HBIGS core course: An Introduction to R Programming

Florian Huber

May 28, 2024

Contents

R basics **2**
Values and variables . . . 2
Doing stuff: functions . . . 3
Vectors and vectorization are fundamental concepts in R . . . 3
More about functions . . . 5
R's data types . . . 6
Vector comparisons . . . 7
Coercion . . . 7
Special types . . . 8
Vectors can be named . . . 9
Where is R running? . . . 9
Some terminology . . . 10

Subsetting and modifying values from vectors **10**
Extracting values from vectors: subsetting . . . 10
Exercises . . . 13
Modifying values in vectors . . . 13
Exercises . . . 14

Lists **14**
Initializing lists . . . 14
Subsetting lists . . . 15
Replacing list elements, extending lists . . . 16
Exercise . . . 17

Data frames **18**
Initializing data frames . . . 18
Subsetting data frames . . . 18
Using logical subsetting to "ask questions" . . . 20
Exercise . . . 20

Functions **21**
Calling functions . . . 21
Running examples from the documentation . . . 21
The "three dots" (ellipsis) . . . 21
Writing your own functions . . . 21
Visibility of variables ('scope') . . . 22
Exercise . . . 22

Control Structures **22**
if - else . . . 22

for loops 23

Importing data and getting an overview **24**
General remarks 24
Importing the `cells` data set 24
Exercise 25

Plotting **25**
Basic functionality: scatter plots 26
Saving ggplots 30
Aesthetic mappings 30
Adding model fits to ggplot2 32
Exercise 35
Displaying quantities and distributions 35
Exercises 46
The time series challenge, aka tidy data aka my data is in a bad shape 46
Exercise 50

The tidyverse **50**
The dplyr package offers an alternative interface for subsetting data 51
Exercise 57

Avoiding repetition: the `map()` function **57**
Exercise 59

A bit of statistics **59**
Linear regression and testing correlations 59
Principles of hypothesis testing: the t-test 64
Confidence intervals 68

License **69**

R basics

Values and variables

R can always be used as a pocket calculator.

```
> 1 + 2 - 4
[1] -1
```

```
> 6 * 3
[1] 18
```

```
> 2 / 3
[1] 0.6666667
```

```
> 3 ^ 3
[1] 27
```

We can save the results of our calculations by storing the values in **variables**. To do this, we have to pick a variable name and then use an **assignment operator** `<-` or `=` to save the values in the **variables**.

Sometimes in this document you will see lines starting with an `#`. The `#` sign is used to put comments into your code. Everything after the `#` is not executed.

```
> # comments are not executed
> # use comments to explain your code
> # <- is used for variable assignment
> a <- 5 # leave spaces around operators
> a
[1] 5
```

```
> b <- 3
> a ^ b
[1] 125
```

```
> x <- a ^ b
> x
[1] 125
```

```
> # When you try to use a variable that does not exist you get an error:
> my_variable
Error in eval(expr, envir, enclos): object 'my_variable' not found
```

Doing stuff: functions

If we want to "do something" with the values that we have, we can use either **operators** such `+`, `-`, `*` etc. or **functions**. Functions are very important in R. They perform actions on values. We say that we **call** a function and usually we pass **arguments** to a function. You can recognise function calls by their parentheses: `func()` is a function call whereas `func` without the `()` would be the variable `func`. Upon **evaluation** functions typically **return** a value. The values returned are said to live in our **workspace** which is also called the **global environment**. You can see which variables are in your workspace by using the function `ls()`.

```
> sqrt(9)
[1] 3
```

```
>
> a <- 3.14159
> round(a)
[1] 3
```

```
>
> ls()
[1] "a" "b" "x"
```

Vectors and vectorization are fundamental concepts in R

- Data in R is stored in **vectors** = sequences of values.
- The function `c()` combines values into a vector. You can pass as many values to `c()` as you like: `c(3, 4, x, y)`. Technically speaking, the values passed to `c()` are `function arguments` that are separated by commas.
- If an expression returns a vector by itself, you can omit the `c()`

```
> a <- c(1, 2, 5, 4)
> a
[1] 1 2 5 4
```

```
> # the following operator creates a sequence of values
> # since its result is a vector we don't need to wrap it in c()
> 1:3
[1] 1 2 3
```

```
> # same as:
> c(1:3)
[1] 1 2 3
```

```
> # but you can't write:
> # 1, 2, 3
>
> # Even single values are vectors:
> is.vector(1)
[1] TRUE
```

```
>
> # c() can itself contain vectors
> b <- c(2, 2, 2, 2)
> v <- c(a, b)
```

The number of elements in a vector determines its **length**, which you can determine with the function `length()`.

```
> v
[1] 1 2 5 4 2 2 2 2
```

```
> length(v)
[1] 8
```

Not only data but also many operators and functions in R are **vectorized**.

Example: arithmetic using vectors.

```
> a
[1] 1 2 5 4
```

```
> b
[1] 2 2 2 2
```

```
> a + b
[1] 3 4 7 6
```

```
> a - b
[1] -1  0  3  2
```

```
> a * b
[1]  2  4 10  8
```

```
> a / b
[1] 0.5 1.0 2.5 2.0
```

```
> a ^ b
[1]  1  4 25 16
```

Vectors are **recycled**, if necessary.

```
> a
[1] 1 2 5 4
```

```
> c <- c(3, 1)
> a + c
[1] 4 3 8 5
```

```
> d <- c(3, 7, 10)
> a + d
```

```
Warning in a + d: longer object length is not a multiple of shorter object
length
[1]  4  9 15  7
```

```
> # but be careful: you are not always warned
```

Exercises

Do exercises 1 and 2 from the "vectors and data types" section.

More about functions

Almost everything in R is done via functions. For example, you can sum all vector elements up with `sum()` or calculate the mean of a vector with `mean()`. Functions are so central to R that it is called a **functional programming language**.

```
> v <- c(1, 2)
> sum(v)
[1] 3
```

```
> mean(v)
[1] 1.5
```

But how do we know what a function does and how to use it? For this, it is helpful to read the **documentation**. You can open the documentation using `?`. In fact, a lot of time during a programmer's work day is spent on reading the documentation. In this course we will learn all the concepts to understand the documentation.

```
> ?round
> round(a, digits = 2)
[1] 1 2 5 4
```

Function arguments and default values

When reading the documentation for `round()` we see that it takes actually two arguments. However, the `digits` arguments is set to a **default value**. When calling a function you can choose to name the arguments. If you don't specify which value refers to which argument, argument matching will happen by position. Sometimes you see `...` listed as an argument. It usually means that you can pass an arbitrary number of arguments. `sum()` for example can sum up as many numbers as you like.

```
> # argument order and naming
> round(3.14159, 2)
[1] 3.14
```

```
> round(2, 3.14159)
[1] 2
```

```
> round(2, x = 3.14159)
[1] 3.14
```

Other useful functions are `rep()` and `seq()`. We will encounter a lot more functions during the course. The base R cheat sheet also lists some useful functions.

```
> # ?rep
> rep(v, times = 3)
[1] 1 2 1 2 1 2
```

```
> # ?seq
> seq(from = 1, to = 2.5, by = 0.5)
[1] 1.0 1.5 2.0 2.5
```

R's data types

So far we have worked only with numbers, the so-called "numeric" data type. However, if we want to work on e.g. text we need other data types.

- Whenever we store data in R, it is of a particular **type** (or **mode**).
- The most common types are:
  - Type "numeric": what we have used so far. Can be double or integer.
  - Type "character", also called "string": any type of text. This data type is recognized by the surrounding quotes: `"`.
  - Type "logical" to represent `TRUE` = `T` and `FALSE` = `F` values. When you type `TRUE` into the R console R will recognize it as a **keyword** and know that you are talking about the logical data type.
- The vectors we have learned about so far can be **of only one type**.
- You can query the type of a vector with `mode()` or `typeof()`.

```
> a <- c(1, 3, 5)
> mode(a)
[1] "numeric"
```

```
> str(a)
 num [1:3] 1 3 5
```

```
> is.numeric(a) # result is a logical vector of length 1
[1] TRUE
```

```
>
> b <- c("p53", "NICN1", "INADL")
> b
[1] "p53"   "NICN1" "INADL"
```

```
> mode(b)
[1] "character"
```

```
> str(b)
 chr [1:3] "p53" "NICN1" "INADL"
```

```
> is.character(b)
[1] TRUE
```

```
> is.numeric(b)
[1] FALSE
```

```
>
> true_or_false <- c(TRUE, FALSE, FALSE)
> str(true_or_false)
 logi [1:3] TRUE FALSE FALSE
```

```
> # types are important: functions need certain input types
> # compare sum(a) and sum(b)
```

Note that there is a difference between `a` and `"a"` (= `'a'`). Without quotes R will usually look for a variable. With quotes R treats everything between the quotes as a "string" (or "character"), a term used in programming to mean words. Both single and double quotes can be used to declare strings.

```
> variable <- 5
> variable
[1] 5
```

```
> "variable"
[1] "variable"
```

```
> notfound
Error in eval(expr, envir, enclos): object 'notfound' not found
```

```
> "notfound"
[1] "notfound"
```

```
>
> # to delete a variable use rm()
> rm(variable)
> variable
Error in eval(expr, envir, enclos): object 'variable' not found
```

Vector comparisons

Logical vectors are useful to compare vectors. There are a number of **comparison operators**: ==, >, <, <=, and >=.

```
> a <- c(1, 2, 5, 4)
> b <- c(2, 2, 2, 2)
> a > 3
[1] FALSE FALSE  TRUE  TRUE
```

```
> a == 5
[1] FALSE FALSE  TRUE FALSE
```

```
> a == b
[1] FALSE  TRUE FALSE FALSE
```

```
> a != b
[1]  TRUE FALSE  TRUE  TRUE
```

```
> a > b
[1] FALSE FALSE  TRUE  TRUE
```

```
> a < b
[1]  TRUE FALSE FALSE FALSE
```

```
> a >= b
[1] FALSE  TRUE  TRUE  TRUE
```

```
> a <= b
[1]  TRUE  TRUE FALSE FALSE
```

Coercion

Forcing a type change is called **coercion**. Unwanted coercion can lead to errors but sometimes coercion is also useful.

```
> cool_genes <- c(IME4 = TRUE, PDC1 = FALSE, ADH1 = TRUE)
> cool_genes
 IME4  PDC1  ADH1
 TRUE FALSE  TRUE
```

```
> sum(cool_genes) # implicit coercion: FALSE is 0, TRUE is 1
[1] 2
```

```
> # fraction of cool genes
> sum(cool_genes) / length(cool_genes)
[1] 0.6666667
```

```
> # more coercion examples
> -1 < T
[1] TRUE
```

```
> as.numeric(cool_genes) # manual coercion
[1] 1 0 1
```

```
> as.character(cool_genes)
[1] "TRUE"  "FALSE" "TRUE"
```

```
> as.numeric(c("apples", "oranges")) # not all coercions make sense
Warning: NAs introduced by coercion
[1] NA NA
```

Special types

There are a few special data types:

- `NA` = 'not available' denotes missing or unknown values
- `NaN` = 'not a number' is similar to `NA` but for arithmetic purposes
- `Inf` and `-Inf`
- `NULL` is a special type for 'empty objects'

The most important one of these special types is `NA`. It can sometimes lead to "unexpected" behaviour.

```
> a <- c(1, 3, 5)
> mean(a)
[1] 3
```

```
>
> b <- c(1, 3, 5, NA)
> mean(b)
[1] NA
```

```
> ?mean
> mean(b, na.rm = TRUE)
[1] 3
```

```
>
> is.na(b)
[1] FALSE FALSE FALSE  TRUE
```

```
>
> # shocking! it is possible! :D
> 1 / 0 # IEEE specification!
[1] Inf
```

```
>
> log10(-5)
Warning: NaNs produced
[1] NaN
```

Vectors can be named

In R, the elements of a vector can be named. Write names to the left of the value, either quoted or unquoted. Use `names()` to set or retrieve names. Use `unname()` to remove names.

```
> # NB: quoting names is optional
> sizes <- c("Luke" = 180, "Leia" = 170, "Chewbacca" = 250)
> str(sizes)
 Named num [1:3] 180 170 250
 - attr(*, "names")= chr [1:3] "Luke" "Leia" "Chewbacca"
```

```
> names(sizes)
[1] "Luke"      "Leia"      "Chewbacca"
```

```
> names(sizes) <- c("Lu", "Le", "Ch")
> sizes
 Lu  Le  Ch 
180 170 250 
```

```
> unname(sizes)
[1] 180 170 250
```

```
> sizes # forgot to assign!
 Lu  Le  Ch 
180 170 250 
```

```
> sizes <- unname(sizes)
> sizes
[1] 180 170 250
```

Where is R running?

```
> # Where are we? note the '()' - always needed to execute functions
> # getwd()
> 
> # './' denotes the current directory: this is a relative path
> setwd("./data/")
> 
> # getwd()
> 
> # to see the files in your working directory
> dir()
[1] "ImagedCells.csv"         "Melanoma.csv"           
[3] "mtec_counts.RData"       "plate_reader"           
[5] "vectors_exercise4.RData"
```

```
> 
> # ../ is an abbreviation for the parent directory
> # so with this command we 'go back'
> setwd("../")
```

```
> # If you are not sure where a file is located use `file.choose()`. It will open
> # a pop-up dialogue and then print the path of the file in the console.
> file.choose()
```

Exercises

Do exercises 3 to 5 from the "vectors and data types" section.

Some terminology

- R is an **interpreted language**.
- `>` is called the **prompt**.
- Sometimes you see a `+` at the beginning of the line. This means that R is expecting additional input.
- Lines beginning with `#` are **comments**.
- Variables are assigned using `<-` or `=`.
- Scripts are executed in a **working directory**.
- All variables you load or create, data you import etc. live in the **workspace**.
- If you are stuck in the console press `Esc` or `Ctrl + c` to stop the current command.

Subsetting and modifying values from vectors

Extracting values from vectors: subsetting

Subsetting by index

- We can extract and modify values of vectors by **subsetting**.
- There are 3 major ways to subset vectors. The first one is **subsetting by index**. Each element of a vector has an *index* which indicates where the element is located. 1 refers to the first element, 2 to the second element etc.
- Subsetting is done using the **subsetting operator:** [].

Study the following examples carefully:

```
> a <- c(4.1, 3.2, 8.3, 19.4) # digit after the comma is the index
> a
[1]  4.1  3.2  8.3 19.4
```

```
> a[1] # first index is 1; can also write a[c(1)]
[1] 4.1
```

```
> a[3]
[1] 8.3
```

```
> a[c(1, 3)]
[1] 4.1 8.3
```

```
> a[c(3, 1)]
[1] 8.3 4.1
```

```
> a[c(1, 1, 3, 3)]
[1] 4.1 4.1 8.3 8.3
```

```
> a[2:4] # same as a[c(2, 3, 4)] or a[2:4]
[1]  3.2  8.3 19.4
```

```
> length(a)
[1] 4
```

```
> a[2:length(a)]
[1]  3.2  8.3 19.4
```

```
> a_subset <- a[c(2, length(a))]
> a_subset
```

```
[1]  3.2 19.4
```

Negative numerical indices are used to exclude values.

```
> a
[1]  4.1  3.2  8.3 19.4
```

```
> # negative indices exclude values
> a[-c(2, 4)]
[1] 4.1 8.3
```

```
> a <- c(a, c(1000, 55))
> a
[1]    4.1    3.2    8.3   19.4 1000.0   55.0
```

```
> max(a)
[1] 1000
```

```
> which.max(a)
[1] 5
```

```
> a[which.max(a)]
[1] 1000
```

```
> a[-which.max(a)]
[1]  4.1  3.2  8.3 19.4 55.0
```

```
> # careful with NA values
> a[NA]
[1] NA NA NA NA NA NA
```

Subsetting by name

Subsetting by name works similar to subsetting by index. The only difference is that - does not work when subsetting by name.

```
> grades <- c("john" = 2, "jane" = 3, "brenda" = 1, "andy" = 5)
> grades[c("john")]
john
   2
```

```
> grades[c("john", "brenda")]
  john brenda
     2      1
```

```
> grades[c("john", "john", "brenda")]
  john   john brenda
     2      2      1
```

```
> # this doesn't work:
> grades[-c("andy")]
Error in -c("andy"): invalid argument to unary operator
```

```
> names(grades)
[1] "john"   "jane"   "brenda" "andy"
```

Subsetting with logical vectors

When subsetting with a logical vector, all the `TRUE` elements are returned whereas all the `FALSE` elements are dropped. This is very useful because we can use this to extract all elements that are greater than a given value.

```
> a <- c(4.1, 3.2, 8.3, 19.4)
> a[c(TRUE, TRUE, FALSE, TRUE)]
[1]  4.1  3.2 19.4
```

```
> # subsetting vector must be the same length as the subsetted vector
> # otherwise vector recycling takes place
> a[c(TRUE, FALSE)] # vector recycling
[1] 4.1 8.3
```

```
> b <- seq(from = 1, to = 3, length.out = 6) # a very useful function
> b
[1] 1.0 1.4 1.8 2.2 2.6 3.0
```

```
> b > 2
[1] FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

```
> b[b > 2]
[1] 2.2 2.6 3.0
```

```
> a > b
Warning in a > b: longer object length is not a multiple of shorter object
length
[1] TRUE TRUE TRUE TRUE TRUE TRUE
```

```
> a[a > b]
Warning in a > b: longer object length is not a multiple of shorter object
length
[1]  4.1  3.2  8.3 19.4   NA   NA
```

Subsetting with logical expressions is very important. We can do complicated comparisons and define conditions so that we only retain certain values. For example, we may want to keep only those gene expression values where the fold-change is greater than 2x *and* the p-value is smaller than 0.01. Or we have two datasets and we only want to keep the genes that are in dataset A *or* in dataset B. To achieve this, we need so-called **logical operators**:

- `&` = logical AND
- `|` = logical OR
- `!` = logical NOT

A so-called **"truth table"** (0 = FALSE and 1 = TRUE):

a	b	a AND b	a OR b	NOT a
0	0	0	0	1
1	0	0	1	0
0	1	0	1	1
1	1	1	1	0

```
> foldchanges <- c('IME4' = 3, 'ACT1' = 1.5, 'HOG1' = 2.5, 'AVL9' = 1.1)
> pvalues <- c(0.005, 0.002, 0.2, 0.003)
>
> thresh_foldchg <- 2
> thresh_pval <- 0.01
>
> foldchanges > thresh_foldchg
 IME4  ACT1  HOG1  AVL9
 TRUE FALSE  TRUE FALSE
```

```
> foldchanges[foldchanges > thresh_foldchg]
IME4 HOG1
 3.0  2.5
```

```
>
> pvalues < thresh_pval
[1]  TRUE  TRUE FALSE  TRUE
```

```
> foldchanges[pvalues < thresh_pval]
IME4 ACT1 AVL9
 3.0  1.5  1.1
```

```
>
> foldchanges > thresh_foldchg & pvalues < thresh_pval
 IME4  ACT1  HOG1  AVL9
 TRUE FALSE FALSE FALSE
```

```
> foldchanges[foldchanges > thresh_foldchg & pvalues < thresh_pval]
IME4
   3
```

When doing logical operations with strings, `%in%` is very useful.

```
> genenames <- names(foldchanges)
>
> genenames %in% c('ACT1', 'IME4')
[1]  TRUE  TRUE FALSE FALSE
```

```
>
> # why did this happen?
> genenames == c('ACT1', 'IME4')
[1] FALSE FALSE FALSE FALSE
```

```
>
> genenames[genenames %in% c('ACT1', 'IME4')]
[1] "IME4" "ACT1"
```

```
> genenames[!genenames %in% c('ACT1', 'IME4')]
[1] "HOG1" "AVL9"
```

Exercises

Do exercises 1 and 2 from the section "Vector subsetting".

Modifying values in vectors

Values in vectors are always modified using a 2-step procedure:

1. Subset the value(s) that you want to replace.
2. Assign new values to the subsetted values.

The subsetted side = left-hand side and the replacement side = right-hand side have to match.

All other rules we have learned about subsetting apply: it can be done using indices, names, or logical vectors.

If you are not sure how to replace a value in a vector try to break down the procedure into these two steps.

Study the following examples carefully:

```
> foldchanges <- c('IME4' = 3, 'ACT1' = 1.5, 'HOG1' = 2.5, 'AVL9' = 1.1)
>
> (foldchanges[3] <- 3.5)
[1] 3.5
```

```
> (foldchanges[c(3, 1)] <- c(2.5, 2.8))
[1] 2.5 2.8
```

```
> (foldchanges['AVL9'] <- 1.0)
[1] 1
```

```
>
> pvalues <- c(0.005, 0.002, 0.2, 0.003)
>
> (foldchanges[pvalues > 0.01] <- NA)
[1] NA
```

```
> (foldchanges[is.na(foldchanges)] <- 0)
[1] 0
```

Exercises

Do exercises 3 and 4 from the section "Vector subsetting".

Lists

Vectors are the basic data structure in R. However, they are not very flexible because all elements of a vector have to be of the same type. They are also called "atomic vectors" because their elements cannot consist of more than one part.

How can we put different types of data into one data structure? One way to do this is by using lists. Lists are the basis of many other important data structures in R.

Dimensionality	Homogeneous	Heterogeneous
1d	Atomic vector	**List**
2d	Matrix	Data frame
nd	Array	

Lists can be thought of as vectors that can hold elements of any type - even other lists.

Initializing lists

Lists are constructed using `list()` instead of `c()`.

```
> my_first_list <-
+   list(
+     1:5,
+     c(TRUE, FALSE),
+     c('hello', 'world')
+   )
>
> my_first_list
```

```
[[1]]
[1] 1 2 3 4 5

[[2]]
[1]  TRUE FALSE

[[3]]
[1] "hello" "world"
```

```
> str(my_first_list)
List of 3
 $ : int [1:5] 1 2 3 4 5
 $ : logi [1:2] TRUE FALSE
 $ : chr [1:2] "hello" "world"
```

```
> # run in your console, can't display in this document
> # View(my_first_list)
> length(my_first_list)
[1] 3
```

Just like vectors, list elements can have names:

```
> # let's make a named list
> my_named_list <-
+     list(
+       "first" = "apples", # instead of 'first', can use any other name
+       "second" = c(3, 4, 5),
+       "third" = T
+     )
>
> str(my_named_list)
List of 3
 $ first : chr "apples"
 $ second: num [1:3] 3 4 5
 $ third : logi TRUE
```

Subsetting lists

- List elements can be accessed with `[]` or `[[]]`.
- `[]` will return a list.
- `[[]]` will return the actual element.
- All the other rules for basic vectors apply: we can subset by (i) index, (ii) name, (iii) logical.

```
> my_named_list
$first
[1] "apples"

$second
[1] 3 4 5

$third
[1] TRUE
```

```
> my_named_list[["first"]] # returns the actual the element
[1] "apples"
```

```
> my_named_list["first"] # returns the same element but wrapped within a list
$first
[1] "apples"
```

```
>
> my_named_list[c(2, 3)]
$second
[1] 3 4 5

$third
[1] TRUE
```

- `$` is a convenient shorthand for `[[]]` when extracting a named element. Instead of subsetting with e.g. `l[["first"]]` we can write `l$first`.
- Difference: When using `[[]]` you need to quote the name: `l[["first"]]`. When using `$` whatever comes after the `$` is automatically quoted: `l$first`.
- This means that if there is a variable that contains a string then you have to use `[[]]`.

```
> my_named_list[["first"]]
[1] "apples"
```

```
> my_named_list$first
[1] "apples"
```

```
> # doesn't work: we don't have a variable called first
> my_named_list[[first]]
Error in eval(expr, envir, enclos): object 'first' not found
```

```
> my_variable <- "first"
> my_named_list[[my_variable]]
[1] "apples"
```

Replacing list elements, extending lists

- To extend a list, assign to an undefined list element using `[[]]` or `$`.
- You can also use the `[[]]` and `$` operators to delete list elements by assigning `NULL` to them.

```
> my_named_list
$first
[1] "apples"

$second
[1] 3 4 5

$third
[1] TRUE
```

```
> my_named_list$another_elmt <- c(TRUE, TRUE) # you could also use [[]]
> my_named_list$first <- NULL
> my_named_list
$second
[1] 3 4 5

$third
[1] TRUE

```

```
$another_elmt
[1] TRUE TRUE
```

- You can also replace elements using `[]` but there is a subtle difference: `[]` will access and replace sublists.

```
> my_named_list
$second
[1] 3 4 5

$third
[1] TRUE

$another_elmt
[1] TRUE TRUE
```

```
> # the following replaces both the second and third list element: so you need to
> # pass a list on the right-hand side! If you don't, R will try to coerce the
> # right-hand side using as.list()
> my_named_list[c(2, 3)] <- list(c('x', 'y', 'z'), c(88, 99))
> my_named_list
$second
[1] 3 4 5

$third
[1] "x" "y" "z"

$another_elmt
[1] 88 99
```

```
>
> # careful!
> my_named_list[c(2, 3)] <- c('oh', 'no!')
> my_named_list
$second
[1] 3 4 5

$third
[1] "oh"

$another_elmt
[1] "no!"
```

```
> as.list(c('oh', 'no'))
[[1]]
[1] "oh"

[[2]]
[1] "no"
```

Exercise

Do exercise 1 from section “Lists”.

Data frames

- Data frames are the equivalent of 'tables' in R. They are 2-dimensional, i.e. they have rows and columns.
- Technically, a data frame is a list of vectors of equal lengths. You can think of each column as a `list element`. Knowing this makes it easier to understand the subsetting behaviour of data frames.

Initializing data frames

- Often, you won't need to create data frames from scratch: when you import data, you will get a data frame automatically. We will cover this a bit later.
- Data frames are created using `data.frame()`, which takes named vectors as input.
- Every vector becomes a column of the data frame.

```
> forecast <- data.frame(day = c("Mon", "Tue", "Wed", "Thu"),
+                        sky = c("cloudy", "rainy", "sunny", "sunny"),
+                        temp = c(16, 15, 22, 35),
+                        is.nice = c(FALSE, FALSE, TRUE, TRUE))
> forecast
  day    sky temp is.nice
1 Mon cloudy   16   FALSE
2 Tue  rainy   15   FALSE
3 Wed  sunny   22    TRUE
4 Thu  sunny   35    TRUE
```

```
> str(forecast)
'data.frame':   4 obs. of  4 variables:
 $ day    : chr  "Mon" "Tue" "Wed" "Thu"
 $ sky    : chr  "cloudy" "rainy" "sunny" "sunny"
 $ temp   : num  16 15 22 35
 $ is.nice: logi  FALSE FALSE TRUE TRUE
```

```
> class(forecast)
[1] "data.frame"
```

```
> # View(forecast)
```

Subsetting data frames

- Data frames have both rows and columns. Therefore, when subsetting them, we have to specify both which row and which column we want to keep. The two dimensions are separated by commas: `my_df[row, column]`.
- Apart from this, what we know about vectors applies: `row` and `column` can be one of numerical indices, logical indices, or names, or a combination of the three.

```
> forecast
  day    sky temp is.nice
1 Mon cloudy   16   FALSE
2 Tue  rainy   15   FALSE
3 Wed  sunny   22    TRUE
4 Thu  sunny   35    TRUE
```

```
> # if the output is one-dimensional, it is by default simplified to a vector:
> forecast[c(1, 3), "sky"]
[1] "cloudy" "sunny"
```

```
> forecast[c(1, 3), c(2)]
[1] "cloudy" "sunny"
```

When columns or rows are not specified but left empty then all the rows or all the columns are returned.

```
> forecast[, c("day", "temp")] # keep all rows
  day temp
1 Mon   16
2 Tue   15
3 Wed   22
4 Thu   35
```

```
> forecast[c(1, 2), ] # keep all columns
  day    sky temp is.nice
1 Mon cloudy   16   FALSE
2 Tue  rainy   15   FALSE
```

```
> forecast[-c(2), ] # keep all columns, exclude the second row
  day    sky temp is.nice
1 Mon cloudy   16   FALSE
3 Wed  sunny   22    TRUE
4 Thu  sunny   35    TRUE
```

- Often, one wants to extract a single column from a data frame. To do this, remember that technically data frames are lists with each column being a list element. Therefore, you can extract columns using `$` or `[[]]`.

```
> # these two are equivalent - data frames are lists!
> forecast$sky
[1] "cloudy" "rainy"  "sunny"  "sunny"
```

```
> forecast[["sky"]]
[1] "cloudy" "rainy"  "sunny"  "sunny"
```

```
>
> # as above, but the data type is preserved (cf. lists)
> forecast["sky"]
     sky
1 cloudy
2  rainy
3  sunny
4  sunny
```

```
> forecast[c(1, 3)]
  day temp
1 Mon   16
2 Tue   15
3 Wed   22
4 Thu   35
```

- To add new columns, assign using `$` or `[]`.
- Alternatively, you can use the `cbind()` function.
- `rbind()` can be used to create new rows.
- Like with lists, you can delete columns by assigning `NULL` to them.

```
> forecast$wind <- c("calm", "storm", "calm", "tornado") # or forecast["wind"]
> air_pressure <- c(1000, 1010, 1050, 980)
> forecast <- cbind(forecast, air_pressure)
> forecast
  day    sky temp is.nice    wind air_pressure
1 Mon cloudy   16   FALSE    calm         1000
2 Tue  rainy   15   FALSE   storm         1010
3 Wed  sunny   22    TRUE    calm         1050
4 Thu  sunny   35    TRUE tornado          980
```

```
> # delete columns by assigning NULL
> forecast$air_pressure <- NULL # deleted columns
```

Using logical subsetting to "ask questions"

Logical subsetting of data frames allows us to "ask questions".

```
> forecast
  day    sky temp is.nice    wind
1 Mon cloudy   16   FALSE    calm
2 Tue  rainy   15   FALSE   storm
3 Wed  sunny   22    TRUE    calm
4 Thu  sunny   35    TRUE tornado
```

```
> forecast$temp > 20
[1] FALSE FALSE  TRUE  TRUE
```

```
> # on which days is the temperature above 20 degrees?
> forecast$day[forecast$temp > 20]
[1] "Wed" "Thu"
```

```
> # what's the average temperature on cloudy days?
> mean(forecast$temp[forecast$sky == "cloudy"])
[1] 16
```

```
> # extract sub data frame where it is both calm and warmer than 20 deg
> forecast[forecast$wind == "calm" & forecast$temp > 20, ]
  day   sky temp is.nice wind
3 Wed sunny   22    TRUE calm
```

`%in%` is very useful for subsetting.

```
> forecast$day %in% c("Mon", "Wed")
[1]  TRUE FALSE  TRUE FALSE
```

```
> # What is the weather forecast on Monday and Wednesday?
> forecast[forecast$day %in% c("Mon", "Wed"), ]
  day    sky temp is.nice wind
1 Mon cloudy   16   FALSE calm
3 Wed  sunny   22    TRUE calm
```

```
> # task for you: why can't you use the == operator here?
```

Exercise

Do exercise 1 from section "data frames".

Functions

Calling functions

- Functions can take any number of arguments.
- Arguments can be specified by name or position: if you don't provide the argument name, you must provide the arguments in the correct order.
- Arguments may take default values.

```
> # ?sample
> dice <- c(1:6)
> sample(x = dice, size = 1)
[1] 4
```

```
> sample(dice, 1) # order of arguments is important
[1] 4
```

```
> sample(dice, size = 6)
[1] 2 4 3 6 5 1
```

```
> sample(dice, 6, replace = T)
[1] 2 4 5 3 3 1
```

```
> mean(sample(dice, 100000, replace = T))
[1] 3.50454
```

Running examples from the documentation

You can always read the documentation for a function by running `?functionname`. Often, it is very useful to look at the examples section at the end of the help page. You can run the examples automatically by running `example(functionname)`.

The "three dots" (ellipsis)

Sometimes in the function documentation you see three dots: `...` This is called ellipsis and can have two meanings. First, it can mean that the function accepts any number of arguments. Example: `sum()`. Second, sometimes R's functions call more specialised functions and these functions may accept different arguments.

Writing your own functions

- You can write your own functions.
- Use a `return()` statement to control which value is "given back".
- After a `return()` statement the function exits.
- If you don't provide a `return()` statement the last evaluated statement is returned.

```
> square_it <- function(x) {
+   x <- x ^ 2
+   return(x)
+ }
>
> my_potentiator <- function(x, power = 2) {
+   x <- x ^ power
+   return(x)
+ }
>
> my_potentiator(c(1:5))
[1]  1  4  9 16 25
```

Visibility of variables (‘scope’)

- Functions form their own little domain.
- Variables that are created or modified within a function stay in that function.
- We say that the variable is **private** to that function.
- While variables from ‘outside’ are still visible, they are not affected.

```
> my_potentiator
function(x, power = 2) {
  x <- x ^ power
  return(x)
}
```

```
> x <- 5
> my_potentiator(x, power = 3)
[1] 125
```

```
> x
[1] 5
```

```
> z <- 3
> zplusone <- function() {
+   z <- z + 1
+   return(z)
+ }
> zplusone()
[1] 4
```

```
> z
[1] 3
```

```
> rm(z)
> zplusone() # now it throws an error
Error in zplusone(): object 'z' not found
```

Exercise

Run the exercise in section “Writing functions” in the exercise sheet.

Control Structures

- **Control structures** affect the flow of execution in a script.
- `R`’s vectorisation make such structures less important than in other languages.

if - else

- `If` statements do something if their argument evaluates to `TRUE`
- `If` can be followed by `else`
- `If` the argument has length > 1, only the first position is checked
- Remember `==`, `!=`, `!`, `&`, `|`

```
> if (TRUE) {
+   99
+ }
[1] 99
```

```
>
> if (FALSE) {
+   99
+ }
>
> if (FALSE) {
+   99
+ } else {
+   44
+ }
[1] 44
```

```
> odd_or_not <- function(number) {
+   if (!is.numeric(number)) {
+     stop("Must provide a single number")
+   } else if ( (number %% 2) == 0  ) {
+     cat("Number is even.")
+   } else {
+     cat("Number is odd.")
+   }
+ }
> odd_or_not(4)
Number is even.
```

```
> odd_or_not(1)
Number is odd.
```

```
> odd_or_not(7)
Number is odd.
```

Often, the vectorised form of if, `ifelse()` is easier to use.

```
> # example: thresholding of values
> genex <- c(0.2, 0.5, 0.3, 1.1, 2.2)
> my_genex <- data.frame(genex = genex)
> my_genex
  genex
1   0.2
2   0.5
3   0.3
4   1.1
5   2.2
```

```
> my_genex$is.expressed <- ifelse(my_genex$genex > 0.25, TRUE, FALSE)
> my_genex
  genex is.expressed
1   0.2        FALSE
2   0.5         TRUE
3   0.3         TRUE
4   1.1         TRUE
5   2.2         TRUE
```

for loops

For loops step through every value of a vector.

```
> x <- 3:8
> for (i in x) {
+   cat("i is now ", i, "\n") # \n is the newline symbol
+ }
i is now  3
i is now  4
i is now  5
i is now  6
i is now  7
i is now  8
```

Example: go through every row of a data frame.

```
> seq(nrow(forecast))
[1] 1 2 3 4
```

```
> for (i in seq(nrow(forecast))) {
+   print(forecast[i, ])
+ }
  day    sky temp is.nice wind
1 Mon cloudy   16   FALSE calm
  day   sky temp is.nice  wind
2 Tue rainy   15   FALSE storm
  day   sky temp is.nice wind
3 Wed sunny   22    TRUE calm
  day   sky temp is.nice    wind
4 Thu sunny   35    TRUE tornado
```

Importing data and getting an overview

General remarks

Data import is a complex topic and is often more challenging than it should be because people use different conventions for separating fields, indicating NA values etc. R can import data both locally and remotely and can also connect to data bases, scrape data from websites, or connect to GoogleDrive.

Here, we will just briefly cover data import from local files. The base R function is `read.table()` and has plenty of options. However, I recommend using the readr package because it has more options for parsing columns. Check out the tutorial or the data import cheat sheet to learn more about its functionality.

If you have to import data from Excel sheets, check out the package readxl.

Importing the `cells` data set

In this session we will talk about data visualisation and exploration using the ggplot2 package. An important resource here is the detailed online documentation.

To illustrate the different needs that commonly arise during data analysis we will work with the "cells" data set. This data set is the result of an imaging experiment where cells were plated into two wells and were either treated or untreated. Imaging was performed in two channels (mCherry and GFP) and at different exposure times. After segmentation, light intensities were recorded for each individual cell.

First, let's import the data from the `./data` directory.

```
> # brackets around an expression will evaluate it so you don't have to type
> # cells again after assignment to see its contents
> library(readr)
```

```
>
> # reminder: "." in Unix denotes the current (= working) directory, so
> # ./data means: the data directory in the current directory
> (cells <- read_delim("./data/ImagedCells.csv", delim = ";"))
Rows: 190 Columns: 14
-- Column specification --------------------------------------------------------
Delimiter: ";"
chr  (4): Well, treated, gene, cellID
dbl (10): roundness, gfp_T0, mCh_T0, cell_area, gfp_T1, gfp_T2, gfp_T3, mCh_...

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
# A tibble: 190 x 14
   Well  treated gene  cellID    roundness gfp_T0 mCh_T0 cell_area gfp_T1 gfp_T2
   <chr> <chr>   <chr> <chr>         <dbl>  <dbl>  <dbl>     <dbl>  <dbl>  <dbl>
 1 A1    no      wt    cell_A1_1      0.89    281    224      546.    345    445
 2 A1    no      wt    cell_A1_2      0.94    329    248      430.    394    502
 3 A1    no      wt    cell_A1_3      0.85    284    224      445.    339    439
 4 A1    no      wt    cell_A1_4      0.92    331    250      568.    395    506
 5 A1    no      wt    cell_A1_5      0.83    280    220      460.    338    437
 6 A1    no      wt    cell_A1_6      0.91    295    228      486.    353    458
 7 A1    no      wt    cell_A1_7      0.98    332    255      512.    397    511
 8 A1    no      wt    cell_A1_8      0.81    300    244      521.    360    472
 9 A1    no      wt    cell_A1_9      0.83    296    235      497.    353    451
10 A1    no      wt    cell_A1_~      0.88    271    222      558.    326    425
# i 180 more rows
# i 4 more variables: gfp_T3 <dbl>, mCh_T1 <dbl>, mCh_T2 <dbl>, mCh_T3 <dbl>
```

After importing data, the first thing that we usually want to do is to get an overview of the data. This includes simply "looking" at the data, understanding which variables there are, what the number ranges are for different variables etc.

There are a number of useful functions to get an overview of your data.

We will explore such functions plus an additional challenge in the following exercise.

Exercise

Do exercise 1 from the section "Data import and overview".

Plotting

We will only talk about ggplot2 in this course. Other R graphics systems include the base system, grid, and lattice.

ggplot2 is a package by Hadley Wickham. It tries to implement a "grammar of graphics" so that there is a consistent translation between the type of plot that one wants to create and the commands that one needs to call. While the theoretical overhead is a bit higher than in the base system, it is worth the effort because at some point one can "speak plot".

Note that making good plots takes time and it is often not worth tweaking and optimising your plot because you do not know which plots you are actually going to use for your papers or presentations.

There are always a lot of options of what one might want to change in a plot. That is why your best friend will be the ggplot2 documentation, the ggplot2 cheat sheet and, of course, Stack Overflow.

You need to have the ggplot2 package loaded, which is part of the tidyverse. You also need to have the cells data loaded.

```
> library(tidyverse)
> (cells <- read_delim("./data/ImagedCells.csv", delim = ";"))
# A tibble: 190 x 14
   Well  treated gene  cellID      roundness gfp_T0 mCh_T0 cell_area gfp_T1 gfp_T2
   <chr> <chr>   <chr> <chr>           <dbl>  <dbl>  <dbl>     <dbl>  <dbl>  <dbl>
 1 A1    no      wt    cell_A1_1        0.89    281    224      546.    345    445
 2 A1    no      wt    cell_A1_2        0.94    329    248      430.    394    502
 3 A1    no      wt    cell_A1_3        0.85    284    224      445.    339    439
 4 A1    no      wt    cell_A1_4        0.92    331    250      568.    395    506
 5 A1    no      wt    cell_A1_5        0.83    280    220      460.    338    437
 6 A1    no      wt    cell_A1_6        0.91    295    228      486.    353    458
 7 A1    no      wt    cell_A1_7        0.98    332    255      512.    397    511
 8 A1    no      wt    cell_A1_8        0.81    300    244      521.    360    472
 9 A1    no      wt    cell_A1_9        0.83    296    235      497.    353    451
10 A1    no      wt    cell_A1_~        0.88    271    222      558.    326    425
# i 180 more rows
# i 4 more variables: gfp_T3 <dbl>, mCh_T1 <dbl>, mCh_T2 <dbl>, mCh_T3 <dbl>
```

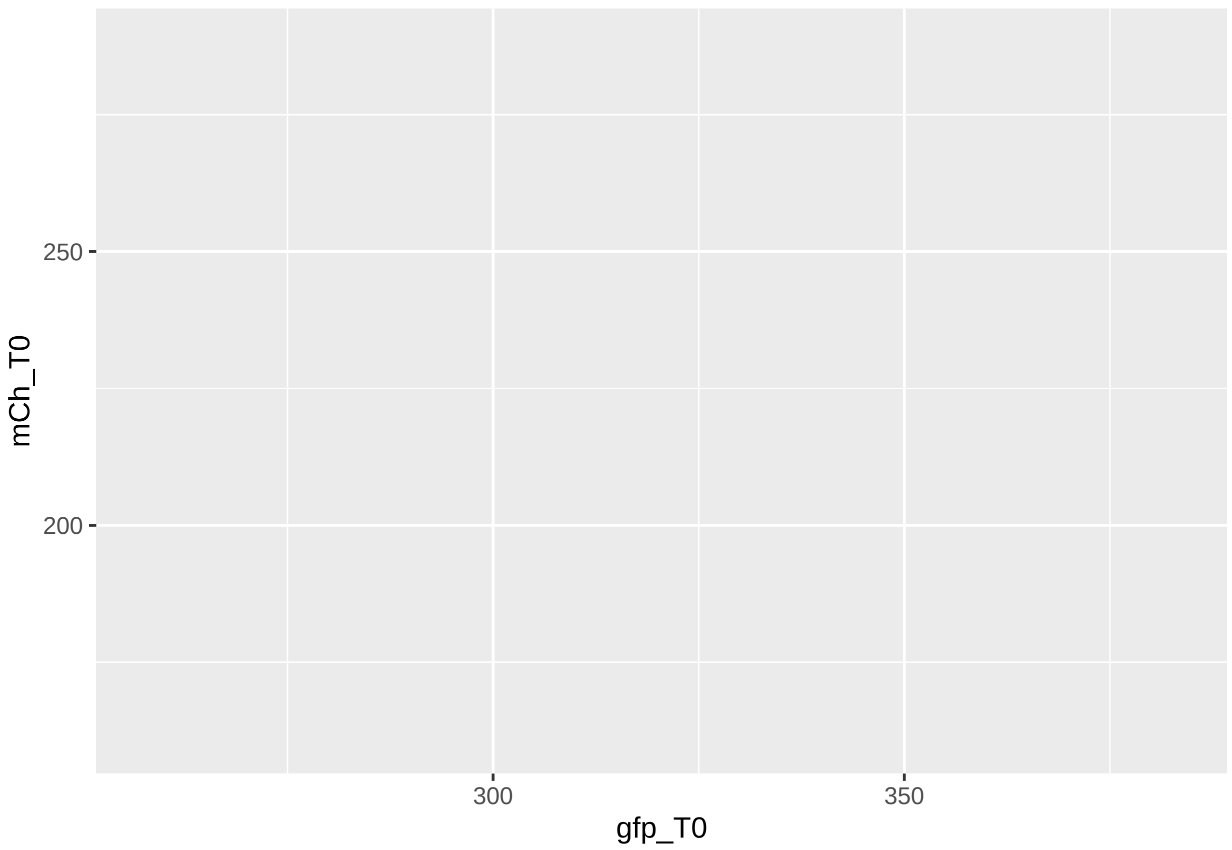
Basic functionality: scatter plots

Scatter plots are very useful to understand the basic working principles of ggplot2.

In ggplot2, plots are created by first calling the function `ggplot(data, mapping)` and then adding different layers to the plot using `+`. These layers can be so-called "geoms" such as points, lines, etc. but also statistical transformations, axis labels, and other modifications.

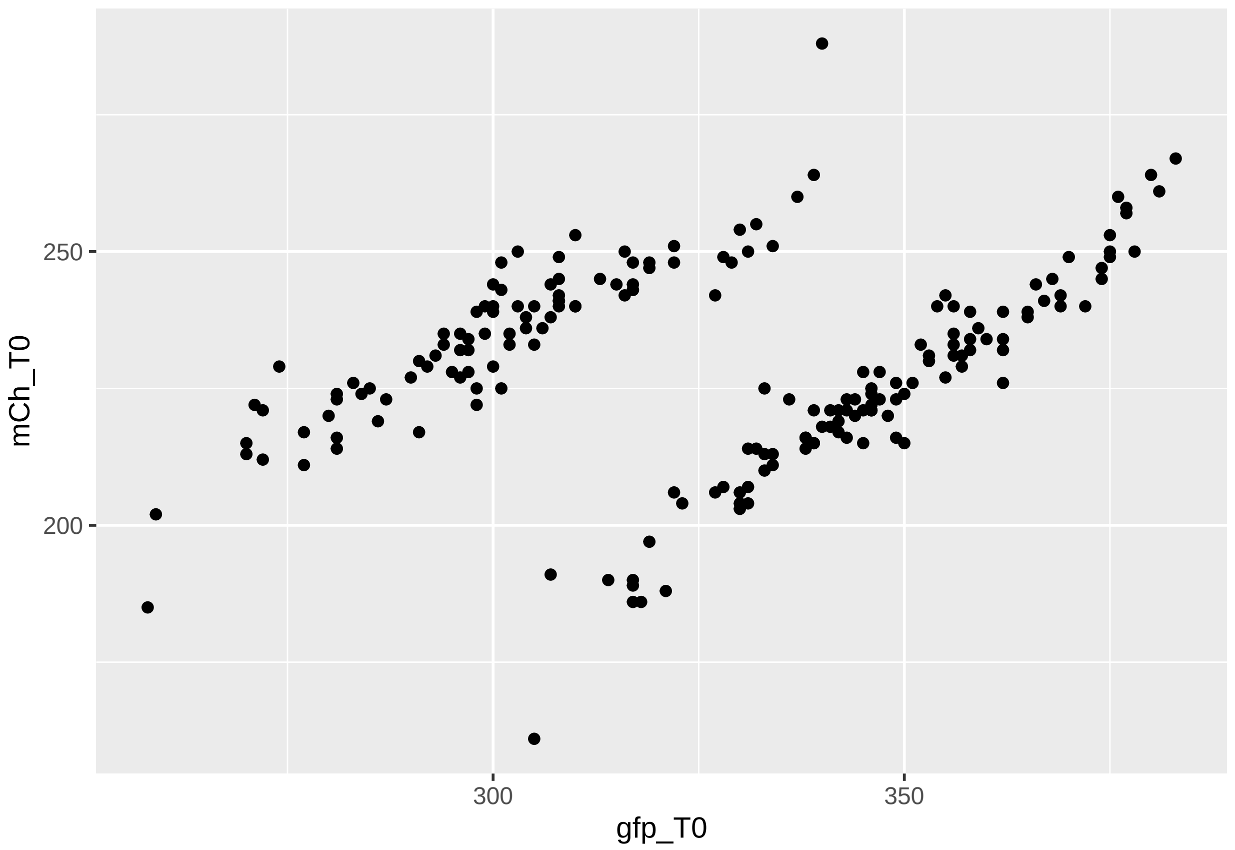
A ggplot always starts with a call to `ggplot(data, mapping)`. You need to specify which data ggplot should operate on and an aesthetic mapping, which indicates which variables should be mapped to which property of the plot.

```
> ggplot(data = cells, aes(x = gfp_T0, y = mCh_T0))
```



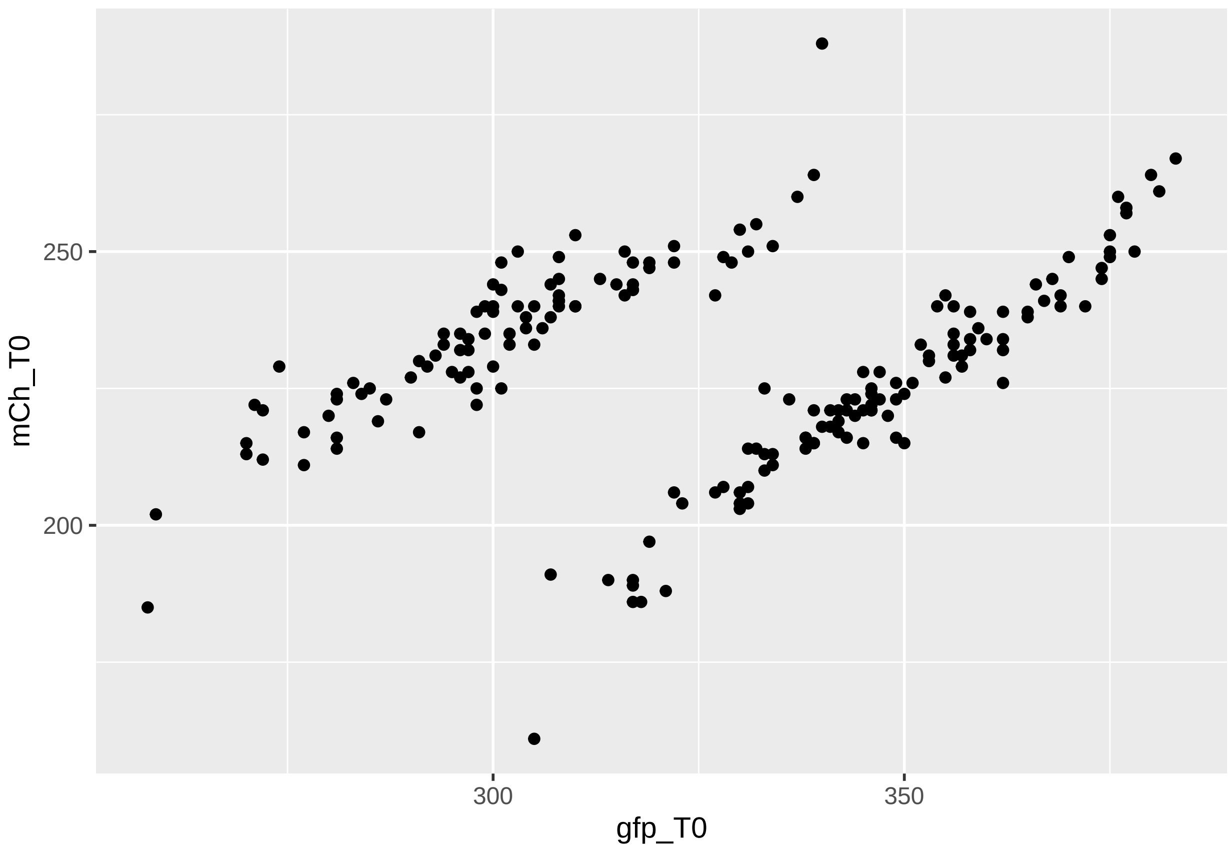
At this stage, you will not be able to see anything because you have not added any "geom" to the plot:

```
> ggplot(data = cells, aes(x = gfp_T0, y = mCh_T0)) +
+   geom_point()
```



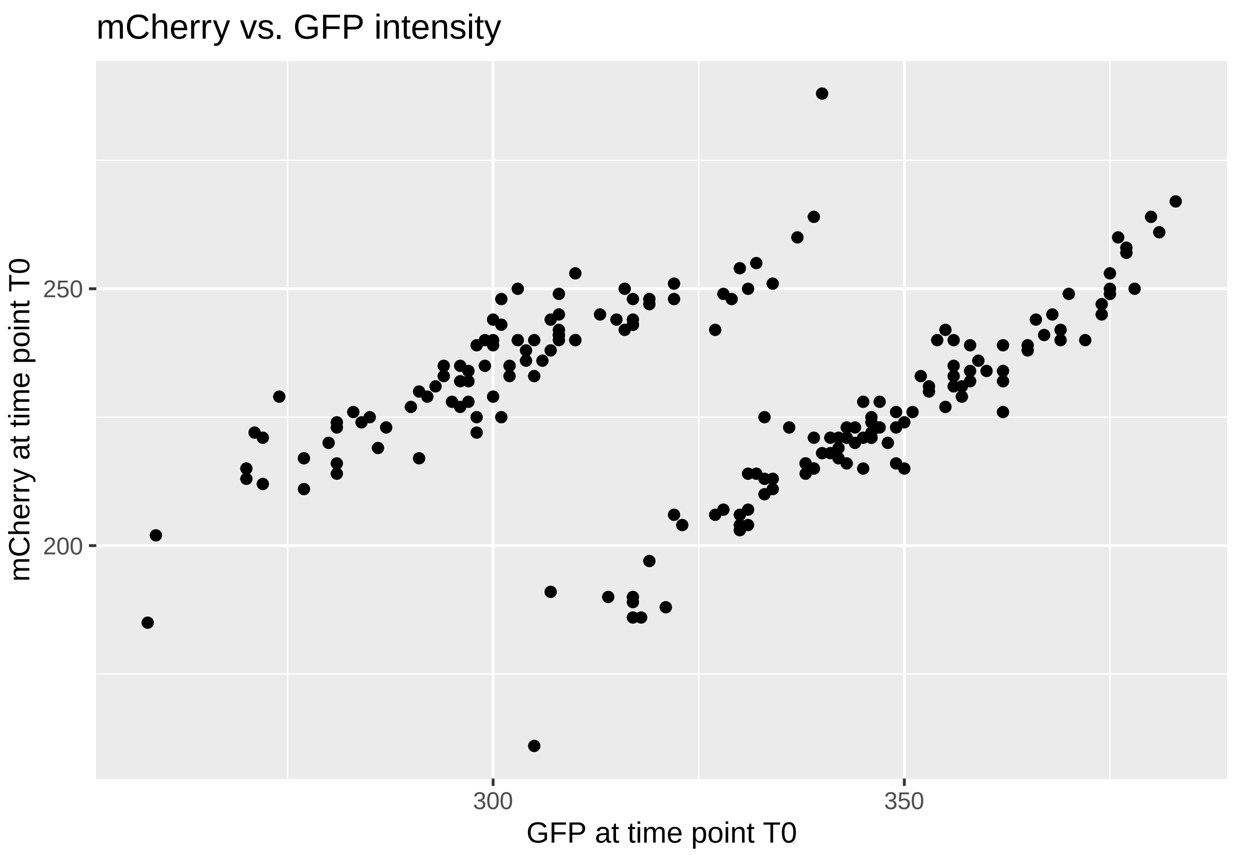
ggplot objects can be assigned to variables and evaluated later on, which will display the plot:

```
> p <- ggplot(data = cells, aes(x = gfp_T0, y = mCh_T0)) +
+   geom_point()
>
> p
```



Add `labs()` to the ggplot command to annotate the axes and add a title.

```
> p <- p + labs(x = "GFP at time point T0", y = "mCherry at time point T0",
+   title = "mCherry vs. GFP intensity")
>
> p
```



Saving ggplots

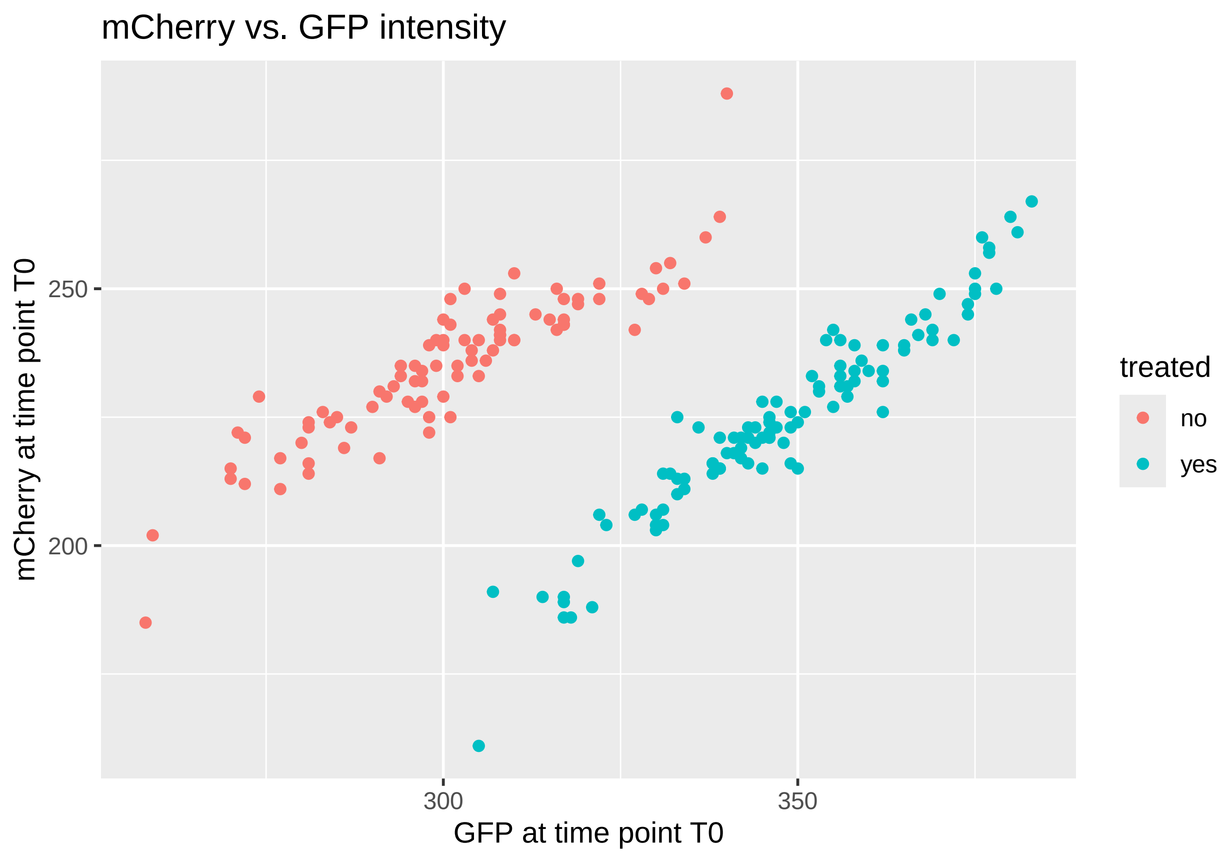
ggplot2 plots can be saved with function `ggsave(filename, plot)`. If the `plot` argument is not specified, the last displayed plot is saved. The filetype is inferred from the extension.

```
> # units are in inches
> ggsave(filename = "A_plot_has_been_born.pdf", plot = p, width = 5, height = 5)
```

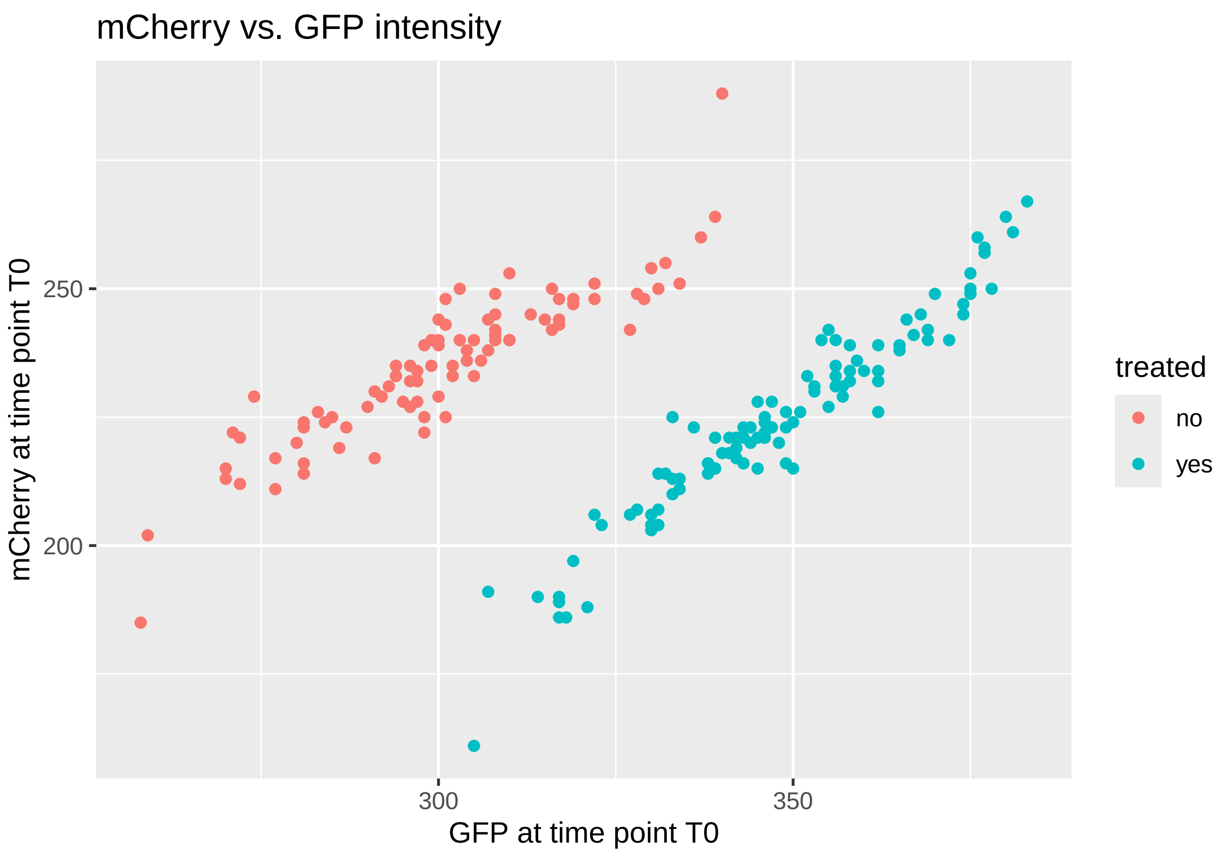
Aesthetic mappings

Aesthetic mappings are used to add additional visual properties to the plot. Let us use colour to indicate whether cells have been treated or not:

```
> ggplot(data = cells, aes(x = gfp_T0, y = mCh_T0)) +
+   geom_point(aes(colour = treated)) +
+   labs(x = "GFP at time point T0", y = "mCherry at time point T0",
+      title = "mCherry vs. GFP intensity")
```



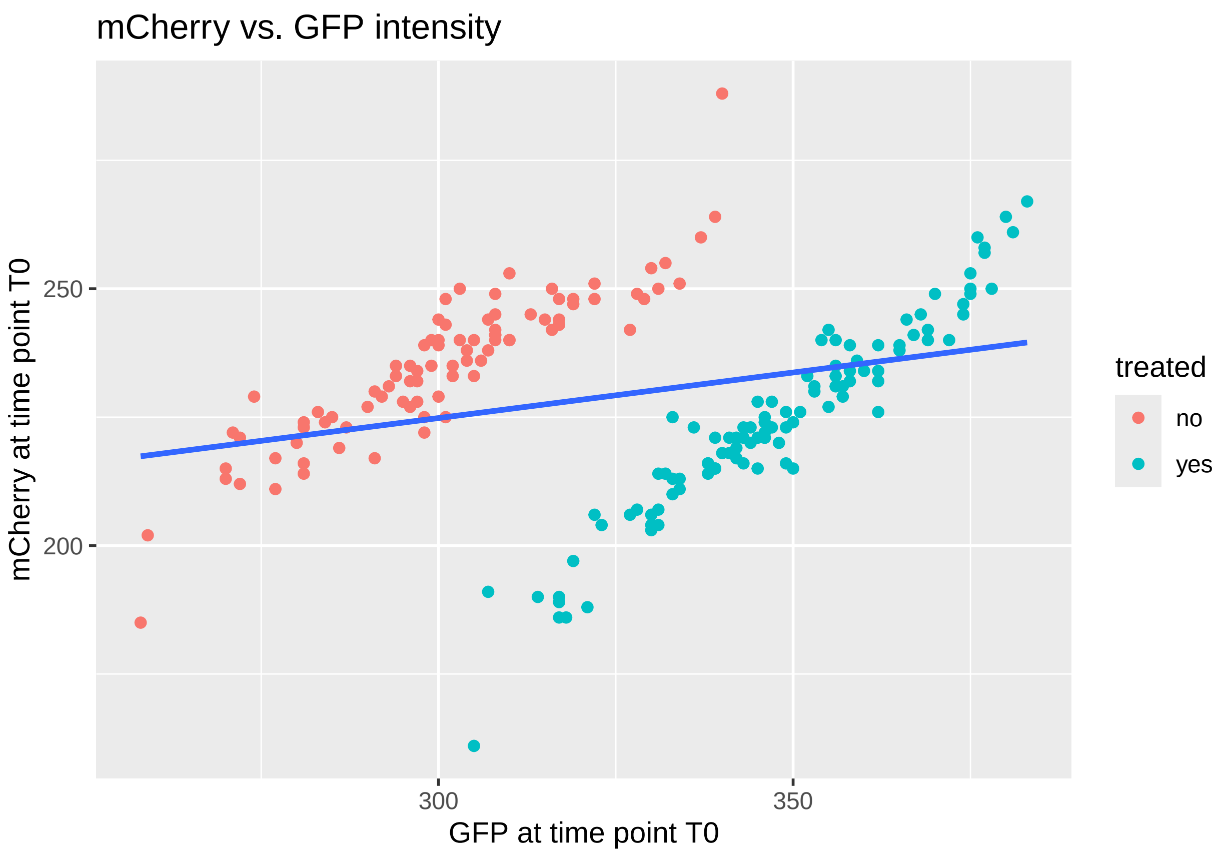
```
>
> # any part of the plotting command can be saved in a variable:
> my_label <- labs(x = "GFP at time point T0", y = "mCherry at time point T0",
+   title = "mCherry vs. GFP intensity")
>
> ggplot(data = cells, aes(x = gfp_T0, y = mCh_T0)) +
+   geom_point(aes(colour = treated)) +
+   my_label
```



Adding model fits to ggplot2

Something that one wants to do quite often is to plot a model fit such as a linear regression in a plot.

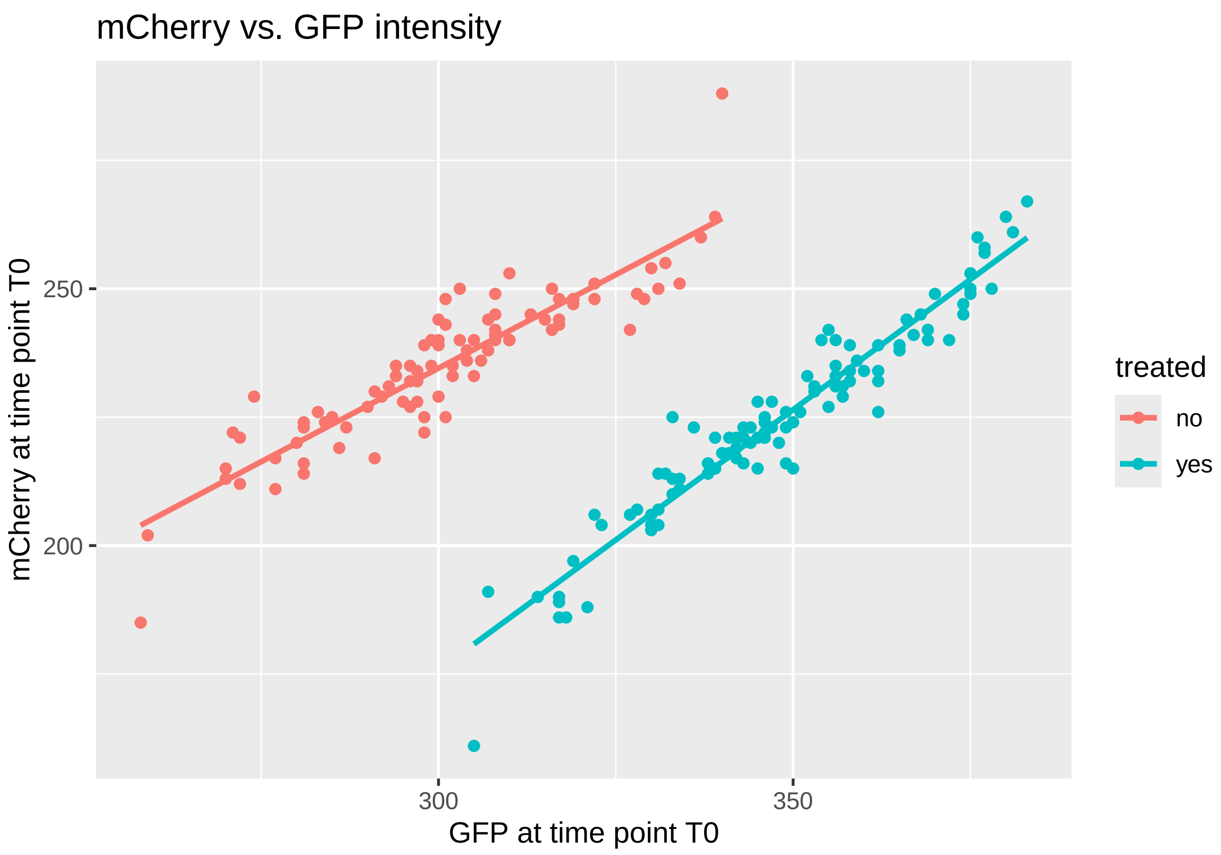
```
> ggplot(data = cells, aes(x = gfp_T0, y = mCh_T0)) +
+   geom_point(aes(colour = treated)) +
+   geom_smooth(method = "lm", se = FALSE) +
+   my_label
`geom_smooth()` using formula = 'y ~ x'
```



This was not quite what we wanted. The colour aesthetic is only used for `geom_point()`. Therefore, while the dots are painted in different colours to discriminate the `treated` variable, the linear regression is plotted for the complete data set.

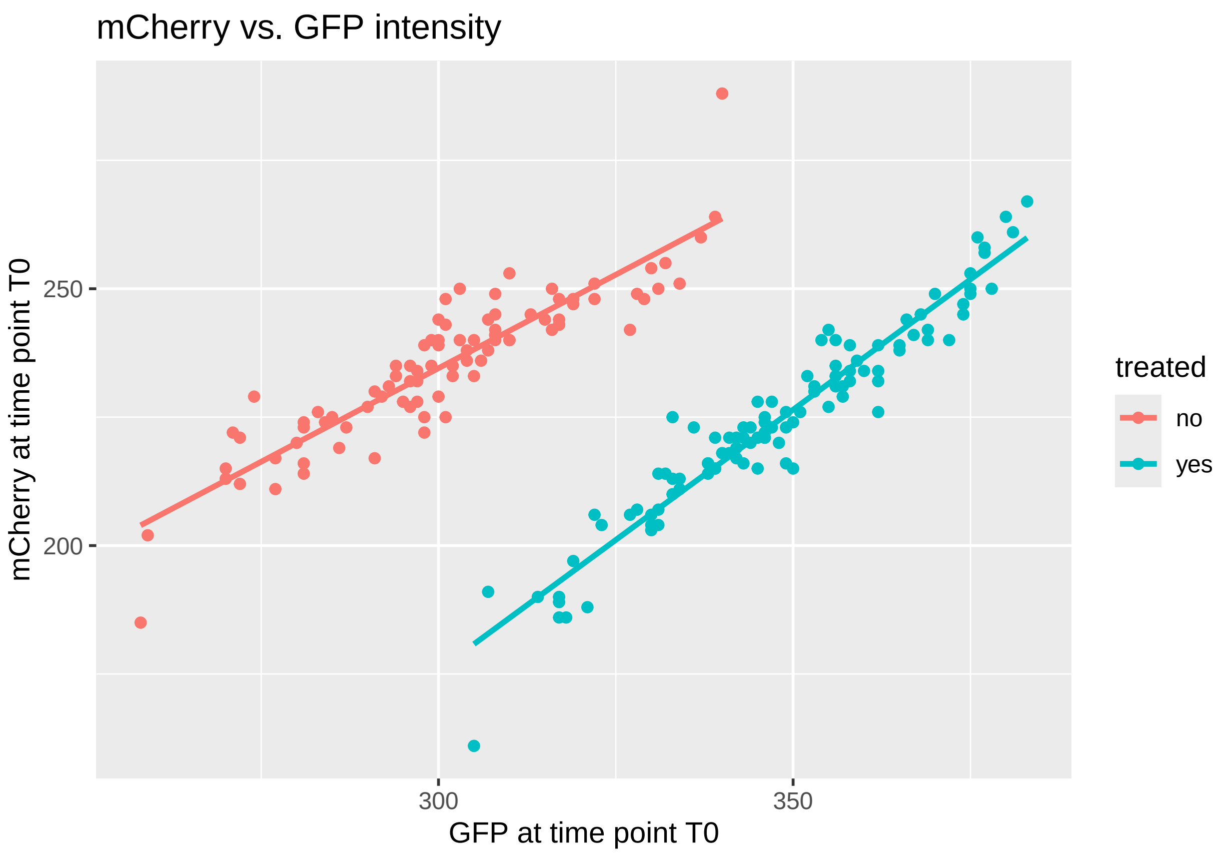
One solution is to specify the colour mapping again in the `geom_smooth()` call:

```
> ggplot(data = cells, aes(x = gfp_T0, y = mCh_T0)) +
+   geom_point(aes(colour = treated)) +
+   geom_smooth(aes(colour = treated), method = "lm", se = FALSE) +
+   my_label
`geom_smooth()` using formula = 'y ~ x'
```



Alternatively, the aesthetic mapping can be specified at the "top level", i.e. in the call to `ggplot()`. We say that the other layers *inherit* the aesthetic from the `ggplot()` command.

```
> # using inheritance
> ggplot(data = cells, aes(x = gfp_T0, y = mCh_T0, colour = treated)) +
+   geom_point() +
+   geom_smooth(method = "lm", se = FALSE) +
+   my_label
`geom_smooth()` using formula = 'y ~ x'
```



Exercise

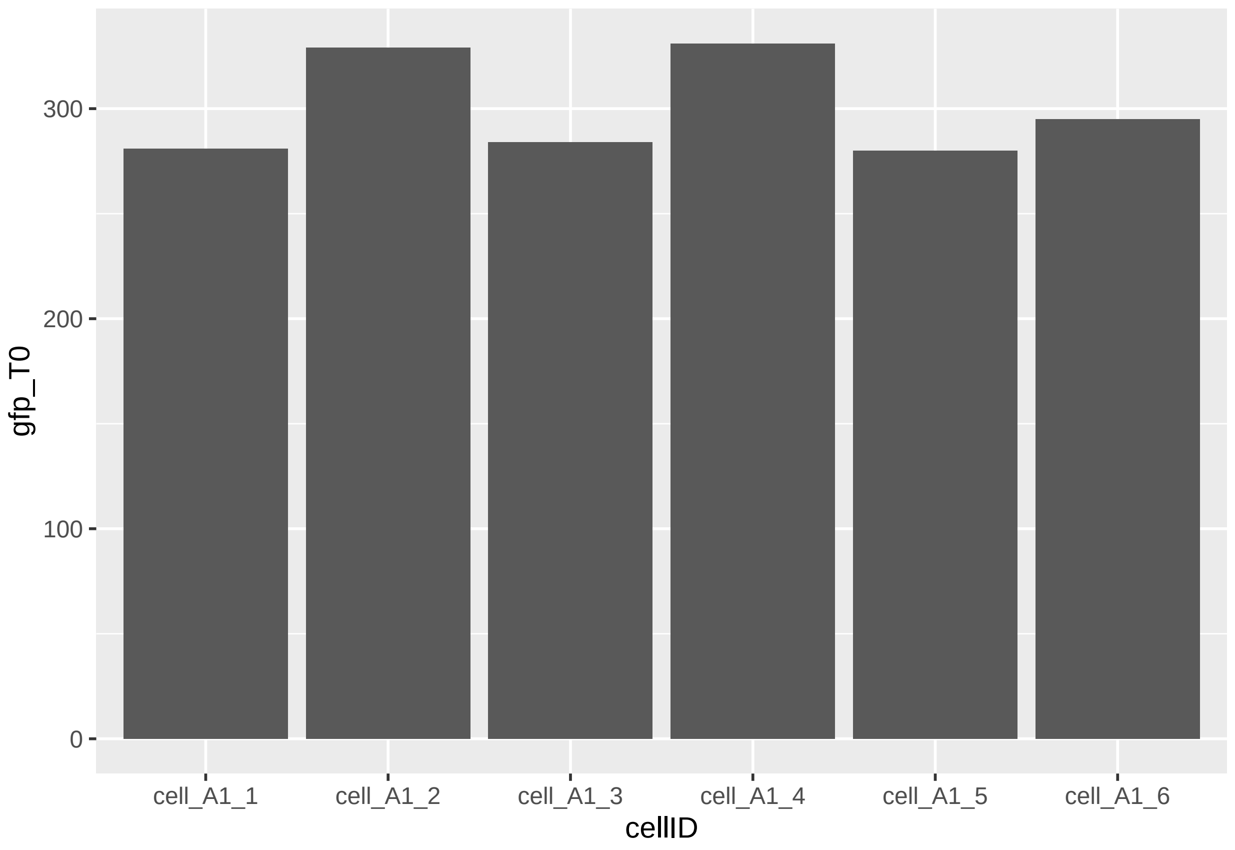
Do exercises 1 and 2 from section "Creating plots"

Displaying quantities and distributions

Barplots

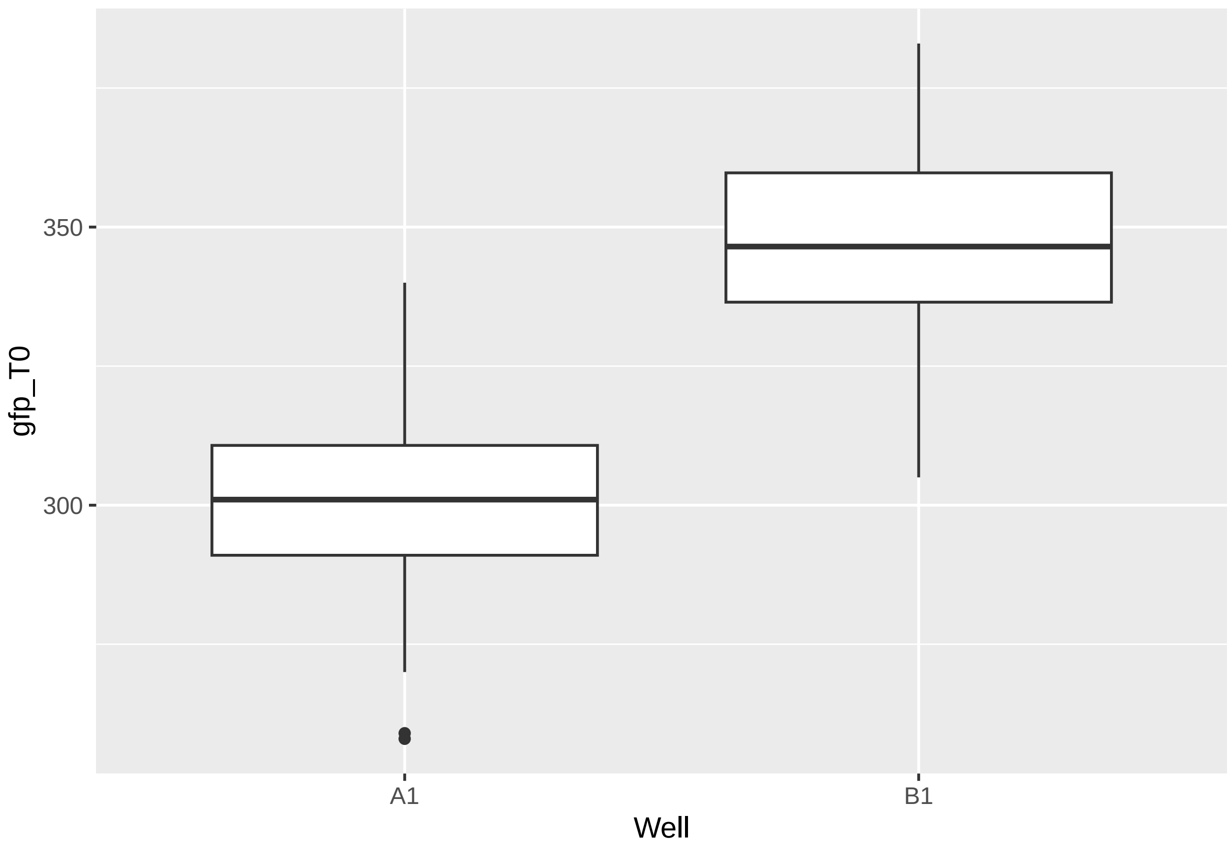
You can get barplots with `geom_col()`.

```
> first_six_cells <- cells[1:6, ]
>
> ggplot(first_six_cells, aes(x = cellID, y = gfp_T0)) +
+   geom_col()
```



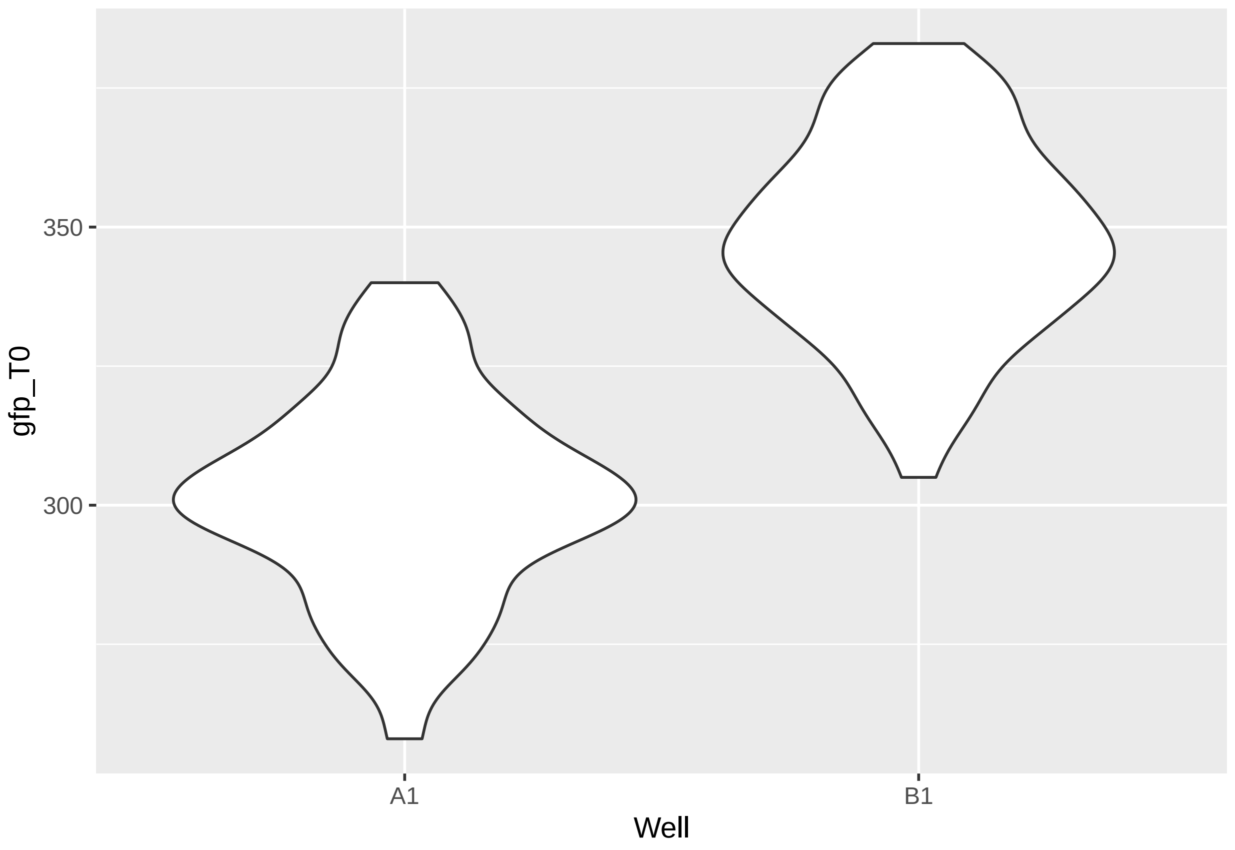
Boxplots

```
> ggplot(cells, aes(x = Well, y = gfp_T0)) +
+   geom_boxplot()
```

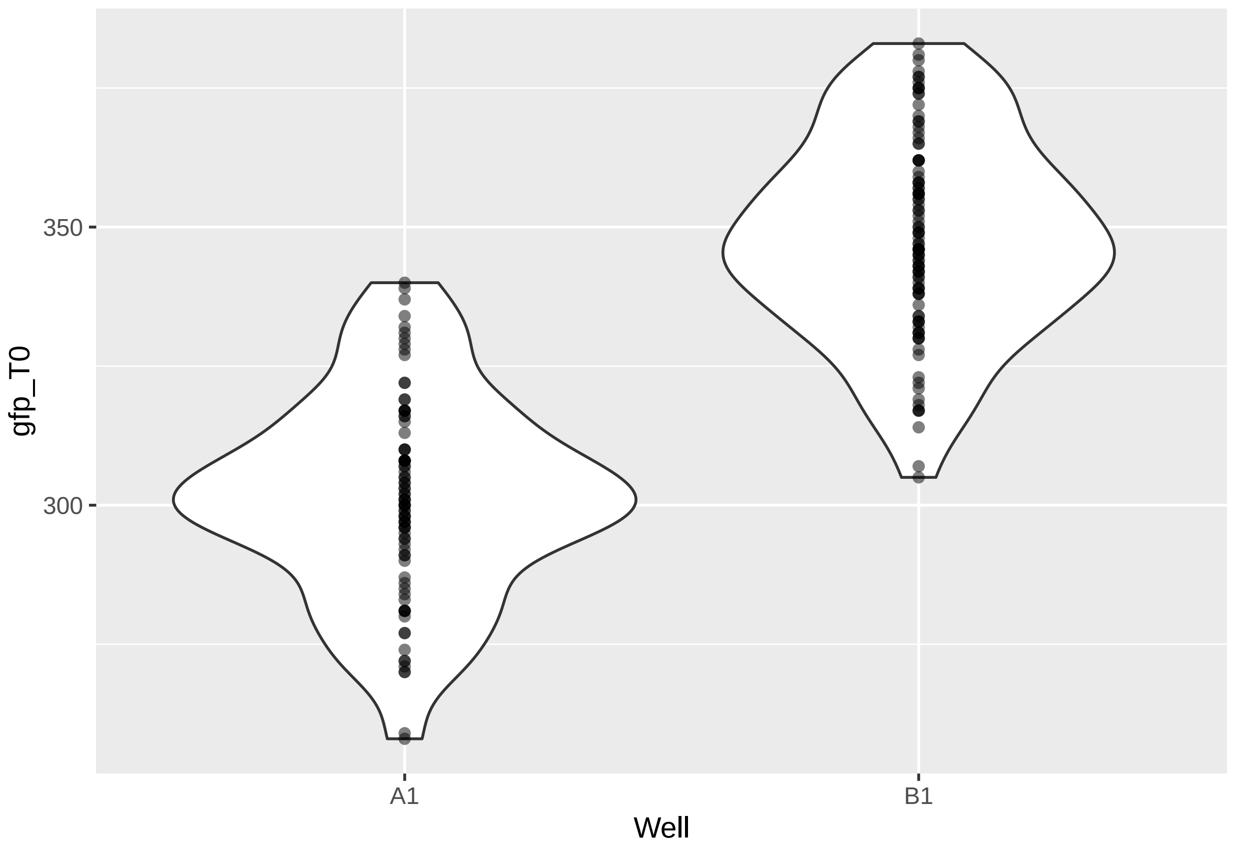


Violin plots and beeswarm plots, jittering

```
> ggplot(cells, aes(x = Well, y = gfp_T0)) +
+   geom_violin()
```

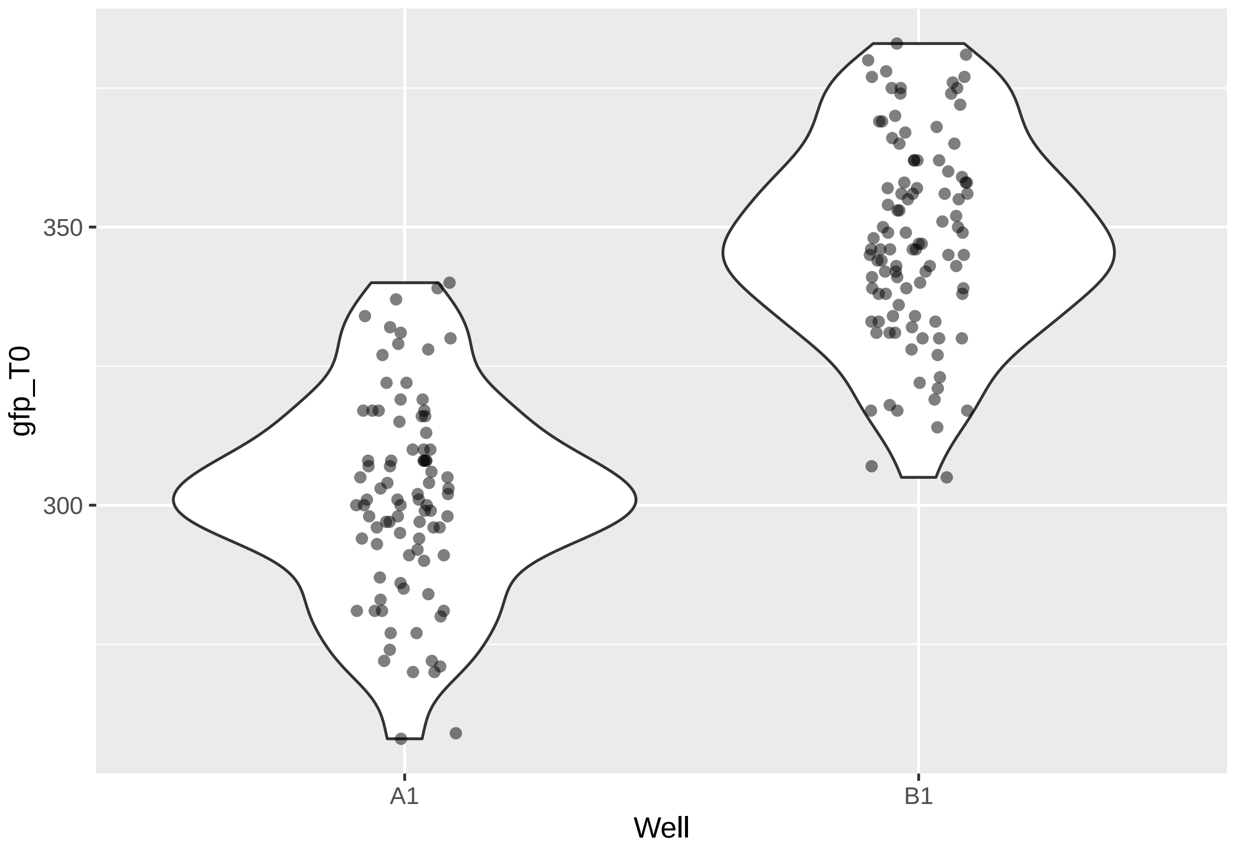


```
>
> ggplot(cells, aes(x = Well, y = gfp_T0)) +
+   geom_violin() +
+   geom_point(alpha = 0.5)
```



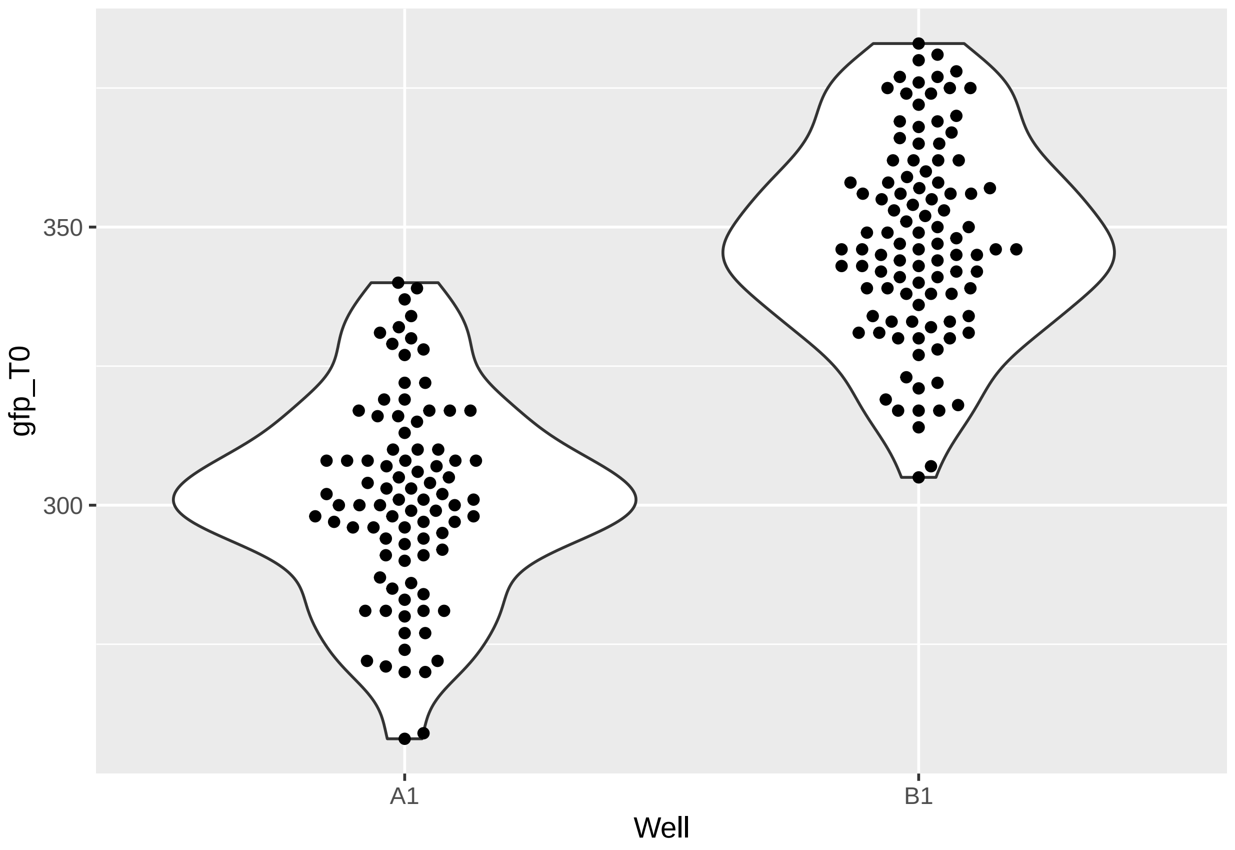
The problem here is that the points are stacked on top of each other. To prevent this, you can add some noise to the points, which is also known as “jittering”. Be careful not to change the actual values if you use this.

```
> # set height to 0!
> ggplot(cells, aes(x = Well, y = gfp_T0)) +
+   geom_violin() +
+   geom_point(alpha = 0.5, position = position_jitter(width = 0.1, height = 0))
```



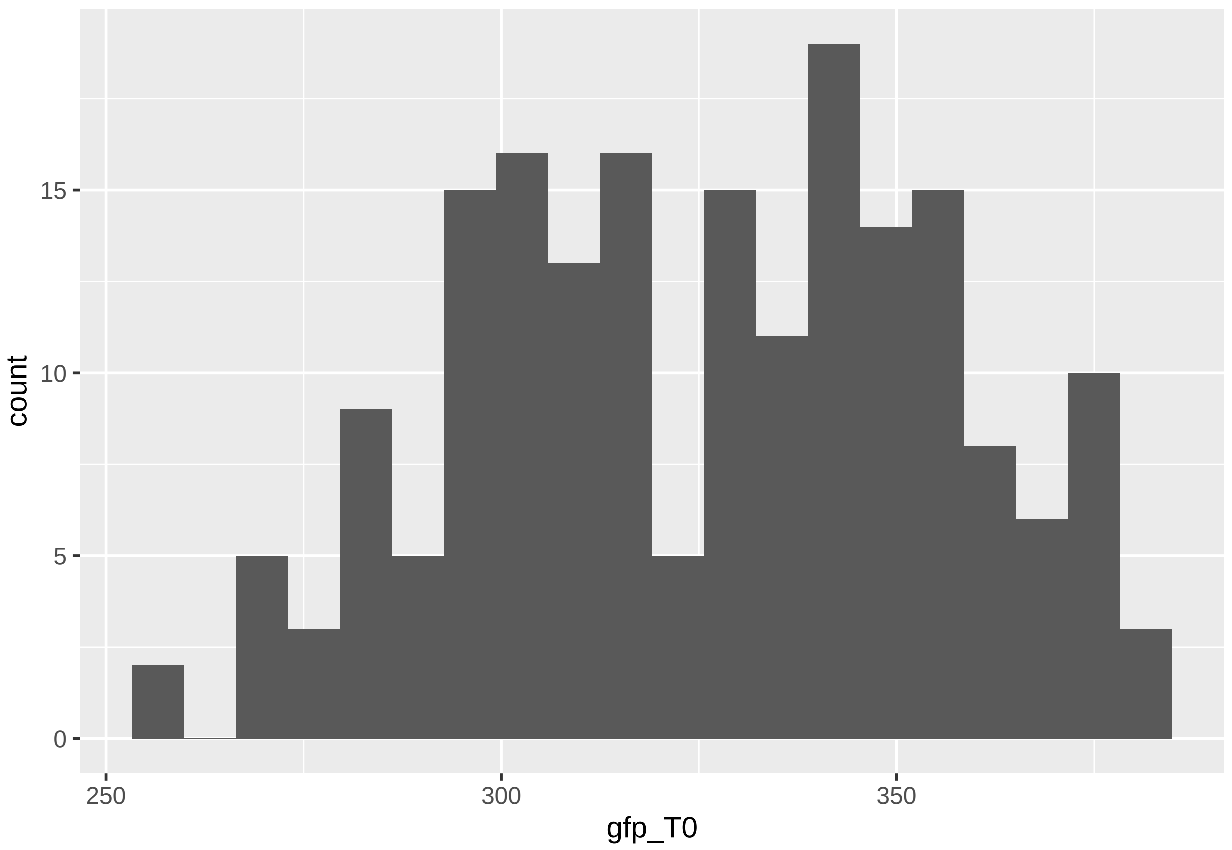
Beeswarm plots are an excellent alternative to jittering, especially for low- or medium-sized data sets.

```
> ggplot(cells, aes(x = Well, y = gfp_T0)) +
+   geom_violin() +
+   ggbeeswarm::geom_beeswarm(cex = 2)
```



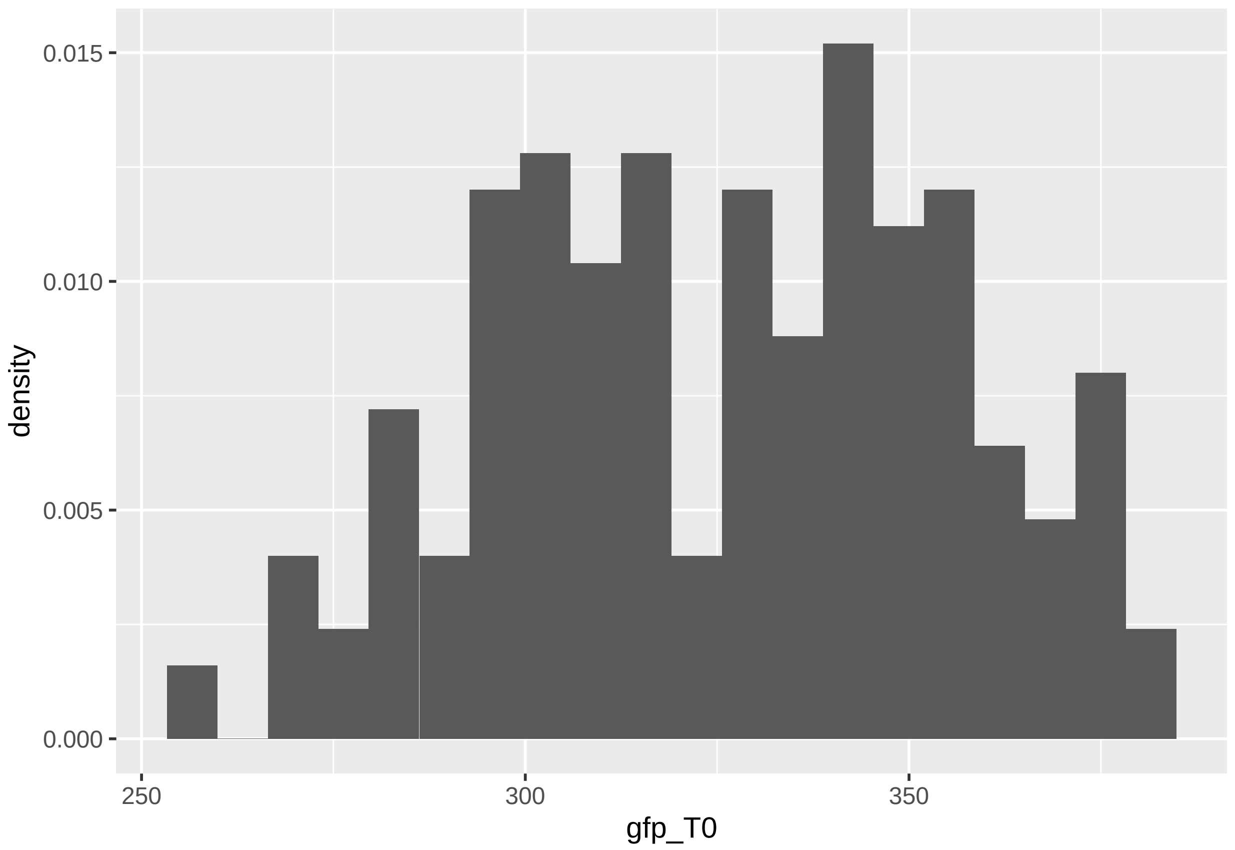
Distributions and density plots

```
> ggplot(cells, aes(x = gfp_T0)) +
+   geom_histogram(bins = 20)
```



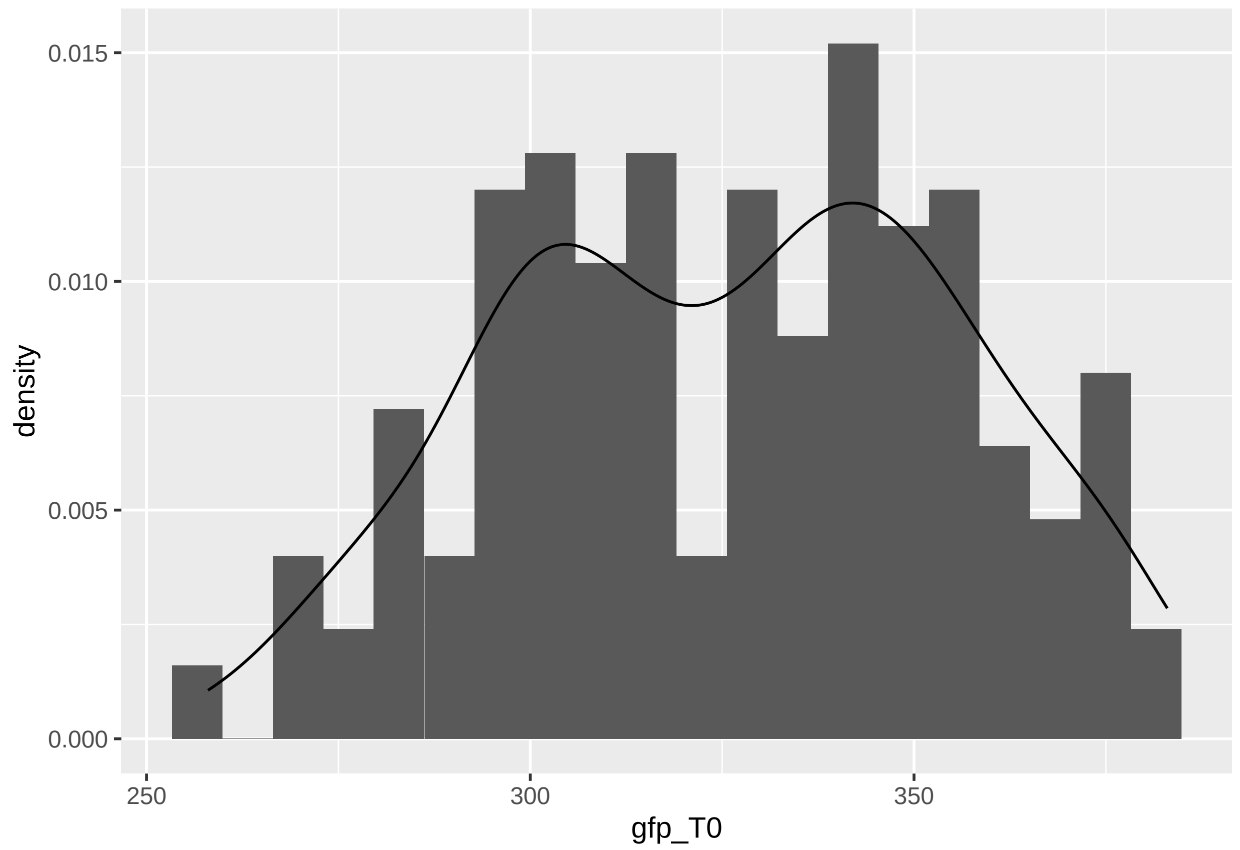
By default, histograms will plot counts. If you want to plot the densities you have to specify `y = ..density..`. The `..` syntax means that you want to access a variable that was computed internally by ggplot2 when running `geom_histogram()`.

```
> ggplot(cells, aes(x = gfp_T0)) +
+   geom_histogram(aes(y = after_stat(density)), bins = 20)
```

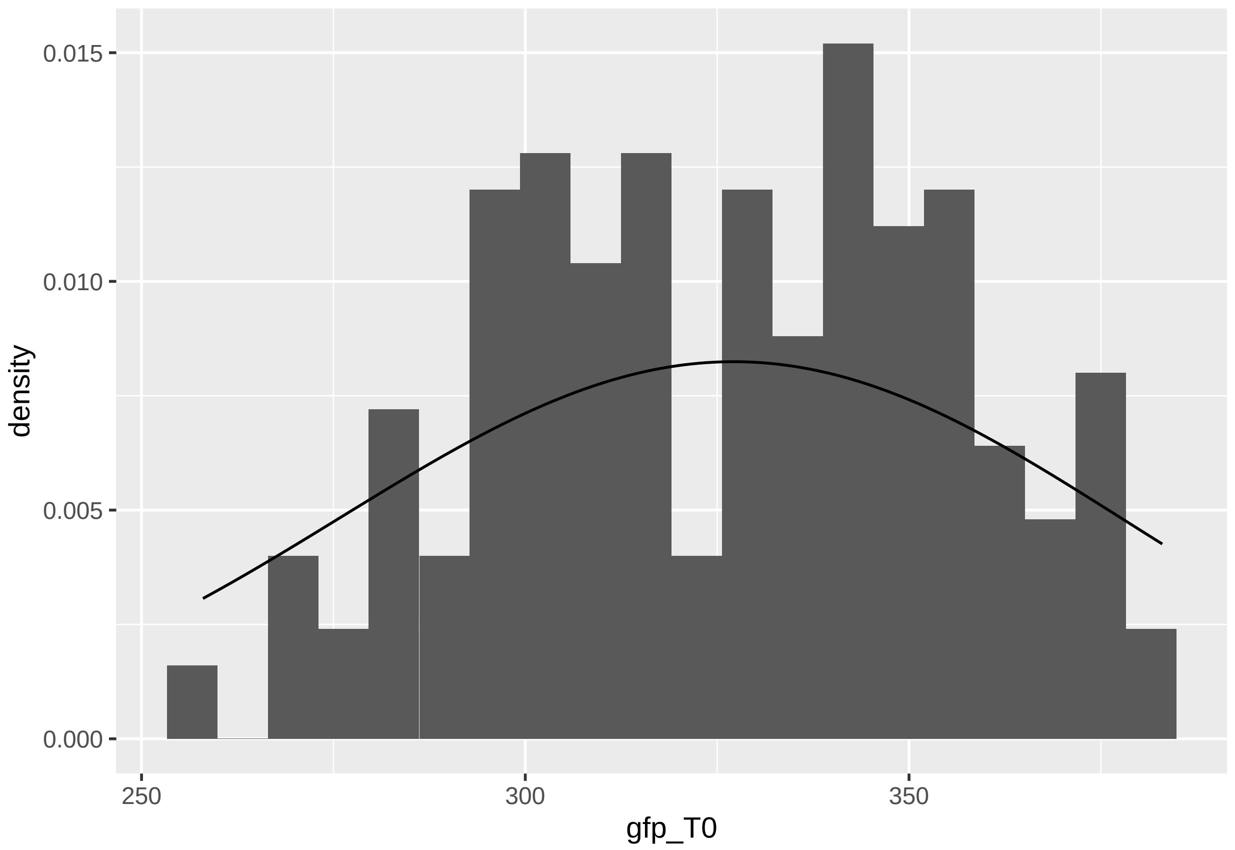


To plot a smoothed density curve, add `geom_density()`.

```
> ggplot(cells, aes(x = gfp_T0)) +
+   geom_histogram(aes(y = after_stat(density)), bins = 20) +
+   geom_density()
```

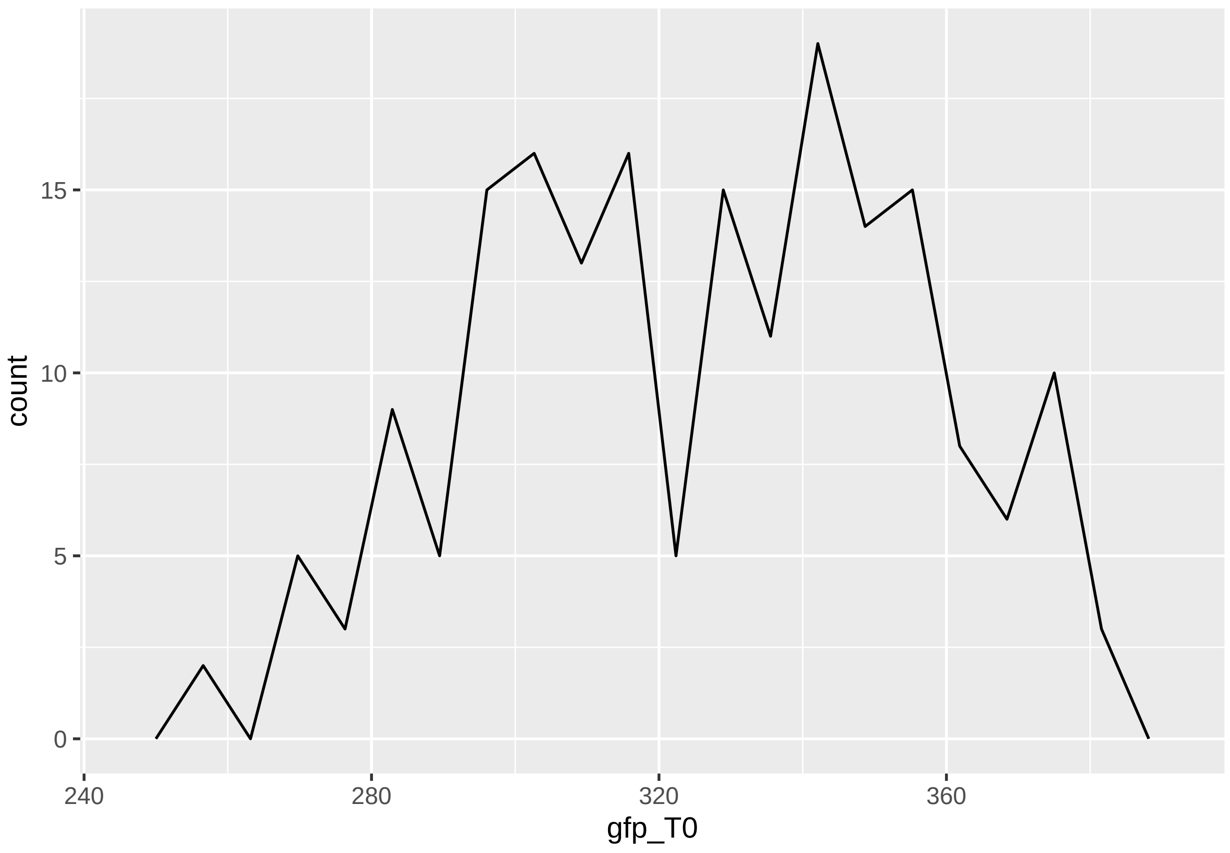


```
>
> ggplot(cells, aes(x = gfp_T0)) +
+   geom_histogram(aes(y = after_stat(density)), bins = 20) +
+   geom_density(adjust = 4)
```



A nice alternative to histograms is the *frequency polygon*.

```
> ggplot(cells, aes(x = gfp_T0)) +
+   geom_freqpoly(bins = 20)
```



Exercises

Do exercises 3 and 4 from the section “Creating plots”.

The time series challenge, aka tidy data aka my data is in a bad shape

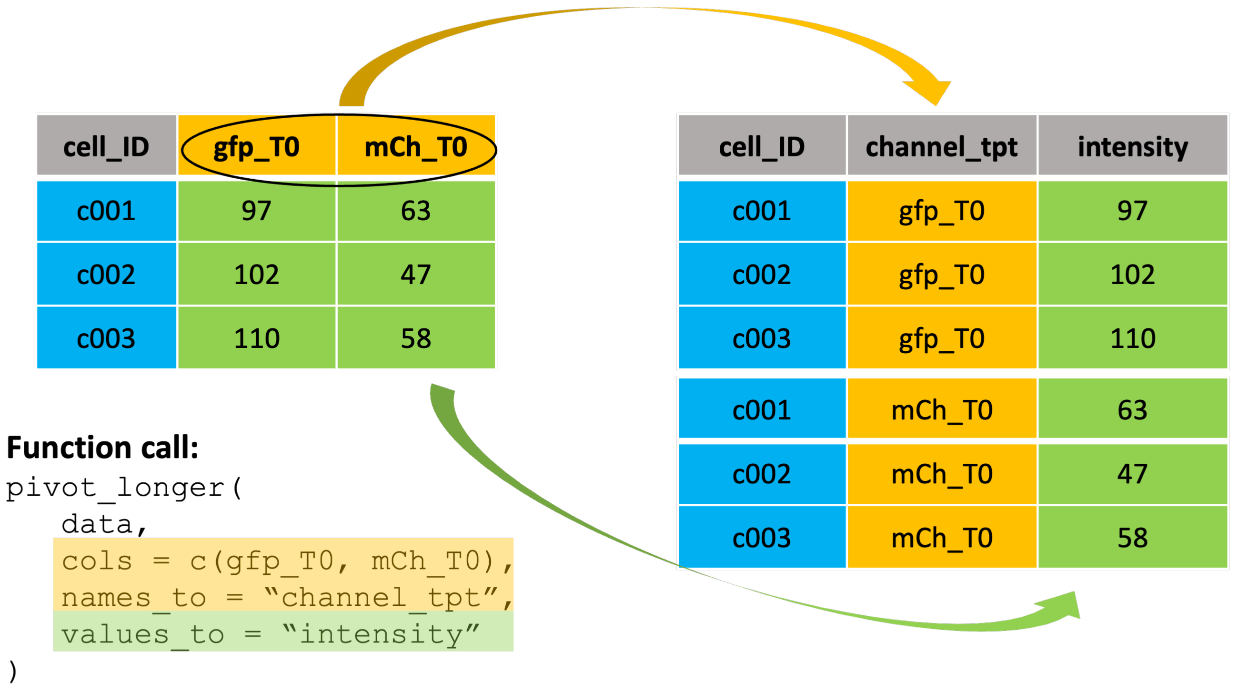
When we look at our `cells` data, we can see that it is in a useful format to plot correlations between channels and time points. However, what if we wanted to plot different colour channels (i.e. gfp vs. mCherry), not wells, in different colours? In principle, all we need to do is to map the colour aesthetic to a channel variable. However, while the information on colour channels is in the table, it is not in a format that can be used for the purpose since the variable we want to map is “spread out” across several columns.

To solve this problem, we need to “reshape” or “melt” our table. Instead of having columns such as `gfp_T0`, which actually represent several pieces of information, we want to have 3 new columns, one for the colour channel, one for the time point, and one for the intensity.

Data where each column represents *one* variable, each row *one* observation, and each cell contains *one* value is also called *tidy data*.

It is usually preferable to have your data in a tidy format. The reason is that there is usually just one way of having tidy data. This means that tidy data is a sort of standard that makes it easier to think about your data. The idea of tidy data is very powerful and inspired the name “tidyverse”. Check the free online book R for Data Science by Hadley Wickham and Garrett Grolemund for a more thorough introduction into the topic.

So how can we make data tidy? It may involve several steps but often one essential operation is called “melting”. A scheme is shown in the figure below. When melting a data frame we transform it in a way that several columns representing the same variable form a new column in the transformed data frame. The values of the columns in the original data are also collected in a new column. We also say that we convert data from a "wide" to a "long" format. In R we can perform this operation using the `pivot_longer()` function from the tidyr package.



cell_ID	gfp_T0	mCh_T0
c001	97	63
c002	102	47
c003	110	58

cell_ID	channel_tpt	intensity
c001	gfp_T0	97
c002	gfp_T0	102
c003	gfp_T0	110
c001	mCh_T0	63
c002	mCh_T0	47
c003	mCh_T0	58

Function call:

```
pivot_longer(
    data,
    cols = c(gfp_T0, mCh_T0),
    names_to = "channel_tpt",
    values_to = "intensity"
)
```

Figure 1: Converting data from 'wide' to 'long' format with `pivot_longer()`

```
> # before:
> cells
# A tibble: 190 x 14
   Well  treated gene  cellID     roundness gfp_T0 mCh_T0 cell_area gfp_T1 gfp_T2
   <chr> <chr>   <chr> <chr>          <dbl>  <dbl>  <dbl>     <dbl>  <dbl>  <dbl>
 1 A1    no      wt    cell_A1_1       0.89    281    224      546.    345    445
 2 A1    no      wt    cell_A1_2       0.94    329    248      430.    394    502
 3 A1    no      wt    cell_A1_3       0.85    284    224      445.    339    439
 4 A1    no      wt    cell_A1_4       0.92    331    250      568.    395    506
 5 A1    no      wt    cell_A1_5       0.83    280    220      460.    338    437
 6 A1    no      wt    cell_A1_6       0.91    295    228      486.    353    458
 7 A1    no      wt    cell_A1_7       0.98    332    255      512.    397    511
 8 A1    no      wt    cell_A1_8       0.81    300    244      521.    360    472
 9 A1    no      wt    cell_A1_9       0.83    296    235      497.    353    451
10 A1    no      wt    cell_A1_~       0.88    271    222      558.    326    425
# i 180 more rows
# i 4 more variables: gfp_T3 <dbl>, mCh_T1 <dbl>, mCh_T2 <dbl>, mCh_T3 <dbl>
```

```
>
> # and after:
> cells_tidy <- pivot_longer(cells,
+                            cols = c(gfp_T0, mCh_T0, gfp_T1, gfp_T2, gfp_T3,
+                            mCh_T1, mCh_T2, mCh_T3),
+                            names_to = "channel_tpt",
+                            values_to = "int")
>
```

```
> cells_tidy
# A tibble: 1,520 x 8
   Well  treated gene  cellID    roundness cell_area channel_tpt   int
   <chr> <chr>   <chr> <chr>         <dbl>     <dbl> <chr>       <dbl>
 1 A1    no      wt    cell_A1_1      0.89      546. gfp_T0        281
 2 A1    no      wt    cell_A1_1      0.89      546. mCh_T0        224
 3 A1    no      wt    cell_A1_1      0.89      546. gfp_T1        345
 4 A1    no      wt    cell_A1_1      0.89      546. gfp_T2        445
 5 A1    no      wt    cell_A1_1      0.89      546. gfp_T3        496
 6 A1    no      wt    cell_A1_1      0.89      546. mCh_T1        228
 7 A1    no      wt    cell_A1_1      0.89      546. mCh_T2        232
 8 A1    no      wt    cell_A1_1      0.89      546. mCh_T3        236
 9 A1    no      wt    cell_A1_2      0.94      430. gfp_T0        329
10 A1    no      wt    cell_A1_2      0.94      430. mCh_T0        248
# i 1,510 more rows
```

However, our data is not yet completely tidied up. We still need to separate the `channel_timepoint` column into two new columns. For this, there is the `separate()` function in the dplyr package.

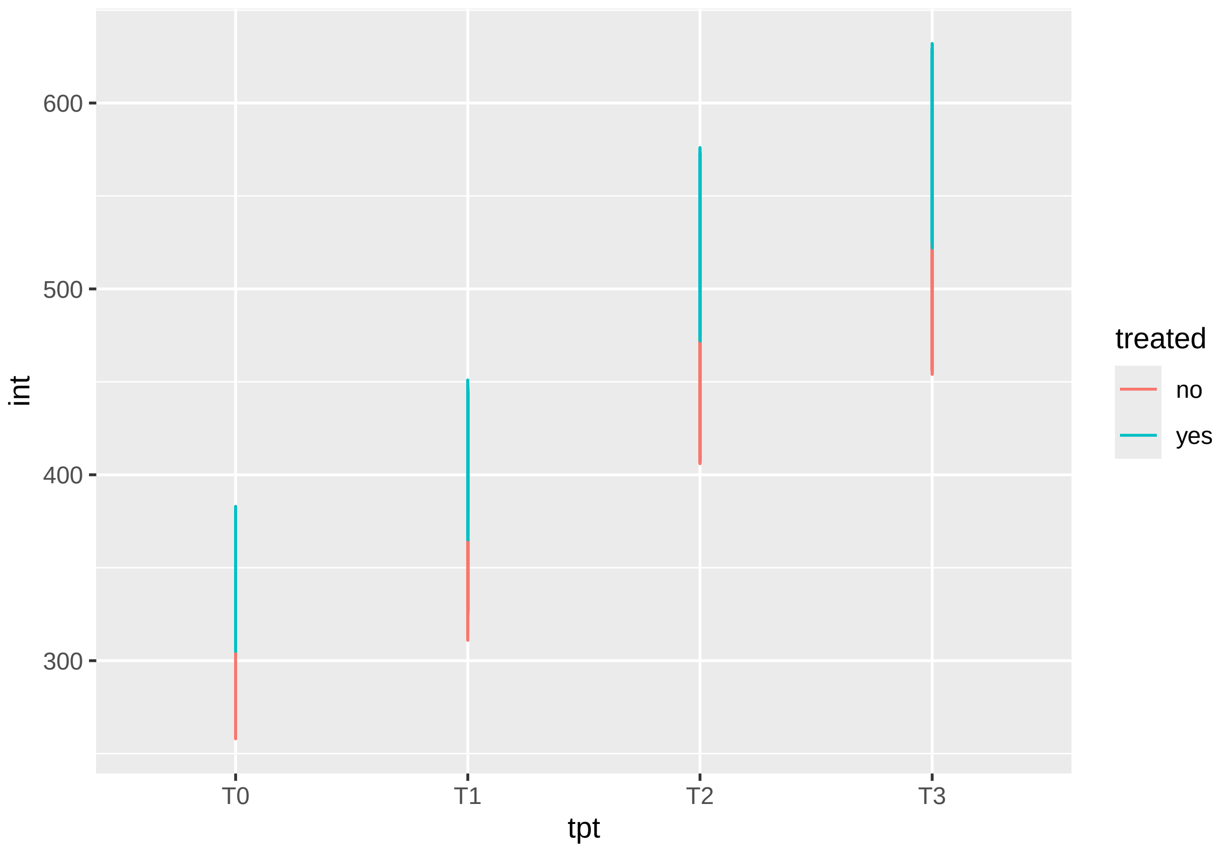
```
> cells_tidy <- separate(cells_tidy, col = channel_tpt,
+   into = c("channel", "tpt"), sep = "_")
> cells_tidy
# A tibble: 1,520 x 9
   Well  treated gene  cellID    roundness cell_area channel tpt     int
   <chr> <chr>   <chr> <chr>         <dbl>     <dbl> <chr>   <chr> <dbl>
 1 A1    no      wt    cell_A1_1      0.89      546. gfp     T0      281
 2 A1    no      wt    cell_A1_1      0.89      546. mCh     T0      224
 3 A1    no      wt    cell_A1_1      0.89      546. gfp     T1      345
 4 A1    no      wt    cell_A1_1      0.89      546. gfp     T2      445
 5 A1    no      wt    cell_A1_1      0.89      546. gfp     T3      496
 6 A1    no      wt    cell_A1_1      0.89      546. mCh     T1      228
 7 A1    no      wt    cell_A1_1      0.89      546. mCh     T2      232
 8 A1    no      wt    cell_A1_1      0.89      546. mCh     T3      236
 9 A1    no      wt    cell_A1_2      0.94      430. gfp     T0      329
10 A1    no      wt    cell_A1_2      0.94      430. mCh     T0      248
# i 1,510 more rows
```

Now we can plot our time series: the x-axis needs to be mapped to the time point, the y-axis to the light intensity. We only visualise "gfp" here.

```
> cells_tidy <- filter(cells_tidy, channel == "gfp")
>
> p <- ggplot(cells_tidy, aes(x = tpt, y = int, colour = treated))
```

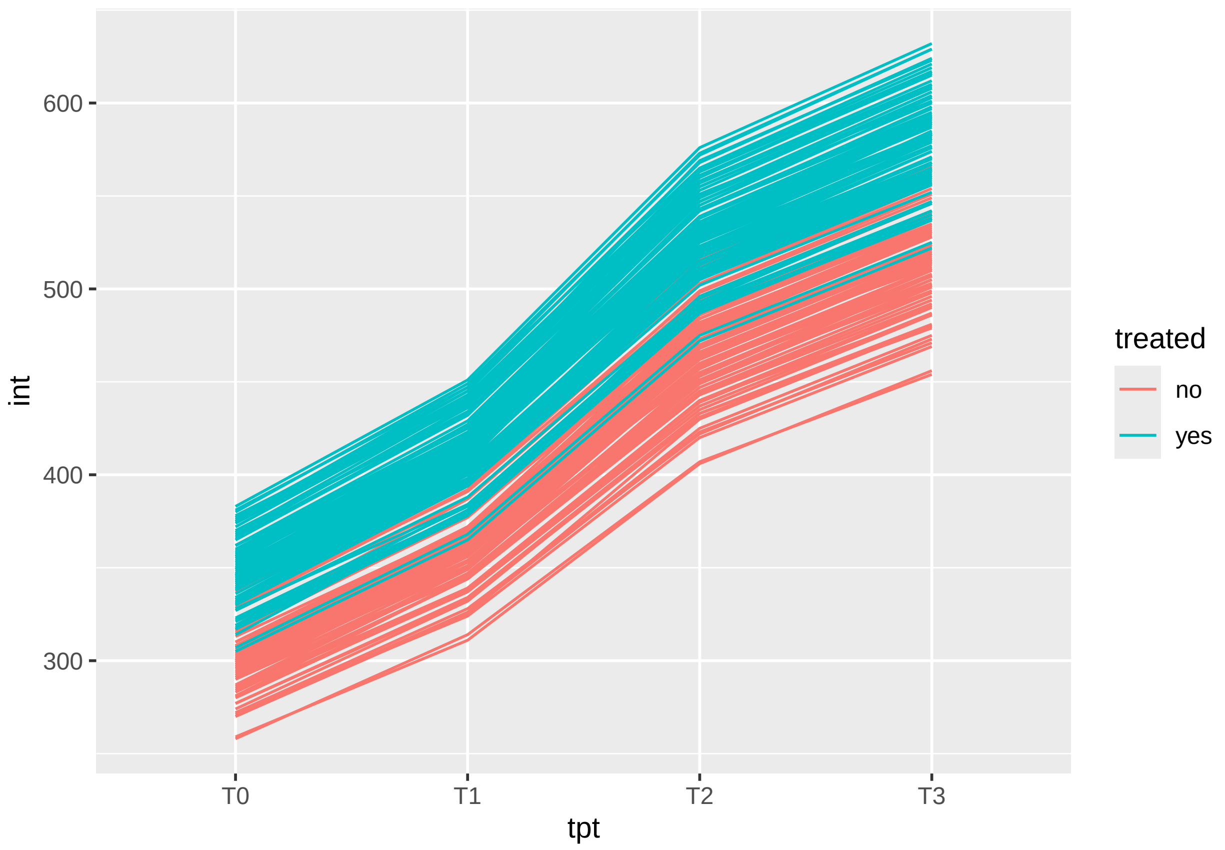
The geom we need is `geom_line()`:

```
> p + geom_line()
```



What happened? We did not tell ggplot which data points it should connect. To do this, we need the `group` aesthetic. In the following command we tell geom_line that it should connect data points if they share a cellID.

```
> p + geom_line(aes(group = cellID))
```



Exercise

Do exercise 5 from the section "Plotting".

The tidyverse

While the R packages that ship with the standard R installation - "base R" - can in principle solve all the data science tasks that you need their usage and behaviour is sometimes not very consistent. Moreover, some problems that tend to be very common when dealing with modern, data-heavy datasets are disappointingly difficult to solve.

For these reasons, Hadley Wickham and other software developers have over the course of the last years written a collection of packages with a shared philosophy of data processing. This set of packages is called the tidyverse. Important tidyverse packages include ggplot2, dplyr, and readr. You can load these packages either separately or all of them at once by running `library(tidyverse)`, which automatically loads all packages that are part of the tidyverse.

An excellent book that shows ways of solving data science challenges with the tidyverse is called R for data science.

We will look at some of the most important tidyverse functions here. You may prefer to use the tidyverse over the base R way of subsetting and modifying data. It is still important to know base R in order to understand the help pages, Stack Overflow posts, Bioconductor code, and other people's code. Importantly, the tidyverse tends to be more difficult to use when programming, e.g. when writing our own functions.

The dplyr package offers an alternative interface for subsetting data

One of the most important tidyverse packages is called "dplyr". It offers a different interface for selecting columns/rows and for creating new columns. It also makes group-wise operations very easy. There is - you guessed it - a dplyr cheat sheet available.

Let us explore the tidyverse using again our `cells` data set. First you need to load the required libraries.

```
> library(tidyverse)
```

To make sure we are on the same page, let's load the `cells` data set again.

```
> (cells <- read_delim("./data/ImagedCells.csv", delim = ";"))
Rows: 190 Columns: 14
-- Column specification --------------------------------------------------------
Delimiter: ";"
chr  (4): Well, treated, gene, cellID
dbl (10): roundness, gfp_T0, mCh_T0, cell_area, gfp_T1, gfp_T2, gfp_T3, mCh_...

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
# A tibble: 190 x 14
   Well  treated gene  cellID    roundness gfp_T0 mCh_T0 cell_area gfp_T1 gfp_T2
   <chr> <chr>   <chr> <chr>         <dbl>  <dbl>  <dbl>     <dbl>  <dbl>  <dbl>
 1 A1    no      wt    cell_A1_1      0.89    281    224      546.    345    445
 2 A1    no      wt    cell_A1_2      0.94    329    248      430.    394    502
 3 A1    no      wt    cell_A1_3      0.85    284    224      445.    339    439
 4 A1    no      wt    cell_A1_4      0.92    331    250      568.    395    506
 5 A1    no      wt    cell_A1_5      0.83    280    220      460.    338    437
 6 A1    no      wt    cell_A1_6      0.91    295    228      486.    353    458
 7 A1    no      wt    cell_A1_7      0.98    332    255      512.    397    511
 8 A1    no      wt    cell_A1_8      0.81    300    244      521.    360    472
 9 A1    no      wt    cell_A1_9      0.83    296    235      497.    353    451
10 A1    no      wt    cell_A1_~      0.88    271    222      558.    326    425
# i 180 more rows
# i 4 more variables: gfp_T3 <dbl>, mCh_T1 <dbl>, mCh_T2 <dbl>, mCh_T3 <dbl>
```

As you can see, the data is presented as a **tibble**. Tibbles are the tidyverse's version of data frames. They tend to be more convenient because they print more nicely to the console and because you can immediately see all the data types contained in a tibble. Apart from that, they behave like data frames in virtually all circumstances.

Selecting columns

We can select which columns we want by putting the name(s) of the column in the `select()` function, separating them by commas.

```
> select(cells, Well, cellID, gfp_T0, mCh_T0)
# A tibble: 190 x 4
   Well  cellID     gfp_T0 mCh_T0
   <chr> <chr>       <dbl>  <dbl>
 1 A1    cell_A1_1     281    224
 2 A1    cell_A1_2     329    248
 3 A1    cell_A1_3     284    224
 4 A1    cell_A1_4     331    250
 5 A1    cell_A1_5     280    220
 6 A1    cell_A1_6     295    228
 7 A1    cell_A1_7     332    255
 8 A1    cell_A1_8     300    244
 9 A1    cell_A1_9     296    235
10 A1    cell_A1_10    271    222
# i 180 more rows
```

IMPORTANT NOTE: When using tidyverse packages you can (must!) often omit the quotes. The function will then look for the respective names in the data set. All tidyverse functions follow the convention: `fun(data, ...)`, i.e. the data set you are working with is the first argument.

The tidyverse offers a rich collection of `selection helpers`. For example, the `starts_with()`, `ends_with()`, and `contains()` functions are useful for using patterns to select sets of columns. Moreover, we can get rid of columns by using the `-` sign.

```
> select(cells, Well, cellID, starts_with('gfp'))
# A tibble: 190 x 6
   Well  cellID     gfp_T0 gfp_T1 gfp_T2 gfp_T3
   <chr> <chr>       <dbl>  <dbl>  <dbl>  <dbl>
 1 A1    cell_A1_1     281    345    445    496
 2 A1    cell_A1_2     329    394    502    554
 3 A1    cell_A1_3     284    339    439    490
 4 A1    cell_A1_4     331    395    506    557
 5 A1    cell_A1_5     280    338    437    487
 6 A1    cell_A1_6     295    353    458    507
 7 A1    cell_A1_7     332    397    511    564
 8 A1    cell_A1_8     300    360    472    521
 9 A1    cell_A1_9     296    353    451    505
10 A1    cell_A1_10    271    326    425    475
# i 180 more rows
```

```
> select(cells, -gfp_T0)
# A tibble: 190 x 13
   Well  treated gene  cellID    roundness mCh_T0 cell_area gfp_T1 gfp_T2 gfp_T3
   <chr> <chr>   <chr> <chr>         <dbl>  <dbl>     <dbl>  <dbl>  <dbl>  <dbl>
 1 A1    no      wt    cell_A1_1      0.89    224      546.    345    445    496
 2 A1    no      wt    cell_A1_2      0.94    248      430.    394    502    554
 3 A1    no      wt    cell_A1_3      0.85    224      445.    339    439    490
 4 A1    no      wt    cell_A1_4      0.92    250      568.    395    506    557
 5 A1    no      wt    cell_A1_5      0.83    220      460.    338    437    487
 6 A1    no      wt    cell_A1_6      0.91    228      486.    353    458    507
 7 A1    no      wt    cell_A1_7      0.98    255      512.    397    511    564
 8 A1    no      wt    cell_A1_8      0.81    244      521.    360    472    521
 9 A1    no      wt    cell_A1_9      0.83    235      497.    353    451    505
10 A1    no      wt    cell_A1_~      0.88    222      558.    326    425    475
# i 180 more rows
# i 3 more variables: mCh_T1 <dbl>, mCh_T2 <dbl>, mCh_T3 <dbl>
```

```
> select(cells, contains('T2'))
# A tibble: 190 x 2
   gfp_T2 mCh_T2
    <dbl>  <dbl>
 1    445    232
 2    502    256
 3    439    232
 4    506    258
 5    437    228
 6    458    236
 7    511    264
 8    472    252
 9    451    243
10    425    230
# i 180 more rows
```

Check the documentation for this and other selection helpers using `?select`.

Filtering rows

If the function `select()` allowed us to pick the columns we're interested in, the `filter()` function lets us choose rows. To keep certain rows, we have to pass to `filter()` a vector of `TRUE` and `FALSE` values, one for each row. The most common way to do that is to use a comparison operator on a column of the data.

```
> filter(cells, gfp_T1 > 350)
# A tibble: 165 x 14
   Well  treated gene  cellID    roundness gfp_T0 mCh_T0 cell_area gfp_T1 gfp_T2
   <chr> <chr>   <chr> <chr>         <dbl>  <dbl>  <dbl>     <dbl>  <dbl>  <dbl>
 1 A1    no      wt    cell_A1_2      0.94    329    248      430.    394    502
 2 A1    no      wt    cell_A1_4      0.92    331    250      568.    395    506
 3 A1    no      wt    cell_A1_6      0.91    295    228      486.    353    458
 4 A1    no      wt    cell_A1_7      0.98    332    255      512.    397    511
 5 A1    no      wt    cell_A1_8      0.81    300    244      521.    360    472
 6 A1    no      wt    cell_A1_9      0.83    296    235      497.    353    451
 7 A1    no      wt    cell_A1_~      0.86    310    240      576.    371    481
 8 A1    no      wt    cell_A1_~      0.94    303    250      454.    361    466
 9 A1    no      wt    cell_A1_~      0.89    300    240      521.    361    468
10 A1    no      wt    cell_A1_~      0.9     303    240      502.    359    464
# i 155 more rows
# i 4 more variables: gfp_T3 <dbl>, mCh_T1 <dbl>, mCh_T2 <dbl>, mCh_T3 <dbl>
```

```
> filter(cells, Well == "A1" & gfp_T1 > 350)
# A tibble: 63 x 14
   Well  treated gene  cellID    roundness gfp_T0 mCh_T0 cell_area gfp_T1 gfp_T2
   <chr> <chr>   <chr> <chr>         <dbl>  <dbl>  <dbl>     <dbl>  <dbl>  <dbl>
 1 A1    no      wt    cell_A1_2      0.94    329    248      430.    394    502
 2 A1    no      wt    cell_A1_4      0.92    331    250      568.    395    506
 3 A1    no      wt    cell_A1_6      0.91    295    228      486.    353    458
 4 A1    no      wt    cell_A1_7      0.98    332    255      512.    397    511
 5 A1    no      wt    cell_A1_8      0.81    300    244      521.    360    472
 6 A1    no      wt    cell_A1_9      0.83    296    235      497.    353    451
 7 A1    no      wt    cell_A1_~      0.86    310    240      576.    371    481
 8 A1    no      wt    cell_A1_~      0.94    303    250      454.    361    466
 9 A1    no      wt    cell_A1_~      0.89    300    240      521.    361    468
10 A1    no      wt    cell_A1_~      0.9     303    240      502.    359    464
# i 53 more rows
# i 4 more variables: gfp_T3 <dbl>, mCh_T1 <dbl>, mCh_T2 <dbl>, mCh_T3 <dbl>
```

Arranging or sorting a tibble

The `arrange()` function lets us sort by one or more columns. By default `arrange()` will sort from smallest to greatest. We can use the function `slice()` to choose the rows to return. In this case `slice(1)` returns the first row, `slice(10:15)` would give you the 10th to 15th rows.

```
> arrange(cells, gfp_T3)
# A tibble: 190 x 14
   Well  treated gene  cellID    roundness gfp_T0 mCh_T0 cell_area gfp_T1 gfp_T2
   <chr> <chr>   <chr> <chr>         <dbl>  <dbl>  <dbl>     <dbl>  <dbl>  <dbl>
 1 A1    no      wt    cell_A1_~      0.96    258    185      531.    314    407
 2 A1    no      wt    cell_A1_~      0.87    259    202      517.    311    406
 3 A1    no      wt    cell_A1_~      0.88    272    212      462.    324    420
 4 A1    no      wt    cell_A1_~      0.86    270    215      427.    326    423
 5 A1    no      wt    cell_A1_~      0.98    272    221      359.    328    422
 6 A1    no      wt    cell_A1_~      0.88    271    222      558.    326    425
 7 A1    no      wt    cell_A1_~      0.95    274    229      471.    332    433
 8 A1    no      wt    cell_A1_~      0.94    270    213      539.    325    430
 9 A1    no      wt    cell_A1_~      0.89    277    217      510.    333    431
10 A1    no      wt    cell_A1_~      0.86    277    211      501.    334    433
# i 180 more rows
# i 4 more variables: gfp_T3 <dbl>, mCh_T1 <dbl>, mCh_T2 <dbl>, mCh_T3 <dbl>
```

```
> # to sort from greatest to smallest use the `desc()` helper function
> arrange(cells, desc(gfp_T3))
# A tibble: 190 x 14
   Well  treated gene  cellID    roundness gfp_T0 mCh_T0 cell_area gfp_T1 gfp_T2
   <chr> <chr>   <chr> <chr>         <dbl>  <dbl>  <dbl>     <dbl>  <dbl>  <dbl>
 1 B1    yes     wt    cell_B1_~      0.76    383    267      523.    451    576
 2 B1    yes     wt    cell_B1_~      0.5     381    261      602.    447    573
 3 B1    yes     wt    cell_B1_~      1.23    380    264      569.    449    572
 4 B1    yes     wt    cell_B1_5      0.54    375    253      467.    445    568
 5 B1    yes     wt    cell_B1_~     -0.52    377    257      554.    445    568
 6 B1    yes     wt    cell_B1_~      2.11    369    240      491.    439    569
 7 B1    yes     wt    cell_B1_~      0.54    376    260      507.    447    565
 8 B1    yes     wt    cell_B1_~      0.64    372    240      559.    440    564
 9 B1    yes     wt    cell_B1_~      1.42    375    250      471.    443    563
10 B1    yes     wt    cell_B1_~      1.5     375    249      583.    442    560
# i 180 more rows
# i 4 more variables: gfp_T3 <dbl>, mCh_T1 <dbl>, mCh_T2 <dbl>, mCh_T3 <dbl>
```

```
> cells_arranged <- arrange(cells, gfp_T3)
> slice(cells_arranged, 1:2)
# A tibble: 2 x 14
  Well  treated gene  cellID     roundness gfp_T0 mCh_T0 cell_area gfp_T1 gfp_T2
  <chr> <chr>   <chr> <chr>          <dbl>  <dbl>  <dbl>     <dbl>  <dbl>  <dbl>
1 A1    no      wt    cell_A1_83      0.96    258    185      531.    314    407
2 A1    no      wt    cell_A1_49      0.87    259    202      517.    311    406
# i 4 more variables: gfp_T3 <dbl>, mCh_T1 <dbl>, mCh_T2 <dbl>, mCh_T3 <dbl>
```

Chaining commands together: meet the pipe operator

Sometimes you want to combine a number of processing steps without saving all the intermediate steps into a variable.

To make this easier, there is an operator to chain together commands. It is called "the pipe" and looks like this: `%>%`.

The basic functionality of the pipe is this: `x %>% f(y)` is equivalent to `f(x, y)`. So the left-hand side of the pipe becomes the first argument of the function to the right-hand side of the pipe. Whenever you see `%>%` you can think of "then do this".

So we could have also written the code above like this:

```
> cells %>%
+     arrange(gfp_T3) %>%
+     slice(1:2)
# A tibble: 2 x 14
  Well  treated gene  cellID      roundness gfp_T0 mCh_T0 cell_area gfp_T1 gfp_T2
  <chr> <chr>   <chr> <chr>           <dbl>  <dbl>  <dbl>     <dbl>  <dbl>  <dbl>
1 A1    no      wt    cell_A1_83       0.96    258    185      531.    314    407
2 A1    no      wt    cell_A1_49       0.87    259    202      517.    311    406
# i 4 more variables: gfp_T3 <dbl>, mCh_T1 <dbl>, mCh_T2 <dbl>, mCh_T3 <dbl>
```

Adding new columns

The `mutate()` function lets us create new columns out of existing columns. Perhaps you would like a column that contains a background subtracted value of `gfp_T0`?

```
> background <- 100
>
> cells %>%
+   mutate(gfp_T0_bgsub = gfp_T0 - background) %>%
+   select(Well, cellID, gfp_T0, gfp_T0_bgsub)
# A tibble: 190 x 4
   Well  cellID     gfp_T0 gfp_T0_bgsub
   <chr> <chr>       <dbl>        <dbl>
 1 A1    cell_A1_1     281          181
 2 A1    cell_A1_2     329          229
 3 A1    cell_A1_3     284          184
 4 A1    cell_A1_4     331          231
 5 A1    cell_A1_5     280          180
 6 A1    cell_A1_6     295          195
 7 A1    cell_A1_7     332          232
 8 A1    cell_A1_8     300          200
 9 A1    cell_A1_9     296          196
10 A1    cell_A1_10    271          171
# i 180 more rows
```

Applying functions group-wise: "split-apply-combine"

Up to now we've seen how we can use **dplyr** functions to 'wrangle' data. We've selected columns and rows of interest, sorted data sets, and created new columns based on existing data.

Using these skills, we can for example retrieve the largest cell for each Well:

```
> cells %>%
+   select(Well, cell_area) %>%
+   filter(Well == "A1") %>%
+   arrange(desc(cell_area)) %>%
+   slice(1)
# A tibble: 1 x 2
  Well  cell_area
  <chr>     <dbl>
1 A1         620.
```

But what if we want to do this for *all* wells, no matter how many? It would be very tedious and error-prone to copy paste the above code for all wells in a plate!

A related problem is if we want to calculate summary statistics for groups, for example the mean `gfp_T0` intensity for each well separately.

Defining sub-groups

We can solve this kind of problem with the "split-apply-combine" pattern of data analysis. Conceptually, first we can *split* the complete data frame into separate data frames, one for each month of the year. Then we can *apply* our logic to get the results we want; in this case, that means sorting the data frame and then getting just the top row with the largest number flight delay. Finally, we *combine* those split apart data frames into a new data frame.

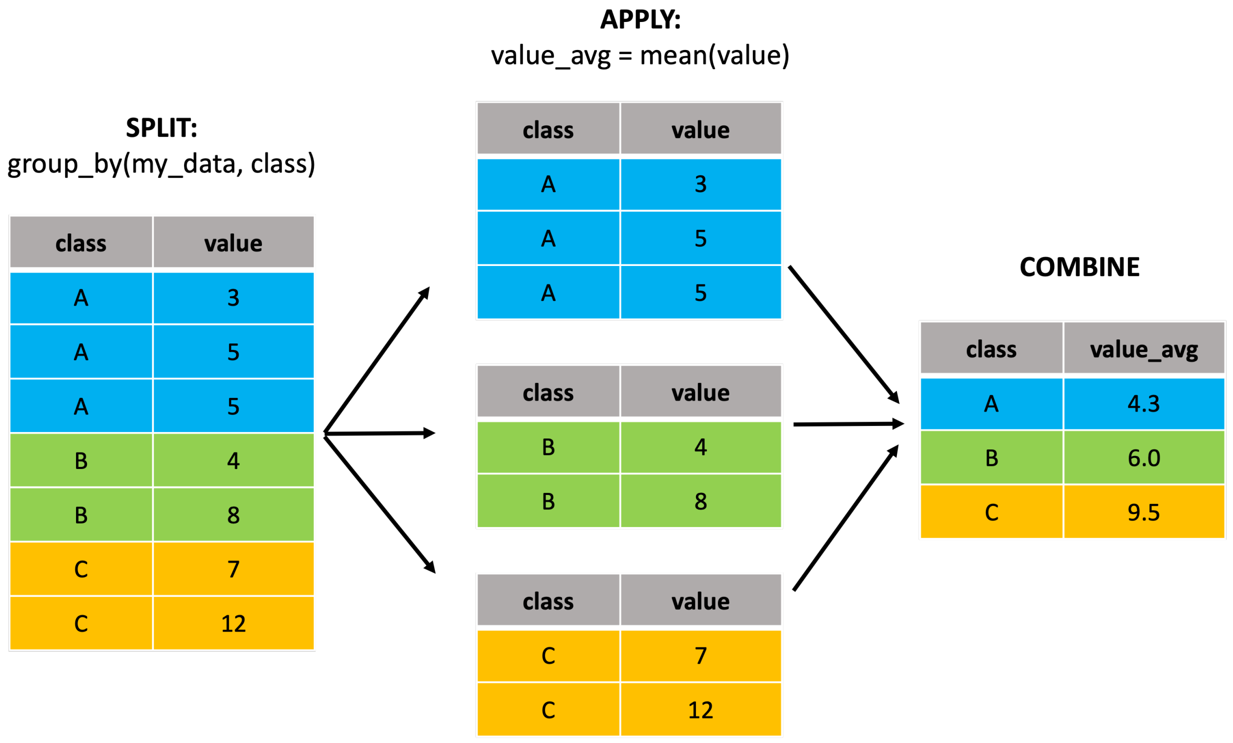


Figure 2: The split-apply-combine pattern for data analysis

To perform the 'splitting' we use the function `group_by()`.

```
> cells %>%
+   select(Well, cell_area) %>%
+   group_by(Well) %>%
+   filter(cell_area == max(cell_area))
# A tibble: 2 x 2
# Groups:   Well [2]
  Well  cell_area
  <chr>     <dbl>
1 A1         620.
2 B1         626.
```

Summarizing or aggregating data (`summarise()`)

The examples above performed the sorting and selecting on our groups using functions in *dplyr*. If we want to perform calculations using other R functions, we can use `summarise()` to apply them across our grouping. The next example will give us the mean delay for each month.

```
> # dplyr will understand both -ise and -ize spellings (summarise and summarize)
> cells %>%
+   group_by(Well) %>%
+   summarise(gfp_T0 = mean(gfp_T0, na.rm = TRUE),
+             mCh_T0 = mean(mCh_T0, na.rm = TRUE))
# A tibble: 2 x 3
  Well  gfp_T0 mCh_T0
  <chr>  <dbl>  <dbl>
1 A1      301.   235.
2 B1      348.   224.
```

Exercise

Do exercise 6 of the section "Plotting"

Avoiding repetition: the `map()` function

When programming it is very common that we want to apply some operation to every item of a list of items. Likewise, sometimes we want to process the same data with different functions.

When faced with such a situation it is tempting to copy and paste code. However, this is usually not a good idea. What if there is a typo or some other mistake in the code you copied? Then you have to go through all lines and correct it manually and chances are you will overlook the mistake in one of the lines. In some programming languages this problem is typically solved with so-called "for loops". In R there is a better solution: `map()` and the `apply()` family of functions.

To illustrate this with an example, let's assume you want to calculate the mean of every column of a data frame.

```
> library(tidyverse)
>
> df <- tibble(
+   col1 = c(1, 3, 7, 2.8),
+   col2 = c(80, 82, 89.7, 60.8),
+   col3 = c(20, 22, 20.3, NA)
+ )
>
> mean_col1 <- mean(df$col1)
> mean_col2 <- mean(df$col2)
> mean_col3 <- mean(df$col3)
```

You can see, this is a lot of typing. In addition, we always have to come up with a variable name. There is an easier way: the `map(x, f)` function applies a function `f` to every element of `x` and returns the result in a list. It is part of the `purrr` package.

```
> map(df, mean)
$col1
[1] 3.45

$col2
[1] 78.125

$col3
[1] NA
```

Remember: under the hood, a data frame is a list with every column being a list element. That's why `map()` applied the function `f()` to every column of our data frame in turn.

The elements in `x` will become the first argument to the function call of `f()`. If you need to provide additional arguments to the function, this can be done after the second argument:

```
> map(df, mean, na.rm = TRUE)
$col1
[1] 3.45

$col2
[1] 78.125

$col3
[1] 20.76667
```

But what if you don't want to get a list back but rather a vector of numbers? This can be achieved by calling `map_dbl()` instead of `map()`.

```
> map_dbl(df, mean, na.rm = TRUE)
    col1     col2     col3 
 3.45000 78.12500 20.76667 
```

What makes this really powerful is that you can provide your own custom functions to `map()`. For example, instead of calculating the mean we could decide to standardise every column, i.e. set the man to 0 and the standard deviation to 1.

```
> map(df, function(x) {
+   x_standardised <- (x - mean(x, na.rm = TRUE)) / sd(x, na.rm = TRUE)
+   return(x_standardised)
+ })
$col1
[1] -0.9676918 -0.1777393  1.4021657 -0.2567346

$col2
[1]  0.1526414  0.3154588  0.9423060 -1.4104062

$col3
[1] -0.7108116  1.1434795 -0.4326679         NA
```

What if you want the result not in a list but put back together in a data frame? You can use `map_dfc()` to do so:

```
> map_dfc(df, function(x) {
+   x_standardised <- (x - mean(x, na.rm = TRUE)) / sd(x, na.rm = TRUE)
+   return(x_standardised)
+ })
# A tibble: 4 x 3
    col1   col2   col3
   <dbl>  <dbl>  <dbl>
1 -0.968  0.153 -0.711
2 -0.178  0.315  1.14 
3  1.40   0.942 -0.433
4 -0.257 -1.41  NA    
```

But it gets even better: what if you don't want to apply one function to all columns but rather different functions to the same column? Again, you could type out all the options.

```
> mean_col1 <- mean(df$col1)
> sd_col1 <- sd(df$col1)
> max_col1 <- max(df$col1)
> min_col1 <- min(df$col1)
```

Again, this can be simplified. Functions can be treated like any other object in R, so we can create a list of functions and then use `map()` to apply every function to our column of interest in turn.

```
> my_functions_of_interest <- list(mean, sd, max, min)
> map(my_functions_of_interest, function(f) {
+   f(df$col1)
+ })
[[1]]
[1] 3.45

[[2]]
[1] 2.531798

[[3]]
[1] 7

[[4]]
[1] 1
```

As you can see, there are a lot of options for using `map()` and it can solve a lot of problems. Check out the `purrr` cheat sheet!

The corresponding functions in base R are `*apply()`, the most important of which is `lapply()`. There is also `sapply()`, `mapply()`, `tapply()`, `vapply()` etc. However, the purrr functions have a more consistent interface. Nevertheless, you will most likely come across the apply family of functions when reading other people's code or base R and Bioconductor documentation.

Reducing code duplication makes code easier to read and maintain. We refer to this as the **DRY principle**: **D**on't **r**epeat **y**ourself.

Exercise

Do exercise 1 from section "Map".

A bit of statistics

Linear regression and testing correlations

Fitting a linear regression

- Linear regressions are a very popular model and also very useful. Just know that R can also do all other kinds of regression you can think of.
- The function for linear regression is `lm(formula, data)` and returns a linear model object.
- `formula` has the syntax `y ~ x`.

```
> cells_wella1 <- cells[cells$Well == "A1", ]
> my_lm <- lm(mCh_T0 ~ gfp_T0, data = cells_wella1)
> my_lm

Call:
lm(formula = mCh_T0 ~ gfp_T0, data = cells_wella1)
```

```
Coefficients:
(Intercept)       gfp_T0
    16.0158       0.7284
```

```
>
> summary(my_lm)

Call:
lm(formula = mCh_T0 ~ gfp_T0, data = cells_wella1)

Residuals:
     Min       1Q   Median       3Q      Max
-18.9395  -2.9328  -0.1059   2.9497  24.3328

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 16.01584   11.24088   1.425    0.158
gfp_T0       0.72839    0.03725  19.552   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 6.274 on 86 degrees of freedom
Multiple R-squared:  0.8164,    Adjusted R-squared:  0.8142
F-statistic: 382.3 on 1 and 86 DF,  p-value: < 2.2e-16
```

Extracting model parameters

- Model objects returned by R will contain every information that you want.
- Such objects are, in effect, somewhat more complicated lists.
- You can therefore extract information directly from the objects using list subsetting.

```
> str(my_lm)
List of 12
 $ coefficients : Named num [1:2] 16.016 0.728
  ..- attr(*, "names")= chr [1:2] "(Intercept)" "gfp_T0"
 $ residuals    : Named num [1:88] 3.308 -7.655 1.122 -7.112 0.036 ...
  ..- attr(*, "names")= chr [1:88] "1" "2" "3" "4" ...
 $ effects      : Named num [1:88] -2208.33 -122.68 0.19 -6.15 -1.06 ...
  ..- attr(*, "names")= chr [1:88] "(Intercept)" "gfp_T0" "" "" ...
 $ rank         : int 2
 $ fitted.values: Named num [1:88] 221 256 223 257 220 ...
  ..- attr(*, "names")= chr [1:88] "1" "2" "3" "4" ...
 $ assign       : int [1:2] 0 1
 $ qr           :List of 5
  ..$ qr   : num [1:88, 1:2] -9.381 0.107 0.107 0.107 0.107 ...
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : chr [1:88] "1" "2" "3" "4" ...
  .. .. ..$ : chr [1:2] "(Intercept)" "gfp_T0"
  .. ..- attr(*, "assign")= int [1:2] 0 1
  ..$ qraux: num [1:2] 1.11 1.18
  ..$ pivot: int [1:2] 1 2
  ..$ tol  : num 1e-07
  ..$ rank : int 2
  ..- attr(*, "class")= chr "qr"
 $ df.residual  : int 86
 $ xlevels      : Named list()
 $ call         : language lm(formula = mCh_T0 ~ gfp_T0, data = cells_wella1)
 $ terms        :Classes 'terms', 'formula'  language mCh_T0 ~ gfp_T0
  .. ..- attr(*, "variables")= language list(mCh_T0, gfp_T0)
  .. ..- attr(*, "factors")= int [1:2, 1] 0 1
  .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. ..$ : chr [1:2] "mCh_T0" "gfp_T0"
  .. .. .. ..$ : chr "gfp_T0"
  .. ..- attr(*, "term.labels")= chr "gfp_T0"
  .. ..- attr(*, "order")= int 1
  .. ..- attr(*, "intercept")= int 1
  .. ..- attr(*, "response")= int 1
  .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
  .. ..- attr(*, "predvars")= language list(mCh_T0, gfp_T0)
  .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
  .. .. ..- attr(*, "names")= chr [1:2] "mCh_T0" "gfp_T0"
 $ model        :'data.frame':  88 obs. of  2 variables:
  ..$ mCh_T0: num [1:88] 224 248 224 250 220 228 255 244 235 222 ...
  ..$ gfp_T0: num [1:88] 281 329 284 331 280 295 332 300 296 271 ...
  ..- attr(*, "terms")=Classes 'terms', 'formula'  language mCh_T0 ~ gfp_T0
  .. .. ..- attr(*, "variables")= language list(mCh_T0, gfp_T0)
  .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
  .. .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. .. ..$ : chr [1:2] "mCh_T0" "gfp_T0"
  .. .. .. .. ..$ : chr "gfp_T0"
  .. .. ..- attr(*, "term.labels")= chr "gfp_T0"
  .. .. ..- attr(*, "order")= int 1
  .. .. ..- attr(*, "intercept")= int 1
  .. .. ..- attr(*, "response")= int 1
  .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
  .. .. ..- attr(*, "predvars")= language list(mCh_T0, gfp_T0)
  .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
  .. .. .. ..- attr(*, "names")= chr [1:2] "mCh_T0" "gfp_T0"
 - attr(*, "class")= chr "lm"
```

However, it is good practice to use **accessory functions**, if available.

```
> # accessory functions:
> coef(my_lm)
(Intercept)      gfp_T0
 16.0158390   0.7283863
```

```
> head(residuals(my_lm))
          1           2           3           4           5           6
 3.30762236 -7.65491804  1.12246358 -7.11169056  0.03600861 -2.88978526
```

```
> summary(my_lm)$r.squared
[1] 0.8163509
```

- To test for correlation and its significance directly, use functions `cor()` and `cor.test()`.
- You can specify the type of regression using the `method` argument. It can be one of "pearson", "spearman", or "kendall".

```
> summary(my_lm)$r.squared
[1] 0.8163509
```

```
> cor(x = cells_wella1$gfp_T0, y = cells_wella1$mCh_T0) ^ 2
[1] 0.8163509
```

```
> # cor.test() by default uses Pearson's correlation coefficient
> cor.test(x = cells_wella1$gfp_T0, y = cells_wella1$mCh_T0)

    Pearson's product-moment correlation

data:  cells_wella1$gfp_T0 and cells_wella1$mCh_T0
t = 19.552, df = 86, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.8560799 0.9358648
sample estimates:
      cor
0.9035214
```

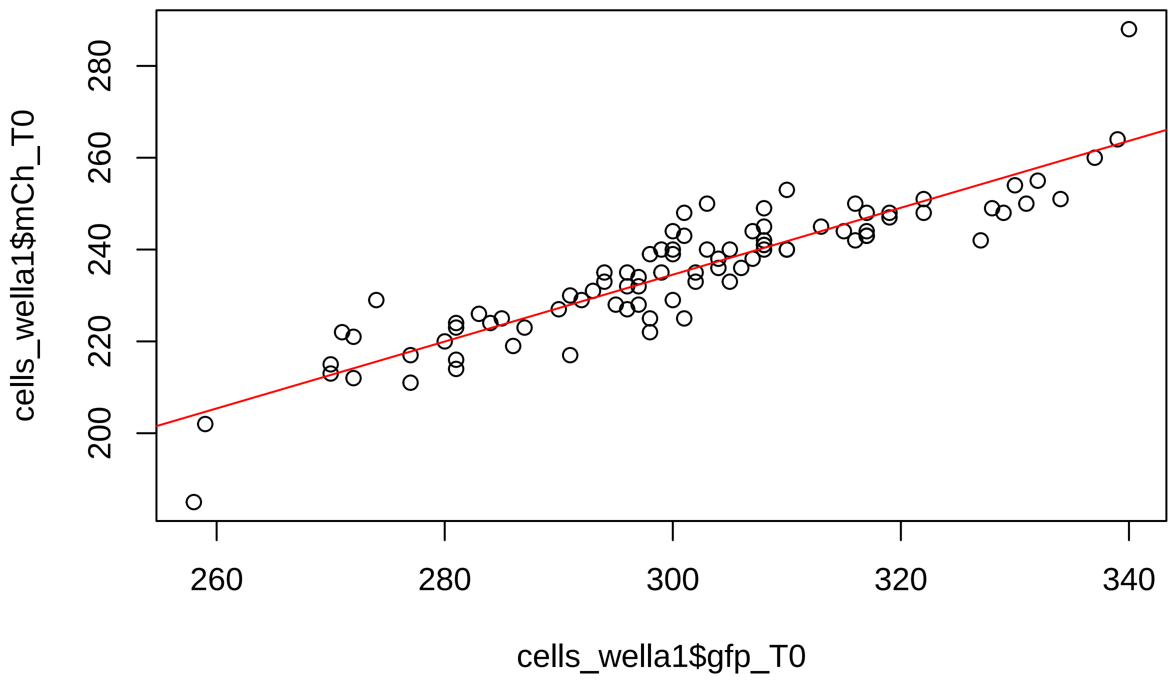
```
>
> cor.test(x = cells_wella1$gfp_T0, y = cells_wella1$mCh_T0,
+          method = "spearman")
Warning in cor.test.default(x = cells_wella1$gfp_T0, y = cells_wella1$mCh_T0, :
Cannot compute exact p-value with ties

    Spearman's rank correlation rho

data:  cells_wella1$gfp_T0 and cells_wella1$mCh_T0
S = 9223.5, p-value < 2.2e-16
alternative hypothesis: true rho is not equal to 0
sample estimates:
      rho
0.9187818
```

Plotting linear regressions

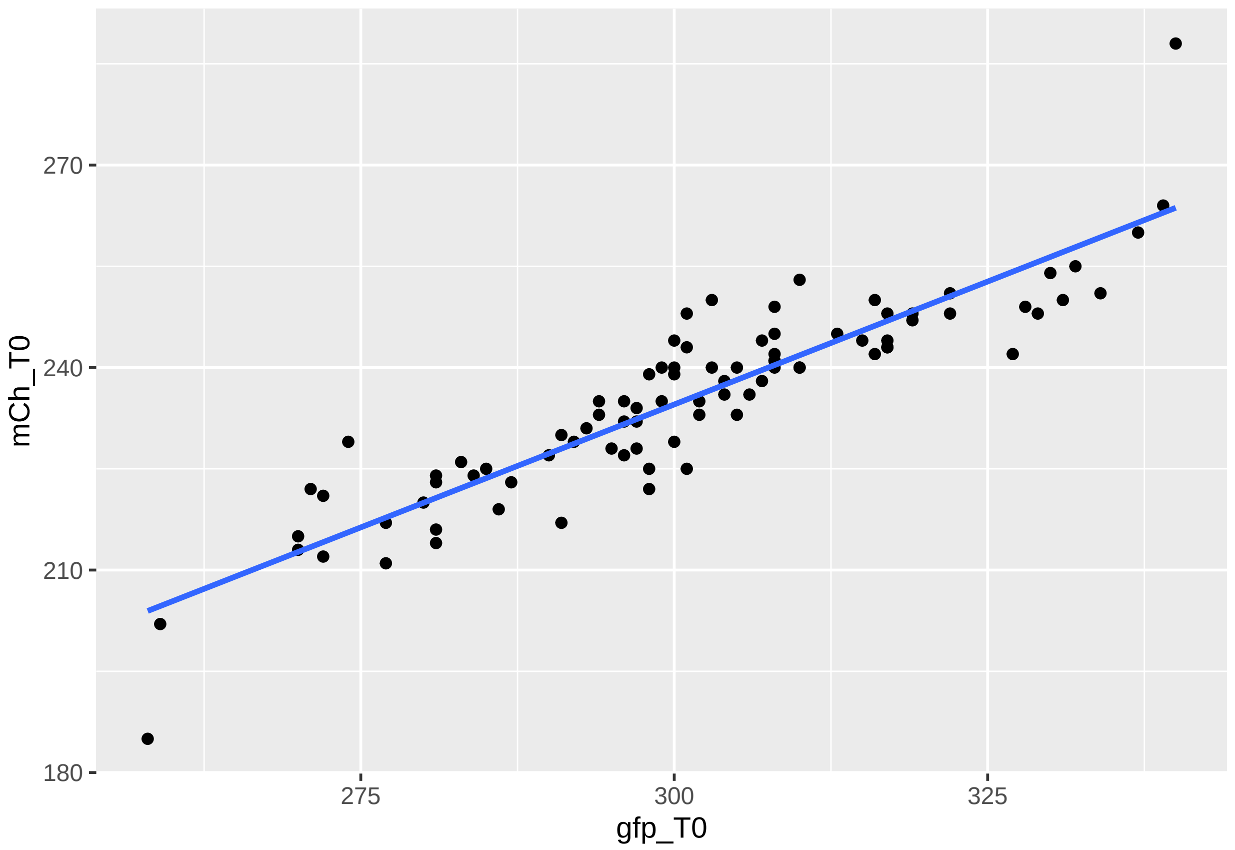
In base plot you add `abline()` to your plot, which accepts a linear model object.

In ggplot, you calculate the linear regression on the fly by adding `geom_smooth(method = "lm")` as a layer. You can decide to display the standard error or not (argument se).

```
> plot(x = cells_wella1$gfp_T0, y = cells_wella1$mCh_T0)
> abline(my_lm, col = "red")
```



```
>
> p <- ggplot(cells_wella1, aes(x = gfp_T0, y = mCh_T0)) +
+   geom_point() +
+   geom_smooth(method = "lm", se = F)
> p
`geom_smooth()` using formula = 'y ~ x'
```



Principles of hypothesis testing: the t-test

When we do empirical science we are interested in comparing groups or in assessing the impact of one variable on the other. Typically, we ask questions such as "Is the average expression of those 2 genes different between the cell types?", "Is growth affected by an increase in the drug concentration?" or, in an omics experiment "Which proteins increase in abundance after knocking out gene X?"

Unfortunately, we cannot simply compare measurements between groups because all measurements are associated with a certain error. Whenever we observe a difference it could be simply due to chance. Estimating when a difference is more likely to be due to chance and when it is more likely to be the result of a real difference is the aim of hypothesis testing.

"Warm-up": distributions of random variables

Random variables can follow lots of different distributions. Important distribution types are, for example, the normal, binomial, Chi-square, Fisher, or the t distribution.

You can always draw random samples from those distributions using the functions `r_$DIST()`, where `$DIST` symbolises the distribution type. So `rnorm(3)`, for example, will return 3 randomly drawn observations from a normal distribution.

```
> rnorm(3)
[1]  1.0131134  0.6973095 -1.6011242
```

```
> rbinom(n = 1, size = 3, prob = 0.5)
[1] 1
```

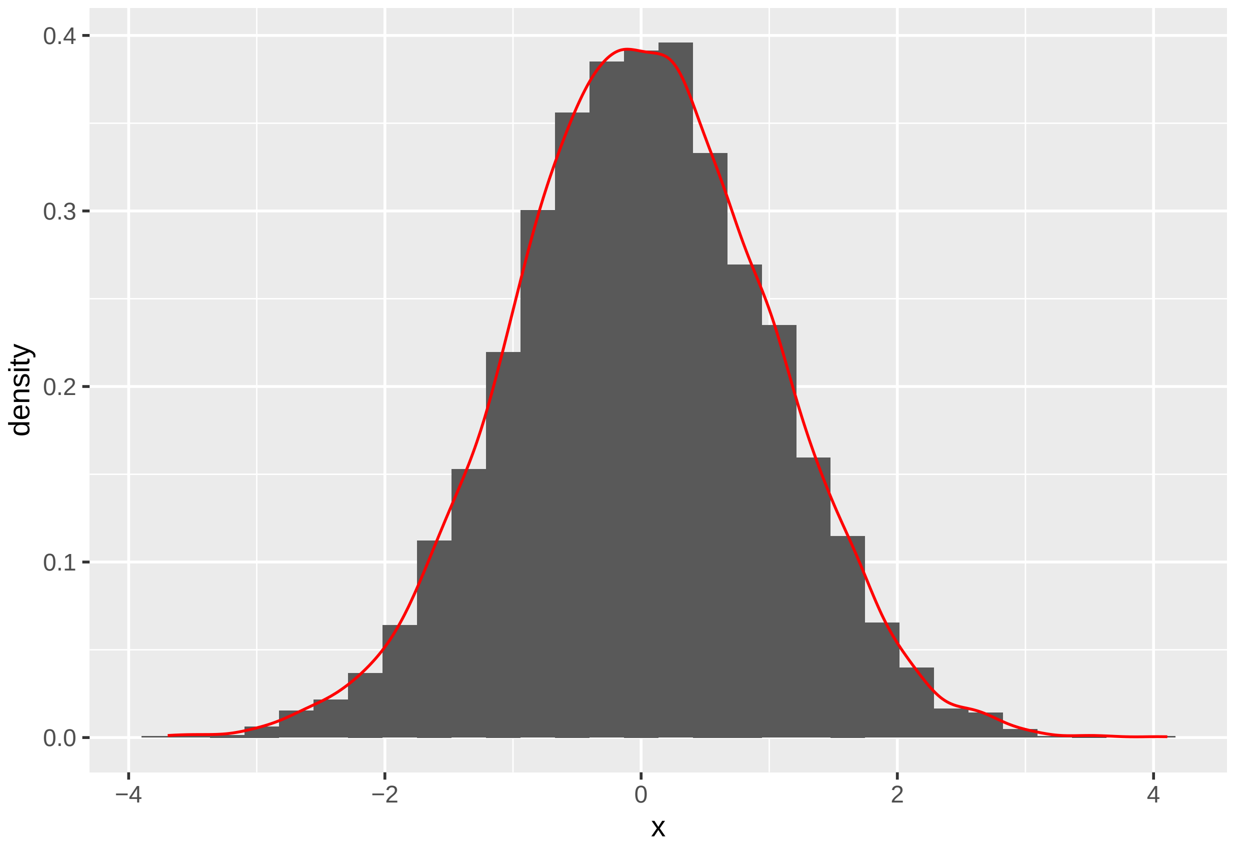
```
> rt(3, df = 2)
[1]  0.6754124 -3.1798893 -0.7984466
```

When we draw a sample that is large enough, the histogram approaches the mathematical shape of the corresponding probability density function. For many distributions we need only one or a few parameters to completely characterise the distribution. For example, the mean and the standard deviation are enough to completely specify a normal distribution. Therefore, those distributions are also called *parametric*.

```
> v <- data.frame(x = rnorm(10000))
> mean(v$x)
[1] -0.002725174
```

```
> sd(v$x)
[1] 1.000845
```

```
>
> ggplot(v, aes(x = x)) +
+   geom_histogram(aes(y = ..density..)) +
+   geom_density(colour = "red")
Warning: The dot-dot notation (`..density..`) was deprecated in ggplot2 3.4.0.
i Please use `after_stat(density)` instead.
This warning is displayed once every 8 hours.
Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
generated.
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



For all the distribution functions mentioned above you can extract the density at a point x using `dnorm()`, (and `dt()`, `dbinom()` etc.).

```
> dnorm(0)
[1] 0.3989423
```

The probability to draw a sample smaller than X (P(X <= x)) is given by `pnorm()`, `pbinom()` etc.

```
> pnorm(0)
[1] 0.5
```

The inverse can be found by using a *quantile function*: `qnorm(p)` returns the quantile ("value") at which the probability to draw a smaller than or equal to this value is `p`.

```
> qnorm(0.5)
[1] 0
```

```
> qnorm(0.975)
[1] 1.959964
```

Hypothesis tests: basic idea

All hypothesis tests share the same basic idea. We first formulate a so-called "null hypothesis" (H0). The null hypothesis is a statement about the distribution of a random variable (e.g. a measurement value). For example, we could say: "My null hypothesis is that my OD values are normally distributed with a mean of 0.5 and a standard deviation of 0.1". Then we perform a measurement and observe a value of, say, 0.6. The probability of this, or a more extreme (i.e. even less probable) value to occur by chance, given that H0 is true, is 0.16. Hence, this event is not that unlikely. But when do we say that an event is so unlikely that we reject H0? This is a matter of convention but typically if the probability is lower than 0.05 or 0.01 (the so-called *significance level alpha*) the convention is to reject H0. In alpha percent of the cases we will do so even though H0 is correct. This type of error is also known as *type I error*.

In the case of a standard normal distribution this means: if a measurement is drawn that is more than 1.96 standard deviations away from 0 then, because this event has a probability less than 5%, we would reject H0 and not believe that it comes from a population with mean = 0 and standard deviation = 1.

Hypothesis testing: the t test

In many measurements we are interested in a question that is very similar to the one above. Often, we are interested if the *mean* of a population is different from, say, 0. Or we want to compare two groups and we want to know if their means are different. In the former case our null hypothesis would be that the mean is 0.

When we calculate the mean from a sample, the resulting mean is again normally distributed. The standard deviation of such a mean is of course smaller than the standard deviation of the distribution it has been calculated from. More precisely its standard deviation is $\sigma/\sqrt(n)$ where σ is the standard deviation of the population.

$\sigma/\sqrt(n)$ is also known as the *standard error of the mean.* It is the standard deviation of the mean. Statisticians call the standard deviation of parameters that we are estimating *standard errors.*

Then the standardised z-score (our "test statistic") is given by

$$Z = \sqrt{n}\frac{\bar{X} - \mu_0}{\sigma}$$

Z is normally distributed and we could check if the likelihood of that z-score to occur is smaller than some significance level alpha.

However, the problem is that we usually do not know the standard deviation of the population. Instead, we have to estimate it using the standard deviation of the sample. The resulting test statistic behaves slightly different from the z-score above. It is now called a *t statistic* and is not normally distributed but it is t-distributed with n-1 degrees of freedom, n being the size of the sample that has been drawn:

$$T = \sqrt{n}\frac{\bar{X} - \mu_0}{S}$$

We can actually show that this is true by simulating some data.

First, we repeatedly “perform 3 measurements” from a random variable with a mean of 0 and standard deviation of 1.

```
> samplesize <- 3
> my_tstats <- replicate(10000, rnorm(samplesize), simplify = FALSE)
> my_tstats <- tibble(sample = my_tstats)
> head(my_tstats)
# A tibble: 6 x 1
  sample
  <list>
1 <dbl [3]>
2 <dbl [3]>
3 <dbl [3]>
4 <dbl [3]>
5 <dbl [3]>
6 <dbl [3]>
```

Next, we calculate for each sample drawn its mean and the standard deviation of the man = standard error of the mean (sem).

```
> my_tstats$mean <- map_dbl(my_tstats$sample, mean)
>
> my_tstats$sem <- map_dbl(my_tstats$sample, function(x) {
+   sd(x) / sqrt(length(x))
+ })
```

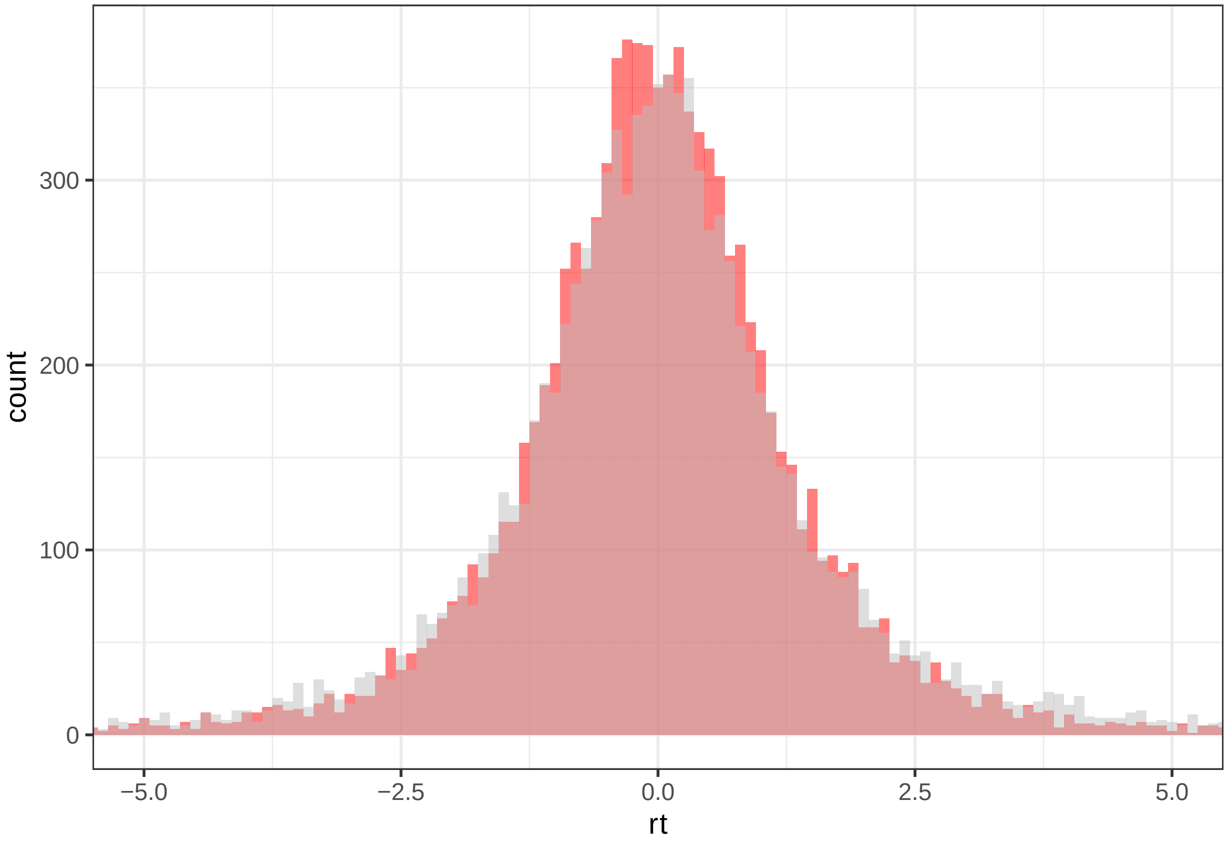
Next, we calculate the t-statistic for each sample. This is like a standardised version of the mean of the sample we have drawn.

```
> get_tstatistic <- function(x) {
+   sqrt(length(x)) * (mean(x)/sd(x))
+ }
>
> my_tstats$tstatistic <- map_dbl(my_tstats$sample, get_tstatistic)
```

Just as a comparison we also make a column containing 10000 randomly drawn observations of a t-distribution with n-1 degrees of freedom. n is our sample size. Remember that the t-statistic we have calculated should be t-distributed. We will check if this is true by plotting both distributions. The numbers sampled from a t-distribution we will plot in red, the t-statistics derived from our samples we will plot in grey.

```
> my_tstats$rt <- rt(10000, df = samplesize)
>
> my_tstats
# A tibble: 10,000 x 5
   sample       mean    sem tstatistic      rt
   <list>      <dbl>  <dbl>      <dbl>   <dbl>
 1 <dbl [3]> -0.236  0.187     -1.27    0.343
 2 <dbl [3]>  0.487  0.408      1.19   -0.397
 3 <dbl [3]>  0.0606 0.837      0.0724  0.737
 4 <dbl [3]> -0.116  0.377     -0.308  -3.57
 5 <dbl [3]> -0.574  0.356     -1.61   -0.0292
 6 <dbl [3]> -0.0444 0.462     -0.0960  0.441
 7 <dbl [3]> -0.0949 0.220     -0.432  -0.376
 8 <dbl [3]>  0.187  0.177      1.05    0.260
 9 <dbl [3]> -0.775  0.156     -4.98   -1.09
10 <dbl [3]>  0.649  0.0799     8.11    0.0692
# i 9,990 more rows
```

```
>
> ggplot(my_tstats) +
+   geom_histogram(aes(x = rt), alpha = 0.5, binwidth = 0.1, fill = "red") +
+   geom_histogram(aes(x = tstatistic), alpha = 0.5, binwidth = 0.1,
+     fill = "grey") +
+   coord_cartesian(xlim = c(-5, 5)) +
+   theme_bw()
```



Confidence intervals

Next, we will explore what the meaning of confidence intervals is. 95% confidence intervals are calculated such that they cover 95% of the data that we would expect to occur by chance. Since our means are t-distributed, we need the 97.5% t-quantile.

```
> t_quantile <- qt(0.975, df = samplesize-1)
```

Using this, we can construct a confidence interval for all samples.

```
> my_tstats$lower <- my_tstats$mean - t_quantile * my_tstats$sem
> my_tstats$upper <- my_tstats$mean + t_quantile * my_tstats$sem
>
```

```
> my_tstats
# A tibble: 10,000 x 7
   sample       mean    sem tstatistic      rt  lower  upper
   <list>      <dbl>  <dbl>      <dbl>   <dbl>  <dbl>  <dbl>
 1 <dbl [3]> -0.236  0.187     -1.27    0.343  -1.04   0.566
 2 <dbl [3]>  0.487  0.408      1.19   -0.397  -1.27   2.24
 3 <dbl [3]>  0.0606 0.837      0.0724  0.737  -3.54   3.66
 4 <dbl [3]> -0.116  0.377     -0.308  -3.57   -1.74   1.51
 5 <dbl [3]> -0.574  0.356     -1.61   -0.0292 -2.11   0.958
 6 <dbl [3]> -0.0444 0.462     -0.0960  0.441  -2.03   1.94
 7 <dbl [3]> -0.0949 0.220     -0.432  -0.376  -1.04   0.851
 8 <dbl [3]>  0.187  0.177      1.05    0.260  -0.575  0.948
 9 <dbl [3]> -0.775  0.156     -4.98   -1.09   -1.45  -0.105
10 <dbl [3]>  0.649  0.0799     8.11    0.0692  0.305  0.993
# i 9,990 more rows
```

How many of the confidence intervals contain the true mean, which we know to be 0?

```
> my_tstats$contains_mean <- my_tstats$lower < 0 & my_tstats$upper > 0
> sum(my_tstats$contains_mean) / nrow(my_tstats)
[1] 0.9504
```

So 95% confidence intervals are defined as follows: if we construct 95% confidence intervals an infinite number of times, then the true mean will be contained in 95% of these confidence intervals.

In summary, we asked: given a certain null hypothesis, which we in this case knew to be: a normal distribution with mean = 0 and sd = 1 (the “null”), how will the standardised mean (the t-statistic) derived from a sample size of 3 drawn from the null distribution behave?

This kind of reasoning is common to all hypothesis tests: we make a certain assumption of how our measurement values behave assuming a certain distribution. If the measurement value we get is then very unlikely (i.e. less likely than our “significance level” α to occur given this “null hypothesis”, then we reject the null hypothesis.

License

Some examples from the sections “R basics” and “Subsetting and modifying values from vectors” were adapted from Hadley Wickham’s “Advanced R” book, 1st edition.