Наука о данных в R для программы Цифровых гуманитарных исследований

Г. А. Мороз, И. С. Поздняков

Оглавление

Глава 1

О курсе

Материалы для курса Наука о данных для магистерской программы Цифровых гуманитарные исследования НИУ ВШЭ.

Глава 2

Введение в R

2.1 Наука о данных

Наука о данных — это новая область знаний, которая активно развивается в последнее время. Она находиться на пересечении компьютерных наук, статистики и математики, и трудно сказать, действительно ли это наука. При этом это движение развивается в самых разных научных направлениях, иногда даже оформляясь в отдельную отрасль:

- биоинформатика
- вычислительная криминалистика
- цифровые гуманитарные исследования
- датажурналистика
- ...

Все больше книг “Data Science for ...”:

- psychologists (Hansjörg, 2019)
- immunologists (Thomas and Pallett, 2019)
- business (Provost and Fawcett, 2013)
- public policy (Brooks and Cooper, 2013)
- fraud detection (Baesens et al., 2015)
- ...

Среди умений датасаентистов можно перечислить следующие:

- сбор и обработка данных
- трансформация данных
- визуализация данных
- статистическое моделирование данных
- представление полученных результатов
- организация всей работы **воспроизводимым способом**

Большинство этих тем в той или иной мере будет представлено в нашем курсе.

2.2 Установка R и RStudio

В данной книге используется исключительно R (R Core Team, 2019), так что для занятий понадобятся:

- R
  - на Windows[1]
  - на Mac[2]
  - на Linux[3], также можно добавить зеркало и установить из командной строки:

```
sudo apt-get install r-cran-base
```

- RStudio — IDE для R (можно скачать здесь[4])
- и некоторые пакеты на R

Часто можно увидеть или услышать, что R — язык программирования для "статистической обработки данных". Изначально это, конечно, было правдой, но уже давно R — это полноценный язык программирования, который при помощи своих пакетов позволяет решать огромный спектр задач. В данной книге используется следующая версия R:

```
## [1] "R version 4.0.2 (2020-06-22)"
```

Некоторые люди не любят устанавливать лишние программы себе на компьютер, несколько вариантов есть и для них:

- RStudio cloud[5] — полная функциональность RStudio, пока бесплатная, но скоро это исправят;
- RStudio on rollApp[6] — облачная среда, позволяющая разворачивать программы.

Первый и вполне закономерный вопрос: зачем мы ставили R и отдельно еще какой-то RStudio? Если опустить незначительные детали, то R — это сам язык программирования, а RStudio — это среда (IDE), которая позволяет в этом языке очень удобно работать.

---

[1] https://cran.r-project.org/bin/windows/base/
[2] https://cran.r-project.org/bin/macosx/
[3] https://cran.rstudio.com/bin/linux/
[4] https://www.rstudio.com/products/rstudio/download/
[5] https://rstudio.cloud/
[6] https://www.rollapp.com/app/rstudio

2.3 Полезные ссылки

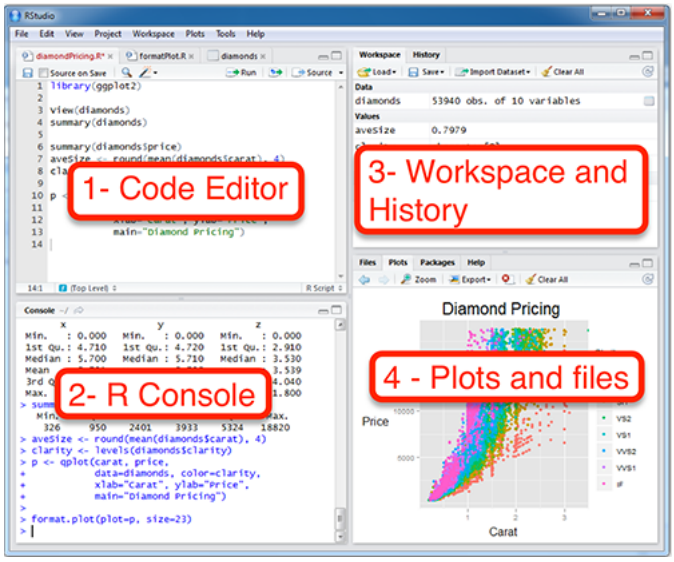
В интернете легко найти документацию и туториалы по самым разным вопросам в R, так что главный залог успеха — грамотно пользоваться поисковиком, и лучше на английском языке.

- книга (Wickham and Grolemund, 2016)[7] является достаточно сильной альтернативой всему курсу
- stackoverflow[8] — сервис, где достаточно быстро отвечают на любые вопросы (не обязательно по R)
- RStudio community[9] — быстро отвечают на вопросы, связанные с R
- русский stackoverflow[10]
- R-bloggers[11] — сайт, где собираются новинки, связанные с R
- чат[12], где можно спрашивать про R на русском (но почитайте правила чата[13], перед тем как спрашивать)
- чат[14] по визуализации данных, чат[15] датажурналистов
- канал про визуализацию[16], дата-блог “Новой газеты”[17], ...

2.4 Rstudio

Когда вы откроете RStudio первый раз, вы увидите три панели: консоль, окружение и историю, а также панель для всего остального. Если ткнуть в консоли на значок уменьшения, то можно открыть дополнительную панель, где можно писать скрипт.

[7] https://r4ds.had.co.nz/
[8] https://stackoverflow.com
[9] https://community.rstudio.com/
[10] https://ru.stackoverflow.com
[11] https://www.r-bloggers.com/
[12] https://t.me/rlang_ru
[13] https://github.com/r-lang-group-ru/group-rules/blob/master/README.md
[14] https://t.me/joinchat/CxZg5goGc6rlWGjcvOYrpA
[15] https://t.me/ddjrus
[16] https://t.me/chartomojka
[17] https://t.me/novaya_data



Существуют разные типы пользователей: одни любят работать в консоли (на картинке это **2** — **R Console**), другие предпочитают скрипты (**1** — **Code Editor**). Консоль позволяет использовать интерактивный режим команда-ответ, а скрипт является по сути текстовым документом, фрагменты которого можно для отладки запускать в консоли.

3 — **Workspace and History**: Здесь можно увидеть переменные. Это поле будет автоматически обновляться по мере того, как Вы будете запускать строчки кода и создавать новые переменные. Еще там есть вкладка с историей последних команд, которые были запущены.

4 — **Plots and files**: Здесь есть очень много всего. Во-первых, небольшой файловый менеджер, во-вторых, там будут появляться графики, когда вы будете их рисовать. Там же есть вкладка с вашими пакетами (Packages) и Help по функциям. Но об этом потом.

2.5 Введение в R

2.5.1 R как калькулятор

Ой-ей, консоль, скрипт че-то все непонятно.
Давайте начнем с самого простого и попробуем использовать R как простой калькулятор. `+`, `-`, `*`, `/`, `^` (степень), `()` и т.д.

Просто запускайте в консоли пока не надоест:

```
40 + 2
```

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

```
3 - 2
```

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

```
5 * 6
```

```
## [1] 30
```

```
99 / 9
```

```
## [1] 11
```

```
2 ^ 3
```

```
## [1] 8
```

```
(2 + 2) * 2
```

```
## [1] 8
```

Ничего сложного, верно? Вводим выражение и получаем результат. Порядок выполнения арифметических операций как в математике, так что не забывайте про скобочки.

> Если Вы не уверены в том, какие операции имеют приоритет, то используйте скобочки, чтобы точно обозначить, в каком порядке нужно производить операции.

2.5.2 Функции

Давайте теперь извлечем корень из какого-нибудь числа. В принципе, тем, кто помнит школьный курс математики, возведения в степень вполне достаточно:

```
16 ^ 0.5
```

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

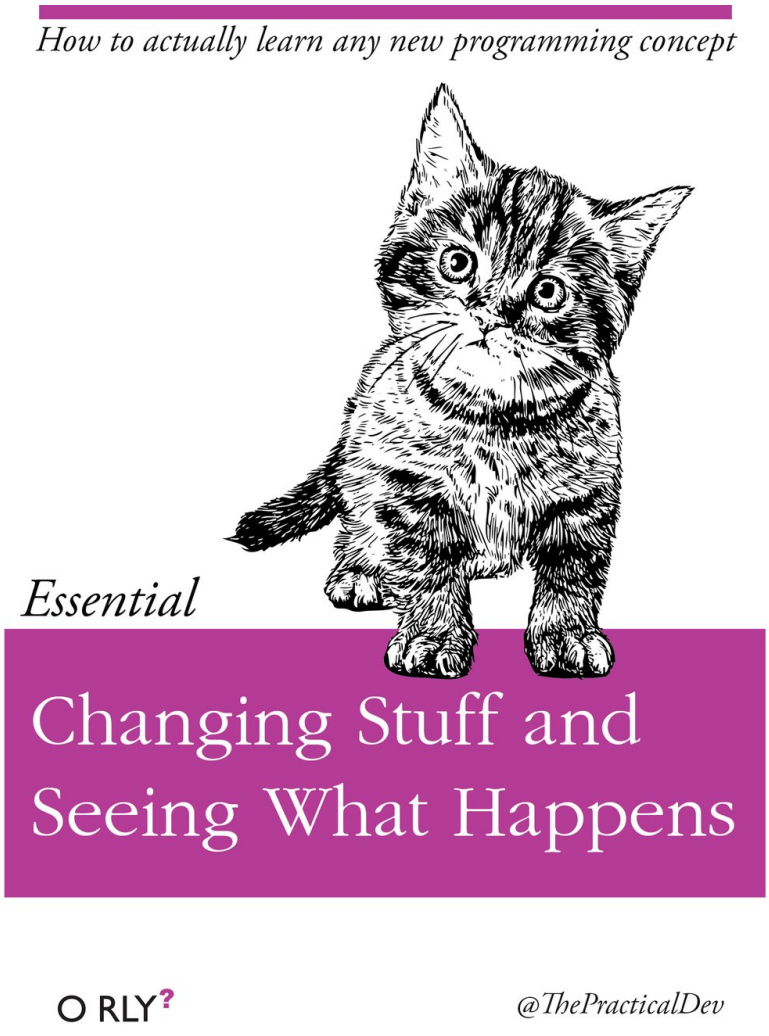

How to actually learn any new programming concept
Essential
Changing Stuff and Seeing What Happens
O RLY?
@ThePracticalDev

Рис. 2.1

Ну а если нет, то можете воспользоваться специальной **функцией**: это обычно какие-то буквенные символы с круглыми скобками сразу после названия функции. Мы подаем на вход (внутрь скобочек) какие-то данные, внутри этих функций происходят какие-то вычисления, которые выдают в ответ какие-то другие данные (или же функция записывает файл, рисует график и т.д.).

Данные на входе называются **аргументом** функции, а иногда — **параметром** функции. В обыденной речи часто говорят **инпут** (калька с английского *input*).

Вот, например, функция для корня:

```
sqrt(16)
```

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

> R — case-sensitive язык, т.е. регистр важен. `SQRT(16)` не будет работать.

А вот так выглядит функция логарифма:

```
log(8)
```

```
## [1] 2.079442
```

Так, вроде бы все нормально, но... Если Вы еще что-то помните из школьной математики, то должны понимать, что что-то здесь не так.

Здесь не хватает основания логарифма!

> Логарифм — показатель степени, в которую надо возвести число, называемое основанием, чтобы получить данное число.

То есть у логарифма 8 по основанию 2 будет значение 3:

$\log_2 8 = 3$

То есть если возвести 2 в степень 3 у нас будет 8:

$2^3 = 8$

Только наша функция считает все как-то не так.

Чтобы понять, что происходит, нам нужно залезть в хэлп этой функции:

```
?log
```

Справа внизу в RStudio появится вот такое окно:

Действительно, у этой функции есть еще аргумент *`base =`*. По умолчанию он равен числу Эйлера (2.7182818...), т.е. функция считает натуральный логарифм. В большинстве функций R есть какой-то основной инпут — данные в том или

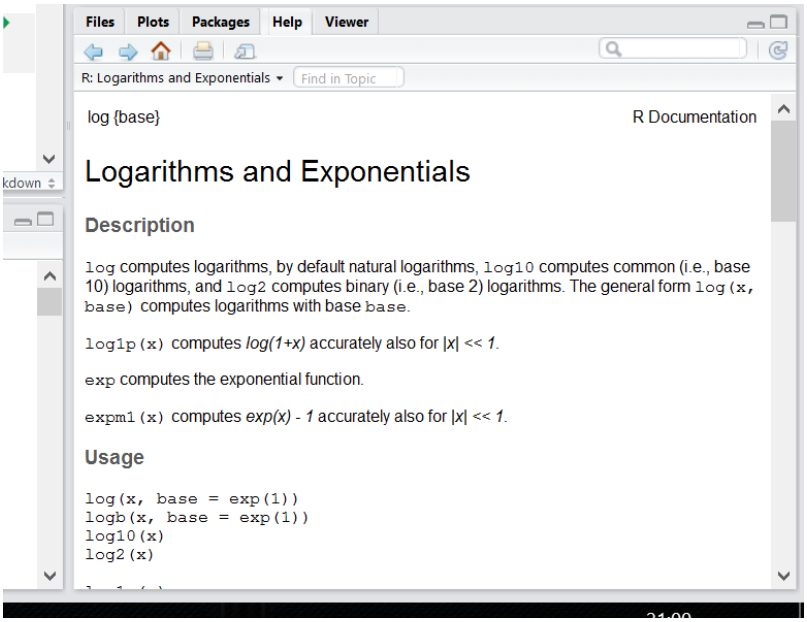


Рис. 2.2

ином формате, а есть и дополнительные параметры, которые можно прописывать вручную, если параметры по умолчанию вас не устраивают.

```
log(x = 8, base = 2)
```

```
## [1] 3
```

...или просто (если Вы уверены в порядке аргументов):

```
log(8, 2)
```

```
## [1] 3
```

Более того, Вы можете использовать результат выполнения одних функций в качестве аргумента для других:

```
log(8, sqrt(4))
```

```
## [1] 3
```

Если эксплицитно писать имена аргументов, то их порядок в функции не важен:

```
log(base = 2, x = 8)
```

```
## [1] 3
```

А еще можно недописывать имена аргументов, если они не совпадают с другими:

```
log(b = 2, x = 8)
```

```
## [1] 3
```

Мы еще много раз будем возвращаться к функциям. Вообще, функции — это одна из важнейших штук в R (примерно так же как и в Python). Мы будем создавать свои функции, использовать функции как инпут для функций и многое-многое другое. В R очень крутые возможности работы с функциями. Поэтому подружитесь с функциями, они клевые.

> Арифметические знаки, которые мы использовали: +,-,/,^ и т.д. называются **операторами** и на самом деле тоже являются функциями:

```
'+'(3,4)
```

```
## [1] 7
```

2.5.3 Переменные

Важная штука в программировании на практически любом языке — возможность сохранять значения в **переменных**. В R это обычно делается с помощью вот этих символов: <- (но можно использовать и обычное =, хотя это не очень принято). Для этого есть удобное сочетание клавиш: нажмите одновременно `Alt` – (или `option` – на Маке).

```
a <- 2
a
```

```
## [1] 2
```

После присвоения переменная появляется во вкладке **Environment** в RStudio:

Можно использовать переменные в функциях и просто вычислениях:

```
b <- a ^ a + a * a
b
```

```
## [1] 8
```

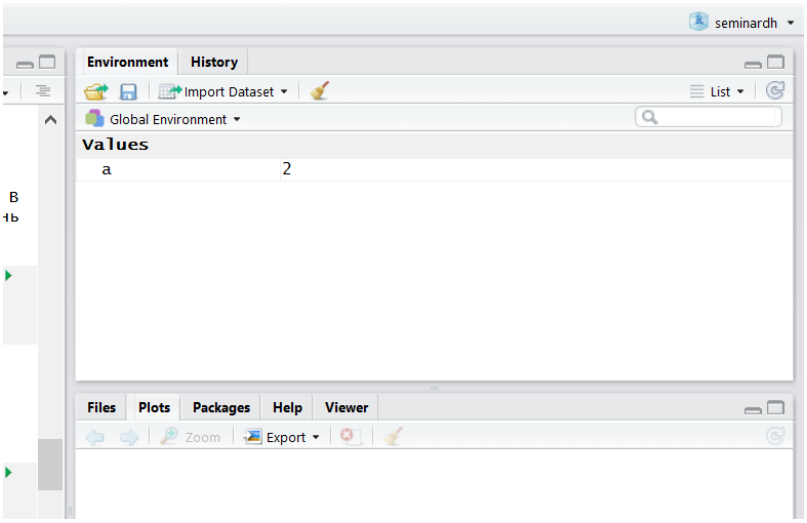


Рис. 2.3

```
log(b, a)
```

```
## [1] 3
```

Вы можете сравнивать разные переменные:

```
a == b
```

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

Заметьте, что сравнивая две переменные мы используем два знака равно ==, а не один =. Иначе это будет означать присвоение.

```
a = b
a
```

```
## [1] 8
```

Теперь Вы сможете понять комикс про восстание роботов на следующей странице (пусть он и совсем про другой язык программирования)

Этот комикс объясняет, как важно не путать присваивание и сравнение (*хотя я иногда путаю до сих пор* =( ).

Иногда нам нужно проверить на *не*равенство:

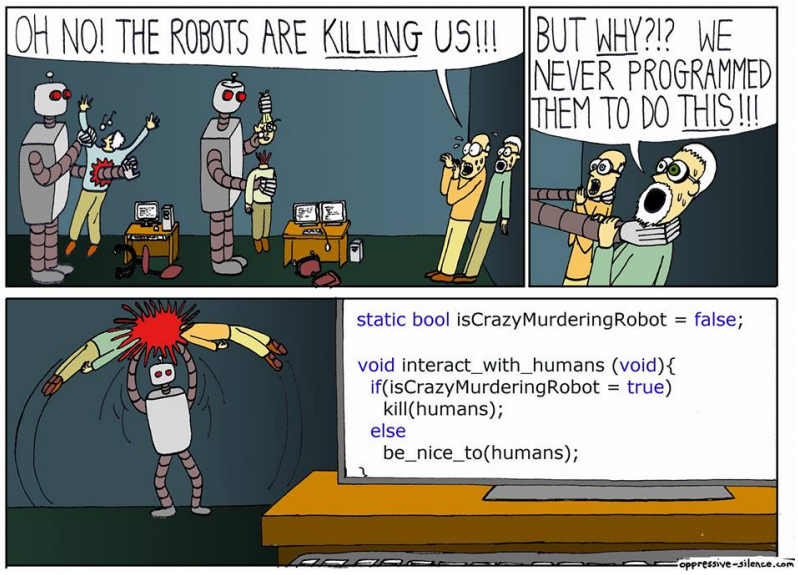


Рис. 2.4

```
a <- 2
b <- 3

a == b
```

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

```
a != b
```

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

Восклицательный язык в программировании вообще и в R в частности стандартно означает отрицание.

Еще мы можем сравнивать на больше/меньше:

```
a>b
```

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

```
a<b
```

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

```
a>=b
```

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

```
a<=b
```

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

2.6 Типы данных

До этого момента мы работали только с числами (numeric):

```
class(a)
```

```
## [1] "numeric"
```

> Вообще, в R много типов numeric: integer (целые), double (с десятичной дробью), complex (комплексные числа). Последние пишутся так: `complexnumber <- 2+2i` Однако в R с этим обычно можно вообще не заморачиваться, R сам будет конвертить между форматами при необходимости. Немного подробностей здесь:

Разница между numeric и integer[18], Как работать с комплексными числами в R[19]

Теперь же нам нужно ознакомиться с двумя другими важными типами данных в R:

1. **character**: строки символов. Они должны выделяться кавычками. Можно использовать как ", так и ' (что удобно, когда строчка внутри уже содержит какие-то кавычки).

```
s <- "       !"
s
```

```
## [1] "       !"
```

```
class(s)
```

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

2. **logical**: просто TRUE или FALSE.

[18] https://stackoverflow.com/questions/23660094/whats-the-difference-between-integer-class-and-numeric-class-in-r

[19] http://www.r-tutor.com/r-introduction/basic-data-types/complex

```
t1 <- TRUE
f1 <- FALSE

t1
```

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

```
f1
```

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

Вообще, можно еще писать `T` и `F` (но не `True` и `False`!)

```
t2 <- T
f2 <- F
```

Это дурная практика, так как R защищает от перезаписи переменные `TRUE` и `FALSE`, но не защищает от этого `T` и `F`

```
TRUE <- FALSE
```

```
## Error in TRUE <- FALSE:         (do_set)
```

```
TRUE
```

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

```
T <- FALSE
T
```

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

Теперь вы можете догадаться, что результаты сравнения, например, числовых или строковых переменных вы можете сохранять в переменные тоже!

```
comparison <- a == b
comparison
```

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

Это нам очень понадобится, когда мы будем работать с реальными данными: нам нужно будет постоянно вытаскивать какие-то данные из датасета, а это как раз и построено на игре со сравнением переменных.

Чтобы этим хорошо уметь пользоваться, нам нужно еще освоить как работать с логическими операторами. Про один мы немного уже говорили — это не (`!`):

```
t1
```

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

```
!t1
```

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

```
!!t1 #          !
```

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

Еще есть И (выдаст `TRUE` только в том случае если обе переменные `TRUE`):

```
t1 & t2
```

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

```
t1 & f1
```

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

А еще ИЛИ (выдаст `TRUE` в случае если хотя бы одна из переменных `TRUE`):

```
t1 | f1
```

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

```
f1 | f2
```

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

Если кому-то вдруг понадобиться другое ИЛИ (строгое ЛИБО) — есть функция `xor()`, принимающая два аргумента.

Поздравляю, мы только что разобрались с самой занудной (хотя и важной) частью. Пора переходить к чему-то более интересному и специфическому для R. Вперед к ВЕКТОРАМ!

2.7 Вектор

Если у вас не было линейной алгебры (или у вас с ней было все плохо), то просто запомните, что **вектор** (или **atomic vector** или **atomic**) — это набор (столбик) чисел в определенном порядке.

Если вы привыкли из школьного курса физики считать вектора стрелочками, то не спешите возмущаться и паниковать. Представьте стрелочки как точки из нуля координат {0,0} до какой-то точки на координатной плоскости, например, {2,3}:

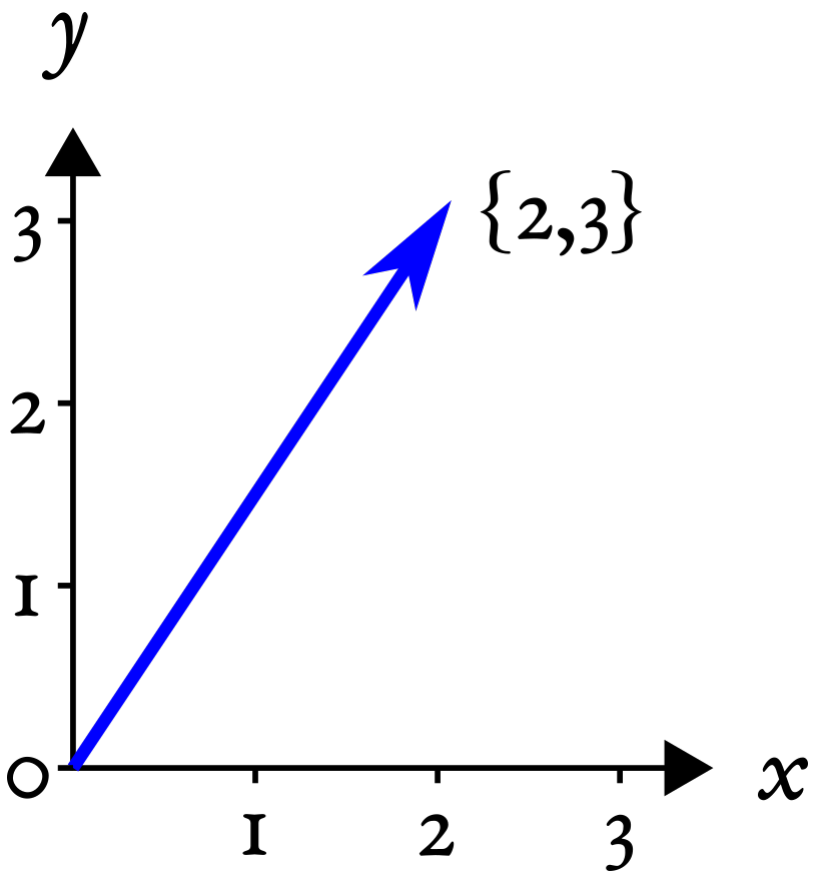


Рис. 2.5

Вот последние два числа и будем считать вектором. Попытайтесь теперь мысленно стереть координатную плоскость и выбросить стрелочки из головы, оставив только последовательность чисел {2,3}:

На самом деле, мы уже работали с векторами в R, но, возможно, Вы об этом даже не догадывались. Дело в том, что в R нет как таковых “значений”, **есть вектора длиной 1**. Такие дела!

Рис. 2.6

Чтобы создать вектор из нескольких значений, нужно воспользоваться функцией *c()*:

```
c(4, 8, 15, 16, 23, 42)
```

```
## [1]  4  8 15 16 23 42
```

```
c("  ", "  ", " ")
```

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

> Одна из самых мерзких и раздражающих причин ошибок в коде — это использование из кириллицы вместо c из латиницы. Видите разницу? И я не вижу. А R видит. И об этом сообщает:

```
(3, 4, 5)
```

```
## Error in (3, 4, 5):                " "
```

Для создания числовых векторов есть удобный **оператор** :

```
1:10
```

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

```
5:-3
```

```
## [1]  5  4  3  2  1  0 -1 -2 -3
```

Этот оператор создает вектор от первого числа до второго с шагом 1. Вы не представляете, как часто эта штука нам пригодится... Если же нужно сделать вектор с другим шагом, то есть функция `seq()`:

```
seq(10,100, by = 10)
```

```
##  [1]  10  20  30  40  50  60  70  80  90 100
```

Кроме того, можно задавать не шаг, а длину вектора. Тогда шаг функция `seq()` посчитает сама:

```
seq(1,13, length.out = 4)
```

```
## [1]  1  5  9 13
```

Другая функция — `rep()` — позволяет создавать вектора с повторяющимися значениями. Первый аргумент — значение, которое нужно повторять, а второй аргумент — сколько раз повторять.

```
rep(1, 5)
```

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

И первый, и второй аргумент могут быть векторами!

```
rep(1:3, 3)
```

```
## [1] 1 2 3 1 2 3 1 2 3
```

```
rep(1:3, 1:3)
```

```
## [1] 1 2 2 3 3 3
```

Еще можно объединять вектора (что мы, по сути, и делали, просто с векторами длиной 1):

```
v1 <- c("Hey", "Ho")
v2 <- c("Let's", "Go!")
c(v1, v2)
```

```
## [1] "Hey"   "Ho"    "Let's" "Go!"
```

2.7.1 Приведение типов

Что будет, если вы объедините два вектора с значениями разных типов? Ошибка?

Мы уже обсуждали, что в *atomic* может быть только один тип данных. В некоторых языках программирования при операции с данными разных типов мы бы получили ошибку. А вот в R при несовпадении типов пройзойдет попытка привести типы к "общему знаменателю", то есть конвертировать данные в более "широкий" тип.

Например:

```
c(FALSE, 2)
```

```
## [1] 0 2
```

`FALSE` превратился в `0` (а `TRUE` превратился бы в `1`), чтобы оба значения можно было объединить в вектор. То же самое произошло бы в случае операций с векторами:

```
2 + TRUE
```

```
## [1] 3
```

Это называется **неявным приведением типов (implicit coercion)**.

Вот более сложный пример:

```
c(TRUE, 3, "   ")
```

```
## [1] "TRUE"   "3"      "    "
```

У R есть иерархия приведения типов:

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

Мы из этого списка еще многого не знаем, сейчас важно запомнить, что логические данные — `TRUE` и `FALSE` — превращаются в `0` и `1` соответственно, а `0` и `1` в строчки `"0"` и `"1"`.

Если Вы боитесь полагаться на приведение типов, то можете воспользоваться функциями `as.` для явного приведения типов (**explicit coercion**):

```
as.numeric(c(T, F, F))
```

```
## [1] 0 0 0
```

```
as.character(as.numeric(c(T, F, F)))
```

```
## [1] "0" "0" "0"
```

Можно превращать и обратно, например, строковые значения в числовые. Если среди числа встретится буква или другой неподходящий знак, то мы получим предупреждение `NA` — пропущенное значение (мы очень скоро научимся с ними работать).

```
as.numeric(c("1", "2", "  "))
```

```
## Warning:                    NA
```

```
## [1]  1  2 NA
```

> Один из распространенных примеров использования неявного приведения типов — использования функций `sum()` и `mean()` для подсчета в логическом векторе количества и доли `TRUE` соответсвенно. Мы будем много раз пользоваться этим приемом в дальнейшем!

2.7.2 Векторизация

Все те арифметические операторы, что мы использовали ранее, можно использовать с векторами одинаковой длины:

```
n <- 1:4
m <- 4:1
n + m
```

```
## [1] 5 5 5 5
```

```
n - m
```

```
## [1] -3 -1  1  3
```

```
n * m
```

```
## [1] 4 6 6 4
```

```
n / m
```

```
## [1] 0.2500000 0.6666667 1.5000000 4.0000000
```

```
n ^ m + m * (n - m)
```

```
## [1] -11   5  11   7
```

Если применить операторы на двух векторах одинаковой длины, то мы получим результат поэлементного применения оператора к двум векторам. Это называется **векторизацией** (**vectorization**).

> Если после какого-нибудь MATLAB Вы привыкли, что по умолчанию операторы работают по правилам линейной алгебры и `m*n` будет давать скалярное произведение (*dot product*), то снова нет. Для скалярного произведения нужно использовать операторы с `%` по краям:

```
n %*% m
```

```
##      [,1]
## [1,]   20
```

> Абсолютно так же и с операциями с матрицами в R, хотя про матрицы будет немного позже.

В принципе, большинство функций в R, которые работают с отдельными значениями, так же хорошо работают и с целыми векторами. Скажем, Вы хотите извлечь корень из нескольких чисел, для этого не нужны никакие циклы (как это обычно делается в других языках программирования). Можно просто "скормить" вектор функции и получить результат применения функции к каждому элементу вектора:

```
sqrt(1:10)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
##  [9] 3.000000 3.162278
```

Таких векторизованных функций в R очень много. Многие из них написаны на более низкоуровневых языках программирования (C, C++, FORTRAN), за счет чего использование таких функций приводит не только к более элегантному, лаконичному, но и к более быстрому коду.

> Векторизация в R — это очень важная фишка, которая отличает этот язык программирования от многих других. Если вы уже имеете опыт программирования на другом языке, то вам во многих задачах захочется использовать циклы типа `for` и `while` **??**. Не спешите этого делать! В очень многих случаях циклы можно заменить векторизацией. Тем не менее, векторизация — это не единственный способ избавить от циклов типа `for` и `while` **??**.

2.7.3 Ресайклинг

Допустим мы хотим совершить какую-нибудь операцию с двумя векторами. Как мы убедились, с этим обычно нет никаких проблем, если они совпадают по длине. А что если вектора не совпадают по длине? Ничего страшного! Здесь будет работать правило **ресайклинга** (*правило переписывания, recycling rule*). Это означает, что если мы делаем операцию на двух векторах разной длины, то если короткий вектор кратен по длине длинному, короткий вектор будет повторяться необходимое количество раз:

```
n <- 1:4
m <- 1:2
n * m
```

```
## [1] 1 4 3 8
```

А что будет, если совершать операции с вектором и отдельным значением? Можно считать это частным случаем ресайклинга: короткий вектор длиной 1 будет повторятся столько раз, сколько нужно, чтобы он совпадал по длине с длинным:

```
n * 2
```

```
## [1] 2 4 6 8
```

Если же меньший вектор не кратен большему (например, один из них длиной 3, а другой длиной 4), то R посчитает результат, но выдаст предупреждение.

```
n + c(3,4,5)
```

```
## Warning in n + c(3, 4, 5):
##
```

```
## [1] 4 6 8 7
```

Проблема в том, что эти предупреждения могут в неожиданный момент стать причиной ошибок. Поэтому не стоит полагаться на ресайклинг некратных по длине векторов. См. здесь[20]. А вот ресайклинг кратных по длине векторов — это очень удобная штука, которая используется очень часто.

2.7.4 Индексирование векторов

Итак, мы подошли к одному из самых сложных моментов. И одному из основных. От того, как хорошо вы научись с этим работать, зависит весь Ваш дальнейший успех на R-поприще!

Речь пойдет об **индексировании** векторов. Задача, которую Вам придется решать каждые пять минут работы в R - как выбрать из вектора (или же списка, матрицы и датафрейма) какую-то его часть. Для этого используются квадратные скобочки `[]` (не круглые - они для функций!).
Самое простое - индексировать по номеру индекса, т.е. порядку значения в векторе.

```
n <- 1:10
n[1]
```

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

```
n[10]
```

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

> Если вы знакомы с другими языками программирования (не MATLAB, там все так же) и уже научились думать, что индексация с 0 — это очень удобно и очень правильно (ну или просто свыклись с этим), то в R Вам придется переучиться обратно. Здесь первый индекс — это 1, а последний равен длине вектора — ее можно узнать с помощью функции `length()`. С обоих сторон индексы берутся включительно.

С помощью индексирования можно не только вытаскивать имеющиеся значения в векторе, но и присваивать им новые:

[20] https://stackoverflow.com/questions/6555651/under-what-circumstances-does-r-recycle

```
n[3] <- 20
n
```

```
##  [1]  1  2 20  4  5  6  7  8  9 10
```

Конечно, можно использовать целые векторы для индексирования:

```
n[4:7]
```

```
## [1] 4 5 6 7
```

```
n[10:1]
```

```
##  [1] 10  9  8  7  6  5  4 20  2  1
```

Индексирование с минусом выдаст вам все значения вектора кроме выбранных:

```
n[-1]
```

```
## [1]  2 20  4  5  6  7  8  9 10
```

```
n[c(-4, -5)]
```

```
## [1]  1  2 20  6  7  8  9 10
```

Минус здесь “выключает” выбранные значения из вектора, а не означает отсчет с конца как в Python.

Более того, можно использовать логический вектор для индексирования. В этом случае нужен логический вектор такой же длины:

```
n[c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE)]
```

```
## [1]  1 20  5  7  9
```

Логический вектор работает здесь как фильтр: пропускает только те значения, где на соответствующей позиции в логическом векторе для индексирования содержится `TRUE`, и не пропускает те значения, где на соответствующей позиции в логическом векторе для индексирования содержится `FALSE`.

Ну а если эти два вектора (исходный вектор и логический вектор индексов) не равны по длине, то тут будет снова работать правило ресайклинга!

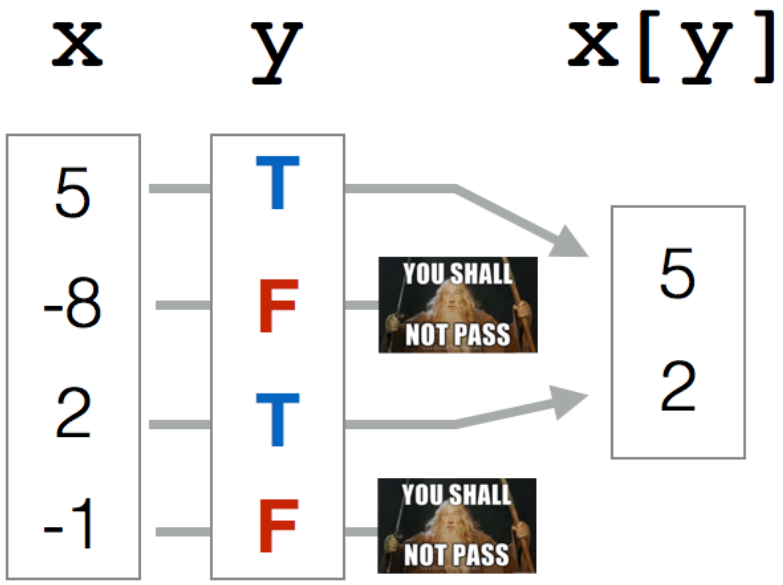


Рис. 2.7

```
n[c(TRUE, FALSE)] #        - recycling rule!
```

```
## [1]  1 20  5  7  9
```

Есть еще один способ индексирования векторов, но он несколько более редкий: индексирование по имени. Дело в том, что для значений векторов можно (но не обязательно) присваивать имена:

```
my_named_vector <- c(first = 1,
                     second = 2,
                     third = 3)
                     my_named_vector['first']
```

```
## first
##     1
```

А еще можно “вытаскивать” имена из вектора с помощью функции `names()` и присваивать таким образом новые имена.

```
d <- 1:4
names(d) <- letters[1:4]
d["a"]
```

```
## a
## 1
```

> `letters` - это "зашитая" в R константа - вектор букв от a до z. Иногда это очень удобно! Кроме того, есть константа `LETTERS` - то же самое, но заглавными буквами. А еще в R есть названия месяцев на английском и числовая константа `pi`.

Теперь посчитаем среднее вектора `n`:

```
mean(n)
```

```
## [1] 7.2
```

А как вытащить все значения, которые больше среднего?

Сначала получим логический вектор — какие значения больше среднего:

```
larger <- n > mean(n)
larger
```

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

А теперь используем его для индексирования вектора `n`:

```
n[larger]
```

```
## [1] 20  8  9 10
```

Можно все это сделать в одну строчку:

```
n[n>mean(n)]
```

```
## [1] 20  8  9 10
```

Предыдущая строчка отражает то, что мы будем постоянно делать в R: вычленять (subset) из данных отдельные куски на основании разных условий.

2.7.5 NA — пропущенные значения

В реальных данных у нас часто чего-то не хватает. Например, из-за технической ошибки или невнимательности не получилось записать какое-то измерение. Для обозначения пропущенных значений в R есть специальное значение `NA`. `NA` — это не строка `"NA"`, не `0`, не пустая строка и не `FALSE`. `NA` — это `NA`. Большинство операций с векторами, содержащими `NA` будут выдавать `NA`:

```
missed <- NA
missed == "NA"
```

```
## [1] NA
```

```
missed == ""
```

```
## [1] NA
```

```
missed == NA
```

```
## [1] NA
```

Заметьте: даже сравнение `NA` с `NA` выдает `NA`!

Иногда `NA` в данных очень бесит:

```
n[5] <- NA
n
```

```
##  [1]  1  2 20  4 NA  6  7  8  9 10
```

```
mean(n)
```

```
## [1] NA
```

Что же делать?
Наверное, надо сравнить вектор с `NA` и исключить этих пакостников. Давайте попробуем:

```
n == NA
```

```
##  [1] NA NA NA NA NA NA NA NA NA NA
```

Ах да, мы ведь только что узнали, что даже сравнение `NA` с `NA` приводит к `NA`!

Чтобы выбраться из этой непростой ситуации, используйте функцию `is.na()`:

```
is.na(n)
```

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

Результат выполнения `is.na(n)` выдает `FALSE` в тех местах, где у нас числа и `TRUE` там, где у нас `NA`. Чтобы вычленить из вектора `n` все значения кроме `NA` нам нужно, чтобы было наоборот: `TRUE`, если это не `NA`, `FALSE`, если это `NA`. Здесь нам понадобится логический оператор НЕ `!` (мы его уже встречали), который инвертирует логические значения:

```
n[!is.na(n)]
```

```
## [1]  1  2 20  4  6  7  8  9 10
```

Ура, мы можем считать среднее!

```
mean(n[!is.na(n)])
```

```
## [1] 7.444444
```

Теперь Вы понимаете, зачем нужно отрицание (`!`)

Вообще, есть еще один из способов посчитать среднее, если есть `NA`. Для этого надо залезть в хэлп по функции *mean*():

```
?mean()
```

В хэлпе мы найдем параметр `na.rm =`, который по умолчанию `FALSE`. Вы знаете, что нужно делать!

```
mean(n, na.rm = TRUE)
```

```
## [1] 7.444444
```

> `NA` может появляться в векторах других типов тоже. На самом деле, `NA` - это специальное значение в логических векторах, тогда как в векторах других типов `NA` появляется как `NA_integer_`, `NA_real_`, `NA_complex_` или `NA_character_`, но R обычно сам все переводит в нужный формат и показывает как просто `NA`.
> Кроме `NA` есть еще `NaN` — это разные вещи. `NaN` расшифровывается как Not a Number и получается в результате таких операций как `0/0`.

2.7.6 В любой непонятной ситуации — ищите в поисковике

Если вдруг вы не знаете, что искать в хэлпе, или хэлпа попросту недостаточно, то ищите в поисковике!

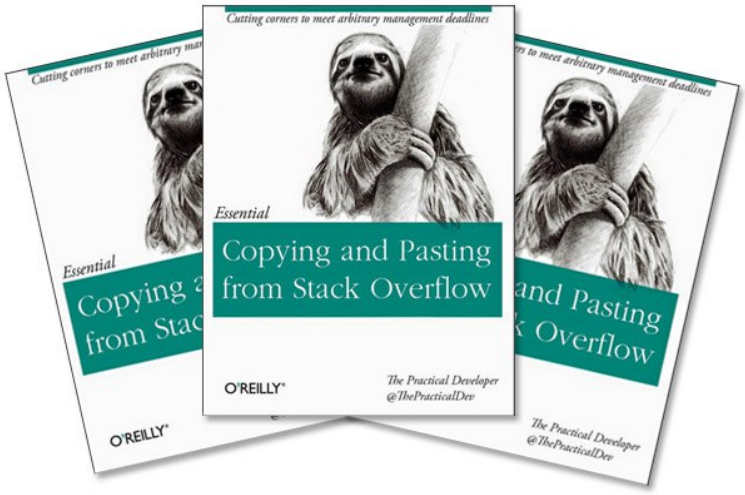


Рис. 2.8

Нет ничего постыдного в том, чтобы искать в Интернете решения проблем. Это абсолютно нормально. Используйте силу интернета во благо и да помогут вам *Stackoverflow* и бесчисленные R-туториалы!

Computer Programming To Be Officially Renamed “Googling Stack Overflow”Source: http://t.co/xu7acfXvFF pic.twitter.com/iJ9k7aAVhd

— Stack Exchange July 20, 2015

Главное, помните: загуглить работающий ответ всегда недостаточно. Надо понять, как и почему он работает. Иначе что-то обязательно пойдет не так.

Кроме того, правильно загуглить проблему — не так уж и просто.

Does anyone ever get good at R or do they just get good at googling how to do things in R

— Lauren M. Seyler, Ph.D. href=“https://twitter.com/mousquemere/status/1125522375141883907?ref_src=twsrc%5Etfw”>May 6, 2019

Итак, с векторами мы более-менее разобрались. Помните, что вектора — это один из краеугольных камней Вашей работы в R. Если Вы хорошо с ними разобрались, то дальше все будет довольно несложно. Тем не менее, вектора — это не все. Есть еще два важных типа данных: списки (**list**) и матрицы (**matrix**). Их можно рассматривать как своеобразное “расширение” векторов, каждый в свою

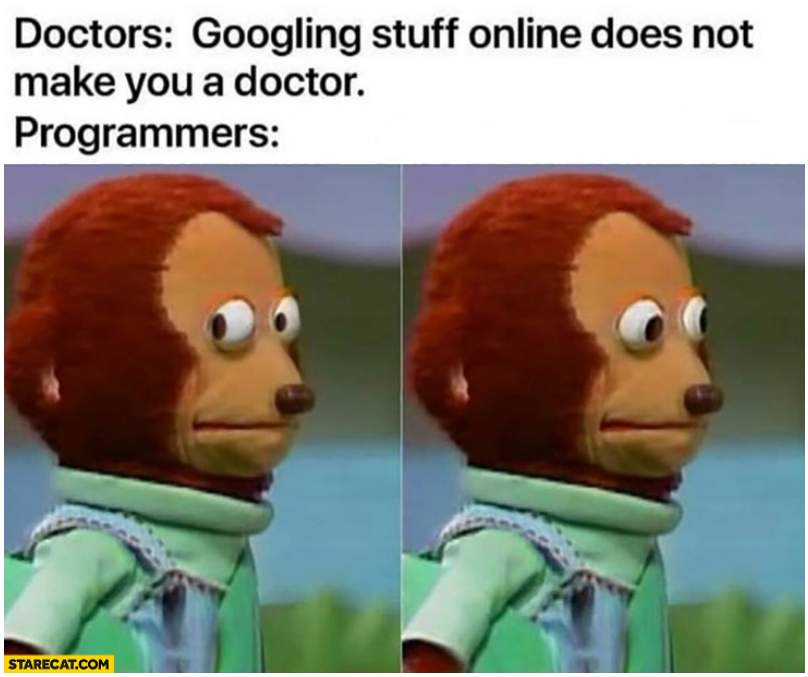


Рис. 2.9

сторону. Ну а списки и матрицы нужны чтобы понять основной тип данных в R — **data.frame**.

2.8 Матрицы (matrix)

Если вдруг Вас пугает это слово, то совершенно зря. Матрица — это всего лишь "двумерный" вектор: вектор, у которого есть не только длина, но и ширина. Создать матрицу можно с помощью функции `matrix()` из вектора, указав при этом количество строк и столбцов.

```
A <- matrix(1:20, nrow=5,ncol=4)
A
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20
```

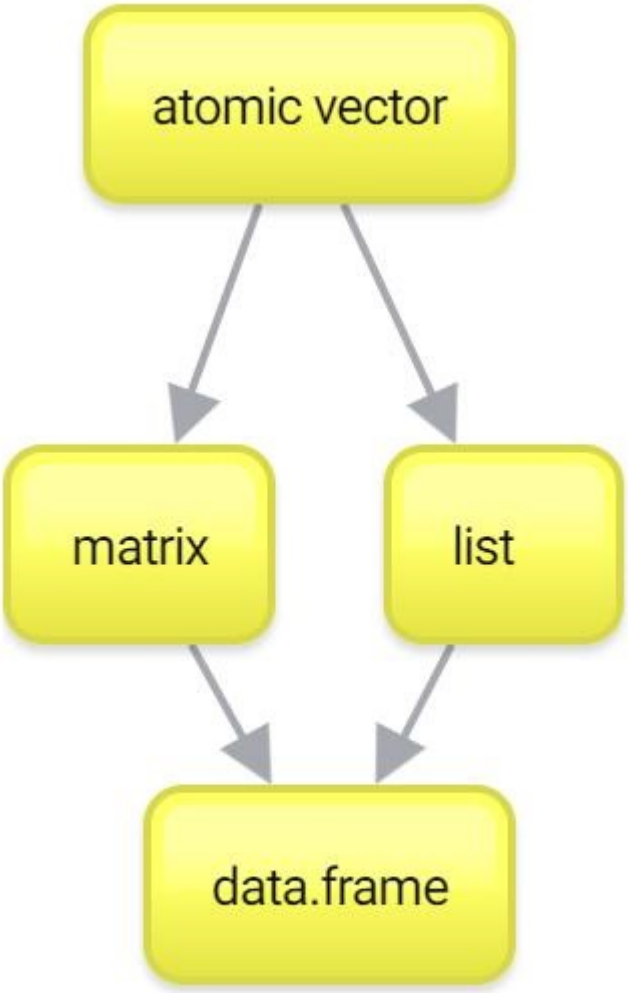


Рис. 2.10

> Заметьте, значения вектора заполняются следующим образом: сначала заполняется первый столбик сверху вниз, потом второй сверху вниз и так до конца, т.е. заполнение значений матрицы идет в первую очередь по вертикали. Это довольно стандартный способ создания матриц, характерный не только для R.

Если мы знаем сколько значений в матрице и сколько мы хотим строк, то количество столбцов указывать необязательно:

```
A <- matrix(1:20, nrow=5)
A
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20
```

Все остальное так же как и с векторами: внутри находится данные только одного типа. Поскольку матрица — это уже двумерный массив, то у него имеется два индекса. Эти два индекса разделяются запятыми.

```
A[2,3]
```

```
## [1] 12
```

```
A[2:4, 1:3]
```

```
##      [,1] [,2] [,3]
## [1,]    2    7   12
## [2,]    3    8   13
## [3,]    4    9   14
```

Первый индекс — выбор строк, второй индекс — выбор колонок. Если же мы оставляем пустое поле вместо числа, то мы выбираем все строки/колонки в зависимости от того, оставили мы поле пустым до или после запятой:

```
A[,1:3]
```

```
##      [,1] [,2] [,3]
## [1,]    1    6   11
## [2,]    2    7   12
## [3,]    3    8   13
## [4,]    4    9   14
```

```
## [5,]    5   10   15
```

```
A[2:4,]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    7   12   17
## [2,]    3    8   13   18
## [3,]    4    9   14   19
```

```
A[,]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20
```

В принципе, это все, что нам нужно знать о матрицах. Матрицы используются в R довольно редко, особенно по сравнению, например, с MATLAB. Но вот индексировать матрицы хорошо бы уметь: это понадобится в работе с датафреймами.

> То, что матрица - это просто двумерный вектор, не является метафорой: в R матрица - это по сути своей вектор с дополнительными *атрибутами* `dim` и `dimnames`. Атрибуты — это неотъемлемые свойства объектов, для всех объектов есть обязательные атрибуты типа и длины и могут быть любые необязательные атрибуты. Можно задавать свои атрибуты или удалять уже присвоенные: удаление атрибута `dim` у матрицы превратит ее в обычный вектор. Про атрибуты подробнее можно почитать здесь[21] или на стр. 99–101 книги “R in a Nutshell” (Adler, 2010).

2.9 Списки (list)

Теперь представим себе вектор без ограничения на одинаковые данные внутри. И получим список!

```
simple_list <- list(42, "    ", TRUE)
simple_list
```

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

[21] `https://perso.esiee.fr/~courivad/R/06-objects.html`

```
##
## [[2]]
## [1] "    "
##
## [[3]]
## [1] TRUE
```

А это значит, что там могут содержаться самые разные данные, в том числе и другие списки и векторы!

```
complex_list <- list(c("Wow", "this", "list", "is", "so", "big"), "16", simple_list)
complex_list
```

```
## [[1]]
## [1] "Wow"  "this" "list" "is"   "so"   "big"
##
## [[2]]
## [1] "16"
##
## [[3]]
## [[3]][[1]]
## [1] 42
##
## [[3]][[2]]
## [1] "    "
##
## [[3]][[3]]
## [1] TRUE
```

Если у нас сложный список, то есть очень классная функция, чтобы посмотреть, как он устроен, под названием `str()`:

```
str(complex_list)
```

```
## List of 3
##  $ : chr [1:6] "Wow" "this" "list" "is" ...
##  $ : chr "16"
##  $ :List of 3
##   ..$ : num 42
##   ..$ : chr "    "
##   ..$ : logi TRUE
```

Как и в случае с векторами мы можем давать имена элементам списка:

```
named_list <- list(age = 24, phd_student = T, language = "Russian")
named_list
```

```
## $age
## [1] 24
##
## $phd_student
## [1] FALSE
##
## $language
## [1] "Russian"
```

К списку можно обращаться как с помощью индексов, так и по именам. Начнем с последнего:

```
named_list$age
```

```
## [1] 24
```

А вот с индексами сложнее, и в этом очень легко запутаться. Давайте попробуем сделать так, как мы делали это раньше:

```
named_list[1]
```

```
## $age
## [1] 24
```

Мы, по сути, получили элемент списка - просто как часть списка, т.е. как список длиной один:

```
class(named_list)
```

```
## [1] "list"
```

```
class(named_list[1])
```

```
## [1] "list"
```

А вот чтобы добраться до самого элемента списка (и сделать с ним что-то хорошее) нам нужна не одна, а две квадратных скобочки:

```
named_list[[1]]
```

```
## [1] 24
```

```
class(named_list[[1]])
```

```
## [1] "numeric"
```

Indexing lists in #rstats. Inspired by the Residence Inn pic.twitter.com/YQ6axb2w7t

— Hadley Wickham (@ href="https://twitter.com/hadleywickham/status/643381054758363136?ref_src=twsrc%5Etfw">September 14, 2015

Как и в случае с вектором, к элементу списка можно обращаться по имени.

```
named_list[['age']]
```

```
## [1] 24
```

Хотя последнее — практически то же самое, что и использование знака $.

> Списки довольно часто используются в R, но реже, чем в Python. Со многими объектами в R, такими как результаты статистических тестов, объекты ggplot и т.д. удобно работать именно как со списками — к ним все вышеописанное применимо. Кроме того, некоторые данные мы изначально получаем в виде древообразной структуры — хочешь не хочешь, а придется работать с этим как со списком. Особенно это характерно для данных, выкачанных из веб-страниц (HTML страницы, XML данные) или полученных с помощью API различных веб-сайтов (например, в формате JSON). Но обычно после этого стоит как можно скорее превратить список в датафрейм.

2.10 Датафрейм

Итак, мы перешли к самому главному. Самому-самому. Датафреймы (**data.frames**). Более того, сейчас станет понятно, зачем нам нужно было разбираться со всеми предыдущими темами.

Без векторов мы не смогли бы разобраться с матрицами и списками. А без последних мы не сможем понять, что такое датафрейм.

```
name <- c("Ivan", "Eugeny", "Lena", "Misha", "Sasha")
age <- c(26, 34, 23, 27, 26)
student <- c(FALSE, FALSE, TRUE, TRUE, TRUE)
df = data.frame(name, age, student)
df
```

```
##     name age student
## 1   Ivan  26   FALSE
```

```
## 2 Eugeny  34   FALSE
## 3   Lena  23    TRUE
## 4  Misha  27    TRUE
## 5  Sasha  26    TRUE
```

```
str(df)
```

```
## 'data.frame':    5 obs. of  3 variables:
##  $ name   : chr  "Ivan" "Eugeny" "Lena" "Misha" ...
##  $ age    : num  26 34 23 27 26
##  $ student: logi  FALSE FALSE TRUE TRUE TRUE
```

Вообще, очень похоже на список, не правда ли? Так и есть, датафрейм — это что-то вроде проименованного списка, каждый элемент которого является atomic вектором фиксированной длины. Скорее всего, список Вы представляли "горизонтально". Если это так, то теперь "переверните" его у себя в голове. Так, чтоб названия векторов оказались сверху, а колонки стали столбцами. Поскольку длина всех этих векторов равна (обязательное условие!), то данные представляют собой табличку, похожую на матрицу. Но в отличие от матрицы, разные столбцы могут имет разные типы данных: первая колонка — character, вторая колонка — numeric, третья колонка — logical. Тем не менее, обращаться с датафреймом можно и как с проименованным списком, и как с матрицей:

```
df$age[2:3]
```

```
## [1] 34 23
```

Здесь мы сначала вытащили колонку age с помощью оператора $. Результатом этой операции является числовой вектор, из которого мы вытащили кусок, выбрав индексы 2 и 3.

Используя оператор $ и присваивание можно создавать новые колонки датафрейма:

```
df$lovesR <- TRUE #     recycling -    ?
df
```

```
##     name age student lovesR
## 1   Ivan  26   FALSE   TRUE
## 2 Eugeny  34   FALSE   TRUE
## 3   Lena  23    TRUE   TRUE
## 4  Misha  27    TRUE   TRUE
## 5  Sasha  26    TRUE   TRUE
```

Ну а можно просто обращаться с помощью двух индексов через запятую, как мы это делали с матрицей:

```
df[3:5, 2:3]
```

```
##   age student
## 3  23    TRUE
## 4  27    TRUE
## 5  26    TRUE
```

Как и с матрицами, первый индекс означает строчки, а второй — столбцы.

А еще можно использовать названия колонок внутри квадратных скобок:

```
df[1:2,"age"]
```

```
## [1] 26 34
```

И здесь перед нами открываются невообразимые возможности! Узнаем, любят ли R те, кто моложе среднего возраста в группе:

```
df[df$age < mean(df$age), 4]
```

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

Эту же задачу можно выполнить другими способами:

```
df$lovesR[df$age < mean(df$age)]
```

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

```
df[df$age < mean(df$age), 'lovesR']
```

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

В большинстве случаев подходят сразу несколько способов — тем не менее, стоит овладеть ими всеми.

Датафреймы удобно просматривать в RStudio. Для это нужно написать команду `View(df)` или же просто нажать на названии нужной переменной из списка вверху справа (там где Environment). Тогда увидите табличку, очень похожую на Excel и тому подобные программы для работы с таблицами. Там же есть и всякие возможности для фильтрации, сортировки и поиска... Но, конечно, интереснее все эти вещи делать руками, т.е. с помощью написания кода.

На этом пора заканчивать с введением и приступать к реальным данным.

2.11 Начинаем работу с реальными данными

Итак, пришло время перейти к реальным данным. Мы начнем с использования датасета (так мы будем называть любой набор данных) по Игре Престолов, а точнее, по книгам цикла *"Песнь льда и пламени"* Дж. Мартина. Да, будут спойлеры, но сериал уже давно закончился и сильно разошелся с книгами...

2.11.1 Рабочая папка и проекты

Для начала скачайте файл по ссылке[22]

Он, скорее всего, появился у Вас в папке "Загрузки". Если мы будем просто пытаться прочитать этот файл (например, с помощью `read.csv()` — мы к этой функцией очень скоро перейдем), указав его имя и разрешение, то наткнемся на такую ошибку:

> Ошибка в file(file, "rt") :не могу открыть соединение Вдобавок: Предупреждение: В file(file, "rt") : не могу открыть файл 'character-deaths.csv': No such file or directory

Это означает, что R не может найти нужный файл. Вообще-то мы даже не сказали, где искать. Нам нужно как-то совместить место, где R ищет загружаемые файлы и сами файлы. Для этого есть несколько способов.

- Магомет идет к горе: перемещение файлов в рабочую папку.

Для этого нужно узнать, какая папка является рабочей с помощью функции `getwd()` (без аргументов), найти эту папку в проводнике и переместить туда файл. После этого можно использовать просто название файла с разрешением:

```
got <- read.csv("character-deaths.csv")
```

- Гора идет к Магомету: изменение рабочей папки.

Можно просто сменить рабочую папку с помощью `setwd()` на ту, где сейчас лежит файл, прописав путь до этой папки. Теперь файл находится в рабочей папке:

```
got <- read.csv("character-deaths.csv")
```

Этот вариант использовать не рекомендуется. Как минимум, это сразу делает невозможным запустить скрипт на другом компьютере.

- Гора находит Магомета по месту прописки: указание полного пути файла.

[22] `https://raw.githubusercontent.com/Pozdniakov/stats/master/data/character-deaths.csv`

```
got <- read.csv("/Users/Username/Some_Folder/character-deaths.csv")
```

Этот вариант страдает теми же проблемами, что и предыдущий, поэтому тоже не рекомендуется.

> Для пользователей Windows есть дополнительная сложность: знак / является особым знаком для R, поэтому вместо него нужно использовать двойной //.

- Магомет использует кнопочный интерфейс: Import Dataset.

Во вкладке Environment справа в окне RStudio есть кнопка “Import Dataset”. Возможно, у Вас возникло непреодолимое желание отдохнуть от написания кода и понажимать кнопочки — сопротивляйтесь этому всеми силами, но не вините себя, если не сдержитесь.

- Гора находит Магомета в интернете.

Многие функции в R, предназначенные для чтения файлов, могут прочитать файл не только на Вашем компьютере, но и сразу из интернета. Для этого просто используйте ссылку вместо пути:

```
got <- read.csv("https://raw.githubusercontent.com/Pozdniakov/stats/master/data/character-deaths.
```

- Каждый Магомет получает по своей горе: использование проектов в RStudio.

На первый взгляд это кажется чем-то очень сложным, но это не так. Это очень просто и ОЧЕНЬ удобно. При создании проекта создается отдельная папочка, где у Вас лежат данные, хранятся скрипты, вспомогательные файлы и отчеты. Если нужно вернуться к другому проекту — просто открываете другой проект, с другими файлами и скриптами. Это еще помогает не пересекаться переменным из разных проектов — а то, знаете, использование двух переменных `data` в разных скриптах чревато ошибками. Поэтому очень удобным решением будет выделение отдельного проекта под этот курс.

2.11.2 Импорт данных

Как Вы уже поняли, импортирование данных - одна из самых муторных и неприятных вещей в R. Если у Вас получится с этим справится, то все остальное - ерунда. Мы уже разобрались с первой частью этого процесса - нахождением файла с данными, осталось научиться их читать.

Здесь стоит сделать небольшую ремарку. Довольно часто данные представляют собой табличку. Или же их можно свести к табличке. Такая табличка, как мы уже

выяснили, удобно репрезентируется в виде датафрейма. Но как эти данные хранятся на компьютере? Есть два варианта: в *бинарном* и в *текстовом* файле.

Текстовый файл означает, что такой файл можно открыть в программе "Блокнот" или ее аналоге и увидеть напечатанный текст: скрипт, роман или упорядоченный набор цифр и букв. Нас сейчас интересует именно последний случай. Таблица может быть представлена как текст: отдельные строчки в файле будут разделять разные строчки таблицы, а какой-нибудь знак-разделитель отделет колонки друг от друга.

Для чтения данных из текстового файла есть довольно удобная функция `read.table()`. Почитайте хэлп по ней и ужаснитесь: столько разных параметров на входе! Но там же вы увидете функции `read.csv()`, `read.csv2()` и некоторые другие — по сути, это тот же `read.table()`, но с другими дефолтными параметрами, соответствующие формату файла, который мы загружаем. В данном случае используется формат .csv, что означает Comma Separated Values (Значения, Разделенные Запятыми). Это просто текстовый файл, в котором "закодирована" таблица: разные строчки разделяют разные строчки таблицы, а столбцы отделяются запятыми. С этим связана одна проблема: в некоторых странах (в т.ч. и России) принято использовать запятую для разделения дробной части числа, а не точку, как это делается в большинстве стран мира. Поэтому есть "другой" формат .csv, где значения разделены точкой с запятой (`;`), а дробные значения - запятой (`,`). В этом и различие функций `read.csv()` и `read.csv2()` — первая функция предназначена для "международного" формата, вторая - для (условно) "Российского".

В первой строчке обычно содержатся названия столбцов - и это чертовски удобно, функции `read.csv()` и `read.csv2()` по дефолту считают первую строчку именно как название для колонок.

Итак, прочитаем наш файл. Для этого используем только параметр `file =`, который идет первым, и для параметра `stringsAsFactors =` поставим значение `FALSE`:

```
got <- read.csv("data/character-deaths.csv", stringsAsFactors = FALSE)
```

> По сути, факторы - это примерно то же самое, что и character, но закодированные числами. Когда-то это было придумано для экономии используемых времени и памяти, сейчас же обычно становится просто лишней морокой. Но некоторые функции требуют именно character, некоторые factor, в большинстве случаев это без разницы. Но иногда непонимание может привести к дурацким ошибкам. В данном случае мы просто пока обойдемся без факторов.

Можете проверить с помощью `View(got)`: все работает! Если же вылезает какая-то странная ерунда или же просто ошибка - попробуйте другие функции и покопаться с параметрами. Для этого читайте Help.

Кроме .csv формата есть и другие варианты хранения таблиц в виде текста. Например, .tsv - тоже самое, что и .csv, но разделитель - знак табуляции. Для чтения таких файлов есть функция `read.delim()` и `read.delim2()`. Впрочем, даже если бы ее и не было, можно было бы просто подобрать нужные параметры для функции `read.table()`. Есть даже функции, которые пытаются сами "угадать" нужные параметры для чтения — часто они справляются с этим довольно удачно. Но не всегда. Поэтому стоит научиться справляться с любого рода данными на входе.

Тем не менее, далеко не всегда таблицы представлены в виде текстового файла. Самый распространенный пример таблицы в бинарном виде — родные форматы Microsoft Excel. Если Вы попробуете открыть .xlsx файл в Блокноте, то увидите кракозябры. Это делает работу с этим файлами гораздо менее удобной, поэтому стоит избегать экселевских форматов и стараться все сохранять в .csv.

> Для работы с экселевскими файлами есть много пакетов: readxl, xlsx, openxlsx. Для чтения файлов SPSS, Stata, SAS есть пакет foreign. Что такое пакеты и как их устанавливать мы изучим позже.

2.12 Препроцессинг данных в R

Вчера мы узнали про основы языка R, про то, как работать с векторами, списками, матрицами и, наконец, датафреймами. Мы закончили день на загрузке данных, с чего мы и начнем сегодня:

```
got <- read.csv("data/character-deaths.csv", stringsAsFactors = F)
```

После загрузки данных стоит немного "осмотреть" получившийся датафрейм `got`.

2.12.1 Исследование данных

Ок, давайте немного поизучаем датасет. Обычно мы привыкли глазами пробегать по данным, листая строки и столбцы — и это вполне правильно и логично, от этого не нужно отучаться. Но мы можем дополнить наш базовый зрительно-поисковой инструментарий несколькими полезными командами.

Во-первых, вспомним другую полезную функцию `str()`:

```
str(got)
```

```
## 'data.frame':    917 obs. of  13 variables:
##  $ Name              : chr  "Addam Marbrand" "Aegon Frey (Jinglebell)" "Aegon Targaryen" "Adra
```

```
##  $ Allegiances       : chr  "Lannister" "None" "House Targaryen" "House Greyjoy" ..
##  $ Death.Year        : int  NA 299 NA 300 NA NA 300 300 NA NA ...
##  $ Book.of.Death     : int  NA 3 NA 5 NA NA 4 5 NA NA ...
##  $ Death.Chapter     : int  NA 51 NA 20 NA NA 35 NA NA NA ...
##  $ Book.Intro.Chapter: int  56 49 5 20 NA NA 21 59 11 0 ...
##  $ Gender            : int  1 1 1 1 1 1 1 0 1 1 ...
##  $ Nobility          : int  1 1 1 1 1 1 1 1 1 0 ...
##  $ GoT               : int  1 0 0 0 0 0 1 1 0 0 ...
##  $ CoK               : int  1 0 0 0 0 1 0 1 1 0 ...
##  $ SoS               : int  1 1 0 0 1 1 1 1 0 1 ...
##  $ FfC               : int  1 0 0 0 0 0 1 0 1 0 ...
##  $ DwD               : int  0 0 1 1 0 0 0 1 0 0 ...
```

Давайте разберемся с переменными в датафрейме:

Колонка `Name` — здесь все понятно. Важно, что эти имена записаны абсолютно по-разному: где-то с фамилией, где-то без, где-то в скобочках есть пояснения. Колонка `Allegiances` — к какому дому принадлежит персонаж. С этим сложно, иногда они меняют дома, здесь путаются сами семьи и персонажи, лояльные им. Особой разницы между `Stark` и `House Stark` нет. Следующие колонки - `Death Year`, `Book.of.Death`, `Death.Chapter`, `Book.Intro.Chapter` — означают номер главы, в которой персонаж впервые появляется, а так же номер книги, глава и год (от завоевания Вестероса Эйгоном Таргариеном), в которой персонаж умирает. `Gender` — 1 для мужчин, 0 для женщин. `Nobility` — дворянское происхождение персонажа. Последние 5 столбцов содержат информацию, появлялся ли персонаж в книге (всего книг пока что 5).

Другая полезная функция для больших таблиц — функция `head()`: она выведет первые несколько (по дефолту 6) строчек датафрейма.

```
head(got)
```

```
##                      Name     Allegiances Death.Year Book.of.Death
## 1          Addam Marbrand       Lannister         NA            NA
## 2 Aegon Frey (Jinglebell)            None        299             3
## 3         Aegon Targaryen House Targaryen         NA            NA
## 4           Adrack Humble   House Greyjoy        300             5
## 5          Aemon Costayne       Lannister         NA            NA
## 6         Aemon Estermont       Baratheon         NA            NA
##   Death.Chapter Book.Intro.Chapter Gender Nobility GoT CoK SoS FfC DwD
## 1            NA                56      1        1   1   1   1   1   0
## 2            51                49      1        1   0   0   1   0   0
## 3            NA                 5      1        1   0   0   0   0   1
## 4            20                20      1        1   0   0   0   0   1
## 5            NA                NA      1        1   0   0   1   0   0
## 6            NA                NA      1        1   0   1   1   0   0
```

Есть еще функция `tail()`. Догадайтесь сами, что она делает.

Для некоторых переменных полезно посмотреть таблицы частотности с помощью функции table():

```
table(got$Allegiances)
```

```
##
##           Arryn       Baratheon         Greyjoy     House Arryn House Baratheon
##              23              56              51               7               8
##   House Greyjoy House Lannister   House Martell     House Stark House Targaryen
##              24              21              12              35              19
##     House Tully    House Tyrell       Lannister         Martell   Night's Watch
##               8              11              81              25             116
##            None           Stark       Targaryen           Tully          Tyrell
##             253              73              17              22              15
##        Wildling
##              40
```

Уау! Очень просто и удобно, не так ли? Функция `table()` может принимать сразу несколько столбцов. Это удобно для получения *таблиц сопряженности*:

```
table(got$Allegiances, got$Gender)
```

```
##
##                    0   1
##   Arryn            3  20
##   Baratheon        6  50
##   Greyjoy          4  47
##   House Arryn      3   4
##   House Baratheon  0   8
##   House Greyjoy    1  23
##   House Lannister  2  19
##   House Martell    7   5
##   House Stark      6  29
##   House Targaryen  5  14
##   House Tully      0   8
##   House Tyrell     4   7
##   Lannister       12  69
##   Martell          7  18
##   Night's Watch    0 116
##   None            51 202
##   Stark           21  52
##   Targaryen        1  16
##   Tully            2  20
```

```
##   Tyrell            6   9
##   Wildling         16  24
```

2.12.2 Subsetting

Как мы обсуждали на прошлом занятии, мы можем сабсеттить (выделять часть датафрейма) датафрейм, обращаясь к нему и как к матрице: *датафрейм*[*вектор_с_номерами_строк, вектор_с_номерами_колонок*]

```
got[100:115, 1:2]
```

```
##                 Name   Allegiances
## 100        Blue Bard  House Tyrell
## 101    Bonifer Hasty     Lannister
## 102           Borcas Night's Watch
## 103  Boremund Harlaw       Greyjoy
## 104     Boros Blount     Baratheon
## 105           Borroq      Wildling
## 106      Bowen Marsh Night's Watch
## 107       Bran Stark   House Stark
## 108   Brandon Norrey         Stark
## 109          Brenett          None
## 110 Brienne of Tarth         Stark
## 111            Bronn     Lannister
## 112    Brown Bernarr Night's Watch
## 113           Brusco          None
## 114   Bryan Fossoway     Baratheon
## 115      Bryce Caron     Baratheon
```

и используя имена колонок:

```
got[508:515, "Name"]
```

```
## [1] "Mance Rayder"    "Mandon Moore"    "Maric Seaworth"  "Marei"
## [5] "Margaery Tyrell" "Marillion"       "Maris"           "Marissa Frey"
```

и даже используя вектора названий колонок!

```
got[508:515, c("Name", "Allegiances", "Gender")]
```

```
##                Name     Allegiances Gender
## 508    Mance Rayder        Wildling      1
## 509    Mandon Moore       Baratheon      1
## 510  Maric Seaworth House Baratheon      1
```

```
## 511           Marei            None      0
## 512 Margaery Tyrell    House Tyrell      0
## 513       Marillion           Arryn      1
## 514           Maris        Wildling      0
## 515    Marissa Frey            None      0
```

Мы можем вытаскивать отдельные колонки как векторы:

```
houses <- got$Allegiances
unique(houses) #                    ---             table()
```

```
##  [1] "Lannister"       "None"            "House Targaryen" "House Greyjoy"
##  [5] "Baratheon"       "Night's Watch"   "Arryn"           "House Stark"
##  [9] "House Tyrell"    "Tyrell"          "Stark"           "Greyjoy"
## [13] "House Lannister" "Martell"         "House Martell"   "Wildling"
## [17] "Targaryen"       "House Arryn"     "House Tully"     "Tully"
## [21] "House Baratheon"
```

Итак, давайте решим нашу первую задачу — вытащим в отдельный датасет всех представителей Ночного Дозора. Для этого нам нужно создать вектор логических значений — результат сравнений колонки `Allegiances` со значением `"Night's Watch"` и использовать его как вектор индексов для датафрейма.

```
vectornight <- got$Allegiances == "Night's Watch"
head(vectornight)
```

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

Теперь этот вектор с `TRUE` и `FALSE` нам надо использовать для индексирования строк. Но что со столбцами? Если мы хотем сохранить все столбцы, то после запятой внутри квадратных скобок нам не нужно ничего указывать:

```
nightswatch <- got[vectornight,]
head(nightswatch)
```

```
##                                Name   Allegiances Death.Year Book.of.Death
## 7  Aemon Targaryen (son of Maekar I) Night's Watch        300             4
## 10                          Aethan Night's Watch         NA            NA
## 13                    Alan of Rosby Night's Watch        300             5
## 16                          Albett Night's Watch         NA            NA
## 24                   Alliser Thorne Night's Watch         NA            NA
## 49                           Arron Night's Watch         NA            NA
##    Death.Chapter Book.Intro.Chapter Gender Nobility GoT CoK SoS FfC DwD
## 7             35                21      1        1   1   0   1   1   0
## 10            NA                 0      1        0   0   0   1   0   0
```

```
## 13             4                 18      1        1   0   1   1   0   1
## 16            NA                 26      1        0   1   0   0   0   0
## 24            NA                 19      1        0   1   1   1   0   1
## 49            NA                 75      1        0   0   0   1   0   1
```

Вуаля! Все это можно сделать проще и в одну строку:

```
nightswatch <- got[got$Allegiances == "Night's Watch",]
```

И не забывайте про запятую!

Теперь попробуем вытащить одновременно всех Одичалых (`Wildling`) и всех представителей Ночного Дозора. Это можно сделать, используя оператор `|` (ИЛИ) при выборе колонок:

```
nightwatch_wildling <- got[got$Allegiances == "Night's Watch" | got$Allegiances == "Wi
head(nightwatch_wildling)
```

```
##                                Name   Allegiances Death.Year Book.of.Death
## 7  Aemon Targaryen (son of Maekar I) Night's Watch        300             4
## 10                          Aethan Night's Watch         NA            NA
## 13                     Alan of Rosby Night's Watch        300             5
## 16                          Albett Night's Watch         NA            NA
## 24                    Alliser Thorne Night's Watch         NA            NA
## 49                           Arron Night's Watch         NA            NA
##    Death.Chapter Book.Intro.Chapter Gender Nobility GoT CoK SoS FfC DwD
## 7             35                 21      1        1   1   0   1   1   0
## 10            NA                  0      1        0   0   0   1   0   0
## 13             4                 18      1        1   0   1   1   0   1
## 16            NA                 26      1        0   1   0   0   0   0
## 24            NA                 19      1        0   1   1   1   0   1
## 49            NA                 75      1        0   0   0   1   0   1
```

> Кажется очевидным следующий вариант: `got[got$Allegiances == c("Night's Watch "Wildling"),]`. Однако это выдаст не совсем то, что нужно, хотя результат может показаться верным на первый взгляд. Попробуйте самостоятельно ответить на вопрос, что происходит в данном случае и чем результат отличается от предполагаемого. Подсказка: вспомните правило recycling.

Для таких случаев есть удобный оператор `%in%`, который позволяет сравнить каждое значение вектора с целым набором значений. Если значение вектора хотя бы один раз встречается в векторе справа от `%in%`, то результат — `TRUE`:

```
1:6 %in% c(1,4,5)
```

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

```
nightwatch_wildling <- got[got$Allegiances %in% c("Night's Watch", "Wildling"),]
head(nightwatch_wildling)
```

```
##                                Name   Allegiances Death.Year Book.of.Death
## 7  Aemon Targaryen (son of Maekar I) Night's Watch        300             4
## 10                           Aethan Night's Watch         NA            NA
## 13                     Alan of Rosby Night's Watch        300             5
## 16                           Albett Night's Watch         NA            NA
## 24                    Alliser Thorne Night's Watch         NA            NA
## 49                            Arron Night's Watch         NA            NA
##    Death.Chapter Book.Intro.Chapter Gender Nobility GoT CoK SoS FfC DwD
## 7             35                21      1        1   1   0   1   1   0
## 10            NA                 0      1        0   0   0   1   0   0
## 13             4                18      1        1   0   1   1   0   1
## 16            NA                26      1        0   1   0   0   0   0
## 24            NA                19      1        0   1   1   1   0   1
## 49            NA                75      1        0   0   0   1   0   1
```

2.12.3 Создание новых колонок

Давайте создадим новую колонку, которая будет означать, жив ли еще персонаж (по книгам). Заметьте, что в этом датасете, хоть он и посвящен смертям персонажей, нет нужной колонки. Мы можем попытаться "вытащить" эту информацию. В колонках `Death.Year`, `Death.Chapter` и `Book.of.Death` стоит `NA` у многих персонажей. Например, у `Arya Stark`, которая и по книгам, и по сериалу живее всех живых и мертвых:

```
got[got$Name == "Arya Stark",]
```

```
##          Name Allegiances Death.Year Book.of.Death Death.Chapter
## 56 Arya Stark       Stark         NA            NA            NA
##    Book.Intro.Chapter Gender Nobility GoT CoK SoS FfC DwD
## 56                  2      0        1   1   1   1   1   1
```

Следовательно, если в `Book.of.Death` стоит `NA`, мы можем предположить, что Джордж Мартин еще не занес своей карающей руки над этим героем.

Мы можем создать новую колонку `Is.Alive`:

```
got$Is.Alive <- is.na(got$Book.of.Death)
```

2.12.4 data.table vs. tidyverse

В принципе, с помощью базового R можно сделать все, что угодно. Однако базовые инструменты R — не всегда самые удобные. Идея сделать работу с датафреймами в R еще быстрее и удобнее сподвигла разработчиков на создание новых инструментов — `data.table` и `tidyverse` (`dplyr`). Это два конкурирующих подхода, которые сильно перерабатывают язык, хотя это по-прежнему все тот же R — поэтому их еще называют “диалектами” R.

Оба подхода обладают своими преимуществами и недостатками, но на сегодняшний день `tidyverse` считается более популярным. Основное преимущество этого подхода — в относительной легкости освоения. Обычно код, написанный в `tidyverse` можно примерно понять, даже не владея им.

Преимущество `data.table` — в суровом лаконичном синтаксисе и наиболее эффективных алгоритмах. Последние обеспечивают очень серьезный прирост в скорости в работе с данными. Чтение файлов и манипуляция данными может быть на порядки быстрее, поэтому если Ваш датасет с трудом пролезает в оперативную память компьютера, а исполнение скрипта занимает длительное время - стоит задуматься о переходе на `data.table`.

Что из этого учить — решать Вам, но знать оба совсем не обязательно: они решают те же самые задачи, просто совсем разными способами. За data.table — скорость, за tidyverse - понятность синтаксиса. Очень советую почитать обсуждение на эту тему здесь[23].

[23] `https://stackoverflow.com/questions/21435339/data-table-vs-dplyr-can-one-do-something-well-the-other-cant-or-does-poorly`

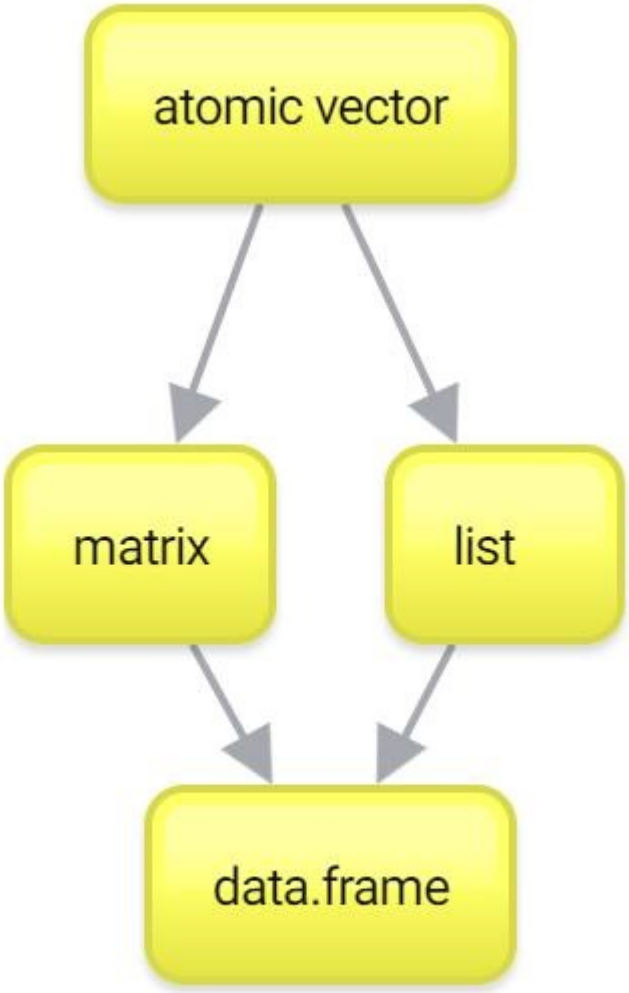


Рис. 2.11

Глава 3

Импорт данных

Итак, пришло время перейти к реальным данным. Мы начнем с использования датасета (так мы будем называть любой набор данных) по супергероям. Этот датасет представляет собой табличку, каждая строка которой - отдельный супергерой, а столбик — какая-либо информация о нем. Например, цвет глаз, цвет волос, вселенная супергероя[1], рост, вес, пол и так далее. Несложно заметить, что этот датасет идеально подходит под структуру датафрейма: прямоугольная табличка, внутри которой есть разные колонки, каждая из которой имеет свой тип (числовой или строковый).

3.1 Рабочая папка и проекты RStudio

Для начала скачайте файл по ссылке[2]

Он, скорее всего, появился у Вас в папке “Загрузки”. Если мы будем просто пытаться прочитать этот файл (например, с помощью `read.csv()` — мы к этой функцией очень скоро перейдем), указав его имя и разрешение, то наткнемся на такую ошибку:

```
read.csv("heroes_information.csv")
```

```
## Warning in file(file, "rt"):              'heroes_information.csv': No
## such file or directory
```

[1] супергерои в комиксах, фильмах и телесериалах часто взаимодействуют друг с другом, однако обычно это взаимодействие происходит между супергероями одного издателя. Два крупнейших издателя комиксов — DC и Marvel, поэтому принято говорить о вселенной DC и Marvel.

[2] `https://raw.githubusercontent.com/agricolamz/2020-2021-ds4dh/master/data/heroes_information.csv`

```
## Error in file(file, "rt"):
```

Это означает, что R не может найти нужный файл. Вообще-то мы даже не сказали, где искать. Нам нужно как-то совместить место, где R ищет загружаемые файлы и сами файлы. Для этого есть несколько способов.

- Магомет идет к горе: перемещение файлов в рабочую папку.

Для этого нужно узнать, какая папка является рабочей с помощью функции `getwd()` (без аргументов), найти эту папку в проводнике и переместить туда файл. После этого можно использовать просто название файла с разрешением:

```
heroes <- read.csv("heroes_information.csv")
```

Кроме того, путь к рабочей папке можно увидеть в RStudio во вкладке с консолью, в самой верхней части (прямо под надписью “Console”):

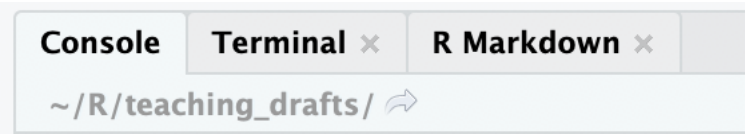


Рис. 3.1

- Гора идет к Магомету: изменение рабочей папки.

Можно просто сменить рабочую папку с помощью `setwd()` на ту, где сейчас лежит файл, прописав путь до этой папки. Теперь файл находится в рабочей папке:

```
heroes <- read.csv("heroes_information.csv")
```

Этот вариант использовать не рекомендуется! Как минимум, это сразу делает невозможным запустить скрипт на другом компьютере.

- Гора находит Магомета по месту прописки: указание полного пути файла.

```
heroes <- read.csv("/Users/Username/Some_Folder/heroes_information.csv")
```

Этот вариант страдает теми же проблемами, что и предыдущий, поэтому тоже не рекомендуется!

> Для пользователей Windows есть дополнительная сложность: знак `/` является особым знаком для R, поэтому вместо него нужно использовать двойной `//`.

- Магомет использует кнопочный интерфейс: Import Dataset.

Во вкладке Environment справа в окне RStudio есть кнопка “Import Dataset”. Возможно, у Вас возникло непреодолимое желание отдохнуть от написания кода и понажимать кнопочки — сопротивляйтесь этому всеми силами, но не вините себя, если не сдержитесь.

- Гора находит Магомета в интернете.

Многие функции в R, предназначенные для чтения файлов, могут прочитать файл не только на Вашем компьютере, но и сразу из интернета. Для этого просто используйте ссылку вместо пути:

```
heroes <- read.csv("https://raw.githubusercontent.com/agricolamz/2020-2021-ds4dh/master/data/hero
```

- Каждый Магомет получает по своей горе: использование проектов в RStudio.

На первый взгляд это кажется чем-то очень сложным, но это не так. Это очень просто и ОЧЕНЬ удобно. При создании проекта создается отдельная папочка, где у Вас лежат данные, хранятся скрипты, вспомогательные файлы и отчеты. Если нужно вернуться к другому проекту — просто открываете другой проект, с другими файлами и скриптами. Это еще помогает не пересекаться переменным из разных проектов — а то, знаете, использование двух переменных `data` в разных скриптах чревато ошибками. Поэтому очень удобным решением будет выделение отдельного проекта под этот курс.

> При закрытии проекта все переменные по умолчанию тоже будут сохраняться, а при открытии — восстанавливаться. Это очень удобно, хотя некоторые рекомендуют от этого отказаться[3]. Это можно сделать во вкладке `Tool - Global Options...`

3.1.1 Табличные данные: текстовые и бинарные данные

Как Вы уже поняли, импортирование данных - одна из самых муторных и неприятных вещей в R. Если у Вас получится с этим справится, то все остальное - ерунда. Мы уже разобрались с первой частью этого процесса - нахождением файла с данными, осталось научиться их читать.

Здесь стоит сделать небольшую ремарку. Довольно часто данные представляют собой табличку. Или же их можно свести к табличке. Такая табличка, как мы уже выяснили, удобно репрезентируется в виде датафрейма. Но как эти данные хранятся на компьютере? Есть два варианта: в *бинарном* и в *текстовом* файле.

Текстовый файл означает, что такой файл можно открыть в программе “Блокнот” или аналоге (например, TextEdit на macOS) и увидеть напечатанный текст: скрипт, роман или упорядоченный набор цифр и букв. Нас сейчас интересует

[3] `https://r4ds.had.co.nz/workflow-projects.html`

именно последний случай. Таблица может быть представлена как текст: отдельные строчки в файле будут разделять разные строчки таблицы, а какой-нибудь знак-разделитель отделять колонки друг от друга.

Для чтения данных из текстового файла есть довольно удобная функция `read.table()`. Почитайте хэлп по ней и ужаснитесь: столько разных параметров на входе! Но там же вы увидете функции `read.csv()`, `read.csv2()` и некоторые другие — по сути, это тот же `read.table()`, но с другими параметрами по умолчанию, соответствующие формату файла, который мы загружаем. В данном случае используется формат .csv, что означает "Comma Separated Values" (Значения, Разделенные Запятыми). Формат .csv — это самый известный способ хранения табличных данных в файде на сегодняшний день. Файлы с расширением .csv можно легко открыть в любой программе, работающей с таблицами, в том числе Microsoft Excel и его аналогах.

Файл с расширением .csv — это просто текстовый файл, в котором "закодирована" таблица: разные строчки разделяют разные строчки таблицы, а столбцы отделяются запятыми (отсюда и название). Вы можете вручную создать такие файлы в Блокноте и сохранять их с форматом .csv - и такая табличка будет нормально открываться в Microsoft Excel и других программах для работы с таблицами. Можете попробовать это сделать самостоятельно!

Как говорилось ранее, в качестве разделителя ячеек по горизонтали — то есть разделителя между столбцами — используется запятая. С этим связана одна проблема: в некоторых странах (в т.ч. и России) принято использовать запятую для разделения дробной части числа, а не точку, как это делается в большинстве стран мира. Поэтому есть альтернативный вариант формата .csv, где значения разделены точкой с запятой (`;`), а дробные значения - запятой (`,`). В этом и различие функций `read.csv()` и `read.csv2()` — первая функция предназначена для "международного" формата, вторая - для (условно) "Российского". Оба варианта формата имеют расширение .csv, поэтому заранее понять какой именно будет вариант довольно сложно, приходится либо пробовать оба, либо заранее открывать файл в текстовом редакторе.

В первой строчке обычно содержатся названия столбцов - и это чертовски удобно, функции `read.csv()` и `read.csv2()` по умолчанию считают первую строчку именно как название для колонок.

Кроме .csv формата есть и другие варианты хранения таблиц в виде текста. Например, .tsv — тоже самое, что и .csv, но разделитель - знак табуляции. Для чтения таких файлов есть функция `read.delim()` и `read.delim2()`. Впрочем, даже если бы ее и не было, можно было бы просто подобрать нужные параметры для функции `read.table()`. Есть даже функции, которые пытаются сами "угадать" нужные параметры для чтения — часто они справляются с этим довольно удачно. Но не всегда. Поэтому стоит научиться справляться с любого рода данными на входе.

Итак, прочитаем наш файл. Для этого используем только параметр `file =`, который идет первым, и для параметра `stringsAsFactors =` поставим значение

FALSE:

```
heroes <- read.csv("data/heroes_information.csv", stringsAsFactors = FALSE)
```

> Параметр `stringsAsFactors =` задает то, как будут прочитаны строковые значения - как уже знакомые нам строки или как факторы. По сути, факторы - это примерно то же самое, что и character, но закодированные числами. Когда-то это было придумано для экономии используемых времени и памяти, сейчас же обычно становится просто лишней морокой. Но некоторые функции требуют именно character, некоторые factor, в большинстве случаев это без разницы. Но иногда непонимание может привести к дурацким ошибкам. В данном случае мы просто пока обойдемся без факторов. Если у вас версия R выше 4.0, то `stringsAsFactors =` будет `FALSE` по умолчанию.

Можете проверить с помощью `View(heroes)`: все работает! Если же вылезает какая-то странная ерунда или же просто ошибка - попробуйте другие функции (`read.table()`, `read.delim()`) и покопаться с параметрами. Для этого читайте `Help`.

3.2 Проверка импортированных данных

При импорте данных обратите внимания на предупреждения (если таковые появляются), в большинстве случаев они указывают на то, что данные импортированы некорректно.

Проверим, что все прочиталось нормально с помощью уже известной нам функции `str()`:

```
str(heroes)
```

```
## 'data.frame':    734 obs. of  11 variables:
##  $ X         : int  0 1 2 3 4 5 6 7 8 9 ...
##  $ name      : chr  "A-Bomb" "Abe Sapien" "Abin Sur" "Abomination" ...
##  $ Gender    : chr  "Male" "Male" "Male" "Male" ...
##  $ Eye.color : chr  "yellow" "blue" "blue" "green" ...
##  $ Race      : chr  "Human" "Icthyo Sapien" "Ungaran" "Human / Radiation" ...
##  $ Hair.color: chr  "No Hair" "No Hair" "No Hair" "No Hair" ...
##  $ Height    : num  203 191 185 203 -99 193 -99 185 173 178 ...
##  $ Publisher : chr  "Marvel Comics" "Dark Horse Comics" "DC Comics" "Marvel Comics" ...
##  $ Skin.color: chr  "-" "blue" "red" "-" ...
##  $ Alignment : chr  "good" "good" "good" "bad" ...
##  $ Weight    : int  441 65 90 441 -99 122 -99 88 61 81 ...
```

> Всегда проверяйте данные на входе и никогда не верьте на слово, если вам говорят, что данные вычищенные и не содержат никаких ошибок.

На что нужно обращать внимание?

1. Прочитаны ли пропущенные значения как `NA`. По умолчанию пропущенные значения обозначаются пропущенной строчкой или "NA", но встречаются самые разнообразные варианты. Возможные варианты кодирования пропущенных значений можно задать в параметре `na.strings =` функции `read.table()` и ее вариантов. В нашем датасете как раз такая ситуация, где нужно самостоятельно задавать, какие значения будут прочитаны как `NA`. Попытайтесь самостоятельно догадаться, как именно.

2. Прочитаны ли те столбики, которые должны быть числовыми, как `int` или `num`. Если в колонке содержатся числа, а написано `chr` (= `"character"`) или `Factor` (в случае если `stringsAsFactors = TRUE`), то, скорее всего, одна из строчек содержит в себе нечисловые знаки, которые не были прочитаны как `NA`.

3. Странные названия колонок. Это может случиться по самым разным причинам, но в таких случаях стоит открывать файл в другой программе и смотреть первые строчки. Например, может оказаться, что первые несколько строчек — пустые или что первая строчка не содержит название столбцов (тогда для параметра `header =` нужно поставить `FALSE`)

4. Вместо строковых данных у вас кракозябры. Это означает проблемы с кодировкой. В первую очередь попробуйте выставить значение `"UTF-8"` для параметра `encoding =` в функции для чтения файла:

```
heroes <- read.csv("data/heroes_information.csv",
                   stringsAsFactors = FALSE,
                   encoding = "UTF-8")
```

В случае если это не помогает, попробуйте разобрать[4], что это за кодировка.

5. Все прочиталось как одна колонка. В этом случае, скорее всего, неправильно подобран разделить колонок — параметр `sep =`. Откройте файл в текстовом редакторе, чтобы понять какой нужно использовать.

6. В отдельных строчках все прочиталось как одна колонка, а в остальных нормально. Скорее всего, в файле есть значения типа `\` или `"`, которые в функциях `read.csv()`, `read.delim()`, `read.csv2()`, `read.delim2()` читаются как символы для закавычивания значений. Это может понадобиться, если у вас в таблице есть строковые значения со знаками `,` или `;`, которые могут восприниматься как разделитель столбцов.

[4] `https://www.artlebedev.ru/decoder/`

7. Появились какие-то новые числовые колонки. Возможно неправильно поставлен разделитель дробной части. Обычно это либо `.` (`read.table()`, `read.csv()`, `read.delim()`), либо `,` (`read.csv2()`, `read.delim2()`).

Конкретно в нашем случае все прочиталось хорошо с помощью функции `read.csv()`, но в строковых переменных есть много прочерков, которые обозначают отсутствие информации по данному параметру супергероя, т.е. пропущенное значение. А вот с числовыми значениями все не так просто: для всех супергероев прописано какое-то число, но во многих случаях это -99. Очевидно, отрицательного роста и массы не бывает, это просто обозначение пропущенных значений (такое часто используется). Таким образом, чтобы адекватно прочитать файл, нам нужно поменять параметр `na.strings =` функции `read.csv()`:

```
heroes <- read.csv("data/heroes_information.csv",
                   stringsAsFactors = FALSE,
                   na.strings = c("-", "-99"))
```

3.3 Импорт таблиц в бинарном формате: таблицы Excel, SPSS

Тем не менее, далеко не всегда таблицы представлены в виде текстового файла. Самый распространенный пример таблицы в бинарном виде — родные форматы Microsoft Excel. Если Вы попробуете открыть .xlsx файл в Блокноте, то увидите кракозябры. Это делает работу с этим файлами гораздо менее удобной, поэтому стоит избегать экселевских форматов и стараться все сохранять в .csv.

Такие файлы не получится прочитать при помощи базового инструментария R. Тем не менее, для чтения таких файлов есть много дополнительных пакетов:

- файлы Microsoft Excel: лучше всего справляется пакет `readxl` (является частью расширенного tidyverse), у него есть много альтернатив (`xlsx`, `openxlsx`).
- файлы SPSS, SAS, Stata: существуют два основных пакета — `haven` (часть расширенного tidyverse) и `foreign`.

Что такое пакеты и как их устанавливать мы изучим очень скоро.

Глава 4

Задания

4.1 Вектор

- Посчитайте логарифм от 8912162342 по основанию 6

```
## [1] 12.7867
```

- Теперь натуральный логарифм 10 и умножьте его на 5

```
## [1] 11.51293
```

- Создайте вектор от 1 до 20

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

- Создайте вектор от 20 до 1

```
##  [1] 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1
```

- Создайте вектор от 1 до 20 и снова до 1. Число 20 должно присутствовать только один раз!

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 19 18 17 16 15
## [26] 14 13 12 11 10  9  8  7  6  5  4  3  2  1
```

- Создайте вектор 2, 4, 6, ... , 18, 20

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

- Создайте вектор из одной единицы, двух двоек, трех троек, ... , девяти девяток

```
##  [1] 1 2 2 3 3 3 4 4 4 4 5 5 5 5 5 6 6 6 6 6 6 7 7 7 7 7 7 7 8 8 8 8 8 8 8 8 9 9
## [39] 9 9 9 9 9 9 9
```

- Сделайте вектор vec, в котором соедините 3, а также значения " " и " ".

```
## [1] "3"      " "    "    "
```

- Вычесть TRUE из 10

```
## [1] 9
```

- Соедините значение 10 и TRUE в вектор vec

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

- Соедините вектор vec и значение "r":

```
## [1] "10" "1"  "r"
```

- Соедините значения 10, TRUE, "r" в вектор.

```
## [1] "10"   "TRUE" "r"
```

4.2 Вектор. Операции с векторами

Создайте вектор p, состоящий из значений 4, 5, 6, 7, и вектор q, состоящий из 0, 1, 2, 3.

```
## [1] 4 5 6 7
```

```
## [1] 0 1 2 3
```

Посчитайте поэлементную сумму векторов p и q:

```
## [1]  4  6  8 10
```

Посчитайте поэлементную разницу p и q:

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

Поделите каждый элемент вектора p на соответствующий ему элемент вектора q:

О, да, Вам нужно делить на 0!

```
## [1]      Inf 5.000000 3.000000 2.333333
```

Возведите каждый элемент вектора p в степень соответствующего ему элемента вектора q:

```
## [1]   1   5  36 343
```

Создайте вектор квадратов чисел от 1 до 10:

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Создайте вектор 0, 2, 0, 4, ... , 18, 0, 20

```
##  [1]  0  2  0  4  0  6  0  8  0 10  0 12  0 14  0 16  0 18  0 20
```

4.3 Вектор. Индексирование

Создайте вектор `vec1`:

```
vec1 <- c(3, 5, 2, 1, 8, 4, 9, 10, 3, 15, 1, 11)
```

- Найдите второй элемент вектора `vec1`:

```
## [1] 5
```

- Найдите последний элемент вектора `vec1`

```
## [1] 11
```

- Найдите все значения вектора `vec1`, которые больше 4

```
## [1]  5  8  9 10 15 11
```

- Найдите все значения вектора vec1, которые больше 4, но меньше 10

```
## [1] 5 8 9
```

- Возведите в квадрат каждое значение вектора `vec1`

```
##  [1]   9  25   4   1  64  16  81 100   9 225   1 121
```

- Возведите в квадрат каждое значение вектора на нечетной позиции и извлеките корень из каждого значения на четной позиции вектора `vec1`

```
##  [1]  9.000000  2.236068  4.000000  1.000000 64.000000  2.000000 81.000000
##  [8]  3.162278  9.000000  3.872983  1.000000  3.316625
```

- Создайте вектор `vec2`, в котором все значения `vec1`, которые меньше 10, будут заменены на `NA`.

```
##  [1] NA NA NA NA NA NA NA 10 NA 15 NA 11
```

- Посчитайте сумму `vec2` с помощью функции `sum()`. Ответ `NA` не считается!

```
## [1] 36
```

- Создайте вектор 2, 4, 6, ... , 18, 20 как минимум 2 новыми способами

  Знаю, это задание может показаться бессмысленным, но это очень базовая операция, с помощью которой можно, например, разделить данные на две части. Чем больше способов Вы знаете, тем лучше!

```
## integer(0)
```

4.4 Списки

Дан список `list_1`:

```
## $numbers
## [1] 1 2 3 4 5
##
## $letters
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
##
## $logic
## [1] FALSE
```

- Найдите первый элемент списка. Ответ должен быть списком.

```
## $numbers
## [1] 1 2 3 4 5
```

- Теперь найдите содержание первого элемента списка двумя разными способами. Ответ должен быть вектором.

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

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

Теперь возьмите первый элемент содержания первого элемента списка. Ответ должен быть вектором.

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

Создайте список `list_2`, содержащий в себе два списка `list_1` с именами `pupa` и `lupa`.

```
## $pupa
## $pupa$numbers
## [1] 1 2 3 4 5
##
## $pupa$letters
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
##
## $pupa$logic
## [1] FALSE
##
##
## $lupa
## $lupa$numbers
## [1] 1 2 3 4 5
##
```

```
## $lupa$letters
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
##
## $lupa$logic
## [1] FALSE
```

Извлеките первый элемент списка, из него - второй полэлемент, а из него - третье значение

```
## [1] "c"
```

4.5 Матрицы

- Создайте матрицу 4x4, состоящую из единиц. Назовите ее `M`

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    1    1    1
## [2,]    1    1    1    1
## [3,]    1    1    1    1
## [4,]    1    1    1    1
```

- Поменяйте все некрайние значения матрицы `M` (то есть значения на позициях [2,2], [2,3], [3,2] и [3,3]) на число 2.

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    1    1    1
## [2,]    1    2    2    1
## [3,]    1    2    2    1
## [4,]    1    1    1    1
```

- Выделите второй и третий столбик из матрицы `M`

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    2
## [3,]    2    2
## [4,]    1    1
```

- Сравните (==) вторую колонку и вторую строчку матрицы `M`

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

- Создайте таблицу умножения (9x9) в виде матрицы. Сохраните ее в переменную `tab`:

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
##  [1,]    1    2    3    4    5    6    7    8    9
##  [2,]    2    4    6    8   10   12   14   16   18
```

```
##  [3,]    3    6    9   12   15   18   21   24   27
##  [4,]    4    8   12   16   20   24   28   32   36
##  [5,]    5   10   15   20   25   30   35   40   45
##  [6,]    6   12   18   24   30   36   42   48   54
##  [7,]    7   14   21   28   35   42   49   56   63
##  [8,]    8   16   24   32   40   48   56   64   72
##  [9,]    9   18   27   36   45   54   63   72   81
```

- Из матрицы `tab` выделите подматрицу, включающую в себя только строчки с 6 по 8 и столбцы с 3 по 7.

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   18   24   30   36   42
## [2,]   21   28   35   42   49
## [3,]   24   32   40   48   56
```

- Создайте матрицу с логическими значениями, где `TRUE`, если в этом месте в таблице умножения (`tab`) двузначное число и `FALSE`, если однозначное.

  Матрица - это почти вектор. К нему можно обращаться с единственным индексом.

```
##        [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9]
##  [1,] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [2,] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [3,] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [4,] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [5,] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [6,] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [7,] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [8,] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [9,] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

- Создайте матрицу `tab2`, в которой все значения `tab` меньше 10 заменены на 0.

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
##  [1,]    0    0    0    0    0    0    0    0    0
##  [2,]    0    0    0    0   10   12   14   16   18
##  [3,]    0    0    0   12   15   18   21   24   27
##  [4,]    0    0   12   16   20   24   28   32   36
##  [5,]    0   10   15   20   25   30   35   40   45
##  [6,]    0   12   18   24   30   36   42   48   54
##  [7,]    0   14   21   28   35   42   49   56   63
##  [8,]    0   16   24   32   40   48   56   64   72
##  [9,]    0   18   27   36   45   54   63   72   81
```

4.6 Создание функций

- Создайте функцию `plus_one()`, которая принимает число и возвращает это же число + 1

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

- Создайте функцию `kvadrat()` возвращающее число в квадрате

```
## [1] 36
```

- Создайте функцию `century()`, которая превращает год в век. Возможно, понадобится вспомнить, как года переводятся в века[1].

  Здесь нужно немного погуглить.

```
century(1999:2002)
```

```
## [1] 20 20 21 21
```

- *А теперь сделайте функцию `century_roman()`, которая переводит год в век, записанный римскими цифрами!

  Здесь нужно просто немного погуглить - возможно, для создания римских цифр есть уже готовая функция? Прежде созданные функции можно использовать для создания новых функций!

```
century_roman(1999:2002)
```

```
## [1] XX  XX  XXI XXI
```

- *Напишите функцию `is_prime()`, которая проверяет, является ли число простым.

  Здесь может понадобиться оператор для получения остатка от деления: `%%`. Еще может пригодиться функция `any()` - она возвращает `TRUE`, если в векторе есть хотя бы один `TRUE`

```
is_prime(2017)
```

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

```
is_prime(2019)
```

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

[1] https://ru.wikipedia.org/wiki/

```
2019/3 #2019       3
```

```
## [1] 673
```

```
is_prime(2020)
```

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

- *Создайте функцию `monotonic()`, которая принимает возвращает `TRUE`, если значения в векторе не убывают (то есть каждое следующее - больше или равно предыдущему) или не возврастают.

```
monotonic(1:7)
```

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

```
monotonic(c(1:5,5:1))
```

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

```
monotonic(6:-1)
```

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

```
monotonic(c(1:5, rep(5, 10), 5:10))
```

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

Глава 5

Решения_заданий

5.1 Вектор

- Посчитайте логарифм от 8912162342 по основанию 6

```
log(8912162342, 6)
```

```
## [1] 12.7867
```

- Теперь натуральный логарифм 10 и умножьте его на 5

```
log(10)*5
```

```
## [1] 11.51293
```

- Создайте вектор от 1 до 20

```
1:20
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

- Создайте вектор от 20 до 1

```
20:1
```

```
##  [1] 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1
```

- Создайте вектор от 1 до 20 и снова до 1. Число 20 должно присутствовать только один раз!

```
c(1:20, 19:1)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 19 18 17 16 15
## [26] 14 13 12 11 10  9  8  7  6  5  4  3  2  1
```

- Создайте вектор 2, 4, 6, ... , 18, 20

```
seq(2,20, 2)
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

- Создайте вектор из одной единицы, двух двоек, трех троек, ... , девяти девяток

```
rep(1:9, 1:9)
```

```
##  [1] 1 2 2 3 3 3 4 4 4 4 5 5 5 5 5 6 6 6 6 6 6 7 7 7 7 7 7 7 8 8 8 8 8 8 8 8 9 9
## [39] 9 9 9 9 9 9 9
```

- Сделайте вектор vec, в котором соедините 3, а также значения " " и "   ".

```
vec <- c(3, "  ", "    ")
vec
```

```
## [1] "3"       "  "     "    "
```

- Вычесть TRUE из 10

```
10 - TRUE
```

```
## [1] 9
```

- Соедините значение 10 и TRUE в вектор vec

```
vec <- c(10, TRUE)
vec
```

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

- Соедините вектор vec и значение "r":

```
c(vec, "r")
```

```
## [1] "10" "1"  "r"
```

- Соедините значения 10, TRUE, "r" в вектор.

```
c(10, TRUE, "r")
```

```
## [1] "10"   "TRUE" "r"
```

5.2 Вектор. Операции с векторами

Создайте вектор p, состоящий из значений 4, 5, 6, 7, и вектор q, состоящий из 0, 1, 2, 3.

```
p <- 4:7
p
```

```
## [1] 4 5 6 7
```

```
q <- 0:3
q
```

```
## [1] 0 1 2 3
```

Посчитайте поэлементную сумму векторов p и q:

```
p + q
```

```
## [1]  4  6  8 10
```

Посчитайте поэлементную разницу p и q:

```
p - q
```

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

Поделите каждый элемент вектора p на соответствующий ему элемент вектора q:

О, да, Вам нужно делить на 0!

```
p/q
```

```
## [1]      Inf 5.000000 3.000000 2.333333
```

Возведите каждый элемент вектора p в степень соответствующего ему элемента вектора q:

```
p^q
```

```
## [1]   1   5  36 343
```

Создайте вектор квадратов чисел от 1 до 10:

```
(1:10)^2
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Создайте вектор 0, 2, 0, 4, ... , 18, 0, 20

```
1:20 * 0:1
```

```
##  [1]  0  2  0  4  0  6  0  8  0 10  0 12  0 14  0 16  0 18  0 20
```

5.3 Вектор. Индексирование

Создайте вектор vec1:

```
vec1 <- c(3, 5, 2, 1, 8, 4, 9, 10, 3, 15, 1, 11)
```

- Найдите второй элемент вектора vec1:

```
vec1[2]
```

```
## [1] 5
```

- Найдите последний элемент вектора vec1

```
vec1[length(vec1)]
```

```
## [1] 11
```

- Найдите все значения вектора vec1, которые больше 4

```
vec1[vec1>4]
```

```
## [1]  5  8  9 10 15 11
```

- Найдите все значения вектора vec1, которые больше 4, но меньше 10

```
vec1[vec1>4 & vec1<10]
```

```
## [1] 5 8 9
```

- Возведите в квадрат каждое значение вектора `vec1`

```
vec1^2
```

```
##  [1]   9  25   4   1  64  16  81 100   9 225   1 121
```

- Возведите в квадрат каждое значение вектора на нечетной позиции и извлеките корень из каждого значения на четной позиции вектора `vec1`

```
vec1 ^ c(2, 0.5)
```

```
##  [1]  9.000000  2.236068  4.000000  1.000000 64.000000  2.000000 81.000000
##  [8]  3.162278  9.000000  3.872983  1.000000  3.316625
```

- Создайте вектор `vec2`, в котором будут значения все значения `vec1`, которые меньше 10 будут заменены на `NA`.

```
vec2 <- vec1
vec2[vec2<10] <- NA
vec2
```

```
##  [1] NA NA NA NA NA NA NA 10 NA 15 NA 11
```

- Посчитайте сумму `vec2` с помощью функции `sum()`. Ответ `NA` не считается!

```
sum(vec2, na.rm = TRUE)
```

```
## [1] 36
```

- Создайте вектор 2, 4, 6, ... , 18, 20 как минимум 2 новыми способами

  Знаю, это задание может показаться бессмысленным, но это очень базовая операция, с помощью которой можно, например, разделить данные на две части. Чем больше способов Вы знаете, тем лучше!

```
(1:20)[c(F,T)]
```

```
## integer(0)
```

```
#(1:10)*2
```

5.4 Списки

Дан список `list_1`:

```
## $numbers
## [1] 1 2 3 4 5
##
## $letters
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
##
## $logic
## [1] FALSE
```

- Найдите первый элемент списка. Ответ должен быть списком.

```
list_1[1]
```

```
## $numbers
## [1] 1 2 3 4 5
```

- Теперь найдите содержание первого элемента списка двумя разными способами. Ответ должен быть вектором.

```
list_1[[1]]
```

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

```
list_1$numbers
```

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

Теперь возьмите первый элемент содержания первого элемента списка. Ответ должен быть вектором.

```
list_1[[1]][1]
```

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

Создайте список `list_2`, содержащий в себе два списка `list_1` с именами `pupa` и `lupa`.

```
list_2 = list(pupa = list_1, lupa = list_1)
list_2
```

```
## $pupa
## $pupa$numbers
## [1] 1 2 3 4 5
##
## $pupa$letters
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
##
## $pupa$logic
## [1] FALSE
##
##
## $lupa
## $lupa$numbers
## [1] 1 2 3 4 5
##
## $lupa$letters
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
##
## $lupa$logic
## [1] FALSE
```

Извлеките первый элемент списка, из него - второй полэлемент, а из него - третье значение

```
list_2[[1]][[2]][3]
```

```
## [1] "c"
```

5.5 Матрицы

- Создайте матрицу 4x4, состоящую из единиц. Назовите ее `M`

```
M <- matrix(rep(1, 16), ncol = 4)
M
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    1    1    1
```

```
## [2,]    1    1    1    1
## [3,]    1    1    1    1
## [4,]    1    1    1    1
```

- Поменяйте все некрайние значения матрицы `M` (то есть значения на позициях [2,2], [2,3], [3,2] и [3,3]) на число 2.

```
M[2:3, 2:3] <- 2
M
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    1    1    1
## [2,]    1    2    2    1
## [3,]    1    2    2    1
## [4,]    1    1    1    1
```

- Выделите второй и третий столбик из матрицы `M`

```
M[,2:3]
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    2
## [3,]    2    2
## [4,]    1    1
```

- Сравните (==) вторую колонку и вторую строчку матрицы `M`

```
M[,2] == M[2,]
```

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

- Создайте таблицу умножения (9x9) в виде матрицы. Сохраните ее в переменную `tab`:

```
tab <- matrix(rep(1:9, rep(9,9))*(1:9), nrow = 9)
tab
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
##  [1,]    1    2    3    4    5    6    7    8    9
##  [2,]    2    4    6    8   10   12   14   16   18
##  [3,]    3    6    9   12   15   18   21   24   27
##  [4,]    4    8   12   16   20   24   28   32   36
##  [5,]    5   10   15   20   25   30   35   40   45
##  [6,]    6   12   18   24   30   36   42   48   54
##  [7,]    7   14   21   28   35   42   49   56   63
```

```
##  [8,]    8   16   24   32   40   48   56   64   72
##  [9,]    9   18   27   36   45   54   63   72   81
```

```
#
#outer(1:9, 1:9, "*")
#1:9 %o% 1:9
```

- Из матрицы tab выделите подматрицу, включающую в себя только строчки с 6 по 8 и столбцы с 3 по 7.

```
tab[6:8, 3:7]
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   18   24   30   36   42
## [2,]   21   28   35   42   49
## [3,]   24   32   40   48   56
```

- Создайте матрицу с логическими значениями, где TRUE, если в этом месте в таблице умножения (tab) двузначное число и FALSE, если однозначное.

  Матрица - это почти вектор. К нему можно обращаться с единственным индексом.

```
tab>=10
```

```
##        [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9]
##  [1,] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [2,] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [3,] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [4,] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [5,] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [6,] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [7,] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [8,] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [9,] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

- Создайте матрицу tab2, в которой все значения tab меньше 10 заменены на 0.

```
tab2 <- tab
tab2[tab<10] <- 0
tab2
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
##  [1,]    0    0    0    0    0    0    0    0    0
```

```
##  [2,]    0    0    0    0   10   12   14   16   18
##  [3,]    0    0    0   12   15   18   21   24   27
##  [4,]    0    0   12   16   20   24   28   32   36
##  [5,]    0   10   15   20   25   30   35   40   45
##  [6,]    0   12   18   24   30   36   42   48   54
##  [7,]    0   14   21   28   35   42   49   56   63
##  [8,]    0   16   24   32   40   48   56   64   72
##  [9,]    0   18   27   36   45   54   63   72   81
```

5.6 Создание функций

- Создайте функцию `plus_one()`, которая принимает число и возвращает это же число + 1

```
plus_one <- function(x) x+1
```

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

- Создайте функцию `kvadrat()` возвращающее число в квадрате

```
kvadrat <- function(x) x*x
```

```
## [1] 36
```

- Создайте функцию `century()`, которая превращает год в век. Возможно, понадобится вспомнить, как года переводятся в века[1].

  Здесь нужно немного погуглить.

```
century <- function(x) floor((x-1)/100)+1
```

```
century(1999:2002)
```

```
## [1] 20 20 21 21
```

- *А теперь сделайте функцию `century_roman()`, которая переводит год в век, записанный римскими цифрами!

  Здесь нужно просто немного погуглить - возможно, для создания римских цифр есть уже готовая функция? Прежде созданные функции можно использовать для создания новых функций!

[1] https://ru.wikipedia.org/wiki/

```
century_roman <- function(x) as.roman(century(x))
```

```
century_roman(1999:2002)
```

```
## [1] XX  XX  XXI XXI
```

- *Напишите функцию `is_prime()`, которая проверяет, является ли число простым.

  Здесь может понадобиться оператор для получения остатка от деления: %%. Еще может пригодиться функция `any()` - она возвращает TRUE, если в векторе есть хотя бы один TRUE

```
is_prime <- function(x) !any(x%%(2:(x-1)) == 0)
```

```
is_prime(2017)
```

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

```
is_prime(2019)
```

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

```
2019/3 #2019        3
```

```
## [1] 673
```

```
is_prime(2020)
```

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

- *Создайте функцию `monotonic()`, которая принимает возвращает TRUE, если значения в векторе не убывают (то есть каждое следующее - больше или равно предыдущему) или не возврастают.

```
monotonic <- function(x) all(diff(x)>=0) | all(diff(x)<=0)
```

```
monotonic(1:7)
```

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

```
monotonic(c(1:5,5:1))
```

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

```
monotonic(6:-1)
```

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

```
monotonic(c(1:5, rep(5, 10), 5:10))
```

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

5.7 Семейство apply()

- Посчитайте, в какой из 5 книг больше всего персонажей.

```
apply(got[, 9:13], 2, sum)
```

```
## GoT CoK SoS FfC DwD
## 250 324 389 250 261
```

```
#   apply():

#colSums(got[, 9:13])

#  :

#sapply(got[,9:13], sum)
```

- Сделайте датафрейм `heroes` с персонажами, которые присутствовали во всех книгах.

```
heroes <- got[apply(got[, 9:13], 1, sum) == 5, ]

heroes
```

```
##                 Name     Allegiances Death.Year Book.of.Death Death.Chapter
## 56         Arya Stark           Stark         NA            NA            NA
## 63        Balon Swann       Lannister         NA            NA            NA
## 104     Boros Blount       Baratheon         NA            NA            NA
```

```
## 131 Cersei Lannister House Lannister         NA            NA            NA
## 302            Grenn   Night's Watch         NA            NA            NA
## 345      Harys Swyft       Lannister         NA            NA            NA
## 383  Jaime Lannister       Lannister         NA            NA            NA
## 410         Jon Snow   Night's Watch         NA            NA            NA
## 436  Kevan Lannister House Lannister        300             5            NA
## 452 Lancel Lannister       Lannister         NA            NA            NA
## 548      Meryn Trant       Lannister         NA            NA            NA
## 652          Pycelle House Lannister        300             5            NA
## 741    Samwell Tarly   Night's Watch         NA            NA            NA
##     Book.Intro.Chapter Gender Nobility GoT CoK SoS FfC DwD Is.Alive
## 56                   2      0        1   1   1   1   1   1     TRUE
## 63                  29      1        1   1   1   1   1   1     TRUE
## 104                  8      1        1   1   1   1   1   1     TRUE
## 131                  4      0        1   1   1   1   1   1     TRUE
## 302                 19      1        0   1   1   1   1   1     TRUE
## 345                 69      1        1   1   1   1   1   1     TRUE
## 383                  5      1        1   1   1   1   1   1     TRUE
## 410                  1      1        1   1   1   1   1   1     TRUE
## 436                 56      1        1   1   1   1   1   1    FALSE
## 452                 47      1        1   1   1   1   1   1     TRUE
## 548                  8      1        1   1   1   1   1   1     TRUE
## 652                 20      1        0   1   1   1   1   1    FALSE
## 741                 70      1        1   1   1   1   1   1     TRUE
```

```
#

#heroes <- got[rowSums(got[, 9:13]) == 5, ]
```

- Создайте функцию `na_n()`, которая будет возвращать количество `NA` в векторе.

```
na_n <- function(x) sum(is.na(x))
```

```
na_n(c(NA, 3:5, NA, 2, NA))
```

```
## [1] 3
```

- Посчитайте количество NA в каждом столбце `got`.

```
apply(got, 2, na_n)
```

```
##               Name        Allegiances         Death.Year      Book.of.Death
```

```
##                 0                 0               612               610
##     Death.Chapter Book.Intro.Chapter            Gender          Nobility
##               618                12                 0                 0
##               GoT               CoK               SoS               FfC
##                 0                 0                 0                 0
##               DwD         Is.Alive
##                 0                 0
```

- Есть список spisok:

```
spisok <- list(1:5, 0:20, 4:24, 6:3, 6:25)
```

- Посчитайте сумму каждого вектора.

```
sapply(spisok, sum)
```

```
## [1]  15 210 294  18 310
```

- А теперь длину.

```
sapply(spisok, length)
```

```
## [1]  5 21 21  4 20
```

- Напишите функцию max_item(), которая будет принимать на входе список, а возвращать - (первый) самый длинный его элемент.

```
max_item <- function (x) spisok[[which.max(sapply(x, length))]]
```

- Теперь мы сделаем сложный список:
- Посчитайте длину каждого вектора в списке, в т.ч. для списка внутри

  Для этого может понадобиться функция rapply(): **recursive lapply**

```
rapply(large_spisok, length, how = "list")
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] 38
##
## [[3]]
## [[3]][[1]]
```

```
## [1] 5
##
## [[3]][[2]]
## [1] 21
##
## [[3]][[3]]
## [1] 21
##
## [[3]][[4]]
## [1] 4
##
## [[3]][[5]]
## [1] 20
```

Литература

Adler, J. (2010). *R in a nutshell: A desktop quick reference*. "O'Reilly Media, Inc.".

Baesens, B., Van Vlasselaer, V., and Verbeke, W. (2015). *Fraud analytics using descriptive, predictive, and social network techniques: a guide to data science for fraud detection*. John Wiley & Sons.

Brooks, H. and Cooper, C. L. (2013). *Science for public policy*. Elsevier.

Hansjörg, N. (2019). *Data Science for Psychologists*. self published.

Provost, F. and Fawcett, T. (2013). *Data Science for Business: What you need to know about data mining and data-analytic thinking*. O'Reilly Media, Inc.

R Core Team (2019). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Thomas, N. and Pallett, L. (2019). *Data Science for Immunologists*. CreateSpace Independent Publishing Platform.

Wickham, H. and Grolemund, G. (2016). *R for data science: import, tidy, transform, visualize, and model data*. O'Reilly Media, Inc.