Наука о данных в R для программы Цифровых гуманитарных исследований

Г. А. Мороз, И. С. Поздняков

Оглавление

О курсе

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

Введение в R

0.1 Наука о данных

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

- биоинформатика
- цифровые гуманитарные исследования
- датажурналистика
- ...

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

- psychologists (**?**)
- immunologists (**?**)
- buisness (**?**)
- public policy (**?**)
- fraud detection (**?**)
- ...

Среди умений датасаентистов можно перечислить следующие:

- сбор и обработка данных
- трансформация данных
- визуализация данных
- моделирование данных
- представление полученных результатов

Все эти темы в той или иной мере будут представлены на нашем курсе.

0.2 Установка R и RStudio

В данной книге используется исключительно R (**?**), так что для занятий понадобятся:

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

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

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

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

```
sessionInfo()$R.version$version.string
```

```
## [1] "R version 3.6.1 (2019-07-05)"
```

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

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

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

0.3 Полезные ссылки

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

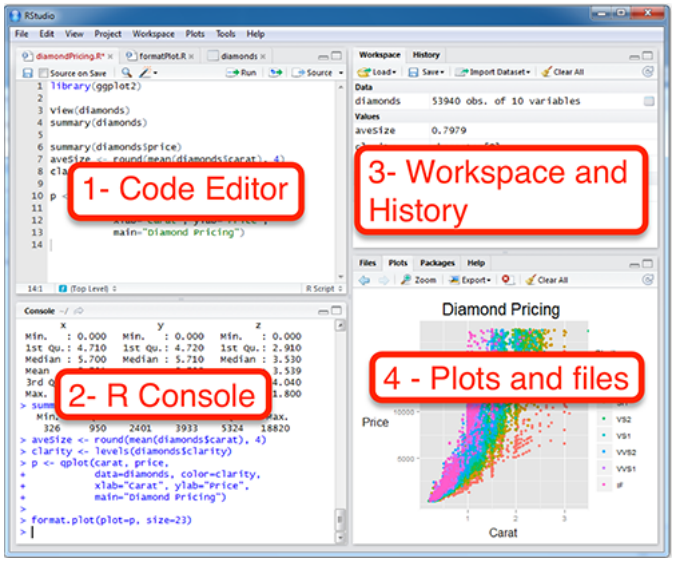
- книга (**?**)[7] является достаточно сильной альтернативой всему курсу

[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
[7] https://r4ds.had.co.nz/

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

0.4 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.



Существуют разные типы пользователей: одни любят работать в консоли (на картинке это **2** — **R Console**), другие предпочитают скрипты (**1** — **Code Editor**). Кон-

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

соль позволяет иметь 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.

0.5 Введение в R

0.5.1 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
```

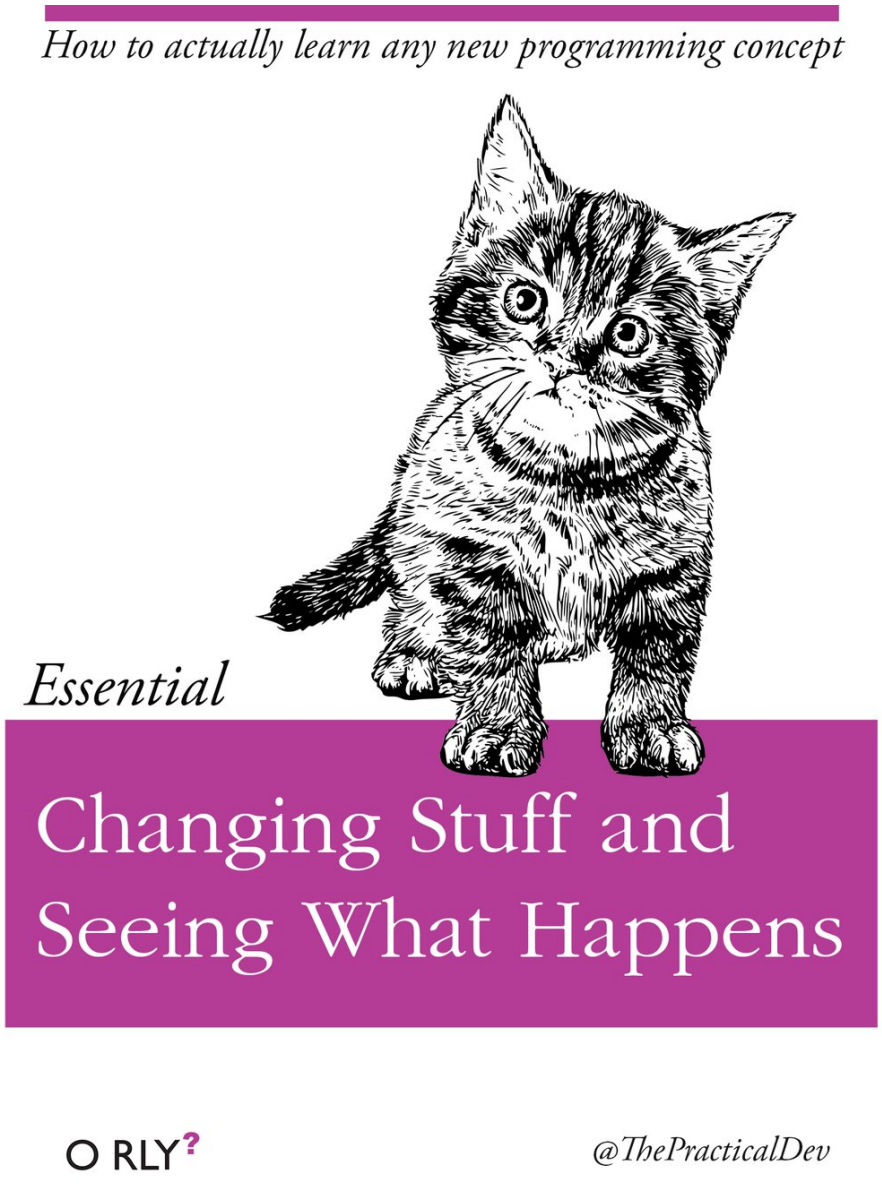
```
## [1] 8
```

```
(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.



0.5.2 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
```

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

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:

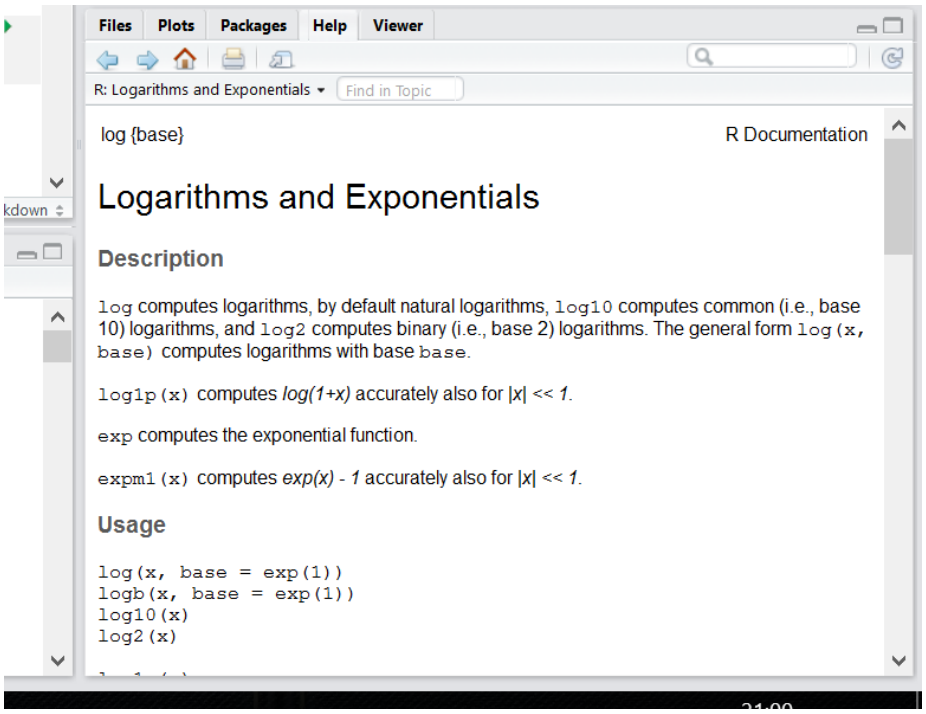
$2^3 = 8$

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 дефолту 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 ином формате, а есть и дополнительные параметры, которые можно прописывать вручную, если параметры по умолчанию нас 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 оутпут одних функций как инпут 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
```

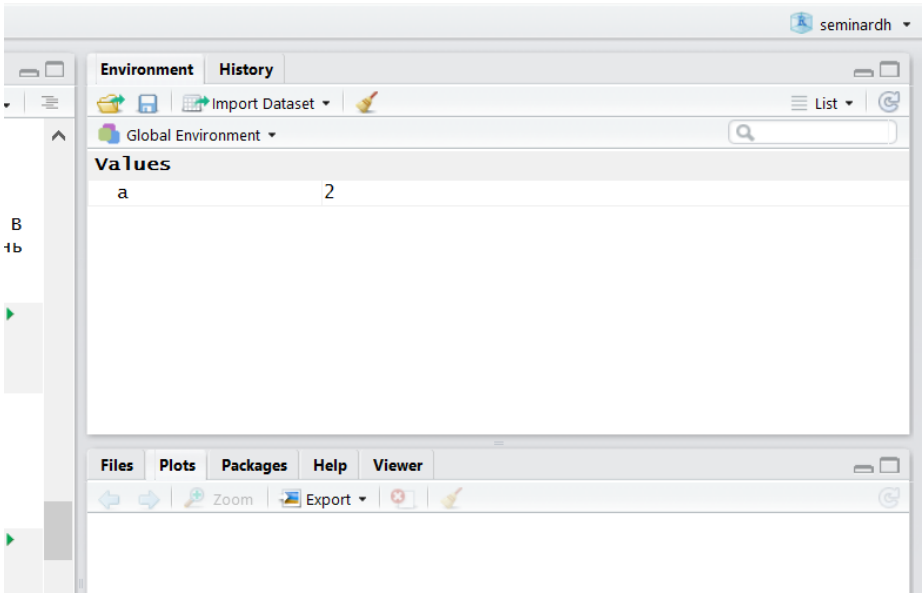
0.5.3 Переменные

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

```
a <- 2
a
```

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

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



Можно использовать переменные в функциях и просто вычислениях:

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

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

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

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

Вы можете сравнивать разные переменные:

```
a == b
```

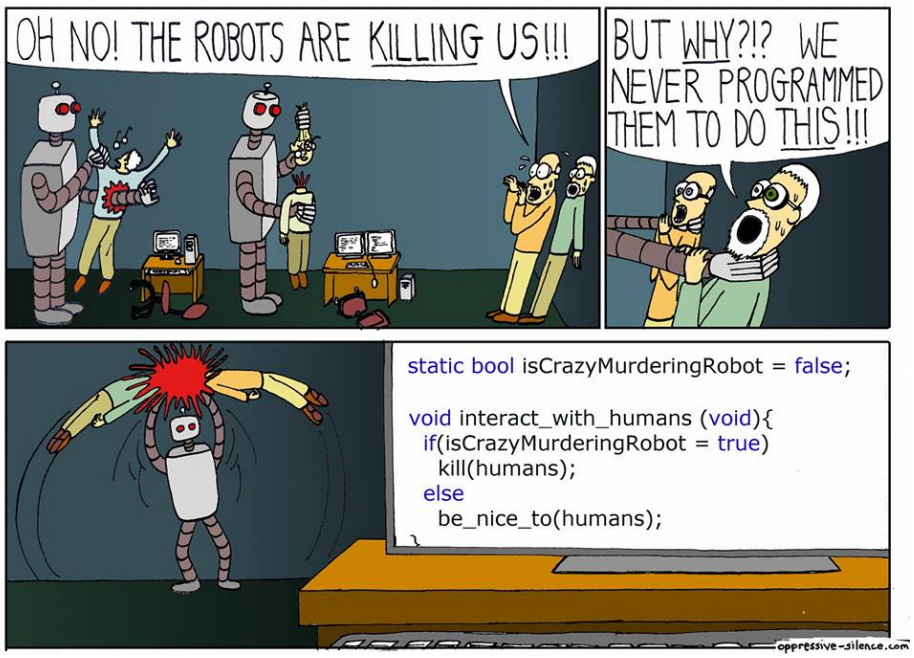
```
## [1] FALSE
```

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

```
a = b
a
```

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

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



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

Иногда нам нужно проверить на *не*равенство:

```
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
```

0.6 Типы данных

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

```
class(a)
```

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

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

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

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

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

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

[18] https://stackoverflow.com/questions/23660094/whats-the-difference-between-integer-class-and-numeric-cl
[19] http://www.r-tutor.com/r-introduction/basic-data-types/complex

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

```
class(s)
```

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

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

```
t1 <- TRUE
f1 <- FALSE

t1
```

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

```
f1
```

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

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

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

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

```
TRUE <- FALSE
```

```
## Error in TRUE <- FALSE: invalid (do_set) left-hand side to assignment
```

```
TRUE
```

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

```
T <- FALSE
T
```

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

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

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

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

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

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

```
t1
```

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

```
!t1
```

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

```
!!t1 #          !
```

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

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

```
t1&t2
```

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

```
t1&f1
```

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

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

```
t1 | f1
```

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

```
f1 | f2
```

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

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

Поздравляю, мы только что разобрались с самой занудной частью. Пора переходить к важному и интересному. ВЕКТОРАМ!

0.7 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.

> P.S. 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, {2,1}. Вот последние два числа и будем считать вектором. Поэтому постарайтесь на время выбросить стрелочки из головы.

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

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

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

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

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

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

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

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

```
## Error in  (3, 4, 5): could not find function " "
```

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!"
```

0.7.1 Coercion

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

Например:

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

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

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

```
2 + TRUE
```

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

Это называется **coercion**. Более сложный пример:

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

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

У R есть иерархия коэрсинга: `NULL < raw < logical < integer < double < complex < character < list < expression`. Мы из этого списка еще многого не знаем, сейчас важно запомнить, что логические данные — `TRUE` и `FALSE` — превращаются в `0` и `1` соответственно, а `0` и `1` в строчки `"0"` и `"1"`.

Если Вы боитесь полагаться на coercion, то можете воспользоваться функциями `as.        :`

```
as.numeric(c(TRUE, FALSE, FALSE))
```

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

```
as.character(as.numeric(c(TRUE, FALSE, FALSE)))
```

```
## [1] "1" "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: NAs introduced by coercion
```

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

0.7.2 Операции с векторами

Все те арифметические операции, 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 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
##  [8] 2.828427 3.000000 3.162278
```

0.7.3 Recycling

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 (**recycling** = *правило переписывания*). Это означает, что если короткий вектор кратен по длине длинному, то он будет повторять короткий необходимое количество раз:

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

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

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

```
n * 2
```

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

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

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

```
## Warning in n + c(3, 4, 5): longer object length is not a multiple of
## shorter object length
```

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

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

0.7.4 Индексирование векторов

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

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

Самое простое - индексировать по номеру индекса, т.е. порядку значения в векторе.

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

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

```
n[10]
```

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

> Если вы знакомы с другими языками программирования (не MATLAB, там все так же) и уже научились думать, что индексация с 0 — это очень удобно и очень правильно (ну или просто свыклись

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

с этим), то в R Вам придется переучиться обратно. Здесь первый индекс — это 1, а последний равен длине вектора — ее можно узнать с помощью функции `length()`. С обоих сторон индексы берутся включительно.

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

```
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 (простите, пользователя Python, которые ожидают здесь отсчет с конца...):

```
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
```

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
```

Ну а если они не равны, то тут будет снова работать правило ресайклинга!

```
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.

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

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

> `letters` - это “зашитая” в R константа - вектор букв от a до z. Иногда это очень удобно! Кроме того, есть константа `LETTERS` - то же самое, но заглавными буквами. А еще 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.

0.7.5 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 этого в R есть `NA`. `NA` — это не строка `"NA"`, не `0`, не пустая строка и не `FALSE`. `NA` — это `NA`. Большинство операций с векторами, содержащими `NA` будут выдавать `NA`:

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

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

```
missed == ""
```

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

```
missed == NA
```

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

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

Иногда `NA` в данных очень бесит:

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

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

```
mean(n)
```

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

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

```
n == NA
```

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

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

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

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

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

Результат выполнения `is.na(n)` выдает FALSE в тех местах, где у нас числа и TRUE там, где у нас NA. Нам нужно сделать наоборот. Здесь нам понадобится оператор `!` (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 дефолту FALSE. Вы знаете, что нужно делать!

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

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

Еееее!

> NA может появляться в векторах других типов тоже. Кроме NA есть еще NaN — это разные вещи. NaN расшифровывается как Not a Number и получается в результате таких операций как 0/0.

0.7.6 В любой непонятной ситуации — ищите в поисковике

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

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

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

— Stack Exchange July 20, 2015

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

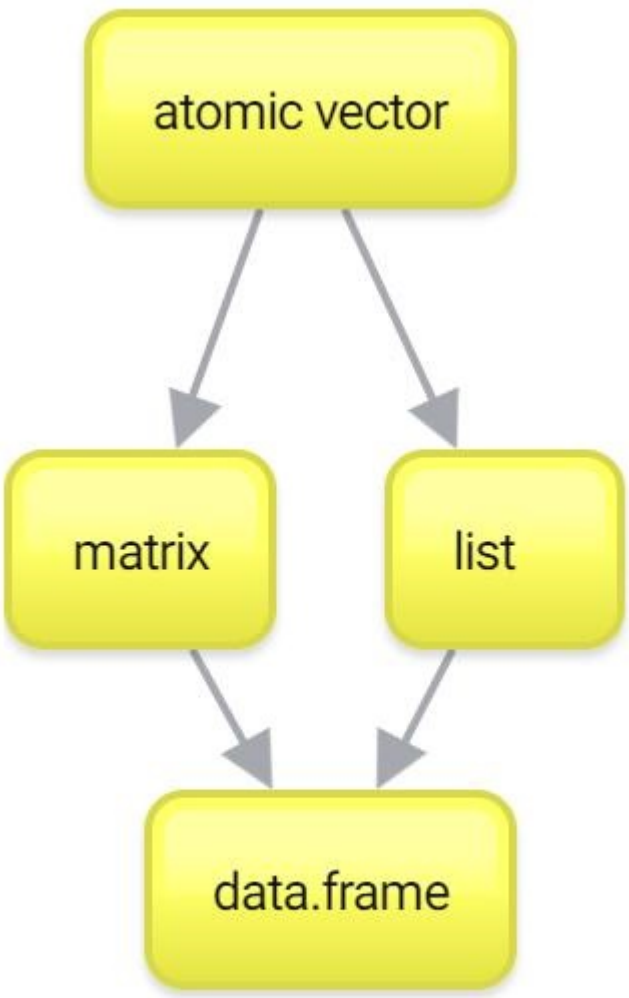
Кроме того, правильно загуглить проблему — не так уж и просто.

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

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

Итак, с векторами мы более-менее разобрались. Помните, что вектора — это

один из краеугольных камней Вашей работы в R. Если Вы хорошо с ними разобрались, то дальше все будет довольно несложно. Тем не менее, вектора — это не все. Есть еще два важных типа данных: списки (**list**) и матрицы (**matrix**). Их можно рассматривать как своеобразное “расширение” векторов, каждый в свою сторону. Ну а списки и матрицы нужны чтобы понять основной тип данных в R — **data.frame**.



0.8 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
```

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

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

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

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

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

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

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

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

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

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

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

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

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

```
A[,]
```

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

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

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

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

0.9 Списки (list)

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

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

```
## [[1]]
## [1] 42
##
## [[2]]
## [1] "    "
##
## [[3]]
## [1] TRUE
```

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

```
lbig <- list(c("Wow", "this", "list", "is", "so", "big"), "16", l)
lbig
```

```
## [[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(lbig)
```

```
## 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:

```
namedl <- list(age = 24, PhDstudent = T, language = "Russian")
namedl
```

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

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

```
namedl$age
```

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

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

```
namedl[1]
```

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

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

```
class(namedl)
```

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

```
class(namedl[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:

```
namedl[[1]]
```

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

```
class(namedl[[1]])
```

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

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

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

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

```
namedl['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. Но обычно после этого стоит как можно скорее превратить список в датафрейм.

0.10 Data.frame

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   : Factor w/ 5 levels "Eugeny","Ivan",..: 2 1 3 4 5
##  $ age    : num  26 34 23 27 26
##  $ student: logi  FALSE FALSE TRUE TRUE TRUE
```

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

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

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

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

Используя оператор $ и присваивание можно создавать новые колонки датафрейма:

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

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

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

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

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

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

А еще можно использовать названия колонок внутри квадратных скобок:

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

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

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

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

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

Эту же задачу можно выполнить другими способами:

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

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

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

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

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

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

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

0.11 Начинаем работу с реальными данными

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

0.11.1 Рабочая папка и проекты

Для начала скачайте файл по ссылке[22]

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

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

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

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

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

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

- Гора идет к Магомету: изменение рабочей папки.

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

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

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

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

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

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

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

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

- Магомет использует кнопочный интерфейс: Import Dataset.

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

- Гора находит Магомета в интернете.

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

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

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

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

0.11.2 Импорт данных

Как Вы уже поняли, импортирование данных - одна из самых муторных и неприятных вещей в R. Если у Вас получится с этим справится, то все остальное - ерун-

да. Мы уже разобрались с первой частью этого процесса - нахождением файла с данными, осталось научиться их читать.

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

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

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

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

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

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

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

данном случае мы просто пока обойдемся без факторов.

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

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

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

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

0.12 Препроцессинг данных в R

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

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

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

0.12.1 Исследование данных

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

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

```
str(got)
```

```
## 'data.frame':    917 obs. of  13 variables:
##  $ Name              : chr  "Addam Marbrand" "Aegon Frey (Jinglebell)" "Aegon Targa
##  $ Allegiances       : chr  "Lannister" "None" "House Targaryen" "House Greyjoy" ..
##  $ Death.Year        : int  NA 299 NA 300 NA NA 300 300 NA NA ...
##  $ Book.of.Death     : int  NA 3 NA 5 NA NA 4 5 NA NA ...
##  $ Death.Chapter     : int  NA 51 NA 20 NA NA 35 NA NA NA ...
##  $ Book.Intro.Chapter: int  56 49 5 20 NA NA 21 59 11 0 ...
##  $ Gender            : int  1 1 1 1 1 1 1 0 1 1 ...
##  $ Nobility          : int  1 1 1 1 1 1 1 1 1 0 ...
##  $ GoT               : int  1 0 0 0 0 0 1 1 0 0 ...
##  $ CoK               : int  1 0 0 0 0 1 0 1 1 0 ...
##  $ SoS               : int  1 1 0 0 1 1 1 1 0 1 ...
##  $ FfC               : int  1 0 0 0 0 0 1 0 1 0 ...
##  $ DwD               : int  0 0 1 1 0 0 0 1 0 0 ...
```

Давайте разберемся с переменными в датафрейме:

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

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

```
head(got)
```

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

```
## 2            51                 49      1        1    0    0    1    0    0
## 3            NA                  5      1        1    0    0    0    0    1
## 4            20                 20      1        1    0    0    0    0    1
## 5            NA                 NA      1        1    0    0    1    0    0
## 6            NA                 NA      1        1    0    1    1    0    0
```

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

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

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

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

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

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

```
##
##                    0   1
##   Arryn            3  20
##   Baratheon        6  50
##   Greyjoy          4  47
##   House Arryn      3   4
##   House Baratheon  0   8
##   House Greyjoy    1  23
##   House Lannister  2  19
##   House Martell    7   5
##   House Stark      6  29
##   House Targaryen  5  14
##   House Tully      0   8
##   House Tyrell     4   7
```

```
##   Lannister        12  69
##   Martell           7  18
##   Night's Watch     0 116
##   None             51 202
##   Stark            21  52
##   Targaryen         1  16
##   Tully             2  20
##   Tyrell            6   9
##   Wildling         16  24
```

0.12.2 Subsetting

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

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

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

и используя имена колонок:

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

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

и даже используя вектора названий колонок!

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

```
##                 Name     Allegiances Gender
## 508    Mance Rayder        Wildling      1
## 509    Mandon Moore       Baratheon      1
## 510  Maric Seaworth House Baratheon      1
## 511           Marei            None      0
## 512 Margaery Tyrell    House Tyrell      0
## 513       Marillion           Arryn      1
## 514           Maris        Wildling      0
## 515    Marissa Frey            None      0
```

Мы можем вытаскивать отдельные колонки как векторы:

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

```
##  [1] "Lannister"       "None"            "House Targaryen"
##  [4] "House Greyjoy"   "Baratheon"       "Night's Watch"
##  [7] "Arryn"           "House Stark"     "House Tyrell"
## [10] "Tyrell"          "Stark"           "Greyjoy"
## [13] "House Lannister" "Martell"         "House Martell"
## [16] "Wildling"        "Targaryen"       "House Arryn"
## [19] "House Tully"     "Tully"           "House Baratheon"
```

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

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

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

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

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

```
##                                Name   Allegiances Death.Year
## 7  Aemon Targaryen (son of Maekar I) Night's Watch        300
```

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

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

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

И не забывайте про запятую!

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

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

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

```
## 24            NA            NA                  19      1        0   1   1
## 49            NA            NA                  75      1        0   0   0
##    SoS FfC DwD
## 7    1   1   0
## 10   1   0   0
## 13   1   0   1
## 16   0   0   0
## 24   1   0   1
## 49   1   0   1
```

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

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

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

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

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

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

```
## 13   1   0   1
## 16   0   0   0
## 24   1   0   1
## 49   1   0   1
```

0.12.3 Создание новых колонок

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

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

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

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

Мы можем создать новую колонку `Is.Alive`:

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

0.12.4 data.table vs. tidyverse

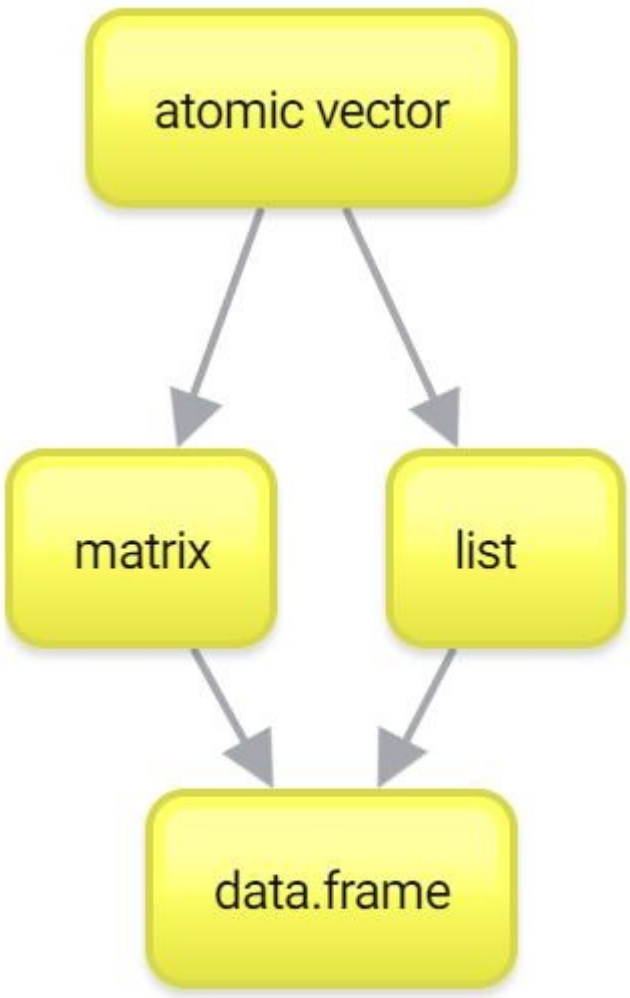
В принципе, с помощью базового R можно сделать все, что угодно. Однако базовые инструменты R — не всегда самые удобные. Идея сделать работу с датафреймами в R еще быстрее и удобнее сподвигла разработчиков на создание новых инструментов — `data.table` и `tidyverse` (`dplyr`). Это два конкурирующих подхода,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

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

Преимущество `data.table` — в суровом лаконичном синтаксисе и наиболее эффективных алгоритмах. Последние обеспечивают очень серьезный прирост в скорости в работе с данными. Чтение файлов и манипуляция данными может

быть на порядки быстрее, поэтому если Ваш датасет с трудом пролезает в оперативную память компьютера, а исполнение скрипта занимает длительное время - стоит задуматься о переходе на `data.table`.

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



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

`tidyverse`: Загрузка и трансформация данных

tidyverse[24] — это набор пакетов:

- *ggplot2*, для визуализации
- *tibble*, для работы с тибблами, современный вариант датафрейма
- *tidyr*, для формата tidy data
- *readr*, для чтения файлов в R
- *purrr*, для функционального программирования
- *dplyr*, для преобразованиия данных
- *stringr*, для работы со строковыми переменными
- *forcats*, для работы с переменными-факторами

Полезно также знать о следующих:

- *readxl*, для чтения .xls и .xlsx
- *jsonlite*, для работы с JSON
- *rvest*, для веб-скреппинга
- *lubridate*, для работы с временем
- *tidytext*, для работы с текстами и корпусами
- *broom*, для перевода в tidy формат статистические модели

```
library("tidyverse")
```

```
## -- Attaching packages --------------- tidyverse 1.2.1 --
## v ggplot2 3.2.1     v purrr   0.3.2
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0
## -- Conflicts ------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
```

[24] https://www.tidyverse.org

```
## x dplyr::lag()    masks stats::lag()
```

0.13 Загрузка данных

0.13.1 Рабочая директория

Все в R происходит где-то. Нужно загружать файлы с данными, нужно их куда-то сохранять. Желательно иметь для каждого проекта некоторую отдельную папку на компьютере, куда складывать все, отнсящееся к этому проекту. Две команды позволят опредить текущую рабочую дерикторию (`getwd()`) и (`setwd(.../path/to/your/directory)`).

0.13.2 Форматы данных: `.csv`

Существет много форматов данных, которые придумали люди. Большинство из них можно загрузить в R. Так как центральный объект в R – таблица $n \times k$, то и работать мы большую часть времени будем с таблицами. Наиболее распространенные способы хранить данные сейчас это `.csv` (разберем в данном разделе) и `.json` (разберем в разделе **??**).

`.csv` (comma separated values) – является обычным текстовым файлом, в котором перечислены значения с некоторым фиксированным разделителем: запятой, табуляцией, точка с запятой, пробел и др. Такие файлы обычно легко открывает LibreOffice, а в Microsoft Excel нужны некоторые трюки[25].

0.13.3 Загрузка данных: readr, readxl

Стандартной функцией для чтения `.csv` файлов в R является функция `read.csv()`, но мы будем использовать функцию `read_csv()` из пакета `readr`.

```
read_csv("...")
```

Вместо многоточия может стоять:

- название файла (если он, есть в текущей рабочей дериктории)

```
read_csv("my_file.csv")
```

- относительный путь к файлу (если он, верен для текущей рабочей дериктории)

[25] https://superuser.com/questions/291445/how-do-you-change-default-delimiter-in-the-text-import-in-excel

```
read_csv("data/my_file.csv")
```

- полный путь к файлу (если он, верен для текущей дериктории)

```
read_csv("/home/user_name/work/data/my_file.csv")
```

- интернет ссылка (тогда, компьютер должен быть подключен к интернету)

```
read_csv("https://my_host/my_file.csv")
```

Для чтения других форматов `.csv` файлов используются другие функции:

- `read_tsv()` – для файлов с табуляцией в качестве разделителя
- `read_csv2()` – для файлов с точкой с запятой в качестве разделителя
- `read_delim(file = "... delim = "...")` – для файлов с любым разделителем, задаваемым аргументом `delim`

Стандартной практикой является создавать первой строкой `.csv` файлов названия столбцов, поэтому по умолчанию функции `read_...()` будут создавать таблицу, считая первую строку названием столбцов. Чтобы изменить это поведение следует использовать аргумент `col_names = FALSE`.

Другая проблема при чтении файлов – кодировка и локаль. На разных компьютерах разные локали и дефолтные кодировки, так что имеет смысл знать про аргумент `locale(encoding = "UTF-8")`.

Попробуйте корректно считать в R файл по этой ссылке[26].

```
## # A tibble: 3 x 3
##   cyrillic ipa_symbols greek
##   <chr>    <chr>       <chr>
## 1
## 2
## 3
```

Благодаря `readxl` пакету Также данные можно скачать напрямую из файлов `.xls` (функция `read_xls`) и `.xlsx` (функция `read_xlsx`), однако эти функции не умеют читать из интернета.

```
library("readxl")
xlsx_example <- read_xlsx("...")
```

[26] `https://raw.githubusercontent.com/agricolamz/DS_for_DH/master/data/scary_letters.csv`

Существует еще один экстравагантный способ хранить данные: это формат файлов R .RData. Создадим data.frame:

```
my_df <- data.frame(letters = c("a", "b"),
                    numbers = 1:2)
my_df
```

```
##   letters numbers
## 1       a       1
## 2       b       2
```

Теперь можно сохранить файл...

```
save(my_df, file = "data/my_df.RData")
```

удалить переменную...

```
rm(my_df)
my_df
```

```
## Error in eval(expr, envir, enclos): object 'my_df' not found
```

и загрузить все снова:

```
load("data/my_df.RData")
my_df
```

```
##   letters numbers
## 1       a       1
## 2       b       2
```

0.13.3.1 Misspelling dataset

Этот датасет я переработал из данных, собранных для статьи The Gyllenhaal Experiment[27], написанной Расселом Гольденбергом и Мэттом Дэниэлсом для издания pudding[28]. Они анализировали ошибки в правописании при поиске имен и фамилий звезд.

```
misspellings <- read_csv("https://raw.githubusercontent.com/agricolamz/DS_for_DH/maste
```

[27] https://pudding.cool/2019/02/gyllenhaal/
[28] https://pudding.cool

```
## Parsed with column specification:
## cols(
##   correct = col_character(),
##   spelling = col_character(),
##   count = col_double()
## )
```

```
misspellings
```

```
## # A tibble: 15,477 x 3
##    correct   spelling    count
##    <chr>     <chr>       <dbl>
##  1 deschanel deschanel   18338
##  2 deschanel dechanel     1550
##  3 deschanel deschannel    934
##  4 deschanel deschenel     404
##  5 deschanel deshanel      364
##  6 deschanel dechannel     359
##  7 deschanel deschanelle   316
##  8 deschanel dechanelle    192
##  9 deschanel deschanell    174
## 10 deschanel deschenal     165
## # ... with 15,467 more rows
```

В датасете следующие переменные:

- `correct` – корректное написание фамилии
- `spelling` – написание, которое сделали пользователи
- `count` – количество случаев такого написания

0.13.3.2 `diamonds`

```
diamonds
```

```
## # A tibble: 53,940 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1 0.23  Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
##  2 0.21  Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
##  3 0.23  Good      E     VS1      56.9    65   327  4.05  4.07  2.31
##  4 0.290 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
##  5 0.31  Good      J     SI2      63.3    58   335  4.34  4.35  2.75
##  6 0.24  Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
##  7 0.24  Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
##  8 0.26  Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
```

```
##  9 0.22  Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
## 10 0.23  Very Good H     VS1      59.4    61   338  4     4.05  2.39
## # ... with 53,930 more rows
```

```
?diamonds
```

0.14 tibble

Пакет `tibble` – является альтернативой штатного датафрейма в R. Существует встроенная переменная `month.name`:

```
month.name
```

```
##  [1] "January"   "February"  "March"     "April"     "May"
##  [6] "June"      "July"      "August"    "September" "October"
## [11] "November"  "December"
```

Можно создать датафрейм таким образом:

```
data.frame(id = 1:12,
           months = month.name,
           n_letters = nchar(months))
```

```
## Error in nchar(months): cannot coerce type 'closure' to vector of type 'character'
```

Однако переменная `months` не создана пользователем, так что данный код выдает ошибку. Корректный способ сделать это базовыми средствами:

```
data.frame(id = 1:12,
           months = month.name,
           n_letters = nchar(month.name))
```

```
##    id    months n_letters
## 1   1   January         7
## 2   2  February         8
## 3   3     March         5
## 4   4     April         5
## 5   5       May         3
## 6   6      June         4
## 7   7      July         4
## 8   8    August         6
## 9   9 September         9
## 10 10   October         7
## 11 11  November         8
```

```
## 12 12  December         8
```

Одно из отличий `tibble` от базового датафрейма – возможность использовать создаваемые "по ходу пьесы переменные"

```
tibble(id = 1:12,
       months = month.name,
       n_letters = nchar(months))
```

```
## # A tibble: 12 x 3
##       id months    n_letters
##    <int> <chr>         <int>
##  1     1 January           7
##  2     2 February          8
##  3     3 March             5
##  4     4 April             5
##  5     5 May               3
##  6     6 June              4
##  7     7 July              4
##  8     8 August            6
##  9     9 September         9
## 10    10 October           7
## 11    11 November          8
## 12    12 December          8
```

Если в окружении пользователя уже есть переменная с датафреймом, его легко можно переделать в `tibble` при помощи функции `as_tibble()`:

```
df <- data.frame(id = 1:12,
                 months = month.name)

df
```

```
##    id    months
## 1   1   January
## 2   2  February
## 3   3     March
## 4   4     April
## 5   5       May
## 6   6      June
## 7   7      July
## 8   8    August
## 9   9 September
## 10 10   October
## 11 11  November
## 12 12  December
```

```
as_tibble(df)
```

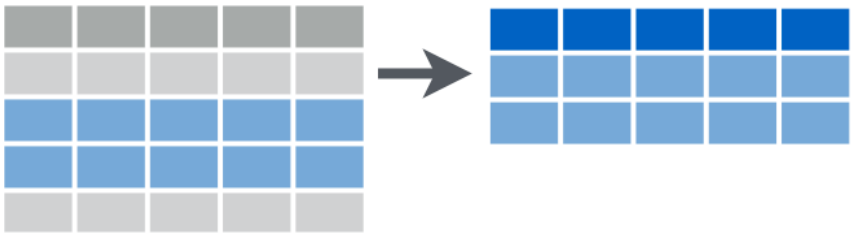
```
## # A tibble: 12 x 2
##       id months
##    <int> <fct>
##  1     1 January
##  2     2 February
##  3     3 March
##  4     4 April
##  5     5 May
##  6     6 June
##  7     7 July
##  8     8 August
##  9     9 September
## 10    10 October
## 11    11 November
## 12    12 December
```

Функционально `tibble` от `data.frame` ничем не отличается, однако существует ряд несущественных отличий. Кроме того стоит помнить, что многие функции из `tidyverse` возвращают именно `tibble`, а не `data.frame`.

0.15 dplyr

В сжатом виде содержание этого раздела хранится вот здесь[29].

0.15.1 dplyr::filter()



Эта функция фильтрует строчки по условиям, основанным на столбцах.

Сколько неправильных произношений, которые написали меньше 10 юзеров?

[29] https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf

```
misspellings %>%
  filter(count < 10)
```

```
## # A tibble: 14,279 x 3
##    correct   spelling    count
##    <chr>     <chr>       <dbl>
##  1 deschanel deshanael       9
##  2 deschanel daychanel       9
##  3 deschanel deschaneles     9
##  4 deschanel dashenel        9
##  5 deschanel deschenael      9
##  6 deschanel deechanel       9
##  7 deschanel deichanel       9
##  8 deschanel dechantel       9
##  9 deschanel deychanel       9
## 10 deschanel daschenell      9
## # ... with 14,269 more rows
```

%>% — конвеер (pipe) отправляет результат работы одной функции в другую.

```
sort(sqrt(abs(sin(1:22))), decreasing = TRUE)
```

```
##  [1] 0.9999951 0.9952926 0.9946649 0.9805088 0.9792468 0.9554817 0.9535709
##  [8] 0.9173173 0.9146888 0.8699440 0.8665952 0.8105471 0.8064043 0.7375779
## [15] 0.7325114 0.6482029 0.6419646 0.5365662 0.5285977 0.3871398 0.3756594
## [22] 0.0940814
```

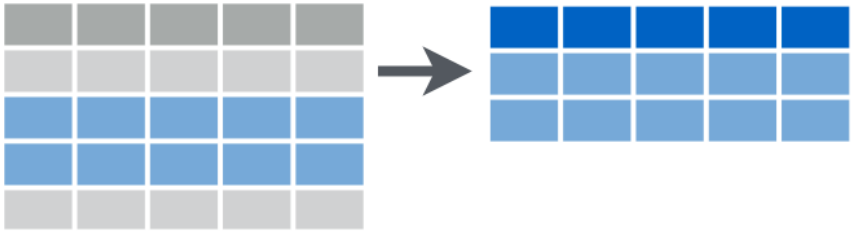
```
1:22 %>%
  sin() %>%
  abs() %>%
  sqrt() %>%
  sort(., decreasing = TRUE) #           ?
```

```
##  [1] 0.9999951 0.9952926 0.9946649 0.9805088 0.9792468 0.9554817 0.9535709
##  [8] 0.9173173 0.9146888 0.8699440 0.8665952 0.8105471 0.8064043 0.7375779
## [15] 0.7325114 0.6482029 0.6419646 0.5365662 0.5285977 0.3871398 0.3756594
## [22] 0.0940814
```

Конвееры в *tidyverse* пришли из пакета *magrittr*. Иногда они работают не корректно с функциями не из *tidyverse*.

0.15.2 dplyr::slice()

Эта функция фильтрует строчки по индексу.



```
misspellings %>%
  slice(3:7)
```

```
## # A tibble: 5 x 3
##   correct   spelling    count
##   <chr>     <chr>       <dbl>
## 1 deschanel deschannel    934
## 2 deschanel deschenel     404
## 3 deschanel deshanel      364
## 4 deschanel dechannel     359
## 5 deschanel deschanelle   316
```

0.15.3 dplyr::select()

Эта функция позволяет выбрать столбцы.

```
diamonds %>%
  select(8:10)
```

```
## # A tibble: 53,940 x 3
##        x     y     z
##    <dbl> <dbl> <dbl>
##  1  3.95  3.98  2.43
##  2  3.89  3.84  2.31
##  3  4.05  4.07  2.31
##  4  4.2   4.23  2.63
##  5  4.34  4.35  2.75
##  6  3.94  3.96  2.48
##  7  3.95  3.98  2.47
##  8  4.07  4.11  2.53
##  9  3.87  3.78  2.49
## 10  4     4.05  2.39
## # ... with 53,930 more rows
```

```
diamonds %>%
  select(color:price)
```

```
## # A tibble: 53,940 x 5
##    color clarity depth table price
##    <ord> <ord>   <dbl> <dbl> <int>
##  1 E     SI2      61.5    55   326
##  2 E     SI1      59.8    61   326
##  3 E     VS1      56.9    65   327
##  4 I     VS2      62.4    58   334
##  5 J     SI2      63.3    58   335
##  6 J     VVS2     62.8    57   336
##  7 I     VVS1     62.3    57   336
##  8 H     SI1      61.9    55   337
##  9 E     VS2      65.1    61   337
```

```
## 10 H     VS1      59.4    61   338
## # ... with 53,930 more rows
```

```
diamonds %>%
  select(-carat)
```

```
## # A tibble: 53,940 x 9
##    cut       color clarity depth table price     x     y     z
##    <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
##  2 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
##  3 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
##  4 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
##  5 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
##  6 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
##  7 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
##  8 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
##  9 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
## 10 Very Good H     VS1      59.4    61   338  4     4.05  2.39
## # ... with 53,930 more rows
```

```
diamonds %>%
  select(-c(carat, cut, x, y, z))
```

```
## # A tibble: 53,940 x 5
##    color clarity depth table price
##    <ord> <ord>   <dbl> <dbl> <int>
##  1 E     SI2      61.5    55   326
##  2 E     SI1      59.8    61   326
##  3 E     VS1      56.9    65   327
##  4 I     VS2      62.4    58   334
##  5 J     SI2      63.3    58   335
##  6 J     VVS2     62.8    57   336
##  7 I     VVS1     62.3    57   336
##  8 H     SI1      61.9    55   337
##  9 E     VS2      65.1    61   337
## 10 H     VS1      59.4    61   338
## # ... with 53,930 more rows
```

```
diamonds %>%
  select(cut, depth, price)
```

```
## # A tibble: 53,940 x 3
##    cut       depth price
```

```
##    <ord>     <dbl> <int>
##  1 Ideal      61.5   326
##  2 Premium    59.8   326
##  3 Good       56.9   327
##  4 Premium    62.4   334
##  5 Good       63.3   335
##  6 Very Good  62.8   336
##  7 Very Good  62.3   336
##  8 Very Good  61.9   337
##  9 Fair       65.1   337
## 10 Very Good  59.4   338
## # ... with 53,930 more rows
```

0.15.4 dplyr::arrange()

Эта функция сортирует (строки по алфавиту, а числа по порядку).

```
misspellings %>%
  arrange(count)
```

```
## # A tibble: 15,477 x 3
##    correct   spelling   count
##    <chr>     <chr>      <dbl>
##  1 deschanel deschil        1
##  2 deschanel deshauneil     1
##  3 deschanel deschmuel      1
##  4 deschanel deshannle      1
##  5 deschanel deslanges      1
##  6 deschanel deshoenel      1
##  7 deschanel dechadel       1
##  8 deschanel dooschaney     1
##  9 deschanel dishana        1
## 10 deschanel deshaneil      1
## # ... with 15,467 more rows
```

```
diamonds %>%
  arrange(desc(carat), price)
```

```
## # A tibble: 53,940 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  5.01 Fair      J     I1       65.5    59 18018 10.7  10.5   6.98
##  2  4.5  Fair      J     I1       65.8    58 18531 10.2  10.2   6.72
##  3  4.13 Fair      H     I1       64.8    61 17329 10     9.85  6.43
##  4  4.01 Premium   I     I1       61      61 15223 10.1  10.1   6.17
```

```
##  5  4.01 Premium   J     I1       62.5    62 15223 10.0   9.94  6.24
##  6  4    Very Good I     I1       63.3    58 15984 10.0   9.94  6.31
##  7  3.67 Premium   I     I1       62.4    56 16193  9.86  9.81  6.13
##  8  3.65 Fair      H     I1       67.1    53 11668  9.53  9.48  6.38
##  9  3.51 Premium   J     VS2      62.5    59 18701  9.66  9.63  6.03
## 10  3.5  Ideal     H     I1       62.8    57 12587  9.65  9.59  6.03
## # ... with 53,930 more rows
```

`dplyr::distinct()` Эта функция возвращает уникальные значения в столбце или комбинации столбцов.

```
misspellings %>%
  distinct(correct)
```

```
## # A tibble: 15 x 1
##    correct
##    <chr>
##  1 deschanel
##  2 mclachlan
##  3 galifianakis
##  4 labeouf
##  5 macaulay
##  6 mcconaughey
##  7 minaj
##  8 morissette
##  9 poehler
## 10 shyamalan
## 11 kaepernick
## 12 mcgwire
## 13 palahniuk
## 14 picabo
## 15 johansson
```

```
misspellings %>%
  distinct(spelling)
```

```
## # A tibble: 15,462 x 1
##    spelling
##    <chr>
##  1 deschanel
##  2 dechanel
##  3 deschannel
##  4 deschenel
##  5 deshanel
##  6 dechannel
```