Romeo and Julia, where Romeo is Basic Statistics

Bartlomiej Lukaszuk

Version: 2024-12-09

Contents

About

Hi, I'm Bart and this is my first 'experimental' book entitled (for now):

"Romeo and Julia, where Romeo is Basic Statistics"

In this book I will explore some basic statistics (the way I see it) with Julia[1]. Actually, I wrote the book for myself from the past. Too bad the past me won't be able to read it. Nevertheless, I hope it is gonna be of value to someone that resembles me from the old days. Additionally, I wrote it to solidify my own knowledge of statistics and Julia, after all they say we best teach that of what we learn :) Still, the book may contain some errors so don't believe everything you read here.

Who am I (not)? I'm not a statistician, a mathematician, or a computer scientist, but a biologist by education. Nowadays, I'm a programming enthusiast. To be honest, statistics was not my favorite subject when I was at college. I didn't quite get it then, I got it somewhat better now. Hopefully all this will make the book easier to digest, although possibly a little biased towards biology.

Oh yeah, I almost forgot, I'm not an English native speaker (keep that in mind while reading this book). Still, despite all the book's (and mine) flaws, I hope you will find it useful (it is available under Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International[2] license).

[1] https://julialang.org/

[2] http://creativecommons.org/licenses/by-nc-sa/4.0/

Why Julia

Before we jump into statistics I think we need to explain why should we use Julia[3]and not, e.g. Python[4]or R[5].

In other words, am I mad to use Julia for statistics instead of R (a project developed for statistical computing) or more popular (also in the field of Data Science) Python?

Well, I hope that I'm just biased. I like Julia because:

1. it's fast
2. it's simple
3. it's a pleasure to write programs with it
4. it's a less mainstream language
5. it's free and open source

Julia is fast

Once upon a time I wrote these three time consuming programs (so hold your horses, you may not want to run them):

```
# file: test.jl
for i in 1:1_000_000_000
    if i == 500_000_000
        println("Half way through. I counted to 500 million.")
    end
end
println("Done. I counted to 1 billion.")
```

```
# file: test.py
for i in range(1_000_000_000):
    if i == 500_000_000:
        print("Half way through. I counted to 500 million.")
print("Done. I counted to 1 billion.")
```

```
# file: test.r
for (i in 1:1000000000) {
```

[3]https://julialang.org/
[4]https://www.python.org/
[5]https://www.r-project.org/

```
  if (i == 500000000) {
    print("Half way through. I counted to 500 million.")
  }
}
print("Done. I counted to 1 billion.")
```

> ***Note:*** Python and Julia allow to write numbers either like this: `1000`, or like that `1_000`. The latter form uses _ to separate thousands, so more typing, but it is more legible.

Each program counts to 1 billion (1 with 9 zeros). Once it is half way through it displays an info on the screen and when it is done counting it prints another message.

The execution times of the scripts on my few-years old laptop (the specification is not that important):

1. Julia: ~1.5 [sec]
2. R: ~33 [sec]
3. Python3: ~50 [sec]

Granted, it's not a proper benchmark, and e.g. Python's numpy[6]library runs with the speed of C[7](so a bit faster than Julia). Nevertheless, the code that I write in Julia is consistently ~5-10 times faster than the code I write in the other two programming languages. This is especially evident when running computer simulations like the ones you may find in this book, still, it is just a subjective feeling.

Fun fact: A human being would likely need more than 32 years to count to 1 billion. Test yourself and show why. *Hint: try to estimate for how long you are alive [in seconds].*

Julia is simple

What I mean by Julia's simplicity is its nice, friendly and terse syntax.

[6]https://github.com/numpy/numpy

[7]https://en.wikipedia.org/wiki/C_(programming_language)

For instance to write a simple Hello world[8]program all I have to do is to type:

```
println("Hello World!")
```

then save and run the file.

For comparison a similar program in Java[9](a popular programming language) looks something like:

```
// file: HelloWorld.java
class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World");
    }
}
```

For me too much boilerplate code. The code that I don't want to type, read or process in my head. Additionally, in general a Java's code will probably not run faster than its Julia's counterpart. Moreover, the difference in lengths may be even greater for more complicated programs.

Pleasure to write

According to this stack overflow's survey[10]Julia got one of the best loved/dreaded ratio among the examined programming languages.

This is also true for me. I like writing programs in Julia (hopefully so will you).

Not mainstream

Not being 'a mainstream programming language' got its drawbacks (missing packages or community support, etc.). Luckily, Julia is big and

[8]https://en.wikipedia.org/wiki/%22Hello,_World!%22_program

[9]https://en.wikipedia.org/wiki/Java_(programming_language)

[10]https://survey.stackoverflow.co/2022/#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages

mature enough, it seems to be growing at a good pace, and got a pretty nice interoperability[11]with other programming languages.

Moreover, not being a mainstream language is like an opportunity, a gap to fill, a venue to explore (hence this book).

Julia is free

Julia is a free and open source programming language as stated on its official website[12]:

> Julia is an open source project with over 1,000 contributors. It is made available under the MIT license. The source code is available on GitHub.

OK, enough preaching, time for our first date with Julia.

[11]https://forem.julialang.org/ifihan/interoperability-in-julia-1m26
[12]https://julialang.org/

Julia - first encounter

Before we begin a warning. This book is not intended to be a comprehensive introduction to Julia programming. If you are looking for one try, e.g. Think Julia[13]. On the other hand, if the above-mentioned book is too much for you, and all you want is a short introduction see learn Julia in Y minutes[14]. For a video introduction try, e.g. A Gentle Introduction to Julia[15].

Still, regarding the current book, I think we need to cover some selected basics of the language in order to use it later. The rest of it we will catch 'on the fly'. Without further ado let's get our hands dirty.

Installation

In order to use Julia we need to install it first. So, now is the time to go to julialang.org[16], click 'Download' and choose the version suitable for your machine's OS.

To check the installation open the Terminal[17] and type:

```
julia --version
```

When I wrote those words the first time I used Julia version ~1.8, currently I'm using:

```
VERSION
```

1.10.7

running on a Gnu/Linux operating system. Keep that in mind, cause sometimes it may make a difference, e.g. reading the contents of a file (file path) may be OS specific.

[13] https://benlauwens.github.io/ThinkJulia.jl/latest/book.html
[14] https://learnxinyminutes.com/docs/julia/
[15] https://www.youtube.com/watch?v=4igzy3bGVkQ
[16] https://julialang.org/
[17] https://en.wikipedia.org/wiki/Terminal_emulator

At the bottom of the Julia's web page you will find 'Editors and IDEs' section presenting the most popular editors that will enable you to effectively write and execute pieces of Julia's code.

For starters I would go with Visual Studio Code[18]a popular, user friendly code editor for Julia. In the link above you will find the installation and configuration instructions for the editor.

From now on you'll be able to use it interactively (to run Julia code from this book).

All You need to do is to create a file, e.g. `chapter03.jl` (or open that file from the code snippets[19]), type the code presented in this chapter and run it by marking the code with your mouse and pressing `Ctrl+Enter`.

Language Constructs

Let's start by looking at some language features, namely:

1. Variables
2. Functions
3. Decision making
4. Repetition

Variables

The way I see it a variable is a box to store some value.

Type

```
x = 1
```

mark it (highlight it with a mouse) and run by pressing `Ctrl+Enter`.

This creates a variable (an imaginary box) named `x` (`x` is a label on the box) that contains the value `1`. The `=` operator assigns `1` (right side) to `x` (left side) [puts `1` into the box].

[18]https://www.julia-vscode.org/docs/dev/gettingstarted/#Installation-and-Configuration-1

[19]https://github.com/b-lukaszuk/RJ_BS_eng/tree/main/code_snippets/ch03

> ***Note:*** Spaces around mathematical operators like = are usually not necessary. Still, they improve legibility of your code.

Now, somewhat below type and execute

```
x = 2
```

Congratulations, now the value stored in the box (I mean variable `x`) is `2` (the previous value is gone).

Sometimes (usually I do this inside of functions, see Section 3.4) you may see variables written like that

```
z::Int = 4
```

or

```
zz::Float64 = 4.4
```

The `::` is a type declaration. Here by using `::Int` you promise Julia that you will store only integers[20](like: ..., −1, 0, 1, ...) in this box. Whereas by typing `::Float64` you declare to place only floats[21](like: ..., 1.1, 1.0, 0.0, 2.2, 3.14, ...) in that box.

> ***Note:*** You can either explicitly declare a type (with `::`) or let Julia guess it (when it's not declared, like in the case of `x` above). In either situation you can check the type of a variable with `typeof` function, e.g. `typeof(x)` or `typeof(zz)`.

Optional type declaration

In Julia type declaration is optional. You don't have to do this, Julia will figure out the types anyway. Still, sometimes it is worth to declare them (explanation in a moment). If you decide to do so, you should

[20]https://en.wikipedia.org/wiki/Integer

[21]https://en.wikipedia.org/wiki/Floating-point_arithmetic

declare a variable's type only once (the time it is first created and initialized with a value).

If you use a variable without a type declaration then you can freely reassign to it values of different types.

> ***Note:*** In the code snippet below # and all the text to the right of it is a comment, the part that is ignored by a computer but read by a human.

```
a = 1 # type is not declared
a = 2.2 # can assign a value of any other type
# the "Hello" below is a string (a text in a form readable by Julia)
a = "Hello"
```

But you cannot assign (to a variable) a value of a different type than the one you declared (you must keep your promises). Look at the code below.

This is OK.

```
b::Int = 1 # type integer declared
b = 2 # value of type integer delivered
```

But this is not OK (it's wrong! it's wroooong!).

```
c::Int = 1 # type integer declared
c = 3.3 # broke the promise, float delivered, it will produce an error
c = 3.1 # again, broke the promise, float delivered, expect error
```

Now a question arises. Why would you want to use a type declaration (like `::Int` or `::Float64`) at all?

In general you put values into variables to use them later. Sometimes, you forget what you placed there and may get an unexpected result (it may even go unnoticed for some time). For instance it makes more sense to use integer instead of string for some operations (e.g. I may wish to multiply `3` by `3` not `"three"` by `"three"`).

```
x = 3
x * x # works as you intended
```

9

```
x = "three"
x * x # the result may be surprising
```

threethree

> ***Note:*** Julia gives you a standard set of mathematical operators, like addition (+), subtraction (-), multiplication (*), division (/) and more (see the docs[22]).

The latter is an example of a so called string concatenation[23], it may be useful (as we will see later in this book), but probably it is not what you wanted.

To avoid such unexpected events (especially if instead of * you use your own function, see Section 3.4) you would like a guarding angel that watches over you. This is what Julia does when you require it by using type declarations (for now you need to take my word for it).

Moreover, declaring types sometimes may make your code run faster (although rather rarely[24]).

Additionally, some IDEs[25]work better (improved code completions, and hints) when you place type declarations in your code.

Personally, I like to use type declarations in my own functions (see the upcoming Section 3.4) to help me reason what they do. At first I write functions without types at all (it's easier that way). Once I got them

[22]https://docs.julialang.org/en/v1/base/math/#math-ops

[23]https://docs.julialang.org/en/v1/manual/strings/#man-concatenation

[24]https://discourse.julialang.org/t/learning-julia-for-scientists-who-are-beginning-programmers/108638/42

[25]https://en.wikipedia.org/wiki/Integrated_development_environment

running I add the types to them (it us useful for future reference, code maintenance, etc.).

Meaningful variable names

Name your variables well. The variable names I used before are horrible (*mea culpa, mea culpa, mea maxima culpa*). We use named variables (like `x = 1`) instead of 'loose' variables (you can type `1` alone in a script file and execute that line) to use them later.

You can use them later in time (reading and editing your code tomorrow or next month/year) or in space (using it 30 or 300 lines below). If so, the names need to be memorable (actually just meaningful will do :D). So whenever possible use: `studentAge = 19`, `bookTitle = "Dune"` (grammatical correctness is not that important) instead of `x = 19`, `y = "Dune"`.

You may want to check Julia's Docs for the allowed variable names[26] and the recommended stylistic conventions[27](for now, always start with a small letter, and use alphanumeric characters from the Latin alphabet). Personally, I prefer to use camelCaseStyle[28]so this is what you're gonna see here.

Floats comparisons

Be careful with = sign. In mathematics = means `equal to` and ≠ means `not equal to`. In programming = is usually an assignment operator (see Section 3.3 before). If you want to compare for equality you should use == (for `equal to`) and (`!=` for `not equal to`), examples:

```
1 == 1
```

true

```
2 == 1
```

[26]https://docs.julialang.org/en/v1/manual/variables/#man-allowed-variable-names

[27]https://docs.julialang.org/en/v1/manual/variables/#Stylistic-Conventions

[28]https://en.wikipedia.org/wiki/Camel_case

false

```
2.0 != 1.0
```

true

```
# comparing float (1.0) with integer (1)
1.0 != 1
```

false

```
# comparing integer (2) with float (2.0)
2 == 2.0
```

true

Be careful though because the comparisons of two floats are sometimes tricky, e.g.

```
(0.1 * 3) == 0.3
```

false

The problem here is not Julia (go ahead, try `(0.1 * 3) == 0.3` in another programming language), but computers in general. The result is `false` since some floats cannot be represented exactly as binary numbers (used internally by a computer), just like the fraction $\frac{1}{3}$ cannot be exactly represented in decimal numeral system ($\frac{1}{3}$ = 0.333...). If you are interested in more technical details see this StackOverflow's thread[29]. Anyway, this is how my computer sees `0.1 * 3`:

```
0.1 * 3
```

0.30000000000000004

and `0.3`

[29] https://stackoverflow.com/questions/8604196/why-0-1-3-0-3

```
0.3
```

0.3

The same caution applies to other comparison operators, like:

- `x > y` (x is greater than y),
- `x >= y` (x is greater than or equal to y),
- `x < y` (x is less than y),
- `x <= y` (x is less than or equal to y).

We will see how to deal with the lack of precision in comparisons later (see Section 3.8.2).

Other types

There are also other types (see Julia's Docs[30]), but we will use mostly those mentioned in this chapter, i.e.:

- floats[31]
- integers[32]
- strings[33]
- booleans[34]

The briefly aforementioned strings contain text of any kind. They are denoted by (optional type declaration) `::String` and you type them within double quotation marks (`"any text"`). If you ever want to place `"` in a string you need to use `\` (backslash) before it [otherwise Julia will terminate the string on the second `"` it encounters and throw an error (because it will be confused by the remaining, stray, characters)]. Moreover, if you wish the text to be displayed in the next line (e.g. in a figure's title like the one in Section 4.7.3) you should place `\n` in it. For instance:

```

```

[30] https://docs.julialang.org/en/v1/manual/types/
[31] https://en.wikipedia.org/wiki/Floating-point_arithmetic
[32] https://en.wikipedia.org/wiki/Integer
[33] https://en.wikipedia.org/wiki/String_(computer_science)
[34] https://en.wikipedia.org/wiki/Boolean_data_type

```
title = "I enjoy reading\n\"Title of my favorite book\"."
println(title)
```

Displays:

```
I enjoy reading
"Title of my favorite book".
```

on the screen.

A string is composed of individual characters (d'ooh!). An individual character (type `::Char`) is enclosed between single quotation marks. For instance, `'a'`, `'b'`, `'c'`, ..., `'z'` (also uppercase) are all individual characters. Whenever you want to type a single character you got a choice, either use `'a'` (single `Char`) or `"a"` (`String` composed of one `Char`). But when typing two or more characters that are 'glued' together you must use double quotations (`"ab"`). In the rest of the book we will focus mostly on strings, still, a bit more knowledge never hurt anyone (or did it?). In Solution to exercise 5 from Section 5.8.5 , we will see how to easily generate a complete alphabet (or a part of it, if you ever need one) with `Chars`. If you want to know more about the Strings[35] and Chars[36] just click the links to the docs that are to be found in this sentence.

The last of the earlier referenced types (boolean) is denoted as `::Bool` (note that in Julia types' names by convention start with a capital letter) and can take only two values: `true` or `false` (see the results of the comparison operations above in Section 3.3.3). `Bool`s are often used in decision making in our programs (see the upcoming Section 3.5) and can be used with a small set of logical operators[37] like AND (&&)

```

```

[35] https://docs.julialang.org/en/v1/manual/strings/

[36] https://docs.julialang.org/en/v1/manual/strings/#man-characters

[37] https://docs.julialang.org/en/v1/manual/mathematical-operations/#Boolean-Operators

```
# && returns true only if both values are true
# those return false:
# true && false
# false && true
# false && false
# this returns true:
true && true
```

true

OR (`||`)

```
# || returns true if any value is true
# those return true:
# true || false
# false || true
# true || true
# this returns false:
false || false
```

false

and NOT (`!`)

```
# ! flips the value to the opposite
# returns false: !true
# returns true
!false
```

true

Collections

Not only do variables may store a single value but they can also store their collections. The collection types that we will discuss here are `Vector` (technically `Vector` is a one dimensional `Array` but don't worry about that now), `Array` and `struct` (it is more like a composite type, but again at that moment we will not be bothered by that fact).

Vectors

```
myMathGrades = [3.5, 3.0, 3.5, 2.0, 4.0, 5.0, 3.0]
```

```
[3.5, 3.0, 3.5, 2.0, 4.0, 5.0, 3.0]
```

Here I declared a variable that stores my mock grades.

The variable type is `Vector` of numbers (each of type `Float64`, run `typeof(myMathGrades)` to check it). I could have declared its type explicitly as `::Vector{Float64}`. Instead I decided to let Julia figure it out.

You can think of a vector as a rectangular cuboid[38] box with drawers (smaller cube[39] shaped boxes). The drawers are labeled with consecutive numbers (indices) starting at 1 (we will get to that in a moment). The variable contains 7 grades in it, which you can check by typing and executing `length(myMathGrades)`.

You can retrieve a single element of the vector by typing `myMathGrades[i]` where `i` is some integer (the aforementioned index). For instance:

```
myMathGrades[3] # returns 3rd element
```

3.5

or

```
myMathGrades[end] # returns last grade
# equivalent to: myMathGrades[7], but here I don't have to count
elements
```

3.0

Be careful though, if You type a non-existing index like `myMathGrades[-1]`, `myMathGrades[0]` or `myMathGrades[10]` you will get an error (e.g. `BoundsError: attempt to access 7-element Vector{Float64} at index [0]`).

You can get a slice (a part) of the vector by typing

[38] https://en.wikipedia.org/wiki/Rectangular_cuboid
[39] https://en.wikipedia.org/wiki/Cube

```
myMathGrades[[2, 5]] # returns Vector with 2nd, and 5th element
```

```
[3.0, 4.0]
```

or

```
myMathGrades[[2, 3, 4]] # returns Vector with 2nd, 3rd, and 4th element
```

```
[3.0, 3.5, 2.0]
```

or simply

```
myMathGrades[2:4] # returns Vector with three grades (2nd, 3rd, and 4th)
# the slicing is [inclusive:inclusive]
```

```
[3.0, 3.5, 2.0]
```

The `2:4` is Julia's range[40] generator, with default syntax `start:stop` (both of which are inclusive). Assume that under the hood it generates a vector (check it by using collect[41] function, e.g, just run `collect(2:4)`). So, it gives us the same result as writing `myMathGrades[[2, 3, 4]]` by hand. However, the range syntax is more convenient (less typing especially for broad ranges). Now, let's say I want to print every other grade out of 100 grades, then I can go with `oneHunderedGrades[1:2:end]` and voila, a magic happened thanks to the `start:step:stop` syntax (`collect(1:2:end)` returns a vector of indices like `[1, 3, 5, 7, ..., 97, 99]`).

Interestingly, you can also choose elements of a vector by using `Bool`s.

```
boolIndices = [true, false, true, false, true, false, true]
```

[40] https://docs.julialang.org/en/v1/base/math/#Base.range

[41] https://docs.julialang.org/en/v1/base/collections/#Base.collect-Tuple%7BType,%20Any%7D

```
Bool[1, 0, 1, 0, 1, 0, 1]
```

Here, we define a vector composed only of `true` and `false` values. The above are printed in their short form as `1`s and `0`s, respectively. Now we may use it to get every other element of `myMathGrades` (actually every element for which the index position is `true`).

```
myMathGrades[boolIndices]
```

```
[3.5, 3.5, 4.0, 3.0]
```

The above may not look very useful right now (after all we need to type `true`/`false` for every index there is), but once we add a bit more syntax it becomes a nice way for data filtering (as we will see in Section 7.5).

One last remark, You can change the elements that are in a vector, e.g. like this:

```
myMathGrades[1] = 2.0
myMathGrades
```

```
[2.0, 3.0, 3.5, 2.0, 4.0, 5.0, 3.0]
```

or like that:

```
myMathGrades[2:3] = [5.0, 5.0]
myMathGrades
```

```
[2.0, 5.0, 5.0, 2.0, 4.0, 5.0, 3.0]
```

Again, remember about proper indexing. What you put inside (right side) should be compatible with indexing (left side), e.g `myMathGrades[2:3] = [2.0, 2.0, 2.0]` will produce an error (placing 3 numbers to 2 slots).

Arrays

A `Vector` is actually a special case of an `Array`, a multidimensional structure that holds data. The most familiar (and useful) form of it is a two-dimensional `Array` (also called `Matrix`). It has rows and columns. Previously, I stored my math grades in a `Vector`, but most likely I would like a place to keep my other grades. Here, I create an array that stores my grades from math (column1) and chemistry (column2).

```
myGrades = [3.5 3.0; 4.0 3.0; 5.0 2.0]
myGrades
```

```
3×2 Matrix{Float64}:
 3.5  3.0
 4.0  3.0
 5.0  2.0
```

I separated the values between columns with a space character and indicated a new row with a semicolon. Typing it by hand is not very interesting, but they come in handy as we will see later in the book.

As with vectors I can use indexing to get specific element(s) from a matrix, e.g.

```
myGrades[[1, 3], 2] # returns second column (rows 1 and 3) as Vector
```

```
[3.0, 2.0]
```

or

```
myGrades[:, 2] # returns second column (and all rows)
```

```
[3.0, 3.0, 2.0]
```

Above, the `:` symbol (when placed alone) means all indices in a row.

```
myGrades[1, :] # returns first row (and all columns)
```

```
[3.5, 3.0]
```

By analogy, here the `:` symbol (when placed alone) means all indices in a column.

```
myGrades[3, 2] # returns a value from third row and second column
```

2.0

Of course, also `Bool`s may be used for indexing.

```
myGrades[:, [false, true]] # all rows, second column
```

```
3×1 Matrix{Float64}:
 3.0
 3.0
 2.0
```

Moreover, we can apply the indexing to replace a particular element in a `Matrix`. For instance.

```
myGrades[3, 2] = 5
myGrades
```

```
3×2 Matrix{Float64}:
 3.5  3.0
 4.0  3.0
 5.0  5.0
```

or

```
myGrades[1:2, 1] = [5, 5]
myGrades
```

```
3×2 Matrix{Float64}:
 5.0  3.0
 5.0  3.0
 5.0  5.0
```

As with a `Vector` also here you must pay attention to proper indexing.

When dealing with `Arrays` (or `Vectors` which are one dimensional arrays) one needs to be cautious not to change their contents accidentally.

In case of atomic variables the values are assigned/passed as copies (i.e. a new number `3` is put to the box, the old number in the variable `x` is unaffected). Observe.

```
x = 2
y = x # y contains the same value as x
y = 3 # y is assigned a new value, x is unaffected

(x, y)
```

```
(2, 3)
```

> ***Note:*** The `(x, y)` returns `Tuple` (see Tuple in the docs[42]) and it is there to show both `x` and `y` in one line. You may think of `Tuple` as something similar to `Vector` but written with parenthesis `()` instead of square brackets `[]`. Additionally, you cannot modify elements of a tuple after it was created (so, if you got `z = (1, 2, 3)`, then `z[2]` will work fine (since it just returns an element), but `z[2] = 8` will produce an error). Technically speaking, you could just type `x, y` and run the line to get a tuple (test it out), but I prefer to use parenthesis to be explicit.

However, the arrays are assigned/passed as references.

```
xx = [2, 2]
yy = xx # yy refers to the same box of drawers as xx
yy[1] = 3 # new value 3 is put to the first drawer of the box pointed by
yy

# both xx, and yy are changed, cause both point at the same box of
```

[42] https://docs.julialang.org/en/v1/manual/functions/#Tuples

```
drawers
(xx, yy)
```

```
([3, 2], [3, 2])
```

As stated in the comments to the code snippet above, here both `xx` and `yy` variables point at (reference to) the same box of drawers (imagine the same box of drawers got two labels `xx` and `yy` stuck to it next to each other). So, when we change a value in one drawer, then both variables reflect the change. If we want to avoid that we can, e.g. make a copy[43] of the `Vector`/`Array` like so:

```
xx = [2, 2]
# yy refers to a different box of drawers
# with the same (copied) numbers inside
yy = copy(xx)
yy[1] = 3 # this does not affect xx

(xx, yy)
```

```
([2, 2], [3, 2])
```

Structs

Another Julia's type worth mentioning is struct[44]. It is a composite type (so it contains other type(s) inside).

Let's say I want to have a thing that resembles fractions that we know from mathematics. It should allow to store the data for numerator and denominator ($\frac{numerator}{denominator}$). Let's use `struct` for that

```
struct Fraction
    numerator::Int
    denominator::Int
end

fr1 = Fraction(1, 2)
fr1
```

[43] https://docs.julialang.org/en/v1/base/base/#Base.copy
[44] https://docs.julialang.org/en/v1/base/base/#struct

```
Fraction(1, 2)
```

> ***Note:*** By convention `Structs`' names start with a capital letter.

If I ever wanted to get a component of the `struct` I can use the dot syntax, like so

```
fr1.numerator
```

1

> ***Note:*** If you type `fr1.` and press TAB key then you should see a hint with the available field names. You may choose one with arrow keys and confirm it with Enter key.

or

```
fr1.denominator
```

2

Of course, as you probably have guessed, there is no need to define your own type for fraction since Julia is already equipped with one. It is called Rational[45]. For convenience the fraction is written as

```
1//2 # equivalent to: Rational(1, 2)
```

1//2

Notice the double slash character (`//`).

In general, `structs` are worth knowing. A lot of libraries (see Section 3.7) define their own `struct` objects and we may want to extract their content using the dot syntax (as we probably sometimes will in the upcoming sections).

[45]https://docs.julialang.org/en/v1/base/numbers/#Base.Rational

OK, enough about the variables, time to meet functions.

Functions

Functions are doers, i.e encapsulated pieces of code that do things for us. Optimally, a function should be single minded, i.e. doing one thing only and doing it well. Moreover since they do stuff their names should contain verbs[46](whereas variables' names should be composed of nouns[47]).

We already met one of many Julia's built in functions, namely `println` (see Section 2.2). As the name suggests it prints something (like a text) to the screen (more precisely standard output[48]).

Mathematical functions

We can also define some functions on our own:

```
function getRectangleArea(lenSideA::Real, lenSideB::Real)::Real
    return lenSideA * lenSideB
end
```

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

Here I declared Julia's version of a mathematical function[49]. It is called `getRectangleArea` and it calculates (surprise, surprise) the area of a rectangle[50].

To do that I used the keyword `function`. The `function` keyword is followed by the name of the function (`getRectangleArea`). Inside the parenthesis are arguments of the function. The function accepts two arguments `lenSideA` (length of one side) and `lenSideB` (length of the other side) and calculates the area of a rectangle (by multiplying `lenSideA` by `lenSideB`). Both `lenSideA` and `lenSideB` are of type `Real`.

[46]https://en.wikipedia.org/wiki/Verb
[47]https://en.wikipedia.org/wiki/Noun
[48]https://en.wikipedia.org/wiki/Standard_streams#Standard_output_(stdout)
[49]https://en.wikipedia.org/wiki/Function_(mathematics)
[50]https://en.wikipedia.org/wiki/Rectangle#Formulae

It is Julia's representation of a real number[51], it encompasses (it's kind of a supertype), among others, `Int` and `Float64` that we encountered before. The ending of the first line, `)::Real`, signifies that the function will return a value of type `Real`. The stuff that function returns is preceded by the `return` keyword. The function ends with the `end` keyword.

> ***Note:*** A Julia's function does not need the `return` keyword since it returns the result of its last expression. Still, I prefer to be explicit.

Time to run our function and see how it works.

```
getRectangleArea(3, 4)
```

12

```
getRectangleArea(1.5, 2)
```

3.0

> ***Note:*** In some other languages, e.g. Python, you could use the function like: `getRectangleArea(3, 4)`, `getRectangleArea(lenSideA=3, lenSideB=4)` or `getRectangleArea(lenSideB=4, lenSideA=3)`. However, for performance reasons (and perhaps due to its Lisp heritage) Julia's functions accept arguments in a positional manner. Therefore, here you may only use `getRectangleArea(3, 4)` form. Internally, the first argument (3) will be assigned to the local variable `lenSideA` and the second (4) to the local variable `lenSideB` inside the `getRectangleArea` function. Keep that in mind since the order of the arguments may sometimes make a difference (e.g. if `getRectangleArea` relied on division instead of multiplication).

[51]https://en.wikipedia.org/wiki/Real_number

Hmm, OK, I got `getRectangleArea` and what if I need to calculate the area of a square[52]. You got it.

```
function getSquareArea(lenSideA::Real)::Real
    return getRectangleArea(lenSideA, lenSideA)
end
```

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

> ***Note:*** The argument (`lenSideA`) of `getSquareArea` is only known inside the function. Another function can use the same name for its arguments and it will not collide with this one. For instance, `getRectangleArea(lenSideA::Real, lenSideB::Real)` will receive the same number twice, which `getSquareArea` knows as `lenSideA`, but `getRectangleArea` will see only the numbers (it will receive their copies) and it will name them `lenSideA` and `lenSideB` for its own usage.

Here I can either write its body from scratch (`return lendSideA * lenSideA`) or reuse (as I did) our previously defined `getRectangleArea`. Lesson to be learned here, functions can use other functions. This is especially handy if those inner functions are long and complicated. Anyway, let's see how it works.

```
getSquareArea(3)
```

9

Appears to be working just fine.

A quick reference to the topic we discussed in Section 3.3.1 . Here typing `getRectangleArea("three", "three")` will produce an error. Now, I can spot it right away, read the error's message and based on that correct my code so the result is in line with my expectations

[52]https://en.wikipedia.org/wiki/Square#Perimeter_and_area

Functions with generics

Now, let's say I want a function `getFirstElt` that accepts a vector and returns its first element (vectors and indexing were briefly discussed in Section 3.3.5).

```
# works fine for non-empty vectors
function getFirstElt(vect::Vector{Int})::Int
    return vect[1]
end
```

It looks OK (test it, e.g. `getFirstElt([1, 2, 3])`. However, the problem is that it works only with integers (or maybe not, test it out). How to make it work with any type, like `getFirstElt(["Eve", "Tom", "Alex"])` or `getFirstElt([1.1, 2.2, 3.3])`?

One way is to declare separate versions of the function for different types of inputs, i.e.

```
function getFirstElt(vect::Vector{Int})::Int
    return vect[1]
end

function getFirstElt(vect::Vector{Float64})::Float64
    return vect[1]
end

function getFirstElt(vect::Vector{String})::String
    return vect[1]
end
```

```
getFirstElt (generic function with 3 methods)
```

> ***Note:*** The function's name is exactly the same in each case. Julia will choose the correct version (aka method, see the output of the code snippet above) based on the type of the argument (`vect`) send to the function, e.g. `getFirstElt([1, 2, 3])`, `getFirstElt([1.1, 2, 3.0])`, and `getFirstElt(["a", "b", "c"])` for the three versions above, respectively.

But that is too much typing (I retyped a few times virtually the same code). The other way is to use no type declarations.

```
function getFirstEltVer2(vect)
    return vect[1]
end
```

It turns out that you don't have to declare function types in Julia (just like in the case of variables, see Section 3.3.1) and a function may work just fine.

Still, a die hard 'typist' (if I may call a person this way) would probably use so called generic types, like:

```
function getFirstEltVer3(vect::Vector{T})::T where T
    return vect[1]
end
```

Here we said that the vector is composed of elements of type `T` (`Vector{T}`) and that the function will return type `T` (see `)::T`). By typing `where T` we let Julia know that `T` is our custom type that we just made up and it can be any Julia's built in type whatsoever (but what it is exactly will be determined once the function is used). We needed to say `where T` otherwise Julia would throw an error (since it wouldn't be able to find its own built in type `T`). Anyway, we could replace `T` with any other letter (or e.g. two letters) of the alphabet (`A`, `D`, or whatever) and the code would still work.

One last remark, it is customary to write generic types with a single capital letter. Notice that in comparison to the function with no type declarations (`getFirstEltVer2`) the version with generics (`getFirstEltVer3`) is more informative. You know that the function accepts a vector of some elements, and you know that it returns a value of the same type as the elements that build that vector.

Of course, that last function we wrote for fun (it was fun for me, how about you?). In reality Julia already got a function with a similar functionality (see Base.first[53]).

> ***Note:*** Usually functions from Base package, like `Base.first` mentioned above, may be used in a shorter form (without the prefix), i.e. this: `first([1, 2, 3, 4])`.

Anyway, as I wrote before if you don't want to use types then don't, Julia gives you a choice. When I begun to write my first computer programs, I preferred to use programming languages that didn't require types. However, nowadays I prefer to use them for the reasons similar to those described in Section 3.3.1 so be tolerant and bear with me.

Functions operating on structs

Functions may also work on custom types like the ones created with `struct`. Do you still remember our `Fraction` type from Section 3.3.8 ? I hope so.

Let's say I want to define a function that adds two fractions. I can proceed like so

```
function add(f1::Fraction, f2::Fraction)::Fraction
    newDenom::Int = f1.denominator * f2.denominator
    f1NewNom::Int = newDenom / f1.denominator * f1.numerator
    f2NewNom::Int = newDenom / f2.denominator * f2.numerator
    newNom::Int = f1NewNom + f2NewNom
    return Fraction(newNom, newDenom)
end

add(Fraction(1, 3), Fraction(2, 6))
```

```
Fraction(12, 18)
```

[53]https://docs.julialang.org/en/v1/base/collections/#Base.first

Note: The variables `newDenom`, `f1NewNom`, `f2NewNom`, `newNom` are local, e.g. they are created and exist only inside the function when it is called (like here with `add(Fraction(1, 3), Fraction(2, 6))`) and do not affect the variables outside the function even if they happened to have the same names.

Works correctly, but the addition algorithm is not optimal (for now you don't have to worry too much about the function's hairy internals). Luckily the built in `Rational` type (Section 3.3.8) is more polished. Observe

```
# equivalent to: Rational(1, 3) + Rational(2, 6)
1//3 + 2//6
```

2//3

Much better ($\frac{12}{18} = \frac{12/6}{18/6} = \frac{2}{3}$). Of course also other operations like subtraction, multiplication and division work for `Rational`.

We will meet some functions operating on `structs` when we use custom made libraries like `HypothesisTests` (abbreviated `Ht`), e.g. `Ht.pvalue` that works on the object (struct) returned by `Ht.OneWayANOVATest` (see the upcoming Section 5.5). Again, for now don't worry about it too much.

Functions modifying arguments

Previously (see Section 3.3.5) we said that we can change elements of a vector. Sometimes even unintentionally, because, e.g. we may forget that `Arrayss`/`Vectors` are assigned/passed by references (as mentioned in Section 3.3.7).

```
function wrongReplaceFirstElt(
    ints::Vector{Int}, newElt::Int)::Vector{Int}
    ints[1] = newElt
    return ints
end

xx = [2, 2]
yy = wrongReplaceFirstElt(xx, 3)
```

```
# unintentionally we changed xx defined outside a function
(xx, yy)
```

```
([3, 2], [3, 2])
```

Let's try to re-write the function that changes the first element improving upon it at the same time.

```
# the function works fine for non-empty vectors
function replaceFirstElt!(vect::Vector{T}, newElt::T) where T
    vect[1] = newElt
    return nothing
end
```

> ***Note:*** The function's name ends with `!` (exclamation mark). This is one of the Julia's conventions to mark a function that modifies its arguments.

In general, you should try to write a function that does not modify its arguments (as modification often causes errors, especially in big programs). However, such modifications are sometimes useful, therefore Julia allows you to do so, but you should always be explicit about it. That is why it is customary to end the name of such a function with `!` (exclamation mark draws attention).

Additionally, observe that `T` can be of any type, but we require `newElt` to be of the same type as the elements in `vect`. Moreover, since we modify the arguments we wrote `return nothing` (to be explicit we do not return a thing) and removed returned type after the function's name, i.e. we used [`) where T` instead of `)::Vector{T} where T`].

Let's see how the function works.

```
x = [1, 2, 3]
y = replaceFirstElt!(x, 4)
(x, y)
```

```
([4, 2, 3], nothing)
```

Let me finish this subsection by mentioning a classical example of a built-in function that modifies its argument. The function is push![54]. It adds elements to a collection (e.g. `Arrays`, or `Vectors`). Observe:

```
xx = [] # empty vector
push!(xx, 1, 2) # now xx is [1, 2]
push!(xx, 3) # now xx is [1, 2, 3]
push!(xx, 4, 5) # now xx is [1, 2, 3, 4, 5]
```

I mentioned it since that was my favorite way of constructing a vector (to start with an empty vector and add elements one by one with a `for` loop that we will meet in Section 3.6.1) back in the day when I started my programming journey. Nowadays I do it a bit differently, but I thought it would be good to mention it in case you find it useful while solving some exercises from this book.

Side Effects vs Returned Values

Notice that so far we encountered two types of Julia's functions:

- those that are used for their side effects (like `println`)
- those that return some results (like `getRectangleArea`)

The difference between the two may not be clear while we use the interactive mode. To make it more obvious let's put them in the script like so:

```
# file: sideEffsVsReturnVals.jl

# you should define a function before you call it
function getRectangleArea(lenSideA::Real, lenSideB::Real)::Real
    return lenSideA * lenSideB
end

println("Hello World!")

getRectangleArea(3, 2) # calling the function
```

[54] https://docs.julialang.org/en/v1/base/collections/#Base.push!

After running the code from terminal:

```
cd folder_with_the_sideEffsVsReturnVals.jl
julia sideEffsVsReturnVals.jl
```

I got printed on the screen:

```
Hello World!
```

That's it. I got only one line of output, the rectangle area seems to be missing. We must remember that a computer does only what we tell it to do, nothing more, nothing less. Here we said:

- print "Hello World!" to the screen (actually standard output[55])
- calculate and return the area of the rectangle (but we did nothing with it)

In the second case the result went into the void ("If a tree falls in a forest and no one is around to hear it, does it make a sound?").

If we want to print both pieces of information on the screen we should modify our script to look like:

```
# file: sideEffsVsReturnVals.jl

# you should define a function before you call it
function getRectangleArea(lenSideA::Real, lenSideB::Real)::Real
    return lenSideA * lenSideB
end

println("Hello World!")

# println takes 0 or more arguments (separated by commas)
# if necessary arguments are converted to strings and printed
println("Rectangle area = ", getRectangleArea(3, 2), " [cm^2]")
```

Now when we run `julia sideEffsVsReturnVals.jl` from terminal, we get:

```

```

[55]https://en.wikipedia.org/wiki/Standard_streams#Standard_output_(stdout)

```
Hello World!
Rectangle area = 6 [cm^2]
```

More information about functions can be found, e.g. in this section of Julia's Docs[56].

If you ever encounter a built in function that you don't know, you may always search for it in the docs[57](search box: top left corner of the page).

Decision Making

In everyday life people have to make decisions and so do computer programs. This is the job for `if ... elseif ... else` constructs.

If ..., or Else ...

To demonstrate decision making in action let's say I want to write a function that accepts an integer as an argument and returns its textual representation. Here we go.

```
function turnInt2string(num::Int)::String
    if num <= 0
        return "zero or less"
    elseif num == 1
        return "one"
    elseif num == 2
        return "two"
    else
        return "three or above"
    end
end

(turnInt2string(2), turnInt2string(5)) # a tuple with results
```

```
("two", "three or above")
```

The general structure of the construct goes like this:

```
```

[56]https://docs.julialang.org/en/v1/manual/functions/

[57]https://docs.julialang.org/en/v1/

```
# pseudocode, don't run this snippet
if (condition_that_returns_Bool)
    what_to_do
elseif (another_condition_that_returns_Bool)
    what_to_do
elseif (another_condition_that_returns_Bool)
    what_to_do
else
    what_to_do
end
```

As mentioned in Section 3.3.4 `Bool` type can take one of two values `true` or `false`. The code inside `if`/`elseif` clause runs only when the condition is `true`. You can have any number of `elseif` clauses. Only the code for the first `true` clause runs. If none of the previous conditions matches (each and every one is `false`) the code in the `else` block is executed. Only `if` and `end` keywords are obligatory, the rest is not, so you may use

```
# pseudocode, don't run this snippet
if (condition_that_returns_Bool)
    what_to_do
end
```

or

```
# pseudocode, don't run this snippet
if (condition_that_returns_Bool)
    what_to_do
else
    what_to_do
end
```

or

```
# pseudocode, don't run this snippet
if (condition_that_returns_Bool)
    what_to_do
elseif (condition_that_returns_Bool)
    what_to_do
else
    what_to_do
end
```

or

```
# pseudocode, don't run this snippet
if (condition_that_returns_Bool)
    what_to_do
elseif (condition_that_returns_Bool)
    what_to_do
elseif (condition_that_returns_Bool)
    what_to_do
else
    what_to_do
end
```

or ..., never mind, I think you got the point.

Below I place another example of a function using `if/elseif/else` construct (in order to remember it better).

```
# works fine for non-empty vectors
function getMin(vect::Vector{Int}, isSortedAsc::Bool)::Int
    if isSortedAsc
        return vect[1]
    else
        sortedVect::Vector{Int} = sort(vect)
        return sortedVect[1]
    end
end

x = [1, 2, 3, 4]
y = [3, 4, 1, 2]

(getMin(x, true), getMin(y, false))
```

```
(1, 1)
```

Here I wrote a function that finds the minimal value in a vector of integers. If the vector is sorted in the ascending order it returns the first element. If it is not, it sorts the vector using the built in sort[58] function and returns its first element (*this may not be the most efficient method but it works*). Note that the `else` block contains two lines of code (it could contain more if necessary, and so could `if` block). I did this for demonstrative purposes. Alternatively instead those two lines

[58] https://docs.julialang.org/en/v1/base/sort/#Base.sort

(in the `else` block) one could write `return sort(vect)[1]` and it would work just fine.

Ternary expression

If you need only a single `if ... else` in your code, then you may prefer to replace it with ternary operator. Its general form is `condition_or_Bool ? result_if_true : result_if_false`.

Let me rewrite `getMin` from Section 3.5.1 using ternary expression.

```
function getMin(vect::Vector{Int}, isSortedAsc::Bool)::Int
    return isSortedAsc ? vect[1] : sort(vect)[1]
end

x = [1, 2, 3, 4]
y = [3, 4, 1, 2]

(getMin(x, true), getMin(y, false))
```

```
(1, 1)
```

Much less code, works the same. Still, I would not overuse it. For more than a single condition it is usually harder to write, read, and process in your head than the good old `if/elseif/else` block.

Dictionaries

Dictionaries in Julia[59]are a sort of mapping. Just like an ordinary dictionary is a mapping between a word and its definition. Here, we say that the mapping is between `key` and `value`. For instance let's say I want to define an English-Polish dictionary.

```
engPolDict::Dict{String, String} = Dict("one" => "jeden", "two" =>
"dwa")
engPolDict # the key order is not preserved on different computers
```

```
Dict{String, String} with 2 entries:
  "two" => "dwa"
  "one" => "jeden"
```

[59]https://docs.julialang.org/en/v1/base/collections/#Dictionaries

Here I defined a dictionary of type `Dict{String, String}`, so, both `key` and `value` are of textual type (`String`). The order of the keys is not preserved (this data structure cares more about lookup performance and not about the order of the keys). Therefore, you may see a different order of items after executing the code on your computer.

If we want to now how to say "two" in Polish I type `aDict[key]` (if the key is not there you will get an error), e.g.

```
engPolDict["two"]
```

dwa

To add a new value to a dictionary (or to update the existing value) write `aDict[key] = newVal`. Right now the key "three" does not exist in `engPolDict` so I would get an error (check it out), but if I type:

```
engPolDict["three"] = "trzy"
```

trzy

Then I create (or update if it was already there) a key-value mapping.

Now, to avoid getting errors due to non-existing keys I can use the built in get[60]function. You use it in the form `get(collection, key, default)`, e.g. right now the word "four" (key) is not in a dictionary so I should get an error (check it out). But wait, there is `get`.

```
get(engPolDict, "four", "not found")
```

not found

OK, what anything of it got to do with `if/elseif/else` and decision making. The thing is that if you got a lot of decisions to make then probably you will be better off with a dictionary. Compare

```

```

[60]https://docs.julialang.org/en/v1/base/collections/#Base.get

```
function translEng2polVer1(engWord::String)::String
    if engWord == "one"
        return "jeden"
    elseif engWord == "two"
        return "dwa"
    elseif engWord == "three"
        return "trzy"
    elseif engWord == "four"
        return "cztery"
    else
        return "not found"
    end
end

(translEng2polVer1("three"), translEng2polVer1("ten"))
```

```
("trzy", "not found")
```

with

```
function translEng2polVer2(engWord::String,
                           aDict::Dict{String, String} =
engPolDict)::String
    return get(aDict, engWord, "not found")
end

(translEng2polVer2("three"), translEng2polVer2("twelve"))
```

```
("trzy", "not found")
```

> ***Note:*** Dictionaries like Arrays (see Section 3.3.7) are passed by references

In `translEng2polVer2` I used a so called optional argument[61]for `aDict` (`aDict::Dict{String, String} = engPolDict`). This means that if the function is provided without the second argument then `engPolDict` will be used as its second argument. If I defined the function as `translEng2polVer2(engWord::String, aDict::Dict{String, String})` then while running the function I

[61]https://docs.julialang.org/en/v1/manual/functions/#Optional-Arguments

would have to write `(translEng2polVer2("three", engPolDict), translEng2polVer2("twelve", engPolDict))`. Of course, I may prefer to use some other English-Polish dictionary (perhaps the one found on the internet) like so `translEng2polVer2("three", betterEngPolDict)` instead of using the default `engPolDict` we got here.

In general, the more `if ... elseif ... else` comparisons you got to do the better off you are when you use dictionaries (especially that they could be written by someone else, you just use them). Still, in the rest of the book we will probably use dictionaries for data storage and a quick lookup.

OK, enough of that. If you want to know more about conditional evaluation check this part of Julia's docs[62].

Repetition

Julia, and computers in general, are good at doing boring, repetitive tasks for us without a word of complaint (and they do it much faster than we do). Let's see some constructs that help us with it.

For loops

A for loop[63]is a standard construct present in many programming languages that does the repetition for us. Its general form in Julia is:

```
# pseudocode, do not run this snippet
for i in sequence
    # do_something_useful
end
```

The loop is enclosed between `for` and `end` keywords and repeats some specific action(s) (`# do_something_useful`) for every element of a `sequence`. On each turnover of a loop consecutive elements of a sequence are referred to by `i`.

[62]https://docs.julialang.org/en/v1/manual/control-flow/#man-conditional-evaluation

[63]https://en.wikipedia.org/wiki/For_loop

> ***Note:*** I could have assigned any name, like: `j`, `k`, `whatever`, it would work the same. Still, `i` and `j` are quite common in for loops[64].

Let's say I want a program that will print hip hip hooray[65]many times for my friend that celebrates some success. I can proceed like this.

```
function printHoorayNtimes(n::Int)
    @assert (n > 0) "n needs to be greater than 0"
    for _ in 1:n
        println("hip hip hooray!")
    end
    return nothing
end
```

Go ahead, run it (e.g. `printHoorayNtimes(3)`).

Notice two new elements. Here it makes no sense for `n` to be less than or equal to 0. Hence, I used @assert[66]construct to test it and print an error message (`"n needs to be greater than 0"`) if it is. The construct is not recommended in serious programs, but for our quick and dirty approach it should do the trick. The `1:n` is a range similar to the one we used in Section 3.3.6 . Here, I used `_` instead of `i` in the example above (to signal that I don't plan to use it further).

OK, how about another example. You remember `myMathGrades`, right?

```
myMathGrades = [3.5, 3.0, 3.5, 2.0, 4.0, 5.0, 3.0]
```

Now, since the end of the school year is coming then I would like to know my average[67](likely this will be my final grade). In order to get that I need to divide the sum by the number of grades. First the sum.

```
function getSum(nums::Vector{<:Real})::Real
    total::Real = 0
```

[64]https://en.wikipedia.org/wiki/For_loop
[65]https://en.wikipedia.org/wiki/Hip_hip_hooray
[66]https://docs.julialang.org/en/v1/base/base/#Base.@assert
[67]https://en.wikipedia.org/wiki/Arithmetic_mean

```
    for i in 1:length(nums)
        total = total + nums[i]
    end
    return total
end

getSum(myMathGrades)
```

24.0

A few explanations regarding the new bits of code here.

In the arguments list I wrote `::Vector{<:Real}`. Which means that each element of nums is a subtype (`<:`) of the type `Real` (which includes integers and floats). I declared a `total` and initialized it to 0. Then in `for` loop I used `i` to hold numbers from 1 to number of elements in the vector (`length(nums)`). Finally, in the for loop body I added each number from the vector (using indexing see Section 3.3.6) to the `total`. The `total = total + nums[i]` means that new total is equal to old total + element of the vector (`nums`) with index `i` (`nums[i]`). Finally, I returned the total.

The body of the `for` loop could be improved. Instead of `for i in 1:length(nums)` I could have written `for i in eachindex(nums)` (notice there is no `1:`, `eachindex` is a built in Julia function, see here[68]). Moreover, instead of `total = total + nums[i]` I could have used `total += nums[i]`. The `+=` is and update operator[69], i.e. a shortcut for updating old value by adding a new value to it. Take a moment to rewrite the function with those new forms and test it.

> ***Note:*** The update operator must be written as `accumulator += updateValue` (e.g. `total += 2`) and not `accumulator =+ updateValue` (e.g. `total =+ 2`). In the latter case Julia will asign `updateValue` (`+2`) as a new value of `accumulator` [it will interpret

[68] https://docs.julialang.org/en/v1/base/arrays/#Base.eachindex

[69] https://docs.julialang.org/en/v1/manual/mathematical-operations/#Updating-operators

`=+ 2` as assign (=) plus/positive two (+2) instead of update (+=) by 2].

Alternatively, I can do this without indexing (although `for` loops with indexing are a classical idiom in programming and it is worth to know them).

```
function getSum(nums::Vector{<:Real})::Real
    total::Real = 0
    for num in nums
        total += num
    end
    return total
end

getSum(myMathGrades)
```

24.0

Here `num` (I could have used `n`, `i` or `whatever` if I wanted to) takes the value of each consecutive element of `nums` and adds it to the total.

OK, and now back to the average[70].

```
function getAvg(nums::Vector{<:Real})::Real
    return getSum(nums) / length(nums)
end

getAvg(myMathGrades)
```

3.4285714285714284

Ups, not quite 3.5, I'll better present some additional projects to improve my final grade.

OK, two more examples that might be useful and will help you master `for` loops even better.

Let's say I got a vector of temperatures in Celsius[71]and want to send it to a friend in the US.

[70]https://en.wikipedia.org/wiki/Arithmetic_mean
[71]https://en.wikipedia.org/wiki/Celsius

```
temperaturesCelsius = [22, 18.3, 20.1, 19.5]
```

```
[22.0, 18.3, 20.1, 19.5]
```

To make it easier for him I should probably change it to Fahrenheit[72] using this formula[73]. I start with writing a simple converting function for a single value of the temperature in Celsius scale.

```
function degCels2degFahr(tempCels::Real)::Real
    return tempCels * 1.8 + 32
end

degCels2degFahr(0)
```

32.0

Now let's convert the temperatures in the vector. First I would try something like this:

```
function degCels2degFahr!(tempsCels::Vector{<:Real})
    for i in eachindex(tempsCels)
        tempsCels[i] = degCels2degFahr(tempsCels[i])
    end
    return nothing
end
```

Notice the `!` in the function name (don't remember what it mean? see here[74]).

Still, this is not good. If I use it (`degCels2degFahr!(temperatureCelsius)`) it will change the values in `temperaturesCelsius` to Fahrenheit which could cause problems (variable name doesn't reflect its contents). A better approach is to write a function that produces a new vector and doesn't change the old one.

[72] https://en.wikipedia.org/wiki/Fahrenheit

[73] https://en.wikipedia.org/wiki/Fahrenheit#Conversion_(specific_temperature_point)

[74] https://docs.julialang.org/en/v1/manual/style-guide/#bang-convention

```
function degCels2degFahr(tempsCels::Vector{<:Real})::Vector{<:Real}
    result::Vector{<:Real} = zeros(length(tempsCels))
    for i in eachindex(tempsCels)
        result[i] = degCels2degFahr(tempsCels[i])
    end
    return result
end
```

```
degCels2degFahr (generic function with 2 methods)
```

Now I can use it like that:

```
temperaturesFahrenheit = degCels2degFahr(temperaturesCelsius)
```

```
[71.6, 64.94, 68.18, 67.1]
```

First of all, notice that so far I defined two functions named `degCels2degFahr`. One of them has got a single value as an argument (`degCels2degFahr(tempCels::Real)`) and another a vector as its argument (`degCels2degFahr(tempsCels::Vector{<:Real})`). But since I explicitly declared argument types, Julia will know when to use each version based on the function's arguments (see next paragraph). The different function versions are called methods (hence the message: `degCels2degFahr (generic function with 2 methods)` under the code snippet above).

In the body of `degCels2degFahr(tempsCels::Vector{<:Real})` first I declare and initialize a variable that will hold the result (hence `result`). I do this using built in zeros[75]function. The function returns a new vector with n elements (where n is equal to `length(tempsCels)`) filled with, you got it, 0s. The 0s are just placeholders. Then, in the `for` loop, I go through all the indices of `result` (`i` holds the current index) and replace each zero (`result[i]`) with a corresponding value in Fahrenheit (`degCels2degFahr(tempsCels[i])`). Here, since I pass a single value (`tempsCels[i]`) Julia knows which version (aka method)

[75]https://docs.julialang.org/en/v1/base/arrays/#Base.zeros

of the function `degCels2degFahr` to use (i.e. this one `degCels2degFahr(tempCels::Real)`).

For loops can be nested[76](even a few times). This is useful, e.g. when iterating over every call in an array (we met arrays in Section 3.3.7). We will use nested loops later in the book (e.g. in Section 6.8.2).

OK, enough for the classic `for` loops. Let's go to some built in goodies that could help us out with repetition.

Built-in Goodies

If the operation you want to perform is simple enough you may prefer to use some of the Julia's goodies mentioned below.

Comprehensions

Another useful constructs are comprehensions[77].

Let's say this time I want to convert inches to centimeters using this function.

```
function inch2cm(inch::Real)::Real
    return inch * 2.54
end

inch2cm(1)
```

2.54

If I want to do it for a bunch of values I can use comprehensions like so.

```
inches = [10, 20, 30]

function inches2cms(inches::Vector{<:Real})::Vector{<:Real}
    return [inch2cm(inch) for inch in inches]
end

inches2cms(inches)
```

[76]https://en.wikibooks.org/wiki/Introducing_Julia/Controlling_the_flow#Nested_loops

[77]https://docs.julialang.org/en/v1/manual/arrays/#man-comprehensions

```
[25.4, 50.8, 76.2]
```

On the right I use the familiar `for` loop syntax, i.e. `for sth in collection`. On the left I place a function (named or anonymous[78]) that I want to use (here `inch2cm`) and pass consecutive elements (`sth`, here `inch`) to that function. The expression is surrounded with square brackets so that Julia makes a new vector out of it (the old vector is not changed).

In general comprehensions are pretty useful, chances are that I'm going to use them a lot in this book so make sure to learn them (e.g. read their description in the link at the beginning of this subchapter, i.e. Section 3.6.3 or look at the examples shown here[79]).

Map and Foreach

Comprehensions are nice, but some people find map[80]even better. The example above could be rewritten as:

```
inches = [10, 20, 30]

function inches2cms(inches::Vector{<:Real})::Vector{<:Real}
    return map(inch2cm, inches)
end

inches2cms(inches)
```

```
[25.4, 50.8, 76.2]
```

Again, I pass a function (note I typed only its name) as a first argument to `map`, the second argument is a collection. Map automatically applies the function to every element of the collection and returns a new collection. Isn't this magic.

[78]https://docs.julialang.org/en/v1/manual/functions/#man-anonymous-functions

[79]https://en.wikibooks.org/wiki/Introducing_Julia/Controlling_the_flow#Comprehensions

[80]https://docs.julialang.org/en/v1/base/collections/#Base.map

If you want to evoke a function on a vector just for side effects (since you don't need to build a vector and return it) use foreach[81]. For instance, `getSum` with `foreach` and an anonymous function would look like this

```
function getSum(vect::Vector{<:Real})::Real
    total::Real = 0
    foreach(x -> total += x, vect) # side effect is to increase total
    return total
end

getSum([1, 2, 3, 4])
```

10

Here, `foreach` will perform an action (its first argument) on each element of its second argument (`vect`). The first argument (`x -> total += x`) is an anonymous function[82]that takes some value `x` and in its body (`->` points at the body) adds `x` to `total` (`total += x`). The `x` takes each value of `vect` (second argument).

> ***Note:*** Anonymous functions will be used quite a bit in this book, so make sure you understand them (read their description in the link above or look at the examples shown here[83]).

Dot operators/functions

Last but not least. I can use a dot operator[84]. Say I got a vector of numbers and I want to add 10 to each of them. Doing this for a single number is simple, I would have just typed `1 + 10`. Hmm, but for a vector? Simple as well. I just need to precede the operator with a `.` like so:

[81]https://docs.julialang.org/en/v1/base/collections/#Base.foreach

[82]https://docs.julialang.org/en/v1/manual/functions/#man-anonymous-functions

[83]https://en.wikibooks.org/wiki/Introducing_Julia/Functions#Anonymous_functions

[84]https://docs.julialang.org/en/v1/manual/mathematical-operations/#man-dot-operators

```
[1, 2, 3] .+ 10
```

```
[11, 12, 13]
```

I can do this also for functions (both built-in and written by myself). Notice `.` goes before `(`

```
inches = [10, 20, 30]

function inches2cms(inches::Vector{<:Real})::Vector{<:Real}
    return inch2cm.(inches)
end

inches2cms(inches)
```

```
[25.4, 50.8, 76.2]
```

Isn't this nice.

OK, the goodies are great, but require some time to get used to them (I suspect at first you're gonna use good old `for` loop syntax). Besides the constructs described in this section are good for simple operations (don't try to put too much stuff into them, they are supposed to be one liners).

In any case choose a construct that you know how to use and that gets the job done for you, mastering them all will take some time.

Still, in general dot operations are pretty useful, chances are that I'm going to use them a lot in this book so make sure to understand them.

Additional libraries

OK, there is one more thing I want to briefly talk about, and it is libraries[85](sometimes called packages).

A library is a piece of code developed by someone else. At the time I'm writing these words there are over 10'000 libraries (aka packages) in

[85]https://en.wikipedia.org/wiki/Library_(computing)

Julia (see here[86]) available under different licenses. If the package is under MIT license[87](a lot of them are) then basically you may use it freely, but without any warranty.

To install a package you use Pkg[88], i.e. Julia's built in package manager. Click the link in the previous sentence to see how to do it (be aware that installation may take some time).

In general there are two ways to use a package in your project:

1. by typing `using Some_pkg_name`
2. by typing `import Some_pkg_name`

Personally, I prefer the latter. Actually, I use it in the form `import Some_pkg_name as Abbreviated_pkg_name` (you will see why in a moment).

Let's see how it works. Remember the `getSum` and `getAvg` functions that we wrote ourselves. Well, it turns out Julia got a built-in sum[89]and Statistics[90]package got a mean[91]function. To use it I type at the top of my file (it is a good practice to do so):

```
import Statistics as Stats
```

Now I can access any of its functions by preceding them with `Stats` (my abbreviation) and `.` like so

```
Stats.mean([1, 2, 3])
```

2.0

And that's it. It just works.

[86]https://julialang.org/packages/
[87]https://en.wikipedia.org/wiki/MIT_License
[88]https://docs.julialang.org/en/v1/stdlib/Pkg/
[89]https://docs.julialang.org/en/v1/base/collections/#Base.sum
[90]https://docs.julialang.org/en/v1/stdlib/Statistics/
[91]https://docs.julialang.org/en/v1/stdlib/Statistics/#Statistics.mean

Note that if you type `import Statistics` instead of `import Statistics as Stats` then in order to use `mean` you will have to type `Statistics.mean([1, 2, 3])`. So in general it is a good idea to give some shorter name for an imported package.

Oh yeah, one more thing. In order to know what are the functions in a library and how to use them you should check the library's documentation.

OK, enough theory, time for some practice.

Julia - Exercises

I once heard that in chess you can get only as much as you give. I believe it is also true for programming (and most likely many other human activities).

So, here are some exercises that you may want to solve to get from this chapter as much as you can.

> ***Note:*** Some readers probably will not solve the exercises. They will not want to (because of the waste of time) or will not be able to solve them (in that case my apology for the inappropriate difficulty level). Either way, I suggest you read the tasks' descriptions and the solutions (and try to understand them). In those sections I may use, e.g. some language constructs that I will not explain again in the upcoming chapters.

Exercise 1

Imagine the following situation. You and your friends make a call to order out a pizza. You got only $50 and you are pretty hungry. But you got a dilemma, for exactly $50 you can either order 2 pizzas 30 cm in diameter each, or 1 pizza 45 cm in diameter. Which one is more worth it?

Hint: Assume that the pizza is flat and that you are eating its surface.

Hint: You may want to search the documentation[92] for `Base.MathConstants` and use one of them.

Exercise 2

When we talked about float comparisons (Section 3.3.3) we said to be careful since

```
(0.1 * 3) == 0.3
```

false

Write a function with the following signature `areApproxEqual(f1::Float64, f2::Float64)::Bool`. It should return `true` when called with those numbers (`areApproxEqual(0.1*3, 0.3)`). For the task you may use round[93] with a precision of, let's say, 16 digits.

> ***Note:*** Probably there is no point of greater precision than 16 digits since your machine won't be able to see it anyway. For technical details see Base.eps[94].

Exercise 3

Remember `getMin` from previous chapter (see Section 3.5.2)

```
function getMin(vect::Vector{Int}, isSortedAsc::Bool)::Int
    return isSortedAsc ? vect[1] : sort(vect)[1]
end
```

Write `getMax` with the following signature `getMax(vect::Vector{Int}, isSortedDesc::Bool)::Int` use only the elements from previous version of the function (you should modify them).

---

[92] https://docs.julialang.org/en/v1/

[93] https://docs.julialang.org/en/v1/base/math/#Base.round-Tuple%7BComplex%7B%3C:AbstractFloat%7D,%20RoundingMode,%20RoundingMode%7D

[94] https://docs.julialang.org/en/v1/base/base/#Base.eps-Tuple%7BType%7B%3C:AbstractFloat%7D%7D

Exercise 4

Someone once told me that the simplest interview question for a candidate programmer is fizz buzz[95]. If a person doesn't know how to do that there is no point of examining them further.

I don't know if that's true, but here we go.

Write a program for a range of numbers 1 to 30.

- If a number is divisible by 3 print "Fizz" on the screen.
- If a number is divisible by 5 print "Buzz" on the screen.
- If a number is divisible by 3 and 5 print "Fizz Buzz" on the screen.
- Otherwise print the number itself.

If you feel stuck right now, don't worry. It sounds difficult, because so far you haven't met all the necessary elements to solve it. Still, I believe you can do this by reading the Julia's docs or using your favorite web search engine.

Here are some constructs that might be useful to solve this task:

- for loop (see Section 3.6.1)
- if/elseif/else (see Section 3.5.1)
- modulo operator or rem function[96]
- 'logical and' (see Section 3.3.4 and this[97] and that[98] section of Julia's docs)
- string function[99]

You may use some or all of them. Or perhaps you can come up with something else. Good luck.

Exercise 5

I once heard a story about chess.

[95] https://en.wikipedia.org/wiki/Fizz_buzz

[96] https://docs.julialang.org/en/v1/base/math/#Base.rem

[97] https://docs.julialang.org/en/v1/manual/missing/#Logical-operators

[98] https://docs.julialang.org/en/v1/manual/missing/#Control-Flow-and-Short-Circuiting-Operators

[99] https://docs.julialang.org/en/v1/base/strings/#Base.string

According to the story the game was created by a Hindu wise man. He presented the invention to his king who was so impressed that he offered to fulfill his request as a reward.

- I want nothing but some wheat grains.
- How many?
- Put 1 grain on the first chess field, 2 grains on the second, 4 on the third, 8 on the fourth, and so on. I want the grains that are on the last field.

A laughingly small request, thought the king. Or is it?

Use Julia to answer how many grains are on the last (64th) field.

Hint. If you get a strange looking result, use BigInt[100]data type instead of Int[101].

Exercise 6

Lastly, to cool down a little write a function `getInit` that takes a vector of any type as an argument and returns the vector without its last element.

You may either use the generics (preferred way to solve it, see Section 3.4.2) or write the function without type declarations (acceptable solution).

Remember about the indexing (see Section 3.3.6). Think (or search for the answer e.g. in the internet) how to get one but last element of an array.

Usage examples:

```
getInit([1, 2, 3, 4])
# output: [1, 2, 3]
```

```
getInit(["ab", "cd", "ef", "gh"])
# output: ["ab", "cd", "ef"]
```

[100]https://docs.julialang.org/en/v1/base/numbers/#BigFloats-and-BigInts

[101]https://docs.julialang.org/en/v1/manual/integers-and-floating-point-numbers/#Integers

```
getInit([3.3])
# output: Float64[]
```

```
getInit([])
# output: Any[]
```

Julia - Solutions

In this sub-chapter you will find exemplary solutions to the exercises from the previous section.

Solution to Exercise 1

Since I'm eating a surface, and the task description gives me diameters, then I should probably calculate area of a circle[102]. I will use Base.MathConstants.pi[103]in my calculations.

```
function getCircleArea(radius::Real)::Real
    return pi * radius * radius
end
```

Now, we can finally get the answer.

```
# radius = diameter / 2
(getCircleArea(30/2) * 2, getCircleArea(45/2))
```

```
(1413.7166941154069, 1590.431280879833)
```

It seems that I will get more food while ordering this one pizza (45 cm in diameter) and not those two pizzas (each 30 cm in diameter).

> ***Note:*** Instead of `pi * radius * radius` I could have used `radius^2`, where `^` is an exponentiation operator in Julia. If I want to raise 2 to the fourth power I can either type `2^4` or `2*2*2*2` and get 16.

[102]https://en.wikipedia.org/wiki/Area_of_a_circle
[103]https://docs.julialang.org/en/v1/base/numbers/#Base.MathConstants.pi

If all the pizzas were cylinders[104] of equal heights (say 2 cm or an inch each) then I would calculate their volumes like so

```
function getCylinderVolume(radius::Real, height::Real=2)::Real
    # hmm, is cylinder just many circles stacked one on another?
    return getCircleArea(radius) * height
end
```

and the results

```
# radius = diameter / 2
(getCylinderVolume(30/2) * 2, getCylinderVolume(45/2))
```

```
(2827.4333882308138, 3180.862561759666)
```

Still, it appears the conclusion is the same.

Solution to Exercise 2

My solution to that problem would look something like

```
function areApproxEqual(f1::Float64, f2::Float64)::Bool
    return round(f1, digits=16) == round(f2, digits=16)
end
```

Let's put it to the test

```
areApproxEqual(0.1*3, 0.3)
```

true

Seems to be working fine. Still, you may prefer to use Julia's built-in isapprox[105]. In general, it is a good idea to use a built in function from the standard library over your own as it should be more robust[106].

Anyway, let's test `isapprox` as well.

[104] https://en.wikipedia.org/wiki/Cylinder

[105] https://docs.julialang.org/en/v1/base/math/#Base.isapprox

[106] https://en.wikipedia.org/wiki/Robustness_(computer_science)

```
isapprox(0.1*3, 0.3)
# compare with
# isapprox(0.11*3, 0.3)
# or to test if the values are not equal
# !isapprox(0.11*3, 0.3)
```

true

It works just fine.

Lesson to be learned here. If you want to do something you can:

1. look for a function in the language documentation
2. look for a function in some library
3. write a function yourself by using what you already got at your disposal

Solution to Exercise 3

Possible solution

```
function getMax(vect::Vector{Int}, isSortedDesc::Bool)::Int
    return isSortedDesc ? vect[1] : sort(vect)[end]
end

(getMax([3, 2, 1], true), getMax([2, 3, 1], false))
```

```
(3, 3)
```

or if you read the documentation for sort[107]

```
function getMax(vect::Vector{Int}, isSortedDesc::Bool)::Int
    return isSortedDesc ? vect[1] : sort(vect, rev=true)[1]
end

(getMax([3, 2, 1], true), getMax([2, 3, 1], false))
```

```
(3, 3)
```

[107] https://docs.julialang.org/en/v1/base/sort/#Base.sort

Sorting an array to get the maximum (or minimum) value is not the most effective method (sorting is based on rearranging elements and takes quite some time). Traveling through an array only once should be faster. Therefore probably a better solution (in terms of performance) would be something like

```
function getMaxUnsorted(unsortedVect::Vector{Int})::Int
    maxVal::Int = unsortedVect[1]
    for elt in unsortedVect[2:end]
        if maxVal < elt
            maxVal = elt
        end
    end
    return maxVal
end

function getMax(vect::Vector{Int}, isSortedDesc::Bool)::Int
    return isSortedDesc ? vect[1] : getMaxUnsorted(vect)
end

(getMax([3, 2, 1], true), getMax([2, 3, 1], false))
```

```
(3, 3)
```

Read it carefully and try to figure out how it works.

> ***Note:*** Julia already got similar functionality to `getMin`, `getMax` that we developed ourselves. See min[108], max[109], minimum[110], and maximum[111].

Solution to Exercise 4

Perhaps the most direct version of the program would be

```
function printFizzBuzz()
    for i in 1:30
        # or: if rem(i, 15) == 0
```

[108] https://docs.julialang.org/en/v1/base/math/#Base.min
[109] https://docs.julialang.org/en/v1/base/math/#Base.max
[110] https://docs.julialang.org/en/v1/base/collections/#Base.minimum
[111] https://docs.julialang.org/en/v1/base/collections/#Base.maximum

```
        if rem(i, 3) == 0 && rem(i, 5) == 0
            println("Fizz Buzz")
        elseif rem(i, 3) == 0
            println("Fizz")
        elseif rem(i, 5) == 0
            println("Buzz")
        else
            println(i)
        end
    end
    return nothing
end
```

> ***Note:*** Julia applies operators based on precedence and associativity[112]. If you are unsure about the order of their evaluation (e.g. in `if rem(i, 3) == 0 && rem(i, 5) == 0`) then check the docs or use parenthesis `()` to enforce the desired order of evaluation (e.g. `if (rem(i, 3) == 0) && (rem(i, 5) == 0)`).

Go ahead, test it out.

If you like challenges try to follow the execution of the following program.

```
function getFizzBuzz(num::Int)::String
    return (
        rem(num, 15) == 0 ? "Fizz Buzz" :
        rem(num, 3) == 0 ? "Fizz" :
        rem(num, 5) == 0 ? "Buzz" :
        string(num)
    )
end

function printFizzBuzz()
    foreach(x -> println(getFizzBuzz(x)), 1:30)
    return nothing
end

# you can use it like so: printFizzBuzz()
```

[112] https://docs.julialang.org/en/v1/manual/mathematical-operations/#Operator-Precedence-and-Associativity

There are probably other more creative [or more (unnecessarily) convoluted] ways to solve this task. Personally, I would be satisfied if you understand the first version.

Solution to Exercise 5

For more information about the legend see this Wikipedia's article[113].

If you want some more detailed mathematical explanation you can read that Wikipedia's article[114].

The Wikipedia's version of the legend differs slightly from mine, but I like mine better.

Anyway let's jump right into some looping.

```
function getNumOfGrainsOnField64()::Int
    noOfGrains::Int = 1 # no of grains on field 1
    for _ in 2:64
        noOfGrains *= 2 # *= is update operator similar to +=
    end
    return noOfGrains
end

getNumOfGrainsOnField64()
```

-9223372036854775808

Hmm, that's odd, a negative number.

Wait a moment. Now I remember, a computer got finite amount of memory. So in order to work efficiently data is stored in small pre-allocated pieces of it. If the number you put into that small 'memory drawer' is greater than the amount of space then you get strange results (imagine that a number sticks out of the drawer but Julia looks only at the part inside the drawer, hence the strange result).

[113] https://en.wikipedia.org/wiki/Sissa_(mythical_brahmin)

[114] https://en.wikipedia.org/wiki/Wheat_and_chessboard_problem

If you are interested in technical stuff then you can read more about it in Julia's docs (sections Integers[115] and Overflow Behavior[116]).

You can check the minimum and maximum value for `Int` by typing `typemin(Int)` and `typemax(Int)` on my laptop those are -9223372036854775808 and 9223372036854775807, respectively.

The broad range of `Int` is enough for most calculations, still if you expect a really big number you should use BigInt[117] (`BigInt` calculations are slower than the ones for `Int`, but now you should be only limited by the amount of memory on your computer).

So let me correct the code.

```
function getNumOfGrainsOnField64()::BigInt
    noOfGrains::BigInt = 1 # no of grains on field 1
    for _ in 2:64
        noOfGrains *= 2
    end
    return noOfGrains
end

getNumOfGrainsOnField64()
```

9223372036854775808

Whoa, that number got like 19 digits. I don't even know how to name it. It cannot be that big, can it?

OK, quick verification with some mathematical calculation (don't remember ^? See Section 3.9.1).

```
BigInt(2)^63 # we multiply 2 by 2 by 2, etc. for fields 2:64
```

9223372036854775808

Yep, the numbers appear to be the same.

---

[115] https://docs.julialang.org/en/v1/manual/integers-and-floating-point-numbers/#Integers

[116] https://docs.julialang.org/en/v1/manual/integers-and-floating-point-numbers/#Overflow-behavior

[117] https://docs.julialang.org/en/v1/base/numbers/#BigFloats-and-BigInts

```
getNumOfGrainsOnField64() == BigInt(2)^63
```

true

So I guess the aforementioned Wikipedia's article[118]is right, it takes much more grain than a country (or the world) could produce in a year.

Solution to Exercise 6

A possible solution with generics looks something like that

```
function getInit(vect::Vector{T})::Vector{T} where T
    return vect[1:(end-1)]
end
```

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

The parenthesis around `end-1` are not necessary. I added them for better clarity of how the last by one index is calculated.

Tests:

```
getInit([1, 2, 3, 4])
```

```
[1, 2, 3]
```

```
getInit(["ab", "cd", "ef", "gh"])
```

```
["ab", "cd", "ef"]
```

```
getInit([3.3])
```

```
Float64[]
```

[118]https://en.wikipedia.org/wiki/Wheat_and_chessboard_problem

```
getInit([])
```

BTW. Try to remove type declarations and see if the function still works (if you do this right then it should).

OK, that's it for now. Let's move to another chapter.

Statistics - introduction

OK, once we got some Julia basics under our belts, it is time to get familiar with statistics.

First of all, what is statistics anyway?

Hmm, actually I have never tried to learn the definition by heart (after all getting such a question during an exam is slim to none). Still, if I were to give a short (2-3 sentences) definition without looking it up I would say something like that.

Statistics is a set of methods for drawing conclusions about big things (populations) based on small things (samples). A statistician observes only a small part of a bigger picture and makes generalization about what he does not see based on what he saw. Given that he saw only a part of the picture he can never be entirely sure of his conclusions.

OK, feel free to visit Wikipedia (see statistics[119]) and see how I did with my definition. The definition given there is probably more accurate and comprehensive than the one given above, but maybe mine will be easier to grasp for a beginner.

Anyway, my definition says “can never be entirely sure” so there needs to be some way to measure the (un)certainty. This is where probability comes into the picture. We will explore this concept in more than a few next pages.

Chapter imports

Later in this chapter we are going to use the following libraries

```
import CairoMakie as Cmk
import Distributions as Dsts
import Random as Rand
```

If you want to follow along you should have them installed on your system. A reminder of how to deal (install and such) with packages

[119] https://en.wikipedia.org/wiki/Statistics

can be found here[120]. But wait, you may prefer to use `Project.toml` and `Manifest.toml` files from the code snippets for this chapter[121]to install the required packages. The instructions you will find here[122].

The imports will be placed in the code snippet when first used, but I thought it is a good idea to put them here, after all imports should be at the top of your file (so here they are at the top of the chapter). Moreover, that way they will be easier to find all in one place.

If during the lecture of this chapter you find a piece of code of unknown functionality, just go to the code snippets mentioned above and run the code from the `*.jl` file. Once you have done that you can always extract a small piece of it and test it separately (modify and experiment with it if you wish).

Probability - definition

To me probability is one of the key concepts in statistics, after all any statistical software will gladly calculate the famous p-value (a form of probability) for you. Still, let's get back to our probability definition (see the sub-chapter name).

As said, at the conclusion of the previous section (Section 4), probability is a way to measure certainty. It's like with the grades in school. In Poland a pupil can score 1 to 6 (lowest to highest grade) and this tells us how well he mastered the subject. If I score 1 then I didn't master it at all, but when I get 6 this means that I got it all. We know from everyday life that probability takes values from 0 to 100%, e.g.

- Are you sure of it?
- Absolutely, one hundred percent.

or

- Do you think he can make it?

[120]https://docs.julialang.org/en/v1/stdlib/Pkg/
[121]https://github.com/b-lukaszuk/RJ_BS_eng/tree/main/code_snippets/ch04
[122]https://pkgdocs.julialang.org/v1/environments/

- I would say it's fifty-fifty.

or even

- What are the chances?
- Pretty much, zero.

When something is bound to happen we assign it the probability of 100%.

When it can go either way we say fifty-fifty (50% it will happen, 50% it will not happen).

When an event is impossible we say zero (probability of it happening is 0%).

And this is the way statisticians use it. OK, maybe not quite. A typical statistics textbook will say that the probability takes values from 0 to 1. It is expressed this way for a few particular reasons (some of the reasons may be given later). Moreover, believe it or not, but it is actually compatible with our understanding that is based on everyday life.

From primary school (see also Wikipedia's definition of percentage[123]) I remember that 1% is actually 1/100th of something which I can write down using proper fraction as $\frac{1}{100}$ or a decimal as 0.01.

Therefore any probability value from 0% to 100% can be written in these few forms. For instance:

- $0\% = \frac{0}{100} = 0.00 = 0$
- $1\% = \frac{1}{100} = 0.01$
- $5\% = \frac{5}{100} = 0.05$
- $10\% = \frac{10}{100} = 0.10 = 0.1$
- $20\% = \frac{20}{100} = 0.20 = 0.2$
- $50\% = \frac{50}{100} = 0.50 = 0.5$
- $100\% = \frac{100}{100} = 1.00 = 1$

[123] https://en.wikipedia.org/wiki/Percentage

To give you a better intuitive grasp of probability written as a decimal take a look at this simplistic graphical depiction of it

```
# prob = 0.0
impossible |||||||||||||||||||||||||||||||||||||||||||||||||| certain
           Δ
# prob = 0.2
impossible |||||||||||||||||||||||||||||||||||||||||||||||||| certain
                     Δ
# prob = 0.5
impossible |||||||||||||||||||||||||||||||||||||||||||||||||| certain
                                    Δ
# prob = 0.8
impossible |||||||||||||||||||||||||||||||||||||||||||||||||| certain
                                                        Δ
# prob = 1.0
impossible |||||||||||||||||||||||||||||||||||||||||||||||||| certain
                                                              Δ
```

Anyway, when written down as a decimal (like a statistician would do it) the probability is easier to type with a keyboard and a software calculator[124]. Additionally, now we will be able to perform some simple but useful calculations with those numbers (see the upcoming sections).

Probability - properties

One of the cool and practical stuff that I learned about probability is that it can be:

- added
- subtracted
- multiplied
- divided (not discussed in this section)

How about I illustrate that with a simple example.

From biology classes I remember that the genetic material (DNA[125]) of a cell is in its nucleus. It is organized in a set of chromosomes. Chromosomes come in pairs (twin or homologous chromosomes[126], we

[124]https://en.wikipedia.org/wiki/Software_calculator
[125]https://en.wikipedia.org/wiki/DNA
[126]https://en.wikipedia.org/wiki/Homologous_chromosome

get one from each of our parents). Each chromosome contains genes (like beads on a thread). Since we got a pair of chromosomes, then each chromosome from a pair contains a copy of the same gene(s). The copies are exactly the same or are different versions of a gene (we call them alleles[127]). In order to create gametes (like the egg cell and sperm cells) the parents' cells undergo division (meiosis[128]). During this process a cell splits in two and each of the child cells gets one chromosome from the pair.

For instance chromosome 9 contains the genes that determine our ABO blood group system[129]. A meiosis process for a person with blood group AB would look something like this (for simplicity I drew only twin chromosomes 9 and only genes for ABO blood group system).

[127] https://en.wikipedia.org/wiki/Allele

[128] https://en.wikipedia.org/wiki/Meiosis

[129] https://en.wikipedia.org/wiki/ABO_blood_group_system#Genetics

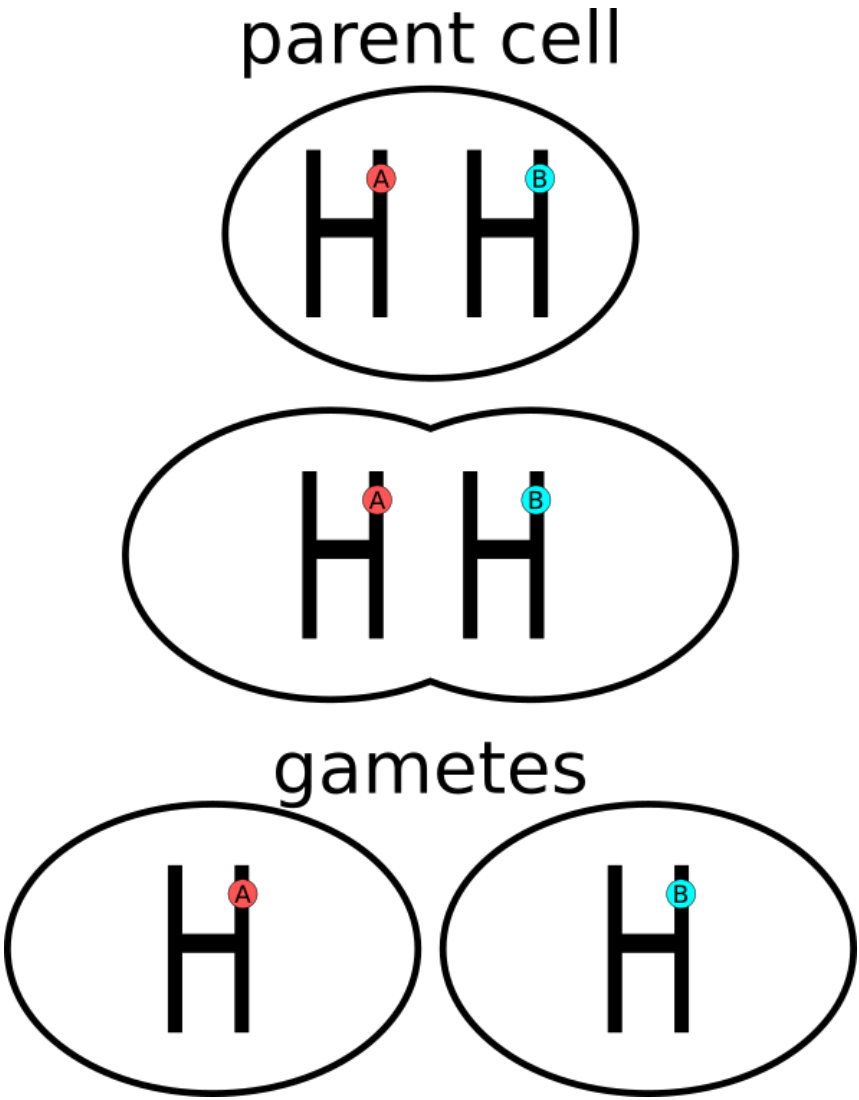

Figure 1: Figure 1: Meiosis. Splitting of a cell of a person with blood group AB.

OK, let's see how the mathematical properties of probability named at the beginning of this sub-chapter apply here.

But first, a warm-up (or a reminder if you will). In the previous part (see Section 4.2) we said that probability may be seen as a percentage,

decimal or fraction. I think that the last one will be particularly useful to broaden our understanding of the concept. To determine probability of an event in the numerator (top) we insert the number of times that a particular event may happen, in the denominator (bottom) we place the number of all possible events, like so:

$$\frac{num\ times\ this\ event\ may\ happen}{num\ times\ any\ event\ may\ happen}$$

Let's test this in practice with a few short Q&As (there may be some repetitions, but they are on purpose).

Q1. In the case illustrated in Figure 1 what is the probability of getting a gamete with allele `C` [for short I'll name it P(`C`)] from a person with blood group AB?

A1. Since we can only get allele `A` or `B`, but no `C` then $P(C) = \frac{0}{2} = 0$ (it is an impossible event).

Q2. In the case illustrated in Figure 1 what is the probability of getting a gamete with allele `A` [for short I'll name it P(`A`)] from a person with blood group AB?

A2. Since we can get only allele `A` or `B` then `A` is 1 of 2 possible events, so $\frac{1}{2} = 0.5$.

It seems that to answer this question we just had to divide the counts of the events satisfying our requirements by the counts of all events.

> ***Note:*** This is exactly the same probability (since it relies on the same reasoning) as for getting a gamete with allele `B` (1 of 2 or $\frac{1}{2} = 0.5$)

Q3. In the case illustrated in Figure 1 , what is the probability of getting a gamete with allele `A` or `B` [for short I'll name it P(`A` or `B`)] from a person with blood group AB?

A3. Since we can only get allele A or B then A or B are 2 events (1 event when A happens + 1 event when B happens) of 2 possible events, so

$P(A\ or\ B) = \frac{1+1}{2} = \frac{2}{2} = 1.$

It seems that to answer this question we just had to add the counts of the both events.

Let's look at it from a slightly different perspective.

Do you remember that in **A2** we stated that the probability of getting gamete A is $\frac{1}{2}$ and the probability of getting gamete B is $\frac{1}{2}$? And do you remember that in primary school we learned that fractions can be added one to another? Let's see will that do us any good here.

$P(A\ or\ B) = P(A) + P(B) = \frac{1}{2} + \frac{1}{2} = \frac{2}{2} = 1$

Interesting, the answer (and calculations) are (virtually) the same despite a slightly different reasoning. So it seems that in this case the probabilities can be added.

Q4. In the case illustrated in Figure 1 , what is the probability of getting a gamete with allele B (for short I'll name it P(B)) from a person with blood group AB?

A4. I know, we already answered it in **A2**. But let's do something wild and use a slightly different reasoning.

Getting gamete A or B are two incidents of two possible events (2 of 2). If we subtract event A (that we are not interested in) from both the events we get:

$P(B) = \frac{2-1}{2} = \frac{1}{2}$

It seems that to answer this question we just had to subtract the count of the events we are not interested in from the counts of the both events.

Let's see if this works with fractions (aka probabilities).

$P(B) = P(A\ or\ B) - P(A) = \frac{2}{2} - \frac{1}{2} = \frac{1}{2}$

Yep, a success indeed.

Q5. Look at Figure 2 .

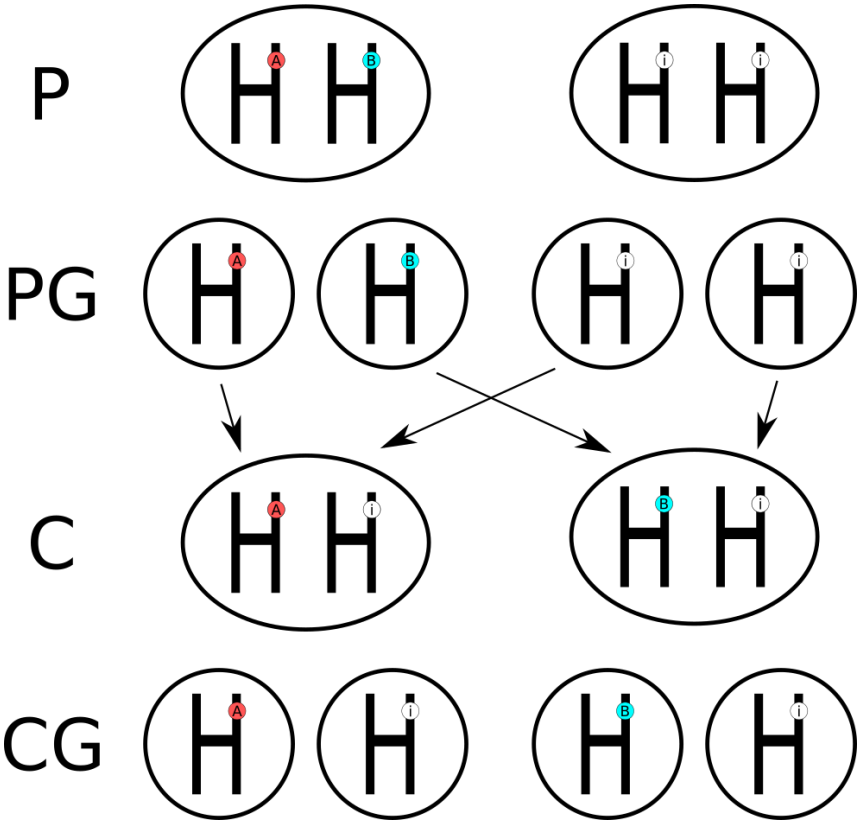

Figure 2: Figure 2: Blood groups, gametes. P - parents, PG - parents' gametes, C - children, CG - children's' gametes.

Here we see that a person with blood group AB got children with a person with blood group O (ii - recessive homo-zygote). The two possible blood groups in children are A (Ai - hetero-zygote) and B (Bi - hetero-zygote).

And now, the question. In the case illustrated in Figure 2 , what is the probability that a child (row C) of those parents (row P) will produce a gamete with allele `A` (row CG)?

A5. One way to answer this question would be to calculate the gametes in the last row (CG). We got 4 gametes in total (`A`, `i`, `B`, `i`) only one of which fulfills the criteria (gamete with allele `A`). Therefore, the probability is

$P(A\ in\ CG) = \frac{1}{4} = 0.25$ and that's it.

Another way to think about this problem is the following. In order for a child to produce a gamete with allele `A` it had to get it first from the parent. So what we are looking for is:

1. what proportion of children got allele `A` from their parents (here, half of them)
2. in the children with allele `A` in their genotype, what proportion of gametes contains allele `A` (here, half of the gametes)

So, in order to get the half of the half we have to multiply two proportions (aka fractions):

$P(A\ in\ CG) = P(A\ in\ C)*P(A\ in\ gametes\ of\ C\ with\ A)$

$P(A\ in\ CG) = \frac{1}{2}*\frac{1}{2} = \frac{1}{4} = 0.25$

So it turns out that probabilities can be multiplied (at least sometimes).

Probability properties - summary

The above was my interpretation of the probability properties explained with biological examples instead of the standard fair coins tosses (not the perfect analogy though, since the events are not quite independent). Let's sum up of what we learned. I'll do this on a coin toss examples (outcome: heads or tails), you compare it with the examples from Q&As above.

1. Probability of an event is a proportion (or fraction) of times this event happens to the total amount of possible distinctive events. Example: $P(heads) = \frac{heads}{heads+tails} = \frac{1}{2} = 0.5$
2. Probability of an impossible event is equal to 0. Probability of a certain event is equal to 1. So, the probability takes values between 0 (inclusive) and 1 (inclusive).

3. Probabilities of the mutually exclusive complementary events add up to
    1. Example:

    $P(heads\ or\ tails) = P(heads) + P(tails) = \frac{1}{2} + \frac{1}{2} = 1$

4. Probability of two mutually exclusive complementary events occurring at the same time is 0 (cannot get both heads and tails in a single coin toss).

5. Probability of two mutually exclusive complementary events occurring one after another is a product of two probabilities.

    Example: probability of getting two tails in two consecutive coin tosses $P(tails\ and\ tails) =$
    $P(tails\ in\ 1st\ toss)$*$P(tails\ in\ 2nd\ toss)$

    $P(tails\ and\ tails) = \frac{1}{2}*\frac{1}{2} = \frac{1}{4} = 0.25$

    Actually, the last is also true for two simultaneous coin tosses (imagine that one coin lands on a floor a few milliseconds before the other). Moreover, notice that here, the result of the first coin toss does not influence the result of the second coin toss (they are independent).

Anyway, the chances are that whenever you say P(this) AND P(that) you should use multiplication. Whereas whenever you say P(this) OR P(that) you ought to use addition. Of course you should always think does it make sense before you do it (if the events are not mutually exclusive and independent then it may not). To check your reasoning it may be easier to think about counts and their proportions. The latter can be translated to probabilities.

Probability - theory and practice

OK, in the previous chapter (see Section 4.3) we said that a person with blood group AB would produce gametes `A` and `B` with probability 50% (p = $\frac{1}{2}$ = 0.5) each. A reference value for sperm count[130]is 16'000'000 per mL or 16'000 per μL. Given that last value, we would

[130]https://en.wikipedia.org/wiki/Semen_analysis#Sperm_count

expect 8'000 cells (16'000 * 0.5) to contain allele `A` and 8'000 (16'000 * 0.5) cells to contain allele `B`.

Let's put that to the test.

Wait! Hold your horses! We're not going to take biological samples. Instead we will do a computer simulation.

```
import Random as Rand
Rand.seed!(321) # optional, needed for reproducibility
gametes = Rand.rand(["A", "B"], 16_000)
first(gametes, 7)
```

```
["B", "A", "B", "A", "B", "A", "A"]
```

First, we import a package to generate random numbers (`import Random as Rand`). Then we set seed to some arbitrary number (`Rand.seed!(321)`) in order to reproduce the results see the docs[131]. Thanks to the above you should get the exact same result as I did (assuming you're using the same version of Julia). Then we draw 16'000 gametes out of two available (`gametes = Rand.rand(["A", "B"], 16_000)`) with function `rand` (drawing with replacement) from `Random` library (imported as `Rand`). Finally, since looking through all 16'000 gametes is tedious we display only first 7 (`first(gametes, 7)`) to have a sneak peak at the result.

Let's write a function that will calculate the number of gametes for us.

```
function getCounts(v::Vector{T})::Dict{T,Int} where T
    counts::Dict{T,Int} = Dict()
    for elt in v
        if haskey(counts, elt) #1
            counts[elt] = counts[elt] + 1 #2
        else #3
            counts[elt] = 1 #4
        end #5
    end
    return counts
end
```

[131] https://docs.julialang.org/en/v1/stdlib/Random/#Random.seed!

Try to figure out what happened here on your own. If you need a refresher on dictionaries in Julia see Section 3.5.3 or the docs[132].

Briefly, first we initialize an empty dictionary (`counts::Dict{T,Int} = Dict()`) with keys of some type `T` (elements of that type compose the vector `v`). Next, for every element (`elt`) in the vector `v` we check if it is present in the `counts` (`if haskey(counts, elt)`). If it is we add 1 to the previous count (`counts[elt] = counts[elt] + 1`). If not (`else`) we put the key (`elt`) into the dictionary with count `1`. In the end we return the result (`return counts`). The `if ... else` block (lines with comments `#1`-`#5`) could be replaced with one line (`counts[elt] = get(counts, elt, 0) + 1`), but I thought the more verbose version would be easier to understand.

Let's test it out.

```
gametesCounts = getCounts(gametes)
gametesCounts
```

```
Dict{String, Int64} with 2 entries:
  "B" => 8082
  "A" => 7918
```

Hmm, that's odd. We were suppose to get 8'000 gametes with allele `A` and 8'000 with allele `B`. What happened? Well, reality. After all "All models are wrong, but some are useful"[133]. Our theoretical reasoning was only approximation of the real world and as such cannot be precise (although with greater sample sizes comes greater precision). For instance, you can imagine that a fraction of the gametes were damaged (e.g. due to some unspecified environmental factors) and underwent apoptosis (aka programmed cell death). So that's how it is, deal with it.

OK, let's see what are the experimental probabilities we got from our hmm… experiment.

[132] https://docs.julialang.org/en/v1/base/collections/#Base.Dict
[133] https://en.wikipedia.org/wiki/All_models_are_wrong

```
function getProbs(counts::Dict{T, Int})::Dict{T,Float64} where T
    total::Int = sum(values(counts))
    return Dict(k => v/total for (k, v) in counts)
end
```

First we calculate total counts no matter the gamete category (`sum(values(counts))`). Then we use a dictionary comprehension, similar to the comprehension we met before (see Section 3.6.3). Briefly, for each key and value in `counts` (`for (k,v) in counts`) we create the same key in a new dictionary with a new value being the proportion of `v` in `total` (`k => v/total`).

And now the experimental probabilities.

```
gametesProbs = getProbs(gametesCounts)
gametesProbs
```

```
Dict{String, Float64} with 2 entries:
  "B" => 0.505125
  "A" => 0.494875
```

One last point. While writing numerous programs I figured out it is sometimes better to represent things (internally) as numbers and only in the last step present them in a more pleasant visual form to the viewer (this way may be faster computationally). In our case we could have used `0` as allele `A` and `1` as allele `B` like so.

```
Rand.seed!(321)
gametes = Rand.rand([0, 1], 16_000)
first(gametes, 7)
```

```
[1, 0, 1, 0, 1, 0, 0]
```

Then to get the counts of the alleles I could type:

```
alleleBCount = sum(gametes)
alleleACount = length(gametes) - alleleBCount
(alleleACount, alleleBCount)
```

```
(7918, 8082)
```

And to get the probabilities for the alleles I could simply type:

```
alleleBProb = sum(gametes) / length(gametes)
alleleAProb = 1 - alleleBProb
(round(alleleAProb, digits=6), round(alleleBProb, digits=6))
```

```
(0.494875, 0.505125)
```

Go ahead. Compare the numbers with those that you got previously and explain it to yourself why this second approach works. Once you're done click the right arrow to explore probability distributions in the next section.

> ***Note:*** Similar functionality to `getCounts` and `getProbs` can be found in StatsBase.jl, see: countmap[134]and proportionmap[135].

Probability distribution

Another important concept worth knowing is that of probability distribution[136]. Let's explore it with some, hopefully interesting, examples.

First, imagine I offer Your a bet. You roll two six-sided dice. If the sum of the dots is 12 then I give you $125, otherwise you give me $5. Hmm, sounds like a good bet, doesn't it? Well, let's find out. By flexing our probabilistic muscles and using a computer simulation this should not be too hard to answer.

```
function getSumOf2DiceRoll()::Int
    return sum(Rand.rand(1:6, 2))
end

Rand.seed!(321)
```

[134]https://juliastats.org/StatsBase.jl/stable/counts/#StatsBase.countmap
[135]https://juliastats.org/StatsBase.jl/stable/counts/#StatsBase.proportionmap
[136]https://en.wikipedia.org/wiki/Probability_distribution

```
numOfRolls = 100_000
diceRolls = [getSumOf2DiceRoll() for _ in 1:numOfRolls]
diceCounts = getCounts(diceRolls)
diceProbs = getProbs(diceCounts)
```

Here, we rolled two 6-sided dice 100 thousand (10^5) times. The code introduces no new elements. The functions: `getCounts`, `getProbs`, `Rand.seed!` were already introduced in the previous chapter (see Section 4.4). And the `for _ in` construct we met while talking about for loops (see Section 3.6.1).

So, let's take a closer look at the result.

```
(diceCounts[12], diceProbs[12])
```

```
(2780, 0.0278)
```

It seems that out of 100'000 rolls with two six-sided dice only 2780 gave us two sixes (6 + 6 = 12), so the experimental probability is equal to 0.0278. But is it worth it? From a point of view of a single person (remember the bet is you vs. me) a person got probability of `diceProbs[12]` = 0.0278 to win $125 and a probability of `sum([get(diceProbs, i, 0) for i in 2:11])` = 0.9722 to lose $5. Since all the probabilities (for 2:12) add up to 1, the last part could be rewritten as `1 - diceProbs[12]` = 0.9722. Using Julia I can write this in the form of an equation like so:

```
function getOutcomeOfBet(probWin::Float64, moneyWin::Real,
                         probLose::Float64, moneyLose::Real)::Float64
    # in mathematics first we do multiplication (*), then subtraction
(-)
    return probWin * moneyWin - probLose * moneyLose
end

outcomeOf1bet = getOutcomeOfBet(diceProbs[12], 125, 1 - diceProbs[12],
5)

round(outcomeOf1bet, digits=2) # round to cents (1/100th of a dollar)
```

-1.39

In total you are expected to lose $ 1.39.

Now some people may say “Phi! What is $1.39 if I can potentially win $125 in a few tries”. It seems to me those are emotions (and perhaps greed) talking, but let’s test that too.

If 200 people make that bet (100 bet $5 on 12 and 100 bet $125 on the other result) we would expect the following outcome:

```
numOfBets = 100

outcomeOf100bets = (diceProbs[12] * numOfBets * 125) -
    ((1 - diceProbs[12]) * numOfBets * 5)
# or
outcomeOf100bets = ((diceProbs[12] * 125) - ((1 - diceProbs[12]) * 5)) *
100
# or simply
outcomeOf100bets = outcomeOf1bet * numOfBets

round(outcomeOf100bets, digits=2)
```

-138.6

OK. So, above we introduced a few similar ways to calculate that. The result of the bets is –138.6. In reality roughly 97 people that bet $5 on two sixes (6 + 6 = 12) lost their money and only 3 of them won $125 dollars which gives us $3*\$125 - 97*\$5 = -\$110$ (the numbers are not exact because based on the probabilities we got, e.g. 2.78 people and not 3).

Interestingly, this is the same as if you placed that same bet with me 100 times. Ninety-seven times you would have lost $5 and only 3 times you would have won $125 dollars. This would leave you over $110 poorer and me over $110 richer ($110 transfer from you to me where the money should be).

It seems that instead of betting on 12 (two sixes) many times you would be better off had you started a casino or a lottery. Then you should find let’s say 1’000 people daily that will take that bet (or buy $5 ticket) and get you $ 1386.0 (`outcomeOf1bet * 1000`) richer every day (well, probably less, because you would have to pay some taxes, still this makes a pretty penny).

OK, you saw right through me and you don't want to take that bet. Hmm, but what if I say a nice, big "I'm sorry" and offer you another bet. Again, you roll two six-sided dice. If you get 11 or 12 I give you $90 otherwise you give me $10. This time you know right away what to do:

```
pWin = sum([diceCounts[i] for i in 11:12]) / numOfRolls
# or
pWin = sum([diceProbs[i] for i in 11:12])

pLose = 1 - pWin

round(pWin * 90 - pLose * 10, digits=2)
# or
round(getOutcomeOfBet(pWin, 90, pLose, 10), digits=2)
```

-1.54

So, to estimate the probability we can either add number of occurrences of 11 and 12 and divide it by the total occurrences of all events OR, as we learned in the previous chapter (see Section 4.3), we can just add the probabilities of 11 and 12 to happen. Then we proceed with calculating the expected outcome of the bet and find out that I wanted to trick you again ("I'm sorry. Sorry.").

Now, using this method (that relies on probability distribution) you will be able to look through any bet that I will offer you and choose only those that serve you well. OK, so what is a probability distribution anyway? Well, it is just the value that probability takes for any possible outcome. We can represent it graphically by using any of Julia's plotting libraries[137].

Here, I'm going to use CairoMakie.jl[138]which seems to produce pleasing to the eye plots and is simple enough (that's what I think after I read its Basic Tutorial[139]). Nota bene also its error messages are quite informative (once you learn to read them).

[137]https://juliapackages.com/c/graphical-plotting
[138]https://docs.makie.org/stable/
[139]https://docs.makie.org/v0.21/tutorials/getting-started

```
import CairoMakie as Cmk

function getSortedKeysVals(d::Dict{A,B})::Tuple{
    Vector{A},Vector{B}} where {A,B}

    sortedKeys::Vector{A} = keys(d) |> collect |> sort
    sortedVals::Vector{B} = [d[k] for k in sortedKeys]
    return (sortedKeys, sortedVals)
end

xs1, ys1 = getSortedKeysVals(diceCounts)
xs2, ys2 = getSortedKeysVals(diceProbs)

fig = Cmk.Figure()
ax1 = Cmk.Axis(fig[1, 1:2],
        title="Rolling 2 dice 100'000 times",
        xlabel="Sum of dots",
        ylabel="Number of occurrences",
        xticks=2:12
)
Cmk.barplot!(ax1, xs1, ys1, color="red")
ax2 = Cmk.Axis(fig[2, 1:2],
        title="Rolling 2 dice 100'000 times",
        xlabel="Sum of dots",
        ylabel="Probability of occurrence",
        xticks=2:12
)
Cmk.barplot!(ax2, xs2, ys2, color="blue")
fig
```

> ***Note:*** Because of the compilation process running Julia's plots for the first time may be slow. If that is the case you may try some tricks recommended by package designers, e.g. this one from the creators of Gadfly.jl[140].

First, we extracted the sorted keys and values from our dictionaries (`diceCounts` and `diceProbs`) using `getSortedKeysVals`. The only new element here is `|>` operator. It's role is piping[141]the output of one function as input to another function. So `keys(d) |> collect |> sort` is just another way of writing `sort(collect(keys(d)))`. In both

[140]http://gadflyjl.org/stable/#Compilation

[141]https://docs.julialang.org/en/v1/manual/functions/#Function-composition-and-piping

cases first we run `keys(d)`, then we use the result of this function as an input to `collect` function, and finally pass its result to `sort` function. Out of the two options, the one with `|>` seems to be clearer to me.

Regarding the `getSortedKeysVals` it returns a tuple of sorted keys and values (that correspond with the sorted keys). In line `xs1, ys1 = getSortedKeysVals(diceCounts)` we unpack and assign them to `xs1` (it gets the sorted keys) and `ys1` (it gets values that correspond with the sorted keys). We do likewise for `diceProbs` in the line below.

In the next step we draw the distributions as bar plots (`Cmk.barplot!`). The code seems to be pretty self explanatory after you read the tutorial[142]that I just mentioned. A point of notice here (in case you wanted to know more): the `axis=`, `color=`, `xlabel=`, etc. are so called keyword arguments[143]. OK, let's get back to the graph. The number of counts (number of occurrences) on Y-axis is displayed using scientific notation, i.e. $1.0x10^4$ is 10'000 (one with 4 zeros) and $1.5x10^4$ is 15'000.

[142]https://docs.makie.org/v0.21/tutorials/getting-started

[143]https://docs.julialang.org/en/v1/manual/functions/#Keyword-Arguments

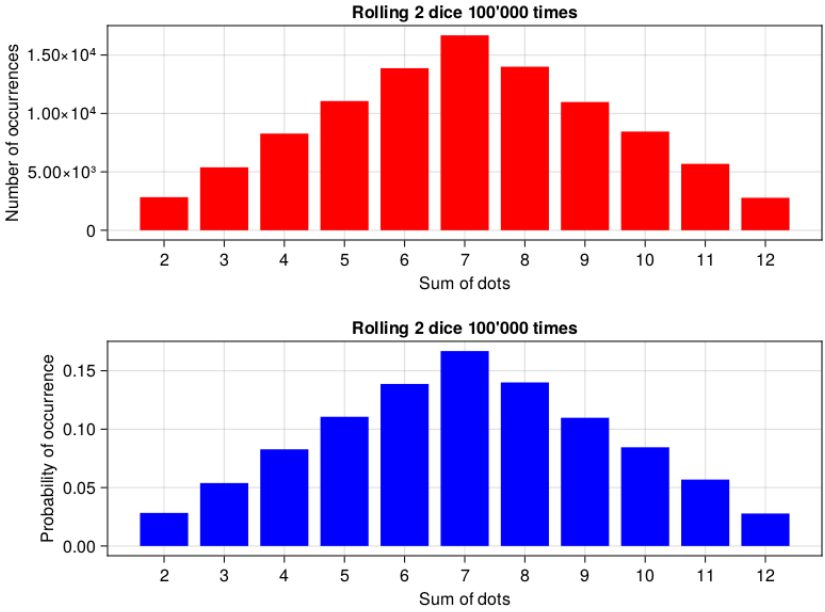

Figure 3: Figure 3: Rolling two 6-sided dice (counts and probabilities).

OK, but why did I even bother to talk about probability distributions (except for the great enlightenment it might have given to you)? Well, because it is important. It turns out that in statistics one relies on many probability distributions. For instance:

- We want to know if people in city A are taller than in city B. We take at random 10 people from each of the cities, we measure them and run a famous Student's T-test[144]to find out. It gives us the probability that helps us answer our question. It does so based on a t-distribution[145](see the upcoming Section 5.3).
- We want to know if cigarette smokers are more likely to believe in ghosts. What we do is we find random groups of smokers and non-smokers and ask them about it (Do you believe in ghosts?). We record the results and run a chi squared test[146]that gives us the probability that helps us answer our question. It does so based on a chi squared distribution[147](see the upcoming Section 6.3).

[144]https://en.wikipedia.org/wiki/Student%27s_t-test

OK, that should be enough for now. Take some rest, and when you're ready continue to the next chapter.

Normal distribution

Let's start where we left. We know that a probability distribution is a (possibly graphical) depiction of the values that probability takes for any possible outcome. Probabilities come in different forms and shapes. Additionally one probability distribution can transform into another (or at least into a distribution that resembles another distribution).

Let's look at a few examples.

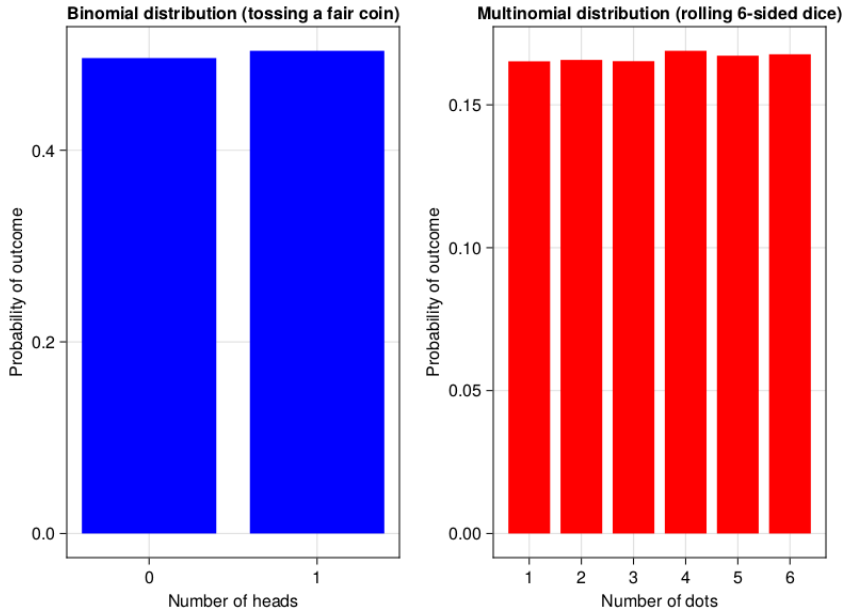

Figure 4: Figure 4: Experimental binomial and multinomial probability distributions.

[145] https://en.wikipedia.org/wiki/Student%27s_t-distribution
[146] https://en.wikipedia.org/wiki/Chi-squared_test
[147] https://en.wikipedia.org/wiki/Chi-squared_distribution

Here we got experimental distributions for tossing a standard fair coin and rolling a six-sided dice. The code for Figure 4 can be found in the code snippets for this chapter[148] and it uses the same functions that we developed previously.

Those are examples of the binomial (`bi` - two, `nomen` - name, those two names could be: heads/tails, A/B, or most general success/failure) and multinomial (`multi` - many, `nomen` - name, here the names are `1:6`) distributions. Moreover, both of them are examples of discrete (probability is calculated for a few distinctive values) and uniform (values are equally likely to be observed) distributions.

Notice that in the Figure 4 (above) rolling one six-sided dice gives us an uniform distribution (each value is equally likely to be observed). However in the previous chapter when tossing two six-sided dice we got the distribution that looks like this.

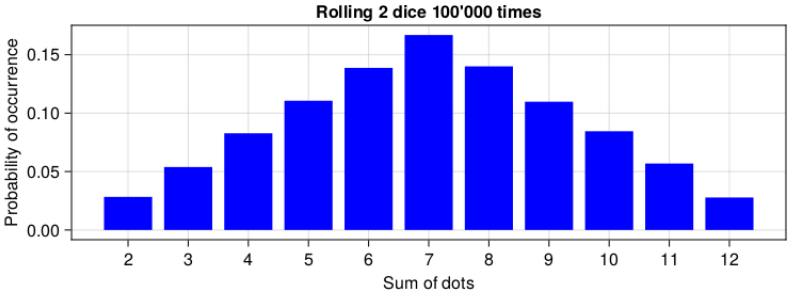

Figure 5: Figure 5: Experimental probability distribution for rolling two 6-sided dice.

What we got here is a bell[149] shaped distribution (c'mon use your imagination). Here the middle values are the ones most likely to occur. It turns out that quite a few distributions may transform into the distribution that is bell shaped (as an exercise you may want to draw a distribution for the number of heads when tossing 10 fair coins simultaneously). Moreover, many biological phenomena got a bell shaped distribution, e.g. men's height or the famous intelligence

[148] https://github.com/b-lukaszuk/RJ_BS_eng/tree/main/code_snippets/ch04
[149] https://en.wikipedia.org/wiki/Bell

quotient[150](aka IQ). The theoretical name for it is normal distribution[151]. Placed on a graph it looks like this.

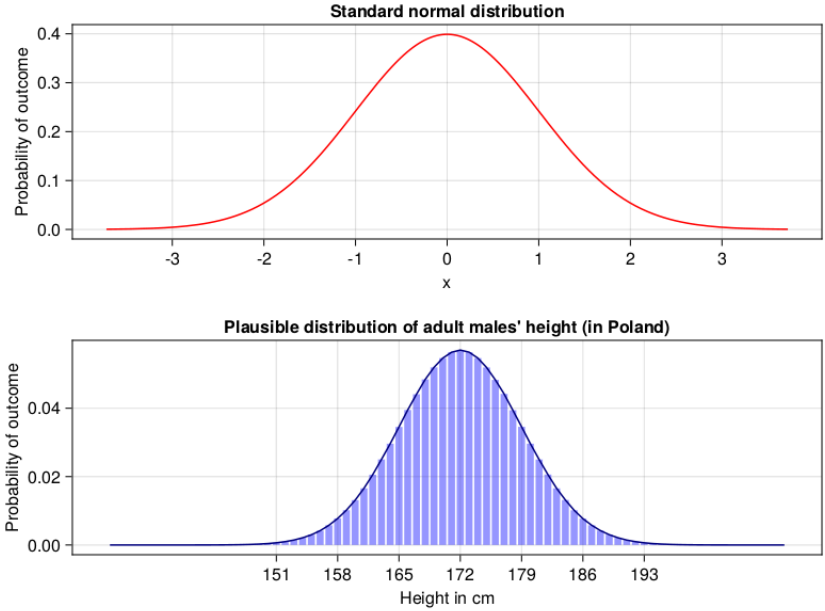

Figure 6: Figure 6: Examples of normal distribution.

In Figure 6 the upper panel depicts standard normal distributions ($\mu = 0, \sigma = 1$, explanation in a moment), a theoretical distribution that all statisticians and probably some mathematicians love. The bottom panel shows a distribution that is likely closer to the adult males' height distribution in my country. Long time ago I read that the average height for an adult man in Poland was 172 [cm] (5.64 [feet]) and the standard deviation was 7 [cm] (2.75 [inch]), hence this plot.

> ***Note:*** In order to get a real height distribution in a country you should probably visit a web site of the country's statistics office instead relying on information like mine.

[150]https://en.wikipedia.org/wiki/Intelligence_quotient
[151]https://en.wikipedia.org/wiki/Normal_distribution

As you can see normal distribution is often depicted as a line plot. That is because it is a continuous distribution (the values on x axes can take any number from a given range). Take a look at the height. In my old identity card[152]next to the field "Height in cm" stands "181", but is this really my precise height? What if during a measurement the height was 180.7 or 181.3 and in the ID there could be only height in integers. I would have to round it, right? So based on the identity card information my real height is probably somewhere between 180.5 and 181.49999... . Moreover, it can be any value in between (like 180.6354551..., although in reality a measuring device does not have such a precision). So, in the bottom panel of Figure 6 I rounded theoretical values for height (`round(height, digits=0)`) obtained from `Rand.rand(Dsts.Normal(172, 7), 10_000_000)` (`Dsts` is `Distributions` package that we will discuss soon enough). Next, I drew bars (using `Cmk.barplot` that you know), and added a line that goes through the middle of each bar (to make it resemble the figure in the top panel).

As you perhaps noticed, the normal distribution is characterized by two parameters:

- the average (also called the mean) (in a population denoted as: μ, in a sample as: $\overline{x}$)
- the standard deviation (in a population denoted as: σ, in a sample as: s, sd or std)

We already know the first one (average) from school and previous chapters (e.g. `getAvg` from Section 3.6.1). However, the last one (standard deviation) requires some explanation.

Let's say that there are two students. Here are their grades.

```
gradesStudA = [3.0, 3.5, 5.0, 4.5, 4.0]
gradesStudB = [6.0, 5.5, 1.5, 1.0, 6.0]
```

[152]https://en.wikipedia.org/wiki/Polish_identity_card

Imagine that we want to send one student to represent our school in a national level competition. Therefore, we want to know who is a better student. So, let's check their averages.

```
avgStudA = getAvg(gradesStudA)
avgStudB = getAvg(gradesStudB)
(avgStudA, avgStudB)
```

```
(4.0, 4.0)
```

Hmm, they are identical. OK, in that situation let's see who is more consistent with their scores.

To test the spread of the scores around the mean we will subtract every single score from the mean and take their average (average of the differences).

```
diffsStudA = gradesStudA .- avgStudA
diffsStudB = gradesStudB .- avgStudB
(getAvg(diffsStudA), getAvg(diffsStudB))
```

```
(0.0, 0.0)
```

> ***Note:*** Here we used the dot operators/functions described in Section 3.6.5

The method is of no use since `sum(diffs)` is always equal to 0 (and hence the average is 0). See for yourself

```
(
    diffsStudA,
    diffsStudB
)
```

```
([-1.0, -0.5, 1.0, 0.5, 0.0],
 [2.0, 1.5, -2.5, -3.0, 2.0])
```

And

```
(sum(diffsStudA), sum(diffsStudB))
```

```
(0.0, 0.0)
```

Personally in this situation I would take the average of diffs without looking at the sign of each difference (`abs` function does that) like so.

```
absDiffsStudA = abs.(diffsStudA)
absDiffsStudB = abs.(diffsStudB)
(getAvg(absDiffsStudA), getAvg(absDiffsStudB))
```

```
(0.6, 2.2)
```

Based on this we would say that student A is more consistent with their grades so he is probably a better student of the two. I would send student A to represent the school during the national level competition. Student B is also good, but choosing him is a gamble. He could shine or embarrass himself (and spot the school's name) during the competition.

For any reason statisticians decided to get rid of the sign in a different way, i.e. by squaring ($x^2 = x*x$) the diffs. Afterwards they calculated the average of it. This average is named variance[153]. Next, they took square root of it ($\sqrt{variance}$) to get rid of the squaring (get the spread of the data in the same scale as the original values, since $\sqrt{x^2} = x$). So, they did more or less this

```
# variance
function getVar(nums::Vector{<:Real})::Real
    avg::Real = getAvg(nums)
    diffs::Vector{<:Real} = nums .- avg
    squaredDiffs::Vector{<:Real} = diffs .^ 2
    return getAvg(squaredDiffs)
end

# standard deviation
function getSd(nums::Vector{<:Real})::Real
```

[153] https://en.wikipedia.org/wiki/Variance

```
    return sqrt(getVar(nums))
end

(getSd(gradesStudA), getSd(gradesStudB))
```

```
(0.7071067811865476, 2.258317958127243)
```

> ***Note:*** In reality the variance and standard deviation for a sample are calculated with slightly different formulas. This is why the numbers returned here may be marginally different from the ones produced by other statistical software. Still, the functions above are easier to understand and give a better feel of the general ideas.

In the end we got similar numbers, reasoning, and conclusions to the ones based on `abs` function. Both the methods rely on a similar intuition, but we cannot expect to get the same results due to the slightly different methodology. For instance given the diffs: `[-2, 3]` we get:

- for squaring: $(-2^2 + 3^2)/2 = (4 + 9)/2 = 13/2 = 6.5$ and $\sqrt{6.5} = 2.55$
- for `abs` values: $(-2 + 3)/2 = (2 + 3)/2 = 5/2 = 2.5$

Although I like my method better the `sd` and squaring/square rooting is so deeply fixed into statistics that everyone should know it. Anyway, as you can see the standard deviation is just an average spread of data around the mean. The bigger value for `sd` the bigger the spread. Of course the opposite is also true.

And now a big question.

Why should we care about the mean (μ, $\overline{x}$) or sd (σ, s, sd, std) anyway?

The answer. For practical reasons that got something to do with the so called three sigma rule[154].

[154] https://en.wikipedia.org/wiki/68%E2%80%9395%E2%80%9399.7_rule

The three sigma rule

The rule[155] says that (here a simplified version made by me):

- roughly 68% of the results in the population lie within $\pm$ 1 sd from the mean
- roughly 95% of the results in the population lie within $\pm$ 2 sd from the mean
- roughly 99% of the results in the population lie within $\pm$ 3 sd from the mean

Example 1

Have you ever tested your blood[156] and received the lab results that said something like

- RBC[157]: 4.45 $[10^6/\mu L]$ (4.2 - 6.00)

The RBC stands for **r**ed **b**lood **c**ell count and the parenthesis contain the reference values (if you are within this normal range then it is a good sign). But where did those reference values come from? This Wikipedia's page[158] gives us a clue. It reports a value for hematocrit[159] (a fraction/percentage of whole blood that is occupied by red blood cells) to be:

- 45 ± 7 (38–52%) for males
- 42 ± 5 (37–47%) for females

Look at this $\pm$ symbol. Have you seen it before? No? Then look at the three sigma rule above.

The reference values were most likely composed in the following way. A large number (let's say 10'000-30'000) of healthy females gave their blood for testing. Hematocrit value was calculated for all of them. The shape of the distribution was established in a similar way to the one we did before (e.g. plotting with a `Cmk` function). The average

[155] https://en.wikipedia.org/wiki/68%E2%80%9395%E2%80%9399.7_rule
[156] https://en.wikipedia.org/wiki/Blood
[157] https://en.wikipedia.org/wiki/Complete_blood_count#Reference_ranges
[158] https://en.wikipedia.org/wiki/Blood
[159] https://en.wikipedia.org/wiki/Hematocrit

hematocrit was 42 units, the standard deviation was 5 units. The majority of the results (roughly 68%) lie within $\pm$ 1 sd from the mean. If so, then we got 42 - 5 = 37, and 42 + 5 = 47. And that is how those two values were considered to be the reference values for the population. Most likely the same is true for other reference values you see in your lab results when you test your blood[160] or when you perform other medical examination.

Example 2

Let's say a person named Peter lives in Poland. Peter approaches the famous IQ test conducted at one of our universities. He read on the internet that there are different intelligence scales[161] used throughout the world. His score is 125. The standard deviation is 24. Is his score high, does it indicate he is gifted (a genius level intellect)? Well, in order to be a genius one has to be in the top 2% of the population with respect to their IQ value. What is the location of Peter's IQ value in the population.

The score of 125 is just a bit greater than 1 standard deviation above the mean (which in an IQ test is always 100). From Section 4.5 we know that when we add the probabilities for all the possible outcomes we get 1 (so the area under the curve in Figure 6 is equal to 1). Half of the area lies on the left, half of it on the right ($\frac{1}{2}$ = 0.5). So, a person with IQ = 100 is as intelligent or more intelligent than half the people ($\frac{1}{2}$ = 0.5 = 50%) in the population. Roughly 68% of the results lies within 1 sd from the mean (half of it below, half of it above). So, from IQ = 100 to IQ = 124 we got (68% / 2 = 34%). By adding 50% (IQ $\leq$ 100) to 34% (100 $\leq$ IQ $\leq$ 124) we get 50% + 34% = 84%. Therefore in our case Peter (with his IQ = 125) is more intelligent than 84% of people in the population (so top 16% of the population). His intelligence is above the average, but it is not enough to label him a genius.

[160] https://en.wikipedia.org/wiki/Complete_blood_count

[161] https://en.wikipedia.org/wiki/Intelligence_quotient#Current_tests

Distributions package

This is all nice and good to know, but in practice it is slow and not precise enough. What if in the previous example the IQ was let's say 139. What is the percentage of people more intelligent than Peter. In the past that kind of questions were to be answered with satisfactory precision using statistical tables at the end of a textbook. Nowadays it can be quickly answered with a greater exactitude and speed, e.g. with the Distributions[162]package. First let's define a helper function that is going to tell us how many standard deviations above or below the mean a given value is (it is called z-score[163])

```
# how many std. devs is value above or below the mean
function getZScore(value::Real, mean::Real, sd::Real)::Float64
    return (value - mean)/sd
end
```

OK, now let's give it a swing. First, something simple IQ = 76, and IQ = 124 (should equal to −1 sd, +1 sd). *Alternatively, look at the value returned by `getZScore` as a value on the x-axis in Figure 6 (top panel).*

```
(getZScore(76, 100, 24), getZScore(124, 100, 24))
```

```
(-1.0, 1.0)
```

Indeed, it seems to be working as expected, and now the value from this task

```
zScorePeterIQ139 = getZScore(139, 100, 24)
zScorePeterIQ139
```

1.625

It is 1.625 sd above the mean. However, we cannot use it directly to estimate the percentage of people above that score because due to the shape of the distribution in Figure 6 the change is not linear: 1 sd ≈

[162]https://juliastats.org/Distributions.jl/stable/

[163]https://en.wikipedia.org/wiki/Standard_score

68%, 2 sd ≈ 95%, 3 sd ≈ 99% (first it changes quickly then it slows down). This is where the `Distributions` package comes into the picture. Under the hood it uses 'scary' mathematical formulas for normal distribution[164]to get us what we want. In our case we use it like this

```
import Distributions as Dsts

Dsts.cdf(Dsts.Normal(), zScorePeterIQ139)
```

0.9479187205847805

Here we first create a standard normal distribution with $\mu = 0$ and $\sigma = 1$ (`Dsts.Normal()`). Then we sum all the probabilities that are lower than or equal to `zScorePeterIQ139` = `getZScore(139, 100, 24)` = 1.625 standard deviation above the mean with `Dsts.cdf`. We see that roughly 0.9479 ≈ 95% of people is as intelligent or less intelligent than Peter. Therefore in this case only ≈0.05 or ≈5% of people are more intelligent than him. Alternatively you may say that the probability that a randomly chosen person from that population is more intelligent than Peter is ≈0.05 or ≈5%.

> ***Note:*** `cdf` in `Dsts.cdf` stands for cumulative distribution function[165]. For more information on `Dsts.cdf` see these docs[166]or for `Dsts.Normal` those docs[167].

The above is a classical method and it is useful to know it. Based on the z-score you can check the appropriate percentage/probability for a given value in a table that is usually placed at the end of a statistics textbook. Make sure you understand it since, we are going to use this method, e.g. in the upcoming chapter on a Student's t-test (see Section 5.2).

---

[164]https://en.wikipedia.org/wiki/Normal_distribution

[165]https://en.wikipedia.org/wiki/Cumulative_distribution_function

[166]https://juliastats.org/Distributions.jl/stable/univariate/#Distributions.cdf-Tuple%7BUnivariateDistribution,%20Real%7D

[167]https://juliastats.org/Distributions.jl/stable/univariate/#Distributions.Normal

Luckily, in the case of the normal distribution we don't have to calculate the z-score. The package can do that for us, compare

```
# for better clarity each method is in a separate line
(
    Dsts.cdf(Dsts.Normal(), getZScore(139, 100, 24)),
    Dsts.cdf(Dsts.Normal(100, 24), 139)
)
```

```
(0.9479187205847805, 0.9479187205847805)
```

So, in this case you can either calculate the z-score for standard normal distribution with $\mu = 0$ and $\sigma = 1$ or define a normal distribution with a given mean and sd (here `Dsts.Normal(100, 24)`) and let the `Dsts.cdf` calculate the z-score (under the hood) and probability (it returns it) for you.

To further consolidate our knowledge. Let's go with another example. Remember that I'm 181 cm tall. Hmm, I wonder what percentage of men in Poland is taller than me if $\mu = 172$ [cm] and $\sigma = 7$ [cm].

```
1 - Dsts.cdf(Dsts.Normal(172, 7), 181)
```

0.09927139684333097

The `Dsts.cdf` gives me left side of the curve (the area under the curve for height ≤ 181). So in order to get those that are higher than me I subtracted it from 1. It seems that under those assumptions roughly 10% of men in Poland are taller than me (approx. 1 out of 10 men that I encounter is taller than me). I could also say: "the probability that a randomly chosen man from that population is higher than me is ≈ 0.1 or $\approx 10\%$. Alternatively I could have used Dsts.ccdf[168]function which under the hood does `1 - Dsts.cdf(distribution, xCutoffPoint)`.

OK, and how many men in Poland are exactly as tall as I am? In general that is the job for `Dsts.pdf` (`pdf` stands for probability density

[168]https://juliastats.org/Distributions.jl/stable/univariate/#Distributions.ccdf-Tuple%7BUnivariateDistribution,%20Real%7D

function[169], see the docs for Dsts.pdf[170]). It works pretty well for discrete distributions (we talked about them at the beginning of this sub-chapter). For instance theoretical probability of getting 12 while rolling two six-sided dice is

```
Dsts.pdf(Dsts.Binomial(2, 1/6), 2)
```

0.02777777777777778

Compare it with the empirical probability from Section 4.5 which was equal to 0.0278. Here we treated it as a binomial distribution (success: two sixes (6 + 6 = 12), failure: other result) hence `Dsts.Binomial` with `2` (number of dice to roll) and `1/6` (probability of getting 6 in a single roll). Then we used `Dsts.pdf` to get the probability of getting exactly two sixes. More info on `Dsts.Binomial` can be found here[171]and on `Dsts.pdf` can be found there[172].

However there is a problem with using `Dsts.pdf` for continuous distributions because it can take any of the infinite values within the range. Remember, in theory there is an infinite number of values between 180 and 181 (like 180.1111, 180.12222, etc.). So usually for practical reasons it is recommended not to calculate a probability density function (hence `pdf`) for a continuous distribution (1 / infinity $\approx$ 0). Still, remember that the height of 181 [cm] means that the value lies somewhere between 180.5 and 181.49999... . Moreover, we can reliably calculate the probabilities (with `Dsts.cdf`) for $\leq$ 180.5 and $\leq$ 181.49999... so a good approximation would be

```
heightDist = Dsts.Normal(172, 7)
# 2 digits after dot because of the assumed precision of a measuring
device
Dsts.cdf(heightDist, 181.49) - Dsts.cdf(heightDist, 180.50)
```

[169]https://en.wikipedia.org/wiki/Probability_density_function

[170]https://juliastats.org/Distributions.jl/stable/univariate/#Distributions.pdf-Tuple%7BUnivariateDistribution,%20Real%7D

[171]https://juliastats.org/Distributions.jl/stable/univariate/#Distributions.Binomial

[172]https://juliastats.org/Distributions.jl/stable/univariate/#Distributions.pdf-Tuple%7BUnivariateDistribution,%20Real%7D

0.024724273314878698

OK. So it seems that roughly 2.5% of adult men in Poland got 181 [cm] in the field "Height" in their identity cards. If there are let's say 10 million adult men in Poland then roughly 250000.0 (so 250 k) people are approximately my height. Alternatively under those assumptions the probability that a random man from the population is as tall as I am (181 cm in the height field of his identity card) is ≈0.025 or ≈2.5%.

If you are still confused about this method take a look at the figure below.

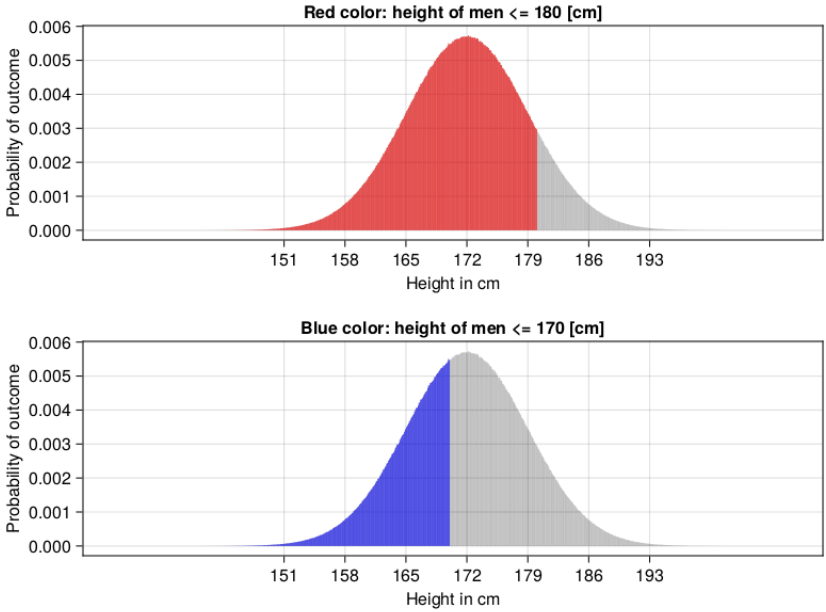

Figure 7: Figure 7: Using cdf to calculate proportion of men that are between 170 and 180 [cm] tall.

Here for better separation I placed the height of men between 170 and 180 [cm]. The method that I used subtracts the area in blue from the area in red (red - blue). That is exactly what I did (but for 181.49 and 180.50 [cm]) when I typed `Dsts.cdf(heightDist, 181.49) - Dsts.cdf(heightDist, 180.50)` above.

OK, time for the last theoretical sub-chapter in this section. Whenever you're ready click on the right arrow.

Hypothesis testing

OK, now we are going to discuss a concept of hypothesis testing. But first let's go through an example from everyday life that we know or at least can imagine. Ready?

A game of tennis

So imagine there is a group of people and among them two amateur tennis players: John and Peter. Everyone wants to know which one of them is a better tennis player. Well, there is only one way to find out. Let's play some games!

As far as I'm aware a tennis match can end with a win of one player, the other loses (there are no draws). Before the games the people set the rules. Everyone agrees that the players will play six games. To prove their supremacy a player must win all six games (six wins in a row are unlikely to happen by accident, I hope we can all agree on that). The series of games ends with the result 0-6 for Peter. According to the previously set rules he is declared the local champion.

Believe it or not but this is what statisticians do. Of course they use more formal methodology and some mathematics, but still, this is what they do:

- before the experiment they start with two assumptions
  - initial assumption: be fair and assume that both players play equally well (this is called the null hypothesis[173], H_0)
  - alternative assumption: one player is better than the other (this is called the alternative hypothesis[174], H_A)
- before the experiment they decide on how big a sample should be (in our case six games).
- before the experiment they decide on the cutoff level, once it is reached they will abandon the initial assumption (H_0) and chose the alternative (H_A). In our case the cutoff is: six games in a row won by a player

- they conduct the experiment (players play six games) and record the results
- after the experiment when the result provides enough evidence (in our case six games won by the same player) they decide to reject H_0, and choose H_A. Otherwise they stick to their initial assumption (they do not reject H_0)

And that's how it is, only that statisticians prefer to rely on probabilities instead of absolute numbers. So in our case a statistician says:

"I assume that H_0 is true. Then I will conduct the experiment and record the result. I will calculate the probability of such a result (or a more extreme result) happening by chance. If it is small enough, let's say 5% or less ($prob \leq 0.05$), then the result is unlikely to have occurred by accident. Therefore I will reject my initial assumption (H_0) and choose the alternative (H_A). Otherwise I will stay with my initial assumption."

Let's see such a process in practice and connect it with what we already know.

Tennis - computer simulation

First a computer simulation.

```
# result of 6 tennis games under H0 (equally strong tennis players)
function getResultOf6TennisGames()
    return sum(Rand.rand(0:1, 6)) # 0 means John won, 1 means Peter won
end

Rand.seed!(321)
tennisGames = [getResultOf6TennisGames() for _ in 1:100_000]
tennisCounts = getCounts(tennisGames)
tennisProbs = getProbs(tennisCounts)
```

Here `getResultOf6TennisGames` returns a result of 6 games under H_0 (both players got equal probability to win a game). When John wins a

[173] https://en.wikipedia.org/wiki/Null_hypothesis

[174] https://en.wikipedia.org/wiki/Alternative_hypothesis

game then we get 0, when Peter we get 1. So if after running `getResultOf6TennisGames` we get, e.g. 4 we know that Peter won 4 games and John won 2 games. We repeat the experiment 100'000 times to get a reliable estimate of the results distribution.

OK, at the beginning of this chapter we intuitively said that a player needs to win 6 games to become the local champion. We know that the result was 0-6 for Peter. Let's see what is the probability that Peter won by chance six games in a row (assuming H_0 is true).

```
tennisProbs[6]
```

0.01538

In this case the probability of Peter winning by chance six games in a row is very small. If we express it graphically it roughly looks like this:

```
# prob = 0.015
impossible |||||||||||||||||||||||||||||||||||||||||||||||||| certain
            Δ
```

So, it seems that intuitively we set the cutoff level well. Let's see if the statistician from the quotation above would be satisfied ("If it is small enough, let's say 5% or less ($prob \leq 0.05$), then the result is unlikely to have occurred by accident. Therefore I will reject my initial assumption (H_0) and choose the alternative (H_A). Otherwise I will stay with my initial assumption.")

First, let's compare them graphically.

```
# prob = 0.05
impossible |||||||||||||||||||||||||||||||||||||||||||||||||| certain
             Δ

# prob = 0.0153
impossible |||||||||||||||||||||||||||||||||||||||||||||||||| certain
            Δ
```

Although our text based graphics is slightly imprecise, we can see that the obtained probability lies below (to the left of) our cutoff level. And now more precise mathematical comparison.

```
# sigLevel - significance level for probability
# 5% = 5/100 = 0.05
function shouldRejectH0(prob::Float64, sigLevel::Float64 = 0.05)::Bool
    @assert (0 <= prob <= 1) "prob must be in range [0-1]"
    @assert (0 <= sigLevel <= 1) "sigLevel must be in range [0-1]"
    return prob <= sigLevel
end

shouldRejectH0(tennisProbs[6])
```

true

Indeed he would. He would have to reject H_0 and assume that one of the players (here Peter) is a better player (H_A).

Tennis - theoretical calculations

OK, to be sure of our conclusions let's try the same with the Distributions[175]package (imported as `Dsts`) that we met before.

Remember one of the two tennis players must win a game (John or Peter). So this is a binomial distributions we met before. We assume (H_0) both of them play equally well, so the probability of any of them winning is 0.5. Now we can proceed like this using a dictionary comprehension similar to the one that we have met before (e.g. see `getProbs` definition from Section 4.4)

```
tennisTheorProbs = Dict(
    i => Dsts.pdf(Dsts.Binomial(6, 0.5), i) for i in 0:6
    )
tennisTheorProbs[6]
```

0.015624999999999977

Yep, the number is pretty close to `tennisProbs[6]` we got before which is 0.01538. So we decide to go with H_A and say that Peter is a better player. Just in case I will place both distributions (experimental

[175]https://juliastats.org/Distributions.jl/stable/

and theoretical) one below the other to make the comparison easier. Behold

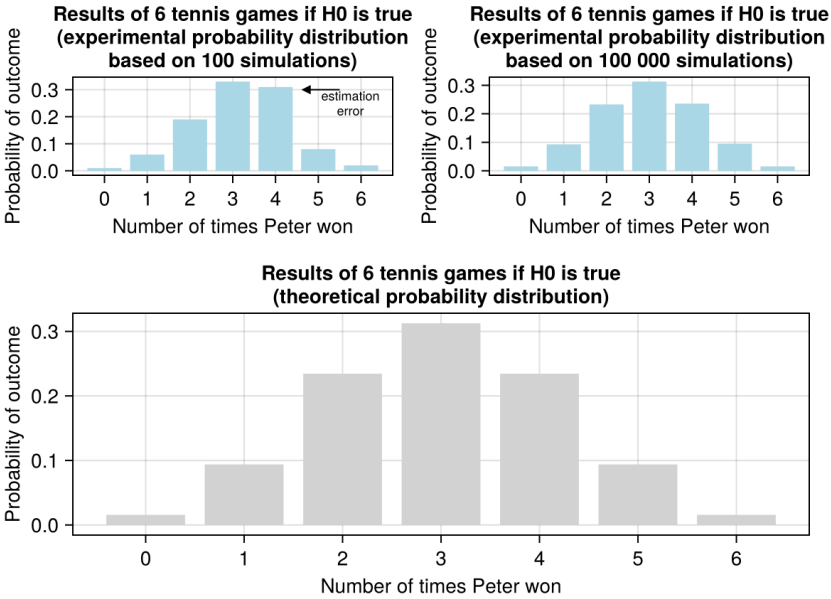

Figure 8: Figure 8: Probability distribution for 6 tennis games if H_0 is true.

Notice that in order to get a satisfactory approximation of theoretical probabilities a sufficiently large number of repetitions needs to be ensured. Figure 8 (row 1, column 1) demonstrates an imprecise probability estimation obtained when only 100 computer simulations were used. In this case it could be noticed in few places, but it is especially evident in the case of overly large bar at x = 4 (indicated by the arrow).

Anyway, once we have warmed up we can even calculate the probability using our knowledge from Section 4.3.1 . We can do this since by assuming our null hypothesis (H_0) we basically compared the result of a game between John and Peter to a fair coin's toss (0 or 1, John or Peter, heads or tails).

The probability of Peter winning a single game is $P(Peter) = \frac{1}{2} = 0.5$. Peter won all six games. In order to get two wins in a row, first he had to won one game. In order to get three wins in a row first he had to won two games in a row, and so on. So he had to win game 1 AND game 2 AND game 3 AND ... . Given the above, and what we stated in Section 4.3.1 , here we deal with a conjunction of probabilities. Therefore we use probability multiplication like so

```
tennisTheorProbWin6games = 0.5 * 0.5 * 0.5 * 0.5 * 0.5 * 0.5
# or
tennisTheorProbWin6games = 0.5 ^ 6

tennisTheorProbWin6games
```

0.015625

Compare it with `tennisTheorProbs[6]` calculated by `Distributions` package

```
(tennisTheorProbs[6], tennisTheorProbWin6games)
```

```
(0.015624999999999977, 0.015625)
```

They are the same. The difference is caused by a computer representation of floats and their rounding (as a reminder see Section 3.3.3 , and Section 3.9.2).

Anyway, I just wanted to present all three methods for two reasons. First, that's the way we checked our reasoning at math in primary school (solving with different methods). Second, chances are that one of the explanations may be too vague for you, if so help yourself to the other methods :)

In general, as a rule of thumb you should remember that the null hypothesis (H_0) assumes lack of differences/equality, etc. (and this is what we assumed in this tennis example).

One or two tails

Hopefully, the above explanations were clear enough. There is a small nuance to what we did. In the beginning of Section 4.7.1 we said 'To prove their supremacy a player must win all six games'. A player, so either John or Peter. Still, we calculated only the probability of Peter winning the six games (`tennisTheorProbs[6]`), Peter and not John. What we did there was calculating one tail probability[176](see the figures in the link). Now, take a look at Figure 8 (e.g. bottom panel) the middle of it is 'body' and the edges to the left and right are tails.

This approach (one-tailed test) is rather OK in our case. However, in statistics it is frequently recommended to calculate two-tails probability (usually this is the default option in many statistical functions/packages). That is why at the beginning of Section 4.7.1 I wrote 'alternative assumption: one player is better than the other (this is called alternative hypothesis, H_A)'.

Calculating the two-tailed probability is very simple, we can either add `tennisTheorProbs[6] + tennisTheorProbs[0]` (remember 0 means that John won all six games) or multiply `tennisTheorProbs[6]` by 2 (since the graph in Figure 8 is symmetrical).

```
(tennisTheorProbs[6] + tennisTheorProbs[0], tennisTheorProbs[6] * 2)
```

```
(0.031249999999999955, 0.031249999999999955)
```

Once we got it we can perform our reasoning one more time.

```
shouldRejectH0(tennisTheorProbs[6] + tennisTheorProbs[0])
```

true

In this case the decision is the same (but that is not always the case). As I said before in general it is recommended to choose a two-tailed test over a one-tailed test. Why? Let me try to explain this with another example.

[176]https://en.wikipedia.org/wiki/One-_and_two-tailed_tests

Imagine I tell you that I'm a psychic that talks with the spirits and I know a lot of the stuff that is hidden from mere mortals (like the rank and suit of a covered playing card[177]). You say you don't believe me and propose a simple test.

You take 10 random cards from a deck. My task is to tell you the color (red or black). And I did, the only problem is that I was wrong every single time! If you think that proves that your were right in the first place then try to guess 10 cards in a row wrongly yourself (if you don't have cards on you go with 10 consecutive fair coin tosses).

It turns out that guessing 10 cards wrong is just as unlikely as guessing 10 of them right (`0.5^10` = 0.0009765625 or 1 per 1024 tries in each case). This could potentially mean a few things, e.g.

- I really talk with the spirits, but in their language "red" means "black", and "black" means "red" (cultural fun fact: they say Bulgarians nod their heads when they say "no", and shake them for "yes"),
- I live in one of 1024 alternative dimensions/realities and in this reality I managed to guess all of them wrong, when the other versions of me had mixed results, and that one version of me guessed all of them right,
- I am a superhero and have an x-ray vision in my eyes so I saw the cards, but I decided to tell them wrong to protect my secret identity,
- I cheated, and were able to see the cards beforehand, but decided to mock you,
- or some other explanation is in order, but I didn't think of it right now.

The small probability only tells us that the result is unlikely to has happened by chance alone. Still, you should always choose your null (H_0) and alternative (H_A) hypothesis carefully. Moreover, it is a good idea to look at both ends of a probability distribution.

[177] https://en.wikipedia.org/wiki/Playing_card

All the errors that we make

Long time ago when I was a student I visited a local chess club. I was late that day, and only one person was without a pair, Paul. I introduced myself and we played a few games. In chess you can either win, lose, or draw a game. Unfortunately, I lost all six games we played that day. I was upset, I assumed I just encountered a better player. I thought: "Too bad, but next week I will be on time and find someone else to play with" (nobody likes loosing all the time). The next week I came to the club, and again the only person without a pair was Paul (just my luck). Still, despite the bad feelings I won all six games that we played that day (what are the odds). Later on it turned out that me and Paul are pretty well matched chess players (we played chess at a similar level).

The story demonstrates that even when there is a lot of evidence (six lost games during the first meeting) we can still make an error by rejecting our null hypothesis (H_0).

In fact, whenever we do statistics we turn into judges, since we can make a mistake in two ways (see Figure 9).

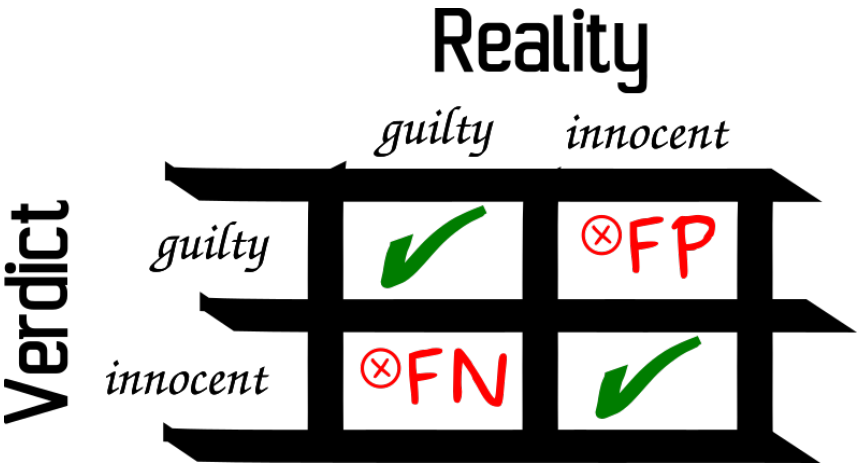

Figure 9: Figure 9: A judge making a verdict. FP - false positive, FN - false negative.

An accused is either guilty or innocent. A judge (or a jury in some countries) sets a verdict based on the evidence.

If the accused is innocent but is sentenced anyway then it is an error, it is usually called **type I error**[178](FP - false positive in Figure 9). Its probability is denoted by the first letter of the Greek alphabet, so alpha (α).

In the case of John and Peter playing tennis the type I probability was ≤ 0.05. More precisely it was `tennisTheorProbs[6]` = 0.015625 (for a one tailed test).

If the accused is guilty but is declared innocent then it is another type of error, it is usually called **type II error** (FN - false negative in Figure 9). Its probability is denoted by the second letter of the Greek alphabet, so beta (β). Beta helps us determine the power of a test[179] (power = 1 - β), i.e. if H_A is really true then how likely it is that we will choose H_A over H_0.

So to sum up, in the judge analogy a really innocent person is H_0 being true and a really guilty person is H_A being true.

Unfortunately, most of the statistical textbooks that I've read revolve around type I errors and alphas, whereas type II error is covered much less extensively (hence my own knowledge of the topic is more limited).

In the tennis example above we rejected H_0, hence here we risk committing the type I error. Therefore, we didn't speak about the type II error, but don't worry we will discuss it in more detail in the upcoming exercises at the end of this chapter (see Section 4.8.5).

Cutoff levels

OK, once we know what are the type I and type II errors it is time to discuss their cutoff values.

Obviously, the ideal situation would be if the probabilities of both type I and type II errors were exactly 0 (no mistakes is always the best). The only problem is that this is not possible. In our tennis example one player won all six games, and still some small risk of a mistake existed

[178]https://en.wikipedia.org/wiki/Type_I_and_type_II_errors

[179]https://en.wikipedia.org/wiki/Power_of_a_test

(`tennisTheorProbs[6]` = 0.015625). If you ever see a statistical package reporting a p-value to be equal, e.g. 0.0000, then this is just rounding to 4 decimal places and not an actual zero. So what are the acceptable cutoff levels for α (probability of type I error) and β (probability of type II error).

The most popular choices for α cutoff values are:

- 0.05, or
- 0.01

Actually, as far as I'm aware, the first of them ($\alpha = 0.05$) was initially proposed by Ronald Fisher[180], a person sometimes named the father of the XX-century statistics. This value was chosen arbitrarily and is currently frowned upon by some modern statisticians as being to lenient. Therefore, 0.01 is proposed as a more reasonable alternative.

As regards β its two most commonly accepted cutoff values are:

- 0.2, or
- 0.1

Actually, as far as I remember the textbooks usually do not report values for β, but for power of the test (if H_A is really true then how likely it is that we will choose H_A over H_0) to be 0.8 or 0.9. However, since as we mentioned earlier power = 1 - β, then we can easily calculate the value for this parameter.

OK, enough of theory, time for some practice. Whenever you're ready click the right arrow to proceed to the exercises that I prepared for you.

Statistics intro - Exercises

So, here are some exercises that you may want to solve to get from this chapter as much as you can (best option). Alternatively, you may read the task descriptions and the solutions (and try to understand them).

[180] https://en.wikipedia.org/wiki/Ronald_Fisher

Exercise 1
Some mobile phones and cash dispensers prevent unauthorized access to the resources by using a 4-digit PIN number.

What is the probability that a randomly typed number will be the right one?

Hint. Calculate how many different numbers you can type. If you get stuck, try to reduce the problem to 1- or 2-digit PIN number.

Exercise 2
A few years ago during a home party a few people bragged that they can recognize beer blindly, just by taste, since, e.g. "the beer of brand X is great, of brand Y is OK, but of brand Z tastes like piss" (hmm, how could they tell?).

We decided to put that to the test. We bought six different beer brands. One person poured them to cups marked 1-6. The task was to taste the beer and correctly place a label on it.

What is the probability that a person would place correctly 6 labels on 6 different beer at random.

Hint. This task may be seen as ordering of different objects. As always you may reduce the problem to a smaller one. For instance think how many different orderings of 3 beer do we have.

Exercise 3
Do you still remember our tennis example from Section 4.7.1 , I hope so. Let's modify it a bit to solidify your understanding of the topic.

Imagine John and Peter played 6 games, but this time the result was 1-5 for Peter. Is the difference statistically significant at the crazy cutoff level for α equal to 0.15. Calculate the probability (the famous p-values) for one- and two-tailed tests.

Exercise 4
In the opening to Section 4.7.5 I told you a story from the old times. The day when I met my friend Paul in a local chess club and lost 6 games in a row while playing with him. So, here is a task for you. If we

were both equally good chess players at that time then what is the probability that this happened by chance (to make it simpler do one-tailed test)?

Exercise 5
Remember how in Section 4.7.5 we talked about a type II error. We said that if we decide not to reject H_0 we risk to commit a type II error or β. It is FN, i.e. false negative, in our judge analogy from Section 4.7.5 (declaring a person that is really guilty to be innocent). In statistics this is when the H_A is true but we fail to say so and stay with our initial hypothesis (H_0).

So here is the task.

Assume that the result of the six tennis games was 1-5 for Peter (like in Section 4.8.3). Write a computer simulation that estimates the probability of type II error that we commit in this case by not rejecting H_0 (if the cutoff level for α is equal to 0.05). To make it easier use one-tailed probabilities.

Hint: assume that H_A is true and that in reality Peter wins with John on average with the ratio 5 to 1 (5 wins - 1 defeat).

Statistics intro - Solutions

In this sub-chapter you will find exemplary solutions to the exercises from the previous section.

Solution to Exercise 1
The easiest way to solve this problem is to reduce it to a simpler one.

If the PIN number were only 1-digit, then the total number of possibilities would be equal to 10 (numbers from 0 to 9).

For a 2-digit PIN the pattern would be as follow:

```
00
01
02
...
09
10
```

```
11
12
...
19
20
21
...
98
99
```

So, for every number in the first location there are 10 numbers (0-9) in the second location. Just like in a counter (see gif below), the number on the left switches to the next only when 10 numbers on the right changed beforehand.

Figure 10: A counter (animation works only in an HTML document) depicting rate of number changes.

Therefore in total we got numbers in the range 00-99, or to write it mathematically 10 * 10 different numbers (numbers per pos. 1 * numbers per pos. 2).

By extension the total number of possibilities for a 4-digit PIN is:

```
# (method1, method2, method3)
(10 * 10 * 10 * 10, 10^4, length(0:9999))
```

```
(10000, 10000, 10000)
```

So 10'000 numbers. Therefore the probability for a random number being the right one is `1/10_000` = 0.0001

Similar methodology is used to assess the strength of a password to an internet website.

Solution to Exercise 2
OK, so let's reduce the problem before we solve it.

If I had only 1 beer and 1 label then there is only one way to do it. The label in my hand goes to the beer in front of me.

For 2 labels and 2 beer it goes like this:

```
a b
b a
```

I place one of two labels on a first beer, and I'm left with only 1 label for the second beer. So, 2 possibilities in total.

For 3 labels and 3 beer the possibilities are as follow:

```
a b c
a c b

b a c
b c a

c a b
c b a
```

So here, for the first beer I can assign any of the three labels (`a`, `b`, or `c`). Then I move to the second beer and have only two labels left in my hand (if the first got `a`, then the second can get only `b` or `c`). Then I move to the last beer with the last label in my hand (if the first two were `a` and `b` then I'm left with `c`). In total I got `3 * 2 * 1` = 6 possibilities.

It turns out this relationship holds also for bigger numbers. In mathematics it can be calculated using the factorial[181] function that is already implemented in Julia (see the docs[182]).

Still, for practice we're gonna implement one on our own with the `foreach` we met in Section 3.6.4 .

```
function myFactorial(n::Int)::Int
    @assert n > 0 "n must be positive"
    product::Int = 1
    foreach(x -> product *= x, 1:n)
    return product
end

myFactorial(6)
```

720

> ***Note:*** You may also just use Julia's prod[183] function, e.g. `prod(1:6)` = 720. Still, be aware that factorial numbers grow pretty fast, so for bigger numbers, e.g. `myFactorial(20)` or above you might want to change the definition of `myFactorial` to use `BigInt` that we met in Section 3.9.5 .

So, the probability that a person correctly labels 6 beer at random is `round(1/factorial(6), digits=5)` = 0.00139 = 1/720.

I guess that is the reason why out of 7 people that attempted to correctly label 6 beer the results were as follows:

- one person correctly labeled 0 beer
- five people correctly labeled 1 beer
- one person correctly labeled 2 beer

I leave the conclusions to you.

[181] https://en.wikipedia.org/wiki/Factorial
[182] https://docs.julialang.org/en/v1/base/math/#Base.factorial
[183] https://docs.julialang.org/en/v1/base/collections/#Base.prod

Solution to Exercise 3

OK, for the original tennis example (see Section 4.7.1) we answered the question by using a computer simulation first (Section 4.7.2). For a change, this time we will start with a 'purely mathematical' calculations. Ready?

In order to get the result of 1-5 for Peter we would have to get a series of games like this one:

```
# 0 - John's victory, 1 - Peter's victory
0 1 1 1 1 1
```

Probability of either John or Peter winning under H_0 (assumption that they play equally well) is $\frac{1}{2}$ = 0.5. So here we got a conjunction of probabilities (John won AND Peter won AND Peter won AND ...). According to what we've learned in Section 4.3.1 we should multiply the probabilities by each other.

Therefore, the probability of the result above is `0.5 * 0.5 * 0.5 * ...` or `0.5 ^ 6` = 0.015625. But wait, there's more. We can get such a result (1-5 for Peter) in a few different ways, i.e.

```
0 1 1 1 1 1
# or
1 0 1 1 1 1
# or
1 1 0 1 1 1
# or
1 1 1 0 1 1
# or
1 1 1 1 0 1
# or
1 1 1 1 1 0
```

> ***Note:*** For a big number of games it is tedious and boring to write down all the possibilities by hand. In this case you may use Julia's binomial[184]function, e.g. `binomial(6, 5)` = 6. This tells us how many different fives of six objects can we get.

[184]https://docs.julialang.org/en/v1/base/math/#Base.binomial

As we said a moment ago, each of this series of games occurs with the probability of 0.015625. Since we used OR (see the comments in the code above) then according to Section 4.3.1 we can add 0.015625 six times to itself (or multiply it by 6). So, the probability is equal to:

```
prob1to5 = (0.5^6) * 6 # parenthesis were placed for the sake of clarity
prob1to5
```

0.09375

Of course we must remember what our imaginary statistician said in Section 4.7.1 : "I assume that H_0 is true. Then I will conduct the experiment and record then result. I will calculate the probability of such a result (or more extreme result) happening by chance."

`More extreme` than 1-5 for Peter is 0-6 for Peter, we previously (see Section 4.7.3) calculated it to be `0.5^6` = 0.015625. Finally, we can get our p-value (for one-tailed test)

```
prob1to5 = (0.5^6) * 6 # parenthesis were placed for the sake of clarity
prob0to6 = 0.5^6
probBothOneTail = prob1to5 + prob0to6

probBothOneTail
```

0.109375

> ***Note:*** Once you get used to calculating probabilities you should use quick methods like those from `Distributions` package (presented below), but for now it is important to understand what happens here, hence those long calculations (of `probBothOneTail`) shown here.

Let's quickly verify it with other methods we met before (e.g. in Section 4.7)

```
# for better clarity each method is in a separate line
(
    probBothOneTail,
```

```
    1 - Dsts.cdf(Dsts.Binomial(6, 0.5), 4),
    Dsts.pdf.(Dsts.Binomial(6, 0.5), 5:6) |> sum,
    tennisProbs[5] + tennisProbs[6] # experimental probability
)
```

```
(0.109375, 0.109375, 0.10937499999999988, 0.11052000000000001)
```

Yep, they all appear the same (remember about floats rounding and the difference between theory and practice from Section 4.4).

So, is it significant at the crazy cutoff level of $\alpha = 0.15$?

```
shouldRejectH0(probBothOneTail, 0.15)
```

true

Yes, it is (we reject H_0 in favor of H_A). And now for the two-tailed test (so either Peter wins at least 5 to 1 or John wins with the exact same ratio).

```
# remember the probability distribution is symmetrical, so *2 is OK here
shouldRejectH0(probBothOneTail * 2, 0.15)
```

false

Here we cannot reject our H_0.

Of course we all know that this was just for practice, because the acceptable type I error cutoff level is usually 0.05 or 0.01. In this case, according to both the one-tailed and two-tailed tests we failed to reject the H_0.

BTW, this shows how important it is to use a strict mathematical reasoning and to adhere to our own methodology. I don't know about you but when I was a student I would have probably accepted the result 1-5 for Peter as an intuitive evidence that he is a better tennis player.

We will see how to speed up the calculations in this solution in one of the upcoming chapters (see Section 6.2).

Solution to Exercise 4

OK, there maybe more than one way to solve this problem.

Solution 4.1

In chess, a game can end with one of the three results: white win, black win or a draw. If we assume each of those options to be equally likely for a two well matched chess players then the probability of each of the three results happening by chance is `1/3` (this is our H_0).

So, similarly to our tennis example from Section 4.7.1 the probability (one-tailed test) of Paul winning all six games is

```
# (1/3) that Paul won a single game AND six games in a row (^6)
(
round((1/3)^6, digits=5),
round(Dsts.pdf(Dsts.Binomial(6, 1/3), 6), digits=5)
)
```

```
(0.00137, 0.00137)
```

So, you might think right now 'That task was a piece of cake' and you would be right. But wait, there's more.

Solution 4.2

In chess played at a top level (>= 2500 ELO) the most probable outcome is draw. It occurs with a frequency of roughly 50% (see this Wikipedia's page[185]). Based on that we could assume that for a two equally strong chess players the probability of:

- white winning is `1/4`,
- draw is `2/4 = 1/2`,
- black winning is `1/4`

So under those assumptions the probability that Paul won all six games is

```

```

[185] https://en.wikipedia.org/wiki/Draw_(chess)#Frequency_of_draws

```
# (1/4) that Paul won a single game AND six games in a row (^6)
(
round((1/4)^6, digits=5),
round(Dsts.pdf(Dsts.Binomial(6, 1/4), 6), digits=5)
)
```

```
(0.00024, 0.00024)
```

So a bit lower, than the probability we got before (which was `(1/3)^6` = 0.00137).

OK, so I presented you with two possible solutions. One gave the probability of `(1/3)^6` = 0.00137, whereas the other `(1/4)^6` = 0.00024. So, which one is it, which one is the true probability? Well, most likely neither. All they really are is just some estimations of the true probability and they are only as good as the assumptions that we make. After all: "All models are wrong, but some are useful"[186].

If the assumptions are correct, then we can get a pretty good estimate. Both the `Solution 4.1` and `Solution 4.2` got reasonable assumptions but they are not necessarily true (e.g. I'm not a >= 2500 ELO chess player). Still, for practical reasons they may be more useful than just guessing, for instance if you were ever to bet on a result of a chess game/match (do you remember the bets from Section 4.5 ?). They may not be good enough for you to win such a bet, but they could allow to reduce the losses.

However, let me state it clearly. The reason I mentioned it is not for you to place bets on chess matches but to point on similarities to statistical practice.

For instance, there is a method named one-way ANOVA[187](we will discuss it, e.g. in the upcoming Section 5.4). Sometimes the analysis requires us to conduct a so called post-hoc test[188]. There are quite a few of them to choose from (see the link above) and they rely on

[186]https://en.wikipedia.org/wiki/All_models_are_wrong
[187]https://en.wikipedia.org/wiki/One-way_analysis_of_variance
[188]https://en.wikipedia.org/wiki/Post_hoc_analysis

different assumptions. For instance one may do the Fisher's LSD test or the Tukey's HSD test. Which one to choose? I think you should choose the test that is better suited for the job (based on your knowledge and recommendations from the experts).

Regarding the above mentioned tests. The Fisher's LSD test was introduced by Ronald Fisher[189](what a surprise). LSD stands for **L**east **S**ignificant **D**ifference. Some time later John Tukey[190]considered it to be too lenient (too easily rejects H_0 and declares significant differences) and offered his own test (operating on different assumptions) as an alternative. For that reason it was named HSD which stands for **H**onestly **S**ignificant **D**ifference. I heard that statisticians recommend to use the latter one (although in practice I saw people use either of them).

Solution to Exercise 5

OK, so we assume that Peter is a better player than John and he consistently wins with John. On average he wins with the ratio 5 to 1 (5:1) with his opponent (this is our true H_A). Let's write a function that gives us the result of the experiment if this H_A is true.

```
function getResultOf1TennisGameUnderHA()::Int
    # 0 - John wins, 1 - Peter wins
    return Rand.rand([0, 1, 1, 1, 1, 1], 1)
end

function getResultOf6TennisGamesUnderHA()::Int
    return [getResultOf1TennisGameUnderHA() for _ in 1:6] |> sum
end
```

The code is fairly simple. Let me just explain one part. Under H_A Peter wins 5 out of six games and John 1 out of 6, therefore we choose one number out of `[0, 1, 1, 1, 1, 1]` (0 - John wins, 1 - Peter wins) with our `Rand.rand([0, 1, 1, 1, 1, 1], 1)`.

[189]https://en.wikipedia.org/wiki/Ronald_Fisher
[190]https://en.wikipedia.org/wiki/John_Tukey

Note: If the H_A would be let's say 1:99 for Peter, then to save you some typing I would recommend to do something like, e.g. `return (Rand.rand(1:100, 1) <= 99) ? 1 : 0`. It draws one random number out of 100 numbers. If the number is 1-99 then it returns 1 (Peter wins) else it returns 0 (John wins). BTW. When a probability of an event is small (e.g. $\leq 1\%$) then to get its more accurate estimate you could/should increase the number of computer simulations [e.g. `numOfSimul` below should be `1_000_000` (shorter form `10^6`) instead of `100_000` (shorter form `10^5`)].

Alternatively the code from the snippet above could be shortened to

```
# here no getResultOf1TennisGameUnderHA is needed
function getResultOf6TennisGamesUnderHA()::Int
    return Rand.rand([0, 1, 1, 1, 1, 1], 6) |> sum
end
```

Now let's run the experiment, let's say `100_000` times, and see how many times we will fail to reject H_0. For that we will need the following helper functions

```
function play6tennisGamesGetPvalue()::Float64
    # result when HA is true
    result::Int = getResultOf6TennisGamesUnderHA()
    # probability based on which we may decide to reject H0
    oneTailPval::Float64 = Dsts.pdf.(Dsts.Binomial(6, 0.5), result:6) |>
sum
    return oneTailPval
end

function didFailToRejectH0(pVal::Float64)::Bool
    return pVal > 0.05
end
```

In `play6tennisGamesGetPvalue` we conduct an experiment and get a p-value (probability of type 1 error). First we get the result of the experiment under H_A, i.e we assume the true probability of Peter winning a game with John to be `5/6` = 0.8333. We assign the result of those 6 games to a variable `result`. Next we calculate the probability

of obtaining such a result by chance under H_0, i.e. probability of Peter winning is `1/2` = 0.5 as we did in Section 4.9.3 . We return that probability.

Previously we said that the accepted cutoff level for alpha is 0.05 (see Section 4.7.6). If p-value ≤ 0.05 we reject H_0 and choose H_A. Here for β we need to know whether we fail to reject H_0 hence `didFailToRejectH0` function with `pVal > 0.05`.

And now, we can go to the promised `100_000` simulations.

```
numOfSimul = 100_000
Rand.seed!(321)
notRejectedH0 = [
    didFailToRejectH0(play6tennisGamesGetPvalue()) for _ in 1:numOfSimul
    ]
probOfType2error = sum(notRejectedH0) / length(notRejectedH0)
```

0.66384

We run our experiment `100_000` times and record whether we failed to reject H_0. We put that to `notRejectedH0` using comprehensions (see Section 3.6.3). We get a vector of `Bool`s (e.g. `[true, false, true]`). When used with `sum` function Julia treats `true` as `1` and `false` as `0`. We can use that to get the average of `true` (fraction of times we failed to reject H_0). This is the probability of type II error, it is equal to 0.66384. We can use it to calculate the power of a test (power = 1 - β).

```
function getPower(beta::Float64)::Float64
    @assert (0 <= beta <= 1) "beta must be in range [0-1]"
    return 1 - beta
end
powerOfTest = getPower(probOfType2error)

powerOfTest
```

0.33616

Finally, we get our results. We can compare them with the cutoff values from Section 4.7.6 , e.g. $\beta \leq 0.2$, $power \geq 0.8$. So it turns out that if in reality Peter is a better tennis player than John (and on average wins with the ratio 5:1) then we will be able to confirm that

roughly in 3 experiments out of 10 (experiment - the result of 6 games that they play with each other). This is because the power of a test should be ≥ 0.8 (accepted by statisticians), but it is 0.33616 (estimated in our computer simulation). Here we can either say that they both (John and Peter) play equally well (we did not reject H_0) or make them play a greater number of games with each other in order to confirm that Peter consistently wins with John on average 5 to 1.

If you want to see a graphical representation of the solution to exercise 5 take a look at the figure below.

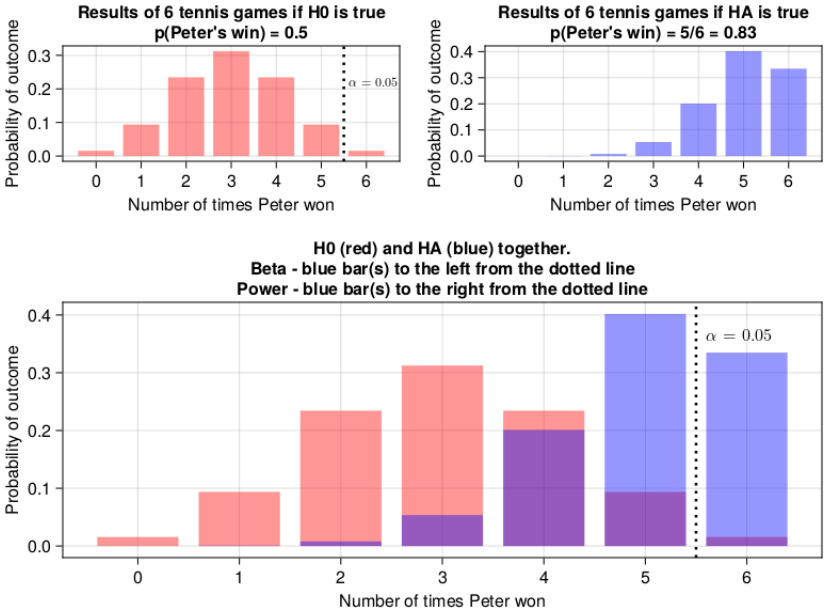

Figure 11: Figure 10: Graphical representation of the estimation process for type II error and the power of a test.

The top panels display the probability distributions for our experiment (6 games of tennis) under H_0 (red bars) and H_A (blue bars). Notice, that the blue bars for 0, 1, and 2 are so small that they are barely (or not at all) visible on the graph. The black dotted vertical line is a cutoff level for type I error (or α), which is 0.05. The bottom panel contains the distributions superimposed one on the other. The probability of

type II error (or β) is the sum of the heights of the blue bar(s) to the left from the black dotted vertical line (the cutoff level for type I error). The power of a test is the sum of the heights of the blue bar(s) to the right from the black dotted vertical line (the cutoff level for type I error).

Hopefully the explanations above were clear enough. Still, the presented solution got a few flaws, i.e. we hard coded 6 into our functions (e.g. `getResultOf1TennisGameUnderHA`, `play6tennisGamesGetPvalue`), moreover running `100_000` simulations is probably less efficient than running purely mathematical calculations. Let's try to add some plasticity and efficiency to our code (plus let's check the accuracy of our computer simulation).

```
# to the right from that point on x-axis (>point) we reject H0 and
choose HA
# n - number of trials (games)
function getXForBinomRightTailProb(n::Int, probH0::Float64,
                                   rightTailProb::Float64)::Int
    @assert (0 <= rightTailProb <= 1) "rightTailProb must be in range
[0-1]"
    @assert (0 <= probH0 <= 1) "probH0 must be in range [0-1]"
    @assert (n > 0) "n must be positive"
    return Dsts.cquantile(Dsts.Binomial(n, probH0), rightTailProb)
end

# n - number of trials (games), x - number of successes (Peter's wins)
# returns probability (under HA) from far left up to (and including) x
function getBetaForBinomialHA(n::Int, x::Int, probHA::Float64)::Float64
    @assert (0 <= probHA <= 1) "probHA must be in range [0-1]"
    @assert (n > 0) "n must be positive"
    @assert (x >= 0) "x mustn't be negative"
    return Dsts.cdf(Dsts.Binomial(n, probHA), x)
end
```

> ***Note:*** The above functions should work correctly if probH0 < probHA, i.e. the probability distribution under H_0 is on the left and the probability distribution under H_A is on the right side of a graph, i.e. the case you see in Figure 10 .

The function `getXForBinomRightTailProb` returns a value (number of Peter's wins, number of successes, value on x-axis in Figure 10) above

which we reject H_0 in favor of H_A (if we feed it with cutoff for α equal to 0.05). Take a look at Figure 10 , it returns the value on x-axis to the right of which the sum of heights of the red bars is lower than the cutoff level for alpha (type I error). It does so by wrapping around Dsts.cquantile[191]function (that runs the necessary mathematical calculations) for us.

Once we get this cutoff point (number of successes, here number of Peter's wins) we can feed it as an input to `getBetaForBinomialHA`. Again, take a look at Figure 10 , it calculates for us the sum of the heights of the blue bars from the far left (0 on x-axis) up-to the previously obtained cutoff point (the height of that bar is also included). Let's see how it works in practice.

```
xCutoff = getXForBinomRightTailProb(6, 0.5, 0.05)
probOfType2error2 = getBetaForBinomialHA(6, xCutoff, 5/6)
powerOfTest2 = getPower(probOfType2error2)

(probOfType2error, probOfType2error2, powerOfTest, powerOfTest2)
```

```
(0.66384, 0.6651020233196159, 0.33616, 0.3348979766803841)
```

They appear to be close enough which indicates that our calculations with the computer simulation were correct.

BONUS

Sample size estimation.

As a bonus to this exercise let's talk about sample sizes.

Notice that after solving this exercise we said that if Peter is actually a better player than John and wins on average 5:1 with his opponent then still, most likely we will not be able to show this with 6 tennis games (`powerOfTest2` = 0.3349). So, if ten such experiments would be conducted around the world for similar Peters and Johns then roughly

[191]https://juliastats.org/Distributions.jl/stable/univariate/#Distributions.cquantile-Tuple%7BUnivariateDistribution,%20Real%7D

only in three of them Peter would be declared a better player after running statistical tests. That doesn't sound right.

In order to overcome this at the onset of their experiment a statistician should also try to determine the proper sample size. First, he starts by asking himself a question: "how big difference will make a difference". This is an arbitrary decision (at least a bit). Still, I think we can all agree that if Peter would win with John on average 99:1 then this would make a practical difference (probably John would not like to play with him, what's the point if he would be still loosing). OK, and how about Peter wins with John on average 51:49. This does not make a practical difference. Here they are pretty well matched and would play with each other since it would be challenging enough for both of them and each one could win a decent amount of games to remain satisfied. Most likely, they would be even unaware of such a small difference.

In real life a physician could say, e.g. "I'm going to test a new drug that should reduce the level of 'bad cholesterol' (LDL-C[192]). How big reduction would I like to detect? Hmm, I know, 30 [mg/dL] or more because it reduces the risk of a heart attack by 50%" or "By at least 25 [mg/dL] because the drug that is already on the market reduces it by 25 [mg/dL]" (the numbers were made up by me, I'm not a physician).

Anyway, once a statistician gets the difference that makes a difference he tries to estimate the sample size by making some reasonable assumptions about rest of the parameters.

In our tennis example we could write the following function for sample size estimation

```
# checks sample sizes between start and finish (inclusive, inclusive)
# assumes that probH0 is 0.5
function getSampleSizeBinomial(probHA::Float64,
    cutoffBeta::Float64=0.2,
    cutoffAlpha::Float64=0.05,
    twoTail::Bool=true,
    start::Int=6, finish::Int=40)::Int
```

[192]https://en.wikipedia.org/wiki/Low-density_lipoprotein

```
    # other probs are asserted in the component functions that use them
    @assert (0 <= cutoffBeta <= 1) "cutoffBeta must be in range [0-1]"
    @assert (start > 0 && finish > 0) "start and finish must be
positive"
    @assert (start < finish) "start must be smaller than finish"

    probH0::Float64 = 0.5
    sampleSize::Int = -99
    xCutoffForAlpha::Int = 0
    beta::Float64 = 1.0

    if probH0 >= probHA
        probHA = 1 - probHA
    end
    if twoTail
        cutoffAlpha = cutoffAlpha / 2
    end

    for n in start:finish
        xCutoffForAlpha = getXForBinomRightTailProb(n, probH0,
cutoffAlpha)
        beta = getBetaForBinomialHA(n, xCutoffForAlpha, probHA)
        if beta <= cutoffBeta
            sampleSize = n
            break
        end
    end

    return sampleSize
end
```

Maybe that is not the most efficient method, but it should do the trick.

First, we initialize a few variables that we will use later (`probH0`, `sampleSize`, `xCutoffForAlpha`, `beta`). Then we compare `probH0` with `probHA`. We do this since `getXForBinomRightTailProb` and `getBetaForBinomialHA` should work correctly only when `probH0` < `probHA` (see the note under the code snippet with the functions definitions). Therefore we need to deal with the case when it is otherwise (`if probH0 >= probHA`). We do this by subtracting `probHA` from 1 and making it our new `probHA` (`probHA = 1 - probHA`). Because of that if we ever type, e.g. `probHA` = 1/6 = 0.166, then the function will transform it to `probHA` = 1 - 1/6 = 5/6 = 0.833 (since in our case the sample size required to demonstrate that Peter wins on

average 1 out of 6 games, is the same as the sample size required to show that John wins on average 5 out of 6 games).

Once we are done with that we go to another checkup. If we are interested in two-tailed probability (`twoTail`) then we divide the number (`cutoffAlpha = 0.05`) by two. Before 0.05 went to the right side (see the black dotted line in Figure 10), now we split it, 0.025 goes to the left side, 0.025 goes to the right side of the probability distribution. This makes sense since before (see Section 4.7.4) we multiplied one-tailed probability by 2 to get the two-tailed probability, here we do the opposite. We can do that because the probability distribution under H_0 (see the upper left panel in Figure 10) is symmetrical (that is why you mustn't change the value of `probH0` in the body of `getSampleSizeBinomial`).

Finally, we use the previously defined functions (`getXForBinomRightTailProb` and `getBetaForBinomialHA`) and conduct a series of experiments for different sample sizes (between `start` and `finish`). Once the obtained `beta` fulfills the requirement (`beta <= cutoffBeta`) we set `sampleSize` to that value (`sampleSize = n`) and stop subsequent search with a `break` statement (so if `sampleSize` of 6 is OK, we will not look at larger sample sizes). If the `for` loop terminates without satisfying our requirements then the value of `-99` (`sampleSize` was initialized with it) is returned. This is an impossible value for a sample size. Therefore it points out that the search failed. Let's put it to the test.

In this exercise we said that Peter wins with John on average 5:1 (H_A, prob = 5/6 = 0.83). So what is the sample size necessary to confirm that with the acceptable type I error ($alpha \leq 0.05$) and type II error ($\beta \leq 0.2$) cutoffs.

```
# for one-tailed test
sampleSizeHA5to1 = getSampleSizeBinomial(5/6, 0.2, 0.05, false)
sampleSizeHA5to1
```

13

OK, so in order to be able to detect such a big difference (5:1, or even bigger) between the two tennis players they would have to play 13 games with each other (for one-tailed test). To put it into perspective and compare it with Figure 10 look at the graph below.

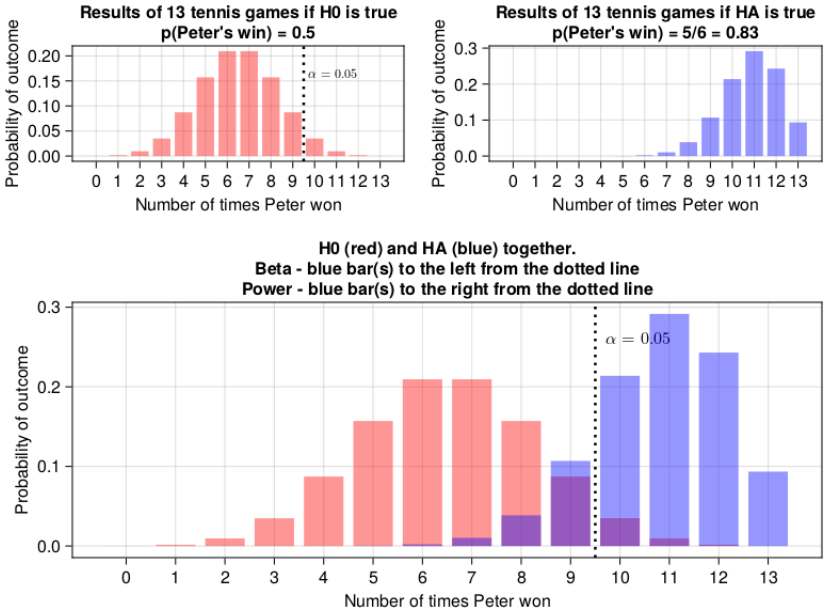

Figure 12: Figure 11: Graphical representation of type II error and the power of a test for 13 tennis games between Peter and John.

If our function worked well then the sum of the heights of the blue bars to the right of the black dotted line should be ≥ 0.8 (power of the test) and to the left should be ≤ 0.2 (type II error or β).

```
(
    # alternative to the line below:
    # 1 - Dsts.cdf(Dsts.Binomial(13, 5/6), 9),
    Dsts.pdf.(Dsts.Binomial(13, 5/6), 10:13) |> sum,
    Dsts.cdf(Dsts.Binomial(13, 5/6), 9)
)
```

```
(0.841922621916511, 0.15807737808348937)
```

Yep, that's correct. So, under those assumptions in order to confirm that Peter is a better tennis player he would have to win ≥ 10 games out of 13.

And how about the two-tailed probability (we expect the number of games to be greater).

```
# for two-tailed test
getSampleSizeBinomial(5/6, 0.2, 0.05)
```

17

Here we need 17 games to be sufficiently sure we can prove Peter's supremacy.

OK. Let's give our `getSampleSizeBinomial` one more swing. How about if Peter wins with John on average 4:2 (H_A)?

```
# for two-tailed test
sampleSizeHA4to2 = getSampleSizeBinomial(4/6, 0.2, 0.05)
sampleSizeHA4to2
```

-99

Hmm, `-99`, so it will take more than 40 games (`finish::Int = 40`). Now, we can either stop here (since playing 40 games in a row is too time and energy consuming so we resign) or increase the value for `finish` like so

```
# for two-tailed test
sampleSizeHA4to2 = getSampleSizeBinomial(4/6, 0.2, 0.05, true, 6, 100)
sampleSizeHA4to2
```

72

Wow, if Peter is better than John in tennis and on average wins 4:2 then it would take 72 games to be sufficiently sure to prove it (who would have thought).

Anyway, if you ever find yourself in need to determine sample size, β or the power of a test (not only for one-sided tests as we did here) then

you should probably consider using PowerAnalyses.jl[193]which is on MIT[194]license.

OK, I think you deserve some rest before moving to the next chapter so why won't you take it now.

[193]https://github.com/rikhuijzer/PowerAnalyses.jl
[194]https://en.wikipedia.org/wiki/MIT_License

Comparisons - continuous data

OK, we finished the previous chapter with hypothesis testing and calculating probabilities for binomial data (`bi` - two `nomen` - name), e.g. number of successes (wins of Peter in tennis).

In this chapter we are going to explore comparisons between the groups containing data on a continuous scale (like the height from Section 4.6).

Chapter imports

Later in this chapter we are going to use the following libraries

```
import CairoMakie as Cmk
import CSV as Csv
import DataFrames as Dfs
import Distributions as Dsts
import HypothesisTests as Ht
import MultipleTesting as Mt
import Random as Rand
import Statistics as Stats
```

If you want to follow along you should have them installed on your system. A reminder of how to deal (install and such) with packages can be found here[195]. But wait, you may prefer to use `Project.toml` and `Manifest.toml` files from the code snippets for this chapter[196]to install the required packages. The instructions you will find here[197].

The imports will be placed in the code snippet when first used, but I thought it is a good idea to put them here, after all imports should be at the top of your file (so here they are at the top of the chapter). Moreover, that way they will be easier to find all in one place.

If during the lecture of this chapter you find a piece of code of unknown functionality, just go to the code snippets mentioned above and run the code from the `*.jl` file. Once you have done that you can

[195]https://docs.julialang.org/en/v1/stdlib/Pkg/
[196]https://github.com/b-lukaszuk/RJ_BS_eng/tree/main/code_snippets/ch05
[197]https://pkgdocs.julialang.org/v1/environments/

always extract a small piece of it and test it separately (modify and experiment with it if you wish).

One sample Student's t-test

Imagine that in your town there is a small local brewery that produces quite expensive but super tasty beer. You like it a lot, but you got an impression that the producer is not being honest with their customers and instead of the declared 500 [mL] of beer per bottle, he pours a bit less. Still, there is little you can do to prove it. Or can you?

You bought 10 bottles of beer (ouch, that was expensive!) and measured the volume of fluid in each of them. The results are as follows

```
# a representative sample
beerVolumes = [504, 477, 484, 476, 519, 481, 453, 485, 487, 501]
```

On a graph the volume distribution looks like this (it was drawn with Cmk.hist[198]function).

[198]https://docs.makie.org/stable/examples/plotting_functions/hist/index.html#hist

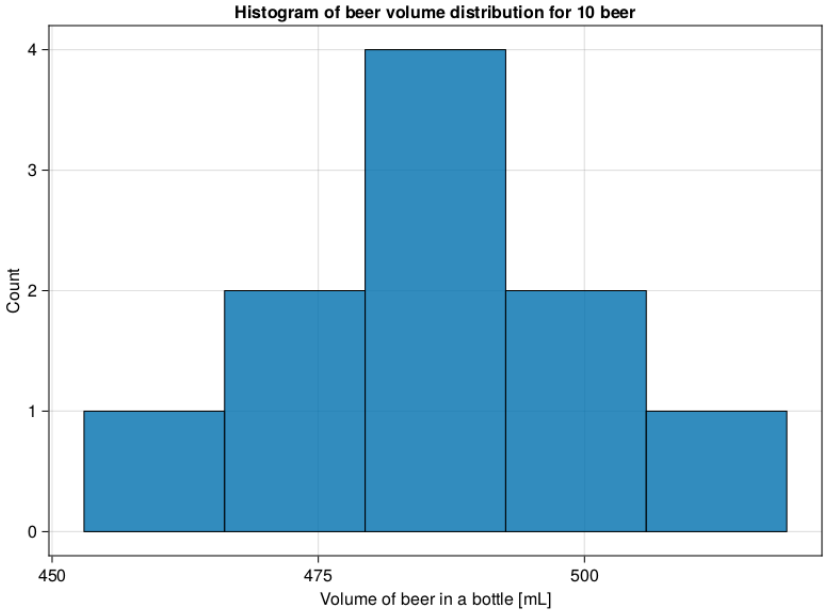

Figure 13: Figure 12: Histogram of beer volume distribution for 10 beer (fictitious data).

You look at it and it seems to resemble a bit the bell shaped curve that we discussed in the Section 4.6 . This makes sense. Imagine your task is to pour let's say 1'000 bottles daily with 500 [mL] of beer in each with a big mug (there is an erasable mark at a bottle's neck). Most likely the volumes would oscillate around your goal volume of 500 [mL], but they would not be exact. Sometimes in a hurry you would add a bit more, sometimes a bit less (you could not waste time to correct it). So it seems like a reasonable assumption that the 1'000 bottles from our example would have a roughly bell shaped (aka normal) distribution of volumes around the mean.

Now you can calculate the mean and standard deviation for the data

```
import Statistics as Stats

meanBeerVol = Stats.mean(beerVolumes)
stdBeerVol = Stats.std(beerVolumes)
```

```
(meanBeerVol, stdBeerVol)
```

```
(486.7, 18.055777776410274)
```

Hmm, on average there was 486.7 [mL] of beer per bottle, but the spread of the data around the mean is also considerable (sd = 18.06 [mL]). The lowest value measured was 453 [mL], the highest value measured was 519 [mL]. Still, it seems that there is less beer per bottle than expected, but is it enough to draw a conclusion that the real mean in the population of our 1'000 bottles is ≈ 487.0 [mL] and not 500 [mL] as it should be? Let's try to test that using what we already know about the normal distribution (see Section 4.6), the three sigma rule (Section 4.6.1) and the `Distributions` package (Section 4.6.2).

Let's assume for a moment that the true mean for volume of fluid in the population of 1'000 beer bottles is `meanBeerVol` = 486.7 [mL] and the true standard deviation is `stdBeerVol` = 18.06 [mL]. That would be great because now, based on what we've learned in Section 4.6.2 we can calculate the probability that a random bottle of beer got >500 [mL] of fluid (or % of beer bottles in the population that contain >500 [mL] of fluid). Let's do it

```
import Distributions as Dsts

# how many std. devs is value above or below the mean
function getZScore(value::Real, mean::Real, sd::Real)::Float64
    return (value - mean)/sd
end

expectedBeerVolmL = 500

fractionBeerLessEq500mL = Dsts.cdf(Dsts.Normal(),
    getZScore(expectedBeerVolmL, meanBeerVol, stdBeerVol))
fractionBeerAbove500mL = 1 - fractionBeerLessEq500mL

fractionBeerAbove500mL
```

0.2306808956300721

I'm not going to explain the code above since for reference you can always check Section 4.6.2 . Still, under those assumptions roughly 0.23 or 23% of beer bottles contain more than 500 [mL] of fluid. In other words under these assumptions the probability that a random beer bottle contains >500 [mL] of fluid is 0.23 or 23%.

There are 2 problems with that solution.

Problem 1

It is true that the mean from the sample is our best estimate of the mean in the population (here 1'000 beer bottles poured daily). However, statisticians proved that instead of the standard deviation from our sample we should use the standard error of the mean[199]. It describes the spread of sample means around the true population mean and it can be calculated as follows

$sem = \frac{sd}{\sqrt{n}}$, where

sem - standard error of the mean

sd - standard deviation

n - number of observations in the sample

Let's enclose it into Julia code

```
function getSem(vect::Vector{<:Real})::Float64
    return Stats.std(vect) / sqrt(length(vect))
end
```

Now we get a better estimate of the probability

```
fractionBeerLessEq500mL = Dsts.cdf(Dsts.Normal(),
    getZScore(expectedBeerVolmL, meanBeerVol, getSem(beerVolumes)))
fractionBeerAbove500mL = 1 - fractionBeerLessEq500mL

fractionBeerAbove500mL
```

0.00992016769999493

[199]https://en.wikipedia.org/wiki/Standard_error

Under those assumptions the probability that a beer bottle contains >500 [mL] of fluid is roughly 0.01 or 1%.

So, to sum up. Here, we assumed that the true mean in the population is our sample mean (μ = `meanBeerVol`). Next, if we were to take many small samples like `beerVolumes` and calculate their means then they would be normally distributed around the population mean (here μ = `meanBeerVol`) with σ (standard deviation in the population) = `getSem(beerVolumes)`. Finally, using the three sigma rule (see Section 4.6.1) we check if our hypothesized mean (`expectedBeerVolmL`) lies within roughly 2 standard deviations (here approximately 2 `sems`) from the assumed population mean (here μ = `meanBeerVol`).

Problem 2

The sample size is small (`length(beerVolumes)` = 10) so the underlying distribution is quasi-normal (*quasi* - almost, as it were). It is called a t-distribution[200](for comparison of an exemplary normal and t-distribution see the figure below). Therefore to get a better estimate of the probability we should use a t-distribution.

[200]https://en.wikipedia.org/wiki/Student%27s_t-distribution

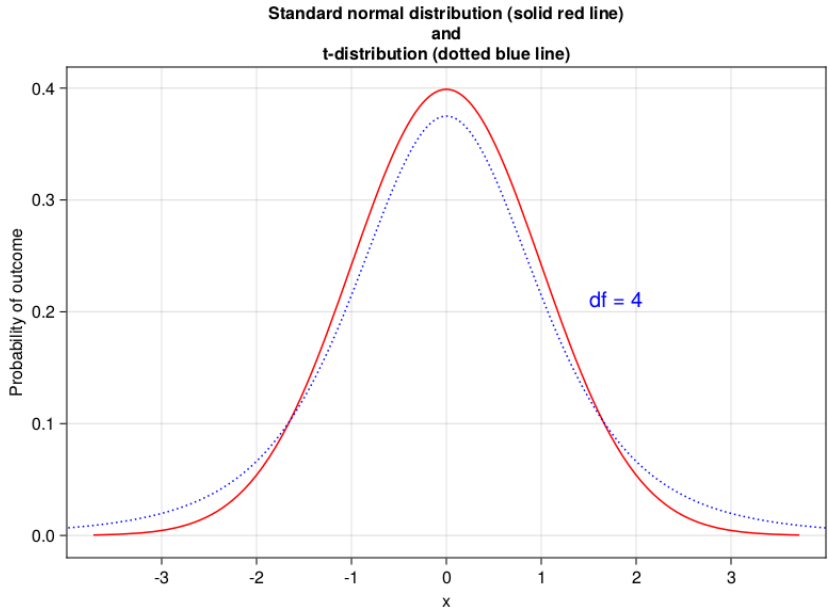

Figure 14: Figure 13: Comparison of normal and t-distribution with 4 degrees of freedom (df = 4).

Luckily our `Distributions` package got the t-distribution included (see the docs[201]). As you remember the normal distribution required two parameters that described it: the mean and the standard deviation. The t-distribution requires only the degrees of freedom[202]. The concept is fairly easy to understand. Imagine that we recorded body masses of 3 people in the room: Paul, Peter, and John.

```
peopleBodyMassesKg = [84, 94, 78]

sum(peopleBodyMassesKg)
```

256

As you can see the sum of those body masses is 256 [kg]. Notice, however, that only two of those masses are independent or free to

[201] https://juliastats.org/Distributions.jl/stable/univariate/#Distributions.TDist
[202] https://en.wikipedia.org/wiki/Degrees_of_freedom_(statistics)

change. Once we know any two of the body masses (e.g. 94, 78) and the sum: 256, then the third body mass must be equal to `sum(peopleBodyMassesKg) - 94 - 78` = 84 (it is determined, it cannot just freely take any value). So in order to calculate the degrees of freedom we type `length(peopleBodyMassesKg) - 1` = 2. Since our sample size is equal to `length(beerVolumes)` = 10 then it will follow a t-distribution with `length(beerVolumes) - 1` = 9 degrees of freedom.

So the probability that a beer bottle contains >500 [mL] of fluid is

```
function getDf(vect::Vector{<:Real})::Int
    return length(vect) - 1
end

fractionBeerLessEq500mL = Dsts.cdf(Dsts.TDist(getDf(beerVolumes)),
    getZScore(expectedBeerVolmL, meanBeerVol, getSem(beerVolumes)))
fractionBeerAbove500mL = 1 - fractionBeerLessEq500mL

fractionBeerAbove500mL
```

0.022397253591088906

> ***Note:*** The z-score (number of standard deviations above or below the mean) for a t-distribution is called the t-score or t-statistics (it is calculated with sem instead of sd).

Finally, we got the result. Based on our representative sample (`beerVolumes`) and the assumptions we made we can see that the probability that a random beer contains >500 [mL] of fluid (500 [mL] is stated on a label) is `fractionBeerAbove500mL` = 0.022 or 2.2% (remember, this is one-tailed probability, the two-tailed probability is 0.022 * 2 = 0.044 = 4.4%).

Given that the cutoff level for α (type I error) from Section 4.7.5 is 0.05 we can reject our H_0 (the assumption that 500 [mL] comes from the population with the mean approximated by μ = `meanBeerVol` = 486.7 [mL] and the standard deviation approximated by σ = `sem` = 5.71 [mL]).

In conclusion, our hunch was right ("...you got an impression that the producer is not being honest with their customers..."). The owner of the local brewery is dishonest and intentionally pours slightly less beer (on average `expectedBeerVolmL - meanBeerVol` = 13.0 [mL]). Now we can go to him and get our money back, or alarm the proper authorities for that monstrous crime. *Fun fact: the story has it that the code of Hammurabi*[203]*(circa 1750 BC) was the first to punish for diluting a beer with water (although it seems to be more of a legend).* Still, this is like 2-3% beer (≈13/500 = 0.026) in a bottle less than it should be and the two-tailed probability (`fractionBeerAbove500mL * 2` = 0.045) is not much less than the cutoff for type 1 error equal to 0.05 (we may want to collect a bigger sample and change the cutoff to 0.01).

HypothesisTests package

The above paragraphs were to further your understanding of the topic. In practice you can do this much faster using HypothesisTests[204] package.

In our beer example you could go with this short snippet (see the docs[205] for `Ht.OneSampleTTest`)

```
import HypothesisTests as Ht

Ht.OneSampleTTest(beerVolumes, expectedBeerVolmL)
```

```
One sample t-test
-----------------
Population details:
    parameter of interest:   Mean
    value under h_0:         500
    point estimate:          486.7
    95% confidence interval: (473.8, 499.6)

Test summary:
    outcome with 95% confidence: reject h_0
    two-sided p-value:           0.0448
```

[203] https://en.wikipedia.org/wiki/Code_of_Hammurabi
[204] https://juliastats.org/HypothesisTests.jl/stable/
[205] https://juliastats.org/HypothesisTests.jl/stable/parametric/#t-test

```
Details:
    number of observations:   10
    t-statistic:              -2.329353706113303
    degrees of freedom:       9
    empirical standard error: 5.70973826993069
```

Let's compare it with our previous results

```
(
expectedBeerVolmL, # value under h_0
meanBeerVol, # point estimate
fractionBeerAbove500mL * 2, # two-sided p-value
getZScore(expectedBeerVolmL, meanBeerVol, getSem(beerVolumes)),# t-
statistic
getDf(beerVolumes), # degrees of freedom
getSem(beerVolumes) # empirical standard error
)
```

```
(500, 486.7, 0.04479450718217781, 2.329353706113303, 9,
5.70973826993069)
```

The numbers are pretty much the same (and they should be if the previous explanation was right). The t-statistic is positive in our case because `getZScore` subtracts `mean` from `value` (`value - mean`) and some packages (like `HypothesisTests`) swap the numbers.

The value that needs to be additionally explained is the 95% confidence interval[206]from the output of `HypothesisTests` above. All it means is that: if we were to run our experiment with 10 beers 100 times and calculate 95% confidence intervals 100 times then 95 of the intervals would contain the true mean from the population. Sometimes people (over?)simplify it and say that this interval [in our case (473.8, 499.6)] contains the true mean from the population with probability of 95% (but that isn't necessarily the same what was stated in the previous sentence). The narrower interval means better, more precise estimate. If the difference is statistically significant (p-value ≤ 0.05) then the interval should not contain the postulated mean (as it is in our case).

[206]https://en.wikipedia.org/wiki/Confidence_interval

Notice that the obtained 95% confidence interval (473.8, 499.6) may indicate that the true average volume of fluid in a bottle of beer could be as high as 499.6 [mL] (so this would hardly make a practical difference) or as low as 473.8 [mL] (a small, ~6%, but a practical difference). In the case of our beer example it is just a curious fact, but imagine you are testing a new drug lowering the 'bad cholesterol' (LDL-C) level (the one that was mentioned in Section 4.9.5). Let's say you got a 95% confidence interval for the reduction of (-132, +2). The interval encompasses 0, so the true effect may be 0 and you cannot reject H_0 under those assumptions (p-value would be greater than 0.05). However, the interval is broad, and its lower value is −132, which means that the true reduction level after applying this drug could be even −132 [mg/dL]. Based on the data from this table[207] I guess this could have a big therapeutic meaning. So, you might want to consider performing another experiment on the effects of the drug, but this time you should take a bigger sample to dispel the doubt (bigger sample size narrows the 95% confidence interval).

In general one sample t-test is used to check if a sample comes from a population with the postulated mean (in our case in H_0 the postulated mean was 500 [mL]). However, I prefer to look at it from the different perspective (the other end) hence my explanation above. The t-test is named after William Sealy Gosset[208] that published his papers under the pen-name Student, hence it is also called a Student's t-test.

Checking the assumptions

Hopefully, the explanations above were clear enough. Still, we shouldn't just jump into performing a test blindly, first we should test its assumptions (see figure below).

[207] https://en.wikipedia.org/wiki/Low-density_lipoprotein#Normal_ranges
[208] https://en.wikipedia.org/wiki/William_Sealy_Gosset

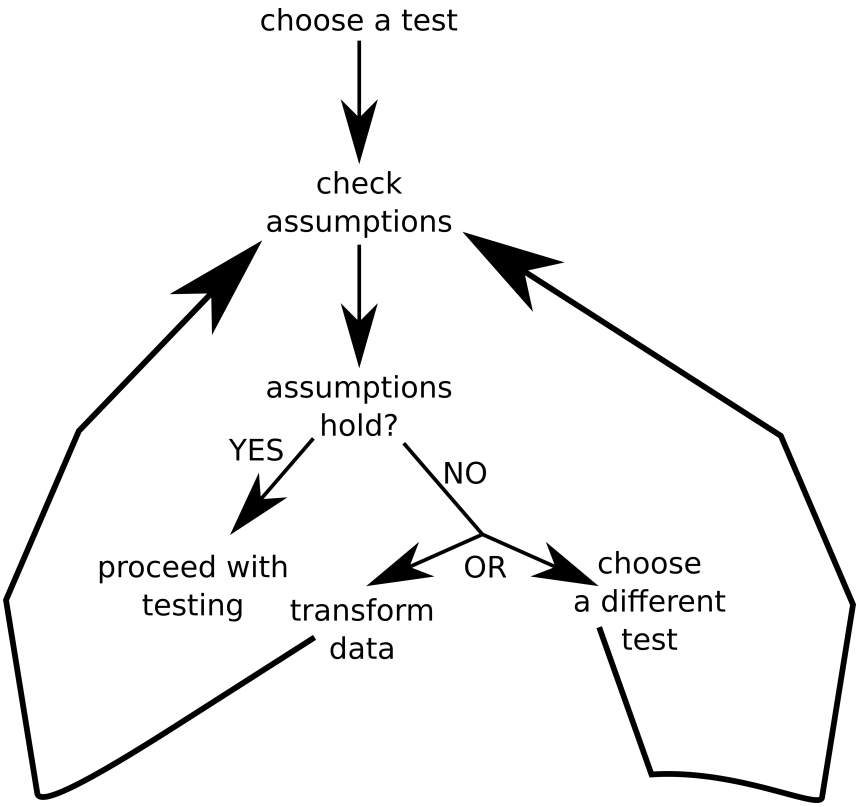

Figure 15: Figure 14: Checking assumptions of a statistical test before running it.

First of all we start by choosing a test to perform. Usually it is a parametric test[209], i.e. one that assumes some specific data distribution (e.g. normal). Then we check our assumptions. If they hold we proceed with our test. Otherwise we can either transform the data (e.g. take a logarithm from each value) or choose a different test (the one that got different assumptions or just less of them to fulfill). We will see an example of a data transformation, and the possible benefits it can bring us, later in this book (see the upcoming Section 7.8.1). Anyway, this different test usually belongs to so called non-parametric tests[210], i.e. tests that make less assumptions about the data, but are likely to be

[209] https://en.wikipedia.org/wiki/Parametric_statistics

[210] https://en.wikipedia.org/wiki/Nonparametric_statistics

slightly less powerful (you remember the power of a test from Section 4.7.5 , right?).

In our case a Student's t-test requires (among others[211]) the data to be normally distributed. This is usually verified with Shapiro-Wilk test[212] or Kolmogorov-Smirnov test[213]. As an alternative to Student's t-test (when the normality assumption does not hold) a Wilcoxon test[214]is often performed (of course before you use it you should check its assumptions, see Figure 14 above).

Both Kolmogorov-Smirnov (see this docs[215]) and Wilcoxon test (see that docs[216]) are at our disposal in `HypothesisTests` package. Behold

```
Ht.ExactOneSampleKSTest(beerVolumes,
    Dsts.Normal(meanBeerVol, stdBeerVol))
```

```
Exact one sample Kolmogorov-Smirnov test
----------------------------------------
Population details:
    parameter of interest:   Supremum of CDF differences
    value under h_0:         0.0
    point estimate:          0.193372

Test summary:
    outcome with 95% confidence: fail to reject h_0
    two-sided p-value:           0.7826

Details:
    number of observations:   10
```

So it seems we got no grounds to reject the H_0 that states that our data are normally distributed (p-value > 0.05) and we were right to perform our one-sample Student's t-test. Of course, I had checked the assumption before I conducted the test (`Ht.OneSampleTTest`). I didn't

[211]https://en.wikipedia.org/wiki/Student%27s_t-test#Assumptions

[212]https://en.wikipedia.org/wiki/Shapiro%E2%80%93Wilk_test

[213]https://en.wikipedia.org/wiki/Kolmogorov%E2%80%93Smirnov_test

[214]https://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test

[215]https://juliastats.org/HypothesisTests.jl/stable/nonparametric/#Kolmogorov-Smirnov-test

[216]https://juliastats.org/HypothesisTests.jl/stable/nonparametric/#Wilcoxon-signed-rank-test

mention it there because I didn't want to prolong my explanation (and diverge from the topic) back there.

And now a question. Is the boring assumption check before a statistical test really necessary?

If you want your conclusions to reflect the reality well then yes. So, even though a statistical textbook for brevity may not check the assumptions of a method you should do it in your analyses if your care about the correctness of your judgment.

Two samples Student's t-test

Imagine a friend that studies biology told you that he had conducted a research in order to write a dissertation and earn a master's degree[217]. As part of the research he tested a new drug (drug X) on mice. He hopes the drug is capable to reduce the body weights of the animals (and if so, then in a distant future it might be even tested on humans). He asks you for help with the data analysis. The results obtained by him are as follows.

```
import CSV as Csv
import DataFrames as Dfs

# if you are in 'code_snippets' folder, then use: "./ch05/miceBwt.csv"
# if you are in 'ch05' folder, then use: "./miceBwt.csv"
miceBwt = Csv.read("./code_snippets/ch05/miceBwt.csv", Dfs.DataFrame)
first(miceBwt, 3)
```

noDrugX	drugX
26	26
26	25
24	25

Table 1: Table 1: Body mass [g] of mice (fictitious data).

> ***Note:*** The path specification above should work fine on GNU/Linux operating systems. I don't know about other OSs.

[217] https://en.wikipedia.org/wiki/Master_of_Science

Here, we opened a table with a made up data for mice body weight [g] (this data set can be found here[218]). For that we used two new packages (CSV[219], and DataFrames[220]).

A `*.csv` file can be opened and created, e.g. with a spreadsheet[221] program. Here, we read it as a `DataFrame`, i.e. a structure that resembles an array from Section 3.3.7 . Since the `DataFrame` could potentially have thousands of rows we displayed only the first three (to check that everything succeeded) using the `first` function.

> ***Note:*** We can check the size of a `DataFrame` with `size` function which returns the information in a friendly `(numRows, numCols)` format.

OK, let's take a look at some descriptive statistics using describe[222] function.

```
Dfs.describe(miceBwt)
```

variable	mean	min	median	max	nmissing	eltype
noDrugX	25.5	23	25.5	29	0	Int64
drugX	24.1	21	24.5	26	0	Int64

Table 2: Table 2: Body mass of mice. Descriptive statistics.

It appears that mice from group `drugX` got somewhat lower body weight. But that could be just a coincidence. Anyway, how should we analyze this data? Well, it depends on the experiment design.

Since we have 10 rows (`size(miceBwt)[1]`). Then, either:

- we had 10 mice at the beginning. The mice were numbered randomly 1:10 on their tails. Then we measured their initial weight

[218] https://github.com/b-lukaszuk/RJ_BS_eng/tree/main/code_snippets/ch05
[219] https://csv.juliadata.org/stable/
[220] https://dataframes.juliadata.org/stable/
[221] https://en.wikipedia.org/wiki/List_of_spreadsheet_software
[222] https://dataframes.juliadata.org/stable/lib/functions/#DataAPI.describe

(noDrugX), administered the drug and measured their body weight after, e.g. one week (drugX), or
- we had 20 mice at the beginning. The mice were numbered randomly 1:20 on their tails. Then first 10 of them (numbers 1:10) became controls (regular food, group: noDrugX) and the other 10 (11:20) received additionally drugX (hence group drugX).

Interestingly, the experimental models deserve slightly different statistical methodology. In the first case we will perform a paired samples t-test, whereas in the other case we will use an unpaired samples t-test. Ready, let's go.

Paired samples Student's t-test

Running a paired Student's t-test with HypothesisTests package is very simple. We just have to send the specific column(s) to the appropriate function. Column selection can be done in one of the few ways, e.g. miceBwt[:, "noDrugX"] (similarly to array indexing in Section 3.3.7 : means all rows, note that this form copies the column), miceBwt[!, "noDrugX"] (! instead of :, no copying), miceBwt.noDrugX (again, no copying).

> ***Note:*** Copying a column is advantageous when a function may modify the input data, but it is less effective for big data frames. If you wonder does a function changes its input then for starter look at its name and compare it with the convention we discussed in Section 3.4.4 . Still, to be sure you would have to examine the function's code.

And now we can finally run the paired t-test.

```
# miceBwt.noDrugX or miceBwt.noDrugX returns a column as a Vector
Ht.OneSampleTTest(miceBwt.noDrugX, miceBwt.drugX)
```

```
One sample t-test
-----------------
Population details:
    parameter of interest:   Mean
```

```
    value under h_0:         0
    point estimate:          1.4
    95% confidence interval: (0.04271, 2.757)

Test summary:
    outcome with 95% confidence: reject h_0
    two-sided p-value:           0.0445

Details:
    number of observations:   10
    t-statistic:              2.3333333333333335
    degrees of freedom:       9
    empirical standard error: 0.6
```

And voila. We got the result. It seems that `drugX` actually does lower the body mass of the animals ($p \leq 0.05$). But wait, didn't we want to do a (paired) two-samples t-test and not `OneSampleTTest`? Yes, we did. Interestingly enough, a paired t-test is actually a one-sample t-test for the difference. Observe.

```
# miceBwt.noDrugX or miceBwt.noDrugX returns a column as a Vector
# hence we can do element-wise subtraction using dot syntax
miceBwtDiff = miceBwt.noDrugX .- miceBwt.drugX
Ht.OneSampleTTest(miceBwtDiff)
```

```
One sample t-test
-----------------
Population details:
    parameter of interest:   Mean
    value under h_0:         0
    point estimate:          1.4
    95% confidence interval: (0.04271, 2.757)

Test summary:
    outcome with 95% confidence: reject h_0
    two-sided p-value:           0.0445

Details:
    number of observations:   10
    t-statistic:              2.3333333333333335
    degrees of freedom:       9
    empirical standard error: 0.6
```

Here, we used the familiar dot syntax from Section 3.6.5 to obtain the differences and then fed the result to `OneSampleTTest` from the

previous section (see Section 5.2). The output is the same as in the previous code snippet.

I don't know about you, but when I was a student I often wondered when to choose a paired and when an unpaired t-test. Now I finally know, and it is so simple. Too bad that most statistical programs/packages separate paired t-test from one-sample t-test (unlike the authors of the `HypothesisTests` package).

Anyway, this also demonstrates an important feature of the data. The data points in both columns/groups need to be properly ordered, e.g. in our case it makes little sense to subtract body mass of a mouse with 1 on its tail from a mouse with 5 on its tail, right? Doing so has just as little sense as subtracting it from mouse number 6, 7, 8, etc. There is only one clearly good way to do this subtraction and this is to subtract mouse number 1 (`drugX`) from mouse number 1 (`noDrugX`). So, if you ever wonder a paired or unpaired t-test then think if there is a clearly better way to subtract one column of data from the other. If so, then you should go with the paired t-test, otherwise choose the unpaired t-test.

BTW, do you remember how in Section 5.2.2 we checked the assumptions of our `oneSampleTTest`, well it turns out that here we should do the same. However, this time instead of Kolmogorov-Smirnov test I'm going to use Shapiro-Wilk's normality test from `HypothesisTests` package (generally Shapiro-Wilk is more powerful).

```
Ht.ShapiroWilkTest(miceBwtDiff)
```

```
Shapiro-Wilk normality test
---------------------------
Population details:
    parameter of interest:   Squared correlation of data and expected
order
                             statistics of N(0,1) (W)
    value under h_0:         1.0
    point estimate:          0.94181

Test summary:
    outcome with 95% confidence: fail to reject h_0
```

```
    one-sided p-value:           0.5733

Details:
    number of observations: 10
    censored ratio:         0.0
    W-statistic:            0.94181
```

There, all normal (p > 0.05). So, we were right to perform the test. Still, the order was incorrect, in general you should remember to check the assumptions first and then proceed with the test. In case the normality assumption did not hold we should consider doing a Wilcoxon test[223] (non-parametric test), e.g. like so `Ht.SignedRankTest(df.noDrugX, df.drugX)` or `Ht.SignedRankTest(miceBwtDiff)`. More info on the test can be found in the link above or on the pages of `HypothesisTests` package (see here[224]).

Unpaired samples Student's t-test

OK, now it's time to move to the other experimental model. A reminder, here we discuss the following situation:

- we had 20 mice at the beginning. The mice were numbered randomly 1:20 on their tails. Then first 10 of them (numbers 1:10) became controls (regular food, group: `noDrugX`) and the other 10 (11:20) received additionally `drugX` (hence group `drugX`).

Here we will compare mice `noDrugX` (miceID: 1:10) with mice `drugX` (miceID: 11:20) using an unpaired samples t-test, but this time we will start by checking the assumptions.

First the normality assumption.

```
function getSWtestPval(v::Vector{<:Real})::Float64
    return Ht.ShapiroWilkTest(v) |> Ht.pvalue
end

# for brevity we will extract just the p-values
(
```

[223] https://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test

[224] https://juliastats.org/HypothesisTests.jl/stable/nonparametric/#Wilcoxon-signed-rank-test

```
    getSWtestPval(miceBwt.noDrugX),
    getSWtestPval(miceBwt.drugX)
)
```

```
(0.6833331724399464, 0.3254417851120679)
```

OK, no reason to doubt the normality (p-vals > 0.05). The other assumption that we may test is homogeneity of variance. Homogeneity means that the spread of data around the mean in each group is similar (var(gr1) ≈ var(gr2)). Here, we are going to use Fligner-Killeen[225]test from the `HypothesisTests` package.

```
Ht.FlignerKilleenTest(miceBwt.noDrugX, miceBwt.drugX)
```

```
Fligner-Killeen test
--------------------
Population details:
    parameter of interest:   Variances
    value under h_0:         "all equal"
    point estimate:          NaN

Test summary:
    outcome with 95% confidence: fail to reject h_0
    p-value:                     1.0000

Details:
    number of observations: [10, 10]
    FK statistic:           4.76242e-31
    degrees of freedom:     1
```

Also this time, the assumption is fulfilled (`p-value` > 0.05), and now for the unpaired test.

```
Ht.EqualVarianceTTest(
    miceBwt.noDrugX, miceBwt.drugX)
```

```
Two sample t-test (equal variance)
----------------------------------
```

[225]https://juliastats.org/HypothesisTests.jl/stable/nonparametric/#Fligner-Killeen-test

```
Population details:
    parameter of interest:   Mean difference
    value under h_0:         0
    point estimate:          1.4
    95% confidence interval: (-0.1877, 2.988)

Test summary:
    outcome with 95% confidence: fail to reject h_0
    two-sided p-value:           0.0804

Details:
    number of observations:   [10,10]
    t-statistic:              1.8525405838431677
    degrees of freedom:       18
    empirical standard error: 0.7557189365836423
```

It appears there is not enough evidence to reject the H_0 (the mean difference is equal to 0) on the cutoff level of 0.05. So, how could that be, the means in both groups are still the same, i.e. `Stats.mean(miceBwt.noDrugX)` = 25.5 and `Stats.mean(miceBwt.drugX)` = 24.1, yet we got different results (reject H_0 from paired t-test, not reject H_0 from unpaired t-test). Well, it is because we calculated slightly different things and because using paired samples usually removes some between subjects variability.

In the case of unpaired t-test we:

1. assume that the difference between the means under H_0 is equal to 0.
2. calculate the observed difference between the means, `Stats.mean(miceBwt.noDrugX) - Stats.mean(miceBwt.drugX)` = 1.4.
3. calculate the sem (with a slightly different formula than for the one-sample/paired t-test)
4. obtain the z-score (in case of t-test it is named t-score or t-statistics)
5. calculate the probability for the t-statistics (slightly different calculation of the degrees of freedom)

When compared with the methodology for one-sample t-test from Section 5.2 it differs only with respect to the points 3, 4 and 5 above. Observe. First the functions

```
function getSem(v1::Vector{<:Real}, v2::Vector{<:Real})::Float64
    sem1::Float64 = getSem(v1)
    sem2::Float64 = getSem(v2)
    return sqrt((sem1^2) + (sem2^2))
end

function getDf(v1::Vector{<:Real}, v2::Vector{<:Real})::Int
    return getDf(v1) + getDf(v2)
end
```

There are different formulas for pooled sem (standard error of the mean), but I only managed to remember this one because it reminded me the famous Pythagorean theorem[226], i.e. $c^2 = a^2 + b^2$, so $c = \sqrt{a^2 + b^2}$, that I learned in a primary school. As for the degrees of freedom they are just the sum of the degrees of freedom for each of the vectors. OK, so now the calculations

```
meanDiffBwtH0 = 0
meanDiffBwt = Stats.mean(miceBwt.noDrugX) - Stats.mean(miceBwt.drugX)
pooledSemBwt = getSem(miceBwt.noDrugX, miceBwt.drugX)
zScoreBwt = getZScore(meanDiffBwtH0, meanDiffBwt, pooledSemBwt)
dfBwt = getDf(miceBwt.noDrugX, miceBwt.drugX)
pValBwt = Dsts.cdf(Dsts.TDist(dfBwt), zScoreBwt) * 2
```

And finally the result that you may compare with the output of the unpaired t-test above and the methodology for the one-sample t-test from Section 5.2 .

```
(
    meanDiffBwtH0, # value under h_0
    round(meanDiffBwt, digits = 4), # point estimate
    round(pooledSemBwt, digits = 4), # empirical standard error
    # to get a positive zScore we should have calculated it as:
    # getZScore(meanDiffBwt, meanDiffBwtH0, pooledSemBwt)
    round(zScoreBwt, digits = 4), # t-statistic
    dfBwt, # degrees of freedom
    round(pValBwt, digits=4) # two-sided p-value
)
```

```
(0, 1.4, 0.7557, -1.8525, 18, 0.0804)
```

[226] https://en.wikipedia.org/wiki/Pythagorean_theorem

Amazing. In the case of the unpaired two-sample t-test we use the same methodology and reasoning as we did in the case of the one-sample t-test from Section 5.2 (only functions for `sem` and `df` changed slightly). Given the above I recommend you get back to the section Section 5.2 and make sure you understand the explanations presented there (if you haven't done this already).

As an alternative to our unpaired t-test we should consider `Ht.UnequalVarianceTTest` (if the variances are not equal) or `Ht.MannWhitneyUTest` (if both the normality and homogeneity assumptions do not hold).

One-way ANOVA

One-way ANOVA is a technique to compare two or more groups of continuous data. It allows us to tell if all the groups are alike or not based on the spread of the data around the mean(s).

Let's start with something familiar. Do you still remember our tennis players Peter and John from Section 4.7.1 . Well, guess what, they work at two different biological institutes. The institutes independently test a new weight reducing drug, called drug Y, that is believed to reduce body weight of an animal by roughly 23%. The drug administration is fairly simple. You just dilute it in water and leave it in a cage for mice to drink it.

So both our friends independently run the following experiment: a researcher takes eight mice, writes at random numbers at their tails (1:8), and decides that the mice 1:4 will drink pure water, and the mice 5:8 will drink water with the drug. After a week body weights of all mice are recorded.

As said, Peter and John run the experiments independently not knowing one about the other. After a week Peter noticed that he messed things up and did not give the drug to mice (when diluted the drug is colorless and by accident he took the wrong bottle). It happened, still let's compare the results that were obtained by both our friends.

```
import Random as Rand

# Peter's mice, experiment 1 (ex1)
Rand.seed!(321)
ex1BwtsWater = Rand.rand(Dsts.Normal(25, 3), 4)
ex1BwtsPlacebo = Rand.rand(Dsts.Normal(25, 3), 4)

# John's mice, experiment 2 (ex2)
ex2BwtsWater = Rand.rand(Dsts.Normal(25, 3), 4)
ex2BwtsDrugY = Rand.rand(Dsts.Normal(25 * 0.77, 3), 4)
```

In Peter's case both mice groups came from the same population `Dsts.Normal(25, 3)` ($\mu = 25, \sigma = 3$) since they both ate and drunk the same stuff. For need of different name the other group is named placebo[227].

In John's case the other group comes from a different distribution (e.g. the one where body weight is reduced on average by 23%, hence $\mu = 25*0.77$).

Let's see the results side by side on a graph.

[227] https://en.wikipedia.org/wiki/Placebo

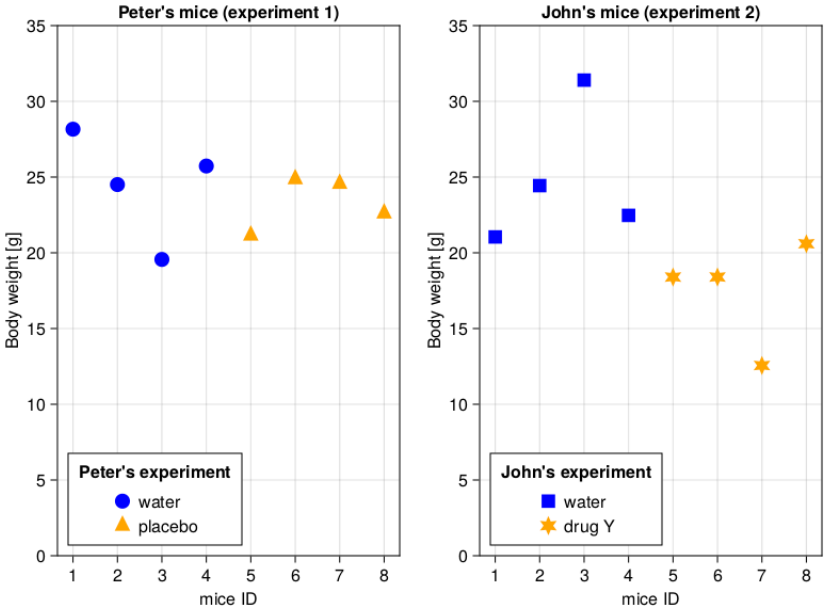

Figure 16: Figure 15: The results of drug Y application on body weights of laboratory mice.

I don't know about you, but my first impression is that the data points are more scattered around in John's experiment. Let's add some means to the graph to make it more obvious.

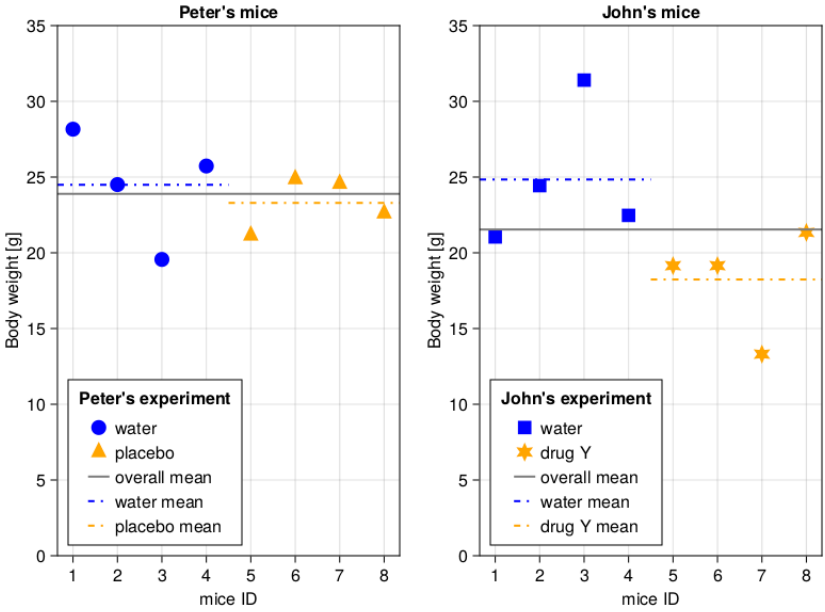

Figure 17: Figure 16: The results of drug Y application on body weights of laboratory mice (with group and overall means).

Indeed, with the lines (especially the overall means) the difference in spread of the data points seems to be even more evident. Notice an interesting fact, in the case of water and placebo the group means are closer to each other, and to the overall mean. This makes sense, after all the animals ate and drunk exactly the same stuff, so they belong to the same population. On the other hand in the case of the two populations (water and drugY) the group means differ from the overall mean (again, think of it for a moment and convince yourself that it makes sense). Since we got Julia on our side we could even try to express this spread of data with numbers. First, the spread of data points around the group means

```
function getAbsDiffs(v::Vector{<:Real})::Vector{<:Real}
    return abs.(Stats.mean(v) .- v)
end

function getAbsPointDiffsFromGroupMeans(
```

```
    v1::Vector{<:Real}, v2::Vector{<:Real})::Vector{<:Real}
    return vcat(getAbsDiffs(v1), getAbsDiffs(v2))
end

ex1withinGroupsSpread = getAbsPointDiffsFromGroupMeans(
    ex1BwtsWater, ex1BwtsPlacebo)
ex2withinGroupsSpread = getAbsPointDiffsFromGroupMeans(
    ex2BwtsWater, ex2BwtsDrugY)

ex1AvgWithinGroupsSpread = Stats.mean(ex1withinGroupsSpread)
ex2AvgWithingGroupsSpread = Stats.mean(ex2withinGroupsSpread)

(ex1AvgWithinGroupsSpread, ex2AvgWithingGroupsSpread)
```

```
(1.941755009754579, 2.87288915817597)
```

The code is pretty simple. Here we calculate the distance of data points around the group means. Since we are not interested in a sign of a difference [+ (above), - (below) the mean] we use `abs` function. We used a similar methodology when we calculated `absDiffsStudA` and `absDiffsStudB` in Section 4.6 . This is as if we measured the distances from the group means in Figure 16 with a ruler and took the average of them. The only new part is the vcat[228]function. All it does is it glues two vectors together, like: `vcat([1, 2], [3, 4])` gives us `[1, 2, 3, 4]`. Anyway, the average distance of a point from a group mean is 1.9 [g] for experiment 1 (left panel in Figure 16). For experiment 2 (right panel in Figure 16) it is equal to 2.9 [g]. That is nice, as it follows our expectations. However, `AvgWithinGroupsSpread` by itself is not enough since sooner or later in `experiment 1` (hence prefix `ex1-`) we may encounter (a) population(s) with a wide natural spread of the data. Therefore, we need a more robust metric.

This is were the average spread of group means around the overall mean could be useful. Let's get to it, we will start with these functions

```
function repVectElts(v::Vector{T}, times::Vector{Int})::Vector{T} where
T
    @assert (length(v) == length(times)) "length(v) not equal
```

[228]https://docs.julialang.org/en/v1/base/arrays/#Base.vcat

```
length(times)"
    @assert all(map(x -> x > 0, times)) "times elts must be positive"
    result::Vector{T} = Vector{eltype(v)}(undef, sum(times))
    currInd::Int = 1
    for i in eachindex(v)
        for _ in 1:times[i]
            result[currInd] = v[i]
            currInd += 1
        end
    end
    return result
end

function getAbsGroupDiffsFromOverallMean(
    v1::Vector{<:Real}, v2::Vector{<:Real})::Vector{<:Real}
    overallMean::Float64 = Stats.mean(vcat(v1, v2))
    groupMeans::Vector{Float64} = [Stats.mean(v1), Stats.mean(v2)]
    absGroupDiffs::Vector{<:Real} = abs.(overallMean .- groupMeans)
    absGroupDiffs = repVectElts(absGroupDiffs, map(length, [v1, v2]))
    return absGroupDiffs
end
```

The function `repVectElts` is a helper function. It is slightly complicated and I will not explain it in detail. Just treat it as any other function from a library. A function you know only by name, input, and output. A function that you are not aware of its insides (of course if you really want you can figure them out by yourself). All it does is it takes two vectors `v` and `times`, then it replicates each element of `v` a number of times specified in `times` like so: `repVectElts([10, 20], [1, 2])` `output [10, 20, 20]`. And this is actually all you care about right now.

As for the `getAbsGroupDiffsFromOverallMean` it does exactly what it says. It subtracts group means from the overall mean `(overallMean .- groupMeans)` and takes absolute values of that [`abs.(`]. Then it repeats each difference as many times as there are observations in the group `repVectElts(absGroupDiffs, map(length, [v1, v2]))` (as if every single point in a group was that far away from the overall mean). This is what it returns to us.

OK, time to use the last function, behold

```
ex1groupSpreadFromOverallMean = getAbsGroupDiffsFromOverallMean(
    ex1BwtsWater, ex1BwtsPlacebo)
ex2groupSpreadFromOverallMean = getAbsGroupDiffsFromOverallMean(
    ex2BwtsWater, ex2BwtsDrugY)

ex1AvgGroupSpreadFromOverallMean =
Stats.mean(ex1groupSpreadFromOverallMean)
ex2AvgGroupSpreadFromOverallMean =
Stats.mean(ex2groupSpreadFromOverallMean)

(ex1AvgGroupSpreadFromOverallMean, ex2AvgGroupSpreadFromOverallMean)
```

```
(0.597596847858199, 3.6750594521844278)
```

OK, we got it. The average group mean spread around the overall mean is 0.6 [g] for experiment 1 (left panel in Figure 16) and 3.7 [g] for experiment 2 (right panel in Figure 16). Again, the values are as we expected them to be based on our intuition.

Now, we can use the obtained before `AvgWithinGroupSpread` as a reference point for `AvgGroupSpreadFromOverallMean` like so

```
LStatisticEx1 = ex1AvgGroupSpreadFromOverallMean /
ex1AvgWithinGroupsSpread
LStatisticEx2 = ex2AvgGroupSpreadFromOverallMean /
ex2AvgWithingGroupsSpread

(LStatisticEx1, LStatisticEx2)
```

```
(0.3077611979143188, 1.2792207599536367)
```

Here, we calculated a so called L-Statistic (`LStatistic`). I made the name up, because that is the first name that came to my mind. Perhaps it is because my family name is Lukaszuk or maybe because I'm selfish. Anyway, the higher the L-statistic (so the ratio of group spread around the overall mean to within group spread) the smaller the probability that such a big difference was caused by a chance alone (hmm, I think I said something along those lines in one of the previous chapters). If only we could reliably determine the cutoff point for my `LStatistic` (we will try to do so in Section 5.8.2).

Luckily, there is no point for us to do that since one-way ANOVA relies on a similar metric called F-statistic (BTW. Did I mention that the ANOVA was developed by Ronald Fisher[229]? Of course, in that case others bestowed the name in his honor). Observe. First, experiment 1:

```
Ht.OneWayANOVATest(ex1BwtsWater, ex1BwtsPlacebo)
```

```
One-way analysis of variance (ANOVA) test
-----------------------------------------
Population details:
    parameter of interest:   Means
    value under h_0:         "all equal"
    point estimate:          NaN

Test summary:
    outcome with 95% confidence: fail to reject h_0
    p-value:                     0.5738

Details:
    number of observations: [4, 4]
    F statistic:            0.353601
    degrees of freedom:     (1, 6)
```

Here, my made up `LStatistic` was 0.31 whereas the F-Statistic is 0.35, so kind of close. Chances are they measure the same thing but using slightly different methodology. Here, the p-value ($p > 0.05$) demonstrates that the groups may come from the same population (or at least that we do not have enough evidence to claim otherwise).

OK, now time for experiment 2:

```
Ht.OneWayANOVATest(ex2BwtsWater, ex2BwtsDrugY)
```

```
One-way analysis of variance (ANOVA) test
-----------------------------------------
Population details:
    parameter of interest:   Means
    value under h_0:         "all equal"
    point estimate:          NaN

Test summary:
```

[229] https://en.wikipedia.org/wiki/Ronald_Fisher

```
    outcome with 95% confidence: reject h_0
    p-value:                     0.0428

Details:
    number of observations: [4, 4]
    F statistic:            6.56001
    degrees of freedom:     (1, 6)
```

Here, the p-value ($p \leq 0.05$) demonstrates that the groups come from different populations (the means of those populations differ). As a reminder, in this case my made up L-Statistic (`LStatisticEx2`) was 1.28 whereas the F-Statistic is 6.56, so this time it is more distant. The differences stem from different methodology. For instance, just like in Section 4.6 here (`LStatisticEx2`) we used `abs` function as our power horse. But do you remember, that statisticians love to get rid of the sign from a number by squaring it. Anyway, let's rewrite our functions in a more statistical manner.

```
# compare with our getAbsDiffs
function getSquaredDiffs(v::Vector{<:Real})::Vector{<:Real}
    return (Stats.mean(v) .- v) .^ 2
end

# compare with our getAbsPointDiffsFromOverallMean
function getResidualSquaredDiffs(
    v1::Vector{<:Real}, v2::Vector{<:Real})::Vector{<:Real}
    return vcat(getSquaredDiffs(v1), getSquaredDiffs(v2))
end

# compare with our getAbsGroupDiffsAroundOverallMean
function getGroupSquaredDiffs(
    v1::Vector{<:Real}, v2::Vector{<:Real})::Vector{<:Real}
    overallMean::Float64 = Stats.mean(vcat(v1, v2))
    groupMeans::Vector{Float64} = [Stats.mean(v1), Stats.mean(v2)]
    groupSqDiffs::Vector{<:Real} = (overallMean .- groupMeans) .^ 2
    groupSqDiffs = repVectElts(groupSqDiffs, map(length, [v1, v2]))
    return groupSqDiffs
end
```

Note: In reality functions in statistical packages probably use a different formula for `getGroupSquaredDiffs` (they do not

replicate `groupSqDiffs`). Still, I like my explanation better, so I will leave it as it is.

The functions are very similar to the ones we developed earlier. Of course, instead of `abs.(` we used `.^2` to get rid of the sign. Here, I adopted the names (`group sum of squares` and `residual sum of squares`) that you may find in a statistical textbook/software.

Now we can finally calculate averages of those squares and the F-statistics itself with the following functions

```
function getResidualMeanSquare(
    v1::Vector{<:Real}, v2::Vector{<:Real})::Float64
    residualSquaredDiffs::Vector{<:Real} = getResidualSquaredDiffs(v1,
v2)
    return sum(residualSquaredDiffs) / getDf(v1, v2)
end

function getGroupMeanSquare(
    v1::Vector{<:Real}, v2::Vector{<:Real})::Float64
    groupSquaredDiffs::Vector{<:Real} = getGroupSquaredDiffs(v1, v2)
    groupMeans::Vector{Float64} = [Stats.mean(v1), Stats.mean(v2)]
    return sum(groupSquaredDiffs) / getDf(groupMeans)
end

function getFStatistic(v1::Vector{<:Real}, v2::Vector{<:Real})::Float64
    return getGroupMeanSquare(v1, v2) / getResidualMeanSquare(v1, v2)
end
```

Again, here I tried to adopt the names (`group mean square` and `residual mean square`) that you may find in a statistical textbook/software. Anyway, notice that in order to calculate `MeanSquares` we divided our sum of squares by the degrees of freedom (we met this concept and developed the functions for its calculation in Section 5.2 and in Section 5.3.2). Using degrees of freedom (instead of `length(vector)` like in the arithmetic mean) is usually said to provide better estimates of the desired values when the sample size(s) is/are small.

OK, time to verify our functions for the F-statistic calculation.

```
(
    getFStatistic(ex1BwtsWater, ex1BwtsPlacebo),
    getFStatistic(ex2BwtsWater, ex2BwtsDrugY),
)
```

```
(0.3536010850042917, 6.560010563323356)
```

To me, they look similar to the ones produced by `Ht.OneWayANOVATest` before, but go ahead scroll up and check it yourself. Anyway, under H_0 (all groups come from the same population) the F-statistic (so $\frac{groupMeanSq}{residMeanSq}$) got the F-Distribution[230](a probability distribution), hence we can calculate the probability of obtaining such a value (or greater) by chance and get our p-value (similarly as we did in Section 4.6.2 or in Section 5.2). Based on that we can deduce whether samples come from the same population (p > 0.05) or from different populations ($p \leq 0.05$). Ergo, we get to know if any group (means) differ(s) from the other(s).

Post-hoc tests

Let's start with a similar example to the ones we already met.

Imagine that you are a scientist and in the Amazon rain forest you discovered two new species of mice (`spB`, and `spC`). Now, you want to compare their body masses with an ordinary lab mice (`spA`) so you collect the data. If the body masses differ perhaps in the future they will become the criteria for species recognition.

```
# if you are in 'code_snippets' folder, then use: "./ch05/
miceBwtABC.csv"
# if you are in 'ch05' folder, then use: "./miceBwtABC.csv"
miceBwtABC = Csv.read("./code_snippets/ch05/miceBwtABC.csv",
Dfs.DataFrame)
```

[230]https://en.wikipedia.org/wiki/F-distribution

spA	spB	spC
18	21	23
21	25	27
20	26	25
23	24	28
22	21	27
19	24	26

Table 3: Table 3: Body mass [g] of three mice species (fictitious data).

Now, let us quickly look at the means and standard deviations in the three groups to get some impression about the data.

```
[
(n, Stats.mean(miceBwtABC[!, n]), Stats.std(miceBwtABC[!, n]))
    for n in Dfs.names(miceBwtABC) # n stands for name
]
```

```
("spA", 20.5, 1.8708286933869707)
```

```
("spB", 23.5, 2.073644135332772)
```

```
("spC", 26.0, 1.7888543819998317)
```

Here, the function `Dfs.names` returns `Vector{T}` with names of the columns. In connection with comprehensions we met in Section 3.6.3 it allows us to quickly obtain the desired statistics without typing the names by hand. Alternatively we would have to type

```
[
("spA", Stats.mean(miceBwtABC[!, "spA"]), Stats.std(miceBwtABC[!,
"spA"])),
("spB", Stats.mean(miceBwtABC[!, "spB"]), Stats.std(miceBwtABC[!,
"spB"])),
("spC", Stats.mean(miceBwtABC[!, "spC"]), Stats.std(miceBwtABC[!,
"spC"])),
]
```

It didn't save us a lot of typing in this case, but think what if we had 10, 30 or even 100 columns. The gain would be quite substantial.

Alternatively, if you read the documentation of the before mentioned (Section 5.3) `Dfs.describe` then you can go with:

```
Dfs.describe(miceBwtABC, :mean, :std)
```

variable	mean	std
spA	20.5	1.8708286933869707
spB	23.5	2.073644135332772
spC	26.0	1.7888543819998317

Table 4: Table 4: Selected summary statistics based on miceBwtABC data frame.

Anyway, based on the means it appears that the three species differ slightly in their body masses. Still, in connection with the standard deviations, we can see that the body masses in the groups overlap slightly. So, is it enough to claim that they are statistically different at the cutoff level of 0.05 (α)? Let's test that with the one-way ANOVA that we met in the previous chapter.

Let's start by checking the assumptions. First, the normality assumption

```
[getSWtestPval(miceBwtABC[!, n]) for n in Dfs.names(miceBwtABC)] |>
pvals -> map(pv -> pv > 0.05, pvals) |>
         all
```

true

All normal. Here we get the p-values from Shapiro-Wilk test for all our groups. Briefly, we obtain p-value (`getSWtestPval`) for each group (`Dfs.names(miceBwtABC)`). Then we pipe (compare with `|>` in `getSortedKeysVals` from Section 4.5) the result to `map` to check if the p-values (`pvals`) are greater than 0.05 (then we do not reject the null hypothesis of normal distribution). Finally, we pipe (`|>`) the

`Vector{Bool}` to the function all[231]. The function returns `true` only if all the elements of the vector are true.

OK, time for the homogeneity of variance assumption

```
Ht.FlignerKilleenTest(
    [miceBwtABC[!, n] for n in Dfs.names(miceBwtABC)]...
    ) |> Ht.pvalue |> pv -> pv > 0.05
```

true

The variances are roughly equal. Here `[miceBwtABC[!, n] for n in Dfs.names(miceBwtABC)]` returns `Vector{Vector{<:Real}}` so vector of vectors, e.g. `[[1, 2], [3, 4], [5, 6]]` but `Ht.FlingerTest` expects separate vectors `[1, 2], [3, 4], [5, 6]` (no outer square brackets). The splat operator (`...`) placed after the array removes the outer square brackets. Then we pipe the result of the test `Ht.FlingerTest` to `Ht.pvalue` because according to the documentation[232]it extracts the p-value from the result of the test. Finally, we pipe (`|>`) the result to an anonymous function (`pv -> pv > 0.05`) to check if the p-value is greater than 0.05 (then we do not reject the null hypothesis of variance homogeneity).

OK, and now for the one-way ANOVA.

```
Ht.OneWayANOVATest(
    [miceBwtABC[!, n] for n in Dfs.names(miceBwtABC)]...
    ) |> Ht.pvalue
```

0.0006608056579183923

Hmm, OK, the p-value is lower than the cutoff level of 0.05. What now. Well, by doing one-way ANOVA you ask your computer a very specific question: "Does at least one of the group means differs from the other(s)?". The computer does exactly what you tell it, nothing more, nothing less. Here, it answers your question precisely with: "Yes" (since $p \leq 0.05$). I assume that right now you are not satisfied

[231]https://docs.julialang.org/en/v1/base/collections/#Base.all-Tuple%7BAny%7D
[232]https://juliastats.org/HypothesisTests.jl/stable/

with the answer. After all, what good is it if you still don't know which group(s) differ one from another: `spA` vs. `spB` and/or `spA` vs `spC` and/or `spB` vs `spC`. If you want your computer to tell you that then you must ask it directly to do so. That is what post-hoc tests are for (`post hoc` means `after the event`, here the event is one-way ANOVA).

The split to one-way ANOVA and post-hoc tests made perfect sense in the 1920s-30s and the decades after the method was introduced. Back then you performed calculations with a pen and a piece of paper (and since ~1970s a pocket calculator as well). Once one-way ANOVA produced a p-value greater than 0.05 you stopped (and saved time and energy on an unnecessary additional calculations). Otherwise, and only then, you performed a post-hoc test (again with a pen and a piece of paper). Anyway, as mentioned in Section 4.9.4 the popular choices for post-hoc tests include Fisher's LSD test and Tukey's HSD test. Here we are going to use a more universal approach and apply a so called `pairwise t-test` (which is just a t-test, that you already know, done between every pairs of groups). Ready, here we go

```
evtt = Ht.EqualVarianceTTest
getPval = Ht.pvalue

# for "spA vs spB", "spA vs spC" and "spB vs spC", respectively
postHocPvals = [
evtt(miceBwtABC[!, "spA"], miceBwtABC[!, "spB"]) |> getPval,
evtt(miceBwtABC[!, "spA"], miceBwtABC[!, "spC"]) |> getPval,
evtt(miceBwtABC[!, "spB"], miceBwtABC[!, "spC"]) |> getPval,
]

postHocPvals
```

```
[0.025111501405268754, 0.0003985445257645916, 0.049332195639921715]
```

OK, here to save us some typing we assigned the long function names (`Ht.EqualVarianceTTest` and `Ht.pvalue`) to the shorter ones (`evtt` and `getPval`). Then we used them to conduct the t-tests and extract the p-values for all the possible pairs to compare (we will develop some more user friendly functions in the upcoming exercises, see Section 5.7.4). Anyway, it appears that here any mouse species differs

with respect to their average body weight from the other two species (all p-vaues are below 0.05). Or does it?

Multiplicity correction

In the previous section we performed a pairwise t-test for the following comparisons:

- `spA` vs `spB`,
- `spA` vs `spC`,
- `spB` vs `spC`.

The obtained p-values were

```
postHocPvals
```

```
[0.025111501405268754, 0.0003985445257645916, 0.049332195639921715]
```

Based on that we concluded that every group mean differs from every other group mean (all p-values are lower than the cutoff level for α equal to 0.05). However, there is a small problem with this approach (see the explanation below).

In Section 4.7.5 we said that it is impossible to reduce the type 1 error (α) probability to 0. Therefore if all our null hypothesis (H_0) were true we need to accept the fact that we will report some false positive findings. All we can do is to keep that number low.

Imagine you are testing a set of random substances to see if they reduce the size (e.g. diameter) of a tumor[233]. Most likely the vast majority of the tested substances will not work (so let's assume that in reality all H_0s are true). Now imagine, that the result each substance has on the tumor is placed in a separate graph. So, you draw a boxplot[234](like the one you will do in the upcoming Section 5.7.5). Now the question. How many graphs would contain false positive results if the cutoff level for α is 0.05? Pause for a moment and come up with the number. That is easy, 100 graphs times 0.05 (probability of

[233]https://en.wikipedia.org/wiki/Neoplasm
[234]https://en.wikipedia.org/wiki/Box_plot

false positive) gives us the expected `100 * 0.05` = 5 figures with false positives. BTW. If you got it, congratulations. If not compare the solution with the calculations we did in Section 4.5 . Anyway, you decided that this will be your golden standard, i.e. no more than 5% ($\frac{5}{100}$ = 0.05) of figures with false positives.

But here (in `postHocPvals` above) you got 3 comparisons and therefore 3 p-values. Imagine that you place such three results into a single figure. Now, the question is: under the conditions given above (all H_0s true, cutoff for α = 0.05) how many graphs would contain false positives if you placed three such comparisons per graph for 100 figures? Think for a moment and come up with the number.

OK, so we got 100 graphs, each reporting 3 comparisons (3 p-values), which gives us in total 300 results. Out of them we expect `300 * 0.05` = 15 to be false positives. Now, we pack those 300 results into 100 figures. In the best case scenario the 15 false positives will land in the first five graphs (three false positives per graph, `5*3` = 15), the remaining 285 true negatives will land in the remaining 95 figures (three true negatives per graph, `95*3` = 285). The golden standard seems to be kept (`5/100` = 0.05). The problem is that we don't know which figures get the false positives. The Murphy's law[235] states: "Anything that can go wrong will go wrong, and at the worst possible time." (or in the worst possible way). If so, then the 15 false positives will go to 15 different figures (1 false positive + 2 true negatives per graph), and the remaining `285 - 2*15` = 255 true negatives will go to the remaining `255/3` = 85 figures. Here, your golden standard (5% of figures with false positives) is violated (`15/100` = 0.15).

This is why we cannot just leave the three `postHocPvals` as they are. We need to act, but what can we do to counteract the problem. Well, if the initial cutoff level for α was 3 times smaller (`0.05/3` = 0.017) then in the case above we would have `300 * (0.05/3)` $\approx$ 5.0 false positives to put into 100 figures and everything would be OK even in the worst case scenario. Alternatively, since division is inverse operation to multiplication we could just multiply every p-value by 3 (number of

[235] https://en.wikipedia.org/wiki/Murphy%27s_law

comparisons) and check its significance at the cutoff level for α = 0.05, like so

```
function adjustPvalue(pVal::Float64, by::Int)::Float64
    @assert (0 <= pVal <= 1) "pVal must be in range [0-1]"
    return min(1, pVal*by)
end

function adjustPvalues(pVals::Vector{Float64})::Vector{Float64}
    return adjustPvalue.(pVals, length(pVals))
end

# p-values for comparisons: spA vs spB, spA vs spC, and spB vs spC
adjustPvalues(postHocPvals)
```

```
[0.07533450421580626, 0.0011956335772937748, 0.14799658691976514]
```

Notice, the since on entry a p-value may be, let's say, 0.6 then multiplying it by 3 would give us 1.8 which is an impossible value for probability (see Section 4.3.1). That is why we set the upper limit to 1 by using `min(1, pVal*by)`. Anyway, after adjusting for multiple comparisons only one species differs from the other (`spA` vs `spC`, adjusted $p - value \leq 0.05$). And this is our final conclusion.

The method we used above (in `adjustPvalue` and `adjustPvalues`) is called the Bonferroni correction[236]. Probably it is the simplest method out there and it is useful if we have a small number of independent comparisons/p-values (let's say up to 6). For a large number of comparisons you are likely to end up with a paradox:

- one-way ANOVA (which controls the overall α at the level of 0.05) indicates that there are some statistically significant differences,
- the corrected p-values (which rely on different assumptions) show no significant differences.

Therefore, for large number of comparisons you may choose a different (less strict) method, e.g. the Benjamini-Hochberg

[236]https://en.wikipedia.org/wiki/Bonferroni_correction

procedure[237]. Both of those (Bonferroni and Benjamini-Hochberg) are available in the MultipleTesting[238]package. Observe

```
import MultipleTesting as Mt
# p-values for comparisons: spA vs spB, spA vs spC, and spB vs spC
resultsOfThreeAdjMethods = (
    adjustPvalues(postHocPvals),
    Mt.adjust(postHocPvals, Mt.Bonferroni()),
    Mt.adjust(postHocPvals, Mt.BenjaminiHochberg())
)

resultsOfThreeAdjMethods
```

```
([0.07533450421580626, 0.0011956335772937748, 0.14799658691976514],
 [0.07533450421580626, 0.0011956335772937748, 0.14799658691976514],
 [0.03766725210790313, 0.0011956335772937748, 0.049332195639921715])
```

As expected, the first two lines give the same results (since they both use the same adjustment method). The third line, and a different method, produces a different result (and hence yields distinctive interpretation).

A word of caution, you shouldn't just apply 10 different adjustment methods on the obtained p-values and choose the one that produces the greatest number of significant differences. Instead you should choose a correction method *a priori* (up front, in advance) and stick to it later (make the final decision of which group(s) differ based on the adjusted p-values). Therefore, it takes some consideration to choose the multiplicity correction well.

OK, enough of theory, time for some practice. Whenever you're ready click the right arrow to go to the exercises for this chapter.

Exercises - Comparisons of Continuous Data

Just like in the previous chapters here you will find some exercises that you may want to solve to get from this chapter as much as you can

[237]https://en.wikipedia.org/wiki/False_discovery_rate#Benjamini%E2%80%93Hochberg_procedure

[238]https://github.com/juliangehring/MultipleTesting.jl

(best option). Alternatively, you may read the task descriptions and the solutions (and try to understand them).

Exercise 1

In Section 5.2 we said that when we draw a small random sample from a normal distribution of a given mean (μ) and standard deviation (σ) then the distribution of the sample means will be pseudo-normal with the mean roughly equal to the population mean and the standard deviation roughly equal to sem (standard error of the mean).

Time to confirm that. Moreover, it's time to practice our plotting skills (I think we neglected them so far).

In this task your population of interest is `Dsts.Normal(80, 20)`. To make it more concrete let's say this is the distribution of body weight for adult humans. To plot you may use CairoMakie[239]or some other plotting library (read the tutorial(s)/docs first).

1) draw a random sample of size 10 from the population `Dsts.Normal(80, 20)`. Calculate `sem` and `sd` for the sample,
2) draw 100'000 random samples of size 10 from the population `Dsts.Normal(80, 200)` and calculate the samples means (100'000 sample means)
3) draw the histogram of the sample means from point 2 using, e.g. Cmk.hist[240]. Afterwards, you may set the y-axis limits from 0 to 4000, with `Cmk.ylims!(0, 4000)`.
4) on the histogram mark the population mean ($\mu = 80$) with a vertical line using, e.g. Cmk.vlines[241]
5) annotate the line from point 4 (e.g. type "population mean = 80") using, e.g. Cmk.text[242]
6) on the histogram mark the means standard deviation using, e.g. Cmk.bracket[243],
7) annotate the histogram (above the bracket from point 6) with the means standard deviation, using, e.g. Cmk.text[244],
8) annotate the histogram with the sample's `sem` and `sd` (from point 1) and compare them with the means standard deviation from point 7.

[239]https://docs.makie.org/stable/documentation/backends/cairomakie/

And that's it. This may look like a lot of work to do, but don't freak out, do it one point at a time, look at the instructions (they are pretty precise on purpose).

Remember that each of those functions may have an equivalent that ends with `!` (a function that modifies an already existing figure). It is for you to decide when to use which version of a plotting function.

Exercise 2

Do you remember how in Section 5.4 we calculated the L-statistic for `ex2BwtsWater` and `ex2BwtsDrugY` and found out its value was equal to `LStatisticEx2` = 1.28? Then we calculated the famous F-statistic for the same two groups (`ex2BwtsWater` and `ex2BwtsDrugY`) and it was equal to `getFStatistic(ex2BwtsWater, ex2BwtsDrugY)` = 6.56. The probability of obtaining an F-value greater than this (by chance) if H_0 is true (i.e. both groups come from the same distribution (`Dsts.Normal(25, 3)`) is equal to:

```
# the way we calculated it in the chapter (more or less)
Ht.OneWayANOVATest(ex2BwtsWater, ex2BwtsDrugY) |> Ht.pvalue
```

0.04283642629899474

Alternatively, we cold calculate it also with our friendly `Distributions` package (similarly to how we used it in, e.g. Section 4.6.2)

```
# the way we can calculate it with Distributions package
# 1 - Dfs for groups (number of groups - 1),
# 6 - Dfs for residuals (number of observations - number of groups)
```

[240] https://docs.makie.org/stable/examples/plotting_functions/hist/index.html#hist

[241] https://docs.makie.org/stable/examples/plotting_functions/hvlines/index.html#vlines

[242] https://docs.makie.org/stable/examples/plotting_functions/text/index.html#text

[243] https://docs.makie.org/stable/examples/plotting_functions/bracket/

[244] https://docs.makie.org/stable/examples/plotting_functions/text/index.html#text

```
1 - Dsts.cdf(Dsts.FDist(1, 6), getFStatistic(ex2BwtsWater,
ex2BwtsDrugY))
```

0.042836426298994756

Hopefully, you remember that. OK, here is the task.

1) write a function `getLStatistic(v1::Vector{<:Real}, v2::Vector{<:Real})::Float64` that calculates the L-Statistic for two given vectors
2) estimate the L-Distribution. To do that:

   2.1) run, let's say 1'000'000 simulations under H_0 that `v1` and `v2` come from the same population (`Dsts.Normal(25, 3)`, draw 4 observations per vector). Calculate the L-Statistic each time (round it to 1 decimal place with `round(getLStatistic(v1, v2), digits=1)`

   2.2) use `getCounts` (Section 4.4), `getProbs` (Section 4.4) and `getSortedKeysVals` (Section 4.5) to obtain the probabilities for each value of the L-Statistic produced in point 2.1

   2.3) based on the data from point 2.2 calculate the probability of L-Statistic being greater than `LStatisticEx2` = 1.28. Compare the probability with the probability obtained for the F-Statistic (presented in the code snippets above)
3) using, e.g. Cmk.lines[245](`color="blue"`) and the data from point 2.2 plot the probability distribution for the L-Distribution
4) add vertical line, e.g with `Cmk.vlines` at L-Statistic = 1.28, annotate the line with `Cmk.text`
5) check what happens if both the samples from point 2.1 come from a different population (e.g. `Dsts.Normal(100, 50)`). Plot the new distribution on the old one (point 3) with, e.g. Cmk.scatter[246] (`marker=:circle`, `color="blue"`).
6) check what happens if the samples from point 2.1 come from the same distribution (`Dsts.Normal(25, 3)`) but are of different size (8

observations per vector). Plot the new distribution on the old one (point 3) with, e.g. `Cmk.scatter` (`marker=:xcross`, `color="blue"`).

Optionally, if you want to make your plots more readable and if you like challenges you may:

7) add the F-Distribution to the plot, e.g. with `Cmk.lines` (`color="red"`)
8) add legends[247] to the plots

Again. This may look like a lot of work to do, but don't freak out, do it one point at a time, look at the instructions (they are pretty precise on purpose). If you get stuck, take a sneak peak at the solution and continue on your own once you get back on the track.

Exercise 3

Let's cool down after the last two demanding exercises.

In this task I want you to write the function `getPValUnpairedTest(v1::Vector{<:Real}, v2::Vector{<:Real})::Float64`. The function accepts two vectors, runs an unpaired test and returns the p-value.

The function should check the:

1) normality (`Ht.ShapiroWilkTest`), and
2) homogeneity of variance (`Ht.FlingerTest`)

assumptions.

If both the assumptions hold then run `Ht.EqualVarianceTTest`.

If only normality assumption holds then run `Ht.UnequalVarianceTTest`.

Otherwise run `Ht.MannWhitneyUTest`.

---

[245] https://docs.makie.org/stable/examples/plotting_functions/lines/index.html#lines

[246] https://docs.makie.org/stable/examples/plotting_functions/scatter/index.html#scatter

[247] https://docs.makie.org/stable/examples/blocks/legend/index.html#multi-group_legends

Exercise 4

Write a function with the following signature:

```
function getPValsUnpairedTests(
    df::Dfs.DataFrame
    )::Dict{Tuple{String,String},Float64}
```

The function accepts a data frame (like `miceBwtABC` we met in Section 5.5). Then it runs the appropriate comparisons (use `getPValUnpairedTest` that you developed in Section 5.7.3) and returns the p-values for comparisons in the form of a dictionary where the keys are the names of the compared groups (`Tuple{String, String}`), and the values are pvalues (e.g. `Dict(("grX", "grY") => 0.3, ("grX", "grZ") => 0.022)`. The function should compare every group with every other group.

Once you are done with this task tweak your function slightly to have the following signature:

```
function getPValsUnpairedTests(
    df::Dfs.DataFrame,
    multCorr
    )::Dict{Tuple{String,String},Float64}
```

This function adjusts the obtained p-values using some sort of multiplicity correction (`multCorr`) from `MultipleTesting` package we discussed before (Section 5.6). I didn't write the type signature for `multCorr` here because it might be frightening at first sight. Still, even without it the function should work just fine.

Test your function on `miceBwtABC` and compare the results with those we obtained in Section 5.5 and in Section 5.6 .

Exercise 5

It appears that when a scientific paper presents a comparison between few groups of continuous variables it does so in a form of bar-plot or

box-plot[248]with some markers for statistically significant differences over the bars/boxes.

So here is your task. For data from `miceBwtABC` from Section 5.5 write a function that draws a plot similar to the one below (it doesn't have to be the exact copy).

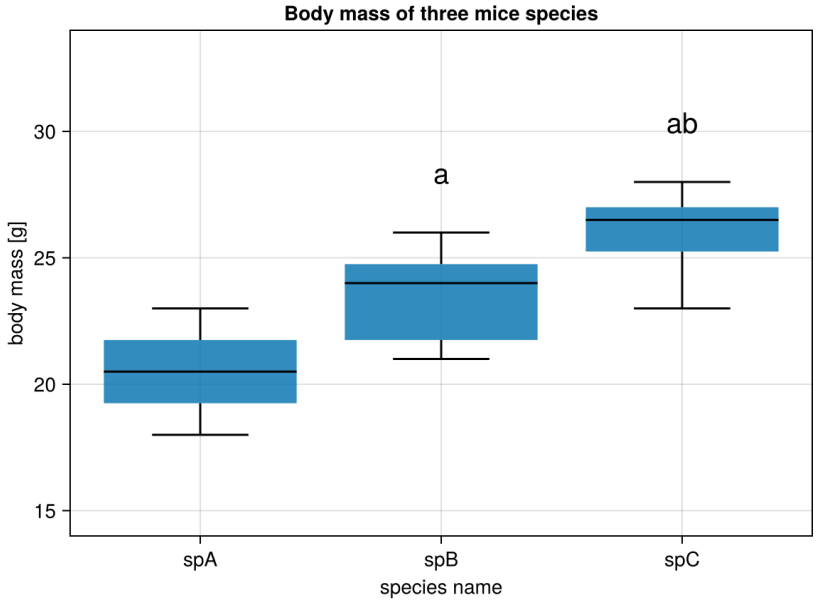

Figure 18: Figure 17: Boxplot of body mass of three mice species (fictitious data). a - difference vs. spA ($p < 0.05$), b - difference vs. spB ($p < 0.05$).

In the graph a middle horizontal line in a box is the median[249], a box depicts interquartile range[250](IQR), the whiskers length is equal to 1.5 * IQR (or the maximum and minimum if they are smaller than 1.5 * IQR).

For the task you may use:

- Cmk.boxplot[251]- to draw the boxplot

[248]https://en.wikipedia.org/wiki/Box_plot
[249]https://en.wikipedia.org/wiki/Median
[250]https://en.wikipedia.org/wiki/Interquartile_range

- Cmk.xticks[252]- to add group labels in x-ticks
- p-values provided by `getPValsUnpairedTests(miceBwtABC, Mt.BenjaminiHochberg)` from the last exercise to generate statistical significance markers.
- Cmk.text[253]to place the markers in the correct positions on the plot.

The function should also work for different data frames of similar kind with different number of groups in the columns.

Solutions - Comparisons of Continuous Data

In this sub-chapter you will find exemplary solutions to the exercises from the previous section.

Solution to Exercise 1

First the sample and the 100'000 simulations:

```
Rand.seed!(321)
ex1sample = Rand.rand(Dsts.Normal(80, 20), 10)
ex1sampleSd = Stats.std(ex1sample)
ex1sampleSem = getSem(ex1sample)
ex1sampleMeans = [
    Stats.mean(Rand.rand(Dsts.Normal(80, 20), 10))
    for _ in 1:100_000]
ex1sampleMeansMean = Stats.mean(ex1sampleMeans)
ex1sampleMeansSd = Stats.std(ex1sampleMeans)
```

The code doesn't contain any new elements, so I will leave it to you to figure out what happened there.

And now, let's move to the plot.

```
fig = Cmk.Figure()
ax1 = Cmk.Axis(fig[1, 1],
               title="Histogram of 100'000 sample means",
               xlabel="Adult human body weight [kg]",
               ylabel="Count")
Cmk.hist!(ax1, ex1sampleMeans, bins=100,
```

[251]https://docs.makie.org/stable/examples/plotting_functions/boxplot/index.html#boxplot

[252]https://docs.makie.org/stable/examples/blocks/axis/index.html#xticks

[253]https://docs.makie.org/stable/examples/plotting_functions/text/index.html#text

```
          color=Cmk.RGBAf(0, 0, 1, 0.3))
Cmk.ylims!(ax1, 0, 4000)
Cmk.vlines!(ax1, 80, ymin=0.0, ymax=0.85, color="black",
linestyle=:dashdot)
Cmk.text!(ax1, 81, 1000, text="population mean = 80")
Cmk.bracket!(ax1,
             ex1sampleMeansMean - ex1sampleMeansSd / 2, 3500,
             ex1sampleMeansMean + ex1sampleMeansSd / 2, 3500,
             style=:square)
Cmk.text!(ax1, 72.5, 3700,
          text="sample means sd = 6.33")
Cmk.text!(ax1, 90, 3200,
          text="single sample sd = 17.32")
Cmk.text!(ax1, 90, 3000,
          text="single sample sem = 5.48")
fig
```

This produces the following graph.

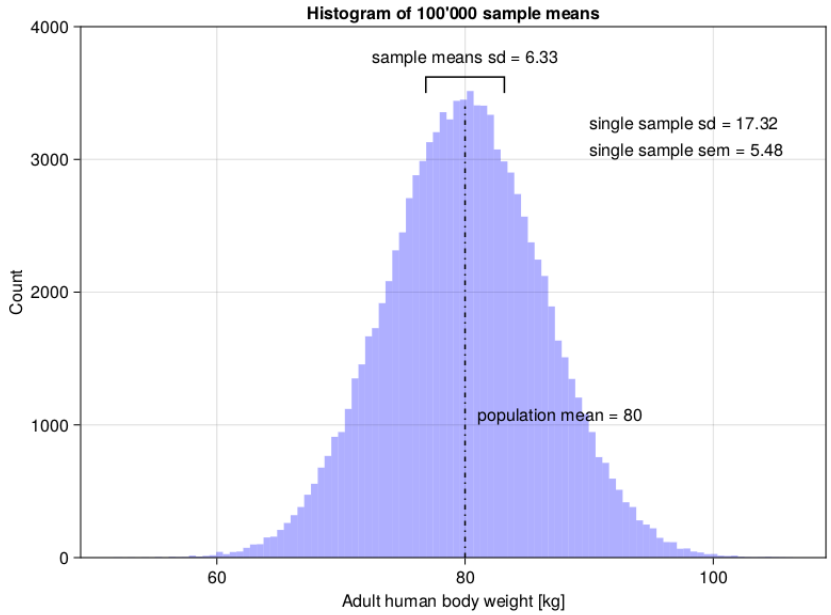

Figure 19: Figure 18: Histogram of drawing 100’000 random samples from a population with $\mu = 80$ and $\sigma = 20$.

The graph clearly demonstrates that a better approximation of the samples means sd is a single sample `sem` and not a single sample `sd` (as stated in Section 5.2).

I'm not gonna explain the code snippet above in great detail since this is a warm up exercise, and the tutorials[254](e.g. the basic tutorial) and the documentation for the plotting functions (see the links in Section 5.7.1) are pretty good. Moreover, we already used `CairoMakie` plotting functions in Section 4.5 . Still, a few quick notes are in order.

First of all, drawing a graph like that is not an enormous feat, you just need some knowledge (you read the tutorial and the function docs, right?). The rest is just patience and replication of the examples. Ah yes, I forgot about the try and error process [that happens from time to time (OK, more often than I would like to admit) in my case]. If an error happens, do not panic try to read the error's message and think what it tells you).

It is always a good idea to annotate the graph, add the title, x- and y-axis labels (to make the reader's, and your own, reasoning easier). Figures are developed from top to bottom (in the code), layer after layer (top line of code -> bottom layer on a graph, next line of code places a layer above the previous layer). First function (`fig`, `Cmk.Axis`, and `Cmk.hist!`) creates the figure, the following functions (e.g. `Cmk.text!` and `Cmk.vlines!`), write/paint something on the previous layers. After some time and tweaking you should be able to produce quite pleasing figures (just remember, patience is the key). One more point, instead of typing strings by hand (like `text="sample sd = 17.32"`) you may let Julia do that by using strings interpolation[255], like `text="sample sd = $(round(ex1sampleSd, digits=2))"`(with time you will appreciate the convenience of this method).

[254]https://docs.makie.org/v0.21/tutorials/getting-started

[255]https://docs.julialang.org/en/v1/manual/strings/#string-interpolation

One more thing, the `:dashdot` (after the keyword argument[256] `linetype`) is a Symbol[257]. For now you may treat it like a string but written differently, i.e. `:dashdot` instead of `"dashdot"`.

Solution to Exercise 2

First let's start with the functions we developed in Section 4 (and its subsections). We already now them, so I will not explain them here.

```
function getCounts(v::Vector{T})::Dict{T,Int} where {T}
    counts::Dict{T,Int} = Dict()
    for elt in v
        counts[elt] = get(counts, elt, 0) + 1
    end
    return counts
end

function getProbs(counts::Dict{T,Int})::Dict{T,Float64} where {T}
    total::Int = sum(values(counts))
    return Dict(k => v / total for (k, v) in counts)
end

function getSortedKeysVals(d::Dict{A,B})::Tuple{
    Vector{A},Vector{B}} where {A,B}
    sortedKeys::Vector{A} = keys(d) |> collect |> sort
    sortedVals::Vector{B} = [d[k] for k in sortedKeys]
    return (sortedKeys, sortedVals)
end
```

Now, time to define `getLstatistic` based on what we learned in Section 5.4 (note, the function uses `getAbsGroupDiffsAroundOverallMean` and `getAbsPointDiffsFromGroupMeans` that we developed in that section).

```
function getLStatistic(v1::Vector{<:Real}, v2::Vector{<:Real})::Float64
    absDiffsOverallMean::Vector{<:Real} =
        getAbsGroupDiffsFromOverallMean(v1, v2)
    absDiffsGroupMean::Vector{<:Real} =
        getAbsPointDiffsFromGroupMeans(v1, v2)
    return Stats.mean(absDiffsOverallMean) /
Stats.mean(absDiffsGroupMean)
end
```

[256] https://docs.julialang.org/en/v1/manual/functions/#Keyword-Arguments
[257] https://docs.julialang.org/en/v1/manual/metaprogramming/#Symbols

OK, that was easy, after all we practically did it all before, we only needed to look for the components in the previous chapters. Now, the function to determine the distribution.

```
function getLStatisticsUnderH0(
    popMean::Real, popSd::Real,
    nPerGroup::Int=4, nIter::Int=1_000_000)::Vector{Float64}

    v1::Vector{Float64} = []
    v2::Vector{Float64} = []
    result::Vector{Float64} = zeros(nIter)

    for i in 1:nIter
        v1 = Rand.rand(Dsts.Normal(popMean, popSd), nPerGroup)
        v2 = Rand.rand(Dsts.Normal(popMean, popSd), nPerGroup)
        result[i] = getLStatistic(v1, v2)
    end

    return result
end
```

This one is slightly more complicated, so I think a bit of explanation is in order here. First we initialize some variables that we will use later. For instance, `v1` and `v2` will hold random samples drawn from a population of interest (`Dsts.Normal(popMean, popSd)`) and will change with each iteration. The vector `result` is initialized with `0`s and will hold the `LStatistic` calculated during each iteration for `v1` and `v2`. The result vector is returned by the function. Later on we will be able to use it to `getCounts` and `getProbs` for the L-Statistics. This should work just fine. However, if we slightly modify our function (`getLStatisticsUnderH0`), we could use it not only with the L-Statistic but also F-Statistic (optional points in this task) or any other statistic of interest. Observe

```
# getXStatFn signature:
fnName(::Vector{<:Real}, ::Vector{<:Real})::Float64
function getXStatisticsUnderH0(
    getXStatFn::Function,
    popMean::Real, popSd::Real,
    nPerGroup::Int=4, nIter::Int=1_000_000)::Vector{Float64}

    v1::Vector{Float64} = []
    v2::Vector{Float64} = []
```

```
    result::Vector{Float64} = zeros(nIter)

    for i in 1:nIter
        v1 = Rand.rand(Dsts.Normal(popMean, popSd), nPerGroup)
        v2 = Rand.rand(Dsts.Normal(popMean, popSd), nPerGroup)
        result[i] = getXStatFn(v1, v2)
    end

    return result
end
```

Here, instead of `getLStatisticsUnderH0` we named the function `getXStatisticsUnderH0`, where `X` is any statistic we can come up with. The function that calculates our statistic of interest is passed as a first argument to `getXStatisticsUnderH0` (`getXStatFn`). The `getXStatFn` should work just fine, if it accepts two vectors (`::Vector{<:Real}`) and returns the statistic of interest (as `Float64`). Both those assumptions are fulfilled by `getLStatistic` (defined above) and `getFStatistic` defined in Section 5.4 . To use our `getXStatisticsUnderH0` we would type, e.g.: `getXStatisticsUnderH0(getFStatistic, 25, 3, 4)` or `getXStatisticsUnderH0(getLStatistic, 25, 3, 4)` instead of `getLStatisticsUnderH0(25, 3, 4)` that we defined in our first try (so more typing, but greater flexibility, and the result would be the same).

Now, to get a distribution of interest we use the following function

```
# getXStatFn signature:
fnName(::Vector{<:Real}, ::Vector{<:Real})::Float64
function getXDistUnderH0(getXStatFn::Function,
    mean::Real, sd::Real,
    nPerGroup::Int=4, nIter::Int=10^6)::Dict{Float64,Float64}

    xStats::Vector{<:Float64} = getXStatisticsUnderH0(
        getXStatFn, mean, sd, nPerGroup, nIter)
    xStats = round.(xStats, digits=1)
    xCounts::Dict{Float64,Int} = getCounts(xStats)
    xProbs::Dict{Float64,Float64} = getProbs(xCounts)

    return xProbs
end
```

First, we calculate the statistics of interest (`xStats`), then we round the statistics to a 1 decimal point (`round.(xStats, digits=1)`). This is necessary, since in a moment we will use `getCounts` so we need some repetitions in our `xStats` vector (e.g. 1.283333331 and 1.283333332 will, both get rounded to 1.3 and the count for this value of the statistic will be 2). Once we got the counts, we change them to probabilities (fraction of times that the given value of the statistic occurred) with `getProbs`.

Now we can finally, use them to estimate the probability that the L-statistic greater than `LStatisticEx2` = 1.28 occurred by chance.

```
Rand.seed!(321)
lprobs = getXDistUnderH0(getLStatistic, 25, 3)
lprobsGTLStatisticEx2 = [v for (k, v) in lprobs if k > LStatisticEx2]
lStatProb = sum(lprobsGTLStatisticEx2)
```

0.045378999999999996

Here, we used a comprehension with `if`. So, for every key-value pair (`(k, v)`) that is in `lprobs` we choose only those whose key (L-Statistic) is greater than `LStatisticEx2` (`if k > LStatisticEx2`). In the last step we take only value (`[v`) from the pair (the value is the probability of such L-Statistic happening by chance alone) to our result `lprobsGTLStatisticEx2`. If this (comprehension with if) is to complicated for you then you may consider using filter[258]and pipe (`|>`) the result to `values |> collect`.

The estimated probability for our L-Statistic is 0.045 which is pretty close to the probability obtained for the F-Statistic (`Ht.OneWayANOVATest(ex2BwtsWater, ex2BwtsDrugY) |> Ht.pvalue` = 0.043) (and well it should).

In virtually the same way we can get the experimental probability of an F-statistic being greater than `getFStatistic(ex2BwtsWater, ex2BwtsDrugY)` = 6.56 by chance. Observe

[258]https://docs.julialang.org/en/v1/base/collections/#Base.filter

```
Rand.seed!(321)
cutoffFStat = getFStatistic(ex2BwtsWater, ex2BwtsDrugY)
fprobs = getXDistUnderH0(getFStatistic, 25, 3)
fprobsGTFStatisticEx2 = [v for (k, v) in fprobs if k > cutoffFStat]
fStatProb = sum(fprobsGTFStatisticEx2)
```

0.043154000000000005

Again, the p-value is quite similar to the one we got from a formal `Ht.OneWayANOVATest` (as it should be).

OK, now it's time to draw some plots. First, let's get the values for x- and y-axes

```
Rand.seed!(321)
# L distributions
lxs1, lys1 = getXDistUnderH0(getLStatistic, 25, 3) |> getSortedKeysVals
lxs2, lys2 = getXDistUnderH0(getLStatistic, 100, 50) |>
getSortedKeysVals
lxs3, lys3 = getXDistUnderH0(getLStatistic, 25, 3, 8) |>
getSortedKeysVals
# F distribution
fxs1, fys1 = getXDistUnderH0(getFStatistic, 25, 3) |> getSortedKeysVals
```

No, big deal L-Distributions start with `l`, the classical F-Distribution starts with `f`. BTW. Notice that thanks to `getXDistUnderH0` we didn't have to write two almost identical functions (`getLDistUnderH0` and `getFDistUnderH0`).

OK, let's place them on the graph

```
fig = Cmk.Figure()
ax1 = Cmk.Axis(fig[1, 1],
               title="F-Distribution (red) and L-Distribution (blue)",
               xlabel="Value of the statistic",
               ylabel="Probability of outcome")
l1 = Cmk.lines!(ax1, fxs1, fys1, color="red")
l2 = Cmk.lines!(ax1, lxs1, lys1, color="blue")
sc1 = Cmk.scatter!(ax1, lxs2, lys2, color="blue", marker=:circle)
sc2 = Cmk.scatter!(ax1, lxs3, lys3, color="blue", marker=:xcross)
Cmk.vlines!(ax1, LStatisticEx2, color="lightblue", linestyle=:dashdot)
Cmk.text!(ax1, 1.35, 0.1,
          text="L-Statistic = 1.28")
Cmk.xlims!(ax1, 0, 4)
Cmk.ylims!(ax1, 0, 0.25)
```

```
Cmk.axislegend(ax1,
    [l1, l2, sc1, sc2],
    [
        "F-Statistic(1, 6) [Dsts.Normal(25, 3), n = 4]",
        "L-Statistic [Dsts.Normal(25, 3), n = 4]",
        "L-Statistic [Dsts.Normal(100, 50), n = 4]",
        "L-Statistic [Dsts.Normal(25, 3), n = 8]"
    ],
    "Distributions
(num groups = 2,
n - num observations per group)",
    position=:rt)
fig
```

Behold

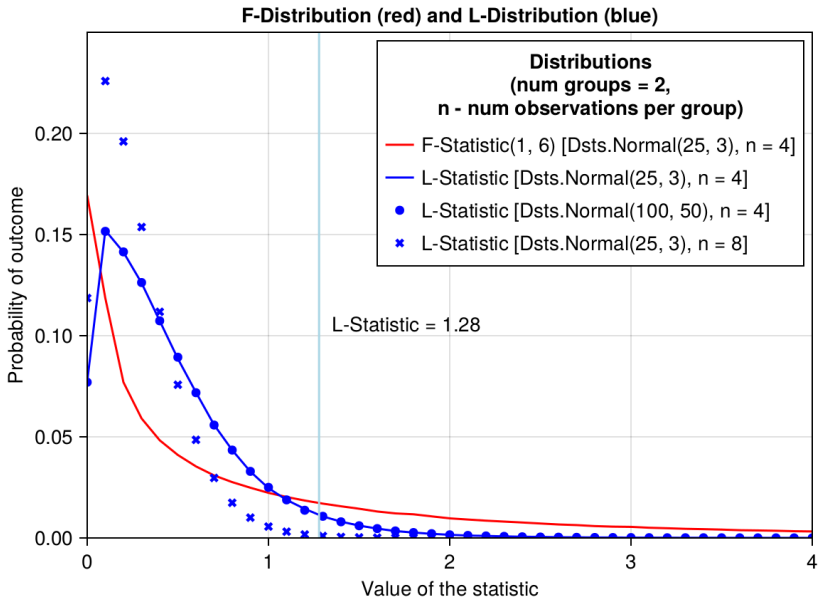

Figure 20: Figure 19: Experimental F- and L-Distributions.

Wow, what a beauty.

A few points of notice. Before, we calculated the probability (`lStatProb`) of getting the L-Statistic value greater than the vertical light blue line (the area under the blue curve to the right of that line). This is a one tail probability only. Interestingly, for the L-Distribution

the mean and sd in the population of origin are not that important (blue circles for `Dsts.Normal(100, 50)` lie exactly on the blue line for `Dsts.Normal(25, 3)`). However, the number of groups and the number of observations per group affect the shape of the distribution (blue xcrosses for `Dsts.Normal(25, 3) n = 8` diverge from the blue curve for `Dsts.Normal(25, 3) n = 4`).

The same is true for the F-Distribution. That is why the F-Distribution depends only on the degrees of freedom (`Dsts.FDist(dfGroup, dfResidual)`). The degrees of freedom depend on the number of groups and the number of observations per group.

Solution to Exercise 3

OK, let's start with functions for checking the assumptions.

```
function
areAllDistributionsNormal(vects::Vector{<:Vector{<:Real}})::Bool
    return [getSWtestPval(v) for v in vects] |>
           pvals -> map(pv -> pv > 0.05, pvals) |>
                    all
end

function areAllVariancesEqual(vects::Vector{<:Vector{<:Real}})
    return Ht.FlignerKilleenTest(vects...) |>
        Ht.pvalue |> pv -> pv > 0.05
end
```

The functions above are basically just wrappers around the code we wrote in Section 5.5 . Now, time for `getPValUnpairedTest`

```
function getPValUnpairedTest(
    v1::Vector{<:Real}, v2::Vector{<:Real})::Float64

    normality::Bool = areAllDistributionsNormal([v1, v2])
    homogeneity::Bool = areAllVariancesEqual([v1, v2])

    return (
        (normality && homogeneity) ? Ht.EqualVarianceTTest(v1, v2) :
        (normality) ? Ht.UnequalVarianceTTest(v1, v2) :
        Ht.MannWhitneyUTest(v1,v2)
        ) |> Ht.pvalue
end
```

The code is rather self-explanatory, of course if you remember the ternary expression from Section 3.5.2 and Section 3.9.4 .

Let's test our newly created function with the data from Section 5.3.2 (`miceBwt`)

```
getPValUnpairedTest([miceBwt[!, n] for n in Dfs.names(miceBwt)]...) |>
x -> round(x, digits=4)
```

0.0804

The p-value is the same as in Section 5.3.2 (as it should be), but this time we didn't have to explicitly check the assumptions before applying the appropriate test.

Solution to Exercise 4

First, let's start with a helper function that will return us all the possible pairs from a vector.

```
function getUniquePairs(uniqueNames::Vector{T})::Vector{Tuple{T,T}}
where T

    @assert (length(uniqueNames) >= 2) "the input must be of length >=
2"

    uniquePairs::Vector{Tuple{T,T}} =
        Vector{Tuple{T,T}}(undef, binomial(length(uniqueNames), 2))
    currInd::Int = 1

    for i in eachindex(uniqueNames)[1:(end-1)]
        for j in eachindex(uniqueNames)[(i+1):end]
            uniquePairs[currInd] = (uniqueNames[i], uniqueNames[j])
            currInd += 1
        end
    end

    return uniquePairs
end
```

The function is generic, so it can be applied to vector of any type (`T`), here designed as `Vector{T}`. It starts by initializing an empty vector (`uniquePairs`) to hold the results. The initialization takes the following form: `Vector{typeOfVectElements}(iniaialValues,`

`lengthOfTheVector)`. The vector is filled with `undefs` (undefined values, some garbage) as placeholders. The size of the new vector is calculated by the binomial[259]function. It is applied in the form `binomial(n, k)` where `n` is number of values to choose from and `k` is number of values per group. The function returns the number of possible groups of a given size. The rest is just iteration (`for` loops) over the indexes (`eachindex`) of the `uniqueNames` vector to get all the possible pairs. Let's quickly check if the function works as expected.

```
(
    getUniquePairs([10, 20]),
    getUniquePairs([1.1, 2.2, 3.3]),
    getUniquePairs(["w", "x", "y", "z"]), # vector of one element
Strings
    getUniquePairs(['a', 'b', 'c']), # vector of Chars
    getUniquePairs(['a', 'b', 'a']) # uniqueNames must be unique (of
course)
)
```

```
([(10, 20)],
 [(1.1, 2.2), (1.1, 3.3), (2.2, 3.3)],
 [("w", "x"), ("w", "y"), ("w", "z"), ("x", "y"), ("x", "z"), ("y",
"z")],
 [('a', 'b'), ('a', 'c'), ('b', 'c')],
 [('a', 'b'), ('a', 'a'), ('b', 'a')])
```

> ***Note:*** The group ("w", "x") is the same group as ("x", "w"). In other words, we don't care about the order of elements in a group. The function works correctly if `uniqueNames` argument contains unique elements (compare with the last example that contains a duplicate value). If you want you can add an additional check to make sure that the `uniqueNames` are really unique (think/search the internet how to do that), but I will leave it as it is.

OK, now it's time for `getPValsUnpairedTests`

```
```

[259]https://docs.julialang.org/en/v1/base/math/#Base.binomial

```
# df - DataFrame: each column is a continuous variable (one group)
# returns uncorrected p-values
function getPValsUnpairedTests(
    df::Dfs.DataFrame)::Dict{Tuple{String,String},Float64}

    pairs::Vector{Tuple{String,String}} = getUniquePairs(Dfs.names(df))
    pvals::Vector{Float64} = [
        getPValUnpairedTest(df[!, a], df[!, b])
        for (a, b) in pairs
    ]

    return Dict(pairs[i] => pvals[i] for i in eachindex(pairs))
end
```

First, we obtain the pairs of group names that we will compare later (`pairs`). In the next few lines we use a comprehension to obtain the p-values. Since each element of `pairs` vector is a tuple (e.g. `[("spA", "spB"), etc.]`) we assign its elements to `a` and `b` (`for (a, b)`) and pass them to `df` to get the values of interest (e.g. `df[!, a]`). The values are send to `getPValUnpairedTest` from the previous section. We terminate (`return`) with another comprehension that creates a dictionary with the desired result.

Let's see how the function works and compare the results with the ones we obtained in Section 5.5 .

```
getPValsUnpairedTests(miceBwtABC)
```

```
Dict{Tuple{String, String}, Float64} with 3 entries:
  ("spA", "spB") => 0.0251115
  ("spA", "spC") => 0.000398545
  ("spB", "spC") => 0.0493322
```

OK, the uncorrected p-values are the same as in Section 5.5 .

Now, the improved version.

```
# df - DataFrame: each column is a continuous variable (one group)
# returns corrected p-values
function getPValsUnpairedTests(
    df::Dfs.DataFrame,
    multCorr::Type{M}
```

```
)::Dict{Tuple{String,String},Float64} where {M<:Mt.PValueAdjustment}

    pairs::Vector{Tuple{String,String}} = getUniquePairs(Dfs.names(df))
    pvals::Vector{Float64} = [
        getPValUnpairedTest(df[!, a], df[!, b])
        for (a, b) in pairs
    ]
    pvals = Mt.adjust(pvals, multCorr())

    return Dict(pairs[i] => pvals[i] for i in eachindex(pairs))
end
```

Don't worry about the strange type declarations like `::Type{M}` and `where {M<:Mt.PValueAdjustment}`. I added them for the sake of consistency (after reading the code in the package repo[260] and some try and error). When properly called, the function should work equally well without those parts.

Anyway, it wasn't that bad, we basically just added a small piece of code (`multCorr` in the arguments list and `pvals = Mt.adjust(pvals, multCorr())` in the function body) similar to the one in Section 5.6 .

Let's see how it works.

```
# Bonferroni correction
getPValsUnpairedTests(miceBwtABC, Mt.Bonferroni)
```

```
Dict{Tuple{String, String}, Float64} with 3 entries:
  ("spA", "spB") => 0.0753345
  ("spA", "spC") => 0.00119563
  ("spB", "spC") => 0.147997
```

That looks quite alright. Time for one more swing.

```
# Benjamini-Hochberg correction
getPValsUnpairedTests(miceBwtABC, Mt.BenjaminiHochberg)
```

```
Dict{Tuple{String, String}, Float64} with 3 entries:
  ("spA", "spB") => 0.0376673
```

[260] https://github.com/juliangehring/MultipleTesting.jl

```
("spA", "spC") => 0.00119563
("spB", "spC") => 0.0493322
```

Again, the p-values appear to be the same as those we saw in Section 5.6 .

Solution to Exercise 5

OK, let's do this step by step. First let's draw a bare box-plot (no group names, no significance markers, titles, etc.).

The docs for Cmk.boxplot[261] show that to do that we need two vectors for xs and ys (values to be placed on the x- and y-axis respectively). Both need to be of numeric types. We can achieve it by typing, e.g.

```
# Step 1
ex5nrows = size(miceBwtABC)[1] #1
ex5names = Dfs.names(miceBwtABC) #2
ex5xs = repeat(eachindex(ex5names), inner=ex5nrows) #3
ex5ys = [miceBwtABC[!, n] for n in ex5names] #4
ex5ys = vcat(ex5ys...) #5

fig = Cmk.Figure()
ax1 = Cmk.Axis(fig[1, 1])
Cmk.boxplot!(ax1, ex5xs, ex5ys)
fig
```

In the first line (`#1`) we get the dimensions of our data frame, `size(miceBwtABC)` returns a tuple `(numberOfRows, numberOfColumns)` from which we take only the first part (`numberOfRows`) that we will need later. In line 3 (`#3`) we assign a number to the names (`eachindex(vect)` returns a sequence `1:length(vect)`, e.g. `[1, 2, 3]`). We multiply each number the same amount of times (`ex5nrows`) using `repeat` (e.g. `repeat([1, 2, 3], inner=2)` returns `[1, 1, 2, 2, 3, 3]`). In line 4 and 5 (`#4` and `#5`) we take all the body weights from columns and put them into a one long vector (`ex5ys`). We end up with two vectors: groups coded as integers

[261] https://docs.makie.org/stable/examples/plotting_functions/boxplot/index.html#boxplot

and body weights. Finally, we check if it works by running `Cmk.boxplot!(fig[1, 1], ex5xs, ex5ys)`. The result is below.

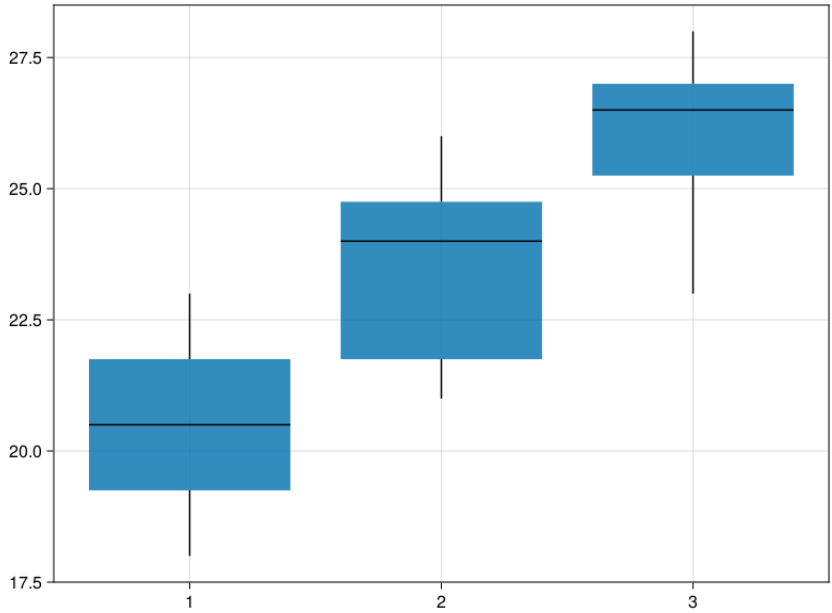

Figure 21: Figure 20: Box-plot for exercise 5. Step 1.

Now, let's add title, label the axes, etc.

```
# Step 2
fig = Cmk.Figure()
ax1 = Cmk.Axis(fig[1, 1],
               title="Body mass of three mice species",
               xlabel="species name", ylabel="body mass [g]",
               xticks=(eachindex(ex5names), ex5names))
Cmk.boxplot!(ax1, ex5xs, ex5ys, whiskerwidth=0.5)
fig
```

The new part here is the `xticks` argument. It takes a tuple of ticks on x axis (`1:3` in Figure 20) and a vector of strings (`ex5names`) to be displayed instead of those values. The meaning of `whiskerwidth` is pretty intuitive, it adds a horizontal bar of desired width at the end of the whiskers. The result is placed below.

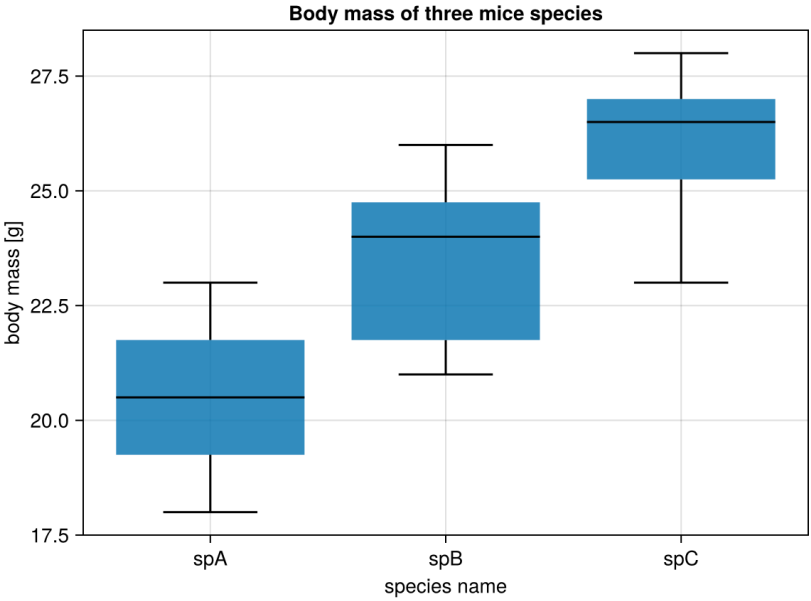

Figure 22: Figure 21: Box-plot for exercise 5. Step 2.

Let's move on to the significance markers. First, let's hard-code them and produce a plot (just to see if it works), then we will introduce some improvements.

```
# Step 3
fig = Cmk.Figure()
ax1 = Cmk.Axis(fig[1, 1],
               title="Body mass of three mice species",
               xlabel="species name", ylabel="body mass [g]",
               xticks=(eachindex(ex5names), ex5names))
Cmk.boxplot!(ax1, ex5xs, ex5ys, whiskerwidth=0.5)
Cmk.text!(ax1,
          eachindex(ex5names), [30, 30, 30],
          text=["", "a", "ab"],
          align=(:center, :top), fontsize=20)
fig
```

OK, we're almost there (see figure below).

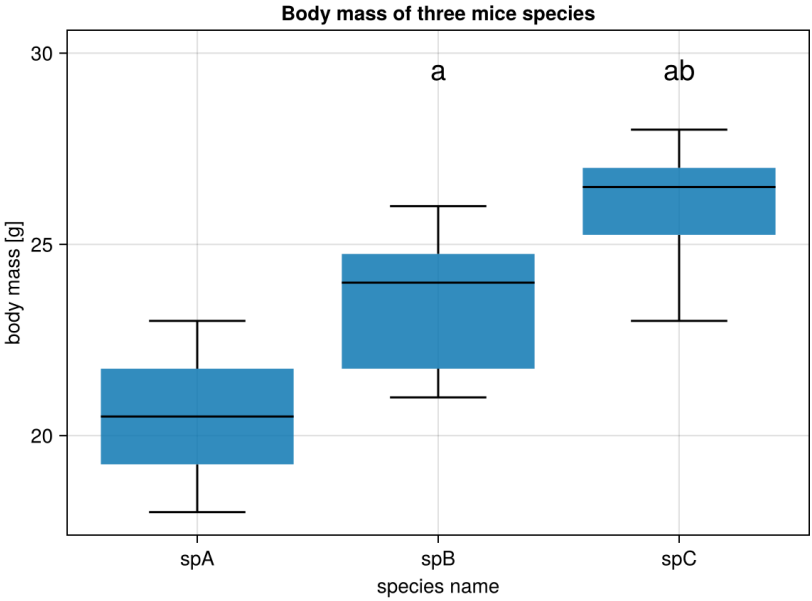

Figure 23: Figure 22: Box-plot for exercise 5. Step 3.

However, it appears that we still need a few things:

1) a way to generate y-values for `Cmk.text!` (for now it is `[30, 30, 30]`, but other dataframes may have different value ranges, e.g. [200-250] and then the markers would be placed too low)
2) a way to generate the markers (e.g. `["", "a", "ab"]` based on p-values) over the appropriate boxes

The first problem can be solved in the following way:

```
# Step 4
ex5marksYpos = [maximum(miceBwtABC[!, n]) for n in ex5names] #1
ex5marksYpos = map(mYpos -> round(Int, mYpos * 1.1), ex5marksYpos) #2
ex5upYlim = maximum(ex5ys * 1.2) |> x -> round(Int, x) #3
ex5downYlim = minimum(ex5ys * 0.8) |> x -> round(Int, x) #4
```

Here, in the first line (`#1`) we get maximum values from every group. Then (`#2`) we increase them by 10% (`* 1.1`) and round them to the closest integers (`round(Int,`). At this height (y-axis) we are going to place our significance markers. Additionally, in lines 3 and 4 (`#3` and

#4) we found the maximum and minimum values (for all the data). We increase (`* 1.2`) and decrease (`* 0.8`) the values by 20%. The rounded (to the nearest integer) values will be the maximum and minimum values displayed on the y-axis of our graph.

Now, time for a function that will translate p-values to significance markers.

```
# Step 5
function getMarkers(
    pvs::Dict{Tuple{String,String},Float64},
    groupsOrder=["spA", "spB", "spC"],
    markerTypes::Vector{String}=["a", "b", "c"],
    cutoffAlpha::Float64=0.05)::Vector{String}

    @assert (
        length(groupsOrder) == length(markerTypes)
    ) "different groupsOrder and markerTypes lengths"
    @assert (0 <= cutoffAlpha <= 1) "cutoffAlpha must be in range [0-1]"

    markers::Vector{String} = repeat([""], length(groupsOrder))
    tmpInd::Int = 0

    for i in eachindex(groupsOrder)
        for ((g1, g2), pv) in pvs
            if (groupsOrder[i] == g1) && (pv <= cutoffAlpha)
                tmpInd = findfirst(x -> x == g2, groupsOrder)
                markers[tmpInd] *= markerTypes[i]
            end
        end
    end

    return markers
end
```

Here, `getMarkers` accepts p-values in the format returned by `getPValsUnpairedTests` defined in Section 5.8.4 . Another input argument is `groupsOrder` which contains the position of groups (boxes, x-axis labels) in Figure 22 from left to right. The third argument is `makrerTypes` so a symbol that is to be used if a statistical difference for a given group is found.

The function defines `markers` (the strings placed over each box with `Cmk.txt`) initialized with a vector of empty strings. Next, it walks through each index in group (`eachindex(groups)`) and checks the

`((g1, g2), pv)` in p-values (`pvs`). If `g1` is equal to the examined group (`groups[i] == g1`) and the p-value (`pv`) is $\leq$ the cutoff level then the appropriate marker (`markerTypes[i]`) is inserted by string concatenation[262] with an update operator[263] (`*=`). Which maker to change is determined by the index of `g2` in the `groups` returned by findfirst[264] function. In general, `g2` receives a marker when it is statistically different from `g1` (`pv < cutoffAlpha`).

Let's test our function

```
(
getMarkers(
    getPValsUnpairedTests(miceBwtABC, Mt.BenjaminiHochberg),
    ["spA", "spB", "spC"],
    ["a", "b", "c"],
    0.05),

getPValsUnpairedTests(miceBwtABC, Mt.BenjaminiHochberg)
)
```

```
(["", "a", "ab"],
Dict(("spA", "spB") => 0.0376672521079031,
("spA", "spC") => 0.001195633577293774,
("spB", "spC") => 0.049332195639921715))
```

The markers appear to be OK (they reflect the p-values well).

Now, it is time to pack it all into a separate function

```
# Step 6

# the function should work fine for up to 26 groups in the df's columns
function drawBoxplot(
    df::Dfs.DataFrame, title::String,
    xlabel::String, ylabel::String)::Cmk.Figure

    nrows, _ = size(df)
    ns::Vector{String} = Dfs.names(df)
```

[262] https://docs.julialang.org/en/v1/manual/strings/#man-concatenation

[263] https://docs.julialang.org/en/v1/manual/mathematical-operations/#Updating-operators

[264] https://docs.julialang.org/en/v1/base/strings/#Base.findfirst-Tuple%7BAbstractString,%20AbstractString%7D

```
    xs = repeat(eachindex(ns), inner=nrows)
    ys = [df[!, n] for n in ns]
    ys = vcat(ys...)
    marksYpos = [maximum(df[!, n]) for n in ns]
    marksYpos = map(mYpos -> round(Int, mYpos * 1.1), marksYpos)
    upYlim = maximum(ys * 1.2) |> x -> round(Int, x)
    downYlim = minimum(ys * 0.8) |> x -> round(Int, x)
    # 'a':'z' generates all lowercase chars of the alphabet
    markerTypes::Vector{String} = map(string, 'a':'z')
    markers::Vector{String} = getMarkers(
        getPValsUnpairedTests(df, Mt.BenjaminiHochberg),
        ns,
        markerTypes[1:length(ns)],
        0.05
    )

    fig = Cmk.Figure()
    ax1 = Cmk.Axis(fig[1, 1],
                   title=title, xlabel=xlabel, ylabel=ylabel,
                   xticks=(eachindex(ns), ns))
    Cmk.boxplot!(ax1, xs, ys, whiskerwidth=0.5)
    Cmk.ylims!(ax1, downYlim, upYlim)
    Cmk.text!(ax1,
              eachindex(ns), marksYpos,
              text=markers, align=(:center, :top), fontsize=20)

    return fig
end
```

and run it

```
drawBoxplot(miceBwtABC,
    "Body mass of three mice species",
    "species name",
    "body mass [g]"
)
```

And voilà this is your result

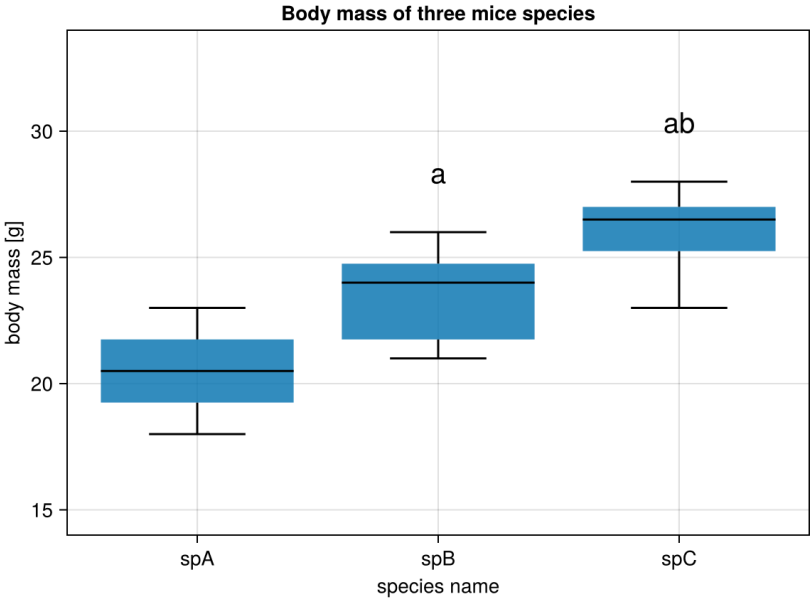

Figure 24: Figure 23: Boxplot of body mass of three mice species (fictitious data). Steps 1-6 (completed). a - difference vs. spA ($p < 0.05$), b - difference vs. spB ($p < 0.05$).

Once again (we said this already in the task description see Section 5.7.5). In the graph above a middle horizontal line in a box is the median[265], a box depicts interquartile range[266](IQR), the whiskers length is equal to 1.5 * IQR (or the maximum and minimum if they are smaller than 1.5 * IQR).

You could make the function more plastic, e.g. by moving some of its insides to its argument list. But this form will do for now. You may want to test the function with some other output, even with `miceBwt` from Section 5.3 (here it should draw a box-plot with no statistical significance markers).

[265]https://en.wikipedia.org/wiki/Median

[266]https://en.wikipedia.org/wiki/Interquartile_range

Note: The code we developed in the exercises (e.g. `getPValsUnpairedTests`, `drawBoxplot`) is to help us automate stuff, still it shouldn't be applied automatically (think before you leap).

Comparisons - categorical data

OK, once we have comparisons of continuous data under our belts we can move to groups of categorical data.

Chapter imports

Later in this chapter we are going to use the following libraries

```
import CairoMakie as Cmk
import DataFrames as Dfs
import Distributions as Dsts
import HypothesisTests as Ht
import MultipleTesting as Mt
import Random as Rand
```

If you want to follow along you should have them installed on your system. A reminder of how to deal (install and such) with packages can be found here[267]. But wait, you may prefer to use `Project.toml` and `Manifest.toml` files from the code snippets for this chapter[268]to install the required packages. The instructions you will find here[269].

The imports will be placed in the code snippet when first used, but I thought it is a good idea to put them here, after all imports should be at the top of your file (so here they are at the top of the chapter). Moreover, that way they will be easier to find all in one place.

If during the lecture of this chapter you find a piece of code of unknown functionality, just go to the code snippets mentioned above and run the code from the `*.jl` file. Once you have done that you can always extract a small piece of it and test it separately (modify and experiment with it if you wish).

Flashback

We deal with categorical data when a variable can take a value from a small set of values. Each element of the set is clearly distinct from the

[267]https://docs.julialang.org/en/v1/stdlib/Pkg/
[268]https://github.com/b-lukaszuk/RJ_BS_eng/tree/main/code_snippets/ch06
[269]https://pkgdocs.julialang.org/v1/environments/

other elements. For instance the results of coin tosses or dice rolls fall into one of a few distinctive categories. As stated in Section 4 and its subsections the result of a coin toss often displays the binomial distribution. In line with that notion, in Exercise 3 (see Section 4.8.3 and Section 4.9.3) we calculated the probability that Peter is a better tennis player than John if he won 5 games out of 6. The two-tailed probability was roughly equal to 0.22. Once we know the logic behind the calculations (see Section 4.9.3) we can fast forward to the solution with Ht.BinomialTest[270]like so

```
import HypothesisTests as Ht

Ht.BinomialTest(5, 6, 0.5)
# or just: Ht.BinomialTest(5, 6)
# since 0.5 is the default prob. for the population
```

```
Binomial test
-------------
Population details:
    parameter of interest:   Probability of success
    value under h_0:         0.5
    point estimate:          0.833333
    95% confidence interval: (0.3588, 0.9958)

Test summary:
    outcome with 95% confidence: fail to reject h_0
    two-sided p-value:           0.2187

Details:
    number of observations: 6
    number of successes:    5
```

Works like a charm. Don't you think. Here we got a two-tailed p-value. By oversimplifying stuff we can say that the 95% confidence interval is an estimate of the true probability of Peter's victory in a game (from data it is 5/6 = 0.83) and it includes 0.5 (our probability under H_0 = 0.5). I leave the rest of the output to decipher to you (as a mini-exercise).

[270]https://juliastats.org/HypothesisTests.jl/stable/nonparametric/#Binomial-test

In general `Ht.BinomialTest` is useful when you want to compare the obtained experimental result that may fall into one of two categories (generally called: success or failure) with a theoretical binomial distribution with a known probability of success (we check if the obtained result is compatible with that distribution). If we interpret this statement in a more creative way we may find other use cases for the test.

Let's look at an interesting example from the field of biological sciences. Imagine that there is some disease that you want to study. Its prevalence in the general population is estimated to be $\approx \frac{10}{100} = 0.1 = 10\%$. You happened to found a human population on a desert island and noticed that 519 adults out of 3'202 suffer from the disease of interest. You run the test to see if that differs from the general population [here success (if I may call it so) is the presence of the disease, and theoretical distribution is the distribution of the disease in the general population].

```
Ht.BinomialTest(519, 3202, 0.1)
```

```
Binomial test
-------------
Population details:
    parameter of interest:   Probability of success
    value under h_0:         0.1
    point estimate:          0.162086
    95% confidence interval: (0.1495, 0.1753)

Test summary:
    outcome with 95% confidence: reject h_0
    two-sided p-value:           <1e-26

Details:
    number of observations: 3202
    number of successes:    519
```

And it turns out that it does. Congratulations, you discovered a local population with a different, clearly higher prevalence of the disease. Now you (or other people) can study the population closer (e.g. gene screening) in order to find the features that trigger the onset of (or predispose to develop) the disease.

The story is not that far fetched since there are human populations that are of particular interest to scientists due to their unusually common occurrence of some diseases (e.g. the Akimel O'odham[271] and their high prevalence of type 2 diabetes[272]).

Chi squared test

We finished the previous section by comparing the proportion of subjects with some feature to the reference population. For that we used `Ht.BinomialTest`. As we learned in Section 4.6 the word binomial means two names. Those names could be anything, like heads and tails, victory and defeat, but most generally they are called success and failure (success when an event occurred and failure when it did not). We can use `a` to denote individuals with the feature of interest and `b` to denote the individuals without that feature. In that case `n` is the total number of individuals (here, individuals with either `a` or `b`). That means that by doing `Ht.BinomialTest` we compared the sample fraction (e.g. $\frac{a}{n}$ or equivalently $\frac{a}{a+b}$) with the assumed fraction of individuals with the feature of interest in the general population.

Now, imagine a different situation. You take the samples from two populations, and observe the eye color[273] of people. You want to know if the percentage of people with blue eyes in the two populations is similar. If it is, then you may deduce they are closely related (perhaps one stems from the other). Let's not look too far, let's just take the population of the US and UK. Inspired by the Wikipedia's page from the link above and supported by the random number generator in Julia I came up with the following counts.

```
import DataFrames as Dfs

dfEyeColor = Dfs.DataFrame(
    Dict(
        "eyeCol" => ["blue", "any"],
        "us" => [161, 481],
        "uk" => [220, 499]
```

[271] https://en.wikipedia.org/wiki/Akimel_O%27odham
[272] https://en.wikipedia.org/wiki/Type_2_diabetes
[273] https://en.wikipedia.org/wiki/Eye_color

```
    )
)
```

eyeCol	uk	us
blue	220	161
any	499	481

Table 5: Table 5: Eye color distribution in two samples (fictitious data).

Here, we would like to compare if the two proportions ($\frac{a_1}{n_1} = \frac{161}{481}$ and $\frac{a_2}{n_2} = \frac{220}{499}$) are roughly equal ($H_0$: they come from the same population with some fraction of blue eyed people). Unfortunately, one look into the docs[274]and we see that we cannot use `Ht.BinomialTest` (the test compares sample with a population, here we got two samples to compare). But do not despair that's the job for Ht.ChisqTest[275](see also this Wikipedia's entry[276]). First we need to change our data slightly, because the test requires a matrix (aka array from Section 3.3.7) with the following proportions in columns: $\frac{a_1}{b_1}$ and $\frac{a_2}{b_2}$ (`b` instead of `n`, where `n` = `a` + `b`). Let's adjust our data for that.

```
# subtracting eye color "blue" from eye color "any"
dfEyeColor[2, 2:3] = Vector(dfEyeColor[2, 2:3]) .-
    Vector(dfEyeColor[1, 2:3])
# renaming eye color "any" to "other" (it better reflects current
content)
dfEyeColor[2, 1] = "other"
dfEyeColor

# all the elements must be of the same (numeric) type
mEyeColor = Matrix{Int}(dfEyeColor[:, 2:3])
mEyeColor
```

```
2×2 Matrix{Int64}:
 220  161
 279  320
```

[274]https://juliastats.org/HypothesisTests.jl/stable/nonparametric/#Binomial-test

[275]https://juliastats.org/HypothesisTests.jl/stable/parametric/#Pearson-chi-squared-test

[276]https://en.wikipedia.org/wiki/Chi-squared_test

OK, we got the necessary data structure. Here, `Matrix{Int}()` closed over `dfEyeColor[:, 2:3]` extracts the needed part of the data frame and converts it to a matrix (aka array) of integers. And now for the χ^2 (chi squared) test.

```
Ht.ChisqTest(mEyeColor)
```

```
Pearson's Chi-square Test
-------------------------
Population details:
    parameter of interest:   Multinomial Probabilities
    value under h_0:         [0.197958, 0.311226, 0.190817, 0.299999]
    point estimate:          [0.22449, 0.284694, 0.164286, 0.326531]
    95% confidence interval:
     [(0.193, 0.2595), (0.2501, 0.322), (0.1369, 0.196), (0.2903,
0.3649)]

Test summary:
    outcome with 95% confidence: reject h_0
    one-sided p-value:           0.0007

Details:
    Sample size:        980
    statistic:          11.616133413434031
    degrees of freedom: 1
    residuals:          [1.86677, -1.48881, -1.90138, 1.51641]
    std. residuals:     [3.40824, -3.40824, -3.40824, 3.40824]
```

OK, first of all we can see right away that the p-value is below the customary cutoff level of 0.05 or even 0.01. This means that the samples do not come from the same population (we reject H_0). More likely they came from the populations with different underlying proportions of blue eyed people. This could indicate for instance, that the population of the US stemmed from the UK (at least partially) but it has a greater admixture of other cultures, which could potentially influence the distribution of blue eyed people. Still, this is just an exemplary explanation, I'm not an anthropologist, so it may well be incorrect. Additionally, remember that the data is fictitious and was generated by me.

Anyway, I'm pretty sure You got the part with the p-value on your own, but what are some of the other outputs. Point estimates are the observed probabilities in each of the cells from `mEyeColor`. Observe

```
# total number of observations
nObsEyeColor = sum(mEyeColor)

chi2pointEstimates = [mEyeColor...] ./ nObsEyeColor
round.(chi2pointEstimates, digits = 6)
```

```
[0.22449, 0.284694, 0.164286, 0.326531]
```

The `[mEyeColor...]` flattens the 2x2 matrix (2 rows, 2 columns) to a vector (column 2 is appended to the end of column 1). The `./ nObsEyeColor` divides the observations in each cell by the total number of observations.

`95% confidence interval` is a 95% confidence interval (who would have guessed) similar to the one explained in Section 5.2.1 for `Ht.OneSampleTTest` but for each of the point estimates in `chi2pointEstimates`. Some (over)simplify it and say that within those limits the true probability for this group of observations most likely lies.

As for the `value under h_0` those are the probabilities of the observations being in a given cell of `mEyeColor` assuming H_0 is true. But how to get that probabilities. Well, in a similar way to the method we met in Section 4.3 . Back then we answered the following question: If parents got blood groups AB and O then what is the probability that a child will produce a gamete with allele `A`? The answer: proportion of children with allele `A` and then the proportion of their gametes with allele `A` (see Section 4.3 for details). We calculated it using the following formula

$$P(A\ in\ CG) = P(A\ in\ C)*P(A\ in\ gametes\ of\ C\ with\ A)$$

Getting back to our `mEyeColor` the expected probability of an observation falling into a given cell of the matrix is the probability of

an observation falling into a given column times the probability of an observation falling into a given row. Observe

```
# cProbs - probability of a value to be found in a given column
cProbs = [sum(c) for c in eachcol(mEyeColor)] ./ nObsEyeColor
# rProbs - probability of a value to be found in a given row
rProbs = [sum(r) for r in eachrow(mEyeColor)] ./ nObsEyeColor

# probability of a value to be found in a given cell of mEyeColor
# under H_0 (the samples are from the same population)
probsUnderH0 = [cp * rp for cp in cProbs for rp in rProbs]
round.(probsUnderH0, digits = 6)
```

```
[0.197958, 0.311226, 0.190817, 0.299999]
```

Here, `[cp * rp for cp in cProbs for rp in rProbs]` is an example of nested for loops[277] enclosed in a comprehension. Notice that in the case of this comprehension there is no comma before the second `for` (the comma is present in the long, non-comprehension version of nested for loops in the link above).

Anyway, note that since the calculations from Section 4.3 assumed the probability independence, then the same assumption is made here. That means that, e.g. a given person cannot be classified at the same time as the citizen of the US and UK since we would have openly violated the assumption (some countries allow double citizenship, so you should think carefully about the inclusion criteria for the categories). Moreover, the eye color also needs to be a clear cut.

Out of the remaining output we are mostly interested in the `statistic`, namely χ^2 (chi square) statistic. Under the null hypothesis (H_0, both groups come from the same population with a given fraction of blue eyed individuals) the probability distribution for counts to occur is called χ^2 (chi squared) distribution. Next, we calculate χ^2 (chi squared) statistic for the observed result (from `mEyeColor`). Then, we obtain the probability of a statistic greater than that to occur by

[277] https://en.wikibooks.org/wiki/Introducing_Julia/Controlling_the_flow#Nested_loops

chance. This is similar to the F-Statistic (Section 5.4) and L-Statistic (Section 5.8.2) we met before. Let's see this in practice

```
observedCounts = [mEyeColor...]
expectedCounts = probsUnderH0 .* nObsEyeColor
# the statisticians love squaring numbers, don't they
chi2Diffs = ((observedCounts .- expectedCounts) .^2) ./ expectedCounts
chi2Statistic = sum(chi2Diffs)

(
    observedCounts,
    round.(expectedCounts, digits = 4),
    round.(chi2Diffs, digits = 4),
    round(chi2Statistic, digits = 4)
)
```

```
([220, 279, 161, 320],
[193.999, 305.001, 187.001, 293.999],
[3.4848, 2.2166, 3.6152, 2.2995],
11.6161)
```

The code is rather self explanatory. BTW. You might have noticed that: a) statisticians love squaring numbers (differences), and b) there are some similarities to the calculations of expected values from Section 4.5 . Anyway, now, we can use the χ^2 statistic to get the p-value, like so

```
import Distributions as Dsts

function getDf(matrix::Matrix{Int})::Int
    nRows, nCols = size(matrix)
    return (nRows - 1) * (nCols - 1)
end

# p-value
# alternative: Dsts.ccdf(Dsts.Chisq(getDf(mEyeColor)), chi2Statistic)
1 - Dsts.cdf(Dsts.Chisq(getDf(mEyeColor)), chi2Statistic) |>
    x -> round(x, digits = 4)
```

0.0007

So, the pattern is quite similar to what we did in the case of F-Distribution/Statistic in Section 5.7.2 . First we created the distribution of interest with the appropriate number of the degrees of freedom

(why only the degrees of freedom matter see the conclusion of Section 5.8.2). Then we calculated the probability of a χ^2 Statistic being greater than the observed one by chance alone and that's it.

Fisher's exact test

This was all nice, but there is a small problem with the χ^2 test, namely it relies on some approximations and works well only for large sample sizes. How large, well, I've heard about the rule of fives (that's what I called it). The rule states that there should be >= 50 (not quite 5) observations per matrix and >= 5 expected observations per cell (applies to every cell). In case this assumption does not hold, one should use, e.g. Fisher's exact test[278](Fisher, yes, I think I heard that name before).

So let's assume for a moment that we were able to collect somewhat less data like in the matrix below:

```
mEyeColorSmall = round.(Int, mEyeColor ./ 20)
mEyeColorSmall
```

```
2×2 Matrix{Int64}:
 11   8
 14  16
```

Here, we reduced the number of observations 20 times compared to the original `mEyeColor` matrix from the previous section. Since the test we are going to apply (Ht.FisherExactTest[279]) requires integers then instead of rounding a number to 0 digits [e.g. `round(12.3, digits = 0)` would return 12.0, so `Float64`] we asked the round function to deliver us the closest integers (e.g. 12).

OK, let's, run the said `Ht.FisherExactTest`. Right away we see a problem, the test requires separate integers as input: `Ht.FisherExactTest(a::Integer, b::Integer, c::Integer, d::Integer)`.

[278]https://en.wikipedia.org/wiki/Fisher%27s_exact_test

[279]https://juliastats.org/HypothesisTests.jl/stable/nonparametric/#Fisher-exact-test

Note: Just like `Real` type from Section 3.4 also `Integer` is a supertype. It encompasses, e.g. `Int` and `BigInt` we met in Section 3.9.5 .

Still, we can obtain the necessary results very simply, by:

```
# assignment goes column by column (left to right), value by value
a, c, b, d = mEyeColorSmall

Ht.FisherExactTest(a, b, c, d)
```

```
Fisher's exact test
-------------------
Population details:
    parameter of interest:   Odds ratio
    value under h_0:         1.0
    point estimate:          1.55691
    95% confidence interval: (0.4263, 5.899)

Test summary:
    outcome with 95% confidence: fail to reject h_0
    two-sided p-value:           0.6373

Details:
    contingency table:
        11   8
        14  16
```

We are not going to discuss the output in detail. Still, we can see that here due to the small sample size we don't have enough evidence to reject the H_0 (p > 0.05) on favor of H_A. Interestingly, due to the small sample size we came to a different conclusion despite the same underlying populations and the same proportions. Let's make an analogy here and let's take it to an extreme. Imagine I got two coins in my pocket, one fair (50/50 heads to tails rate) and one biased (70/30 heads to tails ratio). I give you one to find out which coin it is. That's easy to settle out with 1'000 tosses (since you wold get, e.g. 688/312 heads to tails ratio instead of 494/506), but it is not possible to do it with just one toss (no matter the outcome). With three tosses and two heads we still cannot be sure of it since a fair coin would have produced this exact output with the probability of 37.5% (`HHT`, or `THH`,

or `HTH` each with p = $\frac{1}{2^3} = \frac{1}{8} = 0.125$) and more extreme (`HHH`) with the probability = 12.5% ($\frac{1}{2^3} = \frac{1}{8}$ = 0.125). So, there just wouldn't be enough evidence.

Bigger table

We started Section 6.3 with a fictitious eye color distribution [`blue` and `other`, rows (top-down) in the matrix below] in the US and UK [columns (left-right) in the matrix below].

```
mEyeColor
```

```
2×2 Matrix{Int64}:
 220  161
 279  320
```

But in reality there are more eye colors than just blue and other. For instance let's say that in humans we got three types of eye color: blue, green, and brown. Let's adjust our table for that:

```
# 3 x 2 table (DataFrame)
dfEyeColorFull = Dfs.DataFrame(
    Dict(
        # "other" from dfEyeColor is split into "green" and "brown"
        "eyeCol" => ["blue", "green", "brown"],
        "us" => [161, 78, 242],
        "uk" => [220, 149, 130]
    )
)

mEyeColorFull = Matrix{Int}(dfEyeColorFull[:, 2:3])
mEyeColorFull
```

```
3×2 Matrix{Int64}:
 220  161
 149   78
 130  242
```

Can we say that the two populations differ (with respect to the eye color distribution) given the data in this table? Well, we can, that's the job for ... chi squared (χ^2) test.

Wait, but I thought it is used to compare two proportions found in some samples. Granted, it could be used for that, but in broader sense it is a non-parametric test that determines the probability that the difference between the observed and expected frequencies (counts) occurred by chance alone. Here, non-parametric means it does not assume a specific underlying distribution of data (like the normal or binomial distribution we met before). As we learned in Section 6.3 the expected distribution of frequencies (counts) is assessed based on the data itself.

Let's give it a try with our new data set (`mEyeColorFull`) and compare it with the previously obtained results (for `mEyeColor` from Section 6.3).

```
chi2testEyeColor = Ht.ChisqTest(mEyeColor)
chi2testEyeColorFull = Ht.ChisqTest(mEyeColorFull)

(
    # chi^2 statistics
    round(chi2testEyeColorFull.stat, digits = 2),
    round(chi2testEyeColor.stat, digits = 2),

    # p-values
    round(chi2testEyeColorFull |> Ht.pvalue, digits = 7),
    round(chi2testEyeColor |> Ht.pvalue, digits = 7)
)
```

```
(64.76, 11.62,
0.0, 0.0006538)
```

That's odd. All we did was to split the `other` category from `dfEyeColor` (and therefore `mEyeColor`) into `green` and `brown` to create `dfEyeColorFull` (and therefore `mEyeColorFull`) and yet we got different χ^2 statistics, and different p-values. How come?

Well, because we are comparing different things (and different populations).

Imagine that in the case of `dfEyeColor` (and `mEyeColor`) we actually compare not the eye color, but currency of both countries. So, we change the labels in our table. Instead of `blue` we got `heads` and

instead of `other` we got `tails` and instead of `us` we got eagle[280]and instead of `uk` we got one pound[281]. We want to test if the proportion of heads/tails is roughly the same for both the coins.

Whereas in the case of `dfEyeColorFull` (and `mEyeColorFull`) imagine we actually compare not the eye color, but three sided dice[282]produced in those countries. So, we change the labels in our table. Instead of `blue` we got `1` and instead of `green` we got `2`, instead of `brown` we got `3` (`1`, `2`, `3` is a convention, equally well one could write on the sides of a dice, e.g. `Tom`, `Alice`, and `John`). We want to test if the distribution of `1`s, `2`s, and `3`s is roughly the same for both types of dice.

Now, it so happened that the number of dice throws was the same that the number of coin tosses from the example above. It also happened that the number of `1`s was the same as the number of `heads` from the previous example. Still, we are comparing different things (coins and dices) and so we would not expect to get the same results from our chi squared (χ^2) test. And that is how it is, the test is label blind. All it cares is the difference between the observed and expected frequencies (counts).

Anyway, the value of χ^2 statistic for `mEyeColorFull` is 64.76 and the probability that such a value occurred by chance approximates 0. Therefore, it is below our customary cutoff level of 0.05, and we may conclude that the populations differ with respect to the distribution of eye color (as we did in Section 6.5).

Now, let's get back for a moment to the label blindness issue. The test may be label blind, but we are not. It is possible that sooner or later you will come across a data set where splitting groups into different categories will lead you to different conclusions, e.g. p-value from χ^2 test for `mEyeColorPlSp` for Poland and Spain would be 0.054, and for

[280]https://en.wikipedia.org/wiki/Eagle_(United_States_coin)

[281]https://en.wikipedia.org/wiki/One_pound_(British_coin)

[282]https://www.google.com/search?sca_esv=571684704&q=three+sided+dice&tbm=isch&source=lnms&sa=X&ved=2ahUKEwj1k-bB-uWBAxUa3AIHHWDvDoIQ0pQJegQIDBAB&biw=1437&bih=696&dpr=1.33

`mEyeColorPlSpFull` it would be 0.042 (so it is and it isn't statistically different at the same time). What should you do then?

Well, it happens. There is not much to be done here. We need to live with that. It is like the accused and judge analogy from Section 4.7.5 . In reality the accused is guilty or not. We don't know the truth, the best we can do is to examine the evidence. After that one judge may incline to declare the accused guilty the other will give him the benefit of doubt. There is no certainty or a great solution here (at least I don't know it). In such a case some people suggest to present both the results with the author's conclusions and let the readers decide for themselves. Others suggest to collect a greater sample to make sure which conclusion is right. Still, others suggest that you should plan your experiment (its goals and the ways to achieve them) carefully beforehand. Once you got your data you stick to the plan even if the result is disappointing to you. So, if we had decided to compare `blue` vs `other` and failed to establish the statistical significance we ought stopped there. We should not go fishing for statistical significance by splitting `other` to `green` and `brown`.

Test for independence

Another way to look at the chi squared (χ^2) test is that this is a test that allows to check the independence of the distribution of the data between the rows and columns (see the assumption we made when calculating the expected counts with `probsUnderH0` in Section 6.3). Let's make this more concrete with the following example.

Previously we concerned ourselves with the `mEyeColorFull` table.

```
mEyeColorFull
```

```
3×2 Matrix{Int64}:
 220  161
 149   78
 130  242
```

The rows contain (top to bottom) eye colors: `blue`, `green`, and `brown`. The columns (left to right) are for `us` and `uk`.

Interestingly enough, the eye color depends on the concentration of melanin[283], a pigment that is also present in skin and hair and protects us from the harmful UV radiation. So imagine that the columns contain the data for some skin condition (left column: `diseaseX`, right column: `noDiseaseX`). Now, we are interested to know, if people with a certain eye color are more exposed (more vulnerable) to the disease (if so then some preventive measures, e.g. a stronger sun screen, could be applied by them).

Since this is a fictitious data set on which we only changed the column labels then we already know the answer (see the reminder from Section 6.5 below)

```
(
    round(chi2testEyeColorFull.stat, digits = 2),
    round(chi2testEyeColorFull |> Ht.pvalue, digits = 7)
)
```

```
(64.76, 0.0)
```

OK, so based on the (fictitious) data there is enough evidence to consider that the occurrence of `diseaseX` isn't independent from eye color ($p \leq 0.05$). In other words, people of some eye color get `diseaseX` more often than people with some other eye color. But which eye color (`blue`, `green`, `brown`) carries the greater risk? Pause for a moment and think how to answer the question.

Well, one thing we could do is to collapse some rows (if it makes sense), for instance we could collapse `green` and `brown` into `other` category (we would end up with two eye colors: `blue` and `other`). So in practice we would answer the same question that we did in Section 6.3 for `mEyeColor` (of course here we changed column labels to `diseaseX` and `noDiseaseX`).

```
rowPerc = [r[1] / sum(r) * 100 for r in eachrow(mEyeColor)]
rowPerc = round.(rowPerc, digits = 2)
```

[283] https://en.wikipedia.org/wiki/Melanin

```
(
    round(chi2testEyeColor.stat, digits = 2),
    round(chi2testEyeColor |> Ht.pvalue, digits = 7),
    rowPerc
)
```

```
(11.62, 0.0006538, [57.74, 46.58])
```

We see that roughly 57.74% of `blue` eyed people got `diseaseX` compared to roughly 46.58% of people with `other` eye color and that the difference is statistically significant ($p \leq 0.05$). So people with `other` eye color should be more careful with exposure to sun (of course, these are just made up data).

Another option is to use a method analogous to the one we applied in Section 5.4 and Section 5.5 . Back then we compared three groups of continuous variables with one-way ANOVA [it controls for the overall α (type 1 error)]. Then we used a post-hoc tests (Student's t-tests) to figure out which group(s) differ(s) from the other(s). Naturally, we could/should adjust the obtained p-values by using a multiplicity correction (as we did in Section 5.6). This is exactly what we are going to do in the upcoming exercises (see Section 6.7.5 and Section 6.7.6). For now take some rest and click the right arrow when you're ready.

Exercises - Comparisons of Categorical Data

Just like in the previous chapters here you will find some exercises that you may want to solve to get from this chapter as much as you can (best option). Alternatively, you may read the task descriptions and the solutions (and try to understand them).

Exercise 1

In Section 6.3 and Section 6.5 we dealt with `dfEyeColor` and `dfEyeColorFull`, i.e. the data sets that were already in the form of a contingency table. Usually, this is not the case.

Imagine that you are a researcher and you want to find out if certain professions are associated with a greater risk of smoking cigarettes (perhaps as a way to alleviate the stress). So you prepare a questionnaire. People answer two questions: "Q1. What is your profession?" and "Q2. Do you smoke?". The answers to Q1 are placed in one column of a spreadsheet, the answers to Q2 are placed into another column. An exemplary data could look this way:

```
import Random as Rand

Rand.seed!(321)
smoker = Rand.rand(["no", "yes"], 100)
profession = Rand.rand(["Lawyer", "Priest", "Teacher"], 100)
```

Write a function with the following signature

```
function getContingencyTable(
    rowVect::Vector{String},
    colVect::Vector{String},
    )::Matrix{Int}
```

The function should take two arguments (observations as vectors of strings) and return a contingency table (`Matrix{Int}`) with the counts (similar to `mEyeColor` or `mEyeColorFull`). You may modify the function slightly, e.g to return `Dfs.DataFrame` similar to the one produced by FreqTables.freqtable[284](it doesn't have to be exact).

Test your function with the data presented above. Make sure it works properly also for smaller data sets, i.e.

```
Rand.seed!(321)
smokerSmall = Rand.rand(["no", "yes"], 10)
professionSmall = Rand.rand(["Lawyer", "Priest", "Teacher"], 10)
```

Here, the contingency table should contain zeros in some cells.

[284]https://github.com/nalimilan/FreqTables.jl

Below you may find a list of functions that I found useful (you may check them in the docs[285]). Of course you don't have to use any of them. The functions are sorted alphabetically.

- `Dfs.insertcols!` (DataFrames docs[286])
- `collect`
- `getCounts` (from Section 4.4)
- `sort`
- `unique`
- `zip`

Exercise 2

In Section 6.3 we concluded that the populations of the `us` and `uk` differ with respect to eye color distribution (we used data from `mEyeColor`).

Still, it's often nice to know not just the numbers themselves, but the proportions (or percentage distribution of the data in a table).

So, here is a task for you. Write the following functions

```
function getColPerc(m::Matrix{Int})::Matrix{Float64}

# and

function getRowPerc(m::Matrix{Int})::Matrix{Float64}
```

that should work similarly to FreqTables.prop[287](`prop(tbl2, margins=2)`, and `prop(tbl2, margins=1)`), i.e they should return the column and row percentage of observations, respectively.

To reduce code duplication you may want to combine them into a single function, e.g. `getPerc(m::Matrix{Int}, byRow::Bool)::Matrix{Float64}` that returns row percentages when `byRow` is true, and column percentages otherwise. You my also want to round the numbers (percents) to e.g. 2 decimal points.

[285] https://docs.julialang.org/en/v1/

[286] https://dataframes.juliadata.org/stable/

[287] https://github.com/nalimilan/FreqTables.jl

In my solution I used nested for loops[288], but feel free to write it whatever way you like (as long as it works fine).

Exercise 3

The functions we developed previously (see Section 6.8.2) are nice and useful. Still, we might want to have a visual aid to help us with the interpretation of our data.

So here is another task for you. Using CairoMakie or your favorite plotting library write a function that accepts a data frame like `dfEyeColorFull` and draws a stacked bar plot depicting column percentages (search the documentation for barplot[289]).

You may use the functions we developed before.

If you want, you can make your function also draw row percentages (optional).

Exercise 4

This exercise is pretty easy and straightforward. In Section 6.4 we said that the chi squared (χ^2) test requires the table to fulfill a few assumptions, e.g.:

- total number of observations to be >= 50
- the expected number of observations per a cell to be >= 5

So here is the task. Write a function with the following signature

```
runCategTestGetPVal(m::Matrix{Int})::Float64
# or
runCategTestGetPVal(df::Dfs.DataFrame)::Float64
```

The function takes a 2x2 matrix (like `mEyeColor` or `mEyeColorSmall`) or a data frame (like `dfEyeColor`). Then the function tests the above mentioned assumptions and runs `Ht.ChisqTest` or `Ht.FisherExactTest` on its input and returns the obtained p-value.

[288] https://en.wikibooks.org/wiki/Introducing_Julia/Controlling_the_flow#Nested_loops

[289] https://docs.makie.org/stable/reference/plots/barplot/

Feel free to use the functionalities we developed in this chapter (Section 6) and its sub-chapters.

Exercise 5

In Section 6.6 we analyzed the data in `dfEyeColorFull` (alternatively `mEyeColorFull`) and concluded that the distribution of eye color between the two tested countries differed. Still, we were unable to tell which (two eye colors) distributions differ from each other.

So here is the task. Write a function that accepts a matrix (or a data frame if you will) like `mEyeColor`/`dfEyeColorFull` (where the number of rows and/or columns with counts is greater than 2). The function should return a vector of all possible 2x2 matrices/data frames (I found `getUniquePairs` from Section 5.8.4 to be useful here, but you may use whatever you want).

Once you got the data structure with the data frames write another function that runs the appropriate test (`runCategTestGetPVal` from Section 6.7.4 above) on each of the matrices/data frames from the previous paragraph and return the p-values (choose the appropriate data structure).

In the last step write a function that applies a multiplicity correction (see Section 5.6) to the obtained p-values.

Exercise 6

Too cool down let's end this chapter with something easy but potentially useful.

As you have learned by now in programming we often end up using our old functions (or at least I do), although we tend to tweak them a little to adjust them to the ever changing needs.

In this task I want you to change the `drawColPerc` from Section 6.8.3 (or your own solution to Section 6.7.3). You can name the new function, e.g. `drawColPerc2` (wow, how original). The new function should accept among others a bigger data frame (like `dfEyeColorFull`). Inside it runs `runCategTestsGetPVals` we developed in Section 6.8.5 (with multiplicity correction). Then it

should draw the stacked barplots (it draws one stacked barplot for each data frame, the drawings should be set in one column, but in multiple rows, so a graph under a graph). If the distribution in a data frame is statistically significant add a stroke (`strokewidth` argument) to the barplot.

Solutions - Comparisons of Categorical Data

In this sub-chapter you will find exemplary solutions to the exercises from the previous section.

Solution to Exercise 1

An exemplary `getContingencyTable` could look like this (here, a version that produces output that resembles the result of `FreqTables.freqtable`):

```
function getContingencyTable(
    rowVect::Vector{String},
    colVect::Vector{String},
    rowLabel::String,
    colLabel::String,
    )::Dfs.DataFrame

    rowNames::Vector{String} = sort(unique(rowVect))
    colNames::Vector{String} = sort(unique(colVect))
    pairs::Vector{Tuple{String, String}} = collect(zip(rowVect,
colVect))
    pairsCounts::Dict{Tuple{String, String}, Int} = getCounts(pairs)
    labels::String = "↓" * rowLabel * "/" * colLabel * "→"
    df::Dfs.DataFrame = Dfs.DataFrame()
    columns::Dict{String, Vector{Int}} = Dict()

    for cn in colNames
        columns[cn] = [get(pairsCounts, (rn, cn), 0) for rn in rowNames]
    end

    df = Dfs.DataFrame(columns)
    Dfs.insertcols!(df, 1, labels => rowNames)

    return df
end
```

Here, as we often do, we start by declaring some of the helpful variables. `rowNames` and `colNames` contain all the possible unique groups for each input variable (`rowVect` and `colVect`). Then we get all

the consecutive pairings that are in the data by using `zip` and `collect` functions. For instance `collect(zip(["a", "a", "b"], ["x", "y", "x"]))` will yield us the following vector of tuples: `[("a", "x"), ("a", "y"), ("b", "x")]`. The pairs are then sent to `getCounts` (from Section 4.4) to find out how often a given pair occurs.

In the next step we define a variable `df` (for now it is empty) to hold our final result. We saw in Section 6.3 that a data frame can be created by sending a dictionary to the `Dfs.DataFrame` function. Therefore, we declare `columns` (a dictionary) that will hold the count for every column of our contingency table.

We fill the columns one by one with `for cn in colNames` loop. To get a count for a particular row of a given column (`(rn, cn)`) we use `get` function that extracts it from `pairsCounts`. If the key is not there (a given combination of `(rn, cn)` does not exist) we return `0` as a default value. We fill columns by using comprehensions (see Section 3.6.3).

Finally, we put our counts (`columns`) into the data frame (`df`). Now, we insert a column with `rowNames` at position `1` (first column from left) with `Dfs.insertcols!`.

All that it is left to do is to return the result.

Let's find out how our `getContingencyTable` works.

```
smokersByProfession = getContingencyTable(
    smoker,
    profession,
    "smoker",
    "profession"
)
```

↓smoker/profession⟶	Lawyer	Priest	Teacher
no	14	11	18
yes	17	20	20

Table 6: Table 6: Number of smokers by profession (fictitious data).

It appears to work just fine. Let's swap the inputs and see if we get a consistent result.

```
smokersByProfessionTransposed = getContingencyTable(
    profession,
    smoker,
    "profession",
    "smoker"
)
```

↓profession/smoker→	no	yes
Lawyer	14	17
Priest	11	20
Teacher	18	20

Table 7: Table 7: Number of smokers by profession transposed (fictitious data).

Looks good. And now for the small data set with possible zeros.

```
smokersByProfessionSmall = getContingencyTable(
    smokerSmall,
    professionSmall,
    "smoker",
    "profession"
)
```

↓smoker/profession→	Lawyer	Priest	Teacher
no	2	3	1
yes	0	1	3

Table 8: Table 8: Number of smokers by profession (small data set, fictitious data).

Seems to be OK as well. Of course we can use this function with a data frame, e.g. `getContingencyTable(df[!, "col1"], df[!, "col2"], "col1", "col2")` or adopt it slightly to take a data frame as an input.

Solution to Exercise 2

OK, the most direct solution to the problem (for `getColPerc`) would be something like

```

```

```
function getColPerc(m::Matrix{Int})::Matrix{Float64}
    nRows, nCols = size(m)
    percentages:: Matrix{Float64} = zeros(nRows, nCols)
    for c in 1:nCols
        for r in 1:nRows
            percentages[r, c] = m[r, c] / sum(m[:, c])
            percentages[r, c] = round(percentages[r, c] * 100, digits =
2)
        end
    end
    return percentages
end
```

Here, we begin by extracting the number of rows (`nRows`) and columns (`nCols`). We use them right away by defining `percentages` matrix that will hold our final result (for now it is filled with 0s). Then we use the classical nested for loops[290]idiom to calculate the percentage for every cell in the matrix/table (we use array indexing we met in Section 3.3.7). For that we divide each count (`m[r, c]`) by column sum (`sum(m[:, c])`). Next, we multiply it by 100 (`* 100`) to change the decimal to percentage. We round the percentage to two decimal points (`round` and `digits = 2`).

The algorithm is not super efficient (we calculate `sum(m[:, c])` separately for every cell) or terse (9 lines of code). Still, it is pretty clear and for small matrices (a few/several rows/cols, that we expect in our input) does the trick.

OK, let's move to the `getRowPerc` function.

```
function getRowPerc(m::Matrix{Int})::Matrix{Float64}
    nRows, nCols = size(m)
    percentages:: Matrix{Float64} = zeros(nRows, nCols)
    for c in 1:nCols
        for r in 1:nRows
            percentages[r, c] = m[r, c] / sum(m[r, :])
            percentages[r, c] = round(percentages[r, c] * 100, digits =
2)
        end
    end
```

[290]https://en.wikibooks.org/wiki/Introducing_Julia/Controlling_the_flow#Nested_loops

```
    return percentages
end
```

Hmm, it's almost identical to `getColPerc` (`sum(m[:, c])` was replaced with `sum(m[r, :])`). Let's remove the code duplication and put it into a single function.

```
function getPerc(m::Matrix{Int}, byRow::Bool)::Matrix{Float64}
    nRows, nCols = size(m)
    percentages:: Matrix{Float64} = zeros(nRows, nCols)
    dimSum::Int = 0 # sum in a given dimension of a matrix
    for c in 1:nCols
        for r in 1:nRows
            dimSum = (byRow ? sum(m[r, :]) : sum(m[:, c]))
            percentages[r, c] = m[r, c] / dimSum
            percentages[r, c] = round(percentages[r, c] * 100, digits =
2)
        end
    end
    return percentages
end
```

Here, we replaced the function specific sums with a more general `dimSum` (initialized with 0). Then inside the inner for loop we decide which sum to compute (row sum with `sum(m[r, :])` and column sum with `sum(m[:, c])`) with a ternary expression from Section 3.5.2 . OK, enough of tweaking and code optimization, let's test our new function.

```
mEyeColor
```

```
2×2 Matrix{Int64}:
 220  161
 279  320
```

And now column percentages

```
eyeColorColPerc = getPerc(mEyeColor, false)
eyeColorColPerc
```

```
2×2 Matrix{Float64}:
 44.09  33.47
 55.91  66.53
```

So, based on the data in `mEyeColor` we see that in the `uk` (first column) there is roughly 44.09% of people with blue eyes. Whereas in the `us` (second column) there is roughly 33.47% of people with that eye color.

And now for the row percentages.

```
eyeColorRowPerc = getPerc(mEyeColor, true)
eyeColorRowPerc
```

```
2×2 Matrix{Float64}:
 57.74  42.26
 46.58  53.42
```

So, based on the data in `mEyeColor` we see that among the investigated groups roughly 57.74% of blue eyed people live in the `uk` and 42.26% of blue eyed people live in the `us`.

OK, let's just quickly make sure our function also works fine for a bigger table.

```
mEyeColorFull
```

```
3×2 Matrix{Int64}:
 220  161
 149   78
 130  242
```

And now column percentages.

```
eyeColorColPercFull = getPerc(mEyeColorFull, false)
eyeColorColPercFull
```

```
3×2 Matrix{Float64}:
 44.09  33.47
```

```
 29.86  16.22
 26.05  50.31
```

So, based on the data in `mEyeColor` we see that in the `uk` (first column) there is roughly:

- 44.09% of people with blue eyes,
- 29.86% of people with green eyes, and
- 26.05% of people with brown eyes.

For `us` (second column) we got:

- 33.47% of people with blue eyes,
- 16.22% of people with green eyes, and
- 50.31% of people with brown eyes.

Of course, remember that this is all fictitious data inspired by the lecture of this Wikipedia's page[291].

OK, enough for the task solution. If you want to see a more terse (and mysterious) version of `getPerc` then go to this chapter's code snippets[292].

Solution to Exercise 3

OK, the most straightforward way to draw a stacked bar plot would be to use `Cmk.barplot` with `stack` and `color` keyword arguments[293].

The solution below is slightly different. It allows for greater control over the output and it was created after some try and error.

```
import CairoMakie as Cmk

function drawColPerc(df::Dfs.DataFrame,
    dfColLabel::String,
    dfRowLabel::String,
    title::String,
    dfRowColors::Vector{String})::Cmk.Figure
```

[291] https://en.wikipedia.org/wiki/Eye_color
[292] https://github.com/b-lukaszuk/RJ_BS_eng/tree/main/code_snippets/ch06
[293] https://docs.julialang.org/en/v1/manual/functions/#Keyword-Arguments

```
    m::Matrix{Int} = Matrix{Int}(df[:, 2:end])
    columnPerc::Matrix{Float64} = getPerc(m, false)
    nRows, nCols = size(columnPerc)
    colNames::Vector{String} = names(df)[2:end]
    rowNames::Vector{String} = df[1:end, 1]
    xs::Vector{Int} = collect(1:nCols)
    offsets::Vector{Float64} = zeros(nCols)
    curPerc::Vector{Float64} = []
    barplots = []

    fig = Cmk.Figure()
    ax1 = Cmk.Axis(fig[1, 1],
                   title=title, xlabel=dfColLabel, ylabel="% of data",
                   xticks=(xs, colNames), yticks=0:10:100)

    for r in 1:nRows
        curPerc = columnPerc[r, :]
        push!(barplots,
            Cmk.barplot!(ax1, xs, curPerc,
                         offset=offsets, color=dfRowColors[r]))
        offsets = offsets .+ curPerc
    end
    Cmk.Legend(fig[1, 2], barplots, rowNames, dfRowLabel)

    return fig
end
```

We begin by defining a few helpful variables. Most of them are pretty self explanatory and rely on the constructs we met before. The three most enigmatic are `offsets`, `curPerc`, and `barplots`.

`offsets` are the locations on Y-axis where the bottom edges of the bars will be drawn (it is initialized with zeros). `curPerc` will contain heights of the bars to be drawn. `barplots` will contain a vector of bar plot objects drawn (it is necessary for adding proper legend with `Cmk.Legend`). For each row in `columnPerc` (`for r in 1:nRows`) we take the percentage of the row and put it into `curPerc`. Then we draw bars (`Cmk.barplot!`) of that height that start (their bottom edges) at `offsets` and are of a color of our choosing (`dfRowColors[r]`). The list of allowed named colors can be found here[294]. We append the drawn bars to the `bars` vector by using push![295]function (we met it in Section

[294]https://juliagraphics.github.io/Colors.jl/stable/namedcolors/

[295]https://docs.julialang.org/en/v1/base/collections/#Base.push!

3.4.4). Then we add `curPerc` to the offset so that the bottom edges of the next bars will start where the top edges of the previous bars ended.

Once the for loop ended we finish by adding the appropriate legend.

OK, time to test our function

```
drawColPerc(dfEyeColorFull, "Country", "Eye color",
    "Eye Color distribution by country (column percentages)",
    ["lightblue1", "seagreen3", "peachpuff3"])
```

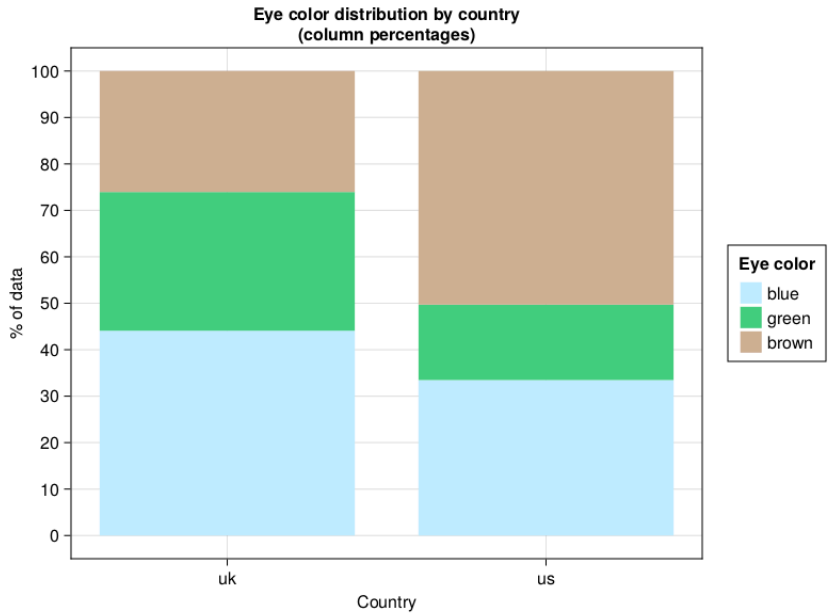

Figure 25: Figure 24: Eye color distribution by country (column percentages, fictitious data).

I don't know about you but to me it looks pretty nice.

OK, now we could write `drawRowPerc` function by modifying our `drawColPerc` slightly. Finally, after some try and error we could write `drawPerc` function that combines both those functionalities and reduces code duplication. Without further ado let me fast forward to the definition of `drawPerc`

```
function drawPerc(df::Dfs.DataFrame, byRow::Bool,
    dfColLabel::String,
    dfRowLabel::String,
    title::String,
    groupColors::Vector{String})::Cmk.Figure

    m::Matrix{Int} = Matrix{Int}(df[:, 2:end])
    dimPerc::Matrix{Float64} = getPerc(m, byRow)
    nRows, nCols = size(dimPerc)
    colNames::Vector{String} = names(df)[2:end]
    rowNames::Vector{String} = df[1:end, 1]
    ylabel::String = "% of data"
    xlabel::String = (byRow ? dfRowLabel : dfColLabel)
    xs::Vector{Int} = collect(1:nCols)
    yticks::Tuple{Vector{Int},Vector{String}} = (
        collect(0:10:100), map(string, 0:10:100)
    )
    xticks::Tuple{Vector{Int},Vector{String}} = (xs, colNames)

    if byRow
        nRows, nCols = nCols, nRows
        xs = collect(1:nCols)
        colNames, rowNames = rowNames, colNames
        dfColLabel, dfRowLabel = dfRowLabel, dfColLabel
        xlabel, ylabel = ylabel, xlabel
        yticks, xticks = (xs, colNames), yticks
    end

    offsets::Vector{Float64} = zeros(nCols)
    curPerc::Vector{Float64} = []
    barplots = []

    fig = Cmk.Figure()
    ax1 = Cmk.Axis(fig[1, 1], title=title,
                   xlabel=xlabel, ylabel=ylabel,
                   xticks=xticks, yticks=yticks)

    for r in 1:nRows
        curPerc = (byRow ? dimPerc[:, r] : dimPerc[r, :])
        push!(barplots,
            Cmk.barplot!(ax1, xs, curPerc,
                         offset=offsets, color=groupColors[r],
                         direction=(byRow ? :x : :y)))
        offsets = offsets .+ curPerc
    end
    Cmk.Legend(fig[1, 2], barplots, rowNames, dfRowLabel)

    return fig
end
```

Ok, let's see how it works.

```
drawPerc(dfEyeColorFull, true,
    "Country", "Eye color",
    "Eye Color distribution by country (row percentages)",
    ["red", "blue"])
```

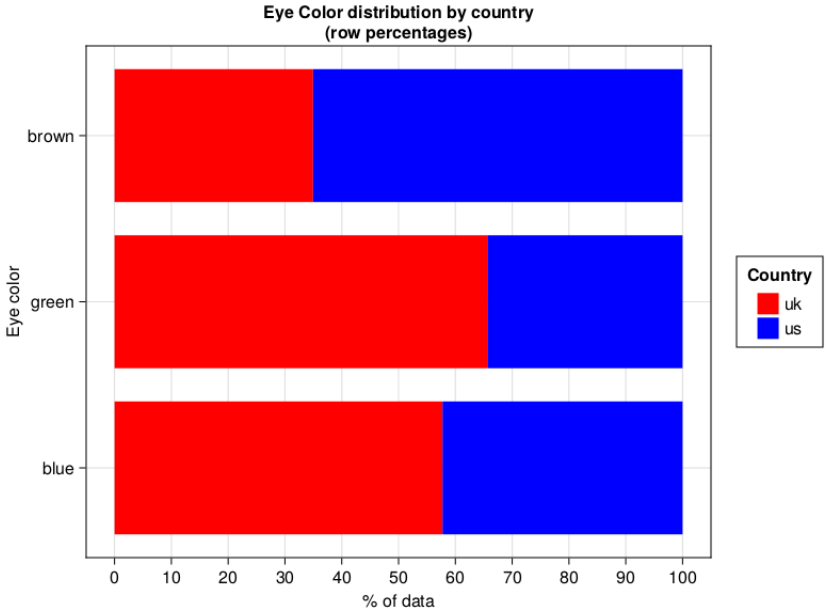

Figure 26: Figure 25: Eye color distribution by country (row percentages, fictitious data).

Pretty, pretty, pretty.

I leave the code in `drawPerc` for you to decipher. Let me just explain a few new pieces.

In Julia (like in Python) we can define two variables in one go by using the following syntax: `a, b = 1, 2` (now `a = 1` and `b = 2`). Let's say that later in our program we decided that from now on `a` should be `2`, and `b` should be `1`. We can swap the variables using the following one line expression: `a, b = b, a`.

Additionally, `drawPerc` makes use of the `direction` keyword argument that accepts symbols[296] `:x` or `:y`. It made the output slightly more visually pleasing but also marginally complicated the code. Anyway, `direction = :y` draws vertical bars (see Figure 24), whereas `direction = :x` draws horizontal bars (see Figure 25).

And that's it for this exercise.

Solution to Exercise 4

OK, let's start by defining helper functions that we will use to test the assumptions.

```
function isSumAboveCutoff(m::Matrix{Int}, cutoff::Int = 49)::Bool
    return sum(m) > cutoff
end

function getExpectedCounts(m::Matrix{Int})::Vector{Float64}
    nObs::Int = sum(m)
    cProbs::Vector{Float64} = [sum(c) / nObs for c in eachcol(m)]
    rProbs::Vector{Float64} = [sum(r) / nObs for r in eachrow(m)]
    probsUnderH0::Vector{Float64} = [
        cp * rp for cp in cProbs for rp in rProbs
        ]
    return probsUnderH0 .* nObs
end

function areAllExpectedCountsAboveCutoff(
    m::Matrix{Int}, cutoff::Float64 = 5.0)::Bool
    expectedCounts::Vector{Float64} = getExpectedCounts(m)
    return map(x -> x >= cutoff, expectedCounts) |> all
end

function areChiSq2AssumptionsOK(m::Matrix{Int})::Bool
    sumGTEQ50::Bool = isSumAboveCutoff(m)
    allExpValsGTEQ5::Bool = areAllExpectedCountsAboveCutoff(m)
    return sumGTEQ50 && allExpValsGTEQ5
end
```

There is not much to explain here, since all we did was to gather the functionality we had developed in the previous chapters (e.g. in Section 6.3).

And now for the tests.

[296] https://docs.julialang.org/en/v1/base/base/#Core.Symbol

```
function runFisherExactTestGetPVal(m::Matrix{Int})::Float64
    @assert (size(m) == (2, 2)) "input matrix must be of size (2, 2)"
    a, c, b, d = m
    return Ht.FisherExactTest(a, b, c, d) |> Ht.pvalue
end

function runCategTestGetPVal(m::Matrix{Int})::Float64
    @assert (size(m) == (2, 2)) "input matrix must be of size (2, 2)"
    if areChiSq2AssumptionsOK(m)
        return Ht.ChisqTest(m) |> Ht.pvalue
    else
        return runFisherExactTestGetPVal(m)
    end
end

function runCategTestGetPVal(df::Dfs.DataFrame)::Float64
    @assert (size(df) == (2, 3)) "input df must be of size (2, 3)"
    return runCategTestGetPVal(Matrix{Int}(df[:, 2:3]))
end
```

Again, all we did here was to collect the proper functionality we had developed in this chapter (Section 6) and its sub-chapters. Therefore, I'll refrain myself from comments. Instead let's test our newly developed tools.

```
round.(
    [
        runCategTestGetPVal(mEyeColor),
        runCategTestGetPVal(mEyeColorSmall),
        runCategTestGetPVal(dfEyeColor)
    ],
    digits = 4
)
```

```
[0.0007, 0.6373, 0.0007]
```

The functions appear to be working as intended, and the obtained p-values match those from Section 6.3 and Section 6.4 .

Solution to Exercise 5

Let's start by writing a function that will accept a data frame like `dfEyeColorFull` and return all the possible 2x2 data frames (2 rows and 2 columns with counts).

```
# previously (ch05) defined function
function getUniquePairs(names::Vector{T})::Vector{Tuple{T,T}} where T
    @assert (length(names) >= 2) "the input must be of length >= 2"
    uniquePairs::Vector{Tuple{T,T}} =
        Vector{Tuple{T,T}}(undef, binomial(length(names), 2))
    currInd::Int = 1
    for i in eachindex(names)[1:(end-1)]
        for j in eachindex(names)[(i+1):end]
            uniquePairs[currInd] = (names[i], names[j])
            currInd += 1
        end
    end
    return uniquePairs
end

function get2x2Dfs(biggerDf::Dfs.DataFrame)::Vector{Dfs.DataFrame}
    nRows, nCols = size(biggerDf)
    @assert ((nRows > 2) || (nCols > 3)) "matrix of counts must be >
2x2"
    rPairs::Vector{Tuple{Int, Int}} = getUniquePairs(collect(1:nRows))
    # counts start from column 2
    cPairs::Vector{Tuple{Int, Int}} = getUniquePairs(collect(2:nCols))
    return [
        biggerDf[[r...], [1, c...]] for r in rPairs for c in cPairs
    ]
end
```

We begin by copying and pasting `getUniquePairs` from Section 5.8.4 . We will use it in `get2x2Dfs`. First we get unique pairs of rows (`rPairs`). Then we get unique pairs of columns (`cPairs`). Finally, using nested comprehension and indexing (for reminder see Section 3.3.7 and Section 5.3.1) we get the vector of all possible 2x2 data frames (actually 2x3 data frames, because first column contains row labels). Since each element of `rPairs` (`r`) or `cPairs` (`c`) is a tuple, and indexing must be a vector, then we convert one into the other using `[r...]` and `[c...]` syntax (e.g. `[(1, 2)...]` will give us `[1, 2]`). In the end we get the list of data frames as a result.

OK, let's write a function to compute p-values (for now unadjusted) for data frames in a vector.

```
function runCategTestsGetPVals(
    biggerDf::Dfs.DataFrame
    )::Tuple{Vector{Dfs.DataFrame}, Vector{Float64}}
```

```
    overallPVal::Float64 = Ht.ChisqTest(
        Matrix{Int}(biggerDf[:, 2:end])) |> Ht.pvalue
    if (overallPVal <= 0.05)
        dfs::Vector{Dfs.DataFrame} = get2x2Dfs(biggerDf)
        pvals::Vector{Float64} = runCategTestGetPVal.(dfs)
        return (dfs, pvals)
    else
        return ([biggerDf], [overallPVal])
    end
end
```

The function is rather simple. First, it checks the overall p-value (`overallPVal`) for the `biggerDf`. If it is less than or equal to our customary cutoff level ($\alpha = 0.05$) then we execute `runCategTestGetPVal` on each possible data frame (`dfs`) using the dot operator syntax from Section 3.6.5 . We return a tuple, its first element is a vector of data frames, its second element is a vector of corresponding (uncorrected) p-values. If `overallPVal` is greater than the cutoff level then we place our `biggerDf` and its corresponding p-value (`overallPVal`) into vectors, and place them into a tuple (which is returned).

Time to test our function.

```
resultCategTests = runCategTestsGetPVals(dfEyeColorFull)
resultCategTests[1]
```

eyeCol	uk	us
blue	220	161
green	149	78

eyeCol	uk	us
blue	220	161
brown	130	242

eyeCol	uk	us
green	149	78
brown	130	242

Looking good, and now the corresponding unadjusted p-values.

```
resultCategTests[2]
```

```
[0.05384721765961758, 3.5949791158435336e-10, 2.761179458504292e-13]
```

Once we got it, adjusting the p-values should be a breeze.

```
import MultipleTesting as Mt

function adjustPVals(
    multCategTestsResults::Tuple{Vector{Dfs.DataFrame},
Vector{Float64}},
    multCorr::Type{<:Mt.PValueAdjustment}
    )::Tuple{Vector{Dfs.DataFrame}, Vector{Float64}}
    dfs, pvals = multCategTestsResults
    adjPVals::Vector{Float64} = Mt.adjust(pvals, multCorr())
    return (dfs, adjPVals)
end
```

Yep. All we did here, was to extract the vector of p-values (`pvals`) and send it as an argument to `Mt.adjust` for correction. Let's see how it works (since we are using the `Bonferroni` method then we expect the adjusted p-values to be 3x greater than the unadjusted ones, see Section 5.6).

```
resultAdjustedCategTests = adjustPVals(resultCategTests, Mt.Bonferroni)
resultAdjustedCategTests[2]
```

```
[0.16154165297885273, 1.07849373475306e-9, 8.283538375512876e-13]
```

OK, it appears to be working just fine.

Solution to Exercise 6

OK, let's look at an exemplary solution.

```
function drawColPerc2(
    biggerDf::Dfs.DataFrame,
    dfColLabel::String,
    dfRowLabel::String,
    title::String,
    dfRowColors::Dict{String,String},
    alpha::Float64=0.05,
    adjMethod::Type{<:Mt.PValueAdjustment}=Mt.Bonferroni)::Cmk.Figure

    multCategTests::Tuple{
        Vector{Dfs.DataFrame},
        Vector{Float64}} = runCategTestsGetPVals(biggerDf)
    multCategTests = adjustPVals(multCategTests, adjMethod)
    dfs, pvals = multCategTests

    fig = Cmk.Figure(size=(800, 400 * length(dfs)))

    for i in eachindex(dfs)
        m::Matrix{Int} = Matrix{Int}(dfs[i][:, 2:end])
        columnPerc::Matrix{Float64} = getPerc(m, false)
        nRows, nCols = size(columnPerc)
        colNames::Vector{String} = names(dfs[i])[2:end]
        rowNames::Vector{String} = dfs[i][1:end, 1]
        xs::Vector{Int} = collect(1:nCols)
        offsets::Vector{Float64} = zeros(nCols)
        curPerc::Vector{Float64} = []
        barplots = []

        ax = Cmk.Axis(fig[i, 1],
            title=title, xlabel=dfColLabel, ylabel="% of data",
            xticks=(xs, colNames), yticks=0:10:100)

        for r in 1:nRows
            curPerc = columnPerc[r, :]
            push!(barplots,
                Cmk.barplot!(ax, xs, curPerc,
                    offset=offsets,
                    color=get(dfRowColors, rowNames[r], "black"),
                    strokewidth=(pvals[i] <= alpha) ? 2 : 0))
            offsets = offsets .+ curPerc
        end
        Cmk.Legend(fig[i, 2], barplots, rowNames, dfRowLabel)
    end

    return fig
end
```

The function definition differs slightly from the original `drawColPerc`. Of note we changed the `colors` parameter from `Vector{String}` to

`Dict{String, String}` (a mapping between row name in column 1 and color by which it will be represented on the graph). Of course, we added two more parameters `alpha` and `adjMethod`.

First, we run multiple categorical tests (`runCategTestsGetPVals`) and adjust the obtained p-values (`adjustPVals`) using functionality developed earlier (Section 6.8.5). Then we, define the figure object with a desired size (`size=(widthPixels, heightPixels)`) adjusted by number of subplots in the figure (`* length(dfs)`).

The next step is pretty simple, basically we enclose the previously developed code from `drawColPerc` in a for loop (`for i in eachindex(dfs)`) that draws consecutive data frames as a stacked bar plots in a separate rows of the figure. If a statistically significant difference for a data frame was detected (`pvals[i] <= alpha`) we add a stroke (`strokewidth`) to the bar plot.

Time to see how it works.

```
drawColPerc2(dfEyeColorFull, "Country", "Eye color", "Eye color by
country",
    Dict("blue" => "lightblue1",
        "green" => "seagreen3",
        "brown" => "peachpuff3"))
```

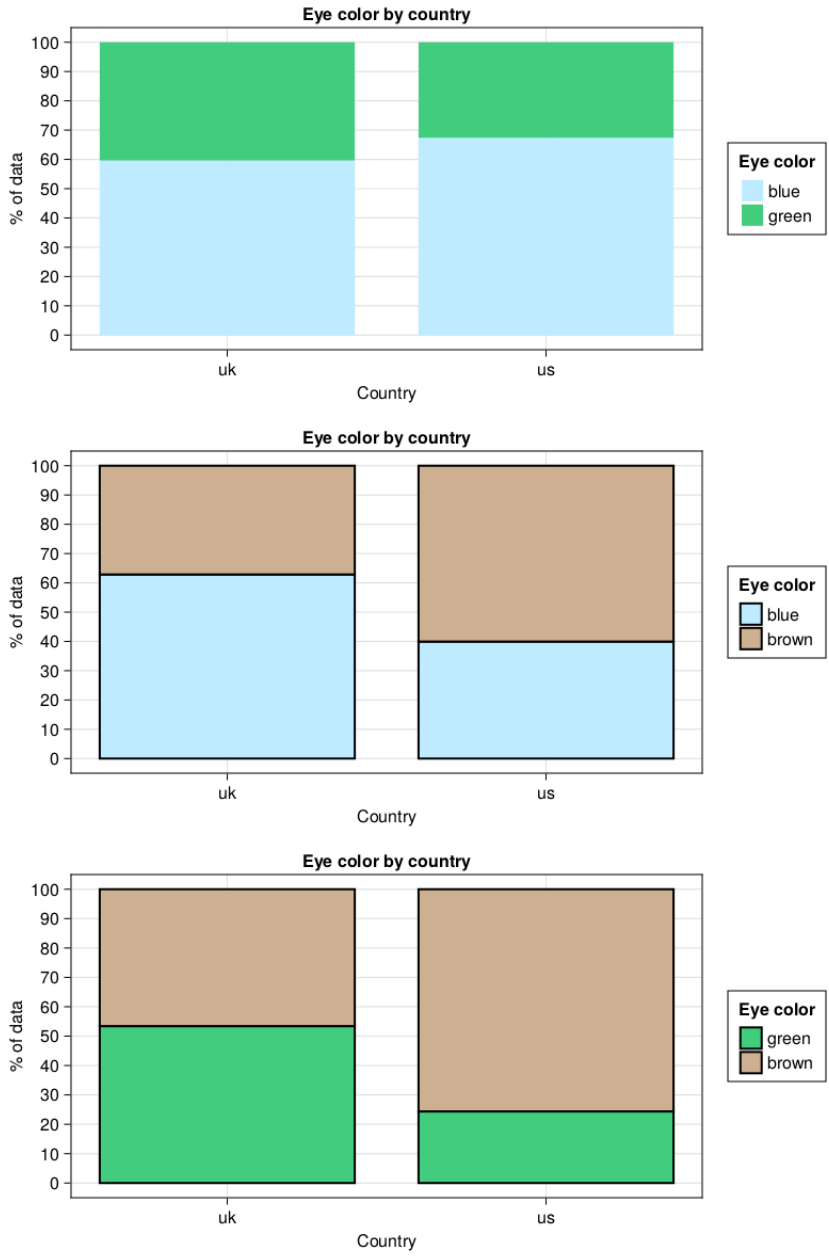

Figure 27: Figure 26: Eye color distribution by country (column percentages, fictitious data). Stroke denotes statistically significant difference ($p \leq 0.05$).

It looks quite OK + it allows us to quickly judge which eye colors distributions differ one from another. For a more complicated layout we should probably follow the guidelines contained in the Layout Tutorial[297].

[297] https://docs.makie.org/stable/tutorials/layout-tutorial/

Association and Prediction

OK, time to talk about association between two variables and how to predict the value of one variable based on the value(s) of other variable(s).

Chapter imports

Later in this chapter we are going to use the following libraries

```
import CairoMakie as Cmk
import CSV as Csv
import DataFrames as Dfs
import Distributions as Dsts
import GLM as Glm
import MultipleTesting as Mt
import Random as Rand
import RDatasets as RD
import Statistics as Stats
```

If you want to follow along you should have them installed on your system. A reminder of how to deal (install and such) with packages can be found here[298]. But wait, you may prefer to use `Project.toml` and `Manifest.toml` files from the code snippets for this chapter[299]to install the required packages. The instructions you will find here[300].

The imports will be placed in the code snippet when first used, but I thought it is a good idea to put them here, after all imports should be at the top of your file (so here they are at the top of the chapter). Moreover, that way they will be easier to find all in one place.

If during the lecture of this chapter you find a piece of code of unknown functionality, just go to the code snippets mentioned above and run the code from the `*.jl` file. Once you have done that you can always extract a small piece of it and test it separately (modify and experiment with it if you wish).

[298]https://docs.julialang.org/en/v1/stdlib/Pkg/
[299]https://github.com/b-lukaszuk/RJ_BS_eng/tree/main/code_snippets/ch07
[300]https://pkgdocs.julialang.org/v1/environments/

Linear relation

Imagine you are a biologist that conducts their research in the Amazon rainforest[301]known for biodiversity and heavy rainfalls (see the name). You divided the area into 20 equal size fields on which you measured the volume of rain (per a unit of time) and biomass of two plants (named creatively `plantA` and `plantB`). The results are contained in `biomass.csv` file, let's take a sneak peak at them.

```
import CSV as Csv
import DataFrames as Dfs

# if you are in 'code_snippets' folder, then use: "./ch07/biomass.csv"
# if you are in 'ch07' folder, then use: "./biomass.csv"
biomass = Csv.read("./code_snippets/ch07/biomass.csv", Dfs.DataFrame)
first(biomass, 5)
```

plantAkg	rainL	plantBkg
20.26	15.09	21.76
9.18	5.32	6.08
11.36	12.5	10.96
11.26	10.7	4.96
9.05	5.7	9.55

Table 12: Table 9: Effect of rainfall on plants biomass (fictitious data).

I think some plot would be helpful to get a better picture of the data (pun intended).

```
import CairoMakie as Cmk

fig = Cmk.Figure()
ax1 = Cmk.Axis(fig[1, 1],
               title="Effect of rainfall on biomass of plant A",
               xlabel="water [L]", ylabel="biomass [kg]")
Cmk.scatter!(ax1, biomass.rainL, biomass.plantAkg,
             markersize=25, color="skyblue",
             strokewidth=1, strokecolor="gray")
ax2 = Cmk.Axis(fig[1, 2],
               title="Effect of rainfall on biomass of plant B",
```

[301]https://en.wikipedia.org/wiki/Amazon_rainforest

```
              xlabel="water [L]", ylabel="biomass [kg]")
Cmk.scatter!(ax2, biomass.rainL, biomass.plantBkg,
             markersize=25, color="linen",
             strokewidth=1, strokecolor="black")
Cmk.linkxaxes!(ax1, ax2)
Cmk.linkyaxes!(ax1, ax2)
fig
```

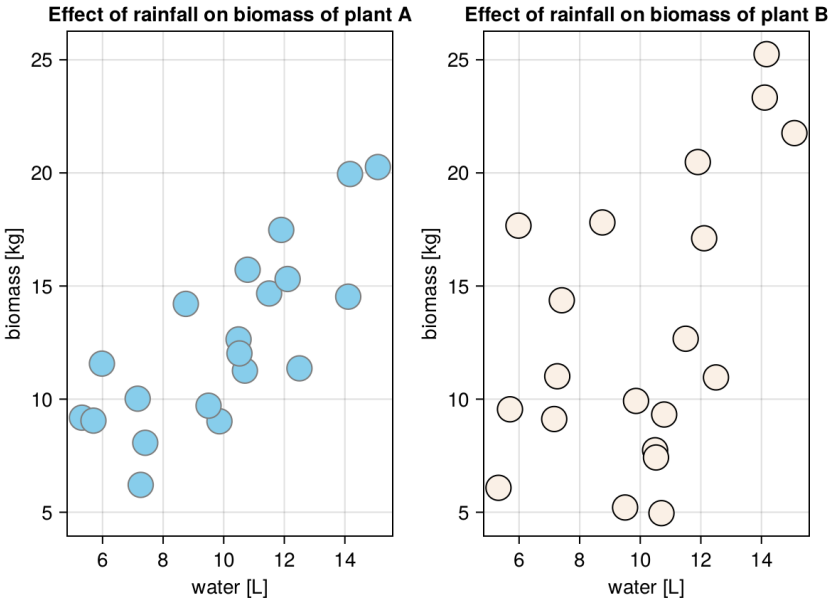

Figure 28: Figure 27: Effect of rainfall on a plant's biomass.

Overall, it looks like the biomass of both plants is directly related (one increases and the other increases) with the volume of rain. That seems reasonable. Moreover, we can see that the points are spread along an imaginary line (go ahead imagine it) that goes through all the points on a graph. We can also see that `plantB` has a somewhat greater spread of points (which may indicate smaller dependency on water). It would be nice to be able to express such a relation between two variables (here biomass and volume of rain) with a single number. It turns out that we can. That's the job for covariance[302].

[302]https://en.wikipedia.org/wiki/Covariance

Covariance

The formula for covariance resembles the one for `variance` that we met in Section 4.6 (`getVar` function) only that it is calculated for pairs of values (here a plant biomass and rainfall for a field), so two vectors instead of one. Observe

```
import Statistics as Stats

function getCov(v1::Vector{<:Real}, v2::Vector{<:Real})::Float64
    @assert length(v1) == length(v2) "v1 and v2 must be of equal
lengths"
    avg1::Float64 = Stats.mean(v1)
    avg2::Float64 = Stats.mean(v2)
    diffs1::Vector{<:Real} = v1 .- avg1
    diffs2::Vector{<:Real} = v2 .- avg2
    return sum(diffs1 .* diffs2) / (length(v1) - 1)
end
```

> ***Note:*** To calculate the covariance you may also use Statistics.cov[303].

A few points of notice. In Section 4.6 in `getVar` we squared the differences (`diffs`), i.e. we multiplied the diffs by themselves (x*$x = x^2$). Here, we do something similar by multiplying parallel values from both vectors of `diffs` (`diffs1` and `diffs2`) by each other (x*y, for a given field). Moreover, instead of taking the average (so `sum(diffs1 .* diffs2)/length(v1)`) here we use the more fine tuned statistical formula that relies on the degrees of freedom we met in Section 5.2 (there we used `getDf` function on a vector, here we kind of use `getDf` on the number of fields that are represented by the points in the Figure 27).

Enough explanations, let's see how it works. First, a few possible associations that roughly take the following shapes on a graph: `/`, `\`, `|`, and `-`.

```
rowLenBiomass, _ = size(biomass)
```

[303] https://docs.julialang.org/en/v1/stdlib/Statistics/#Statistics.cov

```
(
    # assuming: getCov(xs, ys),
    # you may test the distributions with: Cmk.scatter(xs, ys)
    getCov(biomass.rainL, biomass.plantAkg), # /
    getCov(collect(1:1:rowLenBiomass), collect(rowLenBiomass:-1:1)), # \
    getCov(repeat([5], rowLenBiomass), biomass.plantAkg), # |
    getCov(biomass.rainL, repeat([5], rowLenBiomass)) # -
)
```

```
(8.721824210526316, -35.0, 0.0, 0.0)
```

We can see that whenever both variables (on X- and on Y-axis) increase simultaneously (points lie alongside `/` imaginary line like in Figure 27) then the covariance is positive. If one variable increases whereas the other decreases (points lie alongside `\` imaginary line) then the covariance is negative. Whereas in the case when one variable changes and the other is stable (points lie alongside `|` or `-` line) the covariance is equal zero.

OK, time to compare the both plants.

```
covPlantA = getCov(biomass.plantAkg, biomass.rainL)
covPlantB = getCov(biomass.plantBkg, biomass.rainL)

(
    covPlantA,
    covPlantB,
)
```

```
(8.721824210526316, 9.527113684210526)
```

In Section 4.6 greater `variance` (and `standard deviation`) meant greater spread of points around the mean, here the greater covariance expresses the greater spread of the points around the imaginary trend line (in Figure 27). But beware, you shouldn't judge the spread of data based on the covariance alone. To understand why let's look at the graph below.

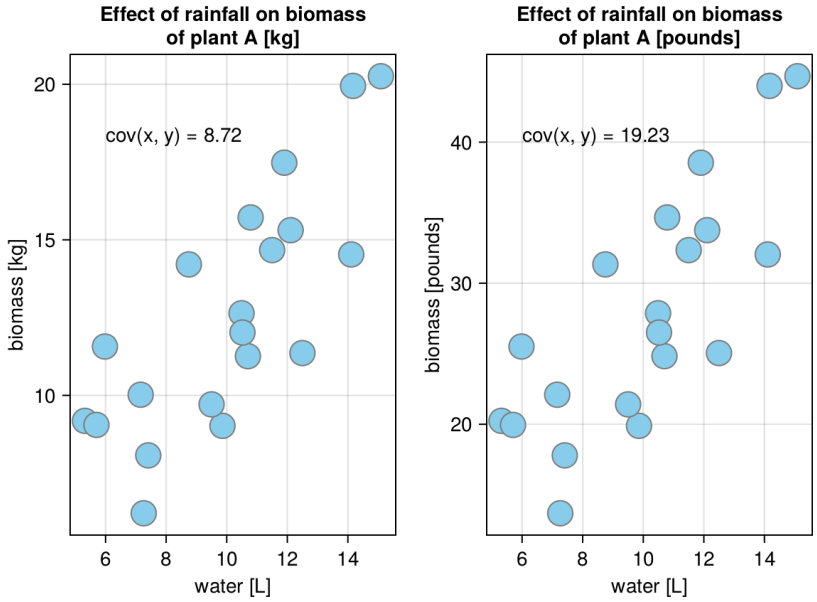

Figure 29: Figure 28: Effect of rainfall on plants' biomass.

Here, we got the biomass of `plantA` in different units (kilograms and pounds). Logic and visual inspection of the points spread on the graph suggest that the covariances should be the same. Or maybe not?

```
(
    getCov(biomass.plantAkg, biomass.rainL),
    getCov(biomass.plantAkg .* 2.205, biomass.rainL),
)
```

```
(8.721824210526316, 19.231622384210525)
```

The covariances suggest that the spread of the data points is like 2 times greater between the two sub-graphs in Figure 28 , but that is clearly not the case. The problem is that the covariance is easily inflated by the units of measurements. That is why we got an improved metrics for association named correlation[304].

[304]https://en.wikipedia.org/wiki/Correlation

Correlation

Correlation is most frequently expressed in the term of the Pearson correlation coefficient[305] that by itself relies on covariance we met in the previous section. Its formula is pretty straightforward

```
# calculates the Pearson correlation coefficient
function getCor(v1::Vector{<:Real}, v2::Vector{<:Real})::Float64
    return getCov(v1, v2) / (Stats.std(v1) * Stats.std(v2))
end
```

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

> ***Note:*** To calculate the Pearson correlation coefficient you may also use Statistics.cor[306].

The correlation coefficient is just the covariance (numerator) divided by the product of two standard deviations (denominator). The lowest absolute value (`abs(getCov(v1, v2))`) possible for covariance is 0. The maximum absolute value possible for covariance is equal to `Stats.std(v1) * Stats.std(v2)`. Therefore, the correlation coefficient (often abbreviated as `r`) takes values from 0 to 1 for positive covariance and from 0 to −1 for negative covariance. The more tightly our points lie on an imaginary trend line the greater is `abs(corCoef)`.

Let's see how it works.

```
biomassCors = (
    getCor(biomass.plantAkg, biomass.rainL),
    getCor(biomass.plantAkg .* 2.205, biomass.rainL), # pounds
    getCor(biomass.plantBkg, biomass.rainL),
    getCor(biomass.plantBkg .* 2.205, biomass.rainL), # pounds
)
round.(biomassCors, digits = 2)
```

```
(0.78, 0.78, 0.53, 0.53)
```

[305] https://en.wikipedia.org/wiki/Pearson_correlation_coefficient
[306] https://docs.julialang.org/en/v1/stdlib/Statistics/#Statistics.cor

Clearly, the new and improved coefficient is more useful than the old one (covariance). Large spread of points along the imaginary line in Figure 27 yields small correlation coefficient (closer to 0). Small spread of points on the other hand results in a high correlation coefficient (closer to −1 or 1). So, now we can be fairly sure of the greater strength of association between `plantA` and rainfall than `plantB` and the condition.

Importantly, the correlation coefficient depends not only on the scatter of points along an imaginary line, but also on the slope of the line. Observe:

```
import Random as Rand

Rand.seed!(321)

jitter = Rand.rand(-0.2:0.01:0.2, 10)
z1 = collect(1:10)
z2 = repeat([5], 10)
(
    getCor(z1 .+ jitter, z1), # / imaginary line
    getCor(z1, z2 .+ jitter) # - imaginary line
)
```

```
(0.9992378634323702, -0.3215268421510342)
```

Feel free to draw side by side scatter plots for the example above (remember to link the axes). In the code snippet above the spread of data points along the imaginary line is the same in both cases. Yet, the correlation coefficient is much smaller in the second case. This is because of the covariance that is present in the `getCor` function (in numerator). The covariance is greater when the points change together is a given direction. The change is smaller and non-systematic in the second case, hence the lower correlation coefficient. You may want to keep that in mind as it will become handy once we talk about correlation pitfalls in Section 7.5 .

Anyway, the interpretation of the correlation coefficient differs depending on a textbook and a field of science, but in biology it is approximated by those cutoffs:

- `abs(r)` = [0 - 0.2) - very weak correlation
- `abs(r)` = [0.2 - 0.4) - weak correlation
- `abs(r)` = [0.4 - 0.6) - moderate correlation
- `abs(r)` = [0.6 - 0.8) - strong correlation
- `abs(r)` = [0.8 - 1] - very strong correlation

> ***Note:*** The Pearson's correlation coefficient is often abbreviated as `r`. Whereas, `]` and `)` signify closed and open interval, respectively. So, x in range `[0, 1]` means 0 <= x <= 1, whereas x in range `[0, 1)` means 0 <= x < 1.

In general, if `x` and `y` are correlated then this may mean one of a few things, the most obvious of which are:

- `x` is a cause, `y` is an effect
- `y` is a cause, `x` is an effect
- changes in `x` and `y` are caused by an unknown third factor(s)
- `x` and `y` are not related but it just happened that in the sample they appear to be related by chance alone (in a small sample drawn from a population they appear to be associated, but in the population they are not).

We can protect ourselves (to a certain extent) against the last contingency with our good old Student's T-test (see Section 5.2). As stated in the Wikipedia's page[307]:

> [...] Pearson's correlation coefficient follows Student's t-distribution with degrees of freedom n – 2. Specifically, if the underlying variables have a bivariate normal distribution the variable

$$t = \frac{r}{\sigma_r} = r^* \sqrt{\frac{n-2}{1-r^2}}$$

[307] https://en.wikipedia.org/wiki/Pearson_correlation_coefficient#Testing_using_Student's_t-distribution

has a student's t-distribution in the null case (zero correlation)

Let's put that knowledge to good use:

```
# calculates the Pearson correlation coefficient and pvalue
# assumption (not tested in the function): v1 & v2 got normal
distributions
function getCorAndPval(
    v1::Vector{<:Real}, v2::Vector{<:Real})::Tuple{Float64, Float64}
    r::Float64 = Stats.cor(v1, v2) # or: getCor(v1, v2)
    n::Int = length(v1) # num of points
    df::Int = n - 2
    t::Float64 = r * sqrt(df / (1 - r^2)) # t-statistics
    leftTail::Float64 = Dsts.cdf(Dsts.TDist(df), t)
    pval::Float64 = (t > 0) ? (1 - leftTail) : leftTail
    return (r, pval * 2) # (* 2) two-tailed probability
end
```

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

The function is just a translation of the formula given above + some calculations similar to those we did in Section 5.2 to get the p-value. And now for our correlations.

```
biomassCorsPvals = (
    getCorAndPval(biomass.plantAkg, biomass.rainL),
    getCorAndPval(biomass.plantAkg .* 2.205, biomass.rainL), # pounds
    getCorAndPval(biomass.plantBkg, biomass.rainL),
    getCorAndPval(biomass.plantBkg .* 2.205, biomass.rainL), # pounds
)
biomassCorsPvals
```

```
((0.7820227869193526, 4.635013786202791e-5),
 (0.7820227869193522, 4.635013786202791e-5),
 (0.526545847035062, 0.017073389709765907),
 (0.5265458470350619, 0.017073389709765907))
```

We can see that both correlation coefficients are unlikely to have occurred by chance alone ($p \leq 0.05$). Therefore, we can conclude that in each case the biomass is associated with the amount of water a plant receives. I don't know a formal test to compare two correlation coefficients, but based on the `rs` alone it appears that the biomass of

`plantA` is more tightly related to (or maybe even it relies more on) the amount of water than the other plant (`plantB`).

Correlation Pitfalls

The Pearson correlation coefficient is pretty useful (especially in connection with the Student's t-test), but it shouldn't be applied thoughtlessly.

Let's take a look at the Anscombe's quartet[308].

```
import RDatasets as RD

anscombe = RD.dataset("datasets", "anscombe")
first(anscombe, 5)
```

X1	X2	X3	X4	Y1	Y2	Y3	Y4
10.0	10.0	10.0	8.0	8.04	9.14	7.46	6.58
8.0	8.0	8.0	8.0	6.95	8.14	6.77	5.76
13.0	13.0	13.0	8.0	7.58	8.74	12.74	7.71
9.0	9.0	9.0	8.0	8.81	8.77	7.11	8.84
11.0	11.0	11.0	8.0	8.33	9.26	7.81	8.47

Table 13: Table 10: DataFrame for Anscombe's quartet

The data frame is a part of RDatasets[309]that contains a collection of standard data sets used with the R programming language[310]. The data frame was carefully designed to demonstrate the perils of relying blindly on correlation coefficients.

```
fig = Cmk.Figure()
i = 0
for r in 1:2 # r - row
    for c in 1:2 # c - column
        i += 1
        xname = string("X", i)
        yname = string("Y", i)
```

[308]https://en.wikipedia.org/wiki/Anscombe%27s_quartet
[309]https://github.com/JuliaStats/RDatasets.jl
[310]https://en.wikipedia.org/wiki/R_(programming_language)

```
        xs = anscombe[:, xname]
        ys = anscombe[:, yname]
        cor, pval = getCorAndPval(xs, ys)
        ax = Cmk.Axis(fig[r, c],
                      title=string("Figure ", "ABCD"[i]),
                      xlabel=xname, ylabel=yname,
                      limits=(0, 20, 0, 15))
        Cmk.scatter!(ax, xs, ys)
        Cmk.text!(ax, 9, 3, text="cor(x, y) = $(round(cor, digits=2))")
        Cmk.text!(ax, 9, 1, text="p-val = $(round(pval, digits=4))")
    end
end

fig
```

There's not much to explain here. The only new part is `string` function that converts its elements to strings (if they aren't already) and glues them together into a one long string. The rest is just plain drawing with `CairoMakie`. Still, take a look at the picture below

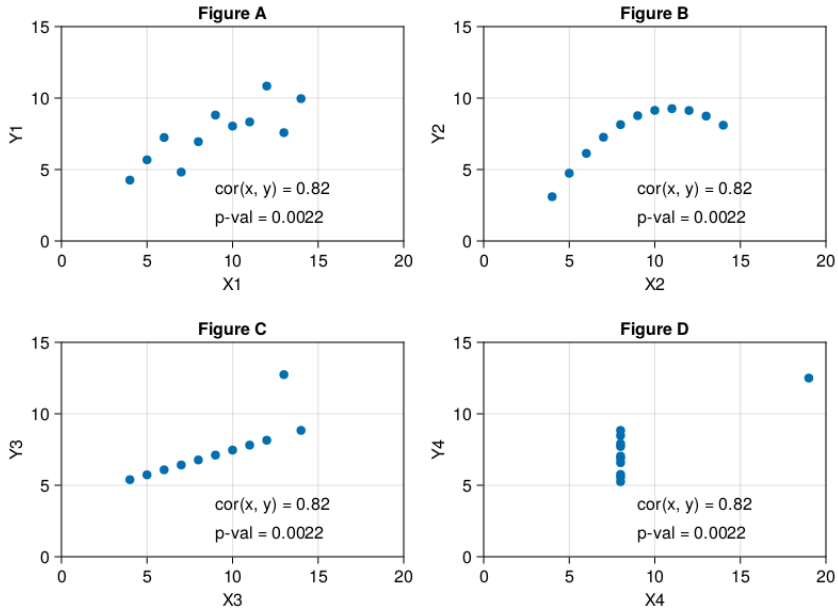

Figure 30: Figure 29: Anscombe's Quartet.

All the sub-figures from Figure 29 depict different relation types between the X and Y variables, yet the correlations and p-values are the same. Two points of notice here. In **Figure B** the points lie in a perfect order on a curve. So, in a perfect word the correlation coefficient should be equal to 1. Yet it is not, as it only measures the spread of the points around an imaginary straight line. Moreover, correlation is sensitive to outliers[311]. In **Figure D** the X and Y variables appear not to be associated at all (for X = 8, Y can take any value). Again, in the perfect world the correlation coefficient should be equal to 0. Still, the outlier on the far right (that in real life may have occurred by a typographical error) pumps it up to 0.82 (or what we could call a very strong correlation). Lesson to be learned here, don't trust the numbers, and whenever you can draw a scatter plot to double check them. And remember, "All models are wrong, but some are useful"[312].

Other pitfalls are also possible. For instance, imagine you measured body and tail length of a certain species of mouse, here are your results.

```
# if you are in 'code_snippets' folder, then use: "./ch07/
miceLengths.csv"
# if you are in 'ch07' folder, then use: "./miceLengths.csv"
miceLengths = Csv.read(
    "./code_snippets/ch07/miceLengths.csv",
    Dfs.DataFrame)
first(miceLengths, 5)
```

[311]https://en.wikipedia.org/wiki/Outlier

[312]https://en.wikipedia.org/wiki/All_models_are_wrong

bodyCm	tailCm	sex
11.3	2.55	f
11.18	2.22	f
9.42	2.54	f
9.21	2.2	f
9.97	2.63	f

Table 14: Table 11: Body lengths of a certain mouse species (fictitious data).

You are interested to know if the tail length is associated with the body length of the animals.

```
getCorAndPval(miceLengths.bodyCm, miceLengths.tailCm)
```

```
(0.8899347709623199, 1.5005298337200657e-7)
```

Clearly it is and even very strongly. Or is it? Well, let's take a look

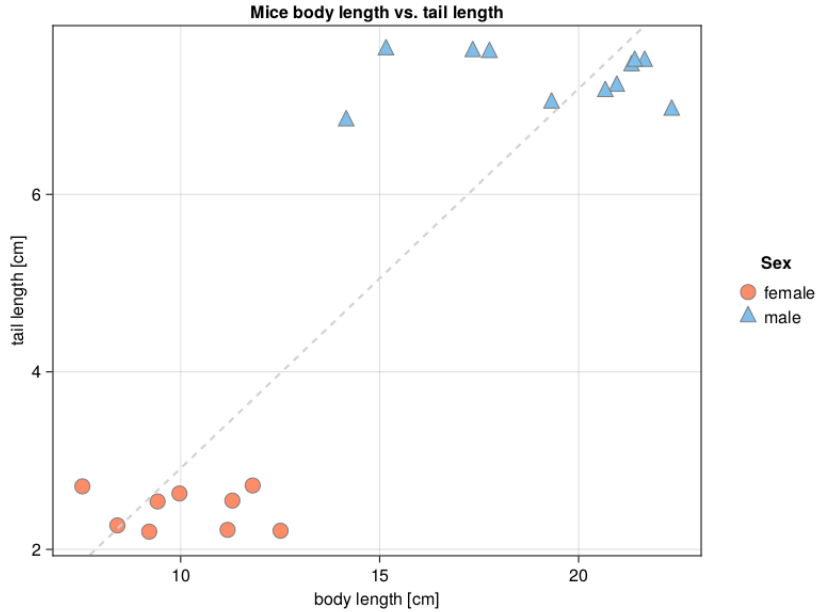

Figure 31: Figure 30: Mice body length vs. tail length.

It turns out that we have two clusters of points. In both of them the points seem to be randomly scattered. This could be confirmed by testing correlation coefficients for the clusters separately.

```
isFemale(value) = value == "f"
isMale(value) = value == "m"

# fml - female mice lengths
# mml - male mice lengths
fml = miceLengths[isFemale.(miceLengths.sex), :] # choose only females
mml = miceLengths[isMale.(miceLengths.sex), :] # choose only males

(
    getCorAndPval(fml.bodyCm, fml.tailCm),
    getCorAndPval(mml.bodyCm, mml.tailCm)
)
```

```
((-0.1593819718041706, 0.6821046994037891),
 (-0.02632446813765734, 0.9387606491398499))
```

Note: The above code snippet uses a single expression functions[313] in the form `functionName(argument) = returnedValue` and Boolean indexing (`isFemale.(miceLengths.sex)` and `isMale.(miceLengths.sex)`) that was discussed briefly in Section 3.3.6 and Section 3.3.7 .

Alternatively, you could read the documentation for the functionality built into `DataFrames.jl` to obtain the desired insight. Doing so takes time, effort, and causes irritation at first (trust me, I know). Still, there are no shortcuts to any place worth going. So, you may decide to use Dfs.groupby[314]and Dfs.combine[315]to get a similar result.

```
# gDf - grouped data frame
Dfs.groupby(miceLengths, :sex) |>
    gDf -> Dfs.combine(gDf, [:tailCm, :bodyCm] => Stats.cor => :r)
```

sex	r
f	−0.1593819718041706
m	−0.02632446813765729

Table 15: Table 12: Pearson correlation coefficients for miceLengths data frame.

Note: You could replace `Stats.cor` with `getCorAndPval` in the snippet above. This should work if you changed the signature of the function from `getCorAndPval(v1::Vector{<:Real}, v2::Vector{<:Real})` to `getCorAndPval(v1::AbstractVector{<:Real}, v2::Abstractvector{<:Real})` first. A more comprehensive `DataFrames` tutorial can be found, e.g. here[316](if you don't know what to do with `*.ipynb` files then you may just click on any of them to see its content in a web browser).

[313]https://en.wikibooks.org/wiki/Introducing_Julia/Functions#Single_expression_functions

[314]https://dataframes.juliadata.org/stable/lib/functions/#DataFrames.groupby

[315]https://dataframes.juliadata.org/stable/lib/functions/#DataFrames.combine

Anyway, the Pearson correlation coefficients are small and not statistically significant ($p > 0.05$). But since the two clusters of points lie on the opposite corners of the graph, then the overall correlation measures their spread alongside the imaginary dashed line in Figure 30 . This inflates the value of the coefficient (compare with the explanation for `z1`, `z2` and `jitter` in Section 7.4). Therefore, it is always good to inspect a graph (scatter plot) to see if there are any clusters of points. The clusters are usually a result of some grouping present in the data (either different experimental groups/treatments or due to some natural grouping). Sometimes we may be unaware of the groups in our data set. Still, if we do know about them, then it is a good idea to inspect the overall correlation and the correlation coefficient for each of the groups separately.

As the last example let's take a look at this data frame.

```
# if you are in 'code_snippets' folder, then use: "./ch07/candyBars.csv"
# if you are in 'ch07' folder, then use: "./candyBars.csv"
candyBars = Csv.read(
    "./code_snippets/ch07/candyBars.csv",
    Dfs.DataFrame)
first(candyBars, 5)
```

total	carb	fat
44.49	30.23	9.67
48.39	29.31	12.48
49.83	30.95	10.58
40.51	25.22	9.89
44.51	29.45	10.15

Table 16: Table 13: Candy bar composition [g] (fictitious data).

Here, we got a data set on composition of different chocolate bars. You are interested to see if the carbohydrate (`carb`) content in bars is associated with their fat mass.

[316]https://github.com/bkamins/Julia-DataFrames-Tutorial/

```
getCorAndPval(candyBars.carb, candyBars.fat)
```

```
(0.12176486958519653, 0.7375535843598793)
```

And it appears it is not. OK, no big deal, and what about `carb` and `total` mass of a candy bar?

```
getCorAndPval(candyBars.carb, candyBars.total)
```

```
(0.822779226943004, 0.0034638410860259317)
```

Now we got it. It's big (r > 0.8) and it's real ($p \leq 0.05$). But did it really make sense to test that?

If we got a random variable `aa` then it is going to be perfectly correlated with itself.

```
Rand.seed!(321)
aa = Rand.rand(Dsts.Normal(100, 15), 10)

getCorAndPval(aa, aa)
```

```
(1.0, 0.0)
```

On the other hand it shouldn't be correlated with another random variable `bb`.

```
bb = Rand.rand(Dsts.Normal(100, 15), 10)

getCorAndPval(aa, bb)
```

```
(0.19399997195558746, 0.5912393958185727)
```

Now, if we add the two variables together we will get the total (`cc`), that will be correlated with both `aa` and `bb`.

```
cc = aa .+ bb
```

```
(
    getCorAndPval(aa, cc),
    getCorAndPval(bb, cc)
)
```

```
((0.7813386818990972, 0.007608814877251513),
 (0.763829856046036, 0.010120085355359132))
```

This is because while correlating `aa` with `cc` we are partially correlating `aa` with itself (`aa .+ bb`). In general, the greater portion of `cc` our `aa` makes the greater the correlation coefficient. So, although possible, it makes little logical sense to compare a part of something with its total. Therefore, in reality running `getCorAndPval(candyBars.carb, candyBars.total)` makes no point despite the interesting result it seems to produce.

Simple Linear Regression

We began Section 7.2 with describing the relation between the volume of water and biomass of two plants of amazon rain forest. Let's revisit the problem.

```
biomass
first(biomass, 5)
```

plantAkg	rainL	plantBkg
20.26	15.09	21.76
9.18	5.32	6.08
11.36	12.5	10.96
11.26	10.7	4.96
9.05	5.7	9.55

Table 17: Effect of rainfall on plants biomass (fictitious data).

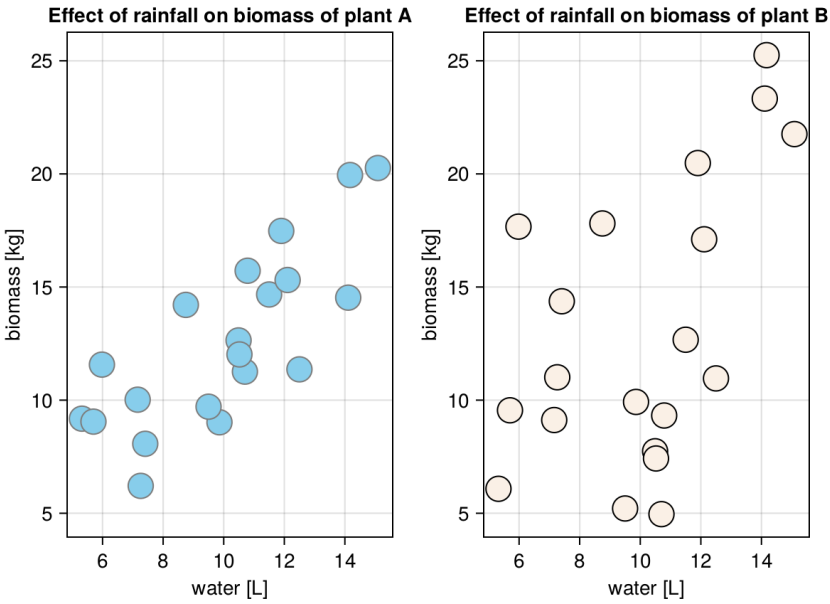

Figure 32: Effect of rainfall on plants' biomass. Revisited.

Previously, we said that the points are scattered around an imaginary line that goes through their center. Now, we could draw that line at a rough guess using a pen and a piece of paper (or a graphics editor). Based on the line we could make a prediction of the values on Y-axis based on the values on the X-axis. The variable placed on the X-axis is called independent (the rain does not depend on a plant, it falls or not), predictor or explanatory variable. The variable placed on the Y-axis is called dependent (the plant depends on rain) or outcome variable. The problem with drawing the line by hand is that it wouldn't be reproducible, a line drawn by the same person would differ slightly from draw to draw. The same is true if a few different people have undertaken this task. Luckily, we got the simple linear regression[317], a method that allows us to draw the same line every single time based on a simple mathematical formula that takes the form:

$y = a + b^*x$, where:

[317]https://en.wikipedia.org/wiki/Simple_linear_regression

- y - predicted value of y
- a - intercept (a point on Y-axis where the imaginary line crosses it at x = 0)
- b - slope (a value by which y increases/decreases when x changes by one unit)
- x - the value of x for which we want to estimate/predict the value of y

The slope (`b`) is fairly easy to calculate with Julia

```
function getSlope(xs::Vector{<:Real}, ys::Vector{<:Real})::Float64
    avgXs::Float64 = Stats.mean(xs)
    avgYs::Float64 = Stats.mean(ys)
    diffsXs::Vector{<:Real} = xs .- avgXs
    diffsYs::Vector{<:Real} = ys .- avgYs
    return sum(diffsXs .* diffsYs) / sum(diffsXs .^ 2)
end
```

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

The function resembles the formula for the covariance that we met in Section 7.3 . The difference is that there we divided `sum(diffs1 .* diffs2)` (here we called it `sum(diffsXs .* diffsYs)`) by the degrees of freedom (`length(v1) - 1`) and here we divide it by `sum(diffsXs .^ 2)`. We might not have come up with the formula ourselves, still, it makes sense given that we are looking for the value by which y changes when x changes by one unit.

Once we got it, we may proceed to calculate the intercept (`a`) like so

```
function getIntercept(xs::Vector{<:Real}, ys::Vector{<:Real})::Float64
    return Stats.mean(ys) - getSlope(xs, ys) * Stats.mean(xs)
end
```

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

And now the results.

```
# be careful, unlike in getCor or getCov, here the order of variables
# in parameters influences the result
plantAIntercept = getIntercept(biomass.rainL, biomass.plantAkg)
plantASlope = getSlope(biomass.rainL, biomass.plantAkg)
plantBIntercept = getIntercept(biomass.rainL, biomass.plantBkg)
plantBSlope = getSlope(biomass.rainL, biomass.plantBkg)

round.([plantASlope, plantBSlope], digits = 2)
```

```
[1.04, 1.14]
```

The intercepts are not our primary interest (we will explain why in a moment or two). We are more concerned with the slopes. Based on the slopes we can say that on average each additional liter or water (`rainL`) translates into 1.04 [kg] more biomass for `plantA` and 1.14 [kg] more biomass for `plantB`. Although, based on the correlation coefficients from Section 7.4 we know that the estimate for `plantB` is less precise. This is because the smaller correlation coefficient means a greater spread of the points along the line as can be seen in the figure below.

```
fig = Cmk.Figure()
ax1 = Cmk.Axis(fig[1, 1],
               title="Effect of rainfall on biomass of plant A",
               xlabel="water [L]", ylabel="biomass [kg]")
Cmk.scatter!(ax1, biomass.rainL, biomass.plantAkg,
             markersize=25, color="skyblue",
             strokewidth=1, strokecolor="gray")
ax2 = Cmk.Axis(fig[1, 2],
               title="Effect of rainfall on biomass of plant B",
               xlabel="water [L]", ylabel="biomass [kg]")
Cmk.scatter!(ax2, biomass.rainL, biomass.plantBkg,
             markersize=25, color="linen",
             strokewidth=1, strokecolor="black")
Cmk.ablines!(ax1, plantAIntercept, plantASlope,
             linestyle=:dash, color="gray")
Cmk.ablines!(ax2, plantBIntercept, plantBSlope,
             linestyle=:dash, color="gray")
Cmk.linkxaxes!(ax1, ax2)
Cmk.linkyaxes!(ax1, ax2)
fig
```

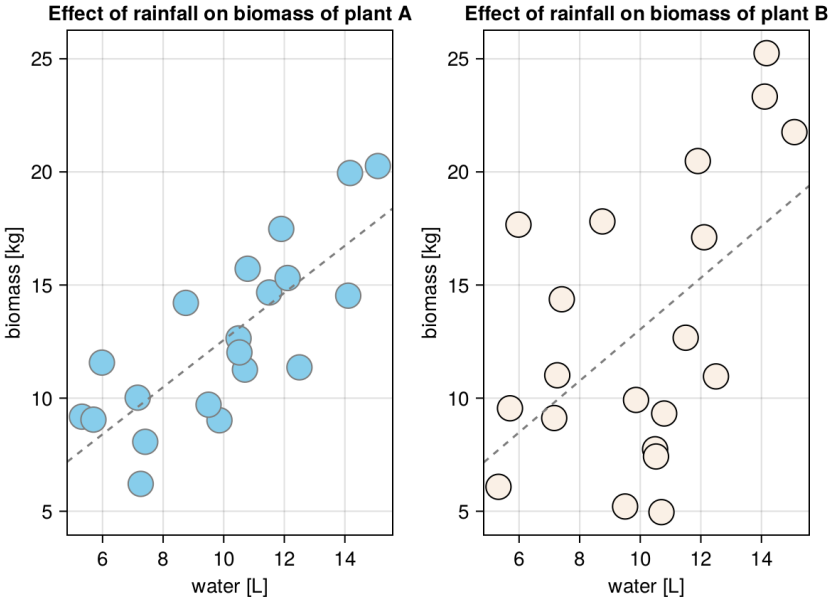

Figure 33: Figure 31: Effect of rainfall on plants' biomass with trend lines superimposed.

The trend line is placed more or less where we would have placed it at a rough guess, so it seems we got our functions right.

Now we can either use the graph (Figure 31) and read the expected value of the variable on the Y-axis based on a value on the X-axis (using a dashed line). Alternatively, we can write a formula based on $y = a + b*x$ we mentioned before to get that estimate.

```
function getPrecictedY(
    x::Float64, intercept::Float64, slope::Float64)::Float64
    return intercept + slope * x
end

round.(
    getPrecictedY.([6.0, 10, 12], plantAIntercept, plantASlope),
    digits = 2)
```

```
[8.4, 12.57, 14.65]
```

It appears to work as expected (to confirm it read from Figure 31 values on Y-axis for the following values on X-axis: [6.0, 10, 12] using the dashed line for `plantA`).

OK, and now imagine you intend to introduce `plantA` into a botanic garden[318]and you want it to grow well and fast. The function `getPrecictedY` tells us that if you pour 35 [L] of water to a field with `plantA` then on average you should get 42 [kg] of the biomass. Unfortunately after you applied the treatment it turned out the biomass actually dropped to 10 [kg] from the field. What happened? Reality. Most likely you (almost) drowned your plant. Lesson to be learned here. It is unsafe to use a model to make predictions beyond the data range on which it was trained. Ultimately, "All models are wrong, but some are useful"[319].

The above is the reason why in most cases we aren't interested in the value of the intercept. The intercept is the value on the Y-axis when X is equal to 0, it is necessary for our model to work, but most likely it isn't very informative (in our case a plant that receives no water simply dies).

So what is regression good for if it only enables us to make a prediction within the range on which it was trained? Well, if you ever underwent spirometry[320]then you used regression in practice (or at least benefited from it). The functional examination of the respiratory system goes as follows. First, you enter your data: name, sex, height, weight, age, etc. Then you breathe (in a manner recommended by a technician) through a mouthpiece connected to an analyzer. Finally, you compare your results with the ones you should have obtained. If, let's say your vital capacity[321]is equal to 5.1 [L] and should be equal to 5 [L] then it is a good sign. However, if the obtained value is equal to 4 [L] when it should be 5 [L] (4/5 = 0.8 = 80% of the norm) then you

[318]https://en.wikipedia.org/wiki/Botanical_garden
[319]https://en.wikipedia.org/wiki/All_models_are_wrong
[320]https://en.wikipedia.org/wiki/Spirometry
[321]https://en.wikipedia.org/wiki/Vital_capacity

should consult your physician. But where does the reference value come from?

One way to get it would be to rely on a large database, of let's say 100-200 million healthy individuals (a data frame with 100-200 million rows and 5-6 columns for age, gender, height, etc. that is stored on a hard drive). Then all you have to do is to find a person (or people) whose data match yours exactly. Then you can take their vital capacity (or their a mean if there is more than one person that matches your features) as a reference point for yours. But this would be a great burden. For once you would have to collect data for a lot of individuals to be pretty sure that an exact combination of a given set of features occurs (hence the 100-200 million mentioned above). The other problem is that such a data frame would occupy a lot of disk space and would be slow to search through. A better solution is regression (most likely multiple linear regression that we will cover in Section 7.7). In that case you collect a smaller sample of let's say 10'000 healthy individuals. You train your regression model. And store it together with the `getPrecictedY` function (where `Y` could be the discussed vital capacity). Now, you can easily and quickly calculate the reference value for a patient even if the exact set of features (values of predictor variables) was not in your training data set (still, you can be fairly sure that the values of the features of the patient are in the range of the training data set).

Anyway, in real life whenever you want to fit a regression line in Julia you should probably use GLM.jl[322]package. In our case an exemplary output for `plantA` looks as follows.

```
import GLM as Glm

mod1 = Glm.lm(Glm.@formula(plantAkg ~ rainL), biomass)
mod1
```

```
plantAkg ~ 1 + rainL
```

[322]https://juliastats.org/GLM.jl/stable/

```
Coefficients:
──────────────────────────────────────────────────────────────────────────
               Coef.  Std. Error     t  Pr(>|t|)  Lower 95%  Upper 95%
──────────────────────────────────────────────────────────────────────────
(Intercept)  2.14751    2.04177   1.05    0.3068  -2.14208     6.43711
rainL        1.04218    0.195771  5.32    <1e-04   0.630877    1.45347
──────────────────────────────────────────────────────────────────────────
```

We begin with `Glm.lm(formula, dataFrame)` (`lm` stands for linear model). Next, we specify our relationship (`Glm.@formula`) in the form `Y ~ X`, where `Y` is the dependent (outcome) variable, ~ is explained by, and `X` is the independent (explanatory) variable. This fits our model (`mod1`) to the data and yields quite some output.

The `Coef.` column contains the values of the intercept (previously estimated with `getIntercept`) and slope (before we used `getSlope` for that). It is followed by the `Std. Error` of the estimation (similar to the `sem` from Section 5.2). Then, just like in the case of the correlation (Section 7.4), some clever mathematical tweaking allows us to obtain a t-statistic for the `Coef.`s and p-values for them. The p-values tell us if the coefficients are really different from 0 (H_0: a `Coeff.` is equal to 0) or estimate the probability that such a big value (or bigger) happened by chance alone (assuming that H_0 is true). Finally, we end up with 95% confidence interval (similar to the one discussed in Section 5.2.1) that (oversimplifying stuff) tells us, with a degree of certainty, within what limits the true value of the coefficient in the population is.

We can use `GLM` to make our predictions as well.

```
round.(
    Glm.predict(mod1, Dfs.DataFrame(Dict("rainL" => [6, 10, 12]))),
    digits = 2
)
```

```
[8.4, 12.57, 14.65]
```

For that to work we feed `Glm.predict` with our model (`mod1`) and a `DataFrame` containing a column `rainL` that was used as a predictor in

our model and voila, the results match those returned by `getPrecictedY` somewhat before in this section.

We can also get the general impression of how imprecise our prediction is by using the residuals (differences between the predicted and actual value on the Y-axis). Like so

```
# an average estimation error in prediction
# (based on abs differences)
function getAvgEstimError(
    lm::Glm.StatsModels.TableRegressionModel)::Float64
    return abs.(Glm.residuals(lm)) |> Stats.mean
end

getAvgEstimError(mod1)
```

2.075254994044967

So, on average our model miscalculates the value on the Y-axis (`plantAkg`) by 2 units (here kilograms). Of course, this is a slightly optimistic view, since we expect that on a new, previously unseen data set, the prediction error will be greater.

Moreover, the package allows us to calculate other useful stuff, like the coefficient of determination[323]that tells us how much change in the variability on Y-axis is explained by our model (our explanatory variable(s)).

```
(
    Glm.r2(mod1),
    Stats.cor(biomass.rainL, biomass.plantAkg) ^ 2
)
```

```
(0.6115596392611107, 0.6115596392611111)
```

The coefficient of determination is called r^2 (r squared) and in this case (simple linear regression) it is equal to the Pearson's correlation coefficient (denoted as `r`) times itself. As we can see our model explains roughly 61% of variability in `plantAkg` biomass.

[323]https://en.wikipedia.org/wiki/Coefficient_of_determination

Multiple Linear Regression

Multiple linear regression is a linear regression with more than one predictor variable. Take a look at the Icecream[324]data frame.

```
ice = RD.dataset("Ecdat", "Icecream")
first(ice, 5)
```

Cons	Income	Price	Temp
0.386	78.0	0.27	41.0
0.374	79.0	0.282	56.0
0.393	81.0	0.277	63.0
0.425	80.0	0.28	68.0
0.406	76.0	0.272	69.0

Table 18: Table 14: Icecream consumption data.

We got 4 columns altogether (more detail in the link above):

- `Cons` - consumption of ice cream (pints),
- `Income` - average family income (USD),
- `Price` - price of ice cream (USD),
- `Temp` - temperature (Fahrenheit)

Imagine you are an ice cream truck owner and are interested to know which factors influence (and in what way) the consumption (`Cons`) of ice-cream by your customers. Let's start by building a model with all the possible explanatory variables.

```
iceMod1 = Glm.lm(Glm.@formula(Cons ~ Income + Price + Temp), ice)
iceMod1
```

```
──────────────────────────────────────────────────────────────────────────
               Coef.  Std. Error      t  Pr(>|t|)  Lower 95%  Upper 95%
──────────────────────────────────────────────────────────────────────────
(Intercept)   0.1973      0.2702   0.73    0.4718    -0.3581     0.7528
Income        0.0033      0.0012   2.82    0.0090     0.0009     0.0057
Price        -1.0444      0.8344  -1.25    0.2218    -2.7595     0.6706
```

[324]https://vincentarelbundock.github.io/Rdatasets/doc/Ecdat/Icecream.html

```
Temp          0.0035      0.0004   7.76    <1e-99     0.0025     0.0044
```

Right away we can see that the price of ice-cream negatively affects (`Coef.` = −1.044) the volume of ice cream consumed (the more expensive the ice cream is the less people eat it, 1.044 pint less for every additional USD of price). The relationship is in line with our intuition. However, there is not enough evidence (p > 0.05) that the real influence of `Price` on consumption isn't 0 (so no influence). Therefore, you wonder should you perhaps remove the variable `Price` from the model like so

```
iceMod2 = Glm.lm(Glm.@formula(Cons ~ Income + Temp), ice)
iceMod2
```

```
                Coef.  Std. Error      t  Pr(>|t|)  Lower 95%  Upper 95%
(Intercept)  -0.1132      0.1083  -1.05    0.3051    -0.3354     0.109
Income        0.0035      0.0012   3.02    0.0055     0.0011     0.0059
Temp          0.0035      0.0004   7.96    <1e-99     0.0026     0.0045
```

Now, we got `Income` and `Temp` in our model, both of which are statistically significant. The values of `Coef.`s for `Income` and `Temp` somewhat changed between the models, but such changes (and even greater) are to be expected. Still, we would like to know if our new `iceMod2` is really better than `iceMod1` that we came up with before.

In our first try to solve the problem we could resort to the coefficient of determination (r^2) that we met in Section 7.6 . Intuition tells us that a better model should have a bigger r^2.

```
round.([Glm.r2(iceMod1), Glm.r2(iceMod2)],
    digits = 3)
```

```
[0.719, 0.702]
```

Hmm, r^2 is bigger for `iceMod1` than `iceMod2`. However, there are two problems with it: 1) the difference between the coefficients is quite small, and 2) r^2 gets easily inflated by any additional variable in the model. And I mean any, if you add, let's say 10 random variables to the `ice` data frame and put them into a model the coefficient of determination will go up even though this makes no sense (we know their real influence is 0). That is why we got an improved metrics called the adjusted coefficient of determination. This parameter (adj. r^2) penalizes for every additional variable added to our model. Therefore the 'noise' variables will lower the adjusted r^2 whereas only truly impactful ones will be able to raise it.

```
round.([Glm.adjr2(iceMod1), Glm.adjr2(iceMod2)],
    digits = 3)
```

```
[0.687, 0.68]
```

`iceMod1` still explains more variability in `Cons` (ice cream consumption), but the magnitude of the difference dropped. This makes our decision even harder. Luckily, `Glm` has `ftest` function to help us determine if one model is significantly better than the other.

```
Glm.ftest(iceMod1.model, iceMod2.model)
```

```
F-test: 2 models fitted on 30 observations
──────────────────────────────────────────────────────────────────
     DOF  ΔDOF     SSR    ΔSSR      R²      ΔR²      F*   p(>F)
──────────────────────────────────────────────────────────────────
[1]    5        0.0353          0.7190
[2]    4    -1  0.0374  0.0021  0.7021  -0.0169  1.5669  0.2218
──────────────────────────────────────────────────────────────────
```

The table contains two rows:

- `[1]` - first model from the left (in `Glm.ftest` argument list)
- `[2]` - second model from the left (in `Glm.ftest` argument list)

and a few columns:

- `DOF` - degrees of freedom (more elements in formula, bigger `DOF`)
- `ΔDOF` - `DOF[2]` - `DOF[1]`
- `SSR` - residual sum of squares (the smaller the better)
- `ΔSSR` - `SSR[2]` - `SSR[1]`
- `R2` - coefficient of determination (the bigger the better)
- `ΔR2` - `R2[2]` - `R2[1]`
- `F*` - F-Statistic (similar to the one we met in Section 5.4)
- `p(>F)` - p-value that you obtain F-statistic greater than the one in the previous column by chance alone (assuming both models are equally good)

Based on the test we see that none of the models is clearly better than the other ($p > 0.05$). Therefore, in line with Occam's razor[325] principle (when two equally good explanations exist, choose the simpler one) we can safely pick `iceMod2` as our final model.

What we did here was the construction of a so called minimal adequate model (the smallest model that explains the greatest amount of variance in the dependent/outcome variable). We did this using top to bottom approach. We started with a 'full' model. Then, we followed by removing explanatory variables (one by one) that do not contribute to the model (we start from the highest p-value above 0.05) until only meaningful explanatory variables remain. The removal of the variables reflects our common sense, because usually we (or others that will use our model) do not want to spend time/money/energy on collecting data that are of no use to us.

OK, let's inspect our minimal adequate model again.

```
iceMod2
```

```
──────────────────────────────────────────────────────────────────────
               Coef.  Std. Error      t  Pr(>|t|)  Lower 95%  Upper 95%
──────────────────────────────────────────────────────────────────────
(Intercept)  -0.1132      0.1083  -1.05    0.3051    -0.3354     0.109
Income        0.0035      0.0012   3.02    0.0055     0.0011     0.0059
```

[325] https://en.wikipedia.org/wiki/Occam%27s_razor

```
Temp          0.0035      0.0004   7.96    <1e-99     0.0026     0.0045
```

We can see that for every extra dollar of `Income` our customers consume 0.003 pint (~1.47 mL) of ice cream more. Roughly the same change is produced by each additional grade (in Fahrenheit) of temperature. So, a simultaneous increase in `Income` by 1 USD and `Temp` by 1 unit translates into roughly 0.003 + 0.003 = 0.006 pint (~2.94 mL) greater consumption of ice cream per person. Now, (remember you were to imagine you are an ice cream truck owner) you could use the model to make predictions (with `Glm.predict` as we did in Section 7.6) to your benefit (e.g. by preparing enough product for your customers on a hot day).

So the time passes by and one sunny day when you open a bottle of beer a drunk genie pops out of it. To compensate you for the lost beer he offers to fulfill one wish (shouldn't there be three?). He won't shower you with cash right away since you will not be able to explain it to the tax office. Instead, he will give you the ability to control either `Income` or `Temp` variable at will. That way you will get your money and none is the wiser. Which one do you choose, answer quickly, before the genie changes his mind.

Hmm, now that's a dilemma, but judging by the coefficients above it seems it doesn't make much of a difference (both `Coef.`s are roughly equal to 0.0035). Or does it? Well, the `Coef.`s are similar, but we are comparing incomparable, i.e. dollars (`Income`) with degrees Fahrenheit (`Temp`) and their influence on `Cons`. We may however, standardize the coefficients[326]to overcome the problem.

```
# fn from ch04
# how many std. devs is a value above or below the mean
function getZScore(value::Real, mean::Real, sd::Real)::Float64
    return (value - mean)/sd
end

# adding new columns to the data frame
```

[326]https://en.wikipedia.org/wiki/Standardized_coefficient

```
ice.ConsStand = getZScore.(
    ice.Cons, Stats.mean(ice.Cons), Stats.std(ice.Cons))
ice.IncomeStand = getZScore.(
    ice.Income, Stats.mean(ice.Income), Stats.std(ice.Income))
ice.TempStand = getZScore.(
    ice.Temp, Stats.mean(ice.Temp), Stats.std(ice.Temp))

iceMod2Stand = Glm.lm(
    Glm.@formula(ConsStand ~ IncomeStand + TempStand), ice)
iceMod2Stand
```

```
ConsStand ~ 1 + IncomeStand + TempStand

Coefficients:
──────────────────────────────────────────────────────────────────────────
                    Coef.  Std. Error      t  Pr(>|t|)  Lower 95%  Upper
95%
──────────────────────────────────────────────────────────────────────────
(Intercept)  -7.92872e-17    0.103281  -0.00    1.0000  -0.211916
0.211916
IncomeStand   0.335122       0.111065   3.02    0.0055   0.107235
0.563009
TempStand     0.884442       0.111065   7.96    <1e-07   0.656555
1.11233
──────────────────────────────────────────────────────────────────────────
```

When expressed on the same scale (using `getZScore` function we met in Section 4.6.2) it becomes clear that the `Temp` (`Coef.` ~0.884) is a much more influential factor with respect to ice cream consumption (`Cons`) than `Income` (`Coef.` ~0.335). Therefore, we can be pretty sure that modifying the temperature by 1 standard deviation (which should not attract much attention) will bring you more money than modifying customers' income by 1 standard deviation. Thanks genie.

Let's look at another example of regression to get a better feel of it and discuss categorical variables and an interaction term in the model. We will operate on agefat[327] data frame.

```
agefat = RD.dataset("HSAUR", "agefat")
```

[327] https://vincentarelbundock.github.io/Rdatasets/doc/HSAUR/agefat.html

Age	Fat	Sex
24	15.5	male
37	20.9	male
41	18.6	male
60	28.0	male
31	34.7	female

Table 19: Table 15: Total body composition.

Here we are interested to predict body fat percentage (`Fat`) from the other two variables. Let's get down to business.

```
agefatM1 = Glm.lm(Glm.@formula(Fat ~ Age + Sex), agefat)
agefatM1
```

```
──────────────────────────────────────────────────────────────────────────
                 Coef.  Std. Error      t  Pr(>|t|)  Lower 95%  Upper 95%
──────────────────────────────────────────────────────────────────────────
(Intercept)   19.6479      4.1078   4.78    0.0001    11.1288    28.1669
Age            0.2656      0.0795   3.34    0.0030     0.1006     0.4305
Sex: male    -10.5489      2.0914  -5.04    <1e-99   -14.8862    -6.2116
──────────────────────────────────────────────────────────────────────────
```

It appears that the older a person is the more fat it has (+0.27% of body fat per 1 extra year of age). Moreover, male subjects got smaller percentage of body fat (on average by 10.5%) than female individuals (this is to be expected: see here[328]). In the case of categorical variables the reference group is the one that comes first in the alphabet (here `female` is before `male`). The internals of the model assign 0 to the reference group and 1 to the other group. This yields us the formula: $y = a + b*x + c*z$ or $Fat = a + b*Age + c*Sex$, where `Sex` is 0 for `female` and 1 for `male`. As before we can use this formula for prediction (either write a new `getPredictedY` function on your own or use `Glm.predict` we met before).

We may also want to fit a model with an interaction term (`+ Age&Sex`) to see if we gain some additional precision in our predictions.

[328]https://en.wikipedia.org/wiki/Body_fat_percentage

```
# or shortcut: Glm.@formula(Fat ~ Age * Sex)
agefatM2 = Glm.lm(Glm.@formula(Fat ~ Age + Sex + Age&Sex), agefat)
agefatM2
```

```
──────────────────────────────────────────────────────────────────────────
                    Coef.  Std. Error      t  Pr(>|t|)  Lower 95%  Upper
95%
──────────────────────────────────────────────────────────────────────────
(Intercept)       25.6686      5.3281   4.82    0.0001    14.5882
36.7491
Age                0.1437      0.1053   1.36    0.1869    -0.0753
0.3627
Sex: male        -21.7625      6.9625  -3.13    0.0051   -36.2418
-7.2833
Age & Sex: male    0.2575      0.1531   1.68    0.1073    -0.0608
0.5758
──────────────────────────────────────────────────────────────────────────
```

Here, we do not have enough evidence that the interaction term (`Age & Sex: male`) matters ($p > 0.05$). Still, let's explain what is this interaction in case you ever find one that is important. For that, take a look at the graph below.

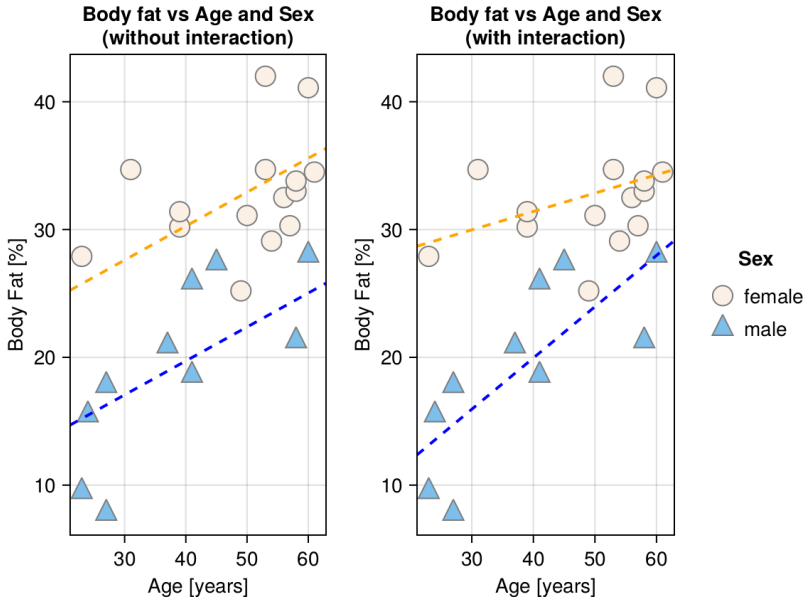

Figure 34: Figure 32: Body fat percentage vs. Age and Sex

As you can see the model without interaction fits two regression lines (one for each `Sex`) with different intercepts, but the same slopes. On the other hand, the model with interaction fits two regression lines (one for each `Sex`) with different intercepts and different slopes. Since the coefficient (`Coef.`) for the interaction term (`Age & Sex: male`) is positive, this means that the slope for `Sex: male` is more steep (more positive). This would suggest that males tend to accumulate fat at a faster rate as they age.

So, when to use an interaction term in your model? The advice I heard was that in general, you should construct simple models and only use an interaction term when there are some good reasons for it. For instance, in the discussed case (`agefat` data frame), we might wanted to answer the research question: Does the accretion of body fat occurs faster in one of the genders as people age?

Exercises - Association and Prediction

Just like in the previous chapters here you will find some exercises that you may want to solve to get from this chapter as much as you can (best option). Alternatively, you may read the task descriptions and the solutions (and try to understand them).

Exercise 1

The `RDatasets` package mentioned in Section 7.5 contains a lot of interesting data. For instance the Animals[329] data frame.

```
animals = RD.dataset("MASS", "Animals")
first(animals, 5)
```

Species	Body	Brain
Mountain beaver	1.35	8.1
Cow	465.0	423.0
Grey wolf	36.33	119.5
Goat	27.66	115.0
Guinea pig	1.04	5.5

Table 20: Table 16: DataFrame for brain and body weights of 28 animal species.

Since this chapter is about association then we are interested to know if body [kg] and brain weights [kg] of the animals are correlated. Let's take a sneak peak at the data points.

[329] https://vincentarelbundock.github.io/Rdatasets/doc/MASS/Animals.html

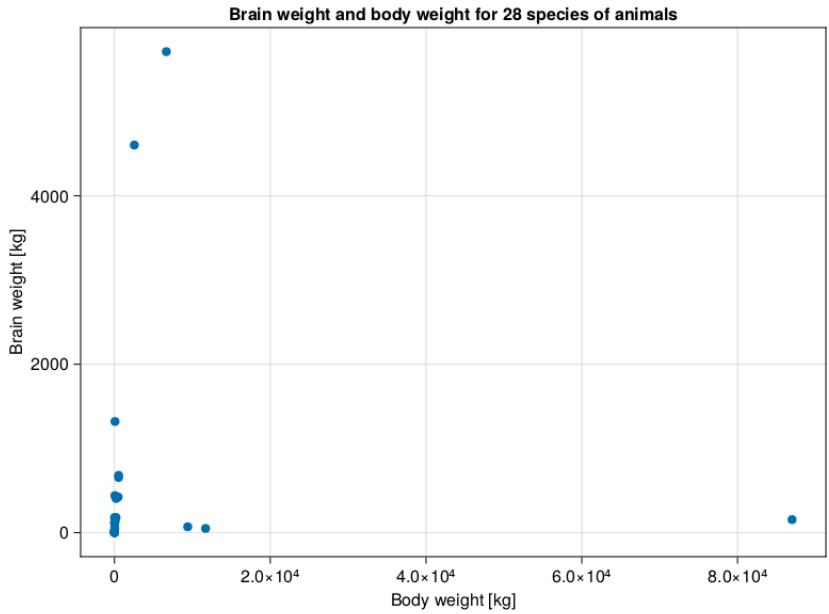

Figure 35: Figure 33: Body and brain weight of 28 animal species.

Hmm, at first sight the data looks like a little mess. Most likely because of the large range of data on X- and Y-axis. Moreover, the fact that some animals got large body mass with relatively small brain weight doesn't help either. Still, my impression is that in general (except for the first three points from the right) greater body weight is associated with a greater brain weight. However, it is quite hard to tell for sure as the points on the left are so close to each other on the scale of X-axis. So, let's put that to the test.

```
getCorAndPval(animals.Body, animals.Brain)
```

```
(-0.005341162561251125, 0.9784802067532018)
```

The Pearson's correlation coefficient is not able to discern the points and confirm that either. Nevertheless, let's narrow our ranges by taking logarithms (with `log10` function) of the data and look at the scatter plot again.

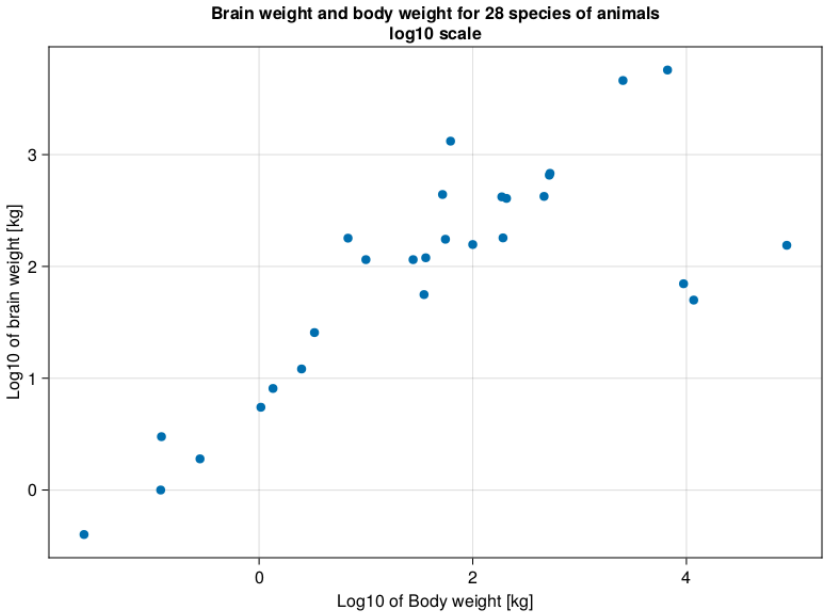

Figure 36: Figure 34: Body (log10) and brain (log10) weight of 28 animal species.

The impression we get is quite different than before. The points are much better separated. The three outliers remain, but they are much closer to the imaginary trend line. Now we would like to express that relationship. One way to do it is with Spearman's rank correlation coefficient[330]. As the name implies instead of correlating the numbers themselves it correlates their ranks.

Note: It might be a good idea to examine the three outliers and see do they have anything in common. If so, we might want to determine the relationship between X- and Y- variable (even on the original, non-`log10` scale) separately for the outliers and the remaining animals. Here, the three outliers are dinosaurs, whereas rest of the animals are mammals. This could explain why the association is different in these two groups of animals.

[330]https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient

So here is a warm up task for you.

Write a `getSpearmCorAndPval` function and run it on `animals` data frame. To do that first you will need a function `getRanks(v::Vector{<:Real})::Vector{<:Float64}` that returns the ranks for you like this.

```
getRanks([500, 100, 1000]) # returns [2.0, 1.0, 3.0]
getRanks([500, 100, 500, 1000]) # returns [2.5, 1.0, 2.5, 4.0]
getRanks([500, 100, 500, 1000, 500]) # returns [3.0, 1.0, 3.0, 5.0, 3.0]
# etc.
```

Personally, I found `Base.findall` and `Base.sort` to be useful while writing `getRanks`, but feel free to employ whatever constructs you want. Anyway, once you got it, you can apply it to get Spearman's correlation coefficient (`getCorAndPval(getRanks(v1), getRanks(v2))`).

> ***Note:*** In real life to calculate the coefficient you would probably use StatsBase.corspearman[331].

Exercise 2

P-value multiplicity correction, a classic theme in this book. Let's revisit it again. Take a look at the following data frame.

```
Rand.seed!(321)

letters = map(string, 'a':'j')
bogusCors = Dfs.DataFrame(
    Dict(l => Rand.rand(Dsts.Normal(100, 15), 10) for l in letters)
)
bogusCors[1:3, 1:3]
```

[331] https://juliastats.org/StatsBase.jl/stable/ranking/#StatsBase.corspearman

a	b	c
102.04452249090404	126.62114430860125	72.58784224875757
81.10997573989799	101.02869856127887	123.65904493232378
85.54321961150684	109.98477666117208	132.32635179854458

Table 21: Table 17: DataFrame with random variables for bogus correlations.

It contains a random made up data. In total we can calculate `binomial(10, 2)` = 45 different unique correlations for the 10 columns we got here. Out of them roughly 2-3 (`binomial(10, 2) * 0.05` = 2.25) would appear to be valid correlations ($p \leq 0.05$), but in reality were the false positives (since we know that each column is a random variable obtained from the same distribution). So here is a task for you. Write a function that will return all the possible correlations (coefficients and p-values). Check how many of them are false positives. Apply a multiplicity correction (e.g. `Mt.BenjaminiHochberg()` we met in Section 5.6) to the p-values and check if the number of false positives drops to zero.

Exercise 3

Sometimes we would like to have a quick visual way to depict all the correlations in one plot to get a general impression of the correlations in the data (and possible patterns present). One way to do this is to use a so called heatmap.

So, here is a task for you. Read the documentation and examples for CairoMakie's heatmap[332](or a heatmap from other plotting library) and for the data in `bogusCors` from the previous section create a graph similar to the one you see below.

[332]https://docs.makie.org/stable/reference/plots/heatmap/

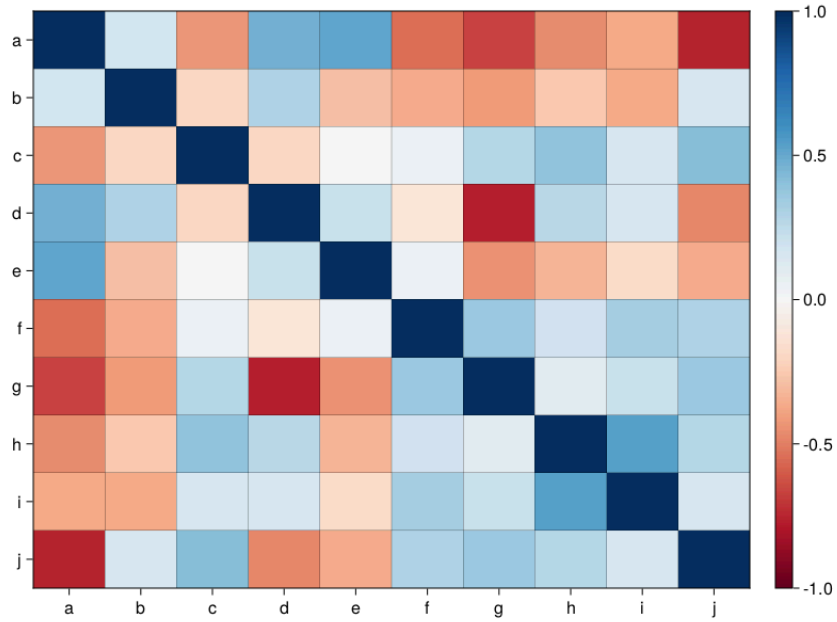

Figure 37: Figure 35: Correlation heatmap for data in `bogusCors`.

The graph depicts the Pearson's correlation coefficients for all the possible correlations in `bogusCors`. Positive correlations are depicted as the shades of blue, negative correlations as the shades of red.

Your figure doesn't have to be the exact replica of mine, for instance you may choose a different color map[333].

If you like challenges you may add (write it in the center of a given square) the value of the correlation coefficient (rounded to let's say 2 decimal digits). Furthermore, you may add a significance marker (e.g. if a 'raw' p-value is ≤ 0.05 put '#' character in a square) for the correlations.

Exercise 4

Linear regression just like other methods mentioned in this book got its assumptions[334]that if possible should be verified. The R

[333]https://docs.makie.org/stable/explanations/colors/

[334]https://en.wikipedia.org/wiki/Regression_analysis#Underlying_assumptions

programming language got plot.lm[335] function to verify them graphically. The two most important plots (or at least the ones that I understand the best) are scatter-plot of residuals vs. fitted values and Q-Q plot[336] of standardized residuals (see Figure 36 below).

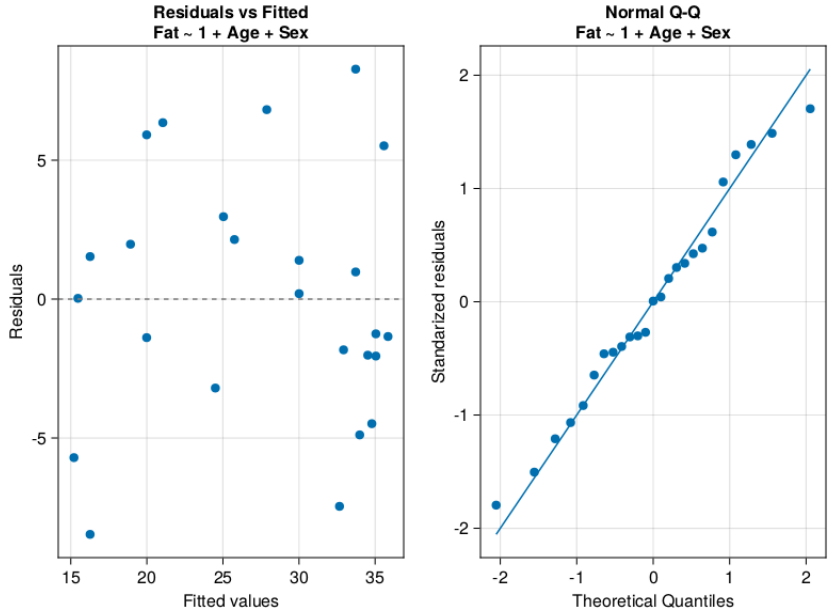

Figure 38: Figure 36: Diagnostic plot for regression model (ageFatM1).

If the assumptions hold, then the points in residuals vs. fitted plot should be randomly scattered around 0 (on Y-axis) with equal spread of points from left to right and no apparent pattern visible. On the other hand, the points in Q-Q plot should lie along the Q-Q line which indicates their normal distribution. To me (I'm not an expert though) the above seem to hold in Figure 36 above. If that was not the case then we should try to correct our model. We might transform one or more variables (for instance by using `log10` function we met in Section 7.8.1) or fit a different model. Otherwise, the model we got may give poor predictions. For instance, if our residuals vs. fitted plot displayed

[335] https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/plot.lm
[336] https://en.wikipedia.org/wiki/Q%E2%80%93Q_plot

a greater spread of points on the right side of X-axis, then most likely our predictions would be more off for large values of explanatory variable(s).

Anyway, your task here is to write a function `drawDiagPlot` that accepts a linear regression model and returns a graph similar to Figure 36 above (when called with `ageFatM1` as an input).

Below you will find some (but not all) of the functions that I found useful while solving this task (feel free to use whatever functions you want):

- `Glm.predict`
- `Glm.residuals`
- `string(Glm.formula(mod))`
- `Cmk.qqplot`

The rest is up to you.

Exercise 5

While developing the solution to exercise 4 (Section 7.9.4) we pointed out on the flaws of `iceMod2`. We decided to develop a better model. So, here is a task for you.

Read about constructing formula programmatically[337] using `StatsModels` package (`GLM` uses it internally).

Next, given the `ice2` data frame below.

```
Rand.seed!(321)

ice = RD.dataset("Ecdat", "Icecream") # reading fresh data frame
ice2 = ice[2:end, :] # copy of ice data frame
# an attempt to remove autocorrelation from Temp variable
ice2.TempDiff = ice.Temp[2:end] .- ice.Temp[1:(end-1)]

# dummy variables aimed to confuse our new function
ice2.a = Rand.rand(-100:1:100, 29)
ice2.b = Rand.rand(-100:1:100, 29)
ice2.c = Rand.rand(-100:1:100, 29)
```

[337] https://juliastats.org/StatsModels.jl/stable/formula/#Constructing-a-formula-programmatically-1

```
ice2.d = Rand.rand(-100:1:100, 29)
ice2
```

Write a function that returns the minimal adequate model.

```
# return a minimal adequate (linear) model
function getMinAdeqMod(
    df::Dfs.DataFrame, y::String, xs::Vector{<:String}
    )::Glm.StatsModels.TableRegressionModel
```

The function accepts a data frame (`df`), name of the outcome variable (`y`), and names of the explanatory variables (`xs`). In its insides the function builds a full additive linear model (`y ~ x1 + x2 + ... + etc.`). Then, it eliminates an `x` (predictor variable) with the greatest p-value (only if it is greater than 0.05). The removal process is continued for all `xs` until only `xs` with p-values ≤ 0.05 remain. If none of the `xs` is impactful it should return the model in the form `y ~ 1` (the intercept of this model is equal to `Stats.mean(y)`). Test it out, e.g. for `getMinAdeqMod(ice2, names(ice2)[1], names(ice2)[2:end])` it should return a model in the form `Cons ~ Income + Temp + TempDiff`.

Hint: You can extract p-values for the coefficients of the model with `Glm.coeftable(m).cols[4]`. GLM got its own function for constructing model terms (`Glm.term`). You can add the terms either using `+` operator or `sum` function (if you got a vector of terms).

Solutions - Association

In this sub-chapter you will find exemplary solutions to the exercises from the previous section.

Solution to Exercise 1

Let's write `getRanks`, but let's start simple and use it on a sorted vector `[100, 500, 1000]` without ties. In this case the body of `getRanks` function would be something like.

```
# for now the function is without types
function getRanksVer1(v)
```

```
    # or: ranks = collect(1:length(v))
    ranks = collect(eachindex(v))
    return ranks
end

getRanksVer1([100, 500, 1000])
```

```
[1, 2, 3]
```

Time to complicate stuff a bit by adding some ties in numbers.

```
# for now the function is without types
function getRanksVer2(v)
    initialRanks = collect(eachindex(v))
    finalRanks = zeros(length(v))
    for i in eachindex(v)
        indicesInV = findall(x -> x == v[i], v)
        finalRanks[i] = Stats.mean(initialRanks[indicesInV])
    end
    return finalRanks
end

(
    getRanksVer2([100, 500, 500, 1000]),
    getRanksVer2([100, 500, 500, 500, 1000])
)
```

```
([1.0, 2.5, 2.5, 4.0],
 [1.0, 3.0, 3.0, 3.0, 5.0])
```

The `findall` function accepts a function (here `x -> x == v[i]`) and a vector (here `v`). Next, it runs the function on every element of the vector and returns the indices for which the result was `true`. Here we are looking for elements in `v` that are equal to the currently examined (`v[i]`) element of `v`. Then, we use `indicesInV` to get the `initialRanks`. The `initialRanks[indicesInV]` returns a `Vector` that contains one or more (if ties occur) `initialRanks` for a given element of `v`. Finally, we calculate the average rank for a given number in `v` by using `Stats.mean`. The function may be sub-optimall as for `[100, 500, 500, 1000]` the average rank for `500` is calculated twice (once for `500` at index 2 and once for `500` at index 3) and for `[100, 500, 500, 500,`

`1000]` the average rank for `500` is calculated three times. Still, we are more concerned with the correct result and not the efficiency (assuming that the function is fast enough) so we will leave it as it is.

Now, the final tweak. The input vector is shuffled.

```
# for now the function is without types
function getRanksVer3(v)
    sortedV = collect(sort(v))
    initialRanks = collect(eachindex(sortedV))
    finalRanks = zeros(length(v))
    for i in eachindex(v)
        indicesInSortedV = findall(x -> x == v[i], sortedV)
        finalRanks[i] = Stats.mean(initialRanks[indicesInSortedV])
    end
    return finalRanks
end

(
    getRanksVer3([500, 100, 1000]),
    getRanksVer3([500, 100, 500, 1000]),
    getRanksVer3([500, 100, 500, 1000, 500])
)
```

```
([2.0, 1.0, 3.0],
 [2.5, 1.0, 2.5, 4.0],
 [3.0, 1.0, 3.0, 5.0, 3.0])
```

Here, we let the built in function `sort` to arrange the numbers from `v` in the ascending order. Then for each number from `v` we get its indices in `sortedV` and their corresponding ranks based on that (`initialRanks[indicesInSortedV]`). As in `getRanksVer2` the latter is used to calculate their average.

OK, time for cleanup + adding some types for future references (before we forget them).

```
function getRanks(v::Vector{<:Real})::Vector{<:Float64}
    sortedV::Vector{<:Real} = collect(sort(v))
    initialRanks::Vector{<:Int} = collect(eachindex(sortedV))
    finalRanks::Vector{<:Float64} = zeros(length(v))
    for i in eachindex(v)
        indicesInSortedV = findall(x -> x == v[i], sortedV)
        finalRanks[i] = Stats.mean(initialRanks[indicesInSortedV])
```

```
    end
    return finalRanks
end

(
    getRanks([100, 500, 1000]),
    getRanks([100, 500, 500, 1000]),
    getRanks([500, 100, 1000]),
    getRanks([500, 100, 500, 1000]),
    getRanks([500, 100, 500, 1000, 500])
)
```

```
([1.0, 2.0, 3.0],
 [1.0, 2.5, 2.5, 4.0],
 [2.0, 1.0, 3.0],
 [2.5, 1.0, 2.5, 4.0],
 [3.0, 1.0, 3.0, 5.0, 3.0])
```

At long last we can define `getSpearmCorAndPval` and apply it to `animals` data frame.

```
function getSpearmCorAndPval(
    v1::Vector{<:Real}, v2::Vector{<:Real})::Tuple{Float64, Float64}
    return getCorAndPval(getRanks(v1), getRanks(v2))
end

getSpearmCorAndPval(animals.Body, animals.Brain)
```

```
(0.7162994456021085, 1.8128636948722132e-5)
```

The result appears to reflect the general relationship well (compare with Figure 34).

Solution to Exercise 2

The solution should be quite simple assuming you did solve exercise 4 from ch05 (see Section 5.7.4 and Section 5.8.4) and exercise 5 from ch06 (see Section 6.7.5 and Section 6.8.5).

Let's start with the helper functions, `getUniquePairs` (Section 5.8.4) and `getSortedKeysVals` (Section 4.5) that we developed previously. For your convenience I paste them below.

```
function getUniquePairs(names::Vector{T})::Vector{Tuple{T,T}} where T
    @assert (length(names) >= 2) "the input must be of length >= 2"
    uniquePairs::Vector{Tuple{T,T}} =
        Vector{Tuple{T,T}}(undef, binomial(length(names), 2))
    currInd::Int = 1
    for i in eachindex(names)[1:(end-1)]
        for j in eachindex(names)[(i+1):end]
            uniquePairs[currInd] = (names[i], names[j])
            currInd += 1
        end
    end
    return uniquePairs
end

function getSortedKeysVals(d::Dict{T1,T2})::Tuple{
    Vector{T1},Vector{T2}} where {T1,T2}
    sortedKeys::Vector{T1} = keys(d) |> collect |> sort
    sortedVals::Vector{T2} = [d[k] for k in sortedKeys]
    return (sortedKeys, sortedVals)
end
```

Now, time to get all possible 'raw' correlations.

```
function getAllCorsAndPvals(
    df::Dfs.DataFrame, colsNames::Vector{String}
)::Dict{Tuple{String,String},Tuple{Float64,Float64}}

    uniquePairs::Vector{Tuple{String,String}} =
getUniquePairs(colsNames)
    allCors::Dict{Tuple{String,String},Tuple{Float64,Float64}} = Dict(
        (n1, n2) => getCorAndPval(df[!, n1], df[!, n2]) for (n1, n2)
        in
        uniquePairs)

    return allCors
end
```

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

We start by getting the `uniquePairs` for the columns of interest `colNames`. Then we use dictionary comprehension to get our result. We iterate through each pair `for (n1, n2) in uniquePairs`. Each `uniquePair` is composed of a tuple `(n1, n2)`, where `n1` - name1, `n2` - name2. While traversing the `uniquePairs` we calculate the correlations and p-values (`getCorAndPval`) by selecting columns of

interest (`df[:, n1]` and `df[:, n2]`). And that's it. Let's see how it works and how many false positives we got (remember, we expect 2 or 3).

```
allCorsPvals = getAllCorsAndPvals(bogusCors, letters)
falsePositves = (map(t -> t[2], values(allCorsPvals)) .<= 0.05) |> sum
falsePositves
```

```
3
```

First, we extract the values from our dictionary with `values(allCorsPvals)`. The values are a vector of tuples `[(cor, pval)]`. To get p-values alone, we use map function that takes every tuple (`t`) and returns its second element (`t[2]`). Finally, we compare the p-values with our cutoff level for type 1 error ($\alpha = 0.05$). And sum the `Bool`s (each `true` is counted as 1, and each `false` as 0).

Anyway, as expected we got 3 false positives. All that's left to do is to apply the multiplicity correction.

```
function adjustPvals(
    corsAndPvals::Dict{Tuple{String,String},Tuple{Float64,Float64}},
    adjMeth::Type{M}
)::Dict{Tuple{String,String},Tuple{Float64,Float64}} where
    {M<:Mt.PValueAdjustment}

    ks, vs = getSortedKeysVals(corsAndPvals)
    cors::Vector{<:Float64} = map(t -> t[1], vs)
    pvals::Vector{<:Float64} = map(t -> t[2], vs)
    adjustedPVals::Vector{<:Float64} = Mt.adjust(pvals, adjMeth())
    newVs::Vector{Tuple{Float64,Float64}} = collect(
        zip(cors, adjustedPVals))

    return Dict(ks[i] => newVs[i] for i in eachindex(ks))
end
```

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

The code is rather self explanatory and relies on step by step operations: 1) getting our p-values (`pvals`), 2) applying an adjustment method (`adjMeth`) on them (`Mt.adjust`), and 3) combining the adjusted p-values (`adjustedPVals`) with `cors` again. For that last purpose we

use `zip` function we met in Section 6.8.1 . Finally we recreate a dictionary using comprehension. Time for some tests.

```
allCorsPvalsAdj = adjustPvals(allCorsPvals, Mt.BenjaminiHochberg)
falsePositves = (map(t -> t[2], values(allCorsPvalsAdj)) .<= 0.05) |>
sum
falsePositves
```

0

We cannot expect a multiplicity correction to be a 100% error-proof solution. Still, it's better than doing nothing and in our case it did the trick, we got rid of false positives.

Solution to Exercise 3

Let's start by writing a function to get a correlation matrix. We could use for that Stats.cor[338]like so `Stats.cor(bogusCors)`. But since we need to add significance markers then the p-values for the correlations are indispensable. As far as I'm aware the package does not have it, then we will write a function of our own.

```
function getCorsAndPvalsMatrix(
    df::Dfs.DataFrame,
    colNames::Vector{String})::Array{<:Tuple{Float64, Float64}}

    len::Int = length(colNames)
    corsPvals::Dict{Tuple{String,String},Tuple{Float64,Float64}} =
        getAllCorsAndPvals(df, colNames)
    mCorsPvals::Array{Tuple{Float64,Float64}} = fill((0.0, 0.0), len,
len)

    for cn in eachindex(colNames) # cn - column number
        for rn in eachindex(colNames) # rn - row number
            corPval = (
                haskey(corsPvals, (colNames[rn], colNames[cn])) ?
                corsPvals[(colNames[rn], colNames[cn])] :
                get(corsPvals, (colNames[cn], colNames[rn]), (1, 1))
            )
            mCorsPvals[rn, cn] = corPval
        end
    end
```

[338]https://docs.julialang.org/en/v1/stdlib/Statistics/#Statistics.cor

```
    return mCorsPvals
end
```

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

The function `getCorsAndPvalsMatrix` uses `getAllCorsAndPvals` we developed previously (Section 7.9.2). Then we define the matrix (our result), we initialize it with the fill function[339]that takes an initial value and returns an array of a given size filled with that value (`(0.0, 0.0)`). Next, we replace the initial values in `mCorsPvals` with the correct ones by using two `for` loops. Inside them we extract a tuple (`corPval`) from the unique `corsPvals`. First, we test if a `corPval` for a given two variables (e.g. "a" and "b") is in the dictionary `corsPvals` (`haskey` etc.). If so then we insert it into the `mCorsPvals`. If not, then we search in `corsPvals` by its reverse (so, e.g. "b" and "a") with `get(corsPvals, (colNames[cn], colNames[rn]), etc.)`. If that combination is not present then we are looking for the correlation of a variable with itself (e.g. "a" and "a") which is equal to `(1, 1)` (for correlation coefficient and p-value, respectively). Once we are done we return our `mCorsPvals` matrix (aka `Array`). Time to give it a test run.

```
getCorsAndPvalsMatrix(bogusCors, ["a", "b", "c"])
```

```
3×3 Matrix{Tuple{Float64, Float64}}:
 (1.0, 1.0)             (0.194, 0.591239)      (-0.432251, 0.212195)
 (0.194, 0.591239)      (1.0, 1.0)             (-0.205942, 0.568128)
 (-0.432251, 0.212195)  (-0.205942, 0.568128)  (1.0, 1.0)
```

The numbers seem to be OK. In the future, you may consider changing the function so that the p-values are adjusted, e.g. by using `Mt.BenjaminiHochberg` correction, but here we need some statistical significance for our heatmap so we will leave it as it is.

Now, let's move to drawing a plot.

[339]https://docs.julialang.org/en/v1/base/arrays/#Base.fill

```
mCorsPvals = getCorsAndPvalsMatrix(bogusCors, letters)
cors = map(t -> t[1], mCorsPvals)
pvals = map(t -> t[2], mCorsPvals)
nRows, _ = size(cors) # same num of rows and cols in our matrix
xs = repeat(1:nRows, inner=nRows)
ys = repeat(1:nRows, outer=nRows)[end:-1:1]

fig = Cmk.Figure()
ax1 = Cmk.Axis(fig[1, 1],
               xticks=(1:1:nRows, letters[1:nRows]),
               yticks=(1:1:nRows, letters[1:nRows][end:-1:1])
)
hm = Cmk.heatmap!(ax1, xs, ys, [cors...],
                  colormap=:RdBu, colorrange=(-1, 1))
Cmk.text!(ax1, xs, ys,
          text=string.(round.([cors...], digits=2)) .*
              getMarkerForPval.([pvals...]),
          align=(:center, :center),
          color=getColorForCor.([cors...]))
Cmk.hlines!(ax1, 1.5:1:nRows, color="black", linewidth=0.25)
Cmk.vlines!(ax1, 1.5:1:nRows, color="black", linewidth=0.25)
Cmk.Colorbar(fig[:, end+1], hm)
fig
```

We begin by preparing the necessary helper variables (`mCorsPvals`, `cors`, `pvals`, `nRows`, `xs`, `ys`). The last two are the coordinates of the centers of squares on the X- and Y-axis. The `cors` will be flattened row by row using `[cors...]` syntax. For your information `repeat([1, 2], inner = 2)` returns `[1, 1, 2, 2]` and `repeat([1, 2], outer = 2)` returns `[1, 2, 1, 2]`. The `ys` vector is then reversed with `[end:-1:1]` to make it reflect better the order of correlations in `cors` (left to right, row by row). The same goes for `yticks` below. The above was determined to be the right option by trial and error. The next important parameter is `colorrange=(-1, 1)` it ensures that `-1` is always the leftmost color (red) from the `:RdBu` colormap and `1` is always the rightmost color (blue) from the colormap. Without it the colors would be set to `minimum(cors)` and `maximum(cors)` which we do not want since the `minimum` will change from matrix to matrix. Over our heatmap we overlay the grid (`hlines!` and `vlines!`) to make the squares separate better from each other. The centers of the squares are at integers, and the edges are at halves, that's why we start the ticks at

`1.5`. Finally, we add `Colorbar` as they did in the docs for `Cmk.heatmap`. The result of this code is visible in Figure 33 from the previous section.

OK, let's add the correlation coefficients and statistical significance markers. But first, two little helper functions.

```
function getColorForCor(corCoeff::Float64)::String
    @assert (0 <= abs(corCoeff) <= 1) "abc(corCoeff) must be in range
[0-1]"
    return (abs(corCoeff) >= 0.65) ? "white" : "black"
end

function getMarkerForPval(pval::Float64)::String
    @assert (0 <= pval <= 1) "probability must be in range [0-1]"
    return (pval <= 0.05) ? "#" : ""
end
```

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

As you can see `getColorForCor` returns a color ("white" or "black") for a given value of correlation coefficient (white color will make it easier to read the correlation coefficient on a dark red/blue background of a square). On the other hand `getMarkerForPval` returns a marker ("#") when a pvalue is below our customary cutoff level for type I error.

```
fig = Cmk.Figure()
ax, hm = Cmk.heatmap(fig[1, 1], xs, ys, [cors...],
    colormap=:RdBu, colorrange=(-1, 1),
    axis=(;
        xticks=(1:1:nRows, letters[1:nRows]),
        yticks=(1:1:nRows, letters[1:nRows][end:-1:1])
    ))
Cmk.text!(fig[1, 1], xs, ys,
    text=string.(round.([cors...], digits=2)) .*
        getMarkerForPval.([pvals...]),
    align=(:center, :center),
    color=getColorForCor.([cors...]))
Cmk.hlines!(fig[1, 1], 1.5:1:nRows, color="black", linewidth=0.25)
Cmk.vlines!(fig[1, 1], 1.5:1:nRows, color="black", linewidth=0.25)
Cmk.Colorbar(fig[:, end+1], hm)
fig
```

The only new element here is `Cmk.text!` function, but since we used it a couple of times throughout this book, then I will leave the

explanation of how the code piece works to you. Anyway, the result is to be found below.

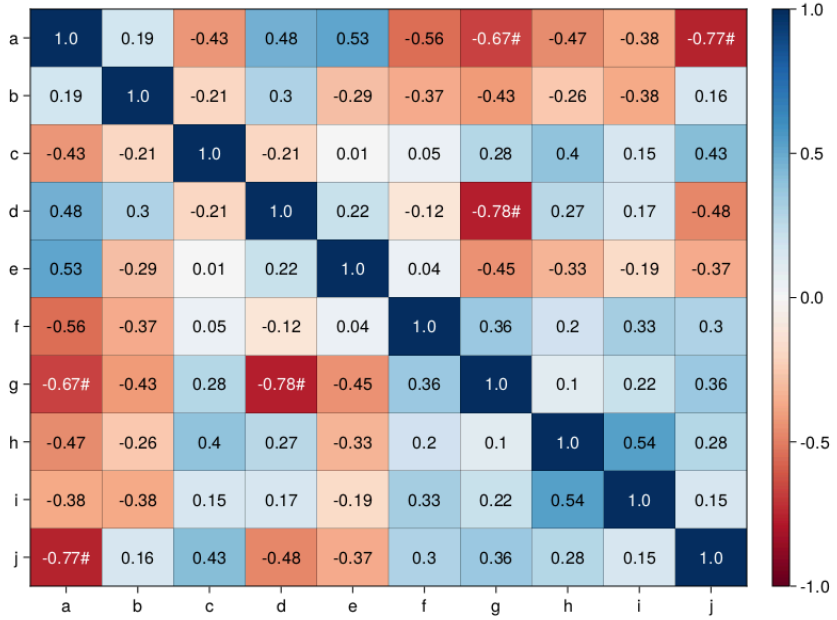

Figure 39: Figure 37: Correlation heatmap for data in `bogusCors` with the coefficients and significance markers.

It looks good. Also the number of significance markers is right. Previously (Section 7.9.2) we said we got 3 significant correlations (based on 'raw' p-values). Since, the upper right triangle of the heatmap is a mirror reflection of the lower left triangle, then we should see 6 significance markers altogether. As a final step (that I leave to you) we could enclose the code from this task into a neat function named, e.g. `drawHeatmap`.

Solution to Exercise 4

OK, the code for this task is quite straightforward so let's get right to it.

```
function drawDiagPlot(
    reg::Glm.StatsModels.TableRegressionModel,
```

```
    byCol::Bool = true)::Cmk.Figure
    dim::Vector{<:Int} = (byCol ? [1, 2] : [2, 1])
    res::Vector{<:Float64} = Glm.residuals(reg)
    pred::Vector{<:Float64} = Glm.predict(reg)
    form::String = string(Glm.formula(reg))
    fig = Cmk.Figure(size=(800, 800))
    ax1 = Cmk.Axis(fig[1, 1],
                   title="Residuals vs Fitted\n" * form,
                   xlabel="Fitted values",
                   ylabel="Residuals")
    Cmk.scatter!(ax1, pred, res)
    Cmk.hlines!(ax1, 0, linestyle=:dash, color="gray")
    ax2 = Cmk.Axis(fig[dim...],
                   title="Normal Q-Q\n" * form,
                   xlabel="Theoretical Quantiles",
                   ylabel="Standarized residuals")
    Cmk.qqplot!(ax2,
                Dsts.Normal(0, 1),
                getZScore.(res, Stats.mean(res), Stats.std(res)),
                qqline=:identity)
    return fig
end
```

We begin with extracting residuals (`res`) and predicted (`pred`) values from our model (`reg`). Additionally, we extract the formula (`form`) as a string. Then, we prepare a scatter plot (`Cmk.scatter`) with `pred` and `res` placed on X- and Y-axis, respectively. Next, we add a horizontal line (`Cmk.hlines!`) at 0 on Y-axis (the points should be randomly scattered around it). All that's left to do is to build the required Q-Q plot (`qqplot`) with X-axis that contains the theoretical standard normal distribution[340](`Dsts.Normal(0, 1)`) and Y-axis with the standardized (`getZScore`) residuals (`res`). We also add `qqline=:identity` (here, identity means x = y) to facilitate the interpretation [if two distributions (on X- and Y-axis)] are alike then the points should lie roughly on the line. Since the visual impression we get may depend on the spacial arrangement (stretching or tightening of the points on a graph) our function enables us to choose (`byCol`) between column (`true`) and row (`false`) alignment of the subplots.

[340]https://en.wikipedia.org/wiki/Normal_distribution#Standard_normal_distribution

For a change let's test our function on the `iceMod2` from Section 7.7 . Behold the result of `drawDiagPlot(iceMod2, false)`.

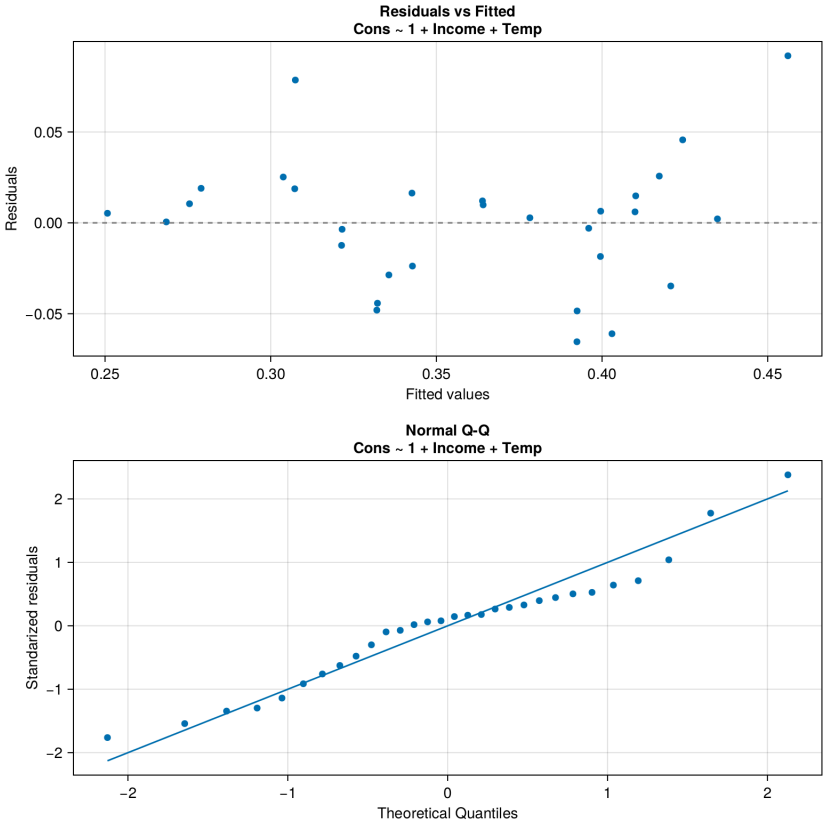

Figure 40: Figure 38: Diagnostic plot for regression model (iceMod2).

Hmm, I don't know about you but to me the bottom panel looks rather normal. However, the top panel seems to display a wave ('w') pattern. This may be a sign of auto-correlation (explanation in a moment) and translate into instability of the estimation error produced by the model across the values of the explanatory variable(s). The error will display a wave pattern (once bigger once smaller). Now we got a choice, either we leave this model as it is (and we bear the consequences) or we try to find a better one.

To understand what the auto-correlation means in our case let's do a thought experiment. Right now in the room that I am sitting the temperature is equal to 20 degrees of Celsius (68 deg. Fahrenheit). Which one is the more probable value of the temperature in 1 minute from now: 0 deg. Cels. (32 deg. Fahr.) or 21 deg. Cels. (70 deg. Fahr.)? I guess the latter is the more reasonable option. That is because the temperature one minute from now is a derivative of the temperature in the present (i.e. both values are correlated).

The same might be true for Icecream[341] data frame, since it contains `Temp` column that we used in our model (`iceMod2`). We could try to remedy this by removing (kind of) the auto-correlation, e.g. with `ice2 = ice[2:end, :]` and `ice2.TempDiff = ice.Temp[2:end] .- ice.Temp[1:(end-1)]` and building our model a new. This is what we will do in the next exercise (although we will try to automate the process a bit).

Solution to Exercise 5

Let's start with a few helper functions.

```
function getLinMod(
    df::Dfs.DataFrame,
    y::String, xs::Vector{<:String}
    )::Glm.StatsModels.TableRegressionModel
    return Glm.lm(Glm.term(y) ~ sum(Glm.term.(xs)), df)
end

function getPredictorsPvals(
    m::Glm.StatsModels.TableRegressionModel)::Vector{<:Float64}
    allPvals::Vector{<:Float64} = Glm.coeftable(m).cols[4]
    # 1st pvalue is for the intercept
    return allPvals[2:end]
end

function getIndsEltsNotEqlM(v::Vector{<:Real}, m::Real)::Vector{<:Int}
    return findall(x -> !isapprox(x, m), v)
end
```

We begin with `getLinMod` that accepts a data frame (`df`), name of the dependent variable (`y`) and names of the independent/predictor

[341] https://vincentarelbundock.github.io/Rdatasets/doc/Ecdat/Icecream.html

variables (`xs`). Based on the inputs it creates the model programmatically using `Glm.term`.

Next, we go with `getPredictorsPvals` that returns the p-values corresponding to a model's coefficients.

Then, we define `getIndsEltsNotEqlM` that we will use to filter out the highest p-value from our model.

OK, time for the main actor of the show.

```
# returns minimal adequate (linear) model
function getMinAdeqMod(
    df::Dfs.DataFrame, y::String, xs::Vector{<:String}
    )::Glm.StatsModels.TableRegressionModel

    preds::Vector{<:String} = copy(xs)
    mod::Glm.StatsModels.TableRegressionModel = getLinMod(df, y, preds)
    pvals::Vector{<:Float64} = getPredictorsPvals(mod)
    maxPval::Float64 = maximum(pvals)
    inds::Vector{<:Int} = getIndsEltsNotEqlM(pvals, maxPval)

    for _ in xs
        if (maxPval <= 0.05)
            break
        end
        if (length(preds) == 1 && maxPval > 0.05)
            mod = Glm.lm(Glm.term(y) ~ Glm.term(1), df)
            break
        end
        preds = preds[inds]
        mod = getLinMod(df, y, preds)
        pvals = getPredictorsPvals(mod)
        maxPval = maximum(pvals)
        inds = getIndsEltsNotEqlM(pvals, maxPval)
    end

    return mod
end
```

We begin with defining the necessary variables that we will update in a for loop. The variables are: predictors (`preds`), linear model (`mod`), p-values for the model's coefficients (`pvals`), maximum p-value (`maxPval`) and indices of predictors that we will leave in our model (`inds`). We start each iteration (`for _ in xs`) by checking if we already reached our minimal adequate model. To that end we make

sure that all the remaining coefficients are statistically significant (`if (maxPval <= 0.05)`) or if we run out of the explanatory variables (`length(preds) == 1 && maxPval > 0.05`) we return our default (`y ~ 1`) model (the intercept of this model is equal to `Stats.mean(y)`). If not then we remove one predictor variable from the model (`preds = preds[inds]`) and update the remaining helper variables (`mod`, `pvals`, `maxPval`, `inds`). And that's it, let's see how it works.

```
ice2mod = getMinAdeqMod(ice2, names(ice2)[1], names(ice2)[2:end])
ice2mod
```

```
──────────────────────────────────────────────────────────────────────
               Coef.  Std. Error      t  Pr(>|t|)  Lower 95%  Upper 95%
──────────────────────────────────────────────────────────────────────
(Intercept)  -0.0672      0.0988  -0.68    0.5024    -0.2707     0.1363
Income        0.0031      0.001    2.99    0.0062     0.001      0.0053
Temp          0.0032      0.0004   7.99    <1e-99     0.0024     0.004
TempDiff      0.0022      0.0007   2.93    0.0071     0.0006     0.0037
──────────────────────────────────────────────────────────────────────
```

It appears to work as expected. Let's compare it with a full model.

```
ice2FullMod = getLinMod(ice2, names(ice2)[1], names(ice2)[2:end])

Glm.ftest(ice2FullMod.model, ice2mod.model)
```

```
F-test: 2 models fitted on 29 observations
──────────────────────────────────────────────────────────────────
     DOF  ΔDOF     SSR    ΔSSR      R²      ΔR²      F*   p(>F)
──────────────────────────────────────────────────────────────────
[1]   10        0.0193          0.8450
[2]    5    -5  0.0227  0.0034  0.8179  -0.0272  0.7019  0.6285
──────────────────────────────────────────────────────────────────
```

It looks good as well. We reduced the number of explanatory variables while maintaining comparable ($p > 0.05$) explanatory power of our model.

Time to check the assumptions with our diagnostic plot (`drawDiagPlot` from Section 7.9.1).

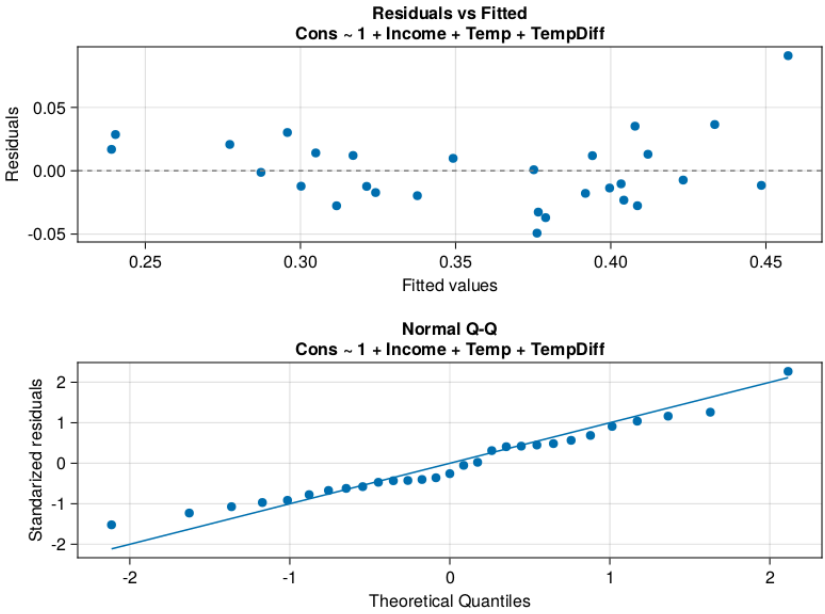

Figure 41: Figure 39: Diagnostic plot for regression model (ice2mod).

To me, the plot has slightly improved and since I run out of ideas how to make our model even better I'll leave it as it is.

Now, let's compare our `ice2mod`, that aimed to counteract the auto-correlation, with its predecessor (`iceMod2`). We will focus on the explanatory powers (adjusted r^2, the higher the better)

```
(
    Glm.adjr2(iceMod2),
    Glm.adjr2(ice2mod)
)
```

```
(0.6799892012945553, 0.796000295561351)
```

and the average prediction errors (the lower the better).

```
(
    getAvgEstimError(iceMod2),
```

```
    getAvgEstimError(ice2mod)
)
```

```
(0.026114993652645798, 0.022116071809225545)
```

Again, it appears that we managed to improve our model's prediction power at a cost of slightly more difficult interpretation (go ahead examine the output tables for `Income + Temp + TempDiff` vs. `Income + Temp` and explain to yourself how each variable influences the value of `Cons`). This is usually the case, the less straightforward the model, the less intuitive is its interpretation.

At a very long last we may check how our `getMinAdeqMod` will behave when there are no meaningful explanatory variables.

```
getMinAdeqMod(ice2, "Cons", ["a", "b", "c", "d"])
```

```
Cons ~ 1

Coefficients:
──────────────────────────────────────────────────────────────────────────
                Coef.  Std. Error      t  Pr(>|t|)  Lower 95%  Upper 95%
──────────────────────────────────────────────────────────────────────────
(Intercept)  0.358517    0.012397  28.92    <1e-21   0.333123   0.383911
──────────────────────────────────────────────────────────────────────────
```

In that case (no meaningful explanatory variables) our best estimate (guess) of the value of `y` (here `Cons`) is the variable's average (`Stats.mean(ice2.Cons)`) which is returned as the `Coef.` for `(Intercept)`. In that case `Std. Error` is just the standard error of the mean that we met in Section 5.2 (compare with `getSem(ice2.Cons)`).

Overall, our `getMinAdeqMod` should work reasonably well for a small number of explanatory variables (the `xs` argument). When the number of predictors grows, some of them are likely to be significant by chance alone (compare with the discussion in Section 5.6).

Anyway, building our minimal adequate model from top to bottom (as we did here) is not the only possible procedure. Equally reasonable is

to apply the bottom to top approach. In that case we start with separate models with 1 explanatory variable each. Of those models we choose the one with the lowest p-value. Next we add to it one explanatory variable at a time (based on the p-value, the lower the better) until we reach our final model (no more significant explanatory variables left). Sadly, the two methods although equally sound do not always produce the same result (the same minimal adequate model). Unfortunately, as far as I'm aware there is not much to be done with it, so we must live with that fact.

Time to say goodbye

They say that all that has its beginning must have its end. So I guess it's time to ..., OK, but before we part let me give you a word of advice.

Julia is a nice programming language with many applications, including statistics (probably way beyond the level covered in this book). Still, if you are new to (Julia) programming and statistics then most likely you should calibrate your tools first. Before you run some statistical analysis you may want to try it out on an example from a textbook written by an expert (not me though) and see if you get the same (or at least comparable) result on your own. Although this is a sound approach, I suspect you are more prone to visit some statistical blog or internet forum and go with the examples that are contained there. One such option is rseek.org[342], i.e. a search engine for the R programming language[343]. In that case RCall.jl[344]will be of assistance.

For instance let's say that I copied the `beerVolumes` example (see Section 5.2) from some R forum (I didn't). Now, without leaving Julia I can paste and execute the R's code (R's code goes between the quotation marks in `RC.R""`).

```
import RCall as RC

RC.R"
beerVolumes <- c(504, 477, 484, 476, 519, 481, 453, 485, 487, 501)
t.test(beerVolumes, mu=500)
"
```

> ***Note:*** For that code to work you need to have the R programming language installed on your machine.

```
        One Sample t-test
```

[342]https://rseek.org/

[343]https://en.wikipedia.org/wiki/R_(programming_language)

[344]https://github.com/JuliaInterop/RCall.jl

```
data:  beerVolumes
t = -2.3294, df = 9, p-value = 0.04479
alternative hypothesis: true mean is not equal to 500
95 percent confidence interval:
 473.7837 499.6163
sample estimates:
mean of x
    486.7
```

Then, I can compare it with the output of `Ht.OneSampleTTest`. That way I can validate it and see if it is a credible Julia's equivalent of R's `t.test`. The above, is also the way to test my understanding of Julia's function that stems from the docs[345].

```
import HypothesisTests as Ht

beerVolumes = [504, 477, 484, 476, 519, 481, 453, 485, 487, 501]
Ht.OneSampleTTest(beerVolumes, 500)
```

```
One sample t-test
-----------------
Population details:
    parameter of interest:   Mean
    value under h_0:         500
    point estimate:          486.7
    95% confidence interval: (473.8, 499.6)

Test summary:
    outcome with 95% confidence: reject h_0
    two-sided p-value:           0.0448

Details:
    number of observations:   10
    t-statistic:              -2.329353706113303
    degrees of freedom:       9
    empirical standard error: 5.70973826993069
```

Once I got both outputs that are similar enough I can be fairly sure I did right. Otherwise I should investigate where the differences come from and possibly make some necessary adjustments.

[345] https://juliastats.org/HypothesisTests.jl/stable/parametric/#t-test

Now, let me follow a word of advice with a word of warning. The book contains a description of statistics the way I see it, not necessarily the way it really is. Additionally, many times I simplified stuff, e.g. by avoiding mathematics and mathematical formulas that go beyond the level of a primary school (in Poland grades 1-8). Moreover, I also tried to limit the number of Julia's constructs in the examples. In the end I wrote that book for myself from the past, so if you ever met me then be sure to pass it on me. I would have loved to read it. But then again, back in the day when I was a student there was no Julia, and my English was too poor. Oh, well, just enjoy the book yourself.

Take care.

Bartłomiej Łukaszuk - author