

Наука о данных в R для программы Цифровых гуманитарных исследований

Г. А. Мороз, И. С. Поздняков

Оглавление

1	О курсе	5
2	Введение в R	7
2.1	Наука о данных	7
2.2	Установка R и RStudio	8
2.3	Полезные ссылки	9
2.4	Rstudio	9
2.5	Введение в R	10
2.6	Типы данных	18
2.7	Вектор	21
2.8	Матрицы (matrix)	35
2.9	Списки (list)	39
2.10	Датафрейм	42
3	Импорт данных	45
3.1	Рабочая папка и проекты RStudio	45
3.2	Проверка импортированных данных	49
3.3	Экспорт данных	51
3.4	Импорт таблиц в бинарном формате: таблицы Excel, SPSS	52
3.5	Быстрый импорт данных	52
4	Условные конструкции и циклы	57
4.1	Выражения if, else, else if	57
4.2	Циклы for	59
4.3	Векторизованные условные конструкции: функции ifelse() и dplyr::case_when()	61
5	Функциональное программирование в R	63
5.1	Создание функций	63
5.2	Когда и зачем создавать функции?	65
5.3	Функции как объекты первого порядка	65
5.4	Семейство функций apply()	66
6	Задания	73

6.1	Вектор	73
6.2	Вектор. Операции с векторами	74
6.3	Вектор. Индексирование	75
6.4	Списки	76
6.5	Матрицы	77
6.6	Создание функций	79
7	Решения_заданий	81
7.1	Вектор	81
7.2	Вектор. Операции с векторами	83
7.3	Вектор. Индексирование	84
7.4	Списки	86
7.5	Матрицы	87
7.6	Создание функций	90
7.7	Семейство <code>apply()</code>	92

Глава 1

О курсе

Материалы для курса Наука о данных для магистерской программы Цифровых гуманитарных 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 находится 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 Windows¹
 - на Mac²
 - на Linux³, также можно добавить зеркало и установить из командной строки:

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

- RStudio — IDE для R (можно скачать здесь⁴)
- 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 cloud⁵ — полная функциональность RStudio, пока бесплатная, но скоро это исправят;
- RStudio on rollApp⁶ — 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 tutorиалы 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](https://stackoverflow.com)⁸ — сервис, где достаточно быстро отвечают на любые вопросы (не обязательно по R)
- [RStudio community](https://community.rstudio.com/)⁹ — быстро отвечают на вопросы, связанные с R
- [русский stackoverflow](https://ru.stackoverflow.com)¹⁰
- [R-bloggers](http://r-bloggers.com)¹¹ — сайт, где собираются новинки, связанные с R
- чат¹², где можно спрашивать про R на русском (но почитайте правила чата¹³, перед тем как спрашивать)
- чат¹⁴ по визуализации данных, чат¹⁵ датажурналистов
- канал про визуализацию¹⁶, 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/>

¹²https://t.me/r_lang_ru

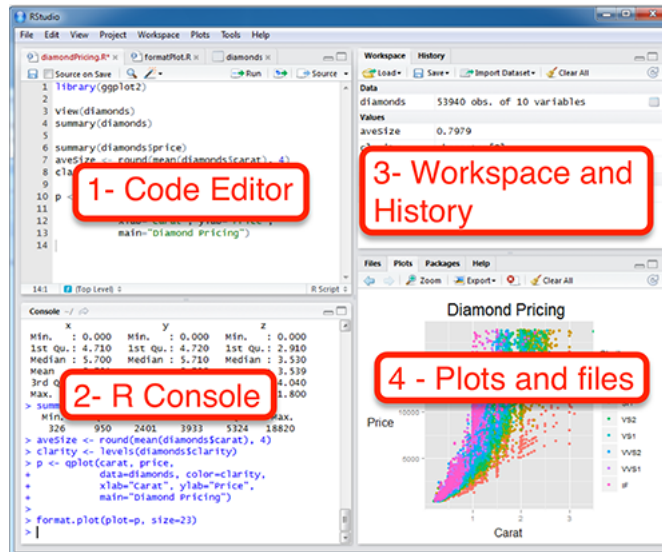
¹³<https://github.com/r-lang-group-ru/group-rules/blob/master/README.md>

¹⁴<https://t.me/joinchat/CxZg5goGc6rlWGjcv0YrpA>

¹⁵<https://t.me/ddjrus>

¹⁶<https://t.me/chartomojka>

¹⁷https://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 (внутри 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

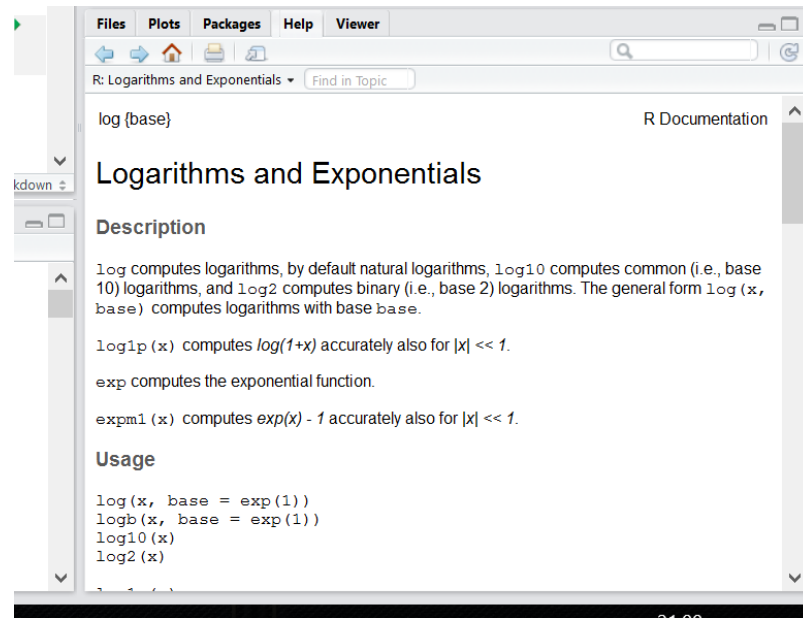


Рис. 2.2

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 Mac).

```
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
```

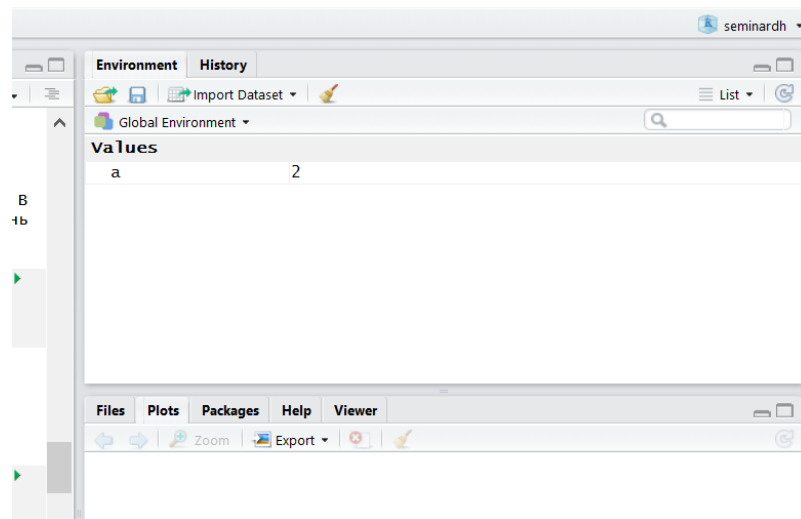


Рис. 2.3

```
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 *неравенство*:

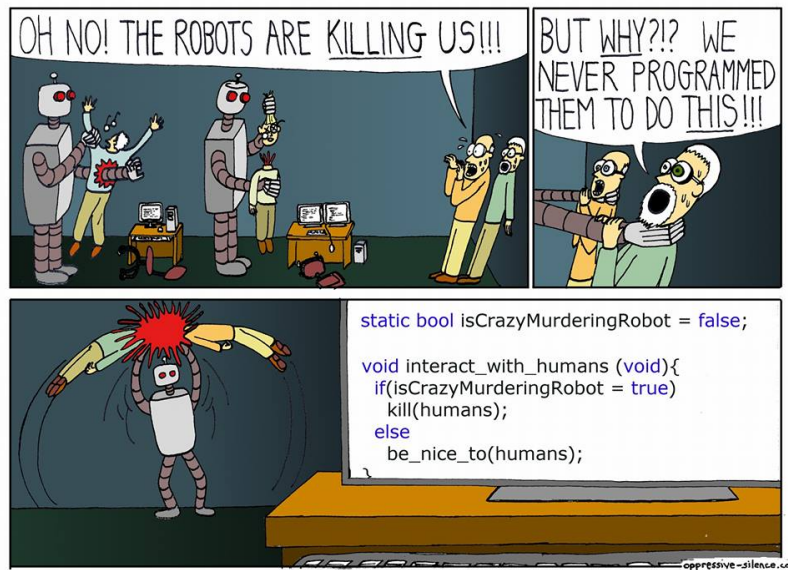


Рис. 2.4

```
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 конвертировать между форматами при необходимости. Немного подробностей здесь:

Разница между numeric и integer¹⁸, Как работать с комплексными числами в R¹⁹

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 Т 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 Т 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 понадобится 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, $\{2,3\}$:

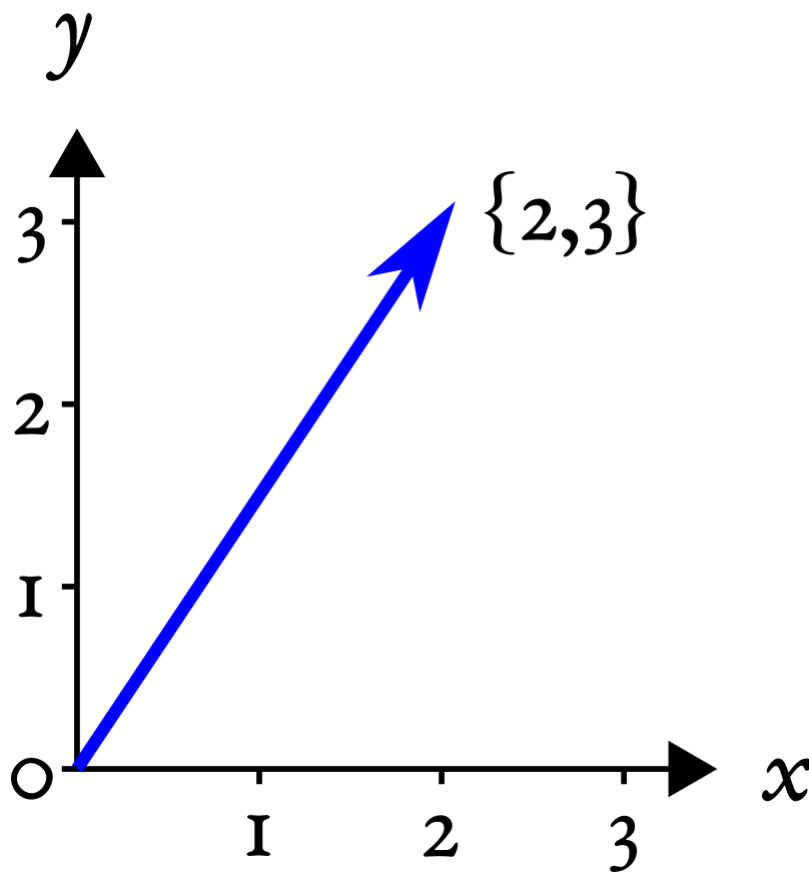


Рис. 2.5

Вот последние два числа и будем считать вектором. Попробуйте 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 \mathbb{R} , но, возможно, Вы об этом даже не догадывались. Дело в том, что в \mathbb{R} нет как таковых “значений”, есть вектора длиной 1. Такие дела!

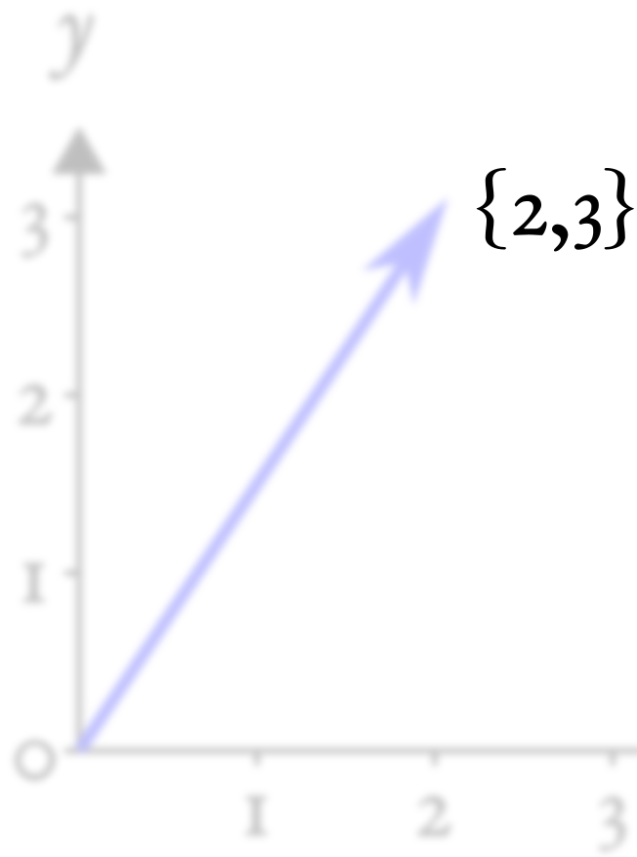


Рис. 2.6

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 с 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 произойдет 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 строки "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 соответственно. 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` 4.2. Не спешите этого делать! В очень многих случаях циклы можно заменить векторизацией. Тем не менее, векторизация — это не единственный способ избавиться от циклов типа `for` и `while` 5.4.1.

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 повторяться 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. здесь²⁰. 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 научитесь 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 `o` — 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 обеих 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!

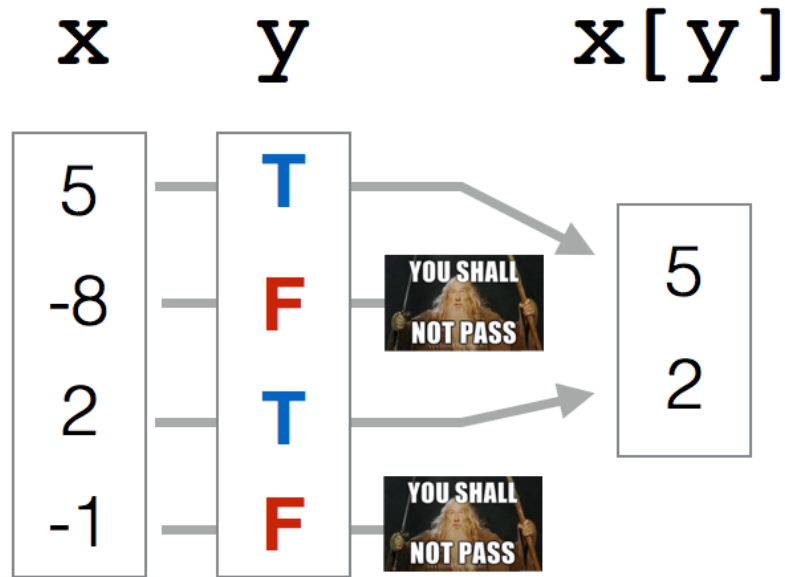


Рис. 2.7

```
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 а 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
```

Результат выполнения `is.na(n)` выдает `FALSE` в тех местах, где у нас числа и `TRUE` там, где у нас `NA`. Чтобы вычлениить 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.](https://twitter.com/mousquemere/status/1125522375141883907?ref_src=twsrc%5Etfw) 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

**Doctors: Googling stuff online does not
make you a doctor.
Programmers:**



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
```

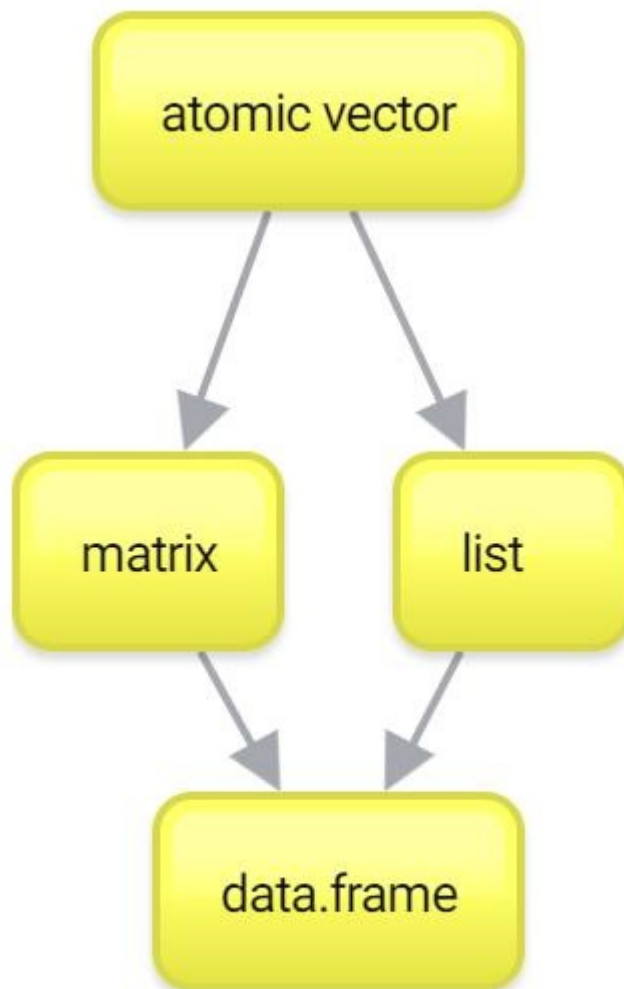


Рис. 2.10

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
```

Если мы выберем только одну колонку/строчку, то на выходе получим уже вектор, а не матрицу:

```
A[2,]
```

```
## [1]  2  7 12 17
```

Это называется “схлопыванием размерности”. Чтобы этого избежать, нужно поставить `drop = FALSE` после второй запятой внутри квадратных скобок.

```
A[2,, drop = FALSE]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    7   12   17
```

Для соединения двух или более матриц можно воспользоваться функциями `rbind()` и `cbind()` для соединения матриц по вертикали и по горизонтали соответственно.

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

```
cbind(A, A)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]    1    6   11   16    1    6   11   16
## [2,]    2    7   12   17    2    7   12   17
## [3,]    3    8   13   18    3    8   13   18
## [4,]    4    9   14   19    4    9   14   19
## [5,]    5   10   15   20    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 свой 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 здесь²¹ 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
##
## [[2]]
```

²¹<https://perso.esiee.fr/~courivad/R/06-objects.html>

```
## [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](https://twitter.com/YQ6axb2w7t)

— Hadley Wickham (@ [href="https://twitter.com/hadleywickham/status/643381054758363136?ref_src=twsrc%5Etfw"](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 иметь 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.

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 супергероя¹, 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 ссылке²

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
```

¹супергерои 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.

²https://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”):

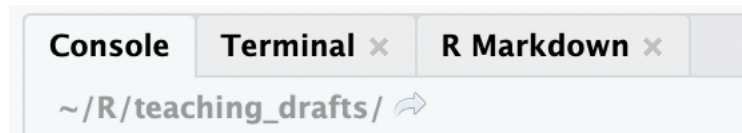


Рис. 3.1

- 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 запуск скрипта 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/heroes.csv")
```

- 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 отказаться³. 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 справиться, 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 увидите 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 файле 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 деления 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" "Ichthyo 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. Попробуйте 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 разобрать⁴, 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 Экспорт данных

Представим, что вы хотите сохранить табличку с данными про супергероев из вселенной DC в виде отдельного файла `.csv`.

```
dc <- heroes[heroes$Publisher == "DC Comics",]
```

Функция `write.csv()` позволит записать датафрейм в файл формата `.csv`:

```
write.csv(dc, "data/dc_heroes_information.csv")
```

Обычно названия строк не используются, и их лучше не записывать, поставив для `row.names` = значение `FALSE`:

```
write.csv(dc, "data/dc_heroes_information.csv", row.names = FALSE)
```

По аналогии с `read.csv2()`, `write.csv2()` позволит записать файлы формата `.csv` с разделителем `,`:

```
write.csv2(dc, "data/dc_heroes_information.csv", row.names = FALSE)
```

3.4 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 попытаетесь открыть .xlsx файл в Блокноте, то увидите кракозябры. Это делает работу с этими 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.

3.5 Быстрый импорт данных

Чтение табличных данных обычно происходит очень быстро. По крайней мере, до тех пор пока ваши данные не содержат очень много значений. Если вы попытаетесь прочитать с помощью `read.csv()` таблицу с миллионами строчками, то заметите, что это происходит довольно медленно. Впрочем, эта проблема эффективно решается дополнительными пакетами.

- Пакет `readr` (часть базового `tidyverse`) предлагает функции, очень похожие на стандартные `read.csv()`, `read.csv2()` и тому подобные, только в названиях используется нижнее подчеркивание: `read_csv()` и `read_csv2()`. Они быстрее и немного удобнее, особенно если вы работаете в `tidyverse`.

```
readr::read_csv("data/heroes_information.csv",  
               na = c("-", "-99"))
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
## Parsed with column specification:
```

```
## cols(
```

```
##   X1 = col_double(),
```

```
##   name = col_character(),
```

```
##   Gender = col_character(),
```

```
##   `Eye color` = col_character(),
```

```
## Race = col_character(),
## `Hair color` = col_character(),
## Height = col_double(),
## Publisher = col_character(),
## `Skin color` = col_character(),
## Alignment = col_character(),
## Weight = col_double()
## )

## # A tibble: 734 x 11
##       X1 name Gender `Eye color` Race `Hair color` Height Publisher
##   <dbl> <chr> <chr> <chr> <chr> <chr> <dbl> <chr>
## 1     0 A-Bo~ Male yellow Human No Hair 203 Marvel C~
## 2     1 Abe ~ Male blue Icth~ No Hair 191 Dark Hor~
## 3     2 Abin~ Male blue Unga~ No Hair 185 DC Comics
## 4     3 Abom~ Male green Huma~ No Hair 203 Marvel C~
## 5     4 Abra~ Male blue Cosm~ Black NA Marvel C~
## 6     5 Abso~ Male blue Human No Hair 193 Marvel C~
## 7     6 Adam~ Male blue <NA> Blond NA NBC - He~
## 8     7 Adam~ Male blue Human Blond 185 DC Comics
## 9     8 Agen~ Female blue <NA> Blond 173 Marvel C~
## 10    9 Agen~ Male brown Human Brown 178 Marvel C~
## # ... with 724 more rows, and 3 more variables: `Skin color` <chr>,
## # Alignment <chr>, Weight <dbl>
```

- Пакет `vroom` - это часть расширенного `tidyverse`. Это такая альтернатива `readr` из того же `tidyverse`, но еще быстрее (отсюда и название).

```
vroom::vroom("data/heroes_information.csv")
```

```
## New names:
## * `` -> ...1

## Rows: 734
## Columns: 11
## Delimiter: ","
## chr [8]: name, Gender, Eye color, Race, Hair color, Publisher, Skin color, Alignment
## dbl [3]: ...1, Height, Weight
##
## Use `spec()` to retrieve the guessed column specification
## Pass a specification to the `col_types` argument to quiet this message

## # A tibble: 734 x 11
##       ...1 name Gender `Eye color` Race `Hair color` Height Publisher
##   <dbl> <chr> <chr> <chr> <chr> <chr> <dbl> <chr>
## 1     0 A-Bo~ Male yellow Human No Hair 203 Marvel C~
## 2     1 Abe ~ Male blue Icth~ No Hair 191 Dark Hor~
```

```
## 3      2 Abin~ Male   blue      Unga~ No Hair      185 DC Comics
## 4      3 Abom~ Male  green     Huma~ No Hair      203 Marvel C~
## 5      4 Abra~ Male  blue      Cosm~ Black        -99 Marvel C~
## 6      5 Abso~ Male  blue      Human No Hair      193 Marvel C~
## 7      6 Adam~ Male  blue      -      Blond        -99 NBC - He~
## 8      7 Adam~ Male  blue      Human Blond      185 DC Comics
## 9      8 Agen~ Female blue      -      Blond        173 Marvel C~
## 10     9 Agen~ Male  brown     Human Brown      178 Marvel C~
## # ... with 724 more rows, and 3 more variables: `Skin color` <chr>,
## #   Alignment <chr>, Weight <dbl>
```

- Пакет `data.table` - это не просто пакет, а целый фреймворк для работы с R, основной конкурент `tidyverse`. Одна из основных фишек `data.table` - быстрота работы. Это касается не только процессинга данных, но и их загрузки и записи. Поэтому некоторые используют функции `data.table` для чтения и записи данных в отдельности от всего остального пакета - они даже и называются соответствующие: `fread()` и `fwrite()`, где `f` означает `fast`⁵.

```
data.table::fread("data/heroes_information.csv")
```

```
##      V1      name Gender Eye color      Race      Hair color
## 1:  0      A-Bomb  Male   yellow      Human      No Hair
## 2:  1      Abe Sapien Male    blue  Ichthyo Sapien      No Hair
## 3:  2      Abin Sur  Male    blue      Ungaran      No Hair
## 4:  3  Abomination Male   green Human / Radiation      No Hair
## 5:  4      Abraxas  Male    blue  Cosmic Entity      Black
## ---
## 730: 729 Yellowjacket II Female    blue      Human Strawberry Blond
## 731: 730      Ymir   Male    white  Frost Giant      No Hair
## 732: 731      Yoda   Male    brown  Yoda's species      White
## 733: 732      Zatanna Female    blue      Human      Black
## 734: 733      Zoom   Male     red      -      Brown
##      Height      Publisher Skin color Alignment Weight
## 1: 203.0      Marvel Comics      -      good      441
## 2: 191.0 Dark Horse Comics    blue      good      65
## 3: 185.0      DC Comics      red      good      90
## 4: 203.0      Marvel Comics      -      bad      441
## 5: -99.0      Marvel Comics      -      bad     -99
## ---
## 730: 165.0      Marvel Comics      -      good      52
## 731: 304.8      Marvel Comics    white      good     -99
## 732:  66.0      George Lucas    green      good      17
## 733: 170.0      DC Comics      -      good      57
```

⁵А еще `friendly`: `fread()` обычно самостоятельно хорошо угадывает формат таблицы на входе. `vroom` тоже так умеет.

```
## 734: 185.0          DC Comics          -          bad      81
```

Чем же пользоваться среди всего этого многообразия? Бенчмарки⁶⁷ показывают, что быстрее всех `vroom` и `data.table`. Если же у вас нет задачи ускорить работу кода на несколько миллисекунд или прочитать датасет на много миллионов строк, то стандартного `read.csv()` (если вы работаете в базовом R) и `readr::read_csv()` (если вы работаете в `tidyverse`) должно быть достаточно.

Все перечисленные пакеты позволяют не только быстро импортировать данные, но и быстро (и удобно!) экспортировать их:

```
readr::write_csv(dc, "data/dc_heroes_information.csv")
readr::write_excel_csv(dc, "data/dc_heroes_information.csv") # Excel
vroom::vroom_write(dc, "data/dc_heroes_information.csv", delim = ",")
data.table::fwrite(dc, "data/dc_heroes_information.csv")
```

В плане скорости записи файлов соотношение сил примерно такое же, как и для чтения: `vroom` и `data.table` обгоняют всех, затем идет `readr`, и только после него - базовые функции R.

⁶<https://www.danielecook.com/speeding-up-reading-and-writing-in-r/>

⁷бенчмаркинг — это тест производительности, в данном случае — сравнение скорости работы конкурирующих пакетов.

Глава 4

Условные конструкции и циклы

4.1 Выражения `if`, `else`, `else if`

Стандартная часть практически любого языка программирования — условные конструкции. R не исключение. Однако и здесь есть свои особенности. Начнем с самого простого варианта с одним условием. Выглядеть условная конструкция будет вот так:

```
if ( )
```

Вот так это будет работать на практике:

```
number <- 1
if (number > 0) "положительное"
```

```
## [1] "положительное"
```

Если выражение (`expression`) содержит больше одной строки, то они объединяются фигурными скобками. Впрочем, использовать их можно, даже если строка всего в выражении всего одна.

```
number <- 1
if (number > 0) {
  "положительное"
}
```

```
## [1] "положительное"
```

В рассмотренной нами конструкции происходит проверка на условие. Если условие верно¹, то происходит то, что записано в последующем выражении. Если же условие неверно², то ничего не происходит.

Оператор `else` позволяет задавать действие на все остальные случаи:

```
if ( ) else
```

Работает это так:

```
number <- -3
if (number > 0) {
  "
} else {
  "
}
```

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

Иногда нам нужна последовательная проверка на несколько условий. Для этого есть оператор `else if`. Вот как выглядит ее применение:

```
number <- 0
if (number > 0) {
  "
} else if (number < 0){
  "
} else {
  " "
}
```

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

Как мы помним, R — язык, в котором векторизация играет большое значение. Но вот незадача — условные конструкции не векторизованы в R! Давайте попробуем применить эти конструкции для вектора значений и посмотрим, что получится.

```
number <- -2:2
if (number > 0) {
  "
} else if (number < 0){
  "
}
```

¹В принципе, необязательно внутри должна быть проверка условий, достаточно просто значения TRUE.

²Аналогично, достаточно просто значения FALSE.

```

} else {
    " "
}

```

```

## Warning in if (number > 0) {:          > 1,
##

```

```

## Warning in if (number < 0) {:          > 1,
##

```

```

## [1] " "

```

R выдает сообщение, что используется только первое значение логического вектора внутри условия. Остальные просто игнорируются. Как же посчитать для всего вектора сразу?

4.2 Циклы for

Во-первых, можно использовать `for`. Синтаксис `for` похож на синтаксис условных конструкций.

```
for(      in      )
```

Теперь мы можем объединить условные конструкции и `for`. Немножко монструозно, но это работает:

```

for (i in number) {
  if (i > 0) {
    print(" " )
  } else if (i < 0) {
    print(" " )
  } else {
    print(" ")
  }
}

```

```

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

```

Чтобы выводить в консоль результат вычислений внутри `for`, нужно использовать `print()`.

Здесь стоит отметить, что `for` используется в R относительно редко. В подавляющем числе ситуаций использование `for` можно избежать. Обычно мы работаем в R с векторами или датафреймами, которые представляют собой множество относительно независимых наблюдений. Если мы хотим провести какие-нибудь операции с этими наблюдениями, то они обычно могут быть выполнены параллельно. Скажем, вы хотите для каждого испытуемого пересчитать его массу из фунтов в килограммы. Этот пересчет осуществляется по одинаковой формуле для каждого испытуемого. Эта формула не изменится из-за того, что какой-то испытуемый слишком большой или слишком маленький - для следующего испытуемого формула будет прежняя. Если Вы встречаете подобную задачу (где функцию можно применить независимо для всех значений), то без цикла `for` вполне можно обойтись.

Даже во многих случаях, где расчеты для одной строки зависят от расчетов предыдущих строк, можно обойтись без `for` векторизованными функциями, например, `cumsum()` для подсчета кумулятивной суммы.

```
cumsum(1:10)
```

```
## [1] 1 3 6 10 15 21 28 36 45 55
```

Если же нет подходящей векторизованной функции, то можно воспользоваться семейством функций `apply()` (см. [@ref\(apply_f\)](#)).

После этих объяснений кому-то может показаться странным, что я вообще упоминаю про эти циклы. Но для кого-то циклы `for` настолько привычны, что их полное отсутствие в курсе может показаться еще более странным. Поэтому лучше от меня, чем на улице.

Зачем вообще избегать конструкций `for`? Некоторые говорят, что они слишком медленные, и частично это верно, если мы сравниваем с векторизованными функциями, которые написаны на более низкоуровневых языках. Но в большинстве случаев низкая скорость `for` связана с неправильным использованием этой конструкции. Например, стоит избегать ситуации, когда на каждой итерации `for` какой-то объект (вектор, список, что угодно) изменяется в размере. Лучше будет создать заранее объект нужного размера, который затем будет наполняться значениями:

```
number_descriptions <- character(length(number)) #
for (i in 1:length(number)) {
  if (number[i] > 0) {
    number_descriptions[i] <- "          "
  } else if (number[i] < 0) {
    number_descriptions[i] <- "          "
  } else {
    number_descriptions[i] <- "  "
  }
}
```

```

    }
  }
  number_descriptions

```

```

## [1] "          " "          " "          "
## [4] "          " "          " "          "

```

В общем, при правильном обращении с `for` особых проблем со скоростью не будет. Но все равно это будет громоздкая конструкция, в которой легко ошибиться, и которую, скорее всего, можно заменить одной короткой строчкой. Кроме того, без конструкции `for` код обычно легко превратить в набор функций, последовательно применяющихся к данным, что мы будем по максимуму использовать, работая в `tidyverse` и применяя пайпы (см. [pipe]).

4.3 Векторизованные условные конструкции: функции `ifelse()` и `dplyr::case_when()`

Альтернатива сочетанию условных конструкций и циклов `for` является использование встроенной функции `ifelse()`. Функция `ifelse()` принимает три аргумента - 1) условие (т.е. просто логический вектор, состоящий из `TRUE` и `FALSE`), 2) что выдавать в случае `TRUE`, 3) что выдавать в случае `FALSE`. На выходе получается вектор такой же длины, как и изначальный логический вектор (условие).

```

ifelse(number > 0, "          ", "          ")

```

```

## [1] "          " "          " "          "
## [3] "          " "          " "          "
## [5] "          " "          " "          "

```

Периодически я встречаю у студентов строчку вроде такой: `ifelse(, TRUE, FALSE)`. Эта конструкция избыточна, т.к. получается, что логический вектор из `TRUE` и `FALSE` превращается в абсолютно такой же вектор из `TRUE` и `FALSE` на тех же самых местах. Выходит, что ничего не меняется!

У `ifelse()` тоже есть недостаток: он не может включать в себя дополнительных условий по типу `else if`. В простых ситуациях можно вставлять `ifelse()` внутри `ifelse()`:

```

ifelse(number > 0,
  "          ",
  ifelse(number < 0, "          ", "          "))

```

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

Достаточно симпатичное решение предлагает пакет `dplyr` (основа `tidyverse`) — функция `case_when()`, которая работает с использованием формулы:

```
dplyr::case_when(
  number > 0 ~ "      ",
  number < 0 ~ "      ",
  number == 0 ~ " " " ")
```

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

Глава 5

Функциональное программирование в R

5.1 Создание функций

Поздравляю, сейчас мы выйдем на качественно новый уровень владения R. Вместо того, чтобы пользоваться теми функциями, которые уже написали за нас, мы можем сами создавать свои функции! В этом нет ничего сложного.

Синтаксис создания функции внешне похож на создание циклов или условных конструкций. Мы пишем ключевое слово `function`, в круглых скобках обозначаем переменные, с которыми собираемся что-то делать. Внутри фигурных скобок пишем выражения, которые будут выполняться при запуске функции. У функции есть свое собственное окружение — место, где хранятся переменные. Именно те объекты, которые мы передаем в скобочках, и будут в окружении, так же как и “обычные” переменные для нас в глобальном окружении. Это означает, что функция будет искать переменные в первую очередь среди объектов, которые переданы в круглых скобочках. С ними функция и будет работать. На выходе функция выдаст то, что вычисляется внутри функции `return()`. Если `return()` появляется в теле функции несколько раз, то до результат будет возвращаться из той функции `return()`, до которой выполнение дошло первым.

```
pow <- function(x, p) {  
  power <- x ^ p  
  return(power)  
}  
pow(3, 2)
```

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

Если функция проработала до конца, а функция `return()` так и не встретилась, то возвращается последнее посчитанное значение.

```
pow <- function(x, p) {
  x ^ p
}
pow(3, 2)
```

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

Если в последней строчке будет присвоение, то функция ничего не вернет обратно. Это очень распространенная ошибка: функция вроде бы работает правильно, но ничего не возвращает. Нужно писать так, как будто бы в последней строчке результат выполнения выводится в консоль.

```
pow <- function(x, p) {
  power <- x ^ p #
}
pow(3, 2) #
```

Если функция небольшая, то ее можно записать в одну строчку без фигурных скобок.

```
pow <- function(x, p) x ^ p
pow(3, 2)
```

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

Вообще, фигурные скобки используются для того, чтобы выполнить серию выражений, но вернуть только результат выполнения последнего выражения. Это можно использовать, чтобы не создавать лишних временных переменных в глобальном окружении.

Мы можем оставить в функции параметры по умолчанию.

```
pow <- function(x, p = 2) x ^ p
pow(3)
```

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

```
pow(3, 3)
```

```
## [1] 27
```


В R работают **ленивые вычисления** (*lazy evaluations*). Это означает, что параметры функций будут только когда они понадобятся, а не заранее. Например, эта функция не будет выдавать ошибку, если мы не зададим параметр `we_will_not_use_this_parameter =`, потому что он нигде не используется в расчетах.

```
pow <- function(x, p = 2, we_will_not_use_this_parameter) x ^ p
pow(x = 3)
```

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

5.2 Когда и зачем создавать функции?

Когда стоит создавать функции? Существует “правило трех”¹ — если у вас есть три куска очень похожего кода, то самое время превратить код в функцию. Это очень условное правило, но, действительно, стоит избегать копипастинга в коде. В этом случае очень легко ошибиться, а сам код становится нечитаемым.

Есть и другой подход к созданию функций: их стоит создавать не столько для того, чтобы использовать тот же код снова, сколько для абстрагирования от того, что происходит в отдельных строчках кода. Если несколько строчек кода были написаны для того, чтобы решить одну задачу, которой можно дать понятное название (например, подсчет какой-то особенной метрики, для которой нет готовой функции в R), то этот код стоит обернуть в функцию. Если функция работает корректно, то теперь не нужно думать над тем, что происходит внутри нее. Вы ее можете мысленно представить как операцию, которая имеет определенный вход и выход — как и встроенные функции в R.

Отсюда следует важный вывод, что хорошее название для функции — это очень важно. Очень, очень, очень важно.

5.3 Функции как объекты первого порядка

Ранее мы убедились, что арифметические операторы — это тоже функции. На самом деле, практически все в R — это функции. Даже `function` — это функция `function()`. Даже скобочки `()`, `{` — это функции!

А сами функции — это объекты первого порядка в R. Это означает, что с функциями вы можете делать практически все то же самое, что и с другими объектами в R (векторами, датафреймами и т.д.). Небольшой пример, который может взорвать ваш мозг:

¹[https://en.wikipedia.org/wiki/Rule_of_three_\(computer_programming\)](https://en.wikipedia.org/wiki/Rule_of_three_(computer_programming))

```
list(mean, min, `{`)
```

```
## [[1]]
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x7f96827c8d48>
## <environment: namespace:base>
##
## [[2]]
## function (... , na.rm = FALSE) .Primitive("min")
##
## [[3]]
## .Primitive("{")
```

Мы можем создать список из функций! Зачем — это другой вопрос, но ведь можем же!

Еще можно создавать функции внутри функций², использовать функции в качестве аргументов функций, сохранять функции как переменные. Пожалуй, самое важное из этого всего - это то, что функция может быть аргументом в функции. Не просто название функции как строковая переменная, не результат выполнения функции, а именно сама функция. Это лежит в основе использования семейства функций `apply()` @ref(apply_f) и многих фишек tidyverse.

В Python дело обстоит похожим образом: функции там тоже являются объектами первого порядка, поэтому все эти фишки функционального программирования (с поправкой на синтаксис, конечно) будут работать и там.

5.4 Семейство функций `apply()`

5.4.1 Применение `apply()` для матриц

Семейство? Да, их целое множество: `apply()`, `lapply()`, `sapply()`, `vapply()`, `tapply()`, `mapply()`, `rapply()`... Ладно, не пугайтесь, всех их знать не придется. Обычно достаточно первых двух-трех. Проще всего пояснить как они работают на простой матрице с числами:

```
A <- matrix(1:12, 3, 4)
A
```

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

²Функция, которая создает другие функции, называется фабрикой функций.

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

Функция `apply()` предназначена для работы с матрицами (или многомерными массивами). Если вы скормите функции `apply()` датафрейм, то этот датафрейм будет сначала превращен в матрицу. Главное отличие матрицы от датафрейма в том, что в матрице все значения одного типа, поэтому будьте готовы, что сработает имплицитное приведение к общему типу данных. Например, если среди колонок датафрейма есть хотя бы одна строковая колонка, то все колонки станут строковыми.

Теперь представим, что нам нужно посчитать что-нибудь (например, сумму) по каждой из строк. С помощью функции `apply()` вы можете в буквальном смысле “применить” функцию к матрице или датафрейму. Синтаксис такой: `apply(X, MARGIN, FUN, ...)`, где `X` — данные, `MARGIN` это 1 (для строк), 2 (для колонок), `c(1,2)` для строк и колонок (т.е. для каждого элемента по отдельности), а `FUN` — это функция, которую вы хотите применить! `apply()` будет брать строки/колонки из `X` в качестве первого аргумента для функции.

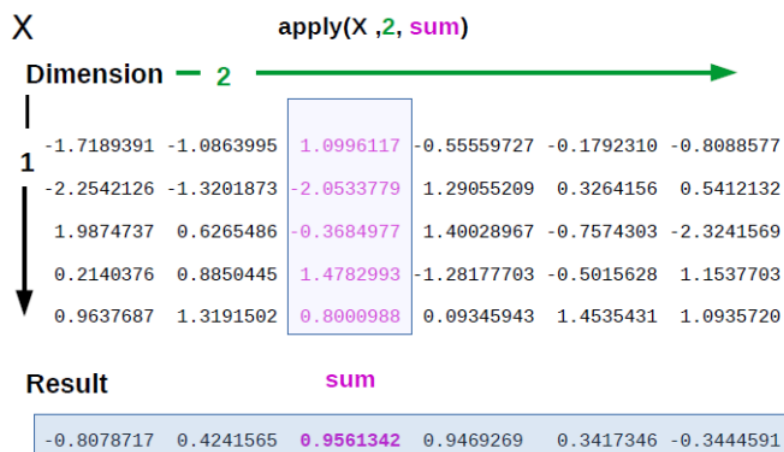


Рис. 5.1: apply

Заметьте, мы вставляем функцию без скобок и кавычек как аргумент в функцию. Это как раз тот случай, когда аргументом в функции выступает сама функция, а не ее название или результат ее выполнения.

Давайте разберем на примере:

```
apply(A, 1, sum) #
```

```
## [1] 22 26 30
```

```
apply(A, 2, sum) #
```

```
## [1] 6 15 24 33
```

```
apply(A, c(1,2), sum) # ...
```

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

Конкретно для подсчета сумм и средних по столбцам и строкам в R есть функции `colSums()`, `rowSums()`, `colMeans()` и `rowMeans()`, которые можно использовать как альтернативы `apply()` в данном случае.

Если же мы хотим прописать дополнительные аргументы для функции, то их можно перечислить через запятую после функции:

```
apply(A, 1, sum, na.rm = TRUE)
```

```
## [1] 22 26 30
```

```
apply(A, 1, weighted.mean, w = c(0.2, 0.4, 0.3, 0.1))
```

```
## [1] 4.9 5.9 6.9
```

5.4.2 Анонимные функции

Что делать, если мы хотим сделать что-то более сложное, чем просто применить одну функцию? А если функция принимает не первым, а вторым аргументом данные из матрицы? В этом случае нам помогут **анонимные функции**.

Анонимные функции - это функции, которые будут использоваться один раз и без названия.

Питонистам знакомо понятие **лямбда-функций**. Да, это то же самое.

Например, мы можем посчитать отклонения от среднего без называния этой функции:

```
apply(A, 1, function(x) x - mean(x)) #
```

```
##      [,1] [,2] [,3]
## [1,] -4.5 -4.5 -4.5
## [2,] -1.5 -1.5 -1.5
## [3,]  1.5  1.5  1.5
## [4,]  4.5  4.5  4.5
```

```
apply(A, 2, function(x) x - mean(x)) #
```

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

```
apply(A, c(1,2), function(x) x - mean(x)) #
```

```
, . .
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   0    0    0    0
## [2,]   0    0    0    0
## [3,]   0    0    0    0
```

Как и в случае с обычной функцией, в качестве `x` выступает объект, с которым мы хотим что-то сделать, а дальше следует функция, которую мы собираемся применить к `.` Можно использовать не `.`, а что угодно, как и в обычных функциях:

```
apply(A, 1, function(whatevername) whatevername - mean(whatevername))
```

```
##      [,1] [,2] [,3]
## [1,] -4.5 -4.5 -4.5
## [2,] -1.5 -1.5 -1.5
## [3,]  1.5  1.5  1.5
## [4,]  4.5  4.5  4.5
```

5.4.3 Другие функции семейства `apply()`

Ок, с `apply()` разобрались. А что с остальными? Некоторые из них еще проще и не требуют индексов, например, `lapply` (для применения к каждому элементу списка) и `sapply` - упрощенная версия `lapply()`, которая пытается по возможности “упростить” результат до вектора или матрицы.

```
some_list <- list(some = 1:10, list = letters)
lapply(some_list, length)
```

```
## $some
## [1] 10
##
## $list
## [1] 26
```

```
sapply(some_list, length)
```

```
## some list
## 10 26
```

Использование `sapply()` на векторе приводит к тем же результатам, что и просто применить векторизованную функцию обычным способом.

```
sapply(1:10, sqrt)
```

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

```
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
```

Зачем вообще тогда нужен `sapply()`, если мы можем просто применить векторизованную функцию? Ключевое слово здесь *векторизованная* функция. Если функция не векторизована, то `sapply()` становится удобным вариантом для того, чтобы избежать итерирования с помощью циклов `for`.

Еще одна альтернатива - это векторизация не векторизованной функции с помощью `Vectorize()`. Эта функция просто оборачивает функцию одним из вариантов `apply()`.

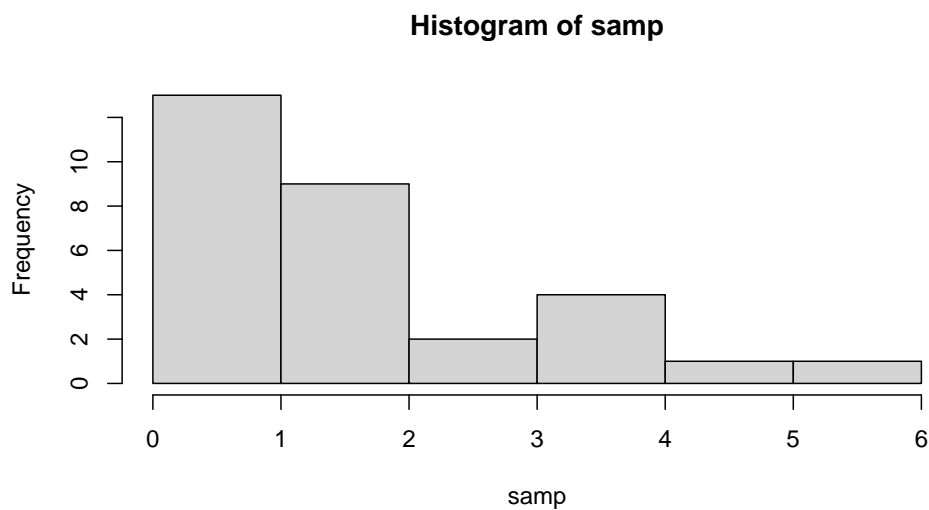
Можно применять функции `lapply()` и `sapply()` на датафреймах. Поскольку фактически датафрейм - это список из векторов одинаковой длины (см. 2.10), то итерироваться эти функции будут по колонкам:

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

```
##           X           name      Gender  Eye.color      Race  Hair.color
##  "integer" "character" "character" "character" "character" "character"
##      Height  Publisher  Skin.color  Alignment      Weight
##  "numeric" "character" "character" "character" "integer"
```

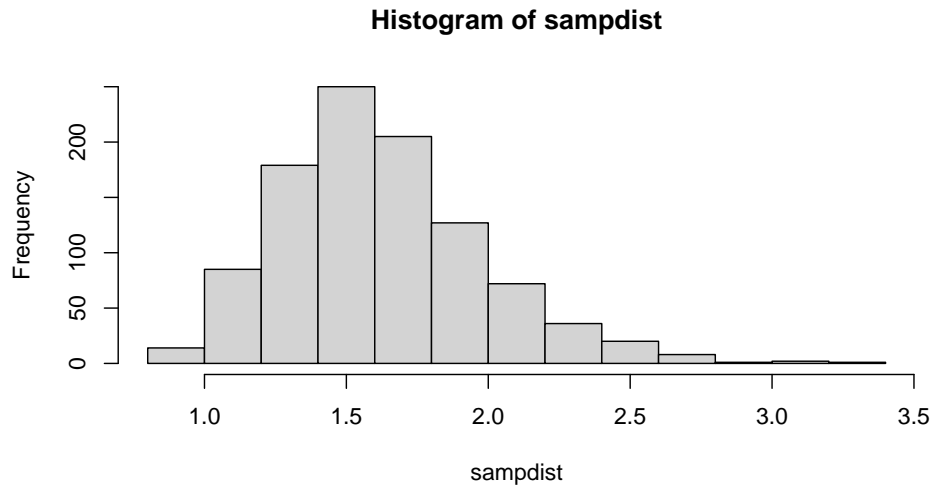
Еще одна функция из семейства `apply()` - функция `replicate()` - самый простой способ повторить одну и ту же операцию много раз. Обычно эта функция используется при симуляции данных и моделировании. Например, давайте сделаем выборку из логнормального распределения:

```
samp <- rlnorm(30)
hist(samp)
```



А теперь давайте сделаем 1000 таких выборок и из каждой возьмем среднее:

```
sampdist <- replicate(1000, mean(rlnorm(30)))
hist(sampdist)
```



Про функции для генерации случайных чисел и про визуализацию мы поговорим в следующие дни.

Если хотите познакомиться с семейством `apply()` чуточку ближе, то рекомендую вот этот [туториал](https://www.datacamp.com/community/tutorials/r-tutorial-apply-family)³.

В заключение стоит сказать, что семейство функций `apply()` — это очень сильное колдунство, но в `tidyverse` оно практически полностью перекрывается функциями из пакета `purrr`. Впрочем, если вы поняли логику `apply()`, то при желании вы легко сможете переключиться на альтернативы из пакета `purrr`.

³<https://www.datacamp.com/community/tutorials/r-tutorial-apply-family>

Глава 6

Задания

6.1 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 9 9
```

```
## [39] 9 9 9 9 9 9 9
```

```

· Сделайте вектор vec, в котором соедините 3, а также значения " " и " ".
## [1] "3"      " "      " "      " "

· Вычестъ 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"

```

6.2 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
```

6.3 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)
```

6.4 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 `rupa` и `lupa`.

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

6.5 Матрицы

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

```
##      [,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 М (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 М

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

- Сравните (==) вторую колонку и вторую строчку матрицы М

```
## [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 строки 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
```

6.6 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 века¹.

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 понадобится 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 и возвращает `TRUE`, если значения в векторе не убывают (то есть каждое следующее - больше или равно предыдущему) или не возрастают.

```
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 7

Решения_заданий

7.1 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 7 7 7 7 7 7 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" " " " "
```

- Вычистить 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"
```

7.2 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
```

7.3 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
```

7.4 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 `ru` 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"
```

7.5 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 %% 1:9
```

- Из матрицы `tab` выделите подматрицу, включающую в себя только строки 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
```

7.6 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 века¹.

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 и возвращает `TRUE`, если значения в векторе не убывают (то есть каждое следующее - больше или равно предыдущему) или не возрастают.

```
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
```

7.7 Семейство `apply()`

- 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
```

- 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) список[[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_sписок, 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.