Analysis of Next Generation Sequencing Data

Introduction to R

Luce Skrabanek, Friederike Dündar

4 February, 2020

Contents

1 Prologue

What is R? R is a free software environment for statistical computing and graphics. It can effectively analyze large-scale datasets, such as those resulting from high-throughput sequencing experiments. It promotes automated and reproducible analyses of scientific data, creates a wide spectrum of publication quality figures, and has an extensive library of add-on packages to facilitate many complex statistical analyses. Because it is free and ubiquitously available (it runs on Windows, Mac, and Linux computers), your investment in learning R will pay dividends for years to come.

What is RStudio? While R is very powerful, it is essentially a command line program and is thus not the friendliest thing to use. Especially when learning R, a friendlier environment is helpful, and RStudio provides this, giving you things you expect in a modern interface like integrated file editing, syntax highlighting, code completion, smart indentation, tools to manage plots, browse files and directories, visualize object structures, etc.

From your computer, choose the RStudio application. This will start R under the hood for you.

2 Introduction

The Console panel (lower left panel) is where you type commands to be run immediately. When R is waiting for a new command, you will see a prompt character, `>`.

2.1 R as a calculator

R understands all the usual mathematical operators, keeping the usual order of operations, including forcing an order with parentheses (PEMDAS):

```
1 + 2
3 - 4
5 * 6
7 / 8
1 + 2 * 3
(1 + 2) * 3
```

Let's focus on `1 + 2` and its result:

```
1+3
```

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

What is the `[1]` before the correct result? All numbers in R are vectors, and the `[1]` is a hint about how the vector is indexed. To see a long vector of random numbers, type:


```
head(rnorm(100), n = 15)
```

```
##  [1]  1.50452530  0.09800236 -0.81114717  0.12167865  1.27631259
##  [6]  0.19246376 -1.67559784  2.61841914 -1.87682488  0.02221273
## [11] -1.88289342 -0.44458393 -0.54254855  1.85580193 -1.11226544
```

We will learn more about vectors and their indeces in Section 3.1.

2.2 R has variables

Just as in UNIX, it can be really useful to assign values to variables, so they can be referred to later. This is done using the assignment operator (`<-`).

```
us.population <- 3.22e8 # From Wolfram|Alpha
us.area <- 3719000      # From Wolfram|Alpha
us.pop.density <- us.population / us.area
us.pop.density
```

```
## [1] 86.58241
```

```
( us.pop.density <- us.population / us.area )
```

```
## [1] 86.58241
```

1. Once a variable is defined, you will see it show up in the environment panel in RStudio.
2. R will not automatically print out the value of an assigned variable. Type the name of the variable by itself to see it. Alternatively, wrapping the assignment in parentheses executes the assignment and prints the result.
3. Case matters: `US.area` is not the same as `us.area`.
4. Word separation in R is traditionally done with periods, but this is slowly losing favor. Other options include snake_case (separated by underscores) or camelCase (capitalize each new word).

Note that in RStudio, the `Tab` key will attempt to autocomplete the variable or function name that your cursor is currently on.

2.3 R is interpreting your input

R is called a "high-level, interpreted" programming language. What this means is that R takes care of a lot of basic tasks for you, which is different to many other (compiled) programming languages such as C or Fortran.

So, when you type:

```
    i <- 5
```

R will do the following helpful leg-work for you to *interpret* your input in a manner that the computer actually understands[1]:

- That "5.0" is a floating-point number.

[1] Taken from Noam Ross' blog entry.

- That i should store numeric-type data.
- To find a place in memory for to put "5".
- To register i as a pointer to that place in memory.

You don't even have to translate 5.0 into its binary representation!

Since R takes care of all of these details for you, it will take more time to interpret your commands than non-interpreted languages as it needs to figure all of these details out on its own.

2.4 Data types

So far, we have only been dealing with numerical data, but in the real world, data takes many forms. R has several basic data types that you can use.

```
has.diabetes    <- TRUE       # logical  (note case!)
patient.name    <- "Jane Doe" # character
moms.age        <- NA         # represents an unknown ("missing") value
NY.socialite.iq <- NULL       # represents something that doesn't exist
```

R can convert between data types using a series of `as.()` methods.

```
as.numeric(has.diabetes)

## [1] 1

as.character(us.population)

## [1] "3.22e+08"

as.character(moms.age)        # still NA - we still don't know!

## [1] NA
```

2.5 Installing packages

One of the major advantages of using R for data analysis is the active community that surrounds it. There is a rich ecosystem of packages to the base R system. Some of these provide general functionality while others address very specific tasks.

There are two main repositories of R packages: CRAN (Comprehensive R Archive Network) and Bioconductor.

CRAN is the somewhat more traditional package library and it contains libraries covering all sorts of functions for different types of analyses including finance, genetics, high performance computing, machine learning, social sciences, geography etc. CRAN can be accessed from `http://cran.r-project.org/` (where you also download the R software). Follow the "Packages" link to browse the 6000+ packages currently available.

Since we will be using the **ggplot2** package for preparing publication-quality figures, here we will download and install the **tidyverse** package, which includes `ggplot2`, as well as `tidyr` and

`magrittr`. `install.packages` will download a source package (= a collection of R functions, see Section 7) and installs it in a library on your computer (e.g. `/usr/lib/R/library`).

```
install.packages("tidyverse") # package name is given as a string
```

If this is the first time a package is being installed on your computer, R may ask you to select a CRAN mirror. Pick something geographically close by. Note that you only have to install a package once (per version of R).

Depending on how your computer (and R installation) is set up, you may receive a message indicating that the central location for packages is not writable; in this case R will ask if you want to use a personalized collection of packages stored in your home directory.

If you happen to have downloaded a package to your computer, you can install it without R checking the remote repositories:

```
install.packages("newpackage.tar.gz", repos = FALSE)
```

Installing a package does not make it ready for use in your current R session. To do this, use the `library()` function. You need to do this in every session or script that will use functions from this library.

```
# the magrittr package brings a new infix operator: %>%
# that can be thought of like the UNIX pipe operator "|"
library(magrittr)     # package name is an object (not a string)
head(state.area) %>% sum
```

```
## [1] 1071319
```

The `library` command (i) *loads* the functions of a previously installed package from the computer's library and (ii) *attaches* the functions to the user's workspace, i.e. new functions are now available for use (such as `%>%` from the `magrittr` package). If you want to use functions from an installed package without attaching it to your workspace, you can invoke them with the `::` or `:::` operators:

```
# use a specific ggplot2 function without previous library() call
ggplot2::ggplot()
```

The `devtools` package also allows for the installation via different routes although installation from the repositories should always be your first bet as those guarantee that the packages can actually be installed on different platforms.

```
devtools::install_github("thomasp85/patchwork")
```

2.6 Getting help

Much work has gone into making R self-documenting. There are extensive built-in help pages for all R commands, which are accessible with the `help()` function. To see how `sqrt()` works, type:

```
help(sqrt)
?sqrt
```


The help page will show up in the Help section of the RStudio window.

The help pages will often group similar functions together (e.g., the related functions `log()` and `exp()` are found on the same page).

3 Data Structures

R has several different types of data structures and knowing what each is used for and when they are appropriate is fundamental to the efficient use of R.

The base data structures can be organised by their dimensionality (1d, 2d, or nd) and whether they are homogeneous (all contents must be of the same type) or heterogeneous (the contents can be of different types).

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

Vectors are ordered collections of elements, where each of the objects must be of the same data type, but can be any data type.

A **matrix** is a rectangular array, having some number of columns and some number of rows. Matrices can only comprise one data type (if you want multiple data types in a single structure, use a data frame).

Lists are like vectors, but whereas elements in vectors must all be of the same type, a single list can include elements from any data type. Elements in lists can be named. A common use of lists is to combine multiple values into a single variable that can then be passed to, or returned by, a function.

Data frames are similar to matrices, in that they can have multiple rows and multiple columns, but in a data frame, each of the columns can be of a different data type, although within a column, all elements must be of the same data type. You can think of a data frame as being like a list, but instead of each element in the list being just one value, each element corresponds to a complete vector.

3.1 Vectors

We've already seen a vector when we ran the `rnorm()` command. Let's run that again, but this time assigning the result to a variable.

```
x <- rnorm(100)
```

Many commands in R take a vector as input; a feature of R that we will again discuss in Section 6.


```
sum(x)
```

```
## [1] -7.122459
```

```
max(x)
```

```
## [1] 2.785203
```

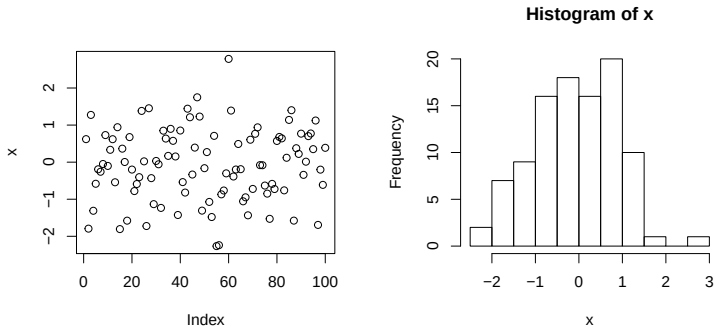
```
summary(x)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -2.26441 -0.73406 -0.06904 -0.07122  0.65005  2.78520
```

```
length(x)
```

```
## [1] 100
```

```
plot(x)
hist(x)
```



3.1.1 Creating vectors

There are many ways of creating vectors. The most common way is using the `c()` function, where `c` stands for concatenation. Here we assign a vector of characters (character strings must be quoted) to a variable `colors`.

```
colors <- c("red", "orange", "yellow", "green",
            "blue", "indigo", "violet")
```

The `c()` function can combine vectors.

```
colors <- c("infrared", colors, "ultraviolet")
# remember that "infrared" and "ultraviolet" are one-element vectors
```

By assigning the result back to the `colors` variable, we are updating its value. The net effect is to both prepend and append new colors to the original `colors` vector.

While `c()` can be used to create vectors of any data type, an easy way to make a numerical vector of sequential numbers is with the ":" operator

```
indices <- 41:50
```


In addition to using the `:` notation to create vectors of sequential numbers, there are a handful of useful functions for generating vectors with systematically created elements. Here we will look at `seq()` and `rep()`.

The `seq()` function can take five different arguments (`from`, `to`, `by`, `length`, `along.with`), although some are mutually exclusive with others. You can pass argments by name rather than position; this is helpful for skipping arguments. Note that all of the arguments have default values, which will be used if you don't specify them.

```
seq(1, 10)      # same as 1:10
seq(1, 4, 0.5) # all numbers from 1 to 4, incrementing by 0.5
seq(0, 1, length.out = 10)
seq(from = 1, to = 4, by = 0.5)
seq(from = 0, to = 1, length.out = 10)
seq(to = 99)
```

Another commonly used function for making regular vectors is `rep()`. This repeats the values in the argument vector as many times as specified. This can be used with character and logical vectors as well as numeric. When using the `length.out` argument, you may not get a full cycle of repetition.

```
rep(colors, 2) # sames as: rep(colors, times = 2)
```

```
##  [1] "infrared"    "red"         "orange"      "yellow"
##  [5] "green"       "blue"        "indigo"      "violet"
##  [9] "ultraviolet" "infrared"    "red"         "orange"
## [13] "yellow"      "green"       "blue"        "indigo"
## [17] "violet"      "ultraviolet"
```

```
rep(colors, each = 2)
```

```
##  [1] "infrared"    "infrared"    "red"         "red"
##  [5] "orange"      "orange"      "yellow"      "yellow"
##  [9] "green"       "green"       "blue"        "blue"
## [13] "indigo"      "indigo"      "violet"      "violet"
## [17] "ultraviolet" "ultraviolet"
```

```
rep(colors, each = 2, times = 2)
```

```
##  [1] "infrared"    "infrared"    "red"         "red"
##  [5] "orange"      "orange"      "yellow"      "yellow"
##  [9] "green"       "green"       "blue"        "blue"
## [13] "indigo"      "indigo"      "violet"      "violet"
## [17] "ultraviolet" "ultraviolet" "infrared"    "infrared"
## [21] "red"         "red"         "orange"      "orange"
## [25] "yellow"      "yellow"      "green"       "green"
## [29] "blue"        "blue"        "indigo"      "indigo"
## [33] "violet"      "violet"      "ultraviolet" "ultraviolet"
```

```
rep(colors, length.out = 10)
```

```
##  [1] "infrared"    "red"         "orange"      "yellow"
```



```
##  [5] "green"       "blue"        "indigo"      "violet"
##  [9] "ultraviolet" "infrared"
```

3.1.2 Accessing and assigning individual vector elements

Individual vector elements are accessed using indexing vectors, which can be numeric, character or logical vectors.

You can access an individual element of a vector by its position (or "index"). In R, the first element has an index of 1.

```
colors[1]
```

```
## [1] "infrared"
```

```
colors[7]
```

```
## [1] "indigo"
```

You can also change the elements of a vector using the same notation as you use to access them.

```
colors[7] <- "purple"
```

You can access multiple elements of a vector by specifying a vector of element indices.

R has many built-in datasets for us to play with. You can view these datasets using the `data()` function. Two examples of vector datasets are `state.name` and `state.area`.

We will get the last ten states (alphabetically) by using the ":" operator, and use that numerical vector to access the elements at those positions.

```
indices <- 41:50
indices[1]
```

```
## [1] 41
```

```
indices[2]
```

```
## [1] 42
```

```
length(indices)
```

```
## [1] 10
```

```
state.name[indices]
```

```
##  [1] "South Dakota"  "Tennessee"     "Texas"         "Utah"
##  [5] "Vermont"       "Virginia"      "Washington"    "West Virginia"
##  [9] "Wisconsin"     "Wyoming"
```

We can test all the elements of a vector at once using logical expressions, generating a logical vector. Let's use this to get a list of small states. First figure out which states are in the bottom quartile, and then compare every element to that number. This returns a vector of logical elements indicating, for every state, whether or not that state is smaller than the cutoff. We use that logical

vector as our indexing vector, returning only those elements which correspond to a TRUE value at that position.

```
summary(state.area)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1214   37317   56222   72368   83234  589757
```

```
cutoff <- 37317
state.area < cutoff
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE
## [12] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE
## [34] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE
## [45]  TRUE FALSE FALSE  TRUE FALSE FALSE
```

```
state.name[state.area < cutoff]
```

```
##  [1] "Connecticut"    "Delaware"       "Hawaii"         "Indiana"
##  [5] "Maine"          "Maryland"       "Massachusetts"  "New Hampshire"
##  [9] "New Jersey"     "Rhode Island"   "South Carolina" "Vermont"
## [13] "West Virginia"
```

We can test for membership in a vector using the `%in%` operator. To see if a state is among the smallest:

```
"New York" %in% state.name[state.area < cutoff]
```

```
## [1] FALSE
```

```
"Rhode Island" %in% state.name[state.area < cutoff]
```

```
## [1] TRUE
```

You can also get the index positions of elements that meet your criteria using the `which()` function.

```
which(state.area > cutoff)
```

```
##  [1]  1  2  3  4  5  6  9 10 12 13 15 16 17 18 22 23 24 25 26 27 28 31
## [23] 32 33 34 35 36 37 38 41 42 43 44 46 47 49 50
```

```
state.name[which(state.area > cutoff)]
```

```
##  [1] "Alabama"        "Alaska"         "Arizona"        "Arkansas"
##  [5] "California"     "Colorado"       "Florida"        "Georgia"
##  [9] "Idaho"          "Illinois"       "Iowa"           "Kansas"
## [13] "Kentucky"       "Louisiana"      "Michigan"       "Minnesota"
## [17] "Mississippi"    "Missouri"       "Montana"        "Nebraska"
## [21] "Nevada"         "New Mexico"     "New York"       "North Carolina"
## [25] "North Dakota"   "Ohio"           "Oklahoma"       "Oregon"
## [29] "Pennsylvania"   "South Dakota"   "Tennessee"      "Texas"
## [33] "Utah"           "Virginia"       "Washington"     "Wisconsin"
## [37] "Wyoming"
```


R also lets us name every element of a vector using the `names()` function, which will allow us to use a character vector to access individual elements directly.

```
names(state.area) <- state.name
state.area["Wyoming"]
```

```
## Wyoming
##   97914
```

```
state.area[c("Wyoming", "Alaska")]
```

```
## Wyoming  Alaska
##   97914  589757
```

R supports sorting, using the `sort()` and `order()` functions.

```
sort(state.area)              # sorts the areas of the states from smallest to largest
order(state.area)             # returns a vector of positions of the sorted elements
state.name[order(state.area)] # sort the state names by state size
state.name[order(state.area, decreasing = TRUE)]
```

We can also randomly sample elements from a vector, using `sample()`.

```
sample(state.name, 4)               # randomly picks four states
sample(state.name)                  # randomly permute the entire vector
sample(state.name, replace = TRUE) # selection with replacement
```

Other miscellaneous useful commands on vectors include:

```
rev(x)    # reverses the vector
sum(x)    # sums all the elements in a numeric or logical vector
cumsum(x) # returns a vector of cumulative sums (or a running total)
diff(x)   # returns a vector of differences between adjacent elements
max(x)    # returns the largest element
min(x)    # returns the smallest element
range(x)  # returns a vector of the smallest and largest elements
mean(x)   # returns the arithmetic mean
```

3.2 Factors

A factor is used to store categorical data, more precisely, it encodes each entry of a vector as an integer. A factor keeps track of all the positions of the distinct values in a given vector. The set of distinct values are called levels. To see (and set) the levels of a factor, you can use the `levels()` function, which will return the levels as a vector.

R has an example factor built in:

```
state.division
```

```
##  [1] East South Central Pacific           Mountain
##  [4] West South Central Pacific           Mountain
##  [7] New England        South Atlantic    South Atlantic
## [10] South Atlantic     Pacific           Mountain
```



```
## [13] East North Central East North Central West North Central
## [16] West North Central East South Central West South Central
## [19] New England        South Atlantic     New England
## [22] East North Central West North Central East South Central
## [25] West North Central Mountain           West North Central
## [28] Mountain           New England        Middle Atlantic
## [31] Mountain           Middle Atlantic    South Atlantic
## [34] West North Central East North Central West South Central
## [37] Pacific            Middle Atlantic    New England
## [40] South Atlantic     West North Central East South Central
## [43] West South Central Mountain           New England
## [46] South Atlantic     Pacific            South Atlantic
## [49] East North Central Mountain
## 9 Levels: New England Middle Atlantic ... Pacific
```

```
levels(state.division)
```

```
## [1] "New England"        "Middle Atlantic"    "South Atlantic"
## [4] "East South Central" "West South Central" "East North Central"
## [7] "West North Central" "Mountain"           "Pacific"
```

To get a hint about how R stores factors (or any other object), we can use the `str()` function to view the structure of that object. You can also use the `class()` function to learn the class of an object, without having to see all the details.

```
str(state.division)
```

```
##  Factor w/ 9 levels "New England",..: 4 9 8 5 9 8 1 3 3 3 ...
```

```
class(state.division)
```

```
## [1] "factor"
```

Note the list of integers corresponds to the level at each position. While factors may behave like character vectors in many ways, they are originally more efficient because they are internally represented as integers and computers are good at working with integers.

You can convert a vector to a factor using the `factor()` function. Here we create a vector using the `sample()` function, where we are storing each color as a character string, and then convert it into a factor.

```
bingo.balls <- sample(colors, size = 40, replace = TRUE)
str(bingo.balls)
bingo.balls.f <- factor(bingo.balls)
str(bingo.balls.f)
```

We will conclude this paragraph with a concise summary by Hadley Wickham:

> While factors look (and often behave) like character vectors, they are actually integers. Be careful when treating them like strings. Some string methods (like gsub() and grepl()) will coerce factors to strings, while others (like nchar()) will throw an error, and still others (like c()) will use the underlying integer values. For this reason, its

usually **best to explicitly convert factors to character vectors if you need string-like behaviour**.

3.3 Matrices and lists

Matrices can be thought of a two-dimensional vectors, where **every element has to be of the same data type**. They are typically created with the `matrix()` function:

```
matrix(data, nrow, ncol, byrow, dimnames)
```

Elements are accessed following the same rules as described above, but given the two-dimensional nature of matrices, you will use 2 indexing vectors: one for rows, and one for columns (`mat[row index, column index]`).

```
testmat <- matrix(c(1:10), nrow = 5) # create a matrix
testmat[3,1] # extract the entry from the third row and first column

## [1] 3

testmat[1,]  # all entries of the 1st row

## [1] 1 6

testmat[,2]  # all entries of the 2nd column

## [1]  6  7  8  9 10

matrix(1:12, nrow = 3)[c(1:3), 2:4] # multiple rows/columns can also be used

##      [,1] [,2] [,3]
## [1,]    4    7   10
## [2,]    5    8   11
## [3,]    6    9   12
```

If both dimensions have been named, you can also use the names for subsetting:

```
head(USPersonalExpenditure)

##                       1940   1945  1950 1955  1960
## Food and Tobacco    22.200 44.500 59.60 73.2 86.80
## Household Operation 10.500 15.500 29.00 36.5 46.20
## Medical and Health   3.530  5.760  9.71 14.0 21.10
## Personal Care        1.040  1.980  2.45  3.4  5.40
## Private Education    0.341  0.974  1.80  2.6  3.64

USPersonalExpenditure["Food and Tobacco", ] # 1D vector result!

## 1940 1945 1950 1955 1960
## 22.2 44.5 59.6 73.2 86.8

USPersonalExpenditure[1:3, c("1940","1950")]

##                       1940  1950
## Food and Tobacco    22.20 59.60
## Household Operation 10.50 29.00
```

```
## Medical and Health   3.53  9.71
```

If your result is one-dimensional, it is by default returned as one-dimensional vector. If you dont want this behavior, you can use `drop=FALSE`:

```
# compare to the output above
USPersonalExpenditure["Food and Tobacco", , drop = FALSE]
```

```
##                   1940 1945 1950 1955 1960
## Food and Tobacco 22.2 44.5 59.6 73.2 86.8
```

Since all elements in a matrix have to be of the same data type, R will automatically change the data type of all entries if a new one of a different data type is entered.

```
str(testmat)
```

```
##  int [1:5, 1:2] 1 2 3 4 5 6 7 8 9 10
```

```
testmat[1,1] <- "X" # add a character
typeof(testmat)
```

```
## [1] "character"
```

```
# notice how the entire matrix is treated the same way as a 1D vector
is.numeric(testmat)
```

```
## [1] FALSE
```

The order of the coercion applied by R is as follows:

```
NULL < raw < logical < integer < double < complex < character < list < expression
```

This holds also true for 1D vectors.

3.4 Lists

Lists are like ragged tables, where every column can be of a different data type, and can have different numbers of elements. Each "column" of a list can be accessed in one of two ways: if the column is not named, we can access a single (!) entry with the double bracket `[[ ]]` notation. If it is named, we can also use a `$` syntax.

```
# assess the structure of a list
str(state.center)
```

```
## List of 2
##  $ x: num [1:50] -86.8 -127.2 -111.6 -92.3 -119.8 ...
##  $ y: num [1:50] 32.6 49.2 34.2 34.7 36.5 ...
```

```
names(state.center)
```

```
## [1] "x" "y"
```

```
# alternative subsetting options for the same list element
state.center$x
state.center[["x"]]
```

```
state.center[[which(names(state.center) == "x")]]

# notice the difference here:
state.center[names(state.center) == "x"] # [ ] returns a list!
# you can also access multiple entries of the list, but only with [ ]
state.center[c(1,2)]
state.center[c("y","x")]
```

An additional important property of lists is that they can be nested, i.e. you can easily construct list of lists:

```
library(magrittr)
list(state_center_coordinates = state.center,
  state_statistics = state.x77) %>% str
```

```
## List of 2
##  $ state_center_coordinates:List of 2
##   ..$ x: num [1:50] -86.8 -127.2 -111.6 -92.3 -119.8 ...
##   ..$ y: num [1:50] 32.6 49.2 34.2 34.7 36.5 ...
##  $ state_statistics        : num [1:50, 1:8] 3615 365 2212 2110 21198 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:50] "Alabama" "Alaska" "Arizona" "Arkansas" ...
##   .. ..$ : chr [1:8] "Population" "Income" "Illiteracy" "Life Exp" ...
```

```
# or just add 'state.x77' to the original list
state.centers <- state.center
state.centers$state_stats <- state.x77
str(state.centers) # notice the difference to the list() example though
```

```
## List of 3
##  $ x          : num [1:50] -86.8 -127.2 -111.6 -92.3 -119.8 ...
##  $ y          : num [1:50] 32.6 49.2 34.2 34.7 36.5 ...
##  $ state_stats: num [1:50, 1:8] 3615 365 2212 2110 21198 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:50] "Alabama" "Alaska" "Arizona" "Arkansas" ...
##   .. ..$ : chr [1:8] "Population" "Income" "Illiteracy" "Life Exp" ...
```

For more information about matrices and lists, see Week 1 of `http://chagall.med.cornell.edu/Rcourse/` and `http://adv-r.had.co.nz/Data-structures.html`.

3.5 Data frames

Data frames are two-dimensional data structures like matrices, but, unlike matrices, they can contain multiple different data types. You can think of a data frame as a list of vectors, where all the vector *lengths* are the same. Data frames are commonly used to represent tabular data such as the data stored in spreadsheets; they are one of the main reasons why statisticians have taken to R since it is tremendously useful to have be able to manipulate different types of data simultaneously.

When we were learning about vectors, we used several parallel vectors, each with length 50 to represent different types of information about US states. The collection of vectors really belongs together, and a data frame is the tool for doing this.

```
state.df <- data.frame(state.name,
  state.abb, state.area, state.center,
  stringsAsFactors = FALSE)
head(state.df)
```

```
##            state.name state.abb state.area         x       y
## Alabama       Alabama        AL      51609  -86.7509 32.5901
## Alaska         Alaska        AK     589757 -127.2500 49.2500
## Arizona       Arizona        AZ     113909 -111.6250 34.2192
## Arkansas     Arkansas        AR      53104  -92.2992 34.7336
## California California        CA     158693 -119.7730 36.5341
## Colorado     Colorado        CO     104247 -105.5130 38.6777
```

The `data.frame()` function combines the four vectors into a single object. There are a couple of noteworthy things that happened here:

- different types of vectors (characters, numeric) were combined
- `state.center` wasn't even a pure vector, but a list with two elements – since the elements were of the same length, they were coerced into separate columns of the data frame (`x, y`)
- `stringsAsFactors = FALSE` argument avoids that the character vectors are parsed as factors (see Section 3.2 of why that's a preferable setting in many cases)

Data frames have a split personality: They behave both like a tagged list of vectors, and like a matrix! This gives you many options for accessing elements.

When accessing a single column, the list notation with the dollar sign is preferred.

```
# different options for subsetting the second column
state.df$state.abb
state.df[[ "state.abb" ]]
state.df[[ 2 ]]
```

When accessing multiple columns or a subset of rows, the matrix notation is used (rows and columns are given indexing vectors, which can be of any type (numeric, character, logical)).

```
state.df[ , 1:2]
state.df[41:50, 1:2]
state.df[c(50, 1), c("state.abb", "x", "y")]
state.df[order(state.df$state.area)[1:5], ]
state.df[order(state.df$state.area), ][1:5, ]
```

The last two examples produce the same output; which is more efficient?

We can give names to both the rows and the columns of a data frame. This makes picking out specific rows less error-prone. Column names are accessed with the `names()` or `colnames()` functions; row names are accessed using `rownames()`.

```
rownames(state.df) <- state.abb
state.df[c("NY", "NJ", "CT", "RI"), c("x", "y")]
```



```
names(state.df) <- c("name", "abb", "area", "long", "lat")
```

Note that if you only fetch data from one column, you'll get a vector back. If you want a one-column data frame, use the `drop = FALSE` option.

You can add a new column the same way you would add one to a list.

```
state.df$division <- state.division # Remember, this is a factor

state.df$z.size <- (state.df$area - mean(state.df$area))/sd(state.df$area)
state.df[ , "z.size", drop = FALSE]
```

3.6 Importing (and exporting) data

In most cases, you won't be typing the data in by hand but rather importing it from spreadsheets or text files. R provides tools for importing a variety of text file formats. If you receive data in Excel format, you'll usually want to save it as tab-delimited or CSV (comma separated values) text[2]. The `read.delim()`, `read.csv()` or `read.table()` functions can then be used to import the data into a data frame.

The `read.table()` function is the most general, giving you exquisite control over how to import your data. One of the defaults of this function is `header = FALSE`. For this reason, we suggest that you always explicitly use the `header` option (you don't want to accidentally miss your first data point).

```
ablation <- read.table("ablation.csv", header = TRUE, sep = ",")
```

You can export a data frame using `write.table()`.

```
write.table(ablation, "my_ablation.txt", quote = FALSE, row.names = FALSE)
```

4 Plotting

Although R has some basic plotting functionality, the **ggplot2** package is more comprehensive and, importantly, consistent, as it is based on a common "grammar"[3]

ggplot2 is written by Hadley Wickham (`http://hadley.nz/`). He maintains a number of other libraries; they are of excellent quality, and are very well documented. However, they are updated frequently, so make sure that you are reading the current documentation. For ggplot2, this can be found at `https://ggplot2.tidyverse.org/reference/`.

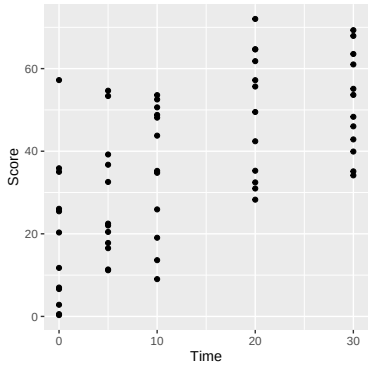
ggplot2 relies entirely on data frames (or Hadley's version of it: `tibbles`) for input.

Let's make our first ggplot with the `ablation` data that we imported earlier.

[2] If you do need to read and write directly to a XLS file, check out the `openxlsx` package.

[3] For an introduction into base R plots, see, e.g., `https://bookdown.org/rdpeng/exdata/the-base-plotting-system-1.html`. For a collection of publication-ready images mostly generated with base R, check out `https://shinyapps.org/apps/RGraphCompendium/index.php`.


```
library(ggplot2) # load the functions of the ggplot2 package into your workspace
ggplot(ablation, aes(x = Time, y = Score)) + geom_point()
```



At a minimum, the two things that you need to give ggplot are:

1. The **dataset** (which must be a data frame or an object that can be interpreted as one), and the variable(s) you want to plot (here: `ablation`)
2. The **type of plot** you want to make (here: a dot plot via `geom_point()`).

This is what the data frame looked like:

```
head(ablation)
```

```
##     Measurement Experiment CellType Direction Time Score
## 1 LDLR-ABLATION      E1909       WT       ABL    0  2.82
## 2 LDLR-ABLATION      E1909       WT       ABL    5 11.37
## 3 LDLR-ABLATION      E1909       WT       ABL   10  9.03
## 4 LDLR-ABLATION      E1909       WT       ABL   20 28.27
## 5 LDLR-ABLATION      E1909       WT       ABL   30 42.86
## 6 LDLR-ABLATION      E1909     A-KD       ABL    0  6.99
```

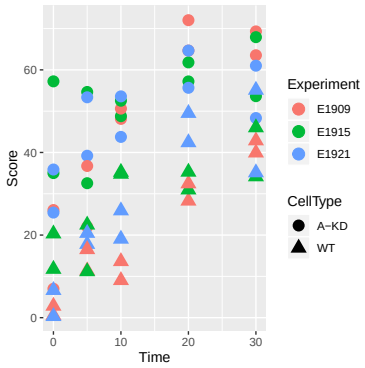
You should notice that the axes are assigned to selected columns of the object just by the name (as if they were objects themselves), not via the usual subsetting routine of character strings.

ggplot gives you ample and fine-grained control over plotting parameters.

```
# Here, we'll change the color and size of the points.
ggplot(ablation, aes(x = Time, y = Score)) + geom_point(color = "red", size = 4)
```

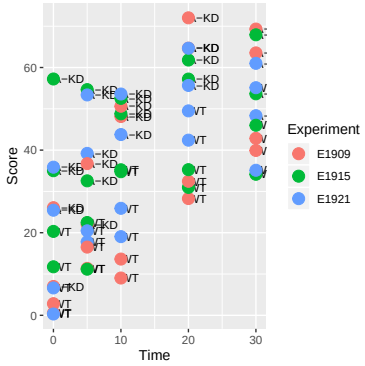
This fixes the color and size for all points. Alternatively, you can use the "aesthetics" parameter to have these properties reflect continuous or discrete values.

```
ggplot(ablation, aes(x = Time, y = Score)) +
  geom_point(aes(color = Experiment, shape = CellType), size = 4)
```

ggplot objects are built up by adding layers. You can add as many layers as you like – the order matters, though, as the first layer may be obstructed by the next layer as shown in the next plot.

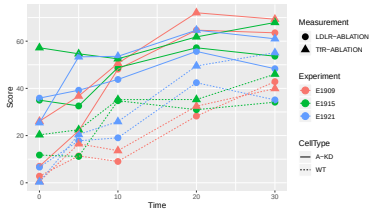
```
ggplot(ablation, aes(x = Time, y = Score)) +
  geom_text(aes(label = CellType), hjust = 0, size = 3) +
  geom_point(aes(color = Experiment), size = 4)
```



It is sometimes useful to save off the base ggplot object and add layers in separate commands. The plot is only rendered when R "prints" the object. This is useful for several reasons:

1. We don't need to create one big huge command to create a plot, we can create it piecemeal.
2. The plot will not get rendered until it has received all of its information, and therefore allows ggplot2 to be more intelligent than R's built-in plotting commands when deciding how large a plot should be, what the best scale is, etc.

```
p <- ggplot(ablation, aes(x = Time, y = Score)) # create the base plot
p <- p + geom_point(aes(color = Experiment, shape = Measurement), size = 4)
# add a line
p <- p +
  geom_line(aes(group = interaction(Experiment, Measurement, CellType),
    color = Experiment,  linetype = CellType))
print(p) # plot gets rendered now
```



Here we've added a layer that plots lines, with a separate line for each unique combination of Experiment, Measurement, and CellType. The `interaction()` function takes a set of factors, and computes a composite factor.

```
library(magrittr)
interaction(ablation$Experiment, ablation$Measurement, ablation$CellType) %>% head
```

```
## [1] E1909.LDLR-ABLATION.WT   E1909.LDLR-ABLATION.WT
## [3] E1909.LDLR-ABLATION.WT   E1909.LDLR-ABLATION.WT
## [5] E1909.LDLR-ABLATION.WT   E1909.LDLR-ABLATION.A-KD
## 12 Levels: E1909.LDLR-ABLATION.A-KD ... E1921.TfR-ABLATION.WT
```

We have also added a new binding to `geom_point()`. The shape of each point is determined by the corresponding Measurement. Note that ggplot prefers six or fewer distinct shapes (i.e., there are no more than six levels in the corresponding factor). You can, however, use more adding a layer like `scale_shape_manual(values = 1:11)`.

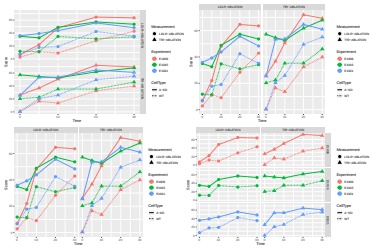
The most commonly used aesthetic types are shown here:

Aestype	
colour	Color of lines and points
fill	Color of area fills (e.g. bar graph)
linetype	Solid/dashed/dotted lines
shape	Shape of points
size	Size of points
alpha	Opacity
x,y	x and y axes

```
# let's jitter the points slightly for better visibility
p + geom_point(aes(color = Experiment, shape = Measurement),
                   size = 4, position = position_dodge(0.5)) +
  scale_shape_manual(values = c(1,16)) # manually specify the shapes
```

This plot is probably showing too much data at once. One approach to resolve this would be to make separate plots for the LDLR and TfR measurements. You can make multiple plots at once using facets. Here are a few options.

```
library(patchwork) # package for easier combined plotting of multiple ggplots
p1 <- p + facet_grid(Measurement ~ .)
p2 <- p + facet_grid(. ~ Measurement)
p3 <- p + facet_grid(Experiment ~ Measurement)
p4 <- p + facet_grid(Measurement ~ Experiment)
(p1 | p2) / (p2 | p3) # patchwork-based combination
```

For more details on using other `scale_()` layers to modify your plots, and all other data types (e.g. `geom_bar`, `geom_hist` etc.) see the **R Graphics Cookbook** (`http://www.cookbook-r.com/Graphs/`) or the online documentation for ggplot (`https://ggplot2.tidyverse.org/reference/`).

5 Data wrangling

You may have noticed that the format of the `ablation` data frame is a bit peculiar. It is, however, in the canonical format for storing and manipulating data that you should be using. The hallmark of this canonical (tidy) format is that there is only one (set of) independently observed value(s) in each row. All of the other columns are identifying values. They explain what exactly was measured, and can be thought of as metadata.

More specifically, a tidy dataset is defined as one where:

- Each variable forms a column.
- Each observation forms a row.

When your data is in this format, it is straightfoward to subset, transform, and aggregate it by any combination of factors of the identifying variables. That is why, for example, the ggplot package essentially requires that your data is in tidy format.

The tidyverse that Hadley Wickham has been instrumental in creating has this format at its core, and his **tidyr** package includes functions to help coerce your data into this format.

5.1 Going long

If you are given data in non-canonical format, you can use the `gather()` function to fix it. This will convert a data frame with several measurement columns (i.e., "fat" or "wide") into a "skinny" or "long" data frame which has one row for every observed (measured) value. The `gather()` function takes multiple columns that all have the same measurement type, and collapses them into key-value pairs, duplicating all other columns as needed.

Let's start with a "fat" data frame that contains data about mouse weights.

```
set.seed(1)
mouse_sim_weights <- data.frame(
  time = seq(as.Date("2017/1/1"), by = "month", length.out = 12),
  Mickey = rnorm(12, 20, 1),  Minnie = rnorm(12, 20, 2),
  Mighty = rnorm(12, 20, 4)
)
head(mouse_sim_weights)
```

```
##         time   Mickey   Minnie   Mighty
## 1 2017-01-01 19.37355 18.75752 22.47930
## 2 2017-02-01 20.18364 15.57060 19.77549
## 3 2017-03-01 19.16437 22.24986 19.37682
## 4 2017-04-01 21.59528 19.91013 14.11699
## 5 2017-05-01 20.32951 19.96762 18.08740
## 6 2017-06-01 19.17953 21.88767 21.67177
```

This dataset consists of only *one type of measurement* – mouse weights – where each column in this dataset represents the weights of a given mouse over a year. The columns 'Mickey', 'Minnie' and 'Mighty' are the names of each mouse, and each of the three columns contain weight data for that respective mouse. This is a format that many spreadsheets will share. The tidy version of this data, however, would have all the weight measurements in one column (the "values") and another column detailing which mouse (or column) that measurement came from (the "keys"). This reformatting can be achieved via the `gather()` function.

```
library(tidyr)
mouse_weights <- gather(data = mouse_sim_weights, # data frame to be manipulated
  key = mouse,    # name of the future column storing the mouse names
  value = weight, # name of the future column storing the weight measurements
  Mickey, Minnie, Mighty) # all the columns that contain the values
head(mouse_weights)
```

```
##         time  mouse   weight
## 1 2017-01-01 Mickey 19.37355
## 2 2017-02-01 Mickey 20.18364
## 3 2017-03-01 Mickey 19.16437
## 4 2017-04-01 Mickey 21.59528
## 5 2017-05-01 Mickey 20.32951
## 6 2017-06-01 Mickey 19.17953
```

```
# this is equivalent
mouse_weights <- gather(data = mouse_sim_weights,
  key = mouse,
  value = weight,
  Mickey:Mighty) # only works if the columns are consecutive, of course
```

After gathering our data, each variable forms a column. Our three variables are `time`, `mouse`, and `weight`. Each row is now an observation. Before tidying our data, each row represented three observations. Note that the arguments to the key and value options become the names of the new columns. Now that the data have been tidied, it is trivial to use as input to ggplot.

```
p1 <- ggplot(mouse_weights, aes(x = mouse, y = weight)) +
  geom_boxplot(aes(fill = mouse))
p2 <- ggplot(mouse_weights, aes(x = time, y = weight)) +
  geom_boxplot(aes(group = time))
p3 <- ggplot(mouse_weights, aes(x = time, y = weight)) +
   geom_boxplot(aes(group = time)) + geom_point(aes(color = mouse))
p4 <- ggplot(mouse_weights, aes(x = time, y = weight)) +
   geom_point(aes(color = mouse)) + geom_line(aes(group = mouse, color = mouse))
```



```
(p1 | p2) / (p3 | p4)
```

5.2 Going wide

The complement of the `gather()` function is the `spread()` function. We can reshape our mouse weights to their original format, or reshape our ablation dataset into a dataframe where there is one row per time point and one column per CellType. For the ablation dataset, note that all of the experimentally measured values in new data frame will come from the original `Score` column (indicated by the `value` option).

```
spread(data = mouse_weights, key = mouse, value = weight)
spread(ablation, key = CellType, value = Score)
```

It is also possible to have columns that are combinations of identifiers, but you will need to include an extra step of manually combining those columns first. Say we wanted a wide table where each of the measurement columns showed the value for a specific combination of Experiment and CellType. We would use another function from the **tidyr** package, `unite()`, `col` is the name of the new column, and we need to supply the columns to paste together. Here, `ExptCell` is the new column that we are defining, as a combination of Experiment and CellType, where the names of the identifiers will be separated by a period.

```
library(magrittr)
abl_united <- unite(ablation, col = ExptCell,
                    Experiment, CellType, sep = ".")
head(abl_united)
```

```
##     Measurement   ExptCell Direction Time Score
## 1 LDLR-ABLATION   E1909.WT       ABL    0  2.82
## 2 LDLR-ABLATION   E1909.WT       ABL    5 11.37
## 3 LDLR-ABLATION   E1909.WT       ABL   10  9.03
## 4 LDLR-ABLATION   E1909.WT       ABL   20 28.27
## 5 LDLR-ABLATION   E1909.WT       ABL   30 42.86
## 6 LDLR-ABLATION E1909.A-KD       ABL    0  6.99
```

```
spread(abl_united, ExptCell, Score) %>% head
```

```
##     Measurement Direction Time E1909.A-KD E1909.WT E1915.A-KD E1915.WT
## 1 LDLR-ABLATION       ABL    0       6.99     2.82      35.01    11.75
## 2 LDLR-ABLATION       ABL    5      22.01    11.37      32.56    11.17
## 3 LDLR-ABLATION       ABL   10      48.13     9.03      48.79    34.77
## 4 LDLR-ABLATION       ABL   20      64.67    28.27      57.20    30.97
```



```
## 5 LDLR-ABLATION       ABL   30      63.54    42.86      53.63    34.15
## 6  TfR-ABLATION       ABL    0      26.06     0.56      57.22    20.32
##   E1921.A-KD E1921.WT
## 1      35.87     6.63
## 2      39.21    17.79
## 3      43.77    19.05
## 4      55.66    42.41
## 5      48.33    35.12
## 6      25.45     0.32
```

Finally, the opposite of the `unite()` function is `separate()`.

```
head(separate(abl_united, ExptCell, c("Expt", "Cell"), sep = "\\."))
```

```
##     Measurement  Expt Cell Direction Time Score
## 1 LDLR-ABLATION E1909   WT       ABL    0  2.82
## 2 LDLR-ABLATION E1909   WT       ABL    5 11.37
## 3 LDLR-ABLATION E1909   WT       ABL   10  9.03
## 4 LDLR-ABLATION E1909   WT       ABL   20 28.27
## 5 LDLR-ABLATION E1909   WT       ABL   30 42.86
## 6 LDLR-ABLATION E1909 A-KD       ABL    0  6.99
```

Note that here, if the separator is a character string, it is interpreted as a regular expression, so we have to escape out the period character. The `separate()` function can be used to split any single column which captures multiple variables.

6 Repeated operations and functions

If you want to perform the same operation multiple times, you would typically turn to a for-loop (remember the scripting exercises for downloading numerous FASTQ files etc.)

For-loops in R have the following syntax:

```
for (value in sequence){
statement
}
```

Consider this simple for-loop:

```
j <- 1
for (i in 1:10) {
    j[i] = i+10
}
```

This is a fairly slow operation because through each round of the for-loop, `j[i]` needs to be allocated, i.e. its size has to be evaluated and an appropriate place in memory must be found – all of this takes time (and remember that R does a lot of interpretation for you!)

You could avoid some of this by calculating how big `j` needs to be to hold all entries.


```
j <- rep(NA, 10) # define a vector of length 10 before the loop
for (i in 1:10) {
    j[i] = i+10
}
```

This is a bit better because every `j[i]` already exists in memory. However, notice how the variable for counting (`i`) is part of your workspace now.

```
ls()
```

```
##  [1] "abl_united"         "ablation"           "colors"
##  [4] "cutoff"             "has.diabetes"       "i"
##  [7] "indices"            "j"                  "moms.age"
## [10] "mouse_sim_weights" "mouse_weights"      "NY.socialite.iq"
## [13] "p"                  "p1"                 "p2"
## [16] "p3"                 "p4"                 "patient.name"
## [19] "seed_value"         "state.area"         "state.centers"
## [22] "state.df"           "testmat"            "us.area"
## [25] "us.pop.density"     "us.population"      "x"
```

Apply family To help you with both the clever memory allocation and to avoid the cluttering of your workspace, R has a couple of in-built functions of the "apply" family. The different apply functions differ in the type of input and output, but all of them enable you to apply a user-defined function to all elements in an object[4].

Function Name	Objects the Function Works On	What the Function Sees as Elements	Result Type
apply	Matrix	Rows or columns	Vector, matrix, array, or list
	Array	Rows, columns, or any dimension	Vector, matrix, array, or list
	Data frame	Rows or columns	Vector, matrix, array, or list
sapply	Vector	Elements	Vector, matrix, or list
	Data frame	Variables	Vector, matrix, or list
	List	Elements	Vector, matrix, or list
lapply	Vector	Elements	List
	Data frame	Variables	List
	List	Elements	List

https://www.dummies.com/programming/r/how-to-use-the-apply-family-of-functions-in-r/

The generally accepted reasons for using these functions rather than for-loops are:

[4]For more details on these functions, see, for example, the data camp tutorial: `https://www.datacamp.com/community/tutorials/r-tutorial-apply-family`.

- They require less code to write.
- They do not need an iteration counter.
- They do not store intermediate results.

```
library(magrittr)
# The basic function is lapply, which returns a list
lapply(1:10, function(z)  z + 10) %>% unlist
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

Under the hood, `lapply` still uses a for-loop on each element of our vector defined by `1:10`, but notice how `z` is not stored in our workspace.

```
ls()
```

```
##  [1] "abl_united"        "ablation"          "colors"
##  [4] "cutoff"            "has.diabetes"      "i"
##  [7] "indices"           "j"                 "moms.age"
## [10] "mouse_sim_weights" "mouse_weights"     "NY.socialite.iq"
## [13] "p"                 "p1"                "p2"
## [16] "p3"                "p4"                "patient.name"
## [19] "seed_value"        "state.area"        "state.centers"
## [22] "state.df"          "testmat"           "us.area"
## [25] "us.pop.density"    "us.population"     "x"
```

Vectorization However, if you find yourself writing a for-loop (even in the disguise of an apply function), you should ask yourself whether there is a way for you to make use of the fact that R is most efficient when performing *vectorized* computations where you supply all of the elements of a vector simultaneously, rather than as sequential, individual components. Our for-loop can easily be solved like this in R:

```
1:10 + 10
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

As you can see, R computet the sum of 10 and each of the vector elements defined by `1:10` without us explicitly stating that it should do the addition for every individual element. The same principle underlies many other seemingy trivial functions such as `sum()`, `log`, `mean` etc. (see Section 3.1 for more examples). All of these accept entire vectors as input, which is typically the fastest way to compute anything in R[5]. The shorter your R code, the faster it usually is! (This is not the case in all other languages.)

Remember that matrices are treated as vectors, too!

```
testmat <- matrix(c(1:10), nrow = 5)
sum(testmat) == sum(1:10)
```

```
## [1] TRUE
```

[5]Under the hood, there's still plenty of for-looping because that is how computers work. But those loops are often implemented in a more efficient language, such as C or Fortran.


```
length(testmat)
```

```
## [1] 10
```

In addition, there are vectorized functions that will work on individual dimensions of the matrix: `rowSums`, `colSums`, `rowMeans`, `colMeans`. The `genefilter` package has even more implemented: `rowSds`, `colSds`, `rowVars`, `colVars`, `rowttests`, `rowFtests`.

```
# example: there will be as many sums as there are rows in testmat
rowSums(testmat)
```

```
## [1]  7  9 11 13 15
```

6.1 Functions

Functions can take different forms in R. The most common type will encompass a function name and some variables/parameters such as `mean()`[6]. In addition, it is often helpful that users define their own functions once they have arrived at a workflow for their data analyses that they are satisfied with.

Following the description by `https://www.tutorialspoint.com/r/r_functions.htm`, the general make-up of a function is as follows:

```
function_name <- function(arg_1, arg_2, ...) {
   Function body
}
```

with:

- Function Name – This is the actual name of the function (here: `function_name`).
- Arguments – An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Arguments can also have default values.
- Function Body - The body of the function can be arbitrarily long and complex (preferably: short and simple). It simply contains a collection of statements and operations that define what the function does. Within the body of a function, all of the input arguments become variables that you can use, including passing them to other functions.
- Return Value - The return value of a function is the last expression in the function body to be evaluated. It is good practice to explicitly use the `return()` function to specify the output though.

```
mySummary <- function(x) {
  my.mean <- mean(x) # making use of other functions
  my.sd <- sd(x)
  out <- list(mean = my.mean, sd = my.sd)
  return(out)
}
```

[6]For an overview of all functions of the `base` R installation, see `library(help = "base")`).

```
# after sourcing the code that defines the function it will become part of your
# workspace
mySummary(x = c(1:100))
```

```
## $mean
## [1] 50.5
##
## $sd
## [1] 29.01149
```

Here, our function, called `mySummary`, takes a single argument called `x`. This function assumes that `x` is a numeric vector, and computes the mean and standard deviation of that vector. The function returns a list with two tagged components. The code executed by our function is enclosed in curly braces .

```
# functions with multiple arguments and default value
raiseNumber <- function(x, power = 1) {
  x ^ power
}

raiseNumber(x = 10)
```

```
## [1] 10
```

```
raiseNumber(x = 10, power = 3)
```

```
## [1] 1000
```

6.1.1 Documenting functions

Our examples so far have been fairly simply, but functions can become quite complex fairly quickly. It is therefore useful to get into the habit of documenting your functions, i.e. to keep track of why you wrote a function at a given point in time and what the details of its arguments are.

The `roxygen2` package has dramatically reduced your ability to find excuses not to document functions as it provides a clear, stringent and powerful framework for keeping documentation together with the actual function in R scripts. Here is an example of a function annotated with `roxygen2`-style comments.

```
#' A Cat Function
#'
#' This function allows you to express your love of cats.
#'
#' This function uses a sophisticated algorithm to determine what type
#' of person you are. The function was initially developed by Hilary
#' Parker and slightly modified here.
#'
#' @param love Boolean. Indicate whether you love cats. Default: TRUE
#' @param nlove Integer. Indicate how much you love cats. Default: 10
#'
#' @return Prints a statment.
```

```
#'
#' @keywords cats
#' @examples
#' cat_function(love = FALSE)
#' @export
cat_function <- function(love=TRUE, nlove = 10){
    if(love==TRUE & !hate){
        print(paste("I love cats! They are", nlove, "times better than
           dogs."))
    }
    else {
        print("Something is wrong with me.")
    }
}
```

From Hadley Wickham's R package book:

> Roxygen comments start with a hash symbol followed by a single quote. They come before a function. All the roxygen lines preceding a function are called a block. Each line should be wrapped in the same way as your code, normally at 80 characters.
>
> Blocks are broken up into tags, which look like `@tagName` details. The content of a tag extends from the end of the tag name to the start of the next tag (or the end of the block).

At the bare minimum, you should strive to add at least the first block covering the first line (= title) and a brief description. Briefly noting what each argument for the function does by adding one tag `@param` per argument is also highly recommended as well as `@return` to describe what type of output the user should obtain from the function.

As you can see above, there are numerous tags available to describe almost all aspects of a user-defined object (such as a function).

6.2 Infix operators

Infix operators are short operators for data manipulation, transformation and customization that are, at their core, also functions. You have already made use of several infix operators, such as

```
$, [ ], [[ ]], +, -,<-
```

Operators for logical assessments are also shipped with base R:

```
& # and
| # or
! # not
== # equal
```

Another very useful infix operator is `%in%`, which matches values:

```
## compare this
"a" %in% c("a","b","d","a")

## [1] TRUE
```

```
## to this
"a" == c("a","b","d","a")

## [1]  TRUE FALSE FALSE  TRUE
```

For a more complete list of infix operators and their explanation see `http://applied-r.com/data-infix-operators-in-r/`.

7 Creating your own package

At the minimum, a package bundles together code and documentation, which allows others (including your future self) to easily re-use your code. You might also add (smallish) data and test functions into a package. Even if you do not plan on sharing your code with anyone else, I have to agree with Karl Broman who insists that *"assembling a few R fucntion with a package will make it way easier for you to use them regularly."*

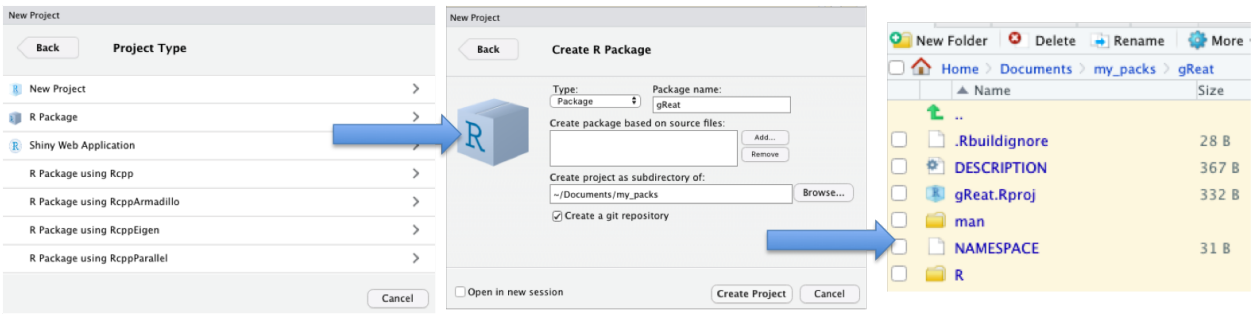
Since R is a highly interactive language, you will usually not immediately start writing a new package once you begin a new type of analysis, but it is good practice to write your functions in a way that will allow to quickly turn them into a package once you find yourself using them repeatedly (see Section 6). All you have to do is to:

- describe each paramater that your function has following the documentation guidelines (see Section 6.1.1)
- avoid giving non-unique names to your functions and objects; try to be reasonably explicit and absolutely avoid overwriting in-built variables such as `t` or `c` within your functions

The `devtools` package along with the `roxygen2` package have made the creation of an R package fairly simple. All you have to type is:

```
devtools::create("gReat")
```

RStudio goes even further as you can initiate a package via "File" ⇒ "New project".



Each R package should have the following components:

- a folder named `R/`: this will contain the scripts with your functions, e.g. `PlottingFunctions.R`, `WranglingFunctions.R`
- a folder named `man/`: this is where the documentation for all of your functions will be kept; each function will have its own `.Rd` file, which contains all the info that users will get when

invoking the generic `help()` function. Thanks to `roxygen2` you won't have to ever generate the files stored here on your own, but the folder has to be there.

- a simple text file named `DESCRIPTION`: describes the dependencies and general information about your package
- a simple text file named `NAMESPACE`: manages the functions that your package depends on; `devtools` and `roxygen2` will again make sure that this is taken care of without you having to manipulate this file manually

Additional folders can contain *optional* components, such as:

- R objects that contain data that will be available after the package installation (kept in `data/` as `.rda` files, with documentation kept in the `R/` folder; see `http://r-pkgs.had.co.nz/data.html` for details). The data stored this way will be available just like its functions – we made use of numerous data sets provided by the base R packages throughout this tutorial (e.g. `state.name`).
- external data (e.g. `.txt` or `.csv` files) that can be imported into R via the usual base R functions such as `read.table()` is kept in `inst/ext_data/`
- additional documentation in the form of short tutorials written in `.Rmd` and kept in `vignettes/`
- if you plan to include functions that are written in other languages than R, you can store the compiled code in `source/`

The `DESCRIPTION` file provides basic informations about the package (see `http://cran.r-project.org/doc/manuals/R-exts.html#The-DESCRIPTION-file`) and has in general the following structure:

```
Package: gReat
Type: Package
Title: Functions For Great Analyses.
Version: 0.1
Date: 2020-02-02
Author: Helmut Kohl
Maintainer: person("Angela", "Merkel", email = "am@example.de",
                   role = c("aut", "cre"))
Description: This packages helps with the analysis of my data.
  It has fantastic functions. Really, the best. They implement
  cool algorithms and enable amazing workflows.
License: Artistic-2.0
LazyLoad: yes # load data from the package when used; see http://r-pkgs
   .had.co.nz/data.html#data
Depends: # Packages that must be present to install this package.
  R (>= 3.1.0) # you can also specify versions
Imports:
  packageA
  packageB
  packageC
Suggests: # packages that are not required to make your package work
  packageE
```

If the package is going to be submitted to Bioconductor, the additional field `biocViews` must includes keywords from the Bioconductor biocViews categories list (`http://wiki.fhcrc.org/bioc/biocViews_categories`).

Difference between `Depends` and `Imports` fields: All the packages listed under `Depends` will have to be installed. In addition, all their functions will be loaded into the interpreter's session environment. This can cause clashes if different packages have functions of the same name. The safer option is to list the needed packages in the `Imports` field, which will not directly attach the functions of those packages, but link them to a package namespace (think `myPackage::myFunction()` style).

7.1 The different states of a package

While you are developing the package, it is helpful to frequently load the functions you've completed into your workspace to test them out. This can be done via

```
devtools::load_all()
```

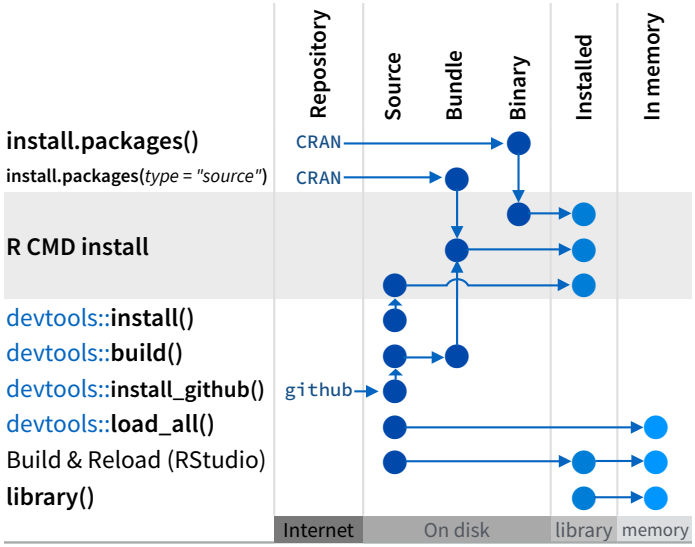
Once you're satisfied with the state of your package, it is time to **build** it, i.e. to bundle all the components up into a tar archive that can be submitted to the common repositories such as CRAN or BioC or that can just be shared with your collaborators.

```
# triad for package building
devtools::load_all()
devtools::document() # this poulates the NAMESPACE file and man/ folder
devtools::build()
```

The resulting `tar.gz` file can then be installed via

```
install.packages("gReat.tar.gz", repos = NULL)
```

The devtools cheat sheet contains an excellent schema to illustrate the different states a package can live in and how the devtools function help with that.



8 References

Karl Broman: R package primer - A minimimal tutorial. `https://kbroman.org/pkg_primer/`

Patrick Burns: The R Inferno (2011). `https://www.burns-stat.com/pages/Tutor/R_inferno.pdf`

Winston Chang: R Graphics Cookbook. `http://www.cookbook-r.com/`

Yen, C.Y., Chang, M.H.W., Chan, C.H. (2019) A Computational Analysis of the Dynamics of R Style Based on 94 Million Lines of Code from All CRAN Packages in the Past 20 Years. Paper presented at the useR! 2019 conference, Toulouse, France. doi:10.31235/osf.io/ts2wq. `https://github.com/chainsawriot/rstyle/blob/master/user2019_poster.pdf`

Bradley J Horn: Applied R Code. Data Science for Immediate Application. `http://applied-r.com/`

Noam Ross: Vectorization in R – why? (2014) `http://www.noamross.net/archives/2014-04-16-vectorization-in-r-why/`

Markus Schröder and Aedìn C. Culhane: Introduction to Sweave and How to Build an R Package (2011). `http://bcb.dfci.harvard.edu/~aedin/courses/ReproducibleResearch/Workshop-reproducible-research.pdf`

Hadley Wickham: R Packages (2015). `http://r-pkgs.had.co.nz/`

The R Project. `https://www.r-project.org/`

See `https://paulvanderlaken.com/2017/08/10/r-resources-cheatsheets-tutorials-books/` for many more R-related resources.

Packages

- `devtools`
- `ggplot2`
- `magrittr`
- `roxygen2`

Also recommended: `data.table`, packages of the tidyverse

