Using Julia for Introductory Statistics

John Verzani

2023-01-11T16:56:10-05:00

Table of contents

Chapter 1

Julia introduction

This is a collection of notes for using `Julia` for Introductory Statistics.

Why? No compelling reason save I had done something similar for `R` when was a fledgling `S-Plus` clone. No more, `R` is a juggernaut, and it is almost certain `Julia` will never replace `R` for this statistics. `Julia` users can already interface with `R` quite easily through `RCall`. *However*, there are some reasons that `Julia` could be a useful language when learning basic inferential statistics, especially if other real strengths of the `Julia` ecosystem were needed. So these notes show how `Julia` can be used for these tasks, and also shows that it works pretty well.

There are some great books published about using `Julia` with data science, within which much of this material is covered. For example, [@JuliaForDataAnalysis] is a quite thorough treatmeant, [@storopoli-huijzeralonso2021juliadatascience] is very well done, [@nazarathy2021statisticsjulia] covers topics here (cf. JuliaCon Workshop. View this as an alternative.

Julia is an open-source programming language suitable for many tasks, like scientific programming. It is designed for high performance – Julia programs compile on the fly to efficient native code. Julia has a relatively easy to learn syntax for many tasks, certainly no harder to pick up than `R` and `Python`, widely used scripting languages for the tasks illustrated herein.

> **Contribute**
>
> These notes are a work in progress. Feel free to click the "edit this page" button or report an issue.

1.1 Installing and running Julia

Julia can be downloaded from julialang.org. The language is evolving rapidly. The latest release should be used. These notes should work with any version since `v"1.6.0"`, though post `v"1.9.0"` there are significant speedups with external packages that make the user experience even better.

Once downloaded and installed the `Julia` installation will provide a *command line* for interactive usage and a binary to run scripts. It is envisioned most users will use an alternative interface, though `Julia` has

an excellent REPL for command-line usages.

Some alternatives to the REPL for interacting with Julia are:

- IJulia: This is a means to use the Jupyter interactive environment to interact with Julia through notebooks. It is made available by installing the package IJulia (details on package installation follow below). This relies on Julia's seamless interaction with Python and leverages many technologies developed for that langauge.
- Pluto: The Pluto environment provides a notebook interface for Julia written in Julia leveraging many JavaScript technologies for the browser. It has the feature of being reactive, making it well suited for many exploratory tasks and pedagogical demonstrations.
- Visual Studio Code: Julia is a supported language for the Visual Studio Code editor of Microsoft, a programmer's IDE.

These notes use quarto to organize the mix of text, code, and graphics. The quarto publishing system is developed by Posit, the developers of the wildly sucessful RStudio interface for R. The code snippets are run as blocks (within IJulia) and the last command executed is shown. (If code is copy-and-pasted into the REPL, each line's output will be displayed.) The code display occurs below the cell, as here, where we show that Julia can handle basic addition:

```
2 + 2
```

```
4
```

1.2 Overview of some basics

This section gives a quick orientation for using Julia. See the collection of tutorials for more comprehensive introductions.

As will be seen Julia use *multiple dispatch* (as does R) where different function methods can be called using the same generic name. Different methods are dispatched depending on the type and number of the arguments. The + sign above, is actually a function call to the + function, which in base Julia has over 200 different methods, as there are many different implementations for addition.

Julia is a *dynamically typed* language, like R and Python, meaning variables can be reassigned to different values and with different types.[1] Dynamicness makes interactive usage at the REPL or through a notebook much easier.

> **Interactive help**
>
> Interacting with Julia primarily involves variables and functions. Most all functions have documentation, which can be called up by prefacing the function name with an immediate question mark, as in ?sqrt to see the documentation for sqrt. More than one method may be documented. A call like ?sqrt(9) will limit the help to the method called by sqrt(9) (the square root function for integers.)

Julia supports the usual mathematical operations familiar to users of a calculator, such as +, -, *, /, and ^. In addition, there a numerous built in functions such as mathematical ones like sqrt or programming

[1] With the one caveat that generic function names can not be reassigned as variables or vice versa.

oriented ones, like `map`.

These functions are called with arguments which may be *positional* (0, 1, or more positional arguments) or specified by *keywords*. Multiple dispatch considers the positions and types of arguments a function is called with.

Values in `Julia` have types. A particular instance will have a concrete type but *abstract* types help to organize code bases and participate in dispatch. Values can be assigned to variable names, or bindings. The ability to simply create new user-defined types makes generic programming quite accessible and `Julia` code very *composable*.

This simple example, taking the average of several numbers, shows most of this:

```
xs = [1, 2, 3, 7, 9]
sum(xs) / length(xs)
```

```
4.4
```

The first line *assigns* to a variable, `xs`, a value that is a *vector* of numbers, integers of type `Int53` in this case. For this illustration, a vector is a container of different numbers. The second line calls three functions: `sum` to add the elements in the vector; `length` to count the number of elements in the vector; and `/` to divide these two quantities. All of these functions are *generic*, with different methods for different types of argument(s). The same pattern would work for different container types:

```
xs = (1, 2, 3, 7, 9) # tuple
sum(xs) / length(xs)
```

```
4.4
```

1.3 Add-on packages

Base `Julia` provides a very useful programming environment which can be extended through *packages*. Some packages are provided by base `Julia`, such as `Dates`, others are external add-on packages, such as `IJulia` mentioned previously. Julia has one key package, `Pkg`, to manage the installation. By default, the installation of a single package will download all dependent packages. On installation, packages are partially compiled. This speeds up the *loading* of a package when it is used within a session.

Packages need be installed just once, but must be loaded each session. Loading a package is done by a command like `using Statistics`, which will load the built in `Statistics` package. At the REPL, calling `using PKGNAME` on an uninstalled package will lead to a prompt to install the package. For other interfaces, packages may need to be installed through the `Pkg` package, loaded through `using Pkg`.

When a package is loaded its exported functions are made available to use directly. Non-exported functions can be accessed by *qualifying* the function with the name of a *module* (usually the name of the package). For example, we will see the command `CSV.read` which calls the `read` function provided in the `CSV` package.

Most packages are set up to *extend* generic functions that may be defined elsewhere. Not all. When there are conflicts, they can be resolved by either just *importing* the packages and qualifying all uses, or qualifying the uses that conflict.

These notes will utilize numerous add-on packages including: `StatsBase`, to extend the built-in `Statistics` package; `StatsPlots`, for easier to make plots; `AlgebraOfGraphics` and `GLMakie`, for more advanced statistical graphics; `CSV`, `DataFrames` for working with tabular data; `RDatasets`, for some handy datasets; `FreqTables` and `CategoricalArrays`, for some needed functionality; `Distributions`, for probability distributions; `HypothesisTests`, for the computation of significance tests and confidence intervals; and `GLM`, `Loess`, and `RobustModels`, for statistical modeling.

Part I

Exploratory Data Analysis

Chapter 2

Univariate data

Statistics involves the study of data. A data set can be a single value, or *scalar*, but more commonly is a collection of values. This chapter covers the basic containers used in `Julia` to store and manipulate data sets for a single variable in statistics. In addition it shows some basic summaries of a single variable.

2.1 Data vectors

`Julia` offers several containers for storing ordered data, such as in a data set $x_1, x_2, \ldots, x_n$.

We consider this data on the number of whale beachings in a certain area by year during a decade:

```
74 122 235 111 292 111 211 133 156 79
```

As the data in a data set is typically of the same type (integer here, or character, number, ...) and may be quite large a vector is a natural choice for storage in `Julia`.

Vectors are created using square brackets with entries separated by commas. Storing this in a vector and assigning to a variable, `whale`, is then done with:

```
whale = [74, 122, 235, 111, 292, 111, 211, 133, 156, 79]
```

```
10-element Vector{Int64}:
  74
 122
 235
 111
 292
 111
 211
 133
 156
  79
```

Without using commas, a matrix is created:

```
whale_matrix = [74 122 235 111 292 111 211 133 156 79]
```

```
1×10 Matrix{Int64}:
 74  122  235  111  292  111  211  133  156  79
```

> **Space saving measure**
>
> For purposes of printing only, we use the trailing `|> permutedims` below to to print a *column* vector, as a *row* vector in the following, as with:
>
> ```
> whale |> permutedims
> ```
>
> ```
> 1×10 Matrix{Int64}:
>  74  122  235  111  292  111  211  133  156  79
> ```
>
> This uses the chaining operation to call the `permutedims` function on the vector.

Internally a vector is a 1-dimensional array, a matrix a 2 dimensional array. `Julia` has a general N dimensional array type that these are specializations of. For a matrix, one uses space to separate row elements and semicolons to separate rows. Vectors are useful for storing data; matrices can be used to store data, but are widely used to represent mathematical values, as they have an associated algebra that can compactly represent many operations; arrays are not used in our discussion.

As seen in the output above, `Julia` prints the size and *element type* of a vector. In this case, a vector of integers (`Int64`). The constructor (`[]`) will promote values to a common type, which may be `Any`, a catch all type.

Vectors have a length, found by `length`, which in this case is the number of data points, often labeled n. More generally, arrays have a `size`:

```
length(whale), size(whale)
```

```
(10, (10,))
```

The size is returned as a `tuple`, in this case with just 1 element, as vectors are 1 dimensional.

The `length` function is a reduction, returning a summary of a vector. Similarly, `sum` will add the elements, returning a number. Here we see how to compute the mean:

```
sum(whale) / length(whale)
```

```
152.4
```

The `mean` function is not part of base `Julia`. It can be found in the `Statistics` package, which comes bundled with `Julia`, but we prefer to use the `StatsBase` package:

```
using StatsBase
mean(whale)
```

```
152.4
```

2.1.1 Missing values

`Julia` uses `missing` to represent missing values. For example, data on the cost of a hip replacement at various hospitals was found from their websites and is given by:

```
10500, 45000, 74100, unavailable, 83500,
86000, 38200, unavailable, 44300, 12500,
55700, 43900, 71900, unavailable, 62000
```

When the cost could not be identified, it was labeled "unavailable." We replace the colloquial "unavailable" with the special value `missing` and enter the values into a vector:

```
hip_cost = [10500, 45000, 74100, missing, 83500, 86000, 38200, missing,
            44300, 12500, 55700, 43900, 71900, missing, 62000]
hip_cost |> permutedims
```

```
1×15 reshape(::Vector{Union{Missing, Int64}}, 1, 15) with eltype Union{Missing, Int64}:
 10500  45000  74100  missing  83500  …  55700  43900  71900  missing  62000
```

We highlight that the type of element in `hip_cost` is `Union{Missing, Int64}`, a type that allows an element to be either an integer or a `missing` element.

The `missing` value propagates through computations:

```
sum(hip_cost)
```

```
missing
```

In particular, all of these basic combinations with `missing` yield `missing`, as they should – if data is not available, combinations based on that data are still not available:

```
1 + missing, 1 - missing, 1*missing, 1/missing, missing^2, missing == true
```

```
(missing, missing, missing, missing, missing, missing)
```

> **Also nothing and something**
>
> In `Julia` there is also `nothing`. The semantics are a bit different, and `missing` is the designed choice for data. (The `nothing` value is useful for general programming purposes.) For floating point values, there is also `NaN`, or not a number. This too ios often used, as a sentinel value, to indicate missingness, though `missing` is suggested. The `something` function is used to skip over its arguments until a non-`nothing` value is found.

To work with missing data, it can be removed or skipped over. The `skipmissing` function provides a wrapper around an object which when iterated over, will skip the missing values. For example:

```
sum(skipmissing(hip_cost))
```

```
627600
```

The `sum` function iterates over the values in the container it is passed to reduce them to a value, `mean` is similar and works with `skipmissing` in an identical manner:

```
mean(skipmissing(hip_cost))
```

```
52300.0
```

(Which is helpful, as `length`, used above to compute a mean, does not work with `skipmissing`.)

2.1.2 Names (named tuples)

It may be natural to assign the year of measurement to the values in `whale`, but vectors, as defined in base `Julia`, do not allow names as an attribute. (There are external packages that allow this, e.g. `NamedArrays`, which arise in our discussion of contingency tables.) If names are important, `Julia` provides a named tuple type, which builds on the basic tuple type.

While vectors are collections of *homogeneous* values, tuples are collections of *heterogeneous* values. Tuples are constructed with commas, and typically parentheses to delimit the comma[1]:

```
whale = (74, 122, 235, 111, 292, 111, 211, 133, 156, 79)
```

```
(74, 122, 235, 111, 292, 111, 211, 133, 156, 79)
```

One-element tuples are distinguished by using a trailing command and parentheses:

```
a_lone_whale = (101,)
```

```
(101,)
```

For the many purposes, tuples can be exchanged for vectors, as both are iterable[2]:

```
sum(whale), mean(whale), length(whale)
```

```
(1524, 152.4, 10)
```

Unlike vectors, but like numbers, tuples can not be *modified* after construction. This allows tuples to be quite useful – and performant – for programming purposes.

[1] The parentheses for constructing tuples with 2 or more elements are technically optional, a fact that is useful for bundling different outputs into one

[2] The output of the following command is displayed as a tuple, as the commas used create the tuple, even if not enclosed in parentheses.

Tuples can also have names. The basic construction uses "key=value" pairs:

```
test_scores = (Alice = 87, Bob = 72, Shirley = 99)
```

```
(Alice = 87, Bob = 72, Shirley = 99)
```

The names are not quoted, and are stored internally as a tuple of symbols. Extra effort is necessary to create names with spaces or number. In the following we show that `var"..."` is useful to create more complicated symbols[3]:

```
children = (var"X Æ A-Xii" = 1,
            var"Vivian Jenna Wilson" = 2,
            var"Exa Dark Sideræl" = 3)
```

```
(var"X Æ A-Xii" = 1, var"Vivian Jenna Wilson" = 2, var"Exa Dark Sideræl" = 3)
```

The above also shows that Unicode values are easily used within `Julia` in strings or as identifiers.

Named tuples can also have their "names" attached via parsing, through a syntax similar to keyword arguments of functions, as identifiers imply names:

```
a = [1,2]; b = [3,4]
(; a, b)  # same as (a=a, b=b)
```

```
(a = [1, 2], b = [3, 4])
```

This gives several different ways to construct 1-element named tuples, where something is done to disambiguate the use of enclosing parentheses:

```
(; a), (; a=a), (; :a => a), (a=a, )
```

```
((a = [1, 2],), (a = [1, 2],), (a = [1, 2],), (a = [1, 2],))
```

Named tuples can have their values accessed by name using `getproperty`, which has a `.` for convenient syntax:

```
test_scores.Alice
```

```
87
```

The generic function `keys` will return the names ("keys" are used to look up a value) and `values` will extract the values.

Named tuples being both named and heterogeneous are a natural container for collecting data on several different variables for a single case, as will be seen in the discussion on tabular data.

[3] The `var` implementation is via a *string macro*, and is documented through `@var_str`.

2.1.2.1 Associative arrays

Abstractly, an *associative array* is a container for storing (key,value) pairs. `Julia` has the notation `key => value` for representing a *pair*. A named tuple is an immutables associative array where the keys are symbols. A *dictionary* is a more general *mutable* type with the keys being arbitrary (numbers, strings, symbols, ...). There are various implementations, the `Dict` constructor the most commonly used. As with a named tuple, the keys are returned by `keys`, the values by `values`, and pairs by `pairs`.

2.1.3 Indexing

The elements of a vector, like a data set, are indexed. For basic vectors Julia uses one-based indexing[4]. The `whale` data, as defined, has 10 elements, we can get the third, fourth, and sixth with:

```
whale = [74, 122, 235, 111, 292, 111, 211, 133, 156, 79]
whale[3], whale[4], whale[6]
```

```
(235, 111, 111)
```

The above uses three function calls, and displays as a tuple does. (Since the commas produce a tuple.)

To retrieve these same values in a single call, we can pass a vector of indices:

```
whale[ [3,4,6] ]
```

```
3-element Vector{Int64}:
 235
 111
 111
```

The `end` keyword refers to the last index of a collection. Similarly, `begin` refers to the first index inside the square brackets.

```
whale[end], whale[begin]
```

```
(79, 74)
```

These values allow simple arithmetic. Here we see how to trim off the first and last through indexing[5]:

```
whale[begin+1:end-1] |> permutedims
```

```
1×8 Matrix{Int64}:
 122  235  111  292  111  211  133  156
```

The colon, `:`, refers to all indices in a vector; `whale[:]` will return a copy of the data in `whale`.

[4] In `Julia` there are options to have offsets for indexing, e.g. `OffsetArrays` that allow for other access patterns, such as would be useful for zero-based indexing.

[5] This example uses the `colon` operator to produce a range of values between it two arguments with increment of 1.

> **Indexing with a container of values**
>
> The range `1:1` specifies the value `1`, as does just `1`, but for indexing the two are different:
>
> ```
> whale[1], whale[1:1]
> ```
>
> ```
> (74, [74])
> ```
>
> The idiom is indexing by a scalar – like `1` – can drop dimensions (the vector becomes a scalar), whereas indexing by a container – like `1:1` or, say, `[1]` – does not drop dimensions. The documentation for indexing of an array has: "If all the indices are scalars, then the result, `X`, is a single element from the array, `A`. Otherwise, `X` is an array with the same number of dimensions as the sum of the dimensionalities of all the indices."

2.1.3.1 Views

The extraction of values from a vector, as above, necessitates the allocation of memory to store the copy of the values. When the data set is large, or is accessed many times, these allocations may be avoided using a "view" of the data. Views create a lazy reference to the underlying data in the array. The `view` function takes the object as its first argument, and indices for its remaining arguments[6]:

```
view(whale, [3,4,6])
```

```
3-element view(::Vector{Int64}, [3, 4, 6]) with eltype Int64:
 235
 111
 111
```

The `end` and `begin` keywords do not work with `view`[7], though the `lastindex` and `firstindex` functions do:

```
view(whale, firstindex(whale)+1:lastindex(whale)-1) |> permutedims
```

```
1×8 reshape(view(::Vector{Int64}, 2:9), 1, 8) with eltype Int64:
 122  235  111  292  111  211  133  156
```

2.1.4 Assignment

The assignment

```
whale = [74, 122, 235, 111, 292, 111, 211, 133, 156, 79]
whale |> permutedims
```

```
1×10 Matrix{Int64}:
 74  122  235  111  292  111  211  133  156  79
```

[6] Arrays can be n dimensional, so there may be 1 or more index to specify

[7] Though `:` does, as it is a function, not a keyword.

binds the name `whale` to the *container* holding the 10 numbers.

If we make another assignment, as in

```
whale_copy = whale
whale_copy  |> permutedims
```

```
1×10 Matrix{Int64}:
 74  122  235  111  292  111  211  133  156  79
```

the container is copied, but – unlike if we had used `copy(whale)` – the two variables point to the same container. When a vector is passed into a function, the function works with the container, not a copy.

The values in the container may be reassigned, which is done by using the indexing notation on the left-hand side of the equals sign:

```
whale[1] = 75
```

```
75
```

The value `75` is returned, as the *right-hand* side of an assignment is always the returned value, but we can see that `whale` was modified:

```
whale[1], whale_copy[1]
```

```
(75, 75)
```

We also see that as `whale_copy` points to the same container, it too was modified.

> **Mutation**
>
> When a vector is passed to a function, if there is no copy made (as opposed to a simple naming), then changes to the vector in the function will effect the original vector as the container used in the body of the function isn't changed. Functions which *modify* an argument that is passed to them (conventionally the first one) are *usually* named with a trailing `!`.

Multiple values can be assigned at once. For example, if the data was mis-arranged chronologically, we might have:

```
whale[[1,2,3]] = [235, 74, 122]
whale |> permutedims
```

```
1×10 Matrix{Int64}:
 235  74  122  111  292  111  211  133  156  79
```

The above modified the original container, so these changes would also be reflected in `whale_copy`.

Julia does not recycle by default, but *broadcasted assignment* (cf. `?.=`) can produce a similar behavior. Broadcasted assignment expands the right hand side and then does in-place assignment. For example, if we wanted to use a sentinel value to indicate unknown data for the first 3 values, we might have:

```
whale[[1,2,3]] .= 999
whale |> permutedims
```

```
1×10 Matrix{Int64}:
 999  999  999  111  292  111  211  133  156  79
```

Notice, without the dot an error will be thrown

```
whale[[4,5,6]] = 999  # errors
```

```
LoadError: ArgumentError: indexed assignment with a single value to possibly many locations is not supported; pe
```

The "recycling" above uses the left-hand side to identify the size needed. Broadcasted assignment also works with the entire collection. For example, the following command replaces the current data with the original data:

```
whale .= [74, 122, 235, 111, 292, 111, 211, 133, 156, 79]
whale |> permutedims
```

```
1×10 Matrix{Int64}:
 74  122  235  111  292  111  211  133  156  79
```

But `whale` is not a new object – as it would be without that dot – but rather, these values are placed into the container `whale` already refers to – which also is the container `whale_copy` points at:

```
whale_copy  |> permutedims
```

```
1×10 Matrix{Int64}:
 74  122  235  111  292  111  211  133  156  79
```

This assignment avoids the needed allocation of more memory.

The use of `[:]` on the *left-hand side* also does in-place assignment[8]. So unlike `whale = [...]` which *replaces* the container `whale` points at, this command reuses the container:

```
whale[:] = [74, 122, 235, 111, 292, 111, 211, 133, 156, 79]  |> permutedims
```

```
1×10 Matrix{Int64}:
 74  122  235  111  292  111  211  133  156  79
```

[8] The `:` *copies* on the right-hand side, but does in-place assignment on the left-hand side

Table 2.1: Various uses of indexing by numeric indices.

Index style	Explanation
`x[1]`	The first element of `x`.
`x[:]`	Copy of all elements of `x`.
`x[end]`	The last element of `x`.
`x[first]`	The first element of `x`.
`x[[2,3]]`	The second and third elements of `x`.
`typeof(x)[]`	0-length vector of same type as `x`.
`eltype(x)`	Element type of container `x`.
`x[1] .= 5`	Assign a value of 5 to first element of `x`.
`x[[2,3]] .= 4`	Broadcasted assignment to second and third elements
`x[[2,3]] = [4,5]`	Assign values to second and third elements of `x`.
`x[:] = [1, 2, 3]`	In-place assignment. Size *and type* of right-hand side must match left-hand side

2.1.4.1 Vector size

A vector has a length which can not be changed during assignment. Attempting to assign to an index beyond the size will result in a `BoundsError`. To extend the size of a vector we can use the following generic functions:

- `push!(v, x)`: extend the vector `v` pushing `x` to the *last* value
- `pushfirst!(v, x)`: extend the vector `v` pushing `x` to the *first* value
- `append!(v, v1)`: append the entries in `v1` to the end of `v`
- `vcat(v, v1)`: **v**ertically con**cat**enate `v` and `v1`. (Unlike `append!` this returns a new vector, so the type of the output will be recomputed).

The vector can also be shrunk. These generic functions for containers are useful:

- `pop!(v)`: remove *last* element from a vector; return element
- `popfirst!(v)`: remove *first* element from a vector; return element
- `deleteat!(v, i)`: remove *i*the element from a vector; returns vector
- `empty!(v)`: remove *all* elements from a vector

Again, the trailing `!` in a function name is a convention to indicate to the user that the function will mutate its arguments, conventionally its first one. That is, a function like `pop!(v)` does two things: it returns the last element and *also* shortens the vector `v` that is passed in.

2.1.4.2 Vectors have an element type

The assignment `whale = [74, 122, 235, ...]` assigns a container of a *specific* type to the variable `whale`. The `eltype(whale)` command will return this element type. Subsequent assignments to this container must contain values that can be *promoted* to that type, otherwise an error will be thrown.

For example, we can't assign a fractional number of whales:

```
whale[1] = 74.5
```

```
LoadError: InexactError: Int64(74.5)
```

An `InexactError` is thrown because, `whale` is a vector of integers, and `74.5` can't be automatically promoted to an integer (as `74.0` could be).

A similar thing happens if we attempt to assign `missing` to a value, as in `whale[1] = missing`. In this case, a `MethodError` is thrown, which comes from the attempt to "`convert`" `missing` into the underlying integer type.

The way to fix this is to create a new container that allows a wider type. For `Float64` values that can be achieved in the cumbersome manner of converting all the values to the wider type:

```
whale = convert(Vector{Float64}, whale)
whale[1] = 74.5
```

```
74.5
```

This assigns `whale` to a *new* container which accepts floating point values, and then reassigns the first one.

Though cumbersome, this is not typical usage, as the constructor used to create the data set will promote to a common type, so it would only matter when adjusting the initial values.

For the special case of assigning a *missing* value, the `allowmissing` function from the `DataFrames` package[9] creates a vector with a type that allows – as well – missing values[10]. Again, re-assignment is necessary:

```
using DataFrames
whale = allowmissing(whale)
whale[1] = missing
```

```
missing
```

2.1.4.3 Broadcasting

As seen, the functions `length` and `sum` are reductions, returning a single number from a vector of numbers. To compute a *sample standard deviation*, say, we follow the formula:

$$s = \sqrt{\frac{\sum_i (x_i - \bar{x})^2}{n-1}}.$$

To do so we would need to:

- Subtract a scalar value from from each element of the data vector: $(x_i - \bar{x})$.
- Apply the squaring function to each element of this difference.
- Reduce the resulting data to a number through `sum`.

[9] With data frames, the `allowmissing!` function is used, as well.

[10] The `allowmissing` function creates a union type with `Missing` in addition to the original data type. It some printouts, a `?` is appended to the type name to indicate this addition.

Embarking on this with a simple attempt: `whale - mean(whale)` will fail.

The subtraction of a scalar value from a vector value is not defined, as `Julia` is not implicitly vectorized. Rather the user must be explicit. For this, the concept of broadcasting is useful. Broadcasting will expand the scalar to match the size of the vector and then use vector subtraction to find the result. Broadcasting is done simply by adding a "." (the dot) to the function. For infix operations like - this is *before* the operator:

```
whale = [74, 122, 235, 111, 292, 111, 211, 133, 156, 79]
whale .- mean(whale)  |> permutedims
```

```
1×10 Matrix{Float64}:
 -78.4  -30.4  82.6  -41.4  139.6  -41.4  58.6  -19.4  3.6  -73.4
```

We also would need to square these values. This could *also* be done by broadcasting `^`, as in:

```
(whale .- mean(whale)).^2  |> permutedims
```

```
1×10 Matrix{Float64}:
 6146.56  924.16  6822.76  1713.96  …  3433.96  376.36  12.96  5387.56
```

To avoid having to use *too* many dots, it is typical to define a function for doing a scalar computation and broadcast that[11]. An example will wait, but to illustrate the syntax, we can broadcast the `sqrt` function using a "." after the function name and before the opening parentheses:

```
sqrt.(whale)  |> permutedims
```

```
1×10 Matrix{Float64}:
 8.60233  11.0454  15.3297  10.5357  …  14.5258  11.5326  12.49  8.88819
```

Broadcasting works with functions of multiple arguments and multiple shapes.

The use of multiple shapes allows scalars and vectors to be broadcast over and is used in `whale .- mean(whale)`. But vector and *row* vectors can also be broadcast over, So `[1,2] .+ [3,4]` does vector addition, `[1,2] .+ [3 4]` pads out the vector to a matrix, the row vector to a matrix, then adds:

```
(v=[1,2], rv=[3 4], va = [1,2] .+ [3,4], ma = [1, 2] .+ [3 4], ck=[1 1; 2 2] + [3 4; 3 4])
```

```
(v = [1, 2], rv = [3 4], va = [4, 6], ma = [4 5; 5 6], ck = [4 5; 5 6])
```

This behavior may not be desirable. Some objects broadcast as scalars, others as containers. But there may be times where broadcasting as a container may be incorrect. To force a value to broadcast like a scalar, the value can be wrapped in `Ref`. That is `mean(whale)` is the same as `mean.(Ref(whale))` *but* `mean.(whale)` would broadcast `mean` over each element in `whale`. As `mean` for a single number is just that number, `mean.(whale)` would just be the same container.

[11] Also the `@.` *macro* can automatically broadcast all operations.

2.2 Data types

An old taxonomy of levels of measurement include nominal, ordinal, interval, and ratio; where nominal values have no rank, ordinal values have a rank, but no meaning is assigned to the different between values, interval data has a meaning between differences but 0 is arbitrary, and ration has a meaningful zero, such as most numeric data. As data can easily be coded (explicitly or behind the scenes) with numbers, this keeps the different types distinct. However, for use with the computer, in particular `Julia` here, we see it makes more sense to emphasize different aspects of the data related to the underlying type.

`Julia` has numerous data types. Some are "abstract" types, such as `Real` or `Integer`, others are "concrete", often indicating how the data is stored, such as `Float64` or `Int64`, with the "`64`" indicating 64 bits of memory. In `Julia`, these data types can be used to direct method dispatch. For statistics a few common types are used to represent data. These are reviewed in the following.

2.2.1 Numeric data types

The `Julia` parser readily identifies the following values of 1 and stores the value using a different type:

```
1, 1.0, 1//1, 1 + 0im, big(1), BigFloat(1)
```

```
(1, 1.0, 1//1, 1 + 0im, 1, 1.0)
```

Respectively, these represent integer, floating point, rational, complex, and two types of "big" numbers. `Julia` uses a promotion machinery when different types are mixed. For example, we have:

```
x = 1 + 1.0 + 1//1 + 1 + 0im + big(1) + BigFloat(1)
```

```
6.0 + 0.0im
```

The type of `x` must be both complex and be able to store the underlying numbers, which may be "big" numbers:

```
typeof(x)
```

```
Complex{BigFloat}
```

The above illustrates that addition of an integer and a floating point yields a floating point, and adding to a complex number returns a complex number, etc.

> **Type stability**
>
> `Julia` programmers try to make functions "type stable," if possible, as this generally leads to more performant code once compiled. A consequence is addition of numbers, like `1 + 1.0` will *always* be a floating point value, even if in the case of these particular *values* an integer could be the answer. That is the output type of `+` here is determined by the input *types*, not the input *values*.

For data consisting of counts, integers are typically used. If storage is an issue (e.g., lots of data, but not a lot of different values), different forms of integers which use less data may be used.

For data on measurements, with a continuous nature, floating point values are the natural choice. Floating point can represent most integer values exactly, and fractional and irrational values either exactly or approximately. Rational numbers can represent fractional data exactly, though it should be expected that operations with rational values are less performant than for floating point values.

Complex values in `Julia` are based on an underlying data type holding the two numbers in `a + bi`.

2.2.2 Categorical data types

There are different options available for the storage of categorical data.

2.2.2.1 Character data

The `String` type in `Julia` is the basic type for holding character data. Strings are created with matching single quotes *or* – for multiline strings – with matching triple quotes:

```
s = "The quick brown fox ..."
t = """
Four score and seven years ago
our fathers brought forth...
"""
```

```
"Four score and seven years ago\nour fathers brought forth...\n"
```

The string type in `Julia` can hold Unicode data, such as non-ASCII letters and even emojis.

Double quotes are used to create a string. Triple quotes can be used to create multi-line strings. Single quotes are for `Char` types. A character represents a Unicode code point. Strings are iterable, and iteration yields back `Char` values. The `collect` function iterates over an object and returns the values. Here we see the `Chars` in a string:

```
s = "Zoë"
collect(s)
```

```
3-element Vector{Char}:
 'Z': ASCII/Unicode U+005A (category Lu: Letter, uppercase)
 'o': ASCII/Unicode U+006F (category Ll: Letter, lowercase)
 'ë': Unicode U+00EB (category Ll: Letter, lowercase)
```

Strings are also indexable. When indexed by a single value, a `Char` type is returned, when indexed by a range of a values a string is returned. Be warned, indexing into non-ascii strings may error. Here is an example using a string from `Julia`'s manual:

```
j = "jμΛIα"
length(j), j[2], j[2:4]  # 2:4 represent the value 2,3,4
```

```
(5, 'μ', "μΛ")
```

The value returned by `j[2]` is a character, whereas that returned by `j[2:4]` is a string[12]. However, attempting to extract `j[3]` will be an error.

2.2.2.2 String operations

Strings can be combined many different ways.

The `*` operation is used to combine strings or chars. The basic usage is straight forward:

```
"a" * "bee" * "see"
```

```
"abeesee"
```

The `^` operation when used with an integer exponent will repeat the argument:[13]

```
"dot "^3
```

```
"dot dot dot "
```

More generally, `join` will combine an *iterable* of strings or characters. If a delimiter is specified, it will be inserted between values (with an option to indicate the last delimiter differently):

```
j = "jμΛIα"
join(collect(j), ","), join(collect(j), ", ", ", and ")
```

```
("j,μ,Λ,I,α", "j, μ, Λ, I, and α")
```

`Julia` makes string *interpolation* easy. Within a string, a value of a variable can be inserted using a dollar sign to reference the variable. This is more general, as computations can also be inserted. Below parentheses are used to delimit the interpolated command:

```
x = "Alice"
"$x knows that 2 + 2 is equal to $(2+2)"
```

```
"Alice knows that 2 + 2 is equal to 4"
```

(To include a dollar sign in the string, it can be escape with a leading slash, "\", or a `raw` string can be used, as in `raw"$(2+2) is not evaluated"`.)

For *formatted* values, such as needed when printing floating point values, the built-in `Printf` package provides support. For more performant solutions, an `IOBuffer` can be useful[14].

A common means to generate strings is from reading in delimited files, such as comma-separated files. These may produce strings with leading or trailing spaces. To strip these off, `Julia` offers `strip`, `lstrip`,

[12] The colon operator `a:b` returns an iterator to produce "a, a+1, ..., b".
[13] The expression `s^i` just calls `repeat(s, i)`, for these types.
[14] These are illustrated in the `stemplot` function defined later.

and `rstrip`. The `lstrip` function strops the left side, `rstrip` the right side, and `strip` combines the two. An option to pass in other characters to strip besides spaces is available.

```
strip("   abc    "),  strip("...abc...", '.')
```

```
("abc", "abc")
```

Discussing data sets for different type of data adds no more complication as vectors in `Julia` are typed so a data set of character data might simply be stored as a vector of string data, such as:

```
job_title = ["Data Scientist", "Machine Learning Scientist", "Big Data Engineer"]
```

```
3-element Vector{String}:
 "Data Scientist"
 "Machine Learning Scientist"
 "Big Data Engineer"
```

2.2.2.3 Symbols

`Julia` as a language can be used to represent the language's code as a data structure in the language. Symbols are needed to refer to the name, or identifier, of a variable as opposed to the values in the variable. Symbols, being part of the language, are used for other purposes, such as keys for a named tuple. The access pattern `nt.a` has been mentioned; this is a convenience for `getfield(nt, :a)`, the symbol being used as a key. When data frames are introduced – essentially a collection of matched data vectors – symbols can be used to reference the individual variables.

The simple constructor for a symbol is `:`, as in `:some_symbol`. The `Symbol` constructor can also be used to create symbols with spaces, e.g., `Symbol("some symbol")`; the string macro `var"..."` is a convenience.

The `string` function (or `String` constructor) will create a string from a symbol.

2.2.2.4 Factors

While character data is useful for representing data that is unique to each case (like an individual's address), there are advantages to using a different representation for data with many anticipated repeated values (like an individual's state). Factors are a data structure that allow the user to see full labels, but internally, the computer only sees an efficient pool of possible levels or values. That is, a factor is essentially a mapping between a label and a corresponding key, with a possible ordering assigned to the labels.

Factors are not a built-in data type, but are provided by the `CategoricalArrays` package, which is loaded as other packages are:

```
using CategoricalArrays
```

We use an example from the package's documentation:

We use an example based on coffee sizes at your neighborhood Starbucks. An order consisted of 4 drinks with these sizes, put into a categorical array:

```
x = categorical(["Grande", "Venti", "Tall", "Venti"], ordered=true)
```

```
4-element CategoricalArray{String,1,UInt32}:
 "Grande"
 "Venti"
 "Tall"
 "Venti"
```

This appears to be like a character vector, but the type is different. First, let's peek to see how values are internally stored. The user-visible values are stored with:

```
x.pool
```

```
CategoricalPool{String, UInt32}(["Grande", "Tall", "Venti"]) with ordered levels
```

Internally, the computer sees:

```
x.refs
```

```
4-element Vector{UInt32}:
 0x00000001
 0x00000003
 0x00000002
 0x00000003
```

The levels are the labels assigned to the internal values. We can see that the levels are internally kept as 32-bit integers, which in general is more space efficient than storing the labels. The command `levelcode.(x)` will show the values using the more readable 64-bit integers.

Working with levels is the key difference. First levels, may be *ordered* either by specifying `ordered=true` to the constructor *or* by calling `ordered!(x, true)`. The example data has an odd order (call `levels(x)` to see), it coming from lexical sorting. So `Grande` is before `Venti`, despite the latter being 16 oz. and the former 20 oz (for hot drinks, 24oz for cold)[15].

```
x[1] < x[2]
```

```
true
```

To reorder, we call the `levels!` function with the desired order:

```
levels!(x, ["Tall", "Venti", "Grande"])
x[1] > x[2]
```

[15] There are also Demi (3oz), short (8oz), and Trenta (31oz).

```
true
```

The levels can be extended through assignment. For example, we might prefer the label "Tall, 12oz" to "Tall" and can readily do this:

```
x[3] = "Tall, 12oz"
```

```
"Tall, 12oz"
```

After this command, there are 4 levels though only 3 used in the vector. To trim out extra levels, the `droplevels!` function can be used.

For multiple replacements, the `replace` function is useful and the levels are adjusted accordingly:

```
replace(x, "Grande" => "Grande, 20oz", "Venti" => "Venti, 16oz")
```

```
4-element CategoricalArray{String,1,UInt32}:
 "Grande, 20oz"
 "Venti, 16oz"
 "Tall, 12oz"
 "Venti, 16oz"
```

Categorical data can be combined, with `vcat`, say, and if ordered, as much as possible, an order will attempt to be matched.

2.2.3 Logical data and operators

The Boolean type in `Julia` has two values: `true` and `false`.

Boolean values have their own algebra. The *short-circuiting* `&&` and `||` implement "and" and "or:"

```
false && false, true && false, false && true, true && true
```

```
(false, false, false, true)
```

```
false || false, true || false, false || true, true || true
```

```
(false, true, true, true)
```

These are called "short circuiting," as the right hand side is only evaluated if it need be (as in `true || false`, the statement is known to be true after the left side of `||` is evaluated, so the right hand side is not evaluated). This is used frequently for error messages, as the `error` is not called when the expression is true. These operations may be broadcasted, so the above might have been illustrated by `[true false] .&& [true, false]`. The operation have left-to-right associativity, so it is common to see them in a sequence.

Many operations promote these values to `1` and `0`. For example, `true + false` is `1`, `true * false` is `0`, and `sum([true,false, true])` is `2`; internally, the complex number i is internally a pair `(false, true)` indicating no real part and an imaginary part.

Boolean values are returned by the *comparison operators* `<`, `<=`, `==`, `===`, `>=`, and `>`. These have the expected meaning, save `==` is a test of equality (not `=`, which is used for assignment) *and* `===` is a test of whether two values are *identical*. (E.g. for a vector `x` we have `x == copy(x)` is true, but `x === copy(x)` is false, as the latter does not point to the exact same container.) The symbol `!` is used for negation.

Boolean values can be used for *indexing*. Suppose `inds` is a vector of `true`s and `false`s with the *same* length as a vector `x`, then `x[inds]` will return those values from `x` where `inds` is `true`.

To create such Boolean vectors, the comparison operators are typically used *combined* with broadcasting. For example, the following redefines the `whale` dataset, then filters out only those values bigger than or equal to 200:

```
whale = [74, 122, 235, 111, 292, 111, 211, 133, 156, 79]
whale[ whale .>= 200]
```

```
3-element Vector{Int64}:
 235
 292
 211
```

Or, this example – which shows the mathematically natural chaining of comparison operators – filters out only the values in $[100, 125)$:

```
whale[ 100 .<= whale .< 125]
```

```
3-element Vector{Int64}:
 122
 111
 111
```

2.2.3.1 Querying elements

There are other helper functions to query the elements of a Boolean vector.

- `any(v)` will return `true` if any of the elements of the Boolean vector `v` are `true`.
- `all(v)` will return `true` if all of the elements of the Boolean vector `v` are `true`.
- `findfirst(v)` will return the index of the *first* `true` value in the Boolean vector `v` or return `nothing` if none is found. A *predicate* function can be used, as in `findfirst(f, v)` which effectively calls `findfirst` on `f` applied to each element of `v`.
- `findlast(v)` will return the index of the *last* `true` value in the Boolean vector `v` or return `nothing` if none is found. A *predicate* function can be used, as in `findlast(f, v)` which effectively calls `findlast` on `f` applied to each element of `v`.
- `findnext(v, i)` will find the first `true` value after *index* `i` in the Boolean vector `v` or return `nothing` if none is found. A *predicate* function can be used, as in `findnext(f, v, i)` which effectively calls `findnext` on `f` applied to each element of `v`.

In general, `x in v` will check if the element `x` is in the vector `v`. The Unicode operator `∈` can replace `in`.

2.2.4 Date and time types

`Julia` provides the built-in `Dates` module for working with date and time data. This module need not be installed, but is not loaded by default, so loading or importing it is needed to access the functionality.

```
using Dates
```

Constructing a date is done with the `Date` constructor which expects a year, followed by an optional month and day. Date and time objects have the `DateTime` constructor. This example uses the vernal equinox, summer solstice, autumnal equinox, and winter solstice in the year 2022 for illustration:

```
ve, ss, ae, ws = Date(2022, 3, 20), Date(2022, 6, 21), Date(2022, 9, 22), Date(2022, 12, 21)
```

```
(Date("2022-03-20"), Date("2022-06-21"), Date("2022-09-22"), Date("2022-12-21"))
```

Dates can also be parsed from a string. The following uses the default format;[16]:

```
Date.(["2022-03-20", "2022-06-21", "2022-09-22", "2022-12-21"])
```

```
4-element Vector{Date}:
 2022-03-20
 2022-06-21
 2022-09-22
 2022-12-21
```

Date objects have the accessors `year`, `month`, and `day`:

```
year(ve), month(ve), day(ve)
```

```
(2022, 3, 20)
```

The objects allow for natural operations, such as comparisons and differences:

```
ve < ss < ae < ws, ae - ve, ve + Day(93)
```

```
(true, Day(186), Date("2022-06-21"))
```

The difference is returned in the number of days. The last command shows the duration of 93 days can be consteucted with `Day` and its value added to a `Date` object.

There are ways to query how a date falls within the calendar

```
dayofyear(ve), dayofweek(ve), dayname(ve), dayofweekofmonth(ve)
```

```
(79, 7, "Sunday", 3)
```

[16] Different date formats are possible, see `?DateFormat` for details.

The last command returning that the vernal equinox in 2022 fell on the third Sunday of the month.

2.2.5 Structured data

Structured data may not represent statistical data, but is useful nonetheless, e.g, for specifying the year of the counts in the `whale` data set.

For a vector of all ones or all zeros, the `ones` and `zeros` functions are useful. The command `ones(n)` will return a vector of `n` zeros using the default `Float64` type. To specify a different type, such as `Int64`, the two-argument form, `ones(T, n)`, is available. Similarly, `zeros` is used to create a vector of zeros. The singular `one()`, `zero()` (one `one(T)` and `zero(T)`) are useful for generic programming.

Arithmetic sequences, $a, a+h, a+2h, \ldots, b$ can be created with the colon operator `a:h:b` or `a:b` when `h` is `1`. This operator returns a recipe for generating the sequence, it is lazy – it does not generate the sequence. The precedence is such that simple arithmetic operations do not need parentheses. That is `a+1:b-1` represents the sequence $a+1, a+2, \ldots, b-1$. Arithmetic sequences are useful for indexing into a vector.

The colon operator for floating point values may or may not stop at `b`. Programming this is harder than it seems. The simple example of `1/10:1/10:3/10` should be $1/10, 2/10, 3/10$, but it turns out that on the computer `1/10 + 2*1/10` is actually *just larger* than `3/10`. See the value of `3/10 - (1/10 + 1/10 + 1/10)` to investigate. However, the algorithm of `:` does produce the result with 3 values here.

```
1/10:1/10:3/10 |> collect
```

```
3-element Vector{Float64}:
 0.1
 0.2
 0.3
```

The `range` function creates sequences. The common usage is `range(start, stop, length)`. That is `a:h:b` specifies a *step* size, whereas the *positional* arguments of `range` specify the number of values between the starting and stopping values. Keyword arguments allow other combinations of `start`, `stop`, `step`, and `length`. The `range` returns a similar expression as the colon operator, it does not realize the entire range of values.

Scalar multiplication, scalar division, and addition of like-sized ranges are defined, as they return an arithmetic sequence.

For example, if `whale` holds beaching numbers for the years 2010 through 2019, we can get the odd years though the following:

```
oddyrs = 2011:2:2019
whale[oddyrs .- 2010 .+ 1] # 1-based offset is why we add 1
```

```
5-element Vector{Int64}:
 122
 111
 111
 133
```

```
79
```

2.3 Functions

`Julia` has simple syntax for basic user-defined functions.

For simple functions, the syntax borrowed from math is useful. For example, here we define a function to find the `mad` defined by the median of the transformed data $|x_i - M_x|$.

```
MAD(x) = median(abs.(x .- median(x)))
```

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

The function is named `MAD` (to distinguish it from the already defined `mad` function in `StatsBase`) and as written accepts a vector and returns a summary number:

```
MAD(whale)
```

```
38.5
```

For functions which are not one liners, a pair of `function`-`end` keywords will define a block. For example, the following computes the fifth standardized moment (skewness and kurtosis related to the 3rd and 4th). The first line is one way to document a function in `Julia`.

```
" Compute 5th standardized momemt: m_5 / m_2^(5/2)"
function fifth_sm(x)
    xbar, n = mean(x), length(x)
    m5 = sum((xi - xbar)^5 for xi in x) / n
    m2 = sum((xi - xbar)^2 for xi in x) / n
    m5 / m2^(5/2)
end
fifth_sm(whale)
```

```
3.6325544657722215
```

(The above uses a *generator*, created by the use of `for` and `in` to loop over the different values of `x` rather than broadcasting.)

The repetition above in `m5` and `m2` could be avoided if we made a function to compute the sample moments about the mean which accepted both the data and a value for the exponent:

```
sample_moment(x, n=2) = sum((xi - x)^n for xi in x) / length(x)
fifth_sm(x) = sample_moment(x, 5) / sample_moment(x)^(5/2)
```

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

The variable `n` is in the second position *and* has a *default* value of `2` which is employed in the denominator of the above, where `sample_moment` is called with just a single argument.

There can be many positional arguments, only the last ones can have default values specified.

Functions can have a variable number of arguments. Here is a way to find the proportions of a set of numbers that is not stored in a container, but rather is passed to the function separated by commas:

```
proportion(xs...) = collect(xs) / sum(xs)
```

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

The splat syntax `...` indicates a variable number of arguments in a function definition, and can be used to expand a list of arguments when used inside a function call. The use of `collect`, above, is needed above to generate a vector, as `xs` is passed to the body of the function as a tuple and tuples do not have division defined for them.

The `mad` function from `StatsBase` has signature `mad(x; center=median(x), normalize=true)`. This shows the use of *keyword arguments*. These have a default value, which, as illustrated, can depend on the data passed in. To call a function without the default, the keyword is typed, as in:

```
mad(whale; center=mean(whale))
```

```
74.13011092528009
```

(We use a semicolon to separate positional from keyword arguments, as that is needed to define a keyword, but commas can be used to call a keyword argument. What is important is the keywords come last.)

Functions, as defined above, are methods of a `generic function`. That is, there can be more than one method for a given name. (There are over 200 methods for the generic function named `+` in base `Julia` – and packages can extend this even more.) To direct or *dispatch* a call to the appropriate method, `Julia` considers the number and types of its positional arguments. That is, like `+`, functions can be defined differently for integers and floating point values.

We might like our `MAD` function to be more graceful than to throw a `MethodError` if a vector of strings is passed to it. A vector of strings has type `Vector{String}` so we could make a method just for that type:[17] 'Julia makes adding methods easy, but the types that are used to extend the function shouldn't be owned by other packages, as this is considered **type-piracy**. (Failing to do so may prompt a request for a *letter of marque*.)

```
MAD(x::Vector{String}) = print("Sorry, MAD is not defined for vectors of strings")
MAD(["one", "three", "four"])
```

```
Sorry, MAD is not defined for vectors of strings
```

(There are many different types that one might wish to exclude; there are many tricks to efficiently code for this. It is common to define a default method which errors and then a special case for the types that

[17] To extend the `mad` function from `StatsBase` requires the extra step of *importing* that function *or* qualifying it with its module, as in `StatsBase.mad(...) = ...`.

can be worked with. As types can be *concrete*, as the above, or *abstract*, it is possible to *parameterize* the type used for dispatch so that subtypes can be identified. For example, some operations return a `SubString` not a `String`. A vector of `SubString` will not match `Vector{String}` as even though both hold string data. The subtleties of parameterization are necessary to understand to write many special cases, but won't be necessary in these notes.)

A *higher order* function is one that takes one or more *functions* as an argument. As example is the calling style of `findfirst(predicate, x)` where `predicate` is a function which returns a boolean value. Higher-order functions are widely used with data wrangling. For such uses it is convenient to be able to define an *anonymous* function. These do not create methods for a generic function, as they have no name (though they can be assigned a name). Anonymous functions are easily defined through the pattern: `argument(s) -> body`, where `body`, when multi-line, can be contained within `begin`/`end` blocks.

For example, to find the index of the first year that there were 200 or more whale beachings we have:

```
findfirst(x -> x >= 200, whale)
```

```
3
```

For the tasks of logical comparisons, there are partially-applied versions of the operators that are essentially defined like anonymous functions (cf. `Base.Fix2`). In particular, `>=(200)` can be used for the anonymous function `x -> x >= 200`. To illustrate, here we see the index of first year where there were 200 or more beachings and the index of the last year, using `!` to negate a call (even though `>=(200)` would be identical):

```
findfirst(>=(200), whale), findlast(!<(200), whale)
```

```
(3, 7)
```

Another example would be to filter out those values less than `100`. The `filter(predicate, v)` function does this:

```
filter(<(100), whale)
```

```
2-element Vector{Int64}:
 74
 79
```

2.3.0.1 Broadcasting alternatives; iteration

As an illustration of a few other concepts, we consider alternatives to broadcasting that can prove useful. Consider the simplest case of broadcasting a function `f` over a single collection `x`.

For example here is broadcasting:

```
x = [1, 4, 9]
f(x) = sqrt(x)
f.(x)
```

```
3-element Vector{Float64}:
 1.0
 2.0
 3.0
```

This use of broadcasting is also called the *mapping* of `f` over `x`. The `map(f, x)` function is defined:

```
map(f, x)
```

```
3-element Vector{Float64}:
 1.0
 2.0
 3.0
```

Either broadcasting or `map` do the following: for each element of `x` apply the function `f`. This can be represented with the chaining operator, `|>`, adjusted to broadcast the values of `x`:

```
x .|> f
```

```
3-element Vector{Float64}:
 1.0
 2.0
 3.0
```

The iteration over `x` can be made explicit with a `for` loop. The basic syntax would be:

```
out = Any[]
for xi in x
   push!(out, f(xi))
end
out
```

```
3-element Vector{Any}:
 1.0
 2.0
 3.0
```

For loops require extra effort for accumulation, which, in the above, required the selection of a container. We used `Any[]` to create a zero-length container that can hold any object and push values onto this. If would be preferable to have a concrete type, which in this case is just `Float64[]`, but that requires some peeking to the output of `f`. This is to point out some background work performed by `map` and broadcasting that simplify other common tasks.

The `for xi in x` part of the for loop assigns `xi` to each iterate of `x`. There are some iterations that return more than one value at once, and it is common to see these *destructured* in the syntax. For example, `enumerate` is a helper function which takes an iterable object, like a vector, and iterates over both the index and the value:

```
for (i, xi) in enumerate(x)
  print("element $i is $xi. ")
end
```

```
element 1 is 1. element 2 is 4. element 3 is 9.
```

A *comprehension* is a good alternative to a for loop when accumulation is required and the computation for each iterate is simple to express. Comprehensions have the basic syntax `[ex for x in xs]` and additional syntax for multiple iterations. The expression `ex` can use the variable `x`; `xs` is some iterable, such as a vector. For example, we might have this alternative to `f.(x)`

```
[xi - mean(x) for xi in x]
```

```
3-element Vector{Float64}:
 -3.666666666666667
 -0.666666666666667
  4.333333333333333
```

Comprehensions use *generators*, which can also be used for many other functions, such as `sum`, which was previously illustrated. This form has the advantage of not needing to allocate temporary space to compute. For example `sum([xi for xi in x])` would have to find space for the vector created by the comprehension, but the similar – and easier to type – `sum(xi for xi in x)` would not.

In this example, we sum the squared differences, passing an optional function to `sum`:

```
f(x) = x^2
sum(f, xi - mean(whale) for xi in whale)
```

```
46020.399999999994
```

Table 2.2: Various convenient iterators in `Julia`.

Iterator	Description
`eachindex`	iterate over each index
`values`	iterate over values in a container
`enumerate`	iterate over index and value
`keys`	iterate over keys for the container
`pairs`	iterate over (key,value) pairs in the container
`zip`	iterate over multiple iterators; the value is a tuple with an element from each
`eachcol`	for tabular data, iterate over the columns
`eachrow`	for tabular data, iterate over the rows

2.4 Numeric summaries

The `StatsBase` package provides numerous functions to compute numeric summaries of a univariate data set.

For **measures of center** we have the mean (average), median (middle value), and mode (most common value):

```
mean(whale), median(whale), mode(whale), mean(trim(whale, prop=0.2))
```

```
(152.4, 127.5, 111, 140.66666666666666)
```

The trimmed mean is computed by composing `mean` with the `trim` function which returns a generator producing the trimmed values (with proportion to trim specified to `prop`).

For **measures of spread** we have the standard deviation, the median absolute deviation, and the interquartile range ($Q_3 - Q_1$):

```
std(whale), mad(whale), iqr(whale)
```

```
(71.5078861229849, 57.08018541246567, 86.25)
```

The `quantile(x, p)` function returns **measures of position**. Keywords `alpha` and `beta` can be used to adjust the algorithm employed. The `0` quantile is the minimum, the `1` quantile is the maximum, and `0.5` the median. The `p` may be an iterable. Here we see that type of `p` is used to compute the value output:

```
quantile(whale, (0, 1//2, 1.0))
```

```
(74, 255//2, 292.0)
```

The `quantile` function is used internally by `StatsBase`: the `iqr` is defined by the difference of `quantile(x, [1/4, 3/4])`; the `summarystats` function returns a summary of a data set, with the 5-number summary of the quantiles (`p=0:1/4:1`), the mean, the length, and the number of missing data values.

A **measure of position** gives a sense of how large a value is relative to the data set. Knowing a value is the 20th percentile, say, says it is larger than 20 percent of the data and smaller than 80 percent. The percent of data less or equal a value can be computed by `sum(data .<= value)` or `sum(x <= value for x in data)`.

The z-score is a different measure of position. The z score is computed by `(value - mean(data))/std(data)` or for all values at once with:

```
zs = (whale .- mean(whale)) / std(whale)
```

```
10-element Vector{Float64}:
 -1.0963825705204249
 -0.4251279355079199
  1.155117351084019
 -0.5789571226982856
```

```
 1.9522322301613686
-0.5789571226982856
 0.8194900335777664
-0.2712987483175542
 0.05034409762593779
-1.0264602127066222
```

A rule of thumb, based on a certain bell-shaped distribution, is that values larger than 3 in absolute value are unusual, none of which are seen in this data. The z score measures the difference from the mean on a scale of standard deviations. It is typical form of *standardization* and is supported directly through either:

```
zs1 = zscore(whale)
zs2 = standardize(ZScoreTransform, Float64.(whale))
zs == zs1 == zs2
```

```
true
```

Why so many measures?

There are two main measures: those based on differences and those based on position.
The mean is defined by

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i.$$

A property of the mean is when we *center* by the mean, i.e. take the transformation $y_i = x_i - \bar{x}$, then the mean of the y_i is just 0. Put another way, the mean is the point where the differences to the left and right of the mean even out when added.
The standard deviation, as a sense of scale, has the property that if we center and *then scale* by the standard deviation, the center will be 0 and the scale will be 1. That is, the z-scores have mean 0 (as we've centered) and standard deviation 1 (as we've scaled) – the z scores speak to the "shape" or distribution of values of a data set.
These measures are sensitive – or not *resistant* – to one or more *outlying* values. For example, the average wealth of people in a bar changes dramatically if someone like a pre-crash Elon Musk walks in. The standard deviation is similar. So measures based on position are better when data is *skewed*, especially if heavily skewed. For these, the median and IQR are not impacted greatly by 1 large value. That is they are resistant to outliers. The extreme values (the minimum and maximum) are less so, so the range of the data is a poor measure of spread, the range of the middle quartiles (the IQR) a resistant measure of spread.

Table 2.3: Various measures of center, spread, and position in a univariate data set.

Measure	Type	Description
`mean`	center	Average value
`median`	center	Middle value
`mean ∘ trim`	center	Trimmed mean

Measure	Type	Description
`var`	spread	"Average" squared distance from mean
`std`	spread	The square of the variance
`mad`	spread	The median absolute deviation from center
`IQR`	spread	Range of middle half of data
`extrema`	spread	Smallest and largest values
`quantile`	position	Value where data would be split for give proportion
`summarystats`	position	The quartiles and extrema
`zscore`	position	Standardizing transformation

2.5 Shape

The 5-number summary gives some insight into the *shape* of a distribution, in that it can show skewness. The `skewness` function numerically quantifies this:

```
skewness(whale)
```

```
0.7785957149991207
```

Symmetric data has 0 skew.

The `kurtosis` function numerically summarizes if a distribution has long or short tails compared to a benchmark distribution, the normal:

```
kurtosis(whale)
```

```
-0.5833928499406711
```

However, the **shape of a distribution** is best seen graphically.

A stem or stem-and-leaf plot is useful summary of a data set that can be computed easily by hand for modest sized data. Basically numbers are coded in terms of a stem and a leaf. The stems are not repeated, so the data can be more compactly displayed. The data is organized so that the extrema, the median, and the shape of the data can be quickly gleaned.

The basic stem plot is not part of the `StatsBase` package. This one, below, is modified from a RosettaCode submission. It is simplified by assuming non-negative data.

```
using Printf
function stemleaf(a::Array{T,1}, leafsize=1) where {T<:Real}
    ls = 10^floor(Int, log10(leafsize))
    aa = floor.(Int, sort(a)/ls)
    out = divrem.(aa, 10)
    stem, leaf = first.(out), last.(out)
    io = IOBuffer()
```

```
    for i in stem[1]:stem[end]
        i != 0 || println(io, "")
        print(io, (@sprintf "%10d | " i))
        println(io, join(map(string, leaf[stem .== i]), ""))
    end
    println(io, " Leaf Unit = " * string(convert(T, ls)))
    String(take!(io))
end
```

```
stemleaf (generic function with 2 methods)
```

For the `whale` data, which is spread out over a wide range, we use a leaf size of 10:

```
stemleaf(whale, 10) |> print
```

```
        0 | 77
        1 | 11235
        2 | 139
 Leaf Unit = 10
```

We can identify `0|7` which is `0*100 + 70` or 70 as the "smallest" value. The *actual* smallest value is 74, but the data is rounded for this graphic so that just two digits (the stem and leaf) can represent all the values. Similarly 290 is identified as the largest values (which is actually 292).

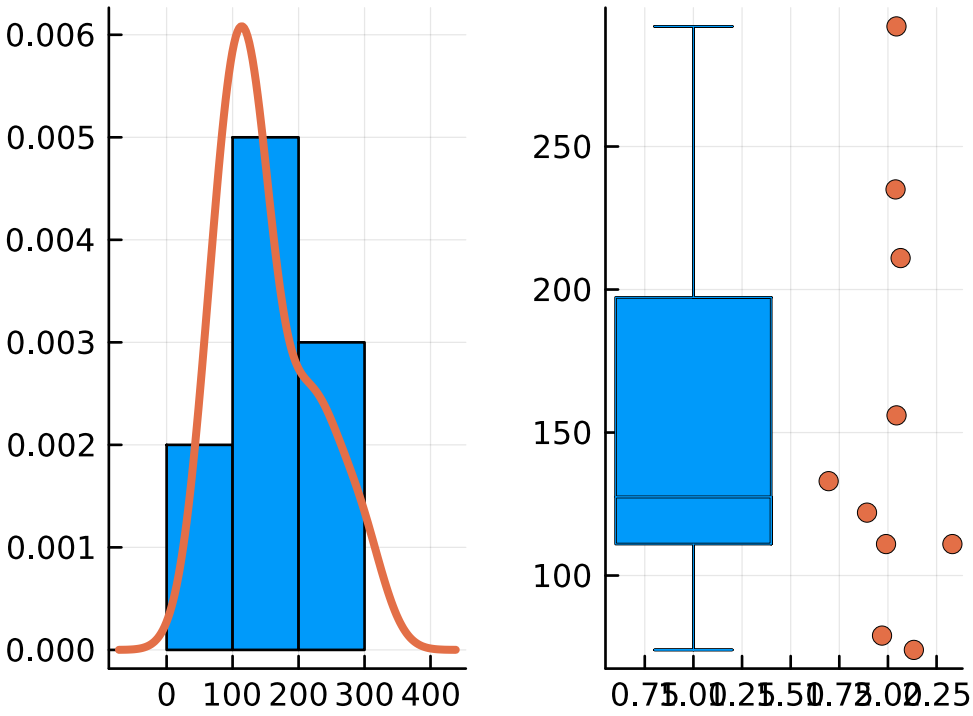
For graphical summaries of data, we will utilize the `StatsPlots` package, which adds to the `Plots` package recipes for many statistical graphics.

```
using StatsPlots
p1 = histogram(whale; normalize=true, legend=false)
density!(p1,whale, linewidth=3)
p2 = boxplot(whale, legend=false)
dotplot!(p2, whale);
```

The figures are shown in Figure 2.1. The `histogram` has `normalize=true` passed to it to have total area of 1 allowing a density curve to be superimposed. The `Plots` packages uses the `!` convention to indicate that `density!` should use the current figure to layer on a density plot. The boxplot and dotplot are plotted side by side, to show how the boxplot summarizes the data through the quartiles, with the data split into quarters by the box and its middle line.

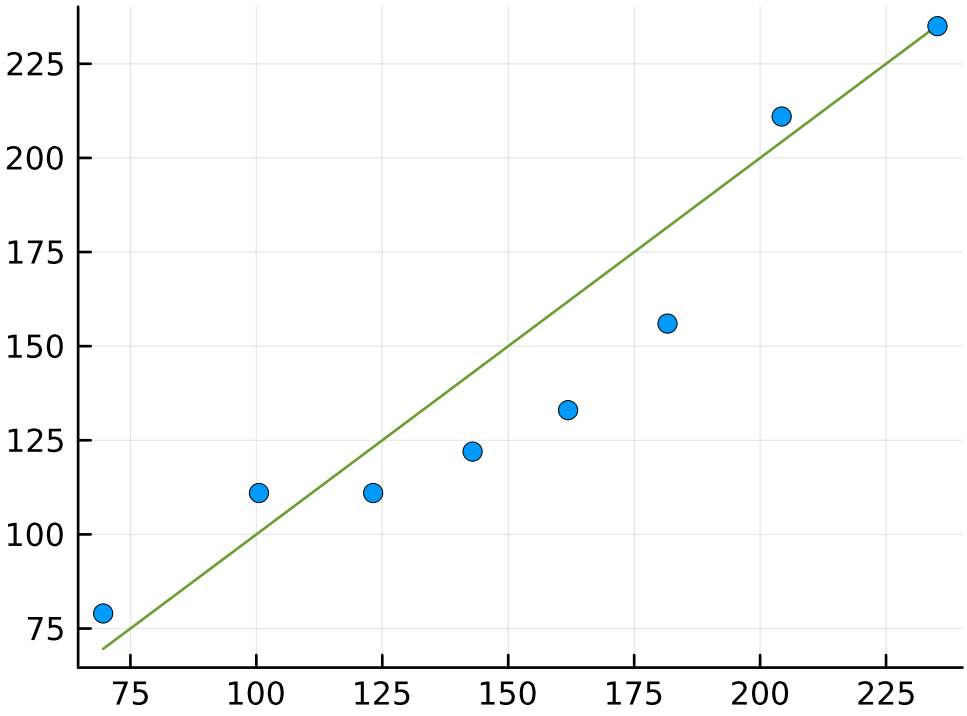
A *quantile-normal* plot is a graphic to assess if a data set comes from a *normal distribution*, a reference bell shaped distribution. The basic graphic is a scatter plot of a quantile of the data set with a corresponding quantile of the reference distribution. If the data set has a roughly bell-shaped distribution with "typical" tails, then the dots will mostly fall on a straight line; if the data set is skewed one edge will deviate from the line; if the data set is symmetric, but has long tails these will shows a deviations in both edges of the line. The `qqnorm` function produces Figure 2.2:

Figure 2.1: Various graphical summaries of the `whale` data, which show a slight skew.



```
qqnorm(whale; legend=false)
```

Figure 2.2: A quantile-normal plot of the `whale` data



Chapter 3

Tabular data

Some data sets are naturally stored in a tabular format, like a spreadsheet. This chapter looks at such data.

In the following, we will utilize these basic packages:

```
using StatsBase, StatsPlots
```

3.1 Data frames

The basic framework for exploratory data analysis is a data set with 1 or more observations for n different cases. For each case (or observation) a *variable* will be a length-n data set, with values measuring the same type of quantity, or perhaps missing. The variables can be stored as vectors. These are *traditionally* arranged in a rectangular manner with each *row* representing the measurements for a given case and each column representing the measured values for each variable. That is the columns are homogeneous (as they measure the same thing), but the rows need not be (as different variables are measuring different quantities).

Matrices contain rectangular data, but are homogeneous. As such, a new structure is needed. A data frame is a concept borrowed from S Plus where they were introduced in 1991. The R language has data frames as a basic built-in data container. In `Julia` the structure is implemented in an external package, `DataFrames`.

```
using DataFrames
```

> **An OG package**
>
> The `DataFrames` package dates back to 2012, at least, and must be one of `Julia`'s oldest packages. Julia was on version `0.1` then. It is also under continual development and has a comprehensive set of documentation and tutorials. This introduction attempts to illustrate some basic usage; consult the provided documentation for additional details and applications.

3.1.1 Data frame construction

There are different ways to construct a data frame.

Consider the task of the Wirecutter in trying to select the best carry on travel bag. After compiling a list of possible models by scouring travel blogs etc., they select some criteria (capacity, compartment design, aesthetics, comfort, ...) and compile data, similar to what one person collected in a spreadsheet. Here we create a much simplified spreadsheet for 3 listed bags with measurements of volume, price, laptop compatability, loading style, and a last checked date – as this market improves constantly.

```
product         v  p    l loads       checked
Goruck GR2      40 395  Y front panel 2022-09
Minaal 3.0      35 349  Y front panel 2022-09
Genius          25 228  Y clamshell   2022-10
```

We see that product is a character, volume and price numeric, laptop compatability a Boolean value, load style one of a few levels, and the last checked date, a year-month date.

We create vectors to hold each. We load the `CategoricalArrays` and `Dates` packages for a few of the variables:

```
using CategoricalArrays, Dates
product = ["Goruck GR2", "Minaal 3.0", "Genius"]
volume = [40, 35, 25]
price = [395, 349, 228]
laptop_compatability = categorical(["Y","Y","Y"])
loading_style = categorical(["front panel", "front panel", "clamshell"])
date_checked = Date.(2022, [9,9,10])
```

```
3-element Vector{Date}:
 2022-09-01
 2022-09-01
 2022-10-01
```

With this, we use the `DataFrame` constructor to combine these into one data set.

```
d = DataFrame(product = product, volume=volume, price=price,
              var"laptop compatability"=laptop_compatability,
              var"loading style"=loading_style, var"date checked"=date_checked)
```

	product	volume	price	laptop compatability	loading style	date checked
	String	Int64	Int64	Cat…	Cat…	Date
1	Goruck GR2	40	395	Y	front panel	2022-09-01
2	Minaal 3.0	35	349	Y	front panel	2022-09-01
3	Genius	25	228	Y	clamshell	2022-10-01

We use `var".."` for names with spaces.

In the above, we repeated the names of the variables to the constructor. A style akin to the construction of named tuple, could also have been used where variables after a semicolon are implicitly assumed to have

the desired name:

```
d = DataFrame(; product, volume, price,
              var"laptop compatability"=laptop_compatability,
              var"loading style"=loading_style, var"date checked"=date_checked)
```

	product	volume	price	laptop compatability	loading style	date checked
	String	Int64	Int64	Cat…	Cat…	Date
1	Goruck GR2	40	395	Y	front panel	2022-09-01
2	Minaal 3.0	35	349	Y	front panel	2022-09-01
3	Genius	25	228	Y	clamshell	2022-10-01

We see that the `DataFrame` type displays values as we might expect with the column header showing the name of the variable and the storage type. The row numbers are added and may be used for reference.

There is a convention for attaching new columns to a data frame that allows an alternate construction:

```
d = DataFrame()   # empty data frame
d.product = product
d.volume = volume
d.price = price
d."laptop compatability" = laptop_compatability
d."loading style" = loading_style
d."date checked" = date_checked
d
```

	product	volume	price	laptop compatability	loading style	date checked
	String	Int64	Int64	Cat…	Cat…	Date
1	Goruck GR2	40	395	Y	front panel	2022-09-01
2	Minaal 3.0	35	349	Y	front panel	2022-09-01
3	Genius	25	228	Y	clamshell	2022-10-01

This uses one of a few different ways to refer to a column in a data frame.

As an alternative, the `insertcols!` function can insert multiple columns, for example the last one in the example, could have been written `insertcols!(d, "date checked" => date_checked)`.

Before exploring how we might query the data frame, we note that an alternate means to describe the data set is by row, or case. For each row, we might use a named tuple with the names indicating the variable, and the values the measurement. The `DataFrame` constructor would accept this, here we shorten the names and only do one row (for now):

```
d = DataFrame([
 (b = "Goruck GR2", v = 40, p = 395, lap = "Y", load = "front panel", d = Date("2022-09-01"))
])
```

	b	v	p	lap	load	d
	String	Int64	Int64	String	String	Date
1	Goruck GR2	40	395	Y	front panel	2022-09-01

Here, the variable types are different, as it takes more effort to make one categorical. The same way we defined a column allows us to *redefine* that column (replacing the container, not re-using it for storage). For example, we could do:

```
d.lap = categorical(d.lap)
d.load = categorical(d.load)
```

```
1-element CategoricalArray{String,1,UInt32}:
 "front panel"
```

and then the two variables would be categorical.

There are other ways to define row by row:

- A data frame with empty variables can be constructed and then values as tuples may be `push!`ed onto the data frame

```
push!(d,  ("Minaal 3.0", 35, 349, "Y", "front panel", Date("2022-09-01")))
```

	b	v	p	lap	load	d
	String	Int64	Int64	Cat…	Cat…	Date
1	Goruck GR2	40	395	Y	front panel	2022-09-01
2	Minaal 3.0	35	349	Y	front panel	2022-09-01

A named tuple can also be used.

- Similarly, rows specified by dictionaries may be pushed onto a data frame

```
push!(d, Dict(:b => "Genius", :v => 25, :p => 228, :lap => "Y",
              :load => "clamshell", :d => Date("2022-10-01")))
```

	b	v	p	lap	load	d
	String	Int64	Int64	Cat…	Cat…	Date
1	Goruck GR2	40	395	Y	front panel	2022-09-01
2	Minaal 3.0	35	349	Y	front panel	2022-09-01
3	Genius	25	228	Y	clamshell	2022-10-01

(A dictionary is a `key => value` container like a named tuple, but keys may be arbitrary `Julia` objects – not always symbols – so we explicitly use symbols in the above command.)

> **The `Tables` interface**
>
> In the `Julia` ecosystem, the `Tables` package provides a lightweight means to describe tabular data, like a data frame, and provides some basic means to query the data. The different ways in which a data frame can be defined, reflect the different ways `Tables` objects can be created.
> The basics involve a vector (a type of array) and a struct (A struct is a composite record, or a named tuple, or a dictionary). Both of these specify a table (provided there is a homogeneity in the inner objects):

- As a struct of arrays. This is used by calling `DataFrame(a=[1,2], b= [2,1])`, say.
- As an array of structs. This is used by calling `DataFrame([(a=1,b=2), (a=2,b=1)])`

3.1.1.1 RDataset

Data frames are often read in from external sources, and not typed in. The `RDataSets` package has collected data sets from many different R packages and bundled them into data sets for `Julia` in the form of data frames. We use these for various examples.

```
using RDatasets
cars = dataset("MASS", "Cars93")  # package, data set name
first(cars, 2)  # first 2 rows only
```

	Manufacturer	Model	Type	MinPrice	Price	MaxPrice	MPGCity	MPGHighway	
	Cat…	String	Cat…	Float64	Float64	Float64	Int32	Int32	
1	Acura	Integra	Small	12.9	15.9	18.8	25	31	…
2	Acura	Legend	Midsize	29.2	33.9	38.7	18	25	…

3.1.1.2 CSV.read

For interacting with spreadsheet data, when it is stored like a data frame, it may be convenient to export the data in some format and then read that into a `Julia` session. A common format is to use comma-separated values, or "csv" data. The `CSV` package reads such files. The `CSV` package reads in delimited text data (the source) using the `Tables` interface and can be instructed to write to a data frame (the sink). The `CSV.read` file is used. Here we write to some file to show that we can read it back:

```
using CSV
f = tempname() * ".csv"
CSV.write(f, d)  # write simple data frame to file `f`
d1 = CSV.read(f, DataFrame)
```

	b	v	p	lap	load	d
	String15	Int64	Int64	String1	String15	Date
1	Goruck GR2	40	395	Y	front panel	2022-09-01
2	Minaal 3.0	35	349	Y	front panel	2022-09-01
3	Genius	25	228	Y	clamshell	2022-10-01

Comma-separated-value files are simple text files, with no metadata to indicate what type a value may be. Some guesswork is necessary. We observe that the two categorical variables were read back in as strings, as that is the default. As seen above, that can be addressed. The date data was properly identified as dates, but a day attribute was added.

The filename, may be more general. For example, it could be `download(url)` for some url that points to a csv file to read data from the internet.

> **Read and write**
>
> The methods `read` and `write` are qualified in the above usage with the `CSV` module. In the `Julia` ecosystem, the `FileIO` package provides a common framework for reading and writing files; it uses the verbs `load` and `save`. This can also be used with `DataFrames`, though it works through the `CSVFiles` package – and not `CSV`, as illustrated above. The read command would look like `DataFrame(load(fname))` and the write command like `save(fname, df)`. Here `fname` would have a ".csv" extension so that the type of file could be detected.

Table 3.1: Basic usage to read/write `.csv` file into a data frame.

Command	Description
`CSV.read(file_name, DataFrame)`	Read csv file from file with given name
`CSV.read(IOBuffer(string), DataFrame)`	Read csv file from a string using an `IOBuffer`
`CSV.read(download(url), DataFrame)`	Read csv file an url
`open(file_name,"w") do io; CSV.write(io, dd); end`	Write data frame `df` to csv file
`DataFrame(load(file_name))`	Read csv file from file with given name using `CSVFiles`
`save(file_name, df)`	Write data frame `df` to a csv file using `CSVFiles`

3.1.1.3 TableScraper

The `TableScraper` package can scrape tables from an HTML file in a manner that can be passed to `DataFrame` for conversion to a data frame. The command `DataFrame.(scrape_tables(url))` may just work to create the tables, but in practice, only well-formed tables can be scraped successfully; tables used for formatting purposes may fail. `TableScraper` is built on `Cascadia` and `Gumbo`, as is the `Harbest` package, which provides more access to the underlying data than `TableScraper`.

3.1.2 Column names

The variable names of a data set are returned by the `names` function, as strings:

```
names(d)
```

```
6-element Vector{String}:
 "b"
 "v"
 "p"
 "lap"
 "load"
 "d"
```

This function has a second argument which can be used to narrow or filter the names that are returned. For example, the following is a convenience for matching columns whose element type is a subtype of `Real`:

```
names(d, Real)
```

```
2-element Vector{String}:
 "v"
 "p"
```

The `rename!` function allows the names to be changed in-place (without returning a new data frame). There are many different calling patterns, but:

- to replace one name, the form `rename!(df, :v => :nv)` is used, where `:v` is the old name, and `:nv` the new one. The form follows the generic `replace!` function. The pair notation can be *broadcast* (`.=>`) to replace more than one name, as with `replace!(df, [:v1, :v2] .=> [:nv1, :nv2])`.
- to replace allnames, the form `renames!(df, [...])` is used, where the vector is of length matching the number of columns in `df`.

3.1.3 Indexing and assignment

The values in a data frame can be referenced by a row number and column number, both 1-based. For example, the 2nd row and 3rd column of `d` can be seen to be `349` by observation

```
d
```

	b	v	p	lap	load	d
	String	Int64	Int64	Cat…	Cat…	Date
1	Goruck GR2	40	395	Y	front panel	2022-09-01
2	Minaal 3.0	35	349	Y	front panel	2022-09-01
3	Genius	25	228	Y	clamshell	2022-10-01

but it can be referenced through indexing (row number first, column number second):

```
d[2, 3]
```

```
349
```

A value can also be assigned. Suppose, that bag went on sale and is now 20% off. This new price could be adjusted with:

```
d[2, 3] = floor(Int, 0.80 * d[2, 3]) # need an integer here
```

```
279
```

3.1.3.1 Column selectors

The 3rd column has a name, `p` in this case. We can use a column selector to avoid having to know which column number a variable is. For this a string or a symbol may be used, as in:

```
d[2, :p], d[2, "p"]
```

```
(279, 279)
```

More generally, it is possible to use more than 1 column in column selection. For example, to get both the volume and price, for the second bag we have:

```
d[2, [:v, :p]]
```

	v	p
	Int64	Int64
2	35	279

As well, either `[2, 3]` or `["v", "p"]` could have been used.

For data frames with many columns, a *regular* expression can be used. The column selector `r"^l"` would match all columns whose name begins with `l` (`lap` and `load`) in this case. (Regular expressions can be very complicated, but here we only assume that `r"name"` will match "name" somewhere in the string; `r"^name"` and `r"name$"` will match "name" at the beginning and ending of a string. Using a regular expression will return a data frame row (when a row index is specified) – not a value – as it is possible to return 0, 1 or more columns in the selection.

Column selectors, as seen may be column number(s), column names as strings, column names as symbols, regular expression.

In addition:

- Boolean vectors of a length matching the number of columns (`d[2, [false, true, true, false, false, false]]` would return the 2nd and 3rd columns.)
- The value `All()` which selects all columns
- the value `Between(a, b)` where, `a` and `b` are column selectors (such as `Between(:v, 3)`).
- `Not(...)` to select those columns not specified in `...`
- `Cols(a,b,...)` which combines different selectors, such as `Cols(:v, 3)`.

3.1.3.2 Slices

The `All()` column selector has a shortcut inherited from the indexing of arrays in base `Julia`, this being a colon, `:`. When used in the column position it refers to all the values in the row:

```
d[2, :]
```

	b	v	p	lap	load	d
	String	Int64	Int64	Cat…	Cat…	Date
2	Minaal 3.0	35	279	Y	front panel	2022-09-01

When used in the *row position*, it refers to all rows:

```
d[:, 3]
```

```
3-element Vector{Int64}:
 395
 279
```

```
228
```

In the above use, a vector is returned, not a data frame, as only one column was selected. But which vector? Is it a *copy* of the one the `d` holds, or is it the same container? In base `Julia` the answer depends on what side of an equals sign the use is on (meaning, if used in assignment the answer is different, as mentioned later). In this usage, a *copy* is returned.

To get the underlying column (not a copy), the selector `!` is used instead:

```
d[!, 3]
```

The notation `d.p` also refers to the 3rd variable. In this case, `d.p` is a view of the underlying data (using `!`) and not a copy.

The notation `d[:,:]` will refer to all rows and all columns of `d`, and in this case will be a *copy* of the data frame.

3.1.3.3 Assignment

Single values can be assigned, as previously illustrated. These values are assigned to the underlying vector in the data frame, so as with vectors, the assigned value will be converted to the column type, if necessary. Assigning a value that can't be converted will throw and error, as it would were such an assignment tried for a vector. To do such an assignment, the underlying vector must be changed.

The row selector `:` or `!` had different semantics when accessing the underlying data, the first returning a copy, the second a view. When used with assignment though, `:` refers to the underlying vector or does in-place assignment; whereas `!` will replace the the underlying container with the assigned container.

To illustrate, suppose the prices are rounded down, but actually contain an extra 99 cents. We can reassign `price` as following:

```
d[!, :p] = price .+ 0.99
```

```
3-element Vector{Float64}:
 395.99
 349.99
 228.99
```

as even though `price .+ 0.99` is no longer an integer vector, the `!` row indicator will instruct `d` to point to this new vector. Had `d[:, :p] = ...` been used above, an error would have been thrown as the underlying container is integer, but the new prices require floating point values to represent them.

Broadcast assignment (using `.=`) will be similar. The right hand side is expanded, and then assigned. Attempts to assign the wrong value type using `:` will error.

3.1.3.4 Missing values

Trying to assign a value of `missing` to a data frame requires the underlying vector to allow missing values. As with vectors, the `allowmissing` function is useful. For example, to give the 2nd bag a price of missing, we could do:

```
allowmissing!(d, :p)
d[2, :p] = missing
```

```
missing
```

By using the mutating form (with the trailing `!`), the data frame `d` is mutated in the `allowmissing!` call. This could be observed by looking a the underlying column type.

The new type is a union of the old type and `Missing`, so we could reassign the old value:

```
d[2, :p] = 349.99
```

```
349.99
```

3.1.3.5 Tables.jl interface

A table, like a data frame can be expected to loop, or iterate, over columns, say to find which columns are numeric. Or, they can be expected to iterate over rows, say to find which values are within a certain range. What should be the case with a data frame? As there are reasons to prefer either, none is specified. Rather, there is a difference based on the calling function. Data frames respect the `Tables` interface, which has methods to iterate over rows or columns. When iterating over columns, the individual vectors are given; when iterating over rows, a `DataFrameRow` object is returned. Calling `copy` on these rows objects will return a named tuple.

The basic iterators are `eachrow` and `eachcol`. For example, to see the column type for a data frame, we can broadcast `eltype` over the variables returned by `eachcol`:

```
eltype.(eachcol(d))  |> permutedims
```

```
1×6 Matrix{Type}:
 String  Int64  Union{Missing, Float64}  …  Date
```

To get the data frame as a vector of named tuples, then broadcasting `copy` over `eachrow` can be used: `copy.(eachrow(d))`.

3.1.4 Sorting

A data frame can be sorted by specifying a permutation of the indices, such as is returned by `sortperm`. For example, `d[sortperm(d.price), :]` will sort by price in increasing order. The keyword argument `rev` can be passed `true` to reverse the default sort order.

The generic `sort` function has a method for data frames, that offers a more convenient, though no additional, functionality. The `sort!` method does in-place sorting. The simplest call is to pass in a column to sort by, as in

```
sort(d, :p)
```

	b	v	p	lap	load	d
	String	Int64	Float64?	Cat…	Cat…	Date
1	Genius	25	228.99	Y	clamshell	2022-10-01
2	Minaal 3.0	35	349.99	Y	front panel	2022-09-01
3	Goruck GR2	40	395.99	Y	front panel	2022-09-01

This sorts by the price (`:p`) variable in *increasing* order. To sort by decreasing order a keyword argument `rev=true` can be specified:

```
sort(d, "d"; rev=true)
```

	b	v	p	lap	load	d
	String	Int64	Float64?	Cat…	Cat…	Date
1	Genius	25	228.99	Y	clamshell	2022-10-01
2	Goruck GR2	40	395.99	Y	front panel	2022-09-01
3	Minaal 3.0	35	349.99	Y	front panel	2022-09-01

There can be one or more column selectors. For this, the `order` function can be used to reverse just one of the columns, as in the following, which first sorts the dates in reverse order (newest first) and then within a data, uses lowest price first to sort:

```
sort(d, [order("d", rev=true), :p])
```

	b	v	p	lap	load	d
	String	Int64	Float64?	Cat…	Cat…	Date
1	Genius	25	228.99	Y	clamshell	2022-10-01
2	Minaal 3.0	35	349.99	Y	front panel	2022-09-01
3	Goruck GR2	40	395.99	Y	front panel	2022-09-01

3.1.5 Filtering rows

Filtering, or subsetting, is a means to select a specified selection of rows.

Let's consider the `cars` data set, previously defined by loading `dataset("MASS", "Cars93")`. This data set consists of 27 measurements on 93 cars from the 1990s. The data set comes from R's `MASS` package, and is documented there.

Two ready ways to filter are by using specific indices, or by using boolean indices generated by a logical comparison.

For the `cars` data set, the latter can be used to extract the Volkswagen models. To generate the indices we compare the `:Manufacturer` variable with the string `Volkswagen` to return a vector of length 93 that indicates if the manufacturer is VW. This is done with *broadcasting*: `cars.Manufacturer .== "Volkswagen"`. We use the easy dot access to variables in `cars`, alternatively `cars[:, :Manufacturer]` (or, more efficiently, `!`, as is used by the dot access) could be used were `:Manufacturer` coming from some variable. With this row selector, we have:

```
cars[cars.Manufacturer .== "Volkswagen", :]
```

	Manufacturer	Model	Type	MinPrice	Price	MaxPrice	MPGCity	MPGHighway	
	Cat…	String	Cat…	Float64	Float64	Float64	Int32	Int32	
1	Volkswagen	Fox	Small	8.7	9.1	9.5	25	33	…
2	Volkswagen	Eurovan	Van	16.6	19.7	22.7	17	21	…
3	Volkswagen	Passat	Compact	17.6	20.0	22.4	21	30	…
4	Volkswagen	Corrado	Sporty	22.9	23.3	23.7	18	25	…

(Yes, there are two dots, *each with different meanings* **and** two colons, *each with different meanings*.)

This approach lends itself to "find all rows matching some value" then "extract the identified rows." Written as two steps to emphasize there are two passes through the data. Another mental model would be loop over the rows, and keep those that match the query. This is done generically by the `filter` function for collections in `Julia` or by the `subset` function of `DataFrames`.

The `filter(predicate, collection)` function is used to identify just the values in the collection for which the predicate function returns `true`. When a data frame is used with `filter`, the iteration is over the rows, so the `eachrow` iterator is not used. We need a predicate function to replace the `.==` above. One follows. It doesn't need `.==`, as `r` is a data frame row and access produces a value not a vector:

```
filter(r -> r.Manufacturer == "Volkswagen", cars)
```

	Manufacturer	Model	Type	MinPrice	Price	MaxPrice	MPGCity	MPGHighway	
	Cat…	String	Cat…	Float64	Float64	Float64	Int32	Int32	
1	Volkswagen	Fox	Small	8.7	9.1	9.5	25	33	…
2	Volkswagen	Eurovan	Van	16.6	19.7	22.7	17	21	…
3	Volkswagen	Passat	Compact	17.6	20.0	22.4	21	30	…
4	Volkswagen	Corrado	Sporty	22.9	23.3	23.7	18	25	…

There are some *conveniences* that have been developed to make the predicate functions even easier to write, which will be introduced incrementally:

The basic binary comparisons, like `==`, are defined by functions essentially of the form `(a,b) -> ==(a,b)`. There are *partially applied* versions, essentially `a -> ==(a,b)`, with `b` applied, formed by `==(b)`. So the syntax `==("Volkswagen")` is a predicate taking a single argument which is then compared to the string `"Volkswagen"`.

However, `r` is a data frame row, what we need to pass to `==("Volkswagen")` is the value of the `:Manufacturer` column. For this another convenience is available.

The `pair` syntax, `=>` of the form `column selector(s) => predicate function` will pass just the value in the column(s) to the predicate function. Combined, the above can be simplified to:

```
filter(:Manufacturer => ==("Volkswagen"), cars)
```

	Manufacturer	Model	Type	MinPrice	Price	MaxPrice	MPGCity	MPGHighway	
	Cat…	String	Cat…	Float64	Float64	Float64	Int32	Int32	
1	Volkswagen	Fox	Small	8.7	9.1	9.5	25	33	…
2	Volkswagen	Eurovan	Van	16.6	19.7	22.7	17	21	…
3	Volkswagen	Passat	Compact	17.6	20.0	22.4	21	30	…
4	Volkswagen	Corrado	Sporty	22.9	23.3	23.7	18	25	…

It doesn't save much for this simple example, but this specification styles is the basis of the `DataFrames` mini language, which provides a standardized and performant means of wrangling a data frame.

Filtering may be done on one or more values. There are different approaches.

For example, to first filter by the manufacturer, then by city mileage, we might write a more complicated predicate function using `&&` to combine Boolean values:

```
pred(r) = r.Manufacturer == "Volkswagen" && r.MPGCity >= 20
filter(pred, cars)
```

	Manufacturer	Model	Type	MinPrice	Price	MaxPrice	MPGCity	MPGHighway	
	Cat…	String	Cat…	Float64	Float64	Float64	Int32	Int32	
1	Volkswagen	Fox	Small	8.7	9.1	9.5	25	33	…
2	Volkswagen	Passat	Compact	17.6	20.0	22.4	21	30	…

The mini language can also be used. On the left hand side of (the first) `=>` a vector of column selectors may be indicated, in which the predicate function (on the right hand side of the first `=>`) will get multiple arguments that can be used to implement the logic:

```
pred(m, mpg) = m == "Volkswagen" && mpg >= 20
filter([:Manufacturer, :MPGCity] => pred, cars)
```

	Manufacturer	Model	Type	MinPrice	Price	MaxPrice	MPGCity	MPGHighway	
	Cat…	String	Cat…	Float64	Float64	Float64	Int32	Int32	
1	Volkswagen	Fox	Small	8.7	9.1	9.5	25	33	…
2	Volkswagen	Passat	Compact	17.6	20.0	22.4	21	30	…

Finally, we could use `filter` twice:

```
cars1 = filter(:Manufacturer => ==("Volkswagen"), cars)
cars2 = filter(:MPGCity => >=(20), cars1)
```

	Manufacturer	Model	Type	MinPrice	Price	MaxPrice	MPGCity	MPGHighway	
	Cat…	String	Cat…	Float64	Float64	Float64	Int32	Int32	
1	Volkswagen	Fox	Small	8.7	9.1	9.5	25	33	…
2	Volkswagen	Passat	Compact	17.6	20.0	22.4	21	30	…

The above required the introduction of an intermediate data frame to store the result of the first `filter` call to pass to the second. This threading through of the modified data is quite common in processing pipelines. The first two approaches with complicated predicate functions can grow unwieldly, so staged modification is common. To support that the chaining or piping operation is often used `|>`:

```
filter(:Manufacturer => ==("Volkswagen"), cars) |>
    d -> filter(:MPGCity => >=(20), d)
```

	Manufacturer	Model	Type	MinPrice	Price	MaxPrice	MPGCity	MPGHighway	
	Cat…	String	Cat…	Float64	Float64	Float64	Int32	Int32	
1	Volkswagen	Fox	Small	8.7	9.1	9.5	25	33	…
2	Volkswagen	Passat	Compact	17.6	20.0	22.4	21	30	…

The right-hand side of `|>` expects a function for application, so an anonymous function is created. There are *macros* available in user-contributed packages that shorten this pattern by using a placeholder.

A popular one is `Chain`, with its basic usage based on two simple principles:

- a new line is a pipe
- a `_` receives the argument from the pipe. If none is given, the first argument does

For example,

```
using Chain
```

```
@chain cars begin
    filter(:Manufacturer => ==("Volkswagen"), _)
    filter(:MPGCity => >=(20), _)
end
```

	Manufacturer	Model	Type	MinPrice	Price	MaxPrice	MPGCity	MPGHighway	
	Cat…	String	Cat…	Float64	Float64	Float64	Int32	Int32	
1	Volkswagen	Fox	Small	8.7	9.1	9.5	25	33	…
2	Volkswagen	Passat	Compact	17.6	20.0	22.4	21	30	…

3.1.5.1 Subset

The calling style `filter(predicate, collection)` is consistently used with several higher-order functions, for example, `map` and `reduce` and is supported by `Julia`'s `do` syntax.

However, with data frames, the convention is to have action functions expect the data frame in the first position.

The `subset` function also returns a copy of the data frame with rows matching a certain selection criteria. It's usage is somewhat similar to `filter` with the order of its arguments reversed, but the predicate must return a vector of indices, so broadcasting or the `ByRow` function may be necessary.

For example, these produce the same data frame:

```
d1 = subset(cars, :Manufacturer => m -> m .== "Volkswagen") # use .== not just ==
d2 = subset(cars, :Manufacturer => ByRow(==("Volkswagen"))) # use `ByRow` to ensure predicate is applied to e
```

	Manufacturer	Model	Type	MinPrice	Price	MaxPrice	MPGCity	MPGHighway	
	Cat…	String	Cat…	Float64	Float64	Float64	Int32	Int32	
1	Volkswagen	Fox	Small	8.7	9.1	9.5	25	33	…
2	Volkswagen	Eurovan	Van	16.6	19.7	22.7	17	21	…
3	Volkswagen	Passat	Compact	17.6	20.0	22.4	21	30	…
4	Volkswagen	Corrado	Sporty	22.9	23.3	23.7	18	25	…

Unlike `filter`, `subset` allows multiple arguments to be applied:

```
subset(cars,
       :Manufacturer => ByRow(==("Volkswagen")),
       :MPGCity => ByRow(>=(20)))
```

	Manufacturer	Model	Type	MinPrice	Price	MaxPrice	MPGCity	MPGHighway	
	Cat…	String	Cat…	Float64	Float64	Float64	Int32	Int32	
1	Volkswagen	Fox	Small	8.7	9.1	9.5	25	33	…
2	Volkswagen	Passat	Compact	17.6	20.0	22.4	21	30	…

The mini language uses a default of `identity` for the function so, if the columns are Boolean, they can be used to subset the data:

```
dd = DataFrame(a = [true, false], b = [true, missing])
subset(dd, :a)  # just first row
```

	a	b
	Bool	Bool?
1	1	1

For data with missing values, like in the `:b` column of `dd`, the comparison operators implement 3-valued logic, meaning the response can be `true`, `false`, *or* `missing`. Using `subset(dd, :b)` above will error. The keyword `skipmissing` can be passed `true` to have these skipped over. The default is false.

```
subset(dd, :b; skipmissing=true)
```

	a	b
	Bool	Bool?
1	1	1

The `dropmissing` function filters out each row that has a `missing` value.

3.1.6 Selecting columns; transforming data

Filtering of a data frame can be viewed as accessing the data in `df` with a boolean set of row indices, as in `df[inds, :]`. (There are flags to have a "view" of the data, as in `df[inds, !]`). Filtering returns a data frame with a possibly reduced number of rows, and the same number of columns.

The `select` function is related in that it can be viewed as passing a set of Boolean vector of column indices to select specific columns or variables, as in `df[:, inds]`. The syntax of `select` adds also extends this allowing transformations of the data and reassignment, as with `df.new_variable = f(other_variables_in_df)`.

The `select` function will return a data frame with the same number of rows (cases) as the original data frame. The variables returned are selected using column selectors. There is support for the `DataFrames` mini language.

First, we can select on or more columns by separating column selectors with commas:

Here, using the backpack data in the variable `d`, we select the first column by column number, the second using a symbol, the third by a string, and the fourth and fifth using a regular expression:[1]

```
select(d, 1, :v, "p", r"^l")
```

	b	v	p	lap	load
	String	Int64	Float64?	Cat…	Cat…
1	Goruck GR2	40	395.99	Y	front panel
2	Minaal 3.0	35	349.99	Y	front panel
3	Genius	25	228.99	Y	clamshell

One use of `select` is to rearrange the column order, say by moving a column to the front. The `:` or `All()` column selector will add the rest. That is this command would move "d" to the front of the other columns: `select(d, :d, All())`.

Selection also allows an easy renaming of column names, using the pairs notation `column name => new column name`:

```
select(d, :v => :volume, "p" => "price")
```

	volume	price
	Int64	Float64?
1	40	395.99
2	35	349.99
3	25	228.99

This usage is a supported abbreviation of `source_column => function => target_column_name`, where the function is `identity`. Here the function is not necessarily a predicate function, as it is with `subset`, but rather a function which returns a vector of the same length as the number of columns in the data frame (as `select` *always* returns the same number of columns as the data frame it is passed.)

The function receives column vectors and should return a column vector. For a scalar function, it must be applied to each entry of the column vector, that is, each row. Broadcasting is one manner to do this. As used previously, the `DataFrames` package provides `ByRow` to also apply a function to each row of the column vector(s).

For example to compute a ratio of price to volume, we might have:

```
select(d, [:p, :v] => ByRow((p,v) -> p/v) => "price to volume")
```

	price to volume
	Float64
1	9.89975
2	9.99971
3	9.1596

The above would be equivalent to `select(d, [:p, :v] => ((p,v) -> p./v) => "price to volume")` (broadcasted `p ./ v` **and** parentheses), but not `select(d, [:p, :v] => (p,v) -> p./v => "price to`

[1] This combination of column selectors is also performed by `Cols(1, :v, "p", r"^l")`.

`volume"`), as the precedence rules for `=>` do not parse the expressions identically. That is, the use of parentheses is around an anonymous function is needed.

The specification of a target column name is not necessary, as one will be generated. In this particular example, it would be `:p_v_function`. For named functions, the automatic namings are a bit more informative, as the method name replaces the "`function`" part. The `renamecols` keyword argument can be set to `false` to drop the addition of a function name, which in this case would leavel `:p_v` as the name. For a single-column selector it would not rename the column, rather replace it.

The above reduced the number of columns to just that specified. The `:` selector will select *all* columns (as it does with indexing) and return them:

```
select(d, [:p, :v] => ByRow((p,v) -> p/v) => "price to volume", :) # return all columns
```

	price to volume	b	v	p	lap	load	d
	Float64	String	Int64	Float64?	Cat…	Cat…	Date
1	9.89975	Goruck GR2	40	395.99	Y	front panel	2022-09-01
2	9.99971	Minaal 3.0	35	349.99	Y	front panel	2022-09-01
3	9.1596	Genius	25	228.99	Y	clamshell	2022-10-01

As seen in this last example, the source column selectors can select more than one column. The function receives multiple arguments. It is possible that there be *multiple* target columns. For example, we could compute both price to volume and volume to price quantities. A function to do so would be

```
pv_and_vp(p,v) = [p/v, v/p]
```

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

When we use this, we see the two values are stored in one column.

```
select(d, [:p, :v] => ByRow(pv_and_vp))
```

	p_v_pv_and_vp
	Array…
1	[9.89975, 0.101013]
2	[9.99971, 0.100003]
3	[9.1596, 0.109175]

To split these up, we can give *two* names:

```
select(d, [:p, :v] => ByRow(pv_and_vp) => [:pv, :vp])
```

	pv	vp
	Float64	Float64
1	9.89975	0.101013
2	9.99971	0.100003
3	9.1596	0.109175

The `AsTable` function has two meanings in the mini language. If on the target side, it assumes the output of the function is a vector of containers (in this example the vectors `[p/v, v/p]`) that should be expanded into multiple columns. It uses the keys (were the containers named tuples) or, in this case generates them, to create multiple columns in the destination:

```
select(d, [:p, :v] => ByRow(pv_and_vp) => AsTable)
```

	x1	x2
	Float64	Float64
1	9.89975	0.101013
2	9.99971	0.100003
3	9.1596	0.109175

Had we used a named tuple in our function, `pv_and_vp(p, v) = (pv = p/v, vp = v/p)`, then the names would come from that.

When `AsTable` is used on the source columns, as in `AsTable([:p,:v])` then the function receives a named tuple with column vector entries. (For this, `pv_and_vp` would be written `pv_and_vp(r) = [r.p/r.v, r.v/r.p]`, with `r` representing a row in a two-column table with names `:p` and `:v`.

3.1.6.1 Transform

Extending the columns in the data frame by `select` is common enough that the function `transform` is supplied which always keeps the columns of the original data frame, though they can also be modified through the mini language. The use of transfrom is equivalent to `select(df, :, args...)`.

> **The DataFrames' mini language**
>
> The mini language is well documented in the documentation string of `transform` and the blog post of Bogumil Kasminski "DataFrames.jl minilanguage explained."

3.1.7 Combine

The `combine` function creates a new data frame whose columns are the result of transformations applied to the source data frame. The name will be more clear after we discuss grouping or splitting of a data set.

A typical usage of combine is to apply some reduction to the data, for example finding an average.

In this example, we use the `mean` function from the `StatsBase` package and apply that to the `:MPGCity` measurements in the `cars` data:

```
combine(cars, :MPGCity => mean)
```

	MPGCity_mean
	Float64
1	22.3656

(A second pair can be used to specify a name for the target, as in `:MPGCity => mean => :AvgMPGCity`)

There are several numeric variables in this data set, we may wish to find the mean of more than 1 at a time. For this, we can broadcast the columns via `.=>`, as in this example:

```
combine(cars, [:MPGCity, :MPGHighway] .=> mean)
```

	MPGCity_mean	MPGHighway_mean
	Float64	Float64
1	22.3656	29.086

(That is an alternate to `combine(cars, :MPGCity => mean, :MPGHighway => mean)`.)

Broadcasting `=>` lends itself to succinctly applying a function to multiple columns.

For example, to apply `mean` to all the numeric values, we might select them first (using the filtering feature of `names`), then pass to the mini language, as follows:

```
nms = names(cars, Real)
combine(cars, nms .=> mean; renamecols=false)
```

	MinPrice	Price	MaxPrice	MPGCity	MPGHighway	EngineSize	Horsepower	RPM	
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64	
1	17.1258	19.5097	21.8989	22.3656	29.086	2.66774	143.828	5280.65	…

More than one transformation at a time is possible, as already seen in the mini language. With `combine` the number of rows is determined by the output of the transformation. In this example, the `mean` reduces to a number, but `unique` reduces `:Origin` to the number of levels. The result of `mean` is repeated, **it is not** the mean for each group (a task we demonstrate shortly):

```
combine(cars, :MPGCity => mean, :Origin => unique)
```

	MPGCity_mean	Origin_unique
	Float64	String
1	22.3656	non-USA
2	22.3656	USA

3.1.8 Flatten

The `repeat` function is used to repeat values in a vector, or other iterable. The simplest usage looks like this:

```
repeat([1,2,3], 2) |> permutedims
```

```
1×6 Matrix{Int64}:
 1  2  3  1  2  3
```

The `2` value is passed to the `outer` argument, and the entire iterable is repeated `2` times. Whereas the `inner` argument repeats each element a certain number of times and combines:

```
repeat([1,2,3], inner=2) |> permutedims
```

```
1×6 Matrix{Int64}:
 1  1  2  2  3  3
```

A common use of repetition is to take two variables, say `x` and `y`, perhaps of unequal length and create a data frame with two columns: one to store the values and one to indicate if a value came from `x` or `y`. This could be achieved like `DataFrame(variable=vcat(repeat([:x], length(x)), repeat([:y], length(y))), values = vcat(x,y))`, but the `flatten` function offers an alternative. Consider the data frame:

```
x = [1,2,3]
y = [4, 5]
d = DataFrame(variable=[:x,:y], value=[x, y])
```

	variable	value
	Symbol	Array…
1	x	[1, 2, 3]
2	y	[4, 5]

The `flatten` function will iterate over the indicated columns and repeat the other variables:

```
flatten(d, :value)
```

	variable	value
	Symbol	Int64
1	x	1
2	x	2
3	x	3
4	y	4
5	y	5

3.1.9 SplitApplyCombine

The split-apply-combine strategy for wrangling data frames was popularized in the influential paper "*The Split-Apply-Combine Strategy for Data Analysis*: [@JSSv040i01] by H. Wickham which introduces this description:"where you break up a big problem into manageable pieces, operate on each piece independently and then put all the pieces back together."

The `groupby` function for data frames does the breaking up, the `combine` function can put things together again, and the `DataFrames` mini language can operate on each piece.

The `groupby` function splits a data frame into pieces of type `GroupedDataFrame`. The basic signature is `groupby(df, cols; sort=nothing, skipmissing=false)`. The `cols` are one or more column selectors. The value of `sort` defaults to `nothing` leaving the order of the groups in the result undefined, but the grouping uses the fastest identified algorithm. The `skipmissing` argument can instruct the skipping of rows which have missing values in the selected columns.

For example, grouping by loading style, returns two sub data frames.

```
gdf = groupby(d, :load)
```

```
LoadError: ArgumentError: column name :load not found in the data frame
```

Indices may be used to get each sub data frame. Moreover, the `keys` method returns keys, like indices, to efficiently look up the different sub data frames in `gdf`. The levels for the keys are found in the property `.load`, as with `keys(gdf)[1].load`. Grouped data frames may be iterated over; the `pairs` iterator iterates over both keys and values.

The `select` and `subset` functions work over `GroupedDataFrame` objects, here we see the returned values are combined:

```
subset(gdf, :p => ByRow(<=(350)))
```

```
LoadError: UndefVarError: gdf not defined
```

(The command `[subset(g, :p => ByRow(<=(350))) for g in gdf]` avoids the `combine` step.)

We can see that `combine` also works over `GroupedDataFrame` objects. This example shows how to find the mean city mileage of groups formed by the value of `:Origin` in the `cars` data set:

```
combine(groupby(cars, :Origin), :MPGCity => mean)
```

	Origin	MPGCity_mean
	Cat…	Float64
1	USA	20.9583
2	non-USA	23.8667

More than one column can be used to group the data. This example first groups by the number of cylinders, then groups by origin. For each of the 9 resulting groups, the group mean for city mileage is taken:

```
combine(groupby(cars, [:Cylinders, :Origin]), :MPGCity => mean)
```

	Cylinders	Origin	MPGCity_mean
	Cat…	Cat…	Float64
1	3	non-USA	39.3333
2	4	USA	24.4091
3	4	non-USA	25.2222
4	5	non-USA	18.5
5	6	USA	18.35
6	6	non-USA	18.5455
7	8	USA	17.0
8	8	non-USA	17.0
9	rotary	non-USA	17.0

The `@chain` macro may make this more readable:

```
@chain cars begin
    groupby([:Cylinders, :Origin])
    combine(:MPGCity => mean)
end
```

	Cylinders	Origin	MPGCity_mean
	Cat…	Cat…	Float64
1	3	non-USA	39.3333
2	4	USA	24.4091
3	4	non-USA	25.2222
4	5	non-USA	18.5
5	6	USA	18.35
6	6	non-USA	18.5455
7	8	USA	17.0
8	8	non-USA	17.0
9	rotary	non-USA	17.0

3.1.9.0.1 Example

For an example, we download a data set from the internet describing all the sets of Legos sold during some time period. This data set is part of the data sets provided by `openintro.org`, an organization trying to make interesting educational products that are free and transparent.

```
csv_data = download("https://www.openintro.org/data/csv/lego_population.csv")
legos = CSV.read(csv_data, DataFrame)
first(legos, 2)
```

	item_number	set_name	theme	pieces	price	amazon_price	year	
	Int64	String	String31	String7	String31	String31	Int64	
1	41916	Extra Dots - Series 2	DOTS	109	3.99	3.44	2020	…
2	41908	Extra Dots - Series 1	DOTS	109	3.99	3.99	2020	…

Let's explore this data.

The size of the data set is printed, though suppressed above as only the first 2 rows are requested to be show, or can be programmatically identified with `size(legos)`.

```
size(legos)
```

```
(1304, 14)
```

There are 1284 different products, and 9 measures for each product.

The types *identified* by `CSV.read` are not perfect. The data uses `NA` for not-available. We read again using `missingstring="NA"`:

```
legos = CSV.read(csv_data, DataFrame; missingstring="NA")
first(legos, 2)  # show first 2 rows
```

	item_number	set_name	theme	pieces	price	amazon_price	year	
	Int64	String	String31?	Int64?	Float64?	Float64?	Int64	
1	41916	Extra Dots - Series 2	DOTS	109	3.99	3.44	2020	…
2	41908	Extra Dots - Series 1	DOTS	109	3.99	3.99	2020	…

The `:theme` and `:packaging` variables are categorical, so we make them so:

```
legos.theme = categorical(legos.theme)
legos.packaging = categorical(legos.packaging);
first(legos, 2)
```

	item_number	set_name	theme	pieces	price	amazon_price	year	
	Int64	String	Cat…?	Int64?	Float64?	Float64?	Int64	
1	41916	Extra Dots - Series 2	DOTS	109	3.99	3.44	2020	…
2	41908	Extra Dots - Series 1	DOTS	109	3.99	3.99	2020	…

We see data on the number of pieces and the age range. Is there some relationship?

The `:age` variable should be an ordered factor, ordered by the youngest age intended for use. The `:ages` variable has a pattern `Ages_startXXX` where `XXX` may be `+` to indicate or up, or a dash to indicate a range. We use this to identify the youngest intended age.

```
# alternative to
# `(m = match(r"Ages_(\d+)", x); m === nothing ? missing : parse(Int, m.captures[1]))`
function pick_first_age(x)
    ismissing(x) && return missing
    nos = collect(string(0:9...))
    out = ""
    for xi in x[6:end] # drop Ages_
        !(xi in nos) && break
        out = out * xi
    end
    out == "" && return missing
    parse(Int, out)
end


transform!(legos, :ages => ByRow(pick_first_age) => :youngest_age)
legos.youngest_age = categorical(legos.youngest_age, ordered=true)
first(legos[:,r"age"], 2)
```

	ages	pages	youngest_age
	String15	Int64?	Cat…?
1	Ages_6+	*missing*	6
2	Ages_6+	*missing*	6

With that ordering, an expected pattern becomes clear – kits for older users have on average more pieces – though there are unexpected exceptions:

```
@chain legos begin
    groupby(:youngest_age)
    combine([:pieces, :unique_pieces] .=> mean∘skipmissing
            .=> [:avg_pieces, :avg_unique_pieces])
end
```

	youngest_age	avg_pieces	avg_unique_pieces
	Cat…?	Float64	Float64
1	*missing*	127.592	40.5604
2	1	37.2083	26.2174
3	2	55.617	31.2979
4	3	140.0	41.0
5	4	194.74	87.2857
6	5	147.712	85.1557
7	6	191.079	87.1289
8	7	265.556	108.816
9	8	491.193	173.801
10	9	859.182	252.773
11	10	417.52	109.768
12	11	2357.86	214.143
13	12	1307.29	256.4
14	14	1617.25	349.5
15	16	2454.79	409.107
16	18	2073.84	214.294

(We composed the two functions `skipmissing` with `mean` using the `\circ[tab]` operator and use `.=>` in both places to avoid defining the transformation function twice.)

> **The Lego Group**
>
> Lego, was the largest toy company in the world by 2021. By 2015 it had produced over 600 billion parts.

Lego is a mature company with an origin dating to 1934. The number of products per year should be fairly stable, though it is easy enough to check. The data set has a `:year` variable, so we would only need to group the data by that, then count the number of cases per each group. The `nrow` function will count the cases. This function is special cased in the `DataFrames`' mini language and can appear by itself:

```
combine(groupby(legos, :year), nrow) # use `nrow => :n`, or some such, to label differently
```

	year	nrow
	Int64	Int64
1	2018	442
2	2019	423
3	2020	439

We can take other summaries by year, for example, here we find the average price by year and size:

```
@chain legos begin
    groupby([:year, :size])
    combine(nrow => :n,
            :price => mean∘skipmissing => :avg_price
            )
end
```

	year	size	n	avg_price
	Int64	String7?	Int64	Float64
1	2018	Small	324	45.7792
2	2018	Large	21	28.59
3	2018	*missing*	97	25.8135
4	2019	Small	322	50.4663
5	2019	Large	14	29.6329
6	2019	*missing*	87	13.8536
7	2020	Small	335	49.8104
8	2020	Large	18	33.5456
9	2020	*missing*	86	21.6959

The youngest age range is a bit long. We wish to collapse it to the ranges 1-6, 7-12, and 12-18. The data has been stored as a categorical array. The `cut` function can group numeric data easily, but to group categorical data, the cut-ranges must be promoted to a `CategoricalValue`. We have the following, where `Ref` is used to stop the broadcasting for that value:

```
vals = CategoricalValue.([1, 6, 12, 18], Ref(legos.youngest_age))
transform!(legos,
           :youngest_age => (x ->cut(x, vals; extend=true))
           => :youngest_age_range);
```

We now check if the number of pieces has dramatically changed over the year. We break up the data by the age range, as we expect variation in that value so it is best to dis-aggregate over that factor:

```
@chain legos begin
    groupby([:youngest_age_range, :year])
    combine(nrow => :n,
            :pieces => mean∘skipmissing => :avg_pieces)
```

```
end
```

	youngest_age_range	year	n	avg_pieces
	Cat...?	Int64	Int64	Float64
1	*missing*	2018	33	165.379
2	*missing*	2019	34	95.7778
3	*missing*	2020	44	111.8
4	[1, 6)	2018	98	131.561
5	[1, 6)	2019	92	122.533
6	[1, 6)	2020	92	153.413
7	[6, 12)	2018	292	341.866
8	[6, 12)	2019	275	389.945
9	[6, 12)	2020	276	365.887
10	[12, 18]	2018	19	1956.21
11	[12, 18]	2019	22	1938.95
12	[12, 18]	2020	27	2111.3

There are many different themes. The command `unique(legos.theme)` shows 41. Which are the most popular? To see, the data is grouped by the theme, each group is counted, missing themes are dropped, then the data frame is sorted and the top 5 rows are shown:

```
@chain legos begin
    groupby(:theme)
    combine(nrow => :n)
    dropmissing
    sort(:n; rev=true)
    first(5)
end
```

	theme	n
	Cat...	Int64
1	Star Wars™	119
2	Friends	103
3	City	101
4	NINJAGO®	78
5	DUPLO®	53

3.1.9.1 The SplitApplyCombine package

There is support for the split-apply-combine paradigm outside of the `DataFrames` package in the package `SplitApplyCombine`. This package is much lighter weight than `DataFrames`. This support targets other representations of tabular data, such as a vector of nested tuples:

```
using SplitApplyCombine
tbl = [
(b = "Goruck GR2", v = 40, p = 395, lap = "Y", load = "front panel", d = Date("2022-09-01")),
```

```
  (b = "Minaal 3.0", v = 35, p = 349, lap = "Y", load = "front panel", d = Date("2022-09-01")),
  (b = "Genius",     v = 25, p = 228, lap = "Y", load = "clamshell",   d = Date("2022-10-01"))
  ]
```

```
3-element Vector{NamedTuple{(:b, :v, :p, :lap, :load, :d), Tuple{String, Int64, Int64, String, String, Date}}}:
 (b = "Goruck GR2", v = 40, p = 395, lap = "Y", load = "front panel", d = Date("2022-09-01"))
 (b = "Minaal 3.0", v = 35, p = 349, lap = "Y", load = "front panel", d = Date("2022-09-01"))
 (b = "Genius", v = 25, p = 228, lap = "Y", load = "clamshell", d = Date("2022-10-01"))
```

Many of the main verbs are generic functions `filter`, `map`, `reduce`, `invert`; `group` does grouping. There are some others. A few examples follow.

For indexing this structure, we can't not index by `[row, col]`, rather we can use `[row][col]` notation, the first to extract the row, the second to access the entries in the column. For example, he we index a selected row by position, by name, and then with multiple names:

```
tbl[2][2], tbl[2].v, tbl[2][ [:v, :p] ]
```

```
(35, 35, (v = 35, p = 349))
```

The `invert` function reverses the access pattern, allowing in this case, `[col][row]` access, with:

```
itbl = invert(tbl)
itbl[3][1], itbl[:p][2]
```

```
(395, 349)
```

To filter out rows, we have the same calling style as with data frames: `filter(pred, tbl)`. For example,

```
filter(r -> r.v >= 35, tbl) |> DataFrame
```

	b	v	p	lap	load	d
	String	Int64	Int64	String	String	Date
1	Goruck GR2	40	395	Y	front panel	2022-09-01
2	Minaal 3.0	35	349	Y	front panel	2022-09-01

(The call to `DataFrame` is to take advantage of the prettier printing. `DataFrame` can consume a vector of named tuples for its input.)

The `DataPipes` package composes a bit more nicely with this approach, but we continue using `Chain` for examples[2], like this one of nested calls to `filter`.

```
@chain tbl begin
    filter(r -> r.v >= 35,  _)
```

[2]The `DataPipes` package inserts a magical `_` in the last positional argument, which is common with higher-order functions like `filter`. The example could become `@p tbl |> filter(_.v >= 25) |> filter(_.p >= 350)` with the new lines replacing the pipe operation as a convenience with longer chains

```
    filter(r -> r.p >= 350, _)
    DataFrame
end
```

	b	v	p	lap	load	d
	String	Int64	Int64	String	String	Date
1	Goruck GR2	40	395	Y	front panel	2022-09-01

Selecting columns and adding a transformation can be achieved through `map`, which iterates over each named tuple in this example:

```
map(r -> (b=r.b, v=r.v, p=r.p, pv = round(r.p/r.v; digits=2)), tbl) |>
    DataFrame
```

	b	v	p	pv
	String	Int64	Int64	Float64
1	Goruck GR2	40	395	9.88
2	Minaal 3.0	35	349	9.97
3	Genius	25	228	9.12

The `group` function can group *by* values of a function call. This example from the package's Readme is instructive:

```
nms = ["Andrew Smith", "John Smith", "Alice Baker",
       "Robert Baker", "Jane Smith", "Jason Bourne"]
group(last, split.(nms))
```

```
3-element Dictionaries.Dictionary{SubString{String}, Vector{Vector{SubString{String}}}}
  "Smith" │ Vector{SubString{String}}[["Andrew", "Smith"], ["John", "Smith"], […
  "Baker" │ Vector{SubString{String}}[["Alice", "Baker"], ["Robert", "Baker"]]
 "Bourne" │ Vector{SubString{String}}[["Jason", "Bourne"]]
```

The names are split on spaces, and the keys of the group are determined by the `last` function, which here gets the last piece after splitting (aka the "last" name, but by coincidence, not intelligence).

For numeric operations, this is quite convenient. Here we find group by the remainder upon division by 3:

```
group(x -> rem(x,3), 1:10)
```

```
3-element Dictionaries.Dictionary{Int64, Vector{Int64}}
 1 │ [1, 4, 7, 10]
 2 │ [2, 5, 8]
 0 │ [3, 6, 9]
```

The grouping basically creates a dictionary and adds to the key determined by the function.

To apply a function to the values of a base `Julia` dictionary is not directly supported by `map`, however the dictionary used by `group` (from the `Dictionaries` packages) does allow this, so we could, for example, add

up the values in each group with:

```
map(sum, group(x -> rem(x,3), 1:10))
```

```
3-element Dictionaries.Dictionary{Int64, Int64}
 1 │ 22
 2 │ 15
 0 │ 18
```

The `group` function can do this in one call, by placing the function to apply in between the `by` function and the table. Here we apply `first` to pick off the first name. That is, `["Andrew", "Smith"]` becomes just `"Andrew"`, the first element of that vector.

```
gps = group(last, first, split.(nms))
```

```
3-element Dictionaries.Dictionary{SubString{String}, Vector{SubString{String}}}
  "Smith" │ SubString{String}["Andrew", "John", "Jane"]
  "Baker" │ SubString{String}["Alice", "Robert"]
 "Bourne" │ SubString{String}["Jason"]
```

Example 3.1 (A tally function)**.** The `Tally` package provides a quick way to tally up numbers. Here we do a simplified version.

To tally, we have three steps: group by each by value, count the number in each groups, sort the values. This is implemented with:

```
tally(v; by=identity) = SplitApplyCombine.sortkeys(map(length, group(by, v)))
```

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

To see, we have:

```
v = rand(1:5, 50)
d = tally(v)
```

```
5-element Dictionaries.Dictionary{Int64, Int64}
 1 │ 10
 2 │ 8
 3 │ 13
 4 │ 8
 5 │ 11
```

Using `map`, we can provide an alternate, oft used view for tallying:

```
tallies = "\u007C"^4
ftally = "𝍸 "
function prison_count(x)
    d, r = divrem(x, 5)
    ftally^d * tallies[1:r]
end


map(prison_count, d)
```

```
5-element Dictionaries.Dictionary{Int64, Any}
 1 │ "𝍸 𝍸 "
 2 │ "𝍸 |||"
 3 │ "𝍸 𝍸 |||"
 4 │ "𝍸 |||"
 5 │ "𝍸 𝍸 |"
```

Let's return to the `cars` data and use `group` to identify the average mileage per type. For this task, we group by `:Type`and then apply a function to extract the mileage from a row. The `copy.(eachrow(cars))` command creates a vector of named tuples.

```
cs = copy.(eachrow(cars))
gps = group(r -> r.Type, r -> r.MPGCity, cs)
```

```
6-element Dictionaries.Dictionary{CategoricalValue{String, UInt8}, Vector{Int32}}
 CategoricalValue{String, UInt8} "Smal… │ Int32[25, 29, 23, 29, 31, 23, 46, 42,…
 CategoricalValue{String, UInt8} "Mids… │ Int32[18, 19, 22, 22, 19, 16, 21, 21,…
 CategoricalValue{String, UInt8} "Comp… │ Int32[20, 25, 25, 23, 22, 22, 24, 26,…
 CategoricalValue{String, UInt8} "Larg… │ Int32[19, 16, 16, 17, 20, 20, 20, 18,…
 CategoricalValue{String, UInt8} "Spor… │ Int32[19, 17, 18, 22, 24, 30, 24, 26,…
  CategoricalValue{String, UInt8} "Van" │ Int32[18, 15, 17, 15, 18, 17, 18, 18,…
```

We now can map `mean` over each group:

```
ms = map(mean, gps)
```

```
6-element Dictionaries.Dictionary{CategoricalValue{String, UInt8}, Float64}
   CategoricalValue{String, UInt8} "Small" │ 29.857142857142858
 CategoricalValue{String, UInt8} "Midsize" │ 19.545454545454547
 CategoricalValue{String, UInt8} "Compact" │ 22.6875
   CategoricalValue{String, UInt8} "Large" │ 18.363636363636363
  CategoricalValue{String, UInt8} "Sporty" │ 21.785714285714285
     CategoricalValue{String, UInt8} "Van" │ 17.0
```

This requires two passes through the grouped data, the first to get just the mileage values, the next to call `mean`. The `DataFrames` syntax hides this:

```
combine(groupby(cars, :Type), :MPGHighway => mean)
```

	Type	MPGHighway_mean
	Cat…	Float64
1	Compact	29.875
2	Large	26.7273
3	Midsize	26.7273
4	Small	35.4762
5	Sporty	28.7857
6	Van	21.8889

Reductions, like sum and count which can be written easily using `reduce` have support to combine the grouping and reduction. The `groupreduce` function takes the additional arguments of `reduce` (an *op*eration and an optional `init` value). For example, go get the group sum, we could call:

```
groupreduce(r -> r.Type, r -> r.MPGCity, +, cs)
```

```
6-element Dictionaries.Dictionary{CategoricalValue{String, UInt8}, Int32}
   CategoricalValue{String, UInt8} "Small" │ 627
 CategoricalValue{String, UInt8} "Midsize" │ 430
 CategoricalValue{String, UInt8} "Compact" │ 363
   CategoricalValue{String, UInt8} "Large" │ 202
  CategoricalValue{String, UInt8} "Sporty" │ 305
     CategoricalValue{String, UInt8} "Van" │ 153
```

The group sum and group count have shortcuts, so the mean could have been computed as:

```
groupsum(r -> r.Type, r -> r.MPGCity, cs) ./ groupcount(r -> r.Type, cs)
```

```
6-element Dictionaries.Dictionary{CategoricalValue{String, UInt8}, Float64}
   CategoricalValue{String, UInt8} "Small" │ 29.857142857142858
 CategoricalValue{String, UInt8} "Midsize" │ 19.545454545454547
 CategoricalValue{String, UInt8} "Compact" │ 22.6875
   CategoricalValue{String, UInt8} "Large" │ 18.363636363636363
  CategoricalValue{String, UInt8} "Sporty" │ 21.785714285714285
     CategoricalValue{String, UInt8} "Van" │ 17.0
```

(Unlike for a `Dict` instance, the dictionaries above allow such division.)

The "combine" part of the paradigm takes the pieces and reassembles. The data frames example returns a data frame, whereas with `SplitApplyCombine.jl` we have a dictionary. A dictionary does not map directly

to a data frame. While both dictionaries and named tuples are associative arrays, we wouldn't necessarily want to map our data to a row. Rather, here we use the `pairs` iterator to iterate over the keys and values to produce an array of named tuples:

```
[(; Type=k, MPG=v) for (k,v) in pairs(ms)] |> DataFrame
```

	Type	MPG
	Cat…	Float64
1	Small	29.8571
2	Midsize	19.5455
3	Compact	22.6875
4	Large	18.3636
5	Sporty	21.7857
6	Van	17.0

3.1.10 Tidy data

The split-apply-combine paradigm suggest that storing data so that it can be split readily is preferable to other ways of storage. The notion of "tidy data," [@tidy-data] as presented in R's `tidyr` package and described in this vignette suggests data sets should follow:

- Every column is a *variable*.
- Every row is an *observation*.
- Every cell is a single *value*.

These terms are defined by

> A dataset is a collection of values, usually either numbers (if quantitative) or strings (if qualitative). Values are organised in two ways. Every value belongs to a variable and an observation. A variable contains all values that measure the same underlying attribute (like height, temperature, duration) across units. An observation contains all values measured on the same unit (like a person, or a day, or a race) across attributes.

The `DataFrames` package provides the functions `stack` and `unstack` for tidying data. We illusrate with an abbreviated version of an example in the `tidyr` vignette.

Consider this data from the Global Historical Climatology Network for one weather station (MX17004) in Mexico for five months in 2010. The full data has 31 days per month, this abbreviated data works with just the first 8:

```
weather = """
"id","year","month","element","d1","d2","d3","d4","d5","d6","d7","d8"
"MX17004","2010","1","tmax","---","---","---","---","---","---","---","---"
"MX17004","2010","1","tmin","---","---","---","---","---","---","---","---"
"MX17004","2010","2","tmax","---","27.3","24.1","---","---","---","---","---"
"MX17004","2010","2","tmin","---","14.4","14.4","---","---","---","---","---"
"MX17004","2010","3","tmax","---","---","---","---","32.1","---","---","---"
"MX17004","2010","3","tmin","---","---","---","---","14.2","---","---","---"
```

```
"MX17004","2010","4","tmax","---","---","---","---","---","---","---","---"
"MX17004","2010","4","tmin","---","---","---","---","---","---","---","---"
"MX17004","2010","5","tmax","---","---","---","---","---","---","---","---"
"MX17004","2010","5","tmin","---","---","---","---","---","---","---","---"
"""

w = CSV.read(IOBuffer(weather), DataFrame; missingstring="---")
first(w, 4)
```

	id	year	month	element	d1	d2	d3	d4	d5	
	String7	Int64	Int64	String7	Missing	Float64?	Float64?	Missing	Float64?	
1	MX17004	2010	1	tmax	*missing*	*missing*	*missing*	*missing*	*missing*	…
2	MX17004	2010	1	tmin	*missing*	*missing*	*missing*	*missing*	*missing*	…
3	MX17004	2010	2	tmax	*missing*	27.3	24.1	*missing*	*missing*	…
4	MX17004	2010	2	tmin	*missing*	14.4	14.4	*missing*	*missing*	…

This example is said to have variables stored in both rows and columns. The first step to tidying the data is to stack days 1 through 8 into a column. We use the `Between` column selector to select the columns for `d1` though `d8`:

```
w1 = @chain w begin
    stack(Between(:d1, :d8); variable_name = :day)
    filter(:value => !ismissing, _)
end
```

	id	year	month	element	day	value
	String7	Int64	Int64	String7	String	Float64?
1	MX17004	2010	2	tmax	d2	27.3
2	MX17004	2010	2	tmin	d2	14.4
3	MX17004	2010	2	tmax	d3	24.1
4	MX17004	2010	2	tmin	d3	14.4
5	MX17004	2010	3	tmax	d5	32.1
6	MX17004	2010	3	tmin	d5	14.2

As in the example being followed, missing values are skipped, requiring the reader to imply the lack of measurement on a given day.

The day is coded with a leading "d" and as a string, not an integer. We can strip this prefix out using a regular expression or, as here, string indexing then parse the result to an integer:

```
transform!(w1, :day => ByRow(x -> parse(Int, x[2:end])) => :day)
```

	id	year	month	element	day	value
	String7	Int64	Int64	String7	Int64	Float64?
1	MX17004	2010	2	tmax	2	27.3
2	MX17004	2010	2	tmin	2	14.4
3	MX17004	2010	2	tmax	3	24.1
4	MX17004	2010	2	tmin	3	14.4
5	MX17004	2010	3	tmax	5	32.1
6	MX17004	2010	3	tmin	5	14.2

Finally, the `element` column does not hold a variable, rather it stores the names of two variables (`tmin` and `tmax`). For this the data is "unstacked" splitting the longer variable `value` in two:

```
w2 = unstack(w1, :element, :value)
```

	id	year	month	day	tmax	tmin
	String7	Int64	Int64	Int64	Float64?	Float64?
1	MX17004	2010	2	2	27.3	14.4
2	MX17004	2010	2	3	24.1	14.4
3	MX17004	2010	3	5	32.1	14.2

As described in the vignette, now each variable is in one column, and each row describes one day.

Chapter 4

Bivariate data

We now consider two variables measured for the same observation, or bivariate data. There are three basic interpretations depending on the type of data.

In the following, we use the widely used `iris` data set which has measurements of sepal and petal width and lengths for 3 species of irises.

```
using StatsBase, StatsPlots,
    RDatasets, DataFrames, Chain, CategoricalArrays

iris = dataset("datasets", "iris")
first(iris, 3)
```

	SepalLength	SepalWidth	PetalLength	PetalWidth	Species
	Float64	Float64	Float64	Float64	Cat…
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa

There are 50 cases per species:

```
combine(groupby(iris, :Species), nrow)
```

	Species	nrow
	Cat…	Int64
1	setosa	50
2	versicolor	50
3	virginica	50

4.1 Independent samples

First we graphically explore one measurement, `PetalWidth`, for the 3 different species. A typical question might be, is the average petal width the same across the three species? Other comparative measures might include the variability or the shape of the distribution.

A common comparison is when data for a measurement is collected across multiple levels of a factor to see if there is something about that factor that has an effect on the measurement. As measurements typically have fluctuations, this "effect" must be understood relative to some understanding of the underlying variability. To gather that, a probability model might be used. For now, we content ourselves with common graphical comparisons, which can be used to distinguish large effects.

The data storage in `iris` is tidy data, each row is an observation, the `PetalWidth` is the variable. The graphics are done by splitting and then applying the graphic to the different groups of data. For this task, the `StatsPlots` package is useful.[1]

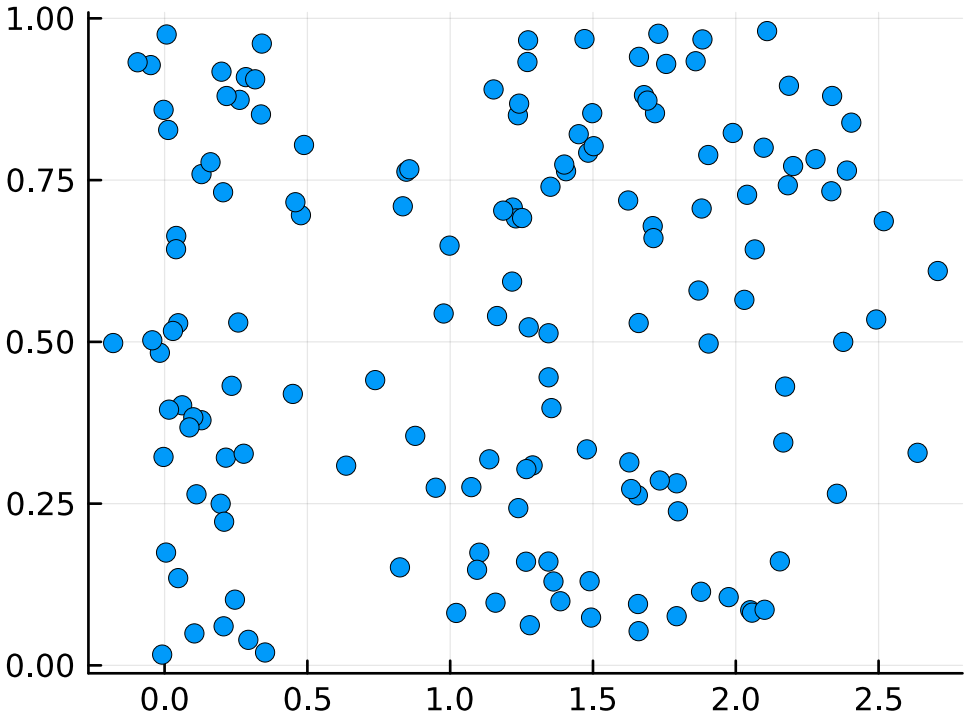
4.1.1 Dot plots

A dot plot is a useful graphic to show the distribution of values on a number line when the data is sufficiently spread out so as not to overlap too much, though there are strategies to jitter values or stack values. A basic dot plot in `StatsPlots` is a scatter plot, where we specify the x and y positions of the data to be plotted. In this case, we create y values:

```
xs = iris.PetalWidth
ys = [rand() for _ in xs] # jitter the data
dotplot(xs, ys; legend=false)
```

[1] For the `Makie` plotting package, the `AlgebraOfGraphics` package is a powerful alternative to `StatsPlots` and reminiscent of R's `ggplot` system. We will illustrate its use in a later chapter.

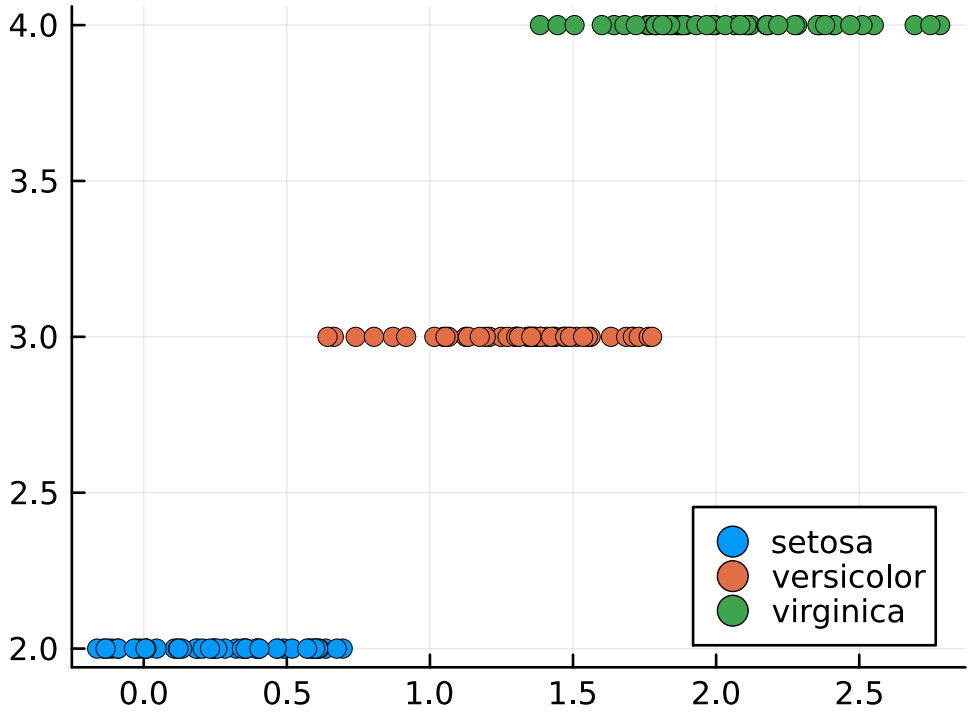
Figure 4.1: Simple dotplot with no separation by species



We can also create plots by each species:

```
byspecies = groupby(iris, :Species)
p = plot()
for (i, k) in enumerate(keys(byspecies)) # iterate over keys
    species = k.Species
    xs = byspecies[k].PetalWidth
    ys = i .+ ones(length(xs))
    dotplot!(p, xs, ys; label=species)
end
p
```

Figure 4.2: A dot plot with each species presented a different y value



The identification of the species after grouping is a bit awkward, but the idea is we split the data, then create a dot plot for each level of the grouping variable, `:Species`. This is more easily done by passing `:Species` as the y value, in which case the categorical variable is used for grouping. Here we use the convenient `@df` macro which looks up the values for each symbol within the data frame specified.

```
p1 = @df iris dotplot(:PetalWidth, :Species; legend=false);
```

Regardless of how the graphic is produced, there appears to be a difference in the centers based on the species, as would be expected – different species have different sizes.

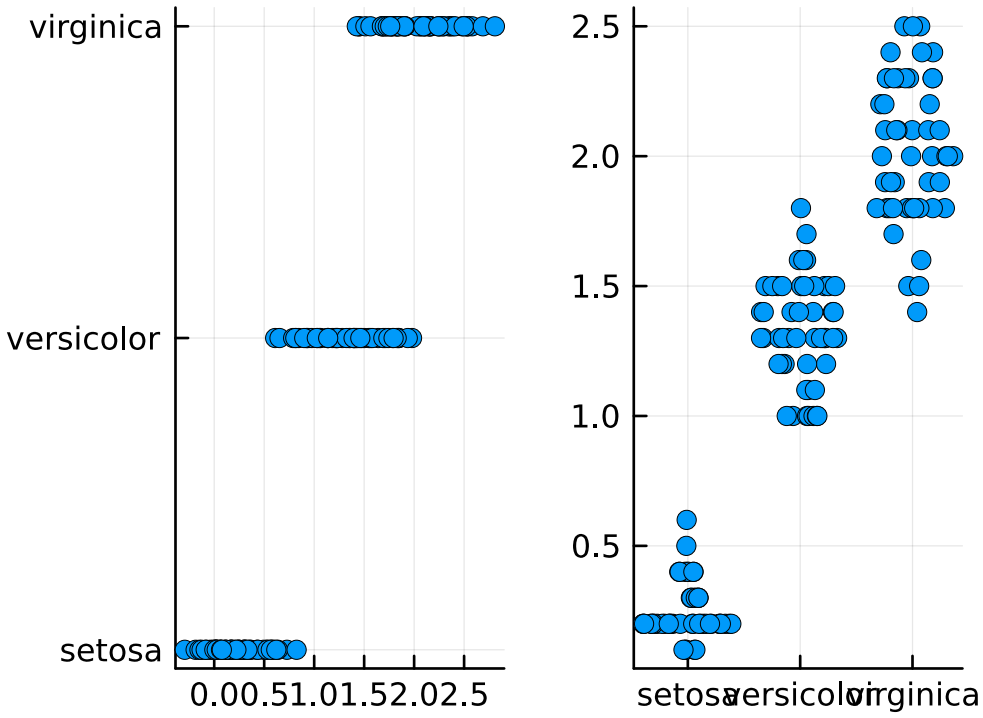
Putting the categorical variable first, presents a graphic which uses jittering to disambiguate equal measurements in a species:

```
p2 = @df iris dotplot(:Species, :PetalWidth; legend=false);
```

4.1.2 Boxplots

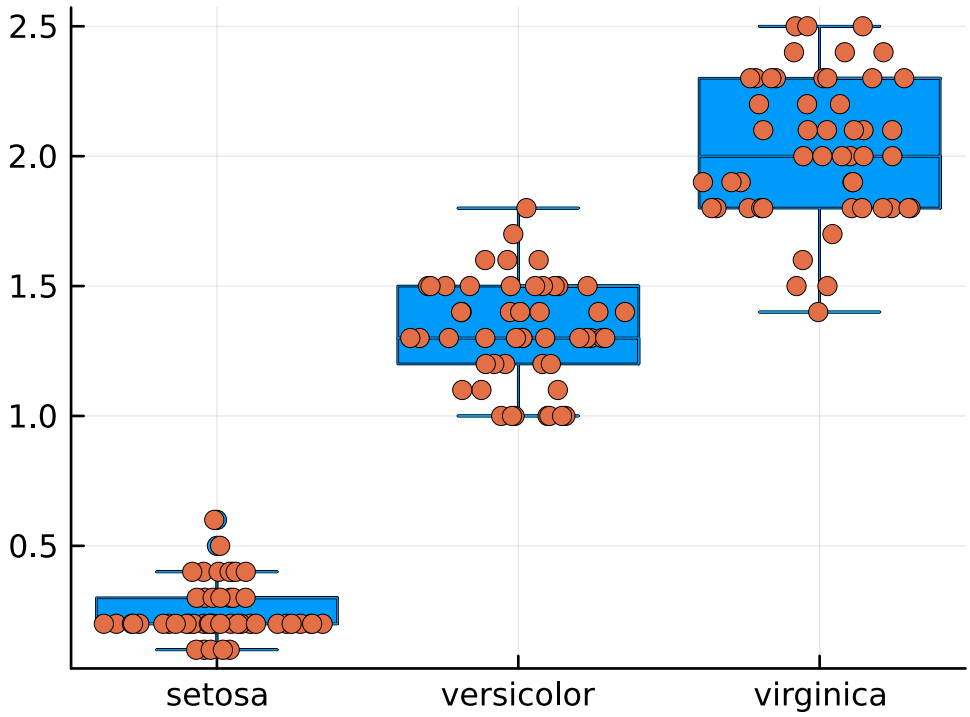
Boxplots are made in a similar manner to the last example using `boxplot`.

Figure 4.3: The left graphic shows a dot plot with groups split by species presented horizonatally. The right one a dot plot with groups split by species presented verticall. Jittering is used by defaut to disambiguate ties. The x axis represents categories, it is not a numeric scale.



```
p1 = @df iris boxplot(:Species, :PetalWidth; legend=false)
@df iris dotplot!(p1, :Species, :PetalWidth; legend=false)
```

Figure 4.4: Boxplots of `PetalWidth` displayed vertically, one for each species level.



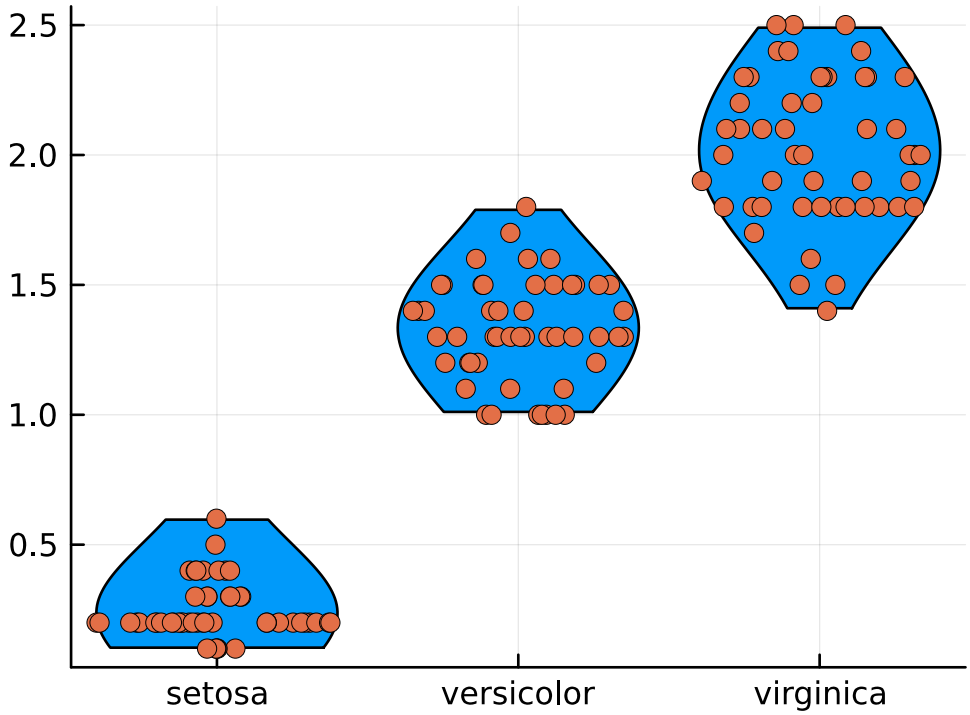
The boxplot shows the quartiles. From this graphic the difference in the medians (the middle line) for the different species is quite apparent. The IQR is the height of the box, we can also see a difference in variation over the species, with "setosa" being less variable than "virginica."

4.1.3 Density plots

A violin plot is like the boxplot, only it is wider where more data is concentrated, and smaller where less is. The `violin` plot creates these:

```
p2 = @df iris violin(:Species, :PetalWidth; legend=false)
@df iris dotplot!(p2, :Species, :PetalWidth; legend=false)
```

Figure 4.5: Violin plots of `PetalWidth` displayed vertically, one for each species level.



The shape of the edges follows from a *density estimate*. A density plot, like a histogram, shows the distribution of values in a way that the center of the distribution is judged by either the balance point (the mean) or the value that splits the area (the median). The spread (IQR) is judged by the range of the middle half of the area.

The `density` function makes a density plot.

As seen below, the grouping variable is a named, not positional argument. In Figure 4.7 the left plot is done without grouping, the right one with grouping by species. The left plot shows a *multi-modal* distribution, which suggests, as we already have seen, that there is a mixture of populations in the data. There appear to be two modes, with a hint of a third. The separation between the "setosa" species and the other two is greater than the separation between the two others.

Ode to the boxplots

The box plot is a great graphic for comparing values across the levels of some factor. Each boxplot shows the center (the median), the range (the IQR), the shape (well basic symmetry about the median), the outlying values (with its identying rule of 1.5 IQRs beyond Q_1 and Q_3. The only drawback is boxplots can't show multi modal data. But if the factor is the cause of that, then the collection of

Figure 4.6: Illustration of boxplot (with dotplot overlay) and violin plot.

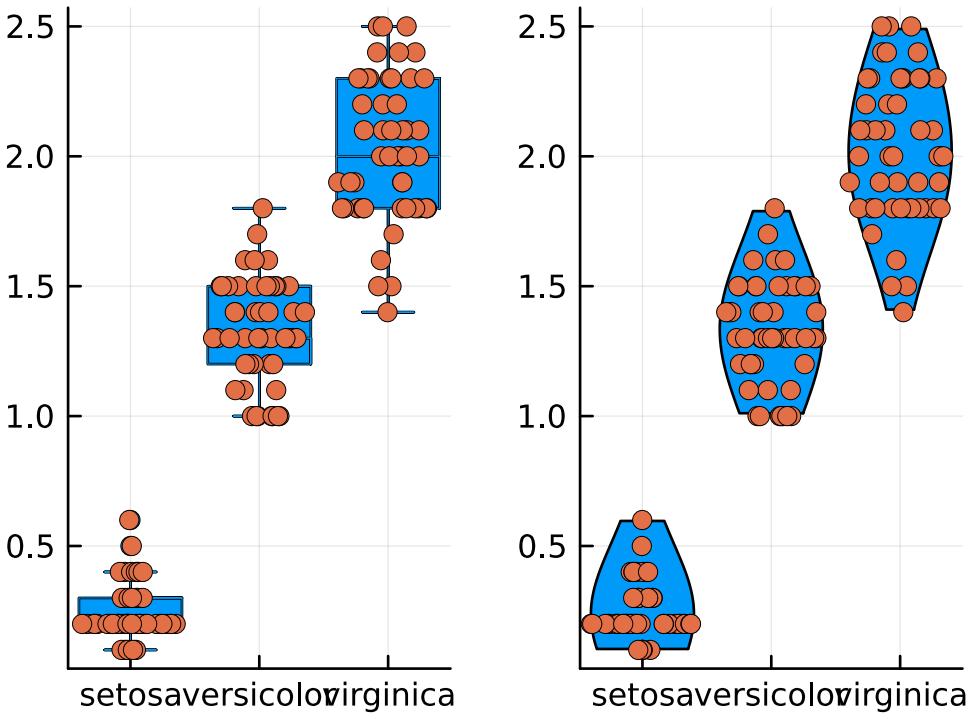
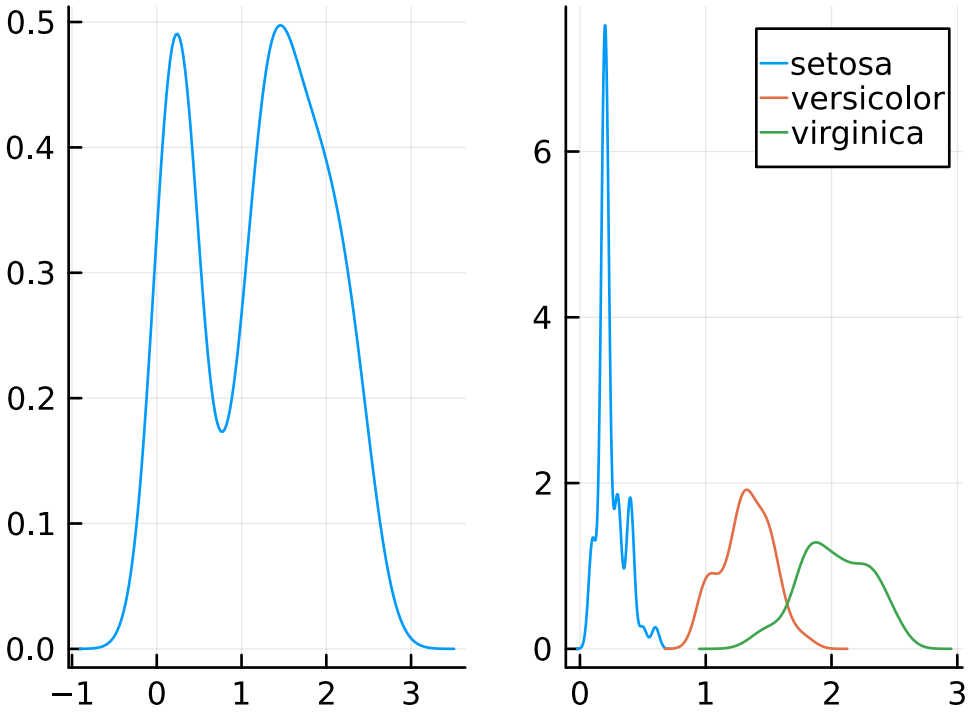


Figure 4.7: Density plots of `PetalWidth` and `PetalWidth` by `Species`. The left graphic suggests multiple modes, which are revealed in the right graphic.



boxplots is great for comparison, as in Figure 4.6.

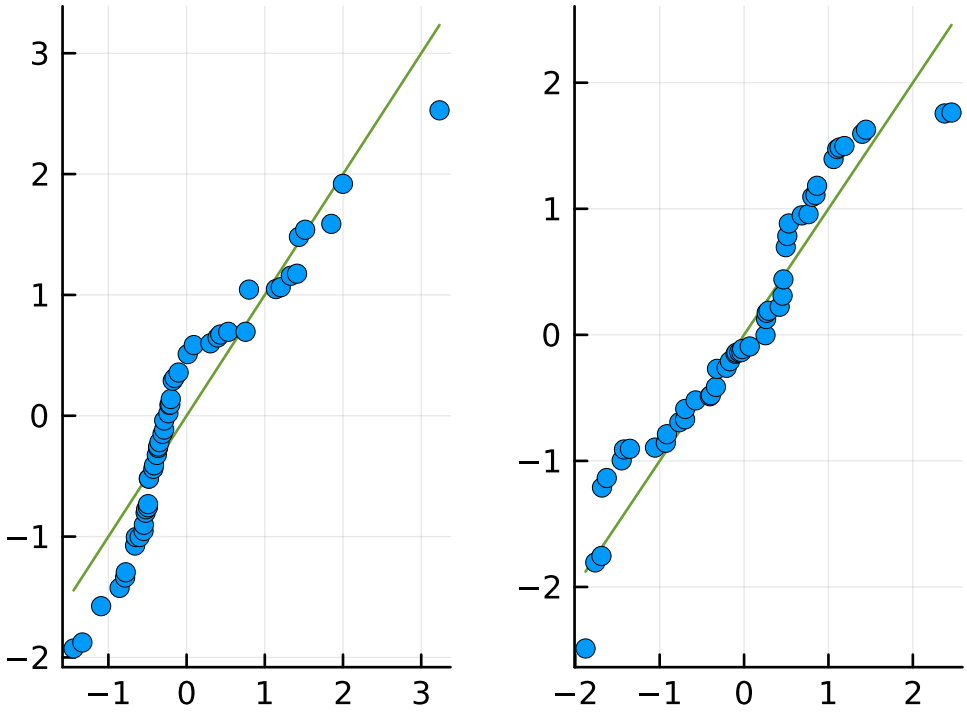
4.1.4 Quantile comparisons

A "quantile-quantile plot" plots pairs of quantiles from two data sets as a scatter plot to gauge is the shape of the two distributions is similar. If it is, the data will fall roughly on a straight line. The quantile-normal plot is identical with one of the distributions being the reference normal distribution.

We use this plot to check if the distribution for two species is roughly the same. To do so we extract the `PetalWidth` values for the different levels of `:Species`. A direct way might be indexing: `iris[iris.Species .== "versicolor", :PetalWidth]` or with `subset`: `subset(iris, :Species => ByRow(==("versicolor")))).PetalWidth`. In the following we group by `:Species` then for each group apply a z-score to `:PetalWidth`. From here, we can compare the 1st and 2nd and then the 2nd and 3rd group with `qqplot` in Figure 4.8. There is not strong reason to expect big differences in shape.

```
gps = [select(gp, :PetalWidth => zscore => :PetalWidth) for gp in groupby(iris, :Species)]
jitter(x, λ = 0.25) = x + λ * randn(length(x))

p1 = qqplot(jitter(gps[1].PetalWidth), jitter(gps[2].PetalWidth))
p2 = qqplot(jitter(gps[2].PetalWidth), jitter(gps[3].PetalWidth))
plot(p1, p2, layout = (@layout [a b]))
```

Figure 4.8: A `qqplot` of the petal width from the versicolor and virginica species. The graphic suggests similar-shaped distributions. The data is jittered to disambiguate identical values.



4.2 Paired data

We have looked at a single variable over several levels of a given factor. If the data is split along the factor, there are possibly a different number of measurements for the given level. A key statistical assumption for inference will be that these different measurements are *independent*.

We might express this data as

$$\begin{aligned} &x_{11}, x_{12}, \ldots, x_{1n_1} \\ &x_{21}, x_{22}, \ldots, x_{2n_2} \\ &\vdots \\ &x_{m1}, x_{m2}, \ldots, x_{mn_m}. \end{aligned}$$

Now we turn our attention to data of two measured variables for a given observation. These two variables must have the same number of values, this being equal to the number of observations. Moreover, it need not be the case that the two measurements should be "independent," indeed, the measurements are from

the same case, so, for example with the iris data, if a plant has large petal width it will likely also have large petal length.

We might express the data mathematically as:

$$x_1, x_2, \ldots, x_n$$
$$y_1, y_2, \ldots, y_n$$

Or – to emphasize how the data is paired off – as $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.

4.2.1 Numeric summaries

The main numeric summary of paired data is the Pearson correlation coefficient. In terms of z scores this is almost the average of their respective products:

$$cor = \frac{1}{n-1} \sum \left(\frac{x_i - \bar{x}}{s_x} \right) \cdot \left(\frac{y_i - \bar{y}}{s_y} \right).$$

The `cor` function computes this measure.

Consider again the `iris` data, only we select below just the `setosa` data and the petal attributes:

```
l, w = @chain iris begin
    subset(:Species => ByRow(==("setosa")))
    select([:PetalLength, :PetalWidth])
    eachcol
end;
```

We have the correlation computed by

```
cor(l, w)
```

```
0.3316300408041188
```

The correlation will be seen to be measure of linear associativity. When it is close to 1 or -1, the variability in the data is well described by a line; when close to 0, not so.

The Spearman correlation is the correlation of the data *after* ranking. It is a measure of monotonicity in the data. There are different ranking algorithms in `StatsBase`, the Spearman correlation uses `tiedrank`. It can be computed with this or with `corspearman`, as shown:

```
corspearman(l, w), cor(tiedrank(l), tiedrank(w))
```

```
(0.2711413763783511, 0.2711413763783511)
```

The primary graphic to assess the relationship between the two, presumably, dependent variables is a scatter plot:

```
scatter(l, w; legend=false, xlab="PetalLength", ylab="PetalWidth")
vline!([mean(l)], linestyle=:dash)
hline!([mean(w)], linestyle=:dash)
```

Figure 4.9: Scatter plot of petal length and petal width for the *setosa* data. The dashed lines show the "center" of the data, $(\bar{x}, \bar{y})$.

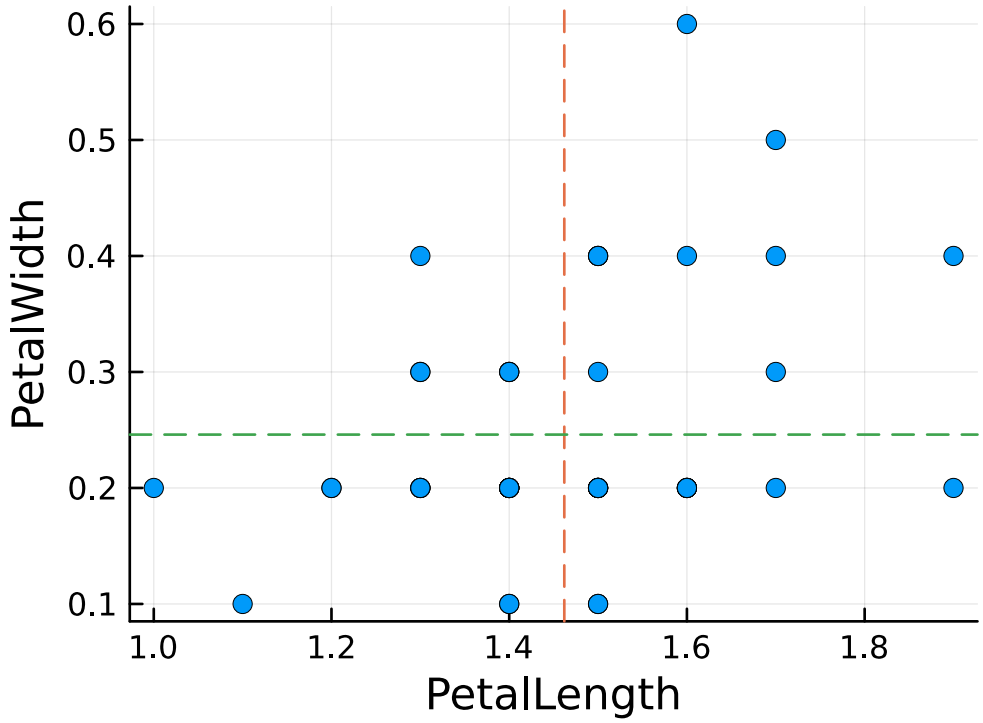


Figure 4.9 shows the length and width data in a scatter plot. The dashed lines are centered at the means of the respective variables. If the mean is the center of a single variable, then $(\bar{x}, \bar{y})$ may be thought of as the center of the paired data. Thinking of the dashed lines meeting at the origin, four quadrants are formed. The correlation can be viewed as a measure of how much the data sits in opposite quadrants. In the figure, there seems to be more data in quadrants I and III then II and IV, which sugests a *positive* correlation, as confirmed numerically.

By writing the correlation in terms of z-scores, the product in that formula is *positive* if the point is in quadrant I or III and negative if in II or IV. So a big positive number suggests data is concentrated in quadrants I and III or that there is a strong association. The scaling by the standard deviations, leaves the mathematical constraint that the correlation is between -1 and 1.

> **Correlation is not causation**
>
> The expression "correlation is not causation" is a common one, due to efforts by the tobacco industry to give plausible alternatives to the statistical association of cigarette smoking and health problems such as lung cancer. Indeed, in some instances correlation between two variables can be related to a lurking or confounding variable. An example might be coffee consumption is positively associated heart disease, yet is not necessarily the cause, as smoking is associated with both variables and may be the cause.

4.2.2 Trend lines

We mentioned that the Pearson correlation is a measure of a linear association and the Spearman rank correlation a measure of monotonic association. We now discuss some basic *trend lines* that highlight the association in paired data.

We use the language *explanatory* variable to describe the x variable, and *response* variable to describe the y variable, the idea that x might be experimentally controlled and y a measured response. The "line" is an explanation of how the y values vary *on average* as the x values vary.

There are many choices for how to identify a trend line, though we will show one such way that is the most common one used. To set it in a mathematical setting, a line can be parameterized by two values, $y = \beta_0 + \beta_1 x$. Use the value $\hat{y}_i = \beta_0 + \beta_1 x_i$, which is the y value of the point on the line corresponding to x_i. The residual amount is usually defined by $e_i = y_i - \hat{y}_i$. That is the vertical distance from the data point to the line. See Figure 4.10 for a visualization. One could argue that the distance to the line might be better than the vertical distance, but this choice leads to a very useful mathematical characterization.

Let $f(x)$ be some function, then a measure of the distance of the data to the line might be the sum $\sum_i f(e_i)$. For exmaple, using $f(x) = |x|$, we have:

```
F(x) = abs(x)

resid_sum(x, y, F, b0, b1) = sum(F, yi - (b0 + b1*xi) for (xi,yi) in zip(x,y))
j(b0, b1) = resid_sum(l, w, F, b0, b1)

xs = ys = -2 : 1/100 : 2
_, k = findmin(j.(xs, ys'))
(b0 = xs[first(k.I)], b1 = ys[last(k.I)])
```

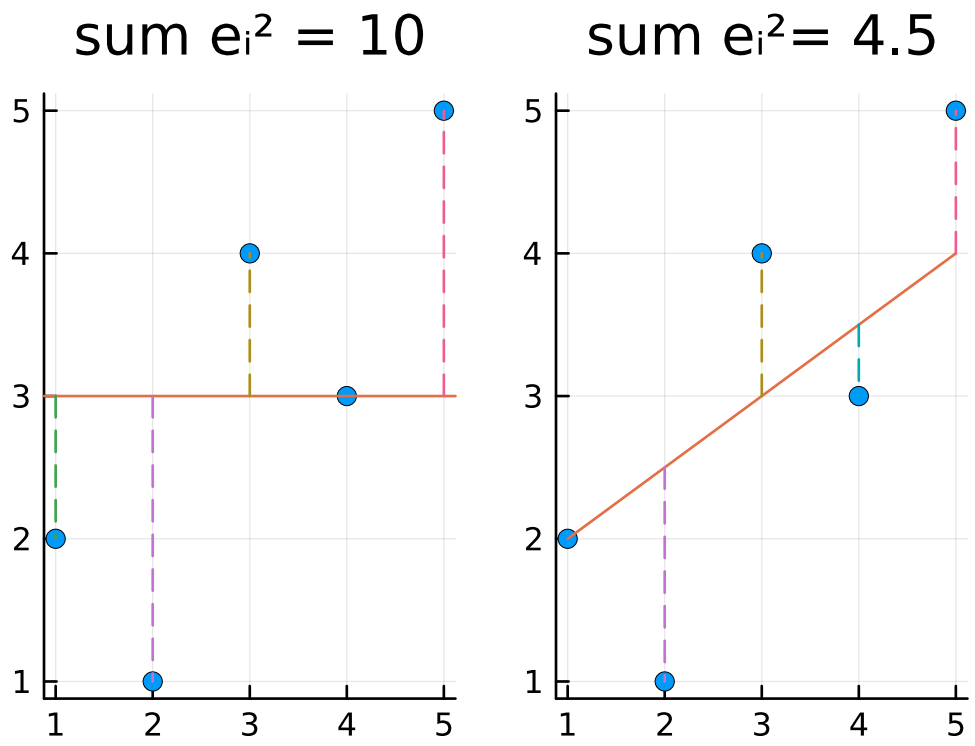
```
(b0 = 0.2, b1 = 0.0)
```

This non-rigorously identifies the values of the coefficients that minimize the sum in $[-2, 2] \times [-2, 2]$ by searching for the smallest over a grid with resolution `1/100`.

Using a different F changes the answer:

```
F(x) = x^2
```

Figure 4.10: Two lines as possible fits to the data with the sum of squared residuals reported.



```
resid_sum(x, y, f, b0, b1) = sum(f, yi - (b0 + b1*xi) for (xi,yi) in zip(x,y))
j(b0, b1) = resid_sum(l, w, F, b0, b1)

xs = ys = -2 : 1/100 : 2
_, k = findmin(j.(xs, ys'))
(b0 = xs[first(k.I)], b1 = ys[last(k.I)])
```

```
(b0 = -0.06, b1 = 0.21)
```

When $F(x) = x^2$, that is we seek to minimize the sum of the squared residuals, there is a mathematical formulation that can be solved to identify values for the constants. This is known as the *method of least squares*. There is a mathematical framework that this fits within that uses linear algebra.

Consider the n linear equations of the form $y_i = \beta_0 + \beta_1 x_i$. Written in a column form these are:

$$\begin{aligned} y_1 &= \beta_0 + \beta_1 x_1 \\ y_2 &= \beta_0 + \beta_1 x_2 \\ &\vdots \\ y_n &= \beta_0 + \beta_1 x_n \end{aligned}$$

There are 2 unknowns, β_0 and β_1. *Were* there two equations ($n = 2$), we could solve by substitution to find an expression for these in terms of (x_1, y_1) and (x_2, y_2). We algebraically solve this below with the help of a symbolic math package:

```
using SymPy
@syms x[1:2] y[1:2] β[0:1]
eqns = [ones(2) x] * β .~ y
```

$$\begin{bmatrix} x_1\beta_1 + 1.0\beta_0 = y_1 \\ x_2\beta_1 + 1.0\beta_0 = y_2 \end{bmatrix}$$

```
solve(eqns, β)
```

```
Dict{Any, Any} with 2 entries:
  β₀ => (x₁*y₂ - x₂*y₁)/(x₁ - x₂)
  β₁ => (y₁ - y₂)/(x₁ - x₂)
```

Mathematically for the *linear problem*, this last line is equivalent to solving the equation $A\beta = y$ where A is a *matrix* with one column of 1s and the other of x:

```
A = [ones(Sym, length(x)) x]
```

$$\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \end{bmatrix}$$

```
A \ y
```

$$\begin{bmatrix} \frac{x_1 y_2 - x_2 y_1}{x_1 - x_2} \\ \frac{y_1 - y_2}{x_1 - x_2} \end{bmatrix}$$

However, it is expected that there are more than 2 data points, so the line will in general be *overdetermined*. The linear algebra problem implemented in the \ operation used above solves the same least squares problem. So for our problem, we can get the coefficients via:

```
A = [ones(length(l)) l]
A \ w
```

```
2-element Vector{Float64}:
 -0.048220327513871966
  0.20124509405873592
```

Algebraically, the above yields these values for the βs in terms of the data, (x_i, y_i):

$$\hat{\beta}_1 = cor(x, y) \cdot \frac{s_y}{s_x}, \quad \hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}.$$

That is, for this line:

- the slope ($\hat{\beta}_1$) is proportional to the correlation coefficient, the proportion being the ratio of the standard deviation in the response variable and standard deviation of the explanatory variable.
- the point $(\bar{x}, \bar{y})$ is on the line.

The underlying machinery of linear algebra is given a different interface along with generalizations in the `GLM` package which implements the *linear* model through its `lm` function. We also add the `StatsModels` package, but this is included with `GLM`.

The statistical model to fit is specified through a formula defined using an interface from `StatsModels`. In this case the model is basically that `w` linearly depends on `l` and is written `w ~ l`, modeling the width of the petal in terms of the length.

```
using StatsModels, GLM
res = lm(@formula(y ~ x), (y=w, x=l))
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, LinearAlgeb
```

```
y ~ 1 + x
```

```
Coefficients:
```

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	-0.0482203	0.121641	-0.40	0.6936	-0.292796	0.196356

```
x             0.201245    0.0826325   2.44    0.0186   0.0351012   0.367389
```

The formula specifies a pattern in a certain manner, the additional arguments indicate how the pattern refers to the data. We *could* use a simple named tuple here, but a data frame is typical, as used here:

```
lm(@formula(PetalWidth ~ PetalLength), subset(iris, :Species => ByRow(==("setosa"))))
```

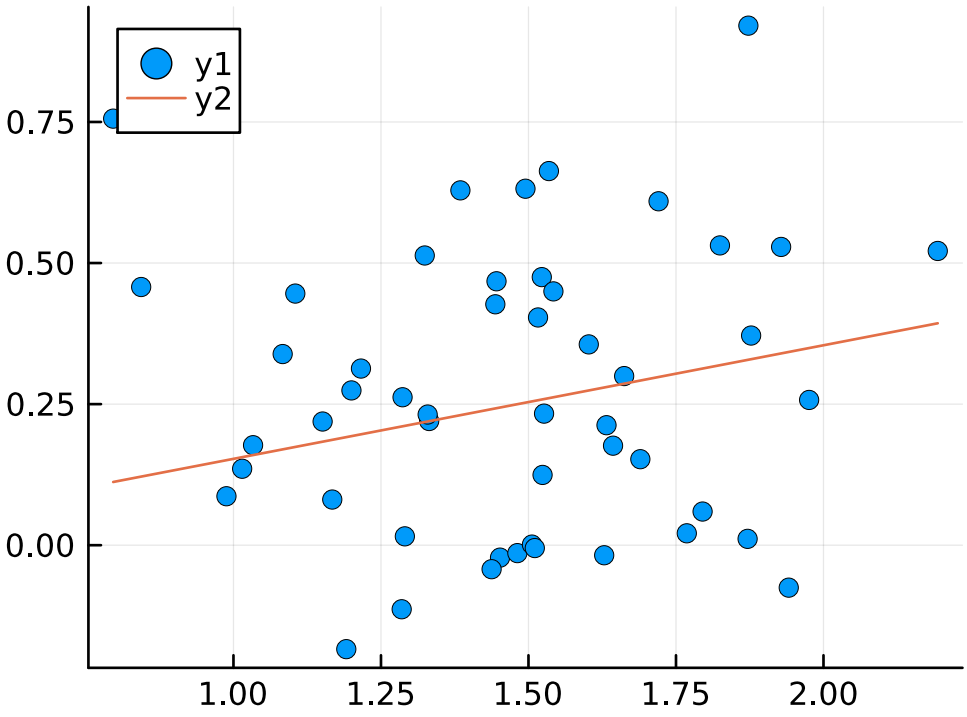
```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, LinearAlgeb

PetalWidth ~ 1 + PetalLength

Coefficients:
```

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	-0.0482203	0.121641	-0.40	0.6936	-0.292796	0.196356
PetalLength	0.201245	0.0826325	2.44	0.0186	0.0351012	0.367389

(The formula is written as a *macro*, so the variables do not need to be symbols, as they are not immediately evaluated because the macro transforms them first.)

The output has more detail to be explained later. For now, we only need to know that the method `coef` will extract the coefficients (in the first column) as a vector of length 2, which we assign to the values `bhat0` and `bhat1` below:

```
scatter(jitter(l), jitter(w))  # spread out values
bhat0, bhat1 = coef(res)       # coef(res) returns the coefficients
plot!(x -> bhat0 + bhat1 * x)
```



A constant model

The linear model has much wider applicability than the *simple* regression model, $y_i = \beta_0 + \beta_1 x_i + \epsilon_i$. To see one example, we can fit the model $y_i = \beta_0 + \epsilon_i$ using least squares:

```
lm(@formula(y ~ 1), (y=l,)) |> coef
```

```
1-element Vector{Float64}:
 1.462
```

The estimate is the `mean` of `l`.

4.2.3 Local regression

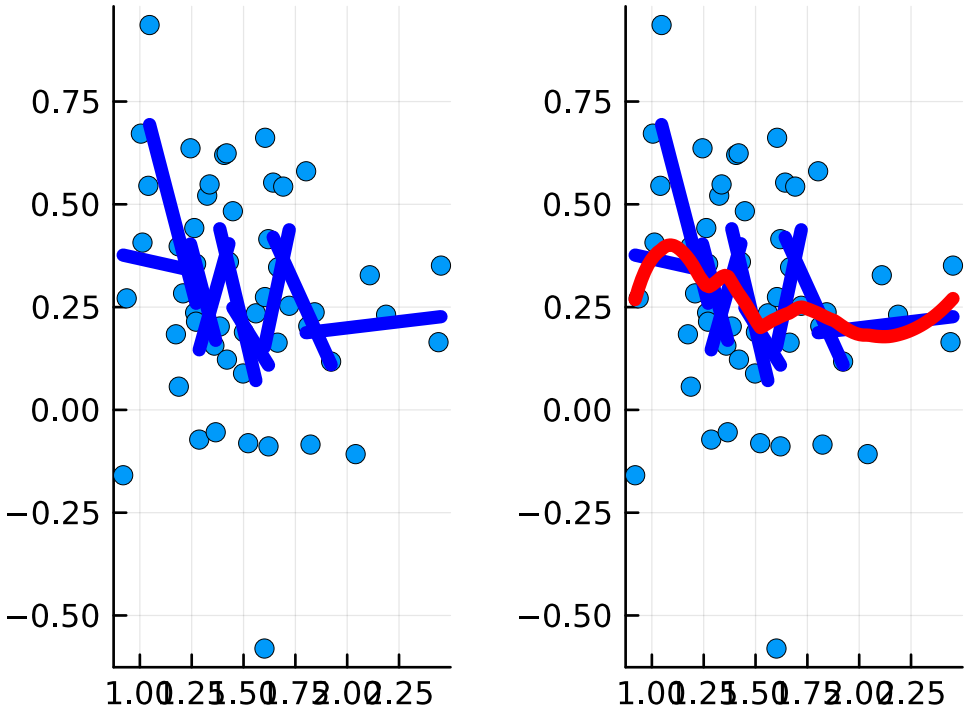
The method of least squares used above is called "regression." It was popularized by Galton [@GALTON_10.2307/2841583] and the phrase "regression to the mean" which will be explained later. The method of local regression (LOESS, with SS meaning scatterplot smoothing) essentially fits a regression line (or a degree d polynomial) to parts of the data then stitches them together. The result is a curve that summarizes any trend in the data between the response variable and the explanatory variable. Figure 4.11 shows some *locally* fit regression lines on the left, and on the right an identified `loess` line to show that the loess line here mostly follows the local regression models. The "loess" line was added to with these commands from the `Loess` package, which are wrapped up into a function to call later:

```
import Loess   # import so as not to conflict with other loaded packages

function loess_line!(p, x, y; normalize=true, span=0.75, degree=2, kwargs...)
  model = Loess.loess(x, y; normalize=normalize, span=span, degree=degree)
  us = range(extrema(x)..., length=100)
  plot!(p, us, Loess.predict(model, us); kwargs...)
end
```

```
loess_line! (generic function with 1 method)
```

The default value of the `degree` keyword is `2` for a *quadratic* fit to the data.

Figure 4.11: On left, a scatter plot with some local regression lines (based on 10 points) layered on; on right, the same with a degree 1 `loess` line added.



4.2.4 Robust linear models

The formula for the slope of the simple regression coefficient depends on s_y/s_x. The standard deviation may be influenced greatly by 1 outlying value. That is, they are not very resistant to outliers. A theory or *Robust Linear Models* has been developed which are useful in allowing a relaxation on the standard distributional assumptions, relaxations that accommodate data with more outliers. The `RobustModels` package imple-

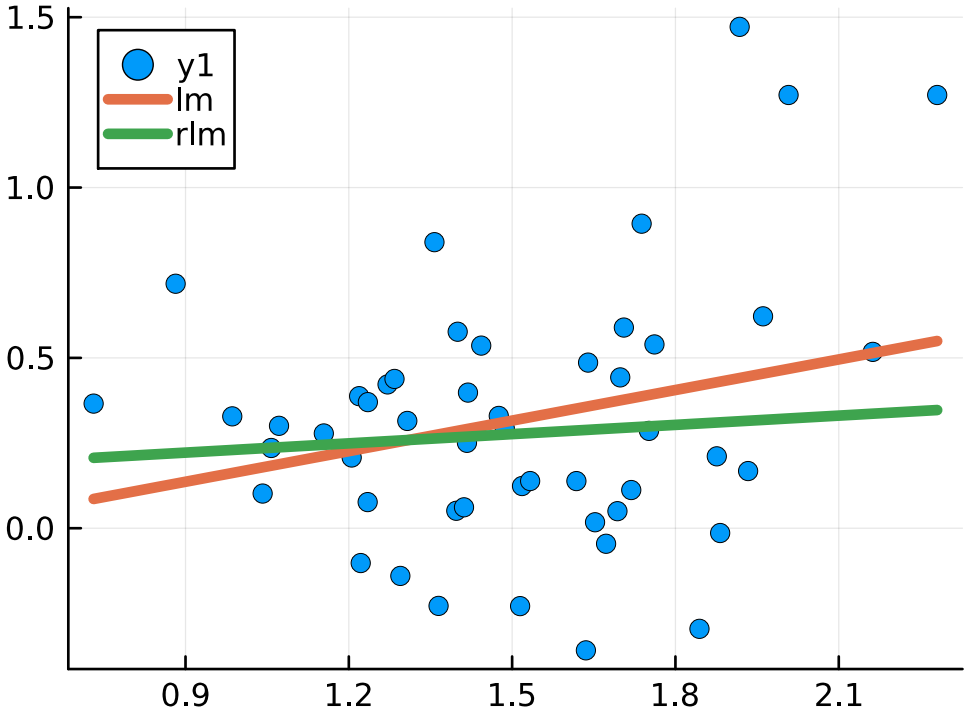
ments a number of such models, of which we illustrate just `MEstimator{TukeyLoss}` without comment as to the implemented algorithm.

One way to view data sets that benefit from robust models is to envision a process where the measured response values are given by the value on a regression line *plus* some "normally" distributed error **plus** some "contaminated" error which interoduces outliers.

To simulate that in our length and width data, we simply add in a contaminant to some of the terms below which appear in Figure 4.12 in the upper-right corner. That figure shows the least-squares regression line is influenced by the 3 contaminated points, but that robust regression line is not.

```
w′ = jitter(w)
l′ = jitter(l)
idx = sortperm(l′)
w′, l′ = w′[idx], l′[idx]

sigma_e = dispersion(res.model) # sqrt(deviance(res)/dof_residual(res))
inds = [45, 48, 50]             # some indices,

w′[inds] = w'[inds] .+ 5*sigma_e  # contaminate 3 values

scatter(l′, w′; legend=:topleft)

fm = @formula(w ~ l)
res = lm(fm, (w=w′, l=l′))

import RobustModels
estimator = RobustModels.MEstimator{RobustModels.TukeyLoss}()
res_robust = RobustModels.rlm(fm, (w=w′, l=l′), estimator)


us = range(minimum(l′), maximum(l′), 100)
plot!(us, GLM.predict(res, (l=us,)); linewidth=4, label="lm")
plot!(us, RobustModels.predict(res_robust, (l=us,)); linewidth=4, label="rlm")
```

Figure 4.12: The 3 data points in the upper right of this graphic are "contaminated" data. Their outlying nature has unde influence on the least-squares regression line ("lm") pulling it up towards the points. The robust regression line ("rlm") is not so influenced.



4.2.5 Prediction

In the definition of `loess_line!` and Figure 4.12 a `predict` method from various packages was used to plot the trend line. The pattern was similar: `predict(model, (l=data,))` created y values for the given explanatory values.

The language comes from the usage of statistical models.

The explanatory "variable", in general, may consist of 0, 1, or more variables. Examples with 0 or 1 for `lm` were given. It is not hard to envision in the `iris` data that the petal length might be influenced by the species, the petal width, and even details of the sepal sizes. The word explanatory hints that the y values should be somehow explained by the covariates. Of course, statistical data is typically subject to variability, so even if all covariates are equal, different responses need not be the same. A statistical model may account for this by expressing a relationship:

response = predicted + error

For our purposes, the "predicted" part comes from some parameterized function of the covariates and forms the structural model for the data. On average, it is assumed the errors are 0, but that an individual error

introduces some randomness.

The parameters in the model are fitted by some process (such as `lm`) leaving the related decomposition

response = fitted + residual

In `GLM` the generic `fit` method is used to fit a model; `lm(...)` is just a shortcut to `fit(LinearModel, ...)`.

The "fitted" and "residual" values are based on the data and are known quantities; the "predicted" and "error" are postulated and may be unknowable. It is common to use the identified model as estimates for predicted values.

In the above examples, the `predict` method was used to do just that. The syntax for `predict` may have seemed more complicated than needed, but the model may have many covariates, so the specification needs a means to identify them. A `Tables` compatible specification of the data used in the prediction is used, in the above that was a named tuple.[2]

The modeling environment of `Julia` has these generic methods for the parts of a model:

- `predict(model, newdata)`: Make predictions for new values of the covariate(s).
- `fitted(model)`: The predicted values for the covariate(s) used to identify the parameters in the model.
- `response(model)`: The response values
- `residuals(model)`: The response values minus the fitted values

The residuals are not the errors, but give a sense of them. The `residuals` function returns the residual values for a model fit. For example, we have these summaries:

```
mean(residuals(res)), std(residuals(res))
```

```
(7.327471962526034e-17, 0.36587303650686603)
```

(The mean of the residuals is **always** 0 for a linear model, a consequence of the method of least squares.)

4.2.6 Assessing the residuals

In the creation of Figure 4.12 the following value was used:

```
sigma_e = dispersion(res.model)
```

```
0.36966456820381355
```

This is the square root of the sum of the squared residuals (computed by `deviance`) divided by the *degrees of freedom* (in this case $n - 2$, but in general given by `dof_residual`):

```
mse = deviance(res) / dof_residual(res) # sense of scale
sqrt(mse)
```

```
0.36966456820381355
```

[2] A named tuple of vectors to specify a data set uses a struct-of-arrays; a vector of named tuples uses an array-of-structs approach.

For the simple linear regression model, the value $\sum e_i^2$ is minimized by the choice of the estimates for the βs. This value is called the SSE, or sum of squared errors. It can be found from the residuals *or* the generic method `deviance`:

```
sum(ei^2 for ei in residuals(res)), deviance(res)
```

```
(6.559290863294972, 6.559290863294972)
```

The deviance gives a sense of scale, but does not reflect the size of the data set, n. The residual *degrees of freedom* of a model is $n - k$ where k is the number of estimated parameters. In the simple linear regression this is 2, which we verify:

```
n = length(l)
n - 2, dof_residual(res)
```

```
(48, 48.0)
```

The *mean squared error* (MSE) is the ratio of the SSE and the degrees of freedom, or $(\sum e_i^2)/(n-k)$. This is like a sample variance for the error terms, the square root of it, *like* the standard deviation, a also measure of variability.

The residuals can be used to graphically assess the ability of the fitted model used to describe the data. There are a few standard graphics:[3]

- A fitted versus residuals plot. The residuals are the data less the structural part of the model. If the model fits well, the residuals should not show a pattern.
- A quantile plot of the residuals. For statistical inference, there are distributional assumptions made on the error values. This plot helps assess if typical assumption on the shape is valid.
- A scale location. For statistical inference, there are distributional assumptions made on the error values. This plot helps assess if typical assumption of equal variances is valid.
- Residuals versus leverage. As regression can be sensitive to influential outliers. This is assessed with Cook's distance which measures the change if point is used to fit the data compared to if it is not.

The support for these graphics in `Julia` is not the same as in R, say. We first define several helper functions:

```
using LinearAlgebra: diag
hat(res) = (X = modelmatrix(res); X * inv(crossmodelmatrix(res)) * X')
hatvalues(res) = diag(hat(res))
rstandard(res) = [ei/sqrt(1-hi) for (ei, hi) in zip(residuals(res), hatvalues(res))] / dispersion(res)
```

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

The `hat` function computes a matrix that arises in the regression model, its diagonal entries are returned by `hatvalues`. These are used to scale the residuals to produce the *standardized residuals*, computed by `rstandard`. The generic `leverage` is intended to compute what `hatvalues` does, but it currently doesn't have a method defined in `GLM`.

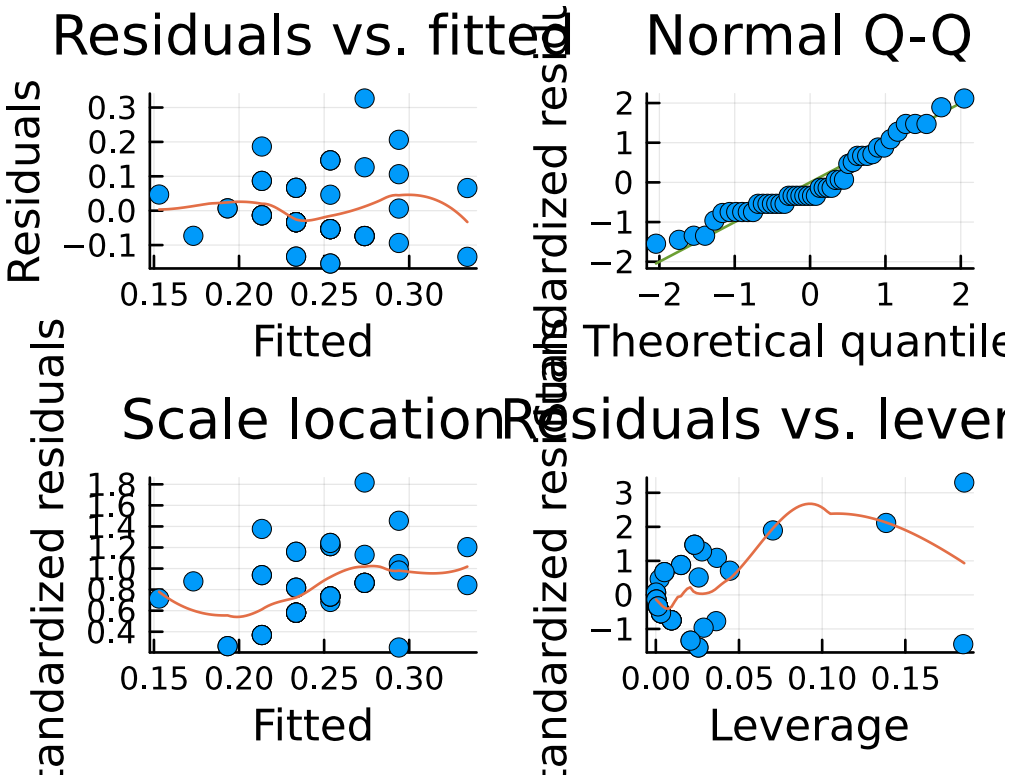
[3] These are graphics that R uses with linear models.

The 4 basic diagnostic plots follow those returned by default in R, though the R versions offer more insight into outlying values. In Figure 4.13 the 4 graphics are shown for the "setosa" species data with PetalWidth modeled by PetalLength.

```
function fitted_versus_residuals(res)
  p = plot(; legend=false)
  scatter!(p, fitted(res), residuals(res);
  xlab="Fitted", ylab="Residuals", title="Residuals vs. fitted")
  loess_line!(p, fitted(res), residuals(res))
  p
end

function quantile_residuals(res)
    qqnorm(rstandard(res);
       legend=false,
       title = "Normal Q-Q",
       xlab="Theoretical quantiles", ylab="Standardized residuals")
end

function scale_location_plot(res)
    x, y = fitted(res), rstandard(res)
    y = sqrt.(abs.(y))

    p = plot(; legend=false)
    scatter!(p, x, y;
        title="Scale location",
        xlab="Fitted", ylab = "Standardized residuals")
    loess_line!(p, x, y)
    p
end

function residual_leverage(res)
   x, y = cooksdistance(res), rstandard(res)

   p = plot(; legend=false)
   scatter!(p, x, y;
    title = "Residuals vs. leverage",
    xlab="Leverage", ylab="Standardized residuals")
   loess_line!(p, x, y)
   p
end
```

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

Figure 4.13: Four diagnostic plots, following the standard ones of R



> **Alternatives**
>
> In the above, we created functions to compute some diagnostic values. These are also defined in the `LinRegOutliers` package [@LinRegOutliers] along with much else. However, the interface is not quite compatible with the model results of `lm`, so we didn't leverage that work here.

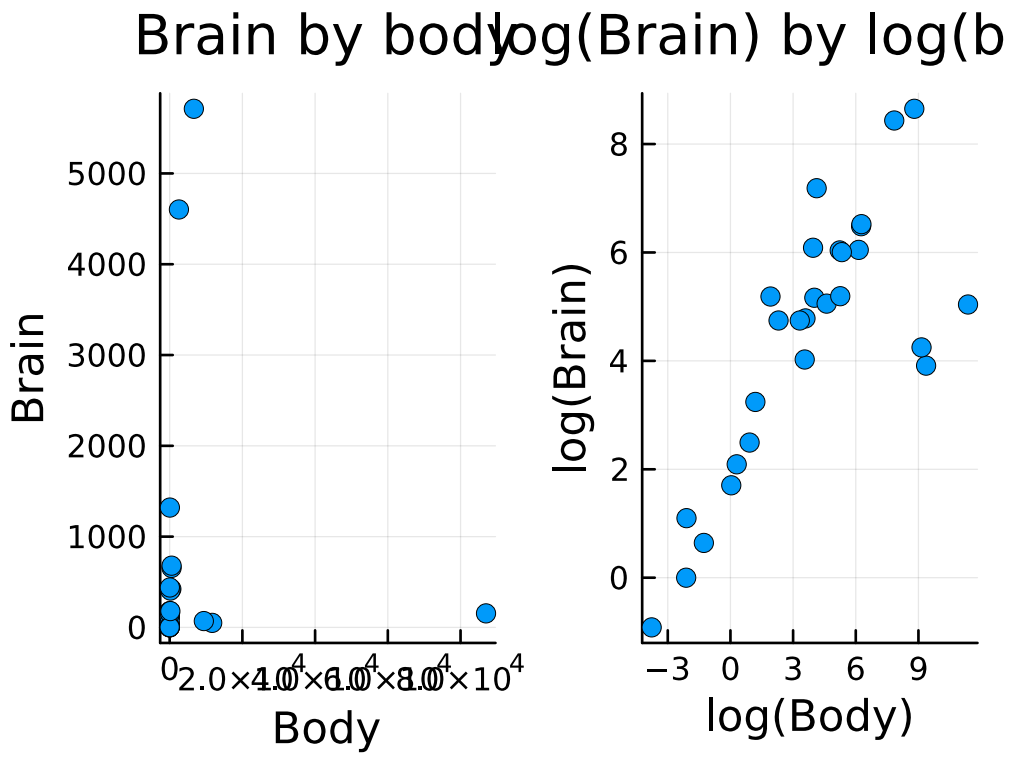
4.2.7 Transformations

The `lm` function, as used above, expects *linear* relationships between the explanatory variable and the response variable. That is, a model of the type $y = \beta_0 + \beta_1 x + \epsilon$. There may be data for which some mathematical transformation is necessary in order to get this structural form.

The `animals` data set, defined below, measures the body size and brain size of various species of animals. The brain size depends on the body size, but it seems to be non-linear and better modeled by $e^y = e^{\beta_0 + \beta_1 x + \epsilon}$. In Figure 4.14 we can see scatter plots of the two data sets before and after a log transform.

```
animals = dataset("MASS", "Animals")
transform!(animals, [:Brain, :Body] .=> ByRow(log) .=> [:logBrain, :logBody])

p1 = @df animals scatter(:Body, :Brain; legend=false,
                         title="Brain by body", xlab="Body", ylab="Brain")
p2 = @df animals scatter(:logBody, :logBrain; legend=false,
                         title="log(Brain) by log(body)",
                         xlab="log(Body)", ylab="log(Brain)")
plot(p1, p2; layout = (@layout [a b]))
```

Figure 4.14: Plot of `animals` data before and after log transform of the body and brain variables.

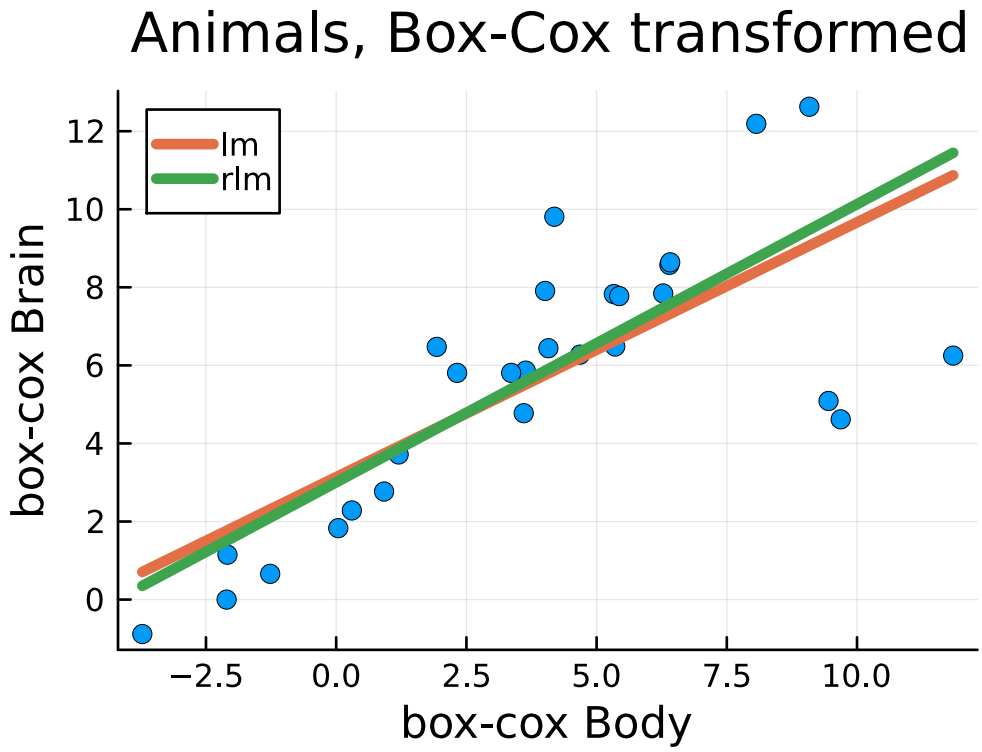


There is an *automated* group of transformations called *power transformations* and are implemented in the `BoxCoxTrans` package. Its `transform` function identifies an appropriate monotonic transformation. We can see the results of the transformations of both variables here along with a regression line and a robust regression line:

```
import BoxCoxTrans
transform!(animals, [:Brain,:Body] .=> BoxCoxTrans.transform .=> [:bcBrain, :bcBody])

p = @df animals scatter(:bcBody, :bcBrain; label=nothing,
    title = "Animals, Box-Cox transformed",
    xlab = "box-cox Body", ylab = "box-cox Brain")

fm = @formula(bcBrain ~ bcBody)
res = lm(fm,  animals)

estimator = RobustModels.MEstimator{RobustModels.TukeyLoss}()
res_robust = RobustModels.rlm(fm, animals, estimator)
```

```
us = range(extrema(animals.bcBody)..., length=100)
plot!(us, GLM.predict(res, (bcBody = us,)), linewidth=4, label="lm")
plot!(us, RobustModels.predict(res_robust, (bcBody = us,)), linewidth=4, label="rlm")
```

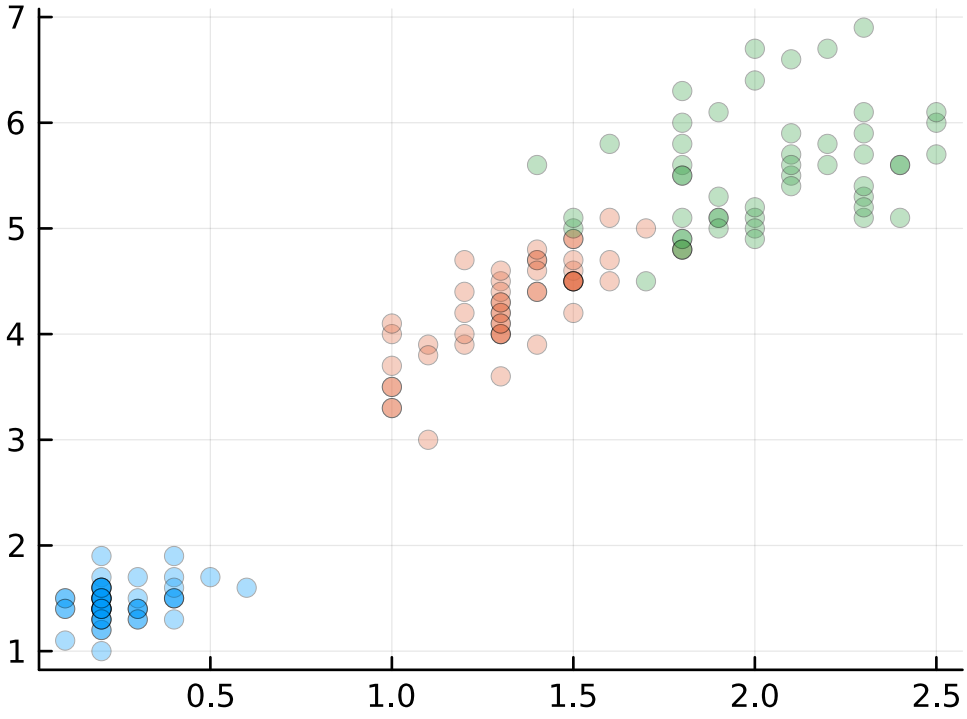
Figure 4.15: The `animals` data after both the `Body` and `Brain` variables have been transformed through the Box-Cox method. This data set has 3 outliers that pull the `lm` regression line downward, but do not impact the robust regression line, which more accurately captures the trend for all but the 3 outliers (which happen to be dinosaur species).



4.2.8 Grouping by a categorical variable

The `iris` data is an example of differences between the species, one more so than the other. The `group` argument to `scatter` instructs the coloring of the markers to vary across the values of the grouping variable, effectively showing three variables at once.

```
p_scatter = @df iris scatter(:PetalWidth, :PetalLength; group=:Species,
      legend=false, alpha=0.33)
```



In Figure 4.16 the basic scatter plot for petal width modeled by petal length is shown in the left figure along with a fitted regression line. We set an alpha value for the scatterplot to better see the regression lines. The line itself was prepared with:

```
m1 = lm(@formula(PetalLength ~ PetalWidth), iris)
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, LinearAlgeb
```

```
PetalLength ~ 1 + PetalWidth
```

```
Coefficients:
```

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	1.08356	0.072967	14.85	<1e-30	0.939366	1.22775
PetalWidth	2.22994	0.0513962	43.39	<1e-85	2.12838	2.33151

```
us = range(extrema(iris.PetalWidth)..., 100)
vs = predict(m1, (PetalWidth = us, ))
```

```
plot!(p_scatter, us, vs; title="iris data");
```

The three clusters can influence the regression line fit to the scatter plot, that is `species` can be an influence, as would be expected. There are different ways for this expectation to be incorporated.

First, suppose we simply adjust the fitted lines up or down for each cluster. That is, there is some additive effect for each type of species. As a example of something we discuss at length later, we can fit this model by including `Species` as an additive term:

```
m2 = lm(@formula(PetalLength ~ PetalWidth + Species), iris)
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, LinearAlgeb
```

```
PetalLength ~ 1 + PetalWidth + Species
```

```
Coefficients:
```

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	1.2114	0.0652419	18.57	<1e-39	1.08246	1.34034
PetalWidth	1.01871	0.152242	6.69	<1e-09	0.717829	1.31959
Species: versicolor	1.69779	0.180948	9.38	<1e-15	1.34018	2.05541
Species: virginica	2.27669	0.281325	8.09	<1e-12	1.7207	2.83269

The second row in the output of `m2` has an identical interpretation as for `m1` – it is the slope of the regression line. The first line of the output in `m1` is the x-intercept, which moves the line up or down. Whereas the first of `m2` is the x intercept for a line that describes *just one* of the species, in this case `setosa`. (A coding for the regression model with a categorical variable chooses one reference level, in this case "setosa."). The 3rd and 4th lines are the slopes for the other two species.

We can plot these individually, one-by-one, in a similar manner as before, however when we call `predict` we include a level for `:Species`. The result is the middle figure in Figure 4.16.

```
p_scatter2 = deepcopy(p_scatter)

gdf = groupby(iris, :Species)
for (k,d) ∈ pairs(gdf)   # GroupKey, SubDataFrame

    s = string(k.Species)

    us = range(extrema(d.PetalWidth)..., 100)
    vs = predict(m2, DataFrame(PetalWidth=us, Species=s))
    plot!(p_scatter2, us, vs; linewidth=5)

end
```

Now we identify different regression lines (slope and intercepts) for each cluster. This is done throuh a *multiplicative* model and is specified in the model formula of `StatsModels` with a `*`:

```
m3 = lm(@formula(PetalLength ~ PetalWidth * Species), iris)
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, LinearAlgeb
```

```
PetalLength ~ 1 + PetalWidth + Species + PetalWidth & Species
```

```
Coefficients:
```

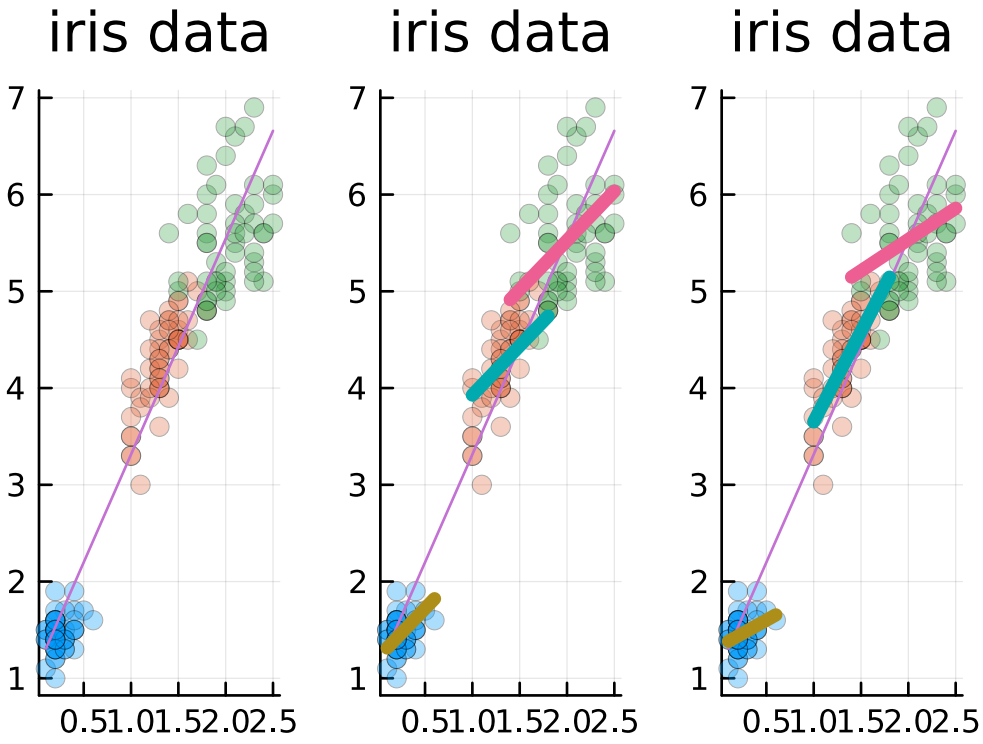
	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	1.32756	0.130933	10.14	<1e-17	1.06877	1.58636
PetalWidth	0.54649	0.490003	1.12	0.2666	-0.422038	1.51502
Species: versicolor	0.453712	0.373701	1.21	0.2267	-0.284935	1.19236
Species: virginica	2.91309	0.406031	7.17	<1e-10	2.11054	3.71564
PetalWidth & Species: versicolor	1.32283	0.555241	2.38	0.0185	0.225359	2.42031
PetalWidth & Species: virginica	0.100769	0.524837	0.19	0.8480	-0.936611	1.13815

The first four coefficients are interpreted similarly to those for `m2`, the remaining 2 summarize the interaction between the petal width and the species type.

We can produce a graphic, the right-most figure in Figure 4.16:

```
p_scatter3 = deepcopy(p_scatter)

gdf = groupby(iris, :Species)
for (k,d) ∈ pairs(gdf)   # GroupKey, SubDataFrame

    s = string(k.Species)

    us = range(extrema(d.PetalWidth)..., 100)
    vs = predict(m3, DataFrame(PetalWidth=us, Species=s))
    plot!(p_scatter3, us, vs; linewidth=5)

end
```

Figure 4.16: Three scatterplots of petal width by petal length in the `iris` data set. The right-hand figure includes the regression line fit to all the data. The middle one fits regression lines identified through an additive model that includes the species. The slopes are parallel, but the lines have an additive shift. The right figure shows a multiplicative model wherein the slopes and intercepts for each species are chosen.



Chapter 5

Categorical data

In the last chapter, the main variable was numeric. Here we consider a variable of a pair of categorical variables.

We use the following, by now standard, packages in this chapter:

```
using StatsBase, StatsPlots,
    RDatasets, DataFrames, Chain, CategoricalArrays
```

5.1 Univariate categorical data

Let's consider a data set from R's `MASS` package on a student survey; in particular the `Smoke` variable, which is stored as a categorical variable. The `levels` method reports 4 levels for this variable:

```
survey = dataset("MASS", "survey")
smokes = survey.Smoke
levels(smokes)
```

```
4-element Vector{String}:
 "Heavy"
 "Never"
 "Occas"
 "Regul"
```

How frequent is each? We would need to tabulate to answer that question. This can be done grouping and applying the `nrow` from `DataFrames`:

```
combine(groupby(survey, :Smoke), nrow)
```

	Smoke	nrow
	Cat…?	Int64
1	*missing*	1
2	Heavy	11
3	Never	189
4	Occas	19
5	Regul	17

Similarly, we can split and apply with `SplitApplyCombine`:

```
using SplitApplyCombine
group(r -> r.Smoke, copy.(eachrow(survey))) .|> length
```

```
5-element Dictionaries.Dictionary{Union{Missing, CategoricalValue{String, UInt8}}, Int64}
 CategoricalValue{String, UInt8} "Never" │ 189
 CategoricalValue{String, UInt8} "Regul" │ 17
 CategoricalValue{String, UInt8} "Occas" │ 19
 CategoricalValue{String, UInt8} "Heavy" │ 11
                                 missing │ 1
```

The above could use some tidying up to read quickly, but easily tabulates the same, and would be useful if the data were not in a data frame.

For tabulation, a separate package has benefits. We use the `FreqTables` package and its `freqtables` function. This returns a *named* vector:

```
using FreqTables
tbl = freqtable(smokes)
```

```
5-element Named Vector{Int64}
Dim1    │
────────┼────
"Heavy" │  11
"Never" │ 189
"Occas" │  19
"Regul" │  17
missing │   1
```

Named vectors are from `NamedArrays` and offer indexing by name, similar to a data frame.

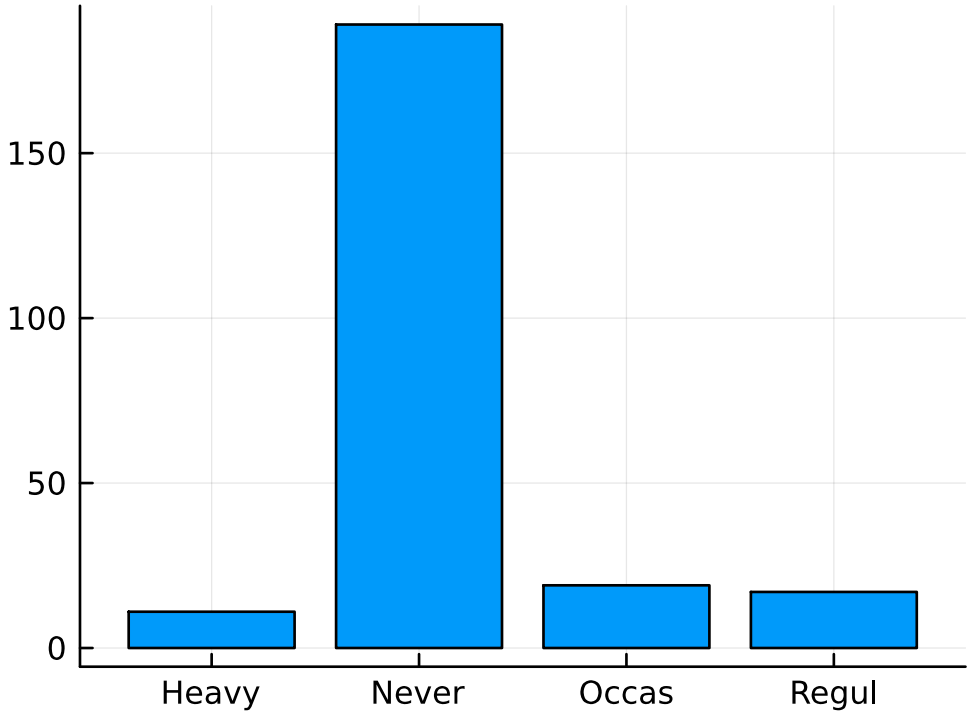
Either way (and there are others), this shows for this data set, "Never" is by far the most common response, and there is 1 missing response.

Such tabulations can be visualized.

A bar chart represents the above table using bars proportional to the counts. The `bar` function makes a bar

chart for the labels in x and the given data. For this example, we have either have to add "missing" to the levels, or, as is done here, excise it from the data set

```
bar(levels(smokes), tbl[1:end-1]; legend=false)
```



Another common graphic is to use a dot, not a bar, to represent the value.

5.2 Paired categorical data

In the `survey` data set we could look at pairs of data where both are categorical and ask questions about the pair. For example, the data contains information on smoking *and* identified gender (`Sex`). Is one gender more likely to smoke?

Again, we can use grouping and apply to see the counts:

```
tbl = combine(groupby(survey, [:Sex, :Smoke]), nrow)
first(tbl, 3) # 3 of 10 rows displayed
```

	Sex	Smoke	nrow
	Cat...?	Cat...?	Int64
1	*missing*	Never	1
2	Female	Heavy	5
3	Female	Never	99

A *contingency table* is the more familiar means to view two-way categorical count data. A count of all combinations of the levels of one and the levels of the other is presented in a grid.

With `unstack` we can do this within `DataFrames`:

```
@chain survey begin
    groupby([:Sex, :Smoke])
    combine(nrow => :value)
    dropmissing
    unstack(:Smoke, :value)
end
```

	Sex	Heavy	Never	Occas	Regul
	Cat...	Int64?	Int64?	Int64?	Int64?
1	Female	5	99	9	5
2	Male	6	89	10	12

The above dropped missing values; to keep them in, the `allowmissing` argument may be specified to `unstack`:

```
@chain survey begin
    groupby([:Sex, :Smoke])
    combine(nrow => :value)
    unstack(:Smoke, :value; allowmissing=true)
end
```

	Sex	Never	Heavy	Occas	Regul	missing
	Cat...?	Int64?	Int64?	Int64?	Int64?	Int64?
1	*missing*	1	*missing*	*missing*	*missing*	*missing*
2	Female	99	5	9	5	*missing*
3	Male	89	6	10	12	1

More conveniently, the `freqtable` command will produce contingency tables:

```
tbl = freqtable(survey, :Sex, :Smoke)
```

```
3×5 Named Matrix{Int64}
Sex ╲ Smoke │ "Heavy"  "Never"  "Occas"  "Regul"  missing
────────────┼─────────────────────────────────────────────
"Female"    │       5       99        9        5        0
"Male"      │       6       89       10       12        1
missing     │       0        1        0        0        0
```

The `freqtable` interface allows the user to pass in two variables of data, or, as above, a tabular data set and two variable names. The `freqtable` method summarized them with the levels of the first variables naming the rows, and levels of the second naming the columns.

5.2.1 Conditional distributions of two-way tables

At first glance, there does not seem to be much difference in the smoking variable between the identified genders. As tables may have many more counts in a given row or column, it can be helpful to take proportions of the rows or columns to compare. The `FreqTables` package provides the `prop` function to do so. By default, it takes a proportion of all the data; pass in `margins=1` to get proportions for each row, `margins=2` to get proportions for each column. For example, to compare the distribution of `Smokes` for each level of `Sex`, we take proportions across each row:

```
prop(tbl; margins=1)  # check `sum(prop(tbl; margins=1); dims=2)` returns 1
```

```
3×5 Named Matrix{Float64}
Sex ╲ Smoke │    "Heavy"     "Never"     "Occas"     "Regul"     missing
────────────┼─────────────────────────────────────────────────────────────
"Female"    │  0.0423729    0.838983   0.0762712   0.0423729         0.0
"Male"      │  0.0508475    0.754237   0.0847458    0.101695  0.00847458
missing     │         0.0         1.0         0.0         0.0         0.0
```

There does not seem to be big differences between the rows, indicating that the gender doesn't seem to have an effect on the smoking prevalency.

What about the exercise variable?

```
tbl = freqtable(survey, :Exer, :Smoke)
prop(tbl; margins=1)
```

```
3×5 Named Matrix{Float64}
Exer ╲ Smoke │    "Heavy"    "Never"    "Occas"    "Regul"    missing
─────────────┼──────────────────────────────────────────────────────────
"Freq"       │ 0.0608696   0.756522   0.104348  0.0782609         0.0
"None"       │ 0.0416667       0.75      0.125  0.0416667   0.0416667
"Some"       │ 0.0306122   0.857143  0.0408163  0.0714286         0.0
```

Again, not much difference across the levels of `Exer`.

Finding the row (or column) proportions as above finds the *conditional distribution* for a given value. (Answering the question, say, what is the distribution of the second variable *given* the first variables has a specific level?)

5.2.2 Marginal distributions of two-way tables

A *marginal* distribution from a two-way table is found by adding all the values in each row, or each column. With two-way tables generated from the full data, there are more direct ways to realize these, but from a

two-way table, we just need to apply `sum` to each row or column. The `sum` function takes a `dims` argument to specify the dimension, which, in this case, is `2` for adding along the columns (the second dimension) and `1` for adding down the rows (the first dimension):

```
sum(tbl, dims=1) # kinda like `freqtable(survey.Smoke)`
```

```
1×5 Named Matrix{Int64}
Exer \ Smoke │ "Heavy"  "Never"  "Occas"  "Regul"  missing
─────────────┼──────────────────────────────────────────────
sum(Exer)    │      11      189       19       17        1
```

```
sum(tbl, dims=2) # like `freqtable(survey.Exer)`
```

```
3×1 Named Matrix{Int64}
Exer \ Smoke │ sum(Smoke)
─────────────┼───────────
"Freq"       │        115
"None"       │         24
"Some"       │         98
```

5.2.3 Two-way tables from summarized data

Suppose a data set was presented in the following two-way table:

Table 5.1: A two-way contingency table of fabricated data showing counts of student grades and mode of instruction.

Grade	In person	Hybrid	Asynchronous Online
A - C	10	5	5
D	10	15	10
F	5	10	10

This table could be *stored* as a two-way table in different ways. Here we show how to make this a data frame, then expand it to variables, then summarize.

```
df = DataFrame([
(Grade="A-C", IP=10, Hybrid=5,  Asynchronous=5),
(Grade="D",   IP=10, Hybrid=15, Asynchronous=10),
(Grade="F",   IP=5,  Hybrid=10, Asynchronous=10)
])
```

	Grade	IP	Hybrid	Asynchronous
	String	Int64	Int64	Int64
1	A-C	10	5	5
2	D	10	15	10
3	F	5	10	10

There are 80 students summarized here:

```
sum([sum(r[2:end]) for r in  eachrow(df)])
```

```
80
```

Here we make a data frame with 80 cases:

```
udf = stack(df, Not(:Grade), variable_name=:InstructionType)
ddf = vcat([repeat(udf[i:i,:],j) for (i,j) in enumerate(udf.value)]...)
select!(ddf, [:Grade, :InstructionType]) # drop :value columns
first(ddf, 3)
```

	Grade	InstructionType
	String	String
1	A-C	IP
2	A-C	IP
3	A-C	IP

(It isn't the most efficient, but we repeat the data frame `udf[i:i,:]` `j` times, where `j` is the value of the `i`th row in `udf.value`. Accessing with `udf[o,:]` returns a `DataFrameRow`, which has no `repeat` method defined.)

To see we could return to the original table, we first give the `InstructionType` the right ordering of the levels, then create a frequency table:

```
ordered_levels = ["IP", "Hybrid", "Asynchronous"]
ddf.InstructionType = categorical(ddf.InstructionType;
                                  ordered=true, levels=ordered_levels)
freqtable(ddf, :Grade, :InstructionType)
```

```
3×3 Named Matrix{Int64}
Grade \ InstructionType │ "IP"  "Hybrid"  "Asynchronous"
────────────────────────┼──────────────────────────────
A-C                     │   10         5              5
D                       │   10        15             10
F                       │    5        10             10
```

5.2.4 Graphical summaries of two-way contingency tables

We review a few visualizations of dependent categorical variables.

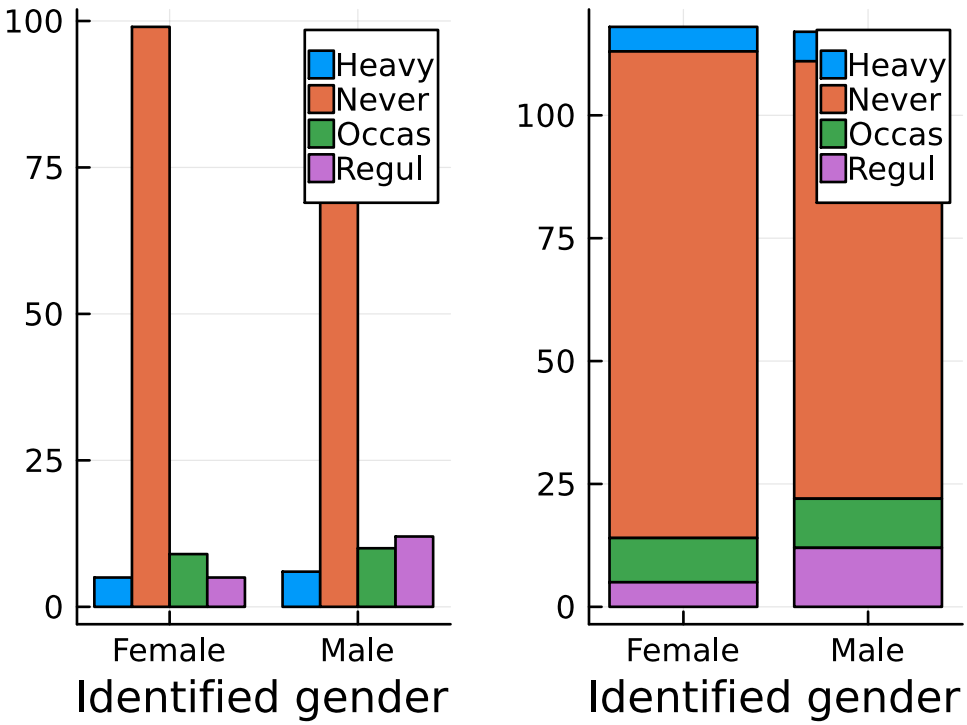
5.2.4.1 Grouped bar plots

The bar plot for a single categorical variable shows frequency counts for each level. A grouped bar plot shows a distribution of the second variable for the grouping variable.

A useful data structure for this graphic is found using `groupby` with 2 variables:

```
tbl = @chain survey begin
    select([:Sex, :Smoke])
    dropmissing
    groupby([:Sex, :Smoke])
    combine(nrow => :value)
end

p1 = @df tbl groupedbar(:Sex, :value, group=:Smoke; xlab="Identified gender")
p2 = @df tbl groupedbar(:Sex, :value, group=:Smoke; xlab="Identified gender",
                        bar_position = :stack)

plot(p1, p2, layout = (@layout [a b]))
```

Figure 5.1: Grouped bar chart of smoking distribution for different levels of the `:Sex` variable. The `bar_position` argument can be passed a value `:stack` to use a stacked display.



As seen in the left graphic of Figure 5.1, there are groups of bars for each level of the first variable (`:Sex`); the groups represent the variable passed to the `group` keyword argument. The values are looked up in the data frame with the computed column that was named `:value` through the `combine` function.

The same graphic on the left – without the labeling – is also made more directly with `groupedbar(freqtable(survey, :Sex, :Smoke))`

5.2.4.2 Andrews plot

An Andrews plot is implemented in `StatsPlots` showing differences in a collection of **numeric** variables for a given categorical variable. For each row, a trigonometric polynomial with coefficients given by the numeric values in the given row creates a function which is plotted. If the values across the categorical variable are similar, the graphs will be; if not, then the groupings will show up.

We first show an example with the `iris` data where the categorical value is `:Species` and the numeric ones the first 4 values. This will be shown in the left graphic of Figure 5.2:

```
iris = dataset("datasets", "iris")
andrews_1 = @df iris andrewsplot(:Species, cols(1:4));
```

For the next plot, we use the `survey` data. There are some efforts needed to wrangle the data: we convert the categorical variables to the numeric levels (`levelcode`) *except* for `:Sex` which we use for the grouping variable. We also drop any `missing` values:

```
iscategorical(x) = isa(x, CategoricalArray) # predicate function
tbl = @chain survey begin
    combine(_, :Sex, findall(iscategorical, eachcol(_))[2:end] .=> ByRow(levelcode); renamecols=false)
    dropmissing
end

andrews_2 = @df tbl andrewsplot(:Sex, cols(2:ncol(tbl)));
```

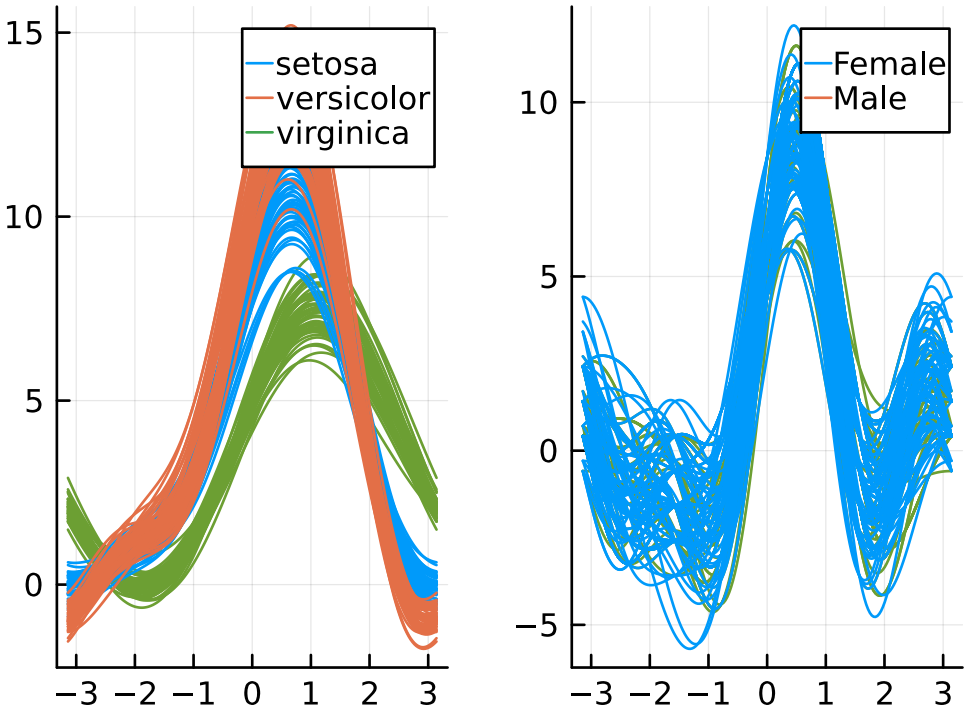
5.2.4.3 Mosaic plots

A mosaic plot presents a graphical view of a two-way contingency table. These are somewhat similar to the grouped bar plot with stacking, but the width of the bars depends on the frequency of the given level.

This graphic is *not* part of `StatsPlots`. We borrow with modification this implementation from OnlineStats:

```
using FreqTables
mosaic_plot(f, g; kwargs...) = mosaic_plot!(Plots.Plot(), f, g; kwargs...)
function mosaic_plot!(p, f, g; xrotation=-45, kwargs...)
    tbl = freqtable(f, g)
    a = sum(tbl, dims=2)
    b = sum(tbl, dims=1)

    a′ = [a.array...]
```

Figure 5.2: Andrews plots of the `iris` data (left graphic) and the `survey` data (right graphic). The `iris` plot shows clear differences based on the `:Species` variable; the `survey` data does not for the `:Sex` variable.



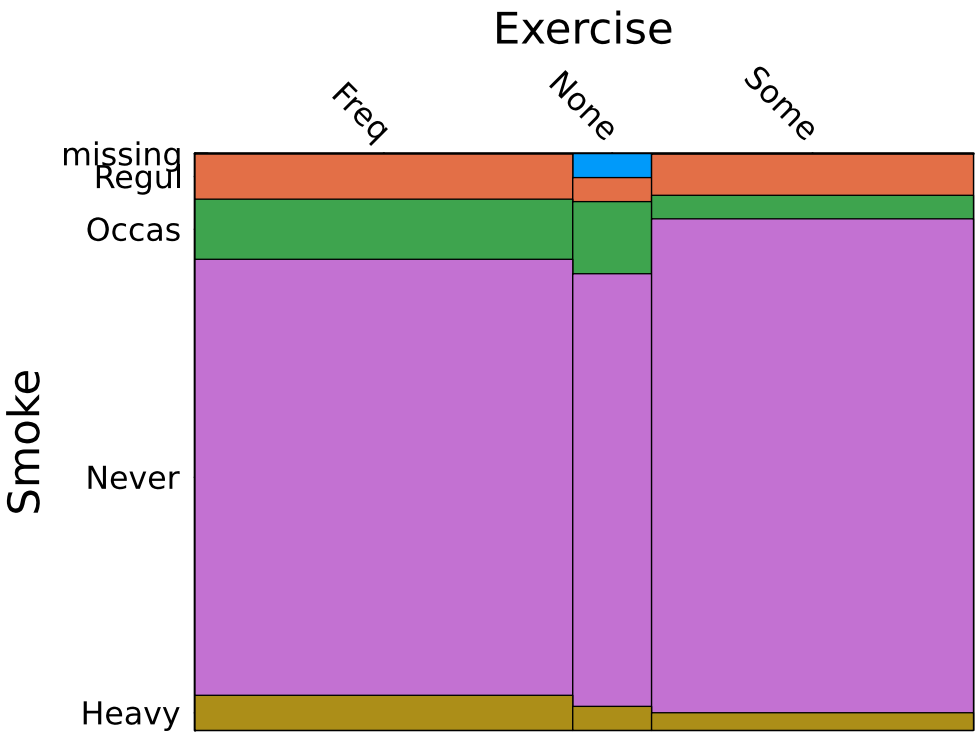
```
    x = vcat(0, cumsum(a')) / sum(a')

    tbl' = convert(Matrix{Float64}, tbl.array)
    tbl'[tbl' .== 0] .+= 1/10  # give some width when missing
    m = prop(tbl', margins=1)
    y = reverse(cumsum(prop(tbl', margins=1), dims=2), dims=2)

    bar!(p,
         midpoints(x), y;
         legend=false,
         bar_width = diff(x),
         xlims=(0,1), ylims=(0,1), linewidth = 0.5,
         xticks =  (midpoints(x), string.(names(a,1))),
         xrotation=xrotation, xmirror=true,
         yticks = (midpoints(vcat(y[1,:],0)), reverse(string.(names(b,2)))),
         kwargs...
         )
end
```

```
mosaic_plot! (generic function with 1 method)
```

For the first variable, displayed on the x-axis, the relative width of the bar is proportional to the marginal proportions; for each level on the x-axis, the vertical bars show the relative proportions of the second variable. For example,

```
@df survey mosaic_plot(:Exer, :Smoke, xlab="Exercise", ylab="Smoke")
```

Figure 5.3: Mosaic plot of the `Exercise` and `Smoke` variables in the `survey` data set.



Chapter 6

The AlgebraOfGraphics

The `StatsPlots` package has been used to illustrate the standard graphics of exploratory statistics. That package leverages `Plots`, a `Julia` interface to multiple plotting backends. The `GR` one renders the images seen. There are a few alternatives, and the `Makie` plotting system along with the `AlgebraOfGraphics` package makes a very compelling alternative.

The `AlgebraOfGraphics` packages offers a *declarative* style to create statistical graphics. An example from the documentation shows the code to do the following "declare the dataset; declare the analysis; declare the arguments used; declare the grouping and the respective visual attribute; draw the visualization." This is all done through a series of composable commands, illustrated by example below. The Pumas project has a much more extensive tutorial.

We will see that it is very easy to visualize multiple variables through an appropriate choice of graphic, and further choices of coloring, faceting, or other means to demarcate different factors.

We begin by loading the packages. The `GLMakie` backend is used here, there are alternatives for web-based graphics and Cairo-based graphics.

```
using StatsBase, DataFrames, RDatasets
using GLMakie, AlgebraOfGraphics
set_aog_theme!()
```

We use the color theme of `aog`, as declared in the last command. The packages are compute-intensive and can take a while to load.

Following the package tutorial we load the Palmer penguins data set of Allison Horst. This includes data collected and made available by Dr. Kristen Gorman and the Palmer Station, Antarctica LTER, a member of the Long Term Ecological Research Network. The data can be downloaded from the GitHub site, but it is also wrapped into a `Julia` package:

```
using PalmerPenguins
penguins = dropmissing(DataFrame(PalmerPenguins.load()))
```

```
irst(penguins, 3)
```

```
LoadError: UndefVarError: irst not defined
```

This data set has several correlated numeric variables on bill length, bill depth, flipper_length, and body_mass_g; and several categorical variables, such as species, island, and sex. A more complete data set can be downloaded from the GitHub site.

6.1 Univariate graphical summaries

We run through the basic graphics for univariate statistics. We shall see that the framework makes multivariate display quite direct, and at times easier than a univariate display.

6.1.1 Boxplot and violin plot

A boxplot (Figure 6.1) for each species is created by composing a series of declarative commands:

```
p = data(penguins) *
    visual(BoxPlot) *
    mapping(:species, :bill_length_mm => "Bill length (mm)", color=:species);
```

The `data(penguins)` command sets up the data. Here a data frame is passed, but this can be any `Tables` compatible structure, such as a struct of arrays: `data( (;species=penguins.species, :bill_length+mm = penguins.bill_length_mm) )`.

The `visual(BoxPlot)` command declares the visualization to be used to view the data. In this case `BoxPlot` is the type associated with the `Makie.boxplot` function.

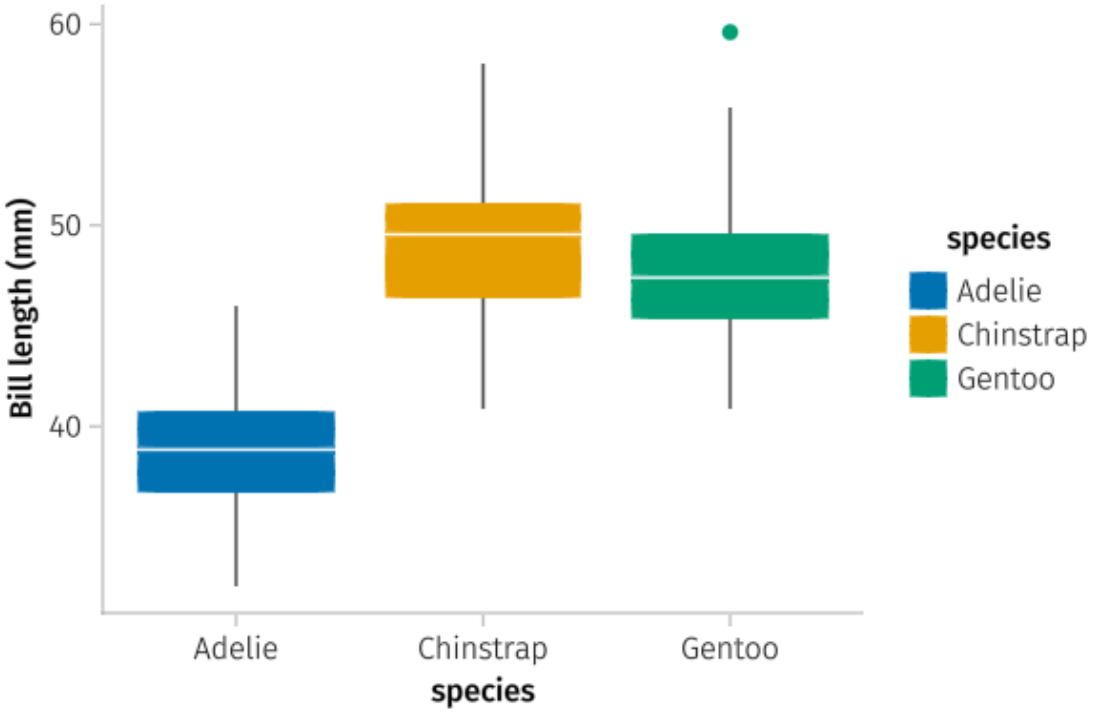
The `mapping` uses *position* to identify the `x`, `y`, and (at times) `z` values for the graphic. The `y` variable uses a mini language nearly identical to the `DataFrames` mini language (only `ByRow` is assumed) to rename the variable, for labeling purposes. This could have also been done in the original data, `penguins`. For a box plot an indicator of the groups goes in the `x` position. The `color=:species` uses a mapping between the levels of the `:Species` variable and color ramp to give each a distinct color. Omitting this will produce a monotone graphic with the chosen theme.

Both `mapping` and `visual` can be used to set attributes; `visual` is used to set attributes for each element independent of the data; `mapping` is used to have attributes depend on a value of a variable, like `color` is used above. The attributes for `visual` are those for the underlying plotting function. For `visual(BoxPlot)`, these can be seen at the help page for `boxplot`, displayed with the command `?boxplot`.

The object `p` can be rendered to the screen with `draw` resulting in Figure 6.1. (Just `draw(p)` will render the graphic, the `figure` keyword arguments takes a named tuple, in this case the figure size is set. Similarly `axis` values can be modified in this manner.)

```
draw(p; figure=(resolution=(600,400),) )
```

Figure 6.1: Boxplots of bill length for the 3 penguin species in `penguins`.



This is the basic pattern where different choices are combined, or merged, with the `*` operation. In the following, we make use of a slightly augmented data set, which we call `d` for reuse:

```
d = data(select(penguins, :, :bill_length_mm => ones ∘ length => :one));
```

Box plots are very effective for quickly comparing distributions of a numeric variable across the levels of some factor. The calling syntax preferences that style, where both an `x` and `y` value are specified to `mapping`. To create a box plot of a *single* variable, without grouping, the graphic takes a bit more to construct. In the above we created an artificial variable in `d` called `one`. This is used to do a simple graphic (upper left graphic in Figure 6.2):

```
p1 = d * visual(BoxPlot) *
    mapping(:one, :bill_length_mm => "Bill length (mm)");
```

The mini language is used above for labeling, but it too could be used to create a a variable with one value: `mapping(:bill_length_mm => one, :bill_length_mm)` would also work to create a single boxplot, in this case centered at 1.

To add another layer, in this case a scatter plot, we can *add* the plotting objects:

```
p2a = d * visual(BoxPlot) * mapping(:species, :bill_length_mm, color=:species)
p2b = d * visual(Scatter) * mapping(:species, :bill_length_mm)
p2 = p2a + p2b;
```

Combinations with `+` add a layer; those with `*` merge layers. The *algebra* name also refers to algebraically desirable short cuts. For example, we repeat `d` and the `mapping` for each `p2a` and `p2b`, but these can be used just once by *distributing* them:

```
m = mapping(:species, :bill_length_mm, color=:species);
p3 =  d * ( visual(BoxPlot) + visual(Scatter) ) * m;
```

Both `p2` and `p3` are shown in the lower row of Figure 6.2. There is just one slight difference, the dots representing the data in `p2` are not colored, as the mapping did not instruct that in forming `p2b`.

Specifying a violin plot requires just a slight modification to the above: we change the `BoxPlot` visual to `Violin`. Violin plots have an argument `side` that allows both sides of the violin to reflect an extra grouping variable. We use `:sex`, as it has only two levels. With this, each side of the violin plot reflects grouping by the `:sex` factor, the legend is used to lookup with level of the factor is represented.

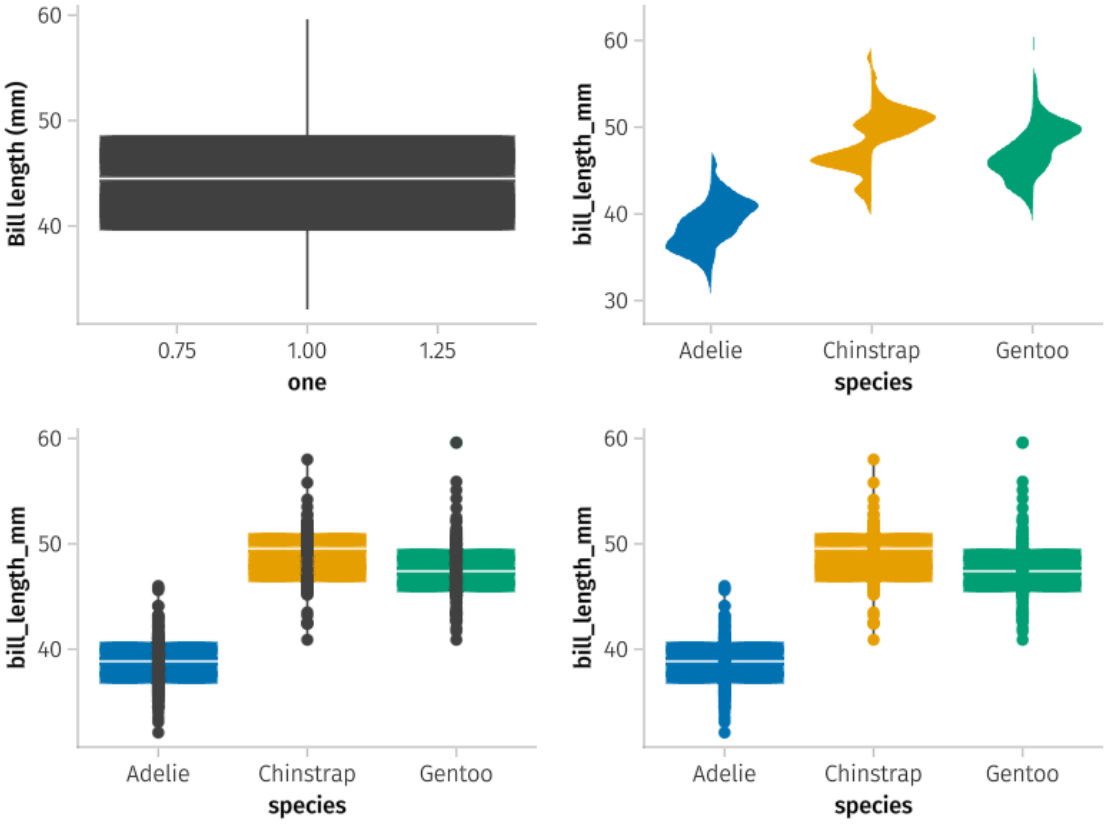
```
p4 = d * visual(Violin) * mapping(:species, :bill_length_mm, color=:species, side=:sex);
```

The `visual(Violin)` call wraps the function `Makie.violin` whose documentation contains additional possible arguments beyond `side`.

The `AlgebraOfGraphics` package builds in `Makie` which has a layout system, which leverages matrix notation to specify cell position. The `draw!` method accepts a figure object as a first argument. In Figure 6.2 we layout 2 rows and 2 columns of figures. It is constructed as follows:

```
f = Figure()
draw!(f[1,1], p1)
draw!(f[1,2], p4)
draw!(f[2,1], p2)
draw!(f[2,2], p3)
f
```

Figure 6.2: Figure showing 4 different graphics displayed. In this case, a single boxplot; a violin plot; a boxplot with scatter; and a similar one with the data and mapping easily resused for each visual.



6.1.2 Faceting

The package also supports *faceting* where different panels share the same scales for each cross comparison. Faceting is specified through the keyword `layout` or either (or both) of `row` and `col` keywords. The `layout` keyword uses levels of the variable name it is passed and arranges the plots over these levels. The `col` will make columns for each level of the specified variable, whereas `row` will create rows for each level of the specified variables.

6.1.3 Histograms

The `AlgebraOfGraphics` has certain functions it refers to as *transformations* of the data. These include `histogram`, `density`, `frequency`, `linear`, `smooth`, and `expectation`; most all will be illustrated by example below.

These are used like `visual` was above, but arguments are passed directly to the transformation.

The `histogram` function plays the role of `visual` in this graphic. (The `visual` function is still useful to apply

data-independent attributes.) Here we arrange to color by `species`:

```
p1 = d * histogram() * mapping(:bill_length_mm, color=:species);
```

The histograms overlap. The `layout` command can be used to declare one panel per level. We do this with `:sex`:

```
p2 = d * histogram() * mapping(:bill_length_mm, color=:species, layout=:sex);
```

See Figure 6.3 for the graphics.

6.1.4 Density plot

The `histogram` function has options for overriding the default bin selection and has several options for scaling the figure through its `normalization` argument. We use this in the next graphic which layers a density plot over a scaled histogram using the `:pdf` scaling. The `density` transformation is qualified with the module name to prevent a conflict with one in `Makie`[1].

```
layers = histogram(normalization=:pdf) + AlgebraOfGraphics.density()
p3 = d * layers * mapping(:bill_length_mm, color=:species, layout=:sex);
```

In this next figure we add in a scatter plot of the data on top of the density plots. For the scatter plot, we use the `Scatter` visual for which we create jittered y values to disambiguate the data, these are added as a column to the data in `d1`, below:

```
p4a = d *  AlgebraOfGraphics.density() *
    mapping(:bill_length_mm, color=:species)

d1 = data(transform(penguins,
                    :bill_length_mm => ByRow(x -> 0.02 * rand()) => :ys))

p4b = d1 * visual(Scatter) * mapping(:bill_length_mm, :ys, color=:species)
p4 = p4a + p4b;
```

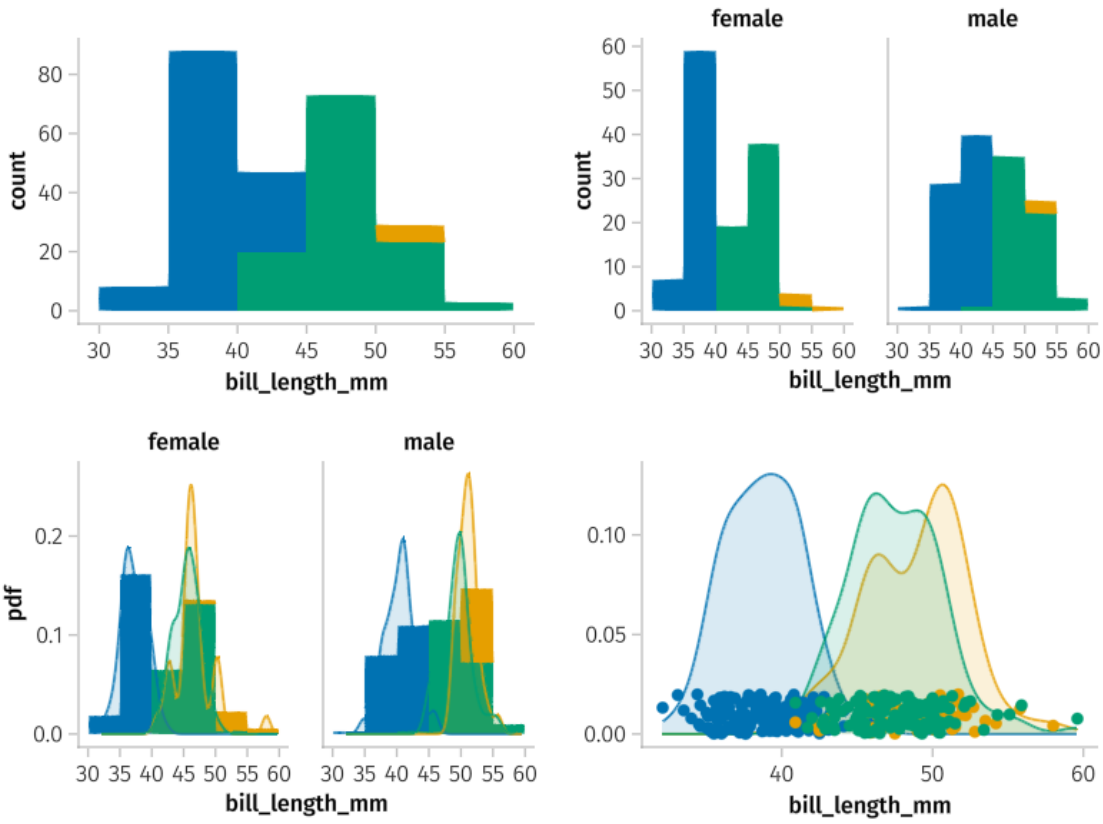
6.1.5 Quantile-normal plots

The `QQNorm` and `QQPlot` visuals are used to make quantile-quantile plots; `QQNorm` expects a mapping to `:x` (first position) whereas `QQPlot` expects mappings to `:x` and `:y` (the first two positions).

The following will give a visual check if bill length is normally distributed, which seems to indicate slightly shorter tails than expected

[1] The `Makie` density could be accessed through `visual(Density)` without module qualification. The `density` function in `AlgebraOfGraphics` has a nice transparency feature which makes its use desirable.

Figure 6.3: Histogram and Density plots.



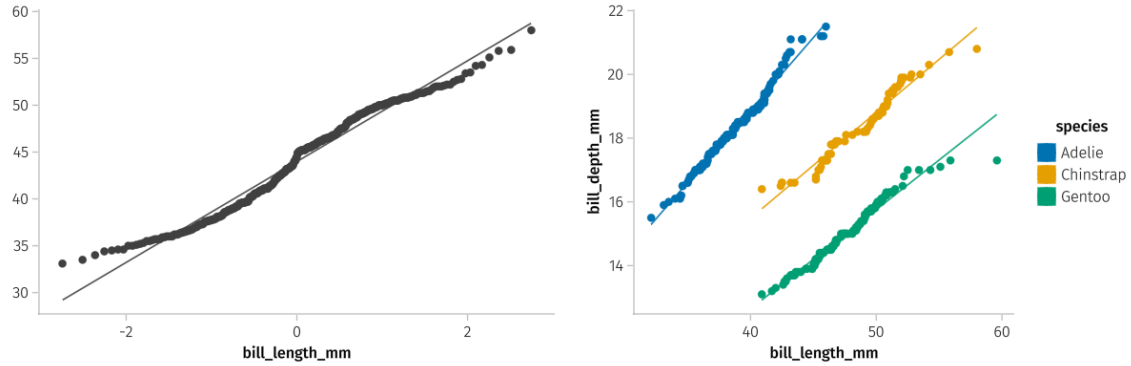
```
p1 = data(penguins) * visual(QQNorm, qqline=:fit) *
    mapping(:bill_length_mm);
```

The following will give a visual check if bill length has a similarly *shaped* distribution as bill depth, in this case with each species highlighted:

```
p2 = data(penguins) * visual(QQPlot, qqline=:fit) *
    mapping(:bill_length_mm, :bill_depth_mm, color=:species);
```

Both are shown in Figure 6.4.

Figure 6.4: Quantile-quantile plots. The left graphic uses a reference normal distribution (through `QQNorm`), the right one uses `QQPlot` to compare the distribution of two variables after grouping by `species`.



6.2 Bivariate relationships

Scatterplots with trend lines are easily produced within the `AlgebraOfGraphics` framework: the `Scatter` visual creates scatter plots; for trend lines there is the `smooth` transformation to fit a loess line, and the `linear` transformation to fit linear models.

This first set of commands shows how to fit a smoother (upper left graphic in Figure 6.5). The `smooth` function has arguments which pass on to `Loess.loess`.

```
layers = visual(Scatter) + smooth()
p1 = d * layers * mapping(:bill_length_mm, :bill_depth_mm);
```

The `linear` function draws the fitted regression line and shades an interval automatically (the `interval` argument). Linear prediction under model assumptions provides a means to identify confidence intervals for the *mean* response (the average value were the covariates held fixed and the response repeatedly samples) and for the *predicted* response for a single observation. The latter are wider, as single observations have more variability than averages of observations. A value of `nothing` suppresses this aspect.

This next set of commands shows (upper-right figure of Figure 6.5) one way to add a linear regression line. As the mapping for `linear` does not include the grouping variable, (`color`) the line is based on all the data:

```
d1 = d * mapping(:bill_length_mm, :bill_depth_mm)
p2a = d1 * visual(Scatter) * mapping(color=:species)
p2b = d1 * linear()
p2 = p2a + p2b;
```

Whereas with this next specification, `color` is mapped for both the `linear` transformation and the `Scatter` visual. This groups the data and separate lines are fit to each. We can see (lower-left figure of Figure 6.5) that whereas the entire data shows a negative correlation, the cohorts are all positively correlated, an example of Simpson's paradox.

```
layers = visual(Scatter) + linear()
p3 = d1 * layers *  mapping(color=:species);
```

Adding `layout=:sex` shows more clearly (lower-right figure of Figure 6.5) that each group has a regression line fit, that is the multiplicative model is fit.

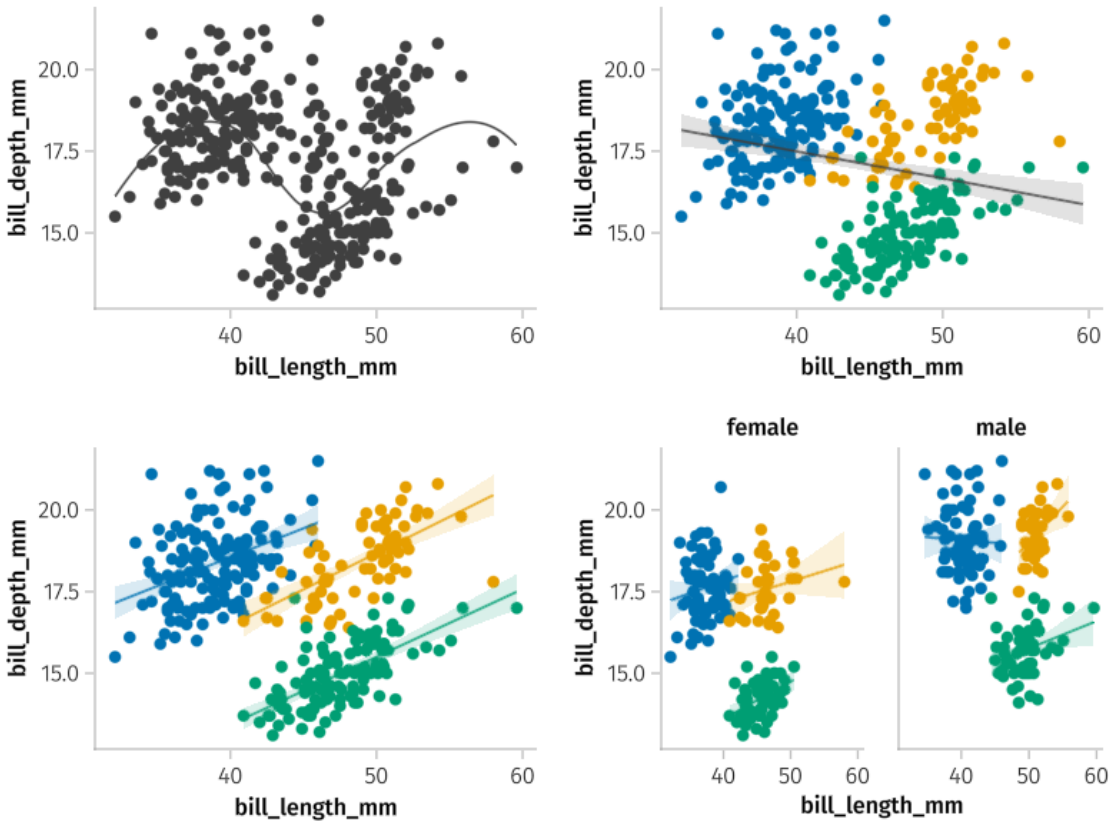
```
p4 = d1 * layers *  mapping(color=:species, layout=:sex);
```

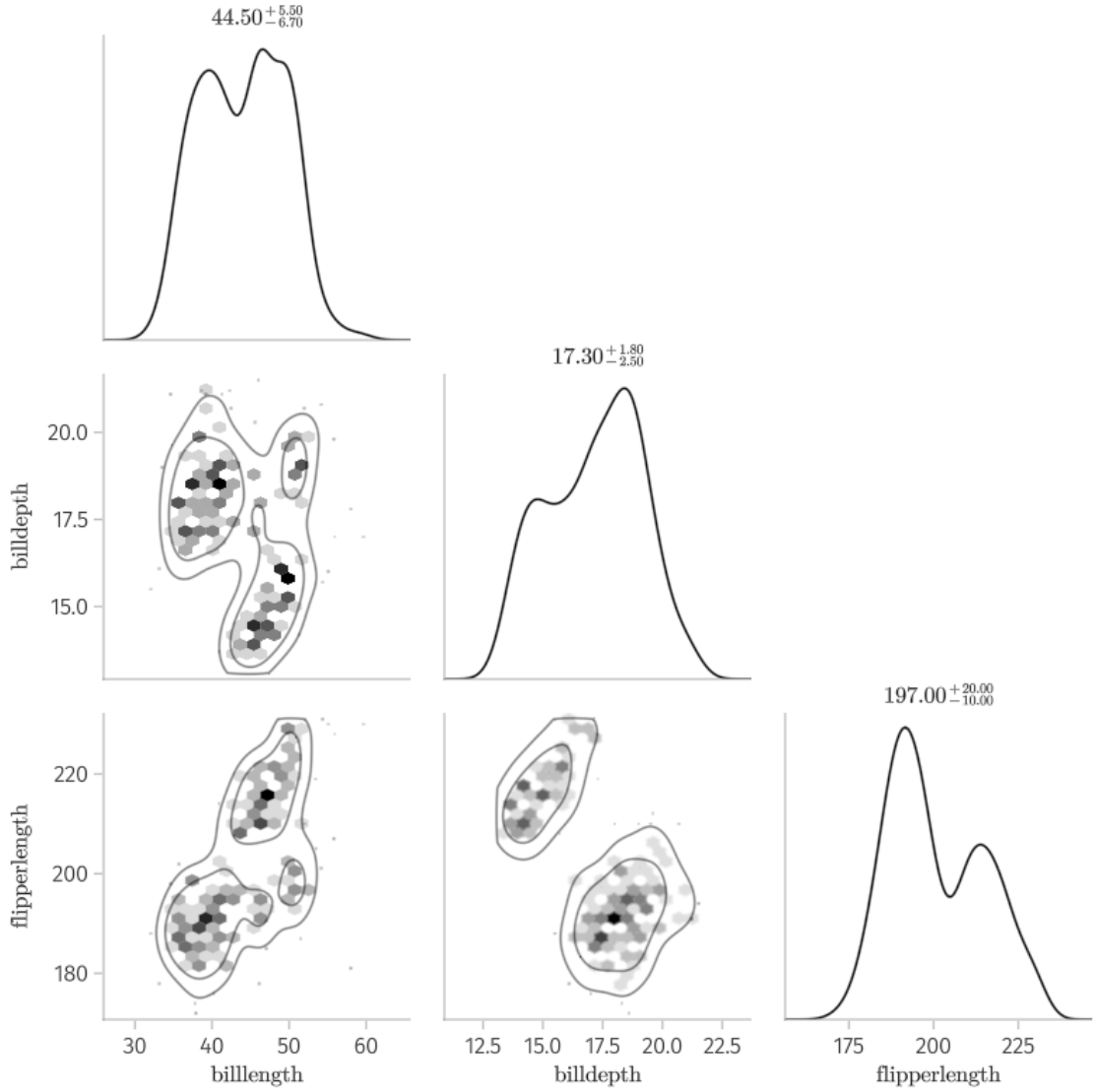
6.2.1 Corner plot

A corner plot, as produced by the `PairPlots` package through its `pairplot` function, is a quick plot to show pair-wise relations amongst multiple numeric values. The graphic uses the lower part of a grid to show paired scatterplots with, by default, contour lines highlighting the relationship. On the diagonal are univariate density plots.

```
using PairPlots
nms = names(penguins, 3:5)
p = select(penguins, nms .=> replace.(nms, "_mm" => "", "_" => " ")) # adjust names
pairplot(p)
```

Figure 6.5: Scatter plots of bill depth by bill width produced by varying specifications.





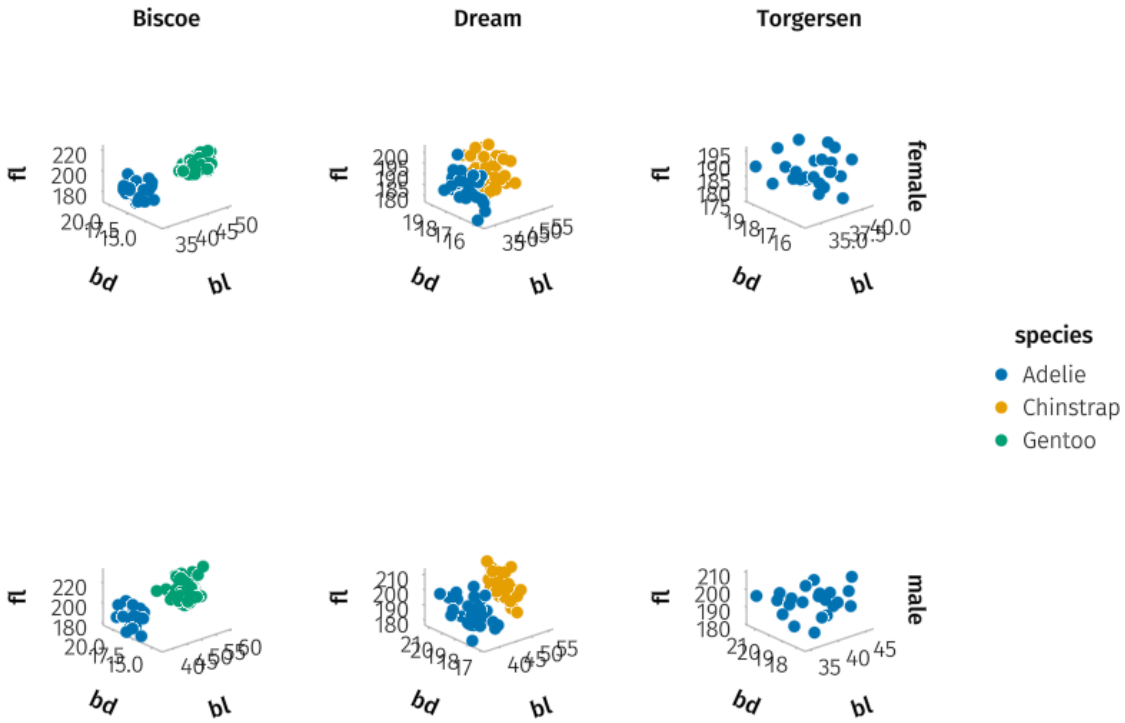
6.2.2 3D scatterplots

A 3-d scatter plot of 3 numeric variables can be readily arranged, with just one unexpected trick:

- The `mapping` object should contain an `x`, `y`, *and* `z` variable specification with numeric variables.
- The `draw` call should include an `axis = (type = Axis3,)` call, specifying that a 3D (`Makie`) axis should be used in the display.

```
d = data(penguins)
p = d * mapping(:bill_length_mm => :bl,  :bill_depth_mm => :bd,  :flipper_length_mm=>:fl; color=:species,
              row=:sex, col=:island)
draw(p, axis=((type=Axis3,)))
```

Figure 6.6: 3D scatter plots of bill length, bill depth, and flipper length with faceting by island and sex variables.



6.3 Categorical data

The distribution of the surveyed species is not the same. A bar chart can illustrate (upper-left graphic of Figure 6.7). The `frequency` transform does the counting:

```
p1 = d * frequency() * mapping(:species);
```

Two categories can be illustrated, we need `dodge` set here to avoid overplotting of the bars. In this example,

following the `AlgebraOfGraphics` tutorial, we add in information about the island. This shows (upper-right graphic of Figure 6.7) that two species are found on just 1 island, whereas Adelie is found on all three.

```
p2 = d * frequency() *
    mapping(:species, color=:island, dodge=:island);
```

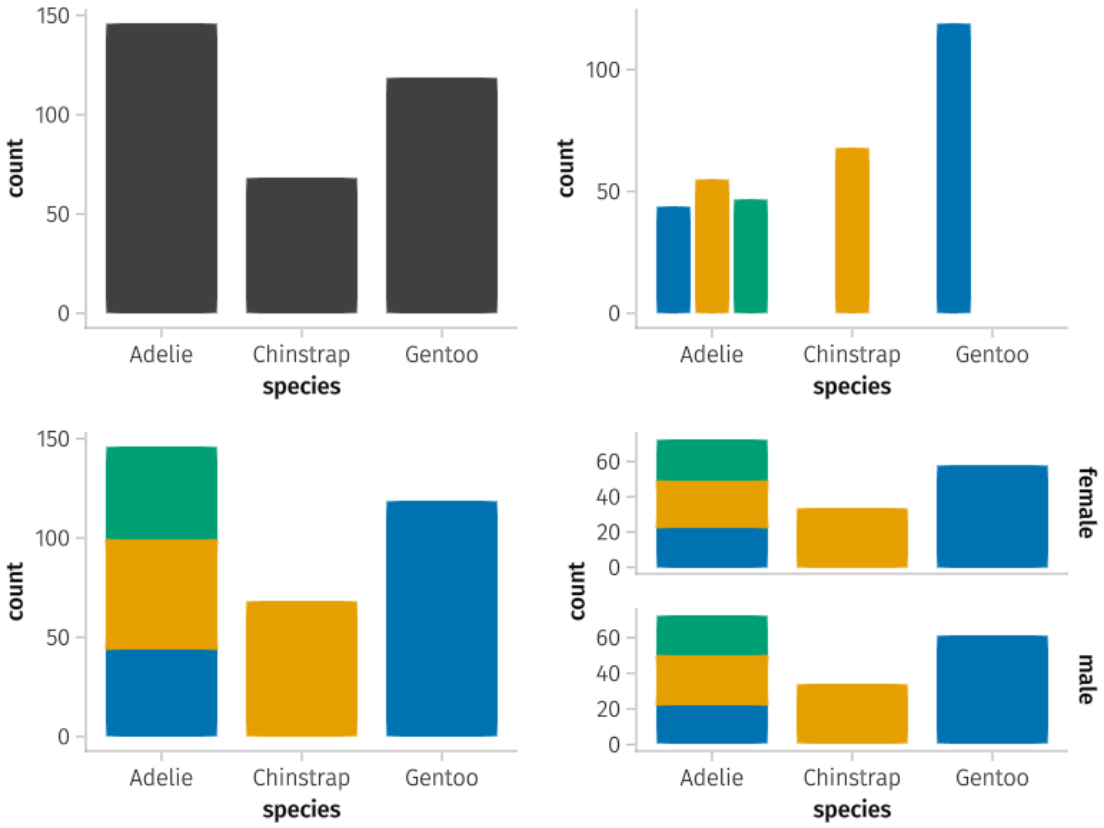
Using `stack` in place of `dodge` presents a stacked bar chart (lower-left graphic of Figure 6.7):

```
p3 = d * frequency() *
    mapping(:species, color=:island, stack=:island);
```

A third category can be introduce using `layout`, `col`, or `row` (lower-right graphic of Figure 6.7):

```
p4 = d * frequency() *
    mapping(:species, color=:island, stack=:island) *
    mapping(row=:sex);
```

Figure 6.7: Scatter plots of bill depth by bill width produced by varying specifications.



Part II

Inference

Chapter 7

Probability distributions

Exploratory data analysis differs from statistics. A statistical model for a measurement involves a description of the random nature of the measurement. To describe randomness, the language of probability is used. Much of this language is implemented in the `Distributions` package of `Julia`, loaded below with other packages:

```
using Distributions
using StatsBase, DataFrames
using GLMakie, AlgebraOfGraphics
```

7.1 Probability

This section quickly reviews the basic concepts of probability.

Mathematically a probability is an assignment of numbers to a collection of sets of a probability space. These values may be understood from a model or through long term frequencies. For example, consider the tossing of a *fair* coin. By writing "fair" the assumption is implicitly made that each side (heads or tails) is equally likely to occur on a given toss. That is a mathematical assumption. This can be reaffirmed by tossing the coin *many* times and counting the frequency of a heads occuring. If the coin is fair, the expectation is that heads will occur in about half the tosses.

The mathematical model involves a formalism of sample spaces and events. There are some subtleties due to infinite sets, but we limit our use of events to subsets of finite or countably infinite sets or intervals of the real line. A probability *measure* is a function P which assigns each event E a number with:

- $0 \leq P(E) \leq 1$
- The probability of the empty event is $P(\emptyset) = 0$, the probability of the the sample space is $P(\Omega) = 1$.
- If events $E_1, E_2, \ldots$ are *disjoint* then the probability of their *union* is the sum of the individual probabilities.

A *random variable*, X, is a function which takes an *outcome* (an element of an event) in a sample space and assigns a number. Random variables naturally generate *events* through sets of the type $\{X = k\}$ for some

k. This event being *all* outcomes for which X would be k. Similarly, $\{X \le a\}$ for some a describes an event. These are the *typical* events of statistics.

The rules of probability lead to a few well used formulas: $P(X \le a) = 1 - P(X > a)$ and *for* **discrete** random variables $P(X \le k) = \sum_{j \le k} P(X = j)$. The former is illustrated in the left graphic of Figure 7.1.

A *distribution* of a random variable is a description of the probabilities of events generated by a random variable. For our purposes, it is sufficient to describe events of the type $\{X \le a\}$, $\{X < a\}$, or $\{X = a\}$, others being formed through intersections and unions. A valid description of the *cumulative distribution function* $F(a) = P(X \le a)$ describes the distribution of a random variable.

There are 2 types of distributions where a related function describes the distribution, discrete and continuous distributions.

A **discrete** random variable is one which has $P(X = k) > 0$ for at most a finite or *countably* infinite set of numbers. For example, if X is the number of coin tosses producing heads in n tosses, then the finite ks are $0, 1, 2, \dots, n$. Whereas, if X is the number of coin tosses needed to toss *one* head. The k would be $1, 2, \dots$ (with no upper bound, as one could get extremely unlucky). For *discrete* random variables it is enough to describe $f(k) = P(X = k)$ for the valid ks. The function $f(k)$ is called the pdf (probability distribution function). An immediate consequence is $\sum_k f(k) = 1$ and $f(k) \ge 0$.

A **continuous** random variable is described by a function $f(x)$ where $P(X \le a)$ is given by the *area* under $f(x)$ between $-\infty$ and a. The function $f(x)$ is called the pdf (probability density function). An immediate consequence is the *total* area under $f(x)$ is 1 and $f(x) \ge 0$.

When defined, the pdf is the basic description of the distribution of a random variable. It says what is *possible* and *how likely* possible things are. For the two cases above, this is different. In the discrete case, the possible values are all k where $f(k) = P(X = k) > 0$, but not all values are equally likely unless $f(k)$ is a constant. For the continuous case there are **no** values with $P(X = k) > 0$, as probabilities are assigned to area, and the corresponding area to this event, for any k, is 0. Rather, the possible values are those for which $f(x) > 0$ and intervals on the x axis where there is more area are more likely.

A data set in statistics, $x_1, x_2, \dots, x_n$, is typically modeled by a collection of random variables, $X_1, X_2, \dots, X_n$. That is, the random variables describe the *possible* values that can be collected, the values $(x_1, x_2, \dots)$ describe the actual values that were collected. Put differently, random variables describe what can happen *before* a measurement, the values are the result of the measurement.

The *joint* cumulative distribution is the probability $P(X_1 \le a_1, X_2 \le a_2, \dots X_n \le a_n)$. A common assumption made for statistics is that each of the random variables is

- *identically distributed*, meaning $f(a) = P(X_i \le a) = P(X_j \le a)$ for each pair (i, j).
- *independent*, which means that knowing the value of X_i does not effect the probabilities regarding values for X_j, $j \ne i$. (If you know a first toss of a fair coin is heads, it doesn't change the odds that the second toss will be heads.)

With these two assumptions, the random variables $X_1, X_2, \cdots, X_n$ are termed an *iid random sample*. For an iid random sample, this simplification applies: $P(X_1 \le a_1, X_2 \le a_2, \dots X_n \le a_n) = F(a_1) \cdot F(a_2) \cdots F(a_n)$ (independence means "multiply").

In general, the distribution of a random variable may be hard to describe. For example, suppose the random variable is the sample median, M, of $X_1, X_2, \cdots, X_n$. With the assumption of an iid random sample, a formula

for the *distribution* of M can be worked out in terms of the underlying pdf, f and cdf, F. But without that assumption, it becomes intractable, in general.

When a distribution can not be fully identified, it can still be described. A few basic summaries of a probability distribution are:

- The *mean* or average value. For a discrete distribution this is $\sum_k kP(X = k)$, which is a *weighted* average of the possible values, weighted by how likely they are. For a continuous distribution, a similar formula using calculus concepts applies. Both can be viewed from a center of mass perspective. The symbol μ or $E(X)$ (expectation) is used to represent the mean.[1]
- The *variance* of a probability distribution describes the spread; the *standard deviation* is the square root of the variance. For a random variable, the variance is described by the *average value* of the centered random variable, squared: $E((X - \mu)^2)$. The symbol σ^2 is used to represent the variance, σ then is the standard deviation. $VAR(X)$ is also used to represent the variance of a random variable, similarly $SD(X)$ is used for the standard deviation.

The transformation $X - \mu = X - E(X)$ centers the random variable X, so that $E(X - \mu) = 0$.

The transformation $Z = (X - \mu)/\sigma$, following the z-score, centers and *scales* the random variable X. The random variable Z has $E(Z) = 0$ and $VAR(Z) = 1$.

For a *random sample* $X_1, X_2, \ldots, X_n$ the sum $S = \sum_k X_k$ has the property that $E(S) = \sum_k E(X_k)$ ("expectations add"). This is true even if the sample is not iid. For example, finding the average number of heads in 100 tosses of a fair coin is easy, it being $100 \cdot (1/2) = 50$, the $1/2$ being the expectation of a single heads where $X_i = 1$ if heads, and $X_i = 0$ if tails.

While $E(X_1 + X_2) = E(X_1) + E(X_2)$, as expectations are *linear* and satisfy $E(aX + bY) = aE(X) + bE(Y)$, it is not the case that $E(X_1 \cdot X_2) = E(X_1) \cdot E(X_2)$ in general – though it is true when the two random variables are independent. As such, the variance of $S = \sum_k X_k$ is:

$$VAR(\sum_k X_k) = \sum_k VAR(X_k) + 2 \sum_{i<j} COV(X_i, X_j),$$

where the *covariance* is $COV(X_i, X_j) = E((X_i - E(X_i)) \cdot (X_j - E(X_j)))$. If, the random sample is iid, then the covariances are 0 and the variance of a sum is the sum of the variances.

7.1.1 Statistical language

In statistics we have seen a random sample is a sequence of random variables $X_1, X_2, \ldots, X_n$. Assume this is an iid random sample.

The *population* is the common distribution of each random variable in the random sample. Populations have a pdf and cdf, often denoted $f_X(x)$ and $F_X(x)$. Populations are summarized by *parameters*, such as the mean (μ) or the standard deviation (σ).

A statistic is some summary of a random sample. For example, the median, or middle value, or the sample mean $\bar{X} = (X_1 + X_2 + \cdots X_n)/n$. Statistics are also *random variables* and so are described by a distribution (when computable) or summarized by values such as the mean or standard deviation.

[1] Notationally, $E(X)$ is used for the mean of a random variable, μ is used for the mean of a probability distribution, though we use them interchangeably.

For the sample mean from an iid random sample, the above says:

$$\begin{aligned} E(\bar{X}) &= E\left(\frac{X_1 + X_2 + \cdots + X_n}{n}\right) \\ &= \frac{1}{n}\sum_k E(X_k) \\ &= \frac{1}{n}\sum_k \mu = \mu; \\ VAR(\bar{X}) &= \frac{1}{n^2}\cdot\left(\sum_k VAR(X_k) + 2\sum_{i<j} COV(X_i, X_j)\right) \\ &= \frac{1}{n^2}\sum_k VAR(X_k) \\ &= \frac{1}{n^2} n\cdot\sigma^2 = \frac{\sigma^2}{n}. \end{aligned}$$

The standard deviation then is $SD(\bar{X}) = \sigma/\sqrt{n}$.

In short, the expected value of the mean is the expected value of the *population*; the variance of the mean (of an iid random sample) is the variance of the population *divided* by the sample size, n. The latter speaks to variability: there is more variability in a single random value than in an average of the random values. The distribution of $\bar{X}$ can be expressed through formulas in terms of $f_X(x)$, but is *well approximated* as n gets large by a distribution characterized by its mean and standard deviation, as will be seen.

There are parallels between random variables and data sets:

- The random sample $X_1, X_2, \ldots, X_n$ is realized in a data set by values $x_1, x_2, \ldots x_n$.
- The distribution, $f_X(x)$, is reflected in the data set.
- The distribution, $f_X(x)$ is *typically* described by key *parameters*
- The data set is *typically* summarized by sample statistics.

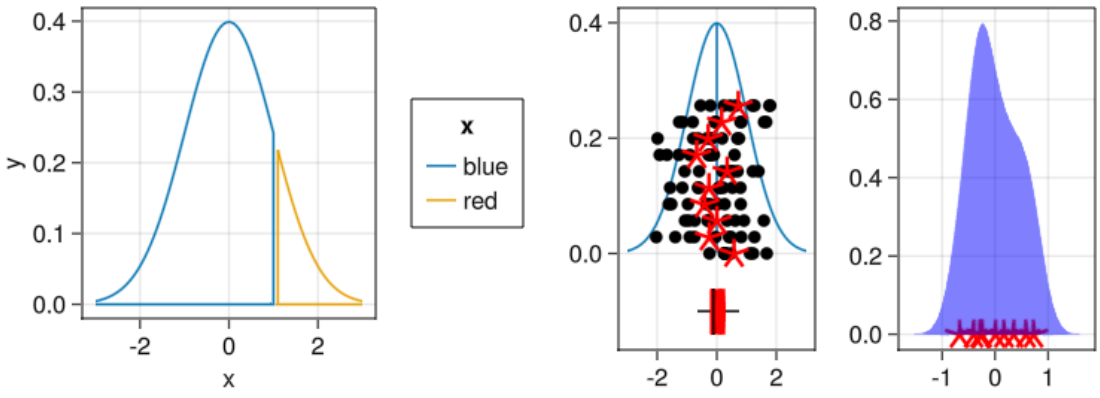
Statistical inference makes statements using the language of probability about the *parameters* in terms of various sample statistics.

An intuitive example is the tossing of a fair coin modeling heads by a 1 and tails by a 0 then we can *parameterize* the distribution by $f(1) = P(X = 1) = p$ and $f(0) = P(X = 0) = 1 - P(X = 1) = 1 - p$. This distribution is summarized by $\mu = p$, $\sigma = \sqrt{p(1-p)}$. A *fair* coin would have $p = 1/2$. A sequence of coin tosses, say H,T,T,H,H might be modeled by a sequence of iid random variables, each having this distribution. The we might expect a few things, where $\hat{p}$ below is the proportion of heads in the n tosses:

- A given data set is not random, but it may be viewed as the result of a random process and had that process been run again would likely result in a different outcome. These different outcomes may be described probabalistically in terms of a distribution.
- If n is large enough, the sample proportion $\hat{p}$ should be *close* to the population proportion p.
- Were the sampling repeated, the variation in the values of $\hat{p}$ should be smaller for larger sample sizes, n.

These imprecise statements can be made more precise, as will be seen. The right graphic in Figure 7.1 shows this for a bell-shaped population. Each of the 10 rows of dots shows a random sample of size $n = 9$ which is summarized by its sample mean with a red star. The boxplot summarizes the red stars. The spread of the bell shaped population is much greater than the spread of the boxplot summarizing the sample means. By the above, we'd expect the standard deviation of the population to be $\sqrt{9}$ times bigger than the standard deviation of the sample mean statistic.

Figure 7.1: The left figure shows how $P(X \leq a) = 1 - P(X > a)$. The right graphic illustrates a population in blue; several samples in black summarized by their sample mean (red star). The boxplot is of the sample means.



However, the center of the population and the center of the boxplot are nearly the same. This is because the mean of the population is the same as the mean of the sample-mean statistic.

Inferential statistics is similar: A population is a distribution summarized by parameters. A random sample drawn from a population is summarized by statistics. A statistic summarizes just one sample, but the language of probability is used to infer from that one sample, statements about the parameters which would be apparent were there many different random samples.

7.2 The Distributions package

While populations are in general described by a cdf, populations which are discrete or continuous are more naturally described by their pdf. There are many standard pdfs used to describe different types of randomness. Many of these are supported in the `Distributions` package. This section reviews several.

```
using Distributions
```

7.2.1 Bernoulli, Binomial, Geometric

The simplest non-trivial distribution is that used to model a coin toss: $P(X = 1) = p$ and $P(X = 0) = 1 - p$. We see this is *parameterized* by p, the probability of heads (say). This also happens to be the mean of this distribution. By the rules of probability, we must have $0 \leq p \leq 1$.

In the `Distributions` package, named distributions are given a type, in this case `Bernoulli` which requires a value of p to be formed. The following code uses the type and confirms the mean of this distribution is indeed p:

```
p = 0.25
Be = Bernoulli(p)
mean(Be)
```

```
0.25
```

The `Bernoulli(p)` distribution is very simple, but combinations of $Bernoulli(p)$ random variables give rise to many interesting distributions. For example, the number of heads in 10 coin tosses could be modeled by the sum of 10 Bernoulli random variables. The distribution of this random variable is described by the `Binomial` distribution. It has two parameters: p for the Bernoulli 10 for the number of tosses and is implemented with the `Binomial(n,p)` type. There are several summary statistics for the types supported by `Distributions`, including: `mean`, `median`, `std`:

```
n, p = 10, 1/2
B = Binomial(n, p)
mean(B),  median(B), std(B)
```

```
(5.0, 5, 1.5811388300841898)
```

The mean and standard deviation for the Binomial can be easily computed when this statistic is viewed as a sample mean of 10 Bernoulli random variables (np and $\sqrt{np(1-p)}$).

More is known. For example, the value of this random variable can only be between 0 and 10. The `extrema` function combines both the `minimum` and `maximum` functions to give this:

```
extrema(B)
```

```
(0, 10)
```

The `insupport` function is similar, return `true` or `false` if a value, `x`, is in the support of the distribution, that is if `x` is a *possible* value:

```
insupport(B, 5), insupport(B, 15), insupport(B, 5.5)
```

```
(true, false, false)
```

We see only `5` is a possible value. Its probability, $P(X = 5)$ for this discrete distribution, is computed by `pdf`:

```
pdf(B, 5)
```

```
0.24609375000000022
```

The *cumulative distribution function* is computed by `cdf` with the same calling style.

The cumulative distribution, $F(a)$ answers $x = P(X \le a)$ for a given value of a. It's inverse operation, answering a in $x = P(X \le a)$ for a given x, is returned by `quantile`, with the same calling style.

A related question is the number of coin tosses needed to get the *first* heads. If X is this distribution, the the event $\{X > k\}$ can be described as the first k tosses were tails. This probability is directly computable. The distribution of this number depends only on the parameter p and is called the *Geometric distribution*.

For example, we can see here the mean, and standard deviation, and also that the the number is unbounded:

```
p = 1/2
G = Geometric(p)
mean(G), std(G), extrema(G) # 1/p, 1/√p, (0, ∞)
```

```
(1.0, 1.4142135623730951, (0, Inf))
```

7.2.2 Uniform and DiscreteNonParametric distributions

A *discrete* uniform random variable on $a, a+1, \dots, b$ assigns equal weight to each value. Hence, each possible value is equally likely. The `DiscreteUniform(a,b)` type models this distribution. For example, to model the roll of a 6-sided die, we might have:

```
D = DiscreteUniform(1,6)
mean(D), std(D)
```

```
(3.5, 1.707825127659933)
```

More generally, a distribution which assigns weight p_k to values $x_1, x_2, \dots, x_n$ is modeled by the `DiscreteNonParametric(xs, ps)` distribution. For example, Benfords "law" is an observation that the first non-zero digit of many data sets follows a certain pattern. For this the *possible* values are 1 through 9 and their probabilities are $f(k) = P(X = k) = \log_{10}(k+1) - \log_{10}(k)$. We can model this with:

```
xs = 1:9
ps = log10.(xs .+ 1) - log10.(xs)
B = DiscreteNonParametric(xs, ps)
```

```
DiscreteNonParametric{Int64, Float64, UnitRange{Int64}, Vector{Float64}}(
support: 1:9
p: [0.3010299956639812, 0.17609125905568124, 0.12493873660829996, 0.09691001300805646, 0.07918124604762478, 0.06
)
```

We can answer questions like the mean, the standard deviation, and what is the probability the number is `5` or less with:

```
mean(B), std(B), cdf(B, 5)
```

```
(3.4402369671232065, 2.4609982997506656, 0.7781512503836436)
```

In `Distributions` the `Categorical` type can alse have been used to construct this distribution, it being a special case of `DiscreteNonParametric` with the `xs` being $1, \dots, k$.

The multinomial distribution is the distribution of counts for a sequence of n iid random variables from a `Categorical` distribution. This generalizes the binomial distribution. Let X_i be the number of type i in n samples. Then $X_1 + X_2 + \cdots + X_k = n$, so these are not independent. They have mean $E(X_i) - np_i$, variance $VAR(X_i) = np_i(1 - p_i)$, like the binomial, but covariance $COV(X_i, X_j) = -np_ip_j, i \neq j$. (Negative, as large values for X_i correlate with smaller values for X_j when $i \neq j$.)

7.2.3 The continuous uniform distribution

The continuous uniform distribution models equally likely outcomes over an interval $[a, b]$. (The endpoints possibly open, as mathematically they have no chance of being selected.) For example, the built-in `rand` function for Float64 values returns random numbers which are modeled by being uniform over $[0, 1)$. (That they can be `0` but not `1` is a reality of how computers and mathematical models aren't always exactly the same, but rather the model is a abstraction of the implementation.)

The `Uniform(a,b)` type models these numbers. Here we see `rand` being used to randomly sample 3 uniform numbers using a more general interface than the use of `rand` in base `Julia`:

```
U = Uniform(0, 1)
rand(U, 3)
```

```
3-element Vector{Float64}:
 0.6412073745180317
 0.9150628175432435
 0.30814569216009957
```

If U is Uniform on $[0, 1]$ then $Y = aU + b$ is uniform on $[b, b + a]$. The difference in means is shifted by $b+E(U)$, the difference in standard deviations is scaled by a. We can do the algebra of linear transformations using overloaded operations:

```
a, b = 2, 3
Y = a * U + b
mean(U) == mean(Y) - (b + mean(U)) == (1 + 0)/2, std(U) == std(Y)/a  == sqrt(1/12)
```

```
(true, true)
```

7.2.4 The Normal distribution

The most important distribution in statistics is called the normal distribution. This is a bell shaped distribution completely described by its mean and standard deviation, `Normal(mu, sigma)` (though some math books use the variance, σ^2 for the second parameter. The *standard* normal has $\mu = 0$ and $\sigma = 1$. Standard normal random variables are generically denoted by Z:

```
Z = Normal(0, 1)
mean(Z), std(Z)
```

```
(0.0, 1.0)
```

There are many facts about standard normals that are useful to know. First we have the 3 rules of thumb – 68, 95, 99.7 – describing the amount of area in $[-1,1]$, $[-2,2]$, and $[-3,3]$. We can see these from the `cdf` with:

```
between(Z, a, b) = cdf(Z,b) - cdf(Z,a)
between(Z, -1, 1), between(Z, -2, 2), between(Z, -3, 3)
```

```
(0.6826894921370861, 0.9544997361036416, 0.9973002039367398)
```

These values are important as the z scores of different data sets are often assumed to be normal or approximately normal, so, for example, about 95% of such a data set should have z scores within -2 and 2.

The transformation $Y = \mu + \sigma \cdot Z$ will produce a normal with mean μ and standard deviation σ, modeled more directly by `Normal(mu, sigma)`.

A common question in introductory statistics is what values a and b correspond to $(1 - \alpha) \cdot 100\%$ of the area under the pdf of Y? There are infinitely many such answers, but only one pair that is symmetric about the mean, μ. This can be found with the help of `quantile` *and* the observation that *since* the pdf of Y is symmetric, there will be area $1 - \alpha$ **plus** $\alpha/2$ to the left of b, a question tailor-made for `quantile`. From b, a can be found from symmetry, it being equidistant from the mean as b is.

For example:

```
alpha = 0.05
mu, sigma = 10, 20
Y = Normal(mu, sigma)
b = quantile(Y, 1 - alpha + alpha/2)
a = mu - (b - mu)
(a, b)
```

```
(-29.19927969080115, 49.19927969080115)
```

The normal distribution was seen as an *excellent* approximation to the distribution of heads in a fixed number of coin tosses, more generally described by the binomial distribution. Suppose we consider the Binomial with n and p such that np and $n \cdot (1-p)$ are both 10 or more. Then the $Binomial(n,p)$ distribution has $\mu = np$ and $\sigma = \sqrt{np(1-p)}$. The cdf of this distribution is *very* well approximated by the cdf of the $Normal(\mu, \sigma)$ distribution, and *asymptotically* they are equal, as noted by De Moivre in 1733. Here we see how well approximated:

```
n, p = 100, 1/3
mu, sigma = n*p, sqrt(n*p*(1-p))
B, Y = Binomial(n,p), Normal(mu, sigma)
ks = 0:100
findmax(abs(cdf(B, k) - cdf(Y, k)) for k in ks)
```

```
(0.046989293624614625, 34)
```

This shows an error of not more than 0.0469... in approximating the binomial with the normal. The *continuity correction* adjusts for the discrete nature of the binomial by comparing k to $k + 1/2$ for the normal. This is more accurate, as can be seen:

```
findmax(abs(cdf(B, k) - cdf(Y, k+1/2)) for k in ks)
```

```
(0.004701503073942237, 34)
```

Of course, computationally it is just as easy to call `cdf(B, k)` as it is to call `cdf(Y, k + 1/2)`, so the advantage here is theoretical for computationally tractable values of n and k. This particular approximation is generalized in the central limit theorem.

7.2.5 The Chi-squared distribution

Let Z be a standard normal random variable. As seen, a linear transform of Z: $Y = \sigma Z + \mu$ will have distribution of $Normal(\mu, \sigma)$, a different transform $Y = Z^2$ can have its distribution computed using some tricks, e.g. ($P(Z^2 < a) = P(-\sqrt{a} < Z < \sqrt{a})$.) If we have an iid random sample, $Z_1, Z_2, \dots, Z_n$ the distribution of $\chi^2 = Z_1^2 + Z_2^2 + \cdots + Z_n^2$ is of interest. (It may be viewed as the distance squared of a randomly chosen point in space, say.) The distribution of χ^2 is the *Chi-squared distribution* with n degrees of freedom and is implemented in the `Chisq(n)` type.

7.2.6 The T and F distributions

There are two main distributions which arise in the distribution of many sample statistics related to the linear regression model, these are the T-distribution of Student and the F-distribution of Fischer. Thesesare *parameterized* by degrees of freedom, which are more naturally discussed with sampling distributions.

Consider two independent random variables, a standard normal Z and a Chi-squared random variable, Y, with ν degrees of freedom. The T-distribution with ν degrees of freedom is the distribution of the ratio:

$$T = \frac{Z}{\sqrt{Y/\nu}}.$$

The T-distribution is a symmetric, bell-shaped distribution like the normal distribution, but has wider, fatter tails hence typically produces larger values in a sample. This can be seen with a relatively small degrees of freedom as follows:

```
T5 = TDist(5)
maximum(abs, rand(T5, 100)), maximum(abs, rand(Z, 100))
```

```
(6.28323894174054, 2.904366928050913)
```

Figure 7.2 shows a quantile plot of $T(5)$ in the upper-right graphic. The long tails cause deviations in the pattern of points from a straight line.

The `skewness` method measures asymmetry. For both the T and the normal distributions this is 0. The `kurtosis` function measures *excess kurtosis* a measure of the size of the tails *as compared* to the normal. We can see:

```
skewness(T5), skewness(Z), kurtosis(T5), kurtosis(Z)
```

```
(0.0, 0.0, 6.0, 0.0)
```

A distribution with excess kurtosis exceeding the normal is call *leptokurtic*. These generally produce values with z scores larger than expected by the normal. A *platykurtic* distribution, on the other hand, would have in general fewer values in the tails and fewer values close to the mean of a normal. The uniform distribution us a good example of a playtkurtic distribution, both conditions have a test in `Distributions`:

```
isleptokurtic(T5), isplatykurtic(U)
```

```
(true, true)
```

For the T-distribution, when the degrees of freedom gets large, say bigger than 100, the distribution is well approximated by the normal. Here we see a similar comparison as above. For a quantile-normal plot see the upper-right graphic in Figure 7.2.

```
T100 = TDist(100)
xs = range(-3, 3, length=100)
findmax(abs(cdf(T100, x) - cdf(Z, x)) for x in xs)
```

```
(0.0015794482887208222, 25)
```

In Figure 7.2 the lower-left graphic is of the uniform distribution, the short tails, lead to a deviation from a straight line in this graphic; the lower-right graphic is of the skewed exponential distribution; the skew shows up in the lack of symmetry about the fitted line.

Consider now two, independent Chi-squared random variables Y_1 and Y_2 with degrees of freedom ν_1 and ν_2. The ratio

$$F = \frac{Y_1/\nu_1}{Y_2/\nu_2},$$

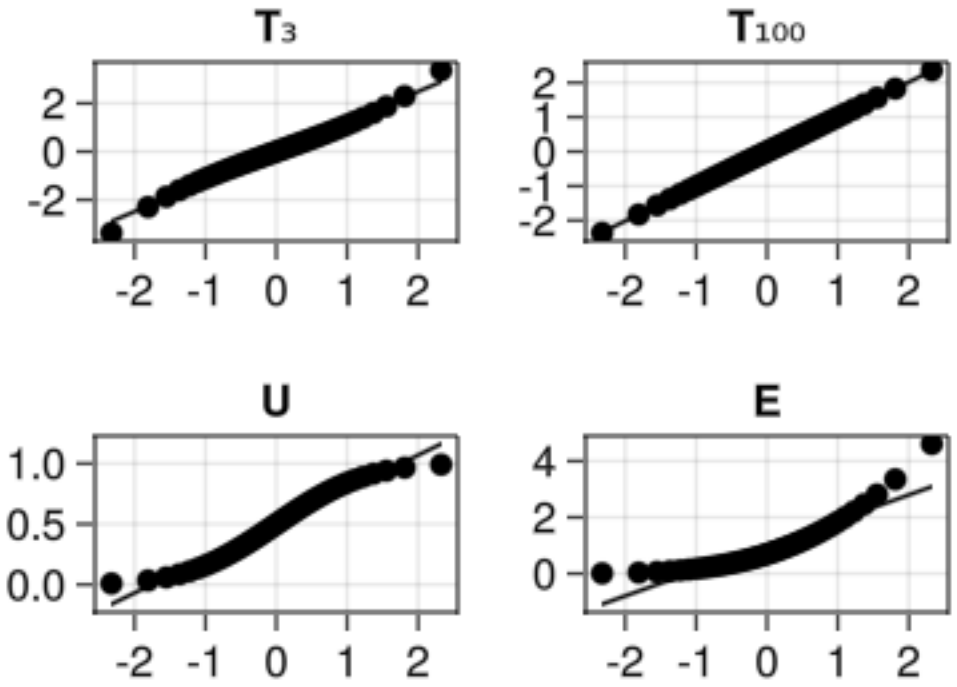
has a known distribution called the F-distribution with degrees of freedom ν_1 and ν_2.

This distribution has different shapes for different parameter values, with the shapes becoming more bell shaped as the two parameters get large.

```
kurtosis(FDist(5, 10)),kurtosis(FDist(5,100)),kurtosis(FDist(100,100))
```

```
(50.86153846153846, 3.1474470779483217, 0.7278883188966445)
```

Figure 7.2: Quantile-quantile plots for different distributions – T_5 is leptokurtic; T_{100} is approximately normal; U is platykurtic; and E is skewed.



7.3 Sampling distributions

The normal distribution, T distribution, and F distribution are important in statistics as they arise as sampling distributions of certain statistics.

7.3.1 The sample mean

For an iid random sample, from a population with mean μ and standard deviation σ we have seen that the sample mean, $\bar{X} = (X_1 + \cdots + X_n)/n$ is described by having mean $\mu_{\bar{X}} = \mu$ and standard deviation $\sigma_{\bar{X}} = \sigma/\sqrt{n}$.

What about the distribution itself? This depends on the underlying population.

If the underlying population is *normal*, then $\bar{X}$ is normal. This is true as the sum of two independent normal random variables is normally distributed. That is $\bar{X}$ is bell shaped, centered at the same place as the distribution of any of the X_is but has a *narrower* shape, as the standard deviation is smaller by a factor of $1/\sqrt{n}$.

7.3.1.1 Central limit theorem

If the population is not normal, then the *central limit theorem* informs us that for large enough n the following is true:

$$P(\frac{\bar{X} - \mu}{\sigma/\sqrt{n}} \leq z) \approx P(Z \leq z),$$

where Z is a standard normal random variable. That is centering $\bar{X}$ by *its* mean and then scaling by its standard deviation results in a random variable that is approximately a standard normal if n is large enough.

The central limit theorem has some assumptions on the underlying population. The assumption that a mean and standard deviation exist are sufficient for application.

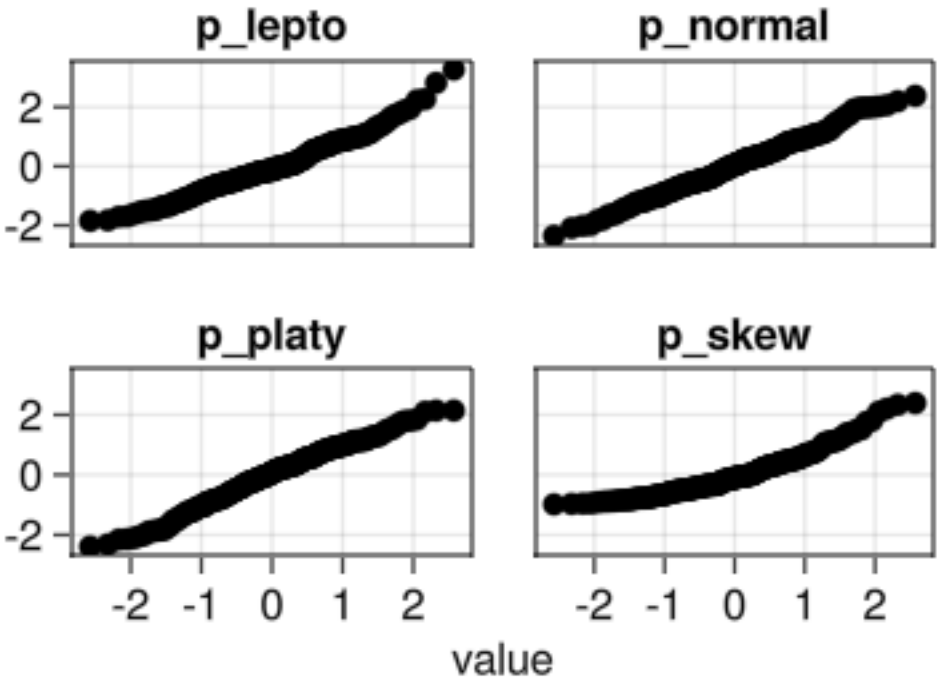
How large n needs to be depends on the underlying shape of the *population*: if the population has long tails (leptokurtic) or is very skewed, then n must be much larger than if the population is symmetric and platykurtic.

Figure 7.3 shows simulations of the distribution of $\bar{X}$ for $n = 10$ for different populations. The key to simulations is the ability to draw random samples from the distribution, which is provided through the `rand` method for a distribution type, with `rand(D,n)` returning a random sample of size n.

```
xbar(D, n, N=200) = [mean(rand(D, n)) for _ in 1:N]
sxbar(D, n, N=200) = (xbar(D,n,N) .- mean(D)) / (std(D)/sqrt(n))

p_normal = sxbar(Normal(0,1), 10)
p_lepto  = sxbar(TDist(3), 10)
p_skew   = sxbar(FDist(2,5), 10)
p_platy  = sxbar(Uniform(0,1), 10)

df = DataFrame(; p_normal, p_lepto, p_skew, p_platy)
d = data(stack(df))
```

```
p = d * visual(QQNorm) * mapping(:value, layout = :variable)
draw(p)
```

Figure 7.3: Quantile-normal plots of different random samples of $\bar{X}$ for $n = 10$. The underlying popluation is either normal, short tailed (playkurtic), long tailed (leptokurtic), or skewed. For the normal and short tailed populations the distribution of $\bar{X}$ is approximately normal for this small value of n; for the other two populations, larger samples sizes are needed to see the aymptotic normality guaranteed by the central limit theorem.



7.3.1.2 The T-statistic

The central limit theorem requires standardizing by the population standard deviation. At times this is neither known or assumed to be known. However, the standard error may be known:

> The *standard error* of a statistic is the standard deviation of the statistic *or* an estimate of the standard deviation.

For $\bar{X}$, the standard deviation is $\sigma/\sqrt{n}$ the standard error is $SD(\bar{X})/\sqrt{n}$, where, for a sample, the sample standard deviation, s, is used to estimate the population standard deviation, σ.

The scaling in the central limit theorem can be redone using the standard error, $S/\sqrt{n}$:

$$T = \frac{\bar{X} - \mu}{SE(\bar{X})} = \frac{\bar{X} - \mu}{S/\sqrt{n}},$$

where S is the sample standard deviation of the data set $X_1, X_2, \ldots, X_n$:

$$S = \sqrt{\frac{1}{n-1}\sum_i^n (X_i - \bar{X}_i)^2}.$$

The variability in the sampling means that the standard error will occasionally be much smaller than the population standard deviation, as this is in the denominator, the random variable T will have longer tails than a standard normal. That is, the sampling distribution of T should be leptokurtic, and it is:

> For a normal population, the sampling distribution of T is the T-distribution with $n-1$ degrees of freedom.

In general, a T-statistic for a parameter, β, is the difference between the estimate for the parameter ($\hat{\beta}$) and the parameter, divided by the standard error of the estimator: $T = (\hat{\beta} - \beta)/SE(\hat{\beta})$. This is of the pattern "(observed - expected) / SE."

7.3.2 The sample variance

For an iid random sample from a normal distribution the distribution of the sample variance S^2 can be identified. Consider this algebraic treatment,

$$\begin{aligned} S^2 &= \frac{1}{n-1}\sum_i^n (X_i - E(X_i))^2 \\ &= \frac{\sigma^2}{n-1}\sum_i^n \left(\frac{X_i - \bar{X}_i}{\sigma}\right)^2 \end{aligned}$$

If the sample is an iid random sample of normals and *were* $\bar{X}_i$ replaced by $\mu = E(X_i)$, this would be $\sigma^2/n - 1$ times a sum of n standard normal random variables, hence described by $Chisq(n)$. However, that isn't quite the case. Cochran's theorem shows the distribution of $\sum(Z_i - \bar{Z})^2$ is $Chisq(n-1)$.

> For a normal population and an iid random sample $(n-1)S^2/\sigma^2$ has a $Chisq(n-1)$ distribution.

For the sample mean and sample variance, we have the distributions for iid random samples from normal distributions. In addition, with these assumptions the two statistics are independent.

7.3.2.1 The F statistic

When there are two independent samples, there are two sample means and standard deviations. The ratio of the sample standard deviations gives insight into the question of whether the two populations have the same spread. The F-statistic is given by:

$$F = \frac{(S_1^2/\sigma_1^2)}{(S_2^2/\sigma_2^2)} = \frac{(\chi^2_{1,n_1-1}/(n_1-1))}{(\chi^2_{2,n_2-1}/(n_2-1))}.$$

If the two populations are normal with means μ_i and standard deviations σ_i, then the distribution of the F statistic is the F distribution with $\nu_1 = n_1 - 1$ and $\nu_2 = n_2 - 1$ degrees of freedom.

A special case is the T statistic for a normal population which has a $F(1, n-1)$ distribution owing to:

$$\begin{aligned} T^2 &= (\frac{\bar{X} - \mu}{S/\sqrt{n}})^2 \\ &= \frac{(\bar{X} - \mu)^2/\sigma^2}{S^2/(n\sigma^2)} \qquad = \frac{(\bar{X} - \mu)^2/(\sigma^2/n)}{S^2/\sigma^2} = \frac{Z^2/1}{\chi^2_{n-1}/(n-1)} \qquad \sim F(1, n-1). \end{aligned}$$

That $\bar{X}$ and S^2 are independent under these assumptions is necessary to identify the distribution.

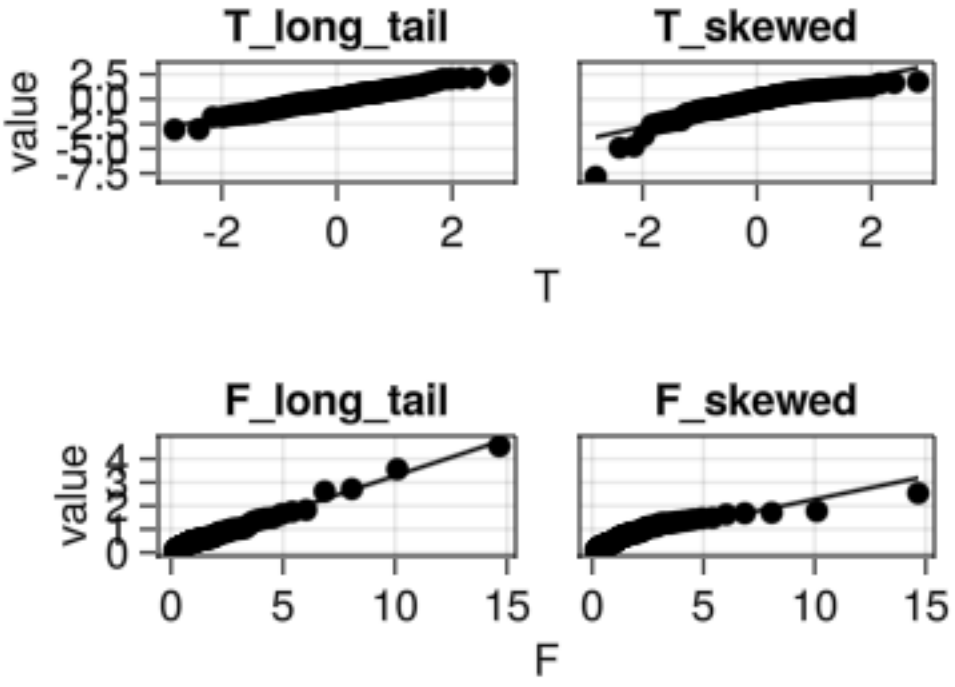
The distribution of the T and F statistics for an iid random sample is known under the assumption of a *normal* population. What if the population is not normal, how robust is the distribution of these statistics to differences in the population? This was studied in [@Box_non_normality_10.1093/biomet/40.3-4.318]. Here we show some examples through simulation.

In the following we use a long-tailed population (T_3) and a skewed exponential population to look:

```
T_n(D, n) = (xs = rand(D,n); (mean(xs) - mean(D)) / (std(xs) / sqrt(n)))
F_n(D, n1, n2) = (xs = rand(D,n1); ys = rand(D, n2); (std(xs)/(n1-1)) / (std(ys)/(n2-1)))

n = n1 = 10
n2 = 5
N = 100

dfT = DataFrame((
    T = quantile.(TDist(n-1), range(0.01, 0.99, N)),
    T_long_tail = [T_n(TDist(3),n) for _ in 1:N],
    T_skewed =    [T_n(Exponential(), n) for _ in 1:N]
    ))
dfF = DataFrame((
    F  = quantile.(FDist(n1-1, n2-1), range(0.01, 0.99, N)),
    F_long_tail = [F_n(TDist(3), n1, n2) for _ in 1:N],
    F_skewed =    [F_n(Exponential(), n1, n2) for _ in 1:N]
    ))
d1, d2 = stack(dfT, 2:3), stack(dfF, 2:3);
```

We visualize these simulations using quantile-quantile plots in Figure 7.4.

Figure 7.4: Smallish degrees of freedom simulations of the T and F distributions for long-tailed and skewed populations. The long-tailed T is closest to distribution when the population is normal.



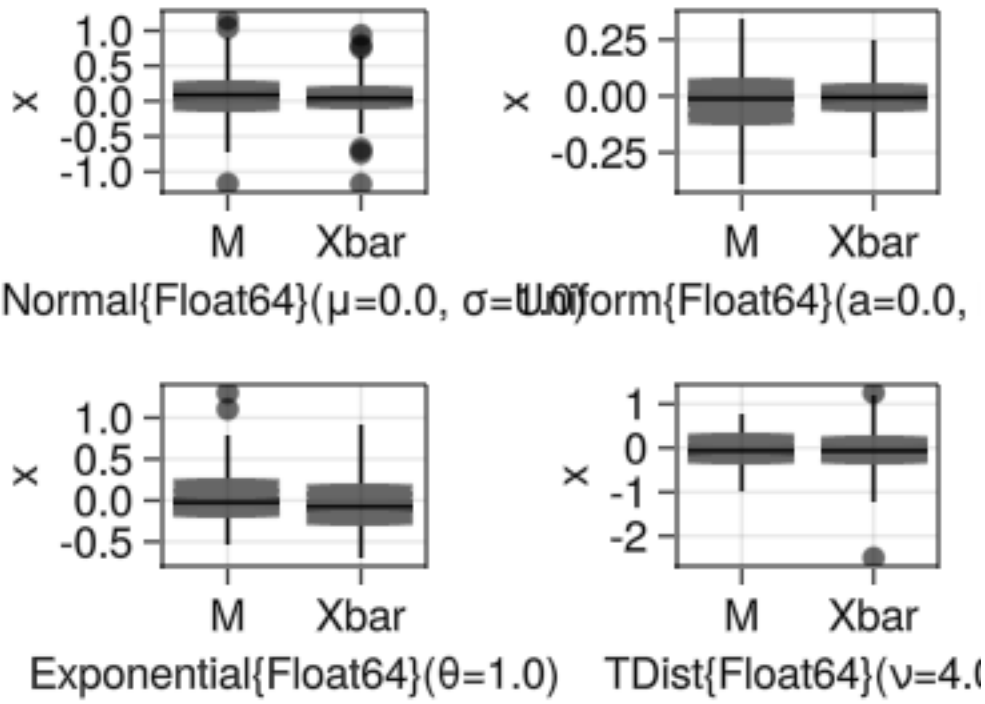
7.3.3 The sample median

The sample median is the "middle" of an iid random sample. Like the sample mean, it is a measure of center, that unlike the mean is resistant to outliers. The central limit theorem instructs us that for most populations the distribution of the sample mean is normally distributed. The distribution of the sample median can be computed. It is asymptotically normal with mean μ and variance given by $(4nf(m)^2)^{-1}$, where n is the sample size, m the median of the *population*, and f the pdf of the population.

Figure 7.5 shows through boxplots the variability of the sample median and sample mean depends on the population. For the standard normal population the ratio of the variances of $\bar{X}$ and M tends to $2/\pi$. This is seen in the smaller IQR for $\bar{X}$ compared to M in the figure for the normal population. This ratio isn't always smaller than 1 for other populations.

```
n = 10
N = 100

MXbar(D,n) = (xs = rand(D,n); (M=median(xs)-median(D), Xbar=mean(xs) - mean(D)))


f = Figure()
Ds = (Normal(0,1), Uniform(0,1), Exponential(1), TDist(4))
for (i,D) in enumerate(Ds)
    z = [MXbar(D,10) for _ in 1:100];
    p = data(stack(combine(DataFrame(;z), :z => AsTable),
        value_name = "x", variable_name=string(D)
    )) *
    visual(BoxPlot) *
    mapping(Symbol(string(D)), :x)

    ax = f[(i-1) ÷ 2 + 1, (i-1) % 2 + 1]
    draw!(ax, p)
end

f
```

Figure 7.5: Comparison of spread of the sample median versus the sample mean for various populations. For skewed and long-tailed distributions, the sample median has smaller variability.



7.3.4 The Chi-squared statistic

For the multinomial distribution for a fixed n with probabilities $p_1, p_2, \ldots, p_k$ and counts $X_1, X_2, \ldots, X_k$, the expected values are $E_i = E(X_i) = np_i$. The difference between the observed, X_i and the expected E_i are often summarized in this statistic:

$$\chi^2 = \sum_k \frac{(X_i - E_i)^2}{E_i}.$$

For this scenario, the *asymptotic* distribution of χ^2 is the Chi-squared distribution with $k - 1$ degrees of freedom. This number comes from the one relationship amongst the k variables (that they all add up to 1.) That is, $k - 1$ p_is are freely chosen, the other a dependency.

Suppose there are s values, $\theta = (\theta_1, \theta_2, \cdots, \theta_s)$ with some identification $f(\theta) = (p_1, p_2, \ldots, p_k)$. If these s values are estimated from the data (using a best asymptotically normal estimator), then there are $k - s - 1$ degrees of freedom in the asymptotic Chi-squared distribution.

The Chis-squared sampling distribution is *asymptotic* meaning for n large enough. A rule of thumb is each count should be expected to be 5 or more.

A related statistic is the Cressie-Read power-divergence statistic, [@CressieRead_10.2307_2345686], parameterized by real λ and given by:

$$2nI^{\lambda} = \frac{2}{\lambda(\lambda+1)} \sum_{i=1}^{k} X_i \left(\left(\frac{X_i}{E_i} \right)^{\lambda} - 1) \right).$$

When $\lambda = 1$, after some algebraic manipulations, this can be identified with the χ^2 statistic above. Other values for λ recover other named statistics, for example $\lambda = 0$ is the maximum-likelihood estimate for this problem.. The Cressie-Read statistic also has an asymptotic $Chisq(k - s - 1)$ distribution for a fixed λ.

Chapter 8

Inference

A goal of statistics is to use data to characterize the process that generated the data.

Suppose $x_1, x_2, \dots, x_n$ is sample. If we assume these are realizations from some iid random sample $X_1, X_2, \dots, X_n$ from some population then when n is *large* we expect the shape of the population to be well described.

We load some useful packages for this chapter:

```
using StatsBase, Distributions
using DataFrames
using AlgebraOfGraphics, GLMakie
```

8.1 Larger data sets

The density plots in Figure 8.1 show a population, a random sample of a certain size from that population, and an density plot found from the random sample. As the sample size gets larger, the density plot resembles more the underlying population.

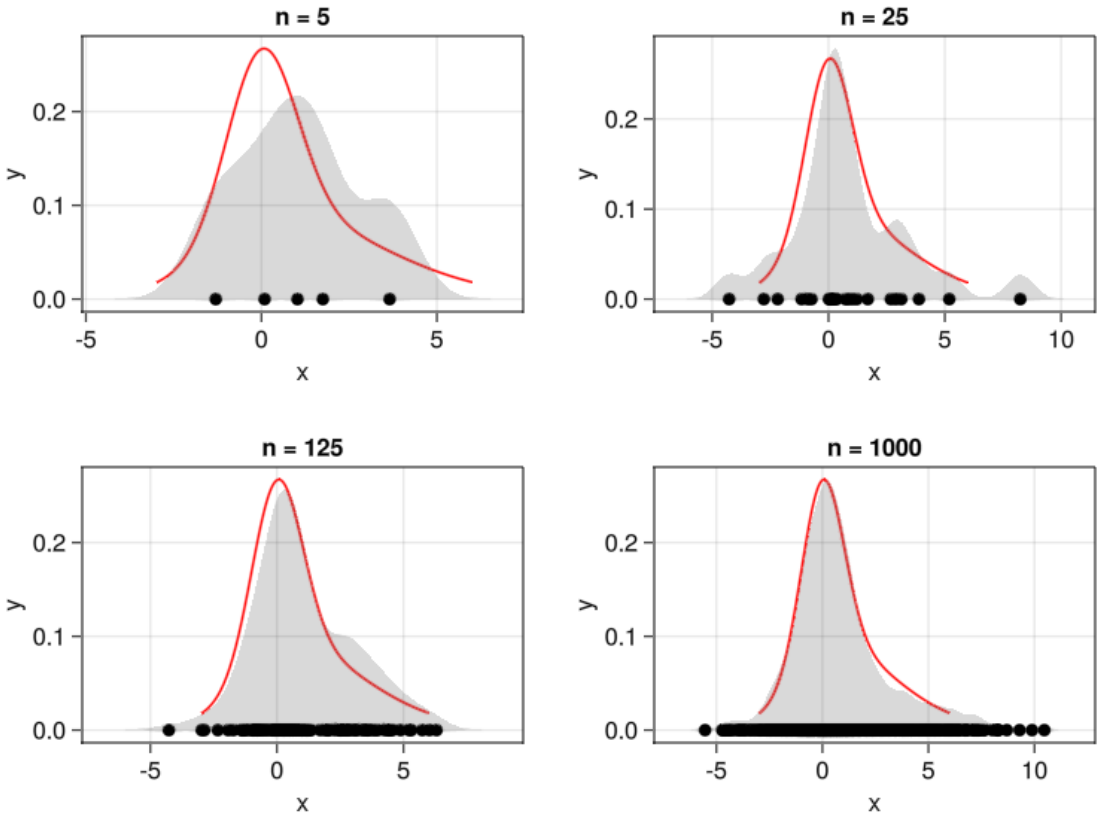
To quantify this, we use the *empirical cdf* defined as

$$F_n(a) = \frac{\#\{i : x_i \le a\}}{n}.$$

That is, the proportion of the sample data less than or equal to a. This estimates the cdf of the population. The `ecdf` function from `StatsBase` returns a function. For example, we have for population given as a mixture of normal distributions:

```
Y = MixtureModel(Normal[
    Normal(0, 1),
    Normal(1, 2),
```

Figure 8.1: A population pdf drawn in red, a sample of size n marked with dots, and a sample density. As n increases, the sample density is an improved estimate for the population pdf.



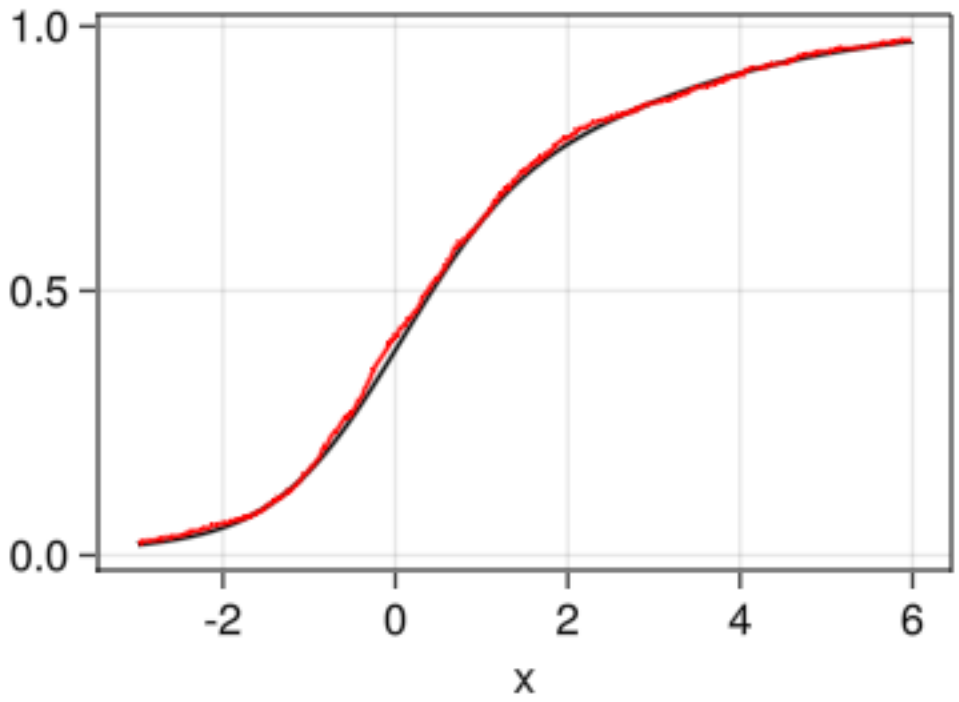
```
    Normal(2.0, 3)], [0.5, 0.2, 0.3])

xs = rand(Y, 1000)
Fₙ = ecdf(xs)

findmax(abs(cdf(Y,x) - Fₙ(x)) for x in range(-5, 5, 1000))
```

```
(0.04339802464070752, 639)
```

The maximum distance between cumulative distributions functions is a useful statistic in itself, which we don't pursue. Rather, we show in Figure 8.2 the empirical cdf and the theoretical cdf for this simulation.

```
Fₙ = ecdf(rand(Y, 1000))
xs = range(-3, 6, 1000)
d = data((x=xs, y=cdf.(Y,xs), y'=Fₙ.(xs)))
p = d * visual(Lines; color=:black) * mapping(:x, :y)
p = p + d * visual(Lines; color=:red) * mapping(:x, :y')
draw(p)
```

Figure 8.2: A cumulative distribution of a mixture model with an empirical cdf generated by a random sample of size 1000.



8.2 Confidence intervals

When there is a much smaller sample size, one can still *infer* things about the underlying population **if** additional assumptions on the population are made. In general, the stronger the assumptions, the more can be said.

8.2.1 Confidence interval for a population mean

For example, suppose you have a data set with n elements, $x_1, x_2, \ldots, x_n$, *and* you assume that:

- the data can be modeled as realizations of some iid collection of random variables, $X_1, X_2, \ldots, X_n$ and
- the population of the random variables is $Normal(\mu, \sigma)$.

With these assumptions the basic facts of probability allow statements about μ based on the data set.

We have that the statistic:

$$Z = \frac{\bar{X} - \mu}{\sigma/\sqrt{n}} = \frac{\bar{X} - \mu}{SD(\bar{X})}$$

has a $Normal(0,1)$ distribution. For any α with $0 < \alpha < 1/2$, we can solve for values $z_{\alpha/2}$ and $z_{1-\alpha/2}$ satisfying:

$$P(z_{\alpha/2} < Z < z_{1-\alpha/2}) = 1 - \alpha.$$

(That is between the z values lies $(1-\alpha)\cdot 100\%$ of the area under the pdf for $Normal(0,1)$.)

By the symmetry of the normal distribution, we have $z_{\alpha/2} = -z_{1-\alpha/2}$.

Rearranging this, we have with probability $1-\alpha$ the following inequality occurs:

$$\bar{X} + z_{1-\alpha/2} \cdot SD(\bar{X}) < \mu < \bar{X} + z_{1-\alpha/2} \cdot SD(\bar{X})$$

Now, the data set is just one possible realization of these random variables, which may or may not be unusual, we can't say. However, we can say the process that produced this data set will produce values where $(1-\alpha)\cdot 100\%$ of the time

$$\bar{x} - z_{1-\alpha/2} \cdot SD(\bar{x}) < \mu < \bar{x} + z_{1-\alpha/2} \cdot SD(\bar{x}).$$

Since a data set is a single realization, and probability speaks to the frequency of many realizations, we can't say for our lone data set that there is a $(1-\alpha)\cdot 100\%$ chance this occurs, rather the language adopted is to say the interval $(\bar{x} - z_{\alpha/2}\cdot\sigma/\sqrt{n}, \bar{x} + z_{1-\alpha/2}\cdot\sigma/\sqrt{n})$ is a $(1-\alpha)\cdot 100\%$ *confidence interval* for an unknown parameter μ.

> For a data set drawn from iid random sample with a $Normal(\mu,\sigma)$ population a $(1-\alpha)\cdot 100\%$ confidence interval is given by $\bar{x} \pm z_{\alpha/2}\cdot\sigma/\sqrt{n}$, where $z_{\alpha/2}$ satisfies $P(-z_{\alpha/2} < Z < z_{\alpha/2})$, Z being a standard normal random variable.

Figure 8.3 illustrates confidence intervals based on several independent random samples. Occasionally – controlled by α – the intervals do not cover the true population mean.

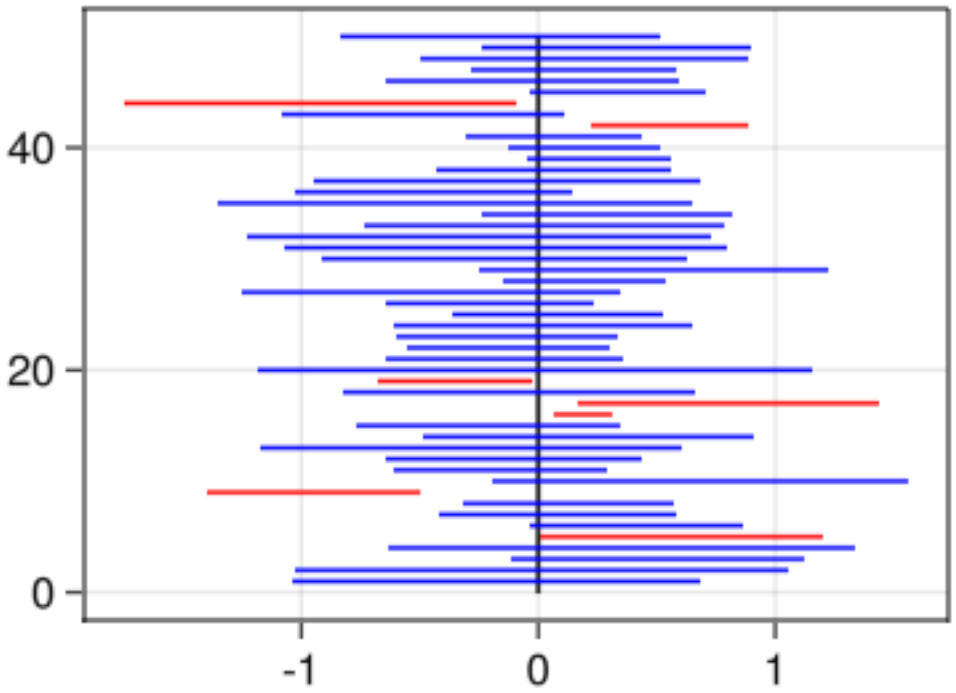
Example 8.1 (Confidence interval for the mean)**.** Consider the task of the beverage dispenser service person. While they may enjoy great benefits and the opportunity to work out of an office, do they get a chance to practice statistics? Well, possibly. Consider the task of calibrating an automatic coffee dispenser. Suppose the engineers have managed to control the variance of the pour so that $\sigma = 0.5$ oz. The 8 oz. cups should not overflow, but should look full when done. As such, the technician aims for a mean dispense of around 7 oz, but not including 7.5. To gauge this, they run the machine 6 times and collect data using a calibrated glass. Assume this data set is from a $N(\mu, \sigma = 1/2)$ population:

```
7.9  7.2  7.1  7.0  7.0  7.1
```

A 95% confidence interval for μ is given by:

```
xs = [7.9,  7.2,  7.1,  7.0,  7.0,  7.1]
n = length(xs)
σ = 1/2
α = 0.05
```

Figure 8.3: 50 simulated confidence intervals. The true mean is $\mu = 0$, the confidence level is 0.90. On average, 10 out of 100 of these confidence intervals will not cover the mean. For this simulation, we expect 5 to be colored red, but this may vary, as each is random.



```
za = quantile(Normal(0, 1), 1 - α + (α/2))
SE = σ / sqrt(n)
(mean(xs) - za*SE, mean(xs) + za*SE)
```

```
(6.816590693637058, 7.616742639696278)
```

That 7.5 is included may be problematic, as larger pours than 8 oz are possible with these assumptions, so the technician calibrates the machine a bit less aggressively.

The assumption of a normal population is used to say the *distribution* of $\bar{X}$ is normal. This would be true if the population weren't normal *but* the sample size, n, were sufficiently large. The important part is having assumptions that allows the sampling distribution of a useful statistic to be known.

The above assumes an unknown mean (μ) but a *known* standard deviation. If that assumption isn't realistic, something similar can be said. Consider the T-statistic:

$$T = \frac{\bar{X} - \mu}{SE(\bar{X})},$$

Under the assumptions above (iid sample, normal population), the standard error is $S/\sqrt{n}$ and the distribution of T is the T-distribution with $n - 1$ degrees of freedom.

> For a data set of size n drawn from an iid random sample with a $Normal(\mu, \sigma)$ population a $(1 - \alpha) \cdot 100\%$ confidence interval is given by $\bar{x} \pm t_{1-\alpha/2} \cdot s$, where $t_{\alpha/2} = -t_{1-\alpha/2}$ satisfies $P(t_{\alpha/2} < T_{n-1} < t_{1-\alpha/2})$, T_{n-1} being T distributed with $n - 1$ degrees of freedom.

Example 8.2 (Confidence interval for the mean, no assumption on σ.)**.** Returning to the coffee-dispenser technician, a cappuccino dispenser has two sources of variance for the amount poured – the coffee and the foam. This is harder to engineer precisely, so is assumed unknown in the calibration process. Suppose the technician again took 6 samples to gauge the value of μ.

With no assumptions on the value of μ *or* σ. A 95% confidence interval would be computed by:

```
xs = [7.9,  7.2,  7.1,  7.0,  7.0,  7.1]
n = length(xs)
s = std(xs)
α = 0.05
za = quantile(TDist(n-1), 1 - α + (α/2))
SE = s / sqrt(n)
(mean(xs) - za*SE, mean(xs) + za*SE)
```

```
(6.856683216914592, 7.576650116418743)
```

These computations – and many others – are carried out by functions in the `HypothesisTests` package. Here we see this problem:

```
using HypothesisTests
confint(OneSampleTTest(xs))
```

```
(6.856683216914592, 7.576650116418743)
```

Of some note, while in general the extra assumption of a known standard deviation will lead to smaller confidence intervals, that is not guaranteed, as seen in this data set.

Example 8.3 (Dependent samples)**.** A method to reduce variability between samples is to match treatments. A classic data set is `shoes`, which collected shoe wear data from 10 boys, each given two different shoes to wear.

```
using RDatasets
shoes = dataset("MASS", "shoes")
first(shoes, 2)
```

	A	B
	Float64	Float64
1	13.2	14.0
2	8.2	8.8

Some boys are assumed to be harder on shoes than others, so by matching the two types, it is expected that the *difference* in the shoe wear per boy could be attributed to the material. That is, if $X_1, \ldots, X_n$ models the one material and $Y_1, \ldots, Y_n$ the other, the difference $Z_i = X_i - Y_i$ should model the difference between the materials. Assuming this data is an iid random sample from a $Normal(\mu, \sigma)$ population, we can find a 90% confidence interval for the mean difference:

```
ds = combine(shoes, [:A,:B] => ByRow(-) => :Z)
confint(OneSampleTTest(ds.Z); level=0.90)
```

```
(-0.634426399199845, -0.18557360080015525)
```

That this does not contain 0, suggests a difference in the mean wear between shoes.

The above illustrates a pattern: a sampling statistic (a *pivotal quantity*) which includes an unknown population parameter with a known sampling distribution independent of the parameters allows the formulation of confidence interval.

8.2.2 Confidence interval for a difference of means

For two iid random samples, $X_1, X_2, \ldots, X_{n_1}$ and $Y_1, Y_2, \ldots, Y_{n_2}$ from two normal populations $Normal(\mu_1, \sigma_1)$ and $Normal(\mu_2, \sigma_2)$ we may have sample data. From that data, the question of the difference between μ_1 and μ_2 can be considered.

With the assumption that the two samples are themselves independent, the standard deviation for $\bar{X} - \bar{Y}$ can be computed as:

$$SD(\bar{X} - \bar{Y}) = \sigma_{\bar{X}-\bar{Y}} = \sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}.$$

Let the T statistic be

$$T = \frac{(\bar{X} - \bar{Y}) - (\mu_1 - \mu_2)}{SE(\bar{X} - \bar{Y})} = \frac{\text{observed} - \text{expected}}{SE}.$$

The distribution of T and the formula for SE depends on assumptions made.

- If both σ_1 and σ_2 are assumed known, $SE(\bar{X} - \bar{Y}) = SD(\bar{X} - \bar{Y})$, and T has a normal distribution.
- If it is *assumed* $\sigma = \sigma_1 = \sigma_2$, but if no value is assumed, then the data can be pooled to estimate σ, the common standard deviation, to get $SE(\bar{X} - \bar{Y}) = s_p\sqrt{1/n_1 + 1/n_2}$ and T has a T distribution with $n_1 + n_2 - 2$ degrees of freedom. The pooled standard deviation is the square root of $(s_1^2(n_1 - 1) + s_2^2(n_2 - 1))/(n_1 + n_2 - 2)$.
- If it is **not** *assumed* $\sigma = \sigma_1 = \sigma_2$ (though it secretly could be), then the standard error is $\sqrt{s_1^2/n_1 + s_2^2/n_2}$. The distribution of T is *approximately* T-distributed with an effective degrees of freedom given by the Welch-Satterhwaite formula, which is always between than the smaller of $n_1 - 1$ and $n_2 - 1$ and $n_1 + n_2 - 2$.

Example 8.4 (Confidence interval for the difference of means)**.** Suppose the coffee-dispenser technician is tasked with calibrating two machines. Again they take samples from the two machines, in this case:

```
Machine 1: 7.0, 7.8, 7.7, 7.6, 8.3
Machine 2: 6.2, 8.0, 6.8, 7.0, 7.3, 7.9, 7.1
```

Do the machines output the same amount *on average*? To answer this, we consider a confidence interval for $\mu_1 - \mu_2$.

If the two machines are assumed to have the same variance, we can compute a 90% confidence interval with:

```
xs = [7.0, 7.8, 7.7, 7.6, 8.3]
ys = [6.2, 8.0, 6.8, 7.0, 7.3, 7.9, 7.1]
confint(EqualVarianceTTest(xs, ys); level=0.90)
```

```
(-0.10761090026945175, 1.0961823288408845)
```

That 0 is in this interval gives evidence that the two means are equal. The sample means do differ:

```
mean(xs), mean(ys)
```

```
(7.6800000000000015, 7.185714285714285)
```

Perhaps were more data available, a difference would be seen, as the variablity is generally smaller for larger sample sizes.

If the machines are from different vendors, or dispense different beverages, perhaps the assumption of equal variances is not appropriate. The `UnequalVarianceTTest` method is available for this comparison. The calling pattern is identical:

```
confint(UnequalVarianceTTest(xs, ys); level=0.90)
```

```
(-0.07723849812971084, 1.0658099267011436)
```

Table 8.1: Different constructors for T-statistic based confidence intervals

Method	Description
OneSampleTTest	Inference on population mean for one sample, unknown population standard deviation
EqualVarianceTTest	Inference on difference of population means for independent samples assuming equal population standard deviations
UnequalVarianceTTest	Inference on difference of population means for independent samples *not* assuming equal population standard deviations

8.2.3 Confidence interval for a proportion

A very common use of confidence intervals is found in the reporting of polls, particularly political polls, where it may be typical to see a statement such as "Candidate A is ahead of her opponent in the polls by 5 percentatge points with a *margin of error* of 5 percent" or "The favorability rating of President B is 54% with a 5 percent margin of error." These are descriptions of confidence intervals, though both leave the confidence level unspecified. When unspecified, it can usually be inferred from the margin of error, as will be seen.

A model for a poll where a person chose one of two options is to use a Bernoulli random variable, X_i, to describe the response. The number of 1s in a fixed number of n respondents can give a proportion. **If** one can assume the X_i are iid random samples from a $Bernoulli(p)$ population, then the number of 1s can be viewed as a realization of a $Binomial(n, p)$ distribution. That is, this statistic will have a known distribution.

Before pursuing, let's note that the assumption implies a few things about the sampling process carried out by the pollster:

- The population p is the proportion of the entire voting population (supposedly of size N) that would respond with a code of 1. A *census* could find p, but random sampling is used as censuses are typically *much more* work to carry out.
- The *independent* assumption technically requires sampling *with replacement*, where a person could be asked 0, 1, *or more* times the question. But *practically* if N is much greater than n this isn't necessary and *sampling without replacement* can be used.
- The *identically distributed* requires the sampling to be representative. For example, if it is a state wide population, a sample concentrated on one possibly partisan district would not be expected to

be identically distributed from the population.

Assuming, a survey is *reasonably* described by a $Binomial(n, p)$ distribution, the `BinomialTest` can be used to identify a confidence interval for a given confidence level.

The `BinomialTest` function can take either two numbers `x` and `n` representing the number of 1s in n sample, or a vector of `true`s and `false`s with `true` being a a "1".

Example 8.5 (A political poll). A poll was taken of 1429 likely voters, with 707 indicating support for candidate A and 722 for candidate B. Let p be the population proportion of voters for candidate A. Find a 95% confidence interval for p.

Assuming the poll was collected in such a way that a binomial model would apply to the data, we have:

```
A = 707
B = 722
n = A + B
ci = confint(BinomialTest(A, n); level = 0.95)
```

```
(0.4685125690214495, 0.521012196604416)
```

As 0.50 is in this interval, there is no suggestion candidate A can't win. (And indeed, they did in this case). The report on this poll would be the sample proportion and a margin of error. The margin of error is the width of the interval divided by two:

```
(last(ci) - first(ci)) * 1/2
```

A bit under 3 percentage points. A rough guide is 3 percentage points is around 1000 people polled, a larger margin of error (MOE) is fewer people polled, a smaller one is more than 1000 polled.

The `BinomialTest` has several different ways to compute the margin of error, or the confidence interval. These are passed to the `method` argument of `confint` as symbols. The default is `:clopper_pearson` which is based on the binomial distribution. The value `:wald` uses a normal approximation to the binomial, which may be very inaccurate when p is close to 0 or 1. There are others.

The default one can be understood through the `quantile` function, and is *essentially* given through:

```
alpha = 0.05
quantile.(Binomial(n, A/n), [alpha/2, 1 - alpha/2]),  ci .*n
```

```
([670, 744], (669.5044611316513, 744.5264289477104))
```

The normal approximation has an explicit formula for the margin of error, based on the standard deviation of the binomial distribution:

$$MOE = z_{1-\alpha/2} \cdot \sqrt{p(1-p)/n} = z_{1-\alpha/2} SE(\hat{p}).$$

As p is unknown, the standard error is used with $\hat{p}$ approximating p.

```
ci = confint(BinomialTest(A, n); level = 0.95, method=:wald)
z_a = quantile(Normal(0,1), 1-alpha/2)
phat = A/n
SE = sqrt(phat * (1-phat)/n)
z_a * SE, (last(ci) - first(ci))/2
```

```
(0.025922569857879177, 0.025922569857879163)
```

The latter allows one to estimate the sample size, n, needed to achieve a certain margin of error, though an estimate for p is useful, taking $p = 1/2$ gives a conservative number as $p(1-p)$ is largest for that value. For example, to see what a 3% margin of error with a 95% confidence level would need in terms of a sample, we have, solving for n:

$$n = p(1-p) \cdot \left(\frac{z_{\alpha/2}}{MOE}\right)^2 \le \frac{1}{4}\left(\frac{z_{\alpha/2}}{MOE}\right)^2.$$

```
MOE = 0.03
alpha = 0.05
z_a = quantile(Normal(0,1), 1-alpha/2)
1/2 * (1- 1/2) * (z_a / MOE)^2
```

```
1067.071894637261
```

A bit more than 1000 in this case ensures that the MOE will be no more than 0.03; it could be less if p is far from 1/2.

8.2.4 Confidence interval for a difference of proportions

For two sample proportions, $\hat{p}_1 = x_1/n_1$ and $\hat{p}_2 = x_2/n_2$, the T statistic:

$$T = \frac{(\hat{p}_1 - \hat{p}_2) - (p_1 - p_2)}{SE(\hat{p}_1 - \hat{p}_2)}$$

has a standard normal distribution if the n values are both large enough. This allows a confidence interval for $p_1 - p_2$ to be given by:

$$z_{\alpha/2} \cdot SE(\hat{p}_1 - \hat{p}_2) < p_1 - p_2 < z_{1-\alpha/2} \cdot SE(\hat{p}_1 - \hat{p}_2),$$

where $SE(\hat{p}_1 - \hat{p}_2) = \sqrt{\hat{p}_1(1-\hat{p}_1)/n_1 + \hat{p}_2(1-\hat{p}_2)/n_2}$.

Example 8.6 (Difference of population proportions)**.** Did some external event cause people to reexamine their choice for a political race? Suppose polls taken several weeks apart yielded:

	Candidate A	Candidate B
Oct	289	259
Nov	513	493

Compute a 90% confidence interval.

```
x1, x2 = 289, 513
n1, n2 = x1 + 259, x2 + 493
phat1, phat2 = x1/n1, x2/n2
SE = sqrt(phat1*(1-phat1)/n1 + phat2*(1-phat2)/n2)
alpha = 0.10
za = quantile(Normal(0,1), 1 - alpha/2)
(phat1 - phat2) .+ [-1,1] * za * SE
```

```
2-element Vector{Float64}:
 -0.026187674280803125
  0.06105148412248293
```

That 0 is in this, suggests the possibility that there was no change in the polls.

8.2.5 Confidence interval for a population standard deviation

Under a normal population assumption for an iid random sample, $(n-1)S^2/\sigma^2$ has a $Chisq(n-1)$ distribution. Solving $\chi^2_{\alpha/2} < (n-1)S^2/\sigma^2 < \chi^2_{1-\alpha/2}$ for σ gives a formula for a $(1-\alpha)\cdot 100\%$ confidence interval for σ^2:

$$\frac{(n-1)S^2}{\chi^2_{1-\alpha/2}} < \sigma^2 < \frac{(n-1)S^2}{\chi^2_{\alpha/2}}.$$

To use this, suppose our data is

```
1.2, -5.2, -8.4, 3.1, -2.1, 3.8
```

We can give a 90% CI by:

```
xs = [1.2, -5.2, -8.4, 3.1, -2.1, 3.8]
n, s2 = length(xs), var(xs)
alpha = 0.10
cl, cr = quantile.(Chisq(n-1), [alpha/2, 1 - alpha/2])
(sqrt((n-1)*s2/cr), sqrt((n-1)*s2/cl))
```

```
(3.2630536146068865, 10.144128513617867)
```

8.2.6 Confidence interval for comparing population standard deviations

The comparison of two sample standard deviations is of interest, as seen in the two-sample T confidence interval, where different formulas are available should there be an assumption of equality. For two *independent* normally distributed iid random samples, the F-statistic can be used to construct a confidence interval, as a scaled ratio of the sample standard deviations will have a certain F distribution. The `VarianceFTest` constructor does not have a `confint` method, though it does compute needed pieces, so we define our own method using:

$$F_{\alpha/2;n_1-1,n_2-1} < (S_1^2/\sigma_1^2)/(S_2^2/\sigma_2^2) < F_{1-\alpha/2;n_1-1,n_2-1}$$

or, after rearrangement:

$$\frac{S_1^2/S_2^2}{F_{1-\alpha/2;n_1-1,n_2-1}} < \frac{\sigma_1^2}{\sigma_2^2} < \frac{S_1^2/S_2^2}{F_{\alpha/2;n_1-1,n_2-1}}.$$

We can create a `confint` method as follows:

```
function HypothesisTests.confint(F::VarianceFTest; level=0.95)
    alpha = 1 - level
    dof1,dof2 = F.df_x, F.df_y
    fl, fr = quantile.(FDist(dof1, dof2), (alpha/2, 1-alpha/2))
    F.F ./ (fr, fl)
end
```

To use this, for example, suppose we have two samples:

```
xs: 1.2, -5.2, -8.4, 3.1, -2.1, 3.8
ys: -1.7, -1.8, -3.3, -6.3, 8.4, -0.1, 2.5, -3.8
```

Then a 90% confidence interval for σ_1^2/σ_2^2 is given by:

```
xs =  [1.2, -5.2, -8.4, 3.1, -2.1, 3.8]
ys =  [-1.7, -1.8, -3.3, -6.3, 8.4, -0.1, 2.5, -3.8]
confint(VarianceFTest(xs, ys); level=0.90)
```

```
(0.28992354789264896, 5.614264355299279)
```

8.2.7 Likelihood ratio confidence intervals

The confidence intervals so far are based on pivotal quantities with known sampling distributions. An alternative is to use the likelihood function, $L(\theta|x) = f(x|\theta)$, where the latter is the joint pdf of the data for a given parameter or parameters. The idea of maximizing the likelihood is to choose the values for θ which maximize the probability of seeing the observed data.

The *maximum likelihood estimate* is an alternative manner to estimate a parameter. Here we apply the method to data that would otherwise be amenable to a a T-test. First we generate some random data:

```
μ, σ = 10, 3
n = 8
D = Normal(μ, σ)
ys = rand(D, n)
permutedims(ys) # save space
```

```
1×8 Matrix{Float64}:
 9.21103  10.4403  8.62943  8.54699  10.899  7.85558  8.67761  8.48412
```

The likelihood function below uses an assumption on on the data. Below it is normally distributed with unknown mean and standard deviation ($y_i \sim Normal(\mu, \sigma)$), which just happens to match how the random data was generated, but typically is an assumption about the data. We write the log-likelihood function so it takes two values the unknown parameters and the data:

```
function loglik(θ, data)
    # yᵢ ~ N(μ, σ) is model
    μ, σ = θ
    D = Normal(μ, σ)
    y = first(data)
    ll = 0.0
    for yᵢ ∈ y
        ll += logpdf(D, yᵢ)
    end
    ll
end
```

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

We use `logpdf` to compute the logarithm of the probability density function.

To to maximize the data. We will use the `ProfileLikelihood` package to do this problem, which relies on the `Optim` package to perform the optimization:

```
using ProfileLikelihood, Optim, OptimizationOptimJL
```

The optimization needs an initial guess, θ_0, and we give names to our parameters, as we set up a "problem" below:

```
θ₀ = [5.0, 2.0]
nms = [:μ, :σ]
dat = (ys,) # data is a container
prob = LikelihoodProblem(loglik, θ₀;
                         data = dat,
                         syms = nms)
```

```
LikelihoodProblem. In-place: true
```

```
θ₀: 2-element Vector{Float64}
     μ: 5.0
     σ: 2.0
```

This problem is solved by `mle`, which needs an optimization algorithm. We use the `NelderMead` one, which is easier, as we don't need to set up means to take a gradient:

```
sol = mle(prob, Optim.NelderMead())
```

```
LikelihoodSolution. retcode: Success
Maximum likelihood: -11.187252311197941
Maximum likelihood estimates: 2-element Vector{Float64}
     μ: 9.093015522996863
     σ: 0.9796871055369571
```

As seen, the `sol` object outputs the estimate for μ. We can compare to `mean(ys)` and `std(ys)`; the mean agrees (the standard deviation has a different divisor, so is systematically different):

```
mean(ys), std(ys)
```

```
(9.09300037254253, 1.0473191427156174)
```

The reason to use the `ProfileLikelihood` package to do the optimization, is we can then find 95% confidence levels for the parameters using a method implemented therein. The profile generation requires a specification of upper and lower bounds for the parameters:

```
lb = [-100.0, 0.0]
ub = [ 100.0, 100.0]  # -100 < μ < 100; 0 < σ < 100
resolutions = [200, 200]
param_ranges = construct_profile_ranges(sol, lb, ub, resolutions)
prof = profile(prob, sol; param_ranges=param_ranges)
```

```
ProfileLikelihoodSolution. MLE retcode: Success
Confidence intervals:
     95.0% CI for μ: (8.323801031112364, 9.862135177919024)
     95.0% CI for σ: (0.6427734005174341, 1.758283638480391)
```

We can compare the confidence interval identified for μ to that identified through the T statistic:

```
confint(OneSampleTTest(ys), level = 0.95)
```

```
(8.21741965760898, 9.96858108747608)
```

The two differ – they use different sampling distributions and methods, though simulations will show both manners create CIs capturing the true mean at the rate of the confidence level.

The above example does not showcase the advantage of the maximimum likelihood methods, but hints at a systematic way to find confidence intervals, which for some cases is optimal, and is more generally

applicable then finding some pivotal quantity (e.g. the T-statistic under a normal population assumption).

8.3 Hypothesis tests

A confidence interval is a means to estimate a population parameter. However, sometimes a different language is sought. For example, we might hear a newer product is *better* than an old one, or amongst two different treatments there is a *difference*. The language of hypothesis tests allows this flexibility. The basic setup is similar to a courtroom trial in the United States – as seen on TV:

- a defendant is judged by a jury with an *assumption of innocence*
- presentation of evidence is given
- the jury weighs the evidence
- If it is a civil trial a *prepoderence of evidence* is enough for the jury to say the defendent is guilty (not innocent); if a criminal trial the standard if the evidence is "beyond a reasonable doubt" then the defendent is deemed not innocent. Otherwise the defendant is said to be "not guilty," though really it should be that they weren't "proven" to be guilty.

In a hypothesis or significance test for parameters, the setup is similar:

- a null hypothesis of "no difference" is assumed; an alternative hypothesis is specified.
- data is collected
- a test statistic is used to summarize the data
- a probability is computed that accounts for the assumption of the null hypothesis and the data, computing the probability that the random values would be as or more extreme than seen in the data *assuming* the null hypothesis is true
- if that probability is small, then one might conclude the null hypothesis is incorrect; otherwise there is no evidence to say it isn't correct, small would depend on the context of application.

To illustrate, suppose a company claims their new product is more effective than the current standard. Further suppose this effectiveness can be measured by a single number, though there is randomness to any single measurement. Over time the old product is known to have a mean of 70 in this measurement.

Then the null and *alternative* hypotheses would be:

$$H_0 : \mu = 70 \quad H_A : \mu > 70,$$

where a bigger number is considered better. The null is an assumption of "no change", the alternative points in the direction of the claim (better, in this case).

To test this, data would be collected. Suppose in this case 8 measurements of the new product were taken with these values:

```
73.0, 66.1, 76.7, 68.0, 60.8, 81.8, 73.5, 75.2
```

Assume these are realizations of an iid random sample from a normal population, then the sample mean would be an estimate for the population mean of the new product:

```
xs = [73.0, 66.1, 76.7, 68.0, 60.8, 81.8, 73.5, 75.2]
mean(xs)
```

```
71.8875
```

This is more than 70, but the skeptic says half the time a sample mean would be more than 70 if the null hypothesis were true. Is it really indicative of "better?" For that question, a sense of how extreme this observed values is under an assumption of the null hypotheses is used.

To test this, we characterize the probability a sample mean would be even more than our observed one, were we to repeat this sampling many times:

> The p-value is the probability the test statistic is as or *more extreme* than the observed value under the null hypothesis.

For this problem the p-value is the probability the sample mean is `mean(xs)` or more assuming the *null hypothesis*. Centering by μ and scaling by the standard error, this is:

$$P(\frac{\bar{X} - \mu}{SE(\bar{X})} > \frac{\bar{x} - \mu}{s/\sqrt{n}} \mid H_0)$$

That is, large values of $\bar{X}$ are not enough to be considered statistically significant, rather large values measured in terms of the number of standard errors are.

This scaling and an assumption that the data is from a normal population makes this a statement about a T-distributed random variable, namely:

```
n = length(xs)
SE = std(xs) / sqrt(n)
obs = (mean(xs) - 70) / SE
p_value = 1 - cdf(TDist(n-1), obs)  # 1 - P(T ≤ obs) = P(T > obs)
```

```
0.22361115388017372
```

Is this p-value suggestive that the null hypothesis is not a valid explanation of the variation seen in the data?

(The `ccdf` function, complementary cumulative distribution function, could also be used to more directly find the p-value.)

The jury has guidance, depending on the type of trial, here there is also guidance in terms of a stated *significance level*. This is done in advance, but is typically just $\alpha = 0.05$. So if the p-value is less than α the null hypothesis is suggested to be incorrect; if not there is no evidence to conclude the null hypothesis is incorrect. Since here the p-value is much greater than α, we would say there is no evidence to *reject* the null hypothesis.

The mechanics here are standard and encapsulated in the `OneSampleTTest` constructor. The `pvalue` method allows the specification of the type of "tail". In this case, as the alternative hypothesis is $H_A : \mu > 70$, the tail is `:right`:

```
pvalue(OneSampleTTest(xs, 70); tail=:right)
```

```
0.22361115388017377
```

Had the alternative been $H_A : \mu \neq 70$, the p-value would be computed differently as then very big or very small values of the observed test statistic would be evidence against the null hypothesis. In this case, the tail is `:both`, which is the default:

```
pvalue(OneSampleTTest(xs, 70))
```

```
0.44722230776034755
```

8.3.0.1 Design of structures in HypothesisTests

The `OneSampleTTest` method creates a structure, also called `OneSampleTTest`. It is illustrative of the other tests in the `HypothesisTests` package. This structure does not keep the data, only sufficient statistics of the data to generate the various summaries. In this case, these are `n`, `xbar` ($\bar{x}$), `df` ($n-1$), `stderr` (SE), `t` (`(mean(xs)-mu)/stderr`), and `μ0` (the null hypothesis). The design is the default show method summarizes most of what may be of interest, specialized methods, like `confint` and `pvalue` allow customization and access to the specific values. The default `show` method for this test is:

```
OneSampleTTest(xs)
```

```
One sample t-test
-----------------
Population details:
    parameter of interest:   Mean
    value under h_0:         0
    point estimate:          71.8875
    95% confidence interval: (66.34, 77.43)

Test summary:
    outcome with 95% confidence: reject h_0
    two-sided p-value:           <1e-07

Details:
    number of observations:   8
    t-statistic:              30.6644467681273
    degrees of freedom:       7
    empirical standard error: 2.3443273098512263
```

A default value for the null was chosen (`0`), a default level for the confidence interval (`0.95`), and a default choice of tail was made for the computed p-value.

8.3.0.2 Equivalence between hypothesis tests and confidence intervals

For many tests, such as the one-sample T-test, there is an equivalence between a significance test and a confidence interval, which is no surprise given the same T-statistic is employed by both.

For example, consider a two-tailed significance test of $H_0 : \mu = \mu_0$ with a two-sided alternative. Let α be the level of significance: we reject if the p-value is less than α. An iid random sample is summarized by

the observed value of the T-statistic, $(\bar{x} - \mu_0)/(s/\sqrt{n})$. Let $t^* = t\_1 - \alpha/2$ be the quantile, then if the p-value is less than α, we must have the observed value in absolute value is greater than t^*. That is

$$|\frac{\bar{x} - \mu_0}{s/\sqrt{n}}| > t^*.$$

Algebraically, this implies that either $\mu_0 < \bar{x} - t^* s/\sqrt{n}$ or $\mu_0 > \bar{x} + t^* s/\sqrt{n}$. That is, the value of μ_0 is not in the $(1 - \alpha) \cdot 100\%$ CI based on the sample. That is, rejecting the null hypothesis is equivalent to μ_0 not being in the corresponding confidence interval. The reverse is also true: any μ_0 not in the $(1 - \alpha) \cdot 100\%$ CI would have a two-tailed test of $H_0 : \mu = \mu_)$ rejected at the α significance level.

8.3.0.3 Alternative hypotheses

The null hypothesis is also described as one of "no effect," the alternative hypothesis the "research hypothesis." The p-value is computed under the null hypothesis. The alternative hypothesis may take different forms: it could be a specification of a point, as will be seen in the next section on power; a one-tailed directional hypothesis, useful to capture expressions like "better;" two-tailed directional, useful to capture expressions like "different;" and non-directional, useful for indicating anything save the null hypothesis. The distinction between the last two is more clear when the null hypothesis speaks to more than one variable, such as is the case when describing, say, different probabilities for a multinomial distribution.

In `HypothesisTests`, for the T tests and many others, the alternative is specified to `pvalue` through the `tail` argument with a value of `:both`, `:right`, or `:left`. Some tests have non-drectional alternatives and no argument. In the case of a T-test, with a symmetric sampling distribution, symmetry relates the values, so knowing one can be used to figure out the other two.

8.3.0.4 Type-I error

The computation of a p-value is the result of a significance test. Often this value is compared to a *significance level*, α: if the p-value is less than α the difference of the test statistic from the null hypothesis is *statistically significant*. In other language, we may "reject" the null hypothesis when the p-value is less than α and "accept" the null if not.

The p-value is computed under the null hypothesis being true. The p-value depends on a random sample. How often, on average, would this p-value be less than α? Why, $\alpha \cdot 100$ percent of the time, as $\alpha = P(\text{reject}|H_0)$.

We can see this through a simulation. Let's look at the T statistic for a sample of size 10 from a $Normal(0, 1)$ population:

```
Z = Normal(0,1)
n = 10
N = 5000
ps = [pvalue(OneSampleTTest(rand(Z, n))) for _ in 1:N]
sum(ps .< 0.01)/N, sum(ps .< 0.05)/N, sum(ps .< 0.10)/N
```

```
(0.01, 0.0464, 0.098)
```

The proportions in the 5000 samples roughly match the threshold specified ($\alpha = 0.01, 0.05, 0.10$).

There are some cases where a *conservative* sampling distribution is used. (For example, with a two sample test of means with no assumption of equal variances, a conservative degrees of freedom is sometimes suggested.) Conservative means that "rejecting" the null (or finding p value less than α) will occur on average less than $\alpha \cdot 100$ percent of the time.

8.3.0.5 Power

Consider a significance test for the population mean with this null and alternative hypothesis:

$$H_0 : \mu = 0, \quad H_A : \mu = 1.$$

When a small p value is found, the language of "rejecting" the null hypothesis is used, whereas for large p values, "accept" is used. If we "reject" when the p-value is less than α, then the probability of making a mistake when the *null hypothesis* is true is α. This is called a *Type-I error*. In the above, as the alternative is a fixed value of the parameter, we could also compute the probability of "accepting" when the *alternative* hypothesis is true. When this error occurs, a *Type-II error* happens.

For example, a sample of size $n = 4$ from a $Normal(\mu, 1)$ distribution would have a standard error of $1/\sqrt{4} = 1/2$. Without thinking too much, were the sample mean close to 0 we would accept the null, were it bigger than 1 we would reject the null. Precisely, if we set $\alpha = 0.05$, then we can compute under the null hypothesis a critical value for which if $\bar{x}$ is less we "accept" and if more we "reject":

```
alpha = 0.05
sigma = 1
n = 4
SD = sigma / sqrt(n)
zc = quantile(Normal(0, SD), 1 - alpha)
```

```
0.8224268134757359
```

This is computed under the null hypothesis. In this case, we can compute under the alternative hypothesis, as it is fully specified. The probability of "accepting" when the alternative is true is called β. For this we have:

```
mu_a = 1
cdf(Normal(mu_a,  SD), zc)
```

```
0.36123996868766456
```

Figure 8.4 shows the sampling distribution under the null hypothesis centered at 0, and the sampling distribution under the alternative hypothesis, centered at 1. The shaded area representing β is much bigger than α.

The value for β is pretty large, over a third, so with such a small sample, it is difficult to detect a difference of $1 = \mu_A - \mu_0$. The basic expectation is: the smaller the difference, the larger the sample is needed to have

"β" be small. The *power* is $1 - \beta$. It may be specified ahead of time, as in: what size n is needed to have power 0.80 with a difference of means being 1.

To solve this, we would need to solve

```
1 - beta = 1 - cdf(Normal(mu_a, sigma/sqrt(n)),
    quantile(Normal(0, sigma/sqrt(n)), 1 - alpha))
```

For this problem we could do this with the following approach, where we first define a function to get the β value for a given n and set of assumptions:

```
function get_beta(n)
    H0 = Normal(0, sigma/sqrt(n))
    HA = Normal(mu_a, sigma/sqrt(n))
    critical_point = quantile(H0, 1 - alpha)
    beta = cdf(HA, critical_point)
end
```

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

This is then solved for a value of `n` where the power for a given `n` matches the desired power. To do so, we equate the values and, here, note a sign change between 2 and 100:

```
beta = 0.2
power = 1 - beta
f(n) = (1 - get_beta(n)) - power
f(2), f(100)
```

```
(-0.3912027802061293, 0.19999999999999996)
```

Somewhere between 2 and 100 lies the answer, and using a zero finding algorithm, a value of 7 can be seen to produce a power greater than or equal to 0.80.

```
using Roots
find_zero(f, (2, 100))
```

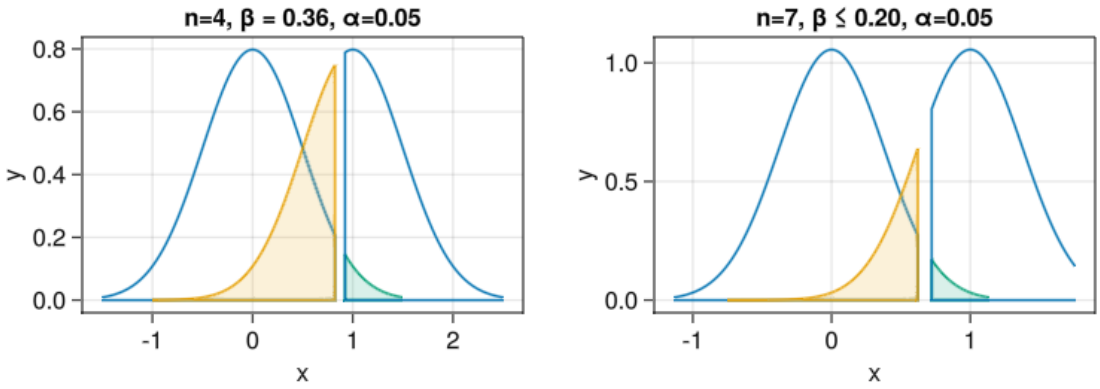
```
6.182557232019764
```

(We round up in calculations involving sample size.)

The computation above was made easier as we used the standard deviation in our test statistic with $\sigma = 1$. To find a sample size for a T-test, we turn to a package, as the underlying sampling distributions are a bit different, especially under H_A.

Let the *effect size*, or standardized mean difference, be the *absolute effect size* (the difference between the mean specified under H_0 and that under H_A) divided by the standard deviation of the population. This value is termed "small" when around 0.20, medium when around 0.50, and large when 0.80 or more. In the above computation, it would be $1 = (1 - 0)/1$.

Figure 8.4: Illustration of power computation. A significance level α is chosen; under H_0 this allows the identification of a critical point. Under H_A, that same critical point is used to find the area in the the "acceptance" region. This probability is β. The power is $1 - \beta$. Adjusting the sample size, n, will narrow the two bell curves allowing β to be set to a specified value. The left-hand figure uses $n = 4$, the right hand one uses $n = 7$.



The power, $1 - \beta$, depends on the value of α, the sample size, and the effect size. Specifying 3 of these 4 values allows the fourth to be numerically identified.

The `PowerAnalyses` package carries out several different power computations. For the one test we are illustrating, the `OneSampleTTest`, the name conflicts with a related, but different method in the `HypothesisTests` package, as `PowerAnalyses` does not extend `HypothesisTests`. We use `import` instead of `using`, and then qualify with the module name each function used.

```
import PowerAnalyses
```

For example, to identify the sample size needed for a left-tailed test with $\alpha = 0.05$, $\beta = 0.20$, and the effect size is small (0.30) we have:

```
effect_size = 0.3
alpha = 0.05
power = 0.80

T = PowerAnalyses.OneSampleTTest(PowerAnalyses.one_tail)
PowerAnalyses.get_n(T; alpha=alpha, es=effect_size, power=power)
```

```
70.06790520005853
```

We see a smaller n is needed to have the same power with a larger effect size.

```
# XXX needs new version of PowerAnalyses
effect_size = 0.8
PowerAnalyses.get_n(T; alpha=alpha, es=effect_size, power=power)
```

```
11.144239681013328
```

There are other power computations provided, but this one illustrates a key point: with large enough sample sizes, any effect size can be discerned. That is the difference is *statistically significant*. However, that does not mean it is *practically significant*. An example from the NIH describes a "Physicians Health Study of aspirin to prevent myocardial infarction. In more than 22,000 subjects over an average of 5 years, aspirin was associated with a reduction in MI (although not in overall cardiovascular mortality) that was highly statistically significant: $p < .00001$. The study was terminated early due to the conclusive evidence, and aspirin was recommended for general prevention. However, the effect size was ... extremely small. As a result of that study, many people were advised to take aspirin who would not experience benefit yet were also at risk for adverse effects. Further studies found even smaller effects, and the recommendation to use aspirin has since been modified.

8.3.1 One sample Z test

The one-sample Z test is similar to the one-sample T test though the *known* population standard deviation is used in the test statistic, so the standard normal distribution is the sample distribution.

For example, consider again the coffee-dispenser technician. They wish to test the hypothesis

$$H_0 : \mu = 7.5, \quad H_A : \mu < 7.5$$

assuming the distribution is $Normal(\mu, 1/2)$. The test statistic used would be $(\bar{x} - \mu)/(\sigma/\sqrt{n})$. The data collected is

```
7.9  7.2  7.1  7.0  7.0  7.1
```

The test can be carried out through the following, which shows evidence to reject the null in favor of the alternative.

```
xs = [7.9,  7.2,  7.1,  7.0,  7.0,  7.1]
Z = OneSampleZTest(xs, 7.5)
pvalue(Z, tail=:left)
```

```
0.02152440439927433
```

(The statistician may say, if the mean is 7.5 and the standard deviation is 0.5, then anything more than 1 standard deviation from the mean will overflow an 8oz cup. But they know that the normal distribution is only a generalization and that the values are always within 0.5 oz of the calibration mean.)

8.3.2 The sign test

The Wilcoxon signed-rank test is an alternative to the Z and T tests. Unlike those, there is no distribution assumption on the population *except* that it be symmetric. For symmetric distributions with reasonable

tails, the median and mean are identical, so this is often termed a test of the median. The test is implemented in the `SignedRankTest` function. Unlike other tests, we use the null $H_0 : M = 0$, so we subtract off the tested median before proceeding.

Returning to the coffee-dispenser technician who took 6 samples and is testing if the center of the data is centered at 7.5 or *something different*, we have:

```
xs = [7.9,  7.2,  7.1,  7.0,  7.0,  7.1]
M = SignedRankTest(xs .- 7.5)
pvalue(M, tail=:both)
```

```
0.15625
```

This p-value is large, there is no suggestion that H_0 does not hold.

8.3.3 One sample test of proportions

A test of survey data where one of two answers are possible can be carried out using the binomial model, assuming the sample data is representative.

For example, suppose it is known that in any given semester, 50% of matriculating students will have a lower semester GPA than their cumulative GPA. A researcher tests if college transfer students are different with p representing the population proportion whose lower semester GPA will be worse than their cumulative one. The test is:

$$H_0 : p = 0.50, \quad , H_A : p > 0.50$$

They collect data and find of 500 transfer students identified, 290 had lower GPAs than their accumulated GPA. Is this data statistically significant?

```
x, n = 290, 500
p = 0.5
B = BinomialTest(x, n, p)
pvalue(B, tail=:right)
```

```
0.00020020776841988704
```

This is an example of "transfer shock," which overtime tends to go away for most transfer students.

8.3.4 Two-sample test of center

A two-sample test of center can test if the population means or medians are equal. The latter could be done with the signed-rank test. Here we illustrate a test of the population means based on the T-statistic:

$$T = \frac{(\bar{X}_1 - \bar{X}_2) - (\mu_1 - \mu_2)}{SE(\bar{X}_1 - \bar{X}_2)}.$$

The null hypothesis will be $H_0 : \mu_1 = \mu_2$, the alternative is either direction or both. The *SE* will depend on an assumption of equal population variances or not (`EqualVarianceTTest` or `UnequalVarianceTTest`). The basic pattern to find the corresponding p-value is identical.

For example, consider the coffee-dispenser technician seeing if two machines have an equal calibration. The hypotheses would be:

$$H_0 : \mu_1 = \mu_2, \quad H_A : \mu_1 \neq \mu_2$$

The collected data is:

```
Machine 1: 7.0, 7.8, 7.7, 7.6, 8.3
Machine 2: 6.2, 8.0, 6.8, 7.0, 7.3, 7.9, 7.1
```

Without assuming equal variances, the test could be carried out with:

```
xs = [7.0, 7.8, 7.7, 7.6, 8.3]
ys = [6.2, 8.0, 6.8, 7.0, 7.3, 7.9, 7.1]
T = UnequalVarianceTTest(xs, ys)
pvalue(T)  # use default tail=:both
```

```
0.14802700278100428
```

The data does not suggest a statistically significant difference between the two machines.

8.3.5 Two-sample test of proportions

For a comparison of two population proportions, we do the math using the test statistic:

$$Z = \frac{(\hat{p}_1 - \hat{p}_2) - (p_1 - p_2)}{SE(\hat{p}_1 - \hat{p}_2)},$$

which for large enough n will have a standard normal distribution, assuming the sampling is well modeled by independent, binomial random variables.

Suppose, we are interested in comparing two randomly chosen groups of transfer students at different institutions. We are testing if the GPA on the first semester after transfer falls off from the accumulated average. The test hypotheses are:

$$H_0 : p_1 = p_2, \quad , H_A : p_1 \neq p_2.$$

The test statistic involves *SE* where the *SD* is $\sqrt{p_1(1-p_1)/n_1 + p_2(1-p_2)/n_2}$. The null hypothesis states $p_1 = p_2$, but not what that value is, so the *SE* is found by estimating $p_1 = p_2$ by *pooling* the data.

The collected data is:

	x	n
School A	120	200
School B	100	190

We proceed to compute the observed value of the test statistic:

```
x1, n1 = 120, 200
x2, n2 = 100, 190
phat1, phat2 = x1/n1, x2/n2
phat = (x1 + x2)/(n1 + n2) # pooled
SE = sqrt(phat*(1-phat) * (1/n1 + 1/n2))
Z_obs = (phat1 - phat2)/SE
```

```
1.4667724206855928
```

This is a two-tailed test, so the p-value incorporates potential test statistics more than `Z_obs` or *less than* `-Z_obs`. (The minus sign comes, as `Z_obs` is positive.) Since the sampling distribution is a symmetric standard normal we have:

```
p_value = 2 * (1 - cdf(Normal(0,1), Z_obs))
```

```
0.14243797452124007
```

This being larger than $\alpha = 0.05$, suggests no reason to reject the null hypothesis.

8.3.6 Test of variances

The question of two indpendent samples having equal variances can be tested through an F-test and is implemented in `VarianceFTest`.

For example, consider a test for equal variances over two levels of some treatment with hypotheses:

$$H_0 : \sigma_1 = \sigma_2, \quad H_A : \sigma_1 \neq \sigma_2$$

The F test considers the ratio σ_1^2/σ_2^2, so this may also have been specified in terms of that.

Two independent samples were collected from normally distributed populations, giving:

```
xs: -2.1, -1.8, 1.3, -0.9, 1.7, -2.0, -1.6, 3.8, -0.8, 5.5
ys: 7.6, 6.4, 7.2, 16.1, 6.6, 10.7, 11.0, 9.4
```

We enter the data and then pass this to the test:

```
xs = [-2.1, -1.8, 1.3, -0.9, 1.7, -2.0, -1.6, 3.8, -0.8, 5.5]
ys = [7.6, 6.4, 7.2, 16.1, 6.6, 10.7, 11.0, 9.4]
F = VarianceFTest(xs, ys)
pvalue(F, tail=:both)
```

```
0.566171938952384
```

> **Robustness of the F test**
>
> The F-test of variance, unlike, say, tests for the population mean, are sensitive to departures from the assumption of a normally distributed population. For this case, this was first noticed by Pearson (cf. [@Box_non_normality_10.1093/biomet/40.3-4.318]). The `LeveneTest` constructor implements Levene's test for comparing two *or* more variances for equality.

8.3.7 Goodness of fit test

Comparing categorical counting data to a specified multinomial model is done through a Chi-squared test. The expected count in each category is $E_i = np_i$, the count modeled by X_i and the statistic is $\sum(X_i - E_i)^2/E_i$.

Example 8.7 (Benford's law). Data to study Benford's law appeared in the linked article. The authors compiled data from COVID websites to get counts on the number of confirmed cases across many data sources. The first digit of each was tallied, and produced this data:

```
  1    2    3    4    5    6    7    8    9
2863 1342 1055  916  744  673  580  461  377
```

To test if the data follow Benford's law ($P(X = k) = \log_{10}(k+1) - \log_{10}(k)$) we have this as the null for p_k with an alternative of not so.

```
xs = [2863, 1342, 1055,  916,  744,  673,  580,  461,  377]

ks = 1:9
pks = log.(10, ks .+ 1) - log.(10, ks)

χ² = ChisqTest(xs, pks)
(pvalue=pvalue(χ²), tstat=χ².stat, df=χ².df)
```

```
(pvalue = 2.6484734636062243e-12, tstat = 71.34745744533376, df = 8)
```

This small p-value suggests the law does not exactly apply, though the general form of the law (monotonically decreasing probabilities) is certainly suggested.

Example 8.8. In [@CressieRead_10.2307_2345686], data on time passage and memory recall is provided. Following Haberman, a log-linear time trend is used to estimate the p_i: $\log(p_i) = \alpha + \beta \cdot i$.

That is, we will test:

$$H_0 : \log(p_i) = \alpha + \beta \cdot i, \quad H_A : \text{not so}$$

The values for α and β are estimated from the data. They use $\alpha \approx -2.1873, \beta \approx -0.0838$. (Their method is not quite the same, but close to, fitting a linear model to $\log(\hat{p}_i) = \alpha + \beta i$.)

Using this we can get the χ^2 statistic as follows, with the data read from the article:

```
xs = [15, 11, 14, 17, 5, 11, 10, 4, 8, 10, 7, 9, 11, 3, 6, 1, 1, 4]
alpha, beta = -2.1873, -0.0838
ps = exp.(alpha .+ beta*(1:18))
ps /= sum(ps) # make sum to exactly 1
χ² = ChisqTest(xs, ps)
χ².stat
```

```
22.716840144539344
```

We don't show the p value, as we need to consider an adjusted degrees of freedom. There are $s = 2$ parameters estimated from the data, so the degrees of freedom of this test statistic is $18 - 2 - 1 = 15$. The p-value is found by computing the area to the *right* of the observed value, giving:

```
1 - cdf(Chisq(18 - 2 - 1), χ².stat)
```

```
0.09033970932839053
```

This data set was used to illustrate the power-divergence statistic, which has a varying parameter λ. The Chi-squared test is $\lambda = 1$; a maximum-likelihood test is $\lambda = 0$; a value of $\lambda = 2/3$ is recommended by Cressie and Read as a robust general purpose value. We can test these, as follows:

```
Dict(l => 1 - cdf(Chisq(18 - 2 - 1), PowerDivergenceTest(xs; lambda=l, theta0=ps).stat)
     for l in [0, 2/3, 1, 5])
```

```
Dict{Float64, Float64} with 4 entries:
  0.0      => 0.0560191
  0.666667 => 0.0824898
  5.0      => 0.00204456
  1.0      => 0.0903397
```

The value of $\lambda = 5$ was used to show that larger values increase the size of the test statistic for this data, making the p-values smaller under the asymptotic distribution. The value for $i = 13$ of (X_i/E_i) is nearly 2; powers over 1 make this one term dominate the value of the statistic.

Example 8.9 (Two-way tables)**.** Let X and Y be two categorical variables summarizing a set of observations. Their counts could be summarized in a two-way table, say, with the X levels across the top, and the Y levels down the side. If there is no relationship between X and Y we would expect the proportions in each row to be *roughly* the same for each level of X. If X depends on Y, this would be expected. Put differently, if X and Y are not related, the expected count in cell row i column j would be $n \cdot p_{ij}$ *or* $(n \cdot P(Y = i)) * P(X = j)$, the latter first finds the expected number in the ith level of Y times the probability of the j level of X. That is, under an assumption of no association, the *marginal* probabilities should determine the cell probabilities.

Let there be c levels of X and r levels of Y. Viewing the table in a flat manner, there are $k = rc$ different

cells. If a multinomial model described the table, then the Chi-squared distribution would asymptotically describe the χ^2 statistic's sampling distribution. However, as there are $s = (r-1)+(c-1)$ parameters that would need estimating ($r-1$ for the Y probabilities as all r must add to 1, for example). So the degrees of freedom would be $k - s - 1 = rc - (r - 1 + c - 1) - 1 = (r-1)(c-1)$.

Suppose, a survey of many individuals was taken including questions on COVID-19 contraction and primary form of mitigation with the following data:

```
Mitigation/Contracted
                 Yes    No
Vaccination       10   500
Isolation          8   250
Face mask         15   600
Ivermectin        20    40
none              20   300
```

We enter this two-way table in aa a matrix, and the call `ChisqTest` to do the analysis:

```
cnts = [10 500;
        8  250;
        15 600
        20  40
        20 300] # r=5, c=2 -> 5 df.
χ² = ChisqTest(cnts)
(pvalue=pvalue(χ²), tstat=χ².stat, df = χ².df)
```

```
(pvalue = 4.5421259053037854e-30, tstat = 143.7051915805636, df = 4)
```

The small p-value for this made-up data suggests an association between primary mitigation and chance of contraction.

Example 8.10 (Two-sample test of proportions)**.** The methodology for two-way tables can be used for a two-sample test of proportions. Letting X count the successes and failures and Y the two different surveys.

Several news articles discussed a divide between "red" and "blue" states and their covid rates based on a politicalization of vaccinations. *Suppose* data was collected on whether a person with COVID-19 needed hospitalization was cross-tabulated with their county, classified broadly as "red" or "blue". The data is

```
Type / Hospitalization
       Yes    No
Red    100  1000
Blue    50   750
```

Test the data under the hypothesis of no association against an alternative of an association. Again, the contingency table is entered as a matrix and `ChisqTest` called to perform the analysis:

```
cnts = [100 1000;
         50  750]
χ² = ChisqTest(cnts)
(pvalue=pvalue(χ²), tstat=χ².stat, df = χ².df)
```

```
(pvalue = 0.023371321878153804, tstat = 5.140692640692601, df = 1)
```

We compare this to the following direct computation:

```
n1, n2 = ns = map(sum, eachrow(cnts))
phat1, phat2 = cnts[:, 1] ./  ns
phat = sum(cnts[:,1]) / sum(cnts)
SE = sqrt(phat*(1-phat) * (1/n1 + 1/n2))

Zobs = abs(phat1 - phat2)/SE  # use abs and double right tail

p_value = 2 * (1 - cdf(Normal(0, 1), Zobs))
```

```
0.023371321878153273
```

Either way, both produced a p-value smaller than the nominal $\alpha = 0.05$ level for this made-up data.

8.3.8 Likelihood ratio tests

Consider again a hypothesis test with a concrete alternative:

$$H_0 : \mu = \mu_0, \quad H_A : \mu = \mu_1$$

Suppose the population is $Normal(\mu, \sigma)$. We had a similar setup in the discussion on power, where for a T-test a specified three of a α, β, n, or an effect size allows the solving of the fourth using known facts about the T-statistic. The Neyman-Pearson lemma speaks to the *uniformly most powerful* test under this scenario with a single unknown parameter (the mean above, but could be the standard devation, etc.). This test can be realized as a likelihood ratio test also covers tests of more generality. Suppose the parameters being tested are call θ which sit in some subset $\Theta_0 \subset \Theta$. The non-direction alternative would be θ is in $\Theta \setminus \Theta_0$.

A test for this can be based on the *likelihood ratio statistic*:

$$\lambda = -2 \ln \frac{\sup_{\theta \in \Theta_0} L(\theta, x)}{\sup_{\theta \in \Theta} L(\theta, x)} = -2(\ln \sup_{\theta \in \Theta_0} L(\theta, x) - \ln \sup_{\theta \in \Theta} L(\theta, x)).$$

The sup just means the maximum value of the set, which above is either the specific set of values in the null or *all* possible values. The ratio here is in $[0, 1]$, the logarithm is then negative, the factor -2 means λ is in $[0, \infty)$. Under some assumptions, λ will have an *asymptotically* Chi-square distribution with the degrees of freedom given by the dimensionality of Θ.

While only in the simple case, the likelihood ratio test is the most powerful, the approach is much more general than the ad hoc tests described previously and mostly agrees with them.

Let's consider the case of a survey modeled by a binomial. We test $H_0 : p = 1/2$ against a two-sided alternative.

The log-likelihood function, $l(p, x)$, for the data is simply:

```
function loglik(θ, data)
    p = first(θ)
    x, n = data
    logpdf(Binomial(n, p), x)
end
```

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

The test statistic is found by computing $-2(l(1/2) - l(\hat{p}))$, where $\hat{p}$ maximizes $l(p, x)$.

We use `findmax` to identify the maximum.

Suppose our data is 60 of 100.

```
dat = (x=60, n=100)
ps = range(0, 1, length=201)
ls = loglik.(ps, Ref(dat))  # don't broadcast over data
l̂, i = findmax(ls)
p̂ = ps[i]
```

```
0.6
```

We see $\hat{p} = x/n$, which can be shown mathematically. The value of the log-likelihood ratio statistic can be found through:

```
p₀ = 1/2
ll_obs = -2*(loglik(p₀, dat) - l̂)
```

```
4.027102710137768
```

The $Chisq(1)$ distribution describes this statistic asymptotically. Supposing that to be valid, the p-value is computed by looking at the probability of a more extreme value under the null hypothesis, which are values larger:

```
1 - cdf(Chisq(1), ll_obs)
```

```
0.04477477232924443
```

Example 8.11 (The linear-regression model)**.** The *simple* linear regression model was given by $y_i = \beta_0 + \beta_1 x_i + e_i$, where for purposes of modeling, we will take e_i to have a $Normal(0, \sigma)$ distribution. We discuss

this model more formally in the susequent chapter, but here we see we can approach the model using maximum likelihood.

For a given data set, we test whether $\beta_1 = 0$ against a two-sided alternative.

Suppose the data is measured dosages of ivermectin with recovery time for COVID-19 infected patients.

```
dose (μg/kg): 100 100 200 200 400 400 600 800
time (days) :   5   5   6   4   5   8   6   6
```

The model has 3 parameters β_0, β_1 and σ, the standard deviation. We have the log-likelihood function given by:

```
function loglik(θ, data)
    # yᵢ ~ N(β₀ + β₁•x, σ) is model
    β₀, β₁, σ = θ
    y, x = data

    ll = 0.0
    for (yᵢ,xᵢ) ∈ zip(y, x)
        μᵢ = β₀ + β₁ * xᵢ
        D = Normal(μᵢ, σ)
        ll += logpdf(D, yᵢ)
    end
    ll
end
```

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

We have the data entered as:

```
x = [100, 100, 200, 200, 400, 400, 600, 800]
y = [5, 5, 6, 4, 5, 8, 6, 6];
```

We use `ProfileLikelihood` to model this. First we set up the model using an initial guess at the maximum likelihood estimators:

```
dat = (y, x)
θ₀ = [5.0, 0.0, 1.0]
nms = [:β₀, :β₁, :σ]
prob = LikelihoodProblem(loglik, θ₀;
                         data = dat,
                         syms = nms)
```

```
LikelihoodProblem. In-place: true
θ₀: 3-element Vector{Float64}
     β₀: 5.0
     β₁: 0.0
```

```
    σ: 1.0
```

We solve the problem through:

```
sol = mle(prob, Optim.NelderMead())
```

```
LikelihoodSolution. retcode: Success
Maximum likelihood: -11.466321890403758
Maximum likelihood estimates: 3-element Vector{Float64}
    β₀: 4.948881523162539
    β₁: 0.0019318326444597892
    σ: 1.0144493565817676
```

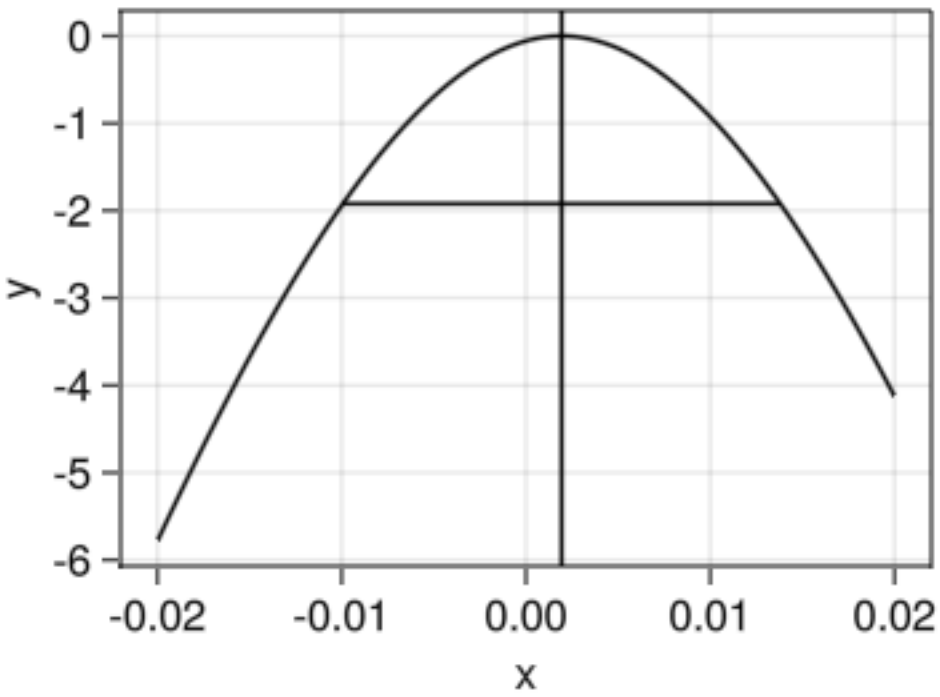
We see a small estimated value for β_1. Is it statistically significant? For this, we choose to profile the value, and rely on the relationship between confidence intervals and significance tests: if the 95% CI for β_1 includes 0, then the significance test would not reject the null.

```
lb = [-100.0, -100.0, 0.0]
ub = [ 100.0,  100.0, 100.0]  # -100 < β₀, β₁ < 100; 0 < σ < 100
resolutions = [200, 200, 200]
param_ranges = construct_profile_ranges(sol, lb, ub, resolutions)
prof = profile(prob, sol; param_ranges=param_ranges)
prof[:β₁]
```

```
Profile likelihood for parameter β₁. MLE retcode: Success
MLE: 0.0019318326444597892
95.0% CI for β₁: (-0.009936997524490673, 0.01380030329166489)
```

We can visualize the log-likelihood over the value in the 95% confidence interval with the following:

```
xs = range(-0.02, 0.02, length=100); ys = prof[:β₁].(xs)
p = data((x=xs, y=ys)) * visual(Lines) * mapping(:x, :y)
p += mapping([sol[:β₁]]) * visual(VLines)
ci = [extrema(prof.confidence_intervals[2])...]
p += data((x=ci, y = prof[:β₁].(ci))) * visual(Lines) * mapping(:x, :y)
draw(p)
```



Were a significance test desired, the test statistic requires one more optimization calculuation, this time the maximum log likelihood under H_0, which assumes a fixed value of β_1:

```
function loglik₀(θ, data)
    # yᵢ ~ N(β₀ + β₁•x, σ) is model
    β₀, σ = θ
    β₁ = 0.0     # H₀: β₁ = 0
    y, x = data

    ll = 0.0
    for (yᵢ,xᵢ) ∈ zip(y, x)
        μᵢ = β₀ + β₁ * xᵢ
        D = Normal(μᵢ, σ)
        ll += logpdf(D, yᵢ)
    end
    ll
end
```

```
loglik₀ (generic function with 1 method)
```

```
prob₀ = LikelihoodProblem(loglik₀, [5.0, 1.0];
                          data = (y, x),
                          syms = [:β₀, :σ])
sol₀ = mle(prob₀, Optim.NelderMead())
```

```
LikelihoodSolution. retcode: Success
Maximum likelihood: -12.193767351012552
Maximum likelihood estimates: 2-element Vector{Float64}
     β₀: 5.62501607088573
     σ: 1.1110596750702828
```

The likelihood ratio statistic is computed with the difference of the respective maximums, available through the `maximum` property of the solution:

```
l = -2 * (sol₀.maximum - sol.maximum)
```

```
1.4548909212175865
```

This observed value can be turned into a p-value using the asymptotically correct $Chisqs(1)$ distribution:

```
1 - cdf(Chisq(1), l)
```

```
0.22774478131827325
```

There is no evidence in the data to reject the null hypothesis of no effect.

Part III

Linear Models

Chapter 9

The linear regression model

We discuss the linear regression model in the following using linear algebra to quickly formulate the main results. For those unfamiliar with linear algebra, its use is only to illustrate various aspects that have pre-defined methods in `GLM`; no user-level linear algebra is necessary for computation.

For this, we utilize the following packages:

```
using StatsBase, Distributions, GLM, HypothesisTests
using CSV, DataFrames, CategoricalArrays, RDatasets
using GLMakie, AlgebraOfGraphics
using LinearAlgebra
```

9.1 Multiple linear regression

The simple linear regression model related a covariate variable, x, to a response variable Y through a formulation:

$$y_i = \beta_0 + \beta_1 \cdot x_i + e_i,$$

where β_0, β_1 are the parameters for the model that describe the average value of the Y variable for a given x value and the random errors, e_i, are *assumed* to be described to be a random sample from some distribution, usually $Normal(0, \sigma)$. Some inferential results require this sample to be iid.

The model for multiple regression is similar, though there are $k = 0, 1$, or more covariates accounted for in the notation:

$$y_i = \beta_0 + \beta_1 \cdot x_{1i} + \beta_2 \cdot x_{2i} + \cdots + \beta_k \cdot x_{ki} + e_i.$$

Following [@WackerlyMendenhallSchaeffer] we formulate the regression model using matrix algebra and quickly review their main results.

If there are n groups of data, then the main model matrix is the $n \times k$ matrix:

$$X = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1k} \\ 1 & x_{21} & x_{22} & \cdots & x_{2k} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nk} \end{bmatrix}.$$

The response, parameters, and errors are recorded in column vectors:

$$Y = [y_1, y_2, \cdots, y_n], \quad \beta = [\beta_0, \beta_1, \ldots, \beta_k], \quad e = [e_1, e_2, \ldots, e_n].$$

The collection of equations can then be written as $Y = X\beta + e$. The notation $\hat{Y} = X\beta$ is common, as the values $X\beta$ are used to *predict* the values of the response. The least squares problem to estimate the parameters is then to find β to minimize:

$$\|Y - \hat{Y}\|^2 = \|Y - X\beta\|^2.$$

Using calculus, it can be seen that any minimizer, $\hat{\beta}$, will satisfy $X'X\hat{\beta} = X'Y$, with X' denoting the transpose[1]. When written out these are called the *normal equations*, which, *if* the matrix X is non-degenerate (rank $k + 1$ in this case), can be solved algebraically by

$$\hat{\beta} = (X'X)^{-1}X'Y.$$

(Though this isn't the formula used in practice where inverse taking is usually done in a roundabout, though more computationally efficient, manner.)

We now assume the errors are an iid random sample from a distribution with mean 0 and variance σ^2. The matrix of covariances $COV(\beta_i, \beta_j)$ is called the covariance matrix and denoted $\Sigma_{\hat{\beta}\hat{\beta}}$. Under these assumptions, it can be shown to satisfy:

$$\Sigma_{\hat{\beta}\hat{\beta}} = \sigma^2(X'X)^{-1}.$$

The parameter σ^2 can be estimated. We have the *residuals* are given in vector form by $\hat{e} = Y - \hat{Y}$. But $\hat{Y} = \hat{H}Y$ where $\hat{H} = X(X'X)^{-1}X'$ is called the "hat" matrix and comes from $\hat{Y} = X\hat{\beta} = X((X'X)^{-1}X'Y) = (X(X'X)^{-1}X')Y$.

The hat matrix has the property of a *projection matrix* taking values in an n dimensional space and projecting onto a subspace described by the columns of the X matrix. This allows a geometric interpretation of the least-squares formulation.[2]

The sum of squared residuals is:

[1] In `Julia`, the `'` notation for a matrix denotes the adjoint of the matrix, which differs from the transpose for complex valued matrices.

[2] This geometric interpretation of projection gives insight into the presence of an F statistic later in this discussion.

$$\sum (Y_i - \hat{Y}_i)^2 = \|Y - \hat{Y}\|^2 = \|(1 - \hat{H})Y\|^2 = Y'(I - \hat{H})Y,$$

where I is the diagonal matrix of all ones that acts like an identity under multiplication.

The expected value can be computed to get $E(\|Y - \hat{Y}\|^2) = (n - 1 - k)\sigma^2$, which is used to *estimate* σ^2:

$$s^2 = \frac{\|Y - \hat{Y}\|^2}{n - 1 - k} = \frac{\sum (Y_i - \hat{Y}_i)^2}{n - 1 - k}.$$

(When $k = 0$, this is the same as the sample standard deviation with $\hat{Y}_i$ simply $\bar{Y}$.)

More is known of the $\hat{\beta}$, in particular the distribution of:

$$\frac{(\hat{\beta} - \beta) \cdot (X'X) \cdot (\hat{\beta} - \beta)}{\sigma_2}$$

Is $Chisq(k + 1)$and if the errors are from an iid random sample with population $Normal(0, \sigma)$, then this is independent of s^2 and so the ratio is F distributed, leading to the following for a $(1 - \alpha) \cdot 100\%$ *joint* confidence interval for β being:

$$(\hat{\beta} - \beta) \cdot (X'X) \cdot (\hat{\beta} - \beta) \leq (1 + k)s^2 F_{1-\alpha;1+k,n-1-k}.$$

While the error terms, e, are assumed to be *independent*, the residuals, $\hat{e}$, are not so, as one large residual must be offset by other smaller ones due to the minimization of the squared residuals. The matrix of their covariances can be expressed as $\Sigma_{\hat{e}\hat{e}} = \sigma^2(I - \hat{H})$. The *standardized* residuals account for the $(I - \hat{H})$ and are given by: $e_i/(s\sqrt{1 - \hat{H}_{ii}})$.

When the errors are an iid sample then the fitted values, $\hat{Y}$, are uncorrelated with the residuals.

If it is assumed the error population is normal, then the least-square estimates for β, given by $\hat{\beta} = (X'X)^{-1}XY$, are linear combinations of *independent* normal random variables, and consequently are normally distributed. (This is assuming the covariates are not random, or the Y values are conditionally independent.) For each i, we have $E(\hat{\beta}_i) = \beta_i$ and $SE(\hat{\beta}_i) = s_{\hat{\beta}_i} = s\sqrt{c_{ii}}$, where c_{ii} is the diagonal entry of $(X'X)^{-1}$. Moreover, the T-statistic:

$$T = \frac{\hat{\beta}_i - \beta_i}{SE(\hat{\beta}_{ii})},$$

will have a T-distribution with $n-k-1$ degrees of freedom, when the errors are iid and normally distributed. For a single estimate, $\hat{\beta}_i \pm t_{1-\alpha/2,n-1-k} s\sqrt{((X'X)^{-1})_{ii}}$ forms a $(1 - \alpha) \cdot 100\%$ confidence interval for β_i.

When the regression model is used for predicting the mean response for a given set of covariates, x_0, the predictor would be $\hat{\mu}_0 = x_0 \cdot \hat{\beta}$ (with the first value for x_0 being 1). The variance can be computed to give $VAR(\hat{\mu}_0) = \sigma^2 x_0'(X'X)^{-1}x_0$, which depends on the value of x_0. Confidence bands drawn by the `linear()`

visualization for a scatterplot of data use this formula and a related one to estimate a single value, not an average value. The dependence on x_0 gives the curve away from the center, $\bar{x}$.

A measure of how much variation in the response is explained by the dependence on the respective covariates (the *coefficient of determination*) is given by R^2 which is computed by

$$R^2 = 1 - \frac{SSR}{SST},$$

where $SSR = \sum \hat{e}_i^2 = \sum (y_i - \hat{y}_i)^2$ is the sum of the squared residuals and $SST = \sum (y_i - \bar{y})^2$ is the total sum of the squares. When the ratio SSR/SST is close to 1, then the model ($\hat{y}$) doesn't explain much of the variation compared to the null model with all the β_i's, $i \geq 1$ being 0 and the sample mean of the y, $\bar{y}$, used for prediction. Conversely, when the ratio is close to 0, then the model explains much of the variation. By subtracting this from 1, as is customary, we have the interpretation that R^2 explains $R^2 \cdot 100\%$ of the variation in the y values.

The value of R^2 can be made equal to 1 with enough variables; the *adjusted* R^2 value is a modification that weights SSR/SST by $(n-1)/(n-k-1)$ so more variables (bigger k) will makes this smaller.

9.1.1 Generic methods for statistical models

The `StatsBase` package defines methods for the above calculations and more. These are generic methods with similar usage for other models than the linear regression model discussed here. Table 9.1 lists several.

Table 9.1: Generic methods for statistical models defined in `StatsBase`. Those marked with a $*$ are defined on the `model` property of the `lm` output.

Method	Description
`coef`	Least squares estimates for intercept, $\hat{\beta}_0, \hat{\beta}_1, \ldots, \hat{\beta}_k$
`coefnames`	Names of coefficients
`stderror`	Standard errors of coefficients
`residuals`	Residuals, $y_i - \hat{y}_i$
`fitted`	$\hat{y}_i$ values
`predict`	Predict future values using $\hat{\beta}$s
`confint`	Confindence interval for estimated parameters
`modelmatrix`	Matrix (X) used in computations
`nobs`	n
`dof`	*Consumed* degrees of freedom (matrix rank plus 1)
`dof_residual`	residual degrees of freedom. $n - k - 1$
`r2`	Coefficient of determination, $1 - SSR/SST$
`adjr2`	Adjusted R^2, $1 - SSR/SST \cdot (n-1)/(n-1-k)$
`vcov`	Variance/Covariance matrix for the $\hat{\beta}$s
`dispersion`*	Estimate for σ, $\hat{\sigma} = \sqrt{SSR/(n-1-k)}$
`deviance`	Residual sum of squares, SSR
`nulldeviance`	Total sum of squares $SST = \sum (y_i - \bar{y})^2$.
`loglikelihood`	Log-likelihood of the model
`nullloglikelihood`	Log-likelihood of null model

Method	Description
`ftest`*	Compute F-test of two or more *nested* models
`aic`	Akaike's Information Criterion, $-2\log(L) + 2(k+2)$
`bic`	Bayesian Information Criterion, $-2\log(L) + (k+2)\log(n)$

Example 9.1 (Example of simple linear regression). Consider some simulated data on dosage amount of Ivermectin and days to recovery of COVID-19 fit by a simple linear model:

```
x = [100, 100, 200, 200, 400, 400, 600, 800]
y = [5, 5, 6, 4, 5, 8, 6, 6];
res = lm(@formula(y ~ x), (; x, y))  # uses named tuple to specify data
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, CholeskyPiv
```

```
y ~ 1 + x
```

```
Coefficients:
```

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	4.94886	0.744004	6.65	0.0006	3.12835	6.76938
x	0.00193182	0.00176594	1.09	0.3159	-0.00238928	0.00625292

The output shows the estimated coefficients, $\hat{\beta}_0$ and $\hat{\beta}_1$. These are computed by $(X'X)^{-1}XY$:

```
X = modelmatrix(res)
inv(X' * X) * X' * y
```

```
2-element Vector{Float64}:
 4.948863636363637
 0.0019318181818181804
```

These are also returned by the `coef` method, as in `coef(res)`.

The default output also computes confidence intervals and performs two-sided tests of whether the parameter is 0. Focusing on $\hat{\beta}_1$, we can find its standard error from $s\sqrt{c_{ii}}$. First we compute s:

```
n, k = length(x), 1
s² = sum(eᵢ^2 for eᵢ in residuals(res)) / (n - k - 1)
s = sqrt(s²)  # also dispersion(res.model)
C = inv(X'*X)
sbetas = s * sqrt.(diag(C))
```

```
2-element Vector{Float64}:
 0.7440036635973516
 0.0017659398031727332
```

More conveniently, these are returned by the `stederror` method:

```
stderror(res)
```

```
2-element Vector{Float64}:
 0.7440036635973516
 0.0017659398031727338
```

These are also the square root of the diagonal of the covariance matrix, $\Sigma_{\hat{\beta}\hat{\beta}}$, computed by the `vcov` method:

```
Σββ = vcov(res)
```

```
2×2 Matrix{Float64}:
  0.553541    -0.00109149
 -0.00109149   3.11854e-6
```

```
sqrt.(diag(Σββ))
```

```
2-element Vector{Float64}:
 0.7440036635973516
 0.0017659398031727338
```

The T statistic for $H_0 : \beta_1 = 0$ is then

```
β̂₁, SE₁ = coef(res)[2], stderror(res)[2]
T_obs = (β̂₁ - 0) / SE₁
```

```
1.0939320685492355
```

The p-value is then found directly with:

```
2 * ccdf(TDist(n-k-1), T_obs)
```

```
0.315945937293384
```

This computation is needed were there different assumed values than $\beta_1 = 0$ for the null.

The confidence intervals are of the form $\hat{\beta}_i \pm t_{1-\alpha/2;n-k-1} \cdot SE(\hat{\beta}_i)$. We find one for the intercept term, β_0:

```
alpha = 0.05
ta = quantile(TDist(n-k-1), 1 - alpha/2)
β̂₀ = coef(res)[1]
```

```
SE₀ = stderror(res)[1]
β̂₀ .+ ta * [-SE₀, SE₀]
```

```
2-element Vector{Float64}:
 3.128352254612003
 6.769375018115272
```

The `confint` method will also compute these, returning the values as rows in a matrix:

```
confint(res)
```

```
2×2 Matrix{Float64}:
  3.12835     6.76938
 -0.00238928  0.00625292
```

We compute the confidence interval for $\hat{\mu}$ when $x = 500$ using the variance formula above.

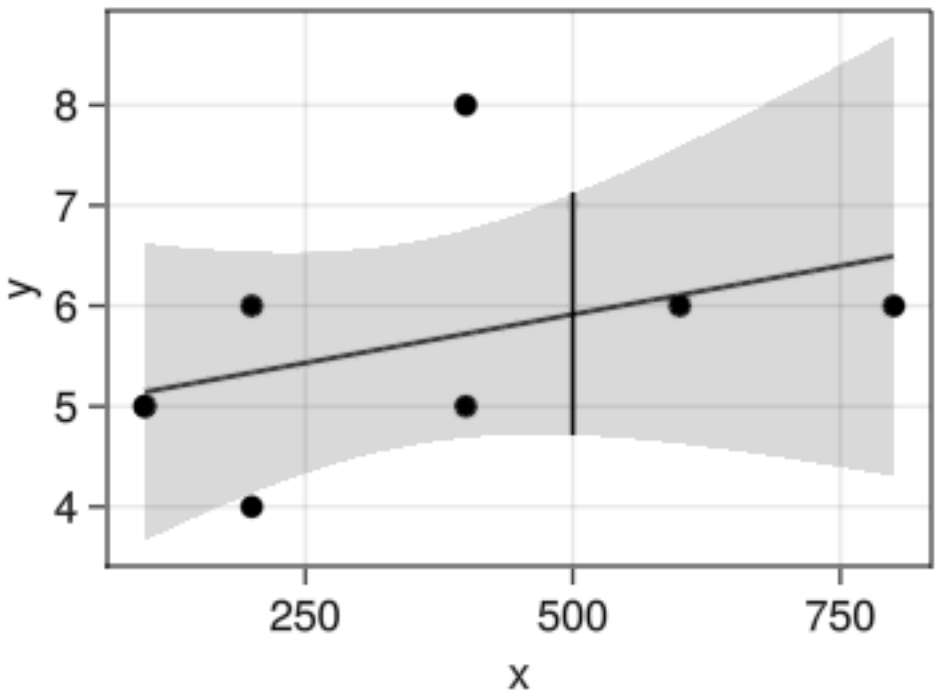
```
x0 = [1, 500]
μ̂ = predict(res, (x=[500],))[1]  # also  dot(inv(X'*X)*X'*y, x0)
SE = s * sqrt(x0' * inv(X' * X) * x0)
ci = μ̂ .+ ta * [-SE, SE]
```

```
2-element Vector{Float64}:
 4.711829667120805
 7.117715787424649
```

We can visualize (Figure 9.1) with the following commands:

```
p = AlgebraOfGraphics.data((;x, y)) *
    (visual(Scatter) + linear(;interval=:confidence)) * mapping(:x, :y)
p += data((x=[500, 500], y=ci)) * visual(Lines) * mapping(:x, :y)
draw(p)
```

Figure 9.1: Illlustration of linear regression model with confidence band drawn. The vertical line is computed directly for the value of $x = 500$.



The value of R^2 can be computed directly:

```
1 - sum(eᵢ^2 for eᵢ in residuals(res)) / sum((yᵢ - mean(y))^2 for yᵢ in y)
```

```
0.166283084004603
```

This can also be computed using several of the methods defined for model outputs by `GLM`:

```
r2(res), 1 - deviance(res)/nulldeviance(res)
```

```
(0.166283084004603, 0.166283084004603)
```

Whichever way, for this model a low R^2 implies the model does not explain much of the variance in the response.

Example 9.2 (Multiple regression example)**.** We give an example of multiple linear regression using a data set on various cereal boxes in a US grocery store.

```
cereal = dataset("MASS", "UScereal")
first(cereal, 2)
```

	Brand	MFR	Calories	Protein	Fat	Sodium	Fibre	Carbo	Sugars	
	String	Cat…	Float64	Float64	Float64	Float64	Float64	Float64	Float64	
1	100% Bran	N	212.121	12.1212	3.0303	393.939	30.303	15.1515	18.1818	…
2	All-Bran	K	212.121	12.1212	3.0303	787.879	27.2727	21.2121	15.1515	…

The data set collected numerous variables, here we consider numeric ones:

```
names(cereal, Real) |> permutedims
```

```
1×9 Matrix{String}:
 "Calories"  "Protein"  "Fat"  "Sodium"  …  "Sugars"  "Shelf"  "Potassium"
```

The initial model we consider has `Calories` as a response, and several covariates:

```
fm = @formula(Calories ~ Protein + Fat + Sodium + Carbo + Sugars)
res = lm(fm, cereal)
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, CholeskyPiv

Calories ~ 1 + Protein + Fat + Sodium + Carbo + Sugars

Coefficients:
──────────────────────────────────────────────────────────────────────────────
                   Coef.  Std. Error      t  Pr(>|t|)    Lower 95%    Upper 95%
──────────────────────────────────────────────────────────────────────────────
(Intercept)  -18.9515       3.61524    -5.24    <1e-05  -26.1856     -11.7175
Protein        3.99092      0.598282    6.67    <1e-08    2.79376      5.18808
Fat            8.86319      0.803833   11.03    <1e-15    7.25472     10.4717
Sodium         0.00266192   0.0107093   0.25    0.8046   -0.0187674    0.0240913
Carbo          4.91706      0.162811   30.20    <1e-36    4.59128      5.24284
Sugars         4.20214      0.216049   19.45    <1e-26    3.76983      4.63446
──────────────────────────────────────────────────────────────────────────────
```

The output shows what might have been anticipated: there appears to be no connection with `Sodium`, though were this data on dinner foods that might not be the case. The T-test for `Sodium` is a test of whether the slope based on `Sodium` is 0 – holding the other variables constant – and the large p-value would lead us to accept that hypotheses.

We drop this variable from the model and refit:

```
res = lm(@formula(Calories ~ Protein + Fat + Carbo + Sugars), cereal)
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, CholeskyPiv
```

```
Calories ~ 1 + Protein + Fat + Carbo + Sugars

Coefficients:
```

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	-18.7698	3.51272	-5.34	<1e-05	-25.7963	-11.7433
Protein	4.05056	0.543752	7.45	<1e-09	2.9629	5.13823
Fat	8.85889	0.797339	11.11	<1e-15	7.26398	10.4538
Carbo	4.92466	0.158656	31.04	<1e-37	4.6073	5.24202
Sugars	4.21069	0.211619	19.90	<1e-27	3.78739	4.634

How to interpret this? Each coefficient (save the intercept) measures the *predicted* change in *mean* number of calories for a 1-unit increase *holding* the other variables fixed. For example, it is suggested that adding 1 unit of protein additional would add nearly 4 calories per serving, on average.

Example 9.3 (Polynomial regression). Dickey provides an analysis of Galileo's falling ball data. Galileo rolled a ball down an elevated ramp at certain distances, the ball then jumped down a certain distance that varied depending on the height of the ramp. The collected data is:

```
release = [1000, 800, 600, 450, 300, 200, 100]
horizontal_distance = [573, 534, 495, 451, 395, 337, 253]
galileo = DataFrame(; release, horizontal_distance)
first(galileo, 3)
```

	release	horizontal_distance
	Int64	Int64
1	1000	573
2	800	534
3	600	495

With an assumption that the horizontal distance was related to $v_x t$ and t was found by solving for $0 = h - (1/2)gt^2$, we might expect h and t to be quadratically related. We consider, somewhat artifcially, the release height modeled *linearly* by the horizontal distance:

```
res = lm(@formula(y ~ x), (y=galileo.release, x=galileo.horizontal_distance))
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, CholeskyPiv

y ~ 1 + x

Coefficients:
```

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%

```
(Intercept)  -713.262    156.449     -4.56    0.0061  -1115.43     -311.098
x               2.77908    0.350392   7.93    0.0005      1.87837     3.67979
```

Without much effort, the small p-value would lead one to conclude the linear term is statistically significant. But Galileo might have expected a *quadratic* relationship and a modern reader might, as well, viewing Figure 9.2, such as modeled by the following:

```
res₂ = lm(@formula(y ~ x + x^2), (y=galileo.release, x=galileo.horizontal_distance))
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, CholeskyPiv
```

```
y ~ 1 + x + :(x ^ 2)
```

```
Coefficients:
```

```
                    Coef.     Std. Error      t  Pr(>|t|)     Lower 95%    Upper 95%
(Intercept)  600.054       110.543         5.43    0.0056  293.136       906.971
x             -4.00484       0.55582      -7.21    0.0020   -5.54805      -2.46164
x ^ 2          0.00818074    0.000665952  12.28    0.0003    0.00633176    0.0100297
```

The rules of `@formula` parse the above as adding a variable `x^2` to the model. Alternatively, the data frame could have been transformed to produce that variable. The output shows the test of $\beta_2 = 0$ would be rejected for reasonable values of α, as Galileo might have expected.

9.2 Categorical covariates

The linear regression model is more flexible than may appear on first introduction through simple regression.

For example, the regression model when there are no covariates is just a one-sample T-test, as seen from this example where a two-sided test of 0 mean is carried out.

```
y = [-0.2, 1.9, 2.7, 2.6, 1.5, 0.6]
lm(@formula(y ~ 1), (;y)) # using a named tuple for the data
```

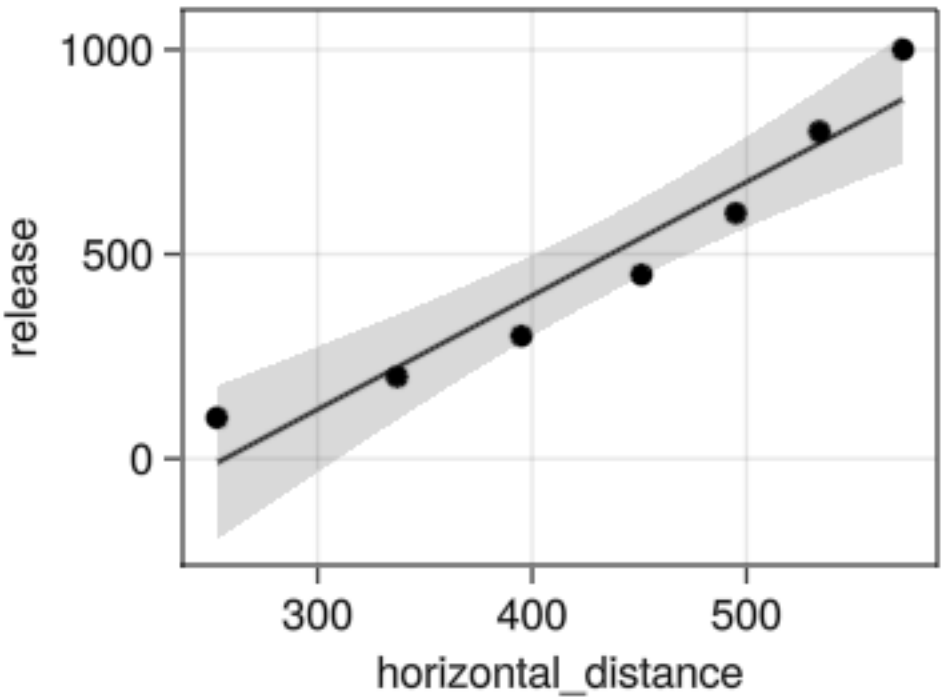
```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, CholeskyPiv
```

```
y ~ 1
```

```
Coefficients:
```

```
              Coef.  Std. Error     t  Pr(>|t|)  Lower 95%  Upper 95%
(Intercept)  1.51667    0.465773  3.26    0.0225   0.319359    2.71397
```

Figure 9.2: Scatter plot of falling-ball data of Galileo with a linear model fit. The curve suggests a quadratic model.



The formula usually has an *implicit* intercept, but here with no covariates listed, it must be made explicit. Compare the values with the following:

```
OneSampleTTest(y)
```

```
One sample t-test
-----------------
Population details:
    parameter of interest:   Mean
    value under h_0:         0
    point estimate:          1.51667
    95% confidence interval: (0.3194, 2.714)

Test summary:
    outcome with 95% confidence: reject h_0
    two-sided p-value:           0.0225

Details:
    number of observations:   6
    t-statistic:              3.2562360146885347
    degrees of freedom:       5
    empirical standard error: 0.46577295374940403
```

Further, the two-sample T-test (with equal variances assumed) can be performed through the regression model. After tidying the data, we fit a model:

```
y1 = [5,4,6,7]
y2 = [7,6,5,4,5,6,7]
df = DataFrame(group=["g1","g2"], value=[y1, y2])
d = flatten(df, [:value])
res = lm(@formula(value ~ group), d)
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, CholeskyPiv
```

```
value ~ 1 + group

Coefficients:
```

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	5.5	0.58757	9.36	<1e-05	4.17083	6.82917
group: g2	0.214286	0.736558	0.29	0.7777	-1.45192	1.88049

We can compare the computed values to those computed a different way:

```
EqualVarianceTTest(y2, y1)
```

```
Two sample t-test (equal variance)
----------------------------------
Population details:
    parameter of interest:   Mean difference
    value under h_0:         0
    point estimate:          0.214286
    95% confidence interval: (-1.452, 1.88)

Test summary:
    outcome with 95% confidence: fail to reject h_0
    two-sided p-value:           0.7777

Details:
    number of observations:   [7,4]
    t-statistic:              0.2909286827258563
    degrees of freedom:       9
    empirical standard error: 0.7365575380122867
```

However, some comments are warranted. We would have found a slightly different answer (a different sign) had we done `EqualVarianceTTest(y1, y2)`. This is because a choice is made if we consider $\bar{y}_1 - \bar{y}_2$ or $\bar{y}_2 - \bar{y}_1$ in the statistic.

In the use of the linear model, there is a new subtlety – the `group` variable is *categorical* and not numeric. A peek at the *model matrix* (`modelmatrix(res)`) will show that the categorical variable was *coded* with a 0 for each `g1` and 1 for each `g2`. The details are handled by the underlying `StatsModels` package which first creates a `ModelFrame` which takes a formula and the data; `ModelMatrix` then creates the matrix, X. The call to `ModelFrame` allows a specification of *contrasts*. The above uses the `DummyCoding`, which picks a base level (`"g1"` in this case) and then creates a variable for *each* other level, these variables having values either being `0` or `1`, and `1` only when the factor has that level. Using the notation $1_j(x_i)$ for this, we have the above call to `lm` fits the model $y_i = \beta_0 + \beta_1 \cdot 1_{g2}(x_i) + e_i$ and the model matrix shows this (2nd row below):

```
modelmatrix(res) |> permutedims # turned on side to save page space
```

```
2×11 Matrix{Float64}:
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 0.0  0.0  0.0  0.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
```

The model can't be $y_i = \beta_0 + \beta_1 \cdot 1_{g2}(x_i) + \beta_2 \cdot 1_{g1}(x_i) + e_i$ as there can't be a unique solution (the model $y_i = (\beta_0 + \beta_2) + \beta_2 \cdot 1_{g2}(x_i)$ would also fit). More mathematically, the model matrix, X, would have 3 columns, but one of them could be expressed as a sum of the other 2. This would mean X would not have full rank and the least-squares formula wouldn't have the form it does.

To fit a model with different contrasts, the `lm` function has a `contrast` keyword argument.

The above formulation does not require the factor to have just 2 levels; if there are k levels, then $k - 1$ variables are formed in the model.

Example 9.4 (Categorical covariates example)**.** Consider the `cereal` data set. The `Shelf` variable is numeric, but really it should be considered categorical for any study using a linear model, as differences between shelf 1 and 2 and shelf 2 and 3 should not be expected to be uniform (as they would were the values treated numerically). The following first ensures shelf is categorical, then fits a model on how the shelf placement impacts the number of calories:

```
cereal.shelf = categorical(cereal.Shelf)
res = lm(@formula(Calories ~ shelf), cereal)
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, CholeskyPiv
```

```
Calories ~ 1 + shelf

Coefficients:
```

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	119.477	13.3488	8.95	<1e-12	92.7936	146.161
shelf: 2	10.3388	18.878	0.55	0.5859	-27.3979	48.0754
shelf: 3	60.6692	16.9939	3.57	0.0007	26.6989	94.6394

The p-value for shelf 2 is consistent with there being no difference between shelf 1 and 2, but that of shelf 3 (counting from the floor) is significantly different from shelf 1 and would be interpreted as having 60 additional calories over shelf 1. (The lowest-shelf traditionally holds the least sold cereals, hence the most healthy in 1993 when this data was collected).

We can check that the model matrix has 2 variables a few ways: directly from the size (with first column being the intercept), and indirectly by the residual degrees of freedom:

```
size(modelmatrix(res)), nobs(res) - dof_residual(res) - 1 # dof_residual = n - k - 1
```

```
((65, 3), 2.0)
```

The omnibus F-test is a statistical test for a null hypothesis that $\beta_i = 0$ for all i except $i = 0$. It is implemented in the `ftest` method of `GLM`. It requires fitting the null model of just a constant, which we do with:

```
res₀ = lm(@formula(Calories ~ 1), cereal)  # null model
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, CholeskyPiv
```

```
Calories ~ 1

Coefficients:
```

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	149.408	7.74124	19.30	<1e-27	133.943	164.873

The test takes the model, which is stored in the `model` property:

```
ftest(res.model, res₀.model)
```

```
F-test: 2 models fitted on 65 observations
```

	DOF	ΔDOF	SSR	ΔSSR	R²	ΔR²	F*	p(>F)
[1]	4		198860.3072		0.2023			
[2]	2	-2	249295.4943	50435.1871	0.0000	-0.2023	7.8623	0.0009

Ignoring for now all but the bottom right number which gives the p-value, we see that this null model would be rejected.

9.3 Interactions

An *interaction* is when the effect of one explanatory variable depends on the values of a different explanatory variable. We see such a case in the following example.

Example 9.5. The `ToothGrowth` data set is included in base R and summarizes an experiment on the effect of vitamin C on tooth growth in guinea pigs. Each of the 60 animals in the study received one of three dose levels of vitamin C (0.5, 1, and 2 mg/day) by one of two delivery methods, orange juice or ascorbic acid. We load the data set using `RDatasets`:

```
ToothGrowth = dataset("datasets", "ToothGrowth")
first(ToothGrowth, 2)
```

	Len	Supp	Dose
	Float64	Cat…	Float64
1	4.2	VC	0.5
2	11.5	VC	0.5

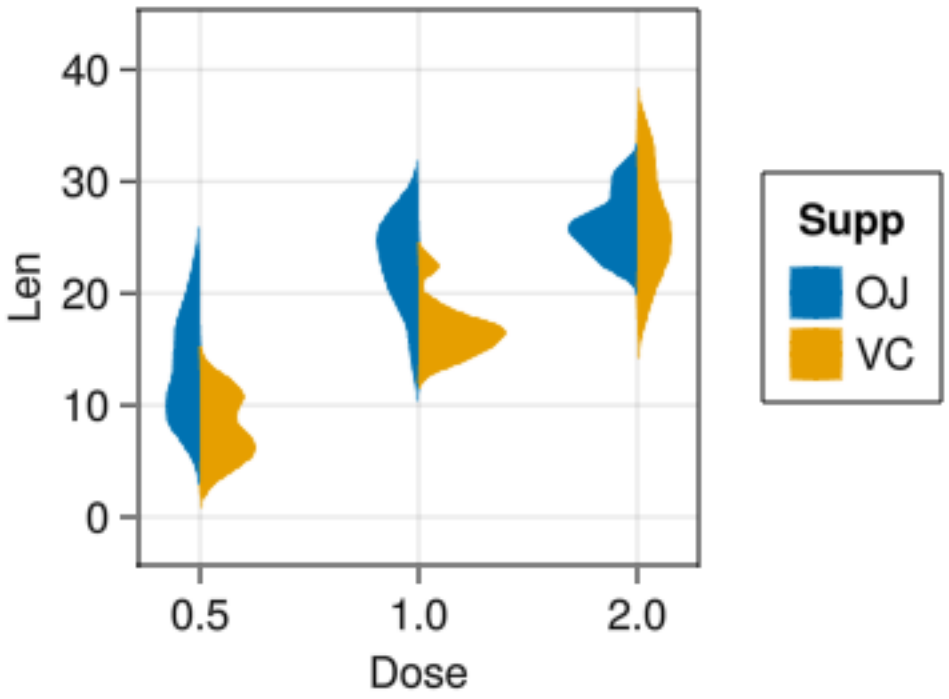
`Dose` is seen to be stored as a numeric variable (`Float64`), but we treat it as a categorical variable in the following. The table shows that the 6 different treatment pairs were tested on 10 animals.

```
ToothGrowth.Dose = categorical(ToothGrowth.Dose)
combine(groupby(ToothGrowth, 2:3), nrow)
```

	Supp	Dose	nrow
	Cat…	Cat…	Int64
1	OJ	0.5	10
2	OJ	1.0	10
3	OJ	2.0	10
4	VC	0.5	10
5	VC	1.0	10
6	VC	2.0	10

Figure 9.3 shows a violinplot with sides reflecting the distribution of the `:Supp` variable. A quick glance suggests that there may be some effect due to the dosage amount and a difference between the `OJ` and `VC` delivery.

Figure 9.3: ToothGrowth data set



We proceed to fit the *additive* model where `Supp` introduces one variable, and `Dose` two:

```
res = lm(@formula(Len ~ Supp + Dose), ToothGrowth)
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, CholeskyPiv
```

```
Len ~ 1 + Supp + Dose
```

Coefficients:

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	12.455	0.98828	12.60	<1e-17	10.4752	14.4348
Supp: VC	-3.7	0.98828	-3.74	0.0004	-5.67976	-1.72024
Dose: 1.0	9.13	1.21039	7.54	<1e-09	6.7053	11.5547
Dose: 2.0	15.495	1.21039	12.80	<1e-17	13.0703	17.9197

The small p-values support the visual observation that there are differences. The value for "`Supp: VC`", for instance, indicates that holding the dose equal, administering the dosage through citamin C and not ascorbic acid had a negative effect of -3.7 units on the predicted average tooth length.

Visually, the distribution of the `VC` variable seems to depend on the dosage. Perhaps there is an *interaction*.

For this data we can fit a model

$$\begin{aligned} y_i = \beta_0 + & \\ & \beta_1 \cdot 1_{\text{VC}}(\text{Supp}_i) + \\ & \beta_2 \cdot 1_{1.0}(\text{Dose}_i) + \beta_3 \cdot 1_{2.0}(\text{Dose}_i) + \\ & \beta_4 \cdot 1_{\text{VC},1.0}(\text{Supp}_i, \text{Dose}_i) + \beta_5 \cdot 1_{\text{VC},2.0}(\text{Supp}_i, \text{Dose}_i) + e_i \end{aligned}$$

The additional terms account for cases where, say, `Supp = VC` *and* `Dose = 1.0`.

Interactions are specified in the modeling formula through `*`. (Which when used also includes the additive terms without interactions. Plain interactions are specified with `&`.). The model is:

```
res₁ = lm(@formula(Len ~ Supp * Dose), ToothGrowth)
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, CholeskyPiv
```

```
Len ~ 1 + Supp + Dose + Supp & Dose
```

Coefficients:

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	13.23	1.14835	11.52	<1e-15	10.9277	15.5323
Supp: VC	-5.25	1.62402	-3.23	0.0021	-8.50596	-1.99404
Dose: 1.0	9.47	1.62402	5.83	<1e-06	6.21404	12.726
Dose: 2.0	12.83	1.62402	7.90	<1e-09	9.57404	16.086
Supp: VC & Dose: 1.0	-0.68	2.29671	-0.30	0.7683	-5.28462	3.92462
Supp: VC & Dose: 2.0	5.33	2.29671	2.32	0.0241	0.725381	9.93462

As expected from the graph, the p-value for the "`Supp: VC & Dose: 2.0`" case is significant.

As before, an F test can test the difference between the model with and without the interaction:

```
ftest(res.model, resᵢ.model)
```

```
F-test: 2 models fitted on 60 observations
```

	DOF	ΔDOF	SSR	ΔSSR	R²	ΔR²	F*	p(>F)
[1]	5		820.4250		0.7623			
[2]	7	2	712.1060	-108.3190	0.7937	0.0314	4.1070	0.0219

The small p-value suggests the interaction is statistically significant.

9.4 F test

Consider the linear regression model with parameters β and a significance test with some constraint on the parameters (e.g. $\beta_1 = 0$ or $\beta_1 = \beta_2$). Suppose the error terms are an iid random sample from a $Normal(0, \sigma)$ distribution. The a test of H_0 against an alternative of not H_0 can be carried out by considering the likelihood ratio statistic. The likelihood function for a set of parameters β is:

$$\begin{aligned} L(\beta, \sigma, x) &= \prod_{i=1}^{n} \frac{1}{(2\pi\sigma^2)^{n/2}} e^{-\frac{1}{2\sigma^2}(y_i - \hat{y}_i)^2} \\ &= \frac{1}{(2\pi)^{n/2}} \frac{1}{\sigma^n} e^{-\frac{1}{2\sigma^2}\sum_i (y_i - \hat{y}_i)^2}, \end{aligned}$$

where $\hat{y}_i = X\beta$ for some X related to the data. As e^{-x} is decreasing, L is maximized in β when $\sum_i (y_i - \hat{y}_i)^2$ is *minimimized* (a least squares estimate), say at $m(\hat{\beta})$. In terms of σ we take a logarithm and seek to maximize:

$$-\frac{n}{\sigma}\ln(2\pi) - n\ln(\sigma) - \frac{1}{2\sigma^2} m(\hat{\beta}).$$

This occurs at

$$\hat{\sigma} = \frac{m(\hat{\beta})}{n} = \frac{1}{n} SSR,$$

where SSR indicates the sum of the squared residuals, $y_i - \hat{y}_i$. (This is a *biased* estimate, as the divisor does not account for the degrees of freedom.)

The log-likelihood ratio statistic consider the two models: the restricted one under H_0 and the unrestricted one. This simplifies to

$$\lambda = -2 \ln \frac{L_0}{L} = n \ln(\frac{SSR_0}{SSR}),$$

with SSR_0 being the sum of the squared residuals under H_0 and SSR the sum of the squared residuals under the full model, which necessarily is smaller than SSR_0. The *asymptotic* distribution is $Chisq(k-p)$ where p variables are free in H_0.

The above says *if* SSR_0/SSR is sufficiently large it is statistically significant. Algebraically, the same thing can be said about

$$F = \frac{n-k-1}{k-p} \cdot \frac{SSR_0 - SSR}{SSR} = \frac{(SSR_0 - SSR)/(k-p)}{SSR/(n-k-1)}.$$

The distribution of $SSR/(n-k-1)$ is $Chisq(n-k-1)$. However, under these assumptions and under the null hypothesis, by Cochran's theorem $SSR_0 - SSR$ is *independent* of SSR and $(SSR_0 - SSR)/(k-p)$ is $Chisq(k-p)$. That is F has a $FDist(n-k-1, k-p)$ distribution. (Not asymptotically.)

This is used by `ftest` to compare two *nested* models. Nested means the parameters in the reduced model are related to those in the full model; no new ones are introduced. More technically, the column space of the reduced model is embedded in that of the full model.

Example 9.6. Consider again the output of the last call to `ftest` which checked for an interaction between the `Supp` and `Dose` variables in the `ToothGrowth` data:

```
res₀ = lm(@formula(Len ~ Supp + Dose), ToothGrowth)
res = lm(@formula(Len ~ Supp * Dose), ToothGrowth)
ftest(res₀.model, res.model)
```

```
F-test: 2 models fitted on 60 observations
```

	DOF	ΔDOF	SSR	ΔSSR	R²	ΔR²	F*	p(>F)
[1]	5		820.4250		0.7623			
[2]	7	2	712.1060	-108.3190	0.7937	0.0314	4.1070	0.0219

The *null* hypothesis is $H_0 : \beta_4 = \beta_5 = 0$. The full model has $k = 5$, the null has $p = 3$. The reported degrees of freedom is the *consumed degrees of freedom* which is this number of (linearly independent) columns in the model matrix $(k + 1)$ plus 1. The sum of squares can be computed directly or through the `deviance` method:

```
SSR₀, SSR = deviance(res₀), deviance(res) # or, say, sum(residuals(res₀).^2)
```

```
(820.4250000000001, 712.1059999999999)
```

The difference between the two is the numerator of the F statistic when divided by $2 = 5 - 3$ (or $7 - 5$). The denominator should be $SSR/(n - k - 1)$:

```
((SS₀ - SSR)/(5 - 3)) / (SSR / (60 - 1 - 5))
```

```
4.1069910940225265
```

The degrees of freedom $(n - 1 - k)$ is also calculated by

```
dof_residual(res)
```

```
54.0
```

The `ftest` can test more than rwo models. For example, suppose we test the null model with just an intercept, as in:

```
resᵢ = lm(@formula(Len ~ 1), ToothGrowth)
ftest(resᵢ.model, res₀.model, res.model)
```

```
F-test: 3 models fitted on 60 observations
```

	DOF	ΔDOF	SSR	ΔSSR	R^2	ΔR^2	F*	p(>F)
[1]	2		3452.2093		0.0000			
[2]	5	3	820.4250	-2631.7843	0.7623	0.7623	59.8795	<1e-16
[3]	7	2	712.1060	-108.3190	0.7937	0.0314	4.1070	0.0219

The output here has two p-values, the first testing if the additive model is statistically significant (with a very small p-value), the second testing, as mentioned, if the model with interaction is statistically significant compared to the additive model.

Example 9.7. We borrow an example from [@Faraway] to illustrate how the F-test can be used to test a null hypothesis of $H_0 : \beta_i = \beta_j$.

The dataset is in the `datasets` package of R:

```
savings = dataset("datasets", "LifeCycleSavings")
first(savings, 2)
```

	Country	SR	Pop15	Pop75	DPI	DDPI
	String15	Float64	Float64	Float64	Float64	Float64
1	Australia	11.43	29.35	2.87	2329.68	2.87
2	Austria	12.07	23.32	4.41	1507.99	3.93

We fit the full model for `SR` through:

```
res = lm(@formula(SR ~ Pop15 + Pop75 + DPI + DDPI), savings)
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, CholeskyPiv
```

```
SR ~ 1 + Pop15 + Pop75 + DPI + DDPI
```

```
Coefficients:
```

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	28.5661	7.35452	3.88	0.0003	13.7533	43.3788
Pop15	-0.461193	0.144642	-3.19	0.0026	-0.752518	-0.169869
Pop75	-1.6915	1.0836	-1.56	0.1255	-3.87398	0.490983
DPI	-0.000336902	0.000931107	-0.36	0.7192	-0.00221225	0.00153844
DDPI	0.409695	0.196197	2.09	0.0425	0.0145336	0.804856

A test of $H_0 : \beta_1 = \beta_2$ is done by preparing a variable `Pop15 + Pop75` (rather than a modification to the formula):

```
res1575 = lm(@formula(SR ~ Pop1575 + DPI + DDPI),
             transform(savings, [:Pop15, :Pop75] => (+) => :Pop1575))
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, CholeskyPiv
```

```
SR ~ 1 + Pop1575 + DPI + DDPI
```

```
Coefficients:
```

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	21.6093	4.88336	4.43	<1e-04	11.7796	31.439
Pop1575	-0.333633	0.103868	-3.21	0.0024	-0.542708	-0.124558
DPI	-0.000845101	0.000844351	-1.00	0.3221	-0.00254469	0.000854489
DDPI	0.390965	0.196871	1.99	0.0530	-0.00531671	0.787247

The `ftest` then can be applied:

```
ftest(res.model, res1575.model)
```

```
F-test: 2 models fitted on 50 observations
```

	DOF	ΔDOF	SSR	ΔSSR	R²	ΔR²	F*	p(>F)
[1]	6		650.7130		0.3385			

```
[2]    5    -1  673.6275  22.9145  0.3152  -0.0233  1.5847  0.2146
```

The large p value suggests no reason to reject this null.