9   9   8   8.81   8.77  7.11   8.84;
  11  11  11  8   8.33   9.26  7.81   8.47;
  14  14  14  8   9.96   8.1   8.84   7.04;
  6   6   6   8   7.24   6.13  6.08   5.25;
  4   4   4   19  4.26   3.1   5.39   12.5;
  12  12  12  8   10.84  9.13  8.15   5.56;
  7   7   7   8   4.82   7.26  6.42   7.91;
  5   5   5   8   5.68   4.74  5.73   6.89], :auto);
```

```
julia> rename!(anscombe, [Symbol.(:X, 1:4); Symbol.(:Y, 1:4)])
```

```
11×8 DataFrames.DataFrame
│ Row │ X1   │ X2   │ X3   │ X4   │ Y1    │ Y2   │ Y3    │ Y4   │
├─────┼──────┼──────┼──────┼──────┼───────┼──────┼───────┼──────┤
│ 1   │ 10.0 │ 10.0 │ 10.0 │ 8.0  │ 8.04  │ 9.14 │ 7.46  │ 6.58 │
│ 2   │ 8.0  │ 8.0  │ 8.0  │ 8.0  │ 6.95  │ 8.14 │ 6.77  │ 5.76 │
│ 3   │ 13.0 │ 13.0 │ 13.0 │ 8.0  │ 7.58  │ 8.74 │ 12.74 │ 7.71 │
│ 4   │ 9.0  │ 9.0  │ 9.0  │ 8.0  │ 8.81  │ 8.77 │ 7.11  │ 8.84 │
│ 5   │ 11.0 │ 11.0 │ 11.0 │ 8.0  │ 8.33  │ 9.26 │ 7.81  │ 8.47 │
│ 6   │ 14.0 │ 14.0 │ 14.0 │ 8.0  │ 9.96  │ 8.1  │ 8.84  │ 7.04 │
│ 7   │ 6.0  │ 6.0  │ 6.0  │ 8.0  │ 7.24  │ 6.13 │ 6.08  │ 5.25 │
│ 8   │ 4.0  │ 4.0  │ 4.0  │ 19.0 │ 4.26  │ 3.1  │ 5.39  │ 12.5 │
│ 9   │ 12.0 │ 12.0 │ 12.0 │ 8.0  │ 10.84 │ 9.13 │ 8.15  │ 5.56 │
│ 10  │ 7.0  │ 7.0  │ 7.0  │ 8.0  │ 4.82  │ 7.26 │ 6.42  │ 7.91 │
│ 11  │ 5.0  │ 5.0  │ 5.0  │ 8.0  │ 5.68  │ 4.74 │ 5.73  │ 6.89 │
```

### Collected datasets

Alternatively you could download and install the RDatasets package, which contains a number of famous datasets, including Anscombe's.

```
julia> ]
(v1.2) pkg> add RDatasets
julia> using RDatasets
julia> anscombe = dataset("datasets","anscombe")
```

There are other ways to create DataFrames, including reading data from files (using CSV.jl).

### Empty DataFrames

You can create simple DataFrames by providing the information about rows, and column names, in arrays:

```
julia> df = DataFrame(A = 1:6, B = 100:105)
```

```
6×2 DataFrame
│ Row │ A     │ B     │
│     │ Int64 │ Int64 │
├─────┼───────┼───────┤
│ 1   │ 1     │ 100   │
│ 2   │ 2     │ 101   │
│ 3   │ 3     │ 102   │
│ 4   │ 4     │ 103   │
│ 5   │ 5     │ 104   │
│ 6   │ 6     │ 105   │
```

To create a completely empty DataFrame, you supply the column names (Julia symbols) and define their types (remembering that the columns are arrays):

```
df = DataFrame(Name=String[],
    Width=Float64[],
    Height=Float64[],
    Mass=Float64[],
    Volume=Float64[])
```

```
0×5 DataFrames.DataFrame
```

```
df = vcat(df, DataFrame(Name="Test", Width=1.0, Height=10.0, Mass=3.0, Volume=5.0))
```

```
1×5 DataFrames.DataFrame
│ Row │ Name │ Width │ Height │ Mass │ Volume │
├─────┼──────┼───────┼────────┼──────┼────────┤
│ 1   │ Test │ 1.0   │ 10.0   │ 3.0  │ 5.0    │
```

## Basics

Once the Anscombe dataset has been loaded, you should see the DataFrame; its appearance varies if you're working in a terminal or using an IJulia notebook. But in either case you can see that you have a table of data, with 8 named columns (X1 to Y4), and 11 rows (1 to 11). The first pair of interest is X1/Y1, the second X2/Y2, and so on. Because the columns are named, it's easy to refer to specific columns when processing or analyzing the data.

```
julia> typeof(anscombe)
DataFrame
```

To obtain a list of column names, use the `names()` function:

```
julia> names(anscombe)
 8-element Array{String,1}:
 "X1"
 "X2"
 "X3"
 "X4"
 "Y1"
 "Y2"
 "Y3"
 "Y4"
```

Other useful functions:

```
julia> size(anscombe) # number of rows, number of columns
(11, 8)
```

The `describe()` function provides a quick overview of each column:

```
julia> describe(anscombe)
```

```
8×8 DataFrame
│ Row │ variable │ mean    │ min  │ median  │ max  │ nunique │ nmissing │ eltype   │
│     │ Symbol   │ Float64 │ Real │ Float64 │ Real │ Nothing │ Nothing  │ DataType │
├─────┼──────────┼─────────┼──────┼─────────┼──────┼─────────┼──────────┼──────────┤
│ 1   │ X1       │ 9.0     │ 4    │ 9.0     │ 14   │         │          │ Int64    │
│ 2   │ X2       │ 9.0     │ 4    │ 9.0     │ 14   │         │          │ Int64    │
│ 3   │ X3       │ 9.0     │ 4    │ 9.0     │ 14   │         │          │ Int64    │
```

```
│ 4   │ X4      │ 9.0     │ 8    │ 8.0    │ 19     │         │          │ Int64    │
│ 5   │ Y1      │ 7.50091 │ 4.26 │ 7.58   │ 10.84  │         │          │ Float64  │
│ 6   │ Y2      │ 7.50091 │ 3.1  │ 8.14   │ 9.26   │         │          │ Float64  │
│ 7   │ Y3      │ 7.5     │ 5.39 │ 7.11   │ 12.74  │         │          │ Float64  │
│ 8   │ Y4      │ 7.50091 │ 5.25 │ 7.04   │ 12.5   │         │          │ Float64  │
```

Notice that some of the columns (all the X columns) contain integer values, and others (all the Y columns) are floating-point numbers. Every element in a column of a DataFrame has the same data type, but different columns can have different types — this makes the DataFrame ideal for storing tabular data - strings in one column, numeric values in another, and so on.

### Referring to specific columns

There are various ways to select columns.

You can use the dot/period (.), the standard Julia field-accessor:

```
julia> anscombe.Y2
```

```
11-element Array{Float64,1}:
 9.14
 8.14
 8.74
 8.77
 9.26
 8.1
 6.13
 3.1
 9.13
 7.26
 4.74
```

Or you can use the general Julia convention for symbol names: precede the column names with a colon (:). So :Y2 refers to the column called Y2, or column number 6.

You can use integers and vectors of integers; here's the sixth column (all rows) of the anscombe dataframe:

```
julia> anscombe[:, 6]
```

```
11-element Array{Float64,1}:
 9.14
 8.14
 8.74
 8.77
 9.26
 8.1
 6.13
 3.1
 9.13
 7.26
 4.74
```

and here are columns 1, 2, 3, 5, and 8:

```
julia> anscombe[:, [1, 2, 3, 5, 8]]
```

```
11×5 DataFrame
│ Row │ X1    │ X2    │ X3    │ Y1      │ Y4      │
│     │ Int64 │ Int64 │ Int64 │ Float64 │ Float64 │
├─────┼───────┼───────┼───────┼─────────┼─────────┤
│ 1   │ 10    │ 10    │ 10    │ 8.04    │ 6.58    │
│ 2   │ 8     │ 8     │ 8     │ 6.95    │ 5.76    │
│ 3   │ 13    │ 13    │ 13    │ 7.58    │ 7.71    │
│ 4   │ 9     │ 9     │ 9     │ 8.81    │ 8.84    │
│ 5   │ 11    │ 11    │ 11    │ 8.33    │ 8.47    │
│ 6   │ 14    │ 14    │ 14    │ 9.96    │ 7.04    │
│ 7   │ 6     │ 6     │ 6     │ 7.24    │ 5.25    │
│ 8   │ 4     │ 4     │ 4     │ 4.26    │ 12.5    │
│ 9   │ 12    │ 12    │ 12    │ 10.84   │ 5.56    │
│ 10  │ 7     │ 7     │ 7     │ 4.82    │ 7.91    │
│ 11  │ 5     │ 5     │ 5     │ 5.68    │ 6.89    │
```

Here are the first X and Y columns:

```
julia> anscombe[:, [:X1, :Y1]]
```

```
11×2 DataFrame
│ Row │ X1    │ Y1      │
│     │ Int64 │ Float64 │
├─────┼───────┼─────────┤
│ 1   │ 10    │ 8.04    │
│ 2   │ 8     │ 6.95    │
│ 3   │ 13    │ 7.58    │
│ 4   │ 9     │ 8.81    │
│ 5   │ 11    │ 8.33    │
│ 6   │ 14    │ 9.96    │
│ 7   │ 6     │ 7.24    │
│ 8   │ 4     │ 4.26    │
│ 9   │ 12    │ 10.84   │
│ 10  │ 7     │ 4.82    │
│ 11  │ 5     │ 5.68    │
```

You can supply a regular expression to grab a set of matching column names. Here's the result showing all the columns whose names end in a “2”:

```
julia> anscombe[:, r".2"]
```

```
11×2 DataFrame
│ Row │ X2    │ Y2      │
│     │ Int64 │ Float64 │
├─────┼───────┼─────────┤
│ 1   │ 10    │ 9.14    │
│ 2   │ 8     │ 8.14    │
│ 3   │ 13    │ 8.74    │
│ 4   │ 9     │ 8.77    │
│ 5   │ 11    │ 9.26    │
│ 6   │ 14    │ 8.1     │
│ 7   │ 6     │ 6.13    │
│ 8   │ 4     │ 3.1     │
│ 9   │ 12    │ 9.13    │
│ 10  │ 7     │ 7.26    │
│ 11  │ 5     │ 4.74    │
```

To access the 3rd and 5th columns of the data set anscombe, you can use any of the following:

```
julia> anscombe[:, [:X3, :Y1]]
```

```
11×2 DataFrame
│ Row │ X3    │ Y1      │
```

```
│      │ Int64 │ Float64 │
├──────┼───────┼─────────┤
│ 1    │ 10    │ 8.04    │
│ 2    │ 8     │ 6.95    │
│ 3    │ 13    │ 7.58    │
│ 4    │ 9     │ 8.81    │
│ 5    │ 11    │ 8.33    │
│ 6    │ 14    │ 9.96    │
│ 7    │ 6     │ 7.24    │
│ 8    │ 4     │ 4.26    │
│ 9    │ 12    │ 10.84   │
│ 10   │ 7     │ 4.82    │
│ 11   │ 5     │ 5.68    │
```

```
julia> anscombe[:, [3, 5]]
```

```
11×2 DataFrame
│ Row │ X3    │ Y1      │
│     │ Int64 │ Float64 │
├─────┼───────┼─────────┤
│ 1   │ 10    │ 8.04    │
│ 2   │ 8     │ 6.95    │
│ 3   │ 13    │ 7.58    │
│ 4   │ 9     │ 8.81    │
│ 5   │ 11    │ 8.33    │
│ 6   │ 14    │ 9.96    │
│ 7   │ 6     │ 7.24    │
│ 8   │ 4     │ 4.26    │
│ 9   │ 12    │ 10.84   │
│ 10  │ 7     │ 4.82    │
│ 11  │ 5     │ 5.68    │
```

```
julia> anscombe[:, ["X3", "Y1"]]
```

```
11×2 DataFrame
│ Row │ X3    │ Y1      │
│     │ Int64 │ Float64 │
├─────┼───────┼─────────┤
│ 1   │ 10    │ 8.04    │
│ 2   │ 8     │ 6.95    │
│ 3   │ 13    │ 7.58    │
│ 4   │ 9     │ 8.81    │
│ 5   │ 11    │ 8.33    │
│ 6   │ 14    │ 9.96    │
│ 7   │ 6     │ 7.24    │
│ 8   │ 4     │ 4.26    │
│ 9   │ 12    │ 10.84   │
│ 10  │ 7     │ 4.82    │
│ 11  │ 5     │ 5.68    │
```

You can access the columns using variables. For example, if `a` = "X3" and `b` = "Y1", then:

```
julia> anscombe[:, [a, b]]
```

```
11×2 DataFrame
│ Row │ X3    │ Y1      │
│     │ Int64 │ Float64 │
├─────┼───────┼─────────┤
│ 1   │ 10    │ 8.04    │
│ 2   │ 8     │ 6.95    │
│ 3   │ 13    │ 7.58    │
│ 4   │ 9     │ 8.81    │
│ 5   │ 11    │ 8.33    │
│ 6   │ 14    │ 9.96    │
│ 7   │ 6     │ 7.24    │
│ 8   │ 4     │ 4.26    │
│ 9   │ 12    │ 10.84   │
```

```
│ 10  │ 7     │ 4.82    │
│ 11  │ 5     │ 5.68    │
```

does what you expect.

### Taking rectangular "slices"

```
julia> anscombe[4:6, [:X2,:X4]]
```

```
3×2 DataFrame
│ Row │ X2    │ X4    │
│     │ Int64 │ Int64 │
├─────┼───────┼───────┤
│ 1   │ 9     │ 8     │
│ 2   │ 11    │ 8     │
│ 3   │ 14    │ 8     │
```

which shows rows 4, 5, and 6, and columns X2 and X4. Instead of a range of rows, you can specify individual rows using commas:

```
julia> anscombe[[4, 6, 9], [:X2,:X4]]
```

```
3×2 DataFrame
│ Row │ X2    │ X4    │
│     │ Int64 │ Int64 │
├─────┼───────┼───────┤
│ 1   │ 9     │ 8     │
│ 2   │ 14    │ 8     │
│ 3   │ 12    │ 8     │
```

which shows rows 4, 6, and 9, and columns X2 and X4.

Or you can use index numbers:

```
julia> anscombe[[4, 6, 8], [2, 6, 8]]
```

```
3×3 DataFrame
│ Row │ X2    │ Y2      │ Y4      │
│     │ Int64 │ Float64 │ Float64 │
├─────┼───────┼─────────┼─────────┤
│ 1   │ 9     │ 8.77    │ 8.84    │
│ 2   │ 14    │ 8.1     │ 7.04    │
│ 3   │ 4     │ 3.1     │ 12.5    │
```

To specify a range of rows and columns, use index numbers:

```
julia> anscombe[4:6, 3:5]
```

```
3×3 DataFrame
│ Row │ X3    │ X4    │ Y1      │
│     │ Int64 │ Int64 │ Float64 │
├─────┼───────┼───────┼─────────┤
│ 1   │ 9     │ 8     │ 8.81    │
│ 2   │ 11    │ 8     │ 8.33    │
│ 3   │ 14    │ 8     │ 9.96    │
```

Notice that the row numbering for the returned DataFrames is different — rows 4, 5, and 6 became rows 1, 2, and 3 in the new DataFrame.

As with arrays, use the colon on its own to specify 'all' columns or rows, when you want to view the contents (when you're modifying the contents, the syntax is different, as described later). So, to see Rows 4, 6, 8, and 11, showing all columns:

```
julia> anscombe[[4,6,8,11], :]
```

```
4×8 DataFrame
│ Row │ X1    │ X2    │ X3    │ X4    │ Y1      │ Y2      │ Y3      │ Y4      │
│     │ Int64 │ Int64 │ Int64 │ Int64 │ Float64 │ Float64 │ Float64 │ Float64 │
├─────┼───────┼───────┼───────┼───────┼─────────┼─────────┼─────────┼─────────┤
│ 1   │ 9     │ 9     │ 9     │ 8     │ 8.81    │ 8.77    │ 7.11    │ 8.84    │
│ 2   │ 14    │ 14    │ 14    │ 8     │ 9.96    │ 8.1     │ 8.84    │ 7.04    │
│ 3   │ 4     │ 4     │ 4     │ 19    │ 4.26    │ 3.1     │ 5.39    │ 12.5    │
│ 4   │ 5     │ 5     │ 5     │ 8     │ 5.68    │ 4.74    │ 5.73    │ 6.89    │
```

All rows, columns X1 and Y1:

```
julia> anscombe[:, [:X1, :Y1]]
```

```
11×2 DataFrame
│ Row │ X1    │ Y1      │
│     │ Int64 │ Float64 │
├─────┼───────┼─────────┤
│ 1   │ 10    │ 8.04    │
│ 2   │ 8     │ 6.95    │
│ 3   │ 13    │ 7.58    │
│ 4   │ 9     │ 8.81    │
│ 5   │ 11    │ 8.33    │
│ 6   │ 14    │ 9.96    │
│ 7   │ 6     │ 7.24    │
│ 8   │ 4     │ 4.26    │
│ 9   │ 12    │ 10.84   │
│ 10  │ 7     │ 4.82    │
│ 11  │ 5     │ 5.68    │
```

### Selecting rows with conditions

You can select all the rows of a DataFrame where the elements satisfy one or more conditions.

Here's how to select rows where the value of the element in column Y1 is greater than 7.0:

```
julia> anscombe[anscombe.Y1 .> 7.0, :]
```

```
7×8 DataFrame
│ Row │ X1    │ X2    │ X3    │ X4    │ Y1      │ Y2      │ Y3      │ Y4      │
│     │ Int64 │ Int64 │ Int64 │ Int64 │ Float64 │ Float64 │ Float64 │ Float64 │
├─────┼───────┼───────┼───────┼───────┼─────────┼─────────┼─────────┼─────────┤
│ 1   │ 10    │ 10    │ 10    │ 8     │ 8.04    │ 9.14    │ 7.46    │ 6.58    │
│ 2   │ 13    │ 13    │ 13    │ 8     │ 7.58    │ 8.74    │ 12.74   │ 7.71    │
│ 3   │ 9     │ 9     │ 9     │ 8     │ 8.81    │ 8.77    │ 7.11    │ 8.84    │
│ 4   │ 11    │ 11    │ 11    │ 8     │ 8.33    │ 9.26    │ 7.81    │ 8.47    │
│ 5   │ 14    │ 14    │ 14    │ 8     │ 9.96    │ 8.1     │ 8.84    │ 7.04    │
│ 6   │ 6     │ 6     │ 6     │ 8     │ 7.24    │ 6.13    │ 6.08    │ 5.25    │
│ 7   │ 12    │ 12    │ 12    │ 8     │ 10.84   │ 9.13    │ 8.15    │ 5.56    │
```

The 'inner' phrase `anscombe.Y1 .> 7.0` carries out an element-wise comparison of the values in column Y1, and returns an array of Boolean true or false values, one for each row. Notice the broadcasting operator `.`. These are then used to select rows from the DataFrame. It's as if you'd entered:

```
julia> anscombe[[true, false, true, true, true, true, true, false, true, false, false], :]
```

In a similar way, here's a result that contains every row where the value of the number in column Y1 is greater than that in column Y2:

```
julia> anscombe[anscombe.Y1 .> anscombe.Y2, :]
```

```
6x8 DataFrame
| Row | X1 | X2 | X3 | X4 | Y1    | Y2   | Y3   | Y4   |
|-----|----|----|----|----|-------|------|------|------|
| 1   | 9  | 9  | 9  | 8  | 8.81  | 8.77 | 7.11 | 8.84 |
| 2   | 14 | 14 | 14 | 8  | 9.96  | 8.1  | 8.84 | 7.04 |
| 3   | 6  | 6  | 6  | 8  | 7.24  | 6.13 | 6.08 | 5.25 |
| 4   | 4  | 4  | 4  | 19 | 4.26  | 3.1  | 5.39 | 12.5 |
| 5   | 12 | 12 | 12 | 8  | 10.84 | 9.13 | 8.15 | 5.56 |
| 6   | 5  | 5  | 5  | 8  | 5.68  | 4.74 | 5.73 | 6.89 |
```

Another way to select matching rows is to use the Julia function `filter`.

```
julia> filter(row -> row.Y1 > 7.0, anscombe)
```

Combining two or more conditions is also possible. Here's a result consisting of rows where the value of Y1 is greater than 5.0 *and* that of Y2 is less than 7.0:

```
julia> anscombe[(anscombe.Y1 .> 5.0) .& (anscombe.Y2 .< 7.0), :]
```

```
2×8 DataFrame
│ Row │ X1    │ X2    │ X3    │ X4    │ Y1      │ Y2      │ Y3      │ Y4      │
│     │ Int64 │ Int64 │ Int64 │ Int64 │ Float64 │ Float64 │ Float64 │ Float64 │
├─────┼───────┼───────┼───────┼───────┼─────────┼─────────┼─────────┼─────────┤
│ 1   │ 6     │ 6     │ 6     │ 8     │ 7.24    │ 6.13    │ 6.08    │ 5.25    │
│ 2   │ 5     │ 5     │ 5     │ 8     │ 5.68    │ 4.74    │ 5.73    │ 6.89    │
```

An equivalent using `filter` would be:

```
julia> filter(row -> row.Y1 > 5 &&  row.Y2 < 7.0, anscombe)
```

## Applying functions to columns and rows

You can apply a function to a column. To find out the mean of the values in the column named X2:

```
julia> using Statistics
julia> mean(anscombe.X2)
9.0
```

The Dataframes package provides two convenient utilities, `eachcol()` and `eachrow()`. These can be used for iterating through every column or every row. Each value is a tuple of Symbol (column heading) and DataArray.

To apply the `mean()` function to every column of the DataFrame, you can either use a comprehension:

```
julia>  [mean(col) for col in eachcol(anscombe)]
```

```
8-element Array{Float64,1}:
 9.0
 9.0
 9.0
 9.0
 7.500909090909093
 7.500909090909091
 7.500000000000001
 7.50090909090909
```

which returns a new array containing the mean values for each column.

Alternatively you can use a broadcasted version:

```
julia>  mean.(eachcol(anscombe))
```

Here's the mean of each column:

```
julia> for col in eachcol(anscombe)
           println(mean(col))
        end
```

```
9.0   9.0   9.0   9.0   7.500909090909093   7.500909090909091   7.500000000000001
7.50090909090909
```

The `eachrow()` function provides an iterator for rows:

```
julia> for r in eachrow(anscombe)
         println(r)
      end
```

```
DataFrameRow
│ Row │ X1    │ X2    │ X3    │ X4    │ Y1      │ Y2      │ Y3      │ Y4      │
│     │ Int64 │ Int64 │ Int64 │ Int64 │ Float64 │ Float64 │ Float64 │ Float64 │
├─────┼───────┼───────┼───────┼───────┼─────────┼─────────┼─────────┼─────────┤
│ 1   │ 10    │ 10    │ 10    │ 8     │ 8.04    │ 9.14    │ 7.46    │ 6.58    │
DataFrameRow
│ Row │ X1    │ X2    │ X3    │ X4    │ Y1      │ Y2      │ Y3      │ Y4      │
│     │ Int64 │ Int64 │ Int64 │ Int64 │ Float64 │ Float64 │ Float64 │ Float64 │
├─────┼───────┼───────┼───────┼───────┼─────────┼─────────┼─────────┼─────────┤
│ 2   │ 8     │ 8     │ 8     │ 8     │ 6.95    │ 8.14    │ 6.77    │ 5.76    │
DataFrameRow
│ Row │ X1    │ X2    │ X3    │ X4    │ Y1      │ Y2      │ Y3      │ Y4      │
│     │ Int64 │ Int64 │ Int64 │ Int64 │ Float64 │ Float64 │ Float64 │ Float64 │
├─────┼───────┼───────┼───────┼───────┼─────────┼─────────┼─────────┼─────────┤
│ 3   │ 13    │ 13    │ 13    │ 8     │ 7.58    │ 8.74    │ 12.74   │ 7.71    │
DataFrameRow
│ Row │ X1    │ X2    │ X3    │ X4    │ Y1      │ Y2      │ Y3      │ Y4      │
│     │ Int64 │ Int64 │ Int64 │ Int64 │ Float64 │ Float64 │ Float64 │ Float64 │
├─────┼───────┼───────┼───────┼───────┼─────────┼─────────┼─────────┼─────────┤
│ 4   │ 9     │ 9     │ 9     │ 8     │ 8.81    │ 8.77    │ 7.11    │ 8.84    │
DataFrameRow
```

```
│ Row │ X1    │ X2    │ X3    │ X4    │ Y1      │ Y2      │ Y3      │ Y4      │
│     │ Int64 │ Int64 │ Int64 │ Int64 │ Float64 │ Float64 │ Float64 │ Float64 │
├─────┼───────┼───────┼───────┼───────┼─────────┼─────────┼─────────┼─────────┤
│ 5   │ 11    │ 11    │ 11    │ 8     │ 8.33    │ 9.26    │ 7.81    │ 8.47    │
DataFrameRow
│ Row │ X1    │ X2    │ X3    │ X4    │ Y1      │ Y2      │ Y3      │ Y4      │
│     │ Int64 │ Int64 │ Int64 │ Int64 │ Float64 │ Float64 │ Float64 │ Float64 │
├─────┼───────┼───────┼───────┼───────┼─────────┼─────────┼─────────┼─────────┤
│ 6   │ 14    │ 14    │ 14    │ 8     │ 9.96    │ 8.1     │ 8.84    │ 7.04    │
DataFrameRow
│ Row │ X1    │ X2    │ X3    │ X4    │ Y1      │ Y2      │ Y3      │ Y4      │
│     │ Int64 │ Int64 │ Int64 │ Int64 │ Float64 │ Float64 │ Float64 │ Float64 │
├─────┼───────┼───────┼───────┼───────┼─────────┼─────────┼─────────┼─────────┤
│ 7   │ 6     │ 6     │ 6     │ 8     │ 7.24    │ 6.13    │ 6.08    │ 5.25    │
DataFrameRow
│ Row │ X1    │ X2    │ X3    │ X4    │ Y1      │ Y2      │ Y3      │ Y4      │
│     │ Int64 │ Int64 │ Int64 │ Int64 │ Float64 │ Float64 │ Float64 │ Float64 │
├─────┼───────┼───────┼───────┼───────┼─────────┼─────────┼─────────┼─────────┤
│ 8   │ 4     │ 4     │ 4     │ 19    │ 4.26    │ 3.1     │ 5.39    │ 12.5    │
DataFrameRow
│ Row │ X1    │ X2    │ X3    │ X4    │ Y1      │ Y2      │ Y3      │ Y4      │
│     │ Int64 │ Int64 │ Int64 │ Int64 │ Float64 │ Float64 │ Float64 │ Float64 │
├─────┼───────┼───────┼───────┼───────┼─────────┼─────────┼─────────┼─────────┤
│ 9   │ 12    │ 12    │ 12    │ 8     │ 10.84   │ 9.13    │ 8.15    │ 5.56    │
DataFrameRow
│ Row │ X1    │ X2    │ X3    │ X4    │ Y1      │ Y2      │ Y3      │ Y4      │
│     │ Int64 │ Int64 │ Int64 │ Int64 │ Float64 │ Float64 │ Float64 │ Float64 │
├─────┼───────┼───────┼───────┼───────┼─────────┼─────────┼─────────┼─────────┤
│ 10  │ 7     │ 7     │ 7     │ 8     │ 4.82    │ 7.26    │ 6.42    │ 7.91    │
DataFrameRow
│ Row │ X1    │ X2    │ X3    │ X4    │ Y1      │ Y2      │ Y3      │ Y4      │
│     │ Int64 │ Int64 │ Int64 │ Int64 │ Float64 │ Float64 │ Float64 │ Float64 │
├─────┼───────┼───────┼───────┼───────┼─────────┼─────────┼─────────┼─────────┤
│ 11  │ 5     │ 5     │ 5     │ 8     │ 5.68    │ 4.74    │ 5.73    │ 6.89    │
```

In this dataset, each element of each row is a number, so we could, if we want to, use `eachrow()` to find the (meaningless) mean of each row:

```
julia> for row in eachrow(anscombe)
          println(mean(row))
        end
```

```
8.6525
7.4525
10.47125
8.56625
9.358749999999999
10.492500000000001
6.3375
7.03125
9.71
6.92625
5.755000000000001
```

## Plotting Anscombe's Quartet

Now let's shift focus to statistics.

The built-in `describe()` function lets you quickly calculate the statistical properties of the columns of a dataset. Supply the symbols for the properties you want to know, choosing from `:mean`, `:std`, `:min`, `:q25`, `:median`, `:q75`, `:max`, `:eltype`, `:nunique`, `:first`, `:last`, and `:nmissing`.

```
julia> describe(anscombe, :mean, :std, :min, :median)
```

```
8×5 DataFrame
│ Row │ variable │ mean    │ std     │ min  │ median  │
│     │ Symbol   │ Float64 │ Float64 │ Real │ Float64 │
├─────┼──────────┼─────────┼─────────┼──────┼─────────┤
│ 1   │ X1       │ 9.0     │ 3.31662 │ 4    │ 9.0     │
│ 2   │ X2       │ 9.0     │ 3.31662 │ 4    │ 9.0     │
│ 3   │ X3       │ 9.0     │ 3.31662 │ 4    │ 9.0     │
│ 4   │ X4       │ 9.0     │ 3.31662 │ 8    │ 8.0     │
│ 5   │ Y1       │ 7.50091 │ 2.03157 │ 4.26 │ 7.58    │
│ 6   │ Y2       │ 7.50091 │ 2.03166 │ 3.1  │ 8.14    │
│ 7   │ Y3       │ 7.5     │ 2.03042 │ 5.39 │ 7.11    │
│ 8   │ Y4       │ 7.50091 │ 2.03058 │ 5.25 │ 7.04    │
```

We can compare the XY datasets too:

```
julia> [describe(anscombe[:, xy], :mean, :std, :median) for xy in [[:X1, :Y1], [:X2, :Y2], [:X3, :Y3], [:X4, :Y4]]]
```

```
4-element Array{DataFrame,1}:
 2×4 DataFrame
│ Row │ variable │ mean    │ std     │ median  │
│     │ Symbol   │ Float64 │ Float64 │ Float64 │
├─────┼──────────┼─────────┼─────────┼─────────┤
│ 1   │ X1       │ 9.0     │ 3.31662 │ 9.0     │
│ 2   │ Y1       │ 7.50091 │ 2.03157 │ 7.58    │
 2×4 DataFrame
│ Row │ variable │ mean    │ std     │ median  │
│     │ Symbol   │ Float64 │ Float64 │ Float64 │
├─────┼──────────┼─────────┼─────────┼─────────┤
│ 1   │ X2       │ 9.0     │ 3.31662 │ 9.0     │
│ 2   │ Y2       │ 7.50091 │ 2.03166 │ 8.14    │
 2×4 DataFrame
│ Row │ variable │ mean    │ std     │ median  │
│     │ Symbol   │ Float64 │ Float64 │ Float64 │
├─────┼──────────┼─────────┼─────────┼─────────┤
│ 1   │ X3       │ 9.0     │ 3.31662 │ 9.0     │
│ 2   │ Y3       │ 7.5     │ 2.03042 │ 7.11    │
 2×4 DataFrame
│ Row │ variable │ mean    │ std     │ median  │
│     │ Symbol   │ Float64 │ Float64 │ Float64 │
├─────┼──────────┼─────────┼─────────┼─────────┤
│ 1   │ X4       │ 9.0     │ 3.31662 │ 8.0     │
│ 2   │ Y4       │ 7.50091 │ 2.03058 │ 7.04    │
```

One last look, at the correlation between the XY datasets:

```
julia> [cor(anscombe[:, first(xy)], anscombe[:, last(xy)]) for xy in [[:X1, :Y1], [:X2, :Y2], [:X3, :Y3], [:X4, :Y4]]]
```

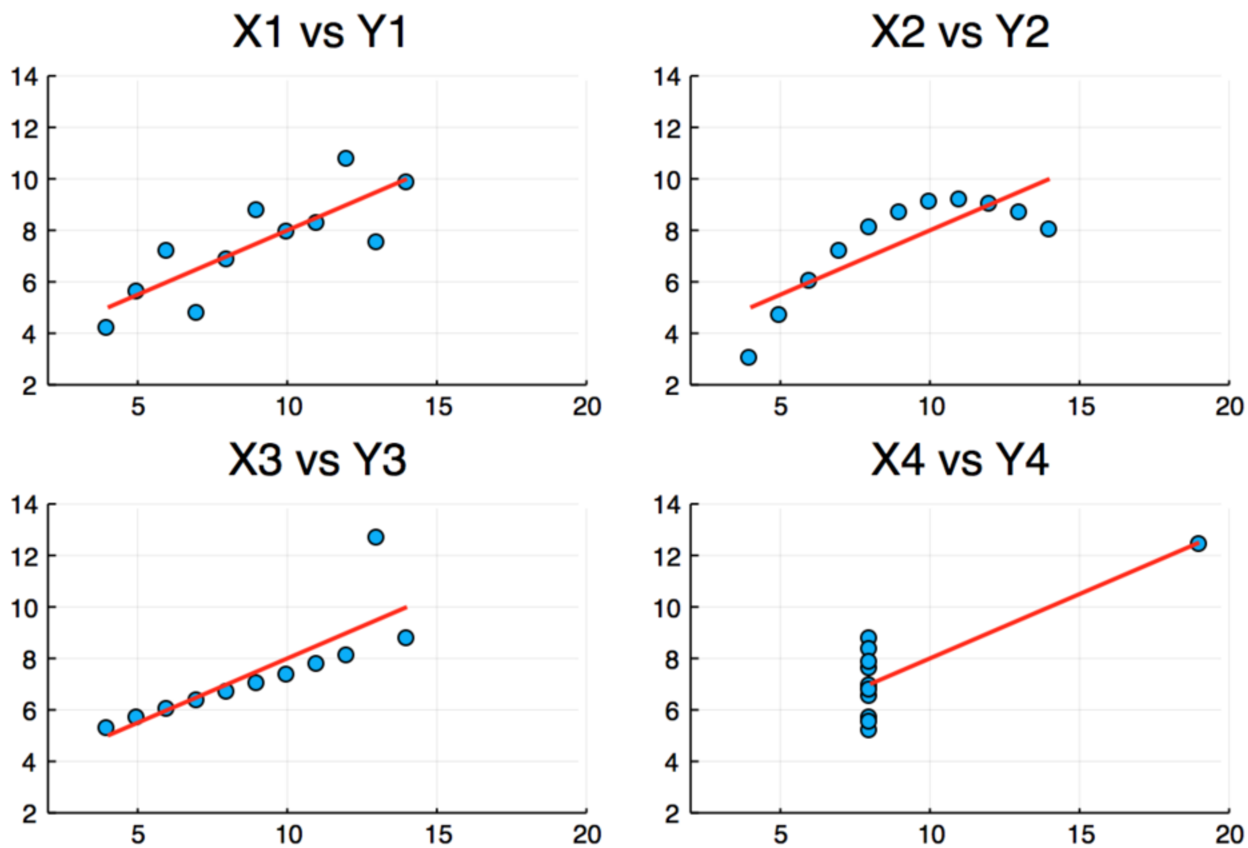
```
4-element Array{Float64,1}:
 0.8164205163448398
 0.8162365060002429
 0.8162867394895982
 0.8165214368885028
```

Notice how similar these correlations are: X1Y1 is the same as X2Y2, X3Y3, X4Y4.

Each of the four datasets has the same mean, median, standard deviation, and correlation coefficient between x and y. Judging by the simple summary statistics, you'd think that they were pretty similar. Let's plot them:

```
using StatsPlots # add this package if necessary
@df anscombe scatter([:X1 :X2 :X3 :X4], [:Y1 :Y2 :Y3 :Y4],
           smooth=true,
```

```
        line = :red,
        linewidth = 2,
        title= ["X$i vs Y$i" for i in (1:4)'],
        legend = false,
        layout = 4,
        xlimits = (2, 20),
        ylimits = (2, 14))
```




Anscombe's Quartet comprises four datasets that have nearly identical simple statistical properties, but are actually very different. Each dataset consists of eleven (x,y) points. They were carefully constructed in 1973 by the statistician Francis Anscombe to demonstrate both the importance of looking at your data, and of graphing that data, before relying on the summary statistics, and the effect of outliers on statistical properties. The first appears to show a simple linear relationship, corresponding to two variables correlated and following the assumption of normality.

The second set of points is not distributed normally; there is an obvious relationship between the two variables, but it isn't linear, and the Pearson correlation coefficient is not really relevant.

In the third set, the distribution is linear, but with a different regression line, which is offset by the one outlier which exerts enough influence to alter the regression line and lower the correlation coefficient from 1 to 0.816.

Finally, the fourth set shows an example when one outlier is enough to produce a high correlation coefficient, even though the relationship between the two variables is not linear.

The quartet is still often used to illustrate the importance of looking at a set of data graphically before starting to analyze according to a particular type of relationship, and the inadequacy of basic statistic properties for describing realistic datasets.

## Regression and Models

If you want to find a linear regression line for the datasets, you can turn to the GLM (Generalized Linear Models) package.

```
using GLM, StatsModels # add these packages if necessary
```

To create a linear model, you specify a formula using the `@formula` macro, supplying the column names and the name of the DataFrame. The result is a regression model.

```
julia> linearmodel = fit(LinearModel, @formula(Y1 ~ X1), anscombe)
```

```
 StatsModels.DataFrameRegressionModel{GLM.LinearModel{GLM.LmResp{Array{Float64,1}},
 GLM.DensePredChol{Float64,Base.LinAlg.Cholesky{Float64,Array{Float64,2}}}},Array{Float64,2}}
Y1 ~ 1 + X1

Coefficients:
──────────────────────────────────────────────────────────────────────────
             Estimate  Std. Error  t value  Pr(>|t|)  Lower 95%  Upper 95%
──────────────────────────────────────────────────────────────────────────
(Intercept)  3.00009    1.12475    2.66735    0.0257   0.455737   5.54444
X1           0.500091   0.117906   4.24146    0.0022   0.23337    0.766812
──────────────────────────────────────────────────────────────────────────
```

Useful functions in the GLM package for working with linear models include `summary()`, and `coef()`.

```
julia> summary(linearmodel)
```

```
 "StatsModels.DataFrameRegressionModel{GLM.LinearModel{GLM.LmResp{Array{Float64,1}},
  GLM.DensePredChol{Float64,Base.LinAlg.Cholesky{Float64,Array{Float64,2}}}}, Array{Float64,2}}"
```

The `coef()` function returns the two useful coefficients that define the regression line: the estimated intercept and the estimated slope:

```
julia> coef(linearmodel)
```

```
2-element Array{Float64,1}:
 3.0000909090909054
 0.5000909090909096
```

It's now easy to produce a function for the regression line in the form `y = a x + c`:

```
julia> f(x) = coef(linearmodel)[2] * x + coef(linearmodel)[1]
```

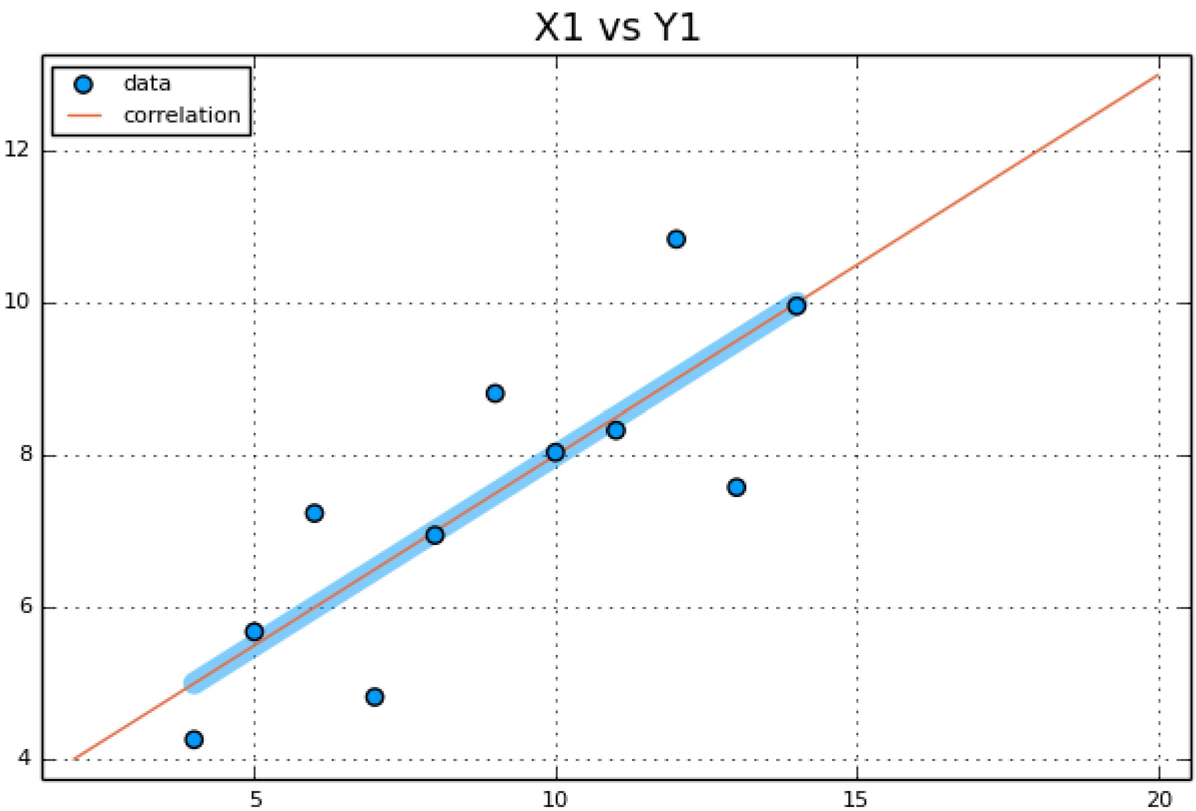
```
f (generic function with 1 method)
```

Now that we have `f()` as a function describing the regression line, it can be drawn in a plot. Here we plot the first series, and add a plot of the function `f(x)` with x running from 2 to 20, and see how it compares with the smoothing line we used earlier.

```
p1 = plot(anscombe[!, :X1], anscombe[!, :Y1],
    smooth=true,
```

```
    seriestype=:scatter,
    title = "X1 vs Y1",
    linewidth=8,
    linealpha=0.5,
    label="data")

plot!(f, 2, 20, label="correlation")
```




# Working with DataFrames

Not all datasets are as consistent and tidy as the examples in RDatasets. In the real world, it's possible that you'll read some data into a DataFrame, only to discover that it's got a few problems, with inconsistently formatted and/or missing elements.

For this section, we'll create a simple test DataFrame by hand, defining the columns one by one. It's a short extract from what might be a periodic table.

```
ptable = DataFrame( Number       =   [1,    2,     6,     8,     26     ],
                    Name         =   ["Hydrogen",   "Helium",    "Carbon",    "Oxygen",    "Iron"    ],
                    AtomicWeight =   [1.0079,     4.0026,  12.0107, 15.9994, 55.845    ],
                    Symbol       =   ["H",      "He",     "C",      "O",    "Fe"  ],
                    Discovered   =   [1776,     1895,     0,      1774,     missing    ])
```

```
5x5 DataFrame
| Row | Number | Name       | AtomicWeight | Symbol | Discovered |
|-----|--------|------------|--------------|--------|------------|
| 1   | 1      | "Hydrogen" | 1.0079       | "H"    | 1776       |
| 2   | 2      | "Helium"   | 4.0026       | "He"   | 1895       |
| 3   | 6      | "Carbon"   | 12.0107      | "C"    | 0          |
| 4   | 8      | "Oxygen"   | 15.9994      | "O"    | 1774       |
| 5   | 26     | "Iron"     | 55.845       | "Fe"   | Missing    |
```

## The case of the missing value

A first look with `describe()` reveals that the Discovered column has some missing values. It's the column that contains the discovered year for Iron, which was marked in the source data as missing.

```
julia> describe(ptable)
```

```
5×8 DataFrame
│ Row │ variable     │ mean    │ min    │ median  │ max    │ nunique │ nmissing │ eltype   │
│     │ Symbol       │ Union…  │ Any    │ Union…  │ Any    │ Union…  │ Union…   │ DataType │
├─────┼──────────────┼─────────┼────────┼─────────┼────────┼─────────┼──────────┼──────────┤
│ 1   │ Number       │ 8.6     │ 1      │ 6.0     │ 26     │         │          │ Int64    │
│ 2   │ Name         │         │ Carbon │         │ Oxygen │ 5       │          │ String   │
│ 3   │ AtomicWeight │ 17.7731 │ 1.0079 │ 12.0107 │ 55.845 │         │          │ Float64  │
│ 4   │ Symbol       │         │ C      │         │ O      │ 5       │          │ String   │
│ 5   │ Discovered   │ 1361.25 │ 0      │ 1775.0  │ 1895   │         │ 1        │ Int64    │
```

The problem of missing fields is one of the important issues you have to confront when working with tabular data. Sometimes, not every field in a table has data. For any number of reasons, there might be missing observations or data points. Unless you're aware of these missing values, they can cause problems. For example, mean and other statistical calculations will be incorrect if you don't notice and account for missing values. (Before the use of special markers for missing values, people used to enter "obviously wrong" numbers such as -99 for a missing temperature reading; not spotting these in temperature datasets have been known to cause problems...) Also, it can be difficult to apply formulas to a mixture of numeric and string values and missing.

You'll come across various ways in which the compilers of the data have indicated that the values are missing. Sometimes the values are represented by 0 or some other 'impossible' number, or by a text string such as "n/a". Sometimes — particularly with Tab and Comma-separated files, they're just left empty.

To address this issue, there's a special data type, `Missing`, which indicates that there isn't a usable value at this location. If used consistently, the DataFrames package and its support for Missing fields will allow you to get the most out of your datasets, while making sure your calculations are accurate and not 'poisoned' by missing values. The `missing` in the Iron/Discovered cell allows the DataFrames package to 'tag' that cell as being Missing.

But there's another problem that's not revealed here. The Discovered year of Carbon was set to 0, chosen to mean 'a long time ago', as it's not a valid year. (After 1 BC comes 1 AD, so Year 0 doesn't exist.) This 0 value should also be marked as being Missing in the DataFrame, before it causes havoc.

There are three approaches to making a DataFrame use missing values consistently:

- Edit the file/data outside Julia before importing, using a text editor or similar. This is sometimes the quickest and easiest way. In our simple example, we used the word missing in the file. This was enough to have the location marked as a Missing value.

- Use the options provided by the CSV package when importing the data. These let you specify rules for identifying certain values as Missing.

- Repair the file before you import it into a DataFrame.

## How to fix missing values with readtable()

There's a lot of flexibility built in to `CSV.read`.

By default, any missing values (empty fields in the table) are replaced by `missing`. With the `missingstrings` option, you can specify a group of strings that will all be converted to NA values when they're encountered in the source text:

```
pt1 = CSV.read("file.tsv", missingstrings=["NA", "na", "n/a", "missing"])
```

This addresses most of the problems with the varying methods for indicating unavailable values in the original file. It's also possible to add "0" (zero as a string) to the list of `nastrings`, if it's the case that 0 isn't used for any legitimate values.

## Using DataFrames with missing values

If a column contains one or more missing values, you'll find that some calculations don't work.

For example, you can apply a function to ordinary columns easily enough. So it's easy to calculate the mean of the AtomicWeight column:

```
julia> mean(ptable[:, :AtomicWeight])
17.77312
```

But, because there is a `missing` value in the Discovered column, you can't apply a function to it:

```
julia> mean(ptable[:, :Discovered])
missing
```

Because just one of the fields contains a missing value, the entire calculation is abandoned, and the missing value propagates back to the top level.

There are two ways to fix this: edit the table so that the missing value is converted to a real value, or, when running calculations, make sure that the relevant field isn't included.

We also have that Year 0 problem to address...

## Looking for missing values and others in DataFrames

To look for missing values in a DataFrame, you can try writing code using the `ismissing()` function. This lets you test a DataFrame cell for missing values:

```
nrows, ncols = size(ptable)
for row in 1:nrows
      for col in 1:ncols
        if ismissing(ptable[row,col])
         println("$(names(ptable)[col]) value for $(ptable[row,:Name]) is missing!")
        end
      end
end
```

```
Discovered value for Iron is missing!
```

You might think to use similar code to check for the 0 values for Discovered, and replace them with missing (should you consider that to be a good way to mark an element as being not-discovered in a particular year). But this doesn't work:

```
for row in 1:nrows
   if ptable[row, :Discovered] == 0
       println("it's zero")
   end
end
```

because:

```
TypeError: non-boolean (Missings.Missing) used in boolean context
```

The problem here is that you're checking the value of each cell in a Boolean comparison, but you can't compare missing values with numbers: any comparison has to return "cannot compare them" rather than simply `true` or `false`.

Instead, you can write the loop like this:

```
for row in 1:nrows
    if ismissing(ptable[row, :Discovered])
        println("skipping missing values")
    else
        println("the value is $(ptable[row, :Discovered])")
    end
end
```

```
the value is 1776
the value is 1895
the value is 0
the value is 1774
skipping missing values
```

The `ismissing()` function is also useful in other contexts. This is a quick way of locating the index numbers of missing values in columns (notice the `.` broadcast operator):

```
julia> findall(ismissing.(ptable[:, :Discovered]))
1-element Array{Int64,1}:
5
```

and this next line returns a new DataFrame of all rows where the Discovered column contains missing values:

```
julia> ptable[findall(ismissing.(ptable[:,:Discovered])), :]
```

```
 1×5 DataFrame
│ Row │ Number │ Name   │ AtomicWeight │ Symbol │ Discovered │
│     │ Int64  │ String │ Float64      │ String │ Int64?     │
├─────┼────────┼────────┼──────────────┼────────┼────────────┤
│ 1   │ 26     │ Iron   │ 55.845       │ Fe     │ missing    │
```

By using `!ismissing` you can return a DataFrame that contains no rows with missing values.

You can use `ismissing()` to select rows with missing values in specific columns and set them all to a new value. For example, this code finds missing discovery years and sets them to 0:

```
julia> ptable[ismissing.(ptable[:, :Discovered]), :Discovered] .= 0
0
```

```
julia> ptable
```

```
5×5 DataFrame
│ Row │ Number │ Name     │ AtomicWeight │ Symbol │ Discovered │
│     │ Int64  │ String   │ Float64      │ String │ Int64?     │
├─────┼────────┼──────────┼──────────────┼────────┼────────────┤
│ 1   │ 1      │ Hydrogen │ 1.0079       │ H      │ 1776       │
│ 2   │ 2      │ Helium   │ 4.0026       │ He     │ 1895       │
│ 3   │ 6      │ Carbon   │ 12.0107      │ C      │ 0          │
│ 4   │ 8      │ Oxygen   │ 15.9994      │ O      │ 1774       │
│ 5   │ 26     │ Iron     │ 55.845       │ Fe     │ 0          │
```

## Repairing DataFrames

To clean your data, you can write a short fragment of code to change values that aren't acceptable. This code looks at every cell and changes any "n/a", "0", or 0 values to `missing`. Notice that the first test is `ismissing()` — that takes care of cases where the element is already a missing value; these are skipped (otherwise the comparisons that follow might fail).

```
for row in 1:size(ptable, 1) # or nrow(ptable)
   for col in 1:size(ptable, 2) # or ncol(ptable)
       println("processing row $row column $col ")
       temp = ptable[row,col]
       if ismissing(temp)
          println("skipping missing")
       elseif temp == "n/a" || temp == "0" || temp == 0
          ptable[row, col] = missing
          println("changed row $row column $col ")
       end
     end
end
```

```
processing row 1 column 1
processing row 1 column 2
processing row 1 column 3
processing row 1 column 4
processing row 1 column 5
processing row 2 column 1
processing row 2 column 2
processing row 2 column 3
processing row 2 column 4
processing row 2 column 5
processing row 3 column 1
processing row 3 column 2
processing row 3 column 3
processing row 3 column 4
processing row 3 column 5
changed row 3 column 5
processing row 4 column 1
processing row 4 column 2
processing row 4 column 3
processing row 4 column 4
processing row 4 column 5
processing row 5 column 1
processing row 5 column 2
processing row 5 column 3
processing row 5 column 4
```

```
processing row 5 column 5
changed row 5 column 5
```

```
julia> ptable
```

```
5×5 DataFrame
│ Row │ Number │ Name     │ AtomicWeight │ Symbol │ Discovered │
│     │ Int64  │ String   │ Float64      │ String │ Int64?     │
├─────┼────────┼──────────┼──────────────┼────────┼────────────┤
│ 1   │ 1      │ Hydrogen │ 1.0079       │ H      │ 1776       │
│ 2   │ 2      │ Helium   │ 4.0026       │ He     │ 1895       │
│ 3   │ 6      │ Carbon   │ 12.0107      │ C      │ missing    │
│ 4   │ 8      │ Oxygen   │ 15.9994      │ O      │ 1774       │
│ 5   │ 26     │ Iron     │ 55.845       │ Fe     │ missing    │
```

Now the Discovered column has two missing values, as the discovery date of Carbon is also now considered to be unknown.

### Working with missing values: completecases() and dropmissing()

To find, say, the maximum value of the Discovered column (which we know contains missing values), you can use the `completecases()` function. This takes a DataFrame and returns flags to indicate which rows are valid. These can then be used to select rows which are guaranteed to not contain missing values:

```
julia> maximum(ptable[completecases(ptable), :].Discovered)
1895
```

This allows you to write code that should work as expected, because rows with one or more missing values are excluded from consideration.

The `dropmissing()` function returns a copy of a data frame without missing values.

```
julia> dropmissing(ptable)
```

```
3×5 DataFrame
│ Row │ Number │ Name     │ AtomicWeight │ Symbol │ Discovered │
│     │ Int64  │ String   │ Float64      │ String │ Int64      │
├─────┼────────┼──────────┼──────────────┼────────┼────────────┤
│ 1   │ 1      │ Hydrogen │ 1.0079       │ H      │ 1776       │
│ 2   │ 2      │ Helium   │ 4.0026       │ He     │ 1895       │
│ 3   │ 8      │ Oxygen   │ 15.9994      │ O      │ 1774       │
```

So an alternative to the `completecases()` approach is:

```
julia> maximum(dropmissing(ptable).Discovered)
1895
```

## Modifying DataFrames

### Adding, deleting, and renaming columns

To add a column, you could do this:

```
hcat(ptable, axes(ptable, 1))
```

which adds another column of integers (which will be called `:x1`) from 1 to *n*. (This creates a copy of the DataFrame, and we haven't changed `ptable` or assigned the new DataFrame to a symbol.

Instead, let's add Melting and Boiling points of our chosen elements:

```
julia> ptable[!, :MP] = [-259, -272, 3500, -218, 1535] # notice the !
julia> ptable[!, :BP] = [-253, -269, 4827, -183, 2750]
julia> ptable
```

```
5×7 DataFrame
│ Row │ Number │ Name     │ AtomicWeight │ Symbol │ Discovered │ MP    │ BP    │
│     │ Int64  │ String   │ Float64      │ String │ Int64?     │ Int64 │ Int64 │
├─────┼────────┼──────────┼──────────────┼────────┼────────────┼───────┼───────┤
│ 1   │ 1      │ Hydrogen │ 1.0079       │ H      │ 1776       │ -259  │ -253  │
│ 2   │ 2      │ Helium   │ 4.0026       │ He     │ 1895       │ -272  │ -269  │
│ 3   │ 6      │ Carbon   │ 12.0107      │ C      │ missing    │ 3500  │ 4827  │
│ 4   │ 8      │ Oxygen   │ 15.9994      │ O      │ 1774       │ -218  │ -183  │
│ 5   │ 26     │ Iron     │ 55.845       │ Fe     │ missing    │ 1535  │ 2750  │
```

Notice the use of the different syntax to access columns when changing them. If we just want to look at values, you can use `[:, ColumnName]`, which provides you with a read-only view of the DataFrame. If you want to change the values, use `[!, ColumnName]`. The `!` is the usual Julian clue that indicates a function that might modify the data arguments.

To illustrate how to create a new column based on things in the other columns, we'll add a column called `Liquid` showing for how many degrees C an element remains liquid (i.e. BP - MP):

```
julia> ptable[!, :Liquid] = map((x, y) -> y - x, ptable[:, :MP], ptable[:, :BP])
```

```
 5-element Array{Int64,1}:
    6
    3
 1327
   35
 1215
```

(or simply:

```
julia> ptable[!, :Liquid] = ptable[:, :BP] - ptable[:, :MP]
```

)

```
5×8 DataFrame
│ Row │ Number │ Name     │ AtomicWeight │ Symbol │ Discovered │ MP    │ BP    │ Liquid │
│     │ Int64  │ String   │ Float64      │ String │ Int64?     │ Int64 │ Int64 │ Int64  │
├─────┼────────┼──────────┼──────────────┼────────┼────────────┼───────┼───────┼────────┤
│ 1   │ 1      │ Hydrogen │ 1.0079       │ H      │ 1776       │ -259  │ -253  │ 6      │
│ 2   │ 2      │ Helium   │ 4.0026       │ He     │ 1895       │ -272  │ -269  │ 3      │
│ 3   │ 6      │ Carbon   │ 12.0107      │ C      │ missing    │ 3500  │ 4827  │ 1327   │
│ 4   │ 8      │ Oxygen   │ 15.9994      │ O      │ 1774       │ -218  │ -183  │ 35     │
│ 5   │ 26     │ Iron     │ 55.845       │ Fe     │ missing    │ 1535  │ 2750  │ 1215   │
```

To add or replace a column of a DataFrame with another column of data (ie an array of the right length), use:

```
julia> ptable[!, :Temp] = axes(ptable, 1)
Base.OneTo(5)
```

Let's do it for real:

```
julia> ptable[!, :Temp] = map((x, y) -> y * x, ptable[:, :Liquid], ptable[:, :AtomicWeight])
```

```
5×9 DataFrame
│ Row │ Number │ Name     │ AtomicWeight │ Symbol │ Discovered │ MP    │ BP    │ Liquid │ Temp    │
│     │ Int64  │ String   │ Float64      │ String │ Int64?     │ Int64 │ Int64 │ Int64  │ Float64 │
├─────┼────────┼──────────┼──────────────┼────────┼────────────┼───────┼───────┼────────┼─────────┤
│ 1   │ 1      │ Hydrogen │ 1.0079       │ H      │ 1776       │ -259  │ -253  │ 6      │ 6.0474  │
│ 2   │ 2      │ Helium   │ 4.0026       │ He     │ 1895       │ -272  │ -269  │ 3      │ 12.0078 │
│ 3   │ 6      │ Carbon   │ 12.0107      │ C      │ missing    │ 3500  │ 4827  │ 1327   │ 15938.2 │
│ 4   │ 8      │ Oxygen   │ 15.9994      │ O      │ 1774       │ -218  │ -183  │ 35     │ 559.979 │
│ 5   │ 26     │ Iron     │ 55.845       │ Fe     │ missing    │ 1535  │ 2750  │ 1215   │ 67851.7 │
```

The values in Temp were replaced with the result of multiplying the atomic weight by the Liquid range.

You might want to add a column that shows the Melting Point in the obsolete Fahrenheit units:

```
julia> ptable[!, :MP_in_F] = map(deg -> 32 + (deg * 1.8), ptable[:, :MP])
```

```
5×10 DataFrame
│ Row │ Number │ Name     │ AtomicWeight │ Symbol │ Discovered │ MP    │ BP    │ Liquid │ Temp    │ MP_in_F │
│     │ Int64  │ String   │ Float64      │ String │ Int64?     │ Int64 │ Int64 │ Int64  │ Float64 │ Float64 │
├─────┼────────┼──────────┼──────────────┼────────┼────────────┼───────┼───────┼────────┼─────────┼─────────┤
│ 1   │ 1      │ Hydrogen │ 1.0079       │ H      │ 1776       │ -259  │ -253  │ 6      │ 6.0474  │ -434.2  │
│ 2   │ 2      │ Helium   │ 4.0026       │ He     │ 1895       │ -272  │ -269  │ 3      │ 12.0078 │ -457.6  │
│ 3   │ 6      │ Carbon   │ 12.0107      │ C      │ missing    │ 3500  │ 4827  │ 1327   │ 15938.2 │ 6332.0  │
│ 4   │ 8      │ Oxygen   │ 15.9994      │ O      │ 1774       │ -218  │ -183  │ 35     │ 559.979 │ -360.4  │
│ 5   │ 26     │ Iron     │ 55.845       │ Fe     │ missing    │ 1535  │ 2750  │ 1215   │ 67851.7 │ 2795.0  │
```

It's easy to rename columns:

```
julia> rename!(ptable, :Temp => :Junk)
```

and

```
julia> rename!(ptable, [f => t for (f, t) = zip([:MP, :BP], [:Melt, :Boil])])
```

```
 5×10 DataFrame
│ Row │ Number │ Name     │ AtomicWeight │ Symbol │ Discovered │ Melt  │ Boil  │ Liquid │ Junk    │ MP_in_F │
│     │ Int64  │ String   │ Float64      │ String │ Int64?     │ Int64 │ Int64 │ Int64  │ Float64 │ Float64 │
├─────┼────────┼──────────┼──────────────┼────────┼────────────┼───────┼───────┼────────┼─────────┼─────────┤
│ 1   │ 1      │ Hydrogen │ 1.0079       │ H      │ 1776       │ -259  │ -253  │ 6      │ 6.0474  │ -434.2  │
│ 2   │ 2      │ Helium   │ 4.0026       │ He     │ 1895       │ -272  │ -269  │ 3      │ 12.0078 │ -457.6  │
│ 3   │ 6      │ Carbon   │ 12.0107      │ C      │ missing    │ 3500  │ 4827  │ 1327   │ 15938.2 │ 6332.0  │
│ 4   │ 8      │ Oxygen   │ 15.9994      │ O      │ 1774       │ -218  │ -183  │ 35     │ 559.979 │ -360.4  │
│ 5   │ 26     │ Iron     │ 55.845       │ Fe     │ missing    │ 1535  │ 2750  │ 1215   │ 67851.7 │ 2795.0  │
```

(There's also a `rename()` function (without the exclamation mark) which doesn't change the original DataFrame.)

The `select!()` function creates a new data frame that contains the selected columns. So to delete columns, use `select!()` together with `Not`, which deselects a column that you don't want to include.

```
julia> select!(ptable, Not(:Junk))
```

```
5×9 DataFrame
│ Row │ Number │ Name     │ AtomicWeight │ Symbol │ Discovered │ Melt  │ Boil  │ Liquid │ MP_in_F │
│     │ Int64  │ String   │ Float64      │ String │ Int64?     │ Int64 │ Int64 │ Int64  │ Float64 │
├─────┼────────┼──────────┼──────────────┼────────┼────────────┼───────┼───────┼────────┼─────────┤
│ 1   │ 1      │ Hydrogen │ 1.0079       │ H      │ 1776       │ -259  │ -253  │ 6      │ -434.2  │
│ 2   │ 2      │ Helium   │ 4.0026       │ He     │ 1895       │ -272  │ -269  │ 3      │ -457.6  │
│ 3   │ 6      │ Carbon   │ 12.0107      │ C      │ missing    │ 3500  │ 4827  │ 1327   │ 6332.0  │
│ 4   │ 8      │ Oxygen   │ 15.9994      │ O      │ 1774       │ -218  │ -183  │ 35     │ -360.4  │
│ 5   │ 26     │ Iron     │ 55.845       │ Fe     │ missing    │ 1535  │ 2750  │ 1215   │ 2795.0  │
```

## Adding and deleting rows

It's easy to add rows. Use `push!` with suitable data of the right length and type:

```
julia> push!(ptable, [29, "Copper", 63.546, "Cu", missing, 1083, 2567, 2567-1083, map(deg -> 32 + (deg * 1.8), 1083)])
```

```
julia> ptable
```

```
6×9 DataFrame
│ Row │ Number │ Name     │ AtomicWeight │ Symbol │ Discovered │ Melt  │ Boil  │ Liquid │ MP_in_F │
│     │ Int64  │ String   │ Float64      │ String │ Int64?     │ Int64 │ Int64 │ Int64  │ Float64 │
├─────┼────────┼──────────┼──────────────┼────────┼────────────┼───────┼───────┼────────┼─────────┤
│ 1   │ 1      │ Hydrogen │ 1.0079       │ H      │ 1776       │ -259  │ -253  │ 6      │ -434.2  │
│ 2   │ 2      │ Helium   │ 4.0026       │ He     │ 1895       │ -272  │ -269  │ 3      │ -457.6  │
│ 3   │ 6      │ Carbon   │ 12.0107      │ C      │ missing    │ 3500  │ 4827  │ 1327   │ 6332.0  │
│ 4   │ 8      │ Oxygen   │ 15.9994      │ O      │ 1774       │ -218  │ -183  │ 35     │ -360.4  │
│ 5   │ 26     │ Iron     │ 55.845       │ Fe     │ missing    │ 1535  │ 2750  │ 1215   │ 2795.0  │
│ 6   │ 29     │ Copper   │ 63.546       │ Cu     │ missing    │ 1083  │ 2567  │ 1484   │ 1981.4  │
```

Those missing values should be replaced soon using the functions from earlier. We can locate the new element by name and change the value for `:Liquid` like this:

```
julia>  ptable[[occursin(r"Copper", elementname) for elementname in ptable[:, :Name]], :][:, :Liquid] .= 2567 - 1083
```

```
1-element view(::Array{Int64,1}, :) with eltype Int64:
 14843
```

or we could use the atomic number to obtain access to the right row and do it that way:

```
julia> ptable[ptable[!, :Number] .== 6, :][:, :Liquid] .= 4827 - 3500
```

```
1-element view(::Array{Int64,1}, :) with eltype Int64:
 1327
```

To delete rows, use the `delete!()` function (carefully), with one or more row specifiers:

```
julia> temp = select(ptable, r".") # make a copy
julia> delete!(temp, 3:5)
```

```
3×9 DataFrame
│ Row │ Number │ Name     │ AtomicWeight │ Symbol │ Discovered │ Melt  │ Boil  │ Liquid │ MP_in_F │
│     │ Int64  │ String   │ Float64      │ String │ Int64?     │ Int64 │ Int64 │ Int64  │ Float64 │
├─────┼────────┼──────────┼──────────────┼────────┼────────────┼───────┼───────┼────────┼─────────┤
│ 1   │ 1      │ Hydrogen │ 1.0079       │ H      │ 1776       │ -259  │ -253  │ 6      │ -434.2  │
│ 2   │ 2      │ Helium   │ 4.0026       │ He     │ 1895       │ -272  │ -269  │ 3      │ -457.6  │
│ 3   │ 26     │ Iron     │ 55.845       │ Fe     │ missing    │ 1535  │ 2750  │ 1215   │ 2795.0  │
```

Alternatively, you could delete rows by specifying a condition. For example, to keep rows where the Boiling Point is less than 100 C, you could just find the rows that are greater than or equal to 100 C, then assign a variable to keep the result:

```
julia> ptable1 = ptable[ptable[:, :Boil] .>= 100, :]
```

```
3×9 DataFrame
│ Row │ Number │ Name   │ AtomicWeight │ Symbol │ Discovered │ Melt  │ Boil  │ Liquid │ MP_in_F │
│     │ Int64  │ String │ Float64      │ String │ Int64?     │ Int64 │ Int64 │ Int64  │ Float64 │
├─────┼────────┼────────┼──────────────┼────────┼────────────┼───────┼───────┼────────┼─────────┤
│ 1   │ 6      │ Carbon │ 12.0107      │ C      │ missing    │ 3500  │ 4827  │ 1327   │ 6332.0  │
│ 2   │ 26     │ Iron   │ 55.845       │ Fe     │ missing    │ 1535  │ 2750  │ 1215   │ 2795.0  │
│ 3   │ 29     │ Copper │ 63.546       │ Cu     │ missing    │ 1083  │ 2567  │ 1484   │ 1981.4  │
```

## Finding values in DataFrames

To find values, the basic idea is to use an elementwise operator or function that examines all rows and returns an array of Boolean values to indicate whether each cell meets the criteria for each row:

```
julia> ptable[:, :Melt] .< 100
```

```
6-element BitArray{1}:
 1
 1
 0
 1
 0
 0
```

then use this Boolean array to select the rows:

```
julia> ptable[ptable[:, :Melt] .< 100, :]
```

```
3×9 DataFrame
│ Row │ Number │ Name     │ AtomicWeight │ Symbol │ Discovered │ Melt  │ Boil  │ Liquid │ MP_in_F │
│     │ Int64  │ String   │ Float64      │ String │ Int64?     │ Int64 │ Int64 │ Int64  │ Float64 │
├─────┼────────┼──────────┼──────────────┼────────┼────────────┼───────┼───────┼────────┼─────────┤
│ 1   │ 1      │ Hydrogen │ 1.0079       │ H      │ 1776       │ -259  │ -253  │ 6      │ -434.2  │
│ 2   │ 2      │ Helium   │ 4.0026       │ He     │ 1895       │ -272  │ -269  │ 3      │ -457.6  │
│ 3   │ 8      │ Oxygen   │ 15.9994      │ O      │ 1774       │ -218  │ -183  │ 35     │ -360.4  │
```

You could use do this to return rows where a value in a column matches a regular expression:

```
julia> ptable[[occursin(r"Co", elementname) for elementname in ptable[:, :Name]], :]
```

```
1×9 DataFrame
│ Row │ Number │ Name   │ AtomicWeight │ Symbol │ Discovered │ Melt  │ Boil  │ Liquid │ MP_in_F │
│     │ Int64  │ String │ Float64      │ String │ Int64?     │ Int64 │ Int64 │ Int64  │ Float64 │
```

```
│ 1   │ 29     │ Copper │ 63.546        │ Cu     │ missing    │ 1083  │ 2567  │ 1484   │ 1981.4  │
```

and you can edit elements in the same way:

```
julia> ptable[[occursin(r"Copper", elementname) for elementname in ptable.Name], :][:, :Liquid] .= Ref(2567 - 1083)
```

```
1-element view(::Array{Int64,1}, :) with eltype Int64:
 1484

6×9 DataFrame
│ Row │ Number │ Name     │ AtomicWeight │ Symbol │ Discovered │ Melt  │ Boil  │ Liquid │ MP_in_F │
│     │ Int64  │ String   │ Float64      │ String │ Int64?     │ Int64 │ Int64 │ Int64  │ Float64 │
├─────┼────────┼──────────┼──────────────┼────────┼────────────┼───────┼───────┼────────┼─────────┤
│ 1   │ 1      │ Hydrogen │ 1.0079       │ H      │ 1776       │ -259  │ -253  │ 6      │ -434.2  │
│ 2   │ 2      │ Helium   │ 4.0026       │ He     │ 1895       │ -272  │ -269  │ 3      │ -457.6  │
│ 3   │ 6      │ Carbon   │ 12.0107      │ C      │ missing    │ 3500  │ 4827  │ 1327   │ 6332.0  │
│ 4   │ 8      │ Oxygen   │ 15.9994      │ O      │ 1774       │ -218  │ -183  │ 35     │ -360.4  │
│ 5   │ 26     │ Iron     │ 55.845       │ Fe     │ missing    │ 1535  │ 2750  │ 1215   │ 2795.0  │
│ 6   │ 29     │ Copper   │ 63.546       │ Cu     │ missing    │ 1083  │ 2567  │ 1484   │ 1981.4  │
```

To find matching entries:

```
julia> ptable[occursin.("Copper", ptable.Name), :]
```

```
1×9 DataFrame
│ Row │ Number │ Name   │ AtomicWeight │ Symbol │ Discovered │ Melt  │ Boil  │ Liquid │ MP_in_F │
│     │ Int64  │ String │ Float64      │ String │ Int64?     │ Int64 │ Int64 │ Int64  │ Float64 │
├─────┼────────┼────────┼──────────────┼────────┼────────────┼───────┼───────┼────────┼─────────┤
│ 1   │ 29     │ Copper │ 63.546       │ Cu     │ missing    │ 1083  │ 2567  │ 1484   │ 1981.4  │
```

```
julia> ptable[occursin.(r"C.*", ptable.Name), :]
```

```
2×9 DataFrame
│ Row │ Number │ Name   │ AtomicWeight │ Symbol │ Discovered │ Melt  │ Boil  │ Liquid │ MP_in_F │
│     │ Int64  │ String │ Float64      │ String │ Int64?     │ Int64 │ Int64 │ Int64  │ Float64 │
├─────┼────────┼────────┼──────────────┼────────┼────────────┼───────┼───────┼────────┼─────────┤
│ 1   │ 6      │ Carbon │ 12.0107      │ C      │ missing    │ 3500  │ 4827  │ 1327   │ 6332.0  │
│ 2   │ 29     │ Copper │ 63.546       │ Cu     │ missing    │ 1083  │ 2567  │ 1484   │ 1981.4  │
```

or:

```
julia> ptable[occursin.(r"Co", ptable.Name), :]
```

```
1×9 DataFrame
│ Row │ Number │ Name   │ AtomicWeight │ Symbol │ Discovered │ Melt  │ Boil  │ Liquid │ MP_in_F │
│     │ Int64  │ String │ Float64      │ String │ Int64?     │ Int64 │ Int64 │ Int64  │ Float64 │
├─────┼────────┼────────┼──────────────┼────────┼────────────┼───────┼───────┼────────┼─────────┤
│ 1   │ 29     │ Copper │ 63.546       │ Cu     │ missing    │ 1083  │ 2567  │ 1484   │ 1981.4  │
```

## Subsets and groups

To investigate subsets and groupings, let's recreate the dataframe:

```
julia> ptable = DataFrame(
          Number       = [1,  2,  6,  8,  26,     29,     ],
          Name         = ["Hydrogen",    "Helium",   "Carbon",   "Oxygen",   "Iron",     "Copper",  ],
```

```
        AtomicWeight = [1.0079,     4.0026,     12.0107,    15.9994,    55.845,     63.546,     ],
        Symbol       = ["H",    "He",    "C",    "O",    "Fe",    "Cu",  ],
        Discovered   = [1776,   1895,   0,  1774,   0,  missing,    ],
        Melt         = [-259,   -272,   3500,   -218,   1535,   1083,   ],
        Boil         = [-253,   -269,   4827,   -183,   2750,   2567,   ],
        Liquid       = [6,  3,  1327,   35,     1215,   1484,   ],
    )
```

and add a another column:

```
julia> ptable[!, :Room] = [:Gas, :Gas, :Solid, :Gas, :Solid, :Solid]
```

```
 6-element Array{Symbol,1}:
  :Gas
  :Gas
  :Solid
  :Gas
  :Solid
  :Solid
```

This column gives the state of each element at room temperature.

It's now possible to collect up and group the elements according to their state. The `groupby()` function splits the original DataFrame into GroupedDataFrames according to the values in the named column. For example, with three elements that are gases at room temperature, and the others which are solid, we can obtain two GroupedDataFrames:

```
julia> gd = groupby(ptable, [:Room])
```

```
GroupedDataFrame with 2 groups based on key: Room
First Group (3 rows): Room = :Gas
│ Row │ Number │ Name     │ AtomicWeight │ Symbol │ Discovered │ Melt  │ Boil  │ Liquid │ MP_in_F │ Room   │
│     │ Int64  │ String   │ Float64      │ String │ Int64?     │ Int64 │ Int64 │ Int64  │ Float64 │ Symbol │
├─────┼────────┼──────────┼──────────────┼────────┼────────────┼───────┼───────┼────────┼─────────┼────────┤
│ 1   │ 1      │ Hydrogen │ 1.0079       │ H      │ 1776       │ -259  │ -253  │ 6      │ -434.2  │ Gas    │
│ 2   │ 2      │ Helium   │ 4.0026       │ He     │ 1895       │ -272  │ -269  │ 3      │ -457.6  │ Gas    │
│ 3   │ 8      │ Oxygen   │ 15.9994      │ O      │ 1774       │ -218  │ -183  │ 35     │ -360.4  │ Gas    │
⋮
Last Group (3 rows): Room = :Solid
│ Row │ Number │ Name   │ AtomicWeight │ Symbol │ Discovered │ Melt  │ Boil  │ Liquid │ MP_in_F │ Room   │
│     │ Int64  │ String │ Float64      │ String │ Int64?     │ Int64 │ Int64 │ Int64  │ Float64 │ Symbol │
├─────┼────────┼────────┼──────────────┼────────┼────────────┼───────┼───────┼────────┼─────────┼────────┤
│ 1   │ 6      │ Carbon │ 12.0107      │ C      │ missing    │ 3500  │ 4827  │ 1327   │ 6332.0  │ Solid  │
│ 2   │ 26     │ Iron   │ 55.845       │ Fe     │ missing    │ 1535  │ 2750  │ 1215   │ 2795.0  │ Solid  │
│ 3   │ 29     │ Copper │ 63.546       │ Cu     │ missing    │ 1083  │ 2567  │ 1484   │ 1981.4  │ Solid  │
```

We're saving the grouped data frame in `gd`.

The `combine()` function lets you group the rows and then apply a function to one of the fields of every row in the group. In this next example, we find the mean melting point of all the gases, and the mean melting point of all the solid elements, and we're using the grouped data frame again:

```
julia>  combine(gd, [:Melt] => mean)
```

```
2×2 DataFrame
│ Row │ Room   │ Melt_mean │
│     │ Symbol │ Float64   │
├─────┼────────┼───────────┤
│ 1   │ Gas    │ -249.667  │
│ 2   │ Solid  │ 2039.33   │
```

## Sorting

The `sort!()` function works with DataFrames as well. You supply the columns on which to sort, using the following syntax:

```
julia> sort!(ptable, [order(:Room), order(:AtomicWeight)])
```

```
6×10 DataFrame
│ Row │ Number │ Name     │ AtomicWeight │ Symbol │ Discovered │ Melt  │ Boil  │ Liquid │ MP_in_F │ Room   │
│     │ Int64  │ String   │ Float64      │ String │ Int64?     │ Int64 │ Int64 │ Int64  │ Float64 │ Symbol │
├─────┼────────┼──────────┼──────────────┼────────┼────────────┼───────┼───────┼────────┼─────────┼────────┤
│ 1   │ 1      │ Hydrogen │ 1.0079       │ H      │ 1776       │ -259  │ -253  │ 6      │ -434.2  │ Gas    │
│ 2   │ 2      │ Helium   │ 4.0026       │ He     │ 1895       │ -272  │ -269  │ 3      │ -457.6  │ Gas    │
│ 3   │ 8      │ Oxygen   │ 15.9994      │ O      │ 1774       │ -218  │ -183  │ 35     │ -360.4  │ Gas    │
│ 4   │ 6      │ Carbon   │ 12.0107      │ C      │ missing    │ 3500  │ 4827  │ 1327   │ 6332.0  │ Solid  │
│ 5   │ 26     │ Iron     │ 55.845       │ Fe     │ missing    │ 1535  │ 2750  │ 1215   │ 2795.0  │ Solid  │
│ 6   │ 29     │ Copper   │ 63.546       │ Cu     │ missing    │ 1083  │ 2567  │ 1484   │ 1981.4  │ Solid  │
```

The resulting DataFrame is sorted first by its state at room temperature (so Gas before Solid), then by its Atomic Weight (so Iron before Copper).